제 28 권 제 1 호
Vol. 28 No. 1

2026

제 28 회

한국 소프트웨어공학 학술대회 논문집

Proceedings of the 28th Korea Conference on Software Engineering (KCSE 2026)

- 일시: 2026년 2월 4일(수) ~ 2월 6일(금)
- 장소: UNIST(울산과학기술원) 114동 경영관

주최: 한국정보과학회, 한국정보처리학회
주관: 한국정보과학회 소프트웨어공학 소사이어티
한국정보처리학회 소프트웨어공학연구회

후원:

Program of the 28th Korea Conference on Software Engineering (KCSE 2026)

초대의 글

소프트웨어공학 학술대회(KCSE 2026) 참가자 여러분을 환영합니다.

KCSE (Korea Conference on Software Engineering)는 기업, 연구소 및 학계에서 활동하고 계신 소프트웨어공학 분야 전문가들의 모임으로, 한국정보과학회 소프트웨어공학 소사이어티와 한국정보처리학회 소프트웨어공학 연구회가 소프트웨어공학 기술의 발전 및 적용 확산을 위하여 1999 년부터 매년 개최하는 학술대회입니다.

이번 제 28 회 학술대회는 "소프트웨어공학 가치의 확산: DevSecOps, MLSecOps, 그리고 그 너머"를 주제로, 기조 연설, 튜토리얼, 신진 연구자 발표, 우수논문 발표 등의 초청 세션과 소프트웨어공학 분야 각계에서 제출한 47 편의 엄선된 논문으로 구성되었습니다. 이번 KCSE 2026 학술대회가 소프트웨어공학을 연구하고, 적용하는 모든 연구자 그리고 전문가 여러분께 즐겁고 활기찬 학술 교류 및 기술 협력의 장이 될 수 있도록 여러분들의 많은 관심과 참여를 부탁드립니다.

제 28 회 KCSE 학술행사를 위해 수고해 주신 조직위원회와 학술위원회 위원들, 후원 기관 관계자 여러분, 그리고 기조 연설을 포함한 학술대회 모든 발표자분들께 깊이 감사드리며 건승을 기원합니다.

한국정보과학회 소프트웨어공학 소사이어티 회장 이정원

한국정보처리학회 소프트웨어공학연구회 운영위원장 유준범

학술대회 준비 위원회

공동대회장: 이정원 교수(아주대), 유준범 교수(건국대)

조직위원장: 이주용 교수(UNIST)
조직위원: 이정원 교수(아주대), 유준범 교수(건국대), 홍 신 교수(충북대),
남재창 교수(한동대), 김영재 박사과정(UNIST), 배경민 교수(POSTECH),
강종구 교수(성신여대)
학술위원장: 김미정 교수(UNIST)
학술위원: 강종구 교수(성신여대), 고인영 교수(KAIST), 김기섭 교수(DGIST)
김동선 교수(고려대), 김문주 교수(KAIST), 김윤호 교수(한양대),
김정아 교수(가톨릭관동대), 김진대 교수(서울과학기술대),
김진현 교수(경상대), 김태호 박사(IITP), 김택수 박사(삼성전자),
김형석 교수(충남대), 남재창 교수(한동대), 류덕산 교수(전북대),
마유승 박사(ETRI), 박수진 교수(서강대), 박지훈 교수(충남대),
배경민 교수(POSTECH), 백종문 교수(KAIST), 서영석 교수(영남대),
손정주 교수(경북대), 송지영 교수(한남대), 안가빈 교수(고려대),
안성수 교수(경상대), 양근석 교수(한경대), 유명성 교수(서울시립대),
유 신 교수(KAIST), 유준범 교수(건국대), 윤회진 교수(협성대),
이선아 교수(경상대), 이우석 교수(한양대), 이은서 교수(국립경국대),
이은주 교수(경북대), 이재권 교수(강원대), 이정원 교수(아주대),
이지현 교수(전북대), 이찬근 교수(중앙대), 이희진 교수(동양미래대),
정우성 교수(서울교대), 정필수 교수(경상대), 지은경 교수(KAIST),
차상길 교수(KAIST), 차수영 교수(성균관대) 채흥석 교수(부산대),
최윤자 교수(경북대), 허기홍 교수(KAIST), 홍 신 교수(충북대),
홍장의 교수(충북대)

문의사항 연락처

학술대회 홈페이지: https://kcse.sigsoft.or.kr/2026/
조 직: 이주용 교수 (E-mail: jooyong@unist.ac.kr)
학 술: 김미정 교수 (E-mail: mijungk@unist.ac.kr)

KCSE 2026 프로그램 개요

2 월 4 일 (수)			
12:00-13:00	KCSE 2026 등록		
13:00-15:30	**튜토리얼 T1. 박상돈 교수** 102 호 (좌장: 배경민 교수)	**튜토리얼 T2. 지은경 교수** 111 호 (좌장: 류덕산 교수)	**튜토리얼 T3. 김진현 교수** 112 호 (좌장: 백종문 교수)
15:30-15:45	휴식		
15:45-16:00	**개회식** 112 호		
16:00-17:00	**기조강연 1. 박인욱 상무(LG 전자)** 112 호		
17:00-17:15	휴식		
17:15-20:00	**석식 및 태화강 국가 정원 은하수길 산책**		

2 월 5 일 (목)				
09:00-09:50	**신진연구자 세미나 N1.** **안가빈 교수** 102 호 (좌장: 김윤호 교수)	**신진연구자 세미나 N2.** **유명성 교수** 111 호 (좌장: 손정주 교수)	**신진연구자 세미나 N3.** **신용준 박사** 112 호 (좌장: 지은경 교수)	
09:50-10:00	휴식			
10:00-11:30	**A1. SW 테스팅 I** 102 호 (좌장: 안가빈 교수)	**A2. 퍼징 및 기호실행** 111 호 (좌장: 김미정 교수)	**A3. SW 결함 I** 112 호 (좌장: 백종문 교수)	**A4. SE 를 위한 AI I** 106 호 (좌장: 유명성 교수)
11:30-13:00	중식			
13:00-13:50	**B1. SW 안전** 102 호 (좌장: 유명성 교수)	**B2. SW 보안 I** 111 호 (좌장: 차상길 교수)	**B3. SW 결함 II** 112 호 (좌장: 남재창 교수)	**B4. 응용 SW** 106 호 (좌장: 김기섭 교수)
13:50-14:00	휴식			
14:00-15:00	**C1. 프로그램 수정** 102 호 (좌장: 김윤호 교수)	**C2. SW 보안 II** 111 호 (좌장: 차상길 교수)	**C3. SE 를 위한 AI II** 112 호 (좌장: 남재창 교수)	**C4. 제조 AI 를 위한 SE** 106 호 (좌장: 김기섭 교수)
15:00-15:15	휴식			
15:15-16:15	**IITP 공청회: AI-native SW 산업 혁신 기술 개발 사업** 112 호			
16:15-16:30	휴식			
16:30-17:30	**기조강연 2. 이희조 교수(고려대학교)** 112 호			
17:30-18:00	휴식			
18:00-20:00	**석식 및 시상식(Banquet)** 102 동 대학본부 4 층 경동홀			

2 월 6 일 (금)			
09:00-10:00	**기조강연 3. 최재식 교수(KAIST)** 112 호		
10:00-10:15	휴식		
10:15-11:45	**D1. SW 테스팅 II** 102 호 (좌장: 손정주 교수)	**D2. SW 디버깅** 111 호 (좌장: 지은경 교수)	**D3. 프로그램 분석 및 모델링** 112 호 (좌장: 배경민 교수)
11:45-11:50	휴식		
11:50-12:00	**폐회식** 112 호		

KCSE 2026 프로그램

세션 A1. SW 테스팅

- 2026년 2월 5일 (목) 오전 10:00-11:30 / 102호
- 좌장: 안가빈 교수(고려대)

[초청발표] An Empirical Study of Web Flaky Tests: Understanding and Unveiling DOM Event Interaction Challenges, Yu Pei(University of Luxembourg), 손정주(경북대), Mike Papadakis(Unibersity of Luxembourg), The 18th IEEE International Conference on Software Testing, Verification and Validation (ICST 2025) 국제학술대회 발표 논문
발표자: 손정주(경북대)

기기의 건전성 테스트를 위한 누적손실기반 평가 메트릭 설계 및 검증
최민서, 김진세, 이정원(아주대)

[초청발표] Automated code-based test case reuse for software product line testing,
정필수, 이선아, 이의천(경상국립대), ICST 2024 Journal First
발표자: 정필수(경상국립대)

MCP 기반 멀티 에이전트 클라우드 DevSecOps 보안·규제·비용 통합 대응 워크플로 (학부생논문)
김수민, 김영서, 심희윤, 장예린(이화여대)

바이트코드 기반 Bugram 기법의 성능 평가 (단편논문)
추새벽(실버든든), 남재창(한동대)

세션 A2. 퍼징 및 기호실행

- 2026년 2월 5일 (목) 오전 10:00-11:30 / 111호
- 좌장: 김미정 교수(UNIST)

톰슨 샘플링 기반 지향성 협력 퍼저 (일반논문)
모현민, 김윤호(한양대)

ExplosionGuard: 예산 제약 심볼릭 실행을 위한 정책 합성 및 가드레일 시스템 (단편논문)
이재영(선린인터넷고등학교), 이건우(충주고등학교)

[초청발표] Lightweight Concolic Testing via Path-Condition Synthesis for Deep Learning Libraries,
김세훈, 김용현, 박다현, 전유석, 이주용, 김미정(UNIST), IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025) 국제학술대회 발표 논문
발표자: 김세훈(UNIST)

딥러닝 라이브러리 계산 오류 탐지의 정확도 개선을 위한 차등 테스팅 파이프라인 (단편논문)
Usmonali Pakhlavonov, 김세훈(UNIST)

[초청발표] Fork State-Aware Differential Fuzzing for Blockchain Consensus Implementations,
김원회, 남호철(KAIST), Muoi Tran(ETH Zurich), Amin Jalilov(KAIST), Zhenkai Liang(National University of Singapore), 차상길(KAIST), 강민석(National University of Singapore), IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025) 국제학술대회 발표 논문
발표자: 김원회(KAIST)

세션 A3. SW 결함 I

- 2026년 2월 5일 (목) 오전 10:00-11:30 / 112호
- 좌장: 백종문 교수(KAIST)

앙상블 머신러닝과 대형 언어 모델의 선택적 통합을 통한 비용 효율적인 Just-In-Time 결함 예측 (일반논문)
Dimitri Romain Bekale Be Ndong, 류덕산, Faisal Mohammad, Junaid Khan Kakar (전북대)

[초청발표] Can We Trust the Actionable Guidance from Explainable AI Techniques in Defect Prediction?,
이기찬, 주한세, Scott Uk-Jin Lee(한양대), The 32nd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2025) 국제학술대회 발표 논문
발표자: 이기찬(한양대)

Tabpfn 기반의 소프트웨어 결함 예측 (단편논문)
임창우, 류덕산(전북대)

TabulaRNN 기반의 소프트웨어 결함 예측 (단편논문)
신중현, 류덕산(전북대)

KAN-BE: 파라미터 효율적 앙상블을 활용한 소프트웨어 결함 예측 (일반논문)
최윤서, 류덕산(전북대), 백종문(KAIST)

세션 A4. SE 를 위한 AI I

- 2026년 2월 5일 (목) 오전 10:00-11:30 / 106호
- 좌장: 유명성 교수(서울시립대)

단일 LLM 기반 테스트 Assertion 생성의 품질 강화: 변이 점수 피드백과 뮤턴트-가드 손실 결합 (일반논문)
김윤기, 양근석(한경국립대)

커버리지 피드백을 활용하는 LLM 기반 단위 테스트 자동 생성 기법 (학부생논문)
류병우(UNIST)

대규모 언어 모델을 활용한 단위 테스트의 적합성 자동 식별 (단편논문)
김대원, 이영규, 유준범(건국대)

진화하는 AI 에이전트를 위한 적응형 런타임 테스팅의 필요성 (단편논문)
Zhaoyan Wang, 안현준, 고인영(KAIST)

Ko-LLM 의 디버깅 성능 및 답변 품질 비교 분석 (학부생논문)
정수정, 정호연, 박효근, 김진대(서울과학기술대)

세션 B1. SW 안전

- 2026년 2월 5일 (목) 오후 1:00-1:50 / 102호
- 좌장: 유명성 교수(서울시립대)

스마트팩토리 예지보전을 위한 모니터링 소프트웨어 설계 (학부생논문)
황세현, 김진세, 최민서, 이정원(아주대)

충돌 위험 인식을 위한 Grad-CAM 과 LLM 결합 설명 시스템 (학부생논문)
신지아, 이선아(경상국립대)

오픈소스 LLM 신뢰성 평가 프레임워크 설계 및 실험 : 신뢰성 5 대 품질 특성 중심 프롬프트 기반 Judge LLM 평가 방법 (단편논문)
김영찬, 김순태(전북대)

세션 B2. SW 보안 I

- 2026년 2월 5일 (목) 오후 1:00-1:50 / 111호
- 좌장: 차상길 교수(KAIST)

[초청발표] LOSVER: Line-Level Modifiability Signal-Guided Vulnerability Detection and Classification,
남도하, 백종문(KAIST), The 40th IEEE/ACM International Conference on Automated Software Engineering (ASE 2025) 국제학술대회 발표 논문
발표자: 남도하(KAIST)

AutoFiC: 취약점 탐지부터 PR 생성까지 자동화된 보안 패치 파이프라인 (학부생논문)
장인영(덕성여대), 오정민(가천대), 김민채(국민대), 김은솔(명지대)

바이너리 코드 (역)어셈블러 자동 생성 방법에 대한 탐구 (단편논문)
김지훈, 정승일, 김준태, 차상길(KAIST)

세션 B3. SW 결함 II

- 2026년 2월 5일 (목) 오후 1:00-1:50 / 112호
- 좌장: 남재창 교수(한동대)

검색 전략이 LLM 기반 버그 리포트 자동 생성 성능에 미치는 영향 분석 (일반논문)
최서진, 양근석(한경국립대)

Code LLaMA 를 활용한 자연어 프롬프트 기반의 소프트웨어 결함 예측 (단편논문)
김민재, 류덕산(전북대)

In-Context Learning 기반 표형 Foundation Model(TabICL)을 활용한 소프트웨어 결함 예측 (단편논문)
심은진, 류덕산(전북대)

세션 B4. 응용 SW

- 2026년 2월 5일 (목) 오후 1:00-1:50 / 106호
- 좌장: 김기섭 교수(DGIST)

생성형 AI 문서 검토에서 출처 라벨이 사용자 판단과 오류 탐지 수행에 미치는 영향 (일반논문)
민소원(KAIST)

EDGE 컴퓨팅 기기에서의 소형 객체 탐지를 위한 선택적 혼합 정밀도 기반 양자화 모델 성능 개선 연구 (단편논문)
임다희, 박지훈(충남대)

도메인 특화 임베딩 학습을 활용한 한국어 법률 질의응답 RAG 시스템 최적화 연구 (산업체논문)
배소연(딥모달), 장진우, 이주형(미디어젠), 박진경(공정거래위원회)

세션 C1. 프로그램 수정

- 2026년 2월 5일 (목) 오후 2:00-3:00 / 102호
- 좌장: 김윤호 교수(한양대)

증거 생성과 구조적 제약을 결합한 자동 프로그램 정정 기법 (일반논문)
이현수, 양근석(한경국립대)

LLM 기반 프로그램 자동 수정을 위한 코드 그래프 활용 방식 비교 (일반논문)
강신엽, 이지광, 권혁민, 남재창(한동대)

프로그램 실행 중 발생한 버그의 실시간 자동 수정 (일반논문)
노준영, 김영재(UNIST)

세션 C2. SW 보안 II

- 2026년 2월 5일 (목) 오후 2:00-3:00 / 111호
- 좌장: 차상길 교수(KAIST)

[초청논문] Automated Attack Synthesis for Constant Product Market Makers,
한수진, 김진서, 이성주, 윤인수(KAIST), The 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2025) 국제학술대회 발표 논문
발표자: 한수진(KAIST)

LLM 질의를 통한 정적 오염분석 허위경보 제거 (일반논문)
주강대, 조한결, 이우석(한양대)

BERT 기반 웹쉘 탐지 모델 성능 평가 (단편논문)
백하현, 정승욱, 남재창(한동대)

세션 C3. SE 를 위한 AI II

- 2026년 2월 5일 (목) 오후 2:00-3:00 / 112호
- 좌장: 남재창 교수(한동대)

EnvAgent: AI/ML 프로젝트의 Conda 환경 자동 구축 시스템 (단편논문)
권혁민, 이지광, 강신엽, 정용빈, 남재창(한동대)

[초청발표] Beyond pip install: Evaluating LLM agents for the automated installation of Python projects,
Louis Milliken, 강성민, 유신(KAIST), The 32nd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2025) 국제학술대회 발표 논문
발표자: 유신(KAIST)

프로젝트 구조 요약을 통한 대규모 언어 모델의 구조적 한계 보완 가능성에 대한 실험적 연구 (학부생논문)
이석인, 이선아(경상대)

세션 C4. 제조 AI 를 위한 SE

- 2026년 2월 5일 (목) 오후 2:00-3:00 / 106호
- 좌장: 김기섭 교수(DGIST)

치공구 설계를 위한 RAG-MCP 기반 멀티 에이전트 FreeCAD 오토코딩 시스템 (산업체논문)
이의천, 고성진, 이선아(경상국립대), 이석원((주)씨엘디)

대규모 언어 모델 기반 무인공장 작업 정책 자동 생성 (일반논문)
주은정, 이정화(미라클에이지아이), 류덕산(전북대), 백종문(KAIST)

X-RAD Engineering Recipe: 하이브리드 그래프와 2 단계 책임 분리를 통한 설명가능한 이상 탐지 (일반논문)
김민지, 허대영(국민대)

세션 D1. SW 테스팅 II

- 2026년 02월 06일 (금) 오전 10:15-11:45 / 102호
- 좌장: 손정주 교수(경북대)

[초청발표] How Effective are Large Language Models in Generating Software Specifications?,
류병우, 김미정(UNIST), Danning Xie, Nan Jiang, Lin Tan, Xiangyu Zhang(Purdue University), The 32nd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2025) 국제학술대회 발표 논문
발표자: 류병우(UNIST)

질의-응답 프롬프트 기반 실용적인 테스트 오라클 생성 (일반논문)
정지나, 김윤호(한양대)

상태 공간 모델 기반 오프라인 강화학습의 강건성 테스팅 (단편논문)
한태현, 김장환, 김영철(홍익대)

모델체킹을 위한 강화학습 기반 휴리스틱 학습 (단편논문)
강혜윤, 손병호, 배경민(POSTECH)

[초청발표] TopSeed: Learning Seed Selection Strategies for Symbolic Execution from Scratch,
이재혁, 차수영(성균관대), IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025) 국제학술대회 발표 논문
발표자: 이재혁(성균관대)

세션 D2. SW 디버깅

- 2026년 02월 06일 (금) 오전 10:15-11:45 / 111호
- 좌장: 지은경 교수(KAIST)

[초청발표] Collaboration failure analysis in cyber-physical system-of-systems using context fuzzy clustering, 현상원(University of Adelaide), 지은경, 배두환(KAIST), Empirical Software Engineering(EMSE 2025) 국제저널 발표 논문
발표자: 현상원 (University of Adelaide)

딥러닝 기반 국방 SW 결함위치추정을 위한 변이 기반 데이터셋 구축의 체계적 연구 (단편논문)
양희찬, 이아청(KAIST), 조규태(LIGNex1), 김문주(KAIST/브이플러스랩)

이슈 설명과 심볼 수준의 목표를 활용한 분류 기반 결함 위치 식별 (단편논문)
Aslan Safarovich Abdinabiev, 홍수지, 이병정(서울시립대)

실행 컨텍스트 정합성에 기반한 LLM 결함 위치 추정 결과의 안정화 기법 (일반논문)
남규민, 최서진, 양근석(한경국립대)

Diff-only 환경에서 단일 에이전트와 다중 에이전트 기반 커밋 메시지 생성의 비교 분석 (일반논문)
안도경, 양근석(한경국립대)

세션 D3. 프로그램 분석 및 모델링

- 2026년 02월 06일 (금) 오전 10:15-11:45 / 112호
- 좌장: 배경민 교수(POSTECH)

바이트코드 의미 보존을 위한 그래프 생성 기법 (단편논문)
권민하, 정용빈, 정승욱, 정다훈, 남재창(한동대)

[초청발표] Forcrat: Automatic I/O API Translation from C to Rust via Origin and Capability Analysis,
홍재민, 류석영(KAIST), The 40th IEEE/ACM International Conference on Automated Software Engineering (ASE 2025) 국제학술대회 발표 논문
발표자: 홍재민(KAIST)

컨트롤러 소프트웨어의 속성 검사를 위한 상태 기반 테스트 생성 기법의 평가 (일반논문)
Za Vinh Le(경북대), 홍신(충북대), 최윤자(경북대)

반응형 시스템을 위한 LLM 기반 상태머신 생성 기법 및 성능평가 (단편논문)
최승빈(소프트웨어재난연구센터), 김요엘, 최윤자(경북대)

통합 그래프 신경망을 통한 계층적 멀티모달 코드 표현 학습 (일반논문)
Junaid Khan Kakar, Faisal Mohammad, 류덕산(전북대)

KCSE 2025 튜토리얼

튜토리얼 T1.

- 일시: 2026년 2월 04일(수) 오후 1:00-3:30
- 장소: 102호 / 좌장: 배경민 교수(POSTECH)

- 제목: AI Red Teaming Toward AI Alignment
- 연사: 박상돈 교수(POSTECH)
- 요약

  우리는 생성형 AI 가 선보이는 놀라움 속에 살고 있습니다. 성능이 뛰어난 생성형 AI 는 지식 베이스, 웹 검색, 개인화된 에이전트, 예술, 코딩, 컴퓨터 보안 등 다양한 분야에서 그 가능성을 대중에게 선보이고 있습니다. 그러나 빛이 밝을수록 그림자는 더욱 짙어집니다. 생성형 AI 는 환각 현상, 편향된 생성, 개인정보 침해 우려, 유해한 콘텐츠 생성 등의 문제로 비판을 받아왔습니다. 이렇게 신뢰할 수 없고 인간의 가치에 정렬되지 않은 생성형 AI 에게 우리의 일상을 공유하는 것을 재고할 필요가 있습니다. 본 튜토리얼에서는 이런 생성형 AI 의 문제를 적극적이고 능동적으로 평가하는 AI Red Teaming 기법의 역사 및 현재 트렌드를 공유하고자 합니다.
- 약력

  박상돈 박사는 POSTECH GSAI/CSE 의 조교수입니다. 그의 연구 관심사는 이론부터 구현까지 아우르는 접근을 통해 인간의 가치에 정렬된 신뢰할 수 있는 AI 시스템을 설계하고, 이를 컴퓨터 보안 및 로보틱스 등 다양한 실제 응용 분야에 적용하는 데 있습니다. 그는 탑티어 머신러닝 국제학회인 NeurIPS, ICLR, ICML 에서 Area Chair 로 활동하고 있습니다. POSTECH 에 합류하기 전에는 조지아 공과대학교(Georgia Institute of Technology)에서 박사후연구원으로 근무하였고, 2021 년 펜실베이니아 대학교(University of Pennsylvania)에서 Computer & Information Science 박사 학위를 취득하였습니다.

튜토리얼 T2.

- 일시: 2026년 2월 04일(수) 오후 1:00-3:30
- 장소: 111호 / 좌장: 류덕산 교수(전북대)

- 제목: 원자력 분야 소프트웨어 안전성 확인 및 검증: 기준, 절차, 방법, 사례 및 현안
- 연사: 지은경 교수(KAIST)
- 요약

  원자력 분야에서 발전소 계측제어 시스템 등 안전 기능 수행에 직접적으로 관여하거나 안전성에 영향을 미칠 수 있는 소프트웨어에 대해서는 규제 기관이 제시하는 다양한 국내외 기준과 표준을 충족해야 하며, 개발 전 생명주기에 걸쳐 체계적인 검증 절차와 객관적인 증거 확보가 필수적이다. 본 튜토리얼은 엄격한 안전성 확인 및 검증(Verification and Validation, V&V)이 요구되는 소프트웨어의 경우, 어떤 기준에 따라 어떤 활동들이 요구되는지, 절차나 방법은 어떠한지, 어떤 현안들이 있는지를 구체적 사례와 함께 다룬다. 본 튜토리얼은 원자력 분야 소프트웨어 안전성 확인 및 검증에 대한 입문 수준의 체계적인 이해를 제공하는 것을 목표로 한다. 원자력 시스템에서 소프트웨어가 수행하는 역할과 안전성 개념을 소개하고, IAEA, IEC, IEEE 등 국제 기준과 국내 규제 체계를 중심으로 원자력 소프트웨어에 적용되는 주요 기준과 요구사항을 설명한다. 이어서 V-모델 기반 개발 생명주기에서의 검증 절차, 독립적 확인 및 검증(IV&V)의 개념, 그리고 대표적 소프트웨어 안전성 확인, 검증 방법들을 소개한다. 실제 적용 사례를 중심으로, 연구·산업 현장의 경험을 바탕으로 실무적인 관점을 공유하고, FPGA, AI 기반 소프트웨어 등 신기술 도입에 따른 규제적 현안과 연구 주제들도 함께 논의한다. 본 튜토리얼은 소프트웨어 안전성 확보가 중요한 분야에서 실무가 어떻게 이루어지는지 관심 있는 학부생, 대학원생, 그리고 관련 연구를 시작하려는 연구자들에게 유용한 기초 지식과 연구 방향성을 제공할 것이다.
- 약력

  지은경 교수는 KAIST 전산학부 연구부교수로 재직 중입니다. 안전 중요 소프트웨어, 소프트웨어 테스팅, 정형 검증, 안전성 분석, 소프트웨어 신뢰성, AI 시스템 안전성 등의 주제들에 대해 연구 중입니다.

튜토리얼 T3.

- 일시: 2026년 2월 04일(수) 오후 1:00-3:30
- 장소: 112호 / 좌장: 백종문 교수(KAIST)

- 제목: RoboRacer AI 경주 로봇 개발의 SW공학적 접근
- 연사: 김진현 교수(경상대)
- 요약

  본 튜토리얼은 RoboRacer 기반의 AI 경주 로봇(소형 자율주행 레이싱 플랫폼)을 대상으로, 자율주행 시스템을 "구현 가능한 제품 수준"으로 끌어올리기 위한 소프트웨어공학적 개발·검증·테스트 접근을 소개합니다. RoboRacer 는 교육과 연구를 동시에 겨냥한 경주형 자율주행 프로그램으로, 제한된 센서·연산 자원과 높은 동역학적 요구 조건 하에서 인지–계획–제어 파이프라인을 안정적으로 통합해야 합니다. 이러한 특성 때문에, 단순히 알고리즘을 "동작"시키는 것을 넘어, 재현성·안전성·성능(랩타임)·견고성을 함께 만족시키는 체계적 개발 방법론이 필수적입니다. 먼저, RoboRacer 자율주행 및 경주-교육 프로그램의 목표, 운영 방식, 플랫폼 구성(차량, 센서, 컴퓨팅, 시뮬레이터/트랙 환경)과 같은 실무적 맥락을 소개합니다. 이어서 자율주행의 핵심 파이프라인을 구성하는 대표 알고리즘들을 레이싱 환경에 적합한 관점에서 정리합니다. 구체적으로, LiDAR 기반 인지 및 로컬라이제이션, 주행 가능 영역/장애물 표현(코스트맵·거리장 등), 경로 생성 및 속도 프로파일링, 그리고 추종 제어(Pure Pursuit/MPC/MPPI 등)를 하나의 시스템으로 결합하는 과정에서 발생하는 설계 이슈를 다룹니다. 그 다음, 본 튜토리얼의 중심으로서 RoboRacer 자율주행을 SW 공학적으로 설계·검증·테스트하는 방법을 제시합니다. 여기에는 (1) 모듈 경계와 인터페이스를 명확히 하는 아키텍처 설계, (2) 시뮬레이션 기반 회귀 테스트와 로그 기반 디버깅, (3) 안전 요구사항과 성능 지표를 동시에 다루는 평가 메트릭 설계, (4) 형식기법 및 V&V(Verification & Validation)를 활용한 신뢰성 강화 전략이 포함됩니다. 또한 최근 각광받는 에이전트형 자동화(예: 테스트 생성–실행–분석–수정 루프)가 로봇 레이싱 시스템 개발에서 어떤 장점과 한계를 갖는지도 함께 논합니다. 마지막으로 실제 RoboRacer 자율주행의 통합 시연을 통해, 동일한 기능을 구현하더라도 테스트 가능성(testability), 관측 가능성(observability), 실패 분석 가능성을 어떻게 설계로부터 확보하는지가 성능과 안정성을 좌우함을 보여줍니다. 본 튜토리얼은 로봇/자율주행 알고리즘에 익숙하지만, 제품 수준의 SW 공학적 개발·검증 체계는 이제 확립하고자 하는 참가자들을 주요 대상으로 합니다.

- 약력

  김진현 교수는 경상국립대학교 AI 정보공학과 부교수로 재직 중입니다. 정형기법(Formal Methods), AI Safety, Medical LLM, Robot Racing 을 중심으로, CPS(사이버물리시스템)의 검증·검증가능 설계 및 의료/자율주행 분야의 신뢰 가능한 AI 개발에 관심을 두고 있습니다. 최근에는 도메인 특화 의료 언어모델 Ophtimus(안과 특화 SLM/LLM) 개발을 포함하여, 안전하고 신뢰 가능한 의사결정 지원을 위한 방법론을 연구하고 있습니다. 또한 UPenn PRECISE Lab, KAIST, Aalborg University, INRIA/IRISA 등과 협력해 왔으며, Kim G. Larsen, Axel Legay, Sungwon Kang 교수 등과의 연구 협업 경험이 있습니다.

KCSE 2025 기조강연

기조강연 I

- 일시: 2026년 2월 04일(수) 오후 4:00-5:00
- 장소: 112호 / 좌장: 김미정 교수(UNIST)

- 제목: AI-Native DevOps: What Should We Do — Accelerating Software Engineering with Shift-Left, CI/CD, and IDP
- 연사: 박인욱 상무(LG전자)

- 요약

  AI-Native DevOps 의 핵심은 "새로운 도구 나열"이 아니라 기존 SW 공학의 가속과 표준화다. SW 개발의 핵심인 Shift-Left 와 CI/CD 전략을 개발자 역량의 연장선으로 보고, 이를 포함한 SDLC 를 IDP(Internal Developer Platform) 위에서 AI 지원을 통해 일관된 가드레일과 골든 패스로 제공하는 방향을 제시한다. 결과적으로, 사람의 역량 × IDP × AI 의 증폭으로 속도와 신뢰를 동시에 끌어올리는 실행 프레임을 제시한다.
- 약력

  박인욱 상무는 현재 LG 전자 webOS SW 개발그룹 DevOps 개발실장으로, 제조사에 DevOps 를 도입하여 정착·고도화하고, webOS Re:New 업그레이드 프로그램과 DevSecOps·플랫폼 엔지니어링·AI 기반 생산성 혁신을 이끌고 있다.

기조강연 II

- 일시: 2026년 2월 05일(목) 오후 4:30-5:30
- 장소: 112호 / 좌장: 이정원 교수(아주대)

- 제목: 재사용을 위한 소프트웨어 산출물 모델링
- 연사: 이희조 교수(고려대)

- 요약

  소프트웨어 공급망 공격이 증가하면서, 소프트웨어는 더 이상 "잘 동작하는가"만으로 평가되기 어려워졌다. 최근 미국과 유럽을 중심으로 강화되고 있는 제품 보안 및 공급망 보안 규제는, 소프트웨어가 "무엇으로 구성되어 있는지(SBOM)"와 발견된 취약점이 실제로 어떤 영향을 미치는지를 설명할 것을 요구하고 있다.
- 약력

  이희조 교수는 고려대학교 정보대학 컴퓨터학과 교수이자 소프트웨어보안연구소(CSSA) 연구소장으로, 지난 20 여 년간 국가·산업·학계를 아우르며 공급망 보안과 취약점 분석 기술 연구, 보안 인재 양성에 힘써왔다. 안랩 CTO 를 역임했으며, 현재는 고려대학교 기술지주 자회사 ㈜래브라도랩스 공동대표로서 SBOM 및 오픈소스 보안 솔루션의 확산을 추진하고 있다.

기조강연 III

- 일시: 2026년 2월 06일(목) 오전 9:00-10:00
- 장소: 112호 / 좌장: 이주용 교수(UNIST)

- 제목: 재사용을 위한 소프트웨어 산출물 모델링
- 연사: 최재식 교수(KAIST)

- 요약

  대형인공지능모델(혹은 대형언어모델, LLMs: Large Language Models)이 많은 응용 분야에 활용되고 있다. 이런 LLM 의 장점과 함께 거짓답변(hallucination)과 같은 문제는 근본적인 원인 뿐만 아니라 해결이 어려운 면이 있다. 최근 설명가능 인공지능의 발전은 이런 LLM 내부를 확인하고 그 의사 결정을 명확히 하는데 기여하고 있다. 이 강의에서는 이런 설명성 기술이 LLM 에 적용되는 최근 기술과 연구 동향을 소개한다.
- 약력

  최재식 교수는 KAIST 김재철 AI 대학원 석좌교수(㈜인이지 대표이사)이자 설명가능 인공지능(XAI) 연구자로, 한국공학한림원 회원으로 활동하며 산업부 제조 AI 전환(M.AX) 얼라이언스 총괄위원으로서 국가 AI 정책 수립과 제조업 AI 전환에 기여하고 있다.

KCSE 2025 신진 연구자 초청 발표

신진 연구자 초청 발표 N1

- 일시: 2026년 2월 05일(목) 오후 9:00-9:50
- 장소: 102호 / 좌장: 김윤호 교수(한양대)

- 제목: 대규모 소프트웨어에서 바늘을 찾다: LLM 기반 결함 위치 식별의 산업 적용
- 연사: 안가빈 교수(고려대)

- 요약

  대규모 소프트웨어에서 결함 위치를 찾는 일은 여전히 큰 도전 과제이다. 본 강연에서는 LLM 에이전트를 활용한 AutoFL 시리즈를 통해, 설명 가능한 결함 위치 식별 연구가 어떻게 산업 규모 소프트웨어의 크래시 분석으로 확장될 수 있는지를 소개한다. 이를 통해 LLM 기반 자동 디버깅 기법의 가능성과 한계를 함께 논의한다
- 약력

  안가빈 교수는 고려대학교 정보대학 컴퓨터학과 조교수로, AI 기반 소프트웨어 공학 연구를 주로 수행하고 있다. 주요 연구 분야는 대형 언어 모델을 활용한 자동 디버깅, 결함 위치 식별, 그리고 소프트웨어 테스트 및 유지보수 자동화이다. 산업 규모의 실제 소프트웨어 시스템을 대상으로 신뢰 가능한 자동 분석 기법을 개발하는 데 주력하고 있다.

신진 연구자 초청 발표 N2

- 일시: 2026년 2월 05일(목) 오후 9:00-9:50
- 장소: 111호 / 좌장: 손정주 교수(경북대)

- 제목: 컨테이너 기반 클라우드 환경을 위한 RDMA-aware CNI
- 연사: 유명성 교수(서울시립대)

- 요약

  분산 마이크로서비스 및 AI 워크로드를 위한 RDMA 채택이 늘고 있으나, 커널 바이패스 특성으로 인한 가시성 부재와 기존 TCP/IP 애플리케이션과의 호환성 문제가 존재한다. 본 강연에서는 유저 공간 eBPF 런타임을 활용해 컨테이너 단위의 RDMA 활동을 정밀 모니터링하고, eBPF 기반 트래픽 처리를 통해 코드 수정 없이도 기존 컨테이너의 통신 성능을 크게 향상시키는 RDMA-aware CNI 기술을 소개한다.

- 약력

  유명성 교수는 서울시립대학교 전자전기컴퓨터공학부 조교수로, 차세대 네트워크 및 시스템 연구실(TNS Lab)을 운영하고 있다. 주요 연구 분야는 고성능 네트워킹(RDMA), 클라우드 시스템 보안, 그리고 AI 기반 소프트웨어 공학이다. 최근에는 LLM 을 활용한 네트워크 프로그램의 자동 생성 및 AI 모델의 보안 취약점(Jailbreaking) 분석 연구를 수행하며, 시스템 네트워킹 기술과 소프트웨어 자동화 및 보안 기술을 융합한 신뢰 가능한 컴퓨팅 인프라 구축에 주력하고 있다.

신진 연구자 초청 발표 N3

- 일시: 2026년 2월 05일(목) 오후 9:00-9:50
- 장소: 112호 / 좌장: 지은경 교수(KAIST)

- 제목: 로보틱스 소프트웨어의 원인-결과 체인 지연시간 검증
- 연사: 신용준 박사(ETRI)

- 요약

  로보틱스 소프트웨어의 원인-결과 체인 지연시간 검증은 시스템의 반응성 및 안전성 보장을 위해 필수적이나, 체인 구성 요소의 실행 비결정성과 소스코드 비가시성은 신뢰 가능한 검증을 어렵게 만든다. 본 강연에서는 이러한 어려움을 고려한 (1) 원인-결과 체인 지연시간 검증을 위한 모델 기반 접근, (2) 블랙박스 원인-결과 체인의 런타임 지연시간 검증 알고리즘, (3) ROS 2 소프트웨어를 대상으로 한 런타임 검증 도구를 소개한다.

- 약력

  신용준 박사는 한국과학기술원(KAIST)에서 공학박사 학위를 취득하고, 현재 한국전자통신연구원(ETRI)에 선임연구원이자 AI 전문교수로 재직 중이며 로보틱스 및 모빌리티 소프트웨어의 안전성 검증 기술을 연구하고 있다. 주요 연구 관심 분야는 자율행동체를 위한 모델 기반 소프트웨어 공학, 런타임 검증 및 안전 제약, 지속적 통합/배포 등이다.

우수 국제학회/학술지 초청 논문발표

◆ **An Empirical Study of Web Flaky Tests: Understanding and Unveiling DOM Event Interaction Challenges**

- The 18th IEEE International Conference on Software Testing, Verification and Validation (ICST 2025)
- Yu Pei(University of Luxembourg), 손정주(경북대), Mike Papadakis(Unibersity of Luxembourg)
- Session A1. SW 테스팅

◆ **Automated Attack Synthesis for Constant Product Market Makers**

- The 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2025)
- 한수진, 김진서, 이성주, 윤인수(KAIST)
- Session B1. SW 안전과 보안

◆ **Automated code-based test case reuse for software product line testing**

- ICST 2024 Journal First
- 정필수, 이선아, 이의천(경상국립대)
- Session A1. SW 테스팅

◆ **Beyond pip install: Evaluating LLM agents for the automated installation of Python projects**

- The 32nd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2025)
- Louis Milliken, 강성민, 유신(KAIST)
- Session C3. SE 를 위한 AI II

◆ **Can We Trust the Actionable Guidance from Explainable AI Techniques in Defect Prediction?**

- The 32nd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2025)
- 이기찬, 주한세, Scott Uk-Jin Lee(한양대)
- Session A3. SW 결함 I

◆ **Collaboration failure analysis in cyber-physical system-of-systems using context fuzzy clustering**

- Empirical Software Engineering (EMSE 2025)
- 현상원(University of Adelaide), 지은경, 배두환(KAIST)
- Session D2. SW 디버깅

◆ **Forcrat: Automatic I/O API Translation from C to Rust via Origin and Capability Analysis**

- The 40th IEEE/ACM International Conference on Automated Software Engineering (ASE 2025)
- 홍재민, 류석영(KAIST)
- Session D3. 프로그램 분석 및 모델링

◆ **Fork State-Aware Differential Fuzzing for Blockchain Consensus Implementations**

- IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025)
- 김원회, 남호철(KAIST), Muoi Tran(ETH Zurich), Amin Jalilov(KAIST), Zhenkai Liang(National University of Singapore), 차상길(KAIST), 강민석(National University of Singapore)
- Session A2. 퍼징 및 기호실행

◆ **How Effective are Large Language Models in Generating Software Specifications?**

- The 32nd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2025)
- 류병우, 김미정(UNIST), Danning Xie, Nan Jiang, Lin Tan, Xiangyu Zhang(Purdue University)
- Session D1. SW 테스팅 II

◆ **Lightweight Concolic Testing via Path-Condition Synthesis for Deep Learning Libraries**

- IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025)
- 김세훈, 김용현, 박다현, 전유석, 이주용, 김미정(UNIST)
- Session A2. 퍼징 및 기호실행

◆ **LOSVER: Line-Level Modifiability Signal-Guided Vulnerability Detection and Classification**

- The 40th IEEE/ACM International Conference on Automated Software Engineering (ASE 2025)
- 남도하, 백종문(KAIST)
- Session B2. SW 보안 I

◆ **TopSeed: Learning Seed Selection Strategies for Symbolic Execution from Scratch**

- IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025)
- 이재혁, 차수영(성균관대)
- Session D1. SW 테스팅 II

KCSE 2026 행사장위치

KCSE 2026 논문 목록

일반 논문

단편 논문

학부생 논문

산업체 논문

기기의 건전성 테스트를 위한 누적손실기반 평가 메트릭 설계 및 검증

최민서 O[1], 김진세 [1], 이정원 [1, 2]

[1] 아주대학교 AI 융합네트워크학과, [2] 아주대학교 전자공학과

minseo24@ajou.ac.kr, Jinsae913@gmail.com, jungwony@ajou.ac.kr

Cumulative Loss based Evaluation Metric for Machinery Health Test: Definition and Verification

Minseo Choi O[1], Jinse Kim[1], Jung-Won Lee[1, 2]

[1]Department of AI Convergence Network, Ajou University

[2]Department of Electrical and Computer Engineering, Ajou University

요 약

스마트팩토리의 핵심 요소인 산업용 기기의 예기치 못한 고장은 막대한 경제적/인적 손실을 초래할 수 있다. 이를 방지하기 위한 기존의 건전성 테스트 기법은 평가 시점 데이터에만 의존하여 점진적인 저하 경향성을 반영하지 못하고 데이터 내 단발적인 이상치에 취약하다. 따라서, 본 논문은 산업용 기기의 건전성 테스트를 위해 누적손실기반 평가 메트릭을 정의하고 그 효과를 검증한다. 제안 메트릭은 과거 평가 주기의 모든 손실 값을 누적하고 평균화하여 과거의 기기 상태 이력을 반영하고, 단발적 이상치의 영향을 완화하여 높은 정확도의 건전성 테스트를 가능하게 한다. 유압 설비의 상태 모니터링을 위한 오픈 데이터셋 기반의 검증 결과, 기존 손실 메트릭 대비 최대 79.91%의 향상된 건전성 테스트 정확도를 달성함으로써 제안 메트릭의 효용성을 입증하였다. 또한, 협동 로봇 기반 사례 연구 결과, 기존 메트릭 대비 건전성 저하 경향성을 명확하게 식별할 수 있음을 확인함으로써 그 실효성을 검증하였다.

1. 서 론

협동 로봇, 밀링 머신, 유압 설비와 같은 산업용 기기(Industrial Machinery)는 제조업, 건설업 등의 다양한 산업 분야에서 공작이나 운반과 같은 특정 작업을 수행하도록 설계된 기계 또는 장치를 의미한다[1]. 이는 스마트팩토리의 필수 조건인 자동화된 공정 환경을 구성하기 위한 핵심 요소로 여겨지며 다양한 산업 분야에서 불가결한 요소로 정착되어 왔다[2]. 산업용 기기는 고속 작업 수행이 가능하고 반복 작업을 자동화할 수 있어 적용 산업의 생산 효율을 크게 향상시키며 이에 따라 보급에 대한 수요가 지속적으로 증가하고 있다. 그러나, 산업용 기기에서 예기치 못한 고장이 발생할 경우 오작동, 전류 누설과 같은 고장 현상에 의해 협착, 화재 등의 중대 산업 재해가 발생할 수 있다. 이는 다운타임(Downtime) 증가에 따른 경제적 손실과 함께 심각한 인명 피해를 야기한다[3], [4]. 따라서, 이와 같은 경제적/인적 손실을 사전에 방지하기 위해 산업용 기기의 초기 대비 건강 상태를 의미하는 건전성의 저하를 평가하고 추적할 수 있는 방법이 필수적으로 요구된다.

산업용 기기 내부에는 고정밀도 작업 수행을 위한 다양한 센서(예: 전류 센서, 가속도 센서)가 탑재되어 있으며, 이러한 센서로부터 추출되는 센서 데이터는 수집 시점에 대한 기기의 상태 정보를 내재한다. 이에 따라 산업용 기기의 건전성 테스트 연구는 대부분 내부 센서 데이터를 기반으로 수행되고 있다[5], [6]. 건전성 테스트를 위한 초기 연구는 주로 정상/저하/고장 상태로 레이블링된 센서 데이터를 활용하여 상태 분류 기준을 수립하고 이를 기반으로 건전성을 평가하거나, RTF(Run-To-Failure) 데이터를 통해 고장 임계점을 사전 정의하여 해당 임계점을 기준으로 평가 시점의 상대적인 건전성 저하 정도를 추적하는 방식으로 수행되었다[7], [8]. 이와 같은 연구는 사전 정의된 실제 상태 정보를 기반으로 건전성을 명확히 평가할 수 있어, 엔진, 베어링 등의 다양한 산업용 기기 및 요소에 적용되었으며 그 효용성이 검증되었다 [8], [9]. 그러나, 산업용 기기의 건전성 저하는 단기간에 발생하는 것이 아닌 점진적으로 장기간에 걸쳐 진행되는 연속적인 과정이기에 고장 데이터 수집이 어려우며, 기존 방식과 같이 이산적인 상태 정의만으로는 저하 과정을 설명하는 데에 한계가 있다. 이에 따라 최근 연구에서는 정상 상태 데이터와 비교 시점 데이터 간의 패턴 차이를 정량화하고 이를 건전성 테스트 메트릭으로

[1] 이 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. 2023R1A2C1006332).

활용하여 저하를 평가하고 추적하는 방법이 제안되었다 [10]. [11].

정상 상태 데이터와 평가 데이터의 패턴 차이를 정량화하는 통계적인 방법으로는 MSE(Mean Squared Error), MAE(Mean Absolute Error)와 같은 손실 함수와 손실 함수의 기저/확장 형태인 거리 기반의 데이터 유사도 측정 메트릭(예: Mahalanobis Distance, Dynamic Time Warping)이 있다. 또한, 머신러닝/딥러닝 관점의 방법으로는 정상 데이터로 학습한 재구성 모델을 통해 입력-복원 데이터간 패턴 차이를 손실 함수로 수치화하는 방법이 존재한다. 이와 같은 접근법은 정상 상태 데이터만을 활용하여 저하 과정에서 나타나는 데이터 패턴의 변화를 효과적으로 포착할 수 있어 다양한 기기에 적용되어왔으며 그 실효성이 검증되었다[11], [12].

그러나, 기존의 손실 활용 방법은 주로 정상 시점과 평가 시점에 해당하는 데이터만을 활용하여 패턴 차이를 산출한다. 이에 따라, 그림 1과 같이 저하 진행과 무관한 통신 오류, 충격 등에 의한 단순 이상치(Anomaly)가 저하 특징으로 식별되어 잘못된 건전성 테스트가 수행될 위험이 있다. 또한, 산업용 기기의 성능 저하는 장기간의 부하 누적에 의해 점진적으로 발생하므로 과거의 가동 패턴을 고려하지 않은 단일 시점의 비교 값만으로는 장기적인 저하 경향성과 단발적인 이상 현상을 구분하기 어렵다. 이 같은 문제를 완화할 수 있는 특정 구간의 평균 수준 추정 기법인 이동평균선(Moving Average, MA)과 과거 관측 값에 지수 가중치를 부여하여 변화 경향을 평가하는 지수 이동 평균(Exponential Moving Averame, EMA) 추세 분석 방법도 존재한다. 그러나, 이동평균선은 고정된 구간 내의 값만을 통해 관측 시점 데이터를 평가하여 과거 시점의 정보가 소실되는 문제가 있다. 또한, 지수 이동 평균은 최근 관측값에 큰 가중치를 부여하기 때문에 과거의 영향성이 매우 작아진다. 이같은 점을 고려할 때, 높은 정확도의 건전성 테스트를 위해서는 점진적인 저하로 인해 발현되는 데이터 패턴의 차이뿐만 아니라 과거부터 관측 시점까지의 변화 경향성을 함께 고려할 수 있는 새로운 건전성 테스트 메트릭이 필요하다.

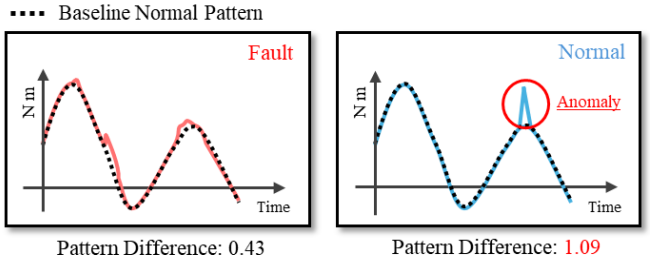


그림 1. 정상 기준 대비 정상/단순 이상치 데이터의 패턴 차이

따라서, 본 논문은 산업용 기기의 효과적인 건전성 테스트를 위해 저하에 따른 데이터 패턴의 변화 경향성을 반영할 수 있는 누적손실기반의 새로운 건전성 테스트 메트릭을 제안한다. 과거부터 평가 시점까지의 평가 주기별 손실 값을 누적한 후 평균화함으로써 시간 흐름에 따른 센서 데이터 패턴의 변화 추세를 효과적으로 반영할 수 있으며, 단발적인 이상치의 영향을 완화할 수 있다. 이를 통해 단일 시점의 손실 값만을 활용하여 평가하는 기존의 건전성 테스트 방법 대비 높은 정확도의 테스트를 가능하게 하며, 센서 데이터 수집이 가능한 산업용 기기에 제약없이 적용 가능하다.

본 연구의 효용성을 평가하기 위해, 유압 설비의 요소별 상태 모니터링을 위해 구축된 오픈 데이터셋을 활용하여 검증을 수행하였다. 건전성 수준별 평가 실험 결과, 최소 86.22%에서 최대 100%의 높은 평가 정확도를 보였으며, 기존의 손실기반 평가 방법 대비 최대 79.91%의 성능 향상을 보임에 따라 건전성 테스트 메트릭로서의 충분한 효용성을 지님을 확인하였다. 또한, 실제 산업용 기기를 대상으로 한 사례 연구 결과, 데이터 내 단발적 이상치의 영향이 완화되어 손실 값의 변동성이 감소하였으며, 이에 따라 건전성 저하의 경향성을 보다 명확하게 식별할 수 있음을 확인함으로써 건전성 테스트 메트릭으로서의 실효성을 검증하였다.

2. 관련 연구

2.1 산업용 기기의 건전성 테스트를 위한 HI 구축

HI(Health Indicator)는 산업용 기기의 건전성을 평가하기 위한 정량적 메트릭이다. 이에 따라, 저하/고장에 대한 사전 정보 없이 저하에 의해 발현되는 센서 데이터 패턴의 변화 양상을 기반으로 HI를 구축하여 건전성을 테스트하고 고장을 예측하는 다양한 연구가 수행되었다.

초기에 제안된 HI는 주로 데이터 내의 통계적 특성(Peak, Kurtosis 등) 또는 신호 특징(주파수 영역, 시간-주파수 영역 등)을 분석하는 방식으로 구축되었다. 관련 연구로 [12]는 베어링의 저하 추적을 위해 RMS(Root Mean Square) 기반 HI 구축 방법을 제안하였다. 해당 연구는 진동 신호를 여러 주파수 대역으로 분해하고 각 대역 신호에 대한 RMS를 산출하여 단조 증가를 보이는 대역을 기기의 HI로 활용하였다. 또 다른 연구 [13]은 신호의 주파수 대역 중 유의미한 대역에 가중치를 부여하고, 이를 반영한 전체 주파수 대역을 HI로 수립하여 통계적 기준(예: 정규 분포를 따르는 정상 상태 데이터의 3σ)을 초과하는 지점을 저하의 시작 지점으로 평가하였다. 이 외에도 [14], [15], [16]은 데이터 내의 통계적 특성 및 신호 특징을 기반으로 HI를 구축하고, 이를 통해 점진적인 건전성 저하를 테스트할 수 있는 방법을

제안하였다. 이러한 연구는 통계 메트릭과 신호 특징을 기반으로 저하 특징 추출이 가능함을 보였으며 실 기기 기반 검증을 통해 그 효용성을 입증하였다. 그러나, 사전 정의된 소수의 특징만을 기반으로 저하를 식별하는 방식은 데이터에 내재된 비선형적 특징을 충분히 반영하기 어려우며 진동에 의한 노이즈나 불규칙한 이상치가 혼재될 경우 저하 패턴을 포착하는 데에 한계가 존재한다.

이와 같은 한계를 보완하기 위해 데이터의 패턴 또는 분포 차이 기반의 HI를 구축하여 건전성을 평가하고 추적하는 연구들이 수행되었다. 해당 연구들은 주로 데이터의 패턴 차이를 정량화할 수 있는 통계/거리 기반의 메트릭이나 비지도 학습 모델의 재구성 오차를 활용하여 HI를 구축하였다. [17]은 진동 데이터를 22개의 특징(예: 시간, 주파수)으로 변환하고 RMS를 예측 대상 값으로 설정하여 정상 특징 데이터로 학습된 회귀 모델의 예측 RMS 값과 실제 RMS 값의 오차로 건전성을 평가하고 추적하는 방법을 제안하였으며, 베어링 고장의 조기 진단이 가능함을 확인하여 그 실효성을 입증하였다. 또 다른 연구로 [18]은 저하 발생에 따른 진동 신호의 주파수 변화를 식별하기 위해 Log Envelope spectrum으로 변환된 정상 데이터로 VAE(Variational AE)를 학습하고, 입력-복원 데이터간 재구성 오차를 HI로 활용하여 기기의 저하를 추적하였다. 이와 같은 데이터 패턴 또는 분포차를 활용한 HI 기반 건전성 테스트 연구들은 복잡한 데이터 패턴 내에서 저하 특징을 효과적으로 추출함으로써 높은 정확도의 건전성 저하 추적이 가능함을 보였다. 그러나, 기존에 제안된 대부분의 HI 구축 방법은 과거의 가동 데이터 패턴은 고려하지 않고 평가 주기에 해당하는 데이터만을 활용하여 건전성을 테스트한다. 이에 따라 충격, 열 잡음 등의 외란으로 인해 가동 중 예기치 않게 발생하는 단발적인 데이터 이상치나 진동에도 매우 민감하게 반응하며, 실제 저하 현상이 아님에도 불구하고 저하 상태로 오판단될 위험이 존재한다.

2.2 산업용 기기의 센서 데이터 기반 건전성 테스트

산업용 기기의 건전성 테스트는 기기의 예기치 않은 고장을 방지하기 위한 핵심 기술이다. 이는 주로 기기의 내부 센서 데이터를 활용하여 수행되며, 초기에는 데이터의 통계적 특성 또는 주파수 대역 분석을 통해 건전성을 평가하고 추적하는 연구가 주를 이루었다.

관련 연구로 [19]은 진동 데이터에 웨이블릿 변환(Wavelet Transform)을 적용하여 신호를 분해하고, 그 에너지 값을 통해 퍼지 논리 기반 신경망을 학습하여 베어링 마모의 심각도를 분류하는 방법을 제안하였다. 또 다른 연구로 [20]는 기기의 비가역적 건전성 저하를 모사하기 위해 이전 시점의 평가 결과를 참조하여 현재의 건전성을 평가할 수 있는 SVM(Support Vector Machine)을 설계하였다. 3가지 상태(정상/저하/고장)로 레이블링된 냉각기 팬의 진동 데이터를 활용한 검증 실험 결과, 평균 97.39%의 정확도를 보임으로써 비가역적 특성을 갖는 건전성 저하를 효과적으로 평가할 수 있음을 확인하였다. 이 외에도 [21], [22]는 각 상태별 데이터 내에서 통계적 특성과 주파수 대역을 기반으로 유의미한 저하 특징을 추출함으로써 높은 정확도의 건전성 테스트가 가능함을 보였다. 그러나, 해당 연구들은 상태를 분류하는 것에 그치기 때문에 건전성을 평가할 수는 있으나 점진적으로 발생하는 기기의 저하 정도를 연속적으로 추적하는 데에는 한계가 존재하였다. 이와 같은 한계를 보완하기 위해 고장 임계점을 기준으로 기기의 건전성 저하를 상대적으로 정량화하고 추적하여 평가하는 [8]과 같은 연구도 제안되었으나, 고장 데이터에 대한 의존성으로 인해 고장 데이터 확보가 어려운 실제 산업환경에서는 여전히 적용에 제약이 존재한다.

따라서, 본 논문은 기기가 건전하게 동작하는 기준 상태 데이터와 평가 시점 데이터의 패턴 차이를 손실 값으로 정량화하고, 과거의 기기 상태 이력을 반영함으로써 점진적인 건전성 저하를 평가할 수 있는 새로운 평가 메트릭을 정의한다. 제안 메트릭은 단일 시점의 데이터 패턴 오차(손실)가 아닌 과거부터 누적된 손실을 기반으로 산출되어, 실제 건전성 저하로 인한 점진적 데이터 패턴 변화를 명확하게 식별할 수 있다. 이를 통해 단발적인 이상치의 영향을 완화하고 보다 높은 정확도로 건전성 테스트를 가능하게 한다.

3. 문제 정의

본 장에서는 일반적인 손실의 개념을 정의하고, 이를 산업용 기기의 건전성 테스트 메트릭으로 활용할 때의 이론적 의미와 한계를 분석한다.

3.1. 손실 정의

통계적 관점의 손실은 기준 데이터와 추정 데이터 간의 차이를 정량화한 값으로 수식 1과 같이 정의되며, 머신러닝/딥러닝 관점의 손실은 정답과 모델 기반 예측 값의 차이를 정량화한 값으로 수식 2와 같이 정의된다.

$$\boldsymbol{Loss} = \boldsymbol{L}(\boldsymbol{x^{Ref}}, \boldsymbol{x^{Est}}) \qquad \text{(수식 1)}$$

$$\boldsymbol{Loss} = \frac{\boldsymbol{1}}{\boldsymbol{n}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{L}(\boldsymbol{x_i^{GT}}, \boldsymbol{x_i^{Pred}}) \qquad \text{(수식 2)}$$

수식 1의 L은 손실 함수, x^{Ref}는 기준 데이터 그리고 x^{Est}는 추정 데이터를 의미하며, 수식 2의 L은 손실 함수, x^{GT}는 정답 데이터 그리고 x^{Pred}는 모델 기반 예측 데이터를 의미한다.

두 가지 관점에서의 손실은 모두 두 데이터 간의 패턴 차이를 평가한 값으로 해석할 수 있다는 점에서

동일한 의미를 가지며, 손실값 도출을 위해 사용 가능한 손실 함수로는 MSE, MAE(Mean Absolute Error) 등이 있다. 이와 같은 손실 함수는 데이터 간의 거리를 평가하는 함수들과 밀접한 관련이 있다. 예시로, 유클리드 거리는 MSE와 MAE의 기저 수식이 되며, 마할라노비스 거리는 MSE를 확장한 형태로 구성된다. 즉, 데이터간 거리 측정 함수 또한 손실 함수의 근간이 되거나 확장된 형태로써 그 결과가 두 데이터간 차이를 의미한다는 점에서 손실과 동일한 해석을 갖는다. 손실 값 산출을 위해 적용 가능한 손실 함수의 예시는 표 1과 같다.

표 1. 손실 함수 예시

Metric	Expression
MSE	$\frac{1}{n}\sum_{i=1}^{n}(x_i-\hat{x_i})^2$
MAE	$\frac{1}{n}\sum_{i=1}^{n}\lvert x_i-\hat{x_i}\rvert$
Euclidean Distance	$\sqrt{\sum_{i=1}^{n}(x_i-\hat{x_i})^2}$
Mahalanobis Distance	$\sqrt{\sum_{i=1}^{n}\frac{(x_i-\hat{x_i})^2}{\sigma_i^2}}$
DTW	$\min_{Path}(\sqrt{\sum_{i=1}^{n}(x_i-\hat{x_i})^2})$

3.2. 손실의 활용 및 건전성 메트릭으로서의 한계

손실은 3.1절에서 정의한 바와 같이 기준 데이터와 비교 데이터의 차이를 정량화한 값으로, 통계 모델과 머신러닝/딥러닝 모델 모두에서 데이터간 차이를 평가하는 메트릭으로 활용된다. 통계적 관점에서는 기준 데이터의 패턴과 분포를 참조하여 비교 대상 데이터와의 차이를 손실 값으로 평가한다. 또한, 머신러닝/딥러닝 관점에서는 학습 방식에 따라 손실 적용 방법이 상이한데, 지도 학습의 경우 모델의 예측 레이블과 정답 레이블 간의 차이를 손실로 산출하며, 비지도 학습의 경우에는 원본(기준) 데이터와 예측(비교) 데이터의 패턴 차이를 손실로 측정한다.

이와 같은 특성을 갖는 손실은 산업용 기기의 건전성 테스트 메트릭으로도 적용되고 있다. 통계적 관점에서는 정상 구간 데이터의 분포를 기준으로 가동 데이터와의 차이 또는 거리를 정량화한 손실을 건전성 메트릭으로 활용한다. 또한, 머신러닝/딥러닝 관점에서는 주로 비지도 학습 모델의 재구성 오차를 손실로 활용하며, 정상 데이터로만 학습한 재구성 모델에 가동 데이터를 입력한 후 입력 데이터와 복원 데이터 간의 재구성 오차(손실)를 건전성 메트릭으로 활용하여 건전성 저하를 평가하고 추적한다.

그러나, 기존의 손실 활용 방식을 건전성 테스트 메트릭으로 적용하는 데에는 두 가지 문제점이 존재한다. 기존의 접근법은 기준이 되는 정상 데이터와 평가 시점에 해당하는 데이터만을 활용하여 건전성을 평가하므로, 가동중 발생 가능한 단발적 이상치가 저하 양상으로 평가되어 오진단이 발생할 수 있다. 또한, 과거의 가동 이력이 반영되지 않기 때문에 장기간에 걸쳐 발생하는 건전성 저하의 경향성을 표현하는 데에 어려움이 있다. 예시로 그림 2는 로봇 관절의 핵심 요소인 하모닉 드라이브의 단일시점 손실기반 건전성 테스트 결과이다. 가동중 발생한 단발적인 이상치로 인해 평가 시점의 손실 값이 이후 평가일 대비 매우 큰 값으로 도출되는 현상이 관찰된다. 이로 인해, 손실 값의 변동이 과도하게 증가하며, 결과적으로 저하에 의해 발생하는 데이터 패턴 변화의 경향성을 파악하기 어려움을 확인할 수 있다.

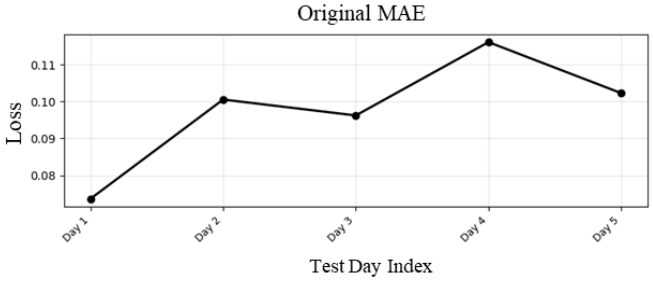


그림 2. 단일시점 손실기반 건전성 테스트 결과

4. 제안 메트릭

본 논문에서는 기존에 제안된 손실기반 건전성 테스트 메트릭의 한계를 보완하기 위해, 산업용 기기의 고성능 건전성 테스트를 가능하게 하는 누적손실기반 건전성 테스트 메트릭을 제안한다. 제안 방법은 건전성 테스트 주기별(예: 일 단위 평가 시 각 날짜) 손실을 계산하는 손실 계산 단계, 과거의 평가 주기별 손실을 순차적으로 합산하는 손실 누적 단계, 마지막으로 평가 주기 수를 기반으로 합산된 손실을 평균화하는 손실 평균화 단계로 구성된다.

(1) 손실 계산 단계

첫 번째 단계는 건전성 테스트 주기의 손실을 계산하는 단계이다. 손실은 기준 패턴과 평가 대상 데이터 간의 패턴 차이를 정량화 한 값을 의미하며, 그 방법은 3.1절에서 정의한 수식 1 또는 수식 2와 같다. 통계적 관점에서는 기준 패턴을 정상 상태의 산업용 기기에서 수집된 정상 데이터 패턴으로

설정할 수 있으며, 머신러닝/딥러닝 관점에서는 비지도 학습 모델의 입력 데이터로 설정될 수 있다.

(2) 손실 누적 단계

두 번째 단계는 첫 번째 단계에서 계산된 평가 대상 주기의 손실과 과거의 모든 평가 주기별 손실을 순차적으로 합산하는 단계이다. 합산된 손실은 하나의 단일 값으로 도출되며, 그 과정은 수식 3과 같이 정의할 수 있다. 해당 수식에서 c는 현재까지 수행된 총 평가 주기의 수를 의미한다.

$$\boldsymbol{Cumulative\ Loss} = \sum_{1}^{c} \boldsymbol{Loss_c} \qquad \text{(수식 3)}$$

(3) 누적손실 평균화 단계

마지막 단계는 두 번째 단계로부터 확보한 누적손실 값을 총 평가 주기 수로 평균화하는 단계이다. 평균화를 통해 단발적인 이상치의 영향을 완화함과 동시에 장기적인 평균 건전성 수준을 평가할 수 있으며, 그 과정은 수식 4와 같이 정의된다.

$$\boldsymbol{HI} = \frac{\boldsymbol{Cumulative\ Loss}}{\boldsymbol{c}} \qquad \text{(수식 4)}$$

이와 같은 과정을 통해 구축된 건전성 테스트 메트릭은 단발적인 이상치에 대한 영향을 완화하고 장기간에 걸쳐 발생하는 건전성 저하 경향성을 명확하게 반영함으로써 산업용 기기의 효과적인 건전성 테스트를 가능하게 한다. 그림 3은 그림 2와 동일한 손실 값을 제안 메트릭에 적용하여 건전성 테스트를 수행한 결과이다. 단일손실 메트릭을 활용한 그림 2와 달리 단발적 이상치의 영향은 완화되는 반면 저하 진행에 따른 정상 데이터와의 패턴 차이가 손실 메트릭에 점진적으로 반영되는 양상을 명확히 보여준다. 이는 제안 메트릭을 기반으로 건전성 테스트를 수행할 경우 높은 정확도로 건전성 저하 추적이 가능함을 시사한다.

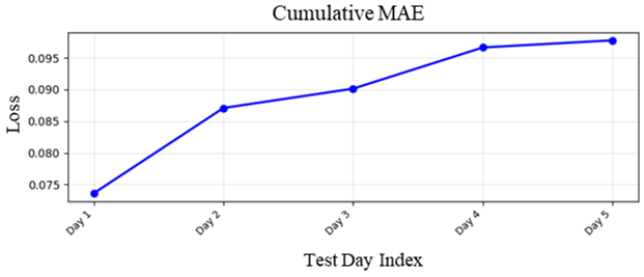


그림 3. 누적손실기반 건전성 테스트 결과

5. 검증 실험

본 장에서는 제안 메트릭의 효용성을 평가하기 위한 검증 실험 및 그 결과에 대해 설명한다.

5.1 실험 설정

(1) 데이터셋 구성

본 실험에서는 산업용 기기의 상태 모니터링을 위해 배포된 오픈 데이터셋 Condition monitoring of hydraulic systems[23]을 활용한다. 해당 데이터셋은 그림 4와 같은 구조를 갖는 유압 설비에서 4가지 구성 요소(Cooler, Pump, Accumulator, Valve)의 상태를 정량적으로 변화시키며 수집된 압력, 유량 등의 센서 데이터로 구성된다. 각 요소별 건전성 테스트를 위해 평가 대상 요소가 점진적인 상태 저하를 보이도록 데이터셋을 재구축하였다. 또한, 다양한 요소로 구성된 산업용 기기에 대한 적용 가능성을 입증하기 위해 분석 대상(Target)이 아닌 요소들을 다양한 상태로 설정하여 실제 운용 환경을 고려한 데이터셋을 구성하였으며, 이는 그림 5와 같다.

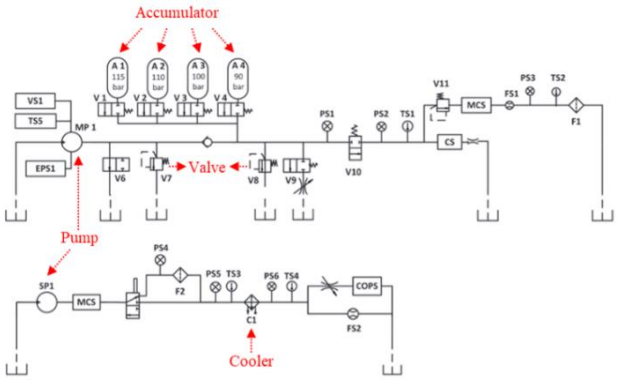


그림 4. 유압 설비 테스트 베드 구조

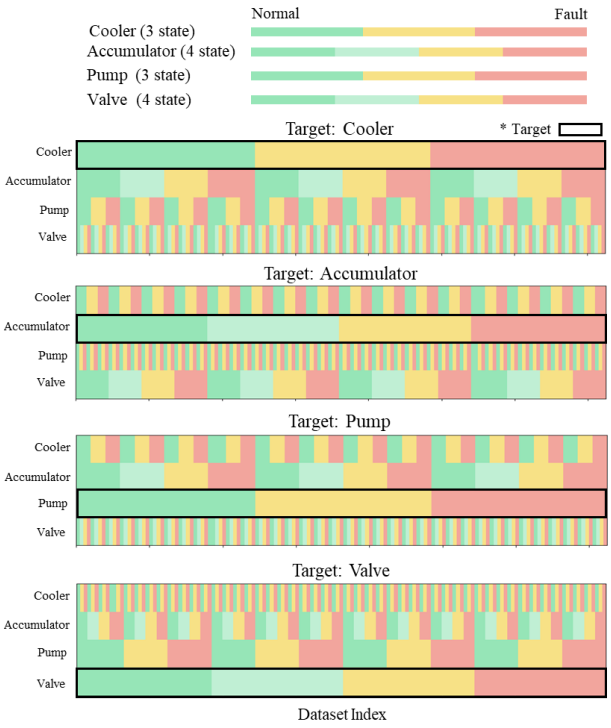


그림 5. 실험 데이터셋 구성

표 4. 건전성 수준별 테스트 정확도 trans.: state transition

Model	Component	MSE			Ours			Euclidean			Ours		
		trans.1	trans.2	trans.3	trans.1	trans.2	trans.3	trans.1	trans.2	trans.3	trans.1	trans.2	trans.3
MLP AE	Cooler	100	100	-	100	100	-	100	100	-	100	100	-
	Pump	24.17	12.08	-	**93.33**	**99.17**	-	34.17	12.71	-	**98.75**	**100**	-
	Accumulator	55.28	21.67	0.27	**80.28**	**95.28**	**98.37**	33.33	21.11	0.81	**82.78**	**90.56**	**92.41**
	Valve	8.33	10.00	21.67	**79.72**	**100**	**100**	5.28	9.17	21.94	**62.50**	**100**	**100**
Conv AE	Cooler	100	100	-	100	100	-	100	100	-	100	100	-
	Pump	39.58	30.42	-	**83.12**	**100**	-	26.25	26.04	-	**98.75**	**100**	-
	Accumulator	52.78	20.83	0.54	**78.33**	**94.17**	**91.33**	33.89	20.83	0.54	**81.67**	**89.72**	**87.26**
	Valve	5.83	23.61	38.06	**60.56**	**100**	**100**	11.11	24.44	33.89	**66.11**	**99.44**	**100**

(2) 모델 구성

제안 기법의 효과를 검증하기 위해 2가지 학습 모델(MLP-AutoEncoder, Conv-AutoEncoder)을 활용하여 비지도 학습 기반 건전성 테스트 실험을 진행하였으며, 각 모델의 구조는 표 2 및 3과 같다. 모델 학습은 평가 대상 요소(Target)의 정상 데이터셋 중 70%를 무작위로 선정하고, 가우시안 노이즈를 추가하여 실데이터셋과 동일한 규모의 증강 데이터를 생성하여 학습에 활용하였다. 이후, 학습에 사용되지 않은 나머지 실데이터셋은 건전성 테스트를 위한 테스트 용도로 활용하였다.

표 2. MLP-AutoEncoder 모델 구조

MLP-AutoEncoder	
Layer sizes	**Activation**
(Encoder) [seq_len, 32, 16, 8]	ReLU (hidden)
(Decoder) [8, 16, 32, seq_len]	ReLU (hidden)

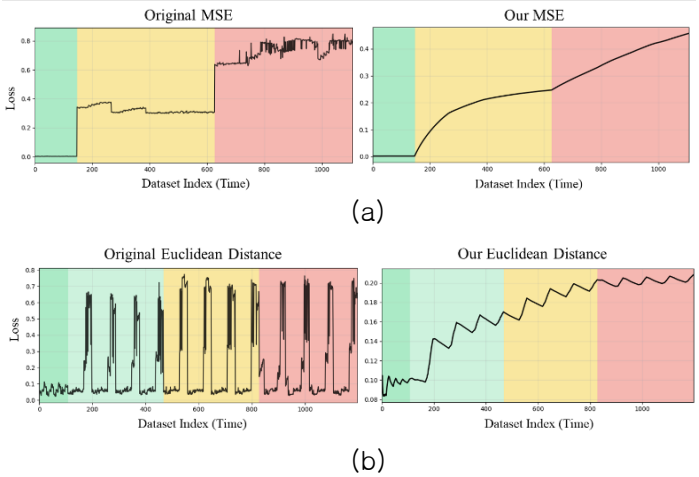
표 3. Conv-AutoEncoder 모델 구조

Conv-AutoEncoder			
Channels	**Kernel sizes**	**Stride**	**Activation**
(Encoder) [1, 8, 16, 32]	[7, 5, 3]	2	ReLU
(Decoder) [32, 16, 8, 1]	[3, 5, 7]	2	ReLU

5.2. 실험 결과

해당 실험은 제안 기법의 건전성 수준별 평가 및 추적 가능 여부를 정성적/정량적으로 검증하기 위한 실험으로, MSE와 Euclidean Distance를 손실 함수로 사용하였다. 정성적 평가는 건전성 저하에 따른 테스트 결과 그래프의 변화 경향성을 기반으로 수행하였으며, 정량적 평가는 이전 상태에서 산출된 손실 값 중 상위 5%를 임계값으로 설정하여 분류 정확도를 산출하였다. 이에 대한 실험 결과는 표 4 및 5와 같다. 임계값은 건전성 테스트의 강도 설정 지표로 엄격한 평가가 요구되는 경우 작은 값으로 설정할 수 있으며, 관대한 평가가 요구되는 경우에는 큰 값으로 설정할 수 있다.

정성적 평가 결과, 그림 6과 같이 기존 메트릭 대비 제안하는 누적손실을 활용할 경우 건전성 저하에 따른 손실 메트릭의 변화 경향성을 보다 명확하게 식별할 수 있음을 확인하였다. 특히, 기존의 방식은 단일 평가 주기만을 활용하기 때문에 저하에 따른 데이터 변동(예: 분산 증가)의 영향을 매우 크게 받아 그림 6-(b)의 좌측 그래프와 같이 저하가 심화될수록 손실 그래프를 활용한 건전성 테스트 및 추적이 매우 어렵다. 그러나, 제안하는 메트릭은 과거의 가동 데이터 패턴을 함께 반영함으로써 그림 6의 우측 그래프와 같이 점진적인 저하에 따라 발생하는 데이터 패턴의 변화 경향성 식별이 용이하여 그래프를 활용한 높은 정확도의 건전성 테스트 및 추적이 가능하다.



(a)

(b)

그림 6. 건전성 저하 테스트 결과 그래프
(a) Cooler의 Conv-AE, (b) Accumulator의 MLP-AE

또한, 정량적 평가 결과에서도 표 4와 같이 유압 시스템 내에서의 물리적 영향력이 매우 커 건전성 변화가 타 요소에 큰 영향을 미치는 Cooler에 대한 정확도가 모두 100%인 경우를 제외하고, 나머지 세

요소의 건전성 상태 전이에 대하여 기존의 손실 메트릭 대비 높은 평가 정확도를 보임을 확인하였다. 특히, 표 5의 요소별 평균 정확도로 분석하였을 때, 최소 56.56%에서 최대 79.91% 매우 큰 성능 향상을 보임으로써 제안하는 누적손실기반의 평가 메트릭이 그림 6과 같이 단발적인 이상치의 영향을 완화함에 따라 건전성 테스트에 있어 매우 높은 효용성을 지님을 검증하였다.

표 5. 건전성 수준별 평가 종합 정확도

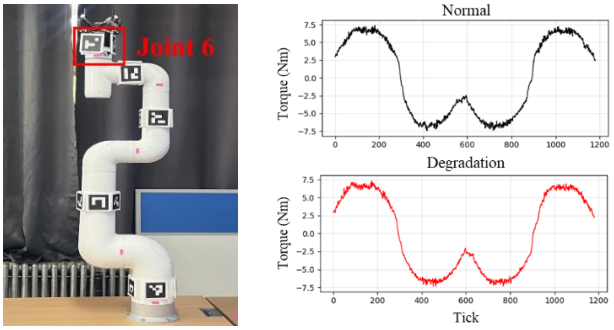
Model	Component	MSE	Ours	Euclidean	Ours
MLP AE	Cooler	100	100	100	100
	Pump	18.13	**96.25**	23.44	**99.38**
	Accumulator	25.74	**91.31**	18.42	**88.58**
	Valve	13.33	**93.24**	12.13	**87.50**
Conv AE	Cooler	100	100	100	100
	Pump	35.00	**91.56**	26.15	**99.38**
	Accumulator	24.72	**87.94**	18.42	**86.22**
	Valve	22.50	**86.85**	23.15	**88.52**

6. 사례 연구

5장의 검증 실험에서는 산업용 기기의 상태 모니터링을 목적으로 구축된 오픈 데이터셋을 활용하였다. 해당 데이터셋은 제안하는 건전성 테스트 메트릭의 정량적 평가와 효용성을 분석하는 데에는 적절하나 사전정의된 건전성 조건에 의해 각 상태가 명확하게 구분된 특성을 갖는다. 그러나, 실제 산업 환경에서의 건전성 저하는 장기간 운용에 의해 점진적으로 발생함에 따라 그 상태 구분이 명확하지 않은 경우가 많다. 이에, 본 장에서는 제안하는 건전성 테스트 메트릭의 실효성 평가를 위해 인위적인 상태 변화 없이 장기간 가동을 통해 점진적인 저하 및 고장이 발생한 산업용 기기의 데이터셋을 활용하여 검증을 수행하고 그 결과를 설명한다.

6.1. 데이터셋

사례 연구에서는 그림 7-(a)와 같이 설계된 Neuromeka 사 Indy7 협동 로봇의 6번 관절 토크 데이터를 활용한다. 해당 데이터셋은 약 7개월간의 실제 운용을 통해 수집되었으며, 로봇의 건전성 테스트를 위해 매 운용일마다 전원 인가 직후 각 관절의 전체 가동 범위 수행 가능 여부를 확인하는 동작 중 수집된 데이터로 구성된다. 그림 7-(b)는 정상 데이터셋과 저하 상태 데이터셋의 예시로, 두 데이터 간의 패턴 차이가 매우 미미하여 단순한 통계적 비교만으로는 건전성 저하를 명확히 평가하기 어려운 양상을 보인다. 이는 효과적인 건전성 테스트 메트릭의 필요성을 보여준다.

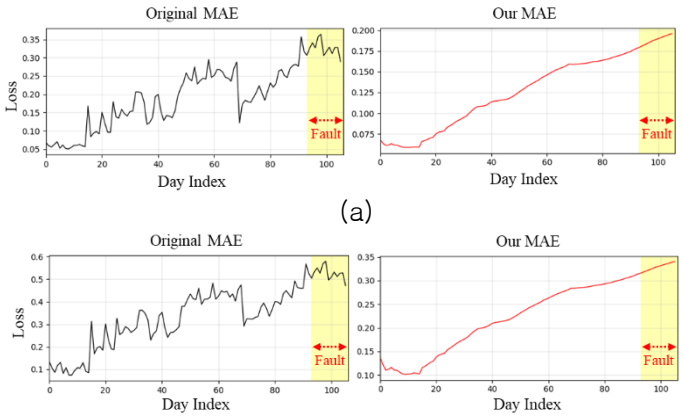


(a) (b)

그림 7. Indy7 협동 로봇 및 정상/저하 데이터셋 예시

6.2. 검증 결과

검증 결과, 두 가지 학습 모델 모두에서 그림 8과 같이 기존의 손실 활용 방식 대비 제안하는 누적손실 기반 평가 메트릭이 건전성 저하 양상을 명확하고 직관적으로 식별하는 데에 효과적임을 확인하였다. 기존 방식의 경우 건전성 저하에 따라 손실 값의 패턴이 전반적으로 증가하는 경향을 보여 전체 구간에 대한 사후 분석을 수행할 경우에는 건전성 테스트가 가능하다. 그러나, 단기적으로 평가하였을 때 손실의 등락 폭이 매우 커 부정확한 건전성 테스트 및 추적이 수행될 위험이 있다. 반면, 제안하는 메트릭은 과거의 가동 패턴을 반영함에 따라 평가 주기의 단발적 이상치에 대한 영향을 매우 적게 받는다. 이에 따라 고장(Fault) 지점까지 손실의 급격한 등락 없이 꾸준한 증가 경향성을 보임으로써 높은 정확도로 건전성 테스트 및 추적이 가능함을 확인하였으며 그 실효성을 검증하였다.



(a)

(b)

그림 8. 협동 로봇 관절의 건전성 테스트 결과
(a) MLP-AE, (b) Conv-AE

7. 결론

본 논문은 산업용 기기의 건전성 테스트를 위한 누적손실기반의 새로운 건전성 테스트 메트릭을 제안한다. 제안 메트릭의 구축 방법은 총 3단계로, 평가 주기의 손실을 산출하는 손실 계산 단계, 모든 평가

주기의 손실을 누적 합산하는 손실 누적 단계 그리고 누적된 손실을 평균화하는 누적손실 평균화 단계로 구성된다. 제안 메트릭의 효용성 검증을 위한 건전성 수준별 평가 실험 결과, 최소 86.22%에서 최대 100%의 높은 평가 정확도를 달성함을 확인하였다. 특히, 기존의 손실 활용 방법 대비 최대 79.91%의 향상된 평가 성능을 보임으로써 건전성 테스트 메트릭으로서 충분한 효용성을 가짐을 입증하였다. 또한, 실적용 가능성 여부를 평가하기 위한 사례 연구 결과, 제안 메트릭이 기존 메트릭에 비해 단발적 이상치의 영향을 효과적으로 완화하여 손실 값의 급격한 변동을 줄이고, 건전성 저하의 진행 양상을 명확하고 직관적으로 나타낼 수 있음을 확인하였다. 이를 통해 제안 메트릭의 실제 산업 현장 적용 가능성과 실효성을 검증하였다. 향후 연구로는 장기적 부하 누적에 의한 건전성 저하와 급격한 고장 징후를 동시에 평가하고, 상태 변화 경향에 기반하여 손실을 초기화할 수 있는 건전성 테스트 기법으로 확장할 예정이다.

References

[1] Jinyang Jiao, Ming Zhao, Jing Lin, Kaixuan Liang, “comprehensive review on convolutional neural network in machine fault diagnosis,” Neurocomputing, vol. 417, pp. 36-63, 2020.

[2] Mohsen Soori, Behrooz Arezoo, Roza Dastres, “Internet of things for smart factories in industry 4.0, a review,” Internet of Things and Cyber-Physical Systems, vol. 3, pp. 192-204, 2023.

[3] T. Roosefert Mohan, J. Preetha Roselyn, R. Annie Uthra, D. Devaraj, K. Umachandran, “Intelligent machine learning based total productive maintenance approach for achieving zero downtime in industrial machinery,” Computers & Industrial Engineering, vol. 157, no. 107267, pp. 1-22, 2021.

[4] R. Langone, C. Alzate, B. De Ketelaere and J. A. K. Suykens, “Kernel spectral clustering for predicting maintenance of industrial machines, 2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), Singapore, pp. 39-45, 2013.

[5] Zhe Li, Yi Wang, Ke-Sheng Wang, “Intelligent predictive maintenance for fault diagnosis and prognosis in machine centers: Industry 4.0 scenario,” Advances in Manufacturing, vol. 5, pp. 377-387, 2017.

[6] Serkan Ayvaz, Koray Alpay, “Predictive maintenance system for production lines in manufacturing: A machine learning approach using IoT data in real-time,” Expert Systems with Applications, vol. 173, pp. 1-10, 2021.

[7] R. Zhao, D. Wang, R. Yan, K. Mao, F. Shen and J. Wang, “Machine Health Monitoring Using Local Feature-Based Gated Recurrent Unit Networks,” IEEE Transactions on Industrial Electronics, vol. 65, no. 2, pp. 1539-1548, 2018.

[8] Stefanos Kontos, Alexandros Bousdekis, Katerina Lepenioti, Gregoris Mentzas, “Degradation Modelling and Prognostics of Rotating Equipment with Automated Machine Learning,” Procedia Computer Science, vol. 253, pp. 1640-1648, 2025.

[9] Maria Grazia De Giorgi, Stefano Campilongo, Antonio Ficarella, “A diagnostics tool for aero-engines health monitoring using machine learning technique,”, Energy Procedia, vol. 148, pp. 860-867, 2018

[10] Huang L, Pan X, Liu Y, Gong L, “Unsupervised Machine Learning Approach for Monitoring Data Fusion and Health Indicator Construction,” Sensor, vol. 23, no. 16, pp. 1-19, 2023.

[11] Zhuang Ye, Jianbo Yu, “Health condition monitoring of machines based on long short-term memory convolutional autoencoder,” Applied Soft Computing, vol. 107, pp. 1-14, 2021.

[12] A. Klausen 1 H.V. Khang 1 K.G. Robbersmyr, “RMS Based Health Indicators for Remaining Useful Lifetime Estimation of Bearings.” Modeling, Identification and Control, vol. 43, no. 1, pp. 21-38, 2022.

[13] T. Yan, D. Wang, J. -Z. Kong, T. Xia, Z. Peng, L. Xi, “Definition of Signal-to-Noise Ratio of Health Indicators and Its Analytic Optimization for Machine Performance Degradation Assessment,” IEEE Transactions on Instrumentation and Measurement, vol. 70, pp. 1-16, 2021.

[14] Haoran Yan, Yi Qin, Sheng Xiang, Yi Wang, Haizhou Chen, “Long-term gear life prediction based on ordered neurons LSTM neural networks,” Measurement, vol. 165, pp. 1-11, 2020.

[15] Guangyao Zhang, Yi Wang, Xiaomeng Li, Yi Qin, Baoping Tang, “Health indicator based on signal probability distribution measures for machinery condition monitoring,” Mechanical Systems and Signal Processing, vol. 198, pp. 1-21, 2023.

[16] A. Rai and J. -M. Kim, “A Novel Health Indicator Based on Information Theory Features for Assessing Rotating Machinery Performance Degradation,” IEEE Transactions on Instrumentation and Measurement, vol. 69, no. 9, pp. 6982-6994, 2020.

[17] Meng Rao, Xingkai Yang, Yuejian Chen, Mingjian Zuo, Zhenfei Bu, Yaqiang Jin, Fulei Chu, “A new health indicator for rotating machinery condition monitoring under variable operation conditions through regression among vibration features,” Mechanical Systems and Signal Processing, vol. 241, pp. 1-20, 2025.

[18] Shun Wang, Yolanda Vidal, Francesc Pozo, “An unsupervised approach to early fault detection and performance degradation assessment in bearings,” Advanced Engineering Informatics, vol. 68, no. Part A, pp. 1-22, 2025.

[19] Xinsheng Lou, Kenneth A Loparo, “Bearing fault diagnosis based on wavelet transform and fuzzy inference,” Mechanical Systems and Signal Processing, vol. 18, no. 5, pp. 1077-1095, 2004.

[20] Qiang Miao, Xin Zhang, Zhiwen Liu, Heng Zhang, “Condition multi-classification and evaluation of system degradation process using an improved support vector machine,” Microelectronics Reliability, vol. 75, pp. 223-232, 2017.

[21] Ying Zhang, Hongfu Zuo, Fang Bai, “Classification of fault location and performance degradation of a roller bearing,” Measurement, vol. 46, no. 3, pp. 1178-1189, 2013.

[22] Yujing Wang, Shouqiang Kang, Yicheng Jiang, Guangxue Yang, Lixin Song, V.I. Mikulovich, “Classification of fault location and the degree of performance degradation of a rolling bearing based on an improved hyper-sphere-structured multi-class support vector machine,” Mechanical Systems and Signal Processing, vol. 29, pp. 404-414, 2012.

[23] Nikolai Helwig, Eliseo Pignanelli, Andreas Schütze, “Condition Monitoring of a Complex Hydraulic System Using Multivariate Statistics,” 2015 IEEE International Instrumentation and Measurement Technology Conference (I2MTC), Italy, pp. 210-215, 2015.

생성형 AI 문서 검토에서 출처 라벨이 사용자 판단과 오류 탐지 수행에 미치는 영향

민소원
한국과학기술원 문화기술대학원
ericasowon@kaist.ac.kr

The Effect of Source Labels on User Judgment and Error Detection Performance in Reviewing Generative AI Documents

Sowon Min
Graduate School of Culture Technology, Korea Advanced Institute of Science and Technology

요 약

본 연구는 생성형 AI가 생성한 문서를 대상으로 한 오류 탐지 과제에서, 문서의 출처 라벨(AI 생성 vs. 인간 작성)이 사용자 판단 인식과 실제 수행에 미치는 영향을 분석하였다. 조건 간 실험을 통해 오류 탐지 수행, 판단 확신, 인지된 신뢰를 측정한 결과, 출처 라벨에 따른 수행 수준의 유의미한 차이는 관찰되지 않았다. 그러나 자유서술 응답에서는 AI 조건에서만 '의심'과 '불신'을 명시적으로 표현하는 언어적 반응들이 나타났다. 이는 출처 라벨이 판단 태도에는 영향을 미치지만, 실제 검증 수행으로 자동 전이되지는 않음을 시사한다. 본 연구는 생성형 AI 사용자 판단에서 인식과 수행 간의 분리 가능성을 제시하며, 단순한 출처 공시를 넘어 검증 행동을 유도하는 공정 중심 설계의 필요성을 논의한다.

Abstract

As generative AI becomes widely used, users increasingly act as evaluators responsible for detecting errors in AI-generated text. This study examines how source labels (AI vs. human) influence users' error detection performance, judgment confidence, and perceived trust when reviewing a document containing errors. A between-subjects experiment was conducted with 25 participants, who reviewed identical content with different source labels. The results showed no significant differences between conditions in error detection performance, judgment confidence, or perceived trust. In addition, error detection performance was not significantly correlated with judgment confidence or perceived trust. These findings suggest that source-based judgments may not readily translate into effective verification behavior, highlighting a potential disconnect between subjective judgment and actual performance in generative AI use.

1. 서 론

생성형 인공지능(Generative AI)의 급격한 확산은 소프트웨어 개발 및 문서화 공정에서 사용자의 역할을 근본적으로 변화시키고 있다. 오늘날 사용자는 AI가 생성한 텍스트를 단순히 소비하는 수동적 존재에서 벗어나, 그 결과물의 정확성과 타당성을 최종적으로 판단하고 결함을 식별해야 하는 적극적인 검토 주체이자 품질 보증(QA)의 핵심 동력으로 기능한다[1][2]. 이러한 변화는 생성형 AI가 요구사항 명세서 초안 작성, 서비스 설명서 생성, 정책 가이드라인 수립 등 높은 정밀도를 요구하는 비정형 텍스트 산출 과정에 깊이 통합됨에 따라, 소프트웨어 서비스 설계 전반의 의사결정 공정에 핵심적인 요소로 자리 잡았기 때문이다 [3][4].

그러나 소프트웨어 V&V(Verification & Validation) 관점에서 볼 때, 생성형 AI의 출력물은 인간 검토자가 품질 판단 역할을 수행하기에 반드시 최적화된 형태로 제공되지 않는다는 구조적 한계를 지닌다. 생성형 AI가 산출하는 문장은 언어적으로 매우 유창하고 구조적으로 완결된 형태를 띠는 경우가 많아 사용자가 이를 신속하게 이해하도록 돕지만, 동시에 내용의 타당성에 대한 별도의 비판적 검증 없이 수용하게 만드는 '인지적 안주'의 위험을 내포한다 [5]. 이러한 표면적 그럴듯함은 사실과 어긋난 내용을 은폐할 수 있으며, 특히 생성형 언어모델의 고질적인 문제인

'환각(Hallucination)' 현상은 소프트웨어 품질의 신뢰성을 저해하는 치명적인 요인으로 작용한다 [6][7]. 따라서 생성형 AI 기반 문서 활용 환경에서 사용자가 출력의 신뢰성과 정확성을 비판적으로 평가하고, 검증과 오류 탐지(error detection)를 수행하는 역할이 필수적으로 요구된다 [8][9]. 소프트웨어 검증의 전통적 관점에서, 검증자는 검토 대상에 결함이 존재할 수 있음을 전제하고 이를 적극적으로 탐지하는 역할을 수행한다. 즉, 검증 활동은 출처와 무관하게 비판적 태도를 요구하며, 검증 대상이 사람에 의해 작성되었는지, 혹은 자동화된 도구에 의해 생성되었는지는 원칙적으로 검증 전략을 달리하는 기준이 되어서는 안 된다. 그러나 생성형 AI 환경에서는 이러한 규범적 전제가 실제 인간의 판단 과정에서 항상 동일하게 작동하는지에 대한 의문이 제기된다.

한편, 인간-AI 상호작용 연구에서는 이러한 판단 상황을 이해하기 위해 오랫동안 자동화 편향(automation bias)과 AI에 대한 신뢰(trust) 개념을 중심으로 논의해 왔다[10][11][12]. 자동화 시스템의 제안은 인간 수행을 향상시키기도 하지만, 과도한 신뢰로 인해 오류를 간과하게 만들 수 있음이 보고되었다 [13][14]. 그러나 최근 생성형 AI 환경에서는 기존의 설명만으로는 포착하기 어려운 변화가 나타나고 있다. 생성형 AI의 오류 가능성에 대한 사회적 인식이 확산되면서, 사용자들은 AI 출력에 대해 이전보다 더 의심적이고 비판적인 태도를 보이는 경향을 드러내고 있다[15]. 실제로 동일한 정보라 하더라도 “AI가 생성했다”는 출처 라벨은 정보의 인지된 정확도와 신뢰도를 유의미하게 낮추는 것으로 보고되었으며, 이러한 효과는 정보의 실제 진위 여부와 무관하게 나타난다[16][17]. 이는 자동화 편향 논의에서 전제해 온 “AI에 대한 과도한 신뢰”와 대비되는 양상으로, 생성형 AI 맥락에서 신뢰의 결여 또는 알고리즘 회피(algorithm aversion)가 함께 고려되어야 함을 시사한다 [18]. 문제는 이러한 인식 변화가 실제 판단 수행으로 어떻게 연결되는지가 여전히 명확하지 않다는 점이다. 사용자가 AI를 덜 신뢰한다고 보고하는 것과 실제로 더 정확하게 오류를 탐지하는 것은 동일한 과정이 아닐 수 있으며, 출처 라벨로 인해 형성된 경계심이 검증 전략의 변화나 오류 탐지 수행 향상으로 이어지는지는 경험적으로 충분히 검증되지 않았다. 또한, 기존 연구들은 주로 신뢰·정확도 평가·수용 의도 등 인식 수준의 지표를 중심으로 출처 라벨의 효과를 분석해 왔고, 인식 변화가 오류 탐지와 같은 수행 수준의 행동으로 전환되는지에 대해서는 제한적으로 다루어져 왔다.

이에 본 연구는 생성형 AI가 생성한 문서를 대상으로 한 오류 탐지 과제에서, 문서의 출처 라벨(AI 생성 vs. 인간 작성)이 (1) 오류 탐지 수행, (2) 판단 확신 및 인지된 신뢰, 그리고 (3) 판단 인식과 수행 간의 관계에 어떠한 영향을 미치는지를 수행 수준에서 분석하고자 한다. 본 연구는 출처 라벨의 효과를 입증하거나 부정하는 데 목적을 두기보다, 출처 기반 판단 인식이 검증 수행으로 자동 전이되지 않을 수 있는 조건을 실험적으로 규명함으로써, 생성형 AI 환경에서의 인간 중심 검증 공정 설계에 대한 시사점을 제공하고자 한다. 본 연구는 다음의 연구 질문을 제시한다.

RQ1. 출처 라벨은 실제 오류 탐지율에 유의미한 차이를 유발하는가?

RQ2. 문서 출처 라벨은 주관적 판단 확신과 신뢰도에 어떠한 영향을 미치는가?

RQ3. 출처로 인한 인식의 변화는 실제 수행 데이터와 어떤 통계적 관계를 갖는가?

이를 위해 본 연구는 (1) 동일한 내용과 오류를 포함한 문서를 AI 생성 조건과 인간 작성 조건으로 제시하고, (2) 참가자들의 오류 탐지 수행을 측정하며, (3) 과제 이후 판단 확신과 인지된 신뢰를 수집한다. 이후 (4) 출처 라벨에 따른 수행 및 인식 차이를 비교하고, (5) 인식 지표와 수행 간의 관계를 탐색적으로 분석함으로써, 생성형 AI 기반 문서 검증 공정에서 인식-행동 연결 구조를 규명한다.

2. 선행 연구

2.1 생성형 AI 출력물의 품질 특성과 환각 현상

생성형 AI가 산출하는 텍스트의 한계는 소프트웨어 공학에서 정의하는 품질 속성 중 신뢰성(reliability) 관점에서 설명될 수 있다. 신뢰성은 시스템이 주어진 조건에서 기대된 기능을 정확하고 일관되게 수행하는 정도를 의미하지만, 대규모 언어모델(LLM)은 본질적으로 확률적 추론에 기반한 비결정론적 시스템으로서 이러한 요구를 안정적으로 충족시키지 못한다. 특히 생성형 AI는 문법적으로 유창하고 맥락적으로 자연스러운 텍스트를 생성하는 데에는 탁월한 성능을 보이는 반면, 생성된 내용의 사실적 정합성이나 논리적 정확성을 보장하지 못하는 환각 문제를 구조적으로 내포하고 있음이 반복적으로 지적되어 왔다 [6][7]. Bender는 이러한 언어모델의 구조적 한계를 “실제 의미를 이해하지 못한 채 통계적 확률에 기반하여 다음 토큰을 예측하는 확률적 앵무새(Stochastic Parrots)”로 개념화하며, 언어적 유창함이 곧 의미적 타당성이나 진실성을 보장하지 않는다는 점을 비판적으로 논의하였다 [6]. 환각 오류는 명시적인 오류 신호나 경고 없이 자연스러운 문장 형태로 제시되기 때문에, 전통적인 소프트웨어 오류와 달리 검토자가 오류의 존재 자체를 인식하기 어렵다는 특징을 가진다. 이러한 특성은 생성형 AI

출력물이 소프트웨어 문서화, 요구사항 명세서, 정책 초안과 같이 높은 정밀도와 검증 가능성을 요구하는 공정에 사용될 경우, 잠재적으로 심각한 품질 리스크로 작용할 수 있다.

더 나아가, 생성형 AI 출력물의 높은 언어적 유창성과 구조적 완결성은 인간 검토자의 인지 과정에도 중요한 영향을 미친다. 기존 연구들은 언어적으로 그럴듯한 텍스트가 사용자로 하여금 내용의 타당성을 직관적으로 신뢰하게 만들며, 추가적인 검증이나 반증 탐색을 수행하려는 동기를 약화시킬 수 있음을 보고하였다 [1][5]. 이러한 현상은 생성형 AI 출력이 실제로 더 정확해서가 아니라, 표면적으로 신뢰 가능해 보이는 형태를 띠기 때문에 발생한다는 점에서 중요하다 [5][8]. 이러한 표면적 신뢰성은 검토자의 비판적 사고를 억제하고 인지적 안주(cognitive complacency)를 유도함으로써, 결과적으로 오류 탐지를 위해 요구되는 인지적 비용을 증가시키는 기제로 작용할 가능성이 있다. 즉, 생성형 AI 출력물은 생산성 측면에서는 검토 대상 문서의 초안을 빠르게 제공함으로써 효율을 향상시킬 수 있으나, 동시에 오류를 식별하고 검증하는 공정의 난이도를 구조적으로 높일 수 있다. 요컨대 생성형 AI 출력물의 문제는 오류의 존재 여부 자체가 아닌, 오류가 인간에게 어떻게 인식되고 검토되는가에 있다. 환각 현상과 표면적 완결성은 결합되어, 검토자가 오류를 탐지하기 어려운 조건을 형성하며, 이는 이후 자동화 편향, 신뢰 교정 실패, 그리고 검증 수행 저하와 같은 인간 요인 문제로 연결될 수 있다. 이러한 맥락에서 생성형 AI 출력의 품질 특성은 기술적 성능 평가를 넘어, 인간 중심의 검증 공정 관점에서 재검토될 필요가 있다.

2.2 자동화 편향과 알고리즘 회피: 검증 수행 실패의 공통 메커니즘

자동화된 시스템의 출력을 검토하는 인간의 판단 과정은 인간-컴퓨터 상호작용(HCI) 및 소프트웨어 공학 연구에서 오랫동안 자동화 편향 개념을 중심으로 논의되어 왔다. 자동화 편향은 인간이 자동화 시스템의 제안이나 판단을 과도하게 신뢰하여, 해당 출력의 정확성을 충분히 검증하지 않은 채 수용함으로써 오류를 간과하는, 과도한 신뢰(over-trust) 현상을 의미한다 [10][13]. Skitka는 자동화된 의사결정 지원 환경에서 사용자가 자신의 정확한 판단보다 시스템의 잘못된 제안을 우선시하거나, 명백한 시스템 오류를 탐지하지 못하는 경향을 실험적으로 입증하였다 [13]. 이후 다수의 연구들은 자동화 시스템이 전반적인 수행 정확도를 향상시킬 수 있음에도 불구하고, 오류 발생 시 인간 검토자의 비판적 개입이 감소함으로써 새로운 유형의 인적 오류(human error)를 유발할 수 있음을 반복적으로 보고하였다. 즉, 시스템이 신뢰할 만하다는 인식이 형성될수록 사용자는 검증 행동을 줄이고, 결과적으로 오류 탐지 수행이 저하될 수 있다는 설명이다. 이 관점에서 자동화 편향은 소프트웨어 품질 검토 공정에서 검증 단계가 형식화되거나 생략되는 위험을 설명하는 중요한 이론적 틀을 제공해 왔다.

그러나 최근 생성형 AI 환경에서는 생성형 AI의 오류 가능성, 특히 환각에 대한 사회적 인식이 확산되면서, 사용자가 시스템의 출력을 충분히 검토하기 이전에 출처에 근거한 선제적 불신을 보이는 경향이 보고되고 있으며, 이는 알고리즘 회피로 개념화된다 [18]. 알고리즘 회피는 자동화 시스템에 대한 과도한 신뢰가 아니라, 오히려 시스템의 오류 가능성을 과대평가함으로써 그 산출물 전반을 부정적으로 평가하거나 회피하는 현상을 의미한다. 이러한 경향은 출처 라벨링 연구에서도 확인된다. Altay 등은 동일한 내용과 품질을 가진 정보라 하더라도 “AI가 생성했다”는 출처 라벨이 부착될 경우, 사용자가 인지하는 정확도와 신뢰도가 유의미하게 하락함을 보고하였다 [16]. 이는 사용자가 정보의 객관적 품질보다는 “AI 생성 여부”라는 외적 단서에 의해 판단을 프레이밍할 수 있음을 시사한다. 다시 말해, 생성형 AI 맥락에서는 시스템의 실제 성능과 무관하게, 출처 정보 자체가 판단의 출발점을 규정하는 요인으로 작동할 수 있다.

자동화 편향과 알고리즘 회피는 표면적으로는 상반된 현상처럼 보인다. 전자는 시스템에 대한 과도한 신뢰에서, 후자는 시스템에 대한 과도한 불신에서 출발하기 때문이다. 그러나 소프트웨어 검증 공정의 관점에서 보면, 두 현상은 공통적으로 검증 수행의 질을 보장하지 못한다는 점에서 중요한 문제를 공유한다. 전자는 과도한 신뢰로 인해 검증이 생략되는 위험을, 후자는 출처 기반 직관적 판단으로 인해 체계적인 오류 탐지가 이루어지지 않는 위험을 내포한다. 즉, 신뢰가 과도하든 부족하든, 그 자체만으로는 효과적인 검증 수행을 보장하지 않는다는 점에서 두 현상은 동일한 검증 실패 메커니즘으로 이해될 수 있다. 나아가 이는 생성형 AI 기반 문서 검토에서 인간 요인을 이해하기 위해, 단순한 신뢰 수준의 증감이 아니라 실제 검토 수행(performance level behavior)을 분석할 필요성을 제기한다. 특히 출처 정보가 제공된 상황에서 사용자가 자동화 편향과 알고리즘 회피 중 어느 방향으로 반응하든, 그 결과가 실제 오류 탐지 수행에 어떠한 영향을 미치는지는 경험적으로 검증될 필요가 있다. 본 연구는 이러한 문제의식에 기반하여, 출처 라벨이 사용자 판단 인식과 더불어 검증 수행에 어떠한 방식으로 작용하는지를 수행 수준에서 분석하고자 한다.

2.3 신뢰 교정 및 인식-수행 간의 불일치

출처 라벨링(source labeling)은 사용자가 시스템의 산출물을 해석하고 검토하는 과정에서 제공되는 대표적인 인지적 보조 도구 중 하나로 논의되어 왔다. 인간-AI 상호작용 연구에서는 사용자가 시스템의 실제 성능과 자신의 신뢰 수준을 정합적으로 조정하는 과정을 신뢰 교정(trust calibration)으로 정의하며, 출처 정보는 이러한 교정을 유도할 수 있는 잠재적 단서로 간주된다 [11][12]. Lee와 See에 따르면, 신뢰가 시스템의 실제 역량보다 높을 경우 과잉 의존(over-reliance)이 발생하고, 반대로 신뢰가 지나치게 낮을 경우 시스템의 유용한 기능이 충분히 활용되지 않는 오용(disuse)이 초래될 수 있다 [11]. 이 관점에서 신뢰 교정은 인간 AI 협업의 효율성과 품질을 유지하기 위한 핵심 조건으로 이해되어 왔다. 최근 생성형 AI 맥락에서 출처 라벨링은 이러한 신뢰 교정을 유도하는 구체적인 설계 요소로 주목받고 있다. 다수의 실증 연구들은 동일한 내용의 정보라 하더라도 “AI가 생성했다”는 라벨이 부착될 경우, 사용자가 인지하는 신뢰도와 정확도 평가가 유의미하게 낮아진다는 결과를 보고하였다 [16][17]. 이러한 발견은 출처 라벨이 사용자의 판단 인식에 분명한 영향을 미친다는 점을 보여주며, 출처 정보가 시스템 산출물의 해석 과정에서 중요한 인지적 단서로 기능할 수 있음을 시사한다.

그러나 이러한 인식 변화가 실제로 검증 수행의 향상으로 이어지는지는 명확히 규명되지 않았다. 일부 연구들은 출처 라벨링이 사용자의 회의론을 증폭시켜 주관적 신뢰를 낮추는 데에는 효과적이지만, 이것이 반드시 더 정밀한 오류 탐지나 체계적인 검증 행동의 강화로 전이되지는 않는다고 지적한다 [2][17]. 즉, “AI를 신뢰하지 않는다”는 인지적 태도의 형성과 “오류를 찾기 위해 추가적인 인지적 노력을 투입한다”는 행동적 실천 사이에는 괴리가 존재할 수 있다.

이러한 현상은 사회과학 및 HCI 연구에서 논의되어 온 의도-행동 간격(intention-behavior gap) 개념과도 연결된다 [2]. 사용자가 낮은 신뢰를 보고하거나 비판적 태도를 표명하더라도, 실제 판단 과정에서 오류를 탐지하기 위한 전략을 적극적으로 적용하지 않거나 검토 깊이를 증가시키지 않을 가능성이 있다는 것이다. 이는 신뢰와 확신과 같은 인식 지표가 검증 수행을 직접적으로 대변하지 못할 수 있음을 의미하며, 주관적 판단 척도만으로 검토 공정의 품질을 평가하는 데 한계가 있음을 시사한다.

소프트웨어 검토 및 품질 보증(QA) 공정의 관점에서 볼 때, 이러한 인식-수행 간의 불일치는 중요한 시사점을 가진다. 출처 라벨링과 같은 인터페이스 수준의 정보 제공은 사용자의 인식과 태도를 조정하는 데에는 기여할 수 있으나, 그것만으로는 인간 검토자가 오류 탐지에 충분한 인지적 자원을 투입하도록 강제하거나 보장하는 충분조건이 아닐 수 있다 [1]. 다시 말해, 출처 라벨은 신뢰 교정을 위한 필요조건일 수는 있으나, 검증 수행의 실질적 향상을 담보하기 위해서는 보다 적극적인 검증 지원 메커니즘이나 공정 차원의 설계 개입이 요구될 수 있다.

이러한 선행연구들은 생성형 AI 기반 문서 검토에서 핵심적인 문제를 “신뢰가 어떻게 변화하는가”가 아니라, “그 변화가 실제 수행으로 어떻게 연결되는가”로 재정의할 필요성을 제기한다. 본 연구는 이러한 문제의식에 기반하여, 출처 라벨이 사용자 판단 인식과 오류 탐지 수행에 각각 어떠한 영향을 미치는지를 수행 수준에서 분석한다.

3. 연구방법

3.1 실험 참가자

총 25명의 성인이 본 실험에 참여하였다. 참가자들은 전공과 무관하게 일반적인 문서 읽기 및 검토 경험을 보유한 성인으로 구성되었다. 모든 참가자는 연구 목적과 절차에 대한 설명을 제공받았으며, 자발적 동의 하에 실험에 참여하였다. 모든 참가자는 2개의 문서 내용을 읽고 오류를 식별하는 과제를 수행할 수 있는 기본적인 읽기 및 이해 능력을 갖추고 있었으며, 모든 사전에 생성형 AI 사용 경험이 있는 것으로 확인되었다. 실험 시작 전 모든 참가자는 연구 목적과 절차에 대한 설명을 제공받았으며, 자발적 동의 하에 참여하였다. 참가자는 무작위로 두 조건 중 하나에 배정되었으며, AI 출처 조건에는 12명, 사람 출처 조건에는 13명이 포함되었다.

3.2 실험 설계

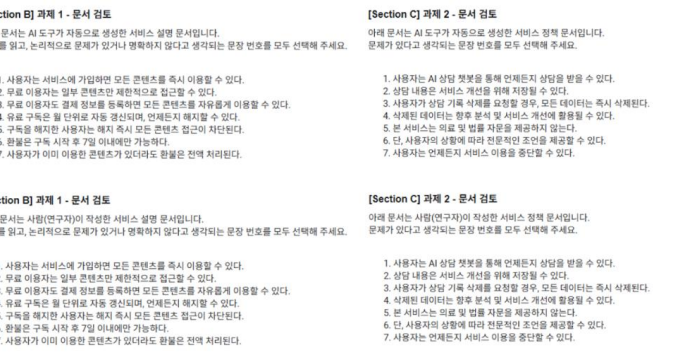


그림 1.

본 연구는 문서 출처 라벨(source label)을 독립변수로 하는 조건 간(between-subjects) 실험 설계를 채택하였다. 출처 라벨은 두 수준으로

구성되었으며, 문서가 AI에 의해 생성되었다고 명시된 조건과 사람이 작성한 문서라고 명시된 조건으로 구분되었으며, 모든 참가자는 내용, 길이, 오류의 수와 유형이 동일한 두 개의 문서를 제공받았다. 즉, 조건 간 차이는 문서 상단에 제시된 출처 라벨 정보에 한정되었다. 이를 통해 출처 정보가 사용자 판단에 미치는 영향을 통제된 환경에서 검증하고자 하였다. 그림 1은 참가자에게 실제로 제시된 오류 탐지 과제 문서를 보여준다.

3.3 실험 자극 및 과업

오류ID	문장	오류 유형	설명
E1	2-3	불일치	접근 권한 모순
E2	3	모호성	구분 불명확
E3	5-7	불일치	즉시 차단 vs 이용 후 환불
E4	6	누락	환불 처리 시점 미명시

표 1.

오류ID	문장	오류 유형	설명
E1	3-4	불일치	삭제된 데이터 재활용
E2	2	모호성	저장 범위 등 불명확
E3	5-6	불일치	원칙과 조항 충돌
E4	7	누락	데이터 처리 방식 미명시

표 2.

실험에 사용된 문서는 소프트웨어 개발 및 서비스 운영 과정에서 생성형 AI가 초안 작성에 활용될 수 있는 정책·설명·요구사항 성격의 문서를 추상화하여 설계되었다. 각 문서는 정보성 텍스트 형태를 띠고 있으며, 논리적 불일치, 의미적 모호성, 정보 누락 등 실제 문서 검토 과정에서 발생할 수 있는 오류 유형을 포함하고 있다. 각 문서에 포함된 오류 문장 및 오류 유형은 표 1과 표 2에 정리되어 있다. 참가자의 과제는 문서를 읽으면서 오류라고 판단되는 모든 문장을 식별하는 것이었으며, 수행 시간에는 제한을 두지 않았다.

3.4 측정지표

3.4.1 오류 탐지 수행

오류 탐지 수행은 문서에 포함된 총 11개의 오류 문장 중 참가자가 정확히 식별한 오류 문장의 비율로 산출하였다. 각 참가자의 오류 탐지율은 정탐한 오류 문장 수를 전체 오류 문장 수로 나누어 계산하였다.

3.4.2 판단 확신

판단 확신은 참가자가 자신의 오류 탐지 판단에 대해 느낀 확신 수준을 측정하기 위해 사용되었다. 총 3개 문항으로 구성되었으며, 각 문항은 7점 리커트 척도로 응답되었다. 분석에는 3개 문항의 평균값을 사용하였다.

3.4.3 인지된 신뢰

인지된 신뢰는 참가자가 제공된 문서의 전반적인 신뢰성을 어떻게 평가하는지를 측정하기 위해 사용되었다. 총 2개 문항으로 구성되었으며, 각 문항은 7점 리커트 척도로 응답되었다. 분석에는 2개 문항의 평균값을 사용하였다.

3.5 탐색적 자유서술 응답 수집

본 연구에서는 출처 라벨이 문서 검토 과정에 미치는 주관적 영향을 탐색적으로 파악하기 위해, 모든 참가자에게 과제 종료 후 자유서술 문항을 추가로 제시하였다. 해당 문항은 각 조건에 맞게 출처 주체를 달리하여 제시되었으며(AI 조건: “AI 도구에 의해 생성되었다는 정보”, 사람 조건: “사람이 작성했다는 정보”), 출처 정보가 문서를 검토하거나 판단하는 과정에 영향을 주었는지와 그 이유를 자유롭게 서술하도록 요청하였다.

이 자유서술 응답은 조건별로 분리하여 (i) 응답 수, (ii) 특정 표현(예: “의심”, “환각”, “상관없다”)의 등장 빈도(포함 여부 기준), (iii) 대표 인용문을 중심으로 기술통계적으로 보고하였다. 해당 분석은 수행 지표를 대체하기 위한 것이 아니라, 정량 결과 해석을 위한 맥락 정보를 제공하기 위한 목적의 기술적(descriptive) 분석이다.

3.6 데이터 분석

조건 간 차이를 검증하기 위해 Welch의 t-검정을 사용하였다. 이는 조건별 표본 수와 분산이 동일하지 않을 가능성을 고려한 것이다. 효과크기는 Cohen’s d로 보고하였다.

오류 탐지율과 판단 확신 및 인지된 신뢰 간의 관계를 분석하기 위해 Pearson 상관분석을 수행하였다. 모든 통계 분석은 유의수준 α = .05에서 수행되었다.

4. 연구 결과

그림 2.

변수	AI(n=12), M(SD)	사람(n=13), M(SD)
오류 탐지율	0.318 (0.999)	0.343 (0.054)
판단 확신	5.278 (1.469)	5.077 (0.992)
인지된 신뢰	4.083 (1.564)	4.000 (1.803)

표 3.

변수	Welch t(df)	p	Cohen's d
오류 탐지율	-0.76(16.82)	0.459	-0.31
판단 확신	0.40(19.11)	0.696	0.16
인지된 신뢰	0.12(22.92)	0.903	0.05

표 4.

4.1 오류 탐지율

오류 탐지율은 두 과제에 포함된 총 11개 오류 문장 중 참가자가 정확히 식별한 오류 문장의 비율로 정의하였다.

AI 조건(n = 12)의 평균 오류 탐지율은 M = 0.318, SD = 0.099였으며, 사람 조건(n = 13)의 평균 오류 탐지율은 M = 0.343, SD = 0.054였다. 두 조건 모두 오류 탐지율은 중간 수준 범위에 분포하였으며, 천장 효과 또는 바닥 효과는 관찰되지 않았다.

조건 간 오류 탐지율 차이를 Welch의 t-검정으로 분석한 결과, 두 조건 간 차이는 통계적으로 유의하지 않았다, $t(16.82) = -0.76$, $p = .459$. 효과크기(Cohen's d)는 -0.31로 나타났다. 조건별 오류 탐지율의 기술통계는 표에 제시하였다.

4.2 판단 확신과 인지된 신뢰

판단 확신은 3개 문항의 평균값으로 산출하였다. AI 조건의 판단 확신 평균은 M = 5.278, SD = 1.469였으며, 사람 조건의 평균은 M = 5.077, SD = 0.992였다. Welch의 t-검정 결과, 조건 간 판단 확신 차이는 통계적으로 유의하지 않았다, $t(19.11) = 0.40$, $p = .696$.

인지된 신뢰는 2개 문항의 평균값으로 산출하였다. AI 조건의 인지된 신뢰 평균은 M = 4.083, SD = 1.564였으며, 사람 조건의 평균은 M = 4.000, SD = 1.803이었다. Welch의 t-검정 결과, 조건 간 인지된 신뢰 차이는 통계적으로 유의하지 않았다, $t(22.92) = 0.12$, $p = .903$.

오류 탐지율, 판단 확신, 인지된 신뢰의 조건별 기술통계는 표에 요약하였다.

4.3 판단 인식 지표와 오류 탐지 수행 간의 관계

오류 탐지율과 판단 확신 간의 Pearson 상관계수는 $r = 0.02$, $p = .916$으로 나타나 거의 상관이 관찰되지 않았다. 오류 탐지율과 인지된 신뢰 간의 상관계수는 $r = -0.27$, $p = .197$로 나타났으며, 통계적으로 유의하지 않았다.

4.4 출처 라벨이 문서 검토에 미친 주관적 영향

과제 종료 후, 모든 참가자에게 문서의 출처 정보가 문서를 검토하거나 판단하는 과정에 영향을 주었는지에 대해 자유서술로 응답하도록 요청하였다. 해당 문항은 조건에 따라 출처 주체를 달리하여 제시되었으며(AI 조건: "AI 도구에 의해 생성되었다는 정보", 사람 조건: "사람이 작성했다는 정보"), 총 17명의 참가자가 응답을 제출하였다(AI 조건: 10명, 사람 조건: 7명).

AI 조건 응답자 10명 중 8명은 출처 정보가 문서 검토 태도에 영향을 주었다고 응답하였다. 이들 응답에서는 출처 기반의 경계적 반응을 직접적으로 지시하는 표현들이 관찰되었으며, 구체적으로 '의심'(2/10), '환각'(1/10), '숨겨진 의도'(1/10), '편향'(1/10), '틀릴 수 있음'(1/10)과 같은 표현이 포함되었다. 예를 들어, 한 참가자는 "AI의 환각이 미묘한 표현의 불명확함으로 나타날 수 있다고 생각해서, 평소보다 더 세심하게 읽어보게 되었다"고 응답하였고, 또 다른 참가자는 "틀릴 수 있다는 생각에 문장 구조와 논리성을 더 집중해서 봤다"고 기술하였다. 반면, AI 조건 응답자 중 2명은 출처 정보가 검토 과정에 영향을 주지 않았다고 응답하였다.

사람 조건 응답자 7명 중 5명은 출처 정보가 문서 검토에 영향을 주지 않았다고 응답하였으며, 이들은 "출처와 무관하게 문장 자체에 집중했다", "사람이 작성했든 AI가 작성했든 상관없다"와 같은 방식으로 출처 무관성을 직접적으로 진술하였다. 나머지 2명은 사람이 작성한 문서라 하더라도 "실수가 있을 수 있다"는 점을 고려하여 비판적으로 읽었다고 응답하였으나, AI 조건에서 관찰된 것과 같은 '의심'이나 '불신'을 명시적으로 언급하지는 않았다.

이러한 자유서술 응답은 정량적 수행 지표를 대체하기 위한 근거가 아니라, 출처 라벨이 참가자에게 어떻게 언어화되어 인식되었는지를 확인하기 위한 설명적 자료로 수집되었다. 특히 AI 조건 응답에서만 출처 기반 경계 반응을 직접 지시하는 표현들이 관찰되고(6/10), 사람 조건에서는 동일한 표현이 관찰되지 않았다는 점(0/7)은, 출처 라벨이 주관적 서술 수준에서 조건별로 다르게 활성화될 수 있음을 보여주는 보조적 맥락 정보로 제시된다. 이러한 결과는 정량적 수행 지표에서 관찰된 조건 간 차이 부재를 해석하기 위한 맥락적 근거로 활용된다.

5. 논의

본 연구는 생성형 AI 활용 맥락에서 문서의 출처 라벨(AI 생성 vs. 인간 작성)이 사용자의 오류 탐지 수행, 판단 확신, 인지된 신뢰에 어떠한 영향을 미치는지를 수행 수준에서 실증적으로 검증하고자 하였다. 정량적 분석 결과, 출처 라벨에 따라 오류 탐지율에서는 통계적으로 유의미한 차이가 관찰되지 않았으며, 판단 확신과 인지된 신뢰 역시 조건 간 차이를 보이지 않았다. 또한 판단 인식 지표(확신, 신뢰)와 실제 오류 탐지 수행 간의 상관관계는 매우 약하거나 거의 관찰되지 않았다. 이러한 결과는 출처 라벨이 사용자 판단의 인식적 측면에 일정한 영향을 미칠 수 있음에도 불구하고, 그러한 인식 변화가 실제 오류 탐지 수행으로 자동 전이되지 않음을 시사한다.

표면적으로 보았을 때 이러한 결과는 출처 라벨의 효과가 제한적이거나 미미한 것으로 해석될 수 있다. 그러나 본 연구는 이러한 차이의 부재 자체가 생성형 AI 사용자 판단의 중요한 특성을 드러낸다고 해석한다. 특히 오류 탐지율이 중간 수준에 분포하고 천장 효과나 바닥 효과가 관찰되지 않았음에도 불구하고, 출처 라벨과 판단 인식 지표가 수행 수준의 행동 변화로 이어지지 않았다는 점은, 사용자 판단에서 인식과 수행이 필연적으로 연동되지 않으며 서로 독립적으로 작동할 수 있음을 시사한다. 다시 말해, 사용자가 무엇을 의심하고 어떻게 인식하는가와, 실제로 어떤 검증 행동을 수행하는가는 동일한 인지 메커니즘의 결과가 아닐 수 있다.

이러한 해석은 자유서술 응답 분석에서 보다 분명하게 드러난다. AI 출처 조건에 노출된 참가자들은 “의심된다”, “틀릴 수 있다”, “환각”, “숨겨진 의도”와 같은 표현을 반복적으로 사용하며, 출처 정보가 문서 검토 태도에 영향을 주었다고 보고하였다. 반면, 인간 작성 조건에 노출된 참가자들은 출처 정보가 판단 과정에 영향을 주지 않았다고 응답하거나, 출처와 무관하게 문장 자체의 내용과 논리적 일관성에 집중했다고 서술하는 경향을 보였다. 즉, AI 조건에서는 출처 라벨이 판단의 출발점으로 언어화되며 명시적으로 활성화된 반면, 사람 조건에서는 출처 정보가 판단 과정에서 거의 호출되지 않거나 비가시적으로 처리되었다.

그러나 이러한 언어적·태도적 차이는 실제 오류 탐지 수행의 차이로는 이어지지 않았다. AI 출처 조건에서 관찰된 의심은 보다 분석적인 검증 행동이나 체계적인 오류 탐지 전략으로 전환되기보다는, 출처 정보에 반응하여 형성된 상징적 회의(symbolic skepticism) 수준에 머물렀을 가능성이 크다. 참가자들은 “AI는 틀릴 수 있다”는 사회적으로 공유된 인식을 언어적으로 표명했지만, 그러한 인식이 실제로 더 많은 인지적 자원 투입, 반복 검토, 또는 대안적 추론 전략의 적용으로 이어지지는 않았다. 판단 확신 및 인지된 신뢰와 오류 탐지율 간의 상관이 거의 관찰되지 않았다는 점은, 이러한 인식-행동 간 분리를 간접적으로 뒷받침한다.

이러한 결과는 기존 자동화 편향(automation bias) 연구에서 강조되어 온 “과도한 신뢰가 오류를 유발한다”는 설명을 보완할 필요성을 제기한다. 본 연구의 결과에 따르면, 신뢰가 낮아졌다고 해서 자동적으로 더 정교한 판단이나 적극적인 검증이 이루어지는 것은 아니다. 즉, 생성형 AI 맥락에서 핵심적인 문제는 사용자가 AI를 얼마나 신뢰하거나 불신하는가가 아니라, 출처 정보로 인해 형성된 판단 태도가 실제 검증 전략으로 어떻게–혹은 왜–전환되지 않는가에 있다.

이 관점은 소프트웨어 검증 및 품질 보증(V&V) 맥락에서도 중요한 시사점을 제공한다. 원칙적으로 검증자는 문서의 출처와 무관하게 결함을 전제하고 검토해야 하지만, 실제 인간 판단에서는 출처 정보가 메타인지적 태도 수준에서만 작동하고, 구체적인 검증 전략의 변화로 이어지지 않을 수 있다. 본 실험에서 출처 라벨은 명시적으로 제공되었으나, 참가자에게 오류를 어떻게 탐지해야 하는지에 대한 절차적 지침, 단계적 검토 프레임, 또는 구조화된 검증 기준은 제공되지 않았다. 그 결과, 출처 정보는 판단 인식이나 태도 수준에서는 작동했을 가능성이 있지만, 실제 오류 탐지 수행을 변화시키는 절차적 전략으로는 충분히 활성화되지 않았을 가능성이 있다.

결과적으로 본 연구는 생성형 AI 사용자 판단 문제를 단순히 신뢰의 증감 문제로 환원하기보다, 검증 공정의 설계 문제로 재정의할 필요성을 제기한다. 출처 라벨링은 사용자의 판단 출발점을 형성하고 메타인지적 평가를 유도할 수는 있으나, 그 자체로 검증 행동을 강화하는 충분조건은 아니다. 생성형 AI 기반 문서 검토 및 소프트웨어 품질 보증 환경에서는, 단순한 정보 공시나 불신 유도에 머무르기보다, 사용자의 실제 검증 행동을 구조적으로 유도할 수 있는 절차적 지침, 검토 구조, 그리고 인터페이스 수준의 지원이 병행되어야 할 필요가 있다.

6. 결론

본 연구는 생성형 AI가 생성한 문서를 대상으로 한 오류 탐지 과제에서, 문서 출처 라벨(AI 생성 vs. 인간 작성)이 사용자 판단 인식과 실제 수행에 어떠한 방식으로 작용하는지를 수행 수준에서 분석하였다. 실험 결과, 출처 라벨은 참가자의 언어적 반응과 판단 태도에는 차이를 유발했으나, 오류 탐지 수행이나 판단 확신, 인지된 신뢰의 정량적 차이로는 이어지지 않았다. 특히 판단 인식 지표와 수행 지표 간의 관계가 매우 약하게 나타났다는 점은, 생성형 AI 사용자 판단에서

인식과 행동이 구조적으로 분리될 수 있음을 시사한다.

본 연구의 주요 기여는 출처 라벨의 효과를 단순히 입증하거나 부정하는 데 있지 않다. 대신, 출처 기반 판단 인식의 변화가 수행 수준의 검증 행동으로 자동 전이되지 않는 조건을 실험적으로 가시화했다는 점에 있다. 이는 생성형 AI 사용자 판단을 이해함에 있어, 신뢰나 태도 지표만으로 검증 수행을 예측하는 접근의 한계를 드러내며, 검증 공정과 행동 수준의 분석이 필요함을 강조한다. 동시에 본 연구는 몇 가지 한계를 가진다. 표본 크기가 제한적이며, 단일 문서 유형과 오류 탐지 과제를 사용하였다는 점에서 결과의 일반화에는 주의가 필요하다. 또한 본 연구에서는 출처 라벨만을 조작 변수로 사용하였으며, 구체적인 검증 전략이나 절차적 지침을 제공하지 않았다. 이러한 설계는 출처 라벨의 단독 효과를 검증하는 데에는 적합했으나, 검증 수행을 촉진할 수 있는 다른 설계 요소들과의 상호작용을 탐색하지는 못했다. 향후 연구에서는 보다 다양한 과제 유형과 문서 맥락, 그리고 검증 전략을 명시적으로 유도하는 인터페이스 설계를 포함함으로써, 출처 정보가 실제 검증 행동으로 전환되는 조건을 보다 정밀하게 분석할 필요가 있다. 이러한 연구는 생성형 AI 기반 문서 지원 도구 및 소프트웨어 품질 보증 시스템에서, 단순한 출처 공개를 넘어 사용자의 검증 행동을 실질적으로 강화하는 공정 중심 설계로 이어질 수 있을 것이다.

참고문헌

[1] Hao-Ping (Hank) Lee, Advait Sarkar, Lev Tankelevitch, Ian Drosos, Sean Rintel, Richard Banks, Nicholas Wilson, The Impact of Generative AI on Critical Thinking: Self-Reported Reductions in Cognitive Effort and Confidence Effects From a Survey of Knowledge Workers, Proceedings of the CHI Conference on Human Factors in Computing Systems, pp. 1-22, 2025.

[2] Yan, et al., Generative AI literacy: Scale development and its influence on privacy protection behaviors and information verification behaviors, Telecommunications Policy, pp. 1-16, 2025.

[3] Saleema Amershi, et al., Guidelines for Human-AI Interaction, Proceedings of the CHI Conference on Human Factors in Computing Systems, pp. 1-13, 2019.

[4] Ben Shneiderman, Human-Centered Artificial Intelligence: Reliable, Safe & Trustworthy, International Journal of Human-Computer Interaction, vol. 36, no. 6, pp. 495-504, 2020.

[5] Laura Weidinger, et al., Ethical and Social Risks of Harm from Language Models, arXiv:2112.04359, pp. 1-64, 2021.

[6] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, Shmargaret Shmitchell, On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?, Proceedings of the ACM Conference on Fairness, Accountability, and Transparency, pp. 610-623, 2021.

[7] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, Ting Liu, A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions, ACM Transactions on Information Systems, vol. 43, no. 2, Article 42, 2025.

[8] Rafal Kocielnik, Saleema Amershi, Paul N. Bennett, Will You Accept an Imperfect AI? Exploring Designs for Adjusting End-User Expectations of AI Systems, Proceedings of the CHI Conference on Human Factors in Computing Systems, 411, pp. 1-14, 2019.

[9] C. Zhai, S. Wibowo, L. D. Li, The Effects of Over-Reliance on AI Dialogue Systems on Students' Cognitive Abilities: A Systematic Review, Smart Learning Environments, vol. 11, no. 28, pp. 1-38, 2024.

[10] Raja Parasuraman, Victor Riley, Humans and Automation: Use, Misuse, Disuse, Abuse, Human Factors, vol. 39, no. 2, pp. 230-253, 1997.

[11] John D. Lee, Katrina A. See, Trust in Automation: Designing for Appropriate Reliance, Human Factors, vol. 46, no. 1, pp. 50-80, 2004.

[12] Kevin A. Hoff, M. Bashir, Trust in Automation: Integrating Empirical Evidence on Factors That Influence Trust, Human Factors, vol. 57, no. 3, pp. 407-434, 2015.

[13] Linda J. Skitka, Kathleen Mosier, Michael Burdick, Does Automation Bias Decision-Making?, International Journal of Human-Computer Studies, vol. 51, no. 5, pp. 991-1006, 1999.

[14] Mary T. Dzindolet, Scott A. Peterson, Roger A. Pomranky, Larry G. Pierce, Hugh P. Beck, The Role of Trust in Automation Reliance, International Journal of Human-Computer Studies, vol. 58, no. 6, pp. 697-718, 2003.

[15] Chunsik Lee, Generative AI Risks and Resilience: How Users Adapt to Hallucination and Privacy Challenges, Telematics and Informatics Reports, vol. 19, Article 100221, 2025.

[16] Sacha Altay, Fabrizio Gilardi, People Are

Skeptical of Headlines Labeled as AI-Generated, Even If True or Human-Made, Because They Assume Full AI Automation, PNAS Nexus, vol. 3, no. 10, pgae403, 2024.
[17] Chloe Wittenberg, Ziv Epstein, Gabrielle Péloquin-Skulski, Adam J. Berinsky, David G. Rand, Labeling AI-Generated Media Online, PNAS Nexus, vol. 4, no. 6, pgaf170, 2025.
[18] Berkeley J. Dietvorst, Joseph P. Simmons, Cade Massey, Algorithm Aversion, Journal of Experimental Psychology: General, vol. 144, no. 1, pp. 114-126, 2015.

톰슨 샘플링 기반 지향성 협력 퍼저

모현민 [1*], 김윤호 [2]

한양대학교 미래자동차공학과(미래자동차-SW 융합전공)[1], 한양대학교 컴퓨터소프트웨어학과 [2]

hyeonminmo@hanyang.ac.kr, yunhokim@hanyang.ac.kr

Thompson Sampling based Directed Collaborative Fuzzer

Hyeonmin Mo[1*], Yunho Kim[2]

Department of Automotive Engineering (Automotive-Computer Convergence), Hanyang University[1]

Department of Computer Science, Hanyang University[2]

요 약

기존 지향성 퍼징은 특정 목표 지점 도달이나 취약점 재현에는 효과적이지만, 퍼저마다 사용하는 분석 정보와 탐색 전략이 달라 대상 프로그램과 목표 지점에 따라 성능이 달라지며, 단일 퍼저로는 일관된 탐색 성능을 제공하기 어렵다. 또한 기존 협력 퍼징 기법들은 주로 커버리지 확장에 초점을 두어 목표 지점 기반 탐색에서 효과적인 퍼저 선택과 자원 집중에 한계가 있다. 이를 해결하기 위해 본 논문은 목표 지점 기반 탐색에 특화된 톰슨 샘플링 기반 지향성 협력 퍼저 프레임워크 DCFuzz를 제안한다. 준비 단계에서는 퍼저별 목표 지점 중심 성능을 수집하고, 집중 단계에서는 순위 기반 퍼저 업데이트 결과를 활용하여 톰슨 샘플링을 통해 가장 효과적인 퍼저를 동적으로 선택한다.

총 25개의 실제 취약점 벤치마크 실험에서 DCFuzz는 단일 지향성 퍼저 대비 최초 목표 지점 도달 시간(TTR)과 취약점 재현 시간(TTE)에서 전반적으로 우수한 성능을 보였고, 일부 목표 지점에서는 기존 퍼저가 장시간 도달하지 못한 경우에도 성공적으로 탐색하였다.

1. 서 론

소프트웨어 퍼징**[1,2]**은 자동으로 대량의 입력을 생성·실행하여 취약점을 탐지하는 대표적인 동적 분석 기법이지만, 실제 취약점 분석에서는 프로그램 전체를 무작위로 탐색하기보다 특정 취약 코드나 관심 지점과 같은 목표 지점에 도달하거나 해당 지점에서 취약 조건을 만족하는 입력을 생성하는 것이 중요하다. 이러한 요구에 따라 지향성 퍼징**[3,4,5]**은 정적 분석이나 실행 경로 정보를 활용해 목표 지점과의 거리 또는 관련성을 정량화하고 이를 탐색 과정에 반영함으로써, 커버리지 중심 퍼징 대비 목표 지점 도달 및 취약점 재현 효율을 향상시킨다.

그러나 기존 지향성 퍼징 기법들**[3,4,5]**은 서로 다른 분석 정보와 휴리스틱에 기반해 탐색을 수행하므로, 대상 프로그램이나 목표 지점의 특성에 따라 성능 편차가 크다. 단일 지향성 퍼저는 일부 목표 지점에서는 효과적일 수 있으나, 모든 목표 지점에 대해 일관된 성능을 보이기 어렵다. 이를 보완하기 위해 여러 퍼저를 함께 활용하는 협력 퍼징**[6,7]**이 제안되었지만, 기존 연구들은 주로 전체 커버리지나 취약점 수와 같은 비지향적 성능 지표를 기준으로 퍼저 선택과 자원 분배를 수행해 지향성 퍼징 환경에는 한계가 있다. 특히 지향성 퍼저들은 서로 다른 거리 측정 기준과 분석 정보를 사용하기 때문에, 동일한 시드에 대해서도 퍼저별 성능을 공통된 기준으로 평가하기 어렵다.

본 연구는 이러한 문제를 해결하기 위해 톰슨 샘플링 기반 지향성 협력 퍼저 프레임워크인 DCFuzz를 제안한다. DCFuzz는 준비 단계에서는 동일한 조건 하에 여러 지향성 퍼저를 실행하여 공통된 평가 기준을 통해 목표 지점 중심 탐색 성능을 정량화하고 수집한다. 집중 단계에서는 수집된 성능을 순위 기반으로 갱신하고 톰슨 샘플링**[8,9]**에 반영하여 현재 목표 지점에 가장 적합한 퍼저를 확률적으로 선택한다. 25개의 실제 취약점을 포함한 벤치마크 실험 결과, DCFuzz는 AFLGo**[3]**, WindRanger**[4]**, DAFL**[5]** 대비 평균 목표 지점 도달 시간(TTR)과 취약점 재현 시간(TTE)에서 전반적으로 우수한 성능을 보였으며, 지향성 퍼징 환경에서 협력 기반 접근과 동적 퍼저 선택의 효과를 실험적으로 입증하였다.

2. DCFuzz(Directed Collaborative Fuzzer)

그림 1은 서로 다른 탐색 특성을 갖는 지향성 퍼저들을 효과적으로 결합하기 위해, 톰슨 샘플링 기반의 지향성 협력 퍼저 프레임워크인 DCFuzz의 전체 흐름을 보여준다. DCFuzz는 준비 단계와 집중 단계의 두 단계로 구성되며, 이 두 단계는 라운드 단위로 반복 수행된다. 이러한 반복 구조는 지향성 퍼징 과정에서 퍼저의 상대적 성능이 시간에 따라 변화할 수 있다는 점을 고려하여, 고정된 퍼저 선택이 아닌 동적인 퍼저 조합과 자원 재분배를 가능하게 한다.

준비 단계에서는 동일한 시간 예산과 동일한 초기 시드 조건 하에서 여러 지향성 퍼저를 독립적으로 실행하고, 공통된 평가 기준을 통해 목표 지점 중심 탐색 성능을 정량화한다. 이는 각 퍼저가 사용하는 내부 휴리스틱이나 고유 지표의 차이로 인해 직접적인 성능 비교가 어려운 문제를 완화하기 위한 설계로, 퍼저 간 성능을 공정하게 비교할 수 있는 기반 정보를 제공한다. 또한 준비 단계 동안 각 퍼저가 생성한 시드 중 목표 지점과 의미적으로 높은 관련성을 갖는 입력을 제한적으로 공유함으로써, 특정 퍼저가 국소적인 실행 경로나 입력 공간에 과도하게 편향되는 현상을 완화하고, 서로 다른 퍼저의 탐색 결과가

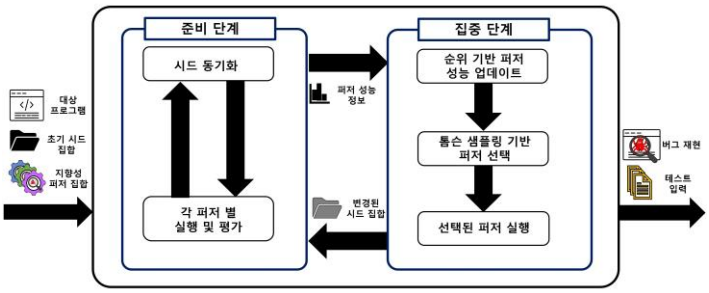


그림 1 톰슨 샘플링 기반 지향성 협력 퍼저 전체 흐름

상호 보완적으로 활용되도록 유도한다. 이러한 과정은 이후 단계에서 효과적인 퍼저 선택과 자원 분배를 수행하기 위한 핵심적인 사전 정보 수집 단계로 기능한다.

집중 단계에서는 준비 단계에서 수집된 성능 정보를 바탕으로 퍼저 간 상대적 성능을 순위 기반으로 갱신하고, 이를 톰슨 샘플링에 반영하여 현재 목표 지점에 가장 적합한 퍼저를 확률적으로 선택한다. 선택된 퍼저는 주어진 시간 동안 퍼징 자원을 집중적으로 할당받아 실행되며, 이 과정에서 생성된 신규 시드와 크래시는 전체 시드 집합과 크래시 집합에 누적 반영된다. 톰슨 샘플링을 활용함으로써, DCFuzz는 관측된 성능이 우수한 퍼저를 우선적으로 활용하는 동시에, 성능 추정의 불확실성을 고려하여 다른 퍼저를 탐색할 기회를 유지한다. 이를 통해 초기 성능이나 일시적인 우수성에 따른 선택 편향을 완화하고, 탐색 진행에 따라 변화하는 목표 지점 도달 특성에 유연하게 대응할 수 있다. 결과적으로 본 프레임워크는 서로 다른 지향성 퍼저들의 상호 보완적 탐색 특성을 지속적으로 결합함으로써, 목표 지점 기반 취약점 탐색의 효율을 점진적으로 향상시키는 적응적 협력 퍼징 메커니즘으로 동작한다.

3. 실험 설정 및 평가

다음의 두 가지 연구 질문을 설정하여 본 연구에서 제시한 DCFuzz를 평가하였다.

- **연구 질문 1. DCFuzz는 단일 지향성 퍼저에 비해 빠르게 목표 지점에 도달하는가?**
- **연구 질문 2. DCFuzz는 단일 지향성 퍼저에 비해 빠르게 목표 지점의 취약점을 재현했는가?**

본 연구에서는 AFLGo[4], WindRanger[5], DAFL[6]의 세 가지 대표적인 지향성 퍼저를 비교 대상으로 하여, 총 25개의 실제 취약점 목표 지점에 대해 DCFuzz의 성능을 평가한다. 각 실험은 목표 지점별로 24시간씩 총 10회 반복 수행되었다. 이를 위해 평균 목표 지점 도달 시간(TTR) 순위와 평균 취약점 재현 시간(TTE) 순위를 지표로 사용하여 협력 지향성 퍼징의 효율성을 분석하였다.

표 1 은 DCFuzz와 단일 지향성 퍼저들 간의 평균 TTR 순위와 평균 TTE 순위 비교한 결과를 보여준다. . DCFuzz는 평균 TTR 순위 1.52를 기록하여, 2위를 차지한 DAFL(2.00)보다 0.48 순위 높은 성능을 보였다. 또한 평균 TTE 순위에서도 1.48을 기록하여, DAFL(2.00) 대비 0.52순위 향상된 결과를 나타냈다. 이는 DCFuzz가 서로 다른 지향성 퍼저들의 탐색 특성을 협력적으로 활용함으로써, 단일 퍼저 대비 목표 지점 도달 및 취약점 재현 과정에서 안정적인 성능 우위를 달성했음을 보여준다.

표 1 DCFuzz와 단일 지향성 퍼저들 간의 평균 TTR 순위와 평균 TTE 순위 비교

	AFLGo	WindRanger	DAFL	DCFuzz
TTR 순위	2.80	2.64	2.00	**1.52**
TTE 순위	2.68	2.56	2.00	**1.48**

4. 결론 및 향후 연구

본 논문은 다양한 지향성 퍼저들의 실시간 성능을 기반으로 목표 지점 기반 퍼징에 적합한 퍼저를 동적으로 선택하는 톰슨 샘플링 기반 지향성 협력 퍼저 프레임워크 DCFuzz를 제안한다. DCFuzz는 동일 조건에서 퍼저별 목표 지점 중심 성능을 수집하는 준비 단계와, 이를 순위 기반으로 정량화하여 톰슨 샘플링을 통해 가장 효과적인 퍼저를 선택·집중 실행하는 단계로 구성된다. 실험 결과, DCFuzz는 단일 지향성 퍼저 대비 목표 지점 도달 시간과 도달 횟수에서 일관되게 우수한 성능을 달성하였다. 본 연구는 향후 목표 지점 도달 이후의 취약점 재현 단계까지 고려한 협력 퍼징 전략으로 확장 가능함을 시사한다.

감사의 글

본 연구는 정부의 재원으로 한국연구재단의 지원을 받아 수행되었으며, 선도연구센터(ERC) 사업의 '소프트웨어재난연구센터' (RS-2021-NR060080)와 우수연구-중견연구 사업의 '프로그램 최적화를 위한 자동 로직 재구성 기술' (RS-2025-24533455)의 지원을 통해 이루어졌다.

참고 문헌

[1] Miller, Barton P.; Fredriksen, Lars; So, Bryan, *An Empirical Study of the Reliability of UNIX Utilities*, Communications of the ACM, Vol. 33, No. 12, pp. 32–44, 1990.

[2] Holler, Christian; Herzig, Kim; Zeller, Andreas, *Fuzzing with Code Fragments*, 21st USENIX Security Symposium (USENIX Security 2012), pp. 445–458, 2012.

[3] Böhme, Marcel; Pham, Van-Thuan; Nguyen, Manh-Dung; Roychoudhury, Abhik, *Directed Greybox Fuzzing*, Proceedings of the ACM Conference on Computer and Communications Security (CCS 2017), pp. 2329–2344, 2017.

[4] Du, Zhengjie; Li, Yuekang; Liu, Yang; Mao, Bing, Du, Zhengjie; Li, Yuekang; Liu, Yang; Mao, Bing, *Windranger: A Directed Greybox Fuzzer Driven by Deviation Basic Blocks*, Proceedings of the International Conference on Software Engineering (ICSE 2022), pp. 2440–2451, 2022.

[5] Kim, Tae Eun; Choi, Jaeseung; Heo, Kihong; Cha, Sang Kil, *DAFL: Directed Grey-box Fuzzing Guided by Data Dependency*, Proceedings of the USENIX Security Symposium, pp. 4931–4948, 2023.

[6] Fu, Yu-Fu; Lee, Jaehyuk; Kim, Taesoo, *autofz: Automated Fuzzer Composition at Runtime*, 32nd USENIX Security Symposium (USENIX Security 2023), pp. 1901–1918, 2023.

[7] Mo, Hyeonmin; Yang, Jongmun; Kim, Yunho, *RCFuzzer: Recommendation-based Collaborative Fuzzer*, Journal of Systems and Software, Vol. 230, Article 112564, 2025.

[8] Thompson, William R., *On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples*, Biometrika, Vol. 25, No. 3/4, pp. 285–294, 1933.

[9] Agrawal, Shipra; Goyal, Navin, *Analysis of Thompson Sampling for the Multi-Armed Bandit Problem*, Proceedings of the Conference on Learning Theory (COLT), JMLR Workshop and Conference Proceedings, Vol. 23 pp. 39–1–39–26, 2012.

컨트롤러 소프트웨어의 속성 검사를 위한 상태 기반 테스트 생성 기법의 평가

러자빈[1], 홍신[2], 최윤자[1]

[1]경북대학교, [2]충북대학교

zavinhle@gmail.com, hongshin@chungbuk.ac.kr, yuchoi76@knu.ac.kr

Stateful System, Stateless Techniques: Initial Assessment on Test Generation Techniques for Property Checking of Controller Software

Za Vinh Le[1], Shin Hong[2], Yunja Choi[1]

[1]Kyungpook National University, [2]Chungbuk National University

Abstract

Property verification is one of the essential activities to ensure the functional correctness of a system that typically requires formal specifications and rigorous verification methods. In search of an alternative technique, this study evaluates state-of-the-art test input generation techniques to assess their ability to detect functional property violations, particularly in controller systems. From a set of benchmark C programs for controller software, we evaluated three representative test generation techniques, dynamic symbolic execution (CREST), greybox fuzzing (AFL, AFL++), and stateful fuzzing (LTL-FUZZER), in terms of their property violation detection ability and detection time. The results show that these test generation techniques are promising for complementing formal methods in property checking and, at the same time, need improvements.

Keywords: property checking, test generation, greybox fuzzing, stateful fuzzing, concolic testing

1. Introduction

A major characteristic of embedded control software, besides hardware dependency, is *statefulness*, meaning that previous interactions or data influence future ones. For example, an automatic transmission controller decides gear-shifting controls depending not only on the current values of throttle position and vehicle speed, but also on the past values. It is known to be common in practice [1] that a functional requirement of a stateful controller program can be defined as an input-output relation condition, namely a functional property, asserted for a specific control state. For example, a functional property for an automatic transmission controller may be specified as “given a throttle position of 50% in the ECO mode, the controller shifts the gears up at moderate RPMs". The practice of testing controller software against functional properties is called Property-based Testing (PBT), and has been actively explored by many researchers [1,2,3,4].

Ensuring that controller software satisfies a functional property is challenging because state variables (e.g., the “mode" in the above example) may influence not only control-flow but also data-flow and output of the controller systems. Rigorous exploration of varying control scenarios with respect to changing internal states requires either formal verification on a formal model [5,6,7], or a testing with varying test sequences carefully designed by domain experts [8]. Either case is costly and limited in scalability.

Recent advances in test generation techniques, such as concolic testing [9] and greybox fuzzing [10], have come to the forefront as new alternatives to automatically generate test inputs. By leveraging both static and dynamic program analysis, they aim to maximize code coverage and failure detection. From the large volume of literature demonstrating their effectiveness and efficiency in uncovering crashes [10,11,12], it is natural to conjecture that these techniques would detect functional property violations with similar efficiency and effectiveness in finding crash bugs. However, this conjecture needs to be formally evaluated as the effectiveness of test generation techniques in detecting functional property violations, especially in stateful controller software, remains largely unexamined.

The functional properties of a controller system often involve complex conditions over multiple variables in different computation cycles, requiring references to the previous state of the system. This distinct challenge of statefulness may limit the fault detection ability of the test generation techniques, which are otherwise highly effective in uncovering crashes. Although several test generation techniques [13,14,15,16,17,18] have been specially designed to uncover crashes and security vulnerabilities in stateful network protocol implementations [19,20,21,13], there exists very limited research with stateful controller software as target programs.

This work presents the first (to our knowledge) empirical evaluation of test generation techniques to find functional property violations in controller software. Specifically, we experimented with four highly popular test generation techniques, a concolic testing tool CREST, coverage-based greybox fuzzers AFL and AFL++ [22], and a stateful fuzzer LTL-Fuzzer [14]. We evaluated their fault detection abilities using mutation testing for seven C programs derived from a popular Simulink Stateflow model benchmark of controller systems, with a total of 331 mutants. As functional properties for the target programs, we derived and used a total of 72 properties from the source code of the controller programs using an active model learning technique (see Section 2.2), which has been proven to be sound through formal completeness checking [23]. For each test generation technique, we measured mutation scores for detecting functional property violations and compared their fault detection performances.

Our evaluation shows that AFL++ performs the best with 99% of detection ratio, but LTL-FUZZER, designed for stateful systems, exhibited more limitations. We discuss our findings with respect to the promises and shortcomings of test generation techniques, and suggest potential improvements.

2. Study Design

2.1. Overview

We designed the empirical studies to answer the following two research questions:

- **RQ1. Effectiveness:** To what extent does a test generation technique generate test inputs that detect property violations?
- **RQ2. Efficiency:** How much time does a test generation technique take to generate test inputs that detect property violations?

We studied the following four test generation techniques covering concolic testing, coverage-based greybox fuzzers and stateful fuzzer:

- **CREST:** a concolic testing technique for C programs. We used an improved version of the original CREST tool [24] such that it supports modeling complex C features such as bitwise operators and floating point arithmetic which are commonly found in embedded controller software, and uses SMT solver Z3 [25].
- **LTL-FUZZER** [14]: a property-directed stateful fuzzer for C programs. Implemented on top of a general-purpose greybox fuzzer AFL v.2.49b, LTL-FUZZER generates inputs by random mutation while guiding the input generation process toward satisfying given temporal properties on state variables. We used LTL-Fuzzer at commit 716ac30, the latest version available at the time of the experiments.
- **AFL++** [22]: the state-of-the-art general-purpose greybox fuzzer for C. AFL++ generates test inputs by random mutation and leads the input generation process toward maximizing code coverage. AFL++ is recognized for its widespread adoption and superior performance in vulnerability detection. We used AFL++ v4.22a, the latest version available at the time of the experiments.

Table 1. Programs used for comparison

Prog.	LoCs	Itr.	Spec	Length (Min/Max/Avg)	Mutants
P1	240	62	8	3 / 8 / 5.50	161 (3152)
P2	137	10	9	4 / 6 / 4.67	31 (333)
P3	177	60	8	3 / 10 / 5.00	112 (1808)
P4	148	25	10	3 / 5 / 3.60	70 (960)
P5	223	30	13	3 / 7 / 3.54	118 (3250)
P6	297	100	8	3 / 8 / 4.63	704 (11240)
P7	473	205	16	5 / 75 / 12.75	2654 (33504)

- **AFL** [26]: a general-purpose greybox fuzzer used as the ground fuzzer for LTL-FUZZER. Specifically, we used AFL v.2.49b embedded in the LTL-FUZZER distribution. We employ this technique for a fair comparison of LTL-FUZZER and its baseline fuzzer that does not use any property and state information.

To experiment with various property-violating programs, we applied property inference and program mutation in sequence, to seven C programs of controller software, and obtained a total of 331 mutants each of which violates a functional property with certain inputs.

2.2. Controller Program Under Tests

Target program selection Table 1 shows the seven target programs, P1 to P7, with their number of lines of code in the second column. These programs were selected from a total of 45 C programs [23] derived from the controller software benchmark [27], through the following steps:

1. randomly choose one program from a set of similar programs, i.e. variations of the same controller system
2. exclude programs that fail to generate functional properties using active learning
3. exclude programs that generate too simple functional properties, e.g., those which have the maximum length of functional properties less than or equal to 3
4. randomly choose a half of the remaining programs at most one program from the same group.

For each target program, we constructed the test harnesses for the four studied techniques in a consistent manner, ensuring that they share the same input format.

Since a controller program is designed to repeat control loops indefinitely, we set the number of loop iterations in a test execution (the third column of Table 1), same as the configurations used in the previous work [23].

Functional property specification We obtained the functional properties of each target controller program using the active model-learning method [23]. This method infers behavior models of a target system from its execution traces by program synthesis [28] and model checking [29]. This approach guarantees to produce a sound model, meaning that the resulting statemachine models admit all observable behaviors of the system. From the inferred statemachine of a target program, we generated a functional property for each state transition condition as an assertion statement. Each assertion statement specifies the expected state transition behavior as a condition on the input, output, and state variables. The fourth and fifth columns of Table 1 show the number of generated assertions and the minimum, maximum, and average number of binary conditions in the assertion statements, respectively.

Figure 1 illustrates an example of a statemachine model inferred from a simple event counting system, with four states and six transitions labeled with specific conditions. For instance, this model asserts that, at State 4 (`IN_Observe`), `if !(1 > rtU.u)` holds, the system transits to State 3 (`IN_Collect_Data`); otherwise, the system remains at State 4. This condition defines a functional property of the system and is specified as an assertion as follows:

```
assert(!(dw.is_c1_model == IN_Observe
         && ! (1 > rtU.u))
      || rtDW.is_c1_model == IN_Collect_Data)
```

Mutant generation To obtain various cases of property violations, we applied mutation operators to the target controller programs and generated mutants, taking the following three steps: (1) a set of distinct mutation operators was applied to each applicable line of a target program except the property-checking assertion statement, generating a comprehensive set of mutants, (2) we

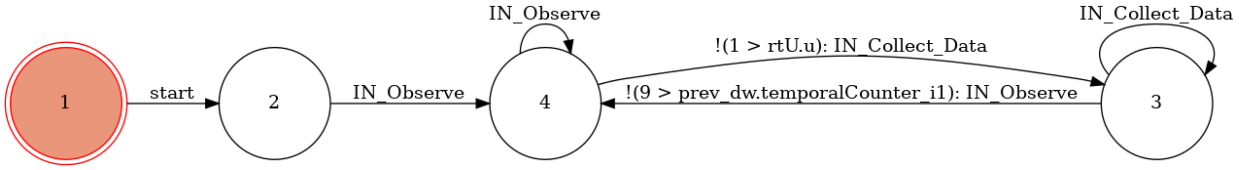


Figure 1. Statemachine generated by active learning

retained only property-violating mutants by checking the property violation using CBMC, and (3) we randomly choose 50 mutants if the number of mutants retained exceeds 50. We chose the following five from the large set of mutation operators [30], considering the earlier works on selecting representative mutation operators [31,32,33]:

- OAAN: Arithmetic operator mutation [31,33]
- OLLN: Logical operator mutation [31,33]
- ORRN: Relational operator mutation [31,33]
- SSDL: Statement deletion [32]
- VSCR: Structural component replacement [33]

We first generated all possible mutants from the target programs using MUSIC [34]. Following mutant generation, we filtered out equivalent mutants that do not violate the given property in any execution, since they are not meaningful for evaluating property violation performance. We used CBMC [29] to identify property-violating mutants by verifying each property-checking mutant, with the same loop bounds configured for the test harnesses (see Section 2.2). In Table 1, the sixth column presents the number of property-violating mutants, alongside the total number of mutants (shown in parentheses). Since too many mutants remain, we randomly sampled them to retain 50 mutants for each target program, except for all 31 mutants of P2. The final dataset consists of a total of 331 mutants spanning all seven programs.

2.3. Test Generation and Measurement

For each technique and mutant studied, we conducted test generation 10 times, each with a time limit of 30 minutes. The result of each test generation run consists of a series of test inputs continuously produced by the technique during the 30 minutes. Given the size of the target programs, this time limit was sufficient to draw reliable conclusions. Since the studied techniques involve randomness, the test generation by each technique with a mutant was repeated 10 times, and the average result was measured.

From the ten test generation results of each technique and mutant, we first measured the ratio of the test generation runs where at least one property-violating test input was generated within the time limit, addressing RQ-1. We counted only the test input that resulted in the property violation while ignoring other crashing inputs. Second, we computed the average time required for the technique to generate the first property-violating test input, addressing RQ 2. We considered that a test generation took 30 minutes if it failed in detecting the property violation within 30 minutes.

We used one zero-filled file as the initial seed corpus for all test generation using the four studied techniques. As the tool-specific configurations, CREST was configured to use DFS (Depth-First Search) as the search strategy. Regarding LTL-FUZZER, we identify target lines by mapping each event in the functional property to the specific program location where the corresponding variable is directly assigned a specific value appearing in the property. All experiments were performed on a 3.3-GHz Intel Xeon Gold 6234 CPU with 200 GB RAM, running Ubuntu 20.04 64-bit version.

3. Results

3.1 RQ1. Effectiveness

Table 2 shows the functional property violation detection ratio of each technique averaged over all experiments of each target program, together with the branch coverage measured for each original program before applying mutation operators. The detection ratio is calculated as the proportion of the total number of functional property violations detected from all mutants, across 10 repetitions. For example of CREST for P1, the detection ratio is 0.98 as

Table 2. Violation detection ratio and branch coverage in parenthesis (%)

Program	CREST	LTL-FUZZER	AFL++	AFL	CBMC
P1	0.98 (100.00)	1.00 (100.00)	1.00 (100.00)	1.00 (100.00)	1.00
P2	1.00 (100.00)	1.00 (100.00)	1.00 (100.00)	1.00 (100.00)	1.00
P3	1.00 (100.00)	1.00 (100.00)	1.00 (100.00)	1.00 (100.00)	1.00
P4	1.00 (100.00)	1.00 (100.00)	1.00 (100.00)	1.00 (100.00)	1.00
P5	1.00 (100.00)	1.00 (96.60)	1.00 (99.15)	1.00 (94.89)	1.00
P6	0.76 (92.50)	0.94 (98.00)	0.94 (99.25)	0.94 (98.25)	1.00
P7	0.30 (50.00)	0.88 (63.44)	1.00 (81.87)	1.00 (77.08)	1.00

CREST detected 49 out of 50 violations across 10 repetitions. As denoted in the parenthesis, CREST achieved 100% branch coverage across 10 repetitions.

In case of programs P2-P5, all four test generation techniques could detect all functional property violations, 50 out of 50 cases in all repetitions. However, all techniques failed to detect violations induced by three mutants in P6, because they are blocked out by segmentation faults before reaching the assert statements. AFL++ and AFL achieve the highest detection rates by detecting all functional property violations in every program except P6. CREST detected fewer violations than others in programs P1 (0.98), P6 (0.76), and its detection ratio dropped significantly in the largest-sized program P7 (0.30). LTL-FUZZER demonstrated comparable performance to AFL++ and AFL, except for P7 where it detected only 44 violations out of 50. It is notable that CREST and LTL-FUZZER show a tendency to decrease the detection rate as their branch coverage decreases, while AFL++ and AFL show the independency between the detection ratio and the branch coverage.

The last column of the table shows that the model checker CBMC detects all property violations without an exception.

3.2 RQ.2 Efficiency

Table 3 shows the average detection time in seconds for each technique for all mutants, together with the number of mutants ranked 1st in detection times using the technique.

AFL++ emerges as the fastest, achieving an average violation detection time of 40.17 seconds. CREST consistently performed well across benchmarks, ranking first 227 times in total, while AFL++, LTL-FUZZER and AFL ranked first 170 times, 134 times and 113 times in total, respectively. CREST performs efficiently for smaller programs, such as achieving the least detection time in P2 (0.09 seconds), but becomes less effective as the program size increases. CREST exhibits the slowest detection time in larger programs, P6 and P7, exceeding 500 seconds on average.

AFL shows better performance than LTL-FUZZER in most cases, except for P6 and P7. This is an unexpected result, as LTL-FUZZER is implemented on top of AFL for a specialized handling of stateful systems, and thus, it is supposed to perform better than AFL. Interestingly, LTL-FUZZER's performance improves with increasing program size. In P6, LTL-FUZZER shows better performance in average detection time and number of mutants ranked first in detection times. In P7, the largest program, although LTL-FUZZER has a slower average detection time than AFL, it ranks first 16 times, while AFL ranks first only once.

The last column of the table shows the detection time by CBMC for comparison. We note that CBMC outperforms the four test generation techniques for programs with higher complexities, e.g., P5, P6 and P7.

4. Discussion

4.1. Stateless techniques vs. stateful techniques

For the first time, this work presents an empirical

Table 3. Average detection time (in second) and the number of the 1st ranks

Program	CREST		LTL-FUZZER		AFL++		AFL		CBMC
	Detection time	# 1st rank	Detection time	# 1st rank	Detection time	# 1st rank	Detection time	# 1st rank	Detection time
P1	37.24	46	5.07	24	0.30	26	0.56	26	0.64
P2	0.09	31	1.00	0	0.28	0	0.59	0	0.18
P3	0.01	50	0.06	44	0.03	44	0.04	44	0.73
P4	87.44	34	20.43	24	0.39	40	1.28	25	0.29
P5	1.42	49	21.82	3	2.54	4	6.76	3	0.43
P6	537.51	17	187.85	23	187.19	23	268.02	14	10.77
P7	1429.16	0	395.68	16	90.43	33	251.59	1	14.85
Average	**298.98**	**-**	**90.27**	**-**	**40.17**	**-**	**75.55**	**-**	**3.98**

comparison of three representative test generation techniques, concolic testing, general purpose fuzzing, and stateful fuzzing, on detecting property violations in controller software, to evaluate their potential as complements of model checking.

Overall, the three stateful and stateless fuzzers effectively detect property violations except for cases blocked by segmentation faults that occurred during the search, whereas CREST shows limitations with larger programs. Interestingly, the stateless fuzzer AFL++ performed the best, demonstrating its effectiveness and efficiency in exploring stateful behaviors.

LTL-FUZZER does not perform as well as expected even though it is specialized for stateful systems. In the nine mutants that LTL-FUZZER failed to detect functional property violations, we found that the mutations changed the operations on the state variables dependent on the assertions for the functional property checking. In these cases, LTL-FUZZER fails in the instrumentation phase or crashes at the runtime phase (i.e., segfault). Even when it operates successfully, LTL-FUZZER often shows an unexpectedly slower detection time when mutations involve state variables (see P7 of LTL-FUZZER compared to that of AFL in Table 3).

4.2 Test generation vs. Model checking

Our initial experiments confirm that model checking is highly effective and efficient as long as the programs under checking are manageable by model checkers. It also shows high potential for state-of-the-art test generation techniques in property-checking stateful systems when the complexity of the program under checking is beyond the capability of model checking.

Although with a great increase in detection time, AFL++ still achieves at least 94% violation detection. However, given that the sizes of the benchmark programs are not large, this result underscores the need for comprehensive empirical evaluations of test generation techniques using large-scale controller programs.

4.3 Limitations and future direction

The scope of our study is limited, highlighting the need for more extensive empirical evaluations: (1) our study focused solely on C programs ranging from 137 to 473 LoCs, (2) the scope of test generation techniques was restricted to the three test generation techniques, and (3) experiments were performed using only syntactic mutations.

Therefore, it is too early to claim that the stateless fuzzer AFL++ is good enough to check property violations in general. However, we believe that our findings on the limitations of current state-of-the-art stateful fuzzers provide valuable insights into future directions. Strictly speaking, those fuzzers known as stateful are not dealing with actual system states, but only assume specific variables as representing states, which could cause various unexpected negative side effects.

We plan to generalize our findings by extending our research to evaluate a wider variety of test generation techniques (e.g., [13,35,36,37]) on larger-scale programs, and investigate improvements on the stateful fuzzing, e.g.,

methods to identify actual states in the stateful system, and to trace states and their dependent variables for providing more effective and efficient search strategy.

5. Conclusion

We empirically evaluate three state-of-the-art test input generation techniques to assess their ability to detect functional property violations, particularly in controller systems through mutation testing. From a set of benchmark C programs for controller software, we generated total 3850 mutants, and then we measured the property violation detection ability and detection time of three representative test generation techniques, dynamic symbolic execution CREST, greybox fuzzers AFL and AFL++, and stateful fuzzer LTL-FUZZER. The results show that these test generation techniques are promising for complementing formal methods in property checking. At the same time, the results demonstrate their limitations, which imply the need for further improvement.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT)(RS-2021-NR060080).

References

[1] H. Goldstein, J. W. Cutler, D. Dickstein, B. C. Pierce, and A. Head, “Property-based testing in practice,” in IEEE/ACM 46th International Conference on Software Engineering, 2024.

[2] C. Hu, W. Dong, Y. Yang, H. Shi, and G. Zhou, “Runtime verification on hierarchical properties of ROS-based robot swarms,” IEEE Transactions on Reliability, vol. 69, no. 2, pp. 674–689, 2019.

[3] M. Park, H. Jang, T. Byun, and Y. Choi, “Property-based testing for LG home appliances using accelerated software-in-the-loop simulation,” in IEEE/ACM 42nd International Conference on Software Engineering (Industry Track), 2020.

[4] E. Bartocci, L. Mariani, D. Nicković, and D. Yadav, “Property-based mutation testing,” in IEEE International Conference on Software Testing, Verification and Validation, 2023, pp. 222–233.

[5] C. Bernardeschi, A. Domenici, and P. Masci, “A PVS-Simulink integrated environment for model-based analysis of cyber-physical systems,” IEEE Transactions on Software Engineering, vol. 44, no. 6, pp. 512–533, 2018.

[6] S. A. Chowdhury, S. Mohian, S. Mehra, S. Gawsane, T. T. Johnson, and C. Csallner, “Automatically finding bugs in a commercial cyber-physical system development tool chain with SLForge,” in Proceedings of the 40th International Conference on Software Engineering (ICSE ’18). ACM, 2018, pp. 981–992.

[7] Y. Jiang, H. Liu, H. Song, H. Kong, R. Wang, Y. Guan, and L. Sha, “Safety-assured model-driven design of the multifunction vehicle bus controller,” IEEE Transactions on Intelligent Transportation Systems, vol. 19, no. 10, pp. 3320–3333, 2018.

[8] I. Drave, S. Hillemacher, T. Greifenberg, S. Kriebel, E. Kusmenko, M. Markthaler, P. Orth, K. S. Salman, J. Richenhagen, B. Rumpe, C. Schulze, M. von Wenckstern, and A. Wortmann, “SmArDT modeling for automotive software testing,” Software: Practice and Experience, vol. 49, no. 2, pp. 301–328, 2019.

[9] K. Sen, “Concolic testing,” in Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE ’07). ACM, 2007, pp. 571–572.

[10] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,” IEEE Transactions on Software Engineering, 2019

[11] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” Communications of the ACM, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[12] Y. Kim, D. Lee, J. Baek, and M. Kim, “Concolic testing for high test coverage and reduced human effort in automotive industry,” in IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019

[13] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, “Stateful greybox fuzzing,” in 31st USENIX Security Symposium (USENIX Security ’22), 2022, pp. 3255–3272.

[14] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, “Linear-time temporal logic guided greybox fuzzing,” in Proceedings of the 44th International Conference on Software Engineering (ICSE ’22). ACM, 2022, pp. 1343–135

[15] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, “Pulsar: Stateful black-box fuzzing of proprietary network protocols,” 2015, pp. 330–347.

[16] C. McMahon Stone, S. L. Thomas, M. Vanhoef, J. Henderson, N. Bailluet, and T. Chothia, “The closer you look, the more you learn: A grey-box approach to protocol state machine learning,” in ACM CCS ’22, 2022, pp. 2265–2278.

[17] R. Natella, “StateAFL: Greybox fuzzing for stateful network servers,” Empirical Software Engineering, vol. 27, no. 7, p. 191, 2022.

[18] C. Daniele, S. B. Andarzian, and E. Poll, “Fuzzers for stateful systems: Survey and research directions,” ACM Computing Surveys, vol. 56, no. 9, pp. 1–23, 2024.

[19] K. Sen, D. Marinov, and G. Agha, “CUTE: A concolic unit

testing engine for C," in ESEC/FSE, 2005, pp. 263–272.

[20] Y. Park, S. Hong, M. Kim, D. Lee, and J. Cho, "Systematic testing of reactive software with non-deterministic events: A case study on LG electric oven," in IEEE/ACM International Conference on Software Engineering, vol. 2, 2015, pp. 29–38.

[21] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," IEEE Transactions on Software Engineering, vol. 47, no. 9, pp. 1980–1997, 2021.

[22] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in 14th USENIX Workshop on Offensive Technologies (WOOT '20), 2020.

[23] N. Yogananda Jeppu, T. Melham, and D. Kroening, "Enhancing active model learning with equivalence checking using simulation relations," Formal Methods in System Design, vol. 61, no. 2–3, pp. 164–197, 2023.

[24] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 443–446.

[25] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in TACAS. Springer, 2008, pp. 337–340.

[26] M. Zalewski, "American fuzzy lop – whitepaper," 2016.

[27] MathWorks, "Stateflow Examples," 2021.

[28] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in TACAS, 2022, pp. 415–442.

[29] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in TACAS, 2004, pp. 168–176.

[30] H. Agrawal, R. DeMillo, B. Hathaway, W. Hsu, E. Krauser, R. Martin, A. Mathur, and E. Spafford, "Design of mutant operators for the C programming language," 1999.

[31] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," ACM Transactions on Software Engineering and Methodology, vol. 5, no. 2, pp. 99–118, 1996.

[32] L. Deng, J. Offutt, and N. Li, "Empirical evaluation of the statement deletion mutation operator," in IEEE International Conference on Software Testing, Verification and Validation, 2013, pp. 84–93.

[33] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," IEEE Transactions on Software Engineering, vol. 32, no. 9, pp. 733–752, 2006.

[34] D. L. Phan, Y. Kim, and M. Kim, "MUSIC: Mutation analysis tool with high configurability and extensibility," in ICST Workshops, 2018, pp. 40–46.

[35] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNet: A greybox fuzzer for network protocols," in ICST, 2020, pp. 460–465.

[36] P. McMinn, "Search-based software test data generation: A survey," Software Testing, Verification and Reliability, vol. 14, no. 2, pp. 105–156, 2004.

[37] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in OSDI, 2008, pp. 209–22

증거 생성과 구조적 제약을 결합한 자동 프로그램 정정 기법

이현수[O1], 양근석[2†]
[1] 한경국립대학교 컴퓨터응용수학부, [2] 한경국립대학교 컴퓨터응용수학부(컴퓨터시스템연구소)
[1] lhs9275@hknu.ac.kr, [2] gsyang@hknu.ac.kr

An Automated Program Repair Approach Combining Evidence Generation and Structural Constraints

Hyunsoo Lee [O1], Geunseok Yang[2†]
[1] Department of Computer Applied Mathematics, Hankyong National University,
[2] Department of Computer Applied Mathematics(Computer System Institute), Hankyong National University
[1] lhs9275@hknu.ac.kr, [2] gsyang@hknu.ac.kr

요 약

본 연구는 로컬 대규모 언어 모델(LLM) 기반의 파이썬 자동 프로그램 정정(APR)에서 발생하는 과도한 코드 재작성(Over-editing)과 구조적 불일치 문제를 해결하기 위한 프레임워크를 제안한다. 해당 기법은 최종 패치를 즉시 생성하는 대신, 정적 분석 정보와 검색 증거를 조건으로 디퓨전 기반 AST 편집 계획을 우선 도출함으로써 모델의 정정 범위와 방향을 구조적으로 제약한다. BugsInPy 데이터셋의 단일 라인 버그 51개를 대상으로 평가한 결과, StarCoder2-7B 모델에서 Success@10 50.98%의 성능을 기록하였다. 실험 결과, 편집 계획의 도입이 탐색 공간을 축소하여 후보 패치의 안정성을 높이고 전반적인 정정 성능을 유의미하게 개선함을 확인하였다.

1. 서 론

소프트웨어의 복잡성 심화는 APR 기술의 필연적 발전을 이끌었으나, 기존 방식은 탐색 공간의 효율적 제어 및 일반화 성능 확보에 어려움을 겪어왔다. 최근 LLM 기반 APR이 대안으로 부상했음에도 불구하고, 로컬 환경의 모델은 수정 지점과 방법론 사이의 괴리로 인해 코드를 전체적으로 재구성하며 구조적 왜곡을 초래하는 한계가 명확하다. 본 연구에서는 이를 극복하기 위해 디퓨전 방식의 AST 편집 가이드와 검색 기반 증거를 결합하여 정정 과정의 정합성을 보장하는 프레임워크를 제안한다.

2. 제안 기법

제안기법은 정정 프로세스를 위치 식별, 편집 방식 결정, 참조 사례 활용의 세 단계로 체계화한다. 기술적 핵심은 AST(Abstract Syntax Tree) 수준의 편집 계획을 선행 도출함으로써, LLM의 생성 자유도를 제어하고 수정 범위를 국소적으로 제한하는 데 있다.

- 학습 단계: 정상적인 함수에 인위적 결함을 주입하여 clean/buggy 코드 쌍을 구축한다. 이후 두 코드 간의 AST 차분 분석[5]을 통해 도출된 편집 연산 시퀀스를 학습 데이터로 활용하여, 정정 방향을 예측하는 디퓨전 모델[3]을 훈련시킨다.
- 추론 단계: 정적 분석을 통해 코드 내 의심 노드 분포를 산출하고, 이를 조건부 입력으로 하여 디퓨전 모델이 구체적인 편집 계획을 샘플링한다. 동시에 RAG[4] 기법으로 검색된 유사 정정 사례를 증거로 결합하여 프롬프트를 구성하며, 최종적으로 로컬 LLM이 해당 제약 조건 하에서 패치를 생성하도록 유도한다.

```
입력:
 - b: 버그가 포함된 함수 코드
 - D*: 학습된 디퓨전 편집 계획 모델
 - R: 유사 패치 데이터베이스
 - N: 계획 샘플링 횟수 (N=12)
 - P: 생성할 프롬프트 조합 수 (P=3)
 - C: 프롬프트당 패치 후보 생성 수 (C=10)
출력: 패치 집합 P 및 평가 지표
알고리즘:
1: ctx ← 버그 문맥 및 에러 메시지 추출(b)
2: S ← 정적 분석 수행(ctx)                  // AST 기반 의심 노드 분포 산출
3: R_sim ← 하이브리드 검색(R, ctx, top-k=5)  // 유사 패치 증거 회수
4: Plans ← 계획 샘플링(D*, S, N)            // N개의 AST 편집 계획 후보 생성
5:  Prompts ← 프롬프트 구성(ctx, R_sim, Plans, P) // 계획과 증거를 결합한 P개의
프롬프트 생성
6: Candidates ← ∅
7: for each prompt p in Prompts do
8:        Candidates ← Candidates ∪ LLM_Generate(p, C) // 각 프롬프트로부터
C개의 후보 생성
9: end for
10: P ← 테스트 검증 및 랭킹(Candidates)      // Docker 샌드박스 내 테스트 수행
11: return P
```

그림 1. 제안기법의 학습 및 추론 절차 의사 코드

3. 실험 및 결과 요약

3.1 실험 설정

본 연구는 로컬 LLM 환경에서 제안기법의 성능을 검증하기 위해 BugsInPy[1] 데이터셋 내 51개의 단일 라인 결함을 실험 대상으로 선정하였다. 기저 모델로는 Qwen[2] 및 StarCoder2[7]등을 활용하였으며, 모든 실험은 동일한 후보 생성 예산 하에 진행되었다. 디퓨전 계획 모델은 L=16, t=50으로 설정하고, 손실 가중치는 Keep 0.1, 편집 연산 1.0을 적용하여 학습을 수행하였다.

* 이 논문은 한경국립대학교 국립대학육성사업(2025)지원을 받아 작성되었음
† 교신저자(Corresponding Author)

3.2 평가 지표 및 비교 방법

정정 성능은 테스트 케이스를 모두 통과한 Plausible Patch 수와 Success@k를 통해 측정하였다. 또한, 제안된 편집 계획의 정밀도를 평가하기 위해 수정 위치 적중률인 LocHit@k, 연산 일치도를 나타내는 OpMatch@k, 그리고 실제 코드 변경이 계획 범위 내에 국한되는 정도를 측정하는 Alignment@k를 평가지표로 채택하여 over-editing 완화 효과를 분석하였다.

3.4 실험 결과 및 분석(RQ1~RQ3)

RQ1: 제안기법이 Baseline 대비 성능을 얼마나 올리나?

표 1. BugsInPy 벤치마크 성능 비교

Method	Plausible (bugs/51)	Success@1 (%)	Success@5 (%)	Success@10 (%)
Qwen1.5-7B-chat (Baseline)	23	36.67	43.08	45.10
Our Methodology (Qwen1.5-7B-chat)	24	37.45	44.83	47.06
Our Methodology (Deepseek-Coder-6.7B)	24	37.45	44.83	47.06
Our Methodology (StarCoder2-7B)	26	40.88	48.15	50.98

실험 결과 표1을 참고하면, StarCoder2-7B[7] 기반의 제안기법이 Success@10에서 50.98%를 기록하며 베이스라인 대비 5.88%p의 가장 높은 성능 개선을 보였다. 이는 구조적 제약 조건의 결합이 상위 후보군 내에 유효 패치가 포함될 확률을 실질적으로 상승했음을 의미한다.

RQ2:증거가 있을 때 편집계획 추가(diff_on)가 얼마나 기여하나?:

표 2. StarCoder2-7B 기반 구성 요소 제거 결과

Setting	Plausible (bugs/51)	Success@1 (%)	Success@5 (%)	Success@10 (%)
diff_off	23	39.46	45.02	45.1
diff_on	26	40.88	48.15	50.98

표2의 구성 요소 제거 실험에서 정적 분석과 검색 증거만 활용한 diff_off 설정은 성능 향상이 미미하였으나, 편집 계획을 통합한 diff_on 설정에서 유의미한 수치 상승이 관찰되었다. 이는 증거 제공만으로는 후보 생성의 분산을 억제하기 어려우며, AST 수준의 계획이 탐색 공간 축소의 핵심 기제로 작용함을 입증한다.

RQ3:계획이 실제 수정(위치/연산)과 얼마나 맞고, over-editing 완화에 기여하나?

표 3. . StarCoder2-7B 기반 편집 계획 품질 지표

k	LocHit@k	OpMatch@K	Alignment@K
1	60.78% (31/51)	93.55% (29/31)	43.14% (173/401)
3	66.67% (34/51)	94.12% (32/34)	44.64% (179/401)
5	68.63% (35/51)	97.14% (34/35)	45.89% (184/401)
10	68.63% (35/51)	100.00% (35/35)	46.63% (187/401)

품질 지표 분석 결과, LocHit@1은 60.78%를 달성하였으며 위치 적중 시 OpMatch@10은 100%에 도달하여 연산 예측의 높은 안정성을 확인하였다. 특히 Alignment@k 수치가 k값 증가에 따라 동반 상승하는 경향은 편집 계획이 LLM의 수정 범위를 효과적으로 구속하여 over-editing을 완화하고 있음을 뒷받침한다.

4.결론

본 연구에서는 로컬 LLM을 활용한 파이썬 프로그램 정정 과정에서 빈번히 발생하는 과도한 코드 재작성과 출력의 불안정성을 제어하기 위해, 정적 분석 정보와 디퓨전 기반 AST 편집 계획, 그리고 하이브리드 검색 증거를 통합적으로 운용하는 기법을 제안하였다. 정상 함수로부터 추출한 AST 차분 시퀀스를 학습 데이터로 활용하여 정정 모델을 구축하였으며, 추론 시에는 의심 노드 국소화와 디퓨전 샘플링을 연계하여 패치 생성의 범위와 방향성을 엄격히 통제하였다. BugsInPy[1] 데이터셋을 활용한 성능 검증 결과, StarCoder2[7] 모델 기반으로 50.98%의 Success@10을 기록하였으며, 소거 실험을 통해 AST 편집 계획이 탐색 공간의 구조적 축소와 정정 성능 향상에 기여하는 핵심 기제임을 실증하였다. 본 연구의 구현체 및 실험 데이터는 저장소[6]를 통해 접근 가능하다. 향후에는 실행 신호(Execution Signals) 및 전역적 데이터 흐름 분석을 체계적으로 통합하여 정정 성능을 고도화하고, 다양한 프로그래밍 언어로 적용 범위를 확장한 범용 APR 모델을 개발할 계획이다.

참 고 문 헌

[1] R. Witschey, O. R. Zielinski, and T. Zimmermann, “BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies,” in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2020, pp. 1556-1560.

[2] J. Bai et al., “Qwen technical report,” arXiv preprint arXiv:2309.16609, 2023.

[3] J. Ho, A. Jain, and P. Abbeel, “Denoising diffusion probabilistic models,” in Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), 2020, pp. 6840-6851.

[4] P. Lewis et al., “Retrieval-augmented generation for knowledge-intensive nlp tasks,” in Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), 2020, pp. 9459-9474.

[5] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” in Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE), 2014, pp. 313-324.

[6] DiffusionRepair GitHub Repository, "DiffusionRepair: Source Code and Supplementary Material." [Online]. Available: https://github.com/lhs9275/KCSE2026_DiffusionRepair

[7] A. Lozhkov et al., “StarCoder 2 and The Stack v2: The Next Generation,” arXiv preprint arXiv:2402.19173, 2024.

Diff-only 환경에서 단일 에이전트와 다중 에이전트 기반 커밋 메시지 생성의 비교 분석

안도경[O1], 양근석[2†]
[1]한경국립대학교 컴퓨터응용수학부, [2]한경국립대학교 컴퓨터응용수학부(컴퓨터시스템연구소)
[1]2023810034@hknu.ac.kr, [2]gsyang@hknu.ac.kr

A Comparative Analysis of Single-Agent and Multi-Agent Commit Message Generation in Diff-Only Settings

Dokyeong An[O1], Geunseok Yang[2†]
[1]Department of Computer Applied Mathematics, Hankyong National University,
[2]Department of Computer Applied Mathematics (Computer System Institute), Hankyong National University
[1]2023810034@hknu.ac.kr, [2]gsyang@hknu.ac.kr

요 약

커밋 메시지는 코드 변경의 목적과 범위를 간결하게 전달하는 핵심 산출물이지만, 실제 개발 환경에서는 시간 제약과 작성 습관 등의 이유로 메시지가 과도하게 축약되거나 변경 의도를 충분히 반영하지 못하는 경우가 빈번하다. 이러한 문제를 완화하기 위해 커밋 메시지 자동 생성 연구가 지속되어 왔으며, 최근에는 코드 변경(diff)만을 입력으로 활용하는 LLM 기반 접근이 널리 시도되고 있다. 그러나 동일한 diff-only 조건에서 단일 에이전트(단일 추론으로 즉시 생성) 와 다중 에이전트(초안–비평–수정의 단계적 상호작용) 구조가 생성 결과에 미치는 영향을 체계적으로 비교한 연구는 상대적으로 제한적이다. 본 연구는 생성 구조의 효과만을 분리해 관찰하기 위해, 입력 정보(오직 diff), 데이터 분포, 전처리(최대 6,000자 제한), 모델 및 생성 규칙을 동일하게 통제한 상태에서 두 구조를 비교하였다. MCMD 데이터셋에서 8,000개 커밋을 추출해 실험을 수행하고, BLEU·METEOR·ROUGE-L로 정량 성능을 평가하는 동시에, 동일 입력에 대한 반복 생성 실험을 통해 Self-BLEU 기반 재현성(출력 안정성/변동성)을 분석하였다. 또한 자동 지표가 포착하기 어려운 차이를 보완하기 위해 일부 샘플에 대한 정성 분석과 추론 시간 측정을 병행하였다. 실험 결과, 단일 에이전트는 BLEU 1.28, METEOR 6.73, ROUGE-L 10.55로 다중 에이전트(1.18/6.02/10.08) 대비 자동 평가 지표에서 소폭 우세했으며, Self-BLEU 98.50으로 동일 입력에 대해 매우 높은 출력 일관성을 보였다. 반면 다중 에이전트는 Self-BLEU 1.28로 출력 변동성이 매우 컸고, 정성적으로는 변경 목적(예: 동시성 개선)이나 핵심 주제를 더 명시적으로 드러내며 파일명/구현 세부를 의미적으로 재구성하는 경향이 관찰되었다. 따라서 diff-only 환경에서 생성 구조 선택은 자동 평가 지표의 정답 표현 유사도, 동일 입력 반복 실행에서의 출력 안정성, 그리고 계산 비용 측면에서 서로 다른 특성을 보이며, 적용 목표가 무엇인지에 따라 합리적인 선택이 달라질 수 있음을 확인하였다.

1. 서론

소프트웨어 개발 과정에서 커밋 메시지는 코드 변경의 목적과 내용을 자연어로 요약하여 전달하는 핵심적인 산출물이다[1,2,3]. 커밋 메시지는 코드 리뷰 과정에서 변경 의도를 빠르게 파악하게 하고, 변경 이력 추적과 유지보수, 협업 과정에서 개발자의 이해를 돕는 중요한 역할을 수행한다[2,3]. 특히 대규모 프로젝트나 다수의 개발자가 참여하는 환경에서는 커밋 메시지가 코드 변경의 맥락을 보존하는 주요 수단으로 기능한다.

그러나 실제 개발 환경에서는 커밋 메시지가 불완전하거나 코드 변경 내용을 충분히 반영하지 못하는 경우가 빈번하게 관찰된다. 메시지가 지나치게 간략하거나 변경 의도와 직접적인 관련이 없는 표현으로 작성되는 사례도 적지 않다. 이러한 문제는 개발자의 작업 부담, 시간 제약, 또는 커밋 메시지 작성에 대한 낮은 우선순위 인식 등 다양한 요인에서 비롯된다. 그 결과, 커밋 메시지 자동 생성은 코드 변경 이력을 보다 체계적으로 관리하기 위한 중요한 소프트웨어 공학 연구 주제로 지속적으로 다루어져 왔다.

최근에는 대규모 언어 모델(Large Language Model, LLM)의 발전과 함께, 코드 변경(diff) 정보를 입력으로 활용하여 커밋 메시지를 자동으로 생성하는 접근법이 활발히 제안되고 있다. 이러한 접근법들은 주로 하나의 모델이 입력된 코드 변경을 바탕으로 메시지를 생성하는 단일 에이전트(single-agent) 구조를 중심으로 발전해 왔다. 단일 에이전트 기반 접근법은 구조가 단순하고 계산 비용이 상대적으로 낮으며, 대규모 데이터셋을 활용한 학습과 추론이 용이하다는 장점을 가진다. 실제로 다수의 기존 연구에서 단일 에이전트 기반 모델은 자동 평가 지표 기준에서 안정적인 성능을 보이는 것으로 보고되었다.

한편, 최근 자연어 처리 및 코드 생성 분야 전반에서는 하나의 모델이 아닌 여러 에이전트 간의 상호작용을 통해 출력을 개선하고자 하는 다중 에이전트(multi-agent) 기반 접근법이 주목받고 있다. 다중 에이전트 구조는 초안 생성, 비평, 수정과 같은 단계적 상호작용을 포함함으로써, 단일 모델이 놓칠 수 있는 오류를 보완하거나 출력의 표현을 점진적으로 개선하는 것을 목표로 한다. 이러한 접근법은 문서 요약, 질의 응답, 코드 생성 등 다양한 과제에서 출력의 풍부함이나 의미적 충실도를 향상시킬 가능성을 보여 왔다. 그러나 커밋 메시지 생성 과제에서 단일 에이전트와 다중 에이전트 기반 접근법을 동일한 조건 하에서 체계적으로 비교한 실험적 분석은 상대적으로 제한적이다. 기존 연구의 다수는 특정 접근법을 제안하고, 자동 평가 지표를 통해 성능을 보고하는 데 초점을 두었으며, 생성 구조 자체의 차이가 결과에 미치는 영향에 대해서는 충분히 분석하지 않았다. 특히 코드 변경(diff) 정보만을 입력으로 사용하는 diff-only 환경에서 두 접근법이 어떠한 생성 특성을 보이는지에 대한 비교 연구는 충분히 이루어지지 않았다.

또한 커밋 메시지 생성 연구에서 널리 사용되는 BLEU, METEOR, ROUGE-L과 같은 자동 평가 지표는 생성된 메시지와 정답(Gold Label) 메시지 간의 표면적 유사도를 중심으로 점수를 산출한다. 이러한 지표는 대규모 실험을 수행하는 데 효율적이지만, 생성된 메시지가 실제 코드 변경 내용을 얼마나 충실히 반영하는지, 혹은 변경의 목적을 명확히 전달하는지와 같은 정성적 특성을 충분히 설명하지 못하는 한계를 가진다. 특히 diff-only 환경에서는 코드 변경의 배경 맥락이 제한적으로 제공되기 때문에, 메시지의 표현 방식이나 정보 선택에 따라 자동 평가 지표 결과와 실제 유용성 간의 차이가 발생할 가능성이 존재한다.

이러한 문제의식에 기반하여, 본 연구는 diff-only 커밋 메시지 생성 환경에서 단일 에이전트와 다중 에이전트 기반 접근법을 동일한 실험 조건 하에서 비교 분석한다. 본 연구에서는 MCMD 데이터셋에서 추출한 8,000개의 커밋을 대상으로 대규모 실험을 수행하였으며[4], 모든 실험에서 입력 정보와 데이터 분포를

* 이 논문은 한경국립대학교 국립대학육성사업(2025)지원을 받아 작성되었음
† 교신저자(Corresponding Author)

동일하게 유지함으로써 생성 구조의 차이만이 결과에 영향을 미치도록 설계하였다. 생성된 커밋 메시지는 BLEU, METEOR, ROUGE-L을 사용하여 정량적으로 평가하였고[5,6,7], 동일 입력에 대한 반복 생성 실험을 통해 출력의 재현성(Self-BLEU)을 함께 분석하였다[8]. 또한 일부 샘플에 대해 정성적 사례 분석을 수행하여 자동 평가 지표로는 포착하기 어려운 생성 특성을 보완적으로 관찰하였다.

실험 결과, 단일 에이전트 기반 접근법은 자동 평가 지표 기준에서 상대적으로 높은 점수를 기록하며 안정적인 출력을 생성하는 경향을 보였다. 반면, 다중 에이전트 기반 접근법은 동일 입력에 대해 다양한 표현의 메시지를 생성하는 특성을 나타냈으며, 정성적 사례 분석에서는 코드 변경의 구체적인 내용이나 변경 목적을 보다 명시적으로 반영하는 사례가 다수 관찰되었다. 또한 재현성(Self-BLEU)과 추론 시간 분석을 통해, 두 접근법은 '정량 점수의 단순 우열'이라기보다 출력 안정성(일관성)과 표현 전략(다양성), 그리고 계산 비용의 관점에서 서로 다른 설계 선택을 유도함을 확인하였다.

본 논문의 주요 기여도는 다음과 같이 정리된다.

- 동일한 입력 조건을 유지한 diff-only 환경에서 단일 에이전트와 다중 에이전트 기반 커밋 메시지 생성 방식을 대규모로 비교하였다. MCMD 데이터셋에서 추출한 8,000개 커밋을 대상으로 입력 정보, 데이터 분포, 평가 절차를 동일하게 통제하여 생성 구조(single-agent vs. multi-agent) 차이만이 결과에 영향을 미치도록 설계했으며, 이를 통해 두 생성 구조 간 특성을 체계적으로 관찰하였다.
- BLEU, METEOR, ROUGE-L로 정량 평가하고, 동일 입력 반복 생성으로 Self-BLEU 기반 재현성을 분석했으며, 일부 샘플에 대해 정성 분석을 병행했다. 그 결과 단일 에이전트는 BLEU 1.28, METEOR 6.73, ROUGE-L 10.55 및 Self-BLEU 98.50으로 높은 점수와 안정성을 보였고, 다중 에이전트는 BLEU 1.18, METEOR 6.02, ROUGE-L 10.08 및 Self-BLEU 1.28로 상대적으로 낮은 점수와 높은 출력 변동성을 나타냈다. 정성 분석에서는 자동 지표로 포착하기 어려운 표현 특성과 코드 변경 반영 양상을 함께 확인했다.
- 실험을 통해 단일·다중 에이전트 구조가 자동 평가 지표 성능, 재현성, 표현 특성, 계산 비용에서 서로 다른 trade-off를 보임을 확인했다. 특히 다중 에이전트는 초안 생성–비평–수정 단계로 인해 단일 에이전트 대비 추론 비용이 약 4.26배 높아, diff-only 환경에서 생성 구조 선택이 성능뿐 아니라 출력 안정성과 계산 효율에도 영향을 준다는 점을 제시한다.

2. 배경지식

커밋 메시지는 코드 변경의 의도와 범위를 짧은 자연어로 요약해 개발자의 이해를 돕는 소프트웨어 산출물이다. 이 메시지는 변경 이력 추적과 유지보수, 코드 리뷰 과정에서 변경 목적을 빠르게 파악하게 하는 역할을 한다. 특히 대규모 프로젝트에서는 커밋 로그 자체가 개발 기록의 "요약 인덱스"처럼 기능하기 때문에, 메시지 품질이 협업 효율과 직결되는 경우가 많다. 그러나 실제 개발 환경에서는 시간 제약이나 우선순위 문제로 인해 메시지가 지나치게 짧거나(예: update, fix), 변경 내용과 충분히 대응하지 않는 경우가 발생한다. 또한 동일한 변경이라도 팀 또는 개인의 작성 습관에 따라 메시지 스타일이 달라 일관성이 깨지기도 한다. 이러한 문제를 보완하기 위해 커밋 메시지 자동 생성 연구가 지속적으로 수행되어 왔으며, 최근에는 대규모 언어 모델을 활용한 생성 방식이 주요 흐름으로 자리 잡고 있다. 본 장에서는 본 연구의 실험 설정과 결과 해석에 필요한 핵심 개념을 정리한다.

2.1 커밋 메시지 자동 생성 과제 개요

커밋 메시지 자동 생성은 코드 변경 정보로부터 변경의 핵심 내용을 자연어로 요약하는 과제이다. 일반적으로 입력은 한 커밋에서 발생한 변경 사항이며, 출력은 해당 변경을 설명하는 한 문장 또는 짧은 구 형태의 메시지이다. 이 과제는 "코드 변경을 자연어로 번역/요약한다"는 관점으로 다룰 수 있으며, 모델은 변경된 파일, 수정된 함수나 로직, 버그 수정이나 기능 추가와 같은 의도 정보를 메시지 형태로 압축해 전달해야 한다. 요약 관점에서 보면, 모델은 diff에 포함된 여러 단서 중 무엇을 핵심으로 선택할지(정보 선택)와, 그 핵심을 어떤 형식으로 표현할지(표현 전략)를 동시에 결정해야 한다.

커밋 메시지는 동일한 변경에 대해서도 허용 가능한 표현이 여러 가지 존재할 수 있다는 특성이 있다. 예를 들어 "Fix null check in parser"와 "Handle null input in parser"는 의미적으로 유사하지만 표면 표현은 다르다. 더 나아가 "Prevent NPE in parser"처럼 약어를 쓰거나, "parser: handle null input"처럼 범위 표기를 섞는 등 표현 형식도 다양하다. 또한 실제 데이터에서 정답 메시지 자체가 항상 변경을 완전하게 설명한다고 가정하기 어렵고, 프로젝트별 작성 관습(짧은 동사형, 범위 표기, 파일명 포함 여부 등)도 다르다. 따라서 커밋 메시지 자동 생성은 단순한 문자열 일치 문제라기보다, "변경의 핵심 정보를 선택해 요약하는 문제"로 이해하는 것이 적절하다. 이런 특성 때문에 연구에서는 정량 평가뿐 아니라, 생성 메시지가 변경 내용을 얼마나 충실히 반영하는지에 대한 정성적 관찰을 함께 수행하는 경우가 많다.

2.2 Diff와 diff-only 입력 설정

본 연구는 코드 변경을 git diff 형식으로 표현한 텍스트를 입력으로 사용한다. diff에는 파일 단위의 변경 경계와 함께 삭제 라인과 추가 라인이 포함되며, 변경 전후 차이가 라인 수준으로 드러난다. 이 입력은 변경된 코드 조각을 직접 제공한다는 점에서 메시지 생성에 필요한 최소 단서를 제공하지만, 변경의 배경이나 개발 의도와 같은 상위 문맥은 포함하지 않는다. 즉 diff는 "무엇이 바뀌었는가"를 비교적 직접적으로 보여주지만, "왜 바뀌었는가"는 충분히 설명하지 못할 수 있다. diff-only 설정은 입력을 diff 텍스트로 제한한다는 의미이며, 다음과 같은 정보는 입력에 포함되지 않는다. 커밋 메시지 히스토리, 저장소 설명, 이슈 트래킹이나 풀 리퀘스트 설명, 이전 커밋의 연속 맥락, 외부 문서나 검색 기반 지식 등이 이에 해당한다. 또한 AST와 같은 구조화된 코드 표현이나 별도의 정적 분석 결과도 사용하지 않는다. 이처럼 입력을 제한하면 모델이 활용할 수 있는 정보원이 명확해지고, 실험에서 관찰되는 차이가 "추가 문맥 덕분"인지 "생성 구조의 차이"인지를 구분하기가 쉬워진다. 본 연구가 diff-only 환경을 채택한 이유는 두 가지 측면에서 설명할 수 있다. 첫째, 생성 구조의 효과를 분리하기 위한 통제된 조건이 필요하다. 입력이 풍부해질수록 성능 변동의 원인이 늘어나고, 단일 에이전트와 다중 에이전트의 차이를 생성 구조 자체로 귀속하기 어려워진다. 둘째, diff-only는 커밋 메시지 생성에서 가장 기본적이며, 다양한 시스템에서 공통적으로 확보 가능한 입력이라는 장점이 있다. 즉 특정 플랫폼 정보(이슈/PR)나 추가 메타데이터에 의존하지 않고도 적용 가능한 설정이므로, 생성 구조 비교라는 본 연구 목적에 적합한 실험 기반을 제공한다.

2.3 단일 에이전트와 다중 에이전트 생성 구조

단일 에이전트 기반 생성은 하나의 언어 모델이 diff를 입력으로 받아 메시지를 한 번에 생성하는 구조를 의미한다. 입력과 출력 사이에 별도의 중간 산출물이나 수정 단계가 없고, 한 번의 추론 과정에서 최종 메시지가 만들어진다. 이 구조는 구현이 단순하고 계산 비용이 낮으며, 같은 입력에 대해 출력이 비교적 안정적으로 유지되는 경향이 나타날 수 있다. 또한 파이프라인 관점에서 실패 지점이 적고, 대규모 데이터셋 전체를 빠르게 처리하기에 유리하다. 반면, 자기 검토나 재작성 과정이 명시적으로 포함되지 않기 때문에, 모델이 초기 선택한 정보(예: 파일명, 함수명, 키워드)에 과도하게 의존하거나, 변경 목적을 충분히 명시하지 못한 상태로 종료될 가능성도 존재한다. 다중 에이전트 기반 생성은 여러 역할의 에이전트가 단계적으로 상호작용하며 결과를 만드는 구조이다. 본 연구에서는 초안 생성, 비평, 수정의 순서를 갖는 3단계 절차로 구성한다. 먼저 초안 단계에서 diff를 요약한 초기 메시지를 만든 뒤, 비평 단계에서 초안과 diff의 대응 관계, 표현의 명확성, 누락 정보, 불필요한 세부 정보 포함 여부 등을 점검하고, 마지막 수정 단계에서 비평을 반영해 최종 메시지를 재작성한다. 이때 핵심은 입력 정보가 늘어나는 것이 아니라, 같은 diff를 두고 "한 번 더 점검하고 고쳐 쓰는 절차"가 추가된다는 점이다. 다중 에이전트 구조는 출력이 한 번에 고정되지 않고

단계적으로 수정되기 때문에, 결과적으로 메시지 표현이 더 다양해지거나 특정 변경 목적이 더 명시적으로 반영되는 양상이 나타날 수 있다. 예를 들어 초안이 단순한 변경 나열에 머물렀다면, 비평 과정에서 "목적(왜) 또는 효과(무엇이 좋아졌는가)"를 강조하도록 유도될 수 있다. 반면 단계 수만큼 추론이 추가되므로 계산 비용이 증가하며, 동일 입력이라도 실행마다 표현이 달라질 가능성이 커진다. 특히 각 단계가 자연어로 피드백을 생성하는 방식이라면, 작은 표현 차이가 다음 단계로 전파되면서 최종 출력 변동성을 키울 수 있다. 따라서 다중 에이전트 접근은 품질 향상 가능성과 함께, 비용과 안정성 측면의 trade-off를 동반한다.

2.4 자동 평가 지표와 재현성 평가

커밋 메시지 생성 연구에서는 대규모 실험을 위해 자동 평가 지표가 널리 사용된다. 본 연구는 BLEU, METEOR, ROUGE-L을 사용해 생성 메시지와 정답(Gold Label) 메시지 사이의 표면적 유사도를 정량화한다. BLEU는 주로 n-gram 중첩을 기반으로 유사도를 측정하며, ROUGE-L은 최장 공통 부분 수열을 활용해 요약 과제에서의 유사도를 평가한다. METEOR는 단어 정렬 및 변형을 고려해 유사도를 산출하는 방식으로 사용된다. 이들 지표는 동일한 데이터셋에서 여러 접근법의 성능 경향을 비교하는 데 유용하고, 실험 재현과 보고가 상대적으로 용이하다는 장점이 있다. 다만 이들 지표는 기본적으로 정답 메시지와의 표면적 중첩을 중심으로 점수를 계산한다. 따라서 의미적으로 타당한 다른 표현이 정답과 겹치지 않는 경우 점수가 낮게 나올 수 있으며, 정답 메시지가 변경 의도를 충분히 반영하지 못하는 경우에도 평가가 정답 중심으로 고정된다는 한계가 있다. 특히 커밋 메시지는 길이가 짧아 단어 하나의 선택이 점수에 미치는 영향이 클 수 있고, 스타일 차이(동사 선택, 범위 표기, 약어 사용)가 중첩도를 크게 흔들 수 있다. 그래서 자동 지표 점수는 "정답 표현과의 유사도"를 측정하는 관찰값으로 해석하고, 생성 결과의 특성은 다른 분석과 함께 종합적으로 판단할 필요가 있다. 본 연구는 출력 특성의 또 다른 축으로 재현성을 함께 분석한다. 재현성은 동일한 diff 입력에 대해 반복 실행 시 출력이 얼마나 일관되게 유지되는지를 의미한다. 이를 정량화하기 위해 Self-BLEU를 사용하며, 동일 입력에 대해 여러 번 생성된 출력들 간의 n-gram 중첩 정도를 계산한다. Self-BLEU가 높으면 출력이 거의 동일하게 반복되는 경향을 의미하고, 낮으면 동일 입력에서도 표현이 다양하게 변동함을 의미한다. 커밋 메시지는 허용 가능한 표현이 여러 가지일 수 있으므로 출력 다양성이 반드시 부정적인 특성은 아니지만, 자동화 파이프라인이나 동일 결과의 반복 생성이 필요한 환경에서는 높은 변동성이 실용적 제약으로 작용할 수 있다. 따라서 본 연구는 자동 평가 지표(유사도), 재현성(Self-BLEU), 정성적 사례 관찰을 함께 사용하여 생성 구조가 만들어내는 차이를 다각도로 해석한다. 즉, 자동 지표가 보여주는 표면적 유사도 경향과, 반복 실행 시 출력이 안정적으로 유지되는지, 그리고 실제 diff 반영 방식이 어떻게 달라지는지를 함께 보면서 단일 에이전트와 다중 에이전트 생성 구조의 차이를 정리한다.

3. 제안한 방법

본 장에서는 diff-only 조건에서 단일 에이전트와 다중 에이전트(Writer–Critic–Refiner) 생성 구조를 동일 설정으로 비교하기 위해, 각 구조의 생성 절차를 요약한다.

두 접근법은 모두 코드 변경 사항을 나타내는 diff 텍스트만을 입력으로 사용하며, 데이터 구성, 입력 전처리, 모델 설정, 생성 규칙은 동일하게 유지된다. 차이는 커밋 메시지에 도달하기까지의 내부 생성 절차에 있으며, 단일 에이전트 기반 접근법은 하나의 추론 단계에서 메시지를 생성하는 반면, 다중 에이전트 기반 접근법은 여러 단계의 역할 분리를 통해 메시지를 점진적으로 완성한다. 본 장에서는 이러한 생성 절차의 흐름과 각 단계가 수행하는 기능을 중심으로 두 접근법을 설명한다.

그림 1은 본 연구에서 비교하는 단일 에이전트 기반 생성 구조와 다중 에이전트 기반 생성 구조의 전체 흐름을 나타낸다. 두 접근법은 동일한 데이터로부터 동일한 diff 입력을 받아 메시지를 생성하며, 생성된 결과는 동일한 평가 절차를 통해 분석된다. 단일 에이전트 기반 기준선에서는 하나의 언어 모델이 입력된 코드 변경(diff)을 받아, 추가적인 중간 단계 없이 최종 커밋 메시지를 직접 생성한다. 이 구조에서는 입력과 출력이 단일 추론 단계에서 연결되며, 메시지 생성 과정이 비교적 단순하게 구성된다. 생성 과정에서 초안이나 검토 단계가 분리되지 않기 때문에, 모델은 diff에 포함된 변경 단서를 한 번의 생성 과정에서 선택하고 요약하여 최종 문장을 구성한다. 이러한 구조에서는 동일한 입력에 대해 반복 실행 시 출력이 유사한 형태로 유지되는 경향이 나타날 수 있으며, 계산 비용 또한 상대적으로 낮게 유지된다. 반면, 다중 에이전트 기반 생성 구조에서는 동일한 diff 입력을 유지한 상태에서 초안 생성, 비평, 수정의 단계가 순차적으로 수행된다. 초안 단계에서는 단일 에이전트 기반 방식과 동일한 입력 조건 하에서 초기 커밋 메시지가 생성되며, 이후 비평과 수정 단계는 해당 초안을 검토하고 재구성하는 역할만을 수행한다. 이 과정에서 입력 정보가 추가되거나 확장되지는 않으며, 모든 단계는 동일한 diff 텍스트를 공통 기준으로 참조한다. 즉, 다중 에이전트 기반 구조는 입력 정보의 확장이 아닌 생성 과정의 절차적 분리를 통해 최종 메시지에 도달하는 방식으로 정의된다. 이와 같이 두 접근법은 입력 데이터, 데이터 분포, 모델 설정을 동일하게 유지하면서, 최종 메시지를 생성하기까지의 절차만을 다르게 구성한다. 따라서 이후 실험 장에서 관찰되는 자동 평가 지표 성능, 출력의 재현성, 추론 시간 차이는 입력 정보나 데이터 조건의 차이가 아니라, 그림 1에 요약된 생성 구조의 차이에서 비롯된 결과로 해석된다.

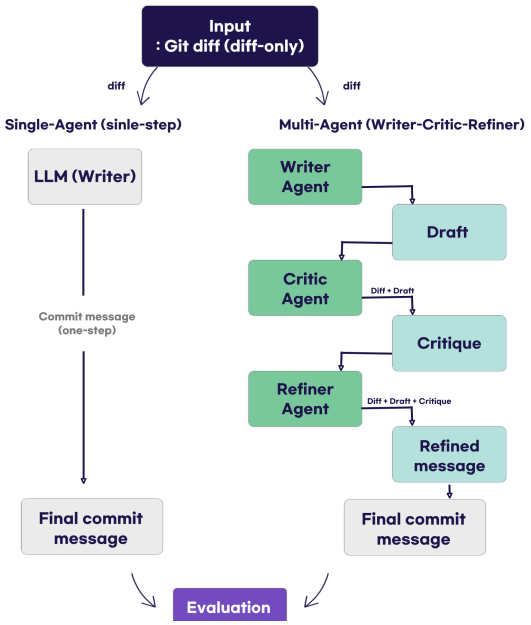


그림 1. 단일 에이전트 기반 기준선과 다중 에이전트 기반 커밋 메시지 생성 프레임워크의 전체 흐름

3.1 단일 에이전트 기반 커밋 메시지 생성

단일 에이전트 기반 접근법에서는 하나의 언어 모델이 입력으로 제공된 코드 변경 사항(diff)을 바탕으로 커밋 메시지를 생성한다. 모델은 git diff 형식으로 표현된 코드 변경 텍스트 전체를 하나의 입력 시퀀스로 받아들이며, 이에 대응하는 단일 커밋 메시지를 출력하도록 구성된다. 이 과정은 입력으로부터 최종 출력이 직접 생성되는 단일 단계의 생성 구조로 이루어지며, 초안 생성, 검토, 수정과 같은 중간 단계는 포함하지 않는다. diff-only 입력은 파일 경로, 수정된 코드 구문, 함수 및 변수와 같은 심볼 이름, 그리고 코드 라인의 추가 및 삭제 정보 등을 텍스트 형태로 포함한다. 단일 에이전트는 이러한 변경 단서를 하나의 입력 공간에서 동시에 고려하여, 코드 변경의 핵심 내용을 요약한 문장을 생성한다. 생성된 커밋 메시지는 일반적으로 변경된 대상이나 변경의 성격을 중심으로 구성되며, "무엇이 변경되었는가"를 간결하게 서술하는 형태를 따른다. 이때 메시지는 한 문장 또는 이에 준하는 짧은 텍스트로 제한되며, diff에 명시적으로 나타난 정보에 기반하여 생성된다. 단일 에이전트 기반 접근법의 주요 특징은 생성 과정이 단일 경로로 고정되어 있다는 점이다. 동일한 diff 입력이 주어질 경우, 모델은 한 번의 생성 과정만을 수행하여 메시지를 완성하며,

생성 결과를 다시 평가하거나 수정하는 절차는 존재하지 않는다. 이로 인해 출력은 모델의 내부 표현과 학습된 생성 패턴에 따라 특정 표현 양식으로 수렴하는 경향을 보일 수 있다. 또한 반복 실행 시에도 유사한 문장 구조나 어휘 선택이 나타나는 경우가 관찰된다. 단일 에이전트는 diff를 입력으로 단 한 번의 추론으로 최종 커밋 메시지를 생성한다. 본 연구에서는 이를 비교 기준선으로 두고, 이후 절의 다중 에이전트 절차와 동일 조건에서 비교한다.

3.2 다중 에이전트 생성 구조 개요 **(Writer–Critic–Refiner)**

다중 에이전트 기반 접근법은 하나의 언어 모델이 입력으로부터 최종 커밋 메시지를 즉시 생성하는 방식과 달리, 서로 다른 역할을 부여받은 에이전트들이 단계적으로 상호작용하면서 메시지를 완성하는 생성 구조이다. 본 연구는 diff-only 환경에서 "입력 정보의 확장 없이도" 생성 절차의 구조적 분리가 결과에 어떤 영향을 주는지 관찰하기 위해, 커밋 메시지 생성 과정을 초안 생성(Writer)–비평(Critic)–수정(Refiner)의 3단계로 분리하여 정의하였다. 각 단계는 동일한 diff 텍스트를 공통 기준으로 참조하되, 역할(관점)만 달리하여 같은 입력을 반복적으로 해석하도록 설계된다.

다중 에이전트 절차에서 핵심은 "더 많은 정보를 넣는 것"이 아니라 "같은 정보를 더 체계적으로 읽게 만드는 것"이다. 즉 Writer가 diff를 보고 1차 메시지를 생성한 뒤, Critic이 초안의 근거 적합성과 문장 규칙 준수 여부를 점검하고, Refiner가 비평을 반영해 최종 문장을 재작성한다. 이때 단계 간에 전달되는 것은 외부 문맥이나 추가 지식이 아니라, 생성 과정 내부에서 산출된 초안과 비평 텍스트뿐이다. 따라서 다중 에이전트는 retrieval, 이슈/PR 설명, 저장소 문서 등 외부 신호를 이용해 메시지를 풍부하게 만드는 방식이 아니라, diff-only 조건하에서 생성 과정 자체의 검토·수정 루프가 메시지 구성에 미치는 영향을 분리해 관찰할 수 있게 한다.

또한 본 연구는 각 단계에서 에이전트의 역할을 명확히 고정하기 위해 시스템 프롬프트를 사용한다. Writer는 전문 소프트웨어 엔지니어로서 간결한 커밋 메시지를 작성하며, Critic은 코드 리뷰어로서 초안의 문제점을 지적하는 비평만 출력하고, Refiner는 비평을 반영해 최종 메시지만 출력하도록 제한한다. 이러한 역할 분리는 다중 에이전트 절차가 단순히 "한 번 더 생성한다"는 형태로 흐려지지 않게 하며, 단계별 목적(초안 작성 vs 근거 점검 vs 재작성)이 실험적으로 구분되도록 한다.

결과적으로 다중 에이전트 생성 구조는 동일한 diff-only 입력을 기반으로 하되, 초안에서 선택된 변경 초점이 비평을 통해 재검토될 수 있고, 누락되거나 과도하게 일반화된 표현이 수정 단계에서 보완될 수 있으며, 규칙 위반(예: 동사로 시작하지 않음, 지나치게 장황함, 모호한 서술)이 최종 문장에 남지 않도록 하는 절차적 장치를 제공한다. 본 연구는 이와 같은 구조적 차이가 자동 평가 지표, 재현성(Self-BLEU), 그리고 정성적 사례에서 어떤 차이로 나타나는지를 비교 분석한다.

3.3 단계별 생성 절차와 프롬프트 구성

본 절에서는 다중 에이전트 기반 생성 구조의 단계별 절차와 프롬프트 구성을 요약한다. 모든 조건은 diff-only로 통제되며, 입력은 전처리된 DIFF만 사용한다(최대 6,000자). 저장소/이슈/PR 정보, 커밋 히스토리, 외부 문서·검색 지식, AST/정적 분석 결과 등 추가 문맥은 사용하지 않는다.

다중 에이전트는 Writer–Critic–Refiner의 3단계로 구성된다. Writer는 DIFF로부터 1줄 커밋 메시지 초안을 생성한다. Critic은 DIFF와 초안을 대조하여 근거 부족, 누락/과잉 정보, 명확성·간결성, 규칙 위반 여부를 불릿 형태로 비평하며(커밋 메시지 출력 금지), Refiner는 DIFF+초안+비평을 입력으로 받아 최종 1줄 커밋 메시지만 출력한다. 단일 에이전트는 DIFF를 입력으로 1회 생성으로 최종 1줄 메시지를 출력한다. 모든 모드에서 출력은 "한 줄", "명령형 동사로 시작", "마침표 없음"으로 제한한다.

생성 파라미터와 디코딩 설정은 단일 에이전트와 다중 에이전트 조건에서 동일하게 고정하였다. 재현성 평가는 동일 DIFF 입력에 대해 각 접근을 N=10회 반복 생성한 출력들로부터 Self-BLEU를 계산하여 산출하였다.

아래는 각 모드의 입력과 출력 제약을 정리한 것이다.

모드	입력	출력 / 제약
단일 에이전트 (Single-Agent)	DIFF (최대 6,000자)	최종 커밋 메시지 1줄 (명령형 동사로 시작, 마침표 없음)
Writer	DIFF (최대 6,000자)	초안 커밋 메시지 1줄 (동일 제약)
Critic	DIFF + 초안(Draft)	비평만 출력(불릿); 커밋 메시지 출력 금지
Refiner	DIFF + 초안(Draft) + 비평(Critique)	최종 커밋 메시지 1줄 (동일 제약)

4. 실험

4.1 실험 세팅

본 연구는 diff-only 환경에서 단일 에이전트 기반과 다중 에이전트 기반 커밋 메시지 생성 방식을 동일 조건에서 비교한다. 모든 실험에서 모델 입력은 코드 변경 사항을 나타내는 diff 텍스트로만 제한했으며, 파일 경로, 커밋 히스토리, 이슈 설명 등 추가 메타데이터와 정답 커밋 메시지(Gold Label)는 생성 과정에 제공하지 않고 평가 단계에서만 활용했다. 또한 두 접근법은 동일한 커밋 샘플 집합과 동일한 데이터 분포를 사용하여, 결과 차이가 데이터 구성이나 입력 범위가 아니라 생성 구조에서 비롯되도록 통제하였다. 생성 품질은 BLEU, METEOR, ROUGE-L로 정량 평가했으며, 단일 출력 평가의 한계를 보완하기 위해 동일 diff에 대한 반복 생성 실험을 수행하고 Self-BLEU로 출력의 안정성과 변동성을 분석했다. 추가로 계산 비용 차이를 확인하기 위해 추론 시간(inference time)도 함께 측정하였다. 실험은 동일 환경(Ubuntu 24.04.2 LTS, Intel Xeon Silver 4310 2.10GHz논리 코어 24개, RAM 125 GiB, RTX 4090 24GB ×2, NVIDIA Driver 560.35.05, CUDA driver 12.6, nvcc 11.8)에서 수행하였다.

4.2 데이터셋 및 평가 척도

본 실험에서는 MCMD 데이터셋을 사용하였다. 본 연구는 MCMD에서 총 8,000개 커밋을 추출하여 사용했으며, 프로그래밍 언어 간 분포 불균형이 결과에 영향을 미치는 것을 줄이기 위해 주요 언어별로 균등한 비율을 유지하도록 샘플을 구성하였다. 각 샘플은 git diff 형식의 텍스트와 이에 대응하는 정답 커밋 메시지로 이루어지며, 정답 메시지는 생성 과정에는 제공하지 않고 평가 단계에서만 활용하였다. 또한 입력 길이 제약을 고려하여 diff 텍스트는 최대 6,000자까지만 사용하도록 전처리하였다.

정량적 성능 평가는 생성된 커밋 메시지와 정답 메시지 간의 표현적 유사도를 측정하기 위해 BLEU, METEOR, ROUGE-L 자동 평가 지표를 사용하였다. BLEU 지표는 n-gram 단위의 중첩 정도를 기반으로 생성 문장과 정답 문장 간의 유사도를 측정하며[5], METEOR 지표는 단순한 n-gram 중첩뿐만 아니라 단어 정렬 기반의 일치 정도를 함께 반영한다[6]. ROUGE-L 지표는 생성 결과와 정답 메시지 간의 최장 공통 부분 수열(Longest Common Subsequence)을 기반으로 문장 수준의 중첩 정도를 측정한다[7]. 본 연구는 커밋 메시지가 일반적으로 길이가 짧고, 동일한 코드 변경을 요약하는 표현이 다양한 형태로 나타날 수 있다는 점을 고려하여, 단일 평가 지표에 의존하지 않고 세 가지 지표를 함께 사용하여 결과를 분석하였다. 이를 통해 각 지표가 포착하는 표현적 유사도의 서로 다른 측면을 종합적으로 관찰할 수 있도록 하였다. 추가적으로, 생성 결과의 재현성 및 출력 안정성을 분석하기 위해 Self-BLEU 지표를 사용하였다. Self-BLEU는 동일한 입력 diff에 대해 반복적으로 생성된 커밋 메시지들 간의 유사도를 측정하는 지표로, 반복 실행 시 출력이 어느 정도 일관되게 유지되는지를 정량적으로 나타낸다. 본 연구에서는 Self-BLEU 값이 상대적으로 높게 나타나는 경우 동일 입력에 대해

생성 결과가 유사하게 유지되는 경향이 크다고 해석하였으며, 반대로 값이 낮게 나타나는 경우 출력 표현의 변동성이 크게 나타나는 경향이 있는 것으로 정의하였다 [8]. 이를 통해 단일 에이전트와 다중 에이전트 접근법이 생성 안정성 측면에서 어떠한 차이를 보이는지를 함께 분석하였다. 본 연구에서 Self-BLEU는 동일 diff 입력에 대해 N회 반복 생성한 출력 집합을 구성한 뒤, 출력들 간 중첩도를 계산하여 산출하였다. 여기서 N은 모델 내부가 아니라 실험 실행 스크립트(반복 생성 루프)에서 고정된 반복 횟수이며, 본 실험에서는 N=10으로 설정하였다. 또한 다중 에이전트(Writer–Critic–Refiner)는 초안과 비평이 다음 단계 입력으로 전달되는 구조이므로, 초기 단계의 작은 표현 차이가 누적되어 동일 diff에 대해서도 최종 메시지의 변동성이 커질 수 있다. 출력 변동성은 디코딩 설정(temperature, top-p 등)과 프롬프트 제약에 의해 달라질 수 있으므로, 본 연구는 반복 횟수(N)와 생성 설정을 실험 실행 스크립트에 명시하여 재현 가능하도록 관리하였다.

4.3 실험 결과

본 절에서는 diff-only 환경에서 단일 에이전트 기반 커밋 메시지 생성 접근법과 다중 에이전트 기반 접근법이 자동 평가 지표 기준에서 어떠한 정량적 차이를 보이는지를 분석한다. 정량 평가는 BLEU, METEOR, ROUGE-L 지표를 사용하여 수행되었으며, MCMD 데이터셋에서 추출한 총 8,000개 커밋 샘플 전체에 대해 각 지표의 평균 점수를 계산하였다.

모든 실험 조건에서는 입력으로 제공되는 정보의 범위, 데이터 분포, 실행 환경이 동일하게 유지되었으며, 이러한 설정은 생성 구조(단일 에이전트 vs. 다중 에이전트)의 차이만이 평가 결과에 영향을 미치도록 통제하기 위한 것이다. 각 접근법의 정량적 성능 비교 결과는 표 1에 정리되어 있다.

표 1. 단일 에이전트와 다중 에이전트 기반 커밋 메시지 생성의 정량적 성능 비교

Generation Method	BLEU	METEOR	ROUGE-L
Single Agent	1.28	6.73	10.55
Multi-Agent (Writer–Critic–Refiner)	1.18	6.02	10.08

표 1은 동일한 diff-only 조건에서 Single-Agent와 Multi-Agent(Writer–Critic–Refiner) 생성 구조를 비교한 자동 평가 지표 결과를 요약한다. 표 1에 제시된 결과에 따르면, 단일 에이전트 기반 접근법은 BLEU, METEOR, ROUGE-L의 세 가지 자동 평가 지표 모두에서 다중 에이전트 기반 접근법보다 높은 평균 점수를 기록하였다. BLEU 점수는 단일 에이전트가 1.28, 다중 에이전트가 1.18로 나타나 약 0.10의 절대 차이가 관찰되었으며, METEOR 점수는 각각 6.73과 6.02로 약 0.71의 차이를 보였다. ROUGE-L 점수는 단일 에이전트가 10.55, 다중 에이전트가 10.08로 약 0.47의 차이가 확인되었다.

모든 지표에서 단일 에이전트 기반 접근법이 상대적으로 높은 점수를 기록하였으나, 각 지표에서 관찰된 차이는 절대값 기준으로는 비교적 제한적인 범위 내에 머물러 있다는 점도 함께 확인된다. 이는 두 접근법 모두 diff-only 환경에서 커밋 메시지를 생성할 수 있으나, 자동 평가 지표가 포착하는 표현적 유사도 측면에서 생성 구조에 따른 차이가 소폭의 수치 차이로 반영되었음을 의미한다. 이러한 결과는 자동 평가 지표의 평가 방식과 생성 구조의 차이를 함께 고려하여 해석할 수 있다. BLEU, METEOR, ROUGE-L 지표는 공통적으로 생성된 메시지와 정답 커밋 메시지(Gold Label) 간의 n-gram 수준 또는 문장 구조 수준의 중첩 정도를 기반으로 점수를 산출한다. 단일 에이전트 기반 접근법은 하나의 추론 과정에서 커밋 메시지를 생성하는 구조를 가지며, 이 과정에서 비교적 간결하고 일반화된 표현을 선택하는 경향이 관찰되었다. 이러한 특성은 정답 메시지가 짧고 요약적인 형태로 구성된 경우, 단어 또는 구 수준의 중첩을 유지하는 방향으로 작용할 가능성이 있다.

반면, 다중 에이전트 기반 접근법은 초안 생성 이후 비평 및 수정 단계를 거치는 다단계 생성 구조를 가지며, 이 과정에서 코드 변경의 세부 요소를 보다 명시적으로 포함하거나 변경의 목적과 효과를 자연어로 확장하여 서술하는 경향이 관찰되었다. 이러한 생성 특성은 메시지의 정보량을 증가시키는 방향으로 작용할 수 있으나, 동시에 정답 메시지와의 표현적 일치도 또는 단어 수준 중첩을 감소시키는 방향으로 작용할 가능성이 있다. 그 결과, 자동 평가 지표 기준에서는 다중 에이전트 기반 접근법의 점수가 상대적으로 낮게 나타나는 양상이 관찰될 수 있다. 재현성 분석은 동일한 코드 변경(diff) 입력에 대해 커밋 메시지 생성을 여러 차례 반복 수행한 후, 반복 실행 과정에서 생성된 출력 간의 유사도를 정량적으로 측정하는 방식으로 수행되었다. 본 분석의 목적은 단일 에이전트 기반 접근법과 다중 에이전트 기반 접근법이 동일 입력 조건에서 어느 정도 일관된 출력을 생성하는지, 혹은 반복 실행 시 출력 표현이 어느 정도 변동하는지를 비교 관찰하는 데 있다. 본 연구에서는 동일 입력에 대한 출력 간 유사도를 수치화하기 위해 Self-BLEU 지표를 사용하였다. Self-BLEU는 하나의 생성 결과를 기준 문장(reference)으로 두고, 동일 입력으로부터 생성된 다른 출력들을 비교 대상으로 삼아 BLEU 점수를 계산하는 방식으로 산출된다. 이 지표는 반복 생성 결과 간의 표현적 중첩 정도를 나타내며, Self-BLEU 값이 높을수록 반복 실행 시 생성 결과가 서로 유사하게 유지되는 경향이 크고, 값이 낮을수록 출력 표현의 다양성 또는 변동성이 크게 나타나는 경향이 있음을 의미한다. 재현성 분석 결과는 표 2에 정리되어 있다.

표 2. 동일 입력에 대한 단일 에이전트와 다중 에이전트의 재현성(Self-BLEU) 비교

Generation Method	Consistency Score (Self-BLEU)	Interpretation
Single Agent	98.50	매우 높음
Multi-Agent	1.28	매우 낮음

표 2에 제시된 결과에 따르면, 단일 에이전트 기반 접근법은 Self-BLEU 98.50이라는 매우 높은 값을 기록하였다. 이는 동일한 코드 변경(diff) 입력에 대해 커밋 메시지 생성을 반복 수행하였을 때, 생성된 메시지들이 표현적으로 거의 동일한 형태를 유지하는 경향이 매우 강하게 나타났음을 의미한다. 즉, 반복 실행 간 출력 간 중첩 정도가 높아, 출력 안정성 측면에서 높은 일관성을 보였음을 확인할 수 있다.

반면, 다중 에이전트 기반 접근법은 Self-BLEU 1.28이라는 매우 낮은 값을 기록하였다. 이는 동일한 코드 변경 입력에 대해서도 반복 실행 시 생성된 커밋 메시지들 간의 표현적 중첩이 극히 낮게 나타났음을 의미하며, 실행마다 서로 다른 표현의 메시지가 생성되는 경향이 크게 나타났음을 보여준다. 이러한 결과는 다중 에이전트 기반 접근법이 동일 입력에 대해서도 다양한 표현을 생성하는 특성을 갖고 있음을 수치적으로 반영한 것으로 볼 수 있다. 단일 에이전트와 다중 에이전트 기반 접근법 간의 Self-BLEU 점수 차이는 약 97.22 포인트로 나타났으며, 이는 동일 입력 조건 하에서 출력의 안정성 및 변동성 측면에서 두 접근법이 매우 상이한 특성을 보였음을 정량적으로 보여주는 결과로 정리할 수 있다. 본 재현성 분석 결과는 앞선 정량 평가 결과와 함께, 생성 구조의 차이가 반복 실행 시 출력 특성에 어떠한 영향을 미치는지를 이해하기 위한 보조적 근거로 활용된다. 정성적 사례 분석은 앞선 정량 평가 결과(BLEU, METEOR, ROUGE-L) 및 재현성 분석(Self-BLEU)에서 관찰된 차이를 보다 구체적으로 이해하기 위해 수행되었다. 본 분석의 목적은 단일 에이전트 기반 접근법과 다중 에이전트 기반 접근법이 동일한 코드 변경(diff) 입력에 대해 어떠한 방식으로 커밋 메시지를 구성하는지, 그리고 그 생성 특성이 출력 내용과 표현 구조에 어떠한 차이로 나타나는지를 질적으로 확인하는 데 있다. 분석 대상은 전체 데이터셋에서 무작위로 선택된 커밋 샘플로 구성되었으며, 각 샘플에 대해 동일한 diff 입력을 기준으로 단일 에이전트와 다중 에이전트가 생성한 커밋 메시지를 직접 비교하였다. 이 과정에서 출력 길이, 포함된 정보의 범위, 변경 내용에 대한 요약 방식, 표현의 추상화 수준 등을 중심으로 관찰을 수행하였다. 분석 결과, 단일 에이전트 기반 접근법은 동일 입력에 대해 출력 표현이 거의 변하지 않는 경향이 관찰되었으며, 이는 재현성 분석에서 확인된 높은 Self-BLEU 값과 일관된 양상으로 나타났다. 단일 에이전트는 하나의 추론 과정에서 메시지를 생성하는 구조를 가지며, diff에

포함된 변경 요소를 비교적 포괄적으로 나열하거나 기술하는 방식으로 메시지를 구성하는 경향이 관찰되었다. 이로 인해 변경된 파일명, 구성 요소, 세부 구현 항목 등이 메시지에 직접적으로 포함되는 사례가 다수 확인되었다.

반면, 다중 에이전트 기반 접근법은 비평 및 수정 단계를 거치는 과정에서 표현 방식과 정보의 강조점이 달라지며, 동일 입력에 대해서도 최종 출력이 여러 표현 후보로 분산되는 경향이 관찰되었다. 이러한 특성은 재현성 분석에서 낮은 Self-BLEU 값으로 나타난 출력 변동성과 대응되는 결과이다. 다중 에이전트는 초기 초안 이후 메시지를 반복적으로 검토·수정하는 구조를 가지며, 이 과정에서 변경 내용을 하나의 핵심 개념이나 목적 중심으로 재구성하는 경향이 관찰되었다.

Self-BLEU 관점에서 Single-Agent의 높은 값은 "동일 입력에 대해 거의 동일한 출력을 재생성할 수 있다"는 의미로, 자동화된 커밋 파이프라인이나 대규모 배치 생성에서 운영 안정성 측면의 장점이다. 반대로 Multi-Agent의 매우 낮은 Self-BLEU는 "동일 diff에서도 결과가 크게 흔들린다"는 뜻이므로, CI/CD나 규정 준수가 필요한 환경에서는 신뢰성 리스크로 작동할 수 있다. 다만 이 변동성은 사람이 최종 선택을 하거나 후보 중 하나를 고르는 워크플로우(예: top-k 후보 생성 후 선택)에서는 '다양한 요약 관점'을 제공하는 장점이 될 수 있다. 따라서 Multi-Agent를 실용적으로 쓰려면 디코딩을 결정적으로 고정하거나, Critic 점수 기반 리랭킹/검증을 추가하거나, 출력 형식 제약을 강화하는 방식으로 변동성을 통제하는 보완책이 필요하다.

구체적인 사례를 통해 이러한 차이를 확인할 수 있다.
Cython 관련 문서 설정 변경 사례에서는, 정답(Gold Label) 메시지가 실제 코드 변경 내용과 직접적인 관련이 없는 라벨로 구성된 경우가 관찰되었다. 이 사례에서 단일 에이전트는 README 파일에 포함된 설치 방법, 스크립트 템플릿, 정의 설명, 실행 방식 등 diff에 포함된 세부 항목을 메시지에 명시적으로 포함하는 양상을 보였다.

<Single-Agent Output>
"Update README.md with Cython installation instructions, script template, and enhanced documentation on definitions and running Cython code"

반면, 다중 에이전트 기반 접근법은 동일한 변경 내용을 하나의 핵심 개념으로 압축하여 표현하는 양상을 보였다.

<Multi-Agent Output>
"Update README.md with Cython setup"

이 사례는 다중 에이전트가 세부 변경 내용을 포괄적으로 나열하기보다는, 변경의 중심 주제를 추출하여 요약 수준을 높인 메시지를 생성하는 경향을 보여준다.

유사한 경향은 자료구조 변경 사례에서도 관찰되었다. HashMap에서 ConcurrentHashMap으로의 변경 사례에서 단일 에이전트는 변경 대상 클래스, 사용 위치, 교체된 자료구조를 메시지에 포함하여 구현 세부 중심의 설명을 제공하였다.

<Single-Agent Output>
"Update Shardingsphere's InlineExpressionParser to use ConcurrentHashMap for SCRIPTS, replacing HashMap for better concurrency"

반면, 다중 에이전트는 동일 변경을 변경의 목적(동시성 개선)을 중심으로 요약하여 표현하였다.

<Multi-Agent Output>
"Update InlineExpressionParser to use ConcurrentHashMap for improved concurrency"

이 사례에서는 단일 에이전트가 '무엇이 어떻게 바뀌었는지'에 초점을 둔 반면, 다중 에이전트는 '왜 변경되었는지'를 중심으로 메시지를 구성하는 경향이 나타났다.

또한 번역 파일 수정 사례에서는, 단일 에이전트가 diff에 포함된 파일명(cs.json)을 그대로 메시지에 포함하는 경향을 보인 반면, 다중 에이전트는 해당 파일이 체코어 번역 파일임을 자연어로 해석하여 표현하는 양상이 관찰되었다.

<Single-Agent Output>
"Update translations for components: cs.json, adding and modifying entries…"

<Multi-Agent Output>
"Update translations for ACMeda in Czech"

이 사례는 다중 에이전트가 파일명이나 코드 수준 정보보다는 의미적 해석을 기반으로 메시지를 재구성하는 경향을 갖고 있음을 보여준다. 따라서, 정성적 사례 분석 결과는 단일 에이전트 기반 접근법과 다중 에이전트 기반 접근법이 동일한 diff 입력을 서로 다른 관점에서 요약하고 표현한다는 점을 보여준다. 단일 에이전트는 변경된 요소를 비교적 포괄적으로 기술하며 출력의 일관성이 높은 반면, 다중 에이전트는 변경의 핵심 개념이나 목적을 중심으로 메시지를 재구성하고, 그 과정에서 표현의 다양성이 크게 나타나는 경향이 관찰되었다. 이러한 정성적 관찰은 앞선 정량 평가 및 재현성 분석 결과에서 확인된 수치적 차이를 구체적인 출력 사례 수준에서 보완적으로 설명하는 근거로 활용될 수 있다. 시간 비용 분석에서는 단일 에이전트 기반 접근법과 다중 에이전트 기반 접근법이 동일한 실험 환경과 동일한 입력 데이터 조건에서 커밋 메시지 생성을 수행할 때 요구되는 추론 시간(inference time)을 함께 측정하였다. 측정 대상은 MCMD 데이터셋에서 추출한 총 8,000개 커밋 샘플 전체이며, 각 접근법에 대해 전체 데이터셋을 처리하는 데 소요된 총 시간과 샘플당 평균 추론 시간을 기록하였다. 측정 결과는 표 3에 정리되어 있다.

표 3. 단일 에이전트와 다중 에이전트 기반 접근법의 추론 시간 비교

Generation Method	Total Time	Time per Sample	Relative Cost
Single Agent	2h 15m	~1.01 sec	1.0x (Base)
Multi-Agent	9h 35m	~4.31 sec	4.26x

표 3에 따르면, 단일 에이전트 기반 접근법은 전체 8,000개 커밋 샘플을 처리하는 데 약 2시간 15분이 소요되었으며, 샘플당 평균 추론 시간은 약 1.01초로 측정되었다. 이는 단일 추론 과정에서 커밋 메시지를 생성하는 구조를 가지는 접근법의 시간적 특성이 반영된 결과로 볼 수 있다.

반면, 다중 에이전트 기반 접근법은 동일한 데이터셋을 처리하는 데 약 9시간 35분이 소요되었으며, 샘플당 평균 추론 시간은 약 4.31초로 측정되었다. 다중 에이전트 기반 접근법은 하나의 커밋 메시지를 생성하기 위해 초안 생성, 비평, 수정 단계가 순차적으로 수행되는 구조를 가지며, 이로 인해 단일 에이전트 기반 접근법에 비해 추론 시간이 크게 증가하는 경향이 관찰되었다. 전체 처리 시간 기준으로 볼 때, 다중 에이전트 기반 접근법은 단일 에이전트 기반 접근법 대비 약 4.26배의 시간 비용이 소요되는 것으로 나타났다. 이러한 결과는 두 접근법 간의 생성 구조 차이가 계산 자원 소요 측면에서 뚜렷한 차이로 반영되었음을 수치적으로 보여주는 결과로 정리할 수 있다. 본 장에서는 diff-only 환경에서 수행된 단일 에이전트 기반 접근법과 다중 에이전트 기반 접근법의 커밋 메시지 생성 실험 결과를 종합적으로 보고하였다. 정량 평가 결과에서는 단일 에이전트 기반 접근법이 BLEU, METEOR, ROUGE-L 자동 평가 지표 기준에서 다중 에이전트 기반 접근법보다 소폭 높은 평균 점수를 기록하는 경향이 관찰되었다. 재현성 분석에서는 단일 에이전트 기반 접근법이 동일 입력에 대해 매우 높은 출력 안정성을 보인 반면, 다중 에이전트 기반

접근법은 반복 실행 시 출력 표현의 변동성이 크게 나타나는 경향이 관찰되었다. 정성적 사례 분석에서는 다중 에이전트 기반 접근법이 코드 변경의 세부 요소를 나열하기보다는 변경의 핵심 개념이나 목적을 중심으로 메시지를 재구성하는 경향을 보였으며, 단일 에이전트 기반 접근법은 변경된 요소를 비교적 포괄적으로 기술하는 경향이 관찰되었다. 시간 비용 분석에서는 이러한 생성 구조의 차이가 추론 시간 증가라는 계산 비용 측면의 차이로 함께 나타났음을 확인하였다.

본 실험에서 자동 평가 지표(BLEU/METEOR/ROUGE-L), 출력 일관성(Self-BLEU), 추론 시간은 모두 Single-Agent가 우세했다. 따라서 본 연구가 관찰한 핵심은 '정량 성능의 상호 교환'이 아니라, 동일한 diff-only 조건에서도 생성 구조에 따라 정보 선택과 표현 전략, 그리고 실행 안정성이 달라진다는 점이다.

5. 관련연구

5.1 커밋 메시지 자동 생성 연구

커밋 메시지 자동 생성은 코드 변경 이력을 자연어로 요약하는 문제로 정의되며, 코드 리뷰, 변경 이력 추적, 유지보수 및 협업 과정에서 개발자의 이해를 지원하는 것을 주요 목표로 한다[2,3]. 이 문제는 주어진 코드 변경 정보와 이에 대응하는 커밋 메시지 간의 대응 관계를 학습하여, 새로운 코드 변경에 대해 적절한 자연어 설명을 생성하는 과제로 다루어져 왔다.

초기 연구들은 커밋 메시지 생성을 전통적인 자연어 처리 문제로 접근하였다. 이들 연구에서는 규칙 기반 방법이나 통계적 기계 번역 기법을 활용하여 코드 변경 내용을 요약하는 방식을 제안하였다. 규칙 기반 접근법은 사전에 정의된 패턴이나 휴리스틱을 활용하여 메시지를 생성하였으며, 통계적 접근법은 코드 변경과 커밋 메시지 간의 대응 관계를 확률적으로 모델링하는 데 초점을 두었다. 그러나 이러한 방법들은 규칙 설계에 대한 수작업 의존도가 높고, 다양한 형태의 코드 변경이나 복잡한 수정 패턴에 대해 일반화하기 어렵다는 한계를 지닌다[1,10]. 이후 신경망 기반 모델의 발전과 함께, 커밋 메시지 생성 문제는 신경 기계 번역(neural machine translation) 관점에서 본격적으로 연구되기 시작하였다[11,12]. 이 접근법에서는 코드 변경 정보를 입력 시퀀스로, 커밋 메시지를 출력 시퀀스로 간주하여 인코더–디코더 구조를 적용하였다. 이를 통해 코드 변경과 자연어 메시지 간의 순차적 대응 관계를 데이터로부터 학습할 수 있게 되었으며, 커밋 메시지 생성 과제를 번역 문제로 정식화하는 데 기여하였다. 이러한 설정은 이후 다수의 연구에서 기본적인 실험 프레임워크로 채택되었고, 커밋 메시지 자동 생성 연구의 표준적인 접근 방식으로 자리 잡았다[10,11]. 최근에는 대규모 언어 모델의 등장으로, 사전 학습된 모델을 활용한 커밋 메시지 생성 접근법이 활발히 제안되고 있다[14,15]. 이러한 모델들은 대규모 코드 데이터와 자연어 데이터를 기반으로 학습되어, 기존 신경 기계 번역 기반 모델에 비해 보다 유연한 표현과 일반화 능력을 보이는 것으로 보고되었다. 특히 코드 변경(diff) 텍스트를 직접 입력으로 사용하면서도 비교적 안정적인 출력을 생성할 수 있다는 점에서 주목받고 있으며, 다양한 프로그래밍 언어와 프로젝트에 대해 적용 가능성이 보고되고 있다[16].

한편, 커밋 메시지 생성 성능을 향상시키기 위해 입력 정보의 범위를 확장하려는 연구들도 다수 제안되었다[17]-[20]. 이러한 연구들은 텍스트 형태의 코드 변경 정보 외에도 추상 구문 트리(AST), 코드 토큰 시퀀스, 변경 전후 코드 스냅샷, 파일 경로 정보 등을 함께 활용함으로써 코드 변경의 구조적 또는 문맥적 정보를 보다 풍부하게 반영하고자 하였다. 일부 연구에서는 이러한 입력 확장이 자동 평가 지표 기준 성능 향상으로 이어졌음을 보고하였다. 그러나 입력 정보와 모델 구조가 동시에 확장되는 경우, 실험 조건 간 비교가 복잡해지고 생성 결과에 영향을 미치는 요인을 개별적으로 분리하여 해석하기 어렵다는 문제점도 함께 제기되었다[21].

특히 기존 연구의 다수는 새로운 입력 정보나 모델 구조를 제안하고, 해당 접근법의 성능을 자동 평가 지표를 통해 보고하는 데 초점을 두었다. 그 결과, 동일한 입력 조건 하에서 메시지 생성 과정의 구조적 차이가 커밋 메시지 생성 결과에 미치는 영향에 대해서는 상대적으로 제한적인 분석만이 이루어졌다. 예를 들어, 단일 에이전트 기반 생성 방식과 다중 에이전트 기반 생성 방식이 동일한 입력 정보와 동일한 데이터 분포 하에서 어떠한 차이를 보이는지에 대한 체계적인 비교는 충분히 다루어지지 않았다. 또한 많은 연구에서 입력 정보의 확장, 모델 용량 증가, 생성 구조 변경이 동시에 이루어졌기 때문에, 생성 결과의 차이가 어떤 요인에 의해 발생했는지를 명확히 분리하여 분석하기 어려운 경우가 많았다.

본 연구는 이러한 기존 연구의 한계를 인식하고, 입력 정보나 모델 용량을 확장하지 않은 상태에서 생성 구조의 차이에 초점을 맞춘 비교 분석을 수행한다는 점에서 기존 연구와 차별화된다. 구체적으로, 코드 변경(diff) 정보만을 입력으로 사용하는 diff-only 환경에서 단일 에이전트 기반 커밋 메시지 생성 방식과 다중 에이전트 기반 생성 방식을 동일한 실험 조건 하에서 비교한다. 이를 통해 입력 정보, 데이터 분포, 평가 절차를 모두 동일하게 유지한 상태에서, 생성 과정의 구조적 차이(single-agent vs. multi-agent)가 커밋 메시지 생성 결과에 어떠한 영향을 미치는지를 실험적으로 관찰하고자 한다. 이러한 관점은 기존 커밋 메시지 자동 생성 연구에서 상대적으로 충분히 분리되어 분석되지 않았던 생성 구조 효과를 보다 명확히 드러내기 위한 시도로 위치 지을 수 있다.

5.2 단일 에이전트 기반 커밋 메시지 생성

단일 에이전트 기반 커밋 메시지 생성은 하나의 언어 모델이 코드 변경 정보를 입력으로 받아, 해당 변경을 설명하는 커밋 메시지를 단일 추론 과정에서 직접 생성하는 방식을 의미한다. 이 구조에서는 코드 변경과 자연어 메시지 사이의 대응 관계가 하나의 생성 경로 내에서 처리되며, 생성 결과에 대해 별도의 검토, 수정, 또는 반복적 상호작용 단계는 포함되지 않는다. 그 결과, 입력과 출력 간의 관계가 비교적 단순한 형태로 구성되며, 시스템 구현과 실행 측면에서 구조적 복잡도가 낮게 유지된다.

이러한 특성으로 인해 단일 에이전트 기반 구조는 커밋 메시지 자동 생성 연구에서 가장 기본적인 기준선 접근법으로 널리 활용되어 왔다. 기존 다수의 연구들은 단일 에이전트 기반 모델을 기준으로 삼아, 입력 정보의 확장이나 모델 구조 변경에 따른 성능 변화를 비교하는 방식으로 실험을 설계하였다[2,3,10,11]. 특히 신경 기계 번역 기반 방법이 도입된 이후, 코드 변경을 입력 시퀀스로, 커밋 메시지를 출력 시퀀스로 처리하는 단일 에이전트 구조는 커밋 메시지 생성 연구에서 표준적인 실험 설정으로 자리 잡았다.

사전 학습된 대규모 언어 모델이 등장하면서, 단일 에이전트 기반 접근법의 표현 능력 또한 크게 확장되었다. 이러한 모델들은 대규모 코드 데이터와 자연어 데이터를 기반으로 학습되었으며, 코드 변경과 자연어 설명 사이의 일반적인 대응 관계를 효과적으로 내재화하고 있는 것으로 알려져 있다. 그 결과, 기존의 신경 기계 번역 기반 모델에 비해 보다 자연스러운 문장 구성과 유연한 표현을 생성할 수 있게 되었고, 다양한 프로그래밍 언어 및 프로젝트 환경에서도 비교적 안정적인 성능을 보이는 사례들이 보고되었다.

단일 에이전트 기반 접근법에서 반복적으로 관찰되는 특징 중 하나는 출력의 일관성이 높다는 점이다. 동일한 코드 변경 입력에 대해 여러 차례 메시지를 생성할 경우, 생성 결과가 유사한 어휘 선택과 문장 구조로 수렴하는 경향이 나타난다. 이러한 출력 안정성은 BLEU, METEOR, ROUGE-L과 같이 정답 메시지와의 표면적 중첩을 기반으로 점수를 산출하는 자동 평가 지표 환경에서 상대적으로 유리하게 작용할 수 있다. 실제로 기존 연구들에서는 단일 에이전트 기반 접근법이 이러한 자동 평가 지표 기준에서 비교적 높은 점수를 기록하는 사례가 다수 보고되었다[3,11]. 이는 단일 에이전트 구조가 특정 코드 변경에 대해 일반화된 표현을 선택함으로써, 정답 메시지와의 어휘적 유사도를 안정적으로 유지하는 경향과 관련된 결과로 해석될 수 있다.

그러나 단일 에이전트 기반 접근법의 이러한 특성은 동시에 한계로 작용할 여지도 존재한다. 생성 과정에 자기 검토나 재작성 단계가 포함되지 않기 때문에, 코드 변경의 세부적인 의미나 변경의 목적이 메시지에 충분히 반영되지 못하는 경우가 발생할 수 있다. 특히 diff-only 환경과 같이 코드 변경 정보만이 입력으로 제공되는 조건에서는, 변경의 배경이나 개발 의도가 명시적으로 드러나지 않는 경우가 많아, 생성된 메시지가 상대적으로

일반적인 표현에 머무르거나 변경의 핵심적인 차이를 명확히 구분하지 못하는 사례가 관찰되었다.

이러한 점은 단일 에이전트 기반 접근법이 자동 평가 지표 기준에서는 안정적인 성능을 보일 수 있으나, 생성된 메시지가 실제 코드 변경을 얼마나 충실하게 설명하는지에 대해서는 추가적인 분석이 필요함을 보인다. 이러한 문제의식은 이후 커밋 메시지 생성 연구에서 검토 또는 수정 단계를 포함하는 생성 구조, 즉 다중 에이전트 기반 접근법이 제안되는 배경 중 하나로 작용하였으며, 생성 과정의 구조적 변화를 통해 메시지 표현 특성을 보완하고자 하는 연구 흐름으로 이어졌다.

5.3 다중 에이전트 기반 생성 접근법

최근 자연어 처리 및 코드 생성 분야에서는 하나의 모델이 단일 추론 경로를 통해 결과를 산출하는 방식만으로는 복잡한 생성 과제를 충분히 다루기 어렵다는 인식이 확산되면서, 여러 에이전트의 단계적 상호작용을 통해 출력을 구성하는 다중 에이전트 기반 접근법이 점차 주목받고 있다[22,23]. 이 접근법에서는 단일 모델이 모든 판단을 수행하는 구조에서 벗어나, 서로 다른 역할을 부여받은 에이전트들이 생성 과정에 순차적으로 참여함으로써 최종 결과를 형성한다.

다중 에이전트 기반 생성 방식에서는 생성 절차가 명시적인 단계로 분리되는 경우가 일반적이다. 대표적으로 초안 작성 단계에서 초기 출력을 생성한 뒤, 이후 단계에서 다른 에이전트가 해당 출력을 점검하거나 재구성하는 흐름이 사용된다. 각 에이전트는 동일한 입력 정보를 공유하거나, 이전 단계에서 생성된 중간 산출물을 참조하여 역할을 수행한다. 이러한 절차적 분리는 단일 모델 내부에서 암묵적으로 이루어지던 판단 과정을 외부의 단계적 상호작용으로 드러내며, 생성 과정에 명시적인 검토 및 수정 메커니즘을 포함할 수 있게 한다[22,23].

기존 연구들에서는 이러한 다중 에이전트 기반 구조가 문서 요약, 질의 응답, 코드 생성 등 다양한 자연어 생성 과제에서 출력 결과의 정확성이나 표현의 다양성을 향상시킬 수 있음을 보고하였다. 특히 단일 단계 생성 과정에서 발생할 수 있는 오류나 부정확한 표현이, 후속 에이전트의 검토 과정을 통해 완화될 수 있다는 점이 주요 장점으로 언급되어 왔다. 이와 같은 결과들은 다중 에이전트 구조가 생성 결과를 한 번에 고정하기보다는, 단계적으로 조정·개선할 수 있는 가능성을 가진다는 점을 실험적으로 보여준다.

커밋 메시지 생성 과제에서도 다중 에이전트 기반 접근법을 적용하려는 시도가 일부 이루어졌다[24,25]. 해당 연구들은 주로 생성된 커밋 메시지의 품질 향상 가능성에 초점을 두고, 단일 에이전트 기반 접근법과의 비교를 통해 자동 평가 지표 기준 성능이 향상되었음을 보고하였다. 다만 이러한 비교는 비교적 제한된 데이터셋이나 특정 실험 조건에 기반한 경우가 많았으며, 코드 변경 정보 외에 추가적인 문맥 정보나 확장된 입력을 함께 사용하는 설정이 포함되는 경우도 적지 않았다.

이러한 이유로 기존 연구 결과만을 바탕으로 다중 에이전트 기반 접근법의 성능 향상이 생성 구조 자체에서 비롯된 효과인지, 혹은 입력 정보 확장이나 모델 설정 차이에 기인한 것인지를 명확히 구분하기에는 한계가 존재한다. 또한 단일 에이전트와 다중 에이전트 기반 접근법을 동일한 입력 조건과 동일한 평가 기준 하에서 체계적으로 비교한 실험적 분석은 상대적으로 제한적으로 이루어져 왔다. 특히 코드 변경 정보만을 입력으로 사용하는 diff-only 환경에서는, 다중 에이전트 기반 생성 구조가 어떠한 출력 특성을 보이는지에 대한 분석이 충분히 축적되지 않았다.

본 연구는 이러한 기존 연구의 제약점을 인식하고, 입력 정보의 범위나 모델 용량을 변경하지 않은 상태에서 생성 구조의 차이에만 초점을 맞춘 비교 분석을 수행한다. 구체적으로, diff-only 환경이라는 제한된 입력 조건을 유지한 채 단일 에이전트 기반 커밋 메시지 생성 방식과 다중 에이전트 기반 방식을 동일한 실험 설정으로 비교함으로써, 생성 절차의 구조적 차이가 커밋 메시지 생성 결과에 어떠한 영향을 미치는지를 보다 명확히 관찰하고자 한다. 이를 통해 기존 연구에서 충분히 분리되어 분석되지 않았던 생성 구조 효과를 실험적으로 검토하는 것을 본 연구의 목표로 한다.

5.4 커밋 메시지 생성의 평가와 재현성

커밋 메시지 자동 생성 연구에서 생성 결과의 품질을 정량적으로 비교하기 위한 수단으로는 자동 평가 지표가 가장 널리 활용되어 왔다. BLEU, METEOR, ROUGE-L과 같은 지표는 본래 기계 번역과 문서 요약 과제에서 사용되어 온 평가 방식으로, 커밋 메시지 생성 연구에서도 표준적인 성능 측정 도구로 자리 잡았다[26,27]. 이들 지표는 생성된 메시지와 정답 메시지 간의 어휘 중첩이나 문장 구조 유사도를 기반으로 점수를 산출하며, 대규모 데이터셋을 대상으로 여러 접근법의 성능 경향을 효율적으로 비교할 수 있다는 장점을 가진다.

다만 자동 평가 지표가 제공하는 점수는 생성된 메시지의 표면적 유사도에 초점을 두고 있으며, 메시지가 코드 변경의 의미를 얼마나 적절하게 전달하는지까지 포괄적으로 설명하지는 못한다. 커밋 메시지 생성 과제에서는 동일한 코드 변경에 대해 여러 자연어 표현이 의미적으로 허용될 수 있고, 실제 데이터에서 정답(Gold Label) 메시지 자체가 항상 변경의 핵심을 정확히 반영한다고 가정하기도 어렵다. 이러한 특성으로 인해 자동 평가 지표 점수와 생성 메시지의 실제 활용 가능성 사이에 차이가 발생할 수 있다는 점이 기존 연구에서 반복적으로 지적되어 왔다.

이러한 한계를 인식한 일부 연구들은 자동 평가 결과를 보완하기 위한 방법으로 사람 평가를 도입하거나, 생성된 메시지에 대한 정성적 사례 분석을 함께 제시하였다. 사람 평가는 생성 메시지가 코드 변경을 얼마나 명확하게 설명하는지를 직접 판단할 수 있다는 점에서 의미 있는 접근으로 평가된다. 그러나 대규모 데이터셋을 대상으로 일관된 기준의 사람 평가를 수행하는 데에는 상당한 시간과 비용이 요구되며, 평가자의 주관적 판단이 결과에 영향을 미칠 수 있다는 점도 동시에 문제로 제기된다. 이러한 현실적 제약으로 인해, 다수의 커밋 메시지 생성 연구에서는 자동 평가 지표를 주요 평가 수단으로 유지하면서 정성적 분석을 보조적으로 활용하는 방식을 채택해 왔다.

한편 자연어 생성 연구 전반에서는 출력 품질 평가와 더불어, 동일 입력에 대해 생성 결과가 얼마나 안정적으로 유지되는지, 또는 얼마나 다양한 표현이 생성되는지 역시 중요한 분석 대상으로 다루어지고 있다. 재현성은 동일한 입력 조건에서 생성 결과가 반복 실행 시 어느 정도 일관되게 유지되는지를 나타내는 개념이며, 출력 다양성은 동일 입력에 대해 표현이 얼마나 폭넓게 변화하는지를 의미한다. 이 두 특성은 생성 모델이 출력 공간을 어떤 방식으로 탐색하는지를 이해하는 데 중요한 정보를 제공한다.

커밋 메시지 생성 과제의 특성을 고려할 때, 출력 다양성은 반드시 부정적인 속성으로만 해석되지는 않는다. 하나의 코드 변경에 대해 여러 요약 표현이 의미적으로 허용될 수 있기 때문에, 다양한 표현을 생성하는 능력은 모델의 표현력이나 요약 전략의 유연성을 반영할 수 있다. 반면, 동일한 입력에 대해 생성 결과가 과도하게 달라지는 경우, 자동화된 개발 도구나 반복 실행이 요구되는 환경에서는 출력의 안정성이 실용적인 제약 요소로 작용할 수 있다. 따라서 재현성과 출력 다양성은 서로 배타적인 개념이라기보다, 사용 목적과 적용 맥락에 따라 균형 있게 해석되어야 할 특성으로 이해할 수 있다.

이러한 관점에서 최근 연구들에서는 동일한 입력에 대해 반복 생성된 출력들 간의 유사도를 정량적으로 측정하기 위한 지표로 Self-BLEU가 활용되고 있다[8,28]. Self-BLEU는 동일 입력으로부터 생성된 여러 출력 간의 n-gram 중첩도를 계산함으로써, 생성 결과의 일관성과 변동성을 수치적으로 표현한다. 본 연구에서는 Self-BLEU를 사용하여 단일 에이전트 기반 접근법과 다중 에이전트 기반 접근법의 재현성 특성을 비교 분석한다. 이를 통해 자동 평가 지표만으로는 충분히 드러나지 않는 출력 안정성과 표현 다양성 측면의 차이를 실험적으로 관찰하고자 한다.

6. 토의

6.1 실험 결과 분석

정량 평가 결과(표 1)에서 단일 에이전트 기반 접근법은 BLEU, METEOR, ROUGE-L 전 지표에서 다중 에이전트 기반 접근법보다 일관되게 높은 점수를 기록하였다. 이 차이는 특정 지표에 국한되지 않고 모든 자동 평가 지표에서 동일한 방향으로 나타났으며, 이는 우연적 변동보다는 두 접근법의 출력 특성이 지표 계산 방식에 체계적으로 다르게 반영되었음을 보인다. 특히

BLEU와 ROUGE-L은 n-gram 및 서열 기반 중첩 비율에 크게 의존하고, METEOR 역시 어휘 정합성을 주요 요소로 포함한다는 점에서, 해당 결과는 두 접근법 간 어휘 선택 분포 및 문장 구조 안정성의 차이가 점수 차이로 누적된 결과로 볼 수 있다.

단일 에이전트 기반 접근법은 단일 추론 단계에서 메시지를 생성하며, 동일하거나 유사한 입력 diff에 대해 반복 실행 시 출력 문장이 매우 높은 수준으로 수렴하는 특성을 보였다. 이 특성은 재현성 분석(표 2)에서 Self-BLEU 98.50이라는 값으로 정량적으로 확인되었으며, 이는 생성 결과가 거의 동일한 어휘 집합과 문장 구조를 반복적으로 사용하는 경향을 가진다는 것을 의미한다. 이러한 출력 수렴성은 자동 평가 지표 관점에서 볼 때, 정답(Gold Label) 메시지와의 어휘 중첩을 안정적으로 유지하는 데 기여했을 가능성이 크다. 즉, 단일 에이전트 기반 접근법은 표현 다양성보다는 표현 일관성(consistency)이 우세한 분포를 형성하고 있으며, 이 분포 특성이 자동 평가 지표 점수의 안정적인 우위를 설명하는 핵심 요인으로 작용한다.

반면 다중 에이전트 기반 접근법은 Self-BLEU 1.28이라는 극히 낮은 값을 기록하였으며, 이는 동일한 입력에 대해 반복 실행 시 출력 문장이 거의 겹치지 않을 정도로 다양한 표현이 생성되고 있음을 의미한다. Self-BLEU가 0에 가까울수록 출력 간 어휘 및 구조 중첩이 낮다는 점을 고려하면, 두 접근법 간 약 97.22포인트의 차이는 단순한 출력 변동성 수준을 넘어, 생성 과정 자체가 서로 다른 확률 분포를 따르고 있음을 보인다. 다중 에이전트 기반 접근법에서는 초안 생성 이후 비평과 수정 단계가 반복적으로 적용되면서, 각 단계에서 강조되는 코드 변경 요소나 설명 방식이 달라지고, 그 결과 최종 출력이 매 실행마다 상이한 조합으로 재구성되는 양상이 나타난다. 이러한 구조적 특성은 표현 다양성을 증가시키는 동시에, 정답 메시지와의 n-gram 중첩 비율을 불안정하게 만드는 요인으로 작용한다. 정성적 사례 분석 결과는 이러한 분포 차이를 보다 구체적으로 뒷받침한다. 다중 에이전트 기반 접근법의 출력에서는 특정 라이브러리 옵션 변경, 내부 자료구조 교체, 다국어 리소스 파일 수정 등 diff에 포함된 세부 변경 사항이 문장 단위로 명시되는 빈도가 높게 관찰되었다. 또한 변경의 원인이나 기대 효과를 설명하는 부가적인 절(clause)이 포함되는 경우도 반복적으로 나타났다. 이는 출력 메시지의 정보 밀도와 서술 범위가 확장되는 방향으로 분포가 이동하고 있음을 의미한다. 반면 단일 에이전트 기반 접근법은 핵심 변경 사항을 하나의 일반화된 표현으로 요약하는 사례가 상대적으로 많았으며, 메시지 길이와 어휘 다양성 분포가 좁은 범위에 집중되는 양상을 보였다. 흥미로운 점은 정답(Gold Label) 메시지가 실제 코드 변경 내용과 불일치하는 일부 샘플에서도 두 접근법의 반응이 다르게 나타났다는 것이다. 단일 에이전트 기반 접근법은 이러한 경우에도 정답 메시지에서 자주 등장하는 표현 패턴과 유사한 문장을 생성하는 경향을 유지한 반면, 다중 에이전트 기반 접근법은 diff에 포함된 변경 요소를 중심으로 메시지를 구성하는 비율이 상대적으로 높게 관찰되었다. 이는 다중 에이전트 기반 접근법이 정답 텍스트 분포보다는 입력 diff 정보에 대한 반응성을 더 크게 반영하는 생성 특성을 가진다는 점을 보인다.

계산 비용 분석 결과(표 3)는 생성 구조의 차이가 시간 비용에 직접적으로 반영됨을 보여준다. 단일 에이전트 기반 접근법은 전체 8,000개 커밋 처리에 약 2시간 15분이 소요되었고, 샘플당 평균 추론 시간은 약 1.01초로 측정되었다. 반면 다중 에이전트 기반 접근법은 동일한 데이터셋 처리에 약 9시간 35분이 소요되었으며, 샘플당 평균 추론 시간은 약 4.31초로 나타났다. 이는 다중 에이전트 기반 접근법이 단일 에이전트 대비 약 4.26배의 시간 비용을 요구함을 의미하며, 초안 생성 이후 비평 및 수정 단계가 추가적인 추론 호출과 토큰 소비를 유발한 결과로 해석할 수 있다.

Multi-Agent가 자동 평가 지표에서 소폭 낮게 측정된 이유는, (평가 지표가 본질적으로 Gold label과의 표면 중첩(n-gram/LCS)에 민감하고, 실제 데이터의 Gold label이 짧고 관습적인 표현(예: “fix”, “update”)으로 수렴하는 경우가 있어, 표현을 재구성하거나 목적/효과를 서술하는 생성물이 오히려 불리하게 채점될 수 있기 때문이다. 본 연구의 정성 사례에서 Multi-Agent는 파일명/구현 세부를 그대로 복제하기보다 의미 단위로 재진술하는 경향이 관찰되었고, 이 과정에서 동의어 치환이나 문장 구조 변화가 발생해 중첩 기반 지표 점수가 하락하는 방향으로 작용할 수 있다. 따라서 본 결과는 “Multi-Agent가 변경을 덜 반영한다”기보다, “정답 표현과의 유사도 관점에서는 불리한 표현 전략을 선택했다”로 해석하는 것이 타당하다.

6.2 위협요소

본 연구의 실험 결과는 해석 시 몇 가지 제한 조건을 함께 고려할 필요가 있다. 본 연구는 diff-only 환경에서 단일 에이전트 기반 접근법과 다중 에이전트 기반 접근법을 비교하였다. 이 설정은 코드 변경 정보만을 입력으로 사용하는 조건에서 생성 구조의 차이가 출력 특성에 미치는 영향을 비교적 명확하게 관찰할 수 있다는 장점을 가지지만, 실제 개발 환경에서 활용될 수 있는 저장소 수준의 문맥 정보, 파일 간 의존 관계, 이전 커밋 히스토리, 이슈 트래킹 정보와 같은 추가적인 맥락을 포함하지 못한다는 한계를 가진다. 따라서 본 연구에서 관찰된 결과는 코드 변경(diff) 자체에 기반한 커밋 메시지 생성 특성으로 해석하는 것이 적절하며, 보다 풍부한 문맥 정보가 제공되는 환경에서는 생성 양상이나 접근법 간 상대적 특성이 다르게 나타날 가능성을 배제할 수 없다. 평가 방법 측면에서도 제한이 존재한다. 본 연구는 8,000개 커밋을 대상으로 BLEU, METEOR, ROUGE-L과 같은 자동 평가 지표를 사용하여 대규모 정량 비교를 수행하였다. 이러한 지표는 접근법 간 전반적인 경향을 효율적으로 비교하는 데에는 유용하지만, 생성된 커밋 메시지가 실제 코드 변경의 의도와 맥락을 얼마나 정확하게 전달하는지, 혹은 개발자가 메시지를 읽었을 때 얼마나 유용하게 인식하는지와 같은 질적 요소를 직접적으로 반영하지는 못한다. 본 연구에서는 이러한 한계를 인식하고 정성적 사례 분석을 병행하였으나, 사람 평가를 포함한 체계적인 사용자 기반 품질 평가는 포함하지 못하였다. 따라서 자동 평가 지표 결과가 실제 개발자 관점의 유용성과 완전히 일치한다고 일반화하기에는 추가적인 검증이 필요하다. 또한 정답(Gold Label) 메시지 자체의 품질 변동 역시 평가 결과에 영향을 줄 수 있는 요인이다. 일부 사례에서는 정답 메시지가 실제 코드 변경 내용과 완전히 일치하지 않거나, 변경의 목적을 충분히 설명하지 못하는 경우가 관찰되었다. 이러한 상황에서는 표면적 유사도에 기반한 자동 평가 지표가 정답 메시지의 오류나 불완전성을 그대로 반영하는 방향으로 작동할 수 있으며, diff 정보를 충실히 반영한 생성 결과가 오히려 낮은 점수로 평가되는 경우도 발생할 수 있다. 본 연구가 정량 지표 결과를 단독으로 해석하지 않고 정성적 사례 분석과 함께 논의한 이유는 이러한 데이터 특성이 결과 해석에 영향을 미칠 수 있음을 고려했기 때문이다. 재현성 분석에서 관찰된 다중 에이전트 기반 접근법의 출력 변동성 또한 적용 관점에서 제한 요소로 고려될 수 있다. 동일한 입력에 대해 실행마다 서로 다른 표현이 생성되는 특성은 표현 다양성이나 탐색 범위 측면에서는 의미 있는 특성으로 해석될 수 있으나, 반복 실행 시 동일한 출력을 요구하는 자동화된 개발 도구나 파이프라인 환경에서는 제약으로 작용할 가능성이 있다. 본 연구에서는 이러한 출력 변동성을 관측된 결과로 보고하였으나, 변동성을 제어하거나 안정화하기 위한 메커니즘에 대한 분석이나 실험은 포함하지 않았다. 계산 비용 역시 실제 적용 시 고려해야 할 요소이다. 실험 결과에서 확인된 바와 같이 다중 에이전트 기반 접근법은 단일 에이전트 기반 접근법 대비 추론 시간이 크게 증가하였다. 이는 생성 품질 측면에서 일부 긍정적인 특성이 관찰되는 경우에도, 시스템 자원이나 응답 시간이 제한된 환경에서는 활용이 제한될 수 있음을 의미한다. 본 연구에서는 시간 비용을 측정하여 보고하는 데에 초점을 두었으며, 다중 에이전트 구조의 단계 축소, 호출 횟수 감소, 또는 효율화 전략에 대한 분석은 범위에 포함하지 않았다. 마지막으로 데이터셋 구성 역시 결과 해석에 영향을 줄 수 있다. 본 연구는 MCMD 데이터셋에서 추출한 8,000개의 커밋을 사용하였으며, 언어별 샘플링을 통해 특정 언어 편향을 완화하고자 하였다. 그럼에도 불구하고 특정 프로젝트나 도메인에 특화된 커밋 메시지 작성 관습, 조직별 스타일 가이드, 혹은 개발 문화의 차이는 충분히 반영되지 않았을 가능성이 있다. 따라서 본 연구의 결과는 개별 프로젝트의 특수한 상황보다는 다양한 오픈소스 프로젝트 전반에서 관찰될 수 있는 일반적인 경향으로 해석하는 것이 적절하다. 향후 연구에서는 프로젝트 단위 분석, 도메인별 비교,

그리고 추가적인 문맥 정보를 포함한 확장 실험을 통해 결과의 일반화 가능성을 보다 정밀하게 검증할 필요가 있다.

7. 결론

본 연구에서는 코드 변경 정보만을 입력으로 사용하는 diff-only 환경에서 단일 에이전트와 다중 에이전트 기반 커밋 메시지 생성 접근법을 비교 분석하였다. 동일한 데이터셋과 동일한 입력 조건을 유지한 상태에서 생성 구조의 차이가 자동 평가 지표, 출력의 재현성, 그리고 생성된 메시지의 표현 특성에 미치는 영향을 실험적으로 관찰하였다. 정량 평가 결과, 단일 에이전트 기반 접근법은 BLEU, METEOR, ROUGE-L 지표에서 다중 에이전트 기반 접근법보다 소폭 높은 점수를 기록하였다. 이는 자동 평가 지표가 생성된 메시지와 정답 메시지 간의 표면적 유사도를 기준으로 점수를 산출한다는 평가 특성과 관련된 결과로 해석될 수 있다. 반면, 다중 에이전트 기반 접근법은 생성 과정에서 초안 생성 이후 비평과 수정 단계를 거치면서 메시지 표현이 변화하는 경향을 보였으며, 그 결과 정답 메시지와의 표현적 차이가 증가하는 양상이 관찰되었다. 재현성 분석에서는 두 접근법 간의 차이가 보다 명확하게 나타났다. 단일 에이전트 기반 접근법은 동일한 코드 변경 입력에 대해 반복 실행 시 매우 유사한 출력을 생성하는 경향을 보였으며, 이는 높은 Self-BLEU 값으로 확인되었다. 반면, 다중 에이전트 기반 접근법은 동일 입력에 대해서도 서로 다른 표현의 메시지를 생성하는 경우가 빈번하게 관찰되었으며, 출력 간 유사도는 상대적으로 낮게 나타났다. 이러한 결과는 생성 구조에 따라 출력의 안정성과 표현 다양성이 서로 다른 양상으로 나타날 수 있음을 보여준다. 정성적 사례 분석을 통해서는 자동 평가 지표만으로는 포착하기 어려운 생성 특성의 차이가 확인되었다. 다중 에이전트 기반 접근법은 코드 변경의 세부적인 내용이나 변경 목적을 메시지에 보다 명시적으로 반영하는 사례가 다수 관찰되었다. 특히 일부 사례에서는 정답 메시지가 실제 코드 변경 내용을 충분히 반영하지 못하는 경우에도, 다중 에이전트 기반 접근법이 코드 변경 정보에 기반한 메시지를 생성하는 양상이 확인되었다. 또한 시간 비용 분석 결과, 다중 에이전트 기반 접근법은 단일 에이전트 기반 접근법에 비해 더 많은 추론 시간이 소요되는 것으로 관찰되었다. 이는 초안 생성, 비평, 수정 단계로 구성된 생성 구조의 특성에 기인한 결과로 볼 수 있다. 이상의 결과를 종합하면, 단일 에이전트와 다중 에이전트 기반 커밋 메시지 생성 접근법은 diff-only 환경에서 자동 평가 지표 성능, 출력의 재현성, 메시지 표현 특성, 그리고 계산 비용 측면에서 서로 다른 특성을 보이는 것으로 관찰된다. 단일 에이전트 기반 접근법은 자동 평가 지표 기준에서 안정적인 성능과 높은 출력 일관성을 보인 반면, 다중 에이전트 기반 접근법은 출력의 다양성과 코드 변경 반영 측면에서 상이한 특성을 나타냈다. 본 연구는 diff-only 환경이라는 제한된 입력 조건 하에서 생성 구조의 차이에 초점을 맞추어, 단일 에이전트와 다중 에이전트 기반 접근법의 특성을 실험적으로 비교하였다. 이러한 분석은 기존 커밋 메시지 생성 연구에서 널리 사용되어 온 자동 평가 지표 중심의 평가 관행을 유지하면서도, 생성 구조에 따른 출력 특성의 차이를 보다 명확히 이해하는 데 기여한다. 향후 연구에서는 저장소 수준의 문맥 정보를 포함한 입력 확장, 출력 안정성을 고려한 생성 제어 기법, 그리고 사람 평가를 포함한 보완적 분석을 통해 본 연구에서 관찰된 결과를 확장할 수 있을 것이다.

참고문헌

[1] A. E. Hassan and R. C. Holt, “Studying the chaos of code development,” in Proc. WCRE, 2003.
[2] M. Tian et al., “What Makes a Good Commit Message?,” in Proc. ICSE, 2022.
[3] Y. Li et al., “Commit Message Matters: Investigating and Predicting Commit Message Quality,” in Proc. ICSE, 2023.
[4] W. Tao et al., “MCMD: Multi-language Commit Message Dataset,” dataset release, 2021.
[5] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: a method for automatic evaluation of machine translation,” in Proc. ACL, 2002.
[6] S. Banerjee and A. Lavie, “METEOR: An automatic metric for MT evaluation with improved correlation with human judgments,” in Proc. ACL Workshop, 2005.
[7] C.-Y. Lin, “ROUGE: A package for automatic evaluation of summaries,” in Proc. ACL Workshop, 2004.
[8] Z. Zhu et al., “Measuring diversity in text generation using self-BLEU,” arXiv preprint, 2018.
[9] kcse-sub01, dataset8000, GitHub repository, 2025. Available: https://github.com/kcse-sub01/dataset8000
[10] S. Jiang, A. Armaly, and C. McMillan, “Automatically generating commit messages from diffs using neural machine translation,” in Proc. ASE, 2017.
[11] Z. Liu et al., “Neural-machine-translation-based commit message generation: How far are we?,” in Proc. ASE, 2018.
[12] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in Proc. NeurIPS, 2014.
[13] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” in Proc. ICLR, 2015.
[14] A. Vaswani et al., “Attention Is All You Need,” in Proc. NeurIPS, 2017.
[15] T. Chen et al., “Automatic commit message generation using large language models,” J. Syst. Softw., 2023.
[16] Y. Xue et al., “Automated Commit Message Generation with Large Language Models: An Empirical Study,” 2024.
[17] L. Shi, Y. Liang, H. Jiang, and L. Yu, “RACE: Retrieval-augmented commit message generation,” in Proc. EMNLP, 2022.
[18] S. Liu et al., “ATOM: Commit message generation based on abstract syntax tree and hybrid ranking,” IEEE Trans. Softw. Eng., 2020.
[19] Y. Huang et al., “Learning Human-Written Commit Messages to Document Code Changes,” J. Comput. Sci. Technol., 2020.
[20] Y. Wang et al., “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation,” in Proc. EMNLP, 2021.
[21] W. Tao et al., “KADEL: Knowledge-Aware Denoising Learning for Commit Message Generation,” arXiv preprint arXiv:2401.08376, 2024.
[22] S. Park et al., “Multi-agent collaboration for text generation with large language models,” in Proc. AAAI, 2023.
[23] J. Wei et al., “Chain-of-thought prompting elicits reasoning in large language models,” in Proc. NeurIPS, 2022.
[24] H. Jung, “Commit message generation using pre-trained neural machine translation model,” in NLP4Prog (Workshop at ACL/IJCNLP), 2021.
[25] Y. Wu et al., “Commit Message Generation via ChatGPT: How Far Are We?,” in Proc. ICSE-Companion, 2024.
[26] W. Tao et al., “On the Evaluation of Commit Message Generation Models: An Experimental Study,” in Proc. ICSME, 2021.
[27] W. Tao et al., “A Large-Scale Empirical Study of Commit Message Generation: Models, Datasets, and Evaluation,” Empirical Softw. Eng., 2022.
[28] A. Madaan et al., “Self-Refine: Iterative Refinement with Self-Feedback,” arXiv preprint arXiv:2303.17651, 2023.

실행 컨텍스트 정합성에 기반한 LLM 결함 위치 추정 결과의 안정화 기법*

남규민 O1, 최서진 1, 양근석 2†

1 한경국립대학교 컴퓨터응용수학부, 2 한경국립대학교 컴퓨터응용수학부(컴퓨터시스템연구소)

O1 kmnam01@hknu.ac.kr, 1 insui12@hknu.ac.kr, 2 gsyang@hknu.ac.kr

Stabilizing LLM-Based Bug Localization Results via Execution-Context Consistency

Gyumin Nam O1, Seojin Choe 1, Geunseok Yang 2†

1 Department of Computer Applied Mathematics, Hankyong National University,
2 Department of Computer Applied Mathematics(Computer System Institute), Hankyong National University

요 약

최근 LLM 기반 결함 위치 추정 기법은 코드와 자연어 간 의미적 관계를 활용하여 기존 통계적·정보검색 기반 기법 대비 경쟁력 있는 성능을 보이고 있다. 그러나 동일한 입력과 추론 설정에서도 반복 실행 시 후보 메서드의 출력 순위가 변동되는 문제가 관찰되며, 이는 결과 일관성과 실험 재현성을 저해한다. 본 논문은 이러한 출력 변동성을 완화하기 위해, LLM의 입력 구성이나 추론 과정을 변경하지 않고, 실패 테스트 실행 과정에서 관측되는 크래시 스택 트레이스로부터 추출한 실행 컨텍스트를 사후적으로 활용하는 정합성 보정 기법을 제안한다. 제안 기법은 LLM이 생성한 초기 Top-k 후보 순위를 유지한 상태에서, 각 후보 메서드와 실제 실행 경로 간의 구조적 정합성을 경량 실행 신호로 평가하고, 최소 변경 원칙과 게이팅 규칙에 따라 Top-1 및 Top-5 경계에서만 제한적인 순위 조정을 수행한다. Defects4J 벤치마크의 835개 결함 인스턴스를 대상으로 두 가지 오픈소스 LLM 환경에서 수행한 실험 결과, 제안 기법은 Top-1 정확도에서 0.5-1.1%p, Top-5 정확도에서 0.8-1.3%p의 개선을 보였으며, MAP는 0.009-0.011, MRR은 0.007-0.013 증가하였다. 또한 반복 실행 간 Top-1 불일치율 감소와 Top-5 후보 집합 유사도 증가를 통해 출력 일관성 측면에서도 개선 효과가 확인되었다. 이러한 결과는 제안 기법이 기존 LLM 기반 결함 위치 추정 파이프라인의 구조적 변경 없이도 실행 경로 정합성을 보조 기준으로 활용하여 순위 품질과 출력 안정성을 함께 개선할 수 있음을 보여준다.

1. 서론

오픈소스 소프트웨어(Open-Source Software, OSS) 프로젝트는 지속적인 기능 확장과 협업 개발로 인해 코드베이스의 규모와 구조적 복잡성이 증가하고 있다. 이러한 환경에서 소프트웨어 결함이 발생했을 때 결함이 포함된 코드 위치를 신속하고 정확하게 식별하는 결함 위치 추정(Fault Localization, FL)은 디버깅 효율성과 유지보수 비용에 직접적인 영향을 미치는 핵심 단계이다. 기존 연구에 따르면 디버깅 과정에서 상당한 시간이 결함 위치 탐색에 소요되며, 부정확한 결함 위치 추정은 불필요한 코드 탐색과 반복적인 검증을 유발할 수 있다 [1]-[3].

이를 해결하기 위해 스펙트럼 기반 결함 위치 추정(SBFL), 정보 검색 기반 결함 위치 추정(IRFL), 그리고 하이브리드 기법 등 다양한 자동화 기법이 제안되어 왔다 [4], [5]. 이러한 기법들은 테스트 실행 결과, 코드 커버리지, 또는 텍스트 유사도와 같은 정적·동적 신호를 활용하여 일정 수준의 성능을 보여왔다. 그러나 실패 테스트가 충분하지 않거나 테스트 인프라가 제한된 환경에서는 성능 저하가 발생하며, 대규모 코드베이스를 대상으로 적용할 경우 탐색 공간이 과도하게 확장되는 한계가 보고되어 왔다 [6].

최근에는 대규모 언어 모델(Large Language Models, LLMs)의 코드 이해 및 의미적 추론 능력을 활용한 결함 위치 추정 기법이 주목받고 있다 [7]. LLM 기반 접근은 버그 리포트,

* 이 논문은 한경국립대학교 국립대학육성사업(2025)지원을 받아 작성되었음

† 교신저자(Corresponding Author)

테스트 정보, 소스 코드 간의 의미적 연관성을 직접 추론할 수 있으며, 사전 학습된 모델을 활용함으로써 별도의 도메인 특화 학습 없이도 적용 가능하다는 장점을 가진다 [8]. 이에 따라 함수 단위 결함 위치 추정 문제에서 기존 통계적 기법을 보완하는 실용적인 대안으로 평가되고 있다 [9].

그러나 기존 연구들에 따르면 LLM 기반 결함 위치 추정 기법은 동일한 입력과 추론 설정을 사용하더라도 반복 실행 시 출력 순위가 변동하는 현상이 관찰된다. 이로 인해 Top-K 후보 순위의 안정성과 결과 일관성이 충분히 확보되지 않으며, 동일한 결함 인스턴스에 대해 Top-1 선택이 달라지거나 상위 후보 간 순서가 재배열되는 문제가 발생할 수 있다 [10], [11]. 이러한 출력 불안정성은 실험 재현성을 저해할 뿐만 아니라, 실제 디버깅 과정에서 결과를 신뢰하고 반복적으로 활용하는 데 장애 요소로 작용한다.

기존 연구들은 입력 문맥 보강, 후보 축소, 다중 추론 결과 집계, 또는 다단계 파이프라인 및 에이전트 기반 추론 구조를 통해 이러한 문제를 완화하고자 하였다 [12], [13]. 그러나 이러한 접근들은 주로 LLM의 입력 구성이나 추론 과정에 직접 개입하는 방식에 집중되어 있으며, 이미 생성된 LLM 출력 결과를 대상으로 한 사후적(post-hoc) 안정화 기법에 대한 논의는 상대적으로 제한적이다. 또한 재학습이나 입력 재구성을 요구하는 기법들은 기존 디버깅 파이프라인에 통합하는 데 추가적인 비용과 복잡성을 수반한다.

본 연구는 이러한 한계를 보완하기 위해 실행 컨텍스트 기반 정합성 보정을 통한 LLM 결함 위치 추정 안정화 기법을 제안한다. 본 논문에서 실행 컨텍스트(execution context)는 실패 테스트 실행 과정에서 관측되는 호출 스택 프레임과 패키지 경로 등 프로그램의 실제 실행 흐름을 반영하는 관측 가능한 정보를 의미한다. 실행 컨텍스트는 결함 위치의 정답을 직접 판별하는 오라클이 아니라, LLM이 생성한 후보 순위가 실제 실행 경로와 구조적으로 얼마나 일관되는지를 평가하기 위한 정합성 기준으로 활용된다.

제안 기법은 LLM의 입력 구성이나 추론 과정에 개입하지 않고, LLM이 생성한 최종 Top-K 후보 순위에 대해 사후적으로 적용된다. 실행 컨텍스트와 후보 함수 간의 정합성을 경량 실행 신호로 평가한 뒤, 최소 변경(minimal-change) 원칙에 따라 Top-K 범위 내에서만 제한적인 순위 보정을 수행한다. 또한 Top-1 및 Top-5 경계에서만 선택적으로 개입하는 게이트 기반 전략을 적용하여, 불필요한 순위 재정렬과 성능 회귀를 억제하도록 설계하였다.

Defects4J 데이터셋에 포함된 835개의 결함 인스턴스를 대상으로 한 실험 결과, 제안 기법은 두 가지 오픈소스 LLM 환경에서 MAP 및 MRR을 포함한 순위 기반 평가 지표에서 기존 LLM 기반 결함 위치 추정 결과 대비 일관된 개선 경향을 보였다. 동시에 정답 결함 함수의 급격한 순위 하락이나 Top-K 범위 이탈과 같은 회귀 사례는 제한적으로 관찰되었다 [14].

본 연구의 주요 기여도는 다음과 같다.

- LLM 기반 결함 위치 추정에서 반복 실행 시 발생하는 Top-K 순위 변동 및 Top-1 선택 불일치를 중심으로 출력 순위 불안정성 문제를 명시적으로 정의하고, 이를 사후 보정의 대상으로 정식화하였다.
- LLM의 입력 문맥이나 추론 구조를 변경하지 않고, 최종 출력 결과에 대해 최소 변경 원칙을 만족하는 실행 컨텍스트 기반 정합성 보정 기법을 제안하였다.
- Top-1 및 Top-5 순위 경계에 한정된 게이트 기반 보정 전략을 도입하여, 출력 안정성을 향상시키면서도 순위 왜곡과 회귀를 구조적으로 억제하였다.
- Defects4J 벤치마크를 활용한 실험을 통해 제안 기법이 기존 LLM 기반 결함 위치 추정 결과 대비 순위 기반 지표에서의 개선과 출력 안정성 향상을 제공함을 실증적으로 확인하였다 [14].

2. 배경지식

전통적 결함 위치 추정(Fault Localization, FL)은 테스트 실행 결과와 프로그램 실행 정보를 활용하여 코드 요소(라인, 블록, 메서드 등)의 결함 가능성을 정량적으로 산출하고 이를 랭킹 형태로 제시하는 기법이다 [1]. 이러한 접근은 디버깅 과정에서 개발자가 확인해야 할 코드 범위를 축소하는 것을 목표로 하며, 활용하는 정보의 유형에 따라 실행 기반 신호와 텍스트 기반 신호를 중심으로 발전해 왔다. 대표적인 실행 기반 접근인 스펙트럼 기반 결함 위치 추정(Spectrum-Based Fault Localization, SBFL)은 테스트 케이스의 성공·실패 여부와 코드 요소의 실행 빈도를 결합하여 의심도(suspiciousness) 점수를 계산한다 [4]. Tarantula [15], Ochiai 계열 [16], DStar [17] 등 다양한 의심도 공식이 제안되었으며, SBFL은 동일한 테스트 집합과 커버리지 정보가 주어질 경우 항상 동일한 랭킹을 산출하는 결정적 특성을 가져 비교 연구의 기준선으로 널리 활용되어 왔다 [6], [18], [19]. 그러나 SBFL은 테스트 및 커버리지 품질에 강하게 의존하며, 실패 테스트가 충분하지 않거나 다중 결함 환경에서는 개별 결함을 명확히 구분하기 어렵다는 한계를 가진다 [2], [3], [20].

이러한 실행 기반 접근의 한계를 보완하기 위해 정보검색(Information Retrieval, IR) 기반 버그 로컬라이제이션 기법이 제안되었다. IR 기반 접근은 버그 리포트를 자연어 질의로 처리하고, 소스 코드 요소를 문서 집합으로 모델링하여 텍스트 유사도를 계산함으로써 결함 후보를 식별한다 [5], [21]. 코드 주석, 식별자, 문자열 리터럴과 같은 정적 텍스트 정보를 활용함으로써 테스트 실행 정보가 제한적인 환경에서도 적용 가능하다는 장점을 가지며, 최근에는 실제 디버깅 효율을 고려하여 메서드 수준 결함 위치 추정으로 연구 범위가 확장되고 있다 [16], [18]. 그러나 IR 기반 기법 역시 코드의 실제 실행 여부나 실패 시점의 실행 경로와 같은 동적 정보를 직접 반영하지 못하며, 버그 리포트가 불완전한 경우 실제 결함과 무관한 코드 요소가 상위에 랭크될 수 있다는 한계를 가진다 [22].

대규모 언어 모델(LLM)의 발전은 결함 위치 추정 문제를

의미 기반 관점에서 재해석할 수 있는 계기를 제공하였다 [7]. LLM은 코드와 자연어를 동시에 학습함으로써 버그 리포트와 소스 코드 간의 의미적 연관성을 직접 추론할 수 있으며, 제로샷 또는 소수샷 환경에서도 활용 가능하다 [23]. 최근 연구들은 이러한 특성을 활용하여, 기존 IR 기반 기법을 보완하거나 테스트 실행 정보가 제한된 환경에서도 결함 위치를 추정하는 LLM 기반 파이프라인을 제안하고 있다 [8], [9], [12], [13]. 일반적으로 이러한 파이프라인은 후보 생성 단계를 통해 검색 공간을 축소한 뒤, LLM이 버그 리포트와 후보 코드 조각을 입력으로 받아 각 코드 요소의 결함 가능성을 판단하는 구조를 가진다 [24]. 이와 같은 접근은 코드와 자연어 간의 의미적 관계를 보다 직접적으로 반영할 수 있다는 장점을 제공한다.

그러나 LLM 기반 결함 위치 추정은 의미적 추론 능력과는 별도로, 결과의 신뢰성과 일관성 측면에서 구조적 한계를 가진다. 선행 연구들에 따르면, LLM은 실제 코드 구조나 실행 맥락과 일치하지 않는 결론을 문맥적으로 타당해 보이도록 생성하는 환각(hallucination) 현상을 보일 수 있으며 [10], [28], 이는 결함과 무관한 코드 요소를 상위 후보로 제시하는 형태로 나타날 수 있다 [25], [29]. 또한 LLM은 입력 컨텍스트의 길이와 구성 방식에 민감하게 반응하여, 장문 컨텍스트 환경에서는 중요한 단서가 충분히 반영되지 못하는 문제가 보고되어 왔다 [11], [25]. 아울러 동일한 후보 집합과 입력 정보가 주어지더라도 프롬프트 표현이나 디코딩 설정에 따라 출력 랭킹이 달라지는 비결정적 특성을 가지며, 이러한 변동성은 반복 실행 시 결과의 안정성과 활용 가능성을 저해한다 [30].

한편, IR 기반 및 LLM 기반 결함 위치 추정 기법은 공통적으로 코드가 실제로 실행되었는지 여부나 실패 시점의 호출 관계와 같은 실행 관점의 객관적 단서를 직접적으로 반영하지 못한다는 구조적 제약을 가진다 [22], [31], [32]. 특히 LLM 기반 접근에서는 의미적으로는 타당해 보이지만 실제 실패 테스트의 실행 경로와는 불일치하는 코드 요소가 상위에 제시되는 사례가 보고되어 왔다 [28], [29]. 실행 컨텍스트(execution context)는 실패 테스트 실행 과정에서 관측되는 스택 트레이스, 호출 프레임, 예외 발생 지점과 같은 경량 동적 정보를 의미하며 [1], 실패 시점의 실제 실행 경로를 직접적으로 반영하는 최소한의 동적 근거를 제공한다. 이러한 실행 컨텍스트는 기존 결함 위치 추정 결과를 대체하기보다는, 이미 생성된 후보 랭킹에 사후적으로(post-hoc) 적용되어 실행 경로와의 구조적 정합성을 평가하고 결과의 신뢰성과 안정성을 보완하는 보정 신호로 활용될 수 있다 [31]-[33].

3. 연구방법

본 연구는 LLM 기반 결함 위치 추정 결과의 출력 안정성을 향상시키기 위해, 실행 컨텍스트를 활용한 정합성 기반 사후 보정(post-hoc calibration) 기법을 제안한다. 제안 기법의 전체 파이프라인은 그림 1에 도식적으로 제시되어 있으며, 기존 LLM 기반 결함 위치 추정 파이프라인의 구조를 변경하거나 LLM의 입력 구성, 프롬프트 설계, 추론 전략을 수정하지 않고, LLM이 생성한 최종 Top-k 후보 순위에 독립적으로 적용되는 경량 후처리 모듈로 설계되었다. 이를 통해 추가적인 재학습이나 모델 수정 없이 기존 시스템에 바로 적용 가능하다.

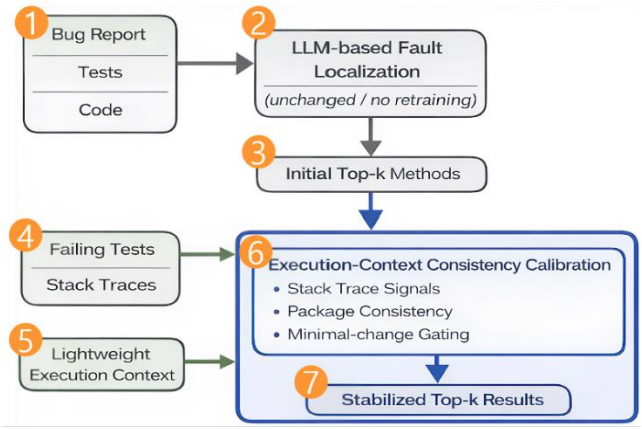


그림1. 제안한 방법 도식도

그림 1의 상단에 해당하는 기존 LLM 기반 결함 위치 추정 파이프라인은 버그 리포트와 소스 코드를 입력으로 받아 의심 메서드의 초기 Top-k 후보 순위를 생성하며, 본 연구에서는 이 결과를 기준 순위(baseline ranking)로 사용한다. 그림 1의 하단에 해당하는 실행 컨텍스트 기반 보정 파이프라인은 실패 테스트 실행 과정에서 관측된 스택 트레이스로부터 실행 컨텍스트를 추출하고, 이를 후보 메서드와의 구조적 정합성을 평가하기 위한 보조 신호로 활용한다. 실행 컨텍스트는 결함 위치의 정답을 직접 지시하는 오라클 정보가 아니며, LLM의 입력이나 추론 과정에는 전달되지 않는다. 이후 최소 변경(minimal-change) 원칙과 게이팅 규칙에 따라, 실행 컨텍스트 관점에서 기존 순위의 타당성을 합리적으로 의심할 수 있는 경우에 한해 제한적인 순위 조정을 수행하고, 그 외의 경우에는 LLM이 생성한 원래의 출력 순위를 그대로 유지한다. 이를 통해 제안 기법은 LLM의 의미적 추론 결과를 최대한 보존하면서도 반복 실행 환경에서의 출력 순위 변동과 결과 일관성을 보수적으로 완화한다.

3.1 실행 컨텍스트 정의 및 경량 신호 추출

본 연구에서 실행 컨텍스트(execution context)는 결함을 유발한 실패 테스트(failing test) 가 수행되는 과정에서 관측되는 호출 스택(call stack) 정보를 의미한다. 실행 컨텍스트는 프로그램의 실제 실행 경로를 반영하는 동적 정보이지만, 본 연구에서는 추가적인 계측이나 고비용의 동적 분석을 수행하지 않는다. 대신 실패 테스트 실행 시 자동으로 생성되는 스택 트레이스 로그만을 활용하여, 최소한의 실행 정보를 기반으로 사후 분석을 수행하는 경량 설계 원칙을 따른다.

각 결함 인스턴스에 대해 실패 테스트 실행 과정에서 생성된 스택 트레이스 파일을 입력으로 사용하며, 호출

프레임으로부터 클래스 및 패키지 경로를 추출한다. 이후 실행 컨텍스트는 호출 순서를 유지하는 시퀀스 형태가 아니라, 패키지 경로 단위의 집합(set) 형태로 구성된다. 구체적으로, 테스트 코드 프레임으로부터 생성된 테스트 패키지 집합과, 비테스트 코드 프레임으로부터 생성된 실행 패키지 집합을 각각 유지한다. 이러한 집합 기반 표현은 호출 순서나 반복 호출에 민감하지 않으면서도, 후보 메서드가 실패 테스트 실행 경로와 구조적으로 일관되는지를 평가하기 위한 최소한의 실행 정보를 제공한다.

본 연구는 위와 같이 구성된 실행 컨텍스트로부터 경량 실행 신호(lightweight execution signals) 를 생성하여, LLM이 생성한 후보 메서드와 관측된 실행 경로 간의 구조적 일관성을 평가한다. 각 후보 메서드에 대해, 해당 메서드가 속한 패키지 경로와 실행 컨텍스트에 포함된 패키지 경로 간의 공통 접두(prefix) 길이를 계산하며, 이 값은 후보 메서드가 실패 시점의 실행 경로와 구조적으로 얼마나 근접한지를 근사적으로 나타내는 지표로 사용된다. 실행 패키지 기반 신호는 주요 정합성 기준으로 활용되며, 테스트 패키지 기반 신호는 보조적인 구조 정보로 사용된다.

생성된 실행 컨텍스트 및 경량 실행 신호는 LLM의 입력으로 직접 사용되지 않는다. 실행 컨텍스트는 LLM의 추론 과정을 변경하거나 보강하기 위한 정보가 아니라, LLM이 이미 생성한 후보 순위가 실제 실행 경로와 구조적으로 일관되는지를 사후적으로 평가하기 위한 참조 정보로만 활용된다. 또한 실행 컨텍스트가 충분한 정보를 제공하지 못한다고 판단되는 경우에는 보정 단계를 적용하지 않으며, 해당 결함 인스턴스에 대해서는 LLM이 생성한 원래의 출력 순위를 그대로 유지한다. 이러한 조건부 적용 방식은 불완전한 실행 정보로 인한 과도한 순위 변경을 방지하고, 보수적이고 안정성 중심의 사후 보정 원칙을 유지하기 위한 설계 선택이다.

3.2 실행 컨텍스트-메서드 정합성 점수 계산

본 절에서는 3.1절에서 정의한 실행 컨텍스트로부터 추출된 경량 실행 신호를 이용하여, 각 후보 메서드가 실패 테스트 실행 경로와 구조적으로 얼마나 일관되는지를 정량적으로 평가하는 정합성 점수(consistency score) 를 정의한다. 제안하는 정합성 점수는 추가적인 학습 과정이나 동적 호출 그래프 구성과 같은 고비용 분석을 요구하지 않으며, 실패 테스트 실행 시 관측된 호출 스택 정보와 후보 메서드의 패키지 경로 간의 구조적 관계를 규칙 기반 방식으로 근사하도록 설계되었다.

정합성 점수 계산을 위해 각 후보 메서드는 완전한 메서드 식별자(fully qualified method name) 를 기준으로 처리된다. 구체적으로, 메서드 시그니처에서 인자 목록을 제거한 후 클래스명과 메서드명을 제외하고 패키지 수준의 경로만을 추출한다. 그 결과, 각 후보 메서드는 점(.)으로 구분되는 패키지 토큰 시퀀스로 표현되며, 이는 실행 컨텍스트와의 구조적 비교를 수행하기 위한 최소 단위로 사용된다.

실행 컨텍스트는 실패 테스트 실행 과정에서 관측된 호출 스택을 기반으로 구성되며, 테스트 실행 맥락을 반영하는 테스트 패키지 집합과 실패 시점의 실제 실행 경로를 반영하는 비테스트 실행 패키지 집합으로 구분된다. 두 집합은 테스트 코드와 결함 코드가 결함 위치 추정 과정에서 서로 다른 역할을 가질 수 있다는 점을 고려하여 독립적으로 유지된다.

후보 메서드와 실행 컨텍스트 간의 기본 정합성은 패키지 계층 구조 상에서 공유하는 상위 모듈의 깊이를 기준으로 평가된다. 이를 위해 후보 메서드의 패키지 경로와 실행 컨텍스트에 포함된 각 패키지 경로 간의 공통 접두(prefix) 길이를 계산한다. 공통 접두 길이는 두 경로가 앞에서부터 연속적으로 일치하는 패키지 토큰의 최대 개수를 의미하며, 값이 클수록 후보 메서드가 실패 테스트 실행 경로와 구조적으로 가까운 위치에 있음을 나타낸다.

실행 패키지 집합에 포함된 모든 경로에 대해 공통 접두 길이를 계산한 뒤, 그중 가장 큰 값을 후보 메서드의 기본 정합성 신호로 사용한다. 이는 실행 컨텍스트 중 하나의 경로라도 후보 메서드와 구조적으로 강한 일관성을 보이는 경우를 보존하기 위한 보수적인 설계 선택이다. 후보 메서드 m과 실행 패키지 집합 P간의 기본 정합성 점수는 다음과 같이 정의된다.

$$\mathrm{s}(\mathrm{pkg}(\mathrm{m}), \mathrm{P}) = max_{p \in P}\ \mathrm{cpl}(\mathrm{pkg}(\mathrm{m}), \mathrm{p})$$

- pkg(m)은 메서드 m이 속한 패키지 식별자를 의미한다.
- P는 실패 테스트 실행 과정에서 크래시 스택을 통해 관측된 실행 패키지 집합을 의미한다.
- cpl(·)은 두 패키지 경로가 앞에서부터 공유하는 공통 접두 토큰의 길이를 계산하는 함수이다.

본 연구에서는 동일 프로젝트 내에서 후보 메서드 간의 상대적 비교에만 해당 값을 사용하므로, 추가적인 정규화는 적용하지 않는다. 이러한 공통 접두 기반 계산 방식은 호출 그래프 분석이나 동적 호출 관계 추적과 같은 고비용 절차 없이도 실행 컨텍스트와 후보 메서드 간의 구조적 근접도를 안정적으로 근사할 수 있으며, 반복 실행 환경에서도 일관된 값을 제공한다.

본 연구에서는 테스트 패키지 집합과 비테스트 실행 패키지 집합에 대해 각각 독립적인 정합성 점수를 계산한다. 비테스트 실행 패키지 기반 정합성 점수는 실패가 발생한 실제 실행 경로와의 구조적 일관성을 반영하는 주요 신호로 사용되며, 테스트 패키지 기반 정합성 점수는 테스트 실행 맥락에서의 구조적 안정성을 보조적으로 반영하는 신호로 활용된다. 또한 패키지 경로 기반 구조 정보만으로 구분하기 어려운 후보 간의 미세한 차이를 반영하기 위해, 실행 컨텍스트에 포함된 패키지 토큰과 후보 메서드 식별자 간의 어휘적 중첩 정도를 측정한 어휘적 정합성 점수(lexical

consistency score)를 추가적인 보조 신호로 사용한다.

최종적으로, 후보 메서드 m에 대한 정합성 점수 $S(m)$는 다음과 같이 정의된다.

$$S(m) = s_{test}(m) + s_{crash}(m) + s_{lex}(m)$$

- $s_{test}(m)$ 은 테스트 패키지 기반 정합성 점수를 의미한다.
- $s_{crash}(m)$ 은 크래시 패키지 기반 정합성 점수를 의미한다.
- $s_{lex}(m)$은 어휘적 정합성 점수를 의미한다.

본 절에서 정의한 정합성 점수는 후보 메서드 간의 상대적 비교를 위한 보조 신뢰도 지표로만 사용되며, LLM이 생성한 원래의 순위를 직접 대체하지 않는다. 실제 순위 조정 여부와 조정 범위는 다음 절에서 설명하는 최소 변경(minimal-change) 원칙 기반의 게이팅 규칙에 의해 제한적으로 결정된다.

3.3 최소 변경 원칙 기반 최종 결함 위치 추정 결과 산출

본 연구에서 제안하는 실행 컨텍스트 기반 보정 단계는 LLM 기반 결함 위치 추정기가 생성한 초기 Top-k 순위를 전면적으로 재구성하거나, 새로운 랭킹 기준에 따라 후보 메서드를 재정렬하는 것을 목표로 하지 않는다. 대신, LLM이 생성한 출력 결과를 최대한 유지한 상태에서 실행 컨텍스트 관점에서 명확한 구조적 불일치가 관찰되는 경우에 한해 제한적인 순위 조정을 수행하는 것을 기본 설계 원칙으로 한다. 본 연구에서는 이러한 설계 철학을 최소 변경(minimal-change) 원칙이라 정의한다.

최소 변경 원칙에 따라, 실행 컨텍스트-메서드 정합성 점수는 후보 메서드의 실행 경로 적합성을 평가하기 위한 보조 신뢰도 신호로만 사용되며, 모든 후보 메서드에 대해 새로운 순위를 일괄적으로 산출하는 기준으로 사용되지 않는다. 즉, 정합성 점수는 LLM이 생성한 원래 순위를 직접 대체하지 않으며, 순위 조정이 정당화될 수 있는지를 판단하기 위한 조건 신호로만 활용된다. 이를 통해 LLM의 의미적 판단 결과는 가능한 한 유지되며, 보정 단계는 구조적 불일치가 충분히 명확한 경우에만 국소적으로 개입한다.

보정은 LLM이 생성한 초기 Top-k 후보 범위 내에서만 수행되며, 후보 메서드를 해당 범위를 벗어나도록 이동시키는 공격적인 재정렬은 허용되지 않는다. 이러한 제한은 보정 과정에서 새로운 오류가 도입되거나 기존 순위 구조가 과도하게 왜곡되는 것을 방지하기 위한 것이다. 결과적으로 보정 단계의 목적은 순위 구조 전체를 변경하는 것이 아니라, 기존 결과 내에서 제한적인 위치 조정을 통해 상위 순위에서 발생하는 불안정성을 완화하는 데 있다.

이러한 최소 변경 원칙에 따른 순위 보정 절차는 Algorithm 1에 요약되어 있다.

Algorithm 1 Execution-Context Consistency Calibration

Input: The bug identifier bug_id; The initial Top-5 ranking from LLM llm_rank; The candidate methods from input candidates; The directory of stack traces trace_dir; Calibration parameters params={λ,α, d_1, ,enable_top5_gate,top5_pos,max_insert,…}.
Output: The stabilized Top-5 ranking stabilized_rank.

```
pool = []
for method in llm_rank do
    pool.add(method)
for method in candidates do
    if method not in pool then pool.add(method)
ctx = extract_context(trace_dir, bug_id)
if is_insufficient(ctx) then return llm_rank
for method in pool do
    score = calc_consistency_score(method, ctx, params)
    scores.put(method, score)
base = llm_rank[0]    ▷ original Top-1 from LLM
best = argmax(scores, pool)
if (scores[best] - scores[base]) ≥ params.d1 then
    promote_to_first(pool, best)
if params.enable_top5_gate is True then
    idx = params.top5_pos
    cnt = 0
    while cnt < params.max_insert do
        gate_sc = scores[pool[idx]]
        cand = find_cand(pool, idx, gate_sc, params)
        if cand is None then break
        remove_and_insert(pool, cand, idx)
        cnt = cnt + 1
stabilized_rank = pool[:5]
return stabilized_rank
```

4. 실험

4.1 데이터셋

본 연구는 LLM 기반 결함 위치 추정 기법의 성능을 평가하기 위해 Defects4J v2.0.0 벤치마크를 사용하였다 [14]. Defects4J는 Java 기반 오픈소스 프로젝트로 구성된 표준 결함 데이터셋으로, 각 결함 인스턴스에 대해 실패 테스트 케이스와 정답 결함 위치 정보를 함께 제공한다. 본 연구는 Defects4J에 포함된 총 835개의 모든 결함 인스턴스를 샘플링 없이 사용하였다.

제안하는 실행 컨텍스트 기반 보정 기법은 각 결함 인스턴스의 실패 테스트 실행 과정에서 관측되는 호출 스택(call stack) 정보를 실행 컨텍스트로 활용한다. 해당 실행 정보는 Defects4J에서 제공하는 공식 테스트 스위트를 통해 수집되었으며, 추가적인 코드 계측이나 고비용의 동적 분석은 수행하지 않았다.

LLM 기반 결함 위치 추정기의 입력으로는 버그 리포트, 실패 테스트 정보, 그리고 소스 코드가 사용되었다. 본 연구의 기법은 기존 LLM 기반 결함 위치 추정기의 출력 결과를 사후적으로 보정하는 방식이므로, 데이터셋 구성과 입력 정보는 선행 연구와 동일하게 유지하였다.

실험에서는 Llama3-8B와 Qwen2.5-7B 두 가지 오픈소스 LLM을 사용하였다. Llama3-8B는 FlexFL 연구에서 사용된

설정을 그대로 적용하였으며 [13], [34], Qwen2.5-7B는 모델 일반화 가능성을 확인하기 위한 보조 모델로 활용하였다 [35]. 모든 모델에 대해 추론 하이퍼파라미터는 선행 연구와의 공정한 비교를 위해 동일하게 유지하였다. 모든 실험은 결함 인스턴스 단위로 독립적으로 수행되었다.

4.2 평가 지표 및 베이스라인

제안하는 실행 컨텍스트 기반 정합성 보정 기법의 효과를 평가하기 위해, 본 연구에서는 결함 위치 추정 연구에서 널리 사용되는 정확도 및 순위 기반 지표와 함께, 출력 안정성을 반영하는 보조 지표를 사용하였다. 모든 평가는 결함 인스턴스 단위로 수행되었으며, 각 지표는 전체 결함 인스턴스에 대해 집계된 결과를 기준으로 산출하였다.

결함 위치 추정의 정확도를 평가하기 위해 Top-1, Top-3, Top-5 정확도를 사용하였다. Top-k 정확도는 정답 결함 메서드가 상위 k개의 후보 목록에 포함되는지를 기준으로 하며, 실제 디버깅 과정에서 개발자가 우선적으로 검토하는 코드 범위를 직접적으로 반영한다.

상위 후보들의 순위 품질을 보다 정밀하게 평가하기 위해 Mean Average Precision (MAP)과 Mean Reciprocal Rank (MRR)를 함께 사용하였다. MAP는 각 결함 인스턴스에 대해 계산된 Average Precision을 평균한 값으로, 정답 결함 메서드가 순위 전반에서 얼마나 일관되게 상위에 위치하는지를 반영한다. MRR은 각 결함 인스턴스에서 정답 결함 메서드가 처음 등장하는 순위의 역수(1/rank)를 평균한 지표로, 정답이 상위에 위치할수록 높은 값을 가진다.

본 연구는 단일 실행에서의 정확도 향상뿐만 아니라, LLM 기반 결함 위치 추정 결과의 출력 안정성과 결과 일관성을 핵심 평가 관점으로 설정하였다. 이를 위해, 실행 컨텍스트 기반 보정 적용 이후 정답 결함 메서드의 순위가 기존 LLM 출력 대비 하락하거나 Top-k 범위를 벗어나는 경우를 회귀(regression)로 정의하고, 해당 사례의 발생 빈도를 함께 분석하였다. 이 지표는 보정 단계가 기존 출력 결과를 과도하게 변경하지 않으면서도 안정적인 개선을 제공하는지를 평가하기 위한 보조 기준으로 사용된다.

베이스라인으로는 실행 컨텍스트 기반 보정을 적용하지 않은 LLM 기반 결함 위치 추정 결과를 사용하였다. Llama3-8B의 경우, 기존 FlexFL 연구에서 공개된 구현과 동일한 추론 구조 및 설정을 적용한 결과를 베이스라인으로 사용하였다 [13], [34]. Qwen2.5-7B의 경우 공식 출력이 공개되어 있지 않으므로, FlexFL 논문에 기술된 절차를 기반으로 동일한 입력 구성과 설정 하에서 추론 파이프라인을 재구성하여 베이스라인 결과를 생성하였다 [13], [35].

모든 비교 실험은 동일한 데이터셋, 입력 정보, 평가 지표를 기준으로 수행되었으며, 실험 조건 간의 차이는 실행 컨텍스트 기반 보정 단계의 적용 여부에 한정된다. 이를 통해 관찰된 성능 변화가 입력 구성이나 추론 설정이 아닌, 제안하는 보정 기법의 효과에 기인함을 분리하여 분석하였다.

4.3 실험 결과

제안하는 실행 컨텍스트 기반 정합성 보정 기법의 정량적 효과는 표 1에 요약되어 있다. 표 1은 Qwen2-7B와 Llama3-8B를 기반으로 한 베이스라인 결과와, 실행 컨텍스트 및 어휘 정합성 기반 보정을 적용한 결과를 Top-k 정확도와 순위 기반 지표 관점에서 비교한 것이다.

표1. 베이스라인 모델 및 제안 기법의 결함 위치 추정 성능 비교

Base Model	Top-1	Top-3	Top-5	MAP	MRR
Qwen2-7B	316 (37.8%)	453 (54.2%)	504 (60.3%)	0.405	0.467
Qwen2-7B(Ours)	325 (38.9%)	463 (55.5%)	517 (61.1%)	0.416	0.480
Llama3-8B	350 (41.9%)	478 (57.3%)	529 (63.4%)	0.439	0.501
Llama3-8B(Ours)	354 (42.4%)	485 (58.1%)	540 (64.7%)	0.448	0.508

두 모델 모두에서 보정 적용 이후 Top-1, Top-3, Top-5 정확도가 일관되게 증가하였다. Qwen2-7B의 경우 Top-1 정확도는 37.8%에서 38.9%로, Top-5 정확도는 60.3%에서 61.1%로 개선되었으며, Llama3-8B에서도 Top-1 정확도는 41.9%에서 42.4%, Top-5 정확도는 63.4%에서 64.7%로 증가하였다. 이는 제안 기법이 서로 다른 LLM 환경에서도 동일한 방향의 성능 변화를 유도함을 보여준다.

순위 품질을 반영하는 MAP와 MRR 역시 두 모델 모두에서 향상되었다. Qwen2-7B에서는 MAP가 0.405에서 0.416으로, MRR은 0.467에서 0.480으로 증가하였으며, Llama3-8B에서도 MAP는 0.439에서 0.448, MRR은 0.501에서 0.508로 각각 개선되었다. 이러한 결과는 제안 기법이 정답 결함 메서드를 상위 후보 집합 내에서 평균적으로 더 앞선 위치로 이동시키는 경향을 보였음을 의미한다.

다만 성능 향상 폭은 절대적인 수치 기준으로는 제한적인 수준에 머문다. 이는 본 기법이 LLM의 출력 순위를 전면적으로 재구성하지 않고, Top-k 범위 내에서만 제한적인 순위 조정을 허용하도록 설계되었기 때문이다. 즉, 제안 기법은 새로운 후보를 생성하거나 대규모 재정렬을 수행하기보다는, 상위 후보 집합 내부에서 국소적인 위치 조정을 수행하는 보수적인 사후 보정 기법으로 작동한다.

정합성 가중치 λ에 대한 민감도 분석 결과는 그림 2에 제시되어 있다. 어휘 정합성 가중치 α를 0.5로 고정한 상태에서 λ 값을 변화시킨 결과, λ=0에서 0.5로 증가할 때 MAP@5는 일관되게 향상되었으며, 이후 값 증가에 따른 성능

변화는 제한적으로 유지되었다. 이에 따라 본 연구에서는 성능 변화와 순위 안정성 간의 균형을 고려하여 λ=0.5를 모든 실험의 기본 설정으로 사용하였다.

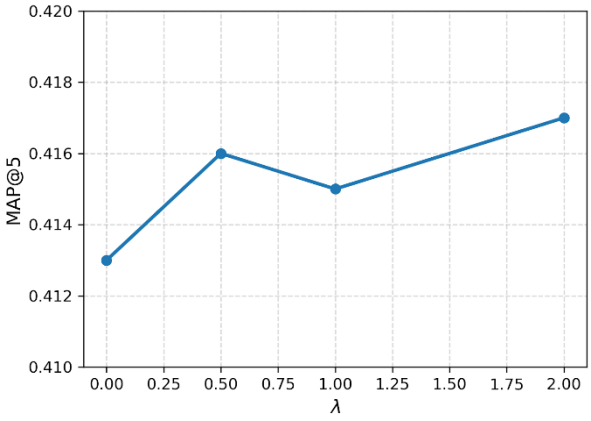


그림 2. 어휘 정합성 가중치(α=0.5)를 고정한 상태에서 정합성 가중치 λ 변화에 따른 MAP@5 민감도 분석(Qwen2-7B)

정확도 및 순위 품질 분석과 더불어, 보정 기법이 출력 결과의 안정성에 미치는 영향도 함께 분석하였다. 동일한 Qwen2-7B 모델과 동일한 입력 설정 하에서 서로 다른 시점에 독립적으로 수행된 두 번의 추론 결과를 비교한 출력 일관성 분석 결과는 표 2에 요약되어 있다. 보정 적용 이후 Top-1 불일치율(Disagreement@1)은 35.93%에서 34.49%로 감소하였으며, Top-5 후보 집합 간 Jaccard 유사도는 0.5315에서 0.5581로 증가하였다. 또한 Top-5 후보 집합의 완전 일치율(Exact Match@5) 역시 12.81%에서 14.13%로 향상되었다.

표2. 반복 실행 간 출력 일관성 비교 (Qwen2-7B)

Metric	Raw LLM Output	After Calibration
Disagreement@1 ↓	35.93%	34.49%
Jaccard@5 ↑	0.5315	0.5581
Exact Match@5 ↑	12.81%	14.13%

이러한 결과는 실행 컨텍스트 기반 보정이 개별 실행 결과를 임의로 변경하는 방식이 아니라, 실패 테스트 실행 경로에서 관측되는 비교적 안정적인 구조적 신호를 기준으로 반복 실행 간 출력 결과를 보다 유사한 방향으로 수렴시키는 역할을 수행했음을 보여준다. 종합하면, 제안 기법은 단일 실행 기준에서의 대규모 성능 향상을 목표로 하기보다는, 반복 실행 환경에서 LLM 기반 결함 위치 추정 결과의 순위 변동과 회귀 현상을 완화하는 데 초점을 둔 안정화 기법으로 해석할 수 있다. 이러한 실험 결과는 실행 컨텍스트 기반 정합성 보정이 LLM의 기존 의미적 판단을 최대한 유지하면서도, 출력 결과의 신뢰성과 일관성을 보조적으로 향상시키는 실용적인 후처리 기법임을 보여준다.

4.4 Case Study

본 절에서는 실행 컨텍스트 기반 정합성 보정 기법이 개별 결함 인스턴스에서 실제로 어떠한 조건 하에서 순위 변화를 유도하는지를 사례 기반으로 분석한다. 평균 성능 지표만으로는 확인하기 어려운 보정 단계의 실제 동작 특성을 명확히 하기 위해, 본 분석은 보정이 적용되어 순위가 개선된 사례, 제한적인 순위 하락이 발생한 사례, 그리고 보정이 적용되지 않아 순위 변화가 발생하지 않은 사례를 함께 포함한다. 이를 통해 제안 기법이 언제 개입하고, 언제 개입하지 않으며, 순위 변화가 어느 범위로 제한되는지를 구체적으로 설명하고자 한다. 분석에 사용된 결함 인스턴스와 보정 전후의 순위 변화는 표 3에 요약되어 있다.

표3. 실행 컨텍스트 기반 정합성 보정의 사례 분석 결과

Model	Bug ID	Rank Change	Outcome
Llama3	Jackson-28	2 → 1	Improved
Llama3	Math-14	4 → 1	Improved
Qwen2	Math-57	3 → 1	Improved
Llama3	Chart-18	1 → 2	Regression
Qwen2	Lang-6	-	No-change

표 3에 제시된 순위 개선 사례는 Llama3 기반의 Jackson-28, Math-14와 Qwen2 기반의 Math-57이다. 이들 결함 인스턴스에서는 보정 적용 이후 정답 결함 메서드의 순위가 각각 2위에서 1위로, 4위에서 1위로, 그리고 3위에서 1위로 이동하였다. 세 사례 모두에서 공통적으로 관찰된 점은, 정답 결함 메서드가 실패 테스트 실행 과정에서 관측된 스택 트레이스에 포함된 패키지 경로와 비교적 긴 공통 접두(prefix)를 공유하고 있었다는 것이다. 이는 정답 메서드가 실제 실행 경로와 구조적으로 근접해 있음을 의미한다. 그러나 이러한 구조적 근접성은 초기 LLM 출력 순위에 명시적으로 반영되지 않았고, 결과적으로 정답 메서드는 최상위 후보로 선택되지 않았다. 보정 단계에서는 실행 컨텍스트-메서드 정합성 점수가 계산되었으며, 정답 메서드와 기존 Top-1 후보 간의 정합성 점수 차이가 사전에 정의된 게이팅 임계값을 초과한 경우에 한해, 최소 변경 원칙에 따라 Top-1 위치에서의 국소적인 순위 교체가 수행되었다. 이 과정에서 Top-5 후보 집합은 유지되었으며, 순위 변화는 최상위 위치로 제한되었다.

반면, Llama3 기반의 Chart-18 사례에서는 보정 적용 이후 정답 결함 메서드의 순위가 1위에서 2위로 하락하는 제한적인 순위 변화가 관찰되었다. 이 사례에서도 정답 메서드는 Top-5 범위 내에 유지되었으며, 순위 변화는 인접한 후보 간의 단일

교환으로 제한되었다. 실행 컨텍스트 기반 정합성 점수 계산 결과, 정답 메서드보다 더 높은 정합성 점수를 갖는 다른 후보 메서드가 식별되었고, 두 후보 간 점수 차이가 게이팅 임계값을 초과함에 따라 Top-1 위치에서만 제한적인 순위 조정이 수행되었다. 이는 제안 기법이 정답 메서드에 대해 항상 단조적인 순위 개선을 보장하지 않으며, 실행 경로 관점에서 구조적 정합성이 더 높은 후보가 존재하는 경우에는 제한적인 순위 하락도 허용함을 보여준다.

한편, Qwen2 기반의 Lang-6 사례에서는 보정 전후의 순위 변화가 관찰되지 않았다. 해당 결함 인스턴스에서는 실패 테스트 실행 과정에서 수집된 스택 트레이스의 호출 프레임 수가 제한적이거나, 후보 메서드 간 실행 컨텍스트 기반 정합성 점수 차이가 게이팅 임계값을 초과하지 않았다. 그 결과 보정 단계는 적용되지 않았으며, LLM이 생성한 원래의 순위 결과가 그대로 유지되었다. 이는 제안 기법이 실행 컨텍스트 정보가 불충분하거나 정합성 차이가 명확하지 않은 경우에는 개입을 수행하지 않도록 설계되었음을 보여준다.

이러한 사례 분석을 통해, 실행 컨텍스트 기반 정합성 보정 기법은 모든 결함 인스턴스에 대해 일괄적으로 순위를 재조정하는 방식이 아니라, 실행 경로 정보가 충분하고 구조적 정합성 차이가 명확하게 관찰되는 경우에만 제한적으로 개입함을 확인할 수 있다. 또한 순위 변화가 발생하더라도, 그 범위는 Top-1 또는 Top-5 경계 내의 국소적인 위치 교환으로 제한되며, 기존 후보 집합이나 순위 구조 전체를 재구성하지 않는다. 이는 제안 기법이 LLM의 기존 출력 결과를 최대한 유지하면서도, 실행 컨텍스트 관점에서 명백한 불일치가 발생하는 경우에만 선택적으로 순위를 조정하는 안정화 중심의 후처리 기법임을 사례 수준에서 뒷받침한다.

5. 토의

5.1 실험 결과 분석

본 절에서는 4장에서 보고한 실험 결과를 바탕으로, 실행 컨텍스트 기반 정합성 보정 기법이 LLM 기반 결함 위치 추정 결과에 미친 영향을 분석한다. 본 연구의 목적은 절대적인 정확도 향상이 아니라, 동일한 입력과 추론 설정 하에서 반복 실행 시 발생하는 출력 순위 변동성을 완화하고, 상위 후보 순위의 신뢰도를 제한적으로 조정하는 사후 보정 기법의 효과를 검증하는 데 있다.

Top-k 정확도 기준에서, 제안 기법은 Qwen2-7B와 Llama3-8B 두 모델 모두에서 Top-1, Top-3, Top-5 정확도가 일관되게 증가하는 경향을 보였다. Top-1 정확도는 약 0.5-1.1%p, Top-5 정확도는 약 0.8-1.3%p 범위에서 개선되었으며, 이는 후보 집합을 변경하지 않고 Top-k 범위 내에서 순위를 제한적으로 조정하는 설계 특성과 일관된 결과로 해석할 수 있다.

순위 품질 지표인 MAP와 MRR 역시 두 모델 모두에서 증가하였다. MAP는 약 0.009-0.011, MRR은 약 0.007-0.013 수준으로 개선되었으며, 이는 정답 결함 메서드의 평균 순위가 상위 구간 전반에서 완만하게 조정되었음을 의미한다. 이러한 결과는 실행 컨텍스트 기반 정합성 점수가 특정 후보를 강하게 선택하는 결정 신호라기보다는, 상위 후보 집합 내부에서 구조적 일관성을 반영하는 보조 신호로 작동했음을 보여준다.

모델별로는 성능 개선의 양상이 다소 상이하게 나타났다. Qwen2-7B에서는 Top-1 정확도의 개선이 상대적으로 두드러졌고, Llama3-8B에서는 Top-3 및 Top-5 정확도의 증가폭이 더 크게 관찰되었다. 이는 제안 기법이 특정 순위 지표를 직접 최적화하기보다는, 각 모델이 생성하는 초기 순위 분포의 특성에 따라 작동 범위가 달라지는 보정 계층임을 보여준다.

보정 이후의 순위 변화는 대부분 상위 후보 간의 국소적인 위치 교환 형태로 나타났으며, 새로운 후보의 추가나 기존 후보의 제거는 발생하지 않았다. 이로 인해 정답 결함 메서드가 급격히 하락하거나 Top-k 범위를 이탈하는 회귀 사례는 제한적으로 관찰되었다.

또한 반복 실행 간 출력 비교 결과, 보정 적용 이후 Top-1 불일치율이 감소하고 Top-5 후보 집합 간 유사도가 증가하였다. 이는 실행 컨텍스트 기반 보정이 개별 실행 결과를 임의로 변경하는 것이 아니라, 실패 테스트 실행 경로라는 비교적 안정적인 기준을 통해 서로 다른 추론 결과를 유사한 방향으로 수렴시키는 역할을 수행했음을 의미한다. 종합하면, 제안 기법은 출력 결과의 절대적 정확도를 크게 변화시키기보다는, 반복 실행 환경에서의 출력 안정성과 상위 후보 순위의 신뢰도를 보조적으로 향상시키는 데 기여한 것으로 해석할 수 있다.

5.2 위협요소

본 연구의 실험 결과는 제안하는 실행 컨텍스트 기반 정합성 보정 기법의 설계 선택과 실험 범위로부터 기인하는 몇 가지 타당성 위협을 내포한다. 본 절에서는 이러한 위협이 결과 해석에 부여하는 제약 조건을 중심으로 논의한다.

내적 타당성(internal validity) 측면에서의 주요 위협은 실행 컨텍스트 정보의 불완전성이다. 본 연구에서 활용한 실행 컨텍스트는 실패 테스트 실행 시 생성되는 스택 트레이스에 기반하므로, 전체 실행 경로나 모든 호출 관계를 포괄하지 않는다. 이로 인해 일부 결함 인스턴스에서는 정답 결함 메서드가 호출 스택에 포함되지 않거나, 제한적인 상위 프레임 정보만 제공될 수 있다. 이러한 경우 실행 컨텍스트 기반 정합성 점수가 실제 결함 관련성을 충분히 반영하지 못할 가능성이 존재한다. 본 연구는 이러한 위험을 완화하기 위해, 실행 컨텍스트가 사전에 정의된 최소 정보 기준을 충족하지 못하는 경우 보정 단계를 적용하지 않는 조건부 전략을 채택하였다.

구성 타당성(construct validity) 측면에서는 출력 안정성을 완전히 정량화하기 어렵다는 한계가 존재한다. 본 연구에서 사용한 Top-k 정확도, MAP, MRR은 순위 품질을 평가하는 표준 지표이지만, 반복 실행 환경에서의 미세한 순위 변동이나 개발자 관점의 신뢰도를 직접 측정하지는 않는다. 이에 따라

본 연구에서는 정답 결함 메서드의 순위 하락 또는 Top-k 범위 이탈을 회귀(regression)로 정의하여 안정성 분석에 포함하였다. 해당 정의는 출력 안정성의 모든 측면을 포착하기보다는, 보정으로 인한 명백한 성능 저하 여부를 검증하기 위한 보수적인 기준으로 해석되어야 한다.

외적 타당성(external validity) 측면의 위협은 실험 대상과 실행 환경의 제한에서 비롯된다. 본 연구는 Defects4J에 포함된 Java 기반 오픈소스 프로젝트를 대상으로 수행되었으며, 실제 산업 환경에서는 실패 테스트의 가용성이나 스택 트레이스 품질이 상이할 수 있다. 이러한 환경에서는 실행 컨텍스트 기반 보정 단계의 적용 빈도가 감소할 수 있으며, 그 결과 본 연구에서 관찰된 효과가 동일하게 재현되지 않을 가능성이 있다. 따라서 본 연구의 결과는 최소한의 실패 테스트 실행 정보가 확보되는 환경을 전제로 해석되어야 한다.

마지막으로, 제안 기법은 LLM이 생성한 초기 후보 순위 분포에 의존하는 구조를 가지므로, 출력 분포가 지나치게 평탄하거나 실행 컨텍스트와의 구조적 정합성이 전반적으로 낮은 모델 환경에서는 보정 효과가 제한될 수 있다. 이는 본 기법이 모델 독립적인 성능 증폭 기법이라기보다는, 초기 출력 분포와 실행 컨텍스트 신호 간의 상호작용에 따라 작동 범위가 결정되는 사후 보정 계층임을 의미한다.

6. 관련연구

최근 LLM 기반 결함 위치 식별 연구는 코드와 자연어 간의 의미적 관계를 활용하여 기존 통계적·정보 검색 기반 기법의 한계를 보완하는 방향으로 발전해 왔다. 대표적인 예로 Xu et al.이 제안한 FlexFL은 LLM을 활용한 이단계 결함 위치 식별 파이프라인을 제시한다 [13]. FlexFL은 후보 공간 축소 단계와 최종 순위 정제 단계를 분리함으로써, LLM의 추론을 제한된 후보 집합에 집중시키는 구조를 채택한다. 1단계에서는 SBFL, IRFL, HybridFL 등 기존 기법과 LLM 기반 에이전트를 결합하여 의심 메서드 후보군을 생성하고, 2단계에서는 LLM이 후보 메서드의 코드 스니펫을 분석하여 최종 순위를 산출한다. 해당 프레임워크는 Llama3-8B [34], Qwen2-7B [35], Mistral-Nemo-12B 등 오픈소스 LLM을 기반으로 구성되어, 재현성과 실용성 측면에서의 장점을 제공하며, Defects4J 벤치마크에서 경쟁력 있는 Top-k 성능을 보고하였다.

한편 테스트 실행 정보에 의존하지 않는 테스트 비의존(test-free) LLM 기반 결함 위치 식별 연구도 제안되었다. Yang et al.의 LLMAO는 소스 코드 자체만을 입력으로 사용하여 결함 위치를 추정하는 접근을 제시하며 [23], LLM이 사전 학습 과정에서 획득한 코드 의미 표현을 활용한다. Defects4J, BugsInPy, Devign 등 다양한 벤치마크에서 실험을 수행하여 테스트 실행이 어려운 환경에서도 적용 가능함을 보였다. 그러나 이러한 접근은 실패 테스트의 실행 경로, 호출 관계와 같은 동적 컨텍스트를 명시적으로 반영하지 않으며, LLM 출력 결과의 변동성이나 반복 실행 간 순위 일관성 문제를 직접적으로 다루지는 않는다.

이와 같이 기존 LLM 기반 결함 위치 식별 연구는 파이프라인 구조 개선, 후보 공간 축소, 테스트 비의존성 확보 등 다양한 방향으로 발전해 왔으나, LLM이 생성한 최종 출력 결과의 신뢰성과 반복 실행 간 일관성 문제는 상대적으로 독립적인 연구 주제로 충분히 다뤄지지 않았다. 특히 동일한 입력과 설정 하에서도 출력 순위가 변동하는 현상은 확률적 언어 모델 구조에 내재된 특성으로 보고되고 있으며 [29], 순위 기반 의사결정이 요구되는 결함 위치 식별 작업에서 실질적인 활용상의 문제로 작용할 수 있다.

전통적인 결함 위치 식별 연구에서는 이러한 불확실성을 완화하기 위해 실행 기반 신호를 핵심 기준으로 활용해 왔다 [4]. 실행 정보는 의미 기반 추론 결과를 보정하는 객관적 기준으로 기능할 수 있으며, 최근 연구들 역시 LLM의 구조적 한계를 전제로 실행 신호를 활용한 보정의 필요성을 제기하고 있다 [31]. 본 논문은 이러한 연구 흐름에 기반하여, LLM의 입력이나 추론 과정을 변경하지 않고 실행 컨텍스트와의 정합성 관점에서 출력 결과를 사후적으로 보정함으로써, LLM 기반 결함 위치 식별 결과의 안정성과 일관성을 향상시키는 접근을 제안한다.

7. 결론

본 논문은 LLM 기반 결함 위치 추정 기법이 의미적 코드 이해 능력 측면에서는 경쟁력 있는 성능을 보이고 있음에도 불구하고, 동일한 입력과 추론 설정 하에서 반복 실행 시 출력 순위가 변동하는 현상과 그로 인한 결과 일관성 저하 문제가 체계적으로 다뤄지지 않았다는 점에 주목하였다. 기존 연구들이 주로 파이프라인 구조 개선이나 추론 전략 확장을 통해 정확도 향상에 초점을 두어 온 반면, LLM이 생성한 최종 출력 결과를 유지한 채 안정성을 사후적으로 조정하는 접근은 상대적으로 명확히 논의되지 않았다.

이를 위해 본 논문에서는 실패 테스트 실행 과정에서 관측되는 호출 스택 정보를 활용한 실행 컨텍스트 기반 정합성 보정 기법을 제안하였다. 제안 기법은 LLM의 입력 구성이나 추론 과정을 변경하지 않고, 경량 실행 신호를 기반으로 후보 메서드 순위와 실제 실행 경로 간의 구조적 정합성을 평가한 뒤, 최소 변경 원칙과 보수적인 게이팅 규칙에 따라 Top-k 범위 내에서만 제한적인 순위 조정을 수행한다. 또한 실행 컨텍스트 정보가 충분하지 않은 경우에는 보정 단계를 비활성화하여, 불완전한 실행 정보로 인한 과도한 개입을 방지하도록 설계하였다.

Defects4J 벤치마크의 835개 결함 인스턴스를 대상으로 두 가지 오픈소스 LLM 환경에서 수행한 실험 결과, 제안 기법은 Top-k 정확도, MAP, MRR 등 순위 기반 지표에서 일관된 개선 경향을 보였다. 절대적인 성능 향상 폭은 제한적이었으나, 이는 출력 순위를 전면적으로 재구성하지 않고 상위 후보 간의 상대적 위치만을 조정하는 설계 특성을 반영한 결과로 해석할 수 있다. 동시에 정답 결함 메서드의 급격한 순위 하락이나 Top-k 범위 이탈과 같은 회귀 사례는 제한적으로 관찰되었으며, 반복 실행 환경에서 상위 후보 집합의 일관성이 완화되는 경향이 확인되었다.

이러한 결과는 실행 컨텍스트 기반 사후 보정이 LLM의 의미적 추론 결과를 대체하지 않더라도, 출력 순위의 안정성과 신뢰성을 보조적으로 향상시킬 수 있음을 보여준다. 본 연구는 LLM 기반 결함 위치 추정을 단순한 정확도 최적화 문제를 넘어, 반복 실행 환경에서의 출력 일관성과 신뢰성 관점에서 함께 분석할 필요성을 제기한다는 점에서 의의를 가진다. 향후 연구로는 다양한 실행 신호를 활용한 정합성 평가 방식의 확장과, 출력 안정성을 보다 정밀하게 포착할 수 있는 평가 지표 설계가 요구된다.

참고문헌

[1] W. E. Wong et al., "A survey on software fault localization," IEEE Trans. Softw. Eng. (TSE), vol. 42, no. 8, pp. 707-740, Aug. 2016.

[2] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in Proc. ISSTA, 2011, pp. 199-209.

[3] P. S. Kochhar et al., "Practitioners' expectations on automated fault localization," in Proc. ISSTA, 2016, pp. 165-176.

[4] R. Abreu et al., "On the accuracy of spectrum-based fault localization," in Proc. TAICPART-MUTATION, 2007, pp. 89-98.

[5] J. Zhou et al., "Where should the bugs be fixed? More accurate information retrieval-based bug localization," in Proc. ICSE, 2012, pp. 14-24.

[6] D. Zou et al., "An empirical study of fault localization families and their combinations," IEEE Trans. Softw. Eng. (TSE), vol. 47, no. 2, pp. 332-347, 2021.

[7] M. Chen et al., "Evaluating large language models trained on code," arXiv:2107.03374, 2021.

[8] Y. Wu et al., "Large language models in fault localisation," arXiv:2308.15276, 2023.

[9] S. Kang et al., "Large language models are few-shot testers: Exploring LLM-based general bug reproduction," in Proc. ICSE, 2023, pp. 2312-2323.

[10] Y. Zhang et al., "Siren's song in the AI ocean: A survey on hallucination in large language models," arXiv:2309.01219, 2023.

[11] N. F. Liu et al., "Lost in the middle: How language models use long contexts," Trans. Assoc. Comput. Linguist. (TACL), vol. 12, pp. 157-173, 2024.

[12] Y. Qin et al., "AgentFL: Scaling LLM-based fault localization to project-level context," arXiv:2403.16362, 2024.

[13] C. Xu et al., "FlexFL: Flexible and effective fault localization with open-source large language models," arXiv:2411.10714, 2024.

[14] R. Just et al., "Defects4J: A database of existing faults to enable controlled testing studies," in Proc. ISSTA, 2014, pp. 437-440.

[15] K. C. Youm et al., "Bug localization based on code change histories and bug reports," in Proc. APSEC, 2015, pp. 190-197.

[16] S. Tsumita et al., "Large-scale evaluation of method-level bug localization with FinerBench4BL," in Proc. SANER, 2023, pp. 815-824.

[17] W. Zhang et al., "FineLocator: A novel approach to method-level fine-grained bug localization," Inf. Softw. Technol. (IST), vol. 110, pp. 121-135, 2019.

[18] A. Razzaq et al., "BoostNSift: A query boosting and code sifting technique for method level bug localization," in Proc. SCAM, 2021, pp. 81-91.

[19] W. E. Wong et al., "The DStar method for effective software fault localization," IEEE Trans. Reliab., vol. 63, no. 1, pp. 290-308, 2014.

[20] M. Zhang et al., "An empirical study of boosting spectrum-based fault localization via PageRank," IEEE Trans. Softw. Eng. (TSE), vol. 47, no. 6, pp. 1089-1113, 2021.

[21] C. D. Manning et al., Introduction to Information Retrieval. Cambridge Univ. Press, 2008.

[22] T. B. Le et al., "Information retrieval and spectrum based bug localization: Better together," in Proc. ESEC/FSE, 2015, pp. 579-590.

[23] A. Z. H. Yang et al., "Large language models for test-free fault localization," in Proc. ICSE, 2024, pp. 1-12.

[24] S. Kang et al., "A quantitative and qualitative evaluation of LLM-based explainable fault localization," Proc. ACM Softw. Eng. (FSE), vol. 1, Jul. 2024.

[25] F. Liu et al., "Exploring and evaluating hallucinations in LLM-powered code generation," arXiv:2404.00971, 2024.

[26] R. Widyasari et al., "Demystifying faulty code: Step-by-step reasoning for explainable fault localization," in Proc. SANER, 2024, pp. 568-579.

[27] J. Wang et al., "Software testing with large language models: Survey, landscape, and vision," IEEE Trans. Softw. Eng. (TSE), 2024.

[28] L. Huang et al., "A survey on hallucination in large language models," arXiv:2311.05232, 2023.

[29] Z. Xu et al., "Hallucination is inevitable: An innate limitation of large language models," arXiv:2401.11817, 2024.

[30] X. Chen et al., "Premise order matters in reasoning with large language models," arXiv:2402.08939, 2024.

[31] Y. Lou et al., "Can automated program repair refine fault localization?" in Proc. ISSTA, 2020, pp. 75-87.

[32] X. Li et al., "DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization," in Proc. ISSTA, 2019, pp. 169-180.

[33] J. Sohn and S. Yoo, "FLUCCS: Using code and change metrics to improve fault localization," in Proc. ISSTA, 2017, pp. 273-283.

[34] Meta, "Meta Llama 3," Meta AI Technical Blog, 2024.

[35] B. Hui et al., "Qwen2.5-Coder technical report," arXiv:2409.12186, 2024.

단일 LLM 기반 테스트 Assertion 생성의 품질 강화: 변이 점수 피드백과 뮤턴트-가드 손실 결합

김윤기[O1], 양근석[2†]
[1] 한경국립대학교 컴퓨터응용수학부, [2] 한경국립대학교 컴퓨터응용수학부 (컴퓨터시스템연구소)
kimyungi0101@hknu.ac.kr, gsyang@hknu.ac.kr

Enhancing Assertion Quality in Single-LLM-Based Test Generation: Combining Mutation Score Feedback with Mutant-Guard Loss

YUNGI KIM[O1], Geunseok Yang[2†]
[1] Department of Computer Applied Mathematics, Hankyong National University
[2] Department of Computer Applied Mathematics (Computer System Institute), Hankyong National University

요 약

대형언어모델(LLM) 기반 자동 테스트 생성은 주로 코드 커버리지로 성능을 평가해 왔으나, 이는 생성된 테스트의 assertion 품질과 결함 탐지 능력을 충분히 반영하지 못합니다. 본 연구는 단일 LLM 환경에서 변이 테스팅 피드백을 반복적으로 활용하여 assertion 품질을 강화하는 테스트 생성 프레임워크를 제안합니다. 제안 방법은 생존 뮤턴트를 타깃으로 한 프롬프트와 변이 점수 및 라인·브랜치 커버리지를 함께 고려하는 뮤턴트-가드 손실, 그리고 비감소 수용 정책을 결합하여 품질 퇴행을 방지합니다. TESTEVAL의 210개 Python 프로그램에서 CodeQwen1.5-7B로 평가한 결과, 평균 변이 점수는 71.8%로 기본 설정 대비 65.0%p 향상되었고, 커버리지는 라인/브랜치 89.0%/88.9% 수준을 유지했습니다. 또한 Mutation@1은 54.4%, Mutation@5는 69.2%로, 단일 LLM 기반 테스트의 assertion 품질과 결함 탐지 능력을 효과적으로 개선함을 보였습니다.

1. 서 론

대형언어모델(LLM)은 단위 테스트 코드의 구문적 완성도는 높게 생성할 수 있으나, 실행 결과의 의미 차이를 판별하는 강한 assertion을 자동으로 구성하는 데는 여전히 어려움이 있다. 그 결과, 커버리지가 높더라도 결함 탐지 능력이 낮게 측정되는 사례가 발생한다.

본 연구는 단일 LLM 설정에서 변이 테스팅(mutation testing) 결과를 반복적으로 활용하여 assertion 품질을 강화하는 프레임워크를 제안한다. 제안 기법은 생존 뮤턴트를 목표로 하는 프롬프트 구성과, 커버리지 및 변이 점수의 퇴행을 허용하지 않는 수용 정책을 결합하여 비용 증가를 제한하면서 결함 구분력을 높인다.

TESTEVAL 210개 Python 프로그램 실험에서 제안 방법은 평균 변이 점수 71.8%를 달성했고, 라인/브랜치 커버리지는 89.0%/88.9%로 유지하였다.

TESTEVAL 210개 Python 프로그램에서 평가한 결과, 제안 방법은 평균 변이 점수 71.8%를 달성하면서 라인/브랜치 커버리지도 89.0%/88.9%로 유지하였다.

2. 제안 기법

제안 프레임워크는 단일 LLM을 반복 호출하며, 각 이터레이션에서 (i) pytest-cov로 미커버 브랜치를 수집하고 (ii) mutmut으로 생존 뮤턴트를 요약한 뒤, 이를 프롬프트에 반영해 다음 테스트를 생성한다. 새 테스트는 라인/브랜치 커버리지와 변이 점수가 모두 감소하지 않을 때만 누적하여 품질 퇴행을 방지한다.

프롬프트 구성은 이터레이션 상태에 따라 동적으로 전환한다. 첫 이터레이션에서는 기본 및 경계값 입력을 포함한 Initial 프롬프트로 탐색을 시작하고, 이후에는 생존 뮤턴트가 존재하면 Assertion-Mutant 프롬프트를 우선 적용하여 특정 뮤턴트를 kill하도록 유도한다. 생존 뮤턴트가 없고 미커버 브랜치가 남아 있는 경우에는 Branch-Target 프롬프트로 분기 조건을 만족하는 입력을 설계하게 하며, 두 조건이 모두 충족되지 않으면 Diversity 프롬프트로 입력 분포를 확장해 정체 구간을 완화한다.

Assertion-Mutant 프롬프트에는 변이 위치, 원본/변이 코드 조각, 변이 연산자 유형(AOR/ROR/COR/COI)을 요약해 포함한다. 특히 ROR의 경우 경계값에서 비교 결과가 뒤집히는 입력을, AOR의 경우 연산 결과 차이가 크게 드러나는 입력을 우선 제안하도록 지시하여 약한 검증(assert result is not None 등)을 기대값 비교나 불변식 검증으로 전환한다.

품질 균형을 위해 변이 점수(M), 라인 커버리지(L), 브랜치 커버리지(B)의 가중합으로 뮤턴트-가드 손실을 정의하고, 로그로 추적한다. 또한 수용 정책은 B, L, M이 모두 감소하지 않아야 하며(비감소 제약), 세 지표 중 하나 이상이 반드시 개선되어야 한다(엄격한 개선). 연속 무개선이 일정 횟수 발생하면 조기 종료하여 변이 테스팅 비용을 제한한다.

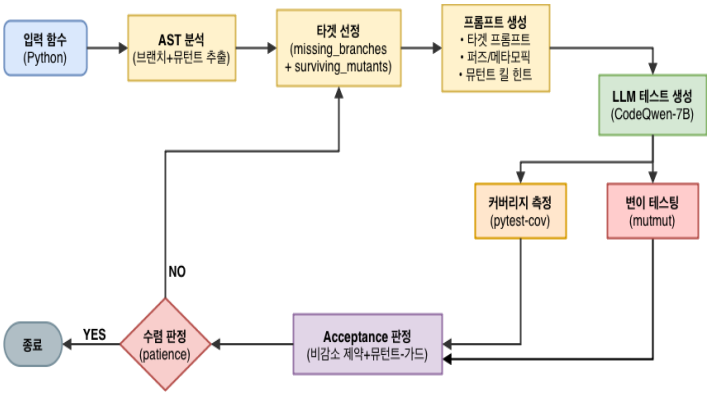


그림1. 제안한 방법의 전반적인 흐름도

* 이 논문은 한경국립대학교 국립대학육성사업(2025)지원을 받아 작성되었음
† 교신저자(Corresponding Author)

3. 실험 및 결과 요약

표 2. 제안 방법과 베이스라인 성능 비교

평가 지표	CodeQwen1.5-7B†	Kim et al. ‡	제안 방법
Syntax Correct	100%	100%	96.19%
Exec. Correct	84.26%	99.63%	96.19%
Avg. Line Cov.	90.7%	91.1%	89.0%
Avg. Branch Cov.	86.9%	86.2%	88.9%
Mutation@1	12.5%	43.6%	54.4%
Mutation@2	8.4%	37.2%	65.6%
Mutation@5	7.9%	30.4%	69.2%
Avg. Mutation	6.8%	26.3%	71.8%

† TESTEVAL 벤치마크 [1]에서 보고된 결과. 변이 점수는 본 연구에서 CodeQwen1.5-7B에 대해 측정함.

‡ Kim et al. [5]의 CodeQwen1.5-7B 기반 커버리지 최적화 방법. 변이 점수는 본 연구에서 측정함.

3.1 실험 설정

TESTEVAL 벤치마크의 210개 Python 프로그램을 대상으로 CodeQwen1.5-7B [4]를 사용하였다. 각 태스크는 최대 20회 이터레이션을 수행하되, 연속 3회 무개선 시 조기 종료한다. 커버리지는 pytest-cov, 변이 점수는 mutmut으로 측정하였다.

벤치마크는 단일 함수 형태로 제공되며, 다양한 제어 흐름과 예외 경로를 포함한다. 각 태스크에 대해 생성된 테스트 스위트는 누적 방식으로 관리하고, 이터레이션마다 새로 생성된 테스트를 기존 스위트에 합쳐 실행한다.

모델 생성 조건은 재현성을 위해 고정하였다(temperature 0.4, max_tokens 2048). 변이 테스팅 비용을 제한하기 위해 함수당 생성 뮤턴트 수를 상한으로 두고, 뮤턴트별 타임아웃을 적용하였다. 실행 실패 테스트는 즉시 누적에서 제외하며, 동일 태스크에서 연속 무개선이 발생하면 종료한다.

3.2 평가지표 및 비교 방법

표 2에서 제안 방법은 단순 생성(CodeQwen1.5-7B) 및 커버리지 최적화 기반 방법(Kim et al. [5]) 대비 변이 점수를 크게 향상시키면서도 커버리지를 유지한다. 이는 생존 뮤턴트 기반 타겟 프롬프팅이 약한 assertion을 강한 검증으로 전환하는 데 효과적임을 보여준다.

Line/Branch 커버리지는 테스트가 경로를 얼마나 폭넓게 탐색했는지를 나타내며, Mutation score는 생성된 assertion이 의미 차이를 구분하도록 작성되었는지를 실행 기반으로 근사한다. 본 연구는 세 지표를 함께 보고하여 '경로 탐색'과 '결함 구분력'을 분리해 해석한다.

비교는 (i) 단순 생성(CodeQwen1.5-7B), (ii) 커버리지 최적화 기반(Kim et al. [5])과 수행하며, 제안 방법은 동일한 이터레이션 예산과 생성 하이퍼파라미터를 적용한다. 또한 변이 점수는 동일 도구 체인으로 재측정하여 설정 차이에 따른 편향을 최소화한다.

3.3 실험 결과 및 분석

커버리지 유지(라인 89.0%, 브랜치 88.9%)는 변이 점수 최적화가 경로 탐색을 억제하지 않음을 보여준다. 평균 7.2회의 이터레이션으로 수렴했으나, mutmut 실행 비용과 동치 뮤턴트의 영향은 한계로 남는다.

Avg. Mutation은 71.8%로 단순 생성(CodeQwen1.5-7B)(6.8%) 대비 65.0%p, 커버리지 최적화(26.3%) 대비 45.5%p 개선되었다.

개선은 생존 뮤턴트 기반 타겟 프롬프팅의 효과에 기인한다. 예를 들어 관계 연산자 변이(ROR)에서는 경계값 입력을 유도해 비교 결과가 뒤집히는 케이스를 만들고, 산술 연산자 변이(AOR)에서는 기대값 비교 또는 불변식 검증을 요구해 'not None' 형태의 약한 검증을 제거한다. 이 과정에서 비감소 수용 정책이 품질 퇴행을 차단하여, 탐색 단계에서 확보한 커버리지를 유지한 채 assertion만 강화되는 방향으로 누적이 진행된다.

효율 측면에서 평균 7.2회의 이터레이션으로 수렴했으며, 조기 종료는 비용을 제한하는 데 기여한다. 다만 mutmut 실행 자체가 비용이 큰 편이며, 동치 뮤턴트 존재로 인해 변이 점수가 과소 추정될 수 있다는 한계가 남는다.

4. 결론

변이 테스팅 피드백과 비감소 수용 정책을 결합한 단일 LLM 테스트 생성 프레임워크를 제안하였다.

향후에는 (i) 생존 뮤턴트의 우선순위화 및 선택적 변이 테스팅으로 비용을 줄이고, (ii) 연산자 유형별로 효과가 큰 프롬프트 템플릿을 자동 선택하며, (iii) 언어/프레임워크 확장 및 더 다양한 오라클 패턴(예외 규약, 메타모픽 관계)을 통합하는 방향으로 확장할 수 있다.

참 고 문 헌

[1] W. Wang et al., "TESTEVAL: Benchmarking large language models for test case generation," in Findings Assoc. Comput. Linguist.: NAACL 2025, 2025, pp. 3547–3562. doi: 10.18653/v1/2025.findings-naacl.197.

[2] Google DeepMind, "Gemma: Open models based on Gemini research and technology," arXiv preprint arXiv:2403.08295, 2024. doi: 10.48550/arXiv.2403.08295.

[3] B. Rozière et al., "Code Llama: Open foundation models for code," arXiv preprint arXiv:2308.12950, 2023. doi: 10.48550/arXiv.2308.12950.

[4] Qwen Team, "CodeQwen1.5: A large language model for code," Alibaba Cloud, Tech. Rep., 2024. [Online]. Available: https://github.com/QwenLM/CodeQwen1.5. Accessed: Jan. 6, 2026.

[5] Y. Kim and G. Yang, "Single-model based coverage optimization test generation: Combining branch-first/line-guard and fuzz/metamorphic testing," in Proc. KIISE Conf., 2025. (in Korean)

질의-응답 프롬프트 기반 실용적인 테스트 오라클 생성

정지나 [1*], 김윤호 [2]

한양대학교 컴퓨터·소프트웨어학과(미래자동차-SW 융합전공)[1], 한양대학교 컴퓨터·소프트웨어학과 [2]

snowgina00@hanyang.ac.kr, yunhokim@hanyang.ac.kr

ANTLION: Practical Test Oracle Generation via Multi-turn LLM Compact Prompting

Gina Jung[1*], Yunho Kim[2]

Dept. of Computer and Software (Automotive-Computer Convergence), Hanyang University[1]

Dept. of Computer and Software, Hanyang University[2]

요 약

본 연구는 단위 테스트에서 버그 탐지의 핵심인 테스트 오라클을 대형 언어 모델(LLM)로 자동 생성하는 문제를 다룬다. 기존 기법에도 오라클 생성은 여전히 어렵고, 미세조정 기반 접근은 비용이 크며 MUT 포함 방식은 결함을 의도된 동작으로 오인할 위험이 있다. 이에 스무고개에서 착안한 대화형 추론 방식 ANTLION을 제안하며, 다중 회차 질의-응답으로 필요한 정보만 선택적으로 보강해 간결한 프롬프트를 유지한다. Defects4J에서 ANTLION은 기존 기법 대비 4%~36% 더 많은 버그를 재현했고, 질의 패턴 분석을 통해 LLM이 오라클 생성에 요구하는 정보도 확인했다.

Abstract

This paper investigates automatic unit-test oracle generation with large language models (LLMs). Oracle generation remains challenging, while fine-tuning is costly and including the method under test (MUT) can misinterpret defects as intended behavior. We propose ANTLION, a "Twenty Questions"-inspired multi-turn dialogue framework that keeps prompts concise by selectively acquiring only necessary information without fine-tuning. On Defects4J, ANTLION reproduces 4%–36% more bugs than prior methods, and our query-pattern analysis reveals what auxiliary information LLMs tend to request.

1. 서론

21세기에 소프트웨어는 현대인의 삶의 없어서는 안 될 존재가 되었다. 이런 소프트웨어에서 아주 사소한 버그일지라도, 대규모 시스템 장애, 보안 침해를 유발할 수 있다[1-3]. 따라서 소프트웨어 테스팅을 통한 버그 탐지를 통해 소프트웨어 신뢰성을 확보하는 것은 매우 중요하다[4-7]. 소프트웨어 테스팅에서 '테스트 오라클'은 소프트웨어 버그를 조기 탐지하는 핵심적 역할을 한다.

일반적으로 단위테스트에서는 각 프로그램의 특정 단위를 실행하는 '테스트 접두사'와 실행되는 동작이 올바른 동작과 일치하는지 검증하는 '테스트 오라클'로 구성되어 있다[8]. 테스트를 수동으로 작성하는 과정은 프로그램 규모가 커질수록 시간과 비용이 급격히 증가하며, 개발자의 도메인 지식과 경험에 크게 의존하여 테스트 품질이 불안정 할 수 있다[9]. 테스트 접두사에 대한 자동화 문제는 기존 연구자들의 노력[13-14]을 통해서 크게 발전하였으며 실제 실무에 효율적으로 반영되었다. 하지만, 효과적인 버그 탐지를 위해서는 테스트 오라클이 정확하고 강력해야 한다[15]. 테스트 오라클을 자동화로 생성하는 문제는 많은 기존 연구자들의 노력[10-11, 16-17, 19-20, 27]에도 오래된 난제로 남아 있다.

연구자들은 코드 주석과 자연어 문서를 활용하여 자연어 처리(NLP) 및 패턴 매칭 기법을 적용하는 연구를 해왔다[16-17]. 인공지능(AI)의 발전으로 테스트 오라클 생성 문제를 해결하기 위한 다양한 자동화 기법이 제안된 바 있다. 테스트 대상 메서드(MUT)를 머신 러닝(ML)의 입력에 포함하여 테스트 오라클을 생성한다. 하지만, 이 방식은 테스트 오라클 품질에 부정 적인 영향을 미칠 수 있다[18]. 예를 들어, 테스트 대상 메서드 구현에 버그가 있는 경우 버그 동작을 의도된 올바른 구현으로 해석하여 통과하면 안 되는 테스트가 부적절하게 통과되는 결과를 초래할 수 있다. 이러한 접근 방식은 주로

회귀(regression) 테스트에서 유용한 오라클을 생성한다. 딥러닝(DL) 기반 기법으로는 테스트 대상 메서드(MUT)와 개발자가 작성한 테스트 케이스로부터 학습하는 트랜스포머 기반 모델이 활용되었다[10-11]. 최근에는 대형 언어 모델(LLM)의 등장으로 미세조정을 통해 코드와 테스트 문맥을 집중적으로 이해하면서 테스트 오라클 생성을 보다 일반적인 생성 문제로 다루는 연구가 활발 해지고 있다[19-20].

테스트 오라클 자동화 생성에 초점을 맞춘 연구임에도, 정확하고 강력한 테스트 오라클 자동 생성은 여전히 어려운 과제로 남아있다. 기존 연구는 미세조정의 과정에서 시간과 노력이 소모되며, 이 학습에서 사용된 데이터 품질이 생성 기법자체의 품질을 결정해 제한적인 적용범위를 가진다. 적절한 형식의 테스트 오라클 구문이 생성되더라도 높은 오탐율을 가지고 있어 일반화가 어렵다.

대형 언어 모델(LLM)은 방대한 분야에서 충분히 일반적인 지식으로 사전학습 되어있기 때문에, 범용적인 생성 능력을 이미 확보하였으므로, 응용 과정에서는 별도의 미세조정 없이 실용 적인 성능을 확보한다. 미세조정 없이 대형 언어 모델을 이용한다면, 데이터 셋 준비와 미세조정을 위한 시간과 노력이 절약되며, 다양한 범위에서 일반적인 성능으로 적용 가능성이 우수 하다는 장점을 가지고 있다. 프롬프트 문맥 길이가 늘어나 더 많은 정보를 한 번에 수용할 수 있지만, 한 번에 많은 정보가 담긴 프롬프트는 대형 언어 모델(LLM)의 집중을 흐려서 환각(hallucination)이 발생되어 나쁜 품질의 결과를 가져온다[22].

대형 언어 모델(LLM)의 장점을 테스트 오라클 생성 문제에 적용하면서도, 대형 언어 모델의 단점인 환각 현상을 줄이기 위한 프롬프트 방식으로 대화형 추론 방식인 스무고개에서 착안한 ANTLION을 제안한다. 간결한 프롬프트를 유지하며 언어 모델과 대화를 이어가면서 언어 모델이 테스트 오라클 추론에 필요하다고 요청되는 정보만 추가로 제공하여 프롬프트 엔지니어링을 했다. 기존에는 테스트 오라클 생성 문제를 '검색(Search)', '번역(Transform)'의 문제로 해결을 했지만 여기서는 '질의-응답(Q&A)'로 해결했다.

본 연구는 관련연구와 공정한 비교를 위해 Java의 실제 버그가 담긴 대규모 데이터 셋인 Defects4j [21]를 사용했다. 테스트 오라클 생성 기존 연구[11,19-20]보다 최소 4% 최대 36% 더 많은 버그를 재현을 이루었다. 대형 언어 모델(LLM)과 비교 실험을 통해서 프롬프트 엔지니어링이 효과적이 였음을 확인하였으며, 스무고개 방식의 회차별 질문 요청 대한 분석을 통해서 효과적인 프롬프트 엔지니어링이었음을 입증했다.

2. 배경지식 및 선행연구

2.1. 대형 언어 모델

대형 언어 모델(Large Language Model, LLM)은 대규모 말뭉치(자연어 텍스트, 코드, 수식 등)로부터 토큰(token) 단위의 확률 분포 학습하여, 주어진 입력(프롬프트) 뒤에 이어질 다음 토큰을 예측하는 방식으로 동작한다. 최근 주류 LLM은 트랜스포머(Transformer) 계열로 구현되며, 입력 시퀀스에 대한 자기-어텐션(self-attention)을 이용해 장거리 의존성을 모델링 한다. 대형 언어 모델은 일반적으로 사전 학습(pre-training)단계에서 대규모 데이터에 대해 자기지도학습을 수행하고, 미세 조정(fine-tuning)단계에서 특정 도메인/작업에 대한 추가 학습을 수행한다.

이러한 언어 모델을 '학습 가능한 파라미터 수'로 규모를 대표할 수 있다.

- 소형(small): ~0.3B(3억 미만 파라미터)
  예) CodeGPT(110M), CodeParrot(110M)
- 중형(Mid): 0.3B~7B(3억 ~70억 파라미터)
  예) Phi-1(1.3B), PolyCoder, CodeGemma-2B
- 대형(Large): 7B~(70억 이상 파라미터)
  예) Code Llama, CodeGemma-7B

오픈 소스 기반의 대형 언어 모델은 파라미터가 명확하여 규모를 구분할 수 있지만, 상업용 대형 언어 모델(API)은 파라미터가 비공개인 경우가 많아 직접적인 비교에는 한계가 있다. 보조적인 비교 지표로 컨텍스트 윈도우의 크기와 최대 출력 토큰 등으로 스펙을 비교할 수 있다. 예를 들어, 오픈소스 언어모델 중 대형으로 분류되는 Code Llama는 기본 16K 토큰 길이로 학습되었고 최대 100K 토큰 컨텍스트까지 안정적 생성이 가능하다고 보고되었다[23]. 하지만 상업용 대형 언어 모델에서 Open Ai의 GPT 계열 중 GPT-5.1의 경우 컨텍스트 윈도우의 크기는 400K이며, 128K 최대 출력을 지원한다[24].

상업용 언어 모델은 일반적으로 모델 가중치에 대한 직접 접근이 제한되어 오픈 소스 언어 모델처럼 사용자가 임의로 자유롭게 재학습하거나 커스텀 미세조정을 수행하기는 한계가 있다. 대신 검색 기반 생성(RAG), 도구 호출 및 컨텍스트 구성 전략(예: few-shot, CoT)을 통해 특정 작업에 대한 적합성을 향상시키는 방식이 실무적으로 널리 활용된다.

2.2. 테스트 오라클 생성

테스트 오라클은 테스트의 기대 결과를 정의하는 핵심 요소이지만, 실제 개발에서는 사람이 수동으로 작성해야 하는 경우가 많아 비용이 크고 누락이 발생하기 쉽다. 이에 따라 최근 소프트웨어 테스트

그림 1. ANTLION 질의-응답 개요[1]

연구에서는 자연어·코드·문서 등 개발 산출물을 활용하여 오라클 또는 오라클에 준하는 명세를 자동 생성하려는 시도가 활발히 이루어지고 있다.

오라클 생성 자체를 직접 목표로 한 신경망 기반 연구로는 TOGA[11]가 대표적이다. 테스트 대상 메서드의 코드 문맥을 이용하여 오라클을 추론하는 트랜스 포머(Transfomer) 기반 방법을 제안하였고, 자동 테스트 생성[13]과 결합했을 때 추가 결함 탐지 성과를 보고하였다. 다만 TOGA[11]를 포함한 기존 학습 기반 방식은 생성된 오라클의 강도(strength) 부족, 구현 결함을 그대로 따라가 잘못된 오라클을 생성할 가능성, 그리고 미지의 코드에 대한 일반화 등에서 한계가 지적되어 왔다.

이러한 흐름은 최근 코드 특화 대형언어모델(Code LLM)의 등장으로 새로운 전환점을 맞이하였다. 여러 코드 LLM을 대상으로 미세조정 및 프롬프팅 변형을 결합해 테스트 오라클을 생성하는 TOGLL[20]을 제시하고, 기존 방법(EvoSuite[13], TOGA[11]) 대비 더 정확하고 강한 오라클을 생성하며 실제 버그 탐지에서도 추가적인 성과를 낼 수 있음을 대규모 실험으로 보였다. 특히 LLM 기반 오라클 생성은 입력 컨텍스트의 구성 방식에 따라 결과가 크게 달라질 수 있는데, Doc2OracLL[19]은 Javadoc과 같은 문서가 메서드의 기대 동작과 예외 조건을 구조적으로 포함한다는 점에 주목하여, 코드만 기반으로 할 때 모델이 결함이 있는 구현을 모사하며 잘못된 오라클을 생성할 수 있는 반면 문서 기반 입력은 의도된 동작을 반영한 오라클 생성을 유도할 수 있음을 논의하고, 문서의 존재 여부가 성능 평가 결과에 미치는 영향을 체계적으로 제시한다. 한편 오라클의 핵심 구성 요소인 단언문 생성의 품질을 높이기 위해, LLM 기반 자동 단언문 생성 가능성을 체계적으로 탐구하며 EASE[27]를 활용한 성능 향상 구성을 제안하였다. 공개된 재현 자료에 따르면 EASE[27]는 유사 테스트-단언문 쌍을 정보검색(IR) 방식으로 찾아 모델 입력으로 제공하는 방식을 통해 LLM이 보다 적절한 단언문을 생성하도록 돕는다.

3. 연구 방법론

3.1절에서는 스무고개 방식의 질의-응답 프롬프트 기법의 동기를 설명한다. 그림 1은 본 기법의 접근 방식의 개요를 보여 준다. 좌측의 입력 컨텍스트 정보가 담긴 사용자의 문제 프롬프트를 통해서 스무고개식 ANTLION의 프롬프트의 시작을 보여준다. 3.2 절에서는 ANTLION이 테스트 오라클을 생성하는 방식을 설명한다. 3.3 절에서는 테스트 오라클 생성 과정에서 사용된 질문목록을 구체적으로 설명한다.

3.1. 동기

대화형 추론 게임인 스무고개는 전세계적으로 오래전부터 남녀노소 즐겨온 놀이로 알려져 있다. 참가자 한 명이 어떤 대상을 마음속으로 정하면, 다른 참가자가 최대 질문 횟수가 정해져 있는 질의-응답 과정을 통해서 그 대상을 추론하는 구술 놀이이다[29].

[1] 본 그림과 3.3절의 질문 구성의 예시는 설명의 편의를 위해 한글로 재구성했으며, 실험은 영어 프롬프트로 수행하였다.

질문자는 제한된 횟수 안에서 정보량이 큰 질문을 선택해 후보를 단계적으로 좁히며, 답변자는 질문에 따라서 단서를 제공한다. 이 과정은 '질문 설계 – 정보 획득 – 가설 갱신 – 탐색 공간 축소'라는 반복적 추론 절차를 직관적으로 보여준다.

일반적 대화 성능이 훌륭한 대형 언어 모델(LLM)을 테스트 오라클 생성 기법에 적용하면서 추론의 불안정성을 완화하기 위한 프롬프트 엔지니어링 관점에서 스무고개 방식 질의 전략을 채택했다. 스무고개 방식은 최소한의 문맥으로도 후보 공간을 체계적으로 축소하도록 유도하므로, 간결한 프롬프트를 유지하면서도 일관된 추론을 도울 수 있다. 또한 불필요한 서술 확장을 억제함으로써 LLM의 환각(hallucination)으로 인한 오류 가능성을 낮추는 데 기여한다.

3.2. ANTLION

그림 1은 본 연구에서 제안하는 ANTLION에서 테스트 오라클 생성 과정의 개요를 보여준다. ANTLION은 테스트 대상 메서드와 테스트 접두 코드를 입력으로 받아, 다중 회차[28] 질의-응답을 통한 단계적 추론을 통해 테스트 오라클을 생성하는 프레임 워크이다.

구체적으로 ANTLION은 (1) 입력 컨텍스트로 테스트 대상 코드, 오라클 위치를 표시하는 플레이스홀더"<AssertPlaceHolder>"를 포함한 테스트 접두 코드, 그리고 사전에 정의된 질문 목록을 구성하여 언어 모델에게 제공함으로써 테스트 오라클 생성 과정을 시작한다. 이후 (2) ANTLION은 스무고개 방식의 질의 전략을 적용하여, 질문에 포함된 질의를 필요에 따라 선택적으로 사용하면서 추가정보를 획득한다. 본 연구에서는 총 5개의 질문(대상 메서드 의도, 테스트 접두 코드 의도, 제어 흐름 정보, 오라클 후보, 문서화 정보)을 제공한다. 5가지 질문은 테스트 생성, 테스트 오라클 생성 연구 분야에서 오라클 생성에 유용하다고 알려진 정보를 포괄하도록 설계했다. 각 질문에 대해서 구현상 사용자(User)가 사전에 정의된 자연어 답변을 제공하며, 이는 최종 정답으로 도출할 오라클 생성을 위한 근거 정보로 활용한다. 마지막으로 (3) 모델은 주어진 대화 문맥과 질의-응답으로 획득한 정보를 종합적으로 추론하여 플레이스 홀더를 치환할 구체적인 테스트 오라클 구문을 출력한다. 이때 모델은 질문 목록의 모든 항목을 반드시 사용해야 하는 것은 아니며, 문제의 난이도나 맥락의 충분성에 따라서 일부 질문만 사용하거나, 질문을 사용하지 않을 수 있다. 이를 통해 ANTLION은 대화의 초점을 잃지 않고 불필요한 질의를 줄여 추론 비용을 완화하면서도, 필요한 경우에 질문 목록을 선택적으로 사용을 통해 오라클 생성에 필요한 정보를 보강하여 정확한 오라클을 도출하는 것을 목표로 한다.

3.3. 질문 구성

ANTLION은 다중 회차 질의-응답 과정에서 사용할 수 있는 사전 정의 질문 목록을 제공한다. 본 절에서는 해당 질문들의 목적과 포함 정보의 범위를 설명한다. 각 질문은 선행연구에서 오라클 생성에 유용하다고 알려진 보조 정보를 체계적으로 활용하기 위해 설계되었다. ANTLION은 각 문제에 대해 모든 질문을 강제하지 않고, 입력 맥락의 충분성 및 불확실성에 따라 질문을 선택적으로 사용함으로써 정보 획득과 추론 비용 간 균형을 달성한다.

3.3.1. 테스트 대상 메서드의 의도

제공 정보. 테스트 대상 메서드(MUT)의 동작을 자연어로 요약한 설명을 제공한다. 구체적으로 메서드의 상위 책임(무엇을 하는지), 반환값 또는 부작용의 의미, 결과에 영향을 주는 주요 조건(분기, 예외 발생 조건 등)을 행동 수준에서 정리한다.

유용한 경우. 메서드가 길거나 도메인 특화 로직을 포함해 코드만으로 의도를 빠르게 파악하기 어려운 경우 유용하다. 특히 다수의 분기/조기 반환/예외 처리가 포함된 메서드에서 "어떤 조건에서 어떤 결과가 나오는지"를 빠르게 정리하는 데 효과적이다.

제공하지 않는 정보/한계. 정답 오라클을 직접 제시하지 않으며, 코드에 존재하지 않는 새로운 가정을 추가하지 않는다. 또한 모든 엣지 케이스를 완전하게 열거하는 것을 보장하지 않는다.

장점. 오라클 생성에서 필요한 "검증 대상의 의미"를 명확히 하여, 반환값/예외/부작용 중 무엇을 목표로 삼아야 하는지 결정하는 데 기여한다. 결과적으로 잘못된 검증 유형 선택을 줄인다.

예시.

- 테스트 대상 메서드:

```
public int add(int a, int b) {
    return a + b;
}
```

- 테스트 대상 메서드의 의도:

> 이 메서드는 두 정수의 합을 계산하도록 설계되었습니다. 두 정수를 입력 매개변수로 받아 산술 덧셈 결과를 반환합니다. 이 메서드는 부작용이 없으며 입력 값에만 기반하여 항상 확정적인 결과를 반환합니다. 조건 분기나 예외 처리가 없습니다.

3.3.2. 테스트 접두 코드의 의도

제공 정보. 테스트 접두 코드가 구성하는 시나리오를 자연어로 설명한다. 즉, 입력 데이터의 형태, 객체 초기화/설정, 선행 호출, 모의 객체 적용 여부 등 테스트가 어떤 전제 조건을 만들고 무엇을 검증하려는 지(값/예외/상태 변화)를 요약한다.

유용한 경우. 테스트 접두 코드에 복잡한 설정(예:

상태 변경, 입력 전처리)이 포함되어 기대 결과가 코드 설정에 강하게 의존하는 경우 유용하다. 또한 접두 코드가 단순 호출 이상으로 "의도된 테스트 목적(what-to-test)"을 암시하는 경우, 이를 분명히 하는 데 도움이 된다.

제공하지 않는 정보/한계. 테스트를 실행하여 실제 결과를 알려주지 않으며, 오라클의 정답 값을 직접 제공하지 않는다. 접두 코드가 기대 값을 충분히 규정하지 못하는 경우, 이 정보만으로 오라클이 단일하게 결정되지 않을 수 있다.

장점. '테스트 접두 코드의 의도'를 명시화 함으로써, 테스트 대상 메서드의 일반 동작과 테스트가 요구하는 특정 맥락을 연결한다. 이를 통해 모델이 불필요한 가정으로 확장하지 않고, 테스트가 겨냥한 조건과 결과에 집중하도록 유도한다.

예시.

- 테스트 대상 메서드:

```
public int add(int a, int b) {
    return a + b;
}
```

- 테스트 접두 코드:

```
int result = add(2, 3);
```

- 테스트 대상 메서드의 의도:

```
이 테스트 접두사는 add 메서드가 두 개의 구체적인 정수 인수 2와 3으로 호출되는 간단한 시나리오를 설정합니다. 이 접두사의 목적은 메서드가 이러한 입력의 합을 올바르게 계산하는지 확인하는 것입니다. 이 테스트는 작고 고정된 값을 사용하여 메서드의 기본 산술 동작을 검증하고 반환 결과가 예상되는 덧셈 결과를 반영하는지 확인하는 것을 목표로 합니다.
```

3.3.3. 테스트 대상 메서드의 제어 흐름 정보

제공 정보. 테스트 대상 메서드(MUT)의 제어 흐름 그래프(CFG)를 제공한다. 일반적으로 정규화된 메서드 표현과 함께, 분기/루프/예외 처리, 조기 반환 등 가능한 실행 경로를 그래프 구조로 나타낸다.

유용한 경우. 오라클이 특정 분기 경로에 종속되는 경우, 혹은 예외 처리/조기 반환이 많아 '실제로 어떤 경로가 실행되는지'를 확인해야 하는 경우 유용하다. 또한 테스트 접두 코드가 유도하는 경로를 추론해야 하는 상황에서, 경로 누락으로 인한 오라클 오류를 줄인다.

제공하지 않는 정보/한계. CFG는 경로 구조를 제공하지만, 각 경로가 어떤 의미를 갖는지에 대한 해석을 자동으로 제공하지 않는다. 또한 경로의 도달 가능성은 입력 조건과 상태에 따라 달라지므로, CFG만으로 특정 경로가 반드시 실행된다고 결론 내릴 수는 없다.

장점. 가능한 실행 경로를 체계적으로 열거하고, 테스트 접두 코드가 어떤 경로를 유도하는지 판단하도록 돕는다. 이를 통해 "잘못된 경로를 가정해 생성한 오라클"을 줄이고, 분기 기반 로직에서의 정확도를 높인다.

예시.

- 테스트 대상 메서드:

```
public int add(int a, int b) {
    return a + b;
}
```

- 제어 흐름 그래프:

```
digraph CFG {
  n0 [label=" 진입"];
  n1 [label=" 종료"];
  n2 [label="return a + b"];
  n0 -> n2;
  n2 -> n1;
}
```

3.3.4. 오라클 후보(top-k)

제공 정보. 현재 문맥에서 가능성이 높다고 판단되는 JUnit 4 오라클 assertion 후보(top-k)를 순위 형태로 제시한다. 후보는 단일 assertion 또는 소수의 assertion 조합으로 제공되며, 정적 import 환경을 가정한 표준 JUnit 4 형식을 따른다.

유용한 경우. 오라클 공간이 넓거나 형식이 까다로운 경우(예: 부동 소수 오차 허용치, 예외 검증 패턴, 문자열/컬렉션 비교 등) 유용하다. 또한 모델이 여러 가능성 중 하나를 선택해야 하는 상황에서 후보 비교를 통해 최종 출력을 안정화하는 데 도움이 된다.

제공하지 않는 정보/한계. 후보는 제안일 뿐 정답을 보장하지 않으며, 정답이 후보 집합에 포함되지 않을 수 있다. 따라서 후보는 "검토 대상"으로 사용되어야 하며, 맥락에 비추어 비판적으로 선택해야 한다.

장점. 오라클의 표현 형식에서 발생하는 실수를 줄이고, 모델이 최종 결정을 내릴 때 탐색 비용을 줄이는 역할을 한다.

예시.

- 테스트 대상 메서드:

```
public int add(int a, int b) {
    return a + b;
}
```

- 테스트 접두 코드:

```
int result = add(2, 3);
```

- 테스트 오라클 후보(top-k):

```
assertEquals(5, result);
assertEquals(4, result);
assertEquals(6, result);
```

3.3.5. 테스트 대상 메서드의 문서화 정보

제공 정보. 테스트 대상 메서드(MUT)의 문서화 정보(예: Javadoc 스타일 docstring)를 제공한다. 메서드의 책임, 파라미터 의미, 반환값 의미(또는 부작용), 코드로부터 명확히 유도되는 제약/조건(예: 예외, null 처리, 경계 조건)을 "계약(contract)"에 가까운 형태로 요약한다.

유용한 경우. 코드가 복잡하거나 구현이 의도를 드러내지 않는 경우, 또는 메서드가 라이브러리/프레임워크의 규약을 따르는 경우 유용하다. 문서에 명시된 규약(예: 특정 입력에서의 예외, 반환의 의미)이 오라클 결정에 직접적인 단서를 제공할 수 있다.

제공하지 않는 정보/한계. 코드에 근거하지 않은 가정을 추가하지 않으며, 문서가 실제 구현과 불일치할 가능성은 존재한다. 또한 문서 역시 정답 오라클을 직접 제공하지 않는다.

장점. 메서드의 '의도된 계약'을 명시 화하여 오라클이 무엇을 검증해야 하는지를 정리하는 데 도움을 준다.

예시.

- 테스트 대상 메서드:

```
public int add(int a, int b) {
    return a + b;
}
```

- 테스트 대상 메서드의 문서화 정보(Javadoc):

```
/**
* 두 정수의 합을 계산합니다.
* <p>이 메서드는 두 정수를 입력 받아 산술 덧셈
결과를 반환합니다.
* 부작용이 없으며 결과는 입력된 인수에만
의존합니다.</p>
*
* @param a 첫 번째 정수 덧셈 수
* @param b 두 번째 정수 덧셈 수
* @return {@code a}와 {@code b}의 합
*/
```

4. 실험 설정

4.1. 연구 질문

RQ1. 본 기법 ANTLION은 얼마나 정확한 테스트 오라클을 생성할 수 있는가?

RQ2. 본 기법 ANTLION은 GPT와 비교했을 때 프롬프트 엔지니어링의 효과가 있는가?

RQ3. 본 기법 ANTLION은 질의-응답 과정에서 얼마나, 어떤 질문을 선호하며, 질문이 정확도에 어떤 영향을 미치는가?

RQ4. 본 기법 ANTLION은 얼마나 다양한 오라클 구문을 생성할 수 있는가?

RQ5. 본 기법 ANTLION은 테스트 오라클을 생성하는데 비용(토큰) 소모량은 어떠 한가?

4.2. 벤치 마크

본 연구 ANTLION를 수행하기 위해, 실제 854개의 Java 버그로 구성된 벤치마크 데이터셋인 Defects4J[21]를 활용한다. 비교 기준과 공정한 비교를 위해 TOGA 저장소[25]에 공개된 동일한 입력 샘플을 사용한다. 374개의 입력 샘플로 구성되며, 각 샘플은 테스트 대상 메서드(MUT), 테스트 접두 코드, Javadoc 문서 쌍으로 구성되어있다. 이 374개의 입력샘플은 Defects4J의 11개 프로젝트와 120개의 고유 버그를 포함한다.

입력 샘플은 테스트 오라클 구문을 포함하며, 해당 테스트 오라클 구문을 플레이스홀더 "<AssertionPlaceHolder>"로 치환하여 ANTLION의 실험을 진행했다.

4.3. 비교 기준

Defects4J[21]의 374개 입력 샘플을 이용한 최신 연구를 비교 기준 선으로 삼았다.

- **TOGA[11]:** 트랜스포머 기반의 신경만 테스트 오라클 생성 기법으로, 가능한 오라클 후보 집합을 생성하고 각 후보의 점수를 순위와 하여 가장 높은 점수를 최종 오라클로 출력하는 연구이다.
- **TOGLL[20]:** 대형 언어 모델을 활용해 정확하고 강한 테스트 오라클을 생성하는 방법으로, 미세조정 및 프롬프트 설계를 통해 TOGA 대비 성능을 비교 보고한 최신 연구이다.
- **Doc2OracLL[19]:** 테스트 오라클 생성에서 문서(Javadoc)이 미치는 영향을 분석하고, 문서 정보를 포함해 오라클을 생성하는 접근을 다룬 최신 연구이다. 프롬프트 조합 중 가장 많은 버그를 탐색한 프롬프트(P6)와 모델(CodeLlama-7B)을 기준선으로 삼았다.
- **GPT-5.1[24]:** 상업용(폐쇠형) 대형 언어 모델이 다중회차 질의-응답 프롬프트 없이 직접 오라클 생성을 진행한다.

4.4. 평가 지표

Defects4J[26] 저장소에는 각 고유 버그 마다 Buggy 버전과 Fixed 버전을 쌍으로 제공한다. 실험에서는 Buggy 버전에서 실패(Fail) 하고 Fixed 버전에서 성공(Pass)한 테스트 오라클을 정확하다 정의하며, 버그를 재현했다고 평가한다. 이는 비교 기준 선과 동일한 판정 기준을 따른다.

4.5. 구현

ANTLION의 실험 파이프라인은 Python 3.12.10 기반으로 구현하였으며, 전체 구현 규모는 1,312 LOC(lines of code)이다. 실험에는 OpenAI의 최신 배포 모델인 GPT-5.1[24]을 사용하였고, 생성 설정은 maximum output tokens 512, temperature 0.5, top_p 1 로 고정하였다. 이 외에는 open ai API 기본값으로 수행하였다. 테스트 실행 및 결함 검증은 Defects4J[21] 벤치마크를 통해 수행하였으며, 테스트 프레임워크로는 JUnit 4를 사용하였다.

질의-응답 과정에서 사용자 답변은 질문 1,2,4의 경우 gpt-5.1 모델에서 temperature 0.0 설정으로 사전에 생성해 두었으며, 질문 3의 경우 라이브러리를 이용하여 그래프 생성 함수를 이용하였다.

4.6. 타당성 위협 요소

LLM 생성 과정의 무작위성(temperature 등)이 결과 변동을 유발할 수 있으므로, 선행연구(TOGLL[20], Doc2OracLL[19])를 참고하되 이를 더 엄밀히 통제하기 위해 동일 설정에서 총 5회 반복 실험을 수행했다. Defects4J[21]의 Buggy/Fixed 쌍에서 'Buggy에서 fail, Fixed에서 pass'를 정확한 오라클로 정의했지만, 이 기준이 결함의 본질을 정확히 기술하는 오라클과 완전히 일치하지는 않아 우연히 결함을 자극하는 취약 오라클이 포함되거나 전제 조건 차이로 패턴이 달라질 가능성이 있다. JUnit4 환경에 한정되어 다른 언어/프레임워크나 산업 코드로의 일반화에 제약이 있을 수 있고, 다중 회차 질의-응답 기반 생성은 프로젝트 관행(테스트 오라클 패턴), 코드/문서 구조, 도메인 특성에 따라 확보 정보의 질이 달라져 성능 편차가 생길 수 있으며, 상업용 API LLM의 모델 업데이트로 시점에 따라 결과가 달라질 수 있어 모델명과 디코딩 설정을 명시하고 당시 배포 모델을 고정해 재현성을 확보했다.

5. 실험 결과

5.1. 정확한 테스트 오라클 생성 평가

RQ1은 "ANTLION이 얼마나 정확한 테스트 오라클을 생성할 수 있는가?"를 평가하는 것이다. 이를 위해 널리 수용되는 현실 세계 Java 결함 벤치마크인 Defects4J를 사용하였다. Defects4J는 각 결함에 대해 buggy 버전과 fixed 버전을 제공하며, 본 연구에서는 Defects4J 관련 세부 구성과 실험 조건을 4.2절에서 제시하였다. 본 절에서는 ANTLION이 생성한 오라클이 실제 결함을 재현할 수 있는지 관점에서 성능을 비교한다.

표 1. Defects4J 벤치마크에서 버그 재현 수

방법론	고유 버그 수
TOGA[11]	57
TOGLL[20]	65
Doc2OracLL[19]	73
ANTLION (Ours)	76

표 2. 프롬프트 엔지니어링을 제거한 GPT와 비교

방법론	고유 버그 수
ChatGPT-5.1[24]	58
ANTLION (Ours)	76

표 1는 Defects4J 벤치마크에서 버그 재현 수(고유 버그 수)를 기준으로, ANTLION과 기존 방법들을 비교한 결과를 보여준다. 비교 대상은 테스트 오라클 생성 분야의 대표적 기존 연구인 TOGA[11], TOGLL[20], Doc2OracLL[19]이며, 각 방법의 수치는 해당 논문에서 보고된 결과를 기반으로 정리하였다.

표 1에서 확인할 수 있듯이 ANTLION은 Defects4J에서 총 76개의 고유 버그를 재현하였다. 이는 TOGA(56), TOGLL(65), Doc2OracLL(73)보다 높은 수치로, 동일한 벤치마크에서 더 많은 실제 결함을 재현할 수 있음을 시사한다. 또한 buggy/fixed 쌍의 동작 차이로 버그가 재현되었다는 점은, 생성된 오라클이 형식적 생성에 그치지 않고 결함을 드러내는 방향으로 작동했음을 보여준다.

5.2. 요소 제거 실험

RQ2는 "ANTLION이 GPT 단독 사용 대비 프롬프트 엔지니어링의 효과를 제공하는가?"를 검증하는 것이다. 이를 위해 ANTLION의 핵심 설계 요소(다중 회차 질의 전략 및 단계적 정보 보강)를 제거하고, 동일한 상용 LLM(GPT-5.1)을 프롬프트 엔지니어링을 제거한 기본 프롬프트로만 호출하는 비교 기준을 구성하였다. 비교의 공정성을 위해 기본 프롬프트 조건에서도 few-shot 예제와 JUnit 4 assertion 작성 지침은 동일하게 제공하고, 질의-응답 기반 정보 보강만 제거해 프롬프트 설계의 기여를 분리해 평가하였다.

표 2은 Defects4J에서의 고유 버그 수를 기준으로 GPT-5.1 기반 비교 기준과 ANTLION의 결과를 제시한다. GPT-5.1을 기본 프롬프트로 적용한 경우 58개의 고유 버그를 재현한 반면, ANTLION은 76개를 재현하였다. 이는 동일한 모델을 사용하더라도 ANTLION의 구조화된 입력 구성과 다중 회차 상호작용이 버그 재현 성과를 개선하며, 성능 향상이

모델 사용만으로 설명되기 어렵다는 점을 뒷받침한다.

5.3. 질의 분석

표 3는 ANTLION의 다중 회차 질의가 얼마나 자주 사용되는지 보여준다. 전체 입력 중 1회 이상 질문을 사용한 경우가 59.8%로 과반을 차지해, 초기 컨텍스트만으로 해결하기보다 질의-응답을 통해 필요한 단서를 보강한 뒤 오라클을 생성하는 사례가 많음을 확인할 수 있다. 질문 횟수 분포는 1회(31.7%)가 가장 크고, 2회(14.8%), 3회(9.6%), 4회(3.3%), 5회(0.1%)로 갈수록 급격히 줄어든다. 즉, 필요할 때는 질문을 활용하지만 대부분은 적은 회차로 정보를 효율적으로 확보한다.

표 4는 멀티 턴 과정에서 요청된 보조 정보의 종류를 요약한다. 가장 많이 호출된 것은 테스트 접두 코드 의도에 대한 자연어 요약(45.8%)이며, 이는 오라클 생성에서 빈번한 불확실성이 메서드 자체의 의미보다 현재 테스트가 구성한 시나리오의 검증 목표에 있음을 시사한다. 접두 코드는 입력 구성, 객체 초기화, 상태 설정 등으로 상황을 만들어내는데, 이 구성이 복잡할수록 코드만으로 무엇을 검증해야 하는지 불명확해질 수 있어 모델이 목표를 먼저 명료화하려는 경향이 나타난다. 그 다음으로는 JUnit 4 오라클 후보 제시(18.5%), 제어 흐름 정보(CFG)(16.7%), 테스트 대상 메서드 의도 요약(13.5%) 순으로 나타나, 목표를 파악한 뒤에도 assertion 형태 선택, 분기·예외로 인한 경로 불확실성, 기능적 요약이 추가로 필요한 경우가 적지 않음을 보여준다. 반면 Javadoc 기반 문서화 정보는 5.4%로 가장 낮아, 다른 단서들이 오라클 결정에 더 직접적으로 사용되었음을 시사한다.

표 5은 최대 5회 질문 허용 조건에서 실제로 N회 질문이 발생한 샘플을 모아, 질문 횟수별로 어떤 유형이 선택되는지 나타낸다. 이는 질문 횟수를 N으로 고정한 실험이 아니라, 생성 과정에서 추가 정보가 필요해 결과적으로 N회 질의가 수행된 경우의 패턴을 보여준다. 1회 질문에서는 테스트 접두 코드 의도가 67.8%로 가장 높아, 단 한 번의 보강으로 해결되는 사례의 핵심 요구가 검증 목표의 명시화임을 재확인한다. 2회에서는 접두 코드 의도가 높은 수준으로 유지되는 가운데 top-k가 늘어나, 목적을 정한 뒤 구체적인 assertion 형태를 정리하는 단계가 병목이 되는 경우를 보여준다. 3회로 늘어나면 CFG와 top-k 비중이 크게 높아져, 어떤 실행 경로를 전제로 오라클을 작성할지와 그에 맞는 표현을 함께 확정하려는 경향이 강화된다. 4회에서는 메서드 의도 요약의 비중이 크게 증가하고 접두 코드 의도, CFG, top-k도 모두 높은 수준을 보이며, 복잡한 입력에서는 메서드 행동 요약, 테스트 목표, 경로 구조, assertion 표현이 결합돼야 안정적으로 결론에 도달함을 시사한다. 문서화 정보는 질문이 많아질수록 비율이 다소 상승하지만 여전히 낮아, 특정 상황에서만 제한적으로 활용되는 보조 근거로 남는다.

종합하면 ANTLION의 질의 선택은 임의가 아니라, 오라클 생성 과정에서 발생하는 불확실성의 성격에 따라 달라진다. 기본 축은 테스트 시나리오의 검증 목표를 분명히 하는 것이며, 질의가 늘어날수록 경로 구조 확인과 assertion 후보 비교가 결합되어 경로 기반 의미와 표현 형식을 동시에 안정화하는 방향으로 확장된다.

표 3. ANTLION의 질의 전략의 질문 호출빈도

	0회	1회	2회	3회	4회	5회
평균(회)	150.4	118.4	55.4	36.0	12.4	0.4
비율(%)	40.2	31.7	14.8	9.6	3.3	0.1

표 4. ANTLION의 질문 유형별 사용빈도

	질문A	질문B	질문C	질문D	질문E
평균(회)	52.6	177.8	65.2	72.0	21.2
비율(%)	13.5	45.8	16.7	18.5	5.4

표 5. 질문 사용 개수(N)별 질문 유형 사용비율(%)

	질문A	질문B	질문C	질문D	질문E
1회	10.6	67.8	12.0	9.5	0.0
2회	26.8	91.2	20.6	46.4	15.0
3회	35.1	96.1	79.5	68.0	21.4
4회	96.7	93.4	88.2	84.0	37.7

5.4. 생성된 오라클의 다양성 분포

표 6은 ANTLION이 생성한 JUnit 4 테스트 오라클을 assertion 유형별로 분류한 결과를 보여준다. 가장 높은 비중은 assertEquals(30.4%)로, 반환값이나 계산 결과처럼 정확한 값 비교가 오라클 생성에서 핵심 형태임을 확인할 수 있다. 다음으로 assertFalse(16.1%), Fail(13.6%), assertTrue(12.7%)가 높은 비율을 차지해, 조건식 기반 검증과 기대한 예외가 발생하지 않을 때 실패를 명시하는 방식도 자주 사용되었다. 또한 assertNotNull(8.1%), assertNull(7.4%) 등 null 관련 검증이 일정 비중을 보였고, assertSame(2.1%), assertArrayEquals(2.0%), assertNotSame(0.5%)는 상대적으로 낮게 나타났다.

이 분포는 ANTLION이 값 비교에만 집중하지 않고, 조건 검증, null 검증, 명시적 실패, 참조 동일성 및 배열 비교까지 입력 맥락에 맞춰 다양한 assertion을 활용해 실행 가능한 JUnit 4 오라클을 폭넓게 구성하는 경향이 있음을 시사한다.

표 6. ANTLION이 생성한 Junit4 테스트 오라클 비율

유형	평균 비율(%)	순위
assertEquals	30.4	1
assertFalse	16.1	2
assertTrue	12.7	4
assertNull	7.4	6
assertNotNull	8.1	5
assertArrayEquals	2.0	8
assertSame	2.1	7
assertNotSame	0.5	9
fail	13.6	3
assertNotEquals	0.0	10

표 7. 오라클 생성의 토큰 소모량 및 API 비용

	입력	출력	합계
토큰 수	2,310,079	28,014	2,338,093
비용($)	2.89	0.28	3.17

5.5. 비용(토큰) 소모량

표 7은 Defects4J[21] 374개 입력 샘플에 대해 ANTLION이 오라클을 생성할 때 사용한 토큰과 비용의 평균을 제시한다. 전체적으로 입력 2,310,079 토큰과 출력 28,014 토큰을 사용해 총 2,338,093 토큰이 소모되었고, 비용은 입력 $2.89, 출력 $0.28로 합계 $3.17이다. 이를 샘플당으로 환산하면 평균 약 $0.0085로, 대규모 벤치마크 실험에서도 비용 부담이 크지 않으며 입력 토큰이 비용의 대부분을 차지하는 구조에서도 전반적으로 합리적인 비용으로 오라클 생성이 가능함을 보여준다.

6. 결론 및 향후 연구 계획

본 연구는 단위 테스트에서 핵심 요소인 테스트 오라클을 대형 언어 모델로 자동 생성하는 문제를 다루며, 다중 회차 질의-응답 프롬프트 기법인 ANTLION을 제안하였다. ANTLION은 테스트 대상 메서드, 테스트 접두 코드, 질문 목록을 기반으로 최소 문맥에서 시작한 뒤, 오라클 추론에 필요한 정보만 선택적으로 추가 제공하여 긴 프롬프트로 인한 환각을 완화하고 생성 성능을 높이고자 한다. Defects4J[21] 실험에서 ANTLION은 최신 비교 기준 기법[11, 19, 20]들보다 더 많은 고유 버그를 재현하였고, 동일 모델(GPT-5.1[24]) 프롬프트 엔지니어링이 제거된 기본 프롬프트 기준과의 프롬프트 엔지니어링 요소 제거 실험에서도 프롬프트 엔지니어링의 효과를 확인하였다. 또한 질의 사용 패턴과 생성된 assertion 분포 분석을 통해, ANTLION이 입력 난이도에 따라 필요한 정보를 단계적으로 보강하며 다양한 형태의 JUnit 4 오라클을 생성함을 확인하였다.

향후 연구에서는 고정된 질문 목록을 넘어, 모델의 불확실성을 자동으로 감지해 필요한 질문을 선택·생성하는 동적 질의 전략을 구현하고자 한다. 또한 질문 응답을 구조화된 형태로 반환하도록 개선하여 오라클 결정을 더 안정화하고, JUnit 5 및 타 언어·테스트 프레임워크로 확장 평가함으로써 일반화 가능성을 높이는 것을 목표로 한다.

감사의 글

이 논문은 정부의 재원으로 한국연구재단의 지원(RS-2021-NR060080, RS-2025-24533455)과 한국산업기술진흥원((P0020536, 2025년 산업혁신인재성장지원사업))의 지원을 받아 수행된 연구임

참고문헌

[1] Synopsys Editorial Team, “Coverity report on the ‘goto fail’ bug,” Synopsys Security Blog. http://security.coverity.com/blog/2014/Feb/a-quick-post-on-apple-security-55471-aka-goto-fail.html , 2014.

[2] Hacker News, “Twitter outage report,” Hacker News. https://news.ycombinator.com/item?id=8810157 , 2016.

[3] A. M. Porrello, “Death and denial: The failure of the Therac-25, a medical linear accelerator,” Death and Denial: The Failure of the THERAC-25, A Medical Linear Accelerator, 2012.

[4] S. B. Hossain, M. B. Dwyer, S. Elbaum, and A. Nguyen-Tuong, “Measuring and mitigating gaps in structural testing,” Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE), pp. 1712–1723, 2023.

[5] S. B. Hossain, A. Filieri, M. B. Dwyer, S. Elbaum, and W. Visser, “Neural-based test oracle generation: A large-scale evaluation and lessons learned,” Proc. ACM Joint ESEC/FSE Conf. (ESEC/FSE), pp. 120–132.

[6] D. Schuler and A. Zeller, “Checked coverage: An indicator for oracle quality,” Softw. Test., Verif. Reliab., vol. 23, no. 7, pp. 531–551, 2013.

[7] Y. Zhang and A. Mesbah, “Assertions are strongly correlated with test suite effectiveness,” Proc. ACM Joint Meeting on Foundations of Software Engineering (FSE), pp. 214–224, 2015.

[8] G. J. Myers, C. Sandler, and T. Badgett, The Art of Software Testing, John Wiley & Sons, 2011.

[9] Diffblue Ltd., “2019 Diffblue Developer Survey: What’s wrong with software speed, quality, and cost?” Technical Report, Diffblue,

https://www.diffblue.com/wp-content/uploads/2019/12/diffblue-2019-developer-survey-report.pdf, 2019.
[10] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri, “Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions?” Proc. ACM Softw. Eng., vol. 1, no. FSE, pp. 1889–1912, 2024.
[11] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, “TOGA: A neural method for test oracle generation,” Proc. Int. Conf. Softw. Eng. (ICSE), pp. 2130–2141, doi: 10.1145/3510003.3510141, 2022.
[12] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, “Code Coverage,” The Fuzzing Book, https://www.fuzzingbook.org/html/Coverage.html, 2024.
[13] G. Fraser and A. Arcuri, “EvoSuite: Automatic Test Suite Generation for Object-Oriented Software,” Proc. ACM Joint ESEC/FSE Conf. (ESEC/FSE), pp. 416–419, doi: 10.1145/2025113.2025179, 2011.
[14] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, “ChatUniTest: A Framework for LLM-Based Test Generation,” Companion Proc. ACM Int. Conf. Found. Softw. Eng. (FSE), pp. 572–576, doi: 10.1145/3663529.3663801, 2024.
[15] Q. Xie and A. M. Memon, “Designing and comparing automated test oracles for GUI-based software applications,” ACM Trans. Softw. Eng. Methodol. (TOSEM), vol. 16, no. 1, Art. 4, doi: 10.1145/1189748.1189752, 2007.
[16] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, “Automatic Generation of Oracles for Exceptional Behaviors,” Proc. Int. Symp. Softw. Testing Anal. (ISSTA), pp. 213–224, doi: 10.1145/2931037.2931061, 2016.
[17] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. Delgado Castellanos, “Translating Code Comments to Procedure Specifications,” Proc. Int. Symp. Softw. Testing Anal. (ISSTA), pp. 242–253, doi: 10.1145/3213846.3213872, 2018.
[18] Z. Liu, K. Liu, X. Xia, and X. Yang, “Towards more realistic evaluation for neural test oracle generation,” Proc. Int. Symp. Softw. Testing Anal. (ISSTA), pp. 589–600, 2023.
[19] S. B. Hossain, R. Taylor, and M. B. Dwyer, “Doc2OracLL: Investigating the Impact of Documentation on LLM-Based Test Oracle Generation,” Proc. ACM Softw. Eng., vol. 2, no. FSE, pp. 1870–1891, doi: 10.1145/3729354, 2025.
[20] S. B. Hossain and M. B. Dwyer, “TOGLL: Correct and Strong Test Oracle Generation with LLMs,” Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE), pp. 1475–1487, doi: 10.1109/ICSE55347.2025.00098, 2025.
[21] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs,” Proc. Int. Symp. Sift. Testing Anal. (ISSTA), pp. 437–440, doi: 10.1145/2610384.2628055, 2014.
[22] A. Sorokin et al., “Q-RAG: Long Context Multi-Step Retrieval via Value-based Embedder Training,” arXiv preprint, arXiv:2511.07328, 2025.
[23] Meta AI, “Introducing Code Llama, a state-of-the-art large language model for coding,” Meta AI Blog. https://ai.meta.com/blog/code-llama-large-language-model-coding.
[24] OpenAI, “GPT-5.1,” OpenAI API Documentation (Models), https://platform.openai.com/docs/models/gpt-5.1, 2026.
[25] Microsoft, “toga: ToGA Artifact (replication artifact for ‘TOGA: A Neural Method for Test Oracle Generation’),” GitHub repository, https://github.com/microsoft/toga, 2022.
[26] R. Just, “Defects4J: A Database of Real Faults and an Experimental Infrastructure to Enable Controlled Experiments in Software Engineering Research,” GitHub repository, ver. 3.0.1, https://github.com/rjust/defects4j, 2024.
[27] Q. Zhang, W. Sun, C. Fang, B. Yu, H. Li, M. Yan, J. Zhou, and Z. Chen, “Exploring Automated Assertion Generation via Large Language Models,” ACM Trans. Softw. Eng. Methodol. (TOSEM), vol. 34, no. 3, Art. 81, doi: 10.1145/3699598, 2025.
[28] K. Zheng, J. Decugis, J. Gehring, T. Cohen, B. Negrevergne, and G. Synnaeve, “What Makes Large Language Models Reason in (Multi-Turn) Code Generation?” Proc. Int. Conf. Learn. Represent. (ICLR). doi: 10.48550/arXiv.2410.08105.
[29] “Twenty questions,” Encyclopaedia Britannica, https://www.britannica.com/topic/twenty-questions , 2026.

프로그램 실행 중 발생한 버그의 실시간 자동 수정

노준영 ○ 김영재

울산과학기술원

nojon123@unist.ac.kr, kyj1411@unist.ac.kr

Axolotl: Automatically Fix Programs' Fault On-the-fly

Joonyeong Noh ○ YoungJae Kim

UNIST

요 약

기존 자동 프로그램 수정 (Automated Program Repair, APR) 연구들은 프로그램의 버그에 의해 실패하는 테스트 케이스가 있다고 전제하고, 해당 테스트를 성공하는 패치를 생성하도록 구성되어 있다. 따라서 프로그램의 테스트 케이스가 제공되고 프로그램을 매번 컴파일 후 실행할 수 있다고 가정하는데, 프로그램이 실제 서비스에 적용되면 테스트 케이스가 포함되어 있지 않은 경우가 많고 버그가 발생한 당시 이미 프로그램이 오랫동안 실행되어 처음부터 재실행하는 것이 힘든 경우가 많다. 본 연구에서는 프로그램 배포 후, 실제로 프로그램이 실행 중일 때 크래시 버그가 발생하여 예외를 일으킬 때, 프로그램을 중단하지 않고 일시정지한 후 런타임에 프로그램을 수정하고 난 다음 이어서 진행하는 방법을 연구하였다. 프로그램을 일시정지한 후 빠르게 좋은 품질의 패치를 얻기 위해 런타임에 얻을 수 있는 정보들과 ToT (Tree-of-Thought), 피드백 기법 등을 이용하여 대형 언어 모델(LLM)로 패치를 생성하였다. 그 후 생성된 패치를 검증하기 위해 black-box 퍼징 기법을 활용하였다. 이 연구를 통해 Python 프로그램에서 실행 도중 예외가 발생할 때 자동으로 수정한 후 프로그램을 이어서 실행하는 Axolotl 을 구현하였으며, 실험에서는 Python 을 사용하는 BugsInPy 벤치마크의 crash 버그들을 사용하여 평균적으로 11.3 분 이내에 92%의 패치 성공률을 보였다.

1. 서론

지난 2009 년에 출판된 Genprog [1] 이후, 최근까지 많은 자동 프로그램 수정 (Automated Program Repair, APR) 연구들이 진행되었다. 기존의 APR 연구들은 고품질의 패치를 생성하여 최대한 많은 버그들을 성공적으로 수정하는 데에 집중하고 있다. 이를 위해 많은 연구 [1-7]에서는 버그에 의해 실패하는 테스트들(failing test)과 버그에 상관없이 성공하는 테스트들(passing test)을 입력으로 받는다고 전제하여, 수많은 패치 후보들을 생성한 후 해당 테스트들을 실행하여 모든 테스트를 성공하면 성공적인 패치로 분류한다. 따라서 기존의 연구들은 이 과정에서 좋은 품질의 패치 후보들을 생성하여 패치 성공률을 높이는 데에 초점을 맞추고 있다.

기존 연구들에서는 프로그램들이 개발 과정 중이었다고 전제하여, 해당 프로그램들에게 실제로 패치 후보를 적용하여 반복적으로 컴파일하고 테스트를 실행하면서 패치를 생성하고 있다. 하지만 프로그램 배포 후에서는 테스트 케이스가 없고, 컴파일과 테스트 실행을 반복하며 패치를 검증하는 것은 매우 오랜 시간이 걸린다. 또 프로그램이 실제로 배포된 후에 버그가 발생하면 심각한 피해를 야기할 수 있다. 따라서 배포 이후에 발생하는 심각한 버그들은 개발자의 개입 이전에 프로그램의 중단이나 오작동을 방지하도록 매우 빠르면서도 심각한 부작용을 일으키지 않는 임시 패치를 진행할 필요가 있다.

NPEX [22]는 Java 프로그램에서 실행 도중 null pointer exception (NPE)이 발생했을 때, 주어진 테스트 없이 모델을 이용하여 생성한 패치를 검증한다. 이를 위해 기존에 개발자들이 NPE 를 패치했던 패턴을 이용해 모델을 학습한다. 그 후 프로그램을 이용해 심볼릭 실행 (symbolic execution)을 진행해 NPE 를 수정할 수 있는 명세를 추출한다. NPEX 는 Java 프로그램이 실행 중 NPE 가 발생한 상황에 특화되어 프로그램 실행 도중에 발생한 모든 예외에 대응할 수 없다.

Casino [23]과 Gresino [24]는 테스트가 주어진 상황에서 수많은 패치 후보들이 있을 때 그 후보들을 더 효율적으로 검증하기 위한 연구들이다. APR 에서 패치를 적용한 후 테스트를 실행하기 위해서는 프로그램을 매번 재컴파일해야 하는데, 이 과정이 오래 걸리고 일반적으로 오픈소스 프로젝트들은 모든 테스트를 실행하는데 시간이 오래 걸린다. Casino 와 Gresino 는 제한된 시간 내에 빠르게 패치를 찾아낼 수 있지만, 이 연구들 또한 프로그램을 중단한 후에 패치를 생성해야 하고 여전히 재컴파일과 테스트 실행이 필요하다.

위의 연구들은 프로그램의 실행 도중 흔히 발생하는 예외에 빠르게 대응할 수 있도록 하지만, 프로그램을 수정하기 위해 여전히 프로그램을 중단해야 한다는 한계가 있다. 또한 생성한 패치가 버그를 잘 고쳤는지 확인하기 위해 매번 프로그램을 재컴파일하고 수많은 테스트들 혹은 검증 과정을 실행해야 한다. 마지막으로 버그 수정이 완료된 후에는 프로그램을 처음부터 다시 실행해야 한다.

위의 한계들을 해결하기 위해 본 연구에서는 프로그램의 실행 도중 예외(exception)가 발생한 경우, 프로그램을 강제로 종료하지 않고 일시정지하여 실행중인 프로그램에 직접 패치를 적용한 후 이어서 실행하는 연구를 진행하였다. 이를 위해, 본 연구에서는 프로세스 체크포인트 기술을 활용하여 미리 실행중인 프로그램을 체크포인트하여 파일로 저장하고, 예외가 발생한 경우 저장된 프로세스를 불러와 패치를 생성 및 적용하고 프로그램을 이어서 실행할 수 있도록 하였다.

또한 최신 초대형 언어 모델 (LLM)과 Tree-of-Thought (ToT) [7], 그리고 피드백 기법을 활용하고 예외 메시지와 예외가 발생했을 때의 함수 인자 등 런타임 정보를 활용해 최대한 좋은 품질의 패치 후보를 생성해 프로그램이 오랜 시간 중단되지 않고 빠르게 패치를 생성하도록 구성하였다. 생성된 패치 후보가 올바른 패치인지 검증하기 위해 퍼징 기법을 활용해, 실행중인 프로그램에 직접 패치를 적용한 후 다양한 입력을 프로그램에 계속 적용하여 프로그램이 일어나서는 안 되는 부작용 (side-effect)를 일으키는지 확인하였다.

본 연구가 실제로 효과가 있는지 확인하기 위해 본 연구를 Python 을 이용해 Axolotl 로 명명한 도구를 구현하였으며, 실제 Python 오픈소스 프로젝트들의 버그들을 모아놓은 BugsInPy [25] 벤치마크에서 실제로 프로그램 내에서 예외가 발생하는 버그들만을 수집하여 실험을 진행하였다. LLM 으로 GPT-5.2 를 사용한 결과, 92%의 버그들을 성공적으로 수정할 수 있었으며 평균적으로 11.3 분이 걸렸다.

요약하면, 본 연구에서는 다음과 같은 내용들을 진행하였다.

- 실제 서비스중인 프로그램의 버그 수정: 실행중인 프로그램을 종료하지 않고 수정 후 이어서 실행
- 테스트들이 주어지지 않은 상황에서 패치 검증: 테스트가 없는 상황에서 퍼징 기법을 이용하여 패치가 버그를 수정하고 다른 부작용을 일으키지 않는지 검증
- 구현물 공개: 본 연구의 구현물(Axolotl)과 실험 데이터를 오픈소스로 공개
  - https://github.com/UNIST-LOFT/axolotl

2. 배경 지식

2.1. 자동 프로그램 수정 (APR)

기존의 APR 연구들은 일반적으로 얼마나 더 많은 버그들을 수정할 수 있는지에 집중하고 있으며, 이를 위해 더 좋은 품질의 패치를 생성하는 연구가 주를 이루고 있다.

따라서 기존의 연구들은 주로 패치를 더 잘 생성하는 방법을 중심으로 진행되고 있다. Genprog [1] 등은 버그가 있는 프로그램의 코드를 버그가 사라질 때까지 변형하여 패치를 생성하며, TBar [2] 등은 패치 템플릿을 사용하여 특정 패턴의 패치들을 생성한다. 최근에는 초대형 언어 모델 (LLM)을 사용하여 패치를 생성하는 연구 [5,6,7]가 활발히 진행되고 있다.

그림 1 은 기존의 일반적인 APR 연구들의 실행 흐름을 보여주고 있다. 입력으로 실패 테스트 (failing test)와 성공 테스트(passing test)를 받아, 패치 후보를 생성한 후 각 후보를 프로그램에 적용하여 실패 테스트와 성공 테스트를 실행한다. 해당 패치 후보가 모든 테스트를 성공하면 해당 패치는 그럴듯한 패치 (plausible patch)로 분류한다.

2.2. LLM 기반 패치 생성

최근 LLM 을 활용하여 패치를 생성하는 연구에서는 Chain-of-Thought (CoT) [11] 기법을 사용하여 패치를 생성한다. CoT 는 LLM 에 패치 생성을 요청할 때 패치를 한 번의 프롬프트만을 이용해 바로 생성하지 않고, 버그를 수정하기 위한 여러 과정을 LLM 을 이용해 진행하는 기법이다.

Nong et al. [10]은 보안 위협 (vulnerability)에 대응하기 위한 패치를 생성하기 위해 CoT 기법을 사용한다. 이 논문에서는 먼저 LLM 을 이용해 코드에 어떠한 보안 위협이 있는지 확인하고, 해당 보안 위협이 어디에 있는지 분석한 후 실제 패치를 생성하는 세 가지 과정을 위해 LLM 을 사용한다.

San2Patch [7]는 보안 위협에 대응하기 위해 CoT 를 확장한 Tree-of-Thought (ToT) 기법을 사용한다. ToT 는 CoT 의 각 과정에서 LLM 이 응답을 생성할 때 여러 개의 응답을 생성한 후, LLM 을 이용해 응답들에 점수를 매겨 높은 점수를 받은 응답에 우선순위를 주면서 샘플링하는 방법을 통해 좋은 응답을 더 많이 사용한다.

또 최근에는 패치 생성 후 테스트를 실행하여 실패한 경우, 해당 패치나 테스트 결과 등을 LLM 에 추가로 제공하여 더 좋은 품질의 패치를 얻는 피드백 기법도 연구가 이루어지고 있다. ThinkRepair [41]은 패치를 적용한 후 컴파일과 테스트 과정에서 나온 결과를 피드백으로 활용해 다음 패치 후보를 생성한다.

2.3. 프로세스 체크포인트

프로세스 체크포인트는 현재 시스템에서 실행중인 프로세스의 현재 상태를 파일로 저장하는 기술로, 추후에 원하는 경우 저장된 체크포인트를 불러와 이어서 실행할 수 있도록 한다.

CRIU [8]는 원하는 프로세스를 명령줄 (CLI) 환경에서 간단히 체크포인트하여 파일로 저장하는 기능을 제공하며, 추후에 해당 파일을 사용하여 프로세스를 복원한 후 이어서 실행할 수 있다. 또한 Docker [9]와 같이 사용하여 컨테이너도 체크포인트할 수 있는 기능을 제공한다. 추가적으로 체크포인트를 저장하고 불러오는 과정 중간에 셸 스크립트를 실행할 수 있다.

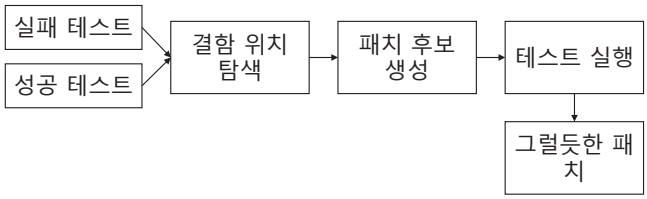


그림 1: APR 의 일반적인 흐름

3. 문제 제기

그림 1 은 기존 APR 연구들의 일반적인 흐름을 보여준다. 입력으로 실패 테스트 (failing test), 성공 테스트 (passing test), 버그가 있는 프로그램을 받고, 먼저 결함 위치 탐색 (fault localization)을 진행해 소스 코드에서 패치를 생성할 위치를 찾는다. 그 후, 다양한 방법들(2.1 절과 7.1 절 참조)을 통해 다량의 패치 후보들을 생성한다. 그 후에는 각 패치 후보들을 하나씩 프로그램에 적용해 실패 테스트를 실행하여 결과를 확인한다. 만약 모든 실패 테스트를 성공했다면 남은 성공 테스트들을 실행해 모두 성공하는지 확인한다. 모든 성공 테스트까지 모두 성공하면 해당 패치는 그럴듯한 패치 (plausible patch)로 분류한다.

이 흐름에서는 해당 프로그램의 테스트 케이스가 주어져야 하고, 각 패치마다 테스트를 실행해야 한다는 문제가 있다. 하지만 프로그램이 이미 서비스 중일 때에는 테스트들을 반복적으로 실행할 시간이 없고 심지어 테스트가 주어지지 않은 경우도 많다. 또한 패치가 완료된 후에 프로그램을 처음부터 다시 실행해야 한다는 문제가 있지만 서비스 중인 프로그램은 이미 많이 실행되어 프로그램을 처음부터 실행해 오류가 발생했던 시점으로 다시 돌아오는 것이 오래 걸리거나 불가능한 경우가 많다. 예를 들어, 본 연구의 실험에 사용한 BugsInPy [25] 벤치마크에 포함된 fastapi 와 httpie 프로그램은 웹 서버에 흔히 사용되어, 서버를 중단하여 버그를 수정한 후 재시작하는 것이 불가능하거나 시간이 오래 걸리는 경우가 많다. 따라서 기존의 APR 연구들은 프로그램의 배포 후가 아닌 개발 과정에서의 실행을 전제한다.

따라서 프로그램의 배포 후에는 예외가 발생한 경우 프로그램의 전체 테스트 케이스 없이 예외가 발생하는 시나리오 하나만으로 부작용이 없는 임시 패치를 빠르게 생성할 수 있어야 한다.

4. Axolotl

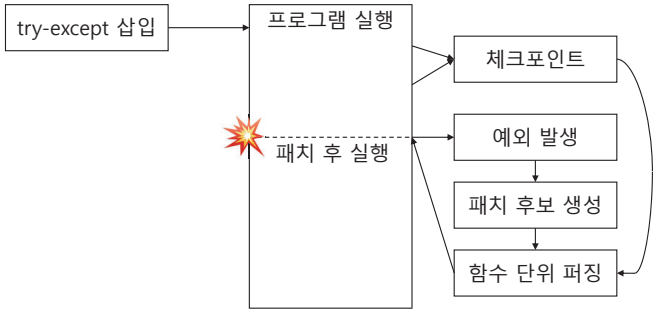


그림 2: Axolotl 의 개요

4.1. 개요

그림 2 는 본 연구를 구현한 Axolotl 의 전체 흐름을 보여준다. 프로그램을 실행하기 전에 자동으로 각 함수들의 몸체(body) 전체를 try-except 로 감싼 후, 프로그램을 실행한다. 프로그램 실행 중에는 패치 생성 후 다시 돌아갈 체크포인트를 지속적으로 저장한다. 예외가 발생하면 먼저 패치 후보들을 생성한 후, 저장해 두었던 체크포인트를 불러와 다양한 입력들을 생성하면서 함수 단위 퍼징을 진행해 각 패치들이 프로그램에서 부작용 (side-effect)을 유발하는지 확인한다. 패치가 부작용을 일으키지 않으면 다시 체크포인트를 불러와 패치를 적용하여 프로그램을 이어서 실행한다.

4.2. try-except 문 삽입

try-except 문은 프로그램 실행 전에 자동으로 각 함수의 몸체(body) 전체를 감싸는 형태로 삽입한다. 이를 통해 각 함수에서 예외가 발생하면 except 문으로 이동하여 Axolotl 의 패치 생성 단계로 넘어간다. try-except 문을 삽입할 때에는 Python 의 bytecode 단위에서 삽입하여 프로그램의 소스 코드 길이가 증가하지 않고 프로그램의 실행 시간 증가를 최소화하였다.

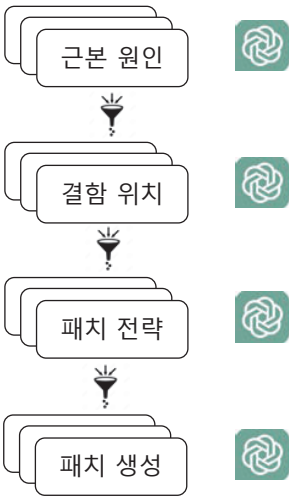


그림 3: 패치 후보 생성 과정

4.3. 프로그램의 체크포인트

예외 발생 시 패치 생성 후 실행 중인 프로그램으로 돌아오기 위해 Axolotl 은 주기적으로 실행 중인 프로그램의 체크포인트를 저장한다. 본 실험에서는 일정 시간마다 CRIU [8]를 이용하여 프로세스 전체를 저장하였다. 체크포인트는 별도의 프로세스를 통해 저장하여 원래 프로그램의 실행을 방해하지 않으며 오버헤드가 없다.

저장된 체크포인트는 패치 생성 및 적용 후 함수 단위 퍼징을 진행할 때 불러온다. 하지만 함수 단위 퍼징을 진행하려면 불러오려는 체크포인트가 패치된 함수에 진입하기 전이어야 패치가 적용할 수 있는데, 외부 프로세스에서는 프로그램 내부 동작을 알 수 없으므로 가장 최근에 저장된 체크포인트가 패치된 함수 이전에 저장되었는지 보장할 수 없다. 이를 위해 Axolotl 은 가장 최근에 저장된 체크포인트만 남기는 것이 아닌 기존에 저장했던 체크포인트를 계속 남겨 퍼징하려는 함수가 실행되기 직전의 체크포인트를 찾아 불러온다.

표 1: 패치 생성 시 사용한 프롬프트

원인 파악	<예외 로그> <stack trace> <Target Buggy Function> <버기 함수> Using the information above, create a detailed yet concise description of the exception. (후략)
결함 위치	<Goal> Your task is to map a provided <근본 원인> to the specific code snippet within the <버기 함수>. </Goal> <Instruction> 1. Read the <근본 원인> carefully. 2. Based on the <예외 종류> and <전략>, identify the **SINGLE most critical code snippet** that needs modification. (후략)
전략 생성	<Goal> (중략) </Goal> <Instruction> Please follow these steps to generate a robust fix strategy: 1. Analyze the exception details. (후략)
패치 생성	<Goal> Your task is to generate a corrected version of the provided <버기 함수> based on the <패치 전략>. </Goal> <버기 함수> <패치 전략> <예외 정보> Message: <예외 메시지> (후략)

4.4. 패치 후보 생성

프로그램 실행 도중 예외가 발생하면, 4.2 절의 try-except 문을 통해 Axolotl 의 패치 후보 생성 단계로 이동한다.

그림 3 에서와 같이 패치 후보를 생성할 때에는 San2Patch [7]의 ToT 를 사용한다. 먼저 예외의 근본적인 원인을 파악하고, 해당 함수에서 어디를 고칠지를 찾은 후 패치 전략을 얻은 다음 실제 패치 후보들을 생성한다. 각 과정에서는 ToT 기법을 적용하여 여러 응답을 받은 후 가장 좋은 응답을 선택한다. 표 1 은 실제 각 과정에서 사용된 프롬프트들을 보여주고 있다.

예외의 근본 원인 (root cause)을 파악할 때에는 먼저 예외 오류 메시지와 예외의 stack trace, 원래 함수의 소스 코드를 이용해 이 예외에 대한 설명과 근본 원인을 응답으로 받는다. 그 후 패치 위치를 찾을 때에는 앞에서 얻은 예외의 설명과 근본 원인, stack trace, 버그가 있는 함수의 소스 코드를 프롬프트로

주어 버기 함수 내에서 결함이 있는 위치를 찾는다. 그 다음 패치 전략을 얻을 때에는 앞에서 찾은 오류의 근본 원인과 버기 함수의 소스 코드, 결함 위치를 주고 패치 전략을 얻는다. 마지막으로 실제 패치를 생성할 때에는 위에서 얻은 모든 정보와 버기 함수를 주고 실제 패치 후보를 찾는다.

각 과정에서는 Tree-of-Thought(ToT) 전략을 사용하고 있다. 각 과정은 총 세 번씩 반복하여 세 개의 응답을 얻고, LLM 을 사용하여 각 응답마다 점수를 매겨 그 중 가장 높은 점수를 가진 응답만을 선택해 다음 단계로 진행한다.

패치 후보 생성 후에는 저장된 체크포인트를 불러와 실제로 적용해서 발생하던 예외가 발생하지 않는지 확인한다. 예외가 발생하지 않으면 그럴듯한 패치로 분류하여 수집한다.

예외가 발생하면 다시 전략 생성 단계로 돌아가는데, 이 때에는 피드백 기법을 사용한다. 기존에 실패했던 패치를 추가로 프롬프트에 제공하여 새로운 전략을 생성한다. 그 후 패치 후보를 새로 생성할 때에도 실패했던 패치를 추가로 제공하여 새로운 패치 후보를 얻는다. 이 과정을 그럴듯한(plausible) 패치를 생성할 때까지 최대 두 번 반복한다.

표 2: 퍼징에 사용된 변이 규칙

객체 타입	규칙
Enum	가능한 값 중 랜덤 선택
bool	부정(negate)
int	비트 뒤집기, 값 증가/감소, 특정 값으로 설정
float	비트 뒤집기
str	랜덤 문자 삽입/삭제, 비트 뒤집기
bytes	랜덤 바이트 삽입/삭제, 비트 뒤집기, 블록 복사/삭제
파일 경로	디렉토리 단위로 분할 후 str 규칙 적용
정규식	str 규칙 적용 후 패턴 유효성 확인
클래스 객체	필드를 따라가 재귀적으로 위 규칙 적용

4.5. 함수 단위 퍼징

그럴듯한 패치들을 생성한 후에는 각 패치들이 프로그램 내에서 부작용(side-effect)을 일으키지 않는지 확인하기 위해 함수 단위 black-box 퍼징 기법을 사용한다. 먼저 기존 프로그램 실행 중에 저장해 두었던 체크포인트를 불러온 후, 패치를 적용한다. 그 후에는 계속 해당 함수의 인자들을 다양하게 생성하여 버그가 있는 원래 함수와 패치된 함수를 각각 실행한다. 원래 함수에서 예외가 발생하지 않은 경우 패치된 함수에서의 실행 결과를 확인하는데, 패치된 함수에서 예외가 발생하면 잘못된 패치로 분류하여 제거한다. 각 패치마다 이 과정을 정해진 시간 동안 반복하여 잘못된 패치가 아니라면 올바른 패치로 분류하여 원래 프로그램에 적용한 후 프로그램을 이어서 실행한다.

LLM 으로부터 패치된 함수를 받으면 먼저 함수에 패치를 적용한다. 패치를 적용할 때에는 Python 내에 기본으로 정의된 compile()함수를 이용하여 bytecode 로 변환한 후 함수의 코드에 변환된 bytecode 를 적용한다.

퍼징을 진행할 때에는 원래 함수의 인자들을 변이(mutate)하여 실행하는데, AFL [30]에서 사용하는 변이 규칙들을 참고하였다. 표 2 가 각 Python 기본형들에 적용된 변이 규칙들을 보여주고 있는데, 인자가 기본형이 아닌 복잡한 클래스일 경우에는 클래스의 멤버 변수(field)를 따라가서 재귀적으로 변이한다. 입력을 하나 생성할 때마다 변이 규칙을 최대 10 번까지 랜덤하게 적용한다.

퍼징은 각 패치마다 10 분씩 진행하며, 매번 새로운 함수 인자를 생성한 후 함수를 실행한다. 그 결과 패치된 함수 안에서 원래 함수에서는 발생하지 않던 새로운 예외가 발생하면 잘못된 패치로 분류한다. 이 때에는 다시 패치 후보 생성 단계로 돌아가는데, 이 때 패치 후보를 생성할 때에는 LLM 에 잘못된 패치 후보를 피드백으로 제공하여 기존 패치와 다른 패치 후보를 생성한다.

5. 구현

Axolotl의 전체 구조는 4055 라인의 Python으로 구현하였으며, 실행중인 프로세스의 체크포인트를 저장하고 불러오기 위해 CRIU [8]를 사용하였다. Python 프로그램에서 예외를 탐지할 수 있도록 bytecode [26] Python 라이브러리를 사용하여 Python의 bytecode 수준에서 함수 전체에 try-except문을 삽입했다. 패치 후보를 생성하기 위해 OpenAI의 GPT-5.2 [31] 모델을 사용하였으나, 다양한 LLM 모델을 간단하게 추가할 수 있다. 패치 검증을 위한 퍼징은 자체적으로 구현하였으나 함수 인자들의 변이 규칙은 AFL [30]을 참고하였다 (4.5절 참고).

6. 실험

6.1. 환경 및 벤치마크

본 연구의 실험에서는 Python을 대상으로 하는 기존 연구 [27-29]에서 흔히 사용되는 BugsInPy [25] 벤치마크에서 프로그램이 직접 예외를 일으키는 버그들 중 재현이 불가능한 버그들을 제외하여 총 25개(표 4)를 선정하였다. 실험은 Ubuntu 22.04.5 및 Anaconda v24.11.3에서 진행하였다.

표 3: Axolotl의 실험 결과. 각 열에서 O는 패치성공을, X는 패치 실패를 의미한다.

	GPT-5.2				Qwen-3-Next			
	L1	L2	L3	실행 시간 (초)	L1	L2	L3	실행 시간 (초)
black-14	O	O	O	679.1	O	O	O	650.73
black-16	O	O	X	714.5	O	O	X	673.79
black-17	O	O	O	668.13	O	O	X	645.73
pandas-49	O	O	O	699.48	X	X	X	109.92
pandas-77	O	O	O	842.0	O	O	O	668.49
pandas-99	X	X	X	325.17	X	X	X	182.92
pandas-102	O	O	X	698.49	X	X	X	101.86
pandas-117	O	O	X	700.17	X	X	X	111.63
pandas-142	O	O	O	709.16	O	O	X	680.81
pandas-146	O	O	O	713.99	X	X	X	153.01
pandas-150	O	O	O	716.2	O	O	O	683.77
pandas-160	O	O	X	742.17	O	O	X	664.85
pandas-168	O	O	O	865.48	X	X	X	288.35
scrapy-15	O	O	O	698.94	O	O	O	676.81
scrapy-17	O	O	O	670.87	O	O	O	647.84
scrapy-29	O	O	O	1338.78	O	O	O	649.81
tornado-9	O	O	O	673.78	O	O	O	656.74
youtube-dl-5	O	O	X	730.1	X	X	X	88.97
youtube-dl-11	O	O	O	691.89	X	X	X	62.21
youtube-dl-16	X	X	X	276.28	X	X	X	157.07
youtube-dl-17	O	O	O	702.73	O	O	X	684.41
youtube-dl-22	O	O	X	730.73	X	X	X	161.4
youtube-dl-28	O	O	O	671.54	O	O	O	652.21
youtube-dl-33	O	O	O	695.02	O	O	X	656.13
youtube-dl-37	O	O	O	668.48	X	X	X	94.01
총합	23/25	23/25	17/25		14/25	14/25	8/25	
성공률	92	92	68	678.19	56	56	32	428.38

LLM 은 상용 모델로 GPT-5.2 [31]를 사용하였으며 오픈소스 모델로 Qwen-3-Next [40]를 사용하였다.

표 4: BugsInPy 벤치마크에서 실험에 사용한 버그

프로그램	버그 개수
black	3
pandas	10
scrapy	3
tornado	1
youtube-dl	8
총합	25

6.2. 연구 질문 (Research question, RQ)

본 연구에서는 Axolotl 을 이용하여 세 가지 연구 질문(RQ)에 답하기 위한 실험을 진행하였다.

- RQ 1: Axolotl 의 효과: Axolotl 이 얼마나 많은 예외들을 수정할 수 있는가?
- RQ 2: Axolotl 의 효율성: 패치를 생성하고 수정하는 데 얼마나 오래 걸리는가?
- RQ 3: Ablation Study: Axolotl 에서 패치 생성에 사용한 세 가지 기법 (런타임 정보, ToT, 피드백)이 패치의 품질에 각각 얼마나 영향을 주는가?

RQ 1 을 위해 Axolotl 을 BugsInPy 벤치마크에 적용하여 25 개의 버그 중 얼마나 많이 수정할 수 있는지 실험하였다. RQ 2 에서는 25 개의 버그들 중 패치에 성공한 버그들에서 평균적으로 어느 정도의 시간이 걸리는지를 측정하였다. RQ 3 에서는

Axolotl 에서 패치 생성에 사용한 런타임 정보와 ToT 기법, 피드백 기법이 각각 패치 생성에 얼마나 기여하는지를 측정하기 위해 각 기법을 사용하지 않고 패치를 생성하여 결과를 비교하였다.

ToT 의 각 단계에서는 3 개의 응답을 생성하여 그 중 가장 높은 점수를 가지는 응답을 선택하였다. 패치 검증에 실패하면 최대 2 번 패치 생성으로 돌아가 피드백을 제공하며, 그 후에도 실패하면 패치 실패로 간주하였다.

6.2. 결과

```
  if src_txt[-1] != "\n":
      src_txt += "\n"
```

그림 4-1: 버그 코드 블럭

```
- if src_txt[-1] != "\n":
+ if src_txt[-1:] != "\n":
      src_txt += "\n"
```

그림 4-2: 개발자 패치

```
- if src_txt[-1] != "\n":
+ if len(src_txt) > 0 and src_txt[-1] != "\n":
      src_txt += "\n"
```

그림 4-3: Qwen-3-Next 가 생성한 패치

```
- if src_txt[-1] != "\n":
+ if not src_txt.endswith("\n"):
      src_txt += "\n"
```

그림 4-4: GPT-5.2 가 생성한 패치

그림 4: black-17

6.2.1. RQ 1: Axolotl 의 효과

표 3 은 Axolotl 의 실험 결과를 보여준다. 표의 왼쪽은 GPT-5.2 의 결과이고 오른쪽은 Qwen-3-Next 의 결과이다. 표에는 3 개의 결과가 있는데, L1 은 목표 예외를 회피하는 데에 성공한 경우이고 L2 는 퍼징의 결과 패치가 함수에 부작용을 만들어내지 않는 경우, L3 는 개발자 패치의 의도와 일치한 패치일 경우 결과를 보여준다. 예를 들어, GPT-5.2 에서 pandas-117 에서는 목표 예외를 성공적으로 회피하면서 해당 함수에 새로운 부작용을 만들어내지 않지만 개발자 패치와 비교했을 때 함수의 동작이 다르다. L3 의 경우, 본 연구의 저자들이 직접 생성된 패치를 확인하여 개발자의 패치와 의미적으로 동등한지 판별하였다.

표 3 의 L1 과 L2 에서 GPT-5.2 는 92%의 성공률을 보였고 Qwen-3-Next 는 56%의 성공률을 보여 Qwen-3-Next 보다 GPT-5.2 가 더 좋은 성공률을 보였다. 또 L3 에서는 GPT-5.2 와 Qwen-3-Next 에서 각각 68%와 32%의 성공률을 보여 L1 과 L2 에 비해 낮은 성공률을 보였다.

L3 에서 상대적으로 낮은 성공률을 보이는 주된 이유는 LLM 이 예외 회피를 최우선 목표로 패치를 생성하기 때문이다. 예를 들어, 그림 4 는 black-17 버그의 코드를 보여준다. 그림 4-1 의 버그 코드에서는 `src_txt` 의 길이가 0 일 경우 `src_txt[-1]`에서 `IndexError` 가 발생한다. 그림 4-2 의 개발자 패치는 `src_txt[-1]`을 `[-1:]`로 변경하여 문자열의 크기가 0 일 때에도 개행 문자(‘\n’)를 삽입한다. 하지만 그림 4-3 의 Qwen-3-Next 가 생성한 패치는 조건식에 `len > 0` 을 추가하여, 빈 문자열일 경우 개행 문자를 삽입하지 않으므로 개발자 패치와 동작이 달라지지만 `IndexError` 예외를 발생시키지는 않는다. 반면 그림 4-4 의 GPT-5.2 가 생성한 패치는 `endswith()` 메소드를 사용하여 빈 문자열일 경우에도 개행 문자열을 삽입한다. 따라서 black-17 버그에서는 Qwen-3-Next 의 경우 예외 회피에는 성공하였으나 프로그램의 동작이 개발자 패치와는 달랐다. 반면 GPT-5.2 는 개발자의 의도와 맞는 패치를 생성하였다.

Axolotl 은 런타임 에러를 회피하는 데에는(L1, L2) 좋은 성능을 보여주었으나, 개발자의 의도까지 파악하여 기능적 완전성을 갖춘 패치(L3)를 생성하는 데에는 한계를 보였다.

6.2.2. RQ 2: Axolotl 의 효율성

표 3 에서는 각 버그마다 실행 시간을 보여주고 있다. GPT-5.2 에서는 평균 678.19 초(약 11.3 분)가 소요되었고 Qwen-3-Next 에서는 평균 428.38 초(약 7.14 분)가 소요되어 Qwen-3-Next 가 더 빠르게 동작하는 것을 관찰하였다. GPT-5.2 는 상용 모델로 네트워크를 통해 프롬프트와 응답을 주고받아야 하므로 로컬에 설치하는 Qwen-3-Next 보다 네트워크 시간이 더 많이 필요하다.

Axolotl 의 실행 시간 대부분은 패치 검증을 위한 함수 단위 퍼징에 사용된다. 퍼징을 하기 전 패치를 생성하는 데에는 GPT-5.2 와 Qwen-3-Next 를 사용하였을 때 평균적으로 각각 126.19 초(약 2.1 분)와 92.38 초(약 1.5 분)가 걸려, 패치 생성은 매우 빠르게 이루어지는 것을 관찰하였다.

또 CRIU 를 이용하여 체크포인트를 생성하는 데에는 평균적으로 0.79 초의 오버헤드가 발생하여 원래 프로그램의 실행 속도를 크게 저해하지 않는다.

Axolotl 과 GPT-5.2 를 사용하였을 때 약 11.3 분이 소요되어, 프로그램 실행 도중 프로그램을 오랜 시간

동안 중지하지 않고 버그를 충분히 빠르게 수정할 수 있었다.

6.2.3. RQ 3: Ablation Study

Ablation Study 를 위해 앞에서 성공률이 더 높았던 GPT-5.2 를 사용해, Axolotl 에서 패치를 생성할 때 사용한 세 가지 요소들을 하나씩 제거하고 패치 생성을 시도하여 각 요소들이 Axolotl 의 성능에 미치는 영향을 관찰하였다. 6.2.1 절에서 L1 과 L2 는 성공률이 완전히 일치하여 Ablation Study 에서는 L2 결과를 제외하였다.

그림 5 는 각 요소를 제거했을 때의 성공률과 실행 시간을 보여준다. ToT 가 없는 경우에는 실행 시간이 가장 빨랐고 Axolotl 과 동일한 L1 성공률을 기록했지만, 버그의 원인과 패치 전략을 추론하는 과정이 없으므로 패치의 품질이 낮아져 L3 성공률은 낮아졌다. 피드백을 제공하지 않은 경우에는 LLM 이 기존에 실패했던 패치를 알지 못하므로, 패치 실패 후 다시 패치를 생성할 때에도 동일하거나 유사한 패치를 생성하였다. 동적 정보를 제공하지 않았을 때는 패치 위치를 식별하지 못해 가장 긴 실행 시간과 가장 낮은 패치 성공률을 보였다.

위 실험 결과는 예외를 효과적으로 회피하며 좋은 품질의 패치를 빠르게 생성하기 위해서 세 가지 요소가 모두 필요함을 보여주고 있다.

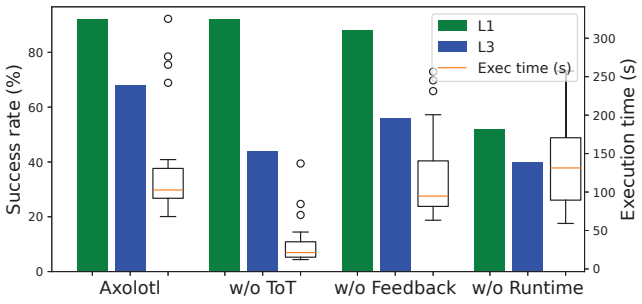


그림 5: Ablation Study 결과. w/o ToT, w/o Feedback, w/o Runtime은 각각 ToT, 피드백, 동적 정보를 사용하지 않았을 때의 결과이다. 초록색과 파란색 막대는 각각 L1과 L3 결과이고, 하얀색 상자는 총 실행 시간이다.

7. 토론 및 한계

본 연구에서 패치는 프로그램 실행 도중 발생한 예외를 회피하기 위한 목적으로 생성하였으며, 생성된 패치는 벤치마크에 포함된 개발자의 패치와 상이한 경우가 있다. 이는 위 실험의 RQ 1 의 결과 (6.2.1 절 참고)에서도 관찰할 수 있는데, L2 에서는 성공했으나 L3 에서는 실패한 경우 해당 함수에서는 부작용이 발생하지 않으나 실제 프로그램의 사용 시나리오에서는 다른 부작용이 발생할 수 있다. 이러한 경우에는 Axolotl 이 부작용을 새로운 예외로 감지하여 대응하는 또 다른 패치를 생성할 수 있다.

또 6.2.1 절의 실험 결과에서 L1 과 L2 의 패치 성공률은 동일한데, 이는 본 연구의 퍼징을 이용한 패치 검증 단계에서 함수의 유효한 인자를 충분히 만들지 못하기 때문이다. 패치가 새로운 예외를 일으키는지 확인하기 위해 먼저 새로 생성한 함수의 입력을 버기 함수에 실행하는데, 이 때 버기 함수가 예외를 일으키지 않아야 패치 검증에 사용할 수 있다. 그러나 예외를 일으키는 입력을 초기 시드로 제공해서 퍼징을 진행하면 예외를 일으키지 않는 입력을 생성하는 것이 어렵다.

또 본 연구에는 함수 전체를 try-except 문으로 감싼 후 프로그램을 실행할 때 해당 함수에서 발생한 예외가 이미 프로그램의 다른 부분에서 처리될 가능성이 있다는 한계가 있다. 즉, 발생한 예외가 의도한 것이고, 함수 밖에서 처리하도록 구현되어 있을 수 있다. 이 경우에는 간단한 정적 분석을 통해 이미 처리되어 있는 예외는 그대로 raise 하고 나머지 예외에만 Axolotl 를 적용할 수 있다.

마지막으로 본 연구에는 LLM 이 가지는 Data leakage 문제가 있다. BugsInPy 는 APR 연구에서 흔히 사용되는 벤치마크로 LLM 이 이미 버그들을 학습했을 가능성이 있으나, 이 문제는 LLM 기반의 많은 연구들 [6,7]도 공유하고 있다.

본 연구의 구현과 실험은 Python 으로 진행하였으나, 본 연구의 기법은 C, C++나 Java 등의 다른 언어에도 적용할 수 있다. 다만 생성된 패치를 실시간으로 프로그램에 적용하기 위해 복잡한 기술과 노력이 필요할 수 있다.

8. 관련 연구

8.1. 패치 생성

현재까지의 APR 연구에서는 여러 기법으로 많은 패치 후보들을 생성한 후, 각 패치들을 하나씩 적용하여 프로그램에 포함된 테스트 케이스들을 실행하여 모든 테스트들이 성공하는 패치들만을 출력하였다. Genprog [1], Arja [12], Arja-e [13]는 패치 후보를 생성하기 위해 유전 프로그래밍 (genetic programming) 기법을 사용한다. 버그가 있는 코드에서 시작해, 유전 프로그래밍을 이용해 프로그램을 조금씩 변형하여 패치를 생성한다. Avatar [14], Fixminer [15], TBar [2]는 템플릿을 이용해 특정 패턴의 패치 후보를 생성한다. Angelix [4], CPR [16], Prophet [17]도 템플릿을 사용하지만, 패치를 효과적으로 생성하기 위해 심볼릭 실행 등의 기법을 사용한다.

최근에는 AI 모델을 이용해 패치를 생성하는 기법들도 활발히 연구가 이루어지고 있다. Recoder [18], ARJANMT [19], SequenceR [20]은 전통적인 인공

신경망 (neural network) 모델을 사용하여 패치를 생성한다. AlphaRepair [5], AutoCodeRover [21], SRepair [6]은 LLM 모델을 사용하여 패치를 생성한다.

8.2. 패치 검증

기존의 일반적인 APR 연구들 [1-7,12-21,23,24]은 패치를 검증하기 위해 프로그램에 제공된 테스트 케이스들을 이용한다. NPEX [22]는 개발자들이 흔히 사용하는 패턴을 학습한 통계 모델을 이용해 패치를 검증한다. SPIDER [37]는 정적 분석을 활용해 AST 가 특정 패턴을 만족해야만 올바른 패치로 분류한다. UC-KLEE [38]는 일반적인 심볼릭 실행과 다른 Under-contrained 심볼릭 실행을 사용하여 초기 테스트 없이 검증을 진행할 수 있다. VulnFix [39]는 스냅샷 퍼징 (snapshot fuzzing)을 이용한다. 스냅샷 퍼징은 프로그램이나 함수의 입력을 변이하는 것이 아닌 프로그램의 상태 자체를 변이하며 퍼징을 진행한다.

9. 결론

본 연구에서는 프로그램 배포 후 실행 도중 크래쉬 버그가 발생한 경우에 프로그램을 중지하지 않고 일시정지한 후 프로그램을 수정하여 이어서 실행하는 기술을 연구하였고 Python 프로그램을 대상으로 하는 도구를 구현하여 Axolotl 로 명명하였다. 실험을 통해 GPT-5.2 를 활용한 경우 평균 11.3 분 안에 92%의 패치 성공률을 확인하였다.

[참고문헌]

[1] Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W., GenProg: A generic method for automatic software repair, IEEE Transactions on Software Engineering, 38, 1, 54-72, 2012.

[2] Liu, K., Koyuncu, A., Kim, D., & Bissyandé, T. F., TBar: Revisiting template-based automated program repair, Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 31-42, 2019.

[3] Yi Li, Shaohua Wang, and Tien N. Nguyen, DEAR: a novel deep learning based approach for automated program repair, Proceedings of the 44th International Conference on Software Engineering, 511-523, 2022.

[4] Mechtaev, S., Yi, J., & Roychoudhury, A., Angelix: Scalable multiline program patch synthesis via symbolic analysis, Proceedings of the 38th International Conference on Software Engineering, 691-701, 2016.

[5] Chunqiu Steven Xia and Lingming Zhang, Less Training, More Repairing Please: Revisiting Automated Program Repair Via Zero-Shot Learning, Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 959-971, 2022.

[6] Xiang, J., Xu, X., Kong, F., Wu, M., Zhang, Z., Zhang, H., & Zhang, Y., How far can we go with practical function-level program repair?, arXiv preprint arXiv:2404.12833, 2024.

[7] Kim, Y., Shin, S., Kim, H., & Yoon, J., Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis, 34th USENIX Security Symposium (USENIX Security 25), 4401-4419, 2025.

[8] CRIU Project, CRIU: Checkpoint/Restore In Userspace, https://criu.org, 2026.

[9] Merkel, D., Docker: lightweight linux containers for consistent development and deployment, Linux Journal, 239, 2, 2014.

[10] Nong, Y., Aldeen, M., Cheng, L., Hu, H., Chen, F., & Cai, H., Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities, arXiv preprint arXiv:2402.17230, 2024.

[11] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., ... & Zhou, D., Chain-of-thought prompting elicits reasoning in large language models, Advances in neural information processing systems, 35, 24824-24837, 2022.

[12] Yuan, Y., & Banzhaf, W., Arja: Automated repair of java programs via multi-objective genetic programming, IEEE Transactions on Software Engineering, 46, 10, 1040-1067, 2020.

[13] Yuan, Y., & Banzhaf, W., Toward better evolutionary program repair: an integrated approach, ACM Transactions on Software Engineering and Methodology, 29, 1, 5:1-5:53, 2020.

[14] Koyuncu, A., Liu, K., Bissyandé, T. F., Kim, D., Klein, J., Monperrus, M., & Le Traon, Y., FixMiner: Mining relevant fix patterns for automated program repair, Empirical Software Engineering, 25, 3, 1980-2024, 2020.

[15] Liu, K., Koyuncu, A., Kim, D., & Bissyandé, T. F., Avatar: Fixing semantic bugs with fix patterns of static analysis violations, Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, 456-467, 2019.

[16] Shariffdeen, R., Noller, Y., Grunske, L., & Roychoudhury, A., Concolic program repair, Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 390-405, 2021.

[17] Long, F., & Rinard, M., Automatic patch generation via learning from successful patches, Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 298-312, 2016.

[18] Zhu, Q., Sun, Z., Xiao, Y., Zhang, W., Yuan, K., Xiong, Y., & Zhang, L., A Syntax-Guided Edit Decoder for Neural

Program Repair, Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 341-353, 2021.
[19] Li, D., Wong, W. E., Jian, M., Geng, Y., & Chau, M., Improving search-based automatic program repair with Neural Machine Translation, IEEE Access, 10, 51167-51175, 2022.
[20] Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L. N., Poshyvanyk, D., & Monperrus, M., Sequencer: Sequence-to-sequence learning for end-to-end program repair, IEEE Transactions on Software Engineering, 47, 9, 1943-1959, 2021.
[21] Zhang, Y., Ruan, H., Fan, Z., & Roychoudhury, A., Autocoderover: Autonomous program improvement, Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 1592-1604, 2024.
[22] Lee, J., Hong, S., & Oh, H., Npex: Repairing java null pointer exceptions without tests, Proceedings of the 44th International Conference on Software Engineering, 1532-1544, 2022.
[23] Kim, Y., Han, S., Khamit, A. Y., & Yi, J., Automated Program Repair from Fuzzing Perspective, Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 854-866, 2023.
[24] Kim, Y., Park, Y., Han, S., & Yi, J., Enhancing the Efficiency of Automated Program Repair via Greybox Analysis, Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 1719-1731, 2024.
[25] Widyasari, R., Sim, S. Q., Lok, C., Qi, H., Phan, J., Tay, Q., ... & Ouh, E. L., BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies, Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 1556-1560, 2020.
[26] Dartiailh, M., bytecode: A Python module to generate and modify bytecode, https://github.com/MatthieuDartiailh/bytecode, 2026.
[27] Oh, W., & Oh, H., PyTER: effective program repair for Python type errors, Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 922-934, 2022.
[28] Cai, X., & Jiang, L., Adapting Knowledge Prompt Tuning for Enhanced Automated Program Repair, Proceedings of the 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering, 360-371, 2025.
[29] Fu, A., Xu, P., Li, J., Kuang, B., & Gao, Y., InstructRepair: Instruct Large Language Models with Rich Bug Information for Automated Program Repair, IEEE Transactions on Information Forensics and Security, 20, , 1113-1125, 2025.
[30] Zalewski, M., AFL, https://github.com/google/AFL, 2026.
[31] OpenAI, ChatGPT (GPT-5.2 version), https://chat.openai.com, 2026.
[32] Fioraldi, A., Maier, D., Eißfeldt, H., & Heuse, M., AFL++: Combining Incremental Steps of Fuzzing Research, Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT), 2020.
[33] Serebryany, K., LibFuzzer – a library for coverage-guided fuzz testing, https://llvm.org/docs/LibFuzzer.html, 2026.
[34] Kim, T. E., Choi, J., Heo, K., & Cha, S. K., DAFL: Directed Grey-box Fuzzing guided by Data Dependency, Proceedings of the 32nd USENIX Security Symposium, 4931-4948, 2023.
[35] Rong, H., You, W., Wang, X., & Mao, T., Toward Unbiased Multiple-Target Fuzzing with Path Diversity, Proceedings of the 33rd USENIX Security Symposium, 2475-2492, 2024.
[36] Bao, A., Zhao, W., Wang, Y., Cheng, Y., McCamant, S., & Yew, P. C., From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification, Proceedings of the 34th USENIX Security Symposium, 6977-6997, 2025.
[37] Machiry, A., Redini, N., Camellini, E., Kruegel, C., & Vigna, G., Spider: Enabling fast patch propagation in related software repositories, Proceedings of the 2020 IEEE Symposium on Security and Privacy, 1562-1579, 2020.
[38] Ramos, D. A., & Engler, D., Under-Constrained Symbolic Execution: Correctness Checking for Real Code, Proceedings of the 24th USENIX Security Symposium, 49-64, 2015.
[39] Zhang, Y., Gao, X., Duck, G. J., & Roychoudhury, A., Program vulnerability repair via inductive inference, Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 691-702, 2022.
[40] Qwen Team, Qwen3 Technical Report, arXiv, 2505.09388, 2025.
[41] Yin, X., Ni, C., Wang, S., Li, Z., Zeng, L., & Yang, X., Thinkrepair: Self-directed automated program repair, Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 1274-1286, 2024.

X-RAD Engineering Recipe: 하이브리드 그래프와 2단계 책임 분리를 통한 설명가능한 이상 탐지

김민지 ○ 허대영

국민대학교

litecj702@gmail.com, dyheo@kookmin.ac.kr

X-RAD Engineering Recipe: Explainable Anomaly Detection via Hybrid Graphs and 2-Stage Responsibility Decoupling

MINJI KIM○ Daeyoung Heo

Kookmin University

요 약

산업제어시스템(ICS) 및 항공 엔진과 같은 복합 공학 시스템에서 이상 탐지는 단순한 성능 지표를 넘어, 조기 경보 능력과 결과에 대한 진단적 근거 제시가 필수적이다. 본 논문에서는 선행 연구된 X-RAD의 핵심 메커니즘을 소프트웨어 공학적 관점에서 재정식화하여, 하이브리드 그래프 구축과 2-Stage 모듈 책임 분리를 핵심 설계 원칙으로 채택한 재현 가능한 개발 및 검증 절차(Engineering Recipe)를 제안한다. 제안된 파이프라인은 (1) 운용 모드별 정상 패턴을 학습하는 조건부 Transformer 기반 전역 탐지기(Stage 1), (2) 물리적 지식과 데이터 근거를 결합한 하이브리드 그래프 위에서 관계 붕괴를 포착하는 진단기(Stage 2), (3) 두 모듈의 출력을 Z-score로 정규화하여 통합하는 Score Fusion 레이어로 구성된다. 특히 Two-Stage Responsibility Decoupling 구조를 통해 탐지(What)와 진단(Why)을 구조적으로 격리함으로써 독립 검증 환경을 구축하고, 반사실적 기여도(ΔCF)를 통해 엣지 수준 설명의 타당성을 정량적으로 검증한다. C-MAPSS 및 SWaT 데이터셋을 이용한 비지도 학습 평가 결과, 평균 AUPRC 0.5295와 NAB-score 92.57을 기록하여 설정된 목표치(AUPRC>0.5, NAB>70)를 상회하였으며, 실제 고장 시점 대비 유의미한 선행 경보(평균 -18.7 step)를 제공함을 입증하였다. 또한 SWaT 환경에서 지목된 상위 엣지가 실제 공정 구조(P&ID)와 높은 정합성을 보임을 확인하여 현장 적용 가능한 진단 체계로서의 유효성을 실증하였다.

1. 서론 (Introduction)

산업제어시스템(ICS), 항공기 엔진, 발전 설비와 같은 복잡 공학 시스템은 수십-수백 개의 센서로부터 고주파 다변량 시계열 데이터를 수집한다. 이러한 시스템에서 발생하는 이상은 단순한 단일 센서의 임계치 초과를 넘어, 운용 조건의 변화(부하, 모드), 센서 간 상호작용의 붕괴, 장시간의 누적 열화 등 복합적인 형태로 표출된다. 따라서 실제 현장에 적용 가능한 이상탐지기는 (i) 조기 경보 능력, (ii) 관계 붕괴에 대한 민감도, (iii) 진단적 설명가능성, 그리고 (iv) 운영 및 검증 관점에서의 소프트웨어 구조적 타당성을 동시에 만족해야 한다.

특히 조기성은 단순히 '이상 발생 후의 빠른 감지'가 아니라, '실제 고장이나 공격 이전에 경보를 제공하여 대응 시간을 선제적으로 확보하는 것'을 의미한다. 기존의 전역 점수(Global score) 중심 방식은 이상 징후의 존재를 신속히 포착하는 데 유리하지만, 변수 간 상호작용이 붕괴되는 유형의 이상에서는 구체적인 원인 관계를 제시하지 못해 분석 시간이 지연되는 한계가 있다. 반면 관계 수준 진단은 원인 후보(센서쌍 및 연결)를 제공하여 해석력을 높여주지만, 전역적 열화 패턴에 대한 민감도가 부족할 경우 조기 경보에 실패할 수 있다. 따라서 조기성과 원인 규명을 동시에 달성하기 위해서는 '전역 탐지'와 '관계 진단'의 유기적인 상보 결합이 필수적이다.

인공지능을 위한 소프트웨어 공학(Software Engineering for AI, 이하 SE4AI) 관점에서는, 단순한 모델 성능 지표의 나열을 넘어 시스템의 실질적인 이식성과 유지보수성을 강조한다. 연구 결과가 실제 공학 시스템에 효과적으로 통합되기 위해서는 데이터 전처리, 그래프 구성 근거, 모듈 간 인터페이스(입·출력 및 책임 범위), 그리고 체계적인 검증 절차가 재현 가능한 형태로 명시되어야 하며, 각 단계는 문서화된 산출물로서 관리되어야 한다. 특히 모듈별 책임(Responsibility)의 명확화는 성능 저하 발생 시 원인을 신속하게 국소화(Localization)하고 대응하기 위한 필수적인 공학적 요건이다.

이러한 문제의식 하에, 본 논문은 선행 연구인 X-RAD의 핵심 메커니즘을 유지하면서도[1], 이를 도메인 확장성(CBM+ ↔ ICS)과 검증 엄밀성을 갖춘 '확장판

엔지니어링 레시피(Engineering Recipe)'로 정식화하여 제시한다. 특정 운영 환경이나 정책에 종속적인 설계를 배제하고, 표준화된 산출물(Artifact)과 검증 체크포인트(Checkpoint)를 중심으로 전체 파이프라인을 재구성함으로써 실무적 범용성과 재현성을 확보하고자 하였다.

본 논문의 주요 기여는 다음과 같다.

(1) 모듈별 책임 분리와 독립 검증: 조건부 전역 탐지(Stage 1)와 하이브리드 그래프 기반 관계 진단(Stage 2)의 2-Stage 구조를 통해 탐지(What)와 진단(Why)의 책임을 명확히 격리하고, 각 모듈에 대한 독립적인 성능 검증(Independent Verification)을 가능하게 한다.

(2) 하이브리드 그래프 레시피 명세: 물리적 도메인 지식과 데이터 기반 근거를 체계적으로 결합하는 Hybrid Graph Recipe를 제시하여, 설명 후보 공간을 설계 단계에서 통제하고 타당성을 확보한다.

(3) 재현 가능한 설계 선택 근거 제시: Score Fusion 가중치 α 및 그래프 구성요소의 효과를 정밀한 민감도 분석으로 제시함으로써, 다양한 산업 도메인으로의 확장 시 근거 있는 설계 선택 지침을 제공한다.

2. 관련 연구 (Related Research)

다변량 시계열 이상탐지(TSAD) 연구는 크게 예측·재구성 기반, 그래프 기반, 그리고 생성 모델 기반 접근으로 분류할 수 있다[2,3]. TranAD[4]와 같은 Transformer 기반 모델은 시계열의 장기 의존성을 포착하여 전역 이상을 탐지하는 데 탁월한 성능을 보이나, 관계 붕괴의 구체적인 원인인 센서쌍(Relation)을 식별하여 설명하는 데에는 한계가 있다. 한편, GDN[5]이나 GAT[6] 기반의 그래프 접근법은 변수 간 구조적 의존성을 명시적으로 반영하지만, 실무 환경에서 그래프를 정의하는 기준(물리적 지식 vs 데이터 통계)과 이를 설명가능성 평가로 연결하는 구체적인 절차적 지침은 여전히 부족한 실정이다.

최근 마스크드 잠재 생성 모델과[7] 같이 설명가능성을 지향하는 TSAD 방법론들이 제안되고 있으나[8], 실제 산업 현장의 요구사항을 충족하기 위해서는 (i) 그래프 구성의 객관적 근거, (ii) 도출된 설명의 타당성 검증, (iii) 구성요소별 명확한 책임 분리가 병행되어야 한다. 특히 기존 연구들이 주로 특정 벤치마크 데이터셋에서의 단순 성능 비교에 집중해 온 결과, 상이한 공학 도메인 간 이동 시 개발자가 준수해야 할 표준화된 개발 및 검증 절차(Engineering Recipe)의 부재는 실무적 확산의 큰 걸림돌이 되고 있다.

본 연구는 이러한 간극을 메우기 위해 선행 연구된 X-RAD의 핵심 개념을 표준화된 '레시피' 형태로 재정식화하고, 성능·조기성·설명력이라는 다각도의 지표를 통합 검증 프레임워크 내에서 제시한다. 산업제어시스템(ICS)과 상태기반정비(CBM+) 도메인은 데이터 생성 메커니즘이 상이함에도 불구하고, 정상 데이터만으로 학습하고 이상 구간에서 성능을 평가한다는 비지도 학습의 공통 구조를 공유한다. 따라서 본 논문에서 제안하는 조건부 입력 처리, 하이브리드 그래프 구성, 모듈별 책임 분리와 같은 재현 가능한 설계 지침은 개발자가 새로운 도메인에 이상탐지 시스템을 구축할 때 겪는 불확실성을 최소화하고 효율적인 도메인 전이를 가능케 할 것이다.

3. 문제 정의 및 연구 질문 (Problem Definition & RQ)

본 연구에서 정의하는 이상탐지 시스템의 입력은 길이 L의 다변량 시계열 윈도우 $X_{t-L+1:t} \in \mathcal{R}^{L\times n}$와 시점 t에서의 운용 모드 및 환경 상태를 나타내는 조건 벡터 $C_t \in \mathcal{R}^c$ 로 구성된다. 시스템의 변수 간 상호작용은 그래프 $G = (V, E^*)$로 정의하며, 여기서 각 엣지 $(i,j) \in E^*$는 변수 i가 j에 미치는 인과적 혹은 물리적 영향을 나타내는 방향성 엣지이다.

시스템의 최종 출력물은 다음 세 가지 요소로 구성된다:

① 시점 t의 최종 이상 점수 A_t : 전역 패턴의 이탈을 포착하는 전역 점수 $s_1(t)$와 관계 붕괴의 정도를 나타내는 집계된 엣지 점수 $s_2(t)$를 Score Fusion 레이어에서 결합하여 산출한다.

② 경보 레이블 $alarm(t)$: 최종 점수 A_t가 사전에 결정된 운영 임계값 τ를 초과할 때 발생하는 이진 판정 결과이다 $(y_t = \mathrm{I}(A_t > \tau))$.

③ 원인 후보 엣지 집합 $E_t(K)$: 이상 경보 발생 시, 각 엣지별 이상 점수를 기준으로 가장 높은 기여도를 보이는 상위 K개의 관계 부분 집합이다 $(E_t(K) \subseteq E^*, |E_t(K)| = K)$.

모델의 학습은 실제 산업 현장의 특성을 반영하여 정상 데이터만을 활용하는 비지도(Unsupervised) 설정을 따른다. C-MAPSS 데이터셋에서는 각 유닛의 초기·중반 정상 구간만을 학습에 사용하며, SWaT 데이터셋은 공격 시나리오가 포함되지 않은 정상 운전 구간을 학습 세트로 활용한다.

본 연구는 제안하는 프레임워크의 공학적 타당성을 검증하기 위해 다음과 같은 세 가지 연구 질문(RQ)을 설정한다:

- RQ1 (탐지 성능 및 조기성) : 제안된 2-Stage 구조는 CBM+와 ICS(수처리 플랜트)라는 서로 다른 특성의 도메인에서 실용적인 수준의 탐지 성능과 조기 경보 능력을 동시에 달성할 수 있는가?
- RQ2 (설명가능성) : Stage 2의 진단 모듈이 제시하는 엣지 수준 설명은 실제 시스템의 물리적 구조(P&ID)와 얼마나 정렬되어, 최종 이상 점수의 변동에 실질적으로 기여하는가?
- RQ3 (구성요소 효과성 및 설계 근거) : Score

Fusion 가중치 α 및 하이브리드 그래프 구성 방식은 탐지 성능과 설명력에 어떠한 영향을 미치며, 실제 시스템 구축 시 엔지니어링 설계 선택의 근거를 어떻게 제공하는가?

4. X-RAD Engineering Recipe

본 절에서는 X-RAD를 '단순 AI 모델'이 아닌, 산업 현장에서 재현 가능한 개발·검증 절차(Engineering Recipe)로 정식화한다. 핵심 설계 원칙은 (i) Stage 1과 Stage 2의 책임 분리(Responsibility Decoupling), (ii) 하이브르드 그래프(E^*)의 체계적 버전 관리, (iii) 산출물(Artifact)과 회귀 방지 게이트(Regression Gate)의 표준화에 있다. 그림 1은 본 논문에서 제안하는 전체 파이프라인의 흐름을 요약한다.

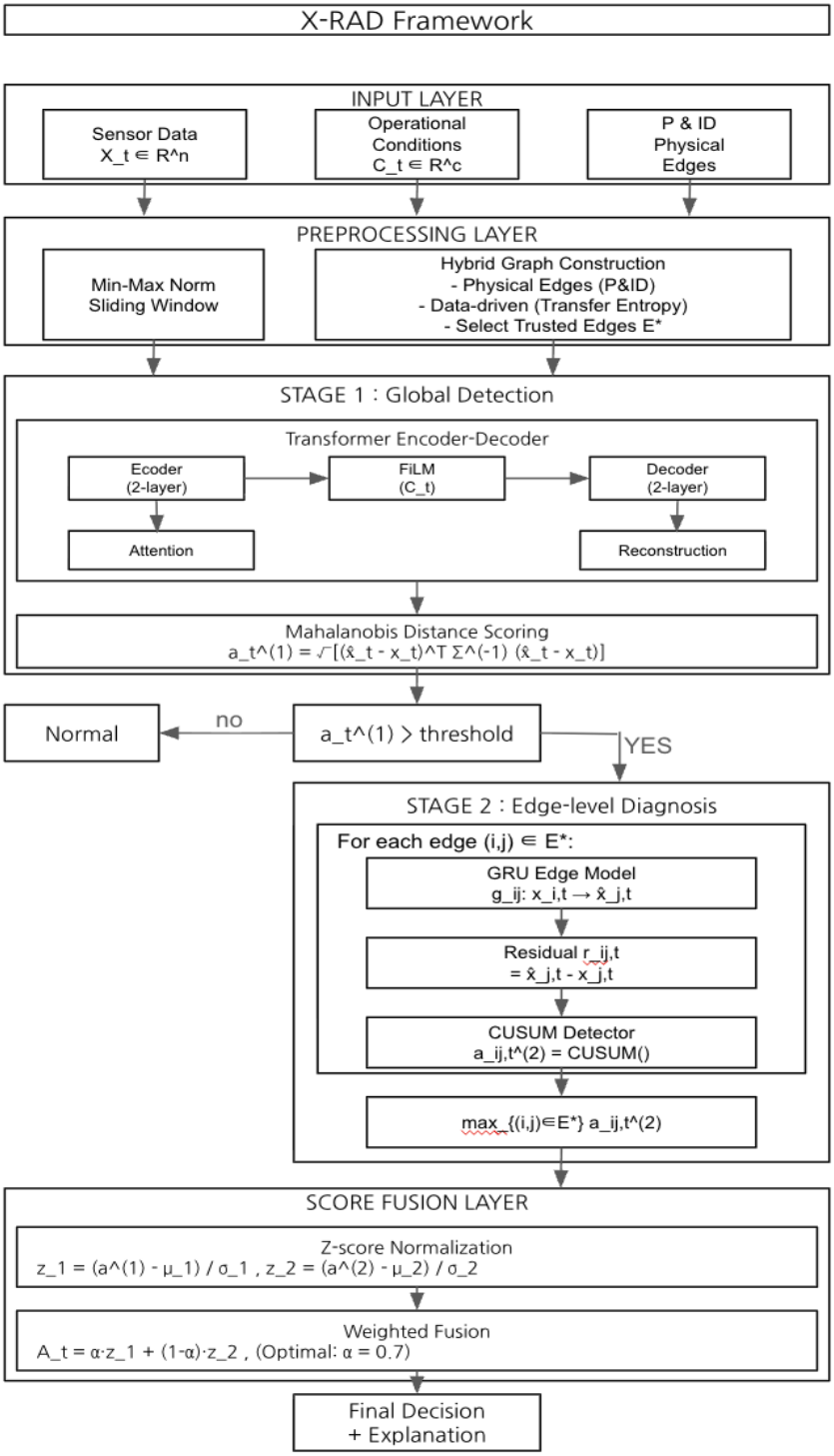


그림 1. X-RAD Overall Architecture. Stage 1(조건부 전역 탐지)은 전역 패턴 이탈(What)을 포착하고, Hybrid Graph Recipe로 정의된 E^*위에서 Stage 2(엣지별 관계 진단)가 원인 후보(Why)를 국소화한다.

4.1. Recipe Summary 및 공학적 원칙

본 레시피의 최상위 원칙인 책임 분리(Responsibility Decoupling)는 시스템의 유지보수성과 신뢰성을 극대화한다. Stage 1은 전역적 이상 징후 포착에만 전념하여 조기 경보 능력을 극대화하고, Stage 2는 구체적인 관계 붕괴를 규명함으로써 진단적 깊이를 더한다. 특히 이러한 책임 분리를 실질적인 검증 가능성으로 전환하기 위해, 본 레시피는 각 모듈이 독립적으로 통과해야 하는 '회귀 게이트(Regression Gates)'를 표준 공정 내에 정의한다. 표 1에 명시된 바와 같이, Stage 1의 탐지 신뢰성은 Stage 2의 진단 모듈 상태와 무관하게 S1-CP1~3 게이트를 통해 독자적으로 보증되며, 이는 모듈 간 의존성을 배제한 '독립 검증(Independent Verification)'을 실현하는 핵심적인 공학적 메커니즘으로 작용한다.

Algorithm 1은 시스템 구현자가 개발 및 운영 단계에서 준수해야 할 표준 운영 절차(Standard Operating Procedure)를 정의한다.

Algorithm: X-RAD Engineering Recipe (Standard Procedure)

- Data : 윈도우 길이 L 선정 및 정상 데이터 D_{normal} 기반 정규화 수행
- Graph : 물리지식(E_{phy})과 통계(E_{data})를 결합하여 버전 관리된 E^* 생성
- Stage 1 : 조건부 전역 탐지기 학습 및 전역 점수 $S_1(t)$ 산출
- Stage 2 : E^* 상의 각 엣지별 경량 모델 학습 및 엣지 점수 $S_{2,ij}(t)$ 산출
- Fuse : Z-score 정규화 후 가중치 a로 결합하여 최종 점수 A_t 생성
- Calib : 검증 세트에서 임계값 τ 결정 및 알람 정책 수립
- Explain : 알람 시 상위 K개 엣지 출력 및 설명 일관성 평가

표 1. Engineering Artifacts and Regression Gates (Verification Contract)

Component	Versioned artifacts (A)	Regression gates (CP, must-pass)
Stage 1 (What)	S1-A1: 학습 파라미터/seed S1-A2: 정상 잔차 통계(Σ 또는 대각 근사) S1-A3: s1(t) 기준선(분포/분위수) S1-A4: 조건별 요약 리포트	S1-CP1: 정상 안정성(분산/스파이크) S1-CP2: 조건 불변성(모드별 편향) S1-CP3: 민감도(이상 신호 주입 시 반응)
Hybrid Graph E^*	G-A1: E_phy 근거(규칙/도면 출처) G-A2: E_data 근거(통계량/윈도우) G-A3: 결합·희소화 규칙 G-A4: graph version ID	G-CP1: 희소성/계산량 제약 G-CP2: 근거 비율(물리 vs 데이터) 점검 G-CP3: 정상 설명 안정성(Top-K 변동) G-CP4: 버전 변경 회귀
Stage 2 (Why)	S2-A1: 엣지 모델 설정(특성/지연) S2-A2: s2_ij(t) 기준선 S2-A3: Top-K 로그 스키마(시간·엣지·점수)	S2-CP1: 정상 구간에서 과민 반응 억제 S2-CP2: 엣지 점수의 일관성(동일 조건 반복 S2-CP3: Top-K 재현성(버전/seed)
Fusion / Alarm	F-A1: α, z-normalization 범위 F-A2: τ 선택 절차(검증 기준) F-A3: alarm 이벤트 로그	F-CP1: α/τ 변경 시 성능 회귀 점검 F-CP2: 알람 빈도·지속성의 비정상 증가 감지(운영 안전장치)
Explain / Release	X-A1: 설명 산출물(Top-K, 근거) X-A2: 반사실 평가 설정(대체/마스킹) X-A3: 릴리스 체크리스트	X-CP1: 설명 정합성(도메인 규칙과 모순 여부) X-CP2: 감사지표 재현성(동일 입력 동일 설명) X-CP3: 운영 적용 전 테스트 케이스 통과

4.2. Stage 1: 조건부 전역 탐지 (Detection of 'What')

Stage 1은 윈도우 시계열 $X_{t-L+1:t}$와 운용 조건을 나타내는 벡터 C_t를 입력 받아 전역 이탈 점수 $S_1(t)$를 산출한다. 본 단계에서는 FiLM 모듈을 통해 모드 변화를 흡수하며, 재구성 오차 $e_t = \hat{x}_t - x_t$ 에 대한 마할라노비스 거리(Mahalanobis Distance)를 다음과 같이 계산한다:

$$s_1(t) = \sqrt{(\hat{x}_t - x_t)^T \sum^{-1} (\hat{x}_t - x_t)}$$

전역 탐지를 분리함으로써 센서 교체나 환경 변화에도 안정적인 기준서를 유지할 수 있으며, 이는 S1-CP(검증 체크포인트)를 통해 성능 회귀 여부를 관리한다.

4.3. Hybrid Graph Recipe: E^* 구축 및 버전 관리

Stage 2의 진단 후보 공간은 하이브리드 그래프 E^*로 정의된다. 이는 물리 기반 그래프 E_{phy}와 데이터 기반 상관 그래프 E_{data} 의 합집합으로 구성되며, 희소화(Sparsification) 규칙을 적용하여 설명의 집중도를 높인다:

$$E^* = E_{phy} \cup Top-K(E_{data})$$

하이브리드 구성은 물리적 구조의 안정성과 실데이터의 유연한 적응력을 유기적으로 결합하여 공정 내 미세한 관계 변화를 정밀하게 포착한다. 이때 도메인 전문가의 주관에 따른 지식 편향을 최소화하기 위해, 본 레시피는 물리적 인접성 자동 추출 가이드라인을 준수한다. 즉, 전문가가 임의로 엣지를 정의하는 대신 P&ID(공정 계통도) 상의 물리적 연결 경로를 인접 행렬(Adjacency Matrix)로 자동 변환하여 그래프의 골격(Skeleton)인 E_{phy}를 우선적으로 형성한다.

이후 전이 엔트로피(Transfer Entropy) 등을 통해 산출된 데이터 기반 상관성(E_{data}) 중 상위 K 개를 결합함으로써, 전문가 지식의 안정성과 데이터의 실증적 근거가 균형을 이루는 진단 공간을 확보한다. 구축된 E^*는 버전 ID(G-A4)와 함께 산출물로 관리되며, 물리 지식과 데이터 근거의 비율을 점검하는 근거 비율 게이트($G-CP2$)를 통과함으로써 최종적인 공학적 신뢰도를 확보한다.

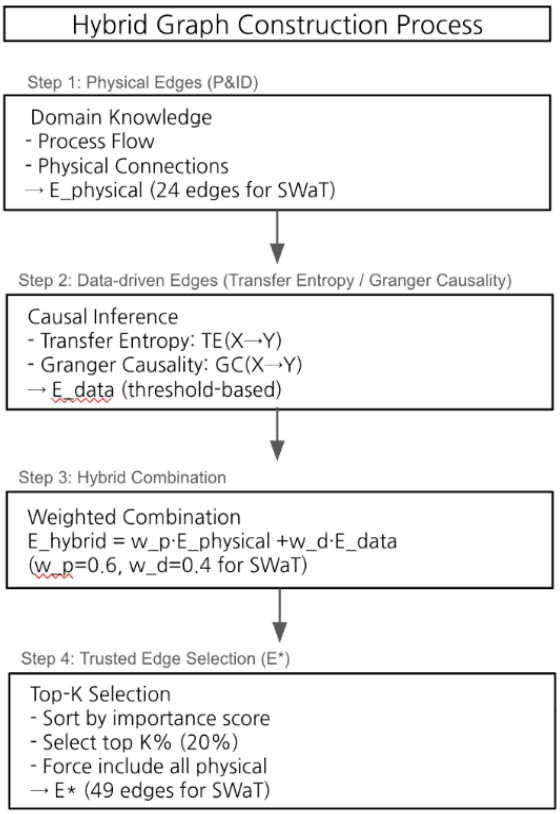


그림 2. Hybrid Graph Recipe. 물리적 연결과 통계적 인과성을 결합하여 설명 후보 공간을 설계한다.

4.4. Stage 2: 엣지별 관계 진단 (Diagnosis of 'Why')

Stage 2는 E^*의 각 엣지 (i,j)에 대해 관계 잔차 $r_{ij,t}$를 계산하고, CUDUM 통계를 통해 지속적인 관계 붕괴를 포착한다.

$$S_{(ij,t)} = \max\left(0, S_{ij,t-1} + \left|r_{(ij,t)}\right| - k\right)$$

최종 관계 점수 $s_2(t)$는 모든 엣지 점수의 최대값으로 정의되며, 이는 S2-CP를 통해 원인 규명의 신뢰도를 보장한다.

4.5. Score Fusion 및 알람 결정 정책

두 단계의 점수를 공통 스케일로 정규화한 뒤 가중 결합하여 최종 점수 를 산출한다:

$$A_t = az_1(t) + (1-a)z_2(t)$$

가중치 a 는 탐지(Stage 1)와 진단(Stage 2)간의 우선순위를 결정하는 공학적 설계 번수이며, 실험 결과 최적값은 $a = 0.7$로 확인되었다.

4.6. 설명 산출물 및 릴리스 게이트 (Release Gate)

경보 발생 시 상위 K개 엣지의 원인으로 지시하며, 반사실적 감소량(ΔCF)을 통해 설명의 타당성을 검증한다. 모든 산출물은 입력 데이터 및 그래프 버전과 함께 로깅되어, 운영 환경에서의 추적성(Traceability)을 확보한다.

5. 실험 설정(Experimental Setup)

본 절에서는 제안한 X-RAD 레시피의 유효성을 검증하기 위한 실험 환경과 평가 프로토콜을 기술한다. 본 연구는 복합 공학 시스템의 다중 도메인 적응성을 입증하기 위해 항공 엔진과 수처리 플랜트라는 서로 다른 특성의 데이터셋을 활용한다.

5.1. 데이터셋 구성 및 프로파일링

실험에는 항공 엔진의 수명 종료(run-to-failure) 시뮬레이션 데이터인 C-MAPSS와 실제 산업제어시스템 데이터인 SWaT를 사용한다. C-MAPSS는 운용 조건과 고장 모드가 복합적으로 구성된 FD002, FD003, FD004 서브셋을 선택하여 조기 경보 능력을 평가하며,[9,10] SWaT는 51개의 센서와 P&ID 기반의 24개 물리 엣지를 포함하고 있어 엣지 수준 설명가능성 검증에 최적화되어 있다[11].

표 2. 데이터셋 상세 명세 및 실험 환경

Ddataset	Type	Sensors	Samples	Train/Val/Test	Operating Conditions	Physical Edges
C-MAPSS, FD001	Turbofan	14	20,631	80/20/100 units	1	-
C-MAPSS, FD002	Turbofan	14	53,759	208/52/259 units	6	-
C-MAPSS, FD003	Turbofan	14	24,720	80/20/100 units	1	-
C-MAPSS, FD004	Turbofan	14	61,249	200/49/248 units	6	-
SWaT	Water Treatment	51	449,919	360K/90K/-	24	24

5.2. 전처리 및 학습 프로토콜

모든 실험은 정상 데이터만을 활용하는 비지도 이상탐지(Unsupervised AD) 설정으로 통일한다.

- 입력 처리 : 모든 입력 시계열은 길이 $L = 100$ 의 슬라이딩 윈도우로 분할되며, 데이터셋별로 계산 효율을 고려한 Stride를 적용한다.
- 정규화 : 학습 데이터의 통계량을 기준으로 한 Min-Max 정규화를 수행하여 모든 센서 값을 $[0,1]$구간으로 스케일링한다.
- 산출물 기록 : 재현성을 위해 전처리 파이프라인 버전, 그래프 버전(E^*), 학습 설정(Seed, Epoch, LR) 등을 '실험 ID'와 함께 패키징하여 관리한다.

5.3. 평가 지표: 조기성, 정확도, 설명가능성

본 연구는 단순 정확도를 넘어 실무적 유용성을 평가하기 위해 다음과 같은 지표를 선정한다.

- NAB-score: 조기 경보에 가중치를 부여하는 시간 가중 점수로, 목표치는 70 이상으로 설정한다.
- AUPRC: 데이터 불균형이 심한 환경에서 임계값 변화에 따른 정밀도-재현율 균형을 평가하며, 목표치는 0.5 이상이다.
- Lead Time: 실제 고장 시점 대비 선행 경보 제공 시간을 step 단위로 측정한다.

5.4. 계산 복잡도 및 적용 고려사항

레시피의 실무 적용을 위해 모델의 계산 비용을 분석한다.

- Stage 1 : Transformer 구조에 따라 $O(L^2 d)$ 의 복잡도를 가지며, $d = 256$ 설정을 통해 전역 패턴을 안정적으로 학습한다.
- Stage 2 : 엣지 수 $|E^*|$에 비례하여 $O(|E^*| \cdot L \cdot h)$로 증가하므로, 하이브리드 그래프의 희소화(Top-K%)를 통해 연산량을 제어한다. 이러한 모듈 책임 분리는 운영 대시보드에서 Stage 1을 '시스템 건강도' 지표로, Stage 2의 $Top - K$를 '트러블슈팅 가이드'로 이원화하여 제공할 수 있는 공학적 이점을 제공한다.

6. 결과 (Results)

본 절에서는 5절에서 설정한 실험 환경을 바탕으로 연구 질문 RQ1~RQ3에 대한 정량적·정성적 검증 결과를 제시한다. 모든 실험 수치는 비지도 학습 설정 하에 도출되었으며, 제안한 'Engineering Recipe'의 구성 요소들이 탐지 성능과 설명력에 미치는 효과를 분석한다.

6.1. RQ1: 탐지 성능 및 조기성 (Detection & Early Warning)

X-RAD의 전체 탐지 성능 요약은 표 3와 같다.

- 정량적 성능 및 목표 달성 검토 : 분석 결과, X-RAD는 모든 평가 데이터셋에서 NAB-score 91.64 이상을 기록하여 목표치(70)를 최대 32% 상회하는 우수한 탐지 능력을 입증하였다. 특히 불균형 데이터 환경에서도 평균 AUPRC 0.5295를 달성하여 탐지 결과의 신뢰성을 확보하였으며, 이는 제안된 레시피가 복합 공학 시스템의 비지도 학습 환경에서 실용적인 탐지 기준을 만족함을 시사한다.
- 설계 의도 기반 조기 경보 효과 분석 : 본 연구의 핵심 지표인 조기 경보 능력(Early Warning) 측면에서, C-MAPSS 데이터셋 전체에서 평균 -18.7 step의 유의미한 Lead Time을 확보하였다. 특히 FD002에서 나타난 -29.73 step의 독보적인 선행 경보 성과는 본 논문의 Stage 1이 센서 간 공분산($\sum$)을 반영함으로써 전역적 이탈을 포착한 결과이다. 이는 4절에서 상술한 '조기성 확보를 위한 전역 패턴 학습' 설계 의도가 실제 물리적 이상 징후의 초기 확산 과정을 마할라노비스 거리 기반의 잔차 분석으로 정확히 포착했음을 방증한다.
- 성능 한계 및 후속 과제 (FD003) : 반면, FD003 서브셋에서 관찰된 상대적으로 낮은 AUPRC(0.3687)는 본 모델이 가진 기술적 한계점을 명확히 시사한다. 이는 복수 고장 모드와 복잡한 열화 패턴이 단일 이상 점수 축 상에서 중첩될 때 발생하는 분별력 저하에 기인하며, 이러한 결과는 7절에서 논의할 '고장 유형별 서브모델 설계' 및 '조건 벡터의 정교화' 연구의 필요성을 뒷받침하는 핵심적인 공학적 근거로 기능한다.

표 3. Overall Detection Performance (RQ1)

Dataset	NAB-score	Range-F1	AUPRC	Lead Time
C-MAPSS FD002	94.65	0.7025	0.6619	-29.73
C-MAPSS FD003	91.64	0.6023	0.3687	-20.55
C-MAPSS FD004	94.20	0.6589	0.5974	-5.82
SWaT	99.76	0.6800	0.7400	-
Average	92.57	0.6240	0.5295	13.20
Target	> 70	-	> 0.5	-
Achievement	+ 32%	-	+6%	+70% improved

6.2. RQ2: 설명가능성 검증 (Explainability)

Stage 2가 산출한 엣지 수준의 진단 결과가 실제 시스템의 물리적 인과 관계를 얼마나 충실히 반영하는지 검증한다. SWaT 데이터셋을 대상으로 물리 그래프 정합성과 반사실적 기여도를 평가한 결과는 표 4와 같다.

표 4. Explainability Results on SWaT (RQ2)

Metric	Value	Target	Interpretation
Edge-Precision@5	0.45	Measure	45% of top-5 match GT
Edge-Precision@10	0.42	Measure	Consistent accuracy
Fidelity (ΔCF)	0.68	Measure	High impact on score
GT Coverage	24/24	100%	All physical edges used

- 정합성 및 기여도: Edge-Precision@5 기준 0.45를 기록하여 모델이 지목한 이상 관계가 실제 공정 구조(P&ID)와 높은 수준으로 일치함을 확인하였다. 특히 Fidelity(ΔCF)가 0.68로 나타난 것은, Stage 2가 지목한 이상 엣지의 잔차를 정상화할 경우 최종 이상 점수가 유의미하게 감소함을 의미하며, 이는 도출된 설명이 경보의 근거로서 실질적인 신뢰도를 확보했음을 방증한다.
- 책임 분리의 이점: 실험적으로 탐지 모델(Stage 1)은 고정한 채 하이브리드 그래프 레시피(E^*)만을 교체하여 설명 품질(RQ2)을 독립적으로 개선함으로써 구조적 격리 검증의 유효성을 확인하였다. 이를 통해 탐지 지표와 설명 지표를 독립적으로 모니터링하고, 탐지 성능의 저하 없이 진단 품질만을 단계적으로 최적화할 수 있는 공학적 유연성을 확보하였다.

6.3. RQ3: 구성요소 효과성 분석 (Ablation Study)

점수 융합 가중치 α 와 하이브리드 그래프 구조가 시스템에 미치는 영향을 분석한다.

- 민감도 분석: 표 5에서 보듯 $\alpha = 0.7$ 일 때 탐지 성능(NAB-score)과 불균형 견고성(AUPRC)이 최적의 균형을 이룬다. 특히 Stage 2를 완전히 배제한 $\alpha = 1.0$ 설정과 비교했을 때, 최종 결합 모델은 NAB-score를 유지하면서도 AUPRC를 유의미하게 개선하였다. 이는 Stage 2의 관계 진단 모듈이 양성 구간의 구분력을 강화하여 경보의 신뢰도를 높이는 핵심 요소임을 입증한다.

표 5. Alpha Sensitivity Analysis for Score Fusion (RQ3)

Alpha	Stage Ratio	NAB-score	Range-F1	AUPRC	Lead Time	Interpretation
0.3	70% S1 + 30% S2	88.84	0.4706	0.5139	+8.23	Low, but positive lead
0.4	60% S1 + 40% S2	90.53	0.5231	0.5999	-1.17	Improving
0.5	50% S1 + 50% S2	91.89	0.6107	0.6225	-13.23	Baseline
0.6	40% S1 + 60% S2	92.69	0.6397	0.5573	-25.48	Better
0.7	30% S1 + 70% S2	94.65	0.7025	0.6619	-29.73	Optimal
0.8	20% S1 + 80% S2	93.88	0.6525	0.5202	-25.52	Slight decrease
1.0	100% S1 (No S2)	94.68	0.6991	0.5120	-29.37	No Stage

- 그래프 구조 효과: SWaT 실험 결과, 하이브리드(Hybrid) 그래프를 적용했을 때 물리 전용(Physical-only) 구성 대비 Range-F1이 36% 향상되었다. 이는 도메인 지식이 제공하는 구조적 안정성과 데이터 기반 패턴이 포착하는 유연한 적응성이 상보적으로 작용하여 복합적인 이상 유형에 효과적으로 대응했음을 의미한다.

6.4. 성공 및 실패 사례 심층 분석 (Case Study)

X-RAD 레시피가 실제 공정 시나리오에서 갖는 실무적 거동을 분석한다.

- 성공 사례 (FD004 및 SWaT): 복합적인 운용 조건 변화가 포함된 FD004 사례에서, Stage 1은 전역적 패턴 이탈을 조기에 감지하여 -5.82 step의 유의미한 선행 경보를 제공하였다. 또한, Stage 2는 미세한 관계 붕괴를 포착하여 경보의 신뢰도를 높였으며, 특히 SWaT 환경에서는 P&ID(공정 계통도) 상의 물리적 연결과 정렬된 엣지 설명이 도출되어(Edge-Precision@5=0.45) 운영자가 즉각적인 트러블슈팅을 수행할 수 있는 객관적 근거를 제공함을 확인하였다.
- 한계 사례 (FD003의 고장 모드 중첩): FD003 서브셋에서 AUPRC가 상대적으로 낮게 나타난 현상을 분석한 결과, 서로 다른 고장 모드가 전역 이상 점수 상에서 유사한 분포를 보여 모델의 구분력이 저하됨을 확인하였다. 이는 본 레시피의 Stage 1이 가진 기술적 해상도의 한계인 동시에, 7절에서 논의할 '고장 유형별 서브모듈 설계'의 필요성을 강력히 뒷받침하는 공학적 근거가 된다.
- 운영 전환을 위한 검증: 각 경보 시점에 생성된 Top-K 엣지 로그는 단순한 결과 보고를 넘어, 시간이 지남에 따라 축적되어 시스템의 '취약 관계 데이터베이스'로 자산화될 수 있음을 확인하였다. 이는 4절에서 제안한 회귀 게이트(Regression Gate)가 실무적으로 필수적인 설계 요소임을 입증한다. 특히 SWaT 실험에서 달성한 GT

Coverage 24/24(100%)는 운영자가 수십 개의 센서를 전수 조사하는 대신 모델이 지목한 상위 엣지를 즉각 점검하게 함으로써, 실제 현장의 평균 수리 시간(MTTR)을 획기적으로 단축하고 운영자의 인지 부하를 경감시킬 수 있는 핵심적인 실무적 효익을 제공한다.

7. 논의 및 위협요인 (Discussion & Threats to Validity)

본 연구를 통해 도출된 실험 결과는 복합 공학 시스템에서 X-RAD 레시피의 유효성을 입증함과 동시에 모델의 기술적 해상도에 대한 중요한 시사점을 제공한다. 우선 FD003 데이터셋에서 관찰된 상대적으로 낮은 AUPRC 성능은 다중 고장 모드와 복잡한 열화 패턴이 동일한 전역 잔차 분포 내에서 중첩될 때 발생하는 단일 점수 축 모델링의 근본적인 한계를 드러낸다 . 이는 향후 고장 유형별 mixture 모델이나 계층적 인코딩 구조를 도입하여 이상 징후의 구분력을 강화해야 할 필요성을 뒷받침하는 핵심적인 공학적 근거가 된다.

실무 적용 관점에서는 데이터 전처리 단계에서의 윈도우 Stride 선정이 시스템의 조기성과 계산 복잡도 사이의 전형적인 트레이드오프 관계를 형성함을 인지해야 한다. Stride를 작게 설정할수록 탐지 해상도가 향상되어 Lead Time을 극대화할 수 있으나, Stage 1의 $O(L^2 d)$ 및 Stage 2의 $O(|E^*| \cdot L \cdot h)$ 연산 부하가 증가하여 실시간 운영 인프라에 부담을 줄 수 있기 때문이다.

또한 하이브리드 그래프(E^*) 구축 과정에서 도메인 전문가의 지식 편향이 개입될 경우 진단 결과의 객관성에 영향을 미칠 수 있는 내부 타당성의 위협이 존재하며, 이는 실제 운영 환경에서 설명에 대한 해석의 주관성 문제로 이어질 수 있다. 이를 완화하기 위해 본 레시피는 표 1의 G-CP2(근거 비율 점검) 게이트를 통해 물리 지식과 데이터 근거의 균형을 정량적으로 검증하는 체크리스트를 포함한다. 마지막으로 오경보에 따른 운영 피로도와 미탐지로 인한 사고 기회비용 사이의 절충은 단순한 수치 이상의 공학적 판단을 요구하므로, 본 레시피의 알람 정책은 운영자의 안전 보수성을 최우선으로 고려하여 교정되어야 함을 강조한다.

8. 결론 (Conclusion)

본 논문은 산업제어시스템 및 항공 엔진과 같은 복합 공학 시스템의 이상탐지를 위해 하이브리드 그래프 구축과 Two-Stage Responsibility Decoupling을 골자로 하는 'X-RAD Engineering Recipe'를 제안하였다 . 비지도 학습 환경에서 C-MAPSS 및 SWaT 데이터셋에 대해 평가한 결과, 평균 NAB-score 92.57과 AUPRC 0.5295를 달성하여 탐지 성능과 조기 경보 능력을 동시에 입증하였으며, 실제 고장 시점 대비 평균 -18.7 step의 선행 경보를 제공하는 실무적 유효성을 확인하였다.

이러한 수치적 성과를 넘어, 본 연구는 AI 모델의 단순 성능 개선을 탈피하여 산출물과 검증 게이트를 표준화함으로써 산업 현장에서 재현 가능한 개발 및 운영 절차를 정식화했다는 점에서 독보적인 공학적 가치를 지닌다. 특히 본 레시피의 산출물은 운영자의 인지 부하를 줄이고 트러블슈팅의 표준 가이드로 활용될 수 있어, AI 모델의 현장 운영 전환(Deployment)을 가속화한다. 향후에는 본 논문에서 제안된 정적 그래프 구조를 온라인 학습 환경으로 확장하여 환경 변화에 동적으로 적응하는 하이브리드 그래프 업데이트 메커니즘을 개발함으로써 시스템의 장기적인 신뢰성을 더욱 강화할 계획이다.

참고 문헌

[1] 김민지, 허대영 "X-RAD: Relation-Aware, Edge-Explainable Anomaly Detection for Industrial Control Systems," Proc. 한국소프트웨어종합학술대회 (KSC), -, -, pp. 244-246, 2025.

[2] Z. Zamanzadeh Darban, et al., "Deep learning for time series anomaly detection: A survey," ACM Computing Surveys, Vol. 57, No. 1, pp. Article 15, 2024.

[3] F. Wang, et al., "A survey of deep anomaly detection in multivariate time series: Taxonomy, applications, and directions," Sensors, Vol. 25, No. 1, pp. 190, 2025.

[4] S. Tuli, G. Casale, and N. R. Jennings, "TranAD: Deep Transformer Networks for Anomaly Detection in Multivariate Time Series Data," Proc. VLDB Endowment, Vol. 15, No. 6, pp. 1201-1214, 2022.

[5] A. Deng, and B. Hooi, "Graph neural network-based anomaly detection in multivariate time series," Proc. AAAI Conf. Artificial Intelligence, Vol. 35, No. 5, pp. 4027-4035, 2021.

[6] H. Zhao, et al., "Multivariate time-series anomaly detection via graph attention network," Proc. IEEE ICDM 2020, -, -, pp. 841-850, 2020.

[7] D. Lee, S. Malacarne, and E. Aune, "Explainable time series anomaly detection using masked latent generative modeling," Pattern Recognition, Vol. 156, -, pp. 110826, 2024.

[8] Y. Wan, D. Zhang, D. Liu, and F. Xiao, "CGAD: A contrastive learning-based framework for anomaly detection in attributed networks," Neurocomputing, Vol. 609, -, pp. 128379, 2024.

[9] A. Saxena, and K. Goebel, "Turbofan Engine Degradation Simulation Data Set," NASA Prognostics

Data Repository, -, -, -, 2008.

[10] A. Saxena, K. Goebel, D. Simon, and N. Eklund, "Damage Propagation Modeling for Aircraft Engine Run-to-Failure Simulation," Proc. PHM'08, -, -, -, 2008.

[11] A. P. Mathur, and N. O. Tippenhauer, "SWaT: A water treatment testbed for research and training on ICS security," Proc. CySWater 2016, -, -, pp. 31-36, 2016.

검색 전략이 **LLM** 기반 버그 리포트 자동 생성 성능에 미치는 영향 분석

최서진[o1], 양근석[2†]

[1] 한경국립대학교 컴퓨터응용수학부, [2] 한경국립대학교 컴퓨터응용수학부(컴퓨터시스템연구소)

[1] insui12@hknu.ac.kr, [2] gsyang@hknu.ac.kr

An Analysis of the Impact of Retrieval Strategies on LLM-Based Automated Bug Report Generation

Seojin Choi[o1], Geunseok Yang[2†]

[1] Department of Computer Applied Mathematics, Hankyong National University,

[2] Department of Computer Applied Mathematics(Computer System Institute), Hankyong National University

[1] insui12@hknu.ac.kr, [2] gsyang@hknu.ac.kr

요 약

버그 리포트는 소프트웨어 결함을 이해하고 재현하기 위한 핵심 산출물이지만, 실제 개발 환경에서는 작성자의 숙련도와 서술 방식 차이로 인해 품질 편차가 크게 발생한다. 최근 대규모 언어 모델(LLM)을 활용한 버그 리포트 자동 생성 기법이 제안되고 있으나, 단일 버그 설명에 의존하는 생성 방식은 입력 문맥의 제약으로 인해 구조적 완성도와 내용 정합성 측면에서 한계를 보인다. 본 논문은 검색 기반 문맥 보강(Retrieval-Augmented Generation, RAG) 환경에서 검색 전략의 선택과 구성 방식이 LLM 기반 버그 리포트 자동 생성 성능에 미치는 영향을 체계적으로 분석한다. 키워드 기반 검색(BM25), 의미 기반 검색(SBERT), 하이브리드 검색 전략을 동일한 생성 조건 하에서 비교하고, 재정렬 기법을 포함한 확장 설정을 함께 평가하였다. Bugzilla 기반 데이터셋(3,966쌍)과 세 가지 공개 LLM(Qwen2.5-7B, LLaMA-3.2-3B, Mistral-7B)을 대상으로 수행한 실험 결과, 검색 기반 문맥 보강은 베이스라인 대비 CTQRS(구조적 완성도)를 약 14-16%p, ROUGE-1 Recall(어휘적 정보 포괄성)을 약 19-21%p 향상시키는 경향을 보였다. SBERT 유사도(의미적 정합성) 역시 약 4-6%p 개선되었으며, 하이브리드 검색은 다양한 설정에서 비교적 일관된 성능 특성을 나타냈다. 반면 예시 수 증가에 따라 ROUGE-1 F1(어휘적 균형)이 저하되는 상충 관계가 관찰되었고, 재정렬 기법의 추가 효과는 어휘적 균형 측면에 한정되었다. 이러한 결과는 검색 전략이 LLM 기반 버그 리포트 자동 생성에서 품질 균형을 결정하는 중요한 설계 요소임을 보인다.

1. 서론

버그 리포트는 소프트웨어 개발 및 유지보수 과정에서 결함을 식별하고 추적하기 위한 핵심 산출물이다. 재현 절차, 실행 환경, 기대 결과와 같은 정보가 명확히 제공될 경우 개발자는 결함의 원인을 보다 신속하게 파악할 수 있으나, 실제 환경에서는 비전문 사용자가 작성하는 경우가 많아 핵심 정보 누락이나 비구조적 서술이 빈번히 발생한다 [1]. 이로 인해 결함의 재현과 분석에 추가적인 노력이 요구되며, 이는 수정 지연과 유지보수 비용 증가로 이어질 수 있다.

이러한 문제는 Bugzilla [2]와 같은 오픈 소스 이슈 추적 시스템뿐만 아니라 Jira [3]와 같은 상용 플랫폼에서도 관찰되며, 저품질 버그 리포트가 결함 분석 시간 증가와 반복적인 커뮤니케이션을 유발하여 유지보수 효율성을 저해한다는 점이 기존 연구들에서도 반복적으로 보고되었다 [1,4]. 이러한 배경에서 비정형 버그 리포트를 자동으로 구조화하고 누락된 정보를 보완하는 기술은 소프트웨어 공학 분야의 중요한 연구 과제로 인식되어 왔다.

최근 대규모 언어 모델(Large Language Model, LLM)의 발전은 저품질 버그 리포트로 인한 유지보수 문제를 완화할 수 있는 가능성을 제시하고 있다. LLM은 높은 문맥 이해와 자연어 생성 능력을 바탕으로, 지시형 프롬프트를 통해 요약, 재현 절차, 기대 결과와 같은 핵심 요소를 포함하는 구조화된 버그 리포트를 자동으로 생성할 수 있다 [5], [6].

기존 LLM 기반 자동 생성 연구들은 주로 지시형 프롬프트와 LoRA 기반 미세조정을 결합한 파이프라인을 제안하였다 [5]. 이러한 접근은 단일 버그 요약만으로도 형식적으로 완전한 리포트를 생성할 수 있음을 보였으나, 입력 문맥이 단일 보고서에 한정되어 실제 환경에서 나타나는 정보 불완전성과 다양한 실행 맥락을 충분히 반영하지 못할 가능성이 있다 [7]. 그 결과, 구조는 완전하지만 결함의 맥락이나 변형된 재현 조건을 충분히 포착하지 못한 리포트가 생성될 수 있다.

이러한 한계를 보완하기 위해, 최근 자연어 처리 분야에서는 검색 기반 문맥 보강(Retrieval-Augmented Generation, RAG) 전략이 널리 활용되고 있다 [8]. RAG는 입력 텍스트와 유사한 외부 문서나 사례를 검색하여 생성 모델의 입력 문맥으로 제공함으로써, 단일 입력에 의존하는 생성 방식의 한계를 완화한다. 특히 버그 리포트 도메인은 과거 유사 결함 사례가 풍부하게 축적되어 있어 RAG가 효과적으로 적용될 수 있는 특성을 가진다.

그러나 기존 연구들은 RAG를 주로 성능 향상을 위한 보조 기법으로 취급하며, 검색 전략의 선택과 구성 방식이 구조적 완성도, 어휘적 정보 포괄성, 어휘적 균형, 의미적 정합성 간의 trade-off에 어떠한 영향을 미치는지는 충분히 분석하지 않았다. 즉, RAG의 적용 여부뿐 아니라 검색 전략의 설계 방식 자체가 생성 품질의 특성과 균형에 영향을 줄 수 있음에도, 이에 대한 체계적인 실증 분석은 아직 제한적이다.

본 연구는 검색 기반 문맥 보강 환경에서 검색 전략을 단순한 보조 기법이 아니라 생성 품질의 특성과 그 균형에 영향을 미치는 설계 요소로 보고, 서로 다른 검색 전략과 구성 선택이 LLM 기반 버그 리포트 자동 생성에서 구조적 완성도, 어휘적 정보 포괄성, 어휘적 균형, 의미적 정합성 간의 trade-off를 어떻게 형성하는지를 분석한다. 구체적으로 BM25, SBERT, 하이브리드 검색 전략과 재정렬 기법을 동일한 생성 조건 하에서 비교하여, 각 선택이 생성 품질에 미치는 영향을 체계적으로 평가한다.

실험은 LoRA 기반으로 미세조정된 대규모 언어 모델을 사용하여 수행되었으며 [9], few-shot 문맥 보강 설정을 통해 예시 수 변화가 생성 품질에 미치는 영향을 분석하였다. 생성된 버그 리포트의 품질은 구조적 완성도를 평가하는 CTQRS [10], 어휘적 포괄성과 어휘적 균형을 각각 반영하는 ROUGE-1 Recall 및 ROUGE-1 F1 [11], 그리고 의미적 정합성을 측정하는 SBERT 유사도를 통해 다각도로 평가하였다 [12].

실험 결과, 검색 기반 문맥 보강은 검색을 사용하지 않는 생성 방식과 비교하여 구조적 완성도와 어휘적 정보 포괄성을 전반적으로 향상시키는 경향을 보였다. 하이브리드 검색 전략은 다양한 설정에서 비교적 안정적인 성능 분포를 나타냈으나, 예시 수 증가에 따라 어휘적 균형과 의미적 정합성이 저하되는 경향이 관찰되어 검색 전략 설계에 따른 trade-off가 확인되었다. 한편 재정렬 기법을 적용한 확장 설정은 의미적 정합성 측면에서는 제한적인 추가 효과를 보였으나, 전반적인 품질 향상은 일관되지 않았다.

* 이 논문은 한경국립대학교 국립대학육성사업(2025)지원을 받아 작성되었음

† 교신저자(Corresponding Author)

본 연구는 단일 최적 기법을 제안하기보다는, 검색 기반 문맥 보강 전략의 설계 선택이 생성 품질의 여러 측면에 어떠한 영향을 미치는지를 분석하고 각 접근법의 특성과 한계를 비교하는 데 목적이 있다. 이를 통해 구조적 완성도, 어휘적 정보 포괄성, 어휘적 균형, 의미적 정합성 간의 상충 관계를 정량적으로 정리하고, 향후 LLM 기반 버그 리포트 자동 생성 연구에서 실험 설정과 결과 해석을 위한 참고 기준을 제공하고자 한다.

본 연구의 주요 기여는 다음과 같다.

- 검색 전략 비교 분석: 검색 기반 문맥 보강 환경에서 키워드 기반 검색(BM25), 의미 기반 검색(SBERT), 하이브리드 검색 전략을 동일한 생성 조건 하에서 비교함으로써, 검색 전략 선택이 LLM 기반 버그 리포트 자동 생성 품질에 미치는 영향을 정량적으로 분석하였다. 모든 검색 전략은 베이스라인 대비 CTQRS를 약 14-16%p, ROUGE-1 Recall을 약 19-21%p 향상시켜 구조적 완성도와 어휘적 정보 포괄성 측면에서 일관된 개선 효과를 보였다.
- 검색 전략별 trade-off 규명: 하이브리드 검색 전략은 다양한 예시 수 설정과 서로 다른 언어 모델 전반에서 상대적으로 안정적인 성능을 유지하였으나, 예시 수 증가에 따라 ROUGE-1 F1과 SBERT 유사도가 감소하는 경향이 관찰되었다. 이를 통해 구조적 완성도와 어휘적 정보 포괄성, 어휘적 균형, 의미적 정합성 간의 상충 관계를 실증적으로 확인하였다.
- 재정렬 기법의 한계 분석: 검색 결과에 재정렬 기법을 적용한 확장 설정에 대해 paired bootstrap 분석을 수행한 결과, 어휘적 지표에서는 일관된 성능 향상이 나타나지 않았으며, 의미적 정합성 측면에서만 제한적인 추가 효과가 확인되었다. 이는 생성 품질 개선의 핵심 요인이 재정렬 단계보다는 검색 전략의 선택과 구성에 있음을 보인다.

2. 배경지식

2.1 버그 리포트 품질

버그 리포트는 소프트웨어 유지보수 과정에서 결함을 식별·재현·수정하기 위한 핵심 정보 원천이며, 재현 절차, 실행 환경, 기대 결과 및 실제 결과의 충실도는 결함 분석의 효율성과 직접적으로 연결된다 [1]. 그러나 실제 개발 환경에서는 보고자의 경험 수준과 문서화 습관, 사용 도구의 차이로 인해 버그 리포트 품질에 큰 편차가 발생한다.

이러한 문제는 다양한 기여자가 참여하는 오픈 소스 프로젝트에서 특히 두드러지며, 많은 리포트가 증상 설명에 치중한 반면 재현 절차나 실행 환경과 같은 핵심 정보는 누락되거나 모호하게 기술되는 경우가 많다 [1]. 그 결과 개발자는 동일한 결함에 대해서도 추가 확인과 반복적인 커뮤니케이션을 수행해야 하며, 이는 분석 시간 증가와 결함 수정 지연으로 이어진다.

기존 실증 연구들은 저품질 버그 리포트가 소프트웨어 유지보수 효율성에 부정적인 영향을 미친다는 점을 반복적으로 보고해 왔다 [1]. 구조적으로 충실한 리포트는 명확한 재현 절차와 환경 정보를 제공하여 결함 재현 가능성을 높이는 반면, 핵심 요소가 누락된 리포트는 추측에 기반한 분석과 추가 비용을 유발한다. 이는 버그 리포트 품질이 문서 완성도를 넘어 실제 개발 생산성과 직결된 문제임을 보인다.

이러한 품질 차이를 정량적으로 분석하기 위해 CTQRS는 핵심 구성 요소의 구조적 포함 정도를 평가하는 지표로 활용되어 왔으며 [10], ROUGE 계열 지표는 어휘적 정보 포괄성을 [11], SBERT 유사도는 의미적 정합성을 평가하는 데 사용되어 왔다 [12]. 이러한 지표들은 버그 리포트 품질을 구조적·어휘적·의미적 관점에서 분석하는 도구로 활용되고 있다.

그러나 품질 평가 지표만으로는 실제 유지보수 과정에서 발생하는 문제를 근본적으로 해결하기에는 한계가 있다. 저품질 리포트를 식별할 수는 있으나, 누락된 정보를 보완하고 리포트를 재구성하는 과정은 여전히 개발자의 수작업에 의존하는 경우가 많으며, 대규모 프로젝트 환경에서는 이러한 한계가 더욱 두드러진다.

이러한 맥락에서 최근에는 비정형 버그 리포트를 구조화된 형태로 자동 변환하거나 보완하려는 자동화된 접근의 필요성이 지속적으로 제기되고 있다 [5], [6]. 특히 정보 불완전성을 내포한 입력 리포트를 대상으로 구조적 완성도와 내용 충실성을 동시에 확보하는 자동 생성 기법은 유지보수 효율성 향상을 위한 중요한 연구 과제로 인식되고 있다.

2.2 LLM 기반 버그 리포트 자동 생성의 한계: 입력 문맥의 부족

최근 자연어 처리 기술의 발전과 함께, 대규모 언어 모델(LLM)을 활용한 버그 리포트 자동 생성 연구가 활발히 이루어지고 있다 [5], [6]. 이러한 접근은 비정형 버그 리포트를 입력으로 받아 요약, 재현 절차, 기대 결과 및 실제 결과와 같은 핵심 요소를 구조화된 형태로 변환하는 것을 목표로 한다. 특히 instruction fine-tuning [13], [14]과 구조화된 프롬프트 설계를 적용한 연구들은 일정 수준 이상의 구조적 완성도와 의미적 정합성을 갖춘 리포트를 자동으로 생성할 수 있음을 실험적으로 보여주었다 [5], [6]. 이는 LLM 기반 자동 생성이 실질적인 적용 가능 단계에 도달했음을 보인다.

그러나 기존의 LLM 기반 자동 생성 접근들은 공통적으로 입력으로 제공되는 단일 버그 리포트에 강하게 의존하는 특성을 갖는다 [5], [6]. 즉, 생성 과정에서 활용되는 정보의 범위는 하나의 비정형 보고서로 제한되며, 해당 보고서에 결함 재현과 분석에 필요한 핵심 정보가 충분히 포함되어 있다는 가정을 전제로 한다. 이러한 가정은 정제된 데이터셋이나 숙련된 개발자가 작성한 리포트에서는 비교적 성립할 수 있으나, 실제 개발 환경에서는 일반적으로 충족되기 어렵다.

현실의 버그 리포트는 다양한 수준의 정보 불완전성을 내포하고 있다. 다수의 리포트는 결함의 증상이나 결과를 중심으로 서술되며, 재현 절차가 불완전하거나 실행 환경 정보가 누락되는 경우가 빈번하다. 또한 기대 결과와 실제 결과가 명확히 구분되지 않거나 혼재되어 기술되는 사례도 자주 관찰된다 [1], [4]. 이러한 특성은 특히 오픈 소스 프로젝트와 같이 기여자의 배경과 숙련도가 다양한 환경에서 더욱 두드러진다. 이로 인해 단일 리포트만을 입력으로 사용하는 자동 생성 모델은 구조적으로는 완전한 형식을 갖춘 결과를 생성하더라도, 실제 결함 맥락을 충분히 반영하지 못한 리포트를 산출할 가능성이 존재한다.

이러한 한계는 자동 생성된 버그 리포트의 품질 평가 결과에서도 확인된다. 구조적 요소의 포함 여부를 중심으로 평가하는 지표에서는 비교적 높은 점수를 획득할 수 있으나, 생성된 내용이 참조 리포트의 핵심 맥락을 충분히 반영하지 못하는 경우 의미적 정합성이나 정보의 정확성 측면에서 성능 저하가 발생할 수 있다 [7]. 이는 생성 품질이 모델의 표현 능력 자체보다는, 입력으로 제공되는 정보의 범위와 충실도에 유의미하게 영향을 받음을 보인다.

따라서 LLM 기반 버그 리포트 자동 생성의 성능을 안정적으로 향상시키기 위해서는, 단일 보고서에 국한된 입력 설정을 넘어 보다 풍부한 문맥 정보를 활용할 필요가 있다 [8]. 결함과 유사한 과거 사례나 관련된 버그 리포트에서 공통적으로 나타나는 재현 패턴, 실행 환경, 결과 기술을 함께 고려할 수 있다면, 자동 생성된 버그 리포트의 구조적 완성도뿐만 아니라 내용적 충실성과 의미적 정합성 역시 보다 체계적으로 개선될 수 있을 것으로 기대된다 [7].

2.3 검색 기반 문맥 보강을 통한 버그 리포트 자동 생성

단일 입력에 의존하는 LLM 기반 버그 리포트 자동 생성의 한계를 보완하기 위한 대안으로, 생성 과정에 외부 문맥을 통합하는 접근이 주목받고 있다. 그중 검색 기반 문맥 보강은 입력과 관련된 과거 사례를 선택적으로 활용함으로써, 생성 모델이 참조할 수 있는 정보 범위를 명시적으로 확장한다는 점에서 특징적이다 [8].

버그 리포트 도메인은 검색 기반 문맥 보강이 효과적으로 적용될 수 있는 특성을 지닌다. 소프트웨어 결함은 특정 모듈이나 환경 조건에서 반복적으로 보고되는 경향이 있으며, 이로 인해 프로젝트 저장소에는 유사한 증상과 재현 조건을 공유하는 다수의 리포트가 축적된다. 이러한 과거 사례에는 개별 리포트에서 충분히 기술되지 않은 재현 절차나 환경 정보가 포함될 가능성이 있으며, 이는 자동 생성 과정에서 유용한 문맥 자원으로 활용될 수 있다 [7].

검색을 통해 제공된 예시는 생성 모델에게 출력 형식과 서술 수준에 대한 기준점으로 작용하며, 재현 절차나 환경 정보와 같은

구조적 요소의 포함을 유도한다. 또한 의미적으로 유사한 사례를 활용할 경우, 어휘 표현의 차이가 존재하더라도 생성 결과의 내용적 충실성과 의미적 정합성을 유지하는 데 기여할 수 있다 [7], [8]. 이러한 점에서 검색 기반 문맥 보강은 단순한 정보 추가가 아니라, 생성 과정의 조건을 조정하는 설계 요소로 해석될 수 있다.

한편 검색 기반 문맥 보강의 효과는 검색 전략에 따라 상이하게 나타날 수 있다. 키워드 기반 검색은 명시적인 기술 용어가 포함된 경우 관련 사례를 효과적으로 회수할 수 있으나, 표현 차이가 큰 경우에는 한계를 가질 수 있다 [15]. 반대로 의미 기반 검색은 의미적 정합성을 반영함으로써 이러한 한계를 완화할 수 있으나, 검색 정밀도가 낮아질 가능성이 존재한다 [12]. 또한 검색 전략의 결합이나 확장 방식은 생성 품질뿐만 아니라 계산 비용에도 영향을 미친다.

이러한 특성은 검색 기반 문맥 보강을 단일한 기법으로 다루기보다, 검색 전략을 하나의 설계 변수로 분석할 필요성을 제기한다. 즉, 검색 적용 여부 자체보다, 검색 전략의 선택과 구성 방식이 구조적 완성도, 어휘적 정보 포괄성, 의미적 정합성과 같은 품질 차원에 어떠한 영향을 미치는지를 구분하여 분석하는 것이 중요하다. 본 연구는 이러한 관점에서 키워드 기반, 의미 기반, 그리고 하이브리드 검색 전략을 동일한 생성 조건 하에서 비교함으로써, 검색 기반 문맥 보강이 버그 리포트 자동 생성에 기여하는 방식과 그 한계를 정량적으로 분석하고자 한다.

3. 방법론

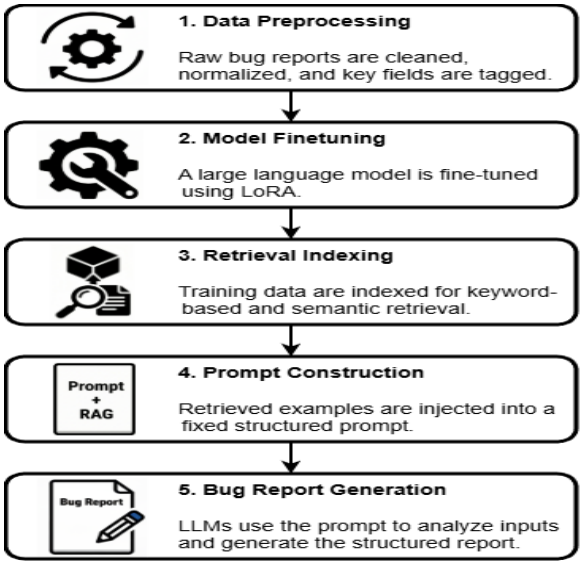


그림1. 본 연구에서 사용한 버그 리포트 자동 생성 실험 파이프라인

본 연구의 전체 방법론은 그림 1에 제시된 처리 흐름을 따른다. 그림 1은 데이터 전처리, 언어 모델 파인튜닝, 학습 세트 기반 검색 인덱싱, 검색 결과를 반영한 프롬프트 구성, 그리고 버그 리포트 생성으로 이어지는 일련의 과정을 단계적으로 나타낸다.
이를 통해 검색 기반 문맥 보강 전략을 적용한 오픈 소스 기반 대규모 언어 모델의 구조화된 버그 리포트가 최종적으로 생성된다.

3.1 데이터 전처리 및 분할

본 연구에서는 기존 연구에서 구축·공개된 Bugzilla 기반 버그 리포트 데이터셋을 사용하였다 [16]. 해당 데이터셋은 LLM 기반 버그 리포트 자동 생성 연구를 위해 정제된 공개 데이터셋으로, 본 연구는 데이터 전처리 및 구성 방식에 따른 영향을 배제하기 위해 기존 연구에서 정의된 최종 데이터셋을 그대로 사용하였다.
기존 연구에서는 원본 Bugzilla 리포트 중 상태가 “fixed” 또는 “closed”로 표시된 사례를 대상으로 데이터를 구성하였다. 이는 이미 해결된 버그 리포트가 결함 재현 절차, 실행 환경, 기대 결과 및 실제 결과와 같은 핵심 정보를 상대적으로 충실히 포함하고 있어, 구조화된 버그 리포트 생성 모델의 학습과 평가에 적합하다는 판단에 기반한다.

데이터 품질을 확보하기 위해, 기존 연구에서는 두 단계의 정제 절차를 적용하였다. 첫 번째 단계에서는 정규 표현식 기반 필터링을 통해 요약(Summary), 재현 절차(Steps to Reproduce), 기대 결과(Expected Result), 실제 결과(Actual Result), 추가 정보(Additional Information)가 명시적으로 구분된 리포트만을 유지하였다. 두 번째 단계에서는 자동화된 CTQRS 평가를 적용하여 [10], 구조적 완성도가 일정 기준 이상으로 확보된 리포트만을 최종 데이터셋에 포함시켰다. 이 과정은 의미적 정합성과 정보 충실성을 동시에 만족하는 리포트만을 학습 및 평가 대상으로 제한하기 위한 것이다.

정제된 데이터셋 중 일부는 기존 연구에서 수동 검토를 통해 추가로 확인되었으며, 자동 생성 모델의 학습 데이터나 검색 기반 예시로 활용하기에 부적합한 사례는 제외되었다. 본 연구는 이러한 검증 과정을 거쳐 공개된 최종 데이터셋을 동일하게 사용함으로써, 데이터 사용의 공정성과 실험 결과의 재현성을 확보하였다.
구성된 데이터셋은 학습, 검증, 테스트 세트(8:1:1)로 분할하여 사용되었으며, 데이터 분할 역시 기존 연구에서 정의된 설정을 그대로 유지하였다. 이를 통해 본 연구에서 관찰되는 성능 차이가 데이터 전처리나 분할 방식의 차이가 아니라, 검색 기반 문맥 보강 전략과 검색 설계의 차이에 기인하도록 실험 환경을 통제하였다.

3.2 학습 세트 임베딩 인덱싱

본 연구에서는 검색 기반 문맥 보강 전략을 체계적으로 분석하기 위해, 학습 세트에 포함된 과거 버그 리포트 사례들을 사전에 인덱싱하고 이를 검색 가능한 형태로 구성하였다. 검색의 목적은 입력으로 주어진 비정형 버그 요약과 유사한 과거 사례를 식별하여, 생성 단계에서 입력-출력 예시로 활용함으로써 버그 리포트 생성 과정에 추가적인 문맥 정보를 제공하는 데 있다. 본 절에서는 검색 공간의 정의, 유사도 계산 방식, 그리고 서로 다른 검색 전략의 설계와 해석을 명확히 기술한다.

검색 대상은 학습 세트에 포함된 버그 요약(summary)으로 한정한다. 검색은 입력 요약 s_q에 대해 학습 세트 내 요약 s_i들과의 유사도를 계산하는 방식으로 수행된다. 검색을 통해 선택된 상위 k개의 사례는 이후 프롬프트 구성 단계에서 입력-출력 예시 쌍(s_i, r_i) 형태로 주입된다. 모든 검색은 학습 세트 범위 내에서만 수행되며, 검증 및 테스트 데이터는 검색 대상에서 제외함으로써 데이터 누수 가능성을 최소화하였다. 학습 세트는 요약-리포트 쌍의 집합으로 정의되며, 다음과 같이 표현된다.

$$D_{train} = \{(s_i, r_i) \mid i = 1, \ldots, N\}$$

- i은 학습 세트에 포함된 개별 버그 리포트 사례의 인덱스를 나타냄
- N은 학습 세트에 포함된 전체 사례 수를 의미함
- r_i는 요약 s_i에 대응하는 구조화된 버그 리포트를 나타냄
- s_q는 생성 단계에서 주어지는 새로운 비정형 버그 요약을 나타냄

BM25 점수가 높을수록 입력 요약과 과거 요약 간의 어휘적 일치도가 높음을 의미하며, 오류 메시지나 컴포넌트 명과 같이 명시적인 기술 용어가 공유되는 사례가 우선적으로 선택된다.

$$BM25(s_q, s_i) = \sum_{t \in s_q} IDF(t) \cdot \frac{f(t, s_i) \cdot (k_1 + 1)}{f(t, s_i) \cdot (1 - b + b \cdot \frac{|s_i|}{avgdl})}$$

- t는 입력 요약 s_q에 포함된 개별용어(term)을 의미함
- $f(t, s_i)$는 요약 s_i에서 용어 t의 등장 빈도를 나타냄
- $|s_i|$는 요약 s_i의 길이를 나타냄
- avgdl은 학습 세트 요약 코퍼스의 평균 길이를 나타냄
- IDF(t)는 학습 세트 요약을 기준으로 계산된 역문서 빈도를 나타냄

- k_1은 용어 빈도의 포화 정도를 조절하는 파라미터를 의미함
- b는 문서 길이 정규화의 영향을 조절하는 파라미터를 의미함

의미 기반 검색은 Sentence-BERT 기반 임베딩을 활용하여 구현하였다 [12]. 각 요약은 임베딩 함수 $\phi(\cdot)$를 통해 동일한 의미 공간의 벡터로 변환되며, 두 요약 간의 의미적 유사도는 코사인 유사도로 계산된다. 이 값은 −1에서 1 사이의 범위를 가지며, 값이 클수록 두 요약이 의미적으로 유사함을 의미한다. SBERT 유사도 검색은 어휘적 중복이 낮은 경우에도 결함의 원인, 증상, 맥락이 유사한 사례를 효과적으로 회수할 수 있다.

$$Sim_{SBERT(s_q, s_i)} = \frac{(e_q \cdot e_i)}{(||e_q||) \cdot (||e_i||)}$$

- $\phi(\cdot)$는 사전 학습된 Sentence-BERT 임베딩 함수를 의미함
- $e_q = \phi(s_q)$는 입력 요약 s_q의 임베딩 벡터
- $e_i = \phi(s_i)$는 입력 요약 s_i의 임베딩 벡터

본 연구에서는 키워드 기반 검색과 의미 기반 검색의 상호 보완적 특성을 결합하기 위해 BM25-SBERT 하이브리드 검색 전략을 정의하였다. 하이브리드 검색은 두 검색 방식에서 산출된 순위를 결합하여 다음과 같은 결합 순위 점수를 계산한다. 특정 검색 방식에 대한 편향을 최소화하기 위해 α = 0.5로 고정하여 동일 가중치를 적용하였다. 결합 순위 점수가 낮을수록 어휘적 단서와 의미적 정합성을 동시에 만족하는 사례임을 의미한다.

$$Rank_{hybrid}(s_i) = \alpha \cdot rank_{BM25}(s_i) + (1 - \alpha) \cdot rank_{SBERT}(s_i)$$

- $rank_{BM25}$는 BM25 기준에서의 요약 s_i의 순위를 나타냄
- $rank_{SBERT}$는 SBERT 유사도 기준에서의 요약 s_i의 순위를 나타냄
- α는 두 검색 신호 간의 기여도를 조절하는 가중치를 의미함

추가적으로, 본 연구는 검색 결과에 재정렬 단계를 적용한 확장 설정을 별도로 구성하였다. 재정렬은 1차 검색을 통해 확보된 상위 k'개의 후보 집합 C에 대해 수행되며, 각 후보에 대해 다음과 같은 재정렬 점수를 계산한다. 재정렬 점수가 높을수록 입력 요약과 과거 사례 간의 정합성이 높다고 판단되며, 이를 기준으로 후보 집합을 재정렬한 뒤 상위 k개의 사례를 최종 검색 결과로 선택한다 [17]. 본 연구에서는 재정렬 기법의 순수한 추가 효과를 분석하기 위해 이를 독립적인 확장 설정으로 포함하였다.

$$Score_{rerank}(s_i, r_i) = \psi(s_q, s_i, r_i)$$

- $\psi(\cdot)$은 입력 요약 s_q와 후보 요약–리포트 쌍 (s_i, r_i)을 하나의 텍스트 시퀀스로 결합하여 관련도 점수를 산출하는 cross-encoder 기반 함수를 나타냄
- (s_i, r_i)는 학습 세트에서 검색된 요약-리포트 쌍을 의미함

이와 같이 본 연구에서는 검색 공간, 유사도 계산 방식, 검색 전략의 결합 및 확장 방식을 명시적으로 정의함으로써, 검색 기반 문맥 보강을 하나의 분석 가능한 설계 요소로 취급한다. 이러한 정의는 이후 실험에서 키워드 기반, 의미 기반, 하이브리드 검색, 그리고 재정렬 확장 설정이 LLM 기반 버그 리포트 자동 생성 품질에 미치는 영향을 일관된 기준 하에서 비교·분석하기 위한 기반을 제공한다.

3.3 프롬프트 설계

본 연구에서는 검색 기반 문맥 보강 전략의 효과를 공정하게 비교·분석하기 위해, 모든 실험에서 동일한 프롬프트 구조와 지시 체계를 사용하였다. 프롬프트는 입력으로 주어진 비정형 버그 요약을 구조화된 버그 리포트로 변환하는 것을 목표로 하며, 검색 전략이나 예시 수를 제외한 모든 구성 요소는 실험 전반에 걸쳐 고정하였다.

검색 기반 문맥 보강 설정에서는 학습 세트로부터 검색된 사례를 프롬프트에 예시 형태로 삽입하였다. 각 예시는 입력 요약과 이에 대응하는 구조화된 버그 리포트 쌍으로 구성되며, 이는 언어 모델이 요구되는 출력 형식과 서술 수준을 명확히 인식하도록 돕는 기준점으로 작동한다. 예시의 수는 1/2/3-shot으로 조절하였으며, 예시 수를 제외한 프롬프트의 나머지 구성 요소와 지시문은 모든 실험에서 동일하게 유지하였다. 이를 통해 검색 전략과 예시 수 변화가 생성 품질에 미치는 영향을 독립적으로 분석할 수 있도록 실험 조건을 통제하였다.

검색을 사용하지 않는 설정에서는 예시 없이 동일한 지시문과 출력 형식 제약만을 적용하였다. 이 설정은 검색 기반 문맥 보강의 효과를 비교하기 위한 베이스라인으로 활용되었으며, 프롬프트 구조 자체의 차이가 아닌 검색 예시의 유무와 구성 방식만이 성능 차이에 영향을 미치도록 설계되었다.

검색된 예시가 프롬프트에 삽입되는 전체 구조는 그림 2에 제시한다. 그림 2는 입력 요약, 검색을 통해 선택된 예시 쌍, 고정된 지시문, 그리고 출력 형식 제약이 하나의 프롬프트 문맥으로 결합되는 과정을 개략적으로 나타낸다. 이를 통해 검색 기반 문맥 보강이 단순한 참고 정보 제공이 아니라, 생성 단계에서 언어 모델의 출력 행동을 제약하고 유도하는 프롬프트 구성 요소로 작동함을 확인할 수 있다.

```
You are an expert QA engineer. Convert the user's unstructured input into a structured bug report.

--- Reference Examples (Retrieved) ---
Example #1:
[Input]
<Example Summary>

[Output]
<Generated Bug Report>

--- End of Examples ---
### Input to Convert:
<Summary>

### Output Bug Report
```

그림 2. 검색 기반 문맥 보강을 활용한 프롬프트 구조 예시

3.4 버그 리포트 생성

본 연구에서는 미세조정된 대규모 언어 모델과 고정된 프롬프트를 사용하여 버그 리포트를 생성하였다. 생성 단계는 검색 기반 문맥 보강 전략과 예시 구성 방식이 생성 결과에 미치는 영향을 비교하기 위한 절차로 정의되었으며, 전략 간의 공정한 비교를 위해 생성 과정 전반의 조건을 일관되게 유지하였다. 이를 통해 생성 결과의 차이가 모델 설정이나 실행 환경의 변화가 아니라, 검색 기반 문맥 보강 전략과 예시 구성의 차이에 의해 발생하도록 실험을 설계하였다.

각 입력 샘플은 버그 요약-리포트 쌍으로 제공되며, 생성 과정에서는 3.3절에서 정의한 출력 형식 제약에 따라 구조화된 버그 리포트로 변환된다. 검색을 사용하지 않는 설정에서는 예시 없이 동일한 지시문과 출력 형식만이 적용되었다. 이때 검색 전략과 예시 수를 제외한 모든 생성 조건은 실험 전반에 걸쳐 동일하게 유지하였다.

4. 실험

4.1 실험 요약 및 설정

본 연구의 실험은 검색 기반 문맥 보강 전략의 설계 차이가 대규모 언어 모델 기반 버그 리포트 자동 생성 품질에 미치는 영향을 비교·분석하기 위해 구성되었다. 이를 위해 검색을 적용하지 않는 생성 설정을 베이스라인으로 정의하고, 키워드

기반 검색(BM25), 의미 기반 검색(SBERT), BM25-SBERT 하이브리드 검색, 그리고 재정렬 단계를 포함한 확장 설정을 동일한 생성 조건 하에서 비교하였다. 모든 실험은 동일한 데이터 분할과 생성 절차를 유지하여, 관찰된 성능 차이가 검색 전략과 문맥 보강 방식의 차이에 기인하도록 통제하였다.

실험에는 Qwen2.5-7B-Instruct [19], LLaMA-3.2-3B-Instruct [20], Mistral-7B-Instruct-v0.3 [21]의 세 가지 LLM을 사용하였다. 모든 모델에는 LoRA 기반 미세조정을 동일한 설정으로 적용하였으며 [9], Unsloth 프레임워크를 기반으로 학습을 수행하였다 [22]. 의미 기반 검색에는 sentence-transformers/all-mpnet-base-v2 모델을 사용하였고 [12], 재정렬 확장 설정에는 BAAI/bge-reranker-v2-m3 모델을 적용하였다 [17].

생성 과정에서는 확률적 변동성을 배제하기 위해 temperature를 0으로 설정하고 샘플링을 비활성화한 결정론적 디코딩을 사용하였다. 이를 통해 검색 전략과 문맥 보강 방식의 차이가 생성 결과에 미치는 영향을 보다 명확하게 분석하였다. 모든 실험은 단일 NVIDIA RTX 4090 GPU를 사용하는 Ubuntu 24.04 (WSL2) 환경에서 수행되었으며, CUDA 12.6, PyTorch 2.5.1+cu121 [23], Transformers 4.55.3 [24], Unsloth 프레임워크 [22]를 동일하게 적용하였다.

4.2 데이터셋

본 연구의 실험은 Bugzilla 기반 버그 리포트 데이터셋을 사용하여 수행되었다 [2]. 해당 데이터셋은 기존 연구에서 구축·공개된 데이터셋을 기반으로 하며 [16], 비정형 버그 요약과 이에 대응하는 구조화된 버그 리포트로 구성된 쌍 형태를 갖는다. 각 쌍은 하나의 버그 요약과, 이를 재현 절차, 기대 결과, 실제 결과, 실행 환경 등의 구조적 요소로 정리한 버그 리포트로 이루어져 있다.

본 연구에서 사용한 최종 데이터셋은 총 3,966개의 요약-리포트 쌍으로 구성되어 있으며, 검색 기반 문맥 보강 전략의 효과를 분석하기에 충분한 규모와 품질을 갖는다. 해당 데이터셋은 자동 생성 모델 학습과 평가에 적합하도록 기존 연구에서 정제된 결과물로, 구조적 완성도와 정보 충실성이 확보된 리포트들로 이루어져 있다 [16].

본 데이터셋은 버그 요약과 구조화된 리포트로 구성된 텍스트 기반 데이터셋으로, 본 연구의 모든 실험은 이러한 텍스트 정보만을 입력과 출력으로 사용하여 수행되었다.

구성된 데이터셋은 학습, 검증, 테스트 세트로 분할하여 사용되었으며, 모든 실험에서는 동일한 분할을 유지하였다. 특히 검색 기반 문맥 보강 과정에서도 학습 세트만을 검색 대상으로 사용하고, 평가 대상 리포트와 동일한 사례가 예시로 활용되지 않도록 엄격히 통제하였다. 이를 통해 데이터 누출 가능성을 차단하고, 검색 전략 및 문맥 보강 설정 간 비교 실험의 공정성과 결과 해석의 신뢰성을 확보하였다.

4.3 평가 지표

본 연구에서는 생성된 버그 리포트의 품질을 단일 관점이 아닌 다각적 관점에서 분석하기 위해, 구조적 완성도, 어휘적 정보 포괄성, 어휘적 균형, 의미적 정합성의 네 가지 측면을 반영하는 자동 평가 지표를 사용하였다. 각 지표는 서로 상보적인 품질 요소를 측정하며, 특정 지표에 대한 편향을 완화하기 위해 복수의 지표를 함께 적용하였다. 지표 간 비교의 직관성과 의미적 정합성을 확보하기 위해, 본 연구에서 보고하는 모든 지표 값은 백분율(%) 형태로 정규화하여 제시한다.

- **CTQRS (**구조적 완성도**) [10]**: CTQRS는 생성된 버그 리포트가 재현 절차, 환경 정보, 기대 결과 및 실제 결과와 같이 재현 가능성과 관련된 핵심 구조 요소를 얼마나 체계적으로 포함하고 있는지를 평가하는 지표이다. 본 연구의 목적은 생성된 리포트가 실제로 재현에 성공하는지(재현 성공률)를 직접적으로 평가하는 것이 아니라, 재현을 가능하게 하는 구조적 전제조건이 충분히 갖추어졌는지를 분석하는 데 있으므로, CTQRS를 구조적 완성도에 대한 대리 지표로 사용하였다. 다만 CTQRS는 자동화된 평가 지표로서 개발자의 이해 용이성이나 실제 재현 효율과 같은 질적 요소를 완전히 반영하지는 못하며, 이에 따라 본 연구에서는 CTQRS 점수의 향상을 재현 성공의 직접적인 보장으로 해석하지 않는다. CTQRS 산출은 기존 연구에서 제시된 자동 평가 스크립트를 활용하여 동일한 기준 하에서 일관되게 측정하였다.
- **ROUGE-1 Recall (**어휘적 정보 포괄성**) [11]**: 참조 리포트의 핵심 정보가 생성된 결과물에 얼마나 누락 없이 포함되었는지를 측정하기 위해 ROUGE-1 Recall을 사용하였다. 이는 생성된 텍스트가 참조 리포트의 주요 단어를 얼마나 폭넓게 포함하는지를 나타내며, 값이 낮을수록 핵심 단서의 누락 가능성이 높음을 보인다. 본 연구에서는 검색 기반 문맥 보강이 핵심 정보의 포함에 미치는 영향을 분석하기 위한 지표로 활용하였다.
- **ROUGE-1 F1 (**어휘적 균형**) [11]**: ROUGE-1 F1은 Recall(어휘적 정보 포괄성)과 Precision(정밀도)의 조화 평균으로, 핵심 정보를 포함하는 동시에 불필요하거나 중복된 표현을 얼마나 억제했는지를 평가한다. 즉, 단순히 정보를 많이 포함하여 Recall을 높이는 것에 그치지 않고, 생성 결과가 장황해지거나 불필요한 토큰을 과도하게 생성하는 현상을 함께 점검할 수 있다. 본 연구에서는 문맥 보강 과정에서 발생할 수 있는 중복과 장황함을 포함한 어휘적 품질의 균형을 분석하기 위해 F1을 함께 고려하였다.
- **SBERT** 유사도 **(**의미적 정합성**) [12]**: 표면적 단어 일치 여부를 넘어, 생성된 리포트와 참조 리포트 간의 의미적 정합성을 평가하기 위해 SBERT(Sentence-BERT) 기반 코사인 유사도를 측정하였다. 이 지표는 사용된 어휘가 다르더라도 문장의 전체적인 의미와 의도가 얼마나 일관되게 유지되는지를 포착할 수 있다. 본 연구에서는 의미 기반 검색 및 하이브리드 검색 전략이 생성 결과의 의미적 정합성에 미치는 영향을 분석하기 위한 지표로 사용하였다.

이와 같은 지표 구성은 검색 기반 문맥 보강이 버그 리포트 자동 생성 품질에 미치는 영향을 구조적, 어휘적, 의미적 측면에서 종합적으로 분석하기 위한 것이다. 다만 자동 평가 지표는 개발자의 이해 용이성이나 실제 재현 효율과 같은 질적 요소를 완전히 반영하지는 못한다는 한계가 있다. 이러한 점을 고려하여, 향후 연구에서는 개발자 참여 기반의 주관적 평가를 통해 생성된 버그 리포트의 실질적 유용성과 재현 가능성을 함께 검증할 계획이다. 본 연구에서는 실험 결과의 통계적 안정성을 확보하기 위해 부트스트랩 리샘플링 기법을 적용하였다. 각 평가 지표에 대해 테스트 데이터셋으로부터 복원 추출을 5,000회 반복 수행하여 95% 신뢰 구간을 산출하였으며, 이를 통해 단일 평균값이 가질 수 있는 우연성을 배제하고 결과의 신뢰도를 검증하였다. 베이스라인 실험의 경우, 개별 생성 결과 데이터가 공개되어 있지 않아 ROUGE-1 F1 및 paired 통계 분석을 수행할 수 없었으며, 따라서 베이스라인 결과는 직접적인 통계 비교가 아닌 참고 기준으로만 활용된다.

4.4 연구 질문 및 실험 결과

본 절에서는 검색 기반 문맥 보강 전략이 대규모 언어 모델 기반 버그 리포트 자동 생성 품질에 미치는 영향을 체계적으로 분석한다. 이를 위해 검색을 적용하지 않는 생성 설정을 베이스라인으로 정의하고, 키워드 기반 검색(BM25), 의미 기반 검색(SBERT), BM25-SBERT 하이브리드 검색, 그리고 재정렬(Re-ranking)을 포함한 확장 설정을 동일한 조건 하에서 비교하였다. 본 연구는 검색 전략의 유형과 결합 방식, 그리고 확장 설정이 생성 품질에 미치는 영향을 단계적으로 검증하기 위해 다음과 같은 연구 질문을 설정하였다.

- RQ1. 키워드 기반 검색의 영향: 키워드 기반 검색(BM25)을 활용한 문맥 보강은 검색을 사용하지 않는 생성 방식과 비교하여 버그 리포트의 구조적 완성도와 어휘적 정보 포괄성을 향상시키는가?
- RQ2. 의미 기반 검색의 특성: 유사도 기반 검색(SBERT)을 활용한 문맥 보강은 키워드 기반 검색과 비교하여 의미적 정합성과 표현 안정성 측면에서 어떠한 특성을 보이는가?

- RQ3. BM25-SBERT 하이브리드 검색의 성능적 특성: BM25-SBERT 하이브리드 검색은 단일 검색 전략 대비 구조적 완성도, 어휘적 균형, 의미적 정합성 측면에서 보다 안정적인 성능을 제공하는가?
- RQ4. 재정렬 확장 설정의 추가 효과: 검색 결과에 대해 재정렬 단계를 추가하는 확장 설정은 하이브리드 검색의 생성 품질에 유의미한 추가 효과를 제공하는가, 아니면 그 효과가 제한적인 범위에 머무르는가?

먼저, 검색 기반 문맥 보강 전략의 효과를 보다 명확히 분석하기 위해 실험에 사용할 기준 모델을 선정하는 사전 비교 실험을 수행하였다. 비교 대상은 Qwen2.5-7B-Instruct, LLaMA-3.2-3B-Instruct, Mistral-7B-Instruct-v0.3의 세 가지 모델이며, 동일한 데이터셋과 학습 설정, 평가 지표 하에서 성능을 비교하였다. 검색을 적용하지 않은 베이스라인 조건에서 Qwen2.5-7B-Instruct는 CTQRS 77%, ROUGE-1 Recall 61%, SBERT 유사도 85%를 기록하여, LLaMA-3.2-3B-Instruct(CTQRS 63%, ROUGE-1 Recall 50%, SBERT 유사도 73%)와 Mistral-7B-Instruct-v0.3(CTQRS 71%, ROUGE-1 Recall 59%, SBERT 유사도 84%)에 비해 전반적으로 안정적인 성능을 보였다. 이는 구조적 완성도와 의미적 정합성 측면에서 Qwen2.5-7B-Instruct 모델이 기본 생성 품질에서 상대적으로 높은 성능을 일관되게 유지함을 보여준다.

검색 기반 문맥 보강을 적용한 이후에는 세 모델 모두에서 CTQRS와 ROUGE-1 Recall이 일관되게 증가하는 공통적인 경향이 관찰되었다. 반면 shot 수가 증가할수록 ROUGE-1 F1 점수는 모든 모델에서 지속적으로 감소하는 양상을 보였는데, 이는 ROUGE-1 F1이 어휘적 정보 포괄성과 정밀도의 균형을 반영하는 지표라는 점을 고려할 때, 관찰된 성능 변화가 포괄성 중심의 생성으로 기울어졌음을 보여주는 결과로 해석할 수 있다. 이러한 변화는 결과적으로 어휘적 균형의 저하로 이어졌음을 의미한다.

모델별 성능 차이를 보다 명확히 비교하기 위해 하이브리드 1-shot 설정을 기준으로 살펴보면, Qwen2.5-7B-Instruct는 CTQRS 92.0%, ROUGE-1 Recall 81.3%, F1 62.4%, SBERT 유사도 91.0%를 기록하여 네 가지 지표 전반에서 균형 잡힌 성능을 보였다. 반면 LLaMA-3.2-3B-Instruct는 동일 설정에서 CTQRS 89.9%, Recall 79.7%, F1 52.1%, SBERT 유사도 87.4%를 기록하여, 특히 ROUGE-1 F1과 SBERT 유사도 측면에서 Qwen 대비 낮은 절대 성능을 나타냈다. 이러한 경향은 BM25, SBERT, 재정렬 설정에서도 유사하게 관찰되었으며, LLaMA 모델은 일정 수준의 점수를 유지한 반면 어휘적 균형과 의미적 정합성 측면에서 상대적으로 불리한 양상을 보였다.

Mistral-7B-Instruct-v0.3는 하이브리드 1-shot에서 CTQRS 93.5%, Recall 81.0%, F1 62.1%, SBERT 유사도 88.7%로 높은 초기 성능을 보였으나, shot 수가 증가함에 따라 성능 저하 폭이 가장 크게 나타났다. BM25 및 SBERT 기반 설정에서도 유사한 감소 양상이 반복되었다. 이는 Mistral 모델이 구조적 완성도는 유지하는 반면, 예시 수 증가 상황에서 의미적 정합성과 어휘적 균형이 빠르게 저하되는 특성을 가짐을 보여준다.

이와 같이 여러 검색 전략과 shot 설정 전반에서 반복적으로 관찰된 결과를 종합하면, Qwen2.5-7B-Instruct는 구조적 완성도, 어휘적 균형, 의미적 정합성 측면에서 가장 안정적인 성능을 보였으며, 검색 전략 변화에 따른 성능 변동도 상대적으로 완만하였다. 이에 따라 이후 RQ1-RQ4에 대한 주 실험은 Qwen2.5-7B-Instruct를 기준 모델로 설정하여 수행하였다.

이제 RQ1-RQ3에 해당하는 검색 전략 비교 결과를 살펴본다. 모든 실험은 재정렬을 적용하지 않은 기본 검색 설정을 기준으로 수행되었으며, 각 검색 전략과 shot 수에 따른 정량적 결과는 표 1에 제시되어 있다. 또한 검색 전략 간 상대적 특성을 보다 명확히 비교하기 위해 시각적 분석에서는 1-shot 설정 결과를 중심으로 함께 제시하였다. 표 1에 제시된 결과는 각 설정의 평균 성능과 함께, 동일한 테스트 인스턴스에 대해 5,000회 부트스트랩 리샘플링으로 산출한 95% 신뢰구간을 포함한다. 이를 통해 관측된 성능 차이가 일부 특정 사례에 의해 발생한 것이 아니라, 입력 전반에서 일관되게 유지되는 경향인지 여부를 함께 분석할 수 있다.

표 1. 검색 전략과 shot 수에 따른 생성 품질 비교 결과. 괄호 안의 값은 5,000회 부트스트랩 리샘플링으로 산출한 95% 신뢰구간의 half-width를 나타낸다.

Method	CTQRS	ROUGE-1Recall	ROUGE-1 F1	SBERT
Baseline	77	61	-	85
BM25 1-shot	92.3(±0.8)	81.5(±1.8)	62.7(±2.2)	91.3(±0.9)
BM25 2-shot	92.4(±0.8)	81.5(±2.0)	61.4(±2.2)	89.3(±1.3)
BM25 3-shot	93.2(±0.7)	80.7(±2.0)	49.1(±2.6)	78.4(±2.1)
SBERT 1-shot	91.6(±0.9)	80.8(±2.0)	62.2(±2.3)	91.1(±1.0)
SBERT 2-shot	91.5(±1.0)	80.4(±2.1)	59.6(±2.4)	88.8(±1.4)
SBERT 3-shot	92.6(±0.9)	78.9(±2.2)	50.4(±2.5)	81.8(±1.9)
Hybrid 1-shot	92(±0.9)	81.3(±1.9)	62.4(±2.2)	91.0(±1.0)
Hybrid 2-shot	91.8(±0.9)	81.4(±2.0)	60.2(±2.3)	88.9(±1.4)
Hybrid 3-shot	92.7(±0.9)	79.8(±2.2)	49.2(±2.6)	79.6(±2.0)

먼저 키워드 기반 검색(BM25)을 적용한 결과를 살펴본다. 그림 3은 BM25 기반 RAG의 1-shot 설정에서의 생성 품질을 네 가지 평가 지표(CTQRS, ROUGE-1 Recall, ROUGE-1 F1, SBERT 유사도) 기준으로 베이스라인과 비교한 결과를 시각적으로 보여준다. 1-shot 설정에서 CTQRS는 77.0%에서 92.3%로 15.3%p 상승하였으며, ROUGE-1 Recall 역시 61.0%에서 81.5%로 20.5%p 증가하였다. 이는 키워드 기반 검색이 재현에 필요한 구조적 요소와 주요 정보가 생성 과정에서 보다 충실히 반영되었음을 보여준다. ROUGE-1 F1 점수는 1-shot 설정에서 62.7%를 기록하여, 어휘적 균형 측면에서도 상대적으로 높은 수준을 유지하였다. 또한 SBERT 유사도는 85.0%에서 91.3%로 6.3%p 향상되어, 의미적 정합성 역시 향상되는 경향이 관찰되었다. 또한 지표들의 신뢰구간 폭이 상대적으로 좁게 유지된 점은, BM25 기반 검색의 성능 향상이 일부 특정 사례에 의해 좌우되지 않고 다양한 입력에 대해 안정적으로 관찰됨을 보인다.

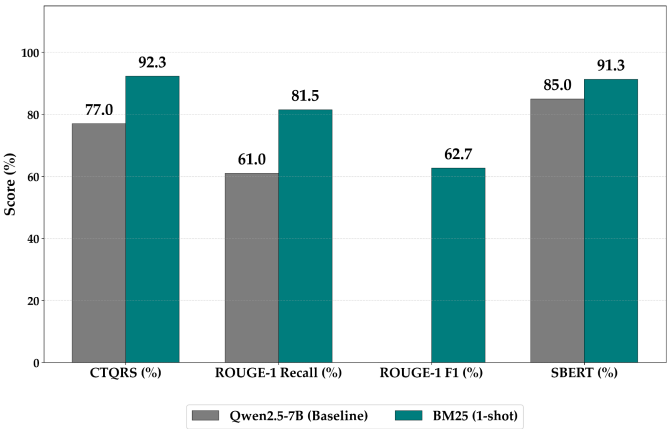


그림 3. BM25 기반 RAG의 1-shot 설정에서 생성된 버그 리포트의 품질을 네 가지 평가 지표로 베이스라인과 비교한 결과.

이후 shot 수가 증가함에 따라 지표 변화 양상을 분석하였다. CTQRS는 2-shot에서 92.4%, 3-shot에서 93.2%를 기록하여, 예시 수 증가에도 불구하고 구조적 완성도는 높은 수준을 유지하였다. 반면 ROUGE-1 F1 점수는 1-shot 62.7%에서 2-shot 61.4%, 3-shot 49.1%로 점진적인 감소를 보였으며, SBERT 유사도 역시 1-shot 91.3%에서 2-shot 89.3%, 3-shot 78.4%로 하락하는 경향이 관찰되었다. 이는 예시 수 증가에 따라 어휘적 정보 포괄성은 강화되는 반면, 어휘적 균형과 의미적 정합성 측면에서는 상대적인 감소가 관찰되었음을 의미한다.

다음으로 의미 기반 검색(SBERT)을 적용한 결과를 분석한다. 그림 4는 SBERT 기반 RAG의 1-shot 설정에서의 네 가지 지표를 베이스라인과 비교한 결과를 제시한다. CTQRS는 77.0%에서 91.6%로 14.6%p 상승하였으며, ROUGE-1 Recall은 61.0%에서 80.8%로 19.8%p 증가하였다. ROUGE-1 F1 점수는 1-shot에서 62.2%를 기록하여, BM25와 유사한 수준의 어휘적 균형을 달성하였다. 특히 SBERT 유사도는 85.0%에서 91.1%로 6.1%p 향상되어, 의미적 정합성이 안정적으로 유지되며 향상되는 경향을 보였다. 이는 의미 기반 검색이 어휘적 일치에 의존하지 않고도 참조 리포트의 핵심 문맥을 효과적으로 반영할 수 있음을 보여준다.

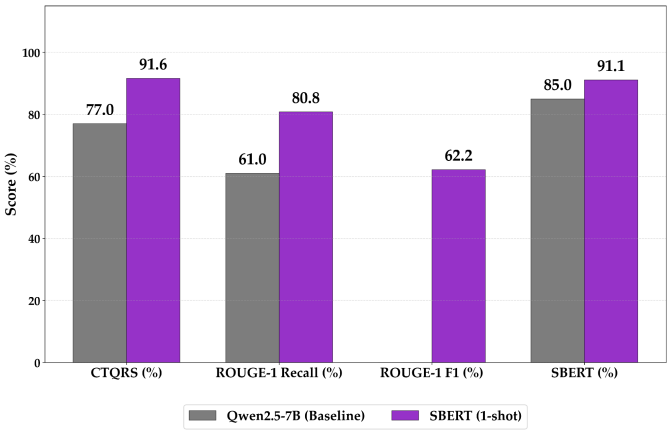


그림 4. SBERT 기반 RAG의 1-shot 설정에서 생성된 버그 리포트의 품질을 네 가지 평가 지표로 베이스라인과 비교한 결과.

shot 수 변화에 따른 결과를 살펴보면, CTQRS는 2-shot에서 91.5%, 3-shot에서 92.6%를 기록하여 비교적 안정적인 수준을 유지하였다. ROUGE-1 F1 점수는 1-shot 62.2%에서 2-shot 59.6%, 3-shot 50.4%로 감소하였으나, SBERT 유사도는 1-shot 91.1%에서 2-shot 88.8%, 3-shot 81.8%로 BM25 대비 상대적으로 완만한 감소 폭을 보였다. 이는 의미 기반 검색이 shot 수 증가 환경에서도 의미적 정합성을 비교적 안정적으로 유지하는 특성을 가짐을 보인다.

마지막으로 BM25-SBERT 하이브리드 검색 결과를 분석한다. 그림 5는 하이브리드 검색의 1-shot 설정에서 네 가지 지표를 베이스라인과 비교하여 제시한다. 하이브리드 검색은 CTQRS가 77.0%에서 92.0%로 15.0%p 상승하였고, ROUGE-1 Recall은 61.0%에서 81.3%로 20.3%p 증가하였다. ROUGE-1 F1 점수는 1-shot에서 62.4%를 기록하였으며, SBERT 유사도는 85.0%에서 91.0%로 6.0%p 향상되었다. 이는 하이브리드 전략이 키워드 기반 검색의 높은 초기 성능을 상당 부분 유지하고 있음을 의미한다.

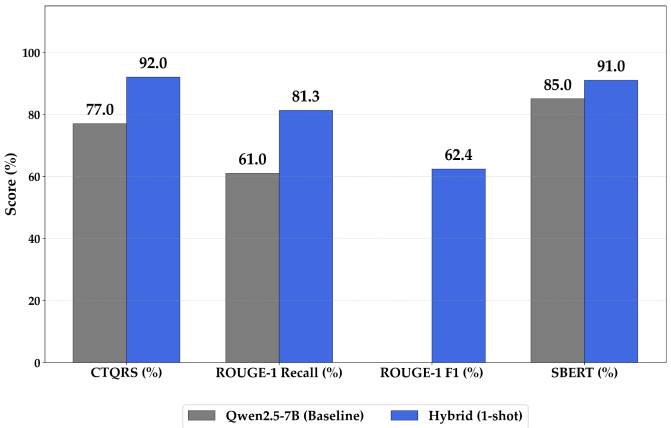


그림 5. BM25-SBERT 하이브리드 기반 RAG의 1-shot 설정에서 생성된 버그 리포트의 품질을 네 가지 평가 지표로 베이스라인과 비교한 결과.

shot 수가 증가함에 따라 하이브리드 검색 역시 일부 지표의 감소가 관찰되었으나, 그 감소 폭은 상대적으로 완만하였다. ROUGE-1 F1 점수는 1-shot 62.4%에서 2-shot 60.2%, 3-shot 49.2%로 감소하였고, SBERT 유사도는 1-shot 91.0%에서 2-shot 88.9%, 3-shot 79.6%로 하락하였다. 이는 BM25 단독 전략 대비 ROUGE-1 F1 및 SBERT 유사도 측면에서 감소 폭이 상대적으로 완만하게 나타난 결과이다. 이러한 관측 결과를 종합하면, 하이브리드 검색은 BM25의 높은 초기 성능과 SBERT의 상대적 안정성을 부분적으로 결합하여, shot 수 증가 환경에서도 비교적 균형 잡힌 성능을 유지하는 접근으로 해석할 수 있다.

종합하면, RQ1-RQ3의 결과는 검색 기반 문맥 보강이 버그 리포트 생성 품질에 구조적, 어휘적, 의미적 측면에서 뚜렷한 변화를 유도함을 보여준다. BM25는 적은 예시 수 환경에서 가장 우수한 성능을 보였고, SBERT는 shot 수 증가 상황에서 의미적 정합성을 비교적 안정적으로 유지하였다. 하이브리드 검색은 이 두 전략의 특성을 절충하여, 다양한 설정에서 균형 잡힌 성능을 제공하는 실용적인 대안으로 기능하였다. 이러한 결과의 의미와 실무적 함의는 다음 절에서 보다 자세히 논의한다.

다음으로 BM25-SBERT 하이브리드 검색 결과에 재정렬 기법을 추가로 적용했을 때 버그 리포트 생성 품질이 어떻게 변화하는지를 분석한다. 재정렬은 1차 검색 단계에서 수집된 후보 문서들을 대상으로 추가적인 정합성 평가를 수행하여, 생성 단계에 주입되는 예시의 순위를 조정하는 확장 설정으로 적용되었다. 본 실험에서는 재정렬을 적용하지 않은 하이브리드 검색 설정을 기준선으로 삼아, 재정렬 추가가 CTQRS, ROUGE-1 Recall, F1, SBERT 유사도에 미치는 영향을 비교하였다. 각 설정의 절대적인 성능 수준은 표 2에 제시되어 있다.

표 2. BM25-SBERT 하이브리드 검색에 재정렬 기법을 적용한 경우와 미적용한 경우를 shot 수별로 비교한 생성 품질 결과. 괄호 안의 값은 5,000회 부트스트랩 리샘플링으로 산출한 95% 신뢰구간의 half-width를 나타낸다.

Method	CTQRS	ROUGE-1Recall	ROUGE-1 F1	SBERT
Hybrid 1-shot	92(±0.9)	81.3(±1.9)	62.4(±2.2)	91(±1.0)
Hybrid 2-shot	91.8(±0.9)	81.4(±2.0)	60.2(±2.3)	88.9(±1.4)
Hybrid 3-shot	92.7(±0.9)	79.8(±2.2)	49.2(±2.6)	79.6(±2.0)
Rerank 1-shot	91.7(±0.8)	80.7(±1.9)	62.7(±2.3)	90.0(±1.2)
Rerank 2-shot	92.4(±0.8)	81.8(±1.9)	59.6(±2.2)	88.7(±1.4)
Rerank 3-shot	93.1(±0.7)	79.6(±2.0)	51.2(±2.4)	81.7(±1.8)

표 2에 따르면, 재정렬을 적용한 하이브리드 검색 설정은 전반적으로 재정렬을 적용하지 않은 경우와 유사한 성능 범위를 보이며, 일부 지표에서는 소폭의 차이가 관찰된다. 그러나 이러한 비교는 각 설정의 평균 성능을 독립적으로 비교한 결과로, 동일한 테스트 인스턴스를 기준으로 재정렬이 실제로 추가적인 개선 효과를 제공하는지를 직접적으로 검증하기에는 한계가 있다.

이에 따라 본 연구에서는 재정렬 기법의 순수한 추가 효과를 보다 엄밀히 분석하기 위해, Hybrid 1-shot 설정을 기준으로 재정렬 적용 여부에 따른 성능 차이(Δ = rerank − no-rerank)에 대해 paired bootstrap 분석을 수행하였다. Hybrid 1-shot 설정은 이전 실험에서 상대적으로 안정적인 성능 분포와 비교적 높은 품질을 보인 설정으로, shot 수 증가에 따른 부가적인 변동 요인의 영향을 최소화한 상태에서 재정렬 기법의 효과를 관찰하기에 적절한 기준 설정으로 선택되었다.

paired bootstrap 분석 결과는 표 3에 제시되어 있다. 분석 결과, CTQRS와 ROUGE 기반 지표(Recall 및 F1)의 경우 재정렬 적용에 따른 평균 성능 차이는 매우 제한적이며, 해당 차이의 95% 신뢰구간이 모두 0을 포함하는 것으로 나타났다. 이는 재정렬 기법의 효과가 구조적 완성도나 어휘적 중복 및 균형 측면에서 입력에 따라 상이하게 나타나며, 해당 지표 전반에 걸쳐 일관된 성능 향상을 제공한다고 보기 어렵다는 점을 보인다. 반면 SBERT 유사도에서는 재정렬 적용 시 평균적으로 양의 성능 차이가

관찰되었으며, 해당 차이의 신뢰구간이 0을 포함하지 않아 의미적 정합성 측면에서는 통계적으로 유의미한 개선 효과가 확인되었다. 이는 재정렬 기법이 동일한 입력에 대해 생성되는 예시들의 의미적 정합성을 비교적 일관되게 개선했음을 의미하며, 구조적·어휘적 지표와는 다른 특성을 보임을 나타낸다.

이러한 결과는 재정렬 기법이 생성된 버그 리포트의 구조적 구성이나 표면적 어휘 품질을 전반적으로 향상시키기보다는, 검색된 예시와의 의미적 정합성을 중심으로 제한적인 추가 이점을 제공함을 보인다. 즉, 하이브리드 검색 기반 문맥 보강 설정에서 재정렬 확장은 의미적 유사도 측면에서는 일정 수준의 기여를 보이나, 전반적인 생성 품질을 구조적, 어휘적, 의미적 측면 전반에서 일관되게 개선하는 핵심 요인으로 작용한다고 보기는 어렵다.

표 3. Hybrid 1-shot 설정에서 재정렬 적용 여부에 따른 성능 차이(Δ = rerank − no-rerank)에 대해 수행한 paired bootstrap 분석 결과.

Metric	Δ mean (±95% CI)
CTQRS	+0.31 (±0.71)
ROUGE-1 Recall	+0.58 (±1.09)
ROUGE-1 F1	−0.33 (±1.42)
SBERT	+1.01 (±0.83)

5. 토의

5.1 실험 결과 해석

본 절에서는 검색 기반 문맥 보강 전략이 LLM 기반 버그 리포트 자동 생성 품질에 미치는 영향을 정량적 결과를 중심으로 분석한다. 분석의 초점은 검색 전략, shot 수, 그리고 평가 지표 간의 관계에 있다.

먼저 검색을 적용하지 않은 베이스라인과 비교했을 때, 모든 검색 기반 설정에서 구조적 완성도와 어휘적 정보 포괄성이 일관되게 향상되었다. 베이스라인의 CTQRS는 77% 수준에 머물렀으나, 검색 기반 설정에서는 전략과 예시 수에 관계없이 약 91-93% 범위로 수렴하였다. 이는 검색된 예시가 생성 결과의 형식과 구성 요소 포함 양상에 유의미한 영향을 미쳤음을 보인다. ROUGE-1 Recall 역시 베이스라인의 61%에서 검색 적용 후 약 80-82%로 상승하여, 검색 기반 문맥 보강이 참조 리포트의 핵심 정보를 보다 폭넓게 포함하는 경향을 보인다.

검색 전략별 차이는 어휘적 균형과 의미적 정합성 지표에서 보다 뚜렷하게 나타났다. 키워드 기반 검색(BM25)은 1-shot 설정에서 ROUGE-1 Recall이 최대 81.5%로 가장 높게 나타났으나, 예시 수 증가에 따라 ROUGE-1 F1과 SBERT 유사도가 급격히 감소하였다. 예를 들어 BM25의 ROUGE-1 F1은 1-shot에서 62.7%였으나 3-shot에서는 49.1%로 감소하였고, SBERT 유사도 역시 91.3%에서 78.4%로 하락하였다. 이는 BM25가 어휘적 일치도를 기반으로 많은 정보를 회수하는 데에는 효과적이나, 예시 수 증가에 따라 어휘적 정보 포괄성이 확대되면서 어휘적 균형과 의미적 정합성이 함께 저하되는 경향이 관찰됨을 보인다.

의미 기반 검색(SBERT)은 CTQRS와 ROUGE-1 Recall에서 BM25와 유사한 수준의 성능을 보이면서도, 예시 수 증가에 따른 성능 저하 폭이 상대적으로 작게 나타났다. SBERT의 ROUGE-1 F1은 1-shot에서 62.2%, 3-shot에서 50.4%로 감소하였으며, SBERT 유사도는 동일 조건에서 91.1%에서 81.8%로 감소하였다. 이는 의미 기반 검색이 어휘적 일치보다는 의미적 정합성을 반영함으로써, 예시 수 증가 환경에서도 의미적 정합성을 상대적으로 안정적으로 유지함을 보여준다.

하이브리드 검색은 단일 검색 전략과 비교하여 전반적으로 중간적인 성능 분포를 보였으며, 지표 간 변동 폭이 상대적으로 작게 나타났다. 1-shot 설정에서 하이브리드 검색은 CTQRS 92.0%, ROUGE-1 Recall 81.3%, ROUGE-1 F1 62.4%, SBERT 유사도 91.0%를 기록하였다. 예시 수가 3-shot으로 증가한 경우에도 ROUGE-1 F1과 SBERT 유사도의 감소 폭은 BM25 단독 전략보다 완만하게 나타났으며, 이는 어휘적 정보 포괄성과 의미적 정합성 간의 균형이 상대적으로 안정적으로 유지되었음을 보인다.

이러한 특성은 두 검색 방식의 상호 보완적 성격에 기인하는 것으로 해석할 수 있다. BM25는 오류 메시지와 같은 명시적 기술 용어를 효과적으로 회수하는 반면, SBERT는 표현 차이가 존재하더라도 결함의 원인이나 동작 맥락이 유사한 사례를 포착하는 데 강점을 가진다. 하이브리드 전략은 이 두 신호를 결합함으로써 어휘 불일치나 의미적 모호성에 따른 검색 실패 가능성을 완화하고, 예시 수 증가 환경에서도 의미적 정합성과 어휘적 균형의 급격한 저하를 일부 완화한 것으로 보인다.

예시 수 변화에 따른 분석에서는 모든 검색 전략에서 CTQRS가 예시 수 증가에 따라 상승하는 경향을 보였다. 반면 ROUGE-1 F1과 SBERT 유사도는 예시 수가 증가할수록 감소하는 일관된 패턴을 보였으며, 특히 3-shot 설정에서 감소 폭이 두드러졌다. 이는 예시 수 증가가 구조적 형식 학습에는 긍정적으로 작용하는 반면, 어휘적 균형과 의미적 정합성 측면에서는 감소로 이어지는 trade-off 구조가 일관되게 관찰되었음을 의미한다.

재정렬 기법을 적용한 확장 설정의 경우, ROUGE-1 F1에서 최대 약 0.3%p 수준의 소폭 개선이 일부 설정에서 관찰되었으나, CTQRS와 ROUGE-1 Recall, SBERT 유사도에서는 일관된 향상이 나타나지 않았다. 또한 예시 수 증가에 따른 성능 감소 패턴은 재정렬 적용 여부와 무관하게 유지되었다. 이는 재정렬 기법이 검색 결과의 순위를 미세하게 조정하는 데에는 기여할 수 있으나, 전반적인 생성 품질의 변화 양상을 구조적, 어휘적, 의미적 지표 전반에서 일관되게 변화시키는 핵심 요인으로 작용하기에는 한계가 있음을 보인다.

5.2 위협 요인

본 연구는 검색 기반 문맥 보강 전략이 LLM 기반 버그 리포트 자동 생성 품질에 미치는 영향을 분석하기 위해, 파인튜닝된 언어 모델과 검색 기반 예시를 결합한 실험 설정을 사용하였다. 이와 같은 실험 구성은 데이터 구성, 학습 설정, 검색 범위, 그리고 결과 해석과 관련된 잠재적 위협 요인을 수반할 수 있다. 본 연구에서는 이러한 위협을 최소화하기 위해 실험 설계 단계에서 가능한 범위의 통제를 적용하였다.

먼저 파인튜닝 데이터 사용과 관련된 위협을 고려할 필요가 있다. 본 연구에서는 기존 연구에서 정제된 Bugzilla 기반 데이터셋을 사용하여 언어 모델을 파인튜닝하였으며, 학습, 검증, 테스트 세트를 사전에 명확히 분리하였다. 또한 검색 인덱싱은 학습 세트에 한정하여 수행함으로써, 평가 대상인 테스트 샘플과 동일한 사례가 검색 예시로 사용되지 않도록 통제하였다. 이러한 설계를 통해 검색 기반 문맥 보강 과정에서 발생할 수 있는 직접적인 데이터 누수 가능성을 최소화하였다. 따라서 본 연구에서 관찰된 성능 향상은 테스트 데이터에 대한 단순한 암기나 정답 노출로 설명되기 어렵다.

한편 파인튜닝 자체가 검색 기반 문맥 보강의 효과를 과대평가했을 가능성 또한 고려될 수 있다. 이를 완화하기 위해 본 연구에서는 모든 실험 설정에서 동일한 파인튜닝 모델과 동일한 하이퍼파라미터를 사용하였다. 즉, 파인튜닝은 검색 전략 간 비교에서 고정된 요소로 유지되었으며, 실험에서 관찰된 성능 차이는 검색 전략의 유형, 예시 수, 그리고 문맥 구성 방식의 차이에 기인하도록 설계되었다. 이에 따라 본 연구의 결과는 파인튜닝의 유무가 아니라, 동일한 파인튜닝 조건 하에서 검색 전략이 생성 품질에 미치는 상대적 영향을 중심으로 해석되어야 한다.

검색 예시가 생성 결과를 과도하게 유도하거나 특정 출력 패턴을 암기하게 만들었을 가능성도 위협 요인으로 고려될 수 있다. 본 연구에서는 검색된 예시를 few-shot 프롬프트 형태로 제공하였으나, 예시 수를 1-shot에서 3-shot까지 단계적으로 변화시키며 성능 변화를 분석하였다. 실험 결과, 예시 수 증가에 따라 구조적 완성도는 향상되었으나 어휘적 균형과 의미적 정합성은 감소하는 경향이 관찰되었다. 이는 검색 예시가 단순한 정답 암기나 복제를 유도했다기보다는, 문맥 보강 수준에 따라 상충 관계가 발생함을 보여주는 결과로 해석할 수 있다.

검색 전략과 파인튜닝 데이터 간의 결합 편향 역시 고려할 필요가 있다. 본 연구의 파인튜닝 데이터와 검색 인덱싱 데이터는 동일한 도메인(Bugzilla)에 기반하지만, 검색은 입력 요약과의 유사도를

기준으로 수행되었으며, 특정 출력 필드나 정답 문장을 직접적으로 회수하도록 설계되지 않았다. 또한 서로 다른 검색 전략(BM25, SBERT, 하이브리드)을 동일한 파인튜닝 모델에서 비교함으로써, 검색 전략 자체의 특성이 생성 품질에 미치는 영향을 분리하여 관찰할 수 있도록 하였다.

데이터셋의 일반화 가능성 또한 제한 요소로 작용할 수 있다. 본 연구는 Bugzilla 기반 데이터셋에 국한되어 수행되었으며, GitHub Issues나 Jira와 같이 리포트 형식과 사용자 서술 방식이 상이한 플랫폼에 대해 동일한 성능 경향이 유지된다고 단정할 수는 없다. 다만 본 연구의 목적은 특정 플랫폼에서의 절대적인 성능을 제시하는 것이 아니라, 검색 전략 선택에 따라 구조적 완성도, 어휘적 정보 포괄성, 어휘적 균형 간의 관계가 어떻게 변화하는지를 분석하는 데 있다. 이러한 분석 프레임워크는 다른 이슈 트래킹 시스템에도 유사한 분석 관점으로 확장 적용될 수 있다.

평가 지표와 관련해서는 자동 평가 지표 사용에 따른 제약이 존재한다. 본 연구는 CTQRS, ROUGE-1 Recall, ROUGE-1 F1, SBERT 유사도를 사용하여 생성 품질을 분석하였으며, 이는 구조적 품질과 내용 정합성을 정량적으로 비교하는 데 적합하다. 그러나 이러한 지표는 실제 개발자가 인지하는 가독성이나 실무적 유용성을 직접적으로 반영하지는 못한다. 특히 검색 기반 문맥 보강은 정보 포함을 증가시키는 특성상, 일부 설정에서 중복 표현이나 장황한 문장이 생성될 수 있으며 이는 ROUGE-1 F1 감소로 간접적으로 반영되었다. 또한 검색 단계의 품질을 Recall@k, nDCG, MRR과 같은 전통적인 정보검색 지표로 직접 평가하지 않고, CTQRS, ROUGE, SBERT 유사도와 같은 생성 기반 평가 지표를 통해 검색 결과가 생성 품질에 미치는 영향을 간접적으로 분석하였다. 이로 인해 검색 실패 사례나 검색 결과의 관련도 분포가 생성 결과의 저하에 어떠한 방식으로 기여하는지에 대한 정밀한 인과 분석은 제한적이다. 향후 연구에서는 검색 품질 지표와 생성 품질 지표를 함께 고려하여, RAG 파이프라인 내 검색 단계와 생성 단계 간의 영향 관계를 보다 체계적으로 분석할 필요가 있다.

통계적 분석 측면에서도 제한이 존재한다. 본 연구는 검색 전략 간 평균 성능 비교를 중심으로 분석하였으며, 모든 연구 질문에 대해 샘플 단위의 짝지은 통계 검정을 일관되게 적용하지는 않았다. 다만 재정렬 확장 설정(RQ4)에 대해서는 동일 테스트 인스턴스를 기준으로 paired bootstrap Δ 분석을 수행하여 추가 효과를 검증하였고, 그 외 설정에 대해서는 5,000회 부트스트랩 리샘플링을 통해 95% 신뢰 구간을 산출하였다. 보다 포괄적인 통계적 유의성 분석은 향후 반복 실험과 다중 시드를 통해 보완될 수 있다.

또한 본 연구에서 관찰된 성능 향상이 실제 유지보수 생산성 향상으로 직접 이어진다고 단정하기는 어렵다. 파인튜닝과 검색 기반 문맥 보강은 추가적인 계산 비용을 수반하며, 실무 적용 효과는 개발 워크플로우에 따라 달라질 수 있다. 따라서 본 연구의 결과는 검색 기반 문맥 보강 전략의 설계와 비교를 위한 분석적 관점에서 해석되어야 하며, 실제 개발자를 대상으로 한 사용자 연구와 시스템 수준의 평가가 필요하다. 본 연구의 분석 프레임워크는 다른 환경에도 적용될 수 있으나, 산업 환경에서의 유효성은 데이터 특성과 시스템 제약을 고려한 추가 검증이 요구된다.

6. 관련 연구

6.1 버그 리포트 요약 및 자동 생성

버그 리포트의 비정형성과 정보 과잉 문제를 완화하기 위해, 초기 연구들은 요약 자동화에 집중하였다. Rastkar et al. [25]은 핵심 문장을 추출하는 기법을, Liu et al. [26]은 딥러닝을 통해 문맥 흐름을 반영하는 요약 기법을 제안하여 결함 파악 효율을 높였다. 그러나 이러한 요약 중심 접근은 텍스트 길이를 줄이는 데에는 효과적이었으나, 결과물이 여전히 비정형 서술에 머물러 재현 절차, 실행 환경, 기대 결과와 같은 구조적 요소를 명시적으로 제공하지 못한다는 한계가 있었다. 즉, 요약 품질은 개선되었으나 실제 유지보수에 즉시 활용 가능한 구조적 완성도를 확보하지는 못한 것이다.

최근에는 LLM을 활용하여 요약을 넘어 구조화된 버그 리포트를 생성하려는 연구가 등장하고 있다. 이들 연구는 지시형 프롬프트를 통해 필수 항목을 포함하는 출력 형식을 유도함으로써 기존 요약 접근의 구조적 한계를 보완하고 있다. 그러나 기존의 생성 연구들은 주로 모델 자체의 성능이나 프롬프트 형식 설계에 초점을 맞추고 있으며, 생성의 재료가 되는 입력 문맥의 구성 방식이 품질에 미치는 영향은 충분히 다루지 않았다. 이에 본 연구는 생성 기법 자체보다는 생성 과정에 제공되는 문맥의 구성 방식(검색 전략)에 주목하여, 서로 다른 검색 전략과 예시 구성 선택이 생성 품질의 여러 속성 간 균형과 그 변화 양상에 어떠한 영향을 미치는지를 체계적으로 분석한다는 점에서 기존 연구들과 구별된다.

6.2 검색 기반 문맥 보강과 프롬프트 전략

유사한 과거 버그 리포트는 결함 재현 절차나 실행 환경과 같은 유용한 문맥을 제공하여 자동 생성 품질 향상에 기여한다. 최근에는 단순 키워드 매칭을 넘어 의미 기반 임베딩 이나 하이브리드 전략 을 통해 검색 안정성을 높이려는 시도가 활발하다. 이러한 검색 기술의 발전은 LLM의 프롬프트 기반 문맥 학습과 결합하며 그 중요성이 확대되었다. Brown et al. [27]이 보인 바와 같이, 검색된 유사 사례는 프롬프트 내의 예시로 주입되어 모델의 출력 방향성을 유도하는 핵심 인터페이스로 기능한다. 즉, 검색 전략은 단순한 정보 조회를 넘어 생성 모델의 문맥 품질을 결정하는 결정적인 설계 변수로 작용한다.

그러나 기존 연구들은 검색 정확도 향상이나 고정된 문맥 하에서의 프롬프트 기법 [28, 29, 30]을 상호 독립적으로 다루는 경향이 강했다. 검색 결과가 프롬프트 문맥으로 통합되는 방식과 그에 따른 생성 품질의 상관관계를 유기적으로 분석한 시도는 드물었다. 이에 본 연구는 검색 기반 문맥 보강을 프롬프트 구성의 핵심 변수로 정의하고, 키워드·의미·하이브리드 등 다양한 검색 신호가 버그 리포트의 구조적 완성도와 의미적 정합성에 미치는 영향을 체계적으로 규명함으로써 기존의 분절된 연구 한계를 극복한다.

6.3 생성 결과 평가 지표의 다차원적 확장

텍스트 생성 연구에서 결과물의 품질을 정량적으로 평가하는 것은 여전히 도전적인 과제이다. 전통적으로 자동 요약 및 생성 분야에서는 ROUGE 계열 지표가 표준으로 활용되어 왔으나 [11], 이는 표면적인 어휘 중복만을 측정할 뿐 생성된 텍스트의 의미적 정합성이나 정보 구조의 적절성을 반영하지 못한다는 한계가 지속적으로 지적되어 왔다 [31]. 이를 보완하기 위해 제안된 BERTScore [32]나 SBERT 유사도 [12]와 같은 의미 기반 지표들은 의미적 정합성을 평가하는 데에는 유용하지만, 버그 리포트와 같이 명확한 구조적 구성 요소를 요구하는 도메인 특화 문서의 품질을 판단하는 데에는 한계가 있다.

소프트웨어 공학 분야에서는 이러한 구조적 특성을 반영하기 위해 CTQRS와 같은 지표를 도입하여 버그 리포트의 구조적 충실도를 정량화해 왔다 [10]. CTQRS는 어휘적 유사도만으로는 포착하기 어려운 재현 가능성의 전제 조건, 즉 필수 항목의 포함 여부를 평가하는 데 적합하다. 그러나 기존의 LLM 기반 버그 리포트 자동 생성 연구들은 여전히 ROUGE-1 Recall과 같은 재현율 중심 지표에 크게 의존하는 경향이 있다 [7].

특히 본 연구와 같이 검색 기반 문맥 보강(RAG)을 적용하는 환경에서는 재현율 중심 평가의 한계가 더욱 두드러진다. 검색을 통해 외부 정보가 프롬프트에 주입될 경우, 모델이 핵심 정보를 더 많이 포함하게 되어 재현율은 자연스럽게 상승하지만, 동시에 불필요한 정보가 과도하게 생성되거나 출력이 장황해지는 부작용이 발생할 수 있기 때문이다. 따라서 생성 품질을 정확히 판단하기 위해서는 정보의 포괄성뿐만 아니라, 불필요한 중복을 억제하고 정보의 밀도를 유지했는지를 나타내는 정밀도 관점의 분석이 필요하다.

이에 본 연구는 기존의 CTQRS, ROUGE-1 Recall, SBERT 유사도 평가에 더해, ROUGE-1 F1 지표를 핵심 분석 도구로 도입하였다 [11]. ROUGE-1 F1 지표는 재현율과 정밀도의 조화 평균으로서, 검색 전략이 정보의 총량을 늘리는 데 기여하는지, 혹은 실질적인 정보의 정확도를 희생시키는지에 대한 상충 관계(trade-off)를 규명하는 결정적인 기준을 제공한다. 이러한 다각적인 지표 구성은 단일 지표 의존으로 인한 편향을 방지하고,

검색 기반 문맥 보강의 효과와 한계를 입체적으로 해석하기 위한 틀을 제공한다.

7. 결론

본 연구는 LLM 기반 버그 리포트 자동 생성 과정에서 검색 기반 문맥 보강(RAG) 전략이 생성 품질에 미치는 영향을 체계적으로 분석하였다. 이를 위해 키워드 기반 검색(BM25), 의미 기반 검색(SBERT), 하이브리드 검색 전략, 그리고 재정렬 기법을 포함한 확장 설정을 동일한 조건 하에서 비교하였다.

실험 결과, 검색 기반 문맥 보강은 검색을 사용하지 않는 생성 방식과 비교하여 버그 리포트의 구조적 완성도와 어휘적 정보 포괄성을 전반적으로 향상시키는 것으로 나타났다. 특히 하이브리드 검색 전략은 단일 검색 방식 대비 다양한 설정과 서로 다른 모델 전반에서 일관된 성능 특성을 보였다. 문맥 보강에 사용되는 예시 수가 증가할수록 구조적 완성도는 향상되는 반면, 어휘적 균형과 의미적 정합성은 저하되는 경향이 관찰되어, 문맥 보강 수준과 생성 품질 간의 상충 관계가 존재함이 관찰되었다. 한편 검색 결과에 재정렬 기법을 적용한 확장 설정은 의미적 정합성 측면에서는 제한적인 추가 효과를 보였으나, 구조적 완성도나 어휘적 품질을 전반적으로 향상시키는 데에는 효과가 일관되지 않았다.

평가 측면에서는 구조적 완성도의 대리 지표로서 CTQRS와 함께, 어휘적 정보 포괄성과 어휘적 균형을 각각 반영하는 ROUGE-1 Recall 및 F1, 그리고 의미적 정합성을 측정하는 SBERT 유사도를 종합적으로 고려함으로써, 검색 기반 문맥 보강이 생성 품질의 다양한 차원에 미치는 영향을 다각도로 분석하였다. 이를 통해 검색 기반 문맥 보강이 정보 포함을 증가시키는 동시에, 설정에 따라 불필요한 표현이나 의미적 정합성 저하를 유발할 수 있음을 정량적으로 확인하였다.

본 연구는 검색 기반 문맥 보강의 적용 여부 자체보다는, 검색 전략의 선택과 구성 방식이 LLM 기반 버그 리포트 자동 생성 품질의 여러 속성 간 상충 관계의 형태와 그 변화 양상에 영향을 미치는 설계 요소임을 실증적으로 분석하였다는 점에서 의의를 가진다. 향후 연구에서는 다양한 데이터 분할과 이슈 추적 시스템을 포함한 추가 실험과 실제 개발자를 대상으로 한 사용자 연구를 통해 본 연구에서 관찰된 결과의 일반성과 실무적 유효성을 추가로 검증할 필요가 있다.

참 고 문 헌

[1] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE), 2008, pp. 308-318.
[2] The Mozilla Foundation, "Bugzilla: A Bug Tracking System," [Online]. Available: https://www.bugzilla.org/. Accessed: Sep. 21, 2025.
[3] Atlassian, "Jira Software," [Online]. Available: https://www.atlassian.com/software/jira. Accessed: Sep. 21, 2025.
[4] S. Choi and G. Yang, "AgentReport: A Multi-Agent LLM Approach for Automated and Reproducible Bug Report Generation," Appl. Sci., vol. 15, no. 22, Art. no. 11931, Nov. 2025.
[5] X. Zhou, Y. Liu, J. Sun, and M. Li, "Automatic Bug Report Generation via Large Language Models," in Proc. IEEE Int. Conf. Softw. Maint. Evol. (ICSME), 2023, pp. 1-11.
[6] J. Chen, S. Kim, and T. Zimmermann, "An Empirical Study on the Use of Large Language Models for Bug Report Generation," IEEE Trans. Softw. Eng., early access, 2024.
[7] A. Acharya and R. Ginde, "RAG-based bug report generation with large language models," in Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE), Ottawa, ON, Canada, 2025.
[8] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), 2020, pp. 9459-9474.
[9] E. J. Hu et al., "LoRA: Low-rank adaptation of large language models," in Proc. Int. Conf. Learn. Represent. (ICLR), 2022.
[10] H. Zhang, Y. Zhao, S. Yu, and Z. Chen, "Automated quality assessment for crowdsourced test reports based on dependency parsing," in Proc. 9th Int. Conf. Dependable Syst. Their Appl. (DSA), 2022, pp. 34-41.
[11] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in Text Summarization Branches Out, 2004, pp. 74-81.
[12] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in Proc. Conf. Empir. Methods Nat. Lang. Process. (EMNLP), 2019, pp. 3982-3992.
[13] L. Ouyang et al., "Training language models to follow instructions with human feedback," in Adv. Neural Inf. Process. Syst. (NeurIPS), 2022, pp. 27730-27744.
[14] H. W. Chung et al., "Scaling instruction-finetuned LMs," arXiv preprint arXiv:2210.11416, 2022.
[15] S. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," Found. Trends Inf. Retr., vol. 3, no. 4, pp. 333-389, 2009.
[16] GindeLab, "Bug Report Summarization Benchmark Dataset," 2025. [Online]. Available: https://github.com/GindeLab/Ease_2025_AI_model
[17] J. Chen et al., "BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation," arXiv preprint arXiv:2402.03216, 2024.
[18] Mozilla, "Bug Writing Guidelines," [Online]. Available: https://bugzilla.mozilla.org/page.cgi?id=bug-writing.html. Accessed: Sep. 21, 2025.
[19] Q. Team, "Qwen2.5 technical report," arXiv preprint arXiv:2409.12121, 2024.
[20] A. Dubey et al., "The Llama 3 Herd of Models," arXiv preprint arXiv:2407.21783, 2024.
[21] A. Q. Jiang et al., "Mistral 7B," arXiv preprint arXiv:2310.06825, 2023.
[22] U. Team, "Unsloth: Efficient Fine-Tuning Framework for LLMs," GitHub Repository, 2025. [Online]. Available: https://github.com/unslothai/unsloth. Accessed: Sep. 21, 2025.
[23] A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), 2019, pp. 8024-8035.
[24] T. Wolf et al., "Transformers: State-of-the-Art Natural Language Processing," in Proc. Conf. Empir. Methods Nat. Lang. Process. (EMNLP): Syst. Demonstrations, 2020, pp. 38-45.
[25] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," IEEE Trans. Softw. Eng., vol. 40, no. 4, pp. 366-380, 2014.
[26] H. Liu et al., "BugSum: Deep context understanding for bug report summarization," in Proc. 28th Int. Conf. Program Comprehension (ICPC), 2020, pp. 94-105.
[27] T. Brown et al., "Language models are few-shot learners," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), 2020, pp. 1877-1901.
[28] X. Gu, H. Zhang, and S. Kim, "Deep code search," in Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE), 2018, pp. 933-944.
[29] F. Bruch, M. Decker, and M. L. Sadowski, "On the Role of Hybrid Retrieval in Modern Information Systems," in Proc. ACM Int. Conf. Inf. Knowl. Manag. (CIKM), 2022, pp. 1-10.
[30] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), 2022.
[31] N. Schluter, "The limits of automatic summarisation according to ROUGE," in Proc. Conf. Eur. Chapter Assoc. Comput. Linguistics (EACL), 2017, pp. 41-45.
[32] T. Zhang et al., "BERTScore: Evaluating text generation with BERT," in Proc. Int. Conf. Learn. Represent. (ICLR), 2020.

대규모 언어 모델 기반 무인공장 작업 정책 자동 생성

주은정 °[1],이정화 [1],류덕산 [2],백종문 [3]
㈜미라클에이지아이 [1],전북대학교 [2],한국과학기술원 [3]
{jeju3146, dlwjdghkl33}@miso.center, duksan.ryu@jbnu.ac.kr,jbaik@kaist.ac.kr

Automatic Generation of Work Policies for Unmanned Factories Based on Large Language Models

Eunjeong Ju °[1], Jeonghwa Lee[1], Duksan Ryu[2], Jongmoon Baik[3]
[1]Miracle-AGI,[2]Jeonbuk National University,[3]Korea Advanced Institute of Science and Technology (KAIST)

요약

무인 공장 환경에서는 작업자의 상시 개입이 제한되기 때문에, 공장 내 다양한 상황에 대해 일관되고 신속한 대응이 가능한 작업 메뉴얼의 중요성이 증가하고 있다. 그러나 기존 작업 메뉴얼은 수작업으로 작성·관리되는 경우가 많아, 공장 상황 변화에 유연하게 대응하기 어렵다는 한계를 가진다. 본 논문에서는 무인 공장에서의 상황 인지 정보를 기반으로 작업 메뉴얼을 자동으로 생성하는 대규모 언어모델 기반 접근 방법을 제안한다. 이를 위해 실제 공장 환경에서 발생할 수 있는 정적 상황을 반영한 시나리오를 구성하고, 각 상황에 대응하는 작업 메뉴얼 데이터셋을 구축하였다. 또한 Qwen, Mistral, LLaMA 계열의 대규모 언어모델을 활용하여 동일한 상황 시나리오에 대한 작업 메뉴얼 생성 결과를 비교·분석하였다. 실험에서는 데이터 규모의 한계로 인해 정량적 성능 지표 대신 전문가 기반 정성 평가를 수행하였으며, 작업 절차의 논리성, 현장 적용 가능성, 표현의 명확성을 중심으로 평가하였다. 실험 결과, 대규모 언어모델이 무인 공장 환경에서 상황 인지 기반 작업 메뉴얼 자동 생성에 활용 가능함을 확인하였다. 본 연구는 무인 공장 환경에서 작업 메뉴얼 자동 생성의 가능성을 제시한다.

Abstract

In unmanned factory environments, the importance of reliable and consistent work manuals has increased due to the limited availability of human operators. However, conventional work manuals are typically created and maintained manually, making them difficult to adapt to changing factory situations.
This paper proposes a large language model-based approach for automatically generating work manuals based on situation awareness in unmanned factories. Static situation scenarios that may occur in real factory environments are defined, and corresponding work manual datasets are constructed. Using these scenarios, work manuals are generated and analyzed using Qwen, Mistral, and LLaMA series large language models. Due to the limited size of the dataset, expert-based qualitative evaluation is conducted instead of quantitative performance metrics. The evaluation focuses on the logical consistency of work procedures, practical applicability, and clarity of expression. The experimental results demonstrate that large language models can effectively generate work manuals for situation-aware scenarios in unmanned factory environments. This study confirms the feasibility of automatic work manual generation and provides a foundation for future intelligent factory operation support systems.

1. 서 론

최근 제조 산업 전반에서 무인 공장 및 자동화 공정의 도입이 확대되면서, 공장 내 다양한 상황에 대응하기 위한 작업 메뉴얼의 중요성이 더욱 강조되고 있다.[1] 무인 공장 환경에서는 작업자의 상시 개입이 제한되기 때문에, 설비 상태나 작업 조건 변화에 따라 적절한 작업 절차를 신속하고 일관되게 제공하는 것이 필수적이다.[2] 그러나 기존의 작업 메뉴얼은 주로 수작업으로 작성·관리되며, 공장 상황 변화에 따른 즉각적인 반영이 어렵고 유지·보수 비용이 크다는 한계를 가진다.[3][4][5]

특히 무인 공장에서는 설비 고장, 공정 이상, 작업 조건 변화 등 다양한 상황이 발생할 수 있으며, 이러한 상황에 대응하기 위한 메뉴얼을 사전에 모두 정의하는 것은 현실적으로 어렵다.[6][7] 이로 인해 현장 적용 시 메뉴얼의 누락이나 비일관성이 발생할 수 있으며, 이는 공장 운영 효율 저하로 이어질 수 있다.

한편, 최근 대규모 언어모델(Large Language Models, LLMs)은 자연어 이해 및 생성 분야에서 뛰어난 성능을 보이며, 산업 도메인에서도 다양한 응용 가능성이 제시되고 있다. 이러한 언어모델은 주어진 상황 정보를 기반으로 절차적이고 일관된 텍스트를 생성할 수 있다는 점에서, 작업 메뉴얼 자동 생성에 활용될 잠

재력을 가진다.

본 논문에서는 무인 공장에서의 상황 인지 단계에서 수집된 공장 상황 정보를 기반으로, 작업 메뉴얼을 자동으로 생성하는 대규모 언어모델 기반 접근 방법을 제안한다. 이를 위해 실제 공장 환경을 고려한 상황 시나리오를 정의하고, 각 상황에 대응하는 작업 메뉴얼 데이터를 구축하였다. 또한 Qwen, Mistral, LLaMA 계열의 대규모 언어모델을 대상으로 동일한 상황 시나리오에 대한 작업 메뉴얼 생성 결과를 비교·분석함으로써, 무인 공장 환경에서 작업 메뉴얼 자동 생성의 가능성을 검토한다.

본 연구의 주요 기여는 다음과 같다.

- 첫째, 무인 공장 환경을 고려한 상황 시나리오–작업 메뉴얼 데이터 구축 방법을 제시한다.
- 둘째, 대규모 언어모델을 활용한 상황 인지 기반 작업 메뉴얼 자동 생성 방식을 제안한다.
- 셋째, 전문가 기반 정성 평가를 통해 모델 별 작업 메뉴얼 생성 특성을 분석하고, 대시보드 기반 결과 시각화를 통해 실용 가능성을 검토한다.

2. 관 련 연 구

제조 산업 분야에서는 공정 자동화와 스마트팩토리 확산에 따라, 공장 운영 과정에서 발생하는 다양한 정보를 효율적으로 활용하기 위한 연구가 지속적으로 이루어져 왔다. 특히 최근에는 대규모 언어모델(Large Language Models, LLMs)의 발전과 함께, 자연어 이해 및 생성 기술을 제조 환경에 적용하려는 시도가 활발히 진행되고 있다.

Li 등은 대규모 언어모델이 제조업 전반의 다양한 태스크를 지원할 수 있음을 체계적으로 분석하였다. 특히 공정 설명, 작업 지시 이해, 기술 문서 생성과 같은 제조 현장의 언어 중심 태스크뿐만 아니라, 설계·품질 관리·공급망 관리 등 복합적인 제조 시나리오에서 LLM의 활용 가능성을 제시하였다 [8]. 해당 연구는 LLM이 복잡한 제조 관련 지식을 자연어 및 멀티모달 형태로 처리·추론할 수 있음을 사례 기반으로 보였으며, 이를 통해 LLM이 단순한 텍스트 생성 도구를 넘어 제조 현장의 지식 관리 및 작업 지원을 위한 범용적 인공지능 도구로 활용될 수 있음을 논의하였다.

제조 도메인 특화 성능을 향상시키기 위한 연구도 수행되었다. Xia 등은 제조 도메인 말뭉치를 활용한 파인튜닝과 오류 기반 보정(error-assisted fine-tuning) 기법을 통해 대규모 언어모델의 도메인 적응 성능을 향상시키는 방법을 제안하였다 [9]. 해당 연구는 일반 목적의 언어모델을 제조 환경에 직접 적용할 경우 발생하는 한계를 지적하고, 오류 정보를 활용한 반복적 보정 과정을 통해 제조 질의 응답 및 코드 생성 성능을 개선할 수 있음을 보였다.

한편, LLM의 제조 산업 적용을 보다 거시적인 관점에서 분석한 연구도 존재한다. Wulf 등은 어포던스 이론(affordance theory)을 기반으로 대규모 언어모델이 제조 산업에서 설계, 품질 관리, 운영 의사결정 등 다양한 영역에서 가치를 창출할 수 있음을 분석하였다 [10]. 이 연구는 LLM이 제조 시스템 전반의 운영 효율성과 의사결정 지원 역량을 향상시킬 잠재력을 지니고 있음을 보여주는 동시에, 실제 현장 적용을 위해서는 구체적인 활용 시나리오와 적용 맥락의 정의가 필요함을 지적하였다.

제조 현장에서의 지식 공유 및 문서 활용을 지원하기 위한 연구도 보고되었다. Rossi 등은 LLM 기반 지식 검색 및 문서 활용 도구를 제안하여, 기술 문서와 운영 지식을 현장 작업 지원에 효과적으로 활용할 수 있음을 보였다 [11]. 해당 연구는 전문가 문서와 현장 데이터를 결합한 질의응답 시스템을 통해 정보 접근성과 문제 해결 효율을 향상시킬 수 있음을 보여주었으나, 상황 인지 결과를 입력으로 받아 작업 메뉴얼을 자동 생성하는 문제까지는 다루지 않았다.

산업 자동화 시스템 전반에서 LLM을 활용하려는 시도 또한 이루어지고 있다. Xia 등은 LLM을 지능형 산업 자동화 프레임워크에 통합하고, 이를 기반으로 사용자 질의 이해부터 작업 계획 및 수행까지 지원할 수 있는 LLM 기반 에이전트 구조를 제안하였다 [12]. 해당 연구는 LLM이 산업 자동화 환경에서 인간–시스템 상호작용을 개선하고, 보다 유연한 의사결정 및 작업 수행을 가능하게 할 수 있음을 사례 기반으로 보여주었다.

또한 Fakih 등은 LLM을 산업 제어 시스템에서 사용되는 PLC 코드 생성에 적용하고, 컴파일 및 형식 검증 결과를 기반으로 반복적으로 개선하는 프레임워크를 제안하였다 [13]. 이 연구는 LLM이 산업 자동화 영역에서 검증 가능한 코드와 같은 실질적인 작업 산출물을 생성할 수 있음을 보여주었으나, 작업 절차나 메뉴얼과 같은 문서 생성 문제를 직접적으로 다루지는 않았다.

이와 같이 기존 연구들은 제조 산업에서의 LLM 활용 가능성을 다양한 관점에서 탐구해 왔으나, **무인 공장 환경에서 상황 인지 결과를 입력으로 받아 작업 메뉴얼을 자동으로 생성하는 문제를 직접적으로 다룬 연구는 제한적이다**. 본 연구는 이러한 연구 공백을 보완하고자, 무인 공장의 상황 인지 정보를 기반으로 작업 메뉴얼을 자동 생성하고, 다양한 대규모 언어모델의 생성 특성을 비교·분석한다는 점에서 기존 연구와 차별성을 가진다.

3. 연 구 방 법

3.1 상황 인지 정보 정의 및 입력 구성

본 연구에서는 무인 공장의 **정적 상황 인지 단계**를 가정한다. 정적 상황 인지는 설비 상태, 공정 조건, 작업 환경 정보 등 일정 시점에서 변화하지 않는 공

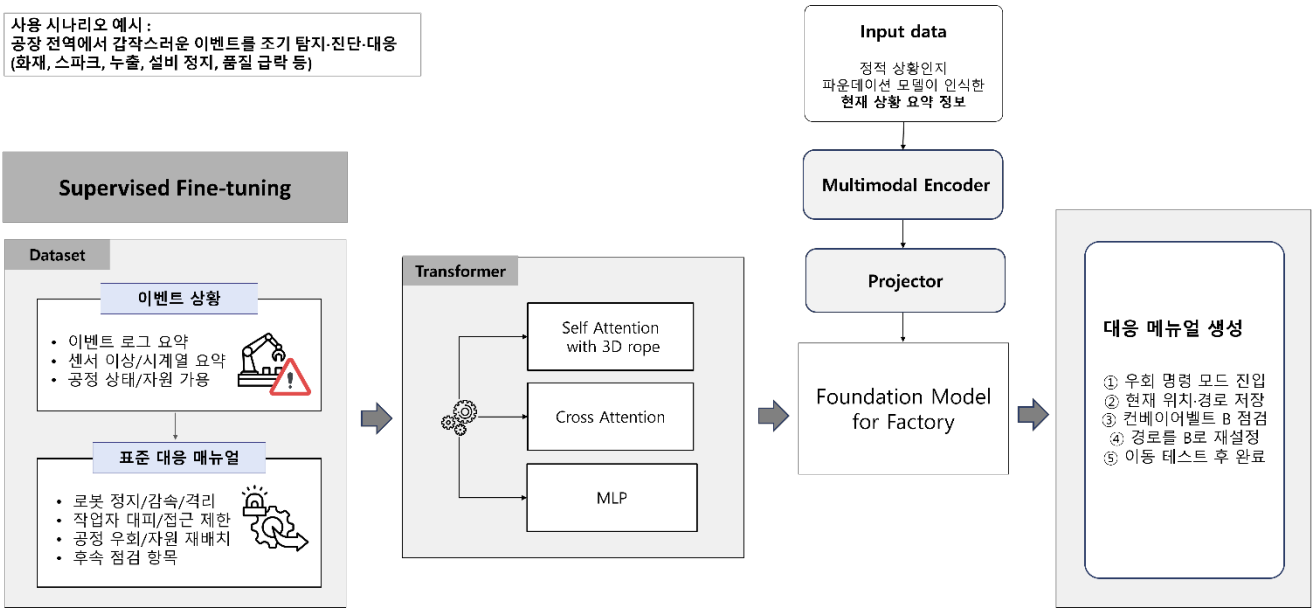


그림 1 전체 연구 과정

장 상황 정보가 사전에 정리되어 제공되는 단계를 의미한다. 이러한 상황 인지 정보는 센서 데이터나 상위 시스템에서 추출된 결과를 바탕으로, 자연어 기반의 시나리오 형태로 구성된다.

각 시나리오는 공장 내 특정 작업 또는 상황을 설명하는 텍스트로 구성되며, 설비 상태, 발생 가능한 문제 상황, 작업 조건 등의 정보를 포함한다. 본 연구에서는 이러한 시나리오를 대규모 언어모델의 입력으로 활용하여, 상황에 대응하는 작업 메뉴얼을 생성하도록 설계하였다.

3.2 작업 메뉴얼 자동 생성 구조

작업 메뉴얼 자동 생성 단계에서는 입력된 상황 시나리오를 바탕으로 대규모 언어 모델이 작업 절차를 자동으로 생성한다. 생성되는 작업 메뉴얼은 작업 단계, 주의 사항, 대응 절차 등을 포함하는 자연어 형태의 문서로 구성된다. Algorithm 1은 본 연구에서 사용한 작업 메뉴얼 자동 생성 절차를 나타낸다.

먼저, **라인 1–2**에서는 무인 공장 환경에서 수집된 상황 인지 정보를 **자연어 기반의 상황 시나리오 텍스트**로 구성한다. 본 연구에서는 모든 입력을 텍스트 형태로 가정하며, 설비 상태, 공정 조건, 작업 환경 정보는 외부 시스템 또는 사전 처리 과정을 통해 **텍스트로 요약된 형태**로 제공된다. 이를 통해 대규모 언어 모델이 직접 처리 가능한 텍스트 표현을 입력으로 사용한다.

이후 **라인 3–5**에서는 입력된 상황 시나리오에 대해 텍스트 정규화 및 전처리를 수행한다. 이 단계에서는 불필요한 표현을 제거하고, 단위 및 용어를 통일하며, 약어를 해소함으로써 상황 정보의 일관성을 확보한다. 전처리된 텍스트를 기반으로 작업 목표, 사용 설비, 자재, 위험 요소 및 제약 조건과 같은 **핵심 작업 요소를 구조적으로 추출**한다.

Algorithm 1 Text-only Work Policy Generation for Unmanned Factories

Require: Textual situation description S (e.g., operator notes, system logs in text, or pre-summarized sensor/tabular reports),
1: Policy template (optional) T,
2: Domain constraints/rules (optional) $\mathcal{R}$
Ensure: Generated work policy/manual P

3: **Step 1: Text preprocessing**
4: Normalize S (remove noise, unify units/terms, resolve abbreviations)
5: $S' \leftarrow \text{Preprocess}(S)$

6: **Step 2: Task element extraction**
7: Extract structured elements $\mathcal{E}$ from S' (e.g., goal, equipment, materials, hazards, constraints)
8: $\mathcal{E} \leftarrow \text{Extract}(S')$

9: **Step 3: Prompt construction**
10: Build prompt p using $\mathcal{E}$, optional template T, and constraints $\mathcal{R}$
11: $p \leftarrow \text{ComposePrompt}(\mathcal{E}, T, \mathcal{R})$

12: **Step 4: Policy generation with LLM**
13: Generate draft policy/manual $\tilde{P}$ using an LLM
14: $\tilde{P} \leftarrow \text{LLMGenerate}(p)$

15: **Step 5: Validation and post-processing**
16: Check format compliance, missing steps, safety constraints, and consistency
17: $P \leftarrow \text{ValidateAndRefine}(\tilde{P}, \mathcal{R})$

18: **return** P

알고리즘 1 작업 메뉴얼 생성 과정

다음으로 **라인 6–8**에서는 추출된 작업 요소와 사전에 정의된 정책 템플릿 및 도메인 제약 조건을 결합하여,

대규모 언어 모델에 입력될 프롬프트를 구성한다. 이 프롬프트는 작업 메뉴얼의 형식과 내용 구조를 유도하기 위한 역할을 한다.

라인 9–10에서는 구성된 프롬프트를 대규모 언어 모델에 입력하여 작업 메뉴얼 초안을 생성한다. 본 연구에서는 Qwen, Mistral, LLaMA 계열의 대규모 언어 모델을 대상으로 동일한 입력 시나리오와 프롬프트를 적용하여 작업 메뉴얼 생성 결과를 비교하였다. 이를 통해 모델 별 작업 메뉴얼 생성 특성과 표현 방식의 차이를 분석하였다.

마지막으로 **라인 11–12**에서는 생성된 작업 메뉴얼 초안에 대해 형식 검증, 누락된 작업 단계 확인, 안전 제약 조건 검토 등의 후처리 및 검증 과정을 수행한다. 이를 통해 작업 단계 중심으로 구조화된 최종 작업 메뉴얼을 출력하며, 생성된 결과는 실제 무인 공장 환경에서 활용 가능한 형태로 정리된다.

3.3 대시보드 기반 결과 시각화

생성된 작업 메뉴얼 결과를 직관적으로 분석하고 비교하기 위해 대시보드 기반 시각화 환경을 구축하였다. 사용자는 상황 시나리오를 입력한 후, 모델 별로 생성된 작업 메뉴얼 결과를 단계적으로 확인할 수 있다. 이를 통해 동일한 상황에 대해 서로 다른 대규모 언어모델이 생성한 작업 메뉴얼의 차이를 직관적으로 비교할 수 있다.

대시보드는 작업 단계별 텍스트 출력, 모델 간 결과 비교, 시나리오별 결과 탐색 기능 등을 제공하며, 실제 현장 적용 가능성을 검토하는 데 활용된다. 이러한 시각화 환경은 작업 메뉴얼 자동 생성 시스템의 실용성을 평가하기 위한 보조 도구로 활용된다.

3.4 접근 방법 요약

제안하는 접근 방법은 무인 공장의 상황 인지 정보를 자연어 시나리오로 변환하고, 이를 대규모 언어모델에 입력하여 작업 메뉴얼을 자동 생성하는 구조를 따른다. 이를 통해 사전에 모든 작업 메뉴얼을 수작업으로 정의하지 않고도, 다양한 공장 상황에 대해 유연하게 대응 가능한 작업 메뉴얼을 생성할 수 있다.

본 연구는 복잡한 제어 로직이나 실시간 판단을 포함하지 않고, 정적 상황 인지 단계를 대상으로 함으로써 시스템 구조를 단순화하였다. 이는 대규모 언어모델을 활용한 작업 메뉴얼 자동 생성의 가능성을 검증하기 위한 초기 단계 접근으로서 의의를 가진다.

4. 실 험 설 정

본 장에서는 제안한 무인 공장 상황 인지 기반 작업 메뉴얼 자동 생성 시스템의 유효성을 검증하기 위한 실험 설정과 평가 결과를 제시한다. 본 연구의 목적은 대규모 언어모델을 활용한 작업 메뉴얼 자동 생성의 가능성을 검토하는 것이므로, 다양한 상황 시나리오에 대해 생성된 작업 메뉴얼의 품질을 중심으로 평가를 수행하였다.

4.1 연구 질문

본 연구에서는 무인 공장 환경에서의 상황 인지 기반 작업 메뉴얼 자동 생성 가능성과, 대규모 언어모델 간 생성 특성의 차이를 분석하기 위해 다음과 같은 연구 질문을 설정하였다.

4.1.1 RQ1. 무인 공장의 정적 상황 인지 정보를 기반으로 대규모 언어모델이 작업 메뉴얼을 자동으로 생성할 수 있는가?

RQ1은 무인 공장 환경에서 상황 인지 정보를 입력으로 활용할 때, 대규모 언어모델이 실제 작업 메뉴얼로 활용 가능한 결과를 생성할 수 있는지를 검증하기 위한 질문이다. 본 연구에서는 설비 상태, 공정 조건, 작업 환경 정보 등을 포함하는 정적 상황 시나리오를 자연어 형태로 구성하고, 이를 대규모 언어모델의 입력으로 제공하였다. 생성된 작업 메뉴얼이 작업 단계, 주의 사항, 대응 절차 등을 포함하는지 여부를 중심으로 분석함으로써, 작업 메뉴얼 자동 생성의 가능성을 평가한다.

4.1.2 RQ2 동일한 상황 시나리오에 대해 서로 다른 대규모 언어모델은 작업 메뉴얼 생성 특성에서 어떤 차이를 보이는가?

RQ2는 동일한 공장 상황 시나리오가 주어졌을 때, 서로 다른 대규모 언어모델이 생성하는 작업 메뉴얼의 특성과 표현 방식에서 어떠한 차이가 나타나는지를 분석하기 위한 질문이다. 이를 위해 본 연구에서는 Qwen, Mistral, LLaMA 계열의 대규모 언어모델을 대상으로 동일한 입력 시나리오와 프롬프트 구조를 적용하였다. 모델 별로 생성된 작업 메뉴얼을 비교함으로써, 작업 절차의 구조화 수준, 표현의 명확성, 현장 적용 가능성 측면에서의 차이를 정성적으로 분석한다.

이러한 연구 질문을 통해 본 연구는 대규모 언어모델을 활용한 무인 공장 작업 메뉴얼 자동 생성의 실현 가능성을 검토하고, 모델 선택 및 향후 시스템 고도화를 위한 기초적인 분석 결과를 제공하고자 한다.

4.2 실험 설정

본 연구의 실험은 무인 공장 환경에서 작업 메뉴얼 자동 생성 과정을 비교·분석하기 위한 목적 하에 설계되었다. 이를 위해 모든 실험은 동일한 조건에서 수행되도록 구성하였으며, 입력 데이터와 프롬프트 구조를 고정한 상태에서 대규모 언어모델의 출력 특성을 관찰하는 방식으로 진행하였다. 상황 시나리오는 사전에 정의된 정적 상황 인지 결과를 기반으로 구성되었으며, 각 시나리오는 독립적으로 모델에 입

력되었다. 실험 과정에서는 하나의 시나리오에 대해 모델을 개별적으로 실행하고, 생성된 작업 메뉴얼 결과를 수집하였다. 모델 실행 순서나 외부 조건에 따른 영향을 최소화하기 위해 동일한 실행 환경과 파라미터 설정을 유지하였다.

생성된 작업 메뉴얼은 후속 분석을 위해 단계별 작업 절차를 중심으로 정리되었으며, 모델 별 결과 비교가 가능하도록 동일한 형식으로 저장되었다. 이러한 실험 설정을 통해 모델 간 생성 결과의 차이가 입력 조건이나 실행 환경이 아닌, 모델 자체의 특성에서 비롯되었는지를 분석할 수 있도록 하였다. 본 실험은 정량적 성능 수치 비교보다는, 동일 조건 하에서 생성된 작업 메뉴얼의 구조와 표현 차이를 관찰하는 데 초점을 두고 수행되었다. 이를 통해 연구 질문에서 설정한 대규모 언어모델의 작업 메뉴얼 생성 가능성과 모델 간 특성 차이를 효과적으로 분석하고자 하였다.

4.3 파인 튜닝 데이터셋 구성

본 연구에서는 무인 공장 환경에서의 작업 매뉴얼 자동 생성을 위해, 상황 시나리오-작업 매뉴얼 쌍으로 구성된 파인 튜닝 데이터셋을 구축하였다. 각 샘플은 특정 공장 상황을 서술한 **자연어 기반 상황 시나리오**와, 이에 대응하는 **작업 매뉴얼 텍스트**로 구성된다.
상황 시나리오는 설비 상태, 작업 조건, 작업 환경 및 발생 가능한 문제 상황을 포함하도록 설계되었으며, 모든 입력은 대규모 언어 모델이 직접 처리할 수 있도록 **텍스트 형태로만 구성**되었다. 작업 매뉴얼은 작업 단계, 주의 사항, 비정상 상황 대응 절차를 포함하는 자연어 문서로 작성되었다.
데이터는 AI 기반 방식으로 초안을 생성한 후, 작업 절차의 논리성, 안전성 및 현장 적용 가능성을 고려한 **수동 정제 과정**을 거쳤다. 구축된 데이터셋은 정상 작업과 이상 상황을 포함한 다양한 작업 유형을 반영하며, 본 연구에서는 데이터 규모의 한계로 인해 **작업 매뉴얼 생성 가능성을 검증하기 위한 경량 파인 튜닝**에 활용되었다.

표 1 데이터셋 설명

구분	내용
데이터 구성 단위	상황 시나리오-작업 매뉴얼 pair
입력 데이터	텍스트 기반 상황 시나리오
출력 데이터	작업 단계/주의사항/대응절차를 포함한 작업 매뉴얼
총 데이터 수	300 pairs
상황 유형	정상 작업, 설비 이상, 공정지연, 작업 오류 대응
도메인 범위	무인 공장 공정 운영 및 설비 관리
생성 방식	Ai 기반 초안 생성 후 수동 정제
정제 기준	의미적 일관성, 작업 절차 순차성, 안전성, 현실성
활용 목적	LLM 파인튜닝 및 작업 매뉴얼 생성 성능 평가

4.4 평가 방법

본 연구에서는 데이터 규모의 한계와 작업 매뉴얼 생성 문제의 특성으로 인해, 전문가 기반 정성 평가(expert-based qualitative evaluation)를 수행하였다. 평가는 데이터 기반 분석 경험을 갖춘 **AI 전문가 5명**에 의해 이루어졌으며, 각 평가자는 머신러닝 또는 대규모 언어모델 분야에서 **3년 이상의 연구 또는 실무 경험**을 보유하고 있다.

동일한 상황 시나리오에 대해 각 대규모 언어모델이 생성한 작업 매뉴얼을 제시하고, **모델 정보를 블라인드 처리한 상태에서** 비교·분석을 수행하였다. 평가는 정상 및 비정상 상황을 포함한 **N개의 시나리오**를 대상으로 진행되었다.

평가 기준은 (1) **작업 절차의 논리성**, (2) **현장 적용 가능성**, (3) **표현의 명확성**의 세 가지 항목으로 정의하였으며, 각 항목은 5점 리커트 척도(1-5)로 평가되었다. 점수 1은 기준을 전혀 충족하지 못하는 경우, 점수 3은 부분적으로 충족하는 경우, 점수 5는 기준을 충분히 충족하는 경우로 정의하였다. 평가 결과는 평가자별 점수를 종합하여 산출하였으며, 평가 일관성을 확인하기 위해 **평가자 간 일치도**를 함께 분석하였다.

본 평가는 BLEU, ROUGE와 같은 자동 정량 지표로는 충분히 반영하기 어려운 **작업 절차의 구조적 완성도와 표현 특성**을 비교·분석하는 데 목적이 있다.

5. 실 험 결 과

본 장에서는 제안한 무인 공장 상황 인지 기반 작업 메뉴얼 자동 생성 시스템의 실험 결과를 제시한다. 동일한 상황 시나리오에 대해 Qwen, Mistral, LLaMA 계열 대규모 언어모델이 생성한 작업 메뉴얼을 대상으로 전문가 기반 정성 평가를 수행하였으며, 모델 별 생성 특성을 비교·분석하였다.

5.1 전문가 정성 평가 결과 (RQ1의 실험 결과)

전문가 평가는 정성 평가를 수행하였으며, 모델 별 생성 특성을 비교·분석하였다.

표 2 모델 별 작업 메뉴얼 정성 평가 결과

모델	작업 절차 논리성	현장 적용 가능성	표현의 명확성	종합 평가
Qwen	3	3	3	3

Mistral	3	4	3	3
LLaMA	5	5	5	5

※ 평가는 전문가 주관적 판단에 따른 상대적 비교 결과임

표 2는 모델 별 작업 메뉴얼 생성 결과에 대한 정성 평가 요약을 나타낸다. **(1점: 기준을 전혀 충족하지 못함, 3점: 기준을 부분적으로 충족함, 5점: 기준을 충분히 충족함(2점,4점은 중간수준으로 해석)**

평가 결과, 동일한 상황 시나리오에 대해서도 대규모 언어모델에 따라 작업 메뉴얼의 구성 방식과 표현 특성에서 차이가 나타났다. 모든 모델은 작업 절차와 대응 방법을 포함하는 작업 메뉴얼을 생성할 수 있었으나, 생성된 결과의 구조화 수준과 구체성에는 차이가 존재하였다.

LLaMA 계열 모델은 작업 절차를 단계적으로 명확하게 구분하여 제시하는 경향을 보였으며, 각 단계에서 수행해야 할 작업 내용과 주의 사항을 비교적 구체적으로 서술하였다. 이로 인해 작업 절차 논리성, 표현의 명확성, 현장 적용 가능성 측면에서 모두 높은 평가를 받았다. 이러한 결과는 LLaMA 계열 모델이 절차 중심 문서 생성 및 지시 기반 응답 생성에 강점을 가지는 특성과 관련된 것으로 해석된다.

Mistral 계열 모델은 전반적으로 안정적인 작업 메뉴얼을 생성하였으며, 작업 단계 간 흐름은 비교적 자연스럽게 유지되었다. 다만 일부 시나리오에서는 작업 절차가 포괄적으로 서술되어, 단계별 세부 설명이 충분히 구체화되지 않은 경우가 관찰되었다. 이에 따라 현장 적용 가능성 측면에서는 중간 수준의 평가를 받은 것으로 분석된다.

Qwen 계열 모델의 경우, 상황에 대한 전반적인 설명은 비교적 충실하였으나, 상황 설명과 작업 지침이 혼합되어 표현되는 사례가 관찰되었다. 이로 인해 작업 단계의 경계가 명확하지 않은 경우가 발생하였으며, 작업 메뉴얼의 구조화 수준 측면에서 상대적으로 낮은 평가를 받았다.

이러한 결과는 대규모 언어모델의 학습 데이터 구성과 생성 전략 차이가 작업 메뉴얼 생성 특성에 직접적인 영향을 미친 것으로 판단된다.

5.2 대시보드 기반 비교 결과(RQ2의 실험 결과)

모델 별 작업 메뉴얼 생성 결과를 보다 직관적으로 비교·분석하기 위해 대시보드 기반 시각화 분석을 수행하였다. 대시보드에서는 동일한 상황 시나리오에 대해 모델 별 작업 메뉴얼을 나란히 제시함으로써, 작업 단계 구성 방식과 표현 차이를 쉽게 확인할 수 있도록 구성하였다.

대시보드 분석 결과, **LLaMA 계열 모델**은 작업 메뉴얼을 명확한 단계 중심 구조로 생성하는 경향을 보였다. 작업 단계가 번호 기반으로 명확히 구분되어 제시되었으며, 각 단계에서 수행해야 할 작업 내용과 주의 사항이 비교적 간결하면서도 구체적으로 서술되었다. 이러한 특성은 표 3에서 제시된 바와 같이 작업 단계 구분, 주의 사항 명시, 절차 일관성 항목에서 모두 높은 평가를 받은 결과와 일관된 경향을 보인다. 특히 전반적 구조화 수준이 우수한 것으로 평가되어, 작업 매뉴얼 로서의 가독성과 현장 적용 가능성이 높은 형태를 제공하는 것으로 분석된다.

Mistral 계열 모델의 경우, 대시보드 상에서 작업 단계는 비교적 명확하게 구분되었으나, 일부 단계에서 설명이 다소 포괄적으로 제시되는 경향이 관찰되었다. 작업 절차의 흐름은 자연스럽게 유지되었으며, 표 3에서도 절차 일관성 항목에서 양호한 평가를 받은 것으로 나타났다. 다만 주의 사항 명시와 작업 단계의 세부 구조 측면에서는 LLaMA 계열 모델에 비해 다소 간략한 표현을 사용하여, 전반적 구조화 수준은 중간 수준으로 평가되었다. 이는 안정적인 생성 특성을 보이지만, 현장 적용 시 추가적인 해석이 요구될 수 있음을 시사한다.

반면, **Qwen 계열 모델**은 대시보드 화면에서 상황 설명과 작업 지침이 하나의 서술 흐름으로 혼합되어 표현되는 사례가 상대적으로 두드러졌다. 작업 단계가 명시적으로 구분되지 않거나 단계 경계가 불명확한 경우가 관찰되었으며, 이로 인해 작업 절차를 순차적으로 파악하는 데 다소 어려움이 있었다. 이러한 특성은 표 3에서 작업 단계 구분 및 전반적 구조화 수준이 낮게 평가된 결과와 일치한다. 다만, 상황에 대한 배경 설명과 문제 원인 서술은 비교적 충실하여, 상황 이해 측면에서는 장점을 가지는 것으로 분석된다.

표 3에 제시된 정성 비교 결과는 대시보드 기반 시각적 분석에서 관찰된 특성과 전반적으로 일관된 경향을 보인다. 즉, LLaMA 계열 모델은 구조화된 작업 메뉴얼 생성에 강점을 보였고, Mistral 계열 모델은 안정적인 절차 흐름을 유지하는 중간적 특성을, Qwen 계열 모델은 서술 중심의 설명 능력을 중심으로 한 생성 특성을 보였다.

종합적으로, 대시보드 및 정성 지표 기반 비교 분석 결과는 동일한 상황 시나리오에 대해서도 대규모 언어모델 간 작업 메뉴얼 생성 특성이 뚜렷하게 다름을 보여준다. 이는 RQ2에서 제기한 모델 간 생성 특성 차이가 실제 출력 결과에서도 명확히 관찰됨을 의미하며, 무인 공장 환경에서 작업 메뉴얼 자동 생성 시스템을 운용할 때 목적에 따라 적합한 모델을 선택할 필요가 있음을 시사한다.

표 3 모델 별 작업 메뉴얼 생성 특성 비교

항목	Qwen	Mistral	LLaMA
작업 단계 구분	미흡	보통	명확
주의 사항 명시	보통	보통	우수
절차 일관성	보통	양호	우수
전반적 구조화 수준	미흡	보통	우수

표 4 Qwen 입출력 예시

모델	Qwen
입력	공장 A 구역에서 자율 이동 로봇이 주행 중 넘어졌고, 주변에 작업자 2명이 존재하는 상황이다.
출력	1. 즉각적으로 해야 할 조치: 작업자들은 로봇 주변을 피하고, 로봇 주변에 있는 물체를 피하며 안전 거리로 이동하고, 로봇 사고 발생 사실을 공장 안전 담당자에게 신고하고 대기한다. 2. 로봇 상태 점검 항목: 로봇의 전원 상태, 로봇의 자율 주행 모드 상태, 로봇의 센서 정상 작동 여부, 로봇의 충돌 시 정지 기능 정상 작동 여부 3. 작업자 안전 관련 조치: 작업자들은 로봇 주변에 가까운 지역에서 작업을 하지 않고, 로봇 사고 발생 후 로봇 주변에 가까운 지역에서 작업을 하지 않는다. 4. 실패 시 대처 조치: 로봇 수리 팀이 로봇을 수리하거나 교체하는 작업을 수행한다. 로봇 수리 시간 동안 작업을 수행하기 위해 대체 로봇을 배치하거나 작업자 수를 증가시킨다. 5. 실패 시 봐야 할 항목: 로봇의 소프트웨어 버그, 하드웨어 문제, 로봇의 센서 정상 작동 여부

표 5 Mistral 입출력 예시

모델	Mistral
입력	공장 A 구역에서 자율 이동 로봇이 주행 중 넘어졌고, 주변에 작업자 2명이 존재하는 상황이다.
출력	1. 즉각적으로 해야 할 조치: 작업자들은 로봇 주변을 피하고, 로봇 정지 버튼을 눌러 로봇을 정지시키고, 공장 안전 담당자에게 사고 상황을 신속하고 대응 방안을 듣는다. 2. 로봇 상태 점검 항목: 로봇의 전원 상태, 로봇의 주행 모드, 로봇의 센서 정상 작동 여부, 로봇의 충돌 이력 3. 작업자 안전 관련 조치: 작업자들은 로봇 주변에 가까운 지역에서 멀리 떨어지고, 안전 장비를 착용하고, 로봇 상태를 점검하기 전에 로봇이 정지되었는지 확인한다. 4. 실패 시 대처 조치: 로봇이 정상적으로 작동하지 않을 경우, 로봇 유지 보수 팀에 문의하고, 로봇 대체 로봇을 배치하여 작업을 계속하거나, 작업을 일시 중단하고 로봇 유지 보수를 수행한다.

표 6 LLaMA 입출력 예시

모델	LLaMA
입력	공장 A 구역에서 자율 이동 로봇이 주행 중 넘어졌고, 주변에 작업자 2명이 존재하는 상황이다.
출력	1. 즉각적으로 해야 할 조치: 작업자들은 로봇 주변을 피하고, 로봇 정지 버튼을 눌러 로봇을 정지시키고, 공장 안전 담당자에게 사고 상황을 신고하고 대응 방안을 듣는다. 2. 로봇 상태 점검 항목: 로봇의 전원 상태, 로봇의 주행 모드, 로봇의 센서 정상 작동 여부, 로봇의 충돌 이력 3. 작업자 안전 관련 조치: 작업자들은 로봇 주변에 가까운 지역에서 멀리 떨어지고, 안전 장비를 착용하고, 로봇 상태를 점검하기 전에 로봇이 정지되었는지 확인한다. 4. 실패 시 대처 조치: 로봇이 정상적으로 작동하지 않을 경우, 로봇 유지 보수 팀에 문의하고, 로봇 대체 로봇을 배치하여 작업을 계속하거나, 작업을 일시 중단하고 로봇 유지 보수를 수행한다.

5.3 성능에 영향을 미치는 주요 요인 분석

앞선 실험 결과를 바탕으로, 무인 공장 상황 인지 기반 작업 메뉴얼 자동 생성 메커니즘을 활용할 때 생성 성능에 영향을 미치는 주요 요인을 분석한다. 본 연구의 메커니즘은 정적 상황 인지 정보를 자연어 시나리오로 변환하고, 이를 대규모 언어모델에 입력하여 작업 메뉴얼을 생성하는 구조를 따른다. 실험 결과를 종합하면, 작업 메뉴얼 생성 성능은 단일 요소가 아닌 여러 요인의 복합적인 영향에 의해 결정되는 것으로 분석된다.

(1) 입력 시나리오의 구조화 수준

작업 메뉴얼 생성 성능에 가장 큰 영향을 미치는 요인 중 하나는 입력되는 상황 시나리오의 구조화 수준이다. 상황 시나리오가 설비 상태, 문제 상황, 작업 조건 등을 명확하게 구분하여 기술할수록, 모델은 이를 기반으로 작업 단계를 보다 논리적으로 구성하는 경향을 보였다. 반면, 상황 정보가 서술적으로 혼합되어 제공될 경우, 작업 단계 구분이 불명확한 결과가 생성되는 사례가 관찰되었다. 이는 작업 메뉴얼 자동 생성에서 입력 시나리오 설계가 중요한 선행 요소임을 시사한다.

(2) 대규모 언어모델의 절차 중심 문서 생성 특성

모델 자체의 학습 특성과 생성 전략 또한 작업 메뉴얼 생성 성능에 중요한 영향을 미친다. 실험 결과, 절차 중심 문서나 지시 기반 응답 생성에 강점을 가진 모델일수록 작업 단계 구분이 명확하고 구조화된

작업 메뉴얼을 생성하는 경향을 보였다. 이는 대규모 언어모델이 학습 과정에서 접한 데이터 유형과 응답 생성 방식이 작업 메뉴얼과 같은 문서 생성 과제에 직접적인 영향을 미침을 의미한다.

(3) 출력 형식에 대한 제약 및 프롬프트 설계

프롬프트에서 작업 단계 구분, 출력 형식, 응답 범위 등을 얼마나 명확히 제약하는지도 생성 성능에 영향을 미치는 요인으로 분석되었다. 출력 형식에 대한 제약이 상대적으로 약한 경우, 모델은 상황 설명 중심의 서술적 응답을 생성하는 경향을 보였으며, 이는 작업 메뉴얼로서의 구조적 명확성을 저하시킬 수 있다. 반면, 단계 중심 출력 형식을 명시적으로 유도한 경우, 작업 절차의 일관성과 가독성이 향상되는 경향이 관찰되었다.

(4) 후처리 및 결과 정제 단계의 역할

작업 메뉴얼 자동 생성 과정에서 후처리 단계 역시 전체 성능에 영향을 미치는 요소로 작용한다. 생성된 결과를 작업 단계 중심으로 정리하고, 중복되거나 불필요한 설명을 제거하는 후처리 과정을 거칠 경우, 작업 메뉴얼의 활용성이 향상되는 것으로 분석되었다. 이는 대규모 언어모델의 출력 결과를 그대로 활용하기보다는, 후처리 과정을 통해 실제 현장 적용에 적합한 형태로 정제할 필요가 있음을 시사한다.

(5) 메커니즘 관점의 종합 분석

이상의 분석을 종합하면, 무인 공장 상황 인지 기반 작업 메뉴얼 자동 생성 메커니즘의 성능은 입력 시나리오 설계, 모델의 생성 특성, 프롬프트 제약 방식, 그리고 후처리 전략이 상호작용한 결과로 결정된다. 즉, 특정 모델의 성능 우수성만으로 작업 메뉴얼 생성 품질이 보장되는 것이 아니라, 전체 메커니즘을 어떻게 구성하고 운용하는지가 중요한 요인으로 작용한다.

이는 대규모 언어모델을 활용한 작업 메뉴얼 자동 생성 시스템을 실제 무인 공장 환경에 적용할 경우, 모델 선택뿐만 아니라 입력 정보 구성과 출력 정제 전략을 함께 고려해야 함을 의미하며, 향후 시스템 고도화를 위한 중요한 시사점을 제공한다.

6. 위 협 요 소

본 연구는 무인 공장에서의 상황 인지 기반 작업 메뉴얼 자동 생성 가능성을 검토하는 초기 단계 연구로서, 다음과 같은 위협 요소를 가진다.

첫째, 데이터 규모의 한계이다. 본 연구에서 사용한 상황 시나리오-작업 메뉴얼 데이터셋은 실제 공장 환경을 반영하여 구성되었으나, 데이터 수가 제한적이어서 모든 공장 상황을 포괄하지는 못한다. 이에 따라 생성 결과의 일반화에는 한계가 존재할 수 있다.

둘째, 평가 방식의 주관성이다. 본 연구에서는 정량적 성능 지표 대신 전문가 기반 정성 평가를 수행하였다. 이는 작업 메뉴얼의 실용성을 평가하는 데 적합한 방법이지만, 평가자의 주관이 결과에 영향을 미칠 가능성이 있다.

셋째, 정적 상황 인지 가정이다. 본 연구는 정적 상황 인지 단계를 가정하고 실험을 수행하였으며, 실시간 상황 변화나 동적 이벤트는 고려하지 않았다. 따라서 실제 무인 공장 환경에서 발생할 수 있는 복합적인 상황을 충분히 반영하지 못할 수 있다.

마지막으로, 모델 의존성 문제이다. 실험에 사용된 대규모 언어모델의 학습 데이터와 구조적 특성에 따라 생성 결과가 영향을 받을 수 있으며, 이는 모델 간 비교 결과에 편향을 유발할 가능성이 있다.

7. 결 론 및 향 후 과 제

본 논문에서는 무인 공장에서의 상황 인지 정보를 기반으로 작업 메뉴얼을 자동으로 생성하는 대규모 언어모델 기반 접근 방법을 제안하였다. 실제 공장 환경을 고려한 정적 상황 시나리오를 구성하고, Qwen, Mistral, LLaMA 계열의 대규모 언어모델을 활용하여 작업 메뉴얼 자동 생성 가능성을 검토하였다. 전문가 기반 정성 평가를 통해, 대규모 언어모델이 무인 공장 환경에서 작업 절차와 대응 방법을 자동으로 생성하는 데 활용될 수 있음을 확인하였다.

향후 연구에서는 다음과 같은 방향으로 연구를 확장할 계획이다. 첫째, 상황 시나리오-작업 메뉴얼 데이터셋을 확장하여 보다 다양한 공장 환경과 상황을 포괄하는 정량적 성능 평가를 수행할 예정이다. 둘째, 정적 상황 인지 단계를 넘어 실시간 상황 변화가 반영되는 동적 환경으로 연구 범위를 확장할 계획이다. 셋째, 생성된 작업 메뉴얼의 신뢰성과 안전성을 향상시키기 위한 후처리 및 검증 기법을 도입할 예정이다. 마지막으로, 실제 무인 공장 시스템과의 연계를 통해 현장 적용 가능성을 검증하는 실증 연구를 진행할 계획이다.

감사의 글

본 논문은 2025년도 정부(과학기술정보통신부)의 재원으로 정보통신산업진흥원-피지컬AI 선도모델 수립 및 PoC 사업의 지원을 받아 수행된 연구임(2025, PJT-25-080036)

참 고 문 헌

[1] Wang, Lihui, et al. "Human-centric assembly in smart factories." CIRP Annals (2025).

[2] Fauska, Christian, and Jaroslava Kniežová. "Information and communication integration in smart factory design." F1000Research 11 (2023): 1026.

[3] Letmathe, Peter, and Marc Rößler. "Should firms use digital work instructions?—Individual learning in an agile manufacturing setting." Journal of operations management 68.1 (2022): 94-109.
[4] Rodriguez, Francisca S., et al. "Performance differences between instructions on paper vs digital glasses for a simple assembly task." Applied Ergonomics 94 (2021): 103423.
[5] Eversberg, Leon, and Jens Lambrecht. "Evaluating digital work instructions with augmented reality versus paper-based documents for manual, object-specific repair tasks in a case study with experienced workers." The International Journal of Advanced Manufacturing Technology 127.3 (2023): 1859-1871.
[6] Han, Pengyu, et al. "Multi-condition fault diagnosis of dynamic systems: A survey, insights, and prospects." IEEE Transactions on Automation Science and Engineering (2025).
[7] Webert, Heiko, et al. "Fault handling in industry 4.0: definition, process and applications." Sensors 22.6 (2022): 2205.
[8] Y. Li, Y. Zhang, H. Wang, et al., “Large Language Models for Manufacturing,” arXiv preprint, arXiv:2410.21418, 2024.
[9] Y. Zhang, J. Liu, S. Wang, et al., “Leveraging Error-Assisted Fine-Tuning of Large Language Models for Manufacturing Domain Adaptation,” Journal of Manufacturing Systems, vol. 73, pp. 1–14, 2024.
[10] J. Wulf, T. Bauernhansl, and A. Verl, “On the Value Potential of Large Language Models in the Manufacturing Industry,” in Proceedings of the Smart Services Summit, Springer, pp. 123–134, 2024.
[11] F. Rossi, M. Gualtieri, and A. Di Nuovo, “Knowledge Sharing in Manufacturing Using LLM-Powered Tools: A User Study,” Frontiers in Artificial Intelligence, vol. 7, Article 1293084, 2024.
[12] Y. Xia, N. Jazdi, and M. Weyrich, “Applying Large Language Models for Intelligent Industrial Automation,” atp edition – Automatisierungstechnische Praxis, vol. 66, no. 4, pp. 56–63, 2024.
[13] M. Fakih, M. B. Alawieh, and I. H. Elhajj, “LLM4PLC: Harnessing Large Language Models for Verifiable Programming of PLCs in Industrial Control Systems,” arXiv preprint, arXiv:2401.05443, 2024.

KAN-BE: 파라미터 효율적 앙상블을 활용한 소프트웨어 결함 예측

최윤서[1O], 류덕산[1], 백종문[2]
전북대학교[1], 한국과학기술원[2]
{dbstj253, duksan.ryu}@jbnu.ac.kr, jbaik@kaist.ac.kr

KAN-BE: Software defect prediction using parameter-efficient ensembles

Yunseo Choi[1O], Duksan Ryu[1], Jongmoon Baik[2]
Jeonbuk National University[1], KAIST[2]

요 약

소프트웨어 결함 예측(Software Defect Prediction, SDP)은 결함 가능성이 높은 모듈을 사전에 식별하여 소프트웨어 품질을 향상시키고 유지보수 비용을 절감하는 데 중요한 역할을 한다. 최근 소프트웨어 메트릭 간 상호작용 학습을 위한 신경망 기반 모델과 성능 향상을 위한 앙상블 기법이 널리 활용되고 있으나, 기존 앙상블 방식은 높은 파라미터 수와 긴 추론 시간으로 인한 효율성 문제가 있다. 본 연구에서는 SDP에서의 성능과 자원 소모 문제를 개선하기 위해, 복잡한 비선형 관계 학습에 강점을 가진 KAN 모델과 파라미터 사용 효율이 높은 BatchEnsemble 기법을 결합한 KAN-BE 모델을 제안한다. 실험 결과, 제안 기법은 비교 모델 대비 전반적인 지표에서 우수한 성능을 보였으며, 특히 KAN-Deep Ensemble 대비 파라미터 수를 약 78% 절감하고 추론속도를 약 2.2배 단축하여 성능과 효율을 동시에 달성하였다. 또한 앙상블 구조의 불확실성 정보를 기반으로 예측 신뢰도를 분석함으로써, 본 연구는 제한된 소프드웨어 품질 보증 자원을 보다 효과적으로 활용하는 데 기여할 수 있을 것으로 기대된다.

Abstract

Software defect prediction (SDP) plays an important role in improving software quality and reducing maintenance costs by identifying modules with high probability of defects in advance. Recently, neural network-based models for learning interactions between software metrics and ensemble techniques for improving performance have been widely used, but existing ensemble methods have efficiency problems due to high number of parameters and long inference time. In this study, to improve the performance and resource consumption problem in SDP, we propose a KAN-BE model that combines a KAN model with strength in learning complex nonlinear relationships and a Batch Ensemble technique with high parameter usage efficiency. As a result of the experiments, the proposed technique showed superior performance in the overall indicators compared to the comparative model, and in particular, it achieved both performance and efficiency by reducing the number of parameters by about 78% compared to KAN-Deep Ensemble and reducing the inference speed by about 2.2 times. In addition, by analyzing the prediction reliability based on the uncertainty information of the ensemble structure, this study is expected to contribute to the more effective utilization of limited software quality assurance resources.

1. 서 론

소프트웨어 결함 예측은 소프트웨어 시스템에서 결함 가능성이 높은 모듈을 사전에 식별하여 소프트웨어 품질을 향상시키고 유지보수 비용을 절감하기 위해 중요한 연구 분야이다. 소프트웨어 결함은 시스템의 신뢰성에 직접적인 영향을 미치며, 결함 수정 시점이 늦어질수록 비용이 급격히 증가한다. 이러한 SDP 분야에서 소프트웨어 메트릭 간의 복잡한 관계는 결함 예측을 어렵게 만드는 주요 요인이다. 전통적인 머신러닝 기법은 이러한 고차원적 비선형 관계를 학습하는 데 한계를 보이며, 이를 보완하기 위해 제안된 다중 모델 기반의 앙상블 방식은 성능 향상을 달성하는 대신 파라미터 수와 추론 시간이 크게 증가하여 제한된 자원 환경에서는 적용이 어렵다는 한계가 있다.

본 연구에서는 이러한 한계를 해결하기 위해,

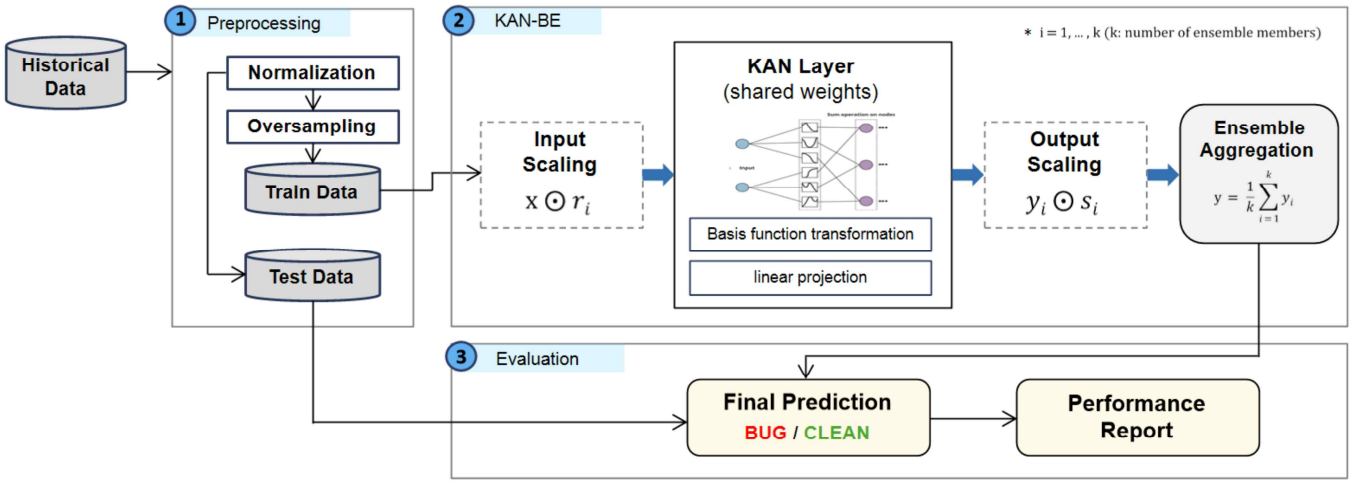


그림 1. 연구 방법

복잡한 비선형 관계 학습과 중요한 패턴 추출에 강점을 가진 KAN[1] 모델과 파라미터 효율성이 높은 BatchEnsemble[2] 기법을 결합한 KAN-BE 모델을 제안한다. KAN 모델을 BatchEnsemble 방식으로 구성함으로써, 동일한 Layer를 공유하는 여러 KAN 기반의 멤버가 결함 예측을 수행하도록 하여 자원 소모를 최소화하면서도 단일 모델 대비 향상된 예측 성능을 달성하고자 한다. 실험 결과, KAN-BE는 Balance 지표 기준 비교 모델 및 앙상블 기법 대비 가장 우수한 성능을 보였으며, 일반 KAN 모델과 유사한 수준의 자원 사용으로 제한된 소프트웨어 품질 보증 자원을 효율적으로 활용할 수 있음을 확인하였다.

또한, 기존 SDP 연구에서는 모델이 산출한 단일 예측 확률 값을 기준으로 결함 여부를 판단하는 경우가 많아, 예측 결과에 내재된 불확실성이나 판단 신뢰도에 대한 고려가 충분히 이루어지지 않았다. 본 연구에서는 KAN-BE의 앙상블 구조에서 도출되는 불확실성 정보를 활용하여 전역적인 결함 우선순위화와 결정 경계 영역에서의 결함 식별 가능성을 단계적으로 분석한다. 이를 통해 불확실성 기반 신뢰도 높은 결함 분석이 가능함을 보이며, 실제 소프트웨어 품질 보증 환경에서의 의사결정 효율성을 향상시키는 데 기여할 것으로 기대된다.

2. 관련 연구

SDP는 소프트웨어 품질 향상과 유지보수 비용 절감을 위해 다양한 접근법이 제안되어 왔다. 기존 연구에서는 예측 성능 개선을 위해 신경망 모델 또는 앙상블 학습 기법을 활용하였으나, 모델의 복잡도 증가에 따른 연산 자원 소모와 판단 신뢰도 측면에서 여전히 한계를 보이고 있다[3].

2.1 신경망 및 앙상블 기반 SDP

SDP 분야에는 Bagging 기반의 앙상블 기법인 Random Forest와 Boosting 기반의 앙상블 학습 기법인 XGBoost, AdaBoost 등이 널리 사용되어 왔으나[4], 이러한 전통적 앙상블 기법은 소프트웨어 메트릭 간의 복잡한 비선형 관계와 상호작용을 충분히 학습하는 데 한계를 보였다. 이에 따라 높은 표현력을 확보하기 위한 신경망 기반 모델이 제안되었다[5].

이후 단일 모델의 일반화 성능 한계와 예측 안정성을 개선하기 위해, 다중 모델을 결합하는 앙상블 접근법이 활발히 연구되었다. HABTEMARIAM et al.[6]은 GRU, Dense NN 등 여러 딥러닝 및 머신러닝 모델을 앙상블하여 단일 모델보다 우수한 성능을 보였으며, Zhang et al.[7]은 CNN과 BiLSTM 구조에 추상 구문 트리(AST) 및 클래스 의존성 네트워크(CDN) 정보를 결합한 앙상블을 통해 예측력과 일반화 성능을 향상시켰다.

그러나 이러한 다중 모델 기반 앙상블 방식은 모델 수 증가에 따라 파라미터 수와 추론 시간이 비례적으로 증가하여, 요구되는 메모리와 연산 자원이 커지는 한계를 가진다. 이에 따라 기존 앙상블의 성능 이점을 유지하면서도 파라미터 증가를 최소화할 수 있는 효율적인 SDP 접근법에 대한 연구가 필요하다. BatchEnsemble은 이러한 접근 중 하나로, 기본 네트워크의 가중치를 공유하면서 각 앙상블 멤버에 대해 저차원의 스케일 파라미터를 적용함으로써, 파라미터 수 증가를 최소화하면서도 앙상블 효과를 유지하는 앙상블 기법이다.

2.2 불확실성 기반 SDP

최근 SDP 연구에서는 평균 예측 확률과 같은 단일 지표만으로는 모델의 판단 신뢰도를 충분히 반영하기 어렵다는 문제가 제기되고 있다. 특히 여러 모델 또는

여러 앙상블 멤버가 예측을 수행하는 경우, 예측 결과 간의 불일치는 모델이 특정 샘플에 대해 확신하지 못하고 있음을 의미한다. 이를 보완하기 위해 예측 결과 분산이나 불확실성을 활용한 접근법이 제안되고 있다[9].

Lakshminarayanan et al.[10]은 Deep Ensemble 구조에서 여러 모델의 예측값 분산을 통해 불확실성을 추정하고, 모델 간 예측 분산이 클수록 모델이 확신하지 못하는 영역이며, 오류 가능성이 높아질 수 있음을 제시하였다. Li et al.[11]은 의사결정 이론에 기반한 three-way decision 프레임워크를 SDP 에 적용하여 경계 영역에 속한 모듈들이 상대적으로 높은 결함 위험을 내포하고 있음을 제시하였다.

이를 바탕으로 본 연구에서는 신경망 모델 기반 앙상블 방식을 활용하여 모델 간 예측 분산으로부터 발생하는 예측 불확실성을 정량적으로 추정하고, 이를 연속적인 지표로 활용하여 결함 가능성을 보다 정밀하게 분석하고자 한다.

3. 연구 방법

본 연구에서는 BatchEnsemble 기법을 SDP 에 적용하여 단일 KAN 모델과 여러 앙상블 멤버를 구성하고, 멤버 간 주요 파라미터를 공유하는 효율적인 구조를 구현하였다. 그림 1 은 연구 방법을, Algorithm 1 은 제안하는 모델의 학습 과정을 나타낸다. 모델의 일반화 성능을 평가하기 위해 10-Fold 교차 검증(Cross Validation)을 수행하였다.

먼저, 피처 스케일링을 위해 Min-Max 정규화 기법을 적용하여 데이터의 범위를 [0, 1]로 축소한다(2~3 행). 이후 합성 소수 샘플링(SMOTE)을 통해 학습 데이터를 1:1 비율로 오버샘플링 한 뒤 모델을 학습시킨다(4 행). 전처리 후, 단일 KAN 모델 내에 여러 앙상블 멤버를 구현한다(5~6 행). 각 멤버가 입력 특성을 다르게 반영할 수 있도록, 입력 데이터에 멤버 고유의 scaling vector r_i 를 element-wise 로 곱한다(7~8 행). 각 앙상블 멤버의 입력은 공유된 KAN 레이어를 통해 각 차원에 다항식 기반 기저 함수를 적용한 후, 선형 결합을 통해 비선형 특징 표현을 학습한다(9~13). 이를 통해, 주요 가중치를 공유하면서도 다양한 예측 표현을 학습할 수 있다. KAN Layer 를 통해 각 멤버의 예측값을 출력하고(14 행), 해당 예측값에 멤버별 scaling vector s_i 를 적용하여 각 멤버의 예측 특성을 조정한다(15~18 행). 이후 수식 (1)과 같이 모든 멤버의 예측값을 평균하여 최종 예측을 수행한다(19~20 행). 마지막으로 수식 (2)의 이진 교차 엔트로피 손실 함수를 통해 최종 예측값과 실제 값 간의 오차를 계산하고, 역전파를 통해 모델의 파라미터를 업데이트한다(21~23 행).

$$\hat{y} = \frac{1}{k}\sum_{i=1}^{k}(f(x \odot r_i) \odot s_i) \quad (1)$$

$$L = -\frac{1}{N}\sum_{n=1}^{N}[y_n \log p_n + (1-y_n)\log(1-p_n)] \quad (2)$$

Algorithm 1. KAN-BatchEnsemble

```
Input: Original Data X
Output: Data Y predicted for fault
1:/* Preprocess                                          */
2: X_train_scaled ← MinMax.fit(X_train)
3: X_test_scaled ← MinMax.transform(X_test)
4: X_train_resampled ← SMOTE(X_train_scaled)

5:/* Input Scaling                                       */
6: for each ensemble member i = 1 to k:
7:     Generate a random scaling vector r_i
8:     Ensemble_Input_i ← X_train_resampled × r_i

9:/* Shared KAN Layer                                    */
10: for each Ensemble_Input_i:
11:    Pass Ensemble_Input_i through the shared KAN Layer
12:        - Apply basis function transformation
13:        - Apply linear projection
14:    Get output y_i

15:/* Output Scaling                                     */
16: for each output y_i:
17:    Generate scaling factor s_i
18:    Ensemble_Output_i ← y_i × s_i

19:/* Aggregation                                        */
20: Final_Prediction = (1/k)  _{i=1}^{k} Ensemble_Output_i

21: Compute Loss(Final_Prediction, true_labels)
22: Backpropagate and update model parameters
23:/* Predict defects                                   */
```

4. 실험 설정

4.1 연구 질문

RQ1: KAN-BE 모델은 다 모델 대비 결함 예측 성능이 우수한가?

RQ2: KAN-BE 모델은 타 앙상블 기법 대비 효율성 측면에서 우수한가?

RQ3: 불확실성 지표는 결함 유무 판단에 유의미한 역할을 하는가?
RQ4: 결정 경계 영역에서 불확실성 지표가 결함 모듈을 얼마나 효과적으로 식별하는가?

본 연구에서는 KAN-BE 모델 기반 SDP 의 효용성을 평가하기 위해, RQ1 에서는 제안 모델의 성능을 타 기법들과 비교하고 RQ2 에서는 기존 앙상블 기법과 성능 대비 효율성을 비교한다.

RQ3 와 RQ4 를 통해서는 KAN-BE 의 불확실성 지표(σ)가 결함 예측에 미치는 영향을 단계적으로 평가한다. RQ3 에서는 모든 테스트 샘플을 대상으로 불확실성 기반의 우선순위 선정이 갖는 효율성을 검토하고, RQ4 에서는 결정 경계 구간에서 불확실성을 이중 판단의 지표로 활용하는 구조의 유효성을 탐구한다.

4.2 데이터

실험을 위해 오픈 데이터셋(AEEEM, ReLink)과 AUDI 자동차 결함 데이터셋을 사용한다. 표 1 은 데이터셋별 세부 정보를 나타낸다.

표 1. 실험 데이터셋

Dataset	Project	#of instances		#of metric	Granularity
		all	buggy		
AEEEM	EQ	324	129(39.81%)	61	class
	JDT	997	206(20.66%)	61	class
	LC	691	64(9.26%)	61	class
AUDI	A	1908	85(4.5%)	12	file
	K	2515	375(14.01%)	12	file
	L	2891	76(2.63%)	12	file
ReLink	apache	194	98(50.52%)	26	file
	safe	56	22(39.29%)	26	file
	zxing	399	118(29.57%)	26	file

4.3 성능 평가 지표

표 2. 혼동 행렬

		Predicted class	
		Defective	Clean
Actual class	Defective	TP(True Positive)	FN(False Negative)
	Clean	FP(False Positive)	TN(True Negative)

성능 평가를 위해 표 2 혼동 행렬을 기반으로, 실제 결함 모듈 중 올바르게 결함으로 예측된 비율을 나타내는 PD 와 정상 모듈 중 결함으로 잘못 예측된 비율을 의미하는 PF 를 사용한다. 또한, PD 와 PF 의 균형을 측정하기 위한 Balance($=1-\sqrt{\frac{(0-PF)^2+(1-PD)^2}{2}}$) 지표와 코드 검사 노력도 감소 효과를 분석하기 위한 FIR(File Inspection Reduction($=\frac{PD-FI}{PD}$)) 지표를 사용한다. FIR 계산에 사용되는 FI(False Identification)는 모델이 결함이라고 예측한 전체 데이터의 비율이다. 효율성 평가는 모델이 테스트 데이터셋 전체를 처리하는 데 소요된 전체 추론 시간과 평균 파라미터 수를 기준으로 비교를 수행하였다.

$$PD = TP/(TP+FN) \quad (1)$$

$$PF = FP/(FP+TN) \quad (2)$$

$$FI = (TP+FP)/(TP+TN+FP+FN) \quad (3)$$

4.4 하이퍼파라미터 설정

표 3. 파라미터 기본값

Parameter	Default
d	128
n	3
degree	3
k	5
bs	128
epoch	50
lr	1e-4
Random Seed	42
Cross-Validation(K)	10-Fold
threshold	0.5

표 3 은 RQ1~RQ4 실험에 사용된 하이퍼파라미터의 설정 값을 정리한 것이다. 위 값들은 성능-자원의 균형과 학습 안정성을 고려하여 선정하였다. 모델의 구조 및 학습과 관련된 주요 파라미터로는 블록 차원 수(d), 블록 수(n), 스플라인 차수(degree), 앙상블 멤버 수(k), 배치 크기(bs), 학습 횟수(epoch), 학습률(lr), 랜덤 시드(Random Seed), 교차 검증 분할 수(K-Fold), 그리고 분류 임계값(threshold)이 포함된다. 이 중 d, n, degree 는 모델의 표현력과 함수 근사 능력에 영향을 미치며, k 는 앙상블 예측의 안정성과 자원 소모 간의 균형을 결정하는 요소이다. 또한 bs 와 lr 은 학습 효율과 수렴 특성에 영향을 미친다.

본 실험에서는 결과의 재현성을 보장하기 위해 랜덤 시드를 42 로 고정하였으며, 모델의 일반화 성능을 평가하기 위해 10-Fold 교차 검증을 적용하였다. 추가적으로, 모델의 예측 결과는 이진 분류를 위해 확률 임계값을 0.5 로 설정하였으며, 해당 값을 기준으로 결함 여부를 판단하였다.

4.5 실험 환경

본 실험은 Intel(R) Xeon(R) w3-2423 CPU, 16 GB RAM, 그리고 4 GB의 VRAM을 갖춘 NVIDIA T400 GPU 환경에서 수행되었으며, 모델 학습 및 추론 과정은 CUDA 12.8 환경에서 진행되었다.

5. 실험 설정

5.1 RQ1: KAN-BE 모델은 타 모델 대비 결함 예측 성능이 우수한가?

표 4. KAN-BE와 타 모델 성능 비교

Models	Metrics			
	PD	PF	Balance	FIR
KAN-BE	**0.7850**	0.1626	**0.7900**	0.5611
KAN	0.6742	0.1701	0.7190	0.5014
MLP	0.7445	0.2468	0.7190	0.4599
RF	0.7531	0.1372	0.7538	**0.7188**
XGB	0.7142	**0.1281**	0.7558	0.7101

표 4는 4가지 성능 지표를 기준으로 KAN-BE와 비교 모델들의 성능을 나타낸다. 비교 대상에는 KAN-BE의 기반 모델인 KAN 모델과 SDP 분야에서 널리 사용되는 MLP, RandomForest(RF), XGBoost(XGB) 모델이 포함된다.

실험 결과, KAN-BE는 PD와 Balance 지표에서 가장 우수한 성능을 보여, 결함 탐지율과 오탐 간의 균형 측면에서 의미 있는 성능 향상을 달성하였다. 반면, PF 지표에서는 XGB가, FIR에서는 RF가 가장 우수한 값을 보였다.

표 5. KAN-BE와 타 모델 Effect size 비교

KAN-BE	Metrics			
vs.	PD	PF	Balance	FIR
KAN	**0.7236(M)**	**-0.1039(N)**	**0.6208(M)**	**0.3713(S)**
MLP	**0.2851(S)**	**-0.8010(L)**	**0.7058(M)**	**0.6553(M)**
RF	**0.1809(N)**	0.2515(S)	**0.2345(S)**	-0.8750(L)
XGB	**0.3929(S)**	0.3506(S)	**0.2500(S)**	-0.8736(L)

표 5는 KAN-BE와 비교 모델 간 성능 차이의 실질적인 유의성을 분석하기 위해 Cohen's d[12]에 기반한 효과 크기를 제시한다.

분석 결과, KAN-BE는 KAN 대비 PD와 Balance 지표에서 각각 Medium 수준의 효과 크기를 보여, 결함 탐지 성능과 예측 균형 측면에서 유의미한 향상을 제공함을 확인하였다. MLP와의 비교에서는 PD를 제외한 모든 지표에서 Medium 수준 이상의 효과 크기를 보여, KAN-BE가 전반적인 지표에서 예측 안정성과 효율성이 개선되었음을 확인하였다. 반면, RF 및 XGB와의 비교에서 FIR 지표가 Large 수준의 음수 효과 크기를 보여, 검사 효율 측면에서는 전통적인 머신러닝 모델들이 우위에 있음을 나타낸다. 그러나 주요 결함 탐지 성능을 나타내는 PD와 Balance 지표에서는 KAN-BE가 유의미한 효과 크기를 나타내어 결함 탐지 측면에서 강점을 지님을 확인하였다.

5.2 RQ2: KAN-BE 모델은 타 앙상블 기법 대비 효율성 측면에서 우수한가?

모델의 효율성을 분석하기 위해 표 6에서는 KAN 모델을 baseline으로 설정하였고, KAN 대비 단순한 구조의 MLP 모델 계열을 참고 비교군으로 포함하였다. KAN과 MLP에 각각 BatchEnsemble과 Deep Ensemble 기법을 적용하여 추론 시간(Inf. Time)과 파라미터 수(Params)를 비교하였다. 본 비교는 앙상블 멤버 수(ensemble size = 5)를 공정성 기준으로 설정하였으며, 이후 동일 앙상블 규모 조건에서 성능 및 자원 효율성 차이를 분석하였다. 추가적으로, 본 절에서는 동일한 표기법을 유지하기 위해 KAN에 Deep Ensemble을 적용한 비교 모델을 KAN-DE, MLP에 BatchEnsemble 및 Deep Ensemble을 적용한 비교 모델을 각각 MLP-BE와 MLP-DE로 약칭한다.

표 6. KAN-BE와 타 앙상블 기법 성능 비교

Models	Metrics					
	PD	PF	Balance	FIR	Inf. Time (ms)	Params
KAN	0.6742	0.1701	0.7190	0.5014	3.13	38,661
KAN-BE	0.7850	0.1626	**0.7900**	0.5611	3.31	42,671
KAN-DE	0.7302	**0.1600**	0.7393	**0.5643**	7.75	193,305
MLP-BE	**0.8436**	0.3418	0.6983	0.4600	**2.57**	**14,763**
MLP-DE	0.7633	0.2577	0.7194	0.4737	3.94	63,365

실험 결과, KAN-BE는 baseline 대비 4가지 성능 지표에서 모두 성능을 향상시키면서도, 추론 시간은 3.13 ms에서 3.47 ms로, 파라미터 수는 38,661개에서 42,671개로 모두 약 1.1배 증가하여 거의 동일한 자원 수준에서 효율적인 성능 향상을 달성함을 보였다. 반면, KAN-DE는 추론 시간이 3.13 ms에서 7.75 ms로 약 2.5배, 파라미터 수가 38,661개에서 193,305로 약 5배 증가하여, 자원 소모가 크게 확대되는 경향을 보였다. 비록 일부 지표(PF, FIR)에서 KAN-DE가 KAN-BE보다 미세하게 우수한 결과를 보였으나, 이러한 성능 향상은 상당한 자원 소모가 동반되었다. 한편, MLP-BE는 가장 낮은 추론 시간과 파라미터 수를 보였으나, PD를 제외한 성능 지표에서는 KAN-BE에 비해 낮은 수준을

보였고, MLP-DE 역시 성능 및 효율성 측면에서 제한적이었다.

결과적으로, KAN-BE는 비교 기법 대비 더 적은 자원으로도 우수한 예측 성능을 달성하여, SDP 환경에서 자원 효율성과 성능을 동시에 만족시키는 균형 잡힌 접근법임을 시사한다.

표 7. KAN-BE와 타 앙상블 기법 Effect size 비교

KAN-BE vs.	Metrics			
	PD	PF	Balance	FIR
KAN-DE	**0.3433(S)**	0.0322(N)	**0.3803(S)**	-0.0200(N)
MLP-BE	-0.5323(M)	**-1.1030(L)**	**0.6537(M)**	**0.6627(M)**
MLP-DE	**0.1574(S)**	**-0.7769(M)**	**0.6142(M)**	**0.5692(M)**

표 7은 KAN-BE와 타 앙상블 기반 모델 간 성능 차이의 실질적인 유의성을 분석하기 위해 Cohen's d 값에 따른 효과 크기를 분석한 결과이다.

분석 결과, KAN-BE는 KAN-DE 대비 PD, Balance 지표에서 Small 수준의 효과 크기를 보여, 제안 기법이 탐지 성능, 예측 균형 측면에서 일관되게 우수한 성능을 보임을 확인하였다. 한편 MLP-BE와의 비교에서는 PF, Balance, FIR 지표에서 Medium 수준 이상의 효과 크기를 보였으며, 특히 PF 지표에서 Large 수준의 효과 크기가 관찰되어 KAN-BE가 단순 구조의 BatchEnsemble 대비 전반적인 예측 성능에서 의미 있는 개선을 제공함을 확인하였다. 또한, MLP-DE와의 비교에서는 모든 지표에서 Small과 Medium 수준의 효과 크기를 보여 Deep Ensemble 대비 성능 차이가 통계적으로 유의미함을 확인하였다.

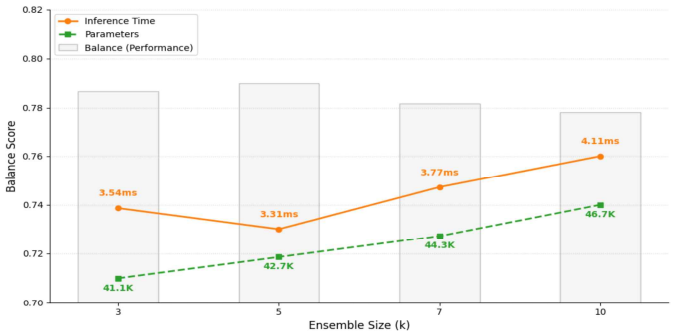


그림 2. k 값에 따른 성능-비용 그래프

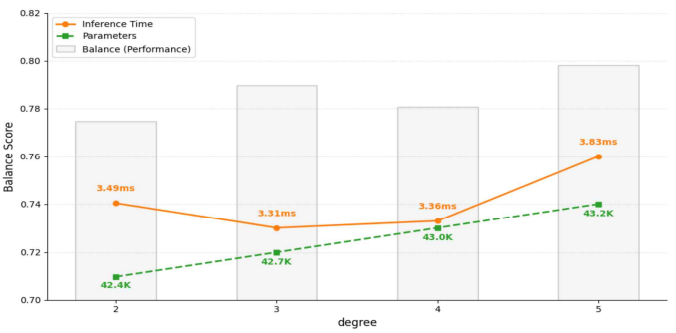


그림 3. degree 값에 따른 성능-비용 그래프

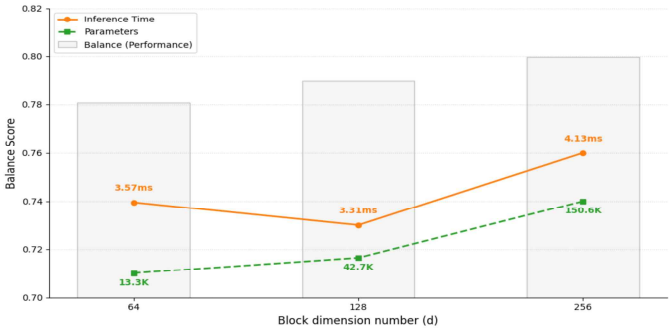


그림 4. d 값에 따른 성능-비용 그래프

그림 2-4는 모델 구조의 핵심 하이퍼파라미터인 k, degree, d 변화에 따른 성능 및 효율성 지표의 변동 추이를 나타낸 것이다. 모든 지표에서 공통적으로 모델 복잡도가 증가함에 따라 파라미터 수와 지연 시간이 상승하는 경향을 보였으나, 성능 개선 폭은 일정 수준 이상에서 줄어드는 것을 알 수 있었다. 특히, degree=5에서는 정교한 학습으로 성능이 향상되었으나 학습이 불안정했고, d=256에서는 약 15만 개의 파라미터가 사용되어 자원 효율성이 급격히 저하되었다. 따라서 비용 대비 성능 개선 효율이 가장 우수한 k=5, degree=3, d=128을 최적의 균형점으로 판단하였다.

5.3 RQ3: 불확실성 지표는 결함 유무 판단에 유의미한 역할을 하는가?

RQ3에서는 불확실성 지표 σ가 결함 식별에 유의미한 정보를 제공하는지를 분석한다. 본 연구에서 불확실성 지표는 이전 실험에서 설정한 5개의 앙상블 멤버 간 결함 확률의 표준편차를 기반으로 산출하였다. 불확실성 기반 우선순위화 효과를 정량적으로 평가하기 위해서는 Lift 지표를 사용한다. Lift는 특정 기준으로 선택된 모듈 집합에서의 결함 비율을 전체 결함 비율로 나눈 값으로, 무작위 선택 대비 결함 모듈이 얼마나 집중되어 있는지를 나타내며, 값이 1보다 클수록 해당 기준이 결함 탐지에 효과적임을 의미한다.

표 8. 상위 불확실성 비율별 Lift 값 비교

Datasct	Projcct	Lift by Top σ percentages		
		Top 5%	Top 10%	Top 20%
AEEEM	EQ	1.63	**1.75**	1.16
	JDT	**1.26**	1.11	**1.26**
	LC	0.93	1.08	**1.48**
AUDI	A	1.17	**1.29**	1.00
	K	**1.17**	1.06	0.93
	L	**2.62**	1.44	0.92
ReLink	apache	**1.39**	1.29	1.02
	safe	1.70	1.70	**2.12**
	zxing	1.18	**1.44**	1.23
Average		**1.45**	1.35	1.24

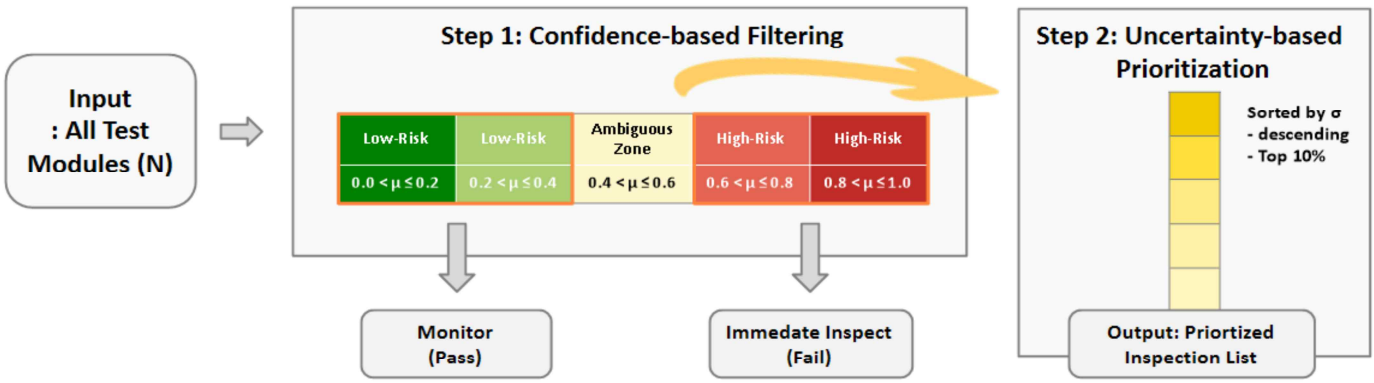


그림 5. 2단계 결함 예측 프레임워크

표 9. 예측 확률 구간별 모듈 개수

Dataset	Project	Modules of predictive probability interval					
		μ[0.0,0.2]	μ[0.2,0.4]	μ[0.4,0.6]	μ[0.6,0.8]	μ[0.8,1.0]	Ambiguous Ratio (%)
AEEEM	EQ	0	86	**206**	25	7	**63.6**
	JDT	1	189	**585**	210	12	**58.7**
	LC	17	178	**460**	32	4	**66.6**
AUDI	A	132	726	**909**	139	2	**47.6**
	K	110	605	**1183**	612	5	**47.0**
	L	214	1710	**870**	97	0	**30.1**
Relink	apache	0	0	**184**	10	0	**94.8**
	safe	0	1	**52**	3	0	**92.9**
	zxing	0	3	**386**	10	0	**96.7**

표 8은 상위 불확실성 비율에 따른 Lift 분석 결과를 나타낸다. 불확실성이 높은 모듈을 상위 5%, 10%, 20%로 선택할 경우 평균적으로 모든 비율에서 Lift 값이 1을 초과하였으며, 특히 상위 5%에서는 평균 Lift 1.45를 기록하였다. 이는 불확실성이 무작위 선택 대비 결함 모듈을 효과적으로 우선 식별할 수 있음을 의미하며, 전역적인 불확실성 기반 우선순위 지표로서의 유용성을 입증한다.

5.4 RQ4: 결정 경계 영역에서 불확실성 지표가 결함 모듈을 얼마나 효과적으로 식별하는가?

5.4.1 2단계 결함 예측 프레임워크

RQ4에서는 평균 예측 확률(μ)만으로 판단이 어려운 결정 경계 영역(0.4<μ<0.6)에서 불확실성이 결함 식별 정밀도를 얼마나 개선할 수 있는지를 분석한다. 이를 위해 본 연구는 평균 확률 기반의 1단계 필터링 이후, 모호한 결정 경계 영역에 대해 불확실성 기반의 2단계 우선순위화를 수행하는 절차를 적용하였다. 그림 5는 제안하는 2단계 결함 예측 프레임워크를 개념적으로 나타낸다.

1단계에서는 평균 예측 확률을 기반으로 모듈을 필터링한다. KAN-BE의 예측 확률이 0.4 이하이거나 0.6을 초과하는 구간은 신뢰 가능한 예측으로 간주하여 각각 모니터링 대상 또는 즉각적인 점검 대상으로 분류한다. 반면, 판단이 모호한 결정 경계 영역에 속한 모듈은 앙상블 멤버가 결과를 확신하지 못하는 구간이므로 2단계로 전달한다. 2단계에서는 모듈들을 불확실성 지표를 기준으로 내림차순 정렬하여, 상위 모듈을 우선 점검 대상으로 선정한다.

5.4.2 예측 확률 분포 분석

표 9는 데이터셋별 예측 확률의 구간 분포를 나타내며, 각 프로젝트에서 KAN-BE 모델이 예측한 결함률에 해당하는 모듈들이 서로 다른 확률 구간에 어떻게 분포하는지를 보여준다. 표의 우측 열에 결정 경계 영역에 포함된 모듈의 비율(Ambiguous Ratio)을 함께 제시함으로써, 평균 예측 확률만을 기준으로 판단할 경우 모호한 예측에 해당하는 모듈이 얼마나 존재하는지를 분석하였다.

분석 결과, 대부분의 프로젝트에서 전체 모듈의 상당 부분이 결정 경계 영역에 분포하는 것으로 나타났으며, 특히 Relink와 같은 소규모 프로젝트에서는 해당 비율이 90%를 초과하였다. 이는 단일 확률 값에 기반한 분류만으로는 상당수의 모듈에 대해 명확한 판단을

내리기 어렵다는 점을 시사하며, 결정 경계 영역에 대한 추가적인 분석이 필요함을 보여준다. 이러한 한계를 보완하기 위해, 본 연구에서는 앙상블 멤버의 예측 분산을 불확실성 지표로 활용하여 기존의 결함 예측 방식을 보완한다.

5.4.3 불확실성 기반 결함 집중도 분석

표 10 은 결정 경계 영역에 속한 모듈을 대상으로, 불확실성 지표를 활용한 우선순위화가 결함 집중도에 미치는 영향을 분석한 결과를 나타낸다. 표에는 결정 경계 영역에 포함된 모듈 수, 전체 모듈 대비 결함 비율(Base Defect Rate), 불확실성 기준 상위 10% 모듈의 결함 비율(Top 10% σ Defect Rate), 그리고 이들의 비율을 비교한 Lift 값을 함께 제시하였다.

표 10. 결정 경계 영역의 불확실성 기반 집중도 분석

Dataset	Project	Ambiguous Zone Analysis			
		μ[0.4,0.6] Modules	Base Defect Rate(A)	Top 10% σ Defect Rate(B)	Lift (B/A)
AEEEM	EQ	74	0.4272	0.6190	**1.45**
	JDT	585	0.1915	0.3051	**1.59**
	LC	460	0.0913	0.0870	0.95
AUDI	A	909	0.0528	0.0549	**1.04**
	K	1183	0.1209	0.1933	**1.60**
	L	870	0.0425	0.0690	**1.62**
ReLink	apache	184	0.4837	0.5789	**1.20**
	safe	52	0.3654	0.8333	**2.29**
	zxing	386	0.2953	0.3846	**1.30**

분석 결과, 대부분의 프로젝트에서 불확실성 기준 상위 10% 모듈은 전체 평균 대비 높은 결함 비율을 보였으며, 이에 따라 Lift 값이 1 을 초과하는 경향을 나타냈다. 특히 safe 프로젝트에서는 Lift 가 2 이상으로 나타나, 동일한 평균 확률 구간 내에서도 불확실성이 큰 모듈일수록 실제 결함 가능성이 높음을 확인할 수 있었다. 이는 평균 예측 확률만으로는 구분이 어려운 결정 경계 영역에서 불확실성 지표가 결함 식별 정밀도를 효과적으로 향상시키는 보조 지표로 작용할 수 있음을 의미한다.

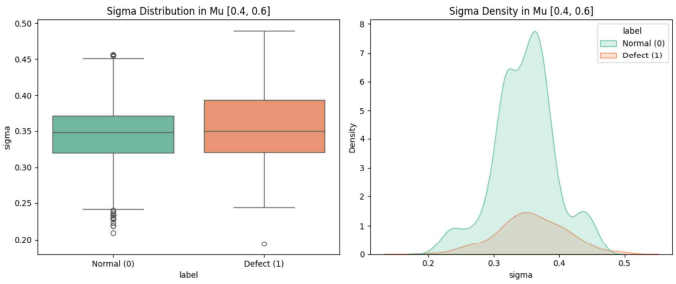


그림 6. 불확실성 분포 - AEEEM/JDT

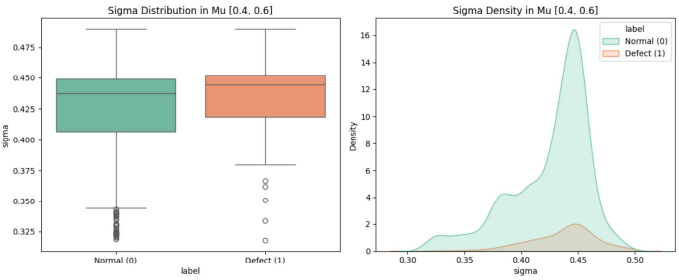


그림 7. 불확실성 분포 - AUDI/K

그림 6 과 그림 7 은 결정 경계 영역 중 모듈 수가 상대적으로 많은 프로젝트를 대상으로, 정상 및 결함 모듈 간 불확실성(Sigma) 분포 특성을 시각화한 것이다. 각 그림의 좌측은 두 클래스 간 불확실성 값의 분포를 boxplot 으로 나타낸 것이며, 우측은 동일 구간에서의 불확실성 밀도(Density) 분포를 나타낸다.

결함 모듈은 정상 모듈에 비해 상대적으로 높은 불확실성 값을 갖는 경향을 보이며, 특히 불확실성 값의 상위 영역으로 갈수록 결함 모듈이 포함될 가능성이 높아진다. 이는 전체 분포 상에서 두 클래스의 불확실성 값이 일부 겹쳐지더라도, 불확실성 기반 정렬을 통해 상위 영역을 우선적으로 검사할 경우 결함 모듈이 상대적으로 집중될 수 있음을 시사한다.

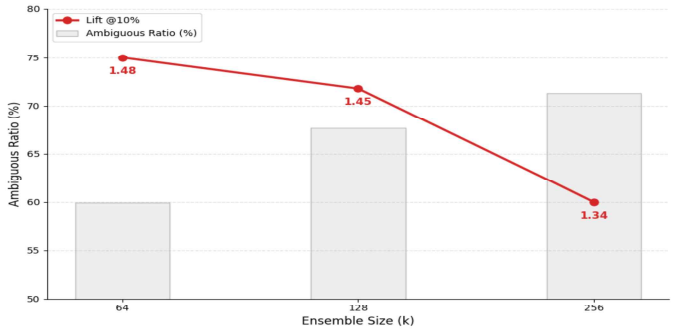


그림 8. k 값에 따른 결정 경계 영역-Lift 그래프

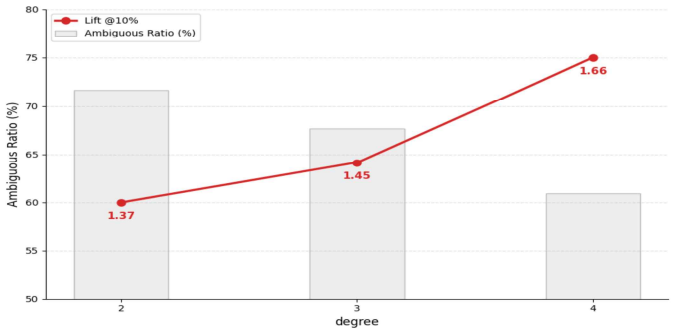


그림 9. degree 값에 따른 결정 경계 영역-Lift 그래프

그림 8과 그림 9는 하이퍼파라미터 k와 degree 변화에 따른 결정 경계 영역과 불확실성 기준 상위 10% 모듈에 대한 탐지 효율(Lift) 간의 관계를 나타낸다.

분석 결과, k 값이 증가함에 따라 결정 경계 영역은

증가하였고, Lift 값은 소폭 감소하는 경향을 보였다. 이는 예측을 수행하는 멤버가 늘어남에 따라 멤버 간 의견 불일치로 불확실한 판단이 많아진 것인데, 여전히 Lift 값은 1이상으로 불확실성 기반의 검사가 무작위 검사보다 효율적임을 알 수 있다. 또한, degree 값이 증가할수록 결정 경계 영역의 비율은 줄어들고 Lift 값은 향상되는 경향을 보였다. 이는 스플라인 차수가 높아질수록 모델이 정상과 결함 사이의 미세한 차이를 정교하게 학습하여, 애매했던 정상 모듈을 정밀하게 식별하기 때문이다. 특히, degree=4에서는 Lift 값이 1.66으로 불확실성 기반 우선순위가 탐지 효율을 극대화할 수 있음을 보였다.

6. 위협 요소

6.1 내적 타당성(Internal validity)

본 연구에서는 BatchEnsemble 기반 앙상블 모델의 예측 분산을 불확실성 지표로 활용하였다. 불확실성 지표에 대해 하이퍼파라미터 변화에 따른 민감도를 확인하였지만, 그럼에도 불구하고 예측 분산은 모델 구조, 초기화 방식, 특정 환경 등에 따라 민감하게 영향을 받을 수 있다. 이러한 설정 차이에 따라 불확실성 추정 값이 달라질 가능성이 있으며, 이는 결과적으로 불확실성 지표와 결함 위험 간의 관계에 대한 해석의 일관성에 영향을 미칠 수 있다.

6.2 외적 타당성(External validity)

본 연구에서는 정적 소프트웨어 메트릭 기반의 표준 벤치마크 데이터셋 3가지를 활용하여 제안 기법을 검증하였다. 이러한 데이터셋은 기존 SDP 연구와의 비교 가능성과 재현성을 확보하는 데 유리하지만, 최근 소프트웨어 개발 환경에서 다루어지는 동적 특성을 반영하지 못하는 한계가 있다. 따라서 본 연구 결과를 다른 프로젝트나 실제 산업 환경 전반으로 일반화하는 데에는 일정한 한계가 있을 수 있다.

7. 결론 및 향후 과제

본 연구에서는 SDP의 성능 향상 및 자원 절감을 위해 KAN 모델에 BatchEnsemble 기법을 결합한 KAN-BE 모델을 제안하고, KAN-BE 구조에서 발생하는 예측 불확실성을 활용하여 결정 경계 영역에서 결함 위험이 높은 모듈을 효과적으로 식별하는 프레임워크를 제시하였다. 실험 결과, KAN-BE는 기존 Deep Ensemble 대비 파라미터 수와 추론 시간을 효율적으로 사용하면서도 앙상블의 성능 이점을 유지하였다. 또한, 불확실성 기반 접근법이 기존 단일 확률 값 기반 예측에 비해 결함 위험이 높은 모듈을 우선적으로 식별하는 능력을 향상시키는 데 기여함을 확인하였다.

향후 연구에서는 정적 메트릭 기반 표준 데이터셋 뿐만아니라 최신 변경 이력 기반 결함 데이터셋을 대상으로 제안 모델을 적용하여, 보다 다양한 개발 환경에서의 일반화 가능성을 검증할 계획이다. 또한, 본 연구에서는 앙상블 멤버 간 예측 분산에 따른 불확실성을 지표로 사용하였는데, 향후 연구에서는 데이터 자체의 노이즈, 라벨 불확실성 등 데이터 불확실성을 함께 고려하는 방향으로 확장함으로써 보다 포괄적인 불확실성 분석이 가능하도록 할 계획이다. 마지막으로, 본 연구에서 제안한 2단계 결함 예측 프레임워크에 대해 데이터 입력부터 결과 출력까지의 과정을 자동화하고, 제안 프레임워크에 대한 실증적 분석을 추가로 수행할 예정이다.

감사의 글

이 논문은 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원-대학 ICT 연구센터(ITRC)의 지원 (IITP-2026-RS-2020-II201795, 50%) 및 정보통신기획평가원-지역지능화혁신인재양성사업의 지원을 받아 수행된 연구임(IITP-2026-RS-2024-00439292, 50%)

참고문헌

[1] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M.Soljacic, T. Y. Hou, and M. Tegmark, “KAN: Kolmogorov-Arnold Networks,” in Proc. of International Conference on Learning Representations (ICLR), 2025.

[2] Y. Wen, D. Tran, and J. Ba, “BatchEnsemble: An Alternative Approach to Efficient Ensemble and Lifelong Learning,” in Proc. of the International Conference on Learning Representations (ICLR), 2020.

[3] G. Giray et al., “On the use of deep learning in software defect prediction,” Journal of Systems and Software, vol. 195, p. 111537, 2023.

[4] S. M. H. Kabir, M. T. Rahman, and A. H. Mridul, “Software Defect Prediction Using Traditional Machine Learning and Ensemble Learning Algorithms,” Smart Wearable Technology, vol. 1, p. A9, 2025.

[5] M. M. T. Thwin and T.-S. Quah, “Application of neural networks for software quality prediction using object-oriented metrics,” Journal of Systems and Software, vol. 76, no. 2, pp. 147-156, 2005.

[6] G. M. Habtemariam, S. K. Mohapatra, and H. W. Seid, “Optimized Ensemble Learning for Software Defect Prediction with Hyperparameter Tuning,” Journal of Theoretical and Applied Information Technology, vol. 102, no. 14, pp. 5628–5639, 2024.
[7] S. Zhang, S. Jiang, and Y. Yan, “A Hierarchical Feature Ensemble Deep Learning Approach for Software Defect Prediction,” International Journal of Software Engineering and Knowledge Engineering, vol. 33, no. 4, pp. 543–573, 2023.
[8] S. Fort, H. Hu, and B. Lakshminarayanan, “Deep Ensembles: A Loss Landscape Perspective,” arXiv preprint, arXiv:1912.02757, 2019.
[9] F. O. Catak, T. Yue, and S. Ali, “Uncertainty-aware prediction validator in deep learning models for cyber-physical system data,” ACM Transactions on Software Engineering and Methodology, vol. 31, no. 4, pp. 1–31, 2022.
[10] B. Lakshminarayanan, A. Pritzel, and C. Blundell, “Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles,” Advances in Neural Information Processing Systems (NeurIPS), vol. 30, pp. 6402–6413, 2017.
[11] W. Li, Z. Huang, and Q. Li, “Three-Way Decisions Based Software Defect Prediction,” Knowledge-Based Systems, vol. 91, pp. 263–274, 2016.
[12] L. Gong, S. Jiang, and L. Jiang, “Conditional Domain Adversarial Adaptation for Heterogeneous Defect Prediction,” IEEE Access, vol. 8, pp. 150738–150749, 2020.
[13] Y. Gorishniy, et al., “TabM: Advancing Tabular Deep Learning with Parameter-Efficient Ensembling,” in Proceedings of the International Conference on Learning Representations (ICLR), 2025.

통합 그래프 신경망을 통한 계층적 멀티모달 코드 표현 학습

주나이드 칸 카카르, 파이살 모하마드, 류덕산

(전북대학교 소프트웨어공학과)

Junaidk@jbnu.ac.kr, mfaisal@jbnu.ac.kr, duksan.ryu@jbnu.ac.kr

Hierarchical Multi-Modal Code Representation Learning via Unified Graph Neural Networks

Junaid Khan Kakar, Faisal Mohammad, Dimitri Bakale Duksan Ryu

(Department of Software Engineering, Jeonbuk National University)

요 약

현대 소프트웨어 공학의 발전은 복잡한 프로그램의 의미론을 포괄적으로 포착할 수 있는 정교한 자동 코드 분석 시스템을 요구한다. 그러나 기존의 신경망 기반 접근법들은 주로 단일 구조적 관점 또는 텍스트 특징에 의존함으로써 소스 코드의 다면적 본질을 충분히 포착하지 못하는 한계를 보인다. 특히, 이러한 방법론들은 추상 구문 트리(AST), 제어 흐름 그래프(CFG), 데이터 흐름 그래프(DFG) 간의 유기적 상호작용을 간과하여 프로그램 이해가 단편적으로 이루어지는 경향이 있다. 본 연구는 이러한 구조적 단절을 해소하고자 AST, CFG, DFG를 포괄적인 코드 속성 그래프(CPG)로 통합하고 이를 그래프 신경망을 통해 처리하는 새로운 계층적 다중 모달 프레임워크를 제안한다. 제안된 방법론은 적응형 어텐션 기반 그래프 융합 전략을 채택하여 지역적 구문 패턴과 전역적 의미 의존성을 동시에 보존하도록 설계되었다. OBG-Code2 및 CodeSearchNet 데이터셋을 활용한 실험 평가 결과, 제안된 통합 모델은 단일 관점 기반 베이스라인 대비 코드 검색 과제에서 24.8%의 상대적 성능 향상을 달성하여 우수한 코드 이해 능력을 입증하였다. 이러한 연구 결과는 다중 모달 구조 정보의 통합이 차세대 구조 인식형 신경 코드 지능 시스템 발전을 위한 필수 요소임을 시사한다.

Abstract

The advancement of modern software engineering necessitates automated systems capable of deeply comprehending complex program semantics. However, existing neural approaches often suffer from a critical limitation: reliance on isolated structural views or textual features, which fail to capture the multifaceted nature of source code. By neglecting the interplay between syntactic structures, controlling flows, and data dependencies, current models frequently result in a fragmented understanding of program logic. To overcome this structural disconnect, this study proposes a hierarchical multi-modal framework that integrates Abstract Syntax Tree (ASTs), Control Flow Graph (CFGs), and Data Flow Graph (DFGs) into a comprehensive Code Property Graph (CPG) utilizing a unified graph neural network. Our method employs a novel graph fusion strategy with adaptive attention mechanisms to preserve both local syntactic patterns and global semantic dependencies. Empirical evaluations on the OBG-Code2 and CodeSearchNet datasets demonstrate that this unified approach significantly outperforms single-view baselines, achieving a 24.8% relative improvement in code search tasks. These results confirm that synthesizing multi-modal structural information is essential for advancing structure-aware neural code intelligence.

1. Introduction

Modern software engineering increasingly relies on automated code understanding systems to support tasks such as code search, summarization, and vulnerability detection. While large language models (LLMs) like CodeBERT and CodeT5 have advanced natural language-code alignment, their text-centric architectures often overlook the rich structural semantics inherent in source code [3]. Graph-based approaches like GraphCodeBERT partially address this by incorporating data flow graphs (DFGs), but they remain limited to a single structural view, ignoring critical syntactic (AST) and control-flow (CFG) relationships. This gap hinders models from fully capturing program behavior, particularly for complex logic or cross-language analysis [1][2].

Recent work highlights the potential of multi-view code representations. Demonstrate through GALLa that aligning abstract syntax trees (ASTs) with LLMs improves structural reasoning, while Guo et al. show that combining multiple code views in CodeSAM enhances model attention patterns. However, existing methods face two critical limitations: 1) Architectural incompatibility with decoder-only LLMs, as seen in GraphCodeBERT's modified transformer layers, and 2) Narrow structural scope most models prioritize either DFGs or ASTs, neglecting the complementary value of control-flow semantics. Nam et al. further emphasizes that current LLMs struggle with control-flow analysis, particularly in identifying data dependencies across execution paths [4][6].

To extract useful information from graph data, early graph analysis methods used graph embedding methods to project the graph into a low-dimensional vector space to create new features for dimensionality reduction, while preserving the essential characteristics of the original data. This makes the original graph more tractable. Subsequently, traditional vector-based machine learning methods can easily complete graph analysis tasks. Although the low-dimensional vector representations obtained in this way make graph learning models or algorithms easier to extract useful information, such methods usually suffer high computation and space overhead. To alleviate this issue, graph neural network (GNN) algorithms [16] have attracted recent attention to automatically capture high-level vertex representations and graph topology information from the given original low-level graph-structured data [18].

This paper proposes a novel framework that integrates AST, CFG, and DFG into a unified CPG while maintaining compatibility with pre-trained LLMs. Building on GALLa's graph alignment strategy , our approach introduces three key innovations: 1) A hierarchical graph encoder that processes multi-relational edges (syntax, control, data) through type-specific attention mechanisms, 2) A parameter-efficient adapter layer that projects unified graph embeddings into the LLM's latent space without architectural modifications, and 3) A multi-stage training protocol that jointly optimizes structural and textual alignment using contrastive learning objectives. Our main contributions are summarized as follows:

- We extend GraphCodeBERT by integrating AST and DFG in addition to the existing CFG, resulting in a unified and enriched code representation CPGs that better captures both syntactic and semantic information.
- In contrast to prior approaches that often neglect syntactic structure, our method explicitly models the complete structural and semantic context of source code, thereby improving the model's capacity for understanding code.
- Extensive ablation studies and benchmark evaluations across multiple tasks, including code summarization, code search, and code generation—demonstrate that incorporating all three graph modalities leads to consistent and significant performance gains.

The organization of this work is as follows. In section 2, we present the background and related work on graph representation and their unified models. In section 3, we present the methodology. Then, in section 4, we talk about experimental setup and results. In section 6, threats or validity to be verified. In section 7, we conclude our work.

2. Related Work

Research on source code understanding seeks to combine textual and structural semantics. Existing methods broadly fall into text-based models, encoder-decoder models with syntactic structure, and graph-based or multi-view approaches. While effective for tasks like code search, these sequential models fail to capture the structural relationships that define program behavior.

(i) Text-Based code Models

Text-based models treat code as token sequences and apply Transformer architecture. CodeBERT and CodeT5 achieve strong performance on tasks such as code search and summarization but lack explicit

structural reasoning [4,5]. TransCoder-IR improves semantic grounding using LLVM intermediate representation during training; however, IR remains text-based and does not explicitly capture control or data-flow structures [7]. While effective for tasks like code search, these sequential models fail to capture the structural relationships that define program behavior.

(ii) Encoder-Decoder models

Encoder-decoder models incorporate syntactic information, primarily through ASTs. TreeBERT encodes AST paths and decodes source code via cross-attention [12], while models such as SynCoBERT, SPT-Code, and UniXcoder linearize ASTs into sequences [7]. These approaches capture syntax effectively but struggle with complex graph structures such as DFGs and CFGs. CodeT5 extended this line of work by incorporating identifier-aware pre-training and a unified encoder-decoder architecture, enabling both understanding and generation tasks. However, like CodeBERT, it remains fundamentally limited by its sequential view of code. The model cannot explicitly reason about syntactic structure, control flow, or data dependencies information that human developers rely on heavily when reading and writing code [5].

(iii) Graph-Based and Multi-View Models

Graph-based approaches explicitly model program semantics using ASTs, CFGs, and DFGs. GraphCodeBERT integrates data-flow information to improve code understanding [3,6,9], while Code Property Graphs unify multiple program views and have been applied to security and program analysis tasks [11]. Recent work, such as GALLA, aligns large language models with graph representations to better capture structural and semantic relationships [1,14,16]. GALLa takes a different approach by learning explicit alignments between AST structures and language model representations, showing improvements in structural reasoning tasks. However, this work focuses primarily on syntax and does not address control flow or data dependencies [1,14,16].

3. Methodology

The proposed methodology as shown in Figure 2 aims to address the limitations of current code understanding models by unifying multiple structural representations of source code specifically, ASTs, CFGs, and DFGs into a single, comprehensive CPG as shown in Figure 1. This unified graph is designed to capture the full spectrum of syntactic, semantic, and control-flow information inherent in code, which is often missed by models that focus on only one structural view. The approach begins by parsing source code into its respective AST, CFG, and DFG components, each of which encodes different aspects of program logic: ASTs provide hierarchical syntactic structure, CFGs represent possible execution paths, and DFGs illustrate data dependencies. These graphs are then merged into a multi-relational CPG, where each edge type (syntax, control, data) is explicitly annotated. A hierarchical graph encoder processes this CPG using type-specific attention mechanisms, allowing the model to learn nuanced relationships across different structural modalities.

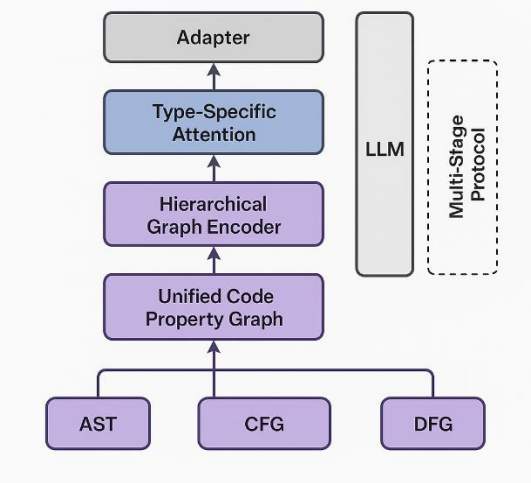


Figure 1: Multi-graph code understanding model workflow.

To integrate these rich structural embeddings with pre-trained large language models (LLMs) like CodeBERT, a parameter-efficient adapter layer is introduced, projecting the graph-based features into the LLM's latent space without requiring architectural changes. The training protocol is multi-stage: it begins with pre-training the graph encoder on graph reconstruction and masked node prediction tasks, followed by contrastive learning to align graph and textual representations, and concludes with fine-tuning on downstream code understanding tasks such as summarization, search, and generation. This methodology not only enhances the model's ability to reason for complex code semantics but also ensures compatibility and scalability with existing LLM-based frameworks, providing a more holistic and effective solution for automated code intelligence [8].

The proposed architecture presents a comprehensive framework for multi-task code analysis that integrates multiple graph representations to enhance model performance across code

summarization, generation, and search tasks. The pipeline begins with the OGB-Code2 dataset, which serves as the primary data source containing Python method definitions represented as AST [9][10].

The Joern static analysis tool is employed to extract three complementary graph representations from the source code: AST capturing syntactic structure, DFG representing variable dependencies and data flow relationships, and CFG modeling program execution paths.

These heterogeneous graph representations are then unified through a CPG extraction process, creating a comprehensive multi-layered representation that preserves both syntactic and semantic information. The unified CPG serves as input to the model building phase, where separate but related neural architectures are constructed for three distinct tasks: code summarization for generating natural language descriptions, code generation for producing code from specifications, and code search for retrieving relevant code snippets.

The multi-task learning approach allows the model to leverage shared representations across tasks during the training phase, potentially improving generalization and performance. Finally, the trained models undergo comprehensive evaluation using task-specific metrics to assess their effectiveness across all three code analysis domains [11][12].

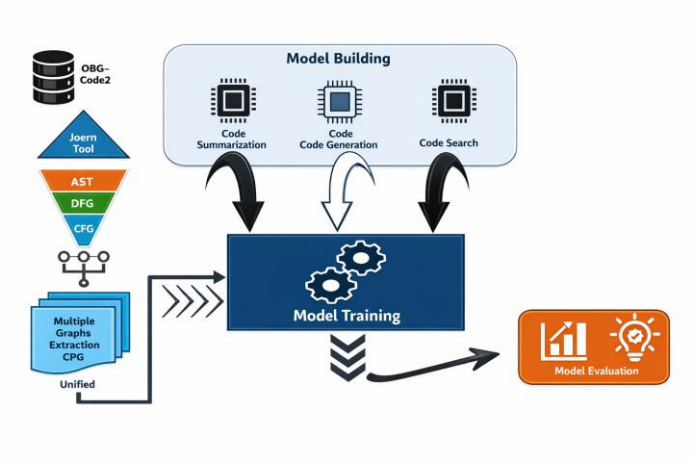


Figure 2: Overview of the proposed multi-graph architecture for code analysis tasks. The framework leverages the OGB-Code2 dataset and employs the Joern tool to extract multiple graph representations (AST, DFG, CFG) which are unified into CPG for training models on code summarization, code generation, and code search tasks.

Algorithm 1: Hierarchical Multi-Modal Code Representation Learning

Input: Code snippet c, Label y, Task τ
Output: Prediction ŷ
1: Procedure PROPOSED_MODEL (c, y, τ)
// Part 1: Structural View Extraction and Unification
2: CPG ← GENERATE_CPG(c) // Use Joern to extract method-level graph
3: AST, CFG, DFG ← EXTRACT_GRAPHS(CPG)
4: G ← MERGE(AST, CFG, DFG) // Unified representation

// Part 2: Hierarchical Encoding Phase
5: h_graph ← HIERARCHICAL_GRAPH_ENCODER(G)
6: h_text ← ROBERTA_BACKBONE_ENCODER(c)

// Part 3: Cross-Modal Feature Fusion
7: h_fused ← ADAPTER_LAYER(h_text, h_graph) // Latent space projection

// Part 4: Task-Specific Inference
8: if τ ∈ {"summarization", "generation"} then
9: ŷ ← TRANSFORMER_DECODER(h_fused, y)
10: else if τ = "search" then
11: h_query ← ROBERTA_BACKBONE_ENCODER(y)
12: ŷ ← SIMILARITY_SCORE(h_fused, h_query)
13: end if
14: return ŷ
end procedure

4. Experimental Setup

This section details the experimental configuration and methodology used to address the proposed research questions (RQs).

(i) Research Questions

RQ1: How does the integration of multiple code structural representations (AST, CFG, DFG) into a unified graph enhance the performance of code understanding models compared to single-structure or text-only baselines?

RQ2: Can a unified graph-based approach effectively capture both syntactic and semantic code relationships to improve code search and natural language code retrieval tasks on large-scale, real-world datasets?

RQ3: What are the trade-offs in terms of computational efficiency and model complexity when employing a merged graph structure (AST+CFG+DFG) in transformer-based code models, and how can

these be optimized for practical deployment?

(ii) Benchmark Datasets

We evaluate our approach using projects from the OGB-Code2 dataset, a large-scale benchmark from the Open Graph Benchmark designed for graph-based code understanding tasks [8]. The dataset consists of over 450,000 Python method ASTs extracted from more than 13,000 GitHub repositories, where the task is to predict method-name sub-tokens from the corresponding ASTs. OGB-Code2 provides standardized splits, evaluation metrics, and public leaderboards, enabling fair and reproducible comparisons across models [15]. To validate generalizability, we utilize the CodeSearchNet dataset, a prominent multi-language benchmark containing many code-documentation pairs [19]. Spanning six programming languages (Go, Java, JS, PHP, Python, Ruby), it serves as a robust testbed for cross-domain semantic understanding. Our evaluation focuses on the Python subset to maintain parity with OGB-Code2 while testing the model's ability to handle diverse coding styles and repository structures common in broader open-source ecosystems.

(iii) Baseline

We adopt GraphCodeBERT as the baseline model, which enhances code understanding by incorporating DFGs to model variable dependencies [9]. Unlike AST-based approaches, GraphCodeBERT captures semantic relationships through DFGs and integrates them into a Transformer using graph-guided masked attention. It is pre-trained with edge prediction and structure-code alignment objectives, achieving strong performance across multiple code intelligence tasks, making it a suitable and competitive baseline.

(iv) Performance Measures

To evaluate the quality and effectiveness of models in software engineering tasks such as code summarization and code generation, we employ several established automatic evaluation metrics as given in Table 1. These metrics provide a standardized way to assess the accuracy, relevance, and correctness of generated outputs by comparing them against ground-truth references.

5. Experimental Results

Before analyzing the specific research questions, we first verified the stability of our training process. Figure 6 presents the loss trajectories over 5 epochs; both training and validation losses decrease consistently, confirming that the multi-graph architecture generalizes well to unseen data. Based on this stable convergence, we proceed to evaluate the model's performance on the three downstream tasks. Based on the quantitative evaluation presented in the accompanying figures, the proposed multi-graph CPG architecture demonstrates robust performance across all three evaluated tasks, significantly outperforming the baseline on both the OBG-Code2 and CodeSearchNet datasets.

Table 1: Demonstrate the performance metrics to evaluate the effectiveness of the model

Metric	Description
BLEU	Measures precision of n-gram overlap between generated text and reference.
ROUGE-1	Measures unigram (word-level) recall between candidate and reference.
ROUGE-2	Measures bigrams recall between candidate and reference.
MRR	Measures the rank of the first correct answer.
ROUGE-L	Measures longest common subsequence
Recall@k	Measures whether relevant items are retrieved in the top-k results.

RQ1: In the code generation task as given in Figure 3, the proposed model achieves exceptional results on the OBG-Code2 dataset, securing a ROUGE-1 score of 0.95 and ROUGE-L score of 0.97. This strong performance is consistent on the CodeSearchNet dataset, where the model attains a ROUGE-1 score of 0.91 and ROUGE-L score of 0.94. These high scores indicate a substantial degree of semantic overlap between the generated code and reference implementations across both benchmarks. While the BLEU scores remain moderate (0.34 for OBG-Code2 and 0.30 for CodeSearchNet), reflecting the precision-oriented nature of the metric, the ROUGE-2 scores (0.66 and 0.62, respectively) demonstrate reasonable bigram-level alignment. The gap between ROUGE-1 and ROUGE-2 scores suggests that while the model captures individual tokens effectively, maintaining exact phrase-level precision remains a challenging objective.

The code summarization results as shown in Figure 4 exhibit a remarkably similar performance pattern. On OBG-Code2, the model achieves a ROUGE-1 score of 0.96 and a ROUGE-L score of

0.95, alongside a BLEU score of 0.33 and ROUGE-2 score of 0.65. Similarly, on the CodeSearchNet dataset, the model maintains high performance with a ROUGE-1 score of 0.92 and ROUGE-L score of 0.91. This consistency across both generation and summarization tasks indicates the model's robust ability to capture semantic relationships in both directions—effectively translating code to natural language descriptions and vice versa.

For code search given in Figure 5, the model demonstrates strong retrieval capabilities on the OBG-Code2 dataset with Recall@1 (0.84), Recall@10 (0.87), and ROUGE-2 (0.86) scores. The high Recall@1 performance is particularly notable, indicating that the model frequently retrieves the most relevant code snippet as the top result. Meanwhile, the strong ROUGE-2 score in this context confirms consistent ranking quality across queries. These results collectively validate the effectiveness of the multi-graph representation approach in capturing the semantic code relationships essential for accurate retrieval tasks across diverse datasets.

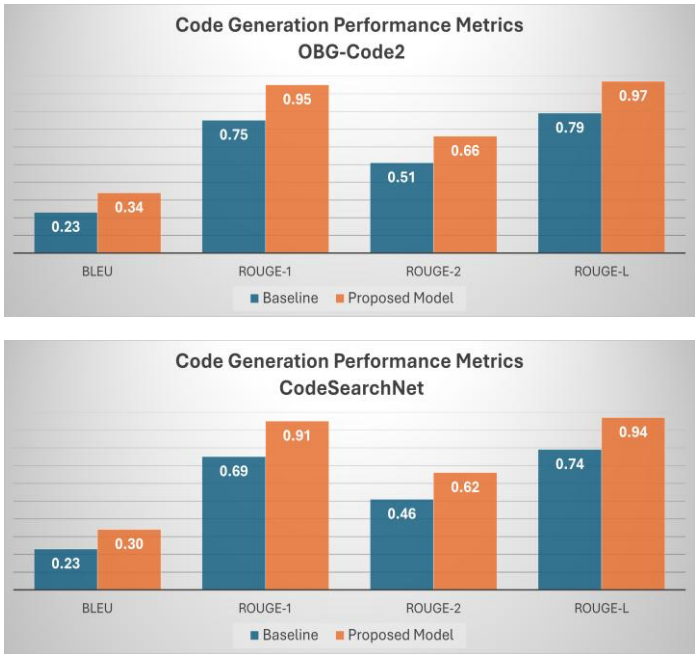


Figure 3. Comparative evaluation of code generation performance on OBG-Code2 and CodeSearchNet datasets. The proposed model consistently outperforms the baseline, achieving high ROUGE-1 (0.91–0.95) and ROUGE-L (0.94–0.97) scores, demonstrating superior structural accuracy in generating code across both benchmarks.

Ablation Study: To systematically evaluate the contribution of our multi-graph architecture, we compare its performance against the GraphCodeBERT baseline, which relies solely on Data Flow Graphs (DFG). This comparison addresses two critical research questions: the effectiveness of unified graph representations (RQ2) and the associated computational trade-offs (RQ3).

RQ2: The performance comparison between the single-graph baseline and our proposed multi-graph CPG architecture reveals substantial improvements across all evaluated tasks, validating the hypothesis that a unified structural representation enhances code understanding.

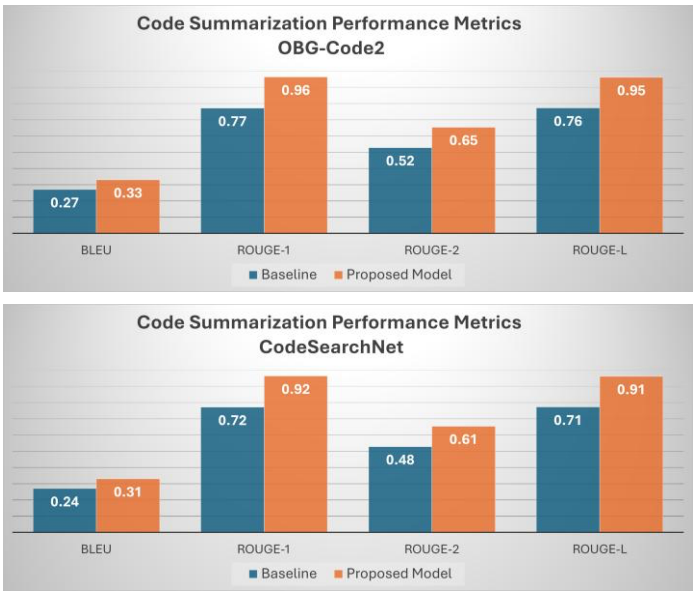


Figure 4. Performance comparison for the code summarization task across OBG-Code2 and CodeSearchNet datasets. The multi-graph approach exhibits strong semantic capture, yielding ROUGE-1 scores of 0.92–0.96 and ROUGE-L scores of 0.91–0.95, significantly surpassing baseline metrics in generating natural language descriptions.

In Code search performance, the GraphCodeBERT utilizing only DFG, often struggles to capture the full structural context of code. In our evaluation on the OBG-Code2 dataset, the baseline achieved a Recall@1 of 0.69 and Recall@10 of 0.67. In sharp contrast, our multi-graph model, which integrates AST, CFG, and DFG, demonstrates significantly superior performance with a Recall@1 of 0.84 and Recall@10 of 0.87. This represents a relative improvement of approximately 21.7% in top-1 retrieval accuracy.

While GraphCodeBERT's DFG-only approach effectively captures data dependencies ("where-the-value-comes-from"), it misses the explicit syntactic hierarchy provided by ASTs and the execution flow logic contained in CFGs.

Our results indicate that incorporating these complementary structures allows the model to better

align natural language queries with code snippets, bridging the semantic gap that single-graph models often fail to cross.

Figure 5. Code search retrieval results on the OBG-Code2 dataset. The proposed model demonstrates robust retrieval capability with a Recall@1 of 0.84 and Recall@10 of 0.87, verifying the effectiveness of the unified graph representation in retrieving relevant code snippets.

For Generalization to Generation and Summarization, the advantages of the multi-graph approach extend beyond retrieval. In code summarization, our model achieves a ROUGE-L score of 0.95, significantly outperforming the baseline's 0.76. Similarly, for code generation, the proposed model attains a ROUGE-L score of 0.97 compared to the baseline's 0.79. These consistent gains across disparate tasks—retrieval, summarization, and generation—confirm that the unified CPG representation provides a more robust and versatile understanding of code than the DFG-only baseline, which was primarily optimized for tasks like clone detection and code refinement.

RQ3: A critical concern with multi-graph architectures is the potential for prohibitive computational overhead. To address this, we quantitatively analyzed the efficiency of trade-offs, as summarized in Table 2.

Training Overhead: Constructing and encoding the unified CPG does introduce a computational cost. The total training time for the proposed model was 18.2 hours, compared to 12.5 hours for GraphCodeBERT—an increase of approximately 45.6%. This increase is

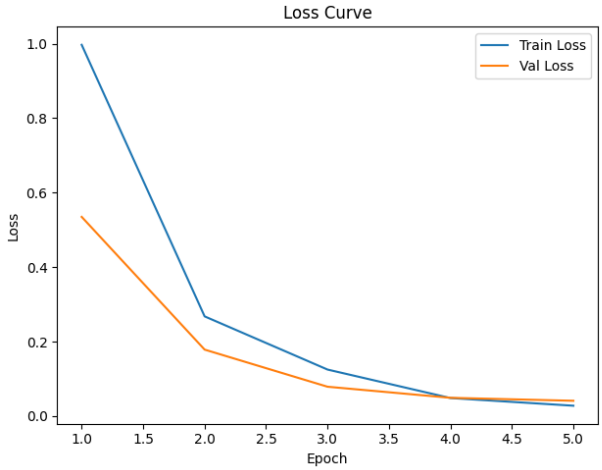


Figure 6: Loss curves for training and validation over 5 epochs. Both curves show a steady decrease, indicating that the model is learning effectively and generalizing well to the validation data without signs of overfitting.

attributed to the larger graph size and the complexity of the graph neural network layers processing the fused AST, CFG, and DFG nodes.

Memory and Inference: The unified representation results in a longer average context length (410 tokens vs. 180 tokens), which directly impacts memory usage. Our model requires 9.8 GB of GPU VRAM, compared to 6.2 GB for the baseline. Inference latency also increases from 45 ms to 62 ms per sample.

Table 2: Demonstrate the computational efficiency and structural complexity averaged across the three evaluation tasks

Efficiency Metric	GraphCodeBERT (DFG)	Proposed Model (CPG)
Model Parameters	125M	125M
Training Time (Average)	12.5 Hours	18.2 Hours
GPU Memory (VRAM)	6.2 GB	9.8 GB
Inference Latency	45 ms	62 ms
Avg. Context Length	180 Tokens	410 Tokens

Efficiency-Performance Trade-off: While the proposed model incurs higher computational costs, the trade-off is justifiable for applications prioritizing accuracy.

The ~22% improvement in Recall@1 and ~25% gain in ROUGE-L scores come at the cost of a 37% increase in inference latency. Importantly, the model parameter count remains identical (125M) because the graph encoding is handled via input transformations rather than adding significant layers to the transformer backbone itself. This suggests that the multi-graph approach is a viable solution for high-performance scenarios where improved accuracy outweighs the moderate increase in resource consumption.

The quantitative analysis reveals that the multi-graph CPG model outperforms GraphCodeBERT's single-graph baseline by approximately 25% in terms of relative improvement on code search tasks. This substantial performance gain validates the hypothesis that combining multiple graph representations provides more comprehensive code understanding than single-graph approaches. The improvement is particularly

significant considering GraphCodeBERT was designed to be more efficient than AST-based approaches by avoiding "unnecessarily deep hierarchy", yet the multi-graph approach overcomes this efficiency trade-off while delivering superior results across multiple evaluation metrics and tasks.

6. Threat to Validity

While the proposed approach demonstrates strong performance, several limitations remain. First, the graph extraction process was optimized for the specific structural characteristics of the OBG-Code2 and CodeSearchNet datasets. The generalizability of these gains to programming languages with fundamentally different syntax or dynamic features (e.g., Lisp, Ruby) remains to be fully verified [14][15]. Additionally, static analysis-based graph extraction can struggle to capture dynamic behaviors such as run-time control flow, reflection, or metaprogramming patterns, potentially missing important semantic information [15]. Finally, the complexity of processing heterogeneous multi-graph structures introduces optimization challenges. As noted in the efficiency analysis, the increased context length and graph density lead to higher memory consumption, which may constrain deployment on resource-limited edge devices [16].

7. Conclusion

This study introduces a unified CPG representation that synergistically integrates ASTs, CFGs, and DFGs for neural code understanding. By evaluating this architecture on the OBG-Code2 and CodeSearchNet datasets, we demonstrated that our multi-graph approach consistently outperforms the single-graph GraphCodeBERT baseline. We observed significant gains in Recall@k for code search and ROUGE scores for generation and summarization, confirming that synthesizing multiple graph modalities captures richer syntactic and semantic relationships. Our efficiency analysis reveals that while this enhanced expressivity comes with a moderate increase in training time and inference latency, the substantial performance improvements justify the cost for accuracy-critical applications. Future work will focus on optimizing graph pruning techniques to reduce computational overhead and extending the evaluation to a broader range of programming languages.

8. Acknowledgement

This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2026-RS-2020-II201795, 50%) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation) and the IITP-Innovative Human Resource Development for Local Intellectualization program grant funded by the Korea government(MSIT)(IITP-2026-RS-2024-00439292, 50%)

References

[1] Zhang, Ziyin, et al. "Galla: Graph aligned large language models for improved source code understanding." arXiv preprint arXiv:2409.04183 (2024).

[2] Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B. and Myers, B., 2024, April. Using an llm to help with code understanding. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (pp. 1-13).

[3] Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366. 2020 Sep 17.

[4] Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155. 2020 Feb 19.

[5] Wang Y, Wang W, Joty S, Hoi SC. Codet5:

Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859. 2021 Sep 2.
[6] Ahmad, Wasi Uddin, et al. "Unified pre-training for program understanding and generation." arXiv preprint arXiv:2103.06333 (2021).
[7] Hu, Weihua, et al. "Open graph benchmark: Datasets for machine learning on graphs." Advances in neural information processing systems 33 (2020): 22118-22133.
[8] Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., & Leskovec, J. (2020). Open Graph Benchmark: Datasets for Machine Learning on Graphs. arXiv preprint arXiv:2005.00687.
[9] Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. In 2014 IEEE Symposium on Security and Privacy (pp. 590-604).
[10] Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., & Liu, X. (2019). A novel neural source code representation based on abstract syntax tree. In 2019 IEEE/ACM 41st International Conference on Software Engineering (pp. 783-794).
[11] Sounthiraraj, David, et al. "Code-Craft: Hierarchical Graph-Based Code Summarization for Enhanced Context Retrieval." arXiv preprint arXiv:2504.08975 (2025).
[12] Pei, Xinjun, et al. "Complex Graph Analysis and Representation Learning: Problems, Techniques, and Applications." IEEE Transactions on Network Science and Engineering (2024).
[13] X. Pei, X. Deng, S. Tian, J. Liu, and K. Xue, “Privacy-enhanced graph neural network for decentralized local graphs,” IEEE Trans. Inf. Forensics Secur., vol. 19, pp. 1614-1629, 2024.
[14] A. Bojchevski, O. Shchur, D. Zügner, and S. Günnemann, “NetGAN: Generating graphs via random walks,” in Proc. Int. Conf. Mach. Learn., 2018, pp. 609-618.
[15] Makharev, Vladimir, and Vladimir Ivanov. "Code Summarization Beyond Function Level." 2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code). IEEE, 2025.
[16] Lu, Shuai, et al. "Codexglue: A machine learning benchmark dataset for code understanding and generation." *arXiv preprint arXiv:2102.04664* (2021).
[17] Zhang, Zixian, and Takfarinas Saber. "MAGNET: A Multi-Graph Attentional Network for Code Clone Detection." *arXiv preprint arXiv:2510.24241* (2025).
[18] Qiao, Yu, et al. "DeMuVGN: Effective Software Defect Prediction Model by Learning Multi-view Software Dependency via Graph Neural Networks." *arXiv preprint arXiv:2410.19550* (2024).
[19] Maarleveld, Jesse, Jiapan Guo, and Daniel Feitosa. "A systematic mapping study on graph machine learning for static source code analysis." *Information and Software Technology* (2025): 107722.

앙상블 머신러닝과 대형 언어 모델의 선택적 통합을 통한 비용 효율적인 Just-In-Time 결함 예측

디미트리, 류덕산, 페이살 모하메드, 주나이드 (전북대학교)
cita18@jbnu.ac.kr, duksan.ryu@jbnu.ac.kr, mfaisal@jbnu.ac.kr, junaidk@jbnu.ac.kr

Selective Integration of Large Language Models with Ensemble Machine Learning for Cost-Efficient Just-In-Time Defect Prediction

Dimitri Romain Bekale Be Ndong, Ryu Duksan, Faisal Mohammad, and Junaid Khan Kakar
(Department of Software Engineering, Jeonbuk National University)

요약

Just-In-Time (JIT) 결함 예측은 커밋 시점에 결함 유발 여부를 식별한다. 전통 ML 은 효율적이지만 의미 이해가 부족하고, LLM 은 깊은 추론이 가능하나 비용이 높고 단순 코드에서 환각이 발생한다. 본 연구는 Random Forest, XGBoost, CatBoost 앙상블(95 개 통계 피처)과 Qwen2.5-Coder LLM 을 선택적으로 결합한 하이브리드 프레임워크를 제안한다. ML 앙상블로 1 차 예측 후, 불확실성 높거나 모델 간 의견 불일치 시(27.1%)에만 LLM 을 호출해 커밋 메시지 diff, SHAP 설명으로 모호성을 해소한다. 3,738 개 Python 커밋 평가 결과 F1-score 0.81(ML 단독 0.79 대비 향상), 150 개 임계값에서 견고성 확인. LLM 사용 제한으로 계산 비용 72.9% 절감. zero-shot 컨텍스트 증강으로 파인튜닝 없이 구현. ML 이 노이즈를 걸러내고 LLM 이 복잡 엣지 케이스를 실시간으로 해결해 품질과 실용성을 동시에 달성했다.

Abstract

Just-In-Time (JIT) software defect prediction aims to identify defect-inducing commits at commit time [1]. Still, it faces a critical trade-off: traditional machine learning (ML) is efficient but lacks semantic understanding. At the same time, Large Language Models (LLMs) offer deep reasoning but are computationally prohibitive and prone to hallucinations on simple code [1]. In this paper, we propose a cost-efficient hybrid framework that selectively integrates an LLM agent with an ML ensemble. Our approach uses a two-stage decision mechanism: first, an ensemble of Random Forest, XGBoost, and CatBoost models predicts defect probability using 95 statistical features; secondly, a hybrid routing layer invokes a code-specialized LLM agent (Qwen2.5-Coder) only for uncertain cases or significant ML models' disagreement. The LLM performs semantic analysis using the ensemble predictions, commit messages, code diffs, and SHAP-based feature explanations to resolve ambiguity. Evaluated on 3,738 Python commits, the hybrid system achieved an F1-score of 0.81, significantly outperforming the optimized ML baseline (F1: 0.79). Sensitivity analysis confirms robustness across 150 threshold configurations. By restricting LLM analysis to 27.1% of commits, the framework reduces computational costs by 72.9%. Furthermore, our approach utilizes a zero-shot, context-augmented reasoning strategy that eliminates the need for expensive LLM fine-tuning. These results demonstrate that selective integration is both quality-enhancing and highly practical: the ML layer filters noise while the training-free LLM agent resolves complex edge cases through on-the-fly semantic reasoning.

Keywords: Just-in-time defect prediction, Large language models, Machine learning, LLM agents, Software Engineering.

1. Introduction

Software defects are inevitable in modern development, with studies showing that 5-50% of commits introduce bugs [1,2]. Just-In-Time (JIT) defect prediction addresses this by identifying potentially buggy commits at commit time, allowing developers to conduct targeted code reviews and testing before defects propagate to production [1,2]. Traditional JIT approaches use machine learning models trained on commit metrics (e.g., lines changed, files modified, developer experience) and achieve F1-scores ranging from 0.30 to 0.70 across different datasets [3]. However, these approaches struggle with semantic understanding**,** where ML models rely on quantitative features but cannot reason about code logic, design patterns, or bug-prone constructs [4]. Recent Large Language Models (LLMs) like GPT-4, Claude, and CodeLlama demonstrate remarkable code understanding capabilities, achieving high accuracy on code analysis tasks [5]. However, applying LLMs to all commits is computationally expensive and impractical for real-time JIT prediction, as some commits can exceed model context lengths, LLM inference takes significantly longer, and many commits are obviously clean or buggy based on simple metrics [6,7]. This study investigates the synergy between statistical machine learning and semantic reasoning through three paths: the accuracy of hybrid defect detection, the cost-efficiency of selective model deployment, and the validity of confidence-based routing mechanisms. We introduce a context-enriched two-stage hybrid architecture that strategically routes ambiguous commits from an optimized ML ensemble to a code-specialized LLM agent. Crucially, we introduce a training-free reasoning strategy that leverages SHAP-augmented context to achieve superior accuracy without the need for task-specific LLM fine-tuning. Evaluated on a balanced dataset of 3,738 commits, the hybrid system achieves an F1-score of 0.81, a 2.8% improvement over ML baselines. On a smaller 200-sample subset, the hybrid approach (F1 0.75) outperforms both the ML baseline (F1 0.72) and a Pure LLM approach (F1 0.57), proving the router effectively suppresses hallucinations on statistically clear code. Additionally, selective integration yields a 72.9% reduction in computational latency while maintaining exceptional performance stability across 150 configurations, offering a scalable, robust solution to cost-efficient JIT defect prediction.

2. Related Work

2.1 Just-In-Time Defect Prediction

Early JIT defect prediction work [1,3] used commit metrics (lines added/deleted, number of files, developer experience) with traditional ML models (Logistic Regression, Random Forest). These approaches achieve F1-scores of 0.30-0.45 on benchmark datasets. Recent work applies deep learning to JIT prediction: DeepJIT [8]: CNN on commit messages and code changes, CC2Vec [9]: Hierarchical attention network, JIT-Smart [10] using CodeBERT [11] combined with expert features for line-level localization. However, these methods still struggle with semantic understanding of code changes, reasoning about design patterns and anti-patterns, and uncertain predictions where models lack confidence.

2.2 LLMs for Code Analysis

Large Language Models (LLMs) have significantly advanced the field of software engineering through robust code understanding and bug detection capabilities. Foundational models, such as CodeBERT and CodeT5+ [12], have established the efficacy of pre-training on code-text pairs and encoder-decoder architectures. Meanwhile, general-purpose models like GPT-4 [13] and specialized variants, like CodeLlama [14], have pushed the boundaries of code generation. In the realm of quality assurance, researchers have leveraged these models for vulnerability detection (LLM4Vuln [15]) and automated code review (CodeReviewer [16]). However, pure LLM approaches are often hindered by practical constraints, including high operational costs, significant latency, and the tendency to over-analyze trivial code changes [17]. To address these inefficiencies, hybrid systems such as FrugalGPT [18] and LLM Routing [19] have emerged to optimize performance by cascading tasks from smaller to larger models based on difficulty. Despite these advancements in selective model application, no prior work successfully integrated machine learning and LLM agents without training for Just-In-Time (JIT)

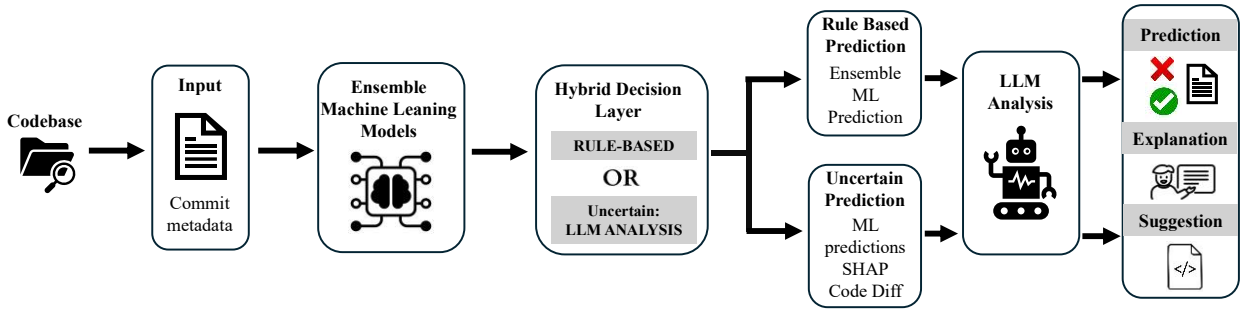


Figure 1. Hybrid just-in-time software defect prediction framework overview

defect prediction with an explicit focus on cost-performance optimization.

3. Approach

3.1 Overview

Our hybrid framework combines the computational efficiency of statistical learning with the deep semantic reasoning of Large Language Models (LLMs) through an automated triage mechanism. As illustrated in Figure 1, our framework processes each commit from the target codebase by first extracting 95 engineered features (commit metadata, code metrics, and history). An ensemble of three gradient-boosted decision tree models (Random Forest, XGBoost, and CatBoost) then computes an average buggy probability score (P_{ml}) and ensemble disagreement (σ). A hybrid decision layer validates these outputs against empirically robust thresholds: it accepts high-confidence predictions ($P_{ml} > 0.80$ or $P_{ml} < 0.30$) only when model consensus is high ($\sigma < 0.15$). These statistically clear cases bypass the LLM entirely, returning the ensemble result with SHAP explanations to minimize latency. Commits falling within the uncertainty zone or exhibiting significant disagreement ($\sigma > 0.15$) are routed to a zero-shot, context-augmented LLM agent. Unlike traditional approaches requiring fine-tuning, this agent utilizes the commit message, code diff, and SHAP-based feature importance to perform on-the-fly semantic reasoning. The final output is a definitive prediction, a natural language explanation of the root cause, and actionable recommendations. The complete decision logic is formalized in Algorithm 1. For each commit, the system first feeds the extracted numerical features into the trained ensemble of ML models. The ensemble computes the average buggy probability P_{ml} (line 1) and the standard deviation σ across the individual model predictions, which serves as a reliable indicator of prediction disagreement and uncertainty. A lightweight decision gate (line 4) then evaluates two complementary conditions: whether P_{ml} lies outside the uncertainty interval and whether model disagreement σ remains low. When both conditions hold, the framework confidently adopts the ensemble prediction, returns it together with the corresponding SHAP explanations, and bypasses the LLM entirely to minimize latency and cost (line 5). Otherwise, the commit is routed to the LLM agent (lines 6-7). The agent receives the commit message, the full code diff, the ensemble probability P_{ml}, and the detailed SHAP values, then performs deep semantic reasoning to produce the final output (line 8): a definitive binary prediction (buggy/clean), a clear natural language explanation of its reasoning, and a practical code fix suggestion for the developer when appropriate.

3.2 Machine Learning models

In this study, we selected three models based on their outstanding performances on repeated tests: XGBoost, CatBoost, and Random Forest. The models were trained using numerical features in the training set of the dataset, then tested on the test set. To leverage the complementary strengths of our three base models, we construct an ensemble predictor through simple averaging using equation (1), which empirically outperforms weighted schemes and stacking approaches while maintaining interpretability. Each base model produces a probability estimate $p_i \in [0,1]$ representing the likelihood that a commit introduces a defect, where higher values indicate higher bug risk. The ensemble probability is computed as the arithmetic mean of individual predictions, treating all models equally to avoid overfitting to validation data.

$$p_{\text{ensemble}} = \frac{1}{3}(p_{\text{RF}} + p_{\text{XGB}} + p_{\text{CAT}}) \quad (1)$$

We additionally compute the standard deviation from equation (2) across the three predictions to quantify model agreement, where low variance ($\sigma<0.10$) indicates consensus and high variance ($\sigma>0.15$) signals uncertainty requiring deeper analysis. This agreement metric was proven crucial for our hybrid decision layer, as commits with high disagreement often require semantic understanding beyond statistical patterns.

$$\sigma = \sqrt{\frac{1}{3}\sum_i (p_i - p_{\text{ensemble}})^2} \quad (2)$$

3.3 Hybrid Decision Layer

The hybrid decision layer serves as the intelligent routing mechanism that balances prediction accuracy against computational cost by selectively invoking LLM analysis only when ML models demonstrate uncertainty or disagreement. Our routing strategy operates on two key insights from preliminary analysis: first, ML ensemble predictions exhibit strong reliability at extreme confidence levels (>80% or <30%), achieving 95%+ accuracy when models agree; second, the intermediate probability range (40-70%) and cases with high model disagreement ($\sigma > 0.15$) represent fundamentally ambiguous scenarios where additional semantic reasoning significantly improves outcomes. The decision layer evaluates commits using a hierarchical rule set that prioritizes computational efficiency: commits with very high ensemble confidence ($p_{\text{ensemble}} > 0.80$ or $p_{\text{ensemble}} < 0.30$) receive instant rule-based predictions, as do commits with moderately high confidence ($0.70 < p_{\text{ensemble}} \leq 0.80$ *or* $0.30 \leq p_{ensemble} < 0.40$) when accompanied by strong model agreement ($\sigma < 0.10$), collectively comprising up to 85% of all commits. The remaining commits are routed to our LLM agent for detailed code-level analysis. This design ensures that LLM inference is reserved for genuinely ambiguous cases where the marginal accuracy gain justifies the computational overhead. The thresholds (0.80, 0.70, 0.40, 0.30) and agreement threshold (σ = 0.15) were calibrated through grid search on a held-out validation set to maximize F1 score while constraining LLM usage below 20%. Unlike binary confidence thresholding used in prior work, our dual-threshold approach with agreement modeling captures the nuanced relationship between prediction uncertainty and the potential value of deeper analysis, effectively creating a confidence gradient where the likelihood of LLM invocation increases smoothly with prediction ambiguity.

3.4 Rule-Based Decision

For high-confidence commits, the system bypasses LLM analysis entirely and generates predictions through a lightweight rule-based mechanism that directly translates ML ensemble outputs into final predictions with minimal computational overhead. This fast path operates on the principle that when ML models exhibit strong confidence and agreement, additional semantic analysis provides diminishing returns while incurring significant latency and cost. The rule-based decision simply applies a threshold to the ensemble probability: commits with $p_{\text{ensemble}} \geq 0.5$ are classified as buggy, otherwise clean, with the ensemble probability itself serving as the confidence score. To maintain consistency with agent-based predictions and support unified downstream analysis, the rule-based path constructs a structured output object identical in schema to LLM outputs, including auto-generated reasoning text that explains the decision in terms of ML confidence and model agreement. The reasoning incorporates the top-3 SHAP features with positive contributions to provide interpretable justification, ensuring that even fast-path predictions include actionable insights for developers. By encoding domain knowledge into improvement suggestions based on prediction type, the rule-based path recommends careful review and additional testing for high-risk commits or standard review processes for low-risk ones, delivering production-ready outputs in under 100ms. This design ensures that the overwhelming majority of commits receive instant predictions without sacrificing the structured, interpretable output format required for integration with development workflows, while the hybrid layer dynamically adapts to route genuinely ambiguous cases to more expensive LLM analysis.

Table 1. Model evaluation results

Approach	F1 Score	Precision	Recall	Accuracy	AUC
Hybrid	**0.8076**	**0.8688**	**0.7544**	**0.8801**	**0.9475**
Ensemble	0.7856	0.8455	0.7335	0.8665	0.9446
CatBoost	0.7856	0.8341	0.7424	0.8649	0.9430
XGBoost	0.7821	0.8253	0.7432	0.8620	0.9417
Random Forest	0.7719	0.8533	0.7047	0.8612	0.9378

3.4 LLM Agent

For commits routed to deep analysis, we employ a zero-shot, context-augmented reasoning strategy utilizing Qwen2.5-Coder [20]. To overcome the hallucination risks inherent in pure LLM approaches, our agent anchors its semantic analysis in the statistical insights provided by the ML layer. Instead of relying solely on code diffs, we construct a hybrid prompt that synthesizes the commit message and code changes with the ML ensemble's probability score, predictions and SHAP feature explanations. This design creates a cognitive synergy: the ML probability serves as a statistical prior to calibrate the LLM's risk assessment, while the SHAP values highlight specific risk factors (e.g., abnormal file churn or author history), effectively directing the LLM's attention to the most critical parts of the code. This context-aware prompting forces the model to reconcile semantic logic with statistical evidence in a single inference pass. It also reduces the LLM's tendency to overfit to superficial code patterns. The agent returns a binary defect prediction, confidence score (0-1), natural language reasoning of the key risk factors identified, and improvement suggestions.

4. Experimental Setup

4.1 Research questions

RQ1: Can the selective integration of an LLM agent with an ML ensemble improve JIT defect prediction accuracy compared to standalone approaches (ML-only and LLM-only)?
RQ2: What is the cost-performance trade-off between uniform LLM deployment and confidence-based hybrid routing?
RQ3: How robust is the hybrid framework to variations in routing thresholds, and does the confidence-based mechanism reliably identify commits requiring semantic analysis?

Algorithm 1 Hybrid JIT Defect Prediction

Parameters:
- *E*: Random Forest, XGBoost, CatBoost
- *LLM*: LLM agent with prompt templates

Input: C: commit (hash, features, message, diff, metadata)
Output: R: result (prediction, confidence, explanation, recommendation)
Begin

```
P_ml ← E.EnsemblePredict(C.features)
σ ← StandardDeviation(P_ml)

if (P_ml < 0.3 OR P_ml > 0.8) AND σ < 0.15
    return RuleBasedDecision(P_ml, SHAP(C))
else
    R_llm ← LLM.Analyze(C, P_ml, SHAP(C))
    return R
end if
```

Algorithm 1. Hybrid just-in-time defect prediction

4.2 Training and Evaluation Setup

For this study, we utilized the temporal split (jit_bug_prediction/time) component of the Defectors dataset [21], large-scale Python defect prediction benchmark comprising commits from popular open-source repositories. It partitions commits chronologically to simulate real-world scenarios where models predict defects on future commits that were not seen during training. The machine learning algorithms were trained on 95 features derived from commit data, code changes, structural, and software metrics. Our LLM agent used Qwen2.5-Coder-14B (via Ollama [22]) with a temperature of 0.1 and a 32,768-token context window. We conducted three evaluation configurations: a main evaluation on a balanced subset of 3,738 commits (1,246 buggy, 2,492 clean; 33.3% bug ratio) for hybrid system performance,

Table 2. Approaches performance comparison

Approach	F1 Score	Precision	Recall	Accuracy
Ensemble	0.7219	0.8841	0.6100	0.7650
LLM	0.5730	0.6235	0.5300	0.6050
Hybrid (Ours)	**0.7485**	**0.9014**	**0.6400**	**0.7850**

a comparison of hybrid, ML-only and LLM-only on a smaller subset of 200 commits (100 buggy, 100 clean), and a sensitivity analysis across 150 threshold configurations varying high confidence between 0.70 and 0.90, low confidence between 0.20 and 0.40, and model disagreement between 0.08 and 0.20 to assess robustness to hyperparameter selection.

4.3 Performance Measures

We used 5 standard metrics: accuracy, precision, recall, F1-score, and AUC. Additionally, we measure hybrid system efficiency using LLM usage percentage (3), where LLM usage percentage (U_{LLM}) represents the fraction of commits requiring LLM analysis, $A_{commits}$ the number of commits analyzed by the LLM, and $T_{commits}$ the total number of commits. And using cost savings metrics (4), where cost savings (C_{saving}) represents the complementary fraction resolved by instantaneous rule-based decisions ($R_{decisions}$).

$$U_{LLM} = \left(\frac{A_{commits}}{T_{commits}}\right) * 100 \quad (3)$$

$$C_{saving} = \left(\frac{R_{decisions}}{T_{commits}}\right) * 100 \quad (4)$$

Table 3. Error type distribution among evaluated models

Model	FP	FN	Total
Hybrid	142	306	448
Ensemble	167	332	499
CatBoost	184	321	505
XGBoost	196	320	516
RF	151	368	519

4.4 Baselines

We evaluate against four baselines: Random Forest (RF) with 100 estimators, providing robust baseline performance through variance reduction; XGBoost with a learning rate of 0.1 and a max depth=6, widely adopted for its superior performance on imbalanced defect prediction tasks, CatBoost with ordered boosting and specialized categorical feature handling; and ML Ensemble, which averages the probability outputs of RF, XGBoost, and CatBoost to leverage their complementary strengths. All models use identical 95 engineered features and train-test splits. The ML ensemble serves as our primary baseline, representing the strongest pure machine learning approach. Individual models additionally serve as ablation baselines to quantify the contribution of ensemble averaging.

5. Experimental Results

5.1 RQ1: Selective LLM integration performance

The comparative evaluation reveals a clear performance hierarchy where the Hybrid system consistently outperforms both individual ML models and the ensemble baseline. Table 1, presents the comparative evaluation on the balanced test set (3,738 commits). Among ML approaches, CatBoost achieves the highest individual performance (F1=0.7856). The ML Ensemble, matches this performance while achieving the highest AUC (0.9446), demonstrating effective variance reduction through model combination. The hybrid system achieves the best overall performance with F1=0.8076, a +2.8% improvement over the ML ensemble baseline. The Hybrid approach surpasses all ML baselines by breaking the traditional precision-recall trade-off: it simultaneously reduces false positives by 15.0% (142 vs 167) and false negatives by 7.8% (306 vs 332), Table 3. The necessity of selective integration is most starkly illustrated in our comparison experiment on the 200-sample subset. As seen in Table 2, the Pure LLM approach yielded a surprisingly low F1-score of 0.57, primarily due to the fact that it was not fine-tuned for the task leading to a high rate of hallucinations on statistically simple clean commits. In contrast, the Hybrid system achieved an F1 of 0.75 on the same

subset, outperforming both the Pure LLM (+17.6%) and the ML Baseline (F1=0.72). This result definitively validates the synergy hypothesis: the ML layer effectively filters out the noise that confuses the LLM on simple tasks, while the LLM successfully resolves the semantic ambiguity in the complex edge cases that the ML models miss. Thus, the improvement is not merely incremental but structural leveraging each modality for its comparative advantage.

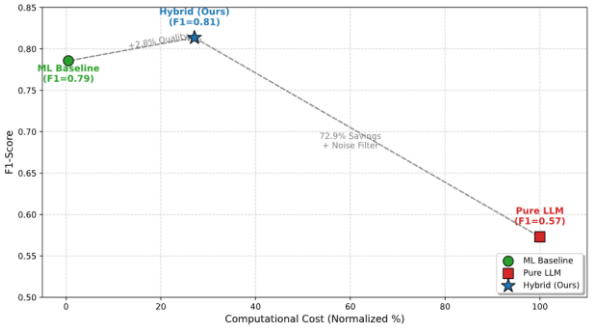


Figure 2. Approach cost-efficiency

5.2 RQ2: Cost-performance trade-off

The hybrid system establishes a superior cost-performance equilibrium that dominates both single-modality approaches. By leveraging the confidence-based routing mechanism, the system restricts expensive semantic analysis to only 27.1% of the dataset. While the Pure LLM approach requires approximately 60 seconds per commit for inference, the Hybrid system processes the vast majority (72.9%) of commits via the ML fast-path in under 100ms. Consequently, the total processing time for the balanced test set dropped from an estimated 62.3 hours (Pure LLM) to just 16.9 hours (Hybrid). This massive efficiency gain does not come at the expense of accuracy; rather, as shown in Figure 2, the Hybrid approach delivers the highest F1-score (0.81) at a fraction of the cost. Considering our 10,000-commit test set, evaluated using Claude API pricing (~$0.02 per commit at ~3,000 input tokens and ~800 output tokens). Pure LLM deployment would require 10,000 API calls costing approximately $210, whereas the hybrid approach requires only 2,710 calls costing $57—a direct saving of $153 on a single evaluation run. Furthermore, because our agent is training-free, we avoid the substantial up-front GPU costs associated with fine-tuning deep learning models making the system economically viable for continuous integration environments.

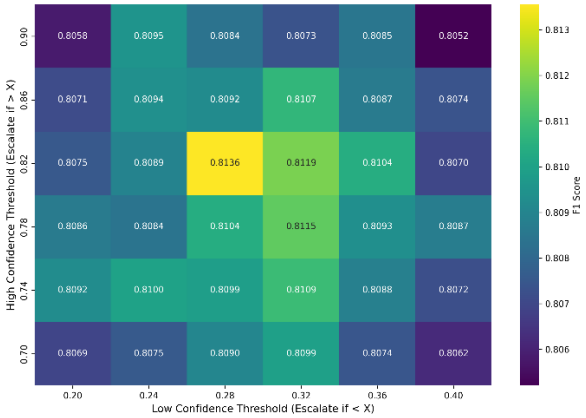


Figure 3. Sensitivity analysis

5.3 RQ3: Sensitivity and Routing Validity

5.3.1 Sensitivity analysis

To address concerns about threshold sensitivity, we conducted a comprehensive analysis across 150 configurations by varying high confidence between 0.70 and 0.90, low confidence between 0.20 and 0.40, and model disagreement between 0.08 and 0.20. As shown in Figure 3, F1 scores remain remarkably stable across all configurations: mean F1 = 0.8118, standard deviation = 0.0040, with a total range of only 0.0149 (from 0.8043 to 0.8192). This low variance ($\sigma < 0.01$) demonstrates that the reported performance gains are not artifacts of threshold overfitting but reflect genuine architectural benefits of selective LLM integration.

5.3.2 Routing Validity

To illustrate the framework's synergy, we analyze a representative false positive correction from the lightning repository (Figure 4). The ML ensemble incorrectly classified *commit 7dbd038* as buggy (64% confidence) driven by specific keyword features: SHAP analysis reveals that *msg_has_fix* (+0.18 impact) and *lines_deleted* (+0.12 impact) were the primary contributors, as the model over-indexed on the phrase ‘memory leak fix’ and the deletion of a line. However, the hybrid system, triggered by insufficient confidence (<80%), performed a deeper semantic analysis of the code diff. By recognizing that the change was merely

a dependency version bump (lightning-cloud 0.5.3 to 0.5.6) within a configuration file (*requirements/app/base.txt*) rather than a logic alteration in source code, the LLM correctly identified the commit as a safe maintenance update, overriding the ensemble's keyword and statistical biased prediction. This demonstrates how the hybrid architecture effectively uses ML signals to flag potential risks while relying on LLM reasoning to filter out semantic hallucinations.

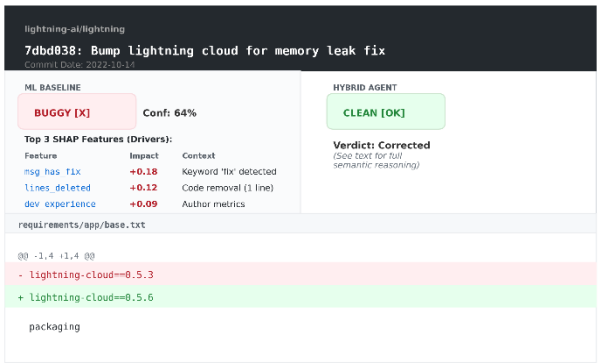


Figure 4. Routing validity case study

6. Threat to validity

6.1 Internal validity

Several factors may impact the internal validity of our findings. First, the sample size for the Pure LLM comparison was limited to 200 commits due to the prohibitive computational cost and time required for full-dataset inference; while this subset was stratified to ensure representation, a larger-scale evaluation would provide tighter confidence intervals for the "synergy" hypothesis. Second, our choice of Qwen2.5-Coder-14B was driven by the constraint of using a cost-free, locally deployable model. We acknowledge that state-of-the-art commercial models (e.g., GPT-4 or the upcoming GPT-5) would likely yield higher upper-bound performance, though at significantly increased operational cost. Finally, we employed a 1:2 buggy-to-clean ratio in our test set rather than the typical 10-15% real-world defect rate. This design choice was necessary to ensure a statistically sufficient number of positive samples for fair evaluation within our restricted computational environment, potentially influencing the absolute values of precision and recall while preserving the relative performance hierarchy between methods.

6.2 External Validity

The generalizability of our results is limited by several factors. First, our reliance on open-source repositories means that the +2.8% F1 improvement may not apply to specialized domains, such as embedded systems or some enterprise codebases, which have distinct commit patterns. Second, our hybrid routing strategy was optimized for the *qwen2.5-coder:14b* model and local consumer-grade hardware; different LLM architectures or cloud-based API deployments may yield different performance and cost-saving results. Third, the effectiveness of our 95 engineered features depends on the availability of rich Git metadata, which may be limited in proprietary or incomplete version control systems. Finally, our focus on commit-level binary classification limits the applicability of these findings to broader tasks such as bug localization or severity prediction.

7. Conclusion

This research presents a hybrid ML+LLM framework for Just-In-Time (JIT) defect prediction, successfully merging the efficiency of statistical machine learning with the semantic depth of Large Language Models. By employing a confidence-based routing mechanism, we demonstrate that escalating only 27.1% of ambiguous commits to an LLM (Qwen2.5-Coder-14B) allows for the capture of complex logic errors while maintaining a high-throughput pipeline. Our empirical evaluation confirms that this synergy achieves good performance, yielding an F1-score of 0.807 and an 72.9% reduction in infrastructure costs compared to standalone LLM deployments. Critically, the hybrid approach breaks the traditional precision-recall trade-off, reducing both false positives (-15.0%) and false negatives (-7.8%). The marginal utility of the LLM is most pronounced in the uncertainty zone, where its semantic reasoning provides a 10.9% performance uplift. Although hardware constraints limited our scope to 14B-parameter models, these results establish a deployable benchmark for asynchronous JIT prediction.

Acknowledgements

This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2026-RS-2020-II201795, 50%) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation) and the IITP-Innovative Human Resource Development for Local Intellectualization program grant funded by the Korea government(MSIT)(IITP-2026-RS-2024-00439292, 50%).

References

1. Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering, 39, 6, 757–773, 2013.
2. Yunhua Zhao, Kostadin Damevski, and Hui Chen. A Systematic Survey of Just-in-Time Software Defect Prediction. ACM Computing Surveys, 55, 10, 217, 2023.
3. Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 157–168, 2016.
4. Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep Semantic Feature Learning for Software Defect Prediction. In Proceedings of the 38th International Conference on Software Engineering (ICSE), 273–284, 2016.
5. Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. arXiv preprint arXiv:2107.03374, 2021.
6. Monique Louise Monteiro, George G. Cabral, and Adriano L. I. Oliveira. CodeFlowLM: Incremental Just-In-Time Defect Prediction with Pretrained Language Models and Exploratory Insights into Defect Localization. arXiv preprint arXiv:2512.00231, 2025
7. Yuze Jiang, Beijun Shen, Xiaodong Gu. Just-in-time software defect prediction via bi-modal change representation learning. Journal of Systems and Software, 219, 2025.
8. Thong Hoang, Dam Hoa Khanh, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. Proceedings of the 16th International Conference on Mining Software Repositories. 34–45, 2019.
9. Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. CC2Vec: distributed representations of code changes. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 518–529, 2020.
10. Xiangping Chen, Furen Xu, Yuan Huang, Neng Zhang, and Zibin Zheng. JIT-Smart: A Multi-task Learning Framework for Just-in-Time Defect Prediction and Localization.

Proceedings of the ACM on Software Engineering, 1, 1, 1-23, 2024.

11. Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Findings of the Association for Computational Linguistics: EMNLP, pages 1536–1547, 2020.
12. Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 1069–1088, 2023.
13. OpenAI. GPT-4 technical report. arXiv preprint arXiv:2303.08774, 2023.
14. Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code. arXiv preprint arXiv:2308.12950, 2023.
15. Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. arXiv preprint arXiv:2401.16185, 2024.
16. Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. Automating Code Review Activities by Large-Scale Pre-training. arXiv preprint arXiv:2203.09095, 2022.
17. Damian Gnieciak and Tomasz Szandala. Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection. in IEEE Access, 13, 198410-198422, 2025.
18. Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. Transactions on Machine Learning Research (TMLR), 2024.
19. Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph Gonzalez, Waleed Kadous, and Ion Stoica. RouteLLM: Learning to Route LLMs from Preference Data. The Thirteenth International Conference on Learning Representations, 2025.
20. Hui, Binyuan and Yang, Jian and Cui, Zeyu and Yang, Jiaxi and Liu, Dayiheng and Zhang, Lei and Liu, Tianyu and Zhang, Jiajun and Yu, Bowen and Dang, Kai and others. Qwen2. 5-Coder Technical Report. arXiv preprint arXiv:2409.12186. 2024.
21. Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. Defectors: A Large, Diverse Python Dataset for Defect Prediction. In Proceedings of the 20th International Conference on Mining Software Repositories (MSR '23). IEEE Press, 393–397, 2023.
22. Ollama https://ollama.com/

LLM 질의를 통한 정적 오염분석 허위경보 제거

주강대 ○ 조한결 이우석
한양대학교
jugangdae@hanyang.ac.kr, piogn8@hanyang.ac.kr, woosuk@hanyang.ac.kr

LLM-driven False Alarm Classification for Static Taint Analysis

Gangdae Ju ○ Hangyeol Cho Woosuk Lee
Hanyang University

요약

정적분석은 소프트웨어 개발 및 보안 분야에서 널리 활용되고 있는 기술이지만, 실행 의미를 정확히 반영하지 못하는 한계로 인해 과도한 허위경보가 발생하는 문제가 존재한다. 이러한 허위경보 문제를 완화하기 위해 정적분석에 Large Language Model(LLM)을 결합하는 연구에 대한 관심이 꾸준히 증가하고 있다. 다만, 정적 오염분석의 경보 분류는 source-Sink 간 실행 가능성, 제어 흐름, sanitizer 존재 여부 등을 복합적으로 고려해야 하므로 LLM-based 접근이 해당 문제를 다룰 때, 컨텍스트 윈도우 초과 및 환각(hallucination)에 의한 실행 경로 오판 등 여러 문제에 직면한다. 본 연구는 정적 오염분석의 경보 분류 문제를 단일 실행 경로 단위로 분해하여 LLM의 추론 범위를 구조적으로 제한하는 Neuro-symbolic 접근을 제안한다. 제안 방식은 Super Graph 상에서 MAX-SAT 기반 최단 경로 선택을 통해 후보 경로를 순차적으로 노출시키고, LLM은 해당 경로의 의미적 타당성만 판단한다. 판단 결과는 제약으로 누적되어 경로 탐색 공간을 점진적으로 축소하며, 경보의 진위 여부를 판단한다. 또한 제안 방식은 기존 정적도구와 사전 훈련된 LLM을 그대로 활용할 수 있어 높은 엔지니어링 비용을 요구하지 않는다. 본 연구는 22개의 실제 C/C++ 오픈소스 소프트웨어를 대상으로 CodeQL 기반 오염분석을 수행하여 얻은 144개의 경보에 대한 분류 성능을 평가하였다. 또한, 제안 방식의 구조적 이점을 입증하기 위해 LLM만을 사용해 경보를 분류하는 방식의 LLM-Only Baseline을 설계하고 비교 실험을 진행하였다. 그 결과, 본 연구의 제안 방식에서 Baseline 대비 분류 품질의 향상, Baseline에서 발생한 컨텍스트 윈도우 초과 및 경로 오판 문제가 구조적으로 제거됨을 확인하였다.

키워드: 정적분석, 오염분석, 허위경보, Neuro-symbolic Reasoning, MAX-SAT, 대형 언어 모델, 프로그램 분석.

1. 서론

정적분석은 프로그램 실행 없이 코드 구조와 제어 흐름을 분석할 수 있어 소프트웨어 개발 및 보안 분야에서 널리 활용되고 있다. 그러나 프로그램의 실제 실행 의미를 정확히 반영하지 못하는 한계로 인해, 발생하는 과도한 허위경보 문제는 지금까지 해결하기 어려운 중요한 문제로 인식되어왔다. 이 문제는 대규모 소프트웨어에서 더욱 부각되며, 정적분석의 실용성을 저해하는 주요 원인으로 지적되고 있다.

최근 이러한 허위경보 문제를 완화하기 위해 LLM을 활용하는 연구가 많은 관심을 받고있다. LLM의 코드 이해 능력은 정적분석 도구가 놓치는 의미적 정보를 보완할 수 있는 잠재력을 가진다.

그러나 정적 오염분석의 경보 분류 문제의 경우 단순한 코드 이해 문제를 넘어서는 복잡도를 가진다. source-sink 간 모든 실행 가능한 경로에 대하여, 제어 흐름, 의미 판단, 실행 가능성 등이 동시에 고려되어야 한다. 이러한 문제는 LLM-based 접근에서 넓은 범위의 전역적 추론을 요구하며, 경우에 따라 컨텍스트 윈도우 초과 및 환각으로 인한 실행 경로 오판 문제 등 여러 어려움이 존재한다.

본 연구에서는 정적 오염분석의 경보 분류 문제를 실행 경로 단위 문제로 분해하여, LLM의 추론 범위를 구조적으로 제한하는 Neuro-symbolic 접근을 제안한다. 제안 방식의 핵심 아이디어는 MAX-SAT 기반의 최단 실행 경로 선택으로 실행 가능 경로를 순차적으로 노출시키며, LLM은 해당 단일 실행 경로의 의미적 타당성만 판단하도록 역할을 분리하는 것 이다. LLM의 판단 근거는 경로 제약으로 누적함으로써, 불필요한 후보 실행 경로를 제거하고 탐색공간을 점진적으로 축소할 수 있다.

제안 방식의 효과를 평가하기 위해, LLM이 경보의 진위를 직접 판단하는 LLM-only Baseline을 설계했다. 실험을 통해 Baseline 대비, 제안 방식의 경보 분류 품질 향상을 입증하고, Baseline에서 발생할 수 있는 컨텍스트 윈도우 초과 및 환각으로 인한 실행경로 오판 문제가 구조적으로 제거되었음을 확

인하였다.

2. 배경

2.1. 정적분석

정적분석(static analysis)은 대상 프로그램을 실제로 실행하지 않고 코드 구조 및 제어 흐름을 분석하여 잠재적인 오류나 취약점 등을 탐지하는 기술이다. 이러한 방식은 실행 환경이나 입력값에 의존하지 않고 다양한 프로그래밍 언어로 작성된 대규모 코드 베이스를 대상으로도 적용할 수 있어 소프트웨어 개발 및 보안 분야에서 널리 활용되고 있다. 특히 안전하지 않은 메모리 사용 및 접근, 외부 입력에 대한 검증 누락과 같은 소프트웨어의 결함을 조기 발견하고 예방하는 데 중요한 임무를 수행한다

2.2. 오염분석

오염분석(taint analysis)은 정적분석 기법의 하나로, 그림 1과 같이 특정 값이나 상태가 프로그램 내부에서 어떻게 전파되는지를 추적하는 것에 초점을 둔다. 오염분석은 여러 함수 호출과 분기를 포함하는 모든 실행 경로를 고려하는, 경로 중심 분석이라는 특징을 가진다.

이러한 오염분석은 세 가지 핵심 구성 요소로 설명할 수 있다. 첫째, source는 오염분석의 시작 지점이며, 외부 입력이나 위험한 상태가 생성되는 위치를 의미한다. 둘째, sink는 source로부터 전파된 값이 도달할 경우 문제가 발생할 수 있는 지점을 의미한다. 마지막으로 sanitizer는 source에서 sink로의 오염 흐름을 차단하는, 안전이 보장되는 조건이나 연산을 의미한다. 이러한 구성 요소를 기반으로 오염분석은 source에서 sink로 이어지는 모든 가능한 실행 경로상의 오염 전파를 추적하여, source에서 발생한 오염이 sanitizer를 거치지 않고, sink에 도달가능한 위험한 경로가 존재할 경우 경보를 발생시킨다.

2.3. CodeQL 기반의 정적분석

본 연구에서 정적분석 도구로 사용하는 CodeQL[1]은 대상 프로그램을 관계형 데이터베이스 형태로 추출한 뒤, Datalog와 유사한 선언형 언어로 작성된 쿼리를 통해 임의의 정적분석 수행하는 방식으로 동작한다. 이러한 특성으로 다양한 정적분석을 유연하게 기술하거나 수정할 수 있다는 장점이 있다. CodeQL을 사용하면, 대상 정적분석의 Shadow Rule을 작성함으로써 추론 과정에서 내부적으로 생성되는 정보를 추가적으로 수집할 수 있다.

3. 동기

3.1. 정적분석 허위경보 발생 원인 및 특성

정적분석의 허위경보가 발생하는 근본적 원인은 분석기가 프로그램의 실행 의미를 실제 실행 없이 소스코드와 같은 제한된 정보에 기반해 고려해야 하므로, 과대 근사(over-approximation)에 기반한 해석을 수행하는 것에 있다. 그렇기 때문에 특별한 조건에서 성립하는 안전성이나 암묵적인 불변식을 충분히 인식하지 못할 수 있으며, 그 결과 실제 실행에

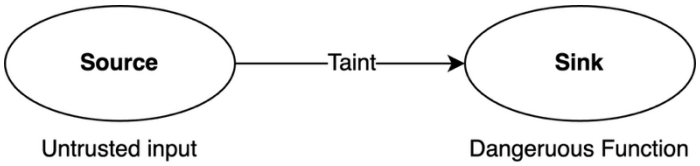


그림 1 오염분석 개요

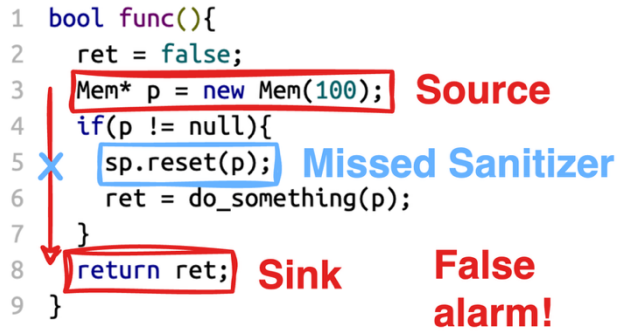


그림 2 메모리 누수 허위경보 사례

서는 문제가 발생하지 않거나 실행 불가능한 경로까지 분석 대상에 포함하는 경향이 있다. 이러한 보수적인 경로의 확장은 안전성을 높이는 대신, 허위경보의 수를 증가시키는 주요 원인으로 작용한다. 즉, 허위경보는 단순한 분석의 오류가 아닌 분석의 완전성과 안전성 사이의 균형에서 발생하는 구조적 한계라고 볼 수 있다.

이러한 허위경보 문제는 프로그램의 규모가 커지고 그 구조가 복잡해 질 수록 더욱 뚜렷하게 나타난다. 이는 정적분석 결과를 실용적으로 활용하는 데 있어 큰 부담으로 작용한다.

3.2. LLM-based 접근의 한계

최근 이러한 정적분석의 허위경보 문제를 개선하기 위해 LLM을 활용하는 연구에 대한 관심이 증가하고 있다. 특히 LLM의 코드 이해 능력은 정적분석 도구가 놓치는 의미적 정보를 보완할 수 있는 가능성을 보여준다[2].

그러나 정적분석과 같이 코드의 제어 흐름 이해와 의미적 판단을 동시에 요구하는 복잡한 문제를 LLM에게만 의존하여 접근하는 방식[3]은 몇 가지의 분명한 한계를 가지고 있다. 이는 LLM에게 과도한 추론 부담 부과될 경우 불완전하거나 근거가 부족한 결론 생성으로 이어질 수 있으며, 실제로 코드에 존재하지 않는 동작이나 조건 등의 사실을 허위로 만들어내는 환각이 종종 발생하여 결과의 신뢰성을 저하시킨다.

3.2. 정적분석의 실제 허위 경보 사례

본 절에서는 메모리누수 분석의 허위경보 예시 사례를 소개한다. 그림 2의 코드를 보면, source에 해당하는 지점에서 변수 p에 메모리가 동적으로 할당되며, 해당 함수가 종료되는 지점인 sink에 도달할 때까지 할당된 메모리를 해제하거나 외부로 소유권 이전을 위한 할당 연산(=)과 같은 코드가 명시적으로 등장하지 않고 있다. 분석기는 이러한 표면적 제어 흐름만으로 메모리 누수 위험 경보를 발생시켰다.

그러나 5행에서 `sp.reset()` 호출 시 포인터 p가 전달된다. 이는 외부에 존재하는 스마트 포인터 객체로 소유권을 이전하는 동작에 해당한다. 이 경우 메모리는 스마트 포인터 객체의 생명주기에 따라 자동으로 관리되며, 메모리 안전이 보장되는 명백한 sanitizer이다. 이는 분석기의 특정 라이브러리 API 명세 누락으로 실제 존재하는 sanitizer를 놓친 전형적인 허위경보 사례이다.

4. 방법

본 장에서는 정적 오염분석의 허위경보 분류 문제를 실행 경로 단위 문제로 재정의하고, 이를 해결하기 위한 Neuro-symbolic 파이프라인을 제안한다.

4.1. 단일 실행 경로 단위 문제로 분해

오염분석의 본질은 source에서 sink로 이어지는 실제 가능한 오염 전파 경로의 존재 여부를 판단하는 것에 있다. 오염분석의 경보가 발생하는 조건을 아래와 같이 표현할 수 있다.

$$Alarm(s,t) = \bigvee_{p \in \mathcal{P}(s,t)} ValidPath(p)$$

여기서 s는 source, t는 sink 노드이며 p는 실행 가능 경로를 나타낸다. s에서 p로 오염이 도달할 수 있는 실행 경로 p가 단 하나라도 존재한다면 분석기는 경보를 발생시킨다. 이때 경보가 허위경보 인지 확인하려면 다음을 보이면 충분하다.

$$\forall p \in \mathcal{P}(s,t), \neg ValidPath(p)$$

즉, s에서 t로의 $ValidPath(\ )$를 만족하는 오염 도달 가능 실행 경로 p가 하나도 존재하지 않을 경우에만 해당 경보는 허위로 분류된다.

경보의 분류 문제를 실행 경로 단위로 문제를 분해하면, 각 단일 실행 경로상의 오염 도달 가능성이 유효한지를 묻는 국소적 질문을 통해 LLM이 한번에 처리해야 하는 정보의 양을 줄이고, 추론 범위를 명시적으로 제한하여 LLM의 추론 부담을 줄일 수 있다.

그 결과, 경보 분류 문제는 가능한 실행 경로를 계산하는 작업과 경로가 실제로 오염을 유발하는지 판단하는 작업으로 나눌 수 있다.

본 연구는 기존의 정적분석과 LLM을 단순히 결합하는 것이 아닌, 각 기술이 잘 수행할 수 있는 역할을 할당하고, 이를 하나의 분석 파이프라인으로 통합하는 것을 목표로 한다. 제어 흐름 및 도달 가능성 탐색과 같은 구조적 분석은 정적분석 및 MAX-SAT과 같은 symbolic 기법이 담당하고 sanitizer 존재 여부나 API 의미 등 의미 기반의 판단은 LLM이 담당한다.

4.2. 전체 파이프라인 개요

그림 3은 본 연구에서 제안하는 방식의 전체적인 흐름을 나타낸다. 이는 크게 네 가지 단계로 구성된다.

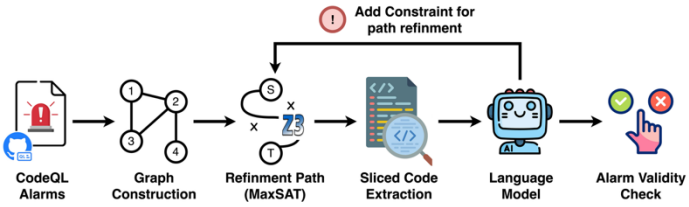


그림 3 파이프라인 개요

(1) **경보 입력 및 제어 흐름 추출** : 분류 대상 경보를 입력 받는다. 각 경보는 Source s의 위치와 Sink t의 위치의 쌍으로 표현된다. 먼저 CodeQL을 이용하여 s를 포함하는 함수 f_s를 식별한다. 이후 함수 f_s의 모든 transitive callee를 계산하여 얻어진 관심 함수 집합 $\mathcal{F}(s)$에 대한 모든 제어 흐름 그래프(CFG) $\{G_{cfg}(f) \mid f \in \mathcal{F}\}$를 추출한다. 이를 통해 전체 프로그램이 아닌 해당 경보와 실제로 연관된 부분 프로그램만을 대상으로 이후 단계를 수행한다. 이때 Super Graph 구성 단계를 위해 각 노드에서 함수 호출이 존재할 경우 추가적인 출력에 Call-chain annotation을 포함하도록 한다.

(2) **Super Graph 구성** : 추출된 각 함수의 CFG $G_{cfg}(f)$에서 반복 구조에 해당하는 cycle edge를 제거하여 방향성 비순환 그래프(DAG) 형태로 변환한다. 변환된 DAG $G_{dag}(f)$들을 함수 호출 관계에 따른 interprocedural edge를 추가하여 하나의 통합된 Super Graph G_{super}를 구성한다. Super Graph G_{super}는 함수 경계를 넘나드는 source-sink의 실행 경로를 단일 그래프상에서 표현하기 위한 기반 구조이다.

(3) **MAX-SAT 기반 최적 경로 선택** : Super Graph G_{super}가 주어졌을 때 source s에서 sink t까지 실행 가능한 최단 경로를 선택하는 문제를 MAX-SAT 기반의 제약 최적화 문제로 모델링한다. 시작, 종료, 흐름 보존과 같은 경로의 구조적 제약은 Hard constraints로 인코딩하고, 최단 실행 경로 선택을 위해서 모든 edge 변수에 대하여 -1의 가중치를 부여는 Soft constraints를 적용한다. MAX-SAT Solver는 모든 Hard constraints를 반드시 만족하면서, Soft constraints의 가중치 합이 최대가 되는, 즉 최단 거리의 source-sink 경로를 반환한다.

(4) **LLM을 통한 경로 정제** : 선택된 단일 실행 경로에 대해, 해당 경로가 실제 실행에서 유효하며, source-sink 간 오염 도달 가능성을 LLM이 판단하도록 한다. LLM이 해당 경로가 실제 문제가 되는 경로로 판단하면 해당 경보를 진짜경보로 분류하고 탐색 과정을 종료한다. 반대로 오염 도달이 불가능한 경로로 판단되면 해당 결론의 근거에 해당하는 부분을 경로에서 배제하는 제약식을 반환한다. 그리고 제약 기반 경로 선택 단계 (3)으로 다시 돌아가 현재까지 누적된 제약을 모두 만족하는 최단 실행 경로를 선택한다. 이러한 과정이 반복될수록 경로 제약은 계속해서 누적되며 더 이상 제약을 만족하는 유효한 경로가 존재하지 않는다면 이는 허위경보로 분류된다.

이러한 **경로 탐색 – LLM 판단 – 경로 제약 누적**의 반복 구

조는 symbolic 기법과 LLM을 하나의 파이프라인으로 통합한 Neuro-symbolic 접근 방식이다. 이를 통해 LLM이 명시적인 단일 실행 경로의 의미적 타당성만 판단하도록 추론 범위를 제한하고, 경로 제약 누적을 통해 경보의 원인 후보가 되는 실행 경로 들을 점진적으로 제거함으로써 허위경보를 분류한다.

4.3. Super Graph 구성

함수 단위의 제어 흐름을 나타내는 CFG는 CodeQL과 같은 정적분석 도구를 사용하면 손쉽게 얻을 수 있지만, 다음과 같은 구조적 한계를 가지고 있다.

• 함수 간 호출 관계를 포함하지 않아 서로 다른 함수 간 제어 흐름을 단일 경로로 표현할 수 없다.

• Loop로 인한 순환이 포함될 수 있다. 이러한 경우 이후 단계에서 MAX-SAT 기반의 경로 문제로 모델링할 수 없다.

따라서 본 연구의 제안 방식에서는 CFG를 그대로 사용하지 않고 다음의 두가지 변환을 통해 단일 경로 탐색에 적합한 Super Graph를 사용한다.

(1) DAG 변환 : 각 관심 함수들의 모든 CFG $\{G_{cfg}(f) \mid f \in \mathcal{F}\}$ 대하여 다음과 같은 변환을 수행한다. loop의 body에서 header로 향하는 cycle edge을 body에서 exit으로 향하도록 재배치를 하여 가능한 경로 공간을 유한하게 만들기 위한 과정이다. 그림 4는 이러한 과정을 시각화 한다.

$$G_{\text{dag}}(f) = \text{RemoveBackEdges}\left(G_{\text{cfg}}(f)\right)$$

(2) Super Graph 구성: CFG 추출 단계에서 부가적으로 얻은 Call-chain annotation 정보를 사용해 함수 호출 관계를 interprocedural edge 형태로 결합한다. 최종 적으로 만들어지는 그래프는 아래와 같이 정의될 수 있다.

$$G_{\text{super}} = \left(\bigcup_{f \in \mathcal{F}(s)} G_{\text{dag}}(f) \right) \cup E_{\text{Inter}}$$

여기서 는 함수 호출 노드에서 피호출 함수의 Entry 노드로 향하는 edge와 피호출 함수의 Exit에서 해당 함수 호출 노드의 successor로 향하는 edge로 구성된다. 이를 통해 Super Graph는 단일 함수 내부 제어흐름 뿐만 아니라 interprocedural한 호출과 복귀 흐름까지 단일 경로로 표현할 수 있다

Algorithm 1은 앞서 설명한 Super Graph를 구성하는 과정을 의사코드로 나타낸 것이다. 2-5행은 CFG를 DAG로 변환하는 단계이며, 6-10행은 모든 함수 호출과 복귀에 대한 inter-

Algorithm 1 BuildSuperGraph()

Global Input CFGs $\{G_{\text{cfg}}(f) \mid f \in \mathcal{F}(s)\}$
Function BUILDSUPERGRAPH()
output Super Graph G_{super}

1: $G_{\text{super}} := \emptyset$
2: **loop** over each function $f \in \mathcal{F}(s)$
3: $G_{\text{dag}}(f) := \text{REMOVEBACKEDGES}(G_{\text{cfg}}(f))$
4: $G_{\text{super}} := G_{\text{super}} \cup G_{\text{dag}}(f)$
5: **end loop**
6: **loop** over each call site c
7: $f := \text{callee}(c)$
8: $c_{\text{succ}} := \text{succ}(c)$
9: $G_{\text{super}} := G_{\text{super}} \setminus \{(c, c_{\text{succ}})\}$
10: $G_{\text{super}} := G_{\text{super}} \cup \{(c, \text{entry}(f))\}$
11: $G_{\text{super}} := G_{\text{super}} \cup \{(\text{exit}(f),\ c_{\text{succ}}\}$
12: **end loop**
13: **return** G_{super}

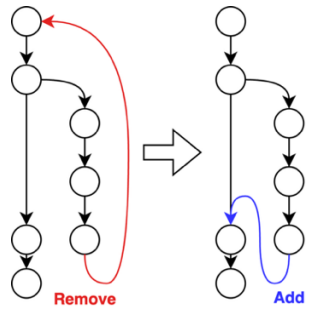


그림 4 CFG-DAG 변환

procedural edge E_{Inter}를 추가하는 단계이다. 이 과정을 통해 MAX-SAT 기반의 경로 문제에 적합한 단일 Super Graph G_{super}가 완성된다.

4.4. MAX-SAT 기반 최적 경로 선택

Super Graph G_{super}가 주어졌을 때, 본 연구의 핵심 아이디어의 첫번째 단계는 경로 선택 문제를 Weighted Partial MAX-SAT (WPMS) 최적화 문제로 정의하는 것이다. WPMS은 제약조건의 성격에 따라 Hard constraints와 Soft constraints로 구분하며, Hard constraints를 모두 만족하면서 동시에 Soft constraints를 가중치의 합을 최대화 하는 해를 찾는 문제이다. 경로 선택 문제에서 Hard constraints는 경로의 구조적 제약으로 인코딩 되고, Soft constraints는 경로 길이 최소화를 위한 비용 함수로 인코딩 되며, 문제의 초기 인코딩은 다음과 같다.

Edge 선택 변수: Super Graph의 각 edge $(u, v) \in E$에 대해서 다음과 같은 Boolean 변수를 도입한다.

$$x_{u,v} = \begin{cases} True & if\ (u,v)\ is\ selected \\ False & otherwise \end{cases}$$

Hard constraints (초기 경로 제약) :

• **시작과 종료 제약** : 선택 되는 경로는 반드시 다음과 같은

시작/종료에 대한 초기 제약을 만족해야 하며, 아래 두가지 제약은 선택되는 경로가 source s에서 시작하여 sink t에 도달하는 경로의 형태를 갖도록 보장한다. 각각 source s의 outgoing edge 중 적어도 하나는 반드시 선택, sink t의 incoming edge 중 적어도 하나는 반드시 선택되어야 함을 나타낸다.

$$\Phi_{\text{start}}(s) = \bigvee_{v \in \text{Out}(s)} x_{s,v}, \quad \Phi_{\text{end}}(t) = \bigvee_{u \in \text{In}(t)} x_{u,t}$$

• **노드의 연속성 제약** : 경로 상에 위치한 노드는 항상 흐름의 연속성이 유지되어야 한다. 어떤 노드 n의 incoming edge가 선택되었다면 반드시 n의 outgoing edge중 하나가 선택되어야 한다

$$\Phi_{\text{flow}}(n) = \left(\bigvee_{p \in \text{In}(n)} x_{p,n} \right) \Rightarrow \text{ExactlyOne}\big(\{ x_{n,q} \mid q \in \text{Out}(n) \} \big)$$

• 따라서 초기 Hard constraints는 다음과 같다.

$$\Phi_{\text{hard}} = \Phi_{\text{start}}(s) \wedge \Phi_{\text{end}}(t) \wedge \bigwedge_{n \in \text{Nodes}} \Phi_{\text{flow}}(n)$$

Soft constraints (최단 경로 목적 함수) :

• 모든 edge 변수 x에 대해 -1의 가중치가 부여된다.

$$\big(x_{u,v}, w_{u,v}\big) \quad \text{where} \quad w_{u,v} = -1$$

• 목적 함수는 다음과 같다. 모든 가중치는 -1 이므로 W가 최대화가 되려면 가장 적은 수의 edge를 선택해야 한다. 따라서 이것은 의미적으로 MIN-SAT과 동일하다.

$$W = \sum_{(u,v) \in E} w(u,v) \cdot x_{u,v}$$

따라서 해당 MAX-SAT의 해는 가장 짧은 source-sink 경로를 선택하는 결과를 갖는다.

4.5. LLM을 통한 경로 정제

Algorithm 2는 Super Graph 생성부터 경로 선택과 LLM의 경로 정제 과정을 통해 경보를 분류하는 전체 과정을 의사코드로 나타낸 것이다.

1행에서 경로 탐색 공간이 되는 Super Graph G_{super}를 구성한다. 이후 3-8행 loop의 4행의 FindMinSAT( )은 $\Phi_{\text{hard}} \wedge \varphi$ 제약을 만족하는 최단 실행 경로를 선택한다. 이 때 Φ_{hard}는 불변 제약식으로 FindMinSAT( ) 연산 내부에서 항상 적용되며, loop가 반복할 동안 변경되지 않으므로, 명시적 표기를 생략한다. 6행에서는 선택된 경로 p의 타당성을 LLM에게 묻는다. 이때 LLM은 둘 중 하나의 판단을 내린다.

Algorithm 2 ClassifyAlarm()

Global Input Alarm (s,t), CFGs $\{G_{\text{cfg}}(f) \mid f \in \mathcal{F}(s)\}$, LLM
Function ClassifyAlarm()
output **TRUE**: true alarm, **FALSE**: false alarm

1: $G_{\text{super}} :=$ BuildSuperGraph($\{G_{\text{cfg}}(f)\}$)
2: $\varphi :=$ true
3: **loop**
4: $p :=$ FindMinSAT(G_{super}, s, t, φ)
5: **if** $p =$ UNSAT **then return FALSE**
6: $\delta :=$ LLM(p)
7: **if** $\delta =$ true **then return TRUE**
8: $\varphi := \varphi \wedge \delta$
9: **end loop**

(1) **해당 경로는 의미적으로 타당하며, 오염이 sink까지 도달 :** 경로 탐색을 중단하고 해당 경보를 진짜경보로 판단.

(2) **해당 경로는 실행될 수 없거나, sanitizer가 존재 :** 해당 경로가 유효하지 않은 근거를 기반으로 경로를 배제하는 경로 제약을 구성한다. 그리고 제약 집합에 새로운 제약을 추가한 뒤 다시 경로 선택 단계로 돌아간다. 해당 loop가 반복될수록 LLM에 의해 경로 제약 φ가 누적되며 탐색 범위를 좁힌다.

Algorithm 2는 아래 두 가지 종료 조건을 가지며 각각의 의미는 다음과 같다.

(1) **LLM이 True를 반환**: 이는 의미적으로 타당한 source-sink 간 오염이 도달 가능한 실행 경로 존재함을 의미하며, 해당 경보는 진짜경보이다.

(2) **경로 선택에서 UNSAT 반환**: 이는 의미적으로 타당한 source-sink 간 오염이 도달 가능한 실행 경로가 존재하지 않음을 의미하며, 해당 경보는 허위경보이다.

4.6. LLM 경로 정제 및 허위경보 분류 예시

해당 절에서는 본 연구의 제안 방식의 정적 오염분석 경보 분류 과정이 어떻게 동작하는지 간단한 예시를 가지고 설명한다. 그림 5는 3장에서 예시로 언급했던 정적분석의 허위경보 사례의 코드를 Super Graph로 구성한 모습을 시각적으로 보여준다. 각 노드에는 실제 코드와 대응하는 행 번호가 표기되어 있다. 그림 6을 통해 단계적으로 동작 과정을 설명할 수 있다.

(1) 최초 반복(iteration)에서는 추가 제약 없이 Super Graph 상의 최단 경로를 선택한다. 이 경로는 `if(p != null)`의 조건을 만족하지 않는 분기를 통해 sink로 도달하는 경로이며, 이는 `new Mem(100)`이 null을 반환해야만 실행 가능한 경로이다. LLM은 이 경로가 의미적으로 오염이 발생하지 않아 오염 도달 가능성을 판단할 필요가 없으므로 유효하지 않은 경로로 판단한다. 따라서 아래와 같은 경로 배제 제약이 추가되며,

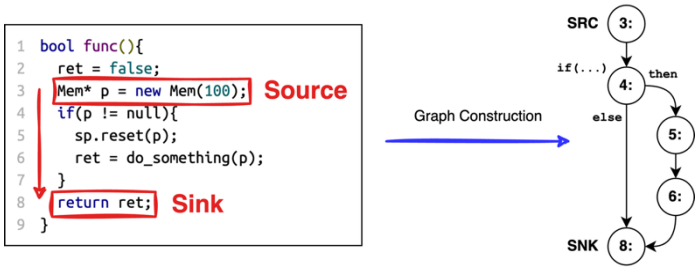


그림 5 Super Graph 변환 예시

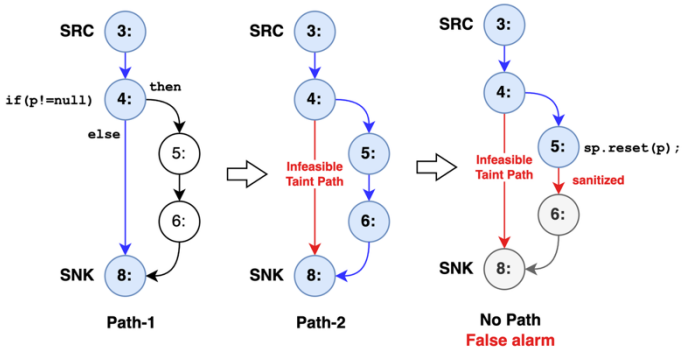


그림 6 경로 정제 예시

이후의 경로 선택 과정에서 2:→7: 방향의 edge는 고려되지 않는다.

(2) 추가된 제약이 반영된 상태에서 두 번째 경로를 선택하면, 이번에는 `if(p != null)` 조건이 참인 실행 경로가 선택되며 `sp.reset(p)`가 등장한다. LLM은 스마트 포인터를 통한 소유권 이전을 sanitizer로 해석하고 해당 경로상 오염 전파가 중단되는 것으로 판단한다. 따라서 아래와 같은 경로 제약 추가

(3) 두 제약이 모두 반영된 상태에서 다시 경로 선택을 시도하면 더 이상 source에서 sink로 이어지는 누적된 제약을 만족하는 유효한 경로가 존재하지 않는다. 즉, MAX-SAT의 관점에서는 경로 선택 문제가 UNSAT으로 판정되어, 해당 경보는 허위경보로 분류된다.

5. Baseline

본 연구는 제안 방식의 효과를 평가하기 위해, LLM만을 활용하여 정적분석 경보를 분류하는 방식을 Baseline으로 사용한다. Baseline은 단순히 LLM의 코드 이해 및 추론 능력에만 의존하여 판단을 내리는 LLM-only 접근이다.

Baseline을 별도로 설계하는 목적은 크게 두 가지이다. 첫째, 실제 소프트웨어 개발 환경에서는 복잡한 프롬프트 엔지니어링, 모델 미세조정 또는 내부 파라미터 수정 없이, 사전 학습된 LLM과 기존의 정적분석 도구를 그대로 활용하여 문제를 해결하려는 사용 시나리오가 일반적이다. 개발자는 정적분석 도구가 보고한 경보를 확인하고, 해당 경보 위치의 코드 또는 일부 관련 문맥을 LLM에게 제시한 뒤 조언을 얻는 방식으로

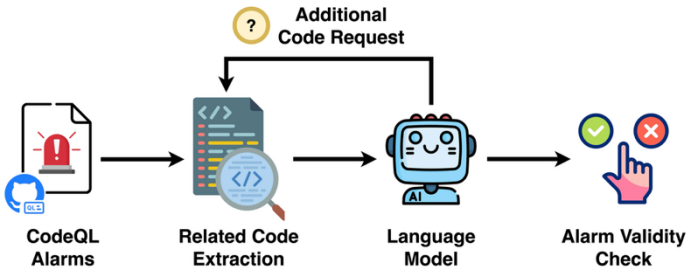


그림 7 Basline 개요

활용할 수 있다. 이러한 환경을 고려할 때, 해당 Baseline은 실무적으로도 그럴듯한 사용 형태를 반영한다. 둘째, Baseline은 LLM이 단독으로 정적분석의 허위경보 문제를 얼마나 해결할 수 있는지를 확인하고, 그 현실적 한계를 평가하기 위한 비교 기준의 역할을 수행한다. 이는 본 연구에서 제안 방식의 구조적 이점을 비교하는 데 중요한 기준이 된다.

5.2. Baseline의 설계

그림 7은 Baseline의 동작 과정 시각화한다.

(1) **초기 질의:** 정적분석 도구(CodeQL)가 생성한 경보의 source-sink 위치를 포함하는 코드 조각을 LLM에게 제공하고, 해당 경보가 실제 발생 가능한지 여부를 판단하도록 요청한다.

(2) **추가 정보 요청:** LLM은 판단을 보류하고, 해당 결론을 내리는 데 필요한 정보(예: 특정 함수 정의, 변수 선언 등)를 명시적으로 요구할 수 있다.

(3) **추가 문맥 제공:** 사용자는 LLM이 요구한 추가 코드 조각을 제공하여 문맥을 확장한다. 이 과정은 LLM의 판단이 가능하다고 판단될 때까지 반복될 수 있다.

(4) **최종 판단 :** LLM이 충분한 문맥 정보를 확보했다고 판단하는 경우, 경보의 진위 여부에 대한 True/False 분류 결과를 반환한다.

6. 실험

본 장에서는 제안 방식에 대하여 아래 세 가지 연구 질문을 설정하고, 실험을 통해 이를 평가한다.

RQ1. 제안 방식은 허위경보와 진짜경보를 얼마나 잘 분류할 수 있는가?
RQ2. 제안 방식은 진짜경보를 놓치지 않고 안전하게 보존할 수 있는가?
RQ3. 최종적으로 검토해야 하는 경보의 수를 얼마나 줄일 수 있는가?

실험은 Intel Xeon Silver 4210R 2.4GHz CPU, 128GB RAM, 4 NVidia RTX A5000 GPUs 사양의 workstation에서 진행했다.

Category	Target	True	False
CWE-78	cadaver	1	0
CWE-78	libmjpegutils-2.1-0	2	0
CWE-78	libss2	2	0
CWE-78	ncurses-bin	0	1
CWE-120	dcraw	0	1
CWE-120	drawxtl	0	3
CWE-120	elvis-tiny	2	1
CWE-120	gap-guava	1	2
CWE-120	gnuplot	0	1
CWE-120	kbd	0	1
CWE-120	le	0	1
CWE-120	libaudio2	0	1
CWE-120	libdebconfclient0	0	3
CWE-120	maildrop	0	6
CWE-120	mlock	1	0
CWE-120	mpg321	0	2
CWE-120	scm	1	0
CWE-120	sc	2	1
CWE-120	scm	1	0
CWE-120	whois	0	2
CWE-120	zip	0	8
MemoryLeak	FastDDS	0	97
Total(22)		**13**	**131**

표 1 오염분석 경보 벤치마크

Metric	Baseline	Ours
Accuracy	(72.57%, 19.45)	(82.36%, 6.90)
Recall	(79.04%, 88.91)	(87.69%, 17.75)
Precision	(20.95%, 20.93)	(33.95%, 15.46)

표 2 실험 결과

4.1. 벤치마크

C/C++ 코드로 작성된 실제 프로그램 22개를 대상으로 CodeQL 기반 정적 오염분석을 수행하여 얻은 결과를 대상으로 한다.(표1 참고) 이 중에는 약 100만 라인 규모의 대규모 산업용 코드베이스 FastDDS가 포함된다. 벤치마크는 Buffer overflow(CWE-120)와 OS command injection(CWE-78) 그리고 메모리 누수 관련 사례를 포함한다. 총 144개의 오염분석 경보가 수집되었고, 이는 문제를 일으킬 수 있는 진짜경보 13개와 허위경보 131개 로 구성된다.

4.1. 실험 세팅

본 실험에서는 제안 방식과 Baseline은 모두 사람이 개입하지 않는 자동화된 파이프라인으로 구성되었으며, OpenAI gpt-5-mini API를 사용하여 벤치마크 세트에 대하여 5회 반복 평가를 수행했다. Baseline의 경우 종료 보장을 위해 정보 요청 가능 횟수는 10번 이하로 제한하여 진행했다. 각 평가에서 아래 세 가지 핵심 지표에 대하여 측정했다.

- **Accuracy:** 전체 경보를 올바르게 분류한 비율
- **Recall:** 진짜경보를 놓치지 않고 식별한 비율
- **Precision:** 진짜라고 분류한 결과 중 실제 진짜경보의 비율

Accuracy는 전체 경보 중 올바르게 분류된 비율을 나타내며, 허위경보 제거와 진짜경보 보존이라는 두 측면을 모두 고려한 전반적인 분류 성능을 의미한다. Recall은 허위경보 제거 과정에서 진짜경보가 함께 제거되는 False negative 상황을 방지할 수 있는지를 나타내므로, Recall이 높을수록 더 안전한(safe) 분류가 이루어졌음을 의미한다. Precision은 허위경보 제거 이후 남겨진 경보 집합 내부의 품질을 나타내며, Precision이 높을수록 실무자가 검토해야 하는 경보 집합이 더 신뢰할 수 있음(trustworthy)을 의미한다.

4.1. 실험 결과

종합적인 결과 비교는 표 2에 정리되어 있으며, 본 절에서는 RQ1-RQ3에 따른 분석 결과를 제시한다.

RQ1. 제안 방식은 허위경보와 진짜경보를 얼마나 잘 분류할 수 있는가? Accuracy 기준 분석 결과, 제안 방식은 Baseline 대비 더 높은 분류 성능을 보였다. Baseline의 평균 정확도는 72.57%이며 분산이 19.45로 높은 변동성을 나타냈다. 반면, 제안 방식은 평균 정확도 82.36%와 분산 6.90을 기록하여 성능과 안정성 모두 향상된 결과를 확인하였다. 이는 본 연구의 제안 방식이 허위경보와 진짜경보를 보다 정확하고 일관되게 구분할 수 있음을 보여준다.

RQ2. 제안 방식은 진짜경보를 놓치지 않고 안전하게 보존할 수 있는가? Recall은 진짜경보가 허위경보로 분류되어 제거되는 False negative 상황을 방지할 수 있는지를 평가하는 지표이다. 실험 결과, Baseline의 평균 Recall은 79.04%였으며 분산은 88.91로 불안정한 성향을 보였다. 반면 제안 방식은 평균 Recall 87.69%와 분산 17.75를 기록하여 진짜경보를 보존하는 측면에서 보다 안전한(Safe) 성능을 보여주었다. 즉, 본 연구의 제안 방식은 허위경보를 제거하는 과정에서 실제 취약점을 함께 제거하는 위험을 감소시키는 효과를 가진다.

RQ3. 최종적으로 개발자가 검토해야 하는 경보의 수를 얼마나 줄일 수 있는가? Precision은 허위경보 제거 이후 남겨진 경보 집합의 품질을 측정하는 지표이며, Precision이 높을수록 최종 검토 대상 경보 집합이 더 신뢰할 수 있음을 나타낸다. Baseline의 평균 Precision은 20.95%였으며 분산은 20.93으로 낮은 품질과 높은 변동성을 나타냈다. 반면, 제안 방식은 평균 Precision 33.95%와 분산 15.46을 기록하여 허위경보 제거로 인한 검토 효율 향상 측면에서 더 실용적인 결과를 보여주었다. 이는 본 연구의 제안 방식이 개발자의 검토 부담을 줄이는 데 기여함을 의미한다.

7. 논의

7.1. Baseline에서만 관찰된 문제들

실험 중 Baseline 방식에서 반복적으로 관찰된 문제점들이 존재했으며, 제안 방식에서는 동일한 문제가 나타나지 않았다. 주요 관찰 내용은 다음과 같다.

(1) **환각에 의한 실행 경로 오판:** Baseline은 전체 코드 조각을 기반으로 실행 가능성을 추론해야 하는데, 이때 제어 흐름

을 정확히 모델링하지 못해 실제로는 도달 불가능한 경로를 가능한 것으로 판단하는 사례가 발생하였다. 반면 제안 방식은 SAT 기반 경로 계산을 통해 실제 실행 가능한 경로만을 고려하므로 이러한 오판 가능성이 구조적으로 감소한다.

(2) **존재하지 않는 함수 또는 변수의 정의를 요청하는 경우:** Baseline은 필요한 맥락을 스스로 수집해야 하므로, 실제로 입력 경로에 포함되지 않는 함수나 정의를 요청하는 경우가 관찰되었다. 제안 방식은 Super Graph를 기반으로 경로 상 호출되는 모든 함수 정의를 포함하여 입력을 구성하므로, 불필요한 정보 요청이 발생하지 않았다.

(3) **정보 요청 누적으로 인한 컨텍스트 윈도우 초과:** Baseline은 함수 정의, 변수 선언 등 추가 정보 요청이 누적되면서 토큰 사용량이 증가하였다. 호출 깊이가 깊거나 코드가 길어질수록 입력이 컨텍스트 윈도우를 초과하는 사례가 관찰되었으며, 이 경우 답변이 중단되어 결론에 도달하지 못하였다. 반면 제안 방식은 단일 경로 단위로 판단이 이루어지며, 최단 실행 경로 기반의 코드 슬라이싱을 통해 입력 코드양이 비교적 작고 일정하게 유지되는 경향을 보였다.

7.2 추론 범위 제한의 효과

두 방식의 가장 큰 차이점은 LLM이 감당해야 할 추론 범위이다. Baseline은 다양한 실행 경로 가능성과 상태 변화를 스스로 탐색해야 하며, 필요시 추가 정보를 요청하는 방식으로 판단을 진행한다. 즉, 제어 흐름 분석과 의미적 판단까지 포함하는 전역적 추론이 요구된다.

제안 방식에서는 LLM이 수행해야 할 역할이 명확히 제한된다. symbolic 기법이 실행 경로 계산을 담당하며, LLM은 주어진 경로가 실제 실행 의미에서 유효한지 여부를 판단하는 국소적 추론만 수행한다. 이러한 역할 분리는 LLM의 추론 부담을 줄이고, 판단의 변동성과 오류 발생 가능성을 낮추는 방향으로 작용하였다.

본 실험의 결과는 제안 방식의 개선 효과가 LLM 자체의 성능 향상에서 비롯된 것이 아니라, 구조적 제약과 입력 전략을 통해 LLM의 추론 범위를 줄인 결과로 해석할 수 있다. 특히 대규모 코드베이스를 대상으로 하는 정적분석 시나리오에서, LLM이 전역적 추론을 수행하도록 요구하는 방식은 비효율적일 뿐만 아니라 컨텍스트 윈도우 한계에 취약할 수 있다. 반면 제안 방식은 입력 크기를 제어하고 실행 의미 중심의 판단을 유도한다는 점에서 실무적 적용 가능성이 있다.

그러나 본 연구는 다음과 같은 한계를 지닌다.

데이터의 독립성 미검증: 본 연구에서 사용된 벤치마크는 오픈소스 소프트웨어로 구성된다. 그렇기 때문에 사전 학습된 LLM이 동일하거나 유사한 코드 패턴을 이미 학습했을 가능성을 완전히 배제하기 어렵다. 이는 LLM-based 접근 방식이 특정 코드베이스에서 과대평가될 가능성을 시사한다. 더욱 엄밀한 평가를 위해, 향후 학습 및 평가 데이터의 독립성 검증이 필요하다. 소스코드가 공개되지 않은 코드베이스를 활용하는 방식이 대안이 될 수 있다.

모델 종속성: 본 연구에서는 gpt-5-mini 모델 기반으로 실험이 진행되었다. 제안 방식의 성능이 특정 모델에 의존하는지 여부를 평가하기 위해, 향후 다양한 모델에 대한 실험 및 추가 검증이 요구된다.

LLM의 판단 근거 검증의 부재: 제안 방식은 LLM이 반환한 판단 근거에 대한 논리적 타당성을 기계적으로 검증하지 않으며, 이는 결과의 정확성이 LLM의 판단 품질에 영향을 받음을 의미한다. 향후 타당성 검증 모듈과의 결합이 가능하다.

8. 관련연구

정적분석의 허위경보 줄이기 위한 연구는 다양한 방향 진행되어 왔다.

SHOVEL[4]은 콜그래프 기반의 실행 경로를 MAX-SAT 기반 제약식으로 모델링하고, 경로 제약을 반복적으로 누적하며 경로를 정제하는 방식으로 허위경보를 효과적으로 제거하는 접근을 제안하였다. 그러나 경로가 의미적으로 타당한가의 여부(예: sanitizer 존재 여부)는 사람이 직접 판단해야 하므로, 제약식 계산과 의미적 판단이 분리되어 있다. 본 연구는 SHOVEL의 MAX-SAT 기반 경로 정제 방식의 아이디어를 참고하여, 의미적 타당성 판단을 LLM으로 자동화하고 분석 단위를 콜그래프에서 Super Graph 기반 제어흐름으로 확장한다는 점에서 차별성을 가진다.

ZeroFalse[5]와 BugLens[6] LLM을 정적분석 경보의 진위여부 판단자로 활용하는 접근이다.

ZeroFalse는 Java 프로그램에서 발생하는 오염분석 허위경보 분류를 목적으로 한다.

해당 dataflow trace 정보를 이용해 인접한 두 dataflow step이 동일 함수 내에 있으면 그 사이 구간을 전부 포함하는 방식으로 문맥을 근사적으로 복원한다. 그렇게 만들어진 경보 관련 코드 조각들을 LLM의 입력으로 삼아 해당 경보의 진위여부를 판별한다. 이때 LLM은 경보 단위에 포함되는 모든 경로를 한 번에 추론하는 방식이다.

ZeroFalse는 flow-sensitive dataflow에 기반한 오염 분석을 대상으로 하며, 해당 기법을 적용하려면 분석 결과로 source-sink에 대한 dataflow trace 정보를 제공할 수 있어야 한다.

BugLens는 Linux Kernel의 메모리 안전 관련 보안 취약점을 대상으로 하는 분석의 경보 중 실제 crash가 발생할 수 있는 메모리 취약점의 실현 가능성(feasibility)을 판단하여 허위 경보를 제거한다.

해당 연구에서 LLM은 두가지 핵심 기능을 SecIA(Security Impact Assessor)와 ConA(Constraint Assessor)라는 역할로 분할하여 수행한다. 먼저 SecIA는 해당 경보의 보안적 영향

(security impact)의 존재 가능성을 visibility 관점에서 판단하여, 보안적으로 의미 있는 경보만을 후속 단계의 분류 대상으로 선별한다. 이후 ConA에서는 선별된 경보에 대한 source-sink 경로의 도달 가능성을 평가한 뒤, 경로 제약을 수집 및 해석하여 해당 경로가 실제로 악용 가능한지(exploit feasibility)를 단계적으로 추론한다. 다만 해당 연구의 경로 제약은 형식적 제약(formal constraints)으로 인코딩 되거나 Solver를 사용해 검증되는 것이 아닌, 제약의 수집 및 해석이 LLM에 의해 수행된다는 점이 특징이다.

BugLens는 source-sink 외 trace 수준의 세부 실행 정보는 요구하지 않지만, 해당 방법은 단순 버그가 아닌 Security impact가 존재하는 취약점에 대해서만 한정적으로 적용 가능하다.

BugLens나 ZeroFalse같은 입력 프롬프트에 의존하는 LLM-based 접근에서는 컨텍스트 윈도우 초과 및 실행 경로 오판 등과 같은 문제에 대하여 근본적으로 취약할 수 있다.

LLM4PFA[7]는 LLM과 SMT 기반의 symbolic 기법을 결합한 Neuro-symbolic 접근을 통해 C/C++을 대상으로 한 메모리 안전 관련 분석의 허위경보 분류를 목적으로 한다.

해당 연구는 source-sink 구간의 call trace상 각 함수에서 sink 도달에 영향을 주는 제약 조건을 수집 및 평가하는 과정을 반복하여, 해당 경보의 실현 가능성을 평가한다.

이 과정에서 LLM의 역할은 실행에 영향을 주는 변수 및 함수 반환 값의 범위 등을 실행 제약 형태로 구성하는 것이다. LLM4PFA는 symbolic execution과 같이 모든 실행 경로 고려하는 것이 아닌, call trace가 이어지기 위해 필요한 최소 제약만을 수집하고 검증하는 전략을 취한다. 구성된 제약은 SMT 제약으로 인코딩 되어 Solver로 검증되며, 최종적으로 call trace 상 단 하나의 함수라도 UNSAT일 경우 허위경보가 된다.

LLM4PFA는 SAT/SMT 기반 제약을 반복적으로 누적하고 이를LLM과 결합한다는 점에서 우리 연구와 유사한 점이 존재하지만, 대상이 메모리 안전 분석으로 한정된다. 또한 해당 기법을 적용하기 위해서는 분석 결과로 source-sink에 대한 call trace 정보를 제공할 수 있어야 한다.

위에서 언급한 연구들은 요구되는 정보의 형식, 대상 언어, 분석 유형, 적용 범위와 같은 측면에서 각자 다른 전제 조건과 목표를 가진다.

또한 본 연구에서는 보안 취약점 중심의 엄격한 오염분석으로 정의하기보다는, source-sink 간 정보 흐름이 구조적으로 표현될 수 있는 넓은 의미의 taint-style 분석을 대상으로 한다. 즉, dataflow 기반 오염 분석(ZeroFalse), security impact가 존재하는 메모리 안전 취약점(BugLens), C/C++ 메모리 안전 value-flow 분석(LLM4PFA)에서 다루는 범주를 모두 포함하는 보다 일반화된 형태의 정적 분석을 전제로 하며, 본 논문의 벤치마크 역시 이러한 포괄적 정의에 따라 구성된다.

9. 결론

본 연구는 정적 오염분석에서 발생하는 허위경보 판단 문제를 실행 경로 단위로 분해하여, LLM이 사전에 계산된 단일 실행 경로만을 고려하도록 한다. 이를 통해 LLM의 추론 범위를 구조적으로 축소하고, 각 경로에 대한 판단을 제약으로 누적하는 반복 구조를 통해 경로를 점진적으로 정제하는 Neuro-symbolic 접근을 제안한다. 또한 실험을 통해 단순 질의 기반의 LLM-only 접근 대비 분류 정확도, 진짜경보 보존, 최종 경보 집합 품질 측면에서 일관된 개선을 입증했다.

감사의 말

본 연구는 한국연구재단(NRF)의 지원을 받아 수행되었으며, 과학기술정보통신부(MSIT)의 재원으로 수행된 NRF 연구과제(No. RS-2021-NR060080)의 지원을 받았다. 또한 본 연구는 과학기술정보통신부(MSIT)의 재원으로 정보통신기획평가원(IITP)이 지원하는 연구과제(Nos. 2022-0-00995, RS-2024-00341722, RS-2024-00423071)의 지원을 받아 수행되었다.

참고 문헌

[1] GitHub, “CodeQL: The libraries and queries that power security analysis,” Available: https://codeql.github.com/, Accessed: Jan. 12, 2026.

[2] H. Jelodar, M. Meymani, and R. Razavi-Far, “Large Language Models (LLMs) for Source Code Analysis: Applications, Models and Datasets,” arXiv preprint arXiv:2503.17502, 2025.

[3] W. Ma, S. Liu, Z. Lin, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, L. Li, and Y. Liu, “LMs: Understanding Code Syntax and Semantics for Code Analysis,” arXiv preprint arXiv:2305.12138, 2024.

[4] J. G. Kim, W. Lee, J. Choi, C. K. Hur, and K. Yi, “SHOVEL: A SAT-based Tool for Information Flow Alarm Classification,” Draft manuscript, Available: https://sf.snu.ac.kr/publications/shovel.pdf, 2014.

[5] M. Iranmanesh, S. Moradi Sabet, S. Marefat, A. Javidi Ghasr, A. Wilson, I. Sharafaldin, and M. A. Tayebi, “ZeroFalse: Improving Precision in Static Analysis with LLMs,” arXiv preprint arXiv:2510.02534, 2025.

[6] Y. Yan, Y. Chen, Z. Wang, J. Sun, Z. Zhao, C. Zhang, Z. Zhang, Z. Qian, N. Abu-Ghazaleh, and K. Chen, “Towards More Accurate Static Analysis for Taint-Style Bug Detection in Linux Kernel,” In Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE ’25), 2025.

[7] X. Du, K. Yu, C. Wang, Y. Zou, W. Deng, Z. Ou, X. Peng, L. Zhang, and Y. Lou, “Minimizing False Positives in Static Bug Detection via LLM-Enhanced Path Feasibility Analysis,” arXiv preprint arXiv:2506.10322, 2025

LLM 기반 자동 프로그램 수정을 위한 코드 그래프 활용 방식 비교

강신엽°, 이지광, 권혁민, 남재창
한동대학교 전산전자공학부
yeob@handong.ac.kr, lucas0606@handong.ac.kr, hyeokkiyaa@gmail.com, jcnam@handong.edu

A Survey on Leveraging Code Graphs for LLM-based Automated Program Repair

Shinyeob Kang, Jikwang Lee, Hyeokmin Kwon, Jaechang Nam
The School of Computer Science and Electrical Engineering, Handong Global University

요 약

현대 소프트웨어 시스템의 복잡도 증가에 따른 디버깅 부담을 완화하기 위해 자동 프로그램 수정(Automated Program Repair, APR) 기술이 활발히 연구되고 있다. 최근에는 대규모 언어 모델(LLM)의 성능 향상에 따라 LLM을 활용하는 기법들이 APR 분야의 주요 연구방향으로 자리 잡았다. 초기 연구들은 코드를 단순 텍스트 형태로만 활용했으나, 최신 연구들은 코드 그래프(Code Graph)를 통해 코드의 구조적 정보를 활용하는 방향으로 발전하고 있다. 본 연구는 코드 그래프를 활용하는 LLM 기반 APR 연구 12편을 체계적으로 정리하고, 향후 코드 그래프를 활용하는 APR 연구가 나아가야 할 연구 방향을 제안한다.

1. 서 론

현대 소프트웨어 시스템의 규모와 복잡성이 증가함에 따라 버그 발생과 디버깅 비용이 증가하고 있으며, 이러한 개발 부담을 줄이기 위한 기술로 자동 프로그램 수정(Automated Program Repair, APR) 기술이 활발히 연구되고 있다 [1]. 최근 대규모 언어 모델 (LLM)의 코드 이해 및 생성 능력이 향상됨에 따라, LLM을 활용하는 APR 기법들이 전통적인 접근법을 능가하는 성능을 달성하며 APR 분야의 주요 연구 방향으로 자리 잡았다[5].

초기 LLM 기반 APR 기법은 코드를 텍스트 형태로 LLM에 입력하였으나, 이러한 방식은 프로그래밍 언어가 가지는 실행 경로, 데이터 흐름, 호출 관계 등의 구조적인 특성과 Semantic 정보를 온전히 반영하지 못하였다. 이를 극복하기 위해 최근 연구에서는 코드 그래프(Code Graph)를 활용하여 LLM의 코드 이해도를 높이는 접근법이 활발히 연구되고 있다. 그러나 LLM은 기본적으로 시퀀스 형태의 텍스트를 입력으로 받기 때문에, 코드 그래프 정보를 LLM에 효과적으로 전달하는 방법이 핵심 과제로 남아있다. 기존 연구들은 (1) 코드 그래프를 텍스트 형식으로 변환하거나, (2) GNN(Graph Neural Network)과 같은 별도의 모듈로 인코딩하여 LLM과 통합하는 접근을 사용했다.

본 연구는 지금까지 제안된 코드 그래프를 활용하는 LLM 기반 APR 연구 12편을 코드 그래프를 활용하는 방식에 따라 체계적으로 정리하고, 각 접근법의 장점과 한계를 분석하여 향후 코드 그래프를 활용하는 APR 연구가 나아가야 할 연구 방향을 제안한다.

2. 연구 방법

본 연구는 LLM 기반 APR 연구에서 코드 그래프를 활용하는 방식을 정리하는 것을 목표로 한다. 2023년 이후 출간된 연구를 대상으로, Google Scholar에서 “APR”, “Code Graph”, “LLM” 관련 키워드로 검색하고 스노우볼 기법을 통해 확장하여 총 12편의 연구를 수집하였다. 각 연구의 제목, 요약, 서론을 검토하여 다음 기준을 만족하는 연구를 선정하였다.

- LLM 기반 APR과 코드 그래프를 모두 다루는 논문
- 코드 그래프를 활용하는 APR 기법을 제안한 논문
- LLM의 코드 그래프 이해도를 높이는 기법을 제안하고 프로그램 수정 성능을 평가한 논문

코드 그래프 정보를 활용하는 다양한 방식을 비교하기 위해 전통적인 코드 그래프 외에 코드 정보를 그래프 형식의 데이터로 변환하여 사용하는 연구도 조사 범위에 포함하였다. 코드 정보 외의 외부 지식을 주로 포함하는 그래프를 활용하는 연구는 제외하였다.

3. 코드 그래프 활용 방식의 분류 및 분석

본 장에서는 코드 그래프를 활용하는 12편의 LLM 기반 APR 연구를 그래프 정보가 LLM과 통합되는 방식에 따라 네 가지 유형으로 분류한다. 그래프를 텍스트로 변환하는 방식은 (1) 그래프 쿼리 기반 검색, (2) 에이전트 주도 탐색, (3) 의존성 인식 동기화로 세분화되며, 그래프를 직접 통합하는 방식은 (4) 그래프 임베딩을 통한 통합으로 구현된다. 본 연구에서는 LLM과 코드 그래프의 결합 밀도를 기준으로 느슨한 결합부터 긴밀한 결합 순서로 이들을 정리하였다.

3.1 그래프 쿼리 기반 검색

그래프 쿼리 기반 검색 방식은 코드 그래프를 구조화된 지식 저장소로 활용한다. 미리 정의된 쿼리 메커니즘을 통해 문제 해결에 필요한 코드 정보를 검색하고, 텍스트 형태로 LLM에 제공한다. CodexGraph[2]는 정적 분석을 통해 코드 저장소를 Neo4j 그래프 데이터베이스로 구축하고, 에이전트가 Cypher 쿼리를 생성하여 다중 홉 순회를 통해 이슈 관련 코드를 검색한다. RepoGraph[3]는 저장소 수준의 개체 간 의존 관계를 그래프로 구축하고, 결함 위치를 중심으로 k-hop ego-graph를 추출한다. 추출된 하위 그래프는 두 가지 형태의 텍스트로 변환되어 기존 APR 프레임워크에 플러그인으로 통합될 수 있도록 설계되었다.

이 방식은 코드 그래프가 가진 구조 및 Semantic 정보를 제공하므로, 단순 텍스트 검색보다 정확한 컨텍스트를 제공하고, 반복적인 도구 호출을 줄여 탐색 비용을 절감한다. 그러나 검색 쿼리의 정확도에 따라 정보 누락이 발생할 수 있으며, 그래프를 텍스트로 변환하는 과정에서 구조 정보가 손실되거나 LLM의 해석 오류가 발생할 위험이 존재한다.

3.2 에이전트 주도 그래프 탐색

에이전트 주도 그래프 탐색 방식은 코드 그래프를 동적 탐색 환경으로 활용한다. 정적 쿼리 기반 검색과 달리, 이 방식에서는 LLM 에이전트가 탐색 도구를 반복 호출하며 그래프를 능동적으로 탐색한다. 에이전트는 각 탐색 단계에서 수집한 정보를 바탕으로 다음 탐색 방향을 스스로 결정하며, 이슈 해결에 필요한 코드 요소를 점진적으로 식별하고 수집한다. AutoCodeRover[4]는 에이전트에게 코드의 계층 구조를 기반으로 하는 탐색 인터페이스를 제공하고, 스펙트럼 기반 결함 위치 파악(Spectrum-Based Fault Localization, SBFL)을 통합하여 효율적으로 탐색한다. LingmaAgent[5]는 몬테카를로 트리 탐색(Monte Carlo Tree Search, MCTS) 알고리즘을 도입해 효율적인 그래프 탐색을 수행한다. OrcaLoca[6]는 우선순위 기반 작업 스케줄링과 거리 기반 가지치기(pruning) 메커니즘을 통해 탐색 컨텍스트를 최적화한다. LocAgent[7]는 Directed Heterogeneous graphs를 사용하여 다양한 코드 개체와 관계를 명시적으로 구분하였다. CoSIL[8]은 사전에 그래프를 구축하는 대신 LLM이 탐색 과정에서 필요에 따라 Call Graph를 동적으로 생성하고, 가지치기 에이전트를 통해 탐색 방향을 정교하게 제어했다.

이 방식은 적응적인 탐색을 통해 여러 파일에 걸친 의존성 문제를

효과적으로 추적할 수 있고, 다양한 알고리즘으로 탐색 효율을 개선한다. 그러나 반복적인 도구 호출로 인한 높은 연산 비용과 시간이 요구되고, 에이전트의 의사결정 과정이 복잡하여 탐색 실패나 오류 발생 시 원인 진단과 수정이 어렵다는 한계를 갖는다.

3.3 의존성 기반 동기화

의존성 기반 동기화 방식은 코드 그래프를 수정 작업의 일관성을 보장하는 제약 조건으로 활용한다. 정보 수집에 집중하는 앞선 접근과 달리, 이 방식은 한 위치의 수정이 의존 관계에 있는 다른 코드에 미치는 파급효과를 분석하고 이를 자동으로 동기화하는 데 초점을 맞춘다. CodePlan[9]은 저장소 전체의 의존성 그래프와 계획 그래프를 구축하고 초기 변경(seed)에서 시작해 영향을 받는 위치들을 식별하고 점진적인 수정 계획을 생성한다. SynFix[10]는 노드별 상세 설명이 포함된 RelationGraph를 이용해 LLM의 의미적 이해를 돕고, 의존성 체인을 추적하여 연관된 모든 결함 지점을 통합적으로 수정한다.

이 방식은 저장소 수준의 일관성을 체계적으로 보장한다는 장점이 있으며, 인터페이스 변경이나 리팩토링이 필요한 복잡한 버그를 해결하는 데 용이하다. 하지만 정적 분석의 불완전성으로 인한 의존성 누락 위험이 있고, 로직 버그나 알고리즘 수정과 같이 의존 관계와 무관한 버그에는 효과가 제한적이다.

3.4 임베딩을 통한 입력

임베딩 기반 구조 통합 방식은 코드 그래프를 텍스트로 변환하지 않고, GNN(Graph Neural Network)을 통해 인코딩하여 LLM의 내부 표현 과정에 통합한다. 그래프 정보를 프롬프트로 입력하는 앞선 방식들과 달리, 모델이 코드의 구조적 특성을 내재적으로 학습하도록 설계된 방식이다. CGM(Code Graph Model)[11]은 저장소 수준의 그래프 정보를 LLM 입력 공간에 투영하고, 인접 관계를 self-attention 마스크에 반영하여 구조적 제약을 학습시킨다. GALLa[12]는 AST(Abstract Syntax Tree)와 DFG(Data Flow Graph)를 GNN으로 인코딩한 후 어댑터를 통해 LLM 임베딩 공간에 정렬시키고, 2단계 학습 전략을 통해 모델을 최적화한다. Graph-LoRA[13]는 패치의 정확성 평가를 위해 APSG (Attributed Patch Semantic Graph)를 제안하고, GNN으로 추출된 그래프 특징을 LoRA 모듈에 결합하여 시퀀스 정보와 통합한다.

이 방식은 텍스트 변환 과정에서 발생하는 정보의 손실 없이 데이터나 제어 흐름과 같은 심층적인 정보를 보존한다는 장점이 있지만, 모델 아키텍처 수정이나 파인튜닝 등 통합을 위한 추가 비용이 발생하므로 범용적으로 적용되기 어렵다.

4. 향후 연구 방향

코드 그래프-텍스트 변환 기법의 투명성과 재현성 기존 연구들은 그래프 정보를 텍스트로 변환하는 과정에서 사용한 변환 기법이나 규칙을 상세하게 다루지 않았다. 그래프-텍스트 변환 기법에 따라 LLM의 코드 이해도와 APR 성능이 크게 달라질 수 있으므로, 향후 연구에서는 변환 방식이 LLM의 코드 이해도에 미치는 영향을 규명하고, 사용한 변환 규칙을 명시하여 연구의 재현성을 확보해야 한다.

표준화된 평가 환경 구축 기존 연구들은 서로 다른 벤치마크와 LLM을 사용하고 있어 객관적인 비교가 어렵다. 또한 전통적인 APR 평가 지표로는 그래프 자체의 효과와 기반 LLM의 추론 능력 향상을 구분하기 어렵다. 향후 연구에서는 표준화된 실험 환경에서 비교가 이루어져야 하고, 그래프 활용 방식의 기여도를 측정할 수 있는 평가 지표를 개발해야 한다.

APR 파이프라인 전반으로의 활용 확장 현재 코드 그래프는 주로 버그 탐색 단계에서 활용되고 있으나, 향후 연구에서는 의존성을 고려한 패치 생성 및 후보 검증 등 APR 각 단계에 적합한 그래프 유형과 활용 방식을 탐구하여 코드 그래프의 잠재력을 보다 폭넓게 활용해야 한다.

5. 결론

본 연구는 LLM 기반 APR 분야에서 코드 그래프를 활용하는 12편의 연구를 체계적으로 분석하였다. 그래프 정보를 LLM에 전달하는 방식에 따라 그래프 쿼리 기반 검색, 에이전트 주도 그래프 탐색, 의존성 기반 동기화, 임베딩 기반 입력의 네 가지 유형으로 분류하고, 각 유형의 장점과 한계를 종합적으로 정리하였다.

또한 기존 연구들을 분석하여 그래프-텍스트 변환 기법의 재현성 확보, 표준화된 평가 환경 구축, APR 파이프라인 전반에서의 활용 확장이라는 세 가지 주요 과제를 도출하였다. 본 연구가 제시한 분류 체계와 분석 결과는 후속 연구자들이 연구 목적에 적합한 그래프 활용 전략을 선택하고, 기존 접근법의 한계를 극복하는 새로운 방법론을 제안하는 데 기여할 것으로 기대된다.

※ 본 연구는 2025년 과학기술정보통신부 및 정보통신기획평가원의 SW중심대학사업(2023-0-00055)과 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원(RS-2024-00457866)을 받아 수행된 연구임

REFERENCES

[1] Zhang, Quanjun, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. "A survey of learning-based automated program repair." *ACM Transactions on Software Engineering and Methodology* 33, no. 2 (2023): 1-69.

[2] Liu, Xiangyan, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. "Codexgraph: Bridging large language models and code repositories via code graph databases." In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 142-160. 2025.

[3] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, Dong Yu. "RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph." In *ICLR 2025*.

[4] Zhang, Yuntong, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. "Autocoderover: Autonomous program improvement." In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1592-1604. 2024.

[5] MA, YINGWEI, and Yue Liu. "Improving Automated Issue Resolution via Comprehensive Repository Exploration." In *ICLR 2025 Third Workshop on Deep Learning for Code*.

[6] Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. "OrcaLoca: An LLM Agent Framework for Software Issue Localization." In ICML 2025.

[7] Chen, Zhaoling, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. "Locagent: Graph-guided llm agents for code localization." In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8697-8727. 2025.

[8] Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. "CoSIL: Software Issue Localization via LLM-Driven Code Repository Graph Searching." In ASE 2025.

[9] Bairi, Ramakrishna, Atharv Sonwane, Aditya Kanade, Vageesh D. C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2024. "CodePlan: Repository-Level Coding Using LLMs and Planning." *Proceedings of the ACM on Software Engineering.* 1 (FSE): 675–98.

[10] Tang, Xunzhu, Jiechao Gao, Jin Xu, Tiezhu Sun, Yewei Song, Saad Ezzini, Wendkûuni C. Ouédraogo, Jacques Klein, and Tegawendé F. Bissyandé. "SynFix: Dependency-aware program repair via RelationGraph analysis." In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 4878-4894. 2025.

[11] Hongyuan Tao, Ying Zhang, Zhenhao Tang, Hongen Peng, Xukun Zhu, Bingchang Liu, Yingguang Yang, Ziyin Zhang, Zhaogui Xu, Haipeng Zhang, Linchao Zhu, Rui Wang, Hang Yu, Jianguo Li, and Peng Di. "Code Graph Model (CGM): A Graph-Integrated Large Language Model for Repository-Level Software Engineering Tasks." In NeurIPS 2025.

[12] Zhang, Ziyin, Hang Yu, Sage Lee, Peng Di, Jianguo Li, and Rui Wang. "Galla: Graph aligned large language models for improved source code understanding." In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13784-13802. 2025.

[13] Yang, Zhenyu, Jingwen Wu, Zhen Yang, and Zhongxing Yu. "Parameter-Efficient Fine-Tuning with Attributed Patch Semantic Graph for Automated Patch Correctness Assessment." arXiv preprint arXiv:2505.02629 (2025).

대규모 언어 모델을 활용한 단위 테스트의 적합성 자동 식별

김대원, 이영규, 유준범
건국대학교 컴퓨터공학부
kdwkdw0078@gmail.com, sontouf@gmail.com, jbyoo@konkuk.ac.kr

Automated Identification of Unit Test Adequacy Using Large Language Models

Daewon Kim, Younggyu Lee, Junbeom Yoo
Department of Computer Science and Engineering, Konkuk University

요 약

소프트웨어가 수정됨에 따라 단위 코드의 동작이 변경되면, 기존 단위 테스트가 여전히 의도된 동작을 검증하고 있는지 판단하기 어렵다. 단위 테스트의 적합성 여부를 확인하기 위해 개발자가 코드를 직접 분석해야 하므로 많은 시간과 노력이 필요하다. 본 논문에서는 대규모 언어 모델(LLM)을 활용하여 단위 테스트의 적합성 여부를 자동으로 식별하는 기법을 제안한다. 제안 기법은 수정 전후의 소스 코드와 테스트 코드를 입력으로 받아, 선행 연구에서 정의한 변경 유형 분류 체계와 판단 기준을 프롬프트에 구조화하여 LLM 이 테스트의 검증 대상 유지 여부를 판단하고 구체적인 판단 근거를 제공하도록 설계되었다. GoogleTest 샘플을 대상으로 한 사례 연구 결과, LLM 은 함수 대체나 재귀 변환과 같이 명확한 변경 유형에서는 테스트 적합성을 효과적으로 식별하였으나, 복잡한 경우에는 일관된 판단에 어려움이 있었다. 본 연구는 LLM 을 활용한 단위 테스트 적합성 자동 식별의 가능성과 한계를 함께 확인하고, 이를 개선하기 위한 향후 연구 방향을 제시한다.

1. 서 론

단위 테스트는 소프트웨어를 구성하는 개별 기능이 명세에 따라 올바르게 동작하는지를 검증하는 핵심 수단이다. 그러나 소프트웨어가 지속적으로 수정되는 상황에서 단위 코드와 테스트 코드 간의 공진화(co-evolution)가 적절히 이루어지지 않으면, 테스트가 수정된 코드의 동작을 더 이상 검증하지 못하는 상황이 발생할 수 있다[1, 2].

선행 연구에서는 테스트 실행 시 발생하는 함수 호출과 객체 간 상호작용을 동적으로 분석하여 시퀀스 다이어그램으로 시각화하고, 테스트 변화 유형을 체계적으로 분류하였다. 이를 통해 개발자는 코드 수정 전후의 실행 흐름 차이를 시각적으로 파악하고, 테스트가 여전히 의도된 동작을 검증하는지 판단할 수 있는 기준을 확보하였다. 하지만, 시퀀스 다이어그램의 변화가 테스트 검증 대상의 실질적 변경인지, 단순한 리팩토링인지를 판단하는 과정은 여전히 개발자의 수작업에 의존한다는 한계가 있다[3, 4].

본 논문에서는 대규모 언어 모델(Large Language Model, LLM)을 활용하여 단위 테스트의 적합성 여부를 자동으로 식별하는 기법을 제안한다. 제안 기법은 수정 전후의 소스 코드와 테스트 코드를 입력으로 받아, 선행 연구의 16 개 변경 유형 분류 체계와 판단 기준을 프롬프트에 구조화하여 LLM 이 테스트의 검증 대상 유지 여부를 판단하도록 한다. 동적 실행이나 계측 없이 정적 코드 분석만으로 작동하며, 판단 결과와 함께 구체적인 근거를 제공하여 개발자의 의사결정을 지원한다[4].

본 논문의 구성은 다음과 같다. 2 장에서는 테스트 적합성 식별 기준과 LLM 기반 자동 분류 기법을 제안하며, 3 장에서는 GoogleTest 샘플을 대상으로 한 사례 연구를 통해 제안 기법을 평가한다. 4 장에서는 연구의 한계와 향후 연구 방향을 논의하고, 5 장에서 결론을 맺는다.

2. LLM 을 활용한 단위 테스트 적합성 자동 식별

단위 테스트의 적합성은 테스트가 의도한 검증 대상과 실제 검증하는 대상의 일치 여부로 판단된다. 개발자가 테스트를 작성할 때는 특정 함수 또는 객체의 특정 동작을 검증하려는 명확한 의도가 존재한다. 선행 연구는 코드 수정 후에도 통과하지만 더 이상 의도된 동작을 검증하지 못하는 테스트를 NLT(No-Longer-Testable)로 정의하였다[3]. 한편, 최근에는 대규모 언어 모델(LLM)을 활용하여 테스트 생성, 테스트 오라클 지원, 테스트 유지보수 등 다양한 소프트웨어 테스트 작업을 자동화하려는 연구가 활발히 진행되고 있다[5, 6]. 본 연구에서는 단위 테스트의 적합성 여부를 LLM 을 활용하여 자동으로 식별하기 위한 판단 기준을 다음과 같이 제시한다.

테스트 적합성 판단에는 세 가지 요소가 고려되어야 한다.

① 테스트가 실제로 호출하는 함수나 메서드가 수정 전후에 동일한지 확인해야 한다. 함수 이름이 같더라도 오버로딩, 오버라이딩, 조건문 변경 등으로 다른 구현이 실행될 수 있다.

② 호출하는 함수가 동일하더라도 그 함수의 입출력 관계가 변경되었는지 확인해야 한다. 내부 구현 방식만 바뀌고 결과가 동일하다면 테스트는 여전히 유효하지만, 같은 입력에 다른 출력을 생성한다면 테스트는 부적합하다.

③ 코드 변경이 명세 갱신에 따른 것인지 리팩토링에 따른 것인지 구분해야 한다. 함수 시그니처 변경이나 메서드 제거는 명세 변경을 시사하며, 변수명 변경이나 내부 로직 재구성은 리팩토링을 시사한다.

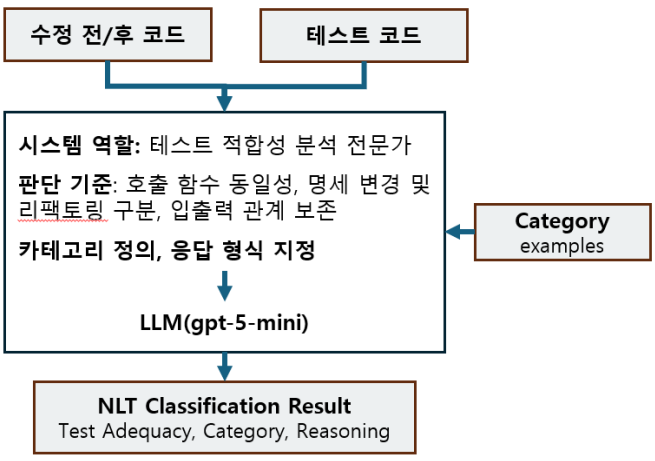


그림 1. 단위 테스트 적합성 자동 식별 과정

<그림 1>은 제안 기법의 전체 과정을 보여준다. 제안 기법은 수정 전 소스 코드, 수정 후 소스 코드, 테스트 코드를 입력으로 받는다. 프롬프트 생성기는 이들 코드와 선행 연구의 16 개 변경 유형 카테고리를 결합하여 구조화된 프롬프트를 생성한다. LLM 은 테스트 적합성 여부, 해당 카테고리, 판단 근거를 출력한다.

프롬프트는 다음 네 부분으로 구성된다. 시스템 역할에서는 LLM 을 테스트 적합성 분석 전문가로 설정하고 테스트 의도와 실제 검증 대상 비교에 집중하도록 지시한다. 판단 기준에는 2 절의 세 가지 요소를 명시하여 LLM 이 함수 동일성, 입출력 관계, 명세 변경 여부를 판단하도록 유도한다. 변경 유형은 <표 1>과 같이 부적합 유형 8 개와 적합 유형 8 개로 제공한다. 각 유형에 특성과 예시를 포함하여 LLM 이 상황을 카테고리에 매칭하도록 한다. 입력 코드는 수정 전후 소스 코드와 테스트 코드를 명확히 구분하여 제시한다.

LLM 의 출력은 분류 결과(적합/부적합), 변경 유형, 판단 근거로 구조화된다. 프롬프트는 증거 기반 추론을 명시적으로 요구하여, LLM 이 어떤 함수가 어떻게 변경되었고 변경 후에도 검증이 유지되는지를 구체적으로 서술하도록 한다. 이를 통해 개발자는 판단 근거를 검토하고 테스트 수정 방향을 결정할 수 있다.

표 1. 단위 테스트 적합성 변경 유형 카테고리

Type	Category	Description
NLT	Overloaded Call	테스트가 다른 오버로드된 메서드를 호출
	Overridden Call	기존 호출에서 오버라이드된 메서드로 변경
	Binding Change	가상 바인딩으로 인해 호출 대상 변경
	Hierarchy Shift	상속 계층 구조의 변경
	Condition Change	조건 로직으로 인해 호출 대상 변경
	Output Change	동일 입력에 대해 다른 출력
	Type Mismatch	입력 데이터 타입 변경
	Call Replacement	다른 동작을 가진 메서드 호출 대체
Non-NLT	Variable Renaming	동일 동작의 변수명 변경
	Method Renaming	동일 동작의 메서드명 변경
	Helper Replacement	동등한 헬퍼 메서드로 대체
	Instance Switch	다른 인스턴스에 의한 동일 동작 실행
	Delegated Call	다른 객체로 호출 위임
	Recursion Replacement	반복문을 재귀 호출로 대체
	Order Change	호출 순서 변경
	Mediator Adjustment	중간 객체 변경으로 동작 영향 없음

3. 사례연구

본 사례 연구는 변경 유형별 영향을 LLM 이 식별하는지 확인하기 위해 GoogleTest 에서 제공하는 10 개 샘플 중 변경 유형을 명확히 적용할 수 있는 8 개 샘플만을 선정하였다[7]. 나머지 2 개 샘플은 테스트 프레임워크 동작을 다루는 예제로, 본 논문에서 정의한 코드 변경 유형을 적용하기에 적합하지 않아 사례 연구 대상에서 제외하였다. 각 샘플에 대해 선행 연구에서 정의한 16 개 변경 유형 카테고리를 모두 적용하여 총 128 개의 테스트 케이스를 생성하였다. LLM 으로는 GPT-4 mini 를 사용하였으며, 2 장에서 제시한 프롬프트 구조를 적용하였다. 각 테스트 케이스에 대해 LLM 이 출력한 식별 결과(적합/부적합)와 카테고리를 정답과 비교하여 정확도를 측정하였다. 카테고리는 코드 수정의 의도와 테스트의 원래 검증 대상을 기준으로 선행 연구에서 사전에 정의되었다[3]. <그림 2>는 시스템 프롬프트의 일부분이고, <그림 3>은 프롬프트에 사용되는 카테고리 형식, <그림 4>는 LLM 의 응답 예시이다.

```
def get_system_prompt() -> str:
    return """You are an expert software testing analyst specializing in test validity assessment.
Your expertise includes:
- Understanding test intent and specification alignment
- Analyzing code changes and their semantic impact
- Evaluating runtime execution behavior changes
- Distinguishing between implementation changes and behavioral changes

Your task is to classify unit tests as either "NLT (No-Longer-Testable)" or
"Non-NLT" based on whether they still validate their intended behavior after code modifications.

CRITICAL RULES:
1. Focus on whether the TEST still validates its ORIGINAL INTENT
2. Consider both code-level semantics and execution-level behavior
3. Distinguish between "different implementation" and "different behavior"
4. Always provide clear, evidence-based reasoning
5. Use only the predefined category names exactly as given
6. Base your analysis on comparing before and after states
7. When multiple files are involved, consider their interactions and dependencies

OUTPUT FORMAT (strict):
Classification: [exactly "NLT" or "Non-NLT"]
Category: [exact category name from provided list]
Reasoning: [clear explanation with specific evidence]"""
```

그림 2. 테스트 적합성 식별을 위한 시스템 프롬프트

```
NON_NLT_CATEGORIES = {
    "Variable Renaming": {
        "description": "Variable names changed but the behavior and logic remain identical.",
        "example": "int count renamed to int total, same computation. ...
    }, ...
```

그림 3. 프롬프트에 사용되는 카테고리 형식

```
{  "classification": "Non-NLT",
   "category": "Recursion Replacement",
   "reasoning": "The original intent of the test was to validate the functionality of
   the Factorial function for negative, zero, and positive inputs. ...
}
```

그림 4. LLM 응답 결과

<표 2>는 128 개 테스트 케이스에 대한 LLM 의 분류 결과를 정답과 비교한 것이다. 총 128 개 케이스 중 44 개를 완전히 정확하게 분류하여 34.4%의 전체 정확도를 기록하였다. <그림 5>는 128 개 케이스의 분류 결과 분포를 시각화한 것으로, 일치 44 개(34%), 카테고리 불일치 36 개(28%), 불일치 39 개(31%), 카테고리 오류 9 개(7%)로 나타났다.

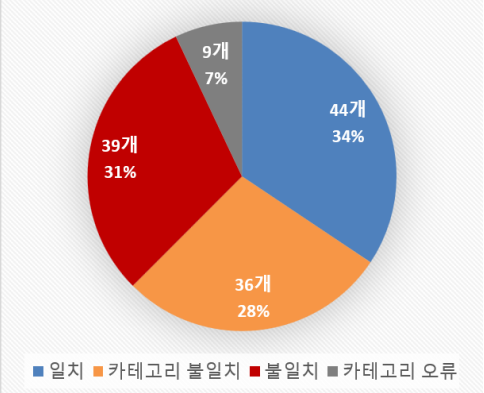


그림 5. 테스트 적합성 예측 결과

표 2 LLM 예측 결과

Type	Category	Accuracy
NLT	Overloaded Call	50%
	Overridden Call	50%
	Binding Change	25%
	Hierarchy Shift	37.5%
	Condition Change	0%
	Output Change	50%
	Type Mismatch	37.5%
	Call Replacement	50%
Non-NLT	Variable Renaming	12.5%
	Method Renaming	12.5%
	Helper Replacement	25%
	Instance Switch	37.5%
	Delegated Call	50%
	Recursion Replacement	50%
	Order Change	25%
	Mediator Adjustment	37.5%

LLM 의 응답 결과는 크게 4 가지로 분류되었다. 먼저 카테고리만 오분류한 경우이다. 36 개 케이스에서 LLM 은 NLT/Non-NLT 적합성은 정확히 판단했으나 세부 카테고리를 잘못 분류하였다. 이 중 17 개는 NLT 유형 내에서 카테고리만 혼동한 경우이며, 19 개는 Non-NLT 유형 내에서 카테고리만 혼동한 경우이다. LLM 은 함수 호출 대상이 변경되는 여러 NLT 카테고리들 (Overloaded Call, Type Mismatch, Hierarchy Shift 등) 간의 미묘한 차이를 구분하는 데 어려움을 보였으며, 구현 변경 방식이 유사한 Non-NLT 카테고리들(Instance Switch, Mediator Adjustment, Delegated Call 등) 간에서도 혼동이 발생하였다. 또한 제시된 카테고리내에서 분류하지 못했지만 LLM 자체적으로

Specification Alignment 등 카테고리를 생성하여 Non-NLT로 성공적으로 식별하였다. 두번째로 적합성 자체를 오판한 경우이다. 39 개 케이스에서 LLM은 NLT를 Non-NLT로 잘못 판단하였다. 주목할 점은 모든 적합성 오판이 NLT→Non-NLT 방향으로만 발생했으며, Non-NLT→NLT로 오판한 경우는 보이지 않았다. 이는 LLM이 코드 변경의 심각성을 과소평가하는 경향이 있음을 보인다. 입출력 관계의 실질적 변화, 가상 함수 바인딩 변경, 오버로딩으로 인한 다른 함수 호출 등 실제로는 테스트의 검증 대상이 바뀐 상황을 단순한 구현 변경으로 잘못 인식하였다. 세번째로 판단 자체를 하지 못한 경우이다. 9 개 케이스에서 LLM이 제시된 16 개 카테고리 내에서 분류하지 못하고 'Specification Alignment', 'No Significant Change', 'Behavioral Equivalence' 등의 자체 카테고리를 생성하였다. 주로 LLM이 조건문 변경에 따른 실행 경로의 차이를 인식하지 못한 경우로, 코드 수준 분석의 한계를 보여준다. 마지막으로 정확히 분류된 44 개 케이스는 모두 명확한 패턴적 특징을 공유하였다. 반복문과 재귀 호출 간의 구조적 전환, 변수명이나 메서드명만 변경된 경우, 동등한 헬퍼 함수로의 대체, 명시적인 위임 패턴 등 코드 수준에서 변경의 의미를 직관적으로 파악할 수 있는 경우에 LLM의 정확도가 높았다.

해당 사례 연구는 몇 가지 한계를 갖는다. 첫째, GoogleTest 샘플만을 대상으로 하여 실험 규모가 제한적이며, 실제 산업 프로젝트의 복잡한 테스트 케이스에 대한 검증이 필요하다. 둘째 34.4%의 정확도는 실용적 활용에 제약이 있다. 특히 적합성을 오판하는 30.5%의 경우(39 개 케이스)는 개발자에게 잘못된 신뢰를 주거나 불필요한 작업을 유발할 수 있어 위험하다. 셋째, 현재는 정적 코드만을 입력으로 사용하였으나, 선행 연구의 실행 로그나 시퀀스 다이어그램 정보를 추가로 활용하면 정확도를 개선할 수 있을 것이다. 넷째, 카테고리 간 혼동 유형을 분석한 결과 유사한 의미의 카테고리에서 빈번한 오류가 발생하였으므로, 프롬프트 최적화를 통해 이러한 카테고리 간 판단 기준을 더욱 구체화할 필요가 있다.

4. 결론

본 논문은 LLM을 활용하여 단위 테스트의 적합성을 자동으로 식별하기 위한 기법을 제안하였다. 선행 연구의 16 개 변경 유형 카테고리와 판단 기준을 프롬프트에 구조화하여, 테스트 적합성을 판단하도록 설계하였다. GoogleTest 샘플을 활용한 사례 연구에서 LLM은 명확한 변경 유형에서는 효과적으로 작동하였으나, 유사 카테고리 간 구분과 복잡한 경우에는 한계가 있음을 확인하였다.

향후 연구는 다음 방향으로 진행할 계획이다. 첫째, 프롬프트 최적화를 통해 유사 카테고리의 판단 기준을 더욱 구체화하고 명확하게 예시를 제공한다. 둘째, 단위 테스트를 직접 실행하여 실행 로그와 프롬프트를 같이 LLM의 입력으로 활용하여 실행 수준의 변화를 더 정확히 파악하도록 한다. 셋째, 대규모 실제 프로젝트를 대상으로 검증하여 실용성을 평가한다. .

Acknowledgment

본 연구는 원자력안전위원회의 재원으로 소형모듈원자로규제연구추진단의 지원을 받아 수행한 원자력 안전연구사업의 연구결과입니다. (No. 1500-1501-409)

참고문헌

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” IEEE Transactions on Software Engineering, vol. 41, no. 5, pp. 507–525, 2015.

[2] 김대원, 이영규, 허윤아, 유준범, "단위 테스트코드 간 추적성 복구를 위한 시나리오 기반 시각화 도구의 비교 및 필요성 연구" 한국정보과학회 학술발표논문집, 2025. pp. 376-378

[3] Y. Lee, D. Kim, and J. Yoo, “How can we find out that unit tests no longer test the unit?” in 19th IEEE International Conference on Software Testing, Verification and Validation (ICST), submitted

[4] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman, “Visualizing testsuites to aid in software understanding,” in 11th European Conference on Software Maintenance and Reengineering (CSMR’07), 2007, pp. 213–222.

[5] S. Rahman, S. Kuhar, B. Cirisci, P. Garg, S. Wang, X. Ma, A. Deoras, and B. Ray, “UTFix: Change Aware Unit Test Repairing using LLM,” Proc. ACM Program. Lang., vol. 9, no. OOPSLA1, art. 85, pp. 1–26, Apr. 2025

[6] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, “Software Testing With Large Language Models: Survey, Landscape, and Vision,” IEEE Transactions on Software Engineering, vol. 50, no. 4, pp. 911–936, Apr. 2024,

[7] GoogleTest Team, “GoogleTest samples,” GoogleTest Documentation, [Online]. Available: https://google.github.io/googletest/ [last accessed Jan. 13, 2026.]

딥러닝 기반 국방 SW 결함위치추정을 위한 변이 기반 데이터셋 구축의 체계적 연구

양희찬[O]
KAIST
heechan.yang@kaist.ac.kr

이아청
KAIST
ahcheong.lee@kaist.ac.kr

조규태
LIG Nex1
kyutae.cho2@lignex1.com

김문주
KAIST/브이플러스랩
moonzoo.kim@gmail.com

Systematic Study of Mutation-Based Dataset Construction for Deep Learning-Based Fault Localization on Military Defense SW

Heechan Yang[O]
KAIST

Ahcheong Lee
KAIST

Kyutae Cho
LIG Nex1

Moonzoo Kim
KAIST/브이플러스랩

요 약

딥러닝 기반 결함 위치 추정(Deep Learning-Based Fault Localization, DLFL)은 소프트웨어 디버깅 자동화 분야에서 혁신적인 성과를 거두고 있으나, 학습 데이터셋 구축에 막대한 계산 비용과 공개된 데이터셋 구축 도구의 부재로인해 실무 적용의 주요한 병목 현상으로 작용하고 있다. 본 논문은 DLFL의 실용성을 확보하기 위해 DLFL 데이터셋 구축 과정을 최적화하는 체계적인 방법론을 제시하고, 이를 자동화하는 도구를 구현하여 실제 L사의 국방 무기체계 소프트웨어에 적용하여 그 유효성을 입증한다.

먼저, 데이터셋 구축 비용에 결정적인 영향을 미치는 두 가지 핵심 파라미터인 (1) '변이체 생성 대상 라인 선택 비율'과 (2) '라인당 변이체 생성 개수'의 최적 임계값을 도출하기 위해 오픈소스 자바 벤치마크인 Defects4J를 대상으로 탐색적 실험을 수행하였다. 실험 결과, 기존 방식 대비 **계산 비용을 74.6% 단축**하면서도 결함 추정 성능을 유지할 수 있는 최적의 설정값을 확인하였다. 또한, 본 연구에서는 개발자의 디버깅 직관을 정량적으로 모델링한 (3) 스택 트레이스(Stack Trace, ST) 관련성 특징을 최초로 설계하여 제안하였으며, 이를 학습 특징으로 추가함으로써 결함 위치 추정 정확도를 기존 대비 **6.8%~11.0% 향상**시켰다.

최종적으로, 도출된 최적 파라미터와 신규 ST 특징 추출 로직을 탑재한 자동화 도구를 L사의 6개 국방 무기체계 소프트웨어(총 61 KLoC, 300개 인공 결함) 시스템에 탑재하여 기술의 실전 배치 가능성을 검증하였다. 그 결과, **Top-5 기준 85.0%의 높은 탐지 정확도**를 달성함과 동시에 데이터셋 구축 비용을 **약 79%(9,081→1,907 CPU-hours) 절감**하였다. 이를 통해 본 연구가 개발한 자동화 도구가 고신뢰 국방 소프트웨어 분야에서 실무자가 별도의 전문 지식 없이도 고성능 DLFL 기술을 즉각 운용할 수 있는 실질적인 기술적 토대를 마련하였음을 입증하였다.

1. 서론

소프트웨어 결함 위치 추정(Fault Localization, FL)은 프로그램 실패의 원인이 되는 특정 코드 요소(파일, 함수, 라인 등)를 식별하여 개발자의 디버깅 노력을 최소화하는 것을 목표로 한다. 전통적인 기법으로는 테스트 커버리지 패턴을 분석하는 스펙트럼 기반 FL(SBFL)[1]과 변이 프로그램을 활용하여 결함의 전파 과정을 분석하는 변이 기반 FL(MBFL)[2,3]이 존재한다. 최근에는 이런 전통적인 기법으로부터 추출된 데이터를 신경망에 학습시킨 딥러닝 기반 결함 위치 추정(DLFL) 기술이 기존 기법들보다 높은 정확도를 보이며 차세대 FL 기술로 주목받고 있다[4, 5, 6].

그러나 기존 MBFL 기반 DLFL 연구들은 주로 모델의 구조적 고도화에만 집중할 뿐, 모델의 성능과 비용을 좌우하는 **데이터셋 구축 방법론**에 대해서는 표준화된 절차없이 연구자의 임의적인 방식에 의존하고 있다. 따라서, 실무 환경에서는 MBFL 기반 DLFL을 적용하는데 다음과 같은 어려움이 따른다. 1. 공개된 도구나 방법론이 없어 적용과정에 많은 불확실성이 생긴다. 2. 데이터셋 구축 시 막대한 계산 자원과 시간 비용이 예상된다. 일례로, 61 KLoC 규모의 소프트웨어에 대한 DLFL 데이터셋을 구축에 임의적인 방법으로 시행할 시 약 9,081 CPU-hours이 소요될 것으로 예상되며, 이는 자원이 한정된 실무 환경에서 MBFL 기반 DLFL 기술을 도입하는 데 큰 진입 장벽이 된다.

본 연구는 이러한 한계를 극복하고 **최종적으로 L사의 실제 국방 무기체계 소프트웨어에 DLFL 기술을 실용적으로 적용하는 것을 목표**로 한다. 이를 위해 본 논문은 자동화 도구 개발, 탐색적 실험, 그리고 실무 사례 연구로 이어지는 연구 방법론을 채택한다.

본 연구의 주요 기여는 다음과 같다:

1. **데이터셋 구축 시간 비용의 74.6% 단축:** 체계적인 실험을 통해 변이체 생성 대상 라인 선택 비율(70%)과 라인당 변이체 생성 개수(3개)의 최적값을 도출, 기존 방식 대비 데이터셋 구축 시간 비용을 74.6% 절감하였다.
2. **개발자의 디버깅 직관을 정량적으로 모델링한 '스택 트레이스(Stack Trace, ST) 관련성 특징 설계를 통한 정확도 향상:** 본 연구에서 최초로 고안한 ST 관련성 특징을 수치화 하여 기존 특징(SBFL, MBFL)과 결합하여 활용하여, DLFL 모델의 결함 탐지 성능을 기존 대비 6.8%~11.0% 향상시켰다. 기존 연구들이 실행 경로의 통계적 유의성에만 의존했던 것과 달리, 본 특징은 프로그램 비정상 종료 시 발생하는 런타임 문맥 정보를 딥러닝 모델이 직접 학습할 수 있도록 설계된 독창적인 기술적 기여이다.
3. **국방 무기체계 SW 적용을 통한 실무적 실효성 검증:** 탐색적 연구에서 얻은 최적 가이드라인과 신규 특징을 바탕으로 직접 개발한 데이터셋 자동 구축 도구를 L사의 실제 개발 환경에 통합하였으며, 6개 국방 무기체계 소프트웨어에 적용하여 Top-5 기준 85.0%의 높은 정확도를 달성하고, 시간 비용을 79% 단축하여 제안 방법론의 실무적 유용성을 입증하였다.

이 논문은 산학연주관 핵심 SW(응용연구) 연구개발 과제(계약번호 UC210018AD)와 정부의 재원으로 한국연구재단의 지원(NRF-2021R1A5A1021944, NRF-RS-2024-00357348)의 지원을 받아 수행된 연구임.

2. 배경

2.1 데이터셋 구성 요소

MBFL 기반 DLFL 모델의 학습 데이터셋은 소스 코드의 각 라인을 하나의 레코드(Record)로 구성한다. 기존의 일반적인 데이터셋은 각 라인에 대하여 총 8개의 특징값을 포함하며, 이는 6개의 서로 다른 SBFL 기법과 2개의 MBFL 기법으로 도출된 의심도 점수로 이루어진다.

2.2 특징 추출 방법

데이터셋을 구성하는 특징들은 프로그램의 동적 실행 정보를 바탕으로 추출된다. SBFL 특징은 결함 프로그램에 테스트 스위트를 실행하여 얻은 라인 커버리지(Line Coverage) 정보를 활용하는 반면, MBFL 특징은 소스 코드에 인위적인 수정을 가하는 변이 분석(Mutation Analysis) 과정을 거쳐 추출된다. 구체적인 산출 방법은 다음과 같다.

2.2.1 스펙트럼 기반 결함위치추정(SBFL) 특징

SBFL 특징은 각 테스트 케이스의 실행 경로와 통과/실패 여부를 결합하여 특정 코드 라인이 결함을 포함하고 있을 가능성(의심도)을 계산한다. 따라서, 본 연구에서는 선행 연구들에서 성능이 검증된 6종의 SBFL 기법(DStar, GP13, Naish1, Naish2, Ochiai, Tarantula)을 활용하여 기본 학습 특징을 구축한다.

2.2.2 변이 기반 결함위치추정(MBFL) 특징

MBFL 특징은 코드 라인에 미세한 변경을 가한 변이체(Mutant)를 생성하고, 이에 대한 테스트 동작 변화를 분석하여 도출한다. 본 연구에서는 변이체 생성 도구인 MUSIC++[7]를 활용하여 대상 라인에 대해 M개의 변이체를 생성한다. MBFL은 변이 적용 전후의 실행 결과 변화를 통해 결함 위치를 추정하며, 본 연구에서는 대표적인 MBFL 기법인 MUSE[2]와 Metallaxis[3] 공식을 활용한다. MBFL은 SBFL보다 높은 정확도를 제공하지만, 모든 라인에 대해 수많은 변이체를 생성하고 테스트하므로 막대한 계산 비용이 발생한다는 한계가 있다.

3. 제안 방법론: 데이터셋 최적화 및 스택 트레이스(ST) 관련성 특징

본 연구는 MBFL 기반 DLFL의 실효성을 확보하기 위해 **(1) 데이터셋 구축의 시간 비용 단축**과 **(2) 결함 추정 정확도 향상**이라는 두 가지 핵심 목표를 설정한다. 이를 달성하기 위해 변이 분석 과정을 최적화하는 파라미터 탐색 전략을 수립하고 (3.1장), 개발자의 디버깅 문맥을 정량적으로 모델링한 새로운 특징 추출 방법을 제안한다 (3.2장).

특히, 본 연구에서는 제안된 방법론을 실제 소프트웨어 개발 현장에 즉각적으로 적용할 수 있도록 약 6,000라인의 파이썬 스크립트로 구성된 '범용적 MBFL 기반 DLFL 데이터셋 구축 자동화 도구'를 직접 설계하고 개발하였다. 이 도구는 복잡한 변이 분석 및 특징 추출 과정을 자동화하여 구축 비용을 절감하는 동시에, 본 연구의 핵심 기술적 기여인 ST 관련성 특징을 생성하는 중추적인 역할을 수행한다.

3.1 데이터셋 구축의 시간 비용 단축

DLFL 데이터셋 구축 과정에서 가장 많은 연산 자원이 소요되는 단계는 MBFL 특징 추출을 위한 변이 분석 단계이다. 본 연구는 변이 분석 비용에 결정적인 영향을 미치는 두 가지 핵심 파라미터를 정의하고, 개발된 자동화 도구를 통해 모델의 성능 저하를 최소화하면서도 비용을 극대화하여 절감할 수 있는 최적값을 탐색한다. 특히 국방 소프트웨어와 같은 대규모 시스템에 기술을 적용하기 전, 통제된 환경(Defects4J)에서의 탐색적 연구를 통해 보편적인 가이드라인을 도출하는 것이 본 전략의 핵심이다.

표 1: RQ1 '변이체 생성 대상 라인 비율' 실험

Line Selection Ratio	#Mutants Per Line	Top-1	Top-3	Top-5	MFR	p-value
100%	10	52.7	98.3	123.4	29.9	1.0000
90%		51.7	96.7	122.3	30.7	0.5536
80%		51.0	96.4	120.8	30.0	0.5967
70%		**50.7**	**92.5**	**117.4**	**31.4**	**0.0750**
60%		49.3	91.3	118.2	33.1	0.0495
50%		47.4	87.7	114.0	34.4	0.0027
40%		45.9	87.2	111.8	32.9	0.0004
30%		42.6	83.8	108.9	36.7	0.0000
20%		42.3	84.3	111.1	38.6	0.0000
10%		43.2	86.6	112.8	40.8	0.0000

첫 번째 파라미터는 **'변이체 생성 대상 라인 선택 비율'**이다. 모든 코드 라인에 대해 변이 분석을 수행하는 것은 막대한 비용을 발생시키므로, 연산이 가벼운 SBFL(Ochiai) 의심도 점수를 기준으로 라인을 정렬한다. 이후 상위 $N\%$의 라인만을 변이체 생성 대상으로 선정함으로써 분석 범위를 전략적으로 제한하고 테스트 실행 횟수를 감축한다.

두 번째 파라미터는 **'라인당 변이체 생성 개수'**이다. 선정된 각 대상 라인에 대해 생성되는 변이체의 수를 조절하여 연산량을 제어한다. 기존 연구들이 임의의 개수를 설정했던 것과 달리, 본 연구는 변이체 개수를 단계적으로 축소하며 DLFL 모델이 유의미한 패턴을 학습하는 데 필요한 최소한의 변이체 수를 도출하고자 한다.

3.2 결함위치추정 정확도 향상 전략

데이터셋의 효율성뿐만 아니라 추정의 정밀도를 높이기 위해, 기존 DLFL 연구에서 사용되던 실행 경로 기반 특징(SBFL, MBFL) 외에 **본 연구에서 최초로 설계하여 제안하는 'ST(Stack Trace) 관련성 특징'**을 추가한다. 해당 로직은 본 연구에서 개발한 자동화 도구의 핵심 모듈로 구현되어 데이터셋 생성 시 자동으로 산출된다.

3.2.1 스택 트레이스(ST) 관련성 특징

본 연구는 딥러닝 모델의 학습 성능을 고도화하기 위해, **실제 개발자의 디버깅 직관을 수치화한 ST 관련성 특징을 세계 최초로 제안**한다.

실무 개발자들은 프로그램 비정상 종료 시 발생하는 스택 트레이스 정보를 가장 먼저 활용하며, 특히 최상단 프레임이 가리키는 코드 위치가 결함과 밀접할 것이라고 판단한다. 본 연구는 이러한 인간의 디버깅 휴리스틱을 딥러닝 모델이 학습 가능한 형태의 정량적 특징으로 설계하였다. 각 코드 라인의 ST 관련성 특징값은 실패 테스트 실행 시 발생하는 스택 트레이스 정보를 바탕으로 다음과 같이 계산된다.

$$STRelevance(s) = max_{f \in stackTrace(P,s)}\left(\frac{1}{position(f)+1} \times e^{-distance(s,f)^2}\right)$$

해당 공식의 주요 구성 요소는 다음과 같다.

- *stackTrace(P, s)*: 프로그램 P에서 발생한 스택 트레이스 중, 라인 s와 동일한 함수 내에 존재하는 프레임들의 집합
- *position(f)*: 해당 프레임의 위치 (최상단 프레임은 0이며, 아래로 갈수록 1씩 증가)
- *distance(s, f)*: 소스 코드 라인 s의 번호와 프레임 f가 가리키는 실제 라인 번호 간의 거리

해당 공식은 비정상 종료 지점(최상단 프레임)과의 거리(position)가 가까울수록, 그리고 프레임이 지칭하는 코드 위치와 물리적으로 인접할수록(distance) 높은 가중치를 부여하도록 설계되었다. 본 연구에서 최초로 제안한 **ST 관련성** 특징은 런타임 오류 문맥을 직접적으로 반영한다.

표 2: RQ2 '라인당 변이체 생성 개수' 실험 결과

Line Selection Ratio	#Mutants Per Line	Top-1	Top-3	Top-5	MFR	p-value
100%	10	52.7	98.3	123.4	29.9	1.0000
70%	10	50.7	92.5	117.4	31.4	0.0750
	9	49.8	93.6	119.4	31.3	0.1977
	8	49.1	93.6	118.8	31.3	0.1533
	7	48.8	94.0	117.6	30.5	0.1286
	6	50.9	93.2	119.4	31.2	0.1836
	5	49.7	92.1	116.4	31.2	0.0753
	4	50.6	94.5	118.6	31.3	0.1024
	3	48.6	93.1	118.3	31.0	0.0792
	2	48.8	92.3	116.0	32.0	0.0296
	1	44.6	88.9	113.2	32.8	0.0015

기존 연구들이 Stack Trace의 단순 텍스트 정보만을 활용했다면, 본 연구에서는 각 라인과 비정상 종료(crash) 간의 관련성을 함수 및 라인 번호를 기반으로 수치화하여 학습 특징으로 활용한다. 결과적으로 실행 경로와 문맥 기반 특징을 결합함으로써, 모델이 결함의 논리적 연관성을 더욱 정교하게 학습할 수 있도록 설계한 것이 본 연구의 주요 기여이다.

4. 탐색적 실험 설정

본 장에서는 제안하는 데이터셋 구축 최적화 방법론과 새롭게 설계된 ST 관련성 특징의 유효성을 검증하기 위한 탐색적 실험 설정을 기술한다. 본 실험의 목적은 가용 자원이 한정되고 복잡도가 높은 국방 소프트웨어에 DLFL 기술을 적용하기에 앞서, 통제된 벤치마크 환경에서 비용과 성능 사이의 최적 임계값을 도출하고 신규 특징의 기여도를 확인하는 데 있다. 본 실험은, MBFL 논문들에서 가장 많이 연구가 된 Defects4J v1.2.0의 5개 프로젝트에서 추출된 257개의 실제 결함 버전을 활용한다.

4.1 연구 질문 (Research Questions)

본 실험은 다음 세 가지 연구 질문에 답함으로써 효율적이고 정밀한 DLFL 모델 구축 가이드를 수립한다.

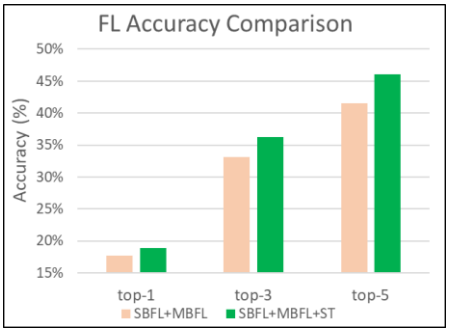
- **RQ1 (변이 라인 선택 비율):** SBFL(Ochiai) 의심도를 기반으로 변이 분석 대상 라인을 상위 ***N%*** (10%~100%)로 제한할 때, ***N***값의 변화가 DLFL 모델의 정확도와 데이터셋 구축 비용에 미치는 영향은 무엇인가?
- **RQ2 (라인당 변이체 개수):** 각 라인당 생성하는 변이체의 수 ***M*** (1~10개)을 조절할 때, ***M***값의 변화가 모델의 정확도와 구축 비용에 미치는 영향은 무엇인가?
- **RQ3 (스택 트레이스 관련성):** 본 연구에서 최초로 설계하여 제안한 스택 트레이스(ST) 관련성 특징을 학습 데이터셋에 추가하는 것이 DLFL 모델의 결함 위치 추정 정확도 향상에 어느 정도 기여하는가?

4.2 실험 모델 및 수행 방법

실험 모델은 최신 DLFL 연구인 CodeHealer[6]의 구조를 계승한 Multi-Layered Perceptron(MLP)을 사용한다. 해당 모델은 입력층의 노드 수를 특징(Feature)의 개수와 동일하게 설정하고, 하나의 은닉층(Hidden Layer)을 포함하도록 설계되었다.

실험 수행을 위해, 앞서 언급한 데이터셋 자동 추출 도구를 활용하여 각 결함 프로그램으로부터 SBFL, MBFL 및 ST 관련성 특징을 자동으로 산출하고 통합 데이터셋을 생성하였다. Java 프로그램에 변이 생성 도구로는 PITest를 활용한다. 결과의 신뢰성을 확보하고 특정 데이터셋에 대한 과적합(Overfitting)을 방지하기 위해 10-fold cross-validation을 적용하였으며, 변이 분석 및 학습 과정의 비결정적 요소를 고려하여 전체 과정을 총 10회 반복 수행한 후 평균값을 최종 결과로 사용하였다.

그림 1: RQ3 '스택 트레이스 관련성 특징' 실험 결과



4.3 평가 지표 및 통계적 검증

본 연구는 데이터셋 구축의 **시간 효율성**과 모델의 **결함위치추정 정확도**를 평가한다.

- **시간 효율성**: 데이터셋 구축부터 모델 학습 완료까지 소요되는 전체 시간(CPU-hours)을 측정하여 비용 절감 효과를 수치화한다.
- **Top-N**: 모델이 예측한 의심도 상위 ***N***개 라인 이내에 실제 결함 코드가 포함된 결함의 총 개수를 의미한다.
- **MFR (Mean First Rank)**: 개발자가 결함을 발견하기 위해 조사해야 하는 첫 번째 실제 결함 코드의 평균 순위이다. 수치가 작을수록 디버깅 효율이 높음을 의미한다.

또한, 서로 다른 설정 간의 성능 차이가 통계적으로 유의미한지 확인하기 위해 Mann-Whitney U Test를 수행한다. 본 연구에서는 유의 수준 0.05를 기준으로 각 설정 간의 성능 변화를 엄격히 검증하여, 성능 저하 없이 최대로 비용을 절감할 수 있는 최적의 파라미터 조합을 도출한다.

5. 탐색적 실험 결과

본 장에서는 4장에서 설정한 세 가지 연구 질문에 따라, 앞서 개발한 자동화 도구를 통해 수행된 탐색적 실험의 결과를 분석한다. 모든 실험 결과는 베이스라인(라인 선택 비율 100%, 라인당 변이체 개수 10개) 대비 성능 변화와 통계적 유의성을 기준으로 평가하였다.

5.1 RQ1: 변이체 생성 대상 라인 선택 비율 (Target Line Selection Ratio)

실험 결과, SBFL 의심도 상위 **70%의 라인만을 변이 분석 대상으로 선정**한 설정은 베이스라인과 비교하여 통계적으로 유의미한 성능 저하를 보이지 않았다(p-value = 0.0750). 반면, 표 1의 60% 이하 설정부터는 정확도가 급격히 하락하는 양상이 관찰되었다. 이를 통해 정확도 손실 없이 데이터셋 구축 시간 비용을 약 29.8% 절감할 수 있는 최적의 임계값이 상위 70%임을 확인하였다.

5.2 RQ2: 라인당 변이체 생성 개수 (Mutant Count Per Line)

표 2은 RQ1에서 도출된 70% 라인 선택 설정을 고정한 후, 변이체 개수(M)를 조절하며 측정한 결과이다. 데이터 분석 결과, **라인당 3개의 변이체**만으로도 베이스라인과 통계적으로 대등한 수준의 성능을 유지할 수 있음을 입증하였다(p-value = 0.0792). 최종적으로 표 3의 '라인 선택 비율 70%'와 '라인당 변이체 3개' 조합을 통해, 베이스라인 대비 전체 구축 시간을 **74.6% (198.2시간→50.4시간)** 단축하는 최적의 효율성을 확보하였으며, 이 가이드라인은 이후 국방 SW 적용의 기술적 근거가 된다.

5.3 RQ3: 스택 트레이스 관련성 특징의 효과

본 연구에서 설계하여 최초로 제안한 ST 관련성 특징의 성능 향상 기여도는 그림 1의 그래프를 통해 명확히 확인할

표 3: 국방 무기체계 소프트웨어
(FT: Failing Tests, PT: Passing Tests, AB: Artificial Bug)

Subjects	Language	#Artifical Buggy Program	Size (LoC)	Line Cov.	Avg. #FTs (on AB)	Avg. #PTs
System_A	C	50	18,854	57.6%	2.5	74.4
System_B	C	50	4,870	55.8%	1.4	11.6
System_C	C	50	6,217	66.9%	3.6	7.1
System_D	CPP	50	10,788	60.9%	3.6	3.2
System_E	CPP	50	8,333	68.4%	3.2	31.9
System_F	CPP	50	12,021	45.8%	5.4	20.9

수 있다. 기존 특징 조합(SBFL+MBFL)에 본 연구의 ST 특징을 추가하여 학습시킨 결과(SBFL+MBFL+ST), 모든 지표에서 성능이 향상되었다. 구체적으로 그림 1에 나타난 바와 같이 Top-1/Top-3/Top-5 정확도는 각각 6.8%, 9.3%, 11.0% 개선되었으며, MFR 지표 또한 14.4% (36.2→30.9) 향상되었다. 이는 실행 경로 중심의 기존 특징들과 본 연구에서 새롭게 제안한 에러 문맥 기반 ST 특징이 모델의 결함 식별력을 극대화했음을 보여준다.

6. 국방 무기체계 소프트웨어 적용

본 장에서는 Java 기반 데이터셋인 Defects4J를 통한 탐색적 실험으로 검증된 가이드라인과 ST 관련성 특징을 실제 C/C++ 기반 국방 무기체계 소프트웨어에 적용한 결과를 기술한다. Java와 C/C++ 간의 구문적 차이가 존재하나, 프로그램의 실행 경로를 변형하여 결함을 식별하는 MBFL의 핵심 기저 원리는 동일하다. 이러한 원리적 공통성으로 인해 Java 환경에서 도출된 최적 가이드라인은 C/C++ 분석에서도 그 유효성이 유지된다. 본 연구에서 개발한 자동화 도구는 L사의 실제 DLFL 운용 시스템에 성공적으로 통합되어 실무적 타당성과 실전 배치 가능성을 입증하였다.

6.1 대상 국방 무기체계 소프트웨어 및 실험 설정

본 사례 연구는 국내 유수의 방산 기업인 L사의 항공, 해양, 유도무기 등 실제 무기체계 시스템에 탑재되어 운용 중인 6종의 C/C++ 미들웨어 소프트웨어를 대상으로 수행되었다. 대상 시스템의 전체 코드 규모는 약 61 KLoC에 달하며, 이는 높은 신뢰성과 복잡한 로직을 요구하는 국방 소프트웨어의 특성을 고스란히 반영하고 있다.

국방 분야의 보안 규정상 과거 결함 데이터 및 실결함 로그에 대한 직접적인 접근이 제한됨에 따라, 본 연구에서는 소스 코드의 각 라인에 변이를 삽입하여 소프트웨어당 50개씩, 총 300개의 인공결함 프로그램을 구축하여 평가를 진행하였다. 표 3는 실험에 사용된 각 시스템의 상세 지표를 보여주며, 보안 유지를 위해 시스템 명칭은 익명화하여 기술한다.

6.2 사례 연구 결과 분석

탐색적 실험에서 도출된 최적 파라미터(라인 선택 비율 70%, 라인당 변이체 3개)와 본 연구의 독창적 기여인 ST 관련성 특징을 국방 SW 데이터셋에 적용한 결과, **자원 소모를 최소화하면서도 압도적인 결함 추정 성능**을 달성하였다.

첫째, 결함 위치 추정 정확도 측면에서 **Top-1 62.7%, Top-3 82.6%, 그리고 Top-5 기준 85.0%**라는 매우 높은 정확도를 기록하였다. 특히, 개발자가 버그를 찾기 위해 조사해야 하는 코드의 평균 순위를 의미하는 MFR 지표는 **18**로 나타났다. 이는 수만 라인의 거대한 코드 베이스 내에서 실무자가 상위 18줄 내외만 조사하더라도 실제 결함 위치를 특정할 수 있음을 입증하는 것이며, 실제 무기체계 개발 현장의 디버깅 노력을 획기적으로 절감할 수 있는 수치이다.

둘째, 데이터셋 구축의 비용 효율성 측면에서 괄목할 만한 성과를 거두었다. 기존의 보수적인 방식(전체 라인 대상, 다수의 변이체 생성)을 고수할 경우 막대한 계산 자원이 요구되지만, 본 연구의 최적화된 가이드라인을 적용한 결과 전체 데이터셋 구축 시간을 합계 약 **1,739 CPU-hours** 수준으로 억제하였다. 이는 이론적 베이스라인 대비 **약 79%의 시간 비용을 단축**한 결과이다. 이러한 극적인 비용 절감은 자원이 제한된 실무 환경에서 DLFL 모델을 주기적으로 갱신하고 실전 배치할 수 있는 강력한 실무적 타당성을 부여한다.

6.3 L사의 사내 DLFL 시스템 구축

본 사례 연구를 통해 확인된 성과는 이론적으로 제안된 방법론이 실제 방산 기업의 DLFL 시스템에 성공적으로 탑재되어 실질적인 가치를 창출했다는 점이다. 본 연구에서 약 6,000줄 규모로 개발된 도구는 자동으로 학습 데이터를 생성하고 정밀한 추론 결과를 도출하여, 국방 SW 개발 과정의 자동화를 앞당겼다.

결과적으로, L사 시스템에 탑재된 본 MBFL 기반 DLFL 데이터셋 자동 구축 도구는 기술적 진입 장벽을 낮추어 실무자가 별도의 딥러닝 및 자동 디버깅 기술에 대한 전문 지식 없이도 결함 위치 추정 기술을 현업에 적용할 수 있도록 하였다. 이는 국방 소프트웨어의 품질 보증 및 유지보수 단계에서 발생하는 디버깅 병목 현상을 해결할 수 있는 실질적인 기술적 토대를 마련한 것으로 평가된다.

7. 결론

본 논문은 국방 소프트웨어 분야의 딥러닝 기반 결함 위치 추정(DLFL) 도입을 위해, 변이 기반 DLFL 데이터셋 구축 과정을 최적화하는 체계적인 연구 방법론을 제시하였다. 실무적 기여 측면에서는 도출된 가이드라인과 ST 특징 추출 로직을 탑재한 자동화 도구를 **L사의 DLFL 시스템에 성공적으로 통합**하였다. 해당 도구는 데이터셋 구축부터 학습 및 추론까지의 전 과정을 자동화하여 사내 직원이 즉시 운용 가능한 실무 인프라를 제공한다. 탐색적 실험에서 도출한 최적의 파라미터와 ST 관련성 특징을 6개 국방 무기체계 C/C++ 소프트웨어에 적용한 결과, **Top-5 기준 85.0%**의 높은 정확도와 약 **79%**의 **구축 비용 단축**을 달성하여 그 유효성을 입증하였다.향후 연구로는 구축된 고품질 변이 데이터셋을 대규모 언어 모델(LLM)[8,9,10]의 의미론적 특징과 결합하여, 실행 분석과 언어적 이해가 통합된 차세대 결함 위치 추정 프레임워크를 개발할 계획이다.

참고 문헌

[1] S. Yoo. (2012). Evolving human competitive spectra-based fault localisation techniques. SSBSE 2012.
[2] S. Moon, Y. Kim, M. Kim and S. Yoo, "Ask the Mutants: Mutating Faulty Programs for Fault Localization," ICST 2014.
[3] M. Papadakis and Y.L. Traon. Metallaxis-FL: mutation-based fault localization. STVR 2015.
[4] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu. Improving fault localization and program repair with deep semantic features and transferred knowledge. ICSE 2022.
[5] X. Wang, H. Yu, X. Meng, H. Cao, H. Zhang, H. Sun, X. Liu, and C. Hu. MTL-TRANSFER: Leveraging Multi-task Learning and Transferred Knowledge for Improving Fault Localization and Program Repair. TOSEM 2024.
[6] L. Zhang, S. Guo, Y. Guo, H. Li, Y. Chai, R. Chen, X. Li, and H. Jiang. Context-based Transfer Learning for Structuring Fault Localization and Program Repair Automation. TOSEM 2025.
[7] D. L. Phan, Y. Kim and M. Kim, "MUSIC: Mutation Analysis Tool with High Configurability and Extensibility," ICSTW 2018.
[8] S. Kang, G. An, and S. Yoo. A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. FSE 2024.
[9] C. Xu, Z. Liu, X. Ren, G. Zhang, M. Liang, and D. Lo. FlexFL: Flexible and Effective Fault Localization With Open-Source Large Language Models. TOSEM 2025.
[10] A.Z. H. Yang, C.L. Goues, R. Martins, and V. Hellendoorn. Large Language Models for Test-Free Fault Localization. ICSE 2024

ExplosionGuard: 예산 제약 심볼릭 실행을 위한 정책 합성 및 가드레일 시스템

이재영[1], 이건우[O 2]
[1] 선린인터넷고등학교 [2] 충주고등학교
lee@jaeyeong.cc, johnkramer1225@gmail.com

ExplosionGuard: Policy Synthesis and Guardrails for Budget-Constrained Symbolic Execution

JAEYEONG LEE[1], GeonWoo Lee[O 2]
[1]Sunrin Internet High School [2]Chungju High School

요 약

심볼릭 실행은 프로그램 입력을 심볼 변수로 모델링한 뒤, 실행 경로에서 마주치는 분기 조건을 제약식으로 누적하는 기법이다. SMT solver는 각 제약식의 만족 가능성을 판단하며, 만족 가능한 경우 해당 경로로 진입하는 구체 입력을 생성할 수 있다. 그러나 분기와 루프가 많은 코드에서는 상태 수가 빠르게 증가한다. 상태가 늘어나면 제약식도 커지고, 실행 step 비용과 solver 호출 비용이 함께 상승하여 처리량이 떨어진다. 예산이 고정된 환경에서는 커버리지가 정체된 구간에서 자원이 소모되어 탐색이 조기에 종료되기 쉽다.

본 연구는 예산 제약 환경에서도 탐색이 지속되도록 ExplosionGuard를 설계한다. 시작 직후 단기 프로파일링으로 폭발, 비용, 진전 지표를 수집하고, 이를 정책 파라미터로 변환해 초기 정책을 합성한다. 이후 실행 중에는 스냅샷 기반 컨트롤러가 지표 변화를 반영해 정책을 갱신한다. 폭발 및 비용 신호가 강해지면 제약을 강화해 상태 증가를 억제하고, 일정 기간 커버리지가 증가하지 않으면 제약을 완화하며 보류 상태를 승격해 탐색 다양성을 회복한다. 오픈소스 파서 3종 실험에서 휴리스틱 정책은 활성 상태 피크를 낮추면서 처리량과 커버리지 진전을 개선했다.

1. 서론

심볼릭 실행은 입력을 심볼 변수로 두고 프로그램을 따라가며, 분기 조건을 경로별 제약식으로 누적한다. SMT solver로 제약식의 만족 가능성을 확인하면, 만족 가능한 경로에 해당하는 입력을 생성할 수 있다. 이 방식은 테스트 입력 생성과 프로그램 분석에서 널리 활용된다.

문제는 현실의 바이너리가 복잡하다는 점이다. 분기와 루프가 누적되면 실행 경로 수가 조합적으로 증가한다. angr에서는 `SimulationManager.active`의 활성 상태 수가 급증하면서 이 현상이 쉽게 드러난다. 상태가 늘수록 제약식 규모도 커지고, step 수행 시간과 solver 비용이 함께 상승한다. 결과적으로 처리량이 감소하고, 커버리지 증가가 느려진다. 예산이 제한된 환경에서는 정체 구간에서 비용만 소모한 채 탐색이 종료되기 쉽다.

정책의 세기를 잡는 일도 어렵다. 제약이 약하면 상태 폭증으로 예산이 빠르게 고갈된다. 반대로 제약이 강하면 유망 경로가 일찍 잘려 커버리지가 빠르게 정체된다. 따라서 폭발을 억제하면서도, 정체가 발생하면 탐색 다양성을 회복하는 제어가 필요하다.

본 연구는 이를 세 단계로 구성한다. Phase I에서 단기 프로파일링으로 지표를 수집하고 초기 정책을 합성한다. Phase II에서는 정책 적용기와 컨트롤러가 스냅샷을 기반으로 정책을 갱신한다. 또한 정책 적용으로 제외된 상태를 `deferred`에 보류하고, 정체 시 일부를 승격해 탐색을 확장한다.

1.1 기여

본 연구의 기여는 다음과 같다.

- 예산 제약 탐색에서 폭발, 비용, 진전을 저비용 프록시 지표로 관측하는 텔레메트리를 정의한다.
- 지표를 정책 파라미터로 사상하는 규칙과 초기 정책 합성 절차를 제시한다.
- 제약 강화 및 완화를 파라미터의 수치 갱신과 적용 시점으로 정의하여 동작을 명확히 한다.
- `deferred` 보류 및 승격을 활용해 정체를 완화하는 전략을 설계한다.

2. 배경 및 문제 정의

2.1 경로 폭발과 예산 소진

경로 폭발은 보통 활성 상태 수의 급격한 증가로 나타난다. 상태가 늘어나면 상태별 제약식도 비대해지고, solver 부담이 커진다. 이 변화는 처리량 저하로 이어진다. 처리량이 낮아지면 같은 예산에서 수행 가능한 step 수가 감소하고, 커버리지 증가도 둔화된다. 특히 예산이 고정된 환경에서는 정체 구간에서 자원만 소모하고 탐색이 종료될 가능성이 높다.

본 연구의 목표는 명확하다. 정해진 예산에서 커버리지를 최대화하되, 상태 폭증으로 탐색이 멈추지 않도록 제어하는 것이다. 이를 위해 ExplosionGuard는 탐색 공간의 크기와 상태 복잡도를 직접 제어하는 파라미터를 정책으로 둔다.

2.2 정책 제어 대상

본 연구에서 정책은 탐색 공간과 상태 복잡도를 제어하는 파라미터 집합이다. ExplosionGuard는 다음 세 축을 제어한다.

- **분기 확장량**: 한 step에서 생성되는 후속 상태 수
- **루프 확장량**: 루프 반복으로 추가 생성되는 상태 수
- **제약식 복잡도**: 선택적 구체화를 통해 제약식을 단순화하는 정도

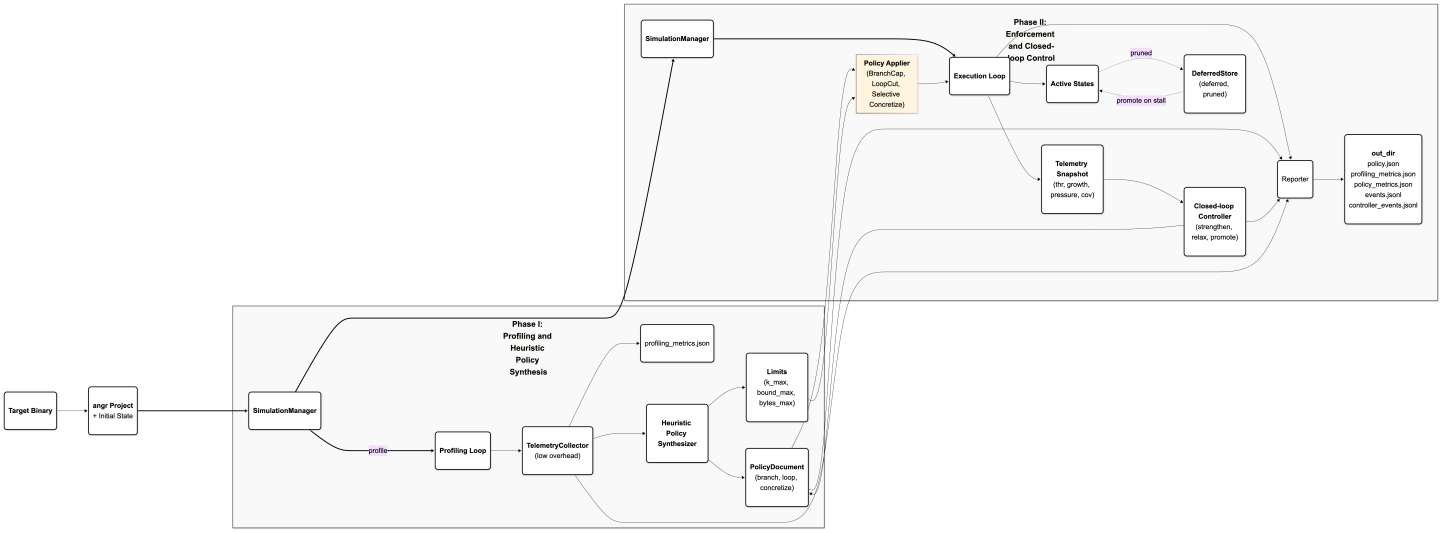


그림 1: ExplosionGuard 구조. Phase I에서 단기 프로파일링으로 초기 정책을 합성한다. Phase II에서 정책 적용기와 컨트롤러가 정책을 적용하고 갱신한다. DeferredStore는 보류, 퇴출, 승격을 담당하며 Reporter는 정책과 지표, 결정을 로그로 기록한다.

컨트롤러는 텔레메트리 스냅샷을 입력으로 받아, 상황에 따라 제약을 강화하거나 완화한다.

2.3 관련 연구

심볼릭 실행의 확장성 문제는 오래전부터 연구되어 왔다. KLEE는 검색 휴리스틱과 쿼리 최적화를 결합해 높은 커버리지를 달성한 대표적 시스템이다 [1]. 실무에서는 활성 상태 제한, 루프 바운딩, 경로 병합 등이 자주 사용된다.

Driller는 퍼징이 정체된 구간에서만 심볼릭 실행을 호출해 비용을 줄이고 퍼징의 한계를 보완한다 [2]. ExplosionGuard는 예산 제약을 전제로 단기 지표 수집, 초기 정책 합성, 실행 중 정책 갱신, 보류 상태 승격을 하나의 제어 루프로 구성한다는 점에서 접근 방식이 다르다.

3. ExplosionGuard 설계

3.1 정책 문서와 파라미터

정책은 JSON 형태의 `PolicyDocument`로 표현한다. 문서는 세 가지 정책 항목을 담는다.

BranchCap은 한 step에서 분기 확장으로 생성되는 후속 상태 수의 상한을 `k`로 둔다. 후속 상태 수가 `k`를 초과하면 점수 상위 `k`개만 `active`로 유지하고, 나머지는 `deferred`로 이동시킨다. 점수는 DeferredStore의 승격 점수와 동일한 기준으로 계산한다.

LoopCut은 루프 헤드 단위로 반복 상한 `bound`를 둔다. 반복 횟수가 `bound`를 초과하면 추가 반복에서 생성되는 상태를 `deferred`로 이동시킨다.

Selective Concretize는 제약식 단순화를 위해 일부 심볼 바이트를 구체화한다. 구체화 강도는 `strategy.k`와 `strategy.bytes`로 표현한다. 대상은 최근 스냅샷에서 비용 신호가 큰 상태부터 선택한다.

모든 파라미터는 `Limits`로 하한과 상한을 둔다. `k`는 `k_min..k_max`, `bound`는 `b_min..b_max`, `bytes`는 `bytes_min..bytes_max` 범위로 제한한다. `deferred` 저장소도 `deferred_max_states`로 용량을 제한한다.

3.2 Phase I: 단기 프로파일링

ExplosionGuard는 시작 직후 단기 프로파일링을 수행한다. 프로파일링 윈도우는 `ProfilingWindow(time_s,steps,max_states)`로 정의하며, 세 조건 중 하나라도 만족하면 종료한다. 비용이 큰 계측을 피하기 위해 다음의 저비용 프록시 지표만 수집한다.

- `throughput`: 초당 수행되는 step 수
- `growth_rate`: 단위 시간당 활성 상태 수 증가량
- `active_peak`: 프로파일링 구간에서 관측된 활성 상태 최댓값
- `step_pressure`: step 1회 수행에 걸리는 평균 시간(초)
- `avg_ast_nodes`: 제약식 규모 프록시로서 AST 노드 수 평균
- `coverage_rate`: 프로파일링 구간에서 새로 도달한 고유 기본 블록 수를 측정 시간(초)으로 나눈 값

`growth_rate`와 `active_peak`는 폭발 위험 신호로, `step_pressure`와 `avg_ast_nodes`는 비용 위험 신호로 해석한다. `coverage_rate`는 탐색 진전의 신호로 사용한다.

3.3 지표에서 정책으로: 초기 정책 합성

Phase I 지표를 초기 정책으로 변환하는 과정의 재현성을 확보하기 위해, ExplosionGuard는 등급화와 사상을 고정 규칙으로 수행한다. 폭발 위험은 `growth_rate`와 `active_peak`로, 비용 위험은 `step_pressure`와 `avg_ast_nodes`로 판단한다. 진전은 `coverage_rate`로 평가한다. 임계값은 고정 상수로 두며 표 1에 정리한다.

폭발 위험이 높으면 `k`와 `bound`를 낮게 설정해 상태 증가를 억제한다. 비용 위험이 높으면 `bytes`를 높게 설정해 제약식 단순화를 강화한다. 진전이 낮고 폭발 및 비용 위험이 모두 높은 경우에는 초기 정책을 한 단계 더 보수적으로 설정한다. 모든 값은 `Limits` 범위로 제한한다.

3.4 Phase II: 제약 강화와 완화

Phase II에서 컨트롤러는 스냅샷을 기반으로 정책 파라미터를 갱신한다. 강화는 폭발 또는 비용 신호가 강할 때 탐색 공간을 줄이고 제약식을 단순화하는 갱신이다. 완화는 정체가 감지될

Algorithm 1 Phase I 초기 정책 합성

Require: profiling metrics M, Limits L, config C, max_states
1: $explode \leftarrow (M.active\_peak \geq C.peak\_ratio \cdot max\_states) \vee (M.growth\_rate > C.growth\_hi)$
2: $cost \leftarrow (M.step\_pressure > C.pressure\_hi) \vee (M.avg\_ast\_nodes \geq C.ast\_hi)$
3: $stallish \leftarrow (M.coverage\_rate < C.coverage\_lo)$
4: Choose $(k_0, bound_0)$ by $explode$ and clamp to $[L.k_{min}, L.k_{max}]$, $[L.b_{min}, L.b_{max}]$
5: Choose $(bytes_0)$ by $cost$ and clamp to $[L.bytes_{min}, L.bytes_{max}]$
6: **if** $stallish$ and $explode$ and $cost$ **then**
7: $k_0 \leftarrow \max(L.k_{min}, k_0 - 1)$
8: $bound_0 \leftarrow \max(L.b_{min}, \lfloor 0.5 \cdot bound_0 \rfloor)$
9: **end if**
10: **return** initial policy $(k_0, bound_0, bytes_0)$

Algorithm 2 Phase II 컨트롤러 갱신

Require: snapshot S, policy $(k, bound, bytes)$, config C, Limits L, max_states
1: $explode \leftarrow (S.growth\_rate > C.growth\_hi) \vee (S.active \geq C.peak\_ratio \cdot max\_states)$
2: $cost \leftarrow (S.step\_pressure > C.pressure\_hi) \vee (S.avg\_ast\_nodes \geq C.ast\_hi)$
3: $stall \leftarrow$ 최근 $C.stall\_steps$ step 동안 신규 고유 기본 블록 도달이 없음
4: **if** $explode$ **then**
5: $k \leftarrow \max(L.k_{min}, k - 1)$
6: $bound \leftarrow \max(L.b_{min}, \lfloor 0.5 \cdot bound \rfloor)$
7: **end if**
8: **if** $cost$ **then**
9: $bytes \leftarrow \min(L.bytes_{max}, bytes + C.bytes\_step)$
10: **end if**
11: **if** $stall$ **then**
12: $k \leftarrow \min(L.k_{max}, k + 1)$
13: $bound \leftarrow \min(L.b_{max}, bound + C.bound\_step)$
14: $bytes \leftarrow \max(L.bytes_{min}, bytes - C.bytes\_step)$
15: Promote top-n states from `deferred` to `active`
16: **end if**
17: **return** updated policy and decisions

때 제한을 일부 풀고 `deferred`에서 상태를 승격하는 갱신이다. 갱신된 정책은 다음 step에서 정책 적용기에 의해 반영된다.

강화 갱신은 다음과 같이 정의한다.

- BranchCap: $k \leftarrow \max(k_{min}, k - 1)$
- LoopCut: $bound \leftarrow \max(b_{min}, \lfloor 0.5 \cdot bound \rfloor)$
- Concretize: $bytes \leftarrow \min(bytes_{max}, bytes + \Delta)$

완화 갱신은 다음과 같이 정의한다.

- BranchCap: $k \leftarrow \min(k_{max}, k + 1)$
- LoopCut: $bound \leftarrow \min(b_{max}, bound + \Delta_b)$
- Concretize: $bytes \leftarrow \max(bytes_{min}, bytes - \Delta)$
- 승격: `deferred`에서 상위 점수 상태 n개를 `active`로 이동

강화와 완화가 같은 step에서 동시에 조건을 만족할 수 있으므로 적용 순서를 고정한다. 강화를 먼저 적용한 뒤, 정체가 참이면 완화를 적용해 일부를 되돌린다. 최종 파라미터는 `Limits` 범위로 제한한다.

3.5 고정 임계값 및 컨트롤러 파라미터

본 실험에서는 Phase I/II 임계값과 컨트롤러 파라미터를 고정 상수로 설정한다. 표 1는 사용한 기본 설정을 정리한 것이다.

임계값은 사전 실험에서 폭발, 비용, 정체 신호를 구분할 수 있는 범위에서 선택한다. 또한 제한된 범위에서 임계값을 조정해도 `off` 대비 `heur`의 경향이 크게 변하지 않음을 확인했다.

표 1: 고정 임계값 및 컨트롤러 파라미터.

Parameter	Value
growth_hi (`growth_rate`)	10.0 (초당 활성 상태 증가량)
pressure_hi (`step_pressure`)	0.05 (step 1회 평균 소요 시간, 초)
coverage_lo (`coverage_rate`)	0.02 (초당 증가한 고유 기본 블록 수)
peak_ratio (활성 상태 과다 기준 비율)	0.8
ast_hi (큰 AST 기준)	4096 (노드 수)
stall_steps	50 (step 수)
bytes_step (Δ)	8 (바이트)
bound_step (Δ_b)	4 (반복 횟수)
promotion n (`deferred`→`active`)	4 (상태 수)

3.6 DeferredStore: 보류, 퇴출, 승격

분기와 루프를 제한하면 탐색 다양성이 감소할 수 있다. ExplosionGuard는 제한으로 제외된 상태를 즉시 폐기하지 않고 `deferred`에 보류한다. `deferred`는 `deferred_max_states`를 넘지 않게 관리하며, 초과분은 점수 하위 상태부터 `pruned`로 이동시킨다.

정체가 감지되면 `deferred`에서 일부 상태를 `active`로 승격한다. 승격 수 n은 재폭발을 피하기 위해 작은 상수로 설정한다(표 1). 승격 점수는 신규 커버리지 기여 가능성을 보상하고 상태 복잡도를 패널티로 반영한다. 예를 들어 상태 s가 새로 도달할 것으로 기대되는 고유 기본 블록 수를 $\Delta_{\text{block}}(s)$로 두면 $score(s) = \Delta_{\text{block}}(s) - \lambda \cdot AST(s) + \mu \cdot depth(s)$로 정의할 수 있다. λ와 μ는 상수이며, 본 실험에서는 고정값을 사용한다.

4. 구현

ExplosionGuard는 angr의 `SimulationManager`에 결합되는 `ExplorationTechnique`로 구현한다 [3]. 실행 루프는 다음 절차를 반복한다. (1) 최신 정책으로 BranchCap, LoopCut, Concretize를 적용한다. (2) `simgr.step()`을 수행한다. (3) 텔레메트리를 갱신하고 스냅샷을 생성한다. (4) 컨트롤러가 정책을 갱신하고 `deferred` 승격을 결정한다. (5) 정책과 결정, 지표를 로그로 기록한다.

Reporter는 `out_dir`에 `policy.json`, `profiling_metrics.json`, `policy_metrics.json`, `events.jsonl`, `controller_events.jsonl`를 기록한다.

5. 실험

5.1 대상 및 환경

오픈소스 파서 3종을 대상으로 평가한다. `inih(v2)`는 inih 저장소의 major 버전 v2 계열을 의미한다.

실험은 Apple Silicon M3, 메모리 36GB 환경에서 수행한다. 각 모드는 동일 예산에서 10회 반복 실행하고 평균을 보고한다. 예산은 `--max-steps200`으로 설정한다.

비교 모드는 `off`(정책 비활성)와 `heur`(휴리스틱 정책 적용)이다. 두 모드는 동일한 바이너리, 동일한 하네스 및 입력 모델, 동일한 angr 옵션, 동일한 step 예산에서 실행한다. `time`은 200 step을 수행하는 데 걸린 실측 시간이다.

본 연구에서 `coverage`는 실행 중 도달한 고유 기본 블록(basic block)의 누적 개수로 정의한다. `coverage_rate`는 프로파일링 구간에서 새로 도달한 고유 기본 블록 수를 측정 시간(초)으로 나눈 값이다. 즉 초당 증가한 고유 기본 블록 수를 의미한다.

표 2: 오픈소스 파서 3종 10회 평균 결과.

Mode	gen	coverage	throughput	time	peak
cJSON					
off	base	4.724	7.663	9.536	216.0
heur	heur	6.699	29.775	6.735	120.0
inih (v2)					
off	base	0.877	4.383	45.747	202.0
heur	heur	3.692	18.455	10.854	147.0
tinyexpr (v2)					
off	base	1.141	5.513	36.956	211.0
heur	heur	2.886	12.545	15.945	156.0

5.2 결과

표 2에서 `heur`는 세 대상 모두에서 활성 상태 피크를 낮추면서 처리량과 커버리지를 개선한다. cJSON에서는 처리량이 증가했고 피크가 감소했으며, 커버리지도 함께 증가했다. inih(v2)에서도 처리량 증가와 실행 시간 감소가 동시에 관측된다. tinyexpr(v2) 역시 유사한 경향을 보이며 커버리지가 증가한다.

이 결과는 다음과 같이 해석할 수 있다. 폭발이 강한 구간에서는 BranchCap과 LoopCut이 상태 증가를 직접 억제한다. 그 결과 동일 예산에서 더 많은 step을 수행할 수 있고, 커버리지 증가로 이어진다. 정체 구간에서는 제한을 일부 완화하고 `deferred`에서 상태를 승격해 탐색 다양성을 확보한다. 이 과정이 커버리지 진전에 기여한다.

5.3 위협 요인 및 한계

본 실험은 파서 3종과 `--max-steps200`이라는 제한된 예산에서 수행했기 때문에 일반화에 한계가 있다. 또한 비용 신호를 `step_pressure`와 `avg_ast_nodes`로 근사했다. solver 내부 시간을 직접 계측하면 비용 모델을 더 정교하게 구성할 수 있다. 마지막으로 임계값과 승격 수 n 같은 하이퍼파라미터는 탐색 안정성에 영향을 준다. 프로그램 특성에 따라 자동으로 보정하는 방법은 향후 과제로 남긴다.

6. 결론

본 연구는 예산 제약 심볼릭 실행에서 경로 폭발과 정체를 함께 다루기 위해 ExplosionGuard를 제안한다. ExplosionGuard는 단기 프로파일링 기반 지표 수집, 지표-정책 사상에 따른 초기 정책 합성, 실행 중 정책 갱신을 결합한다. 또한 `deferred` 보류 및 승격을 통해 정체 시 탐색 다양성을 회복한다. 오픈소스 파서 3종 실험에서 휴리스틱 정책은 활성 상태 피크를 낮추면서 처리량과 커버리지 진전도를 개선했다.

참고문헌

[1] C. Cadar, D. Dunbar, D. Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI, 2008.

[2] N. Stephens et al., Driller: Augmenting Fuzzing Through Selective Symbolic Execution, NDSS, 2016.

[3] Y. Shoshitaishvili et al., angr: A Platform-agnostic Binary Analysis Framework, (project), `https://angr.io/`.

[4] cJSON project, `https://github.com/DaveGamble/cJSON`.

[5] inih project, `https://github.com/benhoyt/inih`.

[6] tinyexpr project, `https://github.com/codeplea/tinyexpr`.

바이트코드 의미 보존을 위한 그래프 생성 기법

권민하○, 정용빈, 정승욱, 정다훈, 남재창
한동대학교
minha20@handong.ac.kr, yongbeanchung@gmail.com, wjdtmddnr123@gmail.com,
wjdekgns3927@gmail.com, jcnam@handong.edu

Graph Generation Method for Preserving Bytecode Semantics

Minha Kwon○, Yongbean Chung, Seungwook Jung, Dahun Jeong, Jaechang Nam
Handong Global University

요 약

최근 Java 바이트코드 사전학습 모델은 소스코드가 부재한 환경에서도 유효한 분석 수단으로 입증받고 있다. 그러나 기존 연구들은 바이트코드를 단순한 토큰 시퀀스로만 처리하여, 프로그램의 실행 의미를 결정하는 명령어 간의 제어 흐름 및 데이터 의존성을 간과한다는 한계가 있다. 또한 현존하는 그래프 분석 도구들은 대부분 중간 표현(IR)으로의 변환을 거치면서 원형 바이트코드의 고유한 16진수(Hex) 정보를 소실하는 문제가 존재한다. 이에 본 연구는 BCEL과 WALA를 결합하여 바이트코드의 16진수 원형 정보와 의미론적 구조를 동시에 보존하는 새로운 그래프 생성 기법을 제안한다.

1. 서 론

최근 Java 바이트코드의 16진수 원형을 활용한 사전학습 모델 연구가 제안되었으며, 소스코드가 부재한 환경에서도 소프트웨어 분석이 가능하다는 점이 입증되고 있다[1]. ByteTok과 같은 최신 기법들은 바이트코드를 토큰화하여 모델의 입력으로 활용함으로써 바이너리 수준에서 유의미한 정보를 추출할 수 있음을 보여주었다.[1] 이러한 흐름은 배포 환경의 특성을 직접 반영할 수 있다는 점에서 바이트코드 기반 분석의 실효성을 높이는 데 기여하고 있다.

그러나 현재의 바이트코드 사전학습 연구들은 대부분 데이터를 단순 토큰 시퀀스로만 취급하며, 소스코드 연구에서 필수적으로 다루어지는 구조적 의존성 모델링을 간과하고 있다[1]. 소스코드 레벨에서는 그래프를 통해 프로그램의 복잡한 논리 구조를 포착하여 모델 성능을 높이는 연구[2]가 보편적이지만, 바이트코드 레벨에서는 여전히 시퀀스 기반의 표면적인 학습에 머물러 있는 실정이다. 프로그램의 실제 실행 의미는 명령어의 나열이 아닌 비선형적인 관계에서 발생하므로, 시퀀스 위주의 학습은 구조적 맥락 파악에 한계를 보일 수밖에 없다.

바이트코드는 개별 명령어 자체보다 명령어 간의 제어 흐름과 데이터 의존성이라는 유기적인 관계를 통해 실행 의미를 가진다[3]. 특히 바이트코드는 고수준 언어에 비해 추상화 수준이 낮기 때문에, 각 인스트럭션이 어떤 경로로 분기되고 데이터가 어떻게 전파되는지를 명시적으로 표현하는 것이 프로그램 이해의 결정적인 단서가 된다[4]. 따라서 바이트코드 모델의 정밀도를 높이기 위해서는 이러한 구조적 특성을 그래프 형태로 추출하여 학습에 반영해야 한다.

기존의 바이트코드 그래프 분석 도구들은 대부분 중간 표현(Intermediate Representation, IR)에 기반하여 생성되므로, 원형 바이트코드가 가진 저수준 16진수(Hex) 의미 정보를 손실하는 한계를 지닌다[5]. Soot나 ASM과 같은 전통적인 도구들은 분석의 편의를 위해 바이트코드를 추상화된 형태로 변환하는데, 이 과정에서 실제 바이너리 값과 그래프 노드 사이의 일대일 대응 관계가 깨지게 된다. 이는 결과적으로 저수준 토큰 정보와 고수준 구조 정보를 통합적으로 처리해야 하는 최신 딥러닝 아키텍처 적용에 큰 장애가 되고 있다.

따라서 본 연구에서는 이러한 공백을 해소하기 위해 BCEL과 WALA를 활용하여 16진수 바이트코드 정보를 직접 담아내는 의미론적 그래프 추출 도구를 제안한다. 제안하는 도구는 원형 바이트코드를 기반으로 제어 흐름 그래프 (Control Flow Graph, CFG), 데이터 흐름 그래프 (Data Flow Graph, DFG), 제어 의존성 그래프 (Control Dependency Graph, CDG), 데이터 의존성 그래프 (Data Dependency Graph, DDG)를 직접 생성하고 저수준 바이너리 특성과 고수준 의미 흐름을 단일 구조 내에 통합한다. 특히 본 연구에서는 기존 도구와 구별되는 두 가지 핵심 기술을 구현하였다. 첫째, WALA와 처리하지 못하는 바이트코드 명령어 수준의 데이터 및 제어 흐름을 BCEL을 활용해 정밀하게 추출하였다. 둘째, 기존 도구에서 분석된 IR 수준의 의존성 정보를, 바이트코드 명령어로 이루어진 노드 수준으로 확장시켰다. 이에 따라 저수준 데이터와 고수준 구조 정보 간의 의미적 불일치를 해결하였다. 이는 단일 바이트코드 명령어를 그래프 노드로 활용하게 하여 기존 도구의 정보 손실 문제를 근본적으로 해결한다. 따라서 향후 바이트코드 사전학습 모델의 정밀도를 극대화하고 소스코드가 부재한 환경에서의 고도화된 프로그램 이해를 가능하게 할 것이다.

본 연구의 기여는 다음과 같다.

- 16진법의 바이트코드 기반으로 그래프를 추출하는 첫 연구
- 바이트코드 그래프를 쉽게 열람할 수 있는 시각화 도구 개발

- Bytecode 16진수 기반 그래프 생성 오픈소스 도구 개발 https://github.com/ISEL-HGU/ByteGraph

2. 관련 연구

2.1 Bytecode 기존 분석 모델 및 연구현황

최근 대형 언어 모델의 발전과 함께 바이트코드의 정형화된 구조에 착안하여 이를 시퀀스로 처리하는 연구가 활발하다. 특히 Kim et al.[1]은 16진수 전용 토크나이저인 ByteTok을 개발하여 byteT5와 byteBERT 모델을 구축하였고, 이는 바이트코드의 물리적 원형을 보존함으로써 소스코드 부재 환경에서의 데이터 가용성 문제를 해결하는 토대가 되었다.

그러나 이러한 시퀀스 기반 접근은 비선형적인 제어 흐름과 데이터 의존성을 명시적으로 표현하는 데 근본적인 제약을 가진다[2]. 프로그램의 실행 의미는 단순한 명령어 나열이 아닌 복잡한 구조적 인과 관계에 의해 결정되므로, 물리적으로 이격된 명령어 간의 논리적 연결을 포착하지 못하는 시퀀스 학습은 고수준 분석에 병목을 야기한다. 따라서 ByteTok이 제공하는 저수준 16진수 정보와 함께, 프로그램의 실행 구조를 온전히 담아낼 수 있는 바이트코드 그래프 표현이 필수적으로 요구된다.

2.2 Intermediate Representation (IR) Graph 도구

스택 기반 바이트코드는 연산과 데이터 흐름이 코드에 명시적으로 드러나지 않고 타입 정보도 부족해 분석과 최적화가 어렵다. 따라서 Soot, WALA와 같은 정적 분석 도구에서는 계산 구조와 데이터 의존성이 명확한 중간 표현(IR)을 사용함으로써 프로그램을 더 효율적으로 분석하였다[5]. 이러한 IR 기반 접근을 바탕으로, 기존 연구들은 CFG, DFG와 같은 구조적 정보를 추출하는 방식을 주로 채택해 왔다[6].

그러나 IR 기반 그래프는 바이트코드의 제어 흐름과 데이터 의존성을 구조적으로 분석하는 데 효과적이지만, IR 변환 과정에서 분석의 편의를 위해 바이트코드의 실제 16진수 값이나 명령어 수준의 물리적 정보를 추상화하거나 생략한다[3]. 반면, 바이트코드 학습 연구에서는 hex 기반 토큰 시퀀스를 모델 입력으로 사용함으로써 바이트코드의 물리적 표현을 보존하고 있다. 이로 인해 hex 시퀀스를 입력으로 사용하는 학습 모델과 IR 기반 그래프 표현을 일관되게 결합하기 어려운 구조적 간극이 존재한다. 이러한 배경에서, hex 표현을 유지하면서도 제어 흐름을 구조적으로 담아낼 수 있는 바이트코드 그래프 표현의 필요성이 제기된다.

3. Bytecode 그래프 생성 기법

3.1 개요

본 연구에서는 바이트코드 생성도구인 ByteGraph를 제안한다. 이 도구는 자바 바이트코드의 저수준 정보와 논리적 의존성을 통합하기 위해 Apache BCEL(Byte Code Engineering Library)과 IBM WALA(Watson Libraries for Analysis)를 상호보완적으로 활용한다. WALA는 수많은 선행 연구를 통해 학술적 신뢰성이 검증된 정적 분석 프레임워크로, 분석 결과의 이론적 타당성을 엄격히 입증할 수 있는 환경을 제공한다.

분석 과정은 그림 1에 표현되어 있다. 분석은 메서드 단위로 진행한다. 명령어의 물리적 속성은 BCEL을 이용하여 추출하고, 의미론적 의존성은 WALA를 이용해 계산한다. 두 분석 결과는 바이트코드 오프셋으로 투영되어 융합된다. 최종적으로 이 도구는 저수준 바이너리 정보와 정밀한 의존성 구조가 결합된 JSON 모델을 생성한다.

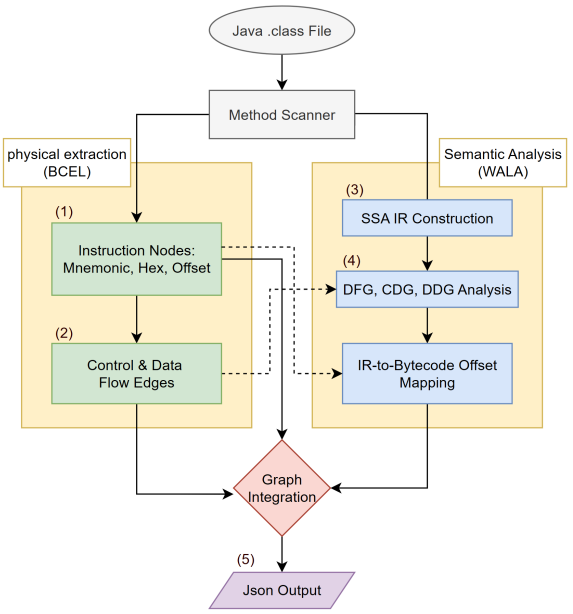


그림 1 : 바이트코드 기반 그래프 추출 흐름도

3.2 분석 절차

본 도구의 분석 과정은 다음과 같이 5단계로 수행된다. 이는 명령어 위치를 정확하게 식별하고 논리적 의존성을 추출하기 위함이다.

3.2.1 바이트코드 및 노드 정보 추출

본 연구는 그래프 노드 데이터 확보를 목적으로 메서드 단위의 Code 속성을 추출한다. 이때 BCEL 라이브러리를 사용한다. 그림 1의 (1)에서 볼 수 있듯, 각 명령어를 순회하며 오프셋 (Offset), 니모닉 (Mnemonic), 피연산자 (Operands) 정보를 수집한다. 특히, 저수준 분석을 위해 원본 바이트열을 16진수 문자열(Hex)로 변환하여 노드 정보에 포함한다.

3.2.2 제어 흐름 및 물리적 데이터 흐름 추출

이 단계는 그림 1의 (2)에 해당하며, 제어 흐름과 물리적 데이터 흐름을 추출한다. 제어 흐름 그래프(CFG)는 명령어 간의 실행 경로를 나타낸다. 제어 흐름은 정상 실행 흐름과 예외 처리 흐름으로 나누어 구축한다.

정상 실행 흐름은 명령어의 성격에 따라 순차 실행(Fall-through) 또는 분기로 처리된다. 분기 명령어 (Goto, Return, ATHROW, Select, IfInstruction)의 경우, 명령어의 피연산자(Operand)에 기록된 목적지 주소 정보를 참조하여 도착지를 결정한다. 예외 처리 흐름은 바이트코드 내부에 정의된 예외 테이블(Exception Table) 정보를 활용한다. 엣지의 출발지는 예외를 감시하는 범위(startPC부터 endPC까지) 내에서 예외가 발생할 가능성이 있는 모든 명령어이다. 도착지는 예외 핸들러(handlerPC) 위치이다.

물리적 데이터 흐름은 레지스터 및 스택 수준에서 추출된다. 먼저 BCEL을 활용하여 로컬 변수 슬롯의 LOAD-STORE 관계를 추적한다. 그리고 스택의 생산·소비 과정을 추적하여 엣지를 생성한다. IR을 사용한 의미적 분석 과정에서는 스택 수준의 세밀한 데이터 흐름이 유실될 수 있다. 본 연구는 바이트코드 명령어 수준의 분석을 통해, WALA만으로는 놓칠 수 있는 세밀한 데이터와 제어 흐름을 추출했다.

3.2.3 SSA(Static Single Assignment) 변환 및 오프셋 매핑

정밀한 의미적 분석을 위해 WALA를 사용하여 바이트코드를 SSA기반의 IR로 변환한다. SSA는 모든 변수에 값이 단 한 번만

할당되는 표현식으로, 복잡한 데이터 흐름과 의존성 계산에 적합하다.

그림 1의 (3)단계에서 가장 중요한 것은 WALA의 IR 인덱스와 바이트코드 오프셋 간의 매핑 테이블을 작성하는 것이다. SSA 인덱스와 인접 바이트코드 오프셋을 통합하는 1:N 매핑 기법으로 매핑 단계에서의 누락을 방지한다. 이렇게 생성된 매핑 테이블은 이후 단계에서 산출된 데이터 흐름 및 의존성 분석 결과를 바이트코드 단위로 연결하는 역할을 한다.

3.2.4 DFG, CDG, DDG 생성

앞선 3.2.3절에서 구축한 매핑 테이블을 바탕으로, 그림 1의 (4)단계에서는 WALA를 이용해 데이터 흐름 그래프(DFG), 데이터 의존성 그래프(DDG), 제어 의존성 그래프(CDG)를 생성한다. 이때, JDK 표준 라이브러리가 사용된 바이트코드를 정밀하게 분석하기 위해 해당 라이브러리 파일(rt.jar, jce.jar)을 분석 범위 안에 포함시켰다.

데이터 흐름 그래프 (DFG) 생성: 데이터 흐름 그래프(DFG)는 데이터의 생성 및 소비 경로를 나타낸다. 이를 만들기 위해 (2)단계에서 추출된 명령어 수준의 물리적 흐름과 현 단계에서 분석하는 변수 중심의 논리적 흐름을 결합한다.

논리적 분석은 WALA의 DefUse 객체를 활용한다. DefUse는 변수가 정의된 명령어와 사용된 명령어 사이의 관계를 관리한다. 모든 IR 명령어를 순회하며 정의된 위치에서 사용된 위치로 엣지를 만든다. 이는 매핑 테이블을 통해 원본 바이트코드 오프셋 단위로 변환된다.

최종적으로, 두 수준의 흐름을 병합하여 데이터 흐름 그래프를 완성한다. 다층적 결합을 통해, 단순한 SSA 분석만으로는 놓칠 수 있는 바이트코드의 세밀한 데이터 이동까지 모두 포함하는 고정밀 통합 DFG가 산출된다.

제어 의존성 그래프 (CDG) 생성: 제어 의존성 그래프(CDG)는 특정 지점에서의 분기 결정에 의해 다른 명령어의 실행 여부가 제어되는 관계를 나타낸다. 본 도구는 WALA의 SSACFG를 바탕으로 Ferrante[7]가 제안한 정식 제어 의존성의 정식 정의를 따른다.

본 시스템은 고정점 (Fixed-point) 알고리즘으로 각 블록의 역우세(Post-Dominator) 집합을 계산하여, 두 노드 사이에 제어 의존성이 존재하는지 판단한다. 또한 조건문뿐만 아니라 예외 처리 과정에서 발생하는 제어 의존성도 포함한다. 만들어진 엣지는 매핑 테이블을 통해 바이트코드 오프셋 단위로 변환된다.

데이터 의존성 그래프 (DDG) 생성: 데이터 의존성 그래프(DDG)는 데이터의 생성 및 참조 관계에 따른 명령어 간의 선행 제약을 나타낸다.

본 도구는 WALA Slicer의 FULL 옵션을 적용하여, 값이 전달되는 Flow 의존성뿐만 아니라 선행 제약을 나타내는 Anti 및 Output 의존성까지도 식별한다. 특히 로컬 변수를 넘어 힙(Heap) 메모리 상의 객체 필드 간 의존성을 정밀하게 포착하기 위해 Zero-CFA 기반의 포인터 분석과 ModRef 분석을 병행한다. 이렇게 계산된 의존 관계는 매핑 테이블을 거쳐 바이트코드 단위로 투영된다.

3.2.5 결과물 출력

그림 1의 (5)단계에서 메서드 단위로 JSON 파일이 생성된다. 최종 결과물은 nodes와 edges 섹션으로 나뉜다. nodes에는 명령어의 물리적 속성(Hex, Mnemonic 등)이 포함된다. edges에는 CFG, EX, DFG, DDG, CDG의 연결 정보가 바이트코드 오프셋(src, dst)을 기준으로 저장된다.

4. Case Study

4.1 결과물

본 절에서는 제안한 도구를 통해 추출된 Json 파일의 자바 바이트코드 그래프 결과를 제시한다. 결과를 보여주기 위한 예시로 Inversoft Prime JWT를 사용했다. 그림 2와 3은 각각 JWTEncoder getInstance 메서드의 노드와 엣지 예시를 보여준다. 그림 4는 해당 메서드에서 만들어진 네 가지 그래프를 시각화한 것이다.


```
"method" : "getInstance()Lorg/primeframework/jwt/JWTEncoder;",
"nodes" : [ {
  "offset" : 0,
  "hex" : "B2 00 07",
  "mnemonic" : "GETSTATIC",
  "operands" : "org.primeframework.jwt.JWTEncoder.instance"
}, {
  "offset" : 3,
  "hex" : "C7 00 0D",
  "mnemonic" : "IFNONNULL",
  "operands" : "-> 0x0010"
}, {
```


그림 2 : 추출된 그래프 노드 정보(JSON)


```
"edges" : {
  "cfg" : [ {
    "src" : 0,
    "dst" : 3
  }, {
    "src" : 3,
    "dst" : 16
  }, {
```


그림 3 : 추출된 그래프 엣지 정보(JSON)

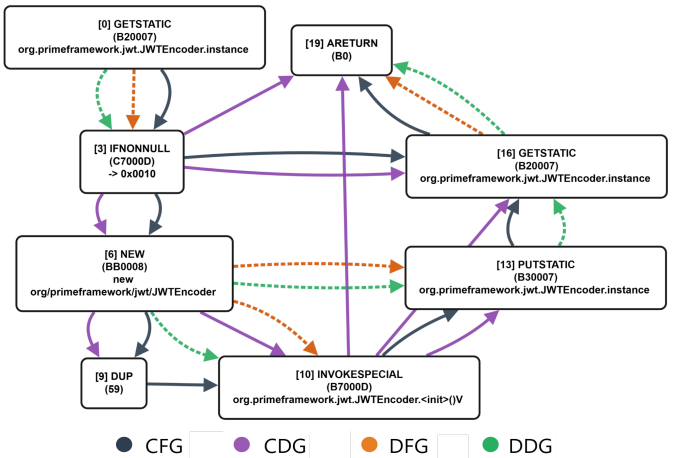


그림 4 : 추출된 그래프 정보 시각화

4.2 결과 분석

본 연구의 실험 대상으로 ANTLR 3.1.3 (Another Tool for Language Recognition)[8]을 선정하였다. ANTLR는 복잡한 상태 머신과 대규모 분기 로직을 포함하여 제어 의존성 검증에 적합하며, 오퍼랜드 스택 활용이 빈번하여 저수준의 물리적 데이터 흐름을 평가하기에 최적의 환경을 제공한다.

ANTLR 라이브러리에서 12개의 클래스, 총 164개의 메서드에서 6,785개의 노드가 생성되었다. 표 1은 이에 따른

20,564개 엣지의 통계를 보여준다. 본 연구는 추출된 엣지를 바이트코드에 직접 기술된 물리적(Physical) 엣지와 정밀 정적 분석을 통해 식별된 의미적(Semantic) 엣지로 구분하여 분석하였다. 예외 경로(EX)는 명령어 내 분기 정보가 없어 물리적 엣지는 존재하지 않으며, 예외 테이블을 통한 의미적 엣지로만 존재한다.

표 1: 물리적 및 의미적 엣지 통계

그래프	엣지 수	물리적 엣지	의미적 엣지
CFG	7,074	6,423	651
EX	906	-	906
DFG	2,313	1,507	806
DDG	2,399	2,313	86
CDG	7,872	2,707	5,165
계	20,564	13,010	7,554

먼저 제어 흐름(CFG) 및 예외 경로(EX) 분석 결과, 100%의 물리적 무결성을 입증하였다. 모든 분기 명령어의 목적지 주소를 바이트코드 오퍼랜드 값과 대조하여 계산된 타겟이 실제 그래프 엣지와 일치함을 전수 검증하였다.

데이터 흐름에서는, 엣지와 저수준 명령어의 정의-사용 관계가 일치하는 비율이 84.39%였다. 고수준에서는 클래스 내 변수에 값을 저장하고 읽는 필드 의존성(357건)과 여러 메서드 호출이 연속될 때 스택을 통해 값이 전달되는 메서드 체이닝(449건)이 식별되었다. 또한, DDG는 일반적인 데이터 흐름에서 놓치기 쉬운 힙(Heap) 메모리상의 고유 의존성 86건을 추가로 포착하여 정밀도를 보강하였다.

제어 의존성(CDG)의 경우, 전체 엣지의 65.6%(5,165건)가 예외 발생 가능 지점(PEI)에서 기인한 의미적 의존성으로 나타났다. 이는 실행 중 예외가 발생할 때 후속 코드의 실행이 중단되는 실제 런타임의 제어 구조를 정밀하게 반영한 결과이다.

결과적으로 본 도구는 전체 엣지의 59.74%가 고수준 의미론적 의존성으로 구성되어 있음을 확인했다. 이는 물리적 실행 위치 정보에 불과했던 바이트코드 노드들이 풍부한 논리적 맥락과 연결되었음을 의미한다. 본 도구는 16진수 바이트코드 데이터와 5종 의미 구조를 단일 모델 내에 성공적으로 통합함으로써, 프로그램의 물리적 특성과 논리적 맥락을 동시에 학습할 수 있는 데이터셋 생성 역량을 입증하였다.

5. 결론 및 향후 연구

본 연구는 Java 바이트코드의 물리적 정보와 논리적 의존성을 단일 구조로 통합하는 그래프 생성 기법을 제안하였다. 제안한 도구는 기존 IR 기반 분석의 한계인 16진수 데이터 손실 문제를 해결했다. 또한 SSA 기반의 정밀한 의존성을 바이트코드 오프셋 레벨로 완벽히 투영하였으며, SSA 분석만으로는 누락될 수 있는 바이트코드 명령어 수준의 세밀한 흐름을 포함하였다.

프로그램의 구조적 정보를 반영한 그래프 표현형은 단순 시퀀스 모델이 포착하기 어려운 복합적인 제어 흐름과 데이터 전파 경로를 식별하는 데 필수적이다. 이는 최근 바이너리 보안 연구[4]와 소스코드 분석 모델인 GraphCodeBERT[2] 등의 사례가 보여주었다. 본 도구는 예외 처리 경로와 힙(Heap) 의존성 정보를 추출한다. 따라서 데이터의 흐름을 정밀하게 추적해야 하는 오염 분석(Taint Analysis)[9]과 같은 보안 분석 기법의 성능을 높이는 데 결정적인 단서를 제공할 수 있다. 이는 기존 시퀀스 기반 모델이 간과하기 쉬운 비정상적 실행 경로상의 취약점을 탐지하기 위한 고품질 학습 데이터셋으로서 높은 실용적 가치를 지닌다.

향후에는 본 도구를 활용해 대규모 바이트코드 취약점 데이터셋을 구축하고, 실제 소프트웨어 보안 분석 환경에서 기존 시퀀스 기반 탐지 모델 대비 구조적 그래프 정보가 취약점 식별 정확도 및 미탐률 개선에 미치는 영향을 정량적으로 검증할 계획이다.

※ 본 연구는 2025년 과학기술정보통신부 및 정보통신기획평가원의 SW중심대학사업(2023-0-00055)과 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원(RS-2024-00457866)을 받아 수행된 연구임

6. 참조문헌

[1] D. Kim, T. Kim, J. Shin, S. Wang, H. Choi and J. Nam, "Pre-trained Models for Bytecode Instructions," *IEEE Conference on Software Testing, Verification and Validation (ICST)*, Napoli, Italy, 2025, pp. 608-612, doi: 10.1109/ICST62969.2025.10989012. (*2025*).

[2] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... & Zhou, M. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*. (2020).

[3] Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., & Leather, H. Programl: Graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536*. (2020).

[4] He, H., Lin, X., Weng, Z., Zhao, R., Gan, S., Chen, L., ... & Xue, Z. Code is not Natural Language: Unlock the Power of {Semantics-Oriented} Graph Representation for Binary Code Similarity Detection. In *33rd USENIX Security Symposium (USENIX Security 24)* (pp. 1759-1776). (2024).

[5] Prakash, J., Tiwari, A., & Hammer, C. Effects of program representation on pointer analyses—an empirical study. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 240-261). Cham: Springer International Publishing. (2021, March).

[6] Allamanis, M., Brockschmidt, M., & Khademi, M. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*. (2017).

[7] Ferrante, Jeanne, Karl J. Ottenstein, and Joe D. Warren. "The program dependence graph and its use in optimization." ACM Transactions on Programming Languages and Systems (TOPLAS) 9.3 : 319-349. (1987)

[8] ANTLR. *antlr/antlr3: antlr v3 repository* [Computer software]. GitHub. https://github.com/antlr/antlr3. (2022)

[9] Sun, M., Wei, T., & Lui, J. C. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (pp. 331-342). (2016, October).

반응형 시스템을 위한 LLM 기반 상태머신 생성 기법 및 성능평가

최승빈[1], 김요엘[2], 최윤자[2†]
경북대학교 소프트웨어재난연구센터[1], 경북대학교 컴퓨터학부[2]
csb8226@naver.com, kimyoel2305@gmail.com, yuchoi76@knu.ac.kr

LLM-Based State Machine Generation Technique for Reactive Systems and Its Performance Evaluation

Seungbin Choi[1], Yoel Kim[2], Yunja Choi[2†]
Software Disaster Research Center, Kyungpook National University[1]
School of Computer Science & Engr., Kyungpook National University[2]

요 약

본 연구에서는 C 코드를 입력받아 LLM을 통해 상태머신을 자동 생성하는 기법(LLM_State_Generate)을 제안하고, 생성된 상태머신을 CBMC 형식 검증을 통해 확인한 뒤 능동학습 기법과 성능을 비교 평가하였다. Simulink 예제 45개 프로그램을 대상으로 실험한 결과, 제안한 방식인 LLM_State_Generate는 단일 생성 기준 프로그램 단위 46.7%의 성공률을 보인 반면 반복적 정제를 포함한 기존의 정형적 능동학습을 통한 상태머신 생성 기법은 93.3%의 성공률을 달성했다. 본 연구는 LLM 기반 접근법의 한계와 개선 방향을 제시한다.

1. 서 론

상태머신(State Machine)은 시스템의 상태와 전이를 정의하는 수학적 모델로, 높은 신뢰성이 요구되는 임베디드 시스템, 반응형 시스템, 제어 시스템 개발에 필수적으로 사용되나, 이러한 상태머신을 수작업으로 작성하는 것은 어렵기 때문에 상태머신 없이 개발되는 경우가 많다. 그래서 개발된 소스코드로부터 상태머신을 학습하는 연구가 진행되었다[1, 2].

상태머신을 정형적 능동학습(Active Learning)을 통해 생성하는 기법[1]은 형식적 방법을 기반으로 하여 높은 정확성을 보장한다. 그러나 능동학습은 엄밀한 모델 정제 과정으로 인하여 상태머신을 생성하는 데 많은 비용이 소요된다. 반면, LLM을 활용한 자연어 기반 접근은 능동학습 기법 대비 빠르게 모델을 생성할 수 있다. 최근 LLM은 테스트 코드 생성, 스펙 생성, 자동 오류 수정 등 다양한 곳에서 활용되고 있다[3,4]. 그러나 상태머신 자동 생성에 적용된 사례는 없으며 본 논문이 최초 시도이다. 다만 LLM 기반 접근은 상태 전이를 누락하거나 잘못된 조건을 만들어내는 등 논리적 정확성과 안정성 측면에서 한계가 존재할 수 있다.

이에 본 연구에서는 C 코드와 프롬프트를 LLM에게 입력해 상태머신을 자동 생성하는 기법(LLM_State_Generate)을 제안한다.

2. LLM 기반 상태머신 생성

2.1. 기법 개요 및 흐름

본 연구에서 제안하는 기법은 C 소스 코드로부터 상태머신의 자동 생성이다. 핵심은 LLM의 패턴 인식을 활용해 입력 대상인 C 코드와 관심 변수를 제공해 상태머신으로 변환하는 것이다.

상태머신 시각화 예시는 그림 1과 같다.

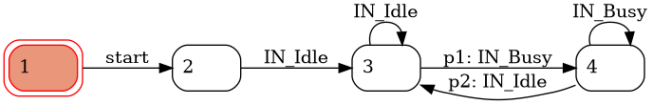


그림 1. 상태머신 예시

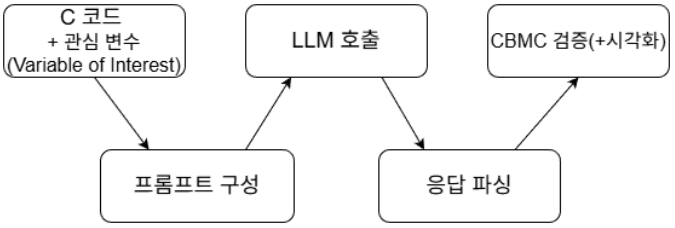


그림 2. LLM_State_Generate의 전체 기법 흐름

본 기법은 먼저 그림 2와 같이 C 코드와 관심 변수를 입력으로 받아 사전 구성한 프롬프트와 함께 LLM에 전달해 상태머신 전이 리스트의 생성을 요청한다. LLM의 응답을 받은 후에는 LLM이 생성한 상태머신의 전이 리스트의 형식 검증 후 파싱하며, 파싱한 전이 리스트를 이미지로 변환하여 사용자가 상태머신 생성이 잘 이루어졌는지 육안으로 확인하기 용이하도록 한다. 마지막으로 생성한 전이 리스트를 CBMC(C Bounded Model Checker)[5]로 모델 검증을 실시하여 상태머신이 C 코드를 올바르게 반영하고 있는지 확인한다.

2.2. LLM 기반 생성 방법

본 연구의 실험 데이터셋인 C 코드들은 변수형과 변수 선언부, 함수선언, 반응형 시스템의 주요 제어논리 등으로 구성되어 있다.

본 기법에서 사용된 프롬프트의 핵심 내용은 다음과 같다.

1. 상태머신 생성 작업을 한다고 프롬프트 도입부에 명시한다.
2. 생성 요구사항(연속적인 상태 ID 사용 규정, 초기 상태는 1로 고정, 고립된 상태 없음)과 전이 형식 규칙을 알려준다.
3. 전이 형식은 [현재상태, '가드: 동작', 다음상태]로 정의되며 조건식은 동일 상태에서 다중 전이가 있을 때만 포함하여 조건식 오류를 최소화하며 상태머신의 일관성을 보장할 수 있도록 한다.

해당 프롬프트로 생성되는 그림 1 예시에 대한 상태머신 전이 리스트는 다음과 같다: "[[1, 'start', 2], [2, 'IN_Idle', 3], [3, 'IN_Idle', 3], [3, '![InputTask == 0]: IN_Busy', 4], [4, 'IN_Busy', 4], [4, '![TaskTime > Counter]: IN_Idle', 3]]"

LLM 모델은 GPT-5를 사용했으며, GPT-5는 API 응답 설정이 Reasoning Effort와 Verbosity로 구성되며, 각각 기본값인 medium으로 설정하여 LLM의 기본 성능을 확인하고자 하며, 각 케이스 별 수행은 1회 수행한다.

2.3. 모델 생성 성공여부 평가 방법

모델 생성 성공 여부는 CBMC[5]를 사용하여 확인한다. CBMC는 C 프로그램의 형식 검증 도구로, 프로그램이 특정 명세(assertion)를 만족하는지 자동으로 확인한다. 일반 테스트가 몇 가지 입력만 검증하는 반면, 형식 검증은 모든 실행 가능한 경로를 수학적으로 분석한다.

검증 방식은 다음과 같다. LLM이 생성한 상태머신 전이 리스트 $[[s, 'g: a', s'], ...]$를 기반으로 assertion으로 변환하여(예: $assert(!(state == s \&\& g) || (next\_state == s'))$) 원본 C 코드에 삽입한다. CBMC를 이용하여 C 코드가 전이 리스트 기반으로 생성된 assertion 들을 모두 만족하는지 검증하며, 모든 assertion들이 만족되면 생성된 상태머신이 원본 코드와 동등한 상태 전이 구조를 가졌다고 볼 수 있다.

3. 실험 설계

3.1. 데이터셋

본 연구에서는 LLM 상태머신 생성 기법의 성능을 객관적인 확인을 위해 Simulink C 코드 변환 예제[1]를 실험 대상으로 선정하였다. 해당 데이터셋은 단순한 구조부터 복잡한 상태 전이까지 폭넓은 복잡도 분포가 존재하며, 자동차 제어, 산업 로봇 등 실제 산업에서 사용되는 제어 로직을 포함하므로 성능 평가에 적합한 표준 벤치마크이다[1].

3.2. 평가 방법

3.2.1. 성공률 평가 기준

LLM_State_Generate와 능동학습 기반 생성 기법의 성능 비교를 위해 두 기법의 상태머신 생성 성공률을 두 가지 기준으로 평가하였다.

첫 번째는 예제 단위 성공으로 45개 예제 프로그램 각각에 대해 예제별로 생성된 assertion이 전부 통과하면 성공으로 분류한다.

두 번째는 전체 실험에서 생성된 모든 assertion 중 검증을 통과한 비율을 측정한다. 케이스 단위로는 실패했더라도 부분적으로 정확한 전이가 있을 수 있으므로, 기법의 잠재적 성능을 평가하는 지표가 된다.

4. 실험 결과

4.1. 전체 성능 비교 (RQ1)

본 연구에서는 45개 예제에 대해 두 기법의 상태머신 생성 성공률을 측정하였다(표 1). 실험 결과, 능동학습 기반 생성 기법은 예제 단위 93.3%, 전체 assertion은 97.5% 성공률을 기록하여 형식적 기법 기반 도구의 높은 신뢰성을 입증하였다. 반면 LLM_State_Generate는 예제 단위에서 46.7%의 성공률을 보여 단순 자연어 기반 접근만으로는 구조적 정확성을 보장하기 어려운 것으로 보이나 전체 assertion에 대한 CBMC 검증 성공률을 보면 LLM의 성공률이 78.1%까지 올라가는 것으로 나타났다. 해당 결과를 통해 LLM 생성 성능이 결코 낮지 않다는 점을 알 수 있다. 그리고 능동학습 기반 생성 방법은 시간 제한(Time-out)을 1시간으로 잡고 진행되었으나, LLM_State_Generate는 전체 예제 중 가장 오래 걸린 케이스가 200초를 넘지 않았다. 이를 통해 생성 속도가 능동학습 기법 대비 상당히 빠르다는 것을 확인할 수 있었다.

표 1. 기법별 성공률 비교

기법	예제 단위	전체 assertion
능동학습 기반 [1]	42/45(93.3%)	384/394(97.5%)
LLM_State_Generate	21/45(46.7%)	349/447(78.1%)

5. 논의 및 결론

본 연구에서는 LLM 기반 상태머신 생성 기법의 성능을 능동학습 기반 생성 기법과 비교 평가하고, 복잡도가 성능에 미치는 영향을 정량적으로 분석하였다. 실험 결과, LLM은 프로그램 단위에서 46.7%의 성공률을 기록하여 93.3%의 성공률을 보인 능동학습 기반 생성 기법에 비해 성능 격차가 확인되었으며, 특히 상태 수와 전이 수가 증가할수록 실패율이 뚜렷하게 증가함을 확인하였다. 다만, assertion 단위로 평가했을 때는 LLM은 성공률 78.1%, 능동학습은 97.5%로 성능 격차가 상대적으로 완화되어 단일 생성 방식으로도 일정 수준의 검증 조건을 추출할 수 있음을 보여준다.

본 연구는 제한된 데이터셋을 사용했으나 단일 생성만으로 예제 단위 46.7%(assertion 단위 78.1%)의 성공률을 달성하였으며 LLM 기반 코드 생성의 정량적 한계를 규명했다는 점에서 의의가 있다. 향후 연구를 통하여, LLM 기반 반복 정제의 효율성과 정확도 향상 정도를 고찰하고, LLM 기반 상태머신 생성이 기존 정형적 방법의 실용적 대안으로 활용될 수 있을지 확인할 계획이다.

Acknowledgement

이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(RS-2021-NR060080).

참고문헌

[1] Jeppu, N. Y., Melham, T., & Kroening, D., "Enhancing active model learning with equivalence checking using simulation relations," Formal Methods in System Design, vol. 61, no. 2–3, pp. 164–197, 2022.

[2] A. Grosche, B. Igel, and O. Spinczyk, "Transformation-and Pattern-based State Machine Mining from Embedded C Code," in Proceedings of the 15th International Conference on Software Technologies (ICSOFT), 2020, pp. 43-54.

[3] Le-Cong, T., Manh, D. T., Inoue, S., and Ghosh, S., "SpecGen: Automated Generation of Formal Program Specifications Assisted by Large Language Models," in Proceedings of the 47th International Conference on Software Engineering (ICSE 2025), 2025.

[4] Fan, Z., Gao, X., Mirchev, M., Marinescu, A., and Tipson, Y., "Automated Repair of Programs from Large Language Models," in Proceedings of the 45th International Conference on Software Engineering (ICSE 2023), pp. 947–958, 2023.

[5] Clarke, E., Kroening, D., and Lerda, F., "A tool for checking ANSI-C programs," in Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 2988, pp. 168-176, Springer, 2004.

바이트코드 기반 Bugram 기법의 성능 평가

추새벽, 남재창
실버든든, 한동대학교
dawnsaebyeokchu@gmail.com, jcnam@handong.edu

Performance Evaluation of A Bytecode-based Bugram Approach

Saebyeok Chu, Jaechang Nam
Silverdndn, Handong Global University

요 약

본 연구는 소스코드 확보가 어려운 환경에서도 통계적 오류 탐지가 가능한지를 검증하기 위해 바이트코드 기반 Bugram 기법의 성능을 실험적으로 평가한다. ASM 프레임워크를 활용하여 추출한 opcode 시퀀스에 n-gram 기반 Bugram을 적용하였다. 실험 결과, 해당 기법은 명령어 시퀀스 자체에 이상이 존재하는 오류 유형에 대해서는 소스코드 기반 모델과 유사한 탐지 성능을 보였다. 이는 바이트코드 표현이 제공하는 구조적 일관성이 통계적 희소성 기반 오류 탐지에 활용될 수 있음을 시사한다. 하지만 의미적 불일치나 제어 흐름의 의미론적 오류에 대해서는 명령어 시퀀스 수준의 통계 정보만으로는 탐지에 한계를 보였다. 이러한 한계를 극복하기 위해, 향후 연구에서는 제어 흐름 정보나 실행 문맥을 추가로 반영하는 확장된 바이트코드 표현을 고려할 수 있을 것이다.

Abstract

This study evaluates a bytecode-based Bugram approach for statistical bug detection where source code is unavailable. We applied n-gram models to opcode sequences extracted via the ASM framework. Results show that the bytecode's structural consistency enables anomaly detection based on data sparsity, achieving performance comparable to source-code models for instruction-level bugs. While limited in capturing complex semantic or control-flow errors, this work suggests that bytecode-level analysis provides a stable statistical baseline. Future models incorporating execution context could further extend these detection capabilities.

1. 서 론

오류 탐지 기법은 소프트웨어 개발 과정에서 품질과 안정성을 확보하기 위한 핵심 요소이다. Wang et al.[1]의 연구에서는 정적 분석, 동적 분석, 테스트 기반 탐지, 로그 기반 탐지 등 다양한 오류 탐지 기법이 제안되어 왔으며, 이 중 규칙 기반 버그 탐지 방식은 비교적 손쉽게 적용할 수 있다는 장점으로 널리 활용되고 있다. 그러나 이러한 방식은 사전에 정의되지 않은 오류를 탐지하는 데 한계를 가진다.

이러한 한계를 보완하기 위해 Bugram 기법은 프로그램의 토큰 시퀀스를 기반으로 한 통계적 오류 탐지 접근 방식을 제안하였으며[1], 기존 규칙 기반 방법으로는 탐지하기 어려웠던 오류를 효과적으로 검출할 수 있음을 보였다. 한편, Choi와 Nam[2]의 연구에서는 바이트코드 명령어 시퀀스 역시 통계적으로 구별 가능한 자연스러움 특성을 지님을 보임으로써, 프로그램 표현 수준에 따른 오류 탐지 가능성을 시사하였다.

그러나 Wang et al.[1]의 연구는 소스코드 기반 분석에 초점을 맞추고 있어 바이트코드 수준에서도 동일한 오류 탐지 특성을 유지할 수 있는지에 대한 실증적 검증은 이루어지지 않았다.

이에 본 연구에서는 Bugram 기법을 바이트코드 기반으로 확장 적용하고, Wang et al.[1]의 연구에서 사용된 실제 오류 사례를 대상으로 바이트코드 기반 Bugram의 탐지 가능성과 한계를 유형별로 분석한다.

본 연구의 주요 기여는 다음과 같다.

- 바이트코드 기반 통계적 버그 탐지의 최초 성능 평가 : 소스코드 기반 Bugram 기법을 바이트코드 수준으로 확장 적용하고, 실제 오류 사례를 대상으로 탐지 성능을 실증적으로 분석하였다.
- 데이터 희소성 (Data Sparsity)[2] 완화 가능성 제시 : 바이트코드가 제공하는 특성이 식별자 수준의 가변성으로 인한 데이터 희소성 문제를 완화할 수 있음을 확인하였다.
- 오류 탐지 기법의 실용적 활용 범위 : 바이트코드 기반 Bugram이 명령어 수준의 이상 탐지에는 효과적이나, 의미론적 오류에는 한계가 있음을 실증하였다.

2. 관련 연구 및 배경

2.1 Bugram 기반 오류 탐지

Wang et al.은 프로그램의 토큰 시퀀스에 n-gram 모델을 적용하여 각 토큰의 출현 확률을 계산하고, 통계적으로 희귀한

패턴을 잠재적인 오류로 간주하는 Bugram 기법을 제안하였다[1]. 이 기법은 프로그램 내부의 반복적인 패턴을 학습하여, 빈도수가 높은 시퀀스를 정상적인 코드로, 빈도수가 낮은 시퀀스를 오류 가능성이 높은 코드로 판단한다.

해당 연구에서는 실제 오픈소스 프로젝트를 대상으로 Bugram 기법을 적용하여 총 25개의 오류를 발견하였으며, 이 중 23개는 기존의 정적 분석 도구(FindBugs 등)가 탐지하지 못한 새로운 오류로 보고되었다. 이러한 결과는 통계적 패턴 분석이 기존 규칙 기반 접근 방식의 한계를 보완할 수 있음을 보여주며, Bugram 기법의 실효성을 입증하였다.

2.2 바이트코드 자연성 기반 오류 분석

한편, Choi et al.은 바이트코드 수준에 n-gram 모델을 적용하여 명령어 시퀀스의 통계적 빈도 분포를 분석한 결과, 소스코드나 AST 표현과는 구별되는 특성을 보임을 확인하였다[2]. 특히, 컴파일러에 의해 생성된 바이트코드는 제한된 명령어 집합과 정형화된 구조를 가지므로, 소스코드에 비해 안정적이고 일관된 빈도 분포를 나타낸다. 이는 프로그램 표현 수준에 따라 자연성 기반 분석이 오류 탐지에 활용될 수 있음을 시사한다.

2.3 연구의 교차 지점

Bugram 기법은 프로그램 토큰 시퀀스에 기반한 통계적 오류 탐지라는 새로운 접근 방식을 제시하였으나, 분석 대상이 소스코드 수준에 한정되어 있다는 한계를 가진다. 첫째, 개발자마다 상이한 명명 규칙이나 코딩 스타일로 인해 토큰 분포의 가변성이 커질 경우, 모델의 민감도가 증가할 수 있다. 둘째, Java 기반 프로젝트에서도 Native Method(C/C++)와 같이 소스코드 형태가 상이한 구성 요소가 포함될 경우, 통합적인 분석이 어렵다.

반면, 바이트코드 기반 분석은 컴파일러에 의해 정제된 명령어 시퀀스를 사용함으로써 프로그램의 통계적 특성을 보다 간결하고 일관되게 표현할 수 있다는 장점을 가진다. 또한, 소스코드의 가용성이나 개발 환경에 의존하지 않고도 동일한 품질의 분석 데이터를 확보할 수 있다.

본 연구는 두 연구 흐름의 교차 지점에서 Bugram의 통계적 오류 탐지 알고리즘을 바이트코드 도메인으로 확장한다. Bugram은 코드 생성이나 의미 이해를 목표로 하는 최근의 신경망 기반 접근과 달리, 통계적으로 극히 희귀한 시퀀스를 이상치로 식별하는 데이터 희소성(Sparsity) 기반의 이상 탐지에 초점을 둔다. 이러한 접근은 PR-Miner[3]로 대표되는 비정상 패턴 마이닝 계보에 속하며, 바이트코드와 같이 구조적 일관성이 높은 도메인에서는 여전히 효과적인 전략이다.

3. 제안 방법

본 장에서는 Bugram 기법을 바이트코드 도메인으로 확장 적용한 방법을 설명한다.

3.1 바이트코드 기반 Bugram 설계 개요

Wang et al.[1]의 방법론을 계승하여, 본 연구에서는 trigram 언어 모델을 학습 모델로 사용하고, 5-gram 길이의 토큰 시퀀스에 대해 로그 확률(log-probability)을 계산하였다. 각 시퀀스에 대해 계산된 평균 로그 확률 값이 낮을수록 통계적으로 자연스럽지 않은 패턴으로 간주되며, 본 연구에서는 이러한 시퀀스 중 상위 K개를 최종 버그 후보로 선정한다. 그림 1은 본 연구에서 사용한 시퀀스별 로그 확률 계산 알고리즘의 의사코드를 나타낸다.

Input: Method tokens T, Window size L
Output: Bug candidates sorted by average log-probability

1. $norm\_tokens \leftarrow normalize(T)$
2. **for each** window seq of length L **in** $norm\_tokens$:
3. $sum\_log\_P \leftarrow 0.0$
4. $t_{i-2,}t_{i-1} \leftarrow BOS, BOS$ //initialize triagram context
5. **for each** token t_i **in** seq: $P(t_i \mid t_{i-2}, t_{i-1}) \leftarrow get\_triagram\_probability(t_{i-2}, t_{i-1}, t_i)$
6. $sum\_log\_P \leftarrow sum\_log\_P + log(P(t_i \mid t_{i-2}, t_{i-1}))$
7. $t_{i-2}, t_{i-1} \leftarrow t_{i-1}, t_i$ // Update context for next token
8. $avg\_log\_P \leftarrow sum\_log\_P/L$
9. $candidates.add(avg\_log\_P, seq)$
10. return $candidates$ sorted by avg_log_P in ascending order

그림 1: Bugram 지수 산출을 위한 시퀀스별 로그 확률(Log-probability) 계산 과정[1]

3.1.1 Bugram 적용 지점에서의 바이트코드 자연성 반영

본 연구와 Wang et al.[1]의 핵심적인 차이점은 분석 대상 입력 데이터의 표현 수준에 있다. 기존 연구에서는 JavaParser 등을 활용하여 소스코드 기반의 토큰 시퀀스나 추상 구문 트리(AST)를 분석 대상으로 삼은 반면, 본 연구에서는 컴파일된 결과물인 Java 바이트코드(.class 파일)를 직접 분석한다.

바이트코드 명령어 추출을 위해 자바 바이트코드 조작 프레임워크인 ASM을 사용하였다. ASM의 MethodVisitor를 활용하여 각 메서드 내부의 명령어를 순회하고, 이 과정에서 피연산자(operand)를 제외한 순수 명령어(opcode) 시퀀스만을 토큰화하여 추출하였다. 그림 2는 ASM 라이브러리를 이용해 opcode를 추출하는 구현 예시를 보여준다.

```
@Override
public void visitInsn(int opcode)
{    tokens.add(Printer.OPCODES[opcode]);    }
```

그림2. ASM 라이브러리를 통해 opcode를 추출하는 구현 예시

4. 실험 설계

본 실험에서는 제안하는 바이트코드 기반 Bugram의 성능을 평가하기 위해 표 1과 같은 환경에서 실험을 진행하였다. 실험은 Windows OS 기반의 환경에서 수행되었으며, 대규모 바이트코드 데이터 파싱 및 통계 모델 연산을 처리하기 위해 고성능 프로세서와 충분한 메모리 자원을 활용하였다.

구분	상세 사양
사용 언어	Java (JDK 11 이상), Python 3.12.3
분석 도구	ASM Framework (Bytecode Parsing)

표 1. 실험 환경

4.1 Bugram 수치 계산

본 장에서는 해당 알고리즘에 사용된 주요 파라미터를 기술한다. 또한, Wang et al.[1]의 실험과 바이트코드로 확장된 제안 방법의 실험 결과를 비교 분석한다.

4.1.1 Bugram 기법 알고리즘 파라미터

본 연구에서 Bugram의 성능 평가를 위해 설정한 세부 파라미터는 표2와 같다. 이 수치들을 버그 후보군을 추출할때 그 범위와 정교함을 결정하는 요소로 작용한다.

파라미터명	설정값	비고
Language Model	Trigram (n=3)	모델 학습 단위
Sequence Length	5	자연스러움 측정 단위
Min Token Freq	3	희귀 토큰 처리 기준
Top-K Candidates	1000	최종 분석 대상 수

표 2. Bugram 기법 알고리즘 파라미터

4.2 실험 대상 및 오류 사례

본 연구는 바이트코드 기반 Bugram 기법의 오류 탐지 특성을 분석하기 위해 Nutch와 ProGuard 두 오픈소스 프로젝트 전체 코드를 실험대상으로 사용하였다. 실험에서는 Wang et al.[1]에서 사용된 것과 동일한 Nutch 2.3.1과 ProGuard 5.2 버전의 소스코드를 사용하였다.

본 실험의 목적은 전체 코드 중에서 이미 알려진 실제 오류 사례들이 제안한 기법에 의해 이상치 후보로 효과적으로 탐지되는지를 확인하는 데 있다. 검증을 위해, Wang et al.[1]에서 실제 오류로 검증된 세 가지 오류 사례를 사용하였으며, 각 다음과 같은 특성을 가진다.

- NUTCH-2256: 로그 레벨 사용 오류로, 의미적 불일치(semantic bug)에 해당
- NUTCH-2076: 예외 처리 구조의 의미론적 오류(control-flow semantics bug)
- PROGUARD-582: 자원 관리 및 예외 처리와 관련된 명령어 수준 오류(instruction-level)

4.3 제안 방법 적용 전 실험

본 장에서는 제안하는 바이트코드 기반 Bugram의 효과를 비교 분석하기 위해 Wang et al.[1]의 기법을 비교 기준선(Baseline)으로 설정하여 실험을 수행하였다 .

- 토큰 추출 및 정규화 : IfStmt, TryStmt 등 주요 제어 구조와 메서드 호출을 의미론적 토큰(예 : <IF>)으로 변화하였다. 기본적으로 제공하지 않는 특수 패턴이나 예외처리 구문은 수동으로 구분하였다.
- 통계 모델 구축 : 추출된 토큰 시퀀스를 바탕으로 n-gram 확률을 계산하기 위해 유니그램(Unigram), 바이그램(Bigram), 트라이그램(Trigram)의 빈도를 각각 산출하였다. 희소성 방지를 위해 3회 미만 토큰을 <RARE>로 치환하였다.
- 로그 확률 산출 : 학습된 통계 모델을 바탕으로 라플라스 스무딩(Laplace Smoothing)이 적용된 트라이그램 조건부 확률을 계산하였으며, 이를 로그 스케일로 합산하여 시퀀스별 평균 로그 확률(Average log-probability)을 산출하였다. 해당 수치가 낮을수록 통계적으로 '이상(Abnormal)'인 코드로 간주하며, 이를 최종 버그 후보 추출의 기준으로 사용하였다.

4.4 제안 방법 적용 후 실험

본 연구의 제안 방법을 적용하여 오류를 탐지한 결과는 표3과 같다. 바이트코드 기반 Bugram 기법의 유효성을 평가하기 위해, 소스코드 대신 컴파일된 클래스 파일(.class)을 분석 대상으로 하여 실험을 수행하였다.

- 토큰 추출 및 정규화 : Wang et al.[1]의 방법과 다르게 ASM이 제공하는 MethodVistor와 Printer를 이용하여 컴파일된 바이너리에서 직접 opcode 시퀀스를 추출하여 분석의 기초 데이터로 사용하였다.

5. 결과

5.1 n-gram 시퀀스 길이에 따른 탐지 결과

표 3은 n-gram 시퀀스 길이를 2부터 9까지 변화시키며 각 오류 사례가 Top-K 후보 내에서 검출된 순위를 나타낸다. 표에서 값이 작을수록 해당 오류가 상위 후보로 검출되었음을 의미하며, 각 값은 이상치로 분류된 Top 1000개의 후보군 중에 상위 몇 번째에 랭크되었는지 표현한다. 'X'는 Top-K 내에서 검출되지 않았음을 의미한다.

오류	2	3	4	5	6	7	8	9
NUTCH-2256	884	X	X	884	X	X	X	X
NUTCH-2076	X	X	X	X	X	X	X	X
PROGUARD-582	X	21	27	24	53	44	277	234

표 3. n-gram 시퀀스 길이에 따른 오류별 검출 순위 변화

실험 결과, PROGUARD-582는 다양한 시퀀스 길이 설정에서 일관되게 검출된 반면, NUTCH-2076은 모든 설정에서 검출되지 않았으며, NUTCH-2256은 일부 설정에서 매우 낮은 순위로만 검출되었다. 이는 바이트코드 기반 Bugram이 명령어 수준의 실행 흐름 이상에는 민감하게 반응하는 반면, 의미적 오류나 제어 흐름의 의미론적 오류에는 반응하지 않음을 보여준다.

5.2 MIN_TOKEN_FREQ 민감도 분석

표 4는 MIN_TOKEN_FREQ 값을 변화시키며 각 오류 사례의 검출 순위를 비교한 결과를 나타낸다. 표에서 값이 작을수록 해당 오류가 상위 후보로 검출되었음을 의미하며,

'X'는 Top-K 내에서 검출되지 않았음을 의미한다.

MIN_TOKEN_FREQ	NUTCH-2256	NUTCH-2076	PROGUARD-582
1	X	X	99
3	884	X	24
5	X	X	21

표 4. MIN_TOKEN_FREQ 값 변화에 따른 오류 검출 민감도 분석 결과

PROGUARD-582의 경우 MIN_TOKEN_FREQ 값이 증가함에 따라 검출 순위가 점진적으로 향상되는 경향을 보였다. 이는 해당 오류가 통계적으로 드문 opcode 조합과 밀접하게 연관되어 있음을 시사한다. 반면, NUTCH 계열 오류는 희귀 토큰 처리 기준의 변화와 관계없이 검출되지 않거나 일부 설정에서 매우 낮은 순위로만 검출되었다.

5.3 Wang et al.[1]의 기법과 비교

표 5는 Wang et al.[1]와 본 연구에서 제안한 바이트코드 기반 Bugram 기법의 오류 검출 결과를 비교한 것이다.

오류이름	기존기법	제안기법
NUTCH-2256	45	884
NUTCH-2076	8	X
PROGUARD-582	11	24

표 5. Wang et al.[1]의 기법과 바이트코드 기반 Bugram의 오류 검출 순위 비교

비교 결과, Wang et al.[1]은 의미적 오류인 NUTCH-2256에 대해 상대적으로 높은 순위로 오류를 검출한 반면, 바이트코드 기반 Bugram은 해당 오류를 탐지하지 못하였다. 반대로, PROGUARD-582와 같은 명령어 수준 오류에 대해서는 바이트코드 기반 Bugram이 안정적인 검출 성능을 보였다. 이는 두 접근 방식이 서로 다른 오류 유형에 대해 상보적인 탐지 특성을 가짐을 의미한다.

6. 논의

연구 결과, 바이트코드는 소스코드에 비해 식별자(identifier) 및 표현 방식의 가변성이 낮아, 보다 일관된 분석 환경을 제공함을 확인하였다. 이러한 특성은 n-gram 모델이 코드의 자연스러움을 학습하는 데 있어 노이즈를 줄이고 안정적인 통계 분포를 형성하는 데 기여한다.

그러나 이러한 장점에도 불구하고, 바이트코드 기반 Bugram의 탐지 성능은 오류 유형에 따라 차이를 보였다. 명령어 수준의 실행 흐름이나 자원 관리와 관련된 오류에 대해서는 소스코드 기반 Bugram과 유사한 탐지 경향을 유지하였으나, 의미적 불일치나 제어 구조의 의도된 의미가 핵심인 오류에 대해서는 탐지 성능이 제한적이었다. 이는 바이트코드 표현이 프로그램의 고수준 의미 정보를 충분히 반영하지 못하기 때문으로 해석된다.

본 연구는 바이트코드 기반 Bugram의 동작 가능성을 검증하기 위한 탐색적 단계로서, 대표적인 3개의 주요 오류 사례를 중심으로 분석을 수행하였다. 상위 랭킹 내 다수의 이상치 후보에 대한 전수 조사 및 사실 검증은 향후 연구에서 다룰 예정이다.

종합하면, 바이트코드 기반 Bugram은 Wang et al.[1]에서 제안된 기법을 전면적으로 대체하기보다는, 명령어 수준 오류 탐지에 특화된 보완적 기법으로 활용될 수 있으며, 향후 operand 정보나 제어 흐름 구조를 결합한 확장 모델을 통해 탐지 범위를 확장할 수 있을 것으로 기대된다.

7. 결론 및 향후 연구

본 연구에서는 Wang et al.[1]의 기법을 바이트코드 수준으로 확장하여, opcode 기반 바이트코드 자연성(bytecode naturalness)이 오류 탐지에 효과적으로 활용될 수 있는지를 실험적으로 분석하였다. 실험 결과, 바이트코드 기반 Bugram은 명령어 시퀀스 자체에 이상(anomaly)이 존재하는 오류 유형에 대해서는 효과적인 탐지 성능을 보였으나, 의미적 불일치나 제어 흐름의 의미론적 오류들은 통계적으로 자연스러운 패턴을 유지하여 탐지되지 않았다. 이를 통해 본 연구는 바이트코드 기반 Bugram이 모든 오류 유형을 포괄적으로 대체하기보다는, 적용 가능한 오류 범위를 명확히 규명하는 데 의의가 있다.

한편, 본 연구는 제한적인 데이터셋과 정량적 평가 지표(Precision, Recall)의 미비함이라는 한계를 가진다. 이는 바이트코드 도메인 적용 가능성을 확인하는 초기 단계 연구의 특성에 기인한다. 따라서 향후 연구에서는 Jira 이슈 트래커를 활용하여 추가 오류 사례를 확보하고(Dataset augmentation), False Positive 분석을 포함한 정량적 성능 지표를 산출할 예정이다. 또한, 기존 연구[2, 4]를 바탕으로 복합 시퀀스 모델 및 심층 학습 기반 모델을 도입하여 탐지 성능을 개선하고자 한다.

References

[1] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 708-719.

[2] Y. Choi and J. Nam, "On the Naturalness of Bytecode Instructions," in Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2022, pp. 1-5.

[3] C. Li and Z. Zhou, "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code," in Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2005, pp. 306-315.

[4] D. Kim, T. Kim, J. Shin, S. Wang, H. Choi, and J. Nam, "Pre-trained Models for Bytecode Instructions," in Proceedings of the 2025 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2025, pp. 1-5.

바이너리 코드 (역)어셈블러 자동 생성 방법에 대한 탐구*

김지훈°, 정승일, 김준태, 차상길
한국과학기술원
{etyu3939, sijung, gnrkr789, sangkilc}@kaist.ac.kr

Towards Automatic Generation of Binary-Code (Dis)Assemblers

Jihun Kim, Seungil Jung, Juntae Kim, Sang Kil Cha
KAIST

요 약

본 논문은 명령어셋 매뉴얼을 자동으로 파싱하여 역어셈블러 코드를 생성하는 방법을 제안한다. 기존 역어셈블러는 매뉴얼을 개발자가 수작업으로 구현하는 방식에 의존하여 오류 발생 가능성과 확장성 한계를 지닌다. 제안 기법은 PowerPC(PPC) 아키텍처를 대상으로 매뉴얼의 명령어 정의를 구조화된 JSON 형식으로 추출하고, 이를 기반으로 역어셈블러 파서를 자동 생성한다. coreutils 바이너리를 대상으로 한 실험 결과, 자동 생성된 역어셈블러가 실제 바이너리에 대해 높은 정확도로 동작함을 확인하였다.

1. 서론

바이너리 분석에서 역어셈블 결과의 정확도는 분석 신뢰도에 직접적인 영향을 미친다. 그러나 기존 (역)어셈블러[1]는 명령어셋 매뉴얼을 사람이 수작업으로 해석하여 구현하는 방식에 의존하고 있어, 복잡한 명령어 규칙이나 예외 입력 처리 과정에서 결함이 발생하기 쉽다. 실제로 다양한 역어셈블러 도구에서 유효하지 않은 명령어나 예외 입력 처리 중 크래시나 오동작이 발생한 사례가 반복적으로 보고되어 왔다[1, 2, 3, 4].

기존 연구는 주로 어셈블러 버그 탐지[5]에 초점을 두었으며, 역어셈블러를 자동으로 생성하는 접근은 머신 코드에 대한 정형화된 명세 부재로 인해 충분히 다루어지지 않았다.

본 논문은 RISC 구조인 PowerPC(PPC) 아키텍처의 명령어 비트 필드 특성을 활용하여, 매뉴얼을 자동으로 파싱하고 역어셈블러를 생성하는 방법을 제안한다. 명령어 패턴을 자동 추출 및 정규화하는 도구를 구현하고, Coreutils[6] 바이너리 10종을 대상으로 실험을 수행하여 생성된 파서의 정확도와 유효성을 검증하였다.

2. 제안 방법

역어셈블러 자동 생성은 매뉴얼에서 추출한 명령어 구조를 구조화한 뒤 이를 기반으로 역어셈블 코드를 생성하는 두 단계로 이루어진다.

PPC32 및 PPC64를 포함한 다수의 아키텍처에서 명령어셋 매뉴얼은 PDF 형식으로 제공되며, 본 논문에서는 Power ISA 3.0C 버전 PDF 매뉴얼을 사용한다. PDF는 직접 처리하기 어려우므로, AI 기반 OCR 대신 텍스트 기반 PDF 파싱 도구인 pdfplumber [7]를 활용하여 매뉴얼을 텍스트로 변환하였다. 이후 텍스트에서 명령어 위치를 특정하기 위해 필드 오프셋 나열을 탐색하는데, PPC 아키텍처는 명령어 길이가 항상 32비트이고 첫 필드가 6비트로 고정되어 있어, 0, 6으로 시작하여 31로 끝나는 오프셋 패턴을 통해 명령어를 효과적으로 식별할 수 있다.

```
{
    "opcode": "addi",
    "operands": ["RT", "RA", "SI"],
    "fields": [
        {"name": "RT", "field_range": [6, 10]},
        {"name": "RA", "field_range": [11, 15]},
        {"name": "SI", "field_range": [16, 31]}
    ],
    "equal_conditions": [
        {"value": 14, "field_range": [0, 5]}
    ]
}
```

그림 1: ADDI 명령어의 JSON 구조

명령어를 구조화된 형태로 저장할 때에는 역어셈블에 필요한 모든 정보를 포함해야 하며, 이를 위해 본 논문에서는 그림 1과 같은 JSON 구조를 사용한다. 각 명령어는 opcode, operands, fields, equal conditions의 네 가지 요소로 표현되며, 각각 명령어 이름, 피연산자 목록, 필드 오프셋 정보, 그리고 바이트 코드 매칭 조건을 저장한다. 이 구조는 PPC의 모든 명령어 패턴을 표현할 수 있으며, 텍스트 파일로부터의 변환은 문자열 파싱을 통해 구현된다.

1) 어셈블러와 역어셈블러는 개념적으로 상호 대응 관계에 있으나, 본 연구의 논의 범위에서는 용어를 통일하여도 무리가 없으므로 역어셈블러로 일괄하여 사용한다.

*이 논문은 2025년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원(No. RS-2025-02263143, 인공위성 지상국 사이버보안 위협 대응 기술 개발)과 2025년도 정부(과학기술정보통신부)의 재원으로 한국인터넷진흥원(KISA) 정보보호전문인력양성(정보보호특성화대학) 사업의 지원을 받아 수행된 연구임

2.1 역어셈블러 코드 자동 생성

각 명령어의 JSON 구조가 주어지면 역어셈블러의 자동 생성은 비교적 직관적이다. 입력된 바이트코드는 `equal conditions`를 통해 명령어를 판별하고, `operands`와 `fields` 정보를 이용해 피연산자 값을 추출한다. 일부 명령어에서는 하나의 피연산자가 여러 필드를 조합해 계산되지만, 이러한 경우는 피연산자 이름에 따라 값 추출 방식이 정해져 있다. PPC 아키텍처에서 이러한 예외적인 피연산자는 소수에 불과하므로, 개별적으로 처리하는 데 큰 어려움은 없다.

3. 실험

Power ISA 3.0C 매뉴얼을 기반으로 역어셈블러 자동 생성을 수행한 결과, 전체 1,169개 명령어 중 1,139개에 대해 역어셈블러 코드를 성공적으로 생성하였다. 나머지 30개 명령어는 매뉴얼 내 오타, 비정형적인 서술 방식, 또는 PDF 파싱 과정에서의 공백 처리 문제로 인해 자동 생성에 실패하였다. 이는 자동 생성 기법 자체의 한계라기보다는 매뉴얼 표기상의 문제에 기인한 것으로, 해당 명령어를 제외하면 자동 생성은 안정적으로 동작하였다.

3.1 파서 정확도 평가

자동 생성된 역어셈블러의 정확성을 검증하기 위해 PPC64 아키텍처를 대상으로 역어셈블러를 구현하고, coreutils에 포함된 10개 바이너리에 대해 실험을 수행하였다. 자동 생성에 실패한 일부 명령어는 수동으로 보완하였으며, 역어셈블 결과를 GNU Assembler를 사용하여 다시 어셈블하여 원래 바이너리와 비교하는 방식으로 정확성을 검증하였다(그림 2). 실험 결과 대부분의 바이너리에 대해 완전한 역어셈블이 가능했으며, 일부 예외는 Power ISA 매뉴얼에 정의되지 않은 구현 종속 명령어에 해당하였다. 이는 매뉴얼 기반 자동 생성 방식으로 RISC 아키텍처의 역어셈블러를 정확하게 구현할 수 있음을 보여준다.

3.2 파서 성능 평가

실행 성능 평가 결과, 자동 생성된 파서는 수작업으로 구현된 파서와 비교하여 전반적으로 유사한 수준의 성능을 유지하였다. 단일 매치 구조를 사용하는 특성상 일부 경우에서 다소 느린 경향이 관찰되었으나, 자동 생성 방식임에도 불구하고 성능 차이는 제한적인 수준에 머물렀다. 이는 파서 자동 생성이 구현 자동화에 따른 이점을 제공하면서도 실행 성능 측면에서는 실용적인 범위 내에 있음을 보여준다. 개발 효율성과 확장성을 함께 고려할 때, 제안 기법은 충분한 활용 가능성을 갖는다.

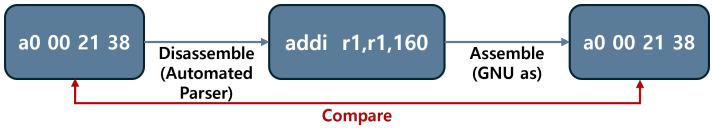


그림 2: 파서 정확도 검증 방법

3.3 역어셈블러 구현 방식에 따른 특성 분석

구현 방식에 따른 개발 효율을 비교한 결과, 수동 구현은 약 3개월이 소요된 반면, 자동 생성 방식은 매뉴얼 분석과 패턴 정의를 포함하여 약 1주일 내에 구현이 가능하였다. 동일한 수준의 개발자가 수행하였다는 점에서, 본 결과는 자동 생성 접근이 역어셈블러 개발의 효율성과 확장성을 크게 향상시킬 수 있음을 정성적으로 보여준다.

4. 결론

본 연구는 PPC 아키텍처를 대상으로 명령어셋 매뉴얼을 자동으로 파싱하여 JSON 형식으로 정규화하고, 이를 기반으로 역어셈블러 파서를 자동 생성하는 방법을 제안하였다. 제안된 접근은 매뉴얼 기반 규칙을 체계적으로 정형화함으로써 수작업 구현에 대한 의존도를 줄이고, coreutils 바이너리를 대상으로 한 실험을 통해 생성된 파서의 정확성을 검증하였다.

향후에는 본 기법을 다른 RISC 아키텍처로 확장하고, 매뉴얼 기술 방식과 구조적 특성을 고려하여 CISC 명령어 체계에 대한 적용 가능성도 탐색할 계획이다. 본 연구의 접근은 대규모 바이너리 분석 환경에서 역어셈블 과정의 신뢰성과 유지보수성을 향상시키는 데 기여할 수 있다.

참고 문헌

[1] “llvm-objdump can crash when it can’t disassemble an instruction.” LLVM Phabricator D73531, 2020.

[2] “disassemble passing invalid instructions.” GitHub Issue #1776, capstone-engine/capstone, 2021.

[3] “Capstone failing to disassemble instruction.” GitHub Issue #2612, capstone-engine/capstone, 2025.

[4] “Error handling in disasm() method.” GitHub Issue #564, capstone-engine/capstone, 2015.

[5] H. Kim, S. Kim, J. Lee, and S. K. Cha, “Asfuzzer: Differential testing of assemblers with error-driven grammar inference,” in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 1099–1111, 2024.

[6] Debian Project, “Gnu coreutils package.” https://packages.debian.org/trixie/coreutils, 2026.

[7] J. Singer-Vine and The pdfplumber contributors, “pdfplumber,” Nov. 2025.

상태 공간 모델 기반 오프라인 강화학습의 강건성 테스팅

한태현, 김장환, 김영철
홍익대학교 소프트웨어공학연구실
taehyun3172@g.hongik.ac.kr, lentoconstante@hongik.ac.kr, bob@hongik.ac.kr

Robustness Testing of Offline Reinforcement Learning based on State Space Models

Taehyun Han, Janghwan Kim, R. Young Chul Kim
Software Engineering Laboratory, Hongik University

요 약

최근 오프라인 강화학습이 실제 환경에 도입됨에 따라, 예측 불가능한 외부 노이즈에 대한 안전성 및 강건성 검증이 필수적이다. 그러나 기존 무작위 노이즈 방식은 취약 시점을 식별하지 못해 비효율적이며, 적대적 공격 기법은 높은 연산 비용으로 인해 실시간 검증에 적용하기 어렵다. 이러한 문제를 해결하기 위해, 상태 공간 모델 기반 Mamba 아키텍처를 활용한 효율적인 블랙박스 테스팅 기법을 제안한다. 본 연구는 노이즈의 주입 시점이 시스템의 생존 및 강건성에 결정적임을 실험적으로 입증하였다. 이를 통해 제안하는 Mamba 기반 노이즈 주입 기법이 효율적인 블랙박스 테스팅 기법으로서 활용될 것을 기대한다.

1. 서론

최근 로보틱스와 같은 복잡한 제어 문제에서 사전에 수집된 데이터를 활용하는 오프라인 강화학습이 핵심 기술로 부상하고 있다. 특히 Decision Transformer (DT)는 강화학습을 시퀀스 모델링으로 재해석하여 탁월한 성과를 거두었다 [1]. 그러나 이러한 모델이 안전 필수 환경에 배포되기 위해서는 예측 불가능한 외부 노이즈에 대한 강건성 검증이 선행되어야 한다 [2].

현재 강건성 평가 방법론은 크게 무작위 노이즈 주입과 적대적 공격 기법으로 나뉜다. 무작위 방식은 구현이 간단하나, 에이전트의 실패를 유발하는 결정적 원인을 파악하기 어렵고 효율이 낮다 [3]. 반면, 적대적 공격 기법들은 탐지 성능은 우수하나, 높은 연산 비용과 학습 복잡도로 인해 실시간 진단 도구로 활용하기에는 제약이 존재한다 [4].

본 논문에서는 이러한 한계를 극복하기 위해 상태 공간 모델(SSM)인 Mamba 아키텍처를 활용한 효율적인 강건성 테스팅 프레임워크를 제안한다 [5,6]. 우리는 Mamba 가 환경의 동역학을 학습하는 과정에서 내부 파라미터인 델타 (Δ) 값이 급증하는 구간이 곧 정보량이 높고 노이즈에 취약한 불안정한 상태라는 가설을 제시한다. 제안 기법은 Mamba 의 델타 파라미터 지표를 모니터링하여 불안정한 시점의 상태를 식별하고, 선택적으로 노이즈를 주입하여 모델의 취약점을 효율적으로 진단한다.

2. 상태 공간 모델 기반 강건성 테스팅 메커니즘

SSM 은 연속적 시간의 시스템을 이산화하며, 이때 이산화 파라미터 델타 (Δ) 는 데이터를 얼마나 반영할지 결정하는 역할을 한다 [5]. 기존 SSM 과 달리 Mamba 는 입력 x에 따라 Δ가 동적으로 변하는 선택적 메커니즘을 도입하였다 [6].

$$\Delta_t = Softplus(W_\Delta x_t + b_\Delta) \quad (1)$$

수식 1 과 같이 Δ_t 는 고정된 값이 아닌 매 시점 입력 데이터와 학습된 파라미터의 조합에 의해 결정되는 동적 행렬이다. 본 연구는 이러한 Δ가 급격히 변하는 시점이 곧 모델이 예측하기 어려운 불안정 상태라는 가설에 기반한다.

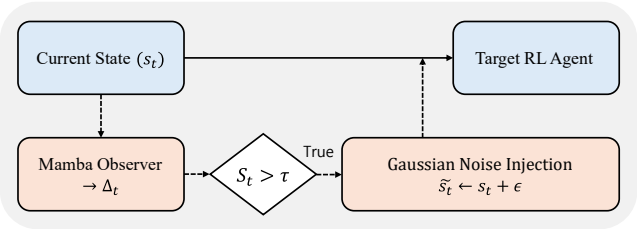


그림 1. 제안하는 Mamba 기반 강건성 테스트 프레임워크

그림 1 은 제안하는 전체 프레임워크의 구조를 보여준다. 제안 기법은 에이전트의 정책과 무관하게 환경의 특성을 분석하기 위해 Mamba 아키텍처를 관측 모델로 활용한다. 오프라인 데이터셋을 사용하여 관측 모델은 현재 시점까지의 상태 $s_{1:t}$를 입력 받아 다음 상태 s_{t+1}을 예측하도록 학습한다 [8,9]. 학습된 관측 모델은 테스팅 단계에서 실시간으로 입력되는 상태 s_t에 대해 내부 파라미터 Δ_t를 산출한다. 이를 지표로서 활용하기 위하여 다음 수식 2 와 같이 L2-Norm 을 적용하여 단일 스칼라 값인 중요도 점수 (S_t)로 변환한다.

$$S_t = ||\Delta_t||_2 = \sqrt{\sum_{i=1}^{D} (\Delta_{t,i})^2} \quad (2)$$

여기서 S_t 값이 클수록 현재 상태가 미래 예측에 중요한 정보를 담고 있음을 의미한다. 제안 기법은 S_t 가 사전에 설정된 임계치 (τ) 를 초과하는 경우 시스템의 취약 시점으로 판단하여 가우시안 노이즈를 주입하고 강건성을 검증한다.

3. 상태 공간 모델 기반 강건성 테스팅 메커니즘 적용 사례

제안 기법의 유효성을 검증하기 위해 D4RL 벤치마크의 MuJoCo 환경(Hopper, Walker2d, HalfCheetah)을 사용하였으며 검증 대상 모델로는 Medium 데이터셋으로 학습된 DT 모델을 사용하였다 [1,7]. 이때 Mamba 관측 모델의 하이퍼 파라미터는 임베딩 차원 $D = 64$, 상태 차원 $N = 16$ 으로 설정하여 대상 모델과 동일한 데이터셋으로 학습시켰다. 노이즈 주입 임계치 (τ)는 사전 에피소드 수행을 통해 수집된 Δ값의 분포의 상위 10%로 결정하였다.

3.1 실험 결과 및 분석

표 1. 환경 별 성능 하락률 결과

Noise		Performance Drop		
Scale	Method	HalfCheetah	Hopper	Walker2d
0.1	Random	6.0%	6.5%	8.3%
	Mamba	9.5%	24.3%	8.3%
0.2	Random	8.1%	14.3%	11.9%
	Mamba	17.8%	39.8%	5.5%
0.3	Random	10.8%	16.6%	14.0%
	Mamba	34.6%	53.5%	19.9%
0.4	Random	10.0%	23.6%	29.9%
	Mamba	26.3%	60.6%	40.2%
0.5	Random	10.0%	22.8%	22.9%
	Mamba	37.8%	61.7%	46.8%

표 1 은 각 환경에서의 제안 기법 적용 결과이다. 실험 결과, 로봇의 물리적 구조에 따라 제안 기법의 효과가 뚜렷한 차이를 보였다. 구조적으로 가장 불안정한 Hopper 에서는 제안 기법의 효용성이 극대화되었다. 최소 노이즈 스케일만으로도 무작위 방식이 최대 스케일을 가했을 때보다 더 큰 성능 하락을 유발하며 취약 시점의 중요성을 입증하였다. 이족 보행 로봇인 Walker2d 에서는 노이즈 강도가 높아질수록 두 방식 간의 격차가 벌어졌다. 최대 스케일에서 제안 기법은 무작위 방식 대비 2 배 이상의 성능 저하를 기록하였다. 마지막으로 가장 안정적인 HalfCheetah 에서도 무작위 방식은 성능 저하가 미미했던 반면, 제안 기법은 약 4 배에 달하는 효과를 보였다. 종합적으로 제안 기법은 모든 환경에서 무작위 방식 대비 월등한 성능 하락을 유도하여, 효율적인 강건성 테스팅 도구임을 증명하였다.

3.2 선택적 메커니즘과 불안정 상태의 상관관계

본 연구는 Mamba 의 내부 파라미터인 델타(Δ)가 급증하는 구간이 곧 에이전트의 불안정 상태라는 가설을 검증하였다. 선택적 메커니즘에서 Δ 는 정보 반영률을 결정하므로, 높은 중요도 점수(S_t)는 해당 상태가 미래 예측에 핵심적인 정보를 담고 있음을 의미한다. 실험 결과, Δ 값이 높은 시점에 노이즈를 주입했을 때 무작위 방식 대비 치명적인 성능 하락이 발생하여 이 가설의 유효성이 입증되었다. 이는 복잡한 연산이나 모델 내부의 기울기 정보 없이도, 관측 모델의 내부 지표만으로 시스템의 취약 시점을 효율적으로 식별할 수 있음을 시사한다.

4. 결론

본 논문에서는 오프라인 강화학습 모델의 강건성을 효율적으로 평가하기 위해, SSM 기반 Mamba 아키텍처를 활용한 테스팅 프레임워크를 제안하였다. 우리는 Mamba 의 내부 지표인 델타(Δ)를 활용하여 에이전트가 외부 노이즈에 민감하게 반응하는 불안정한 상태를 식별하고, 해당 시점에 선별적으로 노이즈를 주입하였다. 실험 결과, 제안 기법은 무작위 방식과 동일한 빈도의 노이즈 주입만으로도 최대 4 배 이상의 보상 하락을 유도하였다. 이는 제안 기법이 에이전트의 제어 실패를 유발하는 결정적 순간을 효과적으로 식별했음을 시사하며, 효율적인 블랙박스 강건성 테스팅 도구로서의 활용 가능성을 보여준다. 본 연구는 무작위 방식과의 비교를 통해 제안 기법의 기초적인 타당성을 확인하였으며, 향후 연구에서는 단순 휴리스틱 및 기존 적대적 공격 기법과의 연산 비용 및 탐지 성능에 대한 정량적 비교를 수행하여 Mamba 내부 지표의 고유한 효용성을 더욱 정밀하게 검증하고자 한다.

감사의 글

본 연구는 한국연구재단의 4 단계 두뇌한국 21 사업(과제명: 초분산 자율 컴퓨팅 서비스 기술 연구팀, 과제번호: 202003520005)의 지원을 받아 수행된 연구임.

5. 참고문헌

[1] L. Chen et al., Decision transformer: Reinforcement learning via sequence modeling, *Advances in Neural Information Processing Systems*, Vol. 34, pp. 15084-15097, 2021.

[2] S. Huang et al., Adversarial attacks on neural network policies, *arXiv preprint arXiv:1702.02284*, 2017.

[3] Y. Wang et al., A systematic review of fuzzing based on machine learning techniques, *PLoS One*, Vol. 15, No. 8, e0237749, 2020.

[4] J. Sun et al., Stealthy and efficient adversarial attacks against deep reinforcement learning, *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34, No. 04, 2020.

[5] A. Gu et al., Efficiently modeling long sequences with structured state spaces, International Conference on Learning Representations (ICLR), 2022.

[6] A. Gu and T. Dao, Mamba: Linear-time sequence modeling with selective state spaces, *First Conference on Language Modeling*, 2024.

[7] J. Fu et al., D4rl: Datasets for deep data-driven reinforcement learning, *arXiv preprint arXiv:2004.07219*, 2020.

EnvAgent: AI/ML 프로젝트의 Conda 환경 자동 구축 시스템

권혁민°, 이지광, 강신엽, 정용빈, 남재창
한동대학교 전산전자공학부
hyeokkiyaa@gmail.com, lucas0606@handong.ac.kr, yeob@handong.ac.kr, yongbeanchung@gmail.com, jcnam@handong.edu

EnvAgent: A System for Automated Conda Environment Generation in AI/ML Projects

Hyeokmin Kwon, Jikwang Lee, Shinyeob Kang, Yongbean Chung, Jaechang Nam
The School of Artificial Intelligence, Computer and Electrical Engineering, Handong Global University

요 약

AI/ML 프로젝트의 복잡한 의존성은 연구 재현성을 저해하는 주요 원인이다. Docker는 배포 단계의 표준이나, 다양한 도메인 지식과 반복적 이미지 빌드로 인해 빠른 실험이 필요한 연구 단계에는 적합하지 않아 Conda가 선호된다. 그러나 기존 도구들은 Docker만을 지원한다. 이에 본 연구에서는 GitHub 저장소로부터 실행 가능한 Conda 환경을 자동 생성하는 하이브리드 에이전트 시스템 EnvAgent를 제안한다. EnvAgent는 AST 기반 정적 분석으로 토큰 비용을 절감하고, 자가 치유 메커니즘으로 환경을 자동 복구한다. 9개 AI/ML 프로젝트 대상 실험에서 77.8%의 성공률을 달성하였다.

ABSTRACT

Complex dependencies in AI/ML projects are a major barrier to research reproducibility. While Docker is the standard for deployment, its steep learning curve and repetitive image builds make it unsuitable for rapid experimentation, leading researchers to prefer Conda. However, existing tools only support Docker. We propose EnvAgent, a hybrid agent system that automatically generates executable Conda environments from GitHub repositories. EnvAgent reduces token costs through AST-based static analysis and automatically recovers from failures via a self-healing mechanism. Experiments on 9 AI/ML projects achieved a 77.8% success rate.

1. 서 론

파이썬은 AI/ML 분야의 사실상 표준으로 자리 잡았으나, 동시에 심각한 재현성 위기에 직면해 있다[1]. 전년 대비 48.78%의 기여자 수 증가가 보여주듯 파이썬 생태계는 급성장하고 있지만[2], TensorFlow, PyTorch 등 딥러닝 프레임워크와 CUDA, cuDNN 같은 시스템 레벨 GPU 의존성의 결합은 환경 구성의 복잡도를 현저히 높였다[3]. 이러한 복잡성은 연구 결과의 재현을 어렵게 만들어, Jupyter Notebook 기반 프로젝트의 재현 성공률은 단 8.5%에 불과하다[4]. 이에 학계에서는 ML 분야의 재현성 문제를 '위기' 수준으로 경고하고 있다[5].

모델 배포 단계에서는 환경의 불변성을 보장하기 위해 Docker가 표준적으로 사용된다[6]. 이에 따라 DockerizeMe[7], PLLM[8], Repo2Run[9] 등 Docker 환경 자동 생성 도구가 활발히 연구되어 왔다.

그러나 Docker는 네트워킹, 운영체제, 클라우드 컴퓨팅 등 다양한 도메인 지식을 요구하여 개발 복잡성을 증가시킨다[10]. 특히 빈번한 실험과 코드 수정이 요구되는 연구 단계에서, Docker의 필수적인 이미지 재빌드 과정은 심각한 개발 병목 현상을 초래한다[11]. 이러한 환경에서는 즉각적인 패키지 설치와 환경 전환이 용이한 Conda가 선호된다[12]. 그럼에도 기존 자동화 도구들은 여전히 Docker 지원에 치중되어 있어, Conda 중심의 연구 환경에 대한 지원은 부족하다.

이에 본 연구에서는 AI/ML 프로젝트로부터 실행 가능한 Conda 환경을 자동으로 생성하고 복구하는 하이브리드 에이전트 시스템인 EnvAgent를 제안한다. EnvAgent는 추상 구문 트리(Abstract Syntax Tree, AST) 기반 정적 분석으로 의존성 정보를 경량화하여 LLM 토큰 비용을 절감하고, 반복적 자가 치유 메커니즘을 통해 환경 생성 성공률을 높인다.

본 연구의 주요 기여는 다음과 같다.

- GitHub 저장소 수준에서 실행 가능한 Conda 환경을 자동 생성하는 하이브리드 에이전트 시스템 제안
- AST 기반 정적 분석과 디렉토리 스코어링을 결합하여 Monorepo 구조에서도 강건하게 의존성을 추출하는 경량화 기법 설계
- OS별 패키지 호환성을 자동 처리하는 플랫폼 인지형 빌드 전략 제안
- 설치 실패 시 오류 유형별 복구 전략을 적용하는 반복적 자가 치유 메커니즘 구현
- EnvAgent를 오픈소스로 공개하여 재현 가능한 연구 환경 제공: https://github.com/ISEL-HGU/EnvAgent

2. 관련 연구

재현성 문제를 해결하기 위해 다양한 의존성 자동 추출 도구가 개발되어 왔다. pipreqs[13]와 pigar[14]는 import문 파싱 기반으로 requirements.txt를 생성하며, DockerizeMe[7]와 PyEGo[15]는 지식 베이스 및 지식 그래프를 활용하여 의존성을 추론한다. 그러나 이들은 정적 지식 베이스에 의존하여 빠르게 변화하는 패키지 생태계를 반영하기 어렵고, Conda를 지원하지 못한다는 공통적인 한계가 있다.

최근에는 LLM 기반 접근법이 등장하였다. PLLM[8]은 LLM과 RAG를 결합한 반복적 피드백 루프를, Repo2Run[9]은 LLM 에이전트 기반 Dockerfile 자동 생성을, Installamatic[16]은 README 분석 기반 환경 구성을 제안하였다. 그러나 PLLM은 단일 파일 수준에 그치며, Repo2Run과 Installamatic은 Docker만을 지원한다는 한계가 있다.

본 연구는 AST 기반 정적 분석으로 토큰 비용을 절감하면서, 피드백 기반 자가 치유 메커니즘을 갖춘 에이전트 파이프라인으로 저장소 전체를 분석하여 Conda 환경을 생성한다는 점에서 차별화된다.

3. EnvAgent 설계

3.1 시스템 개요

그림 1은 EnvAgent의 전체 시스템 아키텍처를 나타낸다. 본 시스템은 사용자의 하드웨어 및 운영체제 환경을 실시간으로 감지하여 실행 가능한 Conda 환경을 자율적으로 구축하며, (1) 로컬 입력 및 전처리, (2) AST 기반 의존성 추출 및 분석, (3) LLM 기반 환경 설정 생성, (4) 실행 및 반복 수정의 4단계 파이프라인으로 구성된다.

본 아키텍처의 핵심은 공유 컨텍스트와 하이브리드 지능의 결합에 있다. System Checker가 수집한 하드웨어 정보는 공유 컨텍스트에 저장되어 모든 에이전트가 플랫폼에 최적화된 결정을 내릴 수 있도록 한다. 또한 AST 분석과 같은 규칙 기반 처리를 선행하여 LLM의 토큰 사용량을 줄인다. Repair Agent는 런타임 오류 발생 시 이 컨텍스트를 기반으로 최대 8회까지 수정을 반복하는 자가 치유 루프를 통해 설치 성공률을 높인다.

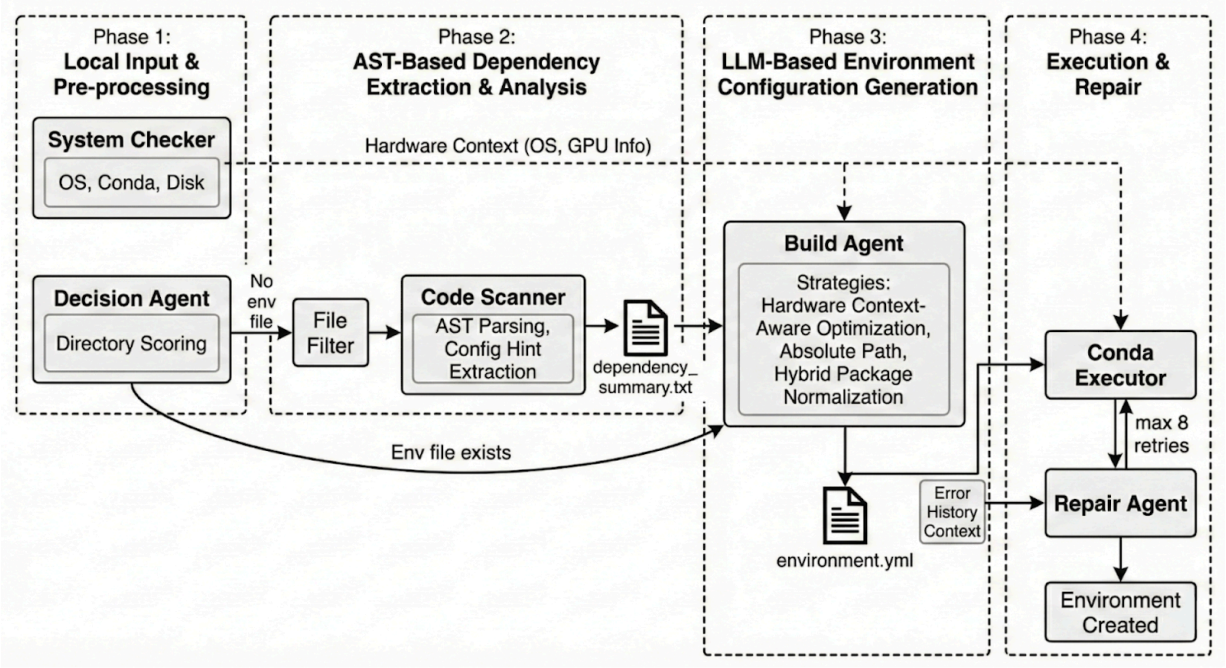


그림 1. EnvAgent 시스템 아키텍처

3.2 Local Input & Pre-processing

첫 번째 단계에서는 GitHub에서 클론한 로컬 AI/ML 프로젝트의 환경을 탐색하고 실행 가능 여부를 확인한다. System Checker가 운영체제, Conda 설치 여부, 디스크 공간 등 필수 요건을 검증하고, 수집된 시스템 정보를 공유 컨텍스트에 저장하여 이후 모든 에이전트가 참조할 수 있도록 한다.

Decision Agent는 프로젝트의 실제 루트 디렉토리를 식별한다. AutoGPT[17]처럼 여러 프로젝트를 단일 저장소에서 관리하는 Monorepo 구조에서는 설정 파일이 하위 디렉토리에 위치하는 경우가 많다. 이를 처리하기 위해 본 연구에서는 디렉토리 스코어링 알고리즘을 도입하였다. 에이전트는 하위 디렉토리를 순회하며 각 디렉토리에 존재하는 파일 유형에 따라 점수를 부여하고, 가장 높은 점수를 받은 디렉토리를 루트로 선정한다. 표 1은 파일 유형별 배점 기준을 나타낸다.

점수	대상 파일	배점 근거
+10	setup.py, pyproject.toml, environment.yml	명시적 프로젝트 설정
+5	requirements.txt, 디렉토리(src, app)	보조적 정황 증거
0	.py 소스 파일	일반적 분포
-10	디렉토리(docs, tests, examples, scripts)	비실행 컨텍스트

표 1. 디렉토리 스코어링 배점 기준

디렉토리 스코어링 알고리즘은 파일 유형과 디렉토리 명칭에 가중치를 부여하여 최적의 프로젝트 루트를 식별한다. 비실행 경로에는 네거티브 스코어링을 적용하고, node_modules나 .git 등은 탐색 대상에서 제외하여 분석 속도를 유지한다.

3.3 AST-Based Dependency Extraction & Analysis

본 단계는 LLM 호출 비용을 최적화하고 분석의 정확도를 높이기 위해 로컬 환경에서 수행되는 전처리 과정이다. Code Scanner Agent는 파이썬의 추상 구문 트리(AST) 분석과 설정 파일 힌트 추출을 결합하여 프로젝트의 의존성 정보를 구조적으로 파악한다.

첫째, AST 기반 소스 코드 순회이다. 에이전트는 .py 파일뿐만 아니라 데이터 과학 분야에서 흔히 사용되는 Jupyter Notebook(.ipynb)의 코드 셀까지 파싱하여 import 구문을 추출한다. 이때 sys, os와 같은 파이썬 표준 라이브러리는 필터링하여 분석 대상에서 제외하고, 외부 라이브러리에만 집중한다. 이를 통해 LLM에 전달되는 토큰 수를 절감한다. 또한 소스 코드 내 cuda, torch.device 등의 키워드를 스캐닝하여 GPU 필요 여부를 식별한다.

둘째, 설정 파일 기반 힌트 추출이다. AST 분석만으로는 동적 임포트나 구체적인 버전 제약 조건을 파악하기 어렵다. 이를 보완하기 위해 requirements.txt, setup.py, pyproject.toml 등의 설정 파일이 존재할 경우, 해당 내용을 텍스트 형태의 힌트로 추출한다.

최종적으로 추출된 정보는 dependency_summary.txt로 통합되며, 전체 코드 대신 이 요약본만 LLM에 전달하여 대규모 프로젝트에서도 효율적으로 처리할 수 있다.

3.4 LLM-Based Environment Configuration Generation

분석된 의존성 정보를 바탕으로 Build Agent가 실제 실행 가능한 Conda 환경 설정 파일(environment.yml)을 생성하는 단계이다. 이 과정에서 LLM은 단순한 텍스트 생성을 넘어, 이전 단계에서 수집되어 공유 컨텍스트에 유지된 실시간 하드웨어 정보를 기반으로 다음과 같은 3가지 지능형 빌드 전략을 수행한다.

첫째, 실시간 하드웨어 컨텍스트 기반의 패키지 최적화이다. System Checker는 초기 진단 단계에서 nvidia-smi를 통해 GPU 드라이버 버전을 감지하거나, macOS의 system_profiler를 통해 Apple Silicon 여부를 식별하여 공유 컨텍스트에 저장한다. Build Agent는 이 컨텍스트를 프롬프트에 주입하여 타겟 시스템에 적합한 패키지를 선별한다. 예를 들어, Apple Silicon 환경에서는 CUDA 패키지를 배제하고 conda-forge 채널의 ARM64 호환 패키지를 우선 선택하여 아키텍처 불일치로 인한 빌드 오류를 방지한다.

둘째, 로컬 의존성의 절대 경로 주입이다. pip install -e .와 같은 현재 디렉토리 기반 설치는 에이전트의 작업 디렉토리에 따라 실패할 수 있다. Build Agent는 타겟 프로젝트의 절대 경로를 파악하여 설정 파일에 명시적으로 주입한다. 이를 통해 LLM이 실행 위치와 무관하게 정확한 경로를 참조할 수 있어 빌드 안정성이 높아진다.

셋째, 임포트 명칭과 패키지 설치 명칭 간 불일치 해결이다. 파이썬에서는 소스 코드의 임포트명(예: cv2, sklearn)과 pip 설치명(예: opencv-python, scikit-learn)이 다른 경우가 많아 의존성 설치가 실패하곤 한다. Build Agent는 사전 정의된 매핑 테이블로 우선 변환을 시도하고, 테이블에 없는 패키지는 LLM이 문맥을 바탕으로 올바른 설치명을 추론한다.

3.5 Execution & Iterative Repair Loop

마지막 단계에서는 생성된 environment.yml 파일로 실제 Conda 환경을 생성한다. Conda Executor는 conda env create 명령어를 실행하고 종료 코드를 확인한다. 앞선 단계의 절대 경로 주입과 OS 호환성 검사를 통해 초기 설치 실패율을 낮출 수 있다.

그럼에도 버전 충돌과 같은 런타임 오류로 설치가 실패할 경우, 오류 로그와 현재 설정 파일이 Repair Agent에게 전달된다. Repair Agent는 LLM을 활용하여 오류 유형을 분석하고 그에 맞는 수정 전략을 수행한다. 이때, 이전 시도의 오류 기록을 LLM 컨텍스트에 포함하여 동일한 실패를 반복하지 않도록 한다. 구체적으로 PackageNotFoundError나 C/C++ 확장 모듈의 빌드 오류가 발생할 경우, 해당 패키지를 pip 섹션에서 Conda 바이너리 섹션으로 이동시키는 전략을 우선 적용한다. UnsatisfiableError 발생 시에는 충돌 패키지의 버전 제약을 단계적으로 완화하거나 conda-forge 등 다른 채널에서 패키지를 탐색한다. LLM이 유효한 수정안을 제시하지 못하거나 수정된 파일이 이전과 동일할 경우, 규칙 기반 대체 전략이 작동한다. 이 대체 전략은 버전 고정(==)을 제거하여 Conda가 호환되는 버전을 자유롭게 선택할 수 있도록 한다.

수정된 설정 파일로 재설치를 시도하는 자가 치유 루프가 형성되며, 최대 재시도 횟수는 8회로 제한하였다. 초기 테스트 결과, 해결 가능한 오류는 8회 이내에 수정되었고, 이후에는 수정 효과가 나타나지 않았다. 8회를 초과하는 실패는 의존성 조정으로 해결할 수 없는 구조적 문제로 간주하여 루프를 종료한다.

4. 결 과

본 장에서는 제안한 EnvAgent 시스템의 성능을 정량적으로 평가한다. 실험은 실제 GitHub의 AI/ML 오픈소스 프로젝트를 대상으로 수행하였으며, 실행 가능한 Conda 환경의 자동 구축 여부를 평가하였다. 본 연구는 다음의 2가지 연구 질문에 답하는 것을 목표로 한다.

- **RQ1**: EnvAgent는 다양한 라이브러리를 포함한 실제 AI/ML 저장소에 대해 실행 가능한 환경을 정확하게 생성하는가?
- **RQ2**: 프로젝트 규모와 복잡도는 EnvAgent의 환경 생성 성공률에 부정적인 영향을 미치는가?

4.1 실험 설정

EnvAgent의 OS별 대응 능력과 GPU 지원 여부를 검증하기 위해, Apple Silicon(macOS)과 NVIDIA GPU가 장착된 Linux 서버에서 실험을 수행하였다. 실험 대상은 AI/ML 라이브러리와 Unit Test 프레임워크를 갖춘 공개 저장소 9개를 선정하고, 소스 코드 라인 수(LOC)를 기준으로 3개 그룹으로 분류하였다.

- **소규모**: 15,000 LOC 이하(3개)
- **중규모**: 15,000 ~ 100,000 LOC(3개)
- **대규모**: 100,000 LOC 이상(3개)

4.2 평가 기준

본 연구에서는 단순히 environment.yml 파일이 생성되는 것을 넘어, 실제 런타임에서의 실행 가능성을 성공의 기준으로 삼는다.

실험 규모	GitHub 저장소	#LOC	Success
소규모 #LOC<15,000	ageitgey/face_recognition[18]	3,653	✔
	scikit-learn-contrib/DESlib[19]	8,743	✔
	keras-team/autokeras[20]	10,748	X
중규모 #LOC<100,000	ultralytics/yolov5[21]	18,587	✔
	scikit-learn-contrib/imbalanced-learn[22]	20,666	✔
	microsoft/LightGBM[23]	97,348	X
대규모 #LOC>100,000	pytorch/vision[24]	100,426	✔
	Netflix/metaflow[25]	120,096	✔
	scikit-learn/scikit-learn[26]	296,096	✔

표 2. *실험 대상 GitHub 저장소 목록 및 규모*

성공 조건은 (1) Conda 환경 오류 없이 생성, (2) 주요 의존성 import 성공, (3) Unit Test 실행 가능의 3가지를 모두 충족하는 경우이다. Repo2Run[9]의 방법론을 따라 개별 테스트의 Pass/Fail 여부는 평가에서 제외하였다.

4.3 실험 결과

4.3.1 RQ1: EnvAgent는 다양한 라이브러리를 포함한 실제 AI/ML 저장소에 대해 실행 가능한 환경을 정확하게 생성하는가?

총 9개 프로젝트 중 77.8%(7/9)에서 실행 가능한 Conda 환경 구축에 성공하였다. torch, tensorflow 등 하드웨어 의존성이 높은 라이브러리도 OS 환경에 맞춰 적절히 설치되었으며, Apple Silicon과 Linux GPU 환경 모두에서 정상 동작을 확인하였다.

EnvAgent는 77.8%의 성공률로 실행 가능한 Conda 환경을 자동 생성하였다.

4.3.2 RQ2: 프로젝트 규모와 복잡도는 EnvAgent의 환경 생성 성공률에 부정적인 영향을 미치는가?

실험 결과, EnvAgent는 소·중규모 프로젝트에서 각각 66.7%(2/3)의 성공률을 보였으며, 대규모 프로젝트에서는 오히려 100%(3/3)의 성공률을 달성하였다.

실패한 두 프로젝트를 분석한 결과, 실패 원인은 프로젝트 규모가 아닌 구조적 요인에 기인하였다. keras-team/autokeras는 상위 프레임워크에 대한 강한 내부 의존성이, microsoft/LightGBM은 C++ 라이브러리의 사전 빌드가 필요한 구조가 원인이었다.

환경 생성 실패는 프로젝트 규모가 아닌 구조적 요인에 기인하였다.

5. 향후 연구

실험에서 식별된 한계점을 바탕으로 향후 연구 방향을 제시한다.

첫째, Monorepo 구조 내 의존성 추론의 고도화이다. 현재의 디렉토리 스코어링 방식은 모듈 간 상호 참조가 빈번한 구조에서 한계를 보였으며, 프로젝트 전체의 의존성 그래프 분석을 통해 이를 개선하고자 한다.

둘째, 에이전트 주도의 런타임 검증 확장이다. 현재 시스템은 설치 완료만을 성공 지표로 삼고 있으나, Unit Test 실행 후 오류 로그를 분석하여 environment.yml을 재수정하는 2차 자가 치유 루프를 도입할 계획이다.

셋째, 시스템 수준의 의존성 자동 해결이다. LightGBM 실패 사례와 같이 C++ 네이티브 빌드가 필요한 프로젝트를 위해, 빌드 설정 파일(CMakeLists.txt 등)을 감지하고 Conda 전용 컴파일러 패키지(예: gxx_linux-64)를 자동 포함시키는 전략 연구가 필요하다.

6. 결 론

본 연구에서는 AI/ML 프로젝트의 복잡한 의존성 문제를 해결하고 실행 가능한 Conda 환경을 자동으로 생성하기 위해, 정적 분석과 거대 언어 모델(LLM)을 결합한 하이브리드 에이전트 시스템인 EnvAgent를 제안하였다. EnvAgent는 AST 기반의 정적 분석을 통해 LLM의 토큰 비용을 최소화하면서도 핵심 의존성을 정확히 추출하며, 최대 8회의 재시도를 수행하는 반복적 자가 치유 메커니즘을 도입하여 설치 과정에서 발생하는 다양한 오류를 자율적으로 복구하도록 설계되었다.

실제 GitHub 오픈소스 프로젝트를 대상으로 한 실험 결과, EnvAgent는 77.8%의 성공률로 런타임에서 정상 동작하는 환경을 구축하였다. 특히 10만 라인(LOC) 이상의 대규모 프로젝트에서도 성능 저하 없이 안정적으로 동작하여 시스템의 확장성과 강건성을 입증하였다. 이러한 결과는 비용 효율적인 정적 분석과 유연한 LLM 추론의 결합이 기존 도구들의 한계를 극복하고 파이썬 환경 재현성 문제를 해결하는 효과적인 접근법임을 시사한다. 향후에는 의존성 그래프 분석, 2차 자가 치유 루프, C++ 빌드 의존성 자동 감지를 통해 시스템의 완성도를 높일 계획이다.

※ 본 연구는 2025년 과학기술정보통신부 및 정보통신기획평가원의 SW중심대학사업(2023-0-00055)과 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원(RS-2024-00457866)을 받아 수행된 연구입니다.

참조문헌

[1] Raschka, Sebastian, Joshua Patterson, and Corey Nolet. "Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence." Information 11, no. 4 (2020): 193. https://doi.org/10.3390/info11040193

[2] GitHub. 2025. "Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1." The GitHub Blog. October 2025. https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/

[3] Huang, Kaifeng, Bihuan Chen, Susheng Wu, Junming Cao, Lei Ma, and Xin Peng. "Demystifying dependency bugs in deep learning stack." In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 450-462. 2023.

[4] Samuel, Sheeba, and Daniel Mietchen. "Computational reproducibility of Jupyter notebooks from biomedical publications." GigaScience 13 (2024): giad113.

[5] Semmelrock, Harald, Tony Ross-Hellauer, Simone Kopeinik, Dieter Theiler, Armin Haberl, Stefan Thalmann, and Dominik Kowald. "Reproducibility in machine-learning-based research: Overview, barriers, and drivers." AI Magazine 46, no. 2 (2025): e70002.

[6] Merkel, Dirk. "Docker: lightweight linux containers for consistent development and deployment." Linux j 239, no. 2 (2014): 2.

[7] Horton, Eric, and Chris Parnin. "Dockerizeme: Automatic inference of environment dependencies for python code snippets." In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 328-338. IEEE, 2019.

[8] Bartlett, Antony, Cynthia Liem, and Annibale Panichella. "Raiders of the Lost Dependency: Fixing Dependency Conflicts in Python using LLMs." arXiv preprint arXiv:2501.16191 (2025).

[9] Hu, Ruida, Chao Peng, Xinchen Wang, Junjielong Xu, and Cuiyun Gao. "Repo2Run: Automated Building Executable Environment for Code Repository at Scale." arXiv preprint arXiv:2502.13681 (2025).

[10] Haque, Mubin Ul, Leonardo Horn Iwaya, and M. Ali Babar. "Challenges in docker development: A large-scale study using stack overflow." In Proceedings of the 14th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM), pp. 1-11. 2020.

[11] Wu, Yiwen, Yang Zhang, Kele Xu, Tao Wang, and Huaimin Wang. "Understanding and predicting docker build duration: An empirical study of containerized workflow of oss projects." In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1-13. 2022.

[12] Grüning, Björn, Ryan Dale, Andreas Sjödin, Brad A. Chapman, Jillian Rowe, Christopher H. Tomkins-Tinch, Renan Valieris, Johannes Köster, and Bioconda Team. "Bioconda: sustainable and comprehensive software distribution for the life sciences." Nature methods 15, no. 7 (2018): 475-476.

[13] Bndr. n.d. "GitHub - Bndr/Pipreqs: Pipreqs - Generate Pip requirements.txt File Based on Imports of Any Project. Looking for Maintainers to Move This Project Forward." GitHub. https://github.com/bndr/pipreqs.

[14] Damnever. n.d. "GitHub - damnever/pigar: :coffee: A tool to generate requirements.txt for Python project, and more than that. (IT IS NOT A PACKAGE MANAGEMENT TOOL)." GitHub. https://github.com/damnever/pigar

[15] Ye, Hongjie, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. "Knowledge-based environment dependency inference for Python programs." In Proceedings of the 44th International Conference on Software Engineering, pp. 1245-1256. 2022.

[16] Milliken, Louis, Sungmin Kang, and Shin Yoo. "Beyond pip install: Evaluating llm agents for the automated installation of python projects." In 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 1-11. IEEE, 2025.

[17] Significant-Gravitas. n.d. "GitHub - Significant-Gravitas/AutoGPT: AutoGPT is the vision of accessible AI for everyone, to use and to build on. Our mission is to provide the tools, so that you can focus on what matters." GitHub. https://github.com/Significant-Gravitas/AutoGPT

[18] Ageitgey. n.d. "GitHub - ageitgey/face_recognition: The world's simplest facial recognition api for Python and the command line." GitHub. https://github.com/ageitgey/face_recognition

[19] Scikit-Learn-Contrib. n.d. "GitHub - scikit-learn-contrib/DESlib: A Python library for dynamic classifier and ensemble selection." GitHub. https://github.com/scikit-learn-contrib/DESlib

[20] Keras-Team. n.d. "GitHub - keras-team/autokeras: AutoML library for deep learning." GitHub. https://github.com/keras-team/autokeras

[21] Ultralytics. n.d. "GitHub - ultralytics/yolov5: YOLOv5 in PyTorch > ONNX > CoreML > TFLite." GitHub. https://github.com/ultralytics/yolov5

[22] Scikit-Learn-Contrib. n.d. "GitHub - scikit-learn-contrib/imbalanced-learn: A Python Package to Tackle the Curse of Imbalanced Datasets in Machine Learning." GitHub. https://github.com/scikit-learn-contrib/imbalanced-learn

[23] Microsoft. n.d. "GitHub - microsoft/LightGBM: A fast, distributed, high performance gradient boosting (GBT, GBDT, GBRT, GBM or MART) framework based on decision tree algorithms, used for ranking, classification and many other machine learning tasks." GitHub. https://github.com/microsoft/LightGBM

[24] Pytorch. n.d. "GitHub - pytorch/vision: Datasets, Transforms and Models specific to Computer Vision." GitHub. https://github.com/pytorch/vision

[25] Netflix. n.d. "GitHub - Netflix/metaflow: Build, Manage and Deploy AI/ML Systems." GitHub. https://github.com/Netflix/metaflow

[26] Scikit-Learn. n.d. "GitHub - scikit-learn/scikit-learn: scikit-learn: machine learning in Python." GitHub. https://github.com/scikit-learn/scikit-learn

진화하는 AI 에이전트를 위한 적응형 런타임 테스팅의 필요성

왕자오얀, 안현준 ○, 고인영

카이스트

zhaoyan123@kaist.ac.kr, a.hyunjun@kaist.ac.kr, iko@kaist.ac.kr

Adaptive Runtime Testing Is Essential for Evolving AI Agents

Zhaoyan Wang, Hyunjun Ahn○, In-Young Ko

KAIST

Abstract

AI systems increasingly operate in dynamic, evolving environments where models must adapt continuously. Traditional testing, however, assumes static architectures and fixed correctness, leaving post-deployment evolution unverified. We suggest that testing must itself become adaptive, evolving alongside the system it evaluates. We identify three critical gaps—conceptual (what counts as testable evolution), methodological (how to verify during runtime), and data-related (how to maintain relevance under drift). We propose an adaptive runtime testing framework structured around three dimensions: the evolution boundary (safe adaptation limits), test memory (preservation of verified behaviors), and acceptance threshold (criteria for trustworthy updates). Integrated across the edge-cloud hierarchy, the framework supports continuous monitoring and adaptive safety assurance throughout system evolution. Adaptive runtime testing transforms verification from a static audit into an ongoing process, ensuring that learning systems remain safe, interpretable, and accountable as they evolve.

1. Introduction

AI-enabled autonomous systems such as autonomous driving system (ADS) have reached a level of complexity where pre-deployment testing alone is no longer sufficient [1]. Traditional testing pipelines are designed under assumptions that the system being tested remains static, where model parameters, operating environment, and behavioral logic are all fixed during evaluation as shown in Figure 1 (a) [2, 3]. However, these assumptions break down once an ADS is released into the real world [3]. As demonstrated in Figure 1 (b), an agent operates in dynamic environments, experiences continual sensor drift, and faces unforeseen road conditions that were never represented in the training data. As a result, decision-making models must update and adapt over time to maintain reliability. Yet every adaptation introduces new uncertainty: the system tested yesterday is no longer identical to the system driving today [1, 4].

Runtime testing emerges as a response to this challenge [5, 6]. Instead of verifying the system only before deployment, runtime testing embeds the evaluation process directly into the operation of the agent itself [7]. The goal is not only to detect errors, but to continuously monitor whether the system remains safe and predictable as it evolves. This means that testing becomes an ongoing, adaptive process that is tightly coupled to the system's perception, decision, and control loops. Such a paradigm shift transforms testing from an external quality check into an intrinsic function of autonomy [6].

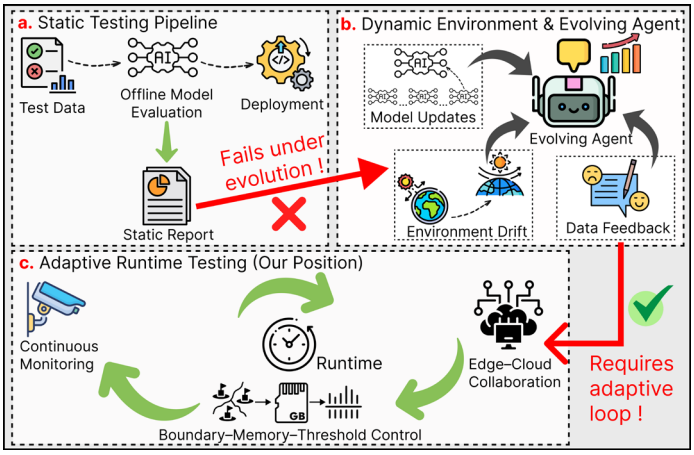


Figure 1 Static versus adaptive testing paradigms. Adaptive runtime testing embeds evaluation within system operation, enabling continuous assurance for evolving agents.

However, applying runtime testing introduces unique difficulties. Unlike static models, an evolving agent continuously modifies its internal state and decision boundaries in response to changing data [8, 9]. Conventional regression tests quickly lose relevance, as each new model version may interpret inputs differently [10].

Moreover, there is no definitive standard to distinguish whether a model's evolution represents a safe and valid adaptation or an unsafe deviation from its intended behavior. In this context, the boundaries of what is "tested" must evolve alongside the system itself.

Adaptive runtime testing requires mechanisms for defining and validating the limits of safe evolution. Specifically, we suggest that three dimensions become central: the **evolution boundary**, which defines how far the system is allowed to change while remaining within safety constraints; the **test memory**, which ensures that new adaptations do not erase previously verified behaviors; and the **acceptance threshold**, which quantifies when an evolved model can be trusted to replace its predecessor.

We propose that AI testing must evolve into an adaptive, runtime process, capable of dynamically validating self-updating models that operate in the real world. Specifically, we stress that testing for self-evolving autonomous driving systems should be structured around three interdependent concepts: (1) **Evolution Boundaries**, (2) **Test Case Forgetting (Memory)**, and (3) **Acceptance Thresholds**. These concepts form the foundation of an adaptive runtime testing framework that maintains trust in evolving systems.

By re-conceptualizing testing as a dynamic, in-operation process, we can enable self-evolving autonomous systems that adapt intelligently without compromising safety. This paper aims not to explain technical details, but to analyze the necessity of ***adaptive runtime testing*** and to present its conceptual framework. This paper contributes (i) an analysis of why static testing paradigms fail under continuous evolution, (ii) a definition of adaptive runtime testing principles grounded in boundary monitoring and dynamic validation, and (iii) a synthesis of unsolved gaps in current testing methodologies. The goal is to transform testing from a one-time certification procedure into an ongoing safety assurance loop, as shown in Figure 1 (c), so that autonomous systems can not only learn, but learn safely.

2. Gaps: Limits of Static Testing

As learning agents begin to evolve, an entity being tested becomes a moving target, and static validation no longer guarantees safety or reliability [2]. The failure of current approaches reveals three fundamental gaps that prevent existing methodologies from supporting self-evolving agents.

2.1. Conceptual Gap

Defining Testable Evolution. Traditional testing presumes a stable definition of correctness. Once an ADS begins to change autonomously, this notion breaks down. There is no shared framework for defining what constitutes "safe" or "testable" evolution. If an adaptive model modifies its internal decision boundaries, should success be judged by its original specification or by its updated objectives? Without clear criteria for acceptable evolution, post-deployment learning cannot be distinguished from system drift. This lack of conceptual clarity prevents testing frameworks from defining clear boundaries for safe evolution.

2.2. Methodological Gap

Testing During Runtime. Even if acceptable evolution could be defined, there remains no method to verify it in real time. Static testing is inherently episodic. It validates a snapshot of system behavior at discrete points in time. In contrast, self-evolving agents continuously update in response to feedback, rendering traditional regression tests obsolete. Once the model changes, earlier test cases may no longer measure equivalent functionality. Current practices provide no mechanism to generate adaptive test cases or validate model updates under live conditions. As a result, model evolution proceeds without synchronized verification.

2.3. Data and Environment Gap

Self-evolving systems operate in open, non-stationary environments [11]. Sensor degradation, seasonal variation, and interactions constantly reshape the data distribution on which the system relies [12, 13]. Static validation datasets quickly become outdated to represent the system's evolving operational domain. This leads to data drift, where the real-world context diverges from the training and test conditions, producing unmeasured failure modes. Without continuous adaptation of test data and contextual evaluation, even well-trained systems may degrade silently while appearing valid under obsolete benchmarks. In other words, current testing paradigms are structurally misaligned with evolving systems. They lack conceptual clarity about what constitutes testable evolution, methodological capacity to perform testing during runtime, and data adaptability to reflect shifting environmen-ts. These gaps underscore the need for new testing paradigms for adaptive runtime testing.

3. Position: Adaptive Runtime Testing

3.1. Addressing the Conceptual Gap

Adaptive testing for self-evolving autonomous systems requires more than monitoring runtime performance: it demands a structured framework to constrain, preserve, and evaluate system evolution. We identify three interdependent dimensions that together define how an evolving system can remain trustworthy throughout the life-cycle of adaptation.

Evolution Boundary: Constraining How Far a System May Evolve. The evolution boundary defines the permissible range of behavioral or model changes that an autonomous system can undergo without compromising safety or core functionality. It acts as a set of predefined safety corridors

that specify how far the system may adapt while still being considered valid. Boundaries are not single numerical limits but multi-dimensional envelopes encompassing perception accuracy, decision latency, control stability, and other safety-critical metrics. For instance, an ADS may be allowed to adjust its perception model to handle new weather conditions as long as detection accuracy does not drop below a defined margin and reaction time remains within certified limits. By explicitly setting these evolution boundaries before runtime adaptation, developers establish a measurable zone of safe learning and change. Crossing a boundary indicates that the system's adaptation has entered unsafe or unverified territory, triggering intervention.

Test Memory: Preserving What Must Not Be Forgotten. The test memory mechanism ensures that, as the system evolves, it retains competence on previously verified scenarios. Self-evolving models risk "catastrophic forgetting", where learning new patterns causes the loss of established capabilities [14]. In the context of adaptive testing, test memory functions as a continuous regression safeguard that automatically re-validates old test cases alongside new adaptations. For an evolving functional model, this may involve rerunning key perception and control tests from previously validated datasets each time the model updates. If the updated model fails cases that were once passed, it indicates functional regression rather than safe adaptation.

Acceptance Threshold: Deciding When Evolution is Safe Enough. The acceptance threshold quantifies whether an evolved model has achieved a sufficient balance of improvement and stability to be trusted for deployment. It operates as a decision rule that integrates both the evolution boundary and test memory: the system must remain within boundary constraints and retain past capabilities before any update. In practice, acceptance thresholds may be defined as numerical margins or composite scores across safety and performance metrics. For example, a new control model may be accepted if it improves trajectory stability by at least 5% in new conditions while maintaining at least 98% success on prior safety tests. If the new model fails to meet these thresholds, the adaptive testing framework rejects the update or flags it for further verification. Acceptance thresholds thus serve as operational gatekeepers that translate adaptive testing results into deployable decisions.

3.2. Addressing the Methodological Gap

Runtime Testing in the Distributed Service Hierarchy. The second challenge lies in how to perform testing during operation. Since static regression frameworks cannot verify models that adapt on short timescales, any re-validation lag immediately undermines real-time safety and performance goals. We propose a distributed testing framework that splits verification responsibilities across the edge-cloud service hierarchy. At the edge, the AI agent works alongside lightweight, co-located monitoring modules that perform continuous, real-time safety checks. These local systems continuously monitor inputs and outputs for immediate unsafe drift and are equipped to trigger instantaneous safety interventions or service rollbacks on the individual instance. The cloud layer serves as the central hub for long-term governance, performing asynchronous large-scale verification through the ingestion and analysis of aggregated operational data. Within this centralized pipeline, comprehensive offline validation is conducted across diverse simulated scenarios to ensure global consistency over time and to update the acceptance rules. The edge-cloud architecture balances fast, local responsiveness at the edge with reliable, large-scale oversight in the cloud, integrating testing as a built-in function across both layers of the AI service pipeline.

3.3. Addressing the Data and Environment Gap

Co-Evolving Test Data. A third challenge arises from environmental and data drift. As real-world conditions change, fixed test datasets quickly lose representativeness, leading to unmeasured degradation. The adaptive testing process must therefore evolve its data basis in tandem with the system and its surroundings. In the proposed edge-cloud architecture, the edge continuously collects contextual data and flags segments where model confidence or consistency deteriorates. These data streams are aggregated by the cloud, which detects distribution shifts and synthesizes updated test suites. The resulting drift-aware test cases are periodically redeployed to the edge for localized revalidation. In this way, testing data evolve along with the operating environment, maintaining relevance as conditions change. We consider testing not as a static audit, but as a living process that adapts in space and time. Conceptually, it redefines correctness as bounded evolution; methodologically, it enables runtime verification through edge-cloud distribution; and environmentally, it ensures representativeness through adaptive data flow. Together, these principles constitute the foundation of adaptive runtime testing, a paradigm where learning systems can evolve safely, transparently, and continuously.

4. Alternative Views

The necessity of adaptive runtime testing for evolving AI systems contrasts with several established positions in AI safety and system engineering. Reviewing these views clarifies the boundaries and rationale of our proposal.

Runtime testing may add operational overhead. Critics argue that runtime validation conflicts with real-time constraints. Adaptive runtime testing, however, relies on

lightweight runtime checks combined with deferred offline or cloud-based analysis, enabling continuous assurance without full in-situ verification.

Continuous evolution challenges regulatory compliance. Regulatory frameworks typically assume static system configurations. We argue that adaptive runtime testing enables certifiable evolution by enforcing explicit boundaries and acceptance criteria, allowing controlled adaptation rather than uncontrolled drift.

Static retraining may be safer than runtime adaptation. Limiting learning to offline retraining may suffice in closed environments but fails in open, dynamic settings. Adaptive runtime testing supports bounded and monitored adaptation when static models become unreliable.

Self-validating systems risk circular assurance. Concerns about circular assurance overlook that runtime testing can be externally grounded through predefined oracles and audit mechanisms, complementing human and regulation.

5. Conclusion

Adaptive runtime testing redefines how we assure safety in learning systems. As AI agents evolve beyond their original specifications, static validation can no longer guarantee reliability. Embedding testing into runtime operation allows systems to adapt while remaining verifiable, accountable, and safe. By integrating evolution boundaries, test memory, and acceptance thresholds within an edge-cloud hierarchy, we enable continuous assurance throughout system evolution. Ultimately, adaptive runtime testing transforms verification from a one-time audit into an enduring safety loop that ensures autonomous systems not only learn but also learn responsibly.

Acknowledgement

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (No. RS-2025-02218761, 75% and No. RS-2024-00406245, 25%).

References

[1] Rauba, Paulius, et al. "Self-healing machine learning: A framework for autonomous adaptation in real-world environments." *Advances in Neural Information Processing Systems 37*. 2024.

[2] Cheng, Mingfei, Yuan Zhou, and Xiaofei Xie. "Behavexplor: Behavior diversity guided testing for autonomous driving systems." *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023.

[3] Fulton, Nathan, et al. "Formal verification of end-to-end learning in cyber-physical systems: Progress and challenges." *ArXiv preprint arXiv:2006.09181*. 2020.

[4] Yu, En, et al. "Online boosting adaptive learning under concept drift for multistream classification." *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. No. 15. 2024.

[5] Carwehl, Marc, et al. "Runtime verification of self-adaptive systems with changing requirements." *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE. 2023.

[6] Caldas, Ricardo, et al. "Runtime verification and field-based testing for ros-based robotic systems." *IEEE Transactions on Software Engineering*. 2024.

[7] Dong, Shuyang, et al. "Quantitative Predictive Monitoring and Control for Safe Human-Machine Interaction." *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 39. No. 25. 2025.

[8] Ghosh, Anurag, et al. "Chanakya: Learning runtime decisions for adaptive real-time perception." *Advances in Neural Information Processing Systems* 36. 2023.

[9] You, Hanmo, et al. "Navigating the Testing of Evolving Deep Learning Systems: An Exploratory Interview Study." *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025.

[10] Liu, Yu, et al. "More precise regression test selection via reasoning about semantics-modifying changes." *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023.

[11] Stocco, Andrea, et al. "Misbehaviour prediction for autonomous driving systems." *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 2020.

[12] Lou, Guannan, et al. "Testing of autonomous driving systems: where are we and where should we go?." *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022.

[13] Sun, Tao, et al. "SHIFT: a synthetic driving dataset for continuous multi-task domain adaptation." *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022.

[14] Kemker, Ronald, et al. "Measuring catastrophic forgetting in neural networks." *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. No. 1. 2018.

EDGE 컴퓨팅 기기에서의 소형 객체 탐지를 위한 선택적 혼합 정밀도 기반 양자화 모델 성능 개선 연구

임다희°, 박지훈*
충남대학교 컴퓨터공학과
seeyldh@naver.com, jihun.park@cnu.ac.kr

Selective Mixed-Precision Quantization for Improving Small Object Detection on Edge Devices

Dahee Lim°, Jihun Park*
Department of Computer Science and Engineering, Chungnam National University

요 약

최근 엣지 컴퓨팅 환경에서 드론 등을 활용한 실시간 객체 탐지 수요가 증가함에 따라 제한된 하드웨어 자원에서 추론 속도를 높이기 위한 모델 경량화 기술이 중요해지고 있다. 그중 INT8 양자화(Quantization)는 널리 사용되는 기법이나, 소형 객체 탐지(Small Object Detection, SOD)와 같이 시각적 정보가 희소한 작업에서는 심각한 성능 저하를 유발하는 경향이 있다. 본 연구는 엣지 컴퓨터인 NVIDIA Jetson Orin Nano 에서 YOLOv11n 모델을 이용해 이러한 성능 저하의 원인을 분석하고 효율적인 해결책을 제안한다. 분석 결과, SOD 데이터셋(UAVDT)은 일반 데이터셋(PASCAL VOC)에 비해 초기 레이어에서 활성값(Activation)의 동적 범위가 넓게 형성되며, 이로 인해 양자화 오차가 증폭되어 탐지 성능이 하락함을 확인하였다. 이를 해결하기 위해 본 논문은 모델 전체를 고정밀도로 연산하는 대신, 양자화 민감도가 높은 초기 특징 추출 구간의 레이어만을 선별하여 FP16 정밀도를 유지하는 선택적 혼합 정밀도(Selective Mixed Precision) 전략을 제안한다. 실험 결과, 제안된 기법은 INT8 기반 엔진 모델 대비 추론 지연시간의 증가는 최소화하면서도 mAP50 성능을 약 2.5%p 향상시켜, 엣지 디바이스 기반 SOD 작업에서 정확도와 효율성의 균형을 효과적으로 달성함을 입증하였다.

1. 서 론

최근 드론, 자율주행 차량, 로봇 등 소형·경량·저전력 컴퓨팅을 필요로 하는 엣지(Edge) 컴퓨팅 환경에서 딥러닝 기술의 활용이 급증하고 있다. 특히 드론을 활용한 감시 정찰이나 조난자 수색과 같은 응용 분야에서는 원거리에서 촬영된 영상 내의 객체가 매우 작게 나타나는 소형 객체 탐지(Small Object Detection, SOD) 능력이 필수적이다. 이러한 환경에서는 대상이 빠르게 이동하거나 관측 조건이 수시로 변화하므로 실시간 탐지가 요구되며, 통신 지연이나 보안 문제를 방지하기 위해 서버 의존도를 줄인 엣지컴퓨터에서의 온디바이스(On-device) 추론이 선호된다. 그러나 엣지 컴퓨터는 연산 자원과 메모리가 제한적이기 때문에, 탐지 정확도를 유지하면서도 추론 지연시간을 최소화해야 하는 Trade-off를 해결하는 것이 중요한 과제이다.

제한된 자원 내에서 추론 속도를 가속화하기 위해 일반적으로 INT8 양자화(Quantization) 기법이 널리 사용되지만, 이는 필연적으로 정밀도 감소를 동반하며 특히 SOD 작업에서 그 성능 저하가 두드러지게 나타난다. 일반적인 객체 탐지와 달리 SOD 환경에서 양자화로 인한 정보 손실이 치명적인 이유는 다음과 같다. Figure 1에서 확인할 수 있듯이, 일반적인 객체 탐지 데이터셋인 PASCAL VOC[1]에 비해 UAVDT[2] 데이터셋의 객체들은 이미지 내에서 차지하는 영역이 극히 작다. 이러한 시각적 정보의 희소성으로 인해 첫째, 소형 객체는 이미지 내에서 차지하는 픽셀 수가 적어 시각적 정보가 희소하므로, 양자화 과정에서 발생하는 미세한 수치 오차에도 특징(Feature)이 쉽게 소실되거나 왜곡될 수 있다. 둘째, 본 연구의 분석에 따르면 SOD 특성이 강한 데이터셋(UAVDT 등)은 일반 데이터셋(PASCAL VOC 등)에 비해 초기 레이어에서 활성값(Activation)의 동적 범위(Dynamic Range)가 넓게 형성되는 경향이 있다. 넓은 활성값 범위는 고정된 8-bit로 매핑될 때 표현 간격(Scale)을 넓혀 양자화 오차(Quantization Error)를 증가시키며, 초기 레이어에서 발생한 이러한 오차는 후속 레이어로 전파되어 최종 탐지 성능에 악영향을 미친다.

기존의 연구들은 주로 일반적인 객체 탐지나 분류 문제에 집중되어 있어, 이러한 SOD 특유의 양자화 민감도와 레이어별 특성을 고려한 최적화 논의는 부족한 실정이다. 이에 본 연구에서는 Jetson Orin Nano

Figure 1 PASCAL VOC 2007 데이터셋(좌) 과 UAVDT 데이터셋(우)의 예시 이미지

환경에서 엣지 실시간 추론을 위한 경량모델인 YOLOv11n을 대상으로, SOD 수행 시 발생하는 INT8 양자화 성능 저하의 원인을 레이어별 Activation 분포 관점에서 분석한다. 분석 결과를 바탕으로, 본 연구는 모델 전체를 고정밀도로 연산하거나 일괄적으로 양자화하는 대신, 양자화 오차에 민감한 초기 특징 추출 구간의 일부 레이어만을 선별하여 FP16 정밀도를 유지하는 선택적 혼합 정밀도(Selective Mixed Precision) 전략을 제안한다. 제안하는 방법은 전체 연산 비용을 크게 증가시키지 않으면서도, 초기 레이어의 정보 손실을 방지하여 INT8 추론의 효율성과 탐지 정확도 사이의 균형을 효과적으로 달성함을 실험적으로 입증하고자 한다.

2. 관련 연구

딥러닝 모델의 엣지 디바이스 배포를 위한 경량화 기법 중, 양자화는 메모리 대역폭과 연산 비용을 획기적으로 절감할 수 있는 핵심 기술이다. 특히 훈련 후 양자화(Post-Training Quantization, PTQ)는 재학습 없이 모델을 변환할 수 있어 실용적이나, 활성값(Activation)의 분포 특성에 따라 성능 편차가 크게 발생한다. 본 절에서는 Activation 분포가 야기하는 양자화 난이도 문제와 이를 해결하기 위한 혼합 정밀도(Mixed Precision) 연구들을 고찰하고, 소형 객체 탐지 환경에서 본 연구가 갖는 차별성을 논의한다.

2.1 Activation 분포와 양자화 난이도

양자화 오차는 가중치보다 활성값의 분포에 더 큰 영향을 받는 경향이 있다. 최근 Xiao et al.의 SmoothQuant[3] 연구나 LLM.int8()[4] 연구 등은 모델의 규모가 커지거나 특정 데이터 입력에서 Activation 내에 극단적인 이상치가 발생할 경우, 고정된 INT8 범위로의 선형 매핑이 큰 양자화 손실을 유발함을 지적하였다. 이러한 연구들은 주로 수학적 평활화기법을 통해 Activation의 난이도를 가중치 쪽으로 이관하거나, 이상치 채널만 분리하여 연산하는 방식을 제안하였다. 하지만 이러한 기법들은 주로 대규모 언어 모델(LLM)이나 트랜스포머 구조에 집중되어 있으며, 엣지 디바이스상의 실시간 객체 탐지 모델(YOLO 등)에 그대로 적용하기에는 연산 오버헤드가 발생하거나 하드웨어 지원 제약이 따를 수 있다. 본 연구는 복잡한 변환 없이, SOD 데이터셋이 갖는 고유한 특성을 분석하여 레이어 단위의 정밀도 조절만으로 문제를 완화한다는 점에서 차이가 있다.

2.2 민감도 기반 혼합 정밀도(Mixed Precision) 양자화

모든 레이어를 동일한 비트 수(예: INT8)로 양자화할 때 발생하는 성능 저하를 방지하기 위해, 레이어 별 민감도를 분석하여 비트 폭을 차등 할당하는 혼합 정밀도 기법이 활발히 연구되었다. 대표적으로 Dong et al.의 HAWQ[5]는 헤시안행렬의 고윳값을 기반으로 각 레이어가 손실 함수에 미치는 영향을 측정하고, 민감한 레이어에 높은 비트 수를 할당하는 프레임워크를 제안하였다. 그러나 헤시안 기반 접근법은 민감도 계산에 추가 계산을 요구하며, 파라미터의 곡률 지표로 계층별 bit를 배분한다. 반면, 본 연구는 소형 객체 탐지라는 구체적 도메인에서 입력 데이터의 특성(작은 객체, 복잡한 배경)이 초기 레이어의 Activation Range를 확장시킨다는 현상적 원인에 집중한다. 이를 통해 복잡한 민감도 탐색 과정 없이 초기 레이어의 정밀도 보존이 성능 회복의 핵심임을 규명하고, 실용적인 설계 가이드를 제시한다.

3. 제안 방법

본 연구는 SOD 환경에서 INT8 양자화 성능 저하가 두드러지는 원인을 레이어별 activation range 관점에서 분석하고, 이를 완화하기 위한 선택적 혼합 정밀도(selective mixed precision) 추론 전략을 제안한다. 먼저 동일한 모델 조건에서 데이터 특성에 따른 activation 분포 차이를 확인하기 위해, 학습된

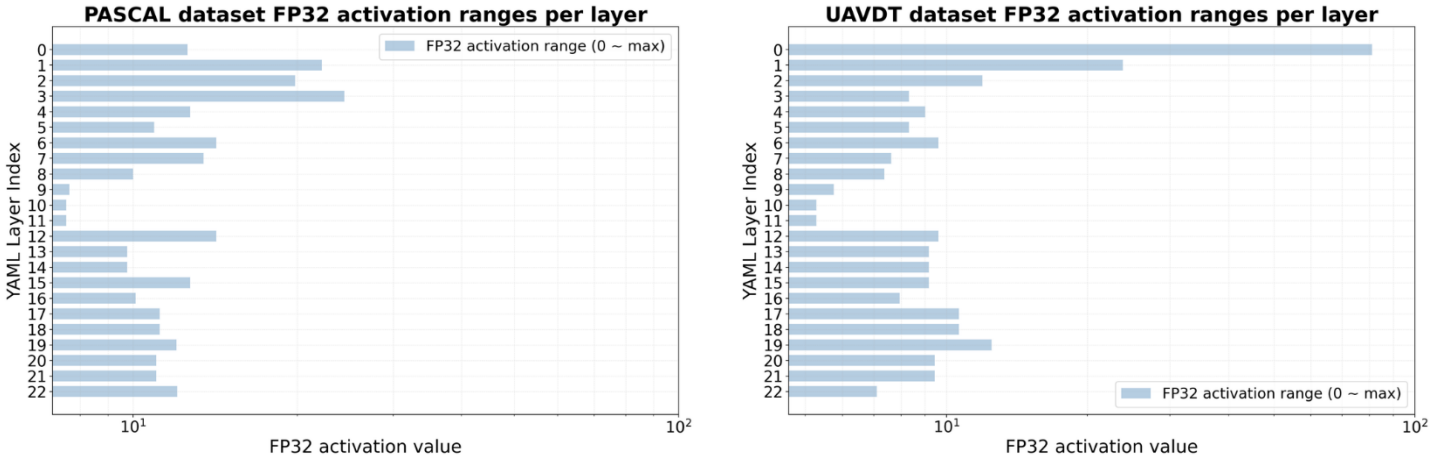


Figure 2 PASCAL VOC 2007(좌)-UAVDT(우)에서 FP32 레이어별 activation 범위 비교.

YOLOv11n 모델을 FP32 ONNX로 변환한 뒤 추론 과정에서 레이어별 activation 값을 출력하여 통계적으로 분석하였다. 분석에는 일반 객체 탐지에서 널리 사용되는 기준 데이터셋인 PASCAL VOC-2007 validation dataset[1]과 항공/원거리 촬영으로 소형 객체 비중이 높은 SOD 조건을 대표하는 UAVDT validation dataset[2]을 사용하였으며, 두 데이터셋을 비교한 결과 초기 레이어 구간에서 activation range가 서로 다르게 나타나는 현상을 확인하였다(Figure 2). 특히 UAVDT는 원거리 객체가 빈번해 작은 객체 비중이 상대적으로 높기 때문에, 초기 특징 추출 구간에서 activation 값의 동적 범위가 더 넓게 형성될 수 있음을 시사한다. SiLU 활성화의 하한(≈−0.278)때문에 레이어별 min 값이 거의 수렴하므로, Figure 2는 0-max 기준으로 activation 범위를 표시하였다.

이러한 관찰은 INT8 양자화에서 중요한 의미를 갖는다. INT8은 제한된 8-bit 표현 범위 내에 값을 매핑하므로, activation range가 넓어질수록 동일 비트폭에서 표현 간격이 커져 근사 오차가 증가할 수 있다. 초기 레이어는 저수준 특징을 추출하는 단계이므로, 이곳의 정보 손실은 후속 레이어 전체에 악영향을 미친다. 초기 고해상도 특징 추출 구간에서 발생한 양자화 오차가 소형 객체의 경계, 텍스처 정보를 먼저 훼손한 뒤 후속 레이어로 누적 전파되므로, 본 연구는 오차 전파 시작점인 초기 일부 레이어만 FP16으로 유지한다. 실수값 x의 INT8 선형 양자화를 $q = round\left(\frac{x}{s}\right) + z$, 역양자화를 $x' = (q - z)s$ 로 두고 오차를 $\delta x = x' - x$ 로 정의한다. 선형 연산 $y = \sum_i a_i w_i$에 대해 양자화된 출력 오차는

$$\Delta y = y' - y \approx \sum_i a_i \delta w_i + \sum_i w_i \delta a_i$$

로 근사할 수 있다. 즉 가중치/활성값의 양자화 오차가 입력 활성값 a 또는 가중치 w에 의해 증폭되어 출력 오차로 전파된다. 또한 활성값의 동적 범위가 커질수록 스케일 s가 커지고, $|\delta a|$가 증가할 수 있어 초기 레이어에서의 오차가 누적적으로 성능저하로 이어질 수 있다. 이에 따라 본 연구는 초기 특징 추출의 일부 컨볼루션 레이어만 FP16으로 유지하고 나머지는 INT8로 수행하는 선택적 혼합 정밀도 전략으로 초기 구간의 오차 전파를 완화한다.

4. 실 험

4.1 실험 환경

본 연구의 실험은 UAVDT 데이터셋으로 YOLOv11n을 학습한 뒤, valid best 모델을 기준으로 TensorRT 기반 추론 엔진을 구성하여 수행하였다. 비교 정밀도는 FP32, FP16, INT8이며 성능 평가는 IoU=0.5에서 계산한 클래스별 AP의 평균 (mAP50)을 기준으로 진행하였다. 여기서 기본 INT8 엔진은 INT8 양자화를 적용하되, TensorRT의 레이어 지원 및 정밀도 선택에 따라 일부 연산이 FP16/FP32로 수행되는 혼합 정밀도 형태로 구성되었다. 본 연구의 제안 설정은 이 기본 엔진을 기준으로, SOD에서 민감할 수 있는 초기 구간의 일부 컨볼루션 레이어 정밀도를 FP16으로 명시적으로 유지하도록 조정한 변형 엔진을 구성하여 비교하였다.

4.2 결과 및 논의

	FP32	FP16	INT8
mAP50(Pascal)	54.4%	54.4%	53.8%
mAP50(UAVDT)	27.1%	27.1%	23.9%

Table 1 Pascal VOC 2007 및 UAVDT 데이터셋에 대한 YOLOv11n 정밀도별 mAP50 성능 비교.

	mAP50	Mean Latency(ms)
INT8(base)	23.9%	7.039
INT8(ours)	26.4%	7.141

Table 2 초기 레이어 혼합 정밀도(Ours) 적용 시 기본 INT8 모델(Base)대비 mAP50 및 지연시간 비교.

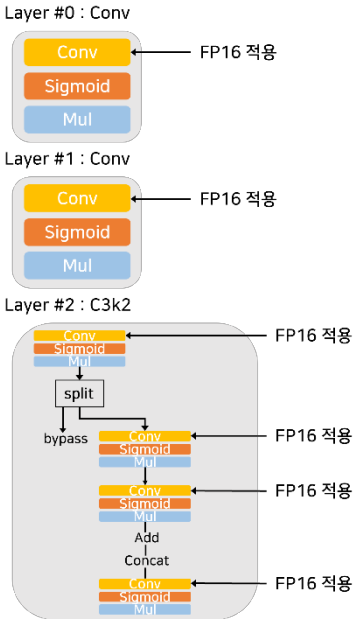


Figure 3 Layer #0-#2의 연산 구조와 합성곱 연산에 대한 FP16 정밀도 적용 위치

정밀도 설정에 따른 기본 성능 비교 결과를 표 1에서 볼 수 있다. FP32/FP16 대비 INT8에서 mAP50이 감소하는 경향이 관찰되었으며, 특히 small object 비중이 높은 SOD 조건에서는 정밀도 감소에 따른 성능 저하가 더 두드러질 수 있음을 확인하였다(Table 1). 이는 엣지 추론에서 INT8 가속이 유효하더라도, SOD에서는 정확도 손실이 무시하기 어려운 문제가 될 수 있음을 시사한다.

INT8 성능 저하를 완화하기 위해 초기 3개 컨볼루션 레이어만 FP16으로 유지하고 나머지는 INT8으로 수행하는 선택적 혼합 정밀도 엔진을 구성하였다.(Figure 3) 이때 해당 레이어들은 TensorRT의 레이어별 정밀도 설정을 통해 가중치와 활성화값 모두 FP16 정밀도로 연산되도록 강제하였으며, 그 결과 기준 엔진 대비 mAP50이 23.9%에서 26.4%로 향상되어 +2.5%p의 성능 개선을 보였다(Table 2). 한편, 추론 속도 측면에서는 초기 레이어의 고정밀도 연산으로 인해 평균 지연시간이 7.039ms에서 7.141ms로 약 1.4%(0.1ms) 소폭 증가하였다. 그러나 이는 실시간성을 해치지 않는 미미한 수준으로 제안하는 방식이 2.5%p의 정확도 향상을 위해 지불하는 비용으로서 충분히 효율적인 Trade-off임을 보여준다. 즉, 전체 정밀도를 상향하지 않고도 초기 구간의 제한적 정밀도 보존만으로 INT8 성능 저하를 유의미하게 완화할 수 있음을 확인하였다.

이러한 개선은 초기 레이어에서 관찰된 activation range 확대가 양자화 민감도를 높일 수 있다는 분석과 일관된 결과로 해석된다. 즉, activation range가 넓게 형성되는 초기 구간을 FP16으로 수행하여 해당 구간의 양자화 근사오차를 완화하고, 소형 객체 탐지에 중요한 미세 특징 정보의 손실을 줄였기 때문에 mAP50 개선에 기여했을 가능성이 있다. 결론적으로 SOD 조건에서 INT8 성능 저하가 실제로 관찰됨을 보았고, 초기 구간의 선택적 FP16 적용만으로도 그 저하를 부분적으로 회복할 수 있음을 볼 수 있다. 이는 혼합 정밀도 설계가 모델 전체를 상향하는 방식이 아니라, 오차가 발생하기 쉬운 구간을 좁혀 조정하는 최소 변경접근으로도 의미 있는 개선을 얻을 수 있음을 시사한다. 따라서 제안 방식은 엣지 배치 상황에서 정확도 개선과 효율 유지 사이의 균형을 맞추기 위한 실무적 설계 선택지로 해석될 수 있다.

5. 결 론

본 연구는 Jetson Orin Nano에서 YOLOv11n 기반 SOD 수행 시 INT8 양자화로 인한 성능 저하 문제를 다루고, 일부 레이어만 FP16으로 유지하는 선택적 혼합 정밀도를 적용해 정확도-지연시간 절충을 개선했다. 그 결과 INT8(base) 대비 mAP50을 +2.5%p 향상시키면서 지연시간 증가는 약 0.1ms로 제한되어, SOD 엣지 추론에서 정밀도 선택이 단순 가속 옵션이 아니라 성능에 직접 영향을 주는 설계요소임을 시사한다.

다만 본 결과는 단일 디바이스, 단일 모델, 제한된 데이터셋 조건에 기반하며, 초기 레이어 선택의 타당성은 확인했으나 레이어 수 및 적용구간(Backbone/Neck/Head)에 따른 민감도 특성은 보다 체계적인 분석이 필요하다. 향후에는 다양한 모델/엔진에서 재현성 검증 및 PTQ/QAT 결합을 통한 자동 정밀도 배치로 확장할 필요가 있다.

6. 참고문헌

[1] Everingham, Mark, et al. "The pascal visual object classes (voc) challenge." *International journal of computer vision* 88.2 (2010): 303-338.

[2] Du, Dawei, et al. "The unmanned aerial vehicle benchmark: Object detection and tracking." *Proceedings of the European conference on computer vision (ECCV)*. 2018.

[3] Xiao, Guangxuan, et al. "Smoothquant: Accurate and efficient post-training quantization for large language models." *International conference on machine learning*. PMLR, 2023.

[4] Dettmers, Tim, et al. "Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale." *Advances in neural information processing systems* 35 (2022): 30318-30332.

[5] Dong, Zhen, et al. "Hawq: Hessian aware quantization of neural networks with mixed-precision." *Proceedings of the IEEE/CVF international conference on computer vision*. 2019.

모델체킹을 위한 강화학습 기반 휴리스틱 학습*

강혜윤O, 손병호, 배경민
포항공과대학교 컴퓨터공학과
{hyoonk, byhoson, kmbae}@postech.ac.kr

RL-based Heuristic Learning for Model Checking

Hyeyoon KangO, Byoungho Son, Kyungmin Bae
Department of Computer Science and Engineering, POSTECH

요 약

모델체킹은 시스템의 성질을 자동으로 검증하는 기법이지만, 일반적으로 모델체킹에서 반복적으로 검증을 수행하는 경우에 기존의 검증 결과를 재활용하는 기술이 널리 연구되지 않았다. 본 연구는 강화학습을 활용하여 모델체킹의 휴리스틱을 자동으로 학습하는 프레임워크를 제안한다. 술어 요약을 통해 구체적 상태를 고정된 차원의 요약 상태로 변환하고, 이 요약 공간에서 테이블 기반 Q-learning 및 DQN을 적용하여 오류 상태 도달 가능성을 예측하는 휴리스틱을 학습한다. Dining Philosophers 벤치마크에 대한 실험 결과, Q-table 기반 휴리스틱은 임의 탐색 대비 최대 5.6배, DQN 기반 휴리스틱은 최대 48배의 탐색 효율 향상을 보였다.

1. 서론

모델체킹(model checking)은 시스템이 주어진 성질을 만족하는지 자동으로 검증하는 기법으로, 하드웨어 및 소프트웨어 시스템의 신뢰성 확보에 널리 활용되어 왔다 [1]. 예를 들어, Amazon Web Services는 분산 시스템의 핵심 알고리즘 검증에 모델체킹을 적용하여 설계 오류를 발견한 바 있다.

시스템 검증 과정에서 오류가 발견되면 이를 수정하고 다시 검증해야 하며, 시스템이 변경될 때마다 기존 오류의 재발 여부를 확인해야 한다. 이러한 반복적 검증 상황에서 매번 시스템의 모든 가능한 상태를 처음부터 탐색하는 것은 비효율적이지만, 기존 검증 결과를 재활용하는 기술은 널리 연구되지 않았다.

모델체킹의 탐색 효율을 높이면 이러한 반복 검증의 부담을 줄일 수 있다. 지향성 모델체킹(directed model checking)은 휴리스틱을 통해 오류 상태에 가까운 상태를 우선 탐색하여 효율을 높이는 기법이다 [2]. 기존 지향성 모델체킹 연구들은 대부분 도메인 지식에 기반한 휴리스틱을 직접 설계하거나, 특정 유형의 시스템에 특화되어 있다.

본 연구는 Q-learning 기반의 강화학습(reinforcement learning)을 활용하여 휴리스틱을 자동으로 학습하는 프레임워크를 제안한다. 강화학습에서는 목표 상태 도달 시 보상을 부여함으로써, 각 상태에서 목표까지 도달하는 경로의 가치를 학습할 수 있다. 오류 상태를 목표로 설정하면, 학습된 Q 함수는 각 상태가 오류 상태에 얼마나 가까운지를 나타내므로 휴리스틱으로 활용될 수 있다.

이러한 접근을 모델체킹의 휴리스틱 학습에 적용하는 데에는 두 가지 어려움이 있다. 첫째, 모델체킹에서는 상태 수가 방대하여 모든 상태의 가치를 Q-table 등에 개별적으로 저장하기 어렵다. 또한 학습에 사용한 시스템보다 큰 시스템을 검증할 때, 학습 중 방문하지 않은 상태에 대해서는 휴리스틱을 제공하지 못한다. 둘째, Q 함수를 Q-table로 명시적으로 저장하는 대신 DQN(Deep Q-Network)와 같은 신경망으로 근사하더라도, 시스템의 상태를 신경망 입력으로 표현하는 방식이 시스템마다 달라 일반적인 적용이 어렵다.

본 연구는 술어 요약(predicate abstraction)을 활용하여 이러한 문제를 해결한다. 술어 요약은 구체적 상태를 미리 정의된 술어들의 만족 여부를 나타내는 불리언 벡터로 변환하는 기법이다. 이를 통해 방대한 상태 공간을 고정된 차원의 요약 공간으로 변환하여, 시스템의 크기가 달라지더라도 동일한 차원의 표현을 사용할 수 있다.

본 연구는 술어 요약과 Q-learning을 결합한 휴리스틱 학습 프레임워크를 제안한다. 요약 상태 공간에서 Q-table을 학습하고, 이를 최선 우선 탐색(best-first search)의 휴리스틱으로 활용한다. 또한 DQN 기반 휴리스틱으로의 확장을 통해 학습 중 방문하지 않은 요약 상태에 대해서도 휴리스틱 값을 예측할 수 있게 하였다.

Dining Philosophers 벤치마크 실험 결과, Q-table 기반 휴리스틱은 임의 탐색 대비 최대 5.6배, DQN 기반 휴리스틱은 최대 48배의 탐색 효율 향상을 보였다.

2. 관련 연구

지향성 모델체킹은 휴리스틱을 활용하여 오류 상태에 가까운 상태를 우선 탐색함으로써 검증 효율을 높이는 기법이다 [2]. 구체적 상태 공간에 대한 지향성 모델체킹 연구로, [3]에서는 명시적 상태 모델체킹에 A* 알고리즘과 FSM 거리, Hamming 거리 등의 휴리스틱을 적용하여 통신 프로토콜 검증에서 효과를 보였다. 이러한 연구들은 도메인 지식에 기반하여 휴리스틱을 직접 설계해야 한다는 한계가 있다.

요약된 상태 공간에서의 지향성 모델체킹 연구도 존재한다.

*이 논문은 2024년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원(No.NRF-2021R1A5A1021944, No.RS-2024-00413202)과 정보통신기획평가원의 지원(No.RS-2024-00439856)을 받아 수행된 연구임

[4]에서는 BDD 기반 심볼릭 모델체킹에 지향성 탐색을 적용하였다. [5]에서는 거리 보존 요약(distance-preserving abstraction)을 제안하여 요약 공간에서 오류 상태까지의 거리를 휴리스틱으로 활용하였다. [6]에서는 술어 요약을 사용하여 휴리스틱 함수를 생성하는 방법을 제안하였다. 이러한 연구들도 휴리스틱을 직접 설계하거나 계산한다.

한편, TSP, Max-Cut, Bin Packing 등 NP-hard 조합 최적화 문제에 강화학습을 적용한 연구들이 활발히 진행되어 왔다 [7]. 이러한 연구들은 조합 최적화 문제에 초점을 맞추고 있으며, 본 연구에서와 같이 강화학습을 통해 모델체킹의 휴리스틱을 학습하는 연구는 거의 없었다.

3. 배경 지식

3.1. 모델체킹

모델체킹(model checking)은 시스템의 모든 도달 가능한 상태를 탐색하여 주어진 성질의 만족 여부를 검증하는 정형 기법이다 [1]. 모델체킹에서 검증 대상 시스템은 상태 전이 시스템(transition system) $\mathcal{S} = (S, \mathrm{Act}, \rightarrow)$ 로 표현되며, 이때 상태 집합 S, 행동 집합 Act, 전이 관계 $\rightarrow \subseteq S \times \mathrm{Act} \times S$이다. 전이 관계 $s \xrightarrow{a} s'$는 상태 s에서 행동 a를 통해 상태 s'로 전이할 수 있음을 의미한다. 검증하고자 하는 성질은 불변성(invariant), 도달 가능성(reachability) 등 다양하며, 본 연구에서는 오류 상태에 도달 가능한지에 대한 성질을 다룬다.

시스템의 구성 요소가 증가하면 상태 공간이 기하급수적으로 증가하는 상태 폭발 문제가 발생한다. 이를 완화하기 위해 부분 순서 축소(partial order reduction), 추상화, 심볼릭 모델체킹 등 다양한 기법이 연구되어 왔다.

3.2. 재작성 논리

재작성 논리(rewriting logic)는 동시성 시스템의 명세에 널리 활용되는 명세 기법이다 [8]. 시스템의 상태를 대수적 항(term)으로 표현하고, 상태 전이를 재작성 규칙으로 기술한다.

다음은 Dining Philosophers 문제에 대한 재작성 논리 명세의 예이다. N명의 철학자가 원형 테이블에 앉아 있고 인접한 철학자 사이에 포크가 있다. 각 철학자는 대수적 항 $p(i, st)$로 표현되며, i는 철학자 위치, st는 상태(think , hungry , single , eat)를 나타낸다. 포크 $c(i)$는 철학자 i의 왼쪽에 위치한다. 전체 상태는 $\|$를 구분자로 가지는 철학자들과 포크들의 집합으로 표현된다:

$$p(0, st_0) \,\|\, p(1, st_1) \,\|\, \cdots \,\|\, p(N-1, st_{N-1}) \\ \|\, c(0) \,\|\, c(1) \,\|\, \cdots \,\|\, c(N-1)$$

각 철학자는 초기에 think 의 상태를 가지며, 다음의 4개의 재작성 규칙으로 표현되는 행동을 할 수 있다:

$$\text{[th]: } p(i, \text{think}) \;\Rightarrow\; p(i, \text{hungry})$$
$$\text{[hs]: } p(i, \text{hungry}) \,\|\, c(j) \;\Rightarrow\; p(i, \text{single}) \text{ if } adj(i, j)$$
$$\text{[se]: } p(i, \text{single}) \,\|\, c(j) \;\Rightarrow\; p(i, \text{eat}) \text{ if } adj(i, j)$$
$$\text{[et]: } p(i, \text{eat}) \;\Rightarrow\; p(i, \text{think}) \,\|\, c\big(lc(i)\big) \,\|\, c\big(rc(i)\big)$$

이때, th, hs, se, et는 각 규칙의 레이블이며, $adj(i, j)$는 포크 i가 철학자 j와 인접함을, $lc(i)$와 $rc(i)$는 각각 철학자 i의 왼쪽과 오른쪽 포크를 나타낸다.

재작성 규칙은 왼쪽 패턴이 현재 상태의 임의의 부분에 매치되면 적용될 수 있다. 예를 들어, 상태 $p(0, \text{think}) \,\|\, p(1, \text{think}) \,\|\, p(2, \text{think}) \,\|\, c(0) \,\|\, c(1) \,\|\, c(2)$ 에는 th 규칙이 적용될 수 있는 세 가지 가능성이 있으며, 만약 철학자 0에 적용되면 다음 상태는 $p(0, \text{hungry}) \,\|\, p(1, \text{think}) \,\|\, p(2, \text{think}) \,\|\, c(0) \,\|\, c(1) \,\|\, c(2)$가 된다.

3.3. 강화학습

강화학습(reinforcement learning)은 에이전트가 환경과 상호작용하며 누적 보상을 최대화하는 행동 정책을 학습하는 기계학습 기법이다 [10]. 에이전트는 현재 상태에서 행동을 선택하고, 환경은 그에 따른 보상과 다음 상태를 반환한다.

Q-learning은 상태-행동 쌍 (s, a)의 가치를 나타내는 Q 함수를 학습하는 강화학습 알고리즘이다. Q 함수 $Q(s, a)$는 상태 s에서 행동 a를 취하고 이후 최적 정책을 따를 때 얻을 수 있는 기대 누적 보상을 나타낸다. 학습 과정에서는 학습된 Q 함수에 따라 최적 행동을 선택하는 활용(exploitation)과 새로운 상태-행동 쌍을 탐색하는 탐험(exploration) 사이의 균형이 필요하며, 이를 위해 ϵ-greedy 정책이 널리 사용된다.

테이블 기반(tabular) Q-learning 방식은 모든 상태-행동 쌍에 대한 Q 값을 테이블에 저장하여, 학습을 통해 Q 값을 갱신한다. 그러나 시스템의 크기에 따라 Q-table의 크기가 급격히 커지는 문제가 있다. DQN(Deep Q-Network)은 Q-table을 신경망으로 근사하여 테이블 기반 방식의 한계를 극복한다. DQN은 상태의 특징 벡터를 입력받아 각 행동에 대한 Q 값을 출력하며, 신경망의 일반화 능력을 통해 학습 중 방문하지 않은 상태에 대해서도 Q 값을 예측할 수 있다.

4. 제안 방법

본 연구는 강화학습 기반 휴리스틱 학습 프레임워크를 제안한다. 서론에 설명한 바와 같이, 먼저 사전에 정의된 술어를 통해 구체적 상태를 요약 상태로 변환하고, 이 요약 공간에서 Q-table 또는 DQN 기반 휴리스틱을 적용하여 오류 상태 도달 가능성을 예측하는 휴리스틱을 학습한다. 학습된 휴리스틱은 반복적인 검증 과정에서 오류 상태를 빠르게 탐색하는 데 활용된다.

이하에서는 술어 요약(4.1장), Q-table 기반 휴리스틱 학습(4.2장), DQN 기반 휴리스틱 학습(4.3장), 그리고 학습된 휴리스틱을 활용한 탐색(4.4장)에 대해 설명한다.

4.1. 술어 요약

술어 요약(predicate abstraction)은 구체적 상태를 술어들의 만족 여부로 요약하는 기법이다 [11]. 술어 집합 $\Pi = \{p_1, \ldots, p_n\}$이 주어졌을 때, $s \vDash p_i$는 상태 s가 술어 p_i를 만족하는지를 참 또는 거짓으로 나타낸다. 요약 함수 $\alpha\colon S \to \mathrm{Bool}^n$는 구체적 상태 s를 각 술어의 만족 여부로 구성된 n 차원 불리언 벡터로 변환한다:

$$\alpha(s) := \langle s \vDash p_1, s \vDash p_2, \ldots, s \vDash p_n \rangle$$

상태 전이 시스템 $\mathcal{S} = (S, \mathrm{Act}, \rightarrow)$에 술어 요약을 적용하면 요약 전이 시스템 $\mathcal{S}_{/\Pi} = \big(\hat{S}, \mathrm{Act}, \widehat{\rightarrow}\big)$를 얻는다. 여기서 $\hat{S} = \mathrm{Bool}^n$이고, 요약 전이 $\hat{s} \xrightarrow{a} \hat{s}'$는 $\alpha(s) = \hat{s}$, $\alpha(s') = \hat{s}'$, $s \xrightarrow{a} s'$를 만족하는 구체적 상태 $s, s' \in S$가 존재할 때 정의된다. 이를 통해 구체적 상태 공간의 크기와 무관하게 최대 2^n 개의 요약 상태를

갖는 시스템을 구성할 수 있다. 본 연구에서 학습하는 Q 함수는 요약 상태에서 정의되며, $\hat{Q}$: $\hat{S} \times \mathrm{Act} \to \mathbb{R}$로 표기한다.

재작성 논리 명세에서는 각 규칙의 왼쪽 패턴이 현재 상태와 매치되는지 여부로 술어를 정의할 수 있다. 본 연구에서는 3.2장에서 설명한 Dining Philosophers의 네 가지 규칙 각각에 대해 적용 가능 여부를 나타내는 술어를 정의하였다.

4.2. Q-table 기반 휴리스틱 학습

그림 1은 Q-table 기반 휴리스틱 학습 알고리즘을 보여준다. 일반적인 Q-learning 기반으로 요약 전이 시스템 $\mathcal{S}_{/\Pi}$을 활용하여 Q 함수 $\hat{Q}$를 학습하는 알고리즘이다. 오류 상태 도달 시 보상 1, 그 외 상태에서 보상 0을 부여한다. 따라서 높은 Q 값은 해당 상태-행동 쌍에서 오류 상태에 도달할 가능성이 높음을 의미한다.

에이전트는 학습 시스템에서 에피소드를 반복 수행하며, 각 전이를 경험할 때마다 Q 값을 갱신한다. 각 에피소드는 초기 상태에서 시작하여 오류 상태에 도달하거나 최대 스텝 수에 도달하면 종료된다. 학습 과정에서는 ϵ-greedy 정책을 사용하여 탐험과 활용의 균형을 유지한다. 이때 상태는 구체적 상태가 아닌 요약 상태 $\alpha(s)$ 가 사용된다. 학습이 완료되면 상태의 가치를 $\hat{V}(s) = \max_a \hat{Q}(\alpha(s), a)$로 정의하여 탐색 휴리스틱으로 활용한다.

Input: 학습 시스템 $\mathcal{S}_{/\Pi}$, 에피소드 수 E, 학습률 β, 할인율 γ
Output: Q-table $\hat{Q}$

Initialize $\hat{Q}(\hat{s}, a) \leftarrow 0$ for all $\hat{s} \in \hat{S}$, $a \in \mathrm{Act}$
for *episode* $= 1$ to E **do**
  $s \leftarrow$ initial state
  while s is not terminal **do**
    $a \leftarrow$ select action (ϵ-greedy based on $\hat{Q}$)
    $s \leftarrow$ execute action a from s
    $r \leftarrow 1$ if s' is error state, 0 otherwise
    $\hat{Q}(\alpha(s), a) \leftarrow \hat{Q}(\alpha(s), a) + \beta \cdot \left(r + \gamma \cdot \max_{a'} \hat{Q}(\alpha(s'), a') - \hat{Q}(\alpha(s), a) \right)$
    $s \leftarrow s'$
return $\hat{Q}$

그림 1. Q-table 기반 휴리스틱 학습 알고리즘

4.3. DQN 기반 휴리스틱 학습

Q-table 기반 방식은 학습 중 방문한 요약 상태에 대해서만 값을 저장한다. 술어 수가 증가하면 가능한 요약 상태 수가 기하급수적으로 증가하며, 학습에서 방문하지 않은 요약 상태에 대해서는 유의미한 휴리스틱을 제공하지 못한다.

이러한 한계를 해결하기 위해 DQN을 도입한다. DQN은 요약 상태 벡터 $\alpha(s)$를 입력받아 각 행동에 대한 Q 값을 출력하는 신경망이다. 네트워크는 n차원 불리언 벡터 입력층, 64개 뉴런과 ReLU 활성화 함수를 갖는 은닉층, $|\mathrm{Act}|$ 개 뉴런의 출력층으로 구성된다. 학습은 experience replay와 target network를 사용하는 표준 DQN 알고리즘을 따른다.

DQN은 신경망의 함수 근사 능력을 통해 학습 중 방문하지 않은 요약 상태에 대해서도 Q 값을 예측할 수 있다. 술어 요약이 고정된 n 차원 특징 공간을 제공하므로, 학습된 DQN은 유사한 요약 상태들에 대해 합리적인 가치를 예측할 수 있다.

4.4. 휴리스틱 탐색

그림 2에 보인 알고리즘과 같이, 학습된 $\hat{Q}$ 함수를 휴리스틱으로 활용하여 상태 전이 시스템 $\mathcal{S}$ 의 구체적 상태 공간에서 최선 우선 탐색을 수행한다. 각 상태 s를 방문할 때 술어 요약을 우선 적용한 뒤, $\hat{V}(s) = \max_a \hat{Q}(\alpha(s), a)$ 를 계산하여 가치가 높은 상태를 우선 탐색한다. $\hat{V}(s)$가 높을수록 해당 상태에서 오류 상태에 도달할 가능성이 높으므로, 탐색은 오류 상태를 향해 유도된다.

Input: 탐색 시스템 $\mathcal{S}$, 초기 상태 s_0, 학습된 $\hat{Q}$
Output: 오류 상태 또는 탐색 실패

queue $\leftarrow \{(s_0, \hat{V}(s_0))\}$
visited $\leftarrow \{s_0\}$
while *queue* is not empty **do**
  $s \leftarrow$ pop state with highest $\hat{V}$ from *queue*
  if s is an error state **then return** s
  for each (a, s') such that $s \xrightarrow{a} s'$ **do**
    visited $\leftarrow$ *visited* $\cup \{s'\}$
    insert $(s', \hat{V}(s'))$ into *queue*
return "No error state found"

그림 2. 휴리스틱 기반 최선 우선 탐색 알고리즘

5. 실험

본 논문에서 제안된 기법의 효과를 평가하기 위하여, 재작성 논리 기반 명세 분석 도구인 Maude [9]를 사용하여 제안 기법을 구현하고 실험을 수행하였다. 본 실험은 다음 두 가지 Research Question에 대해 답하고자 한다.

(i) 제안한 강화학습 기반 휴리스틱이 기존 탐색 방법 대비 얼마나 효과적으로 오류 상태를 탐색하는가?
(ii) 학습된 휴리스틱이 학습에 사용되지 않은 더 큰 인스턴스에서도 효과적으로 동작하는가?

5.1. 실험 설정

벤치마크로 3.2장에서 설명한 Dining Philosophers 문제를 사용한다. 이 문제는 N 이 증가함에 따라 상태 공간이 기하급수적으로 증가하므로 휴리스틱의 효과를 평가하기에 적합하다. 검증 목표는 모든 철학자가 한쪽 포크를 들고 대기하는 교착 상태 도달 여부이다.

술어는 4.1장에서 설명한 방법에 따라 네 가지 규칙 각각에 대해 철학자 0, 철학자 1, 나머지 철학자들의 적용 가능성을 나타내는 술어를 정의하고, 교착 여부를 나타내는 술어를 포함하여 총 13개의 술어를 사용하였다. 이를 통해 다양한 크기의 인스턴스가 동일한 13차원 불리언 벡터로 표현된다.

Q-table 기반 및 DQN 기반의 휴리스틱 학습을 $N = 3$ 인스턴스에서 500 에피소드 동안 수행하였다. 두 방법 모두 ϵ-greedy의 ϵ을 1에서 시작하여 0.05까지 지수적으로 감소시켰다. Q-table 학습에는 학습률 0.7, 할인율 0.95를 사용하였다. DQN 학습에는 학습률 0.02, 할인율 0.95를 사용하였으며, experience replay의 배치 크기는 32, target network 갱신 주기는 50 에피소드로 설정하였다.

학습된 휴리스틱의 일반화 성능을 평가하기 위해 $N = 4$ 부터 $N = 10$ 까지의 인스턴스에서 테스트하였다. 비교를 위한

기준선으로 너비 우선 탐색(BFS)과 임의 탐색(Random)을 사용하였다. Random은 각 상태에 무작위 우선순위를 부여하여 최선 우선 탐색을 수행하며, 10회 독립 실행의 평균을 사용하였다.

5.2. 실험 결과

표 1과 표 2는 각 방법의 탐색 결과를 비교한다. BFS와 Random은 기준선, Q-table과 DQN은 학습 기반 휴리스틱을 나타낸다. 표 1은 오류 상태 도달까지 탐색한 상태 수를 나타내며, Hit ratio는 Q-table에서 학습된 요약 상태가 테스트 시 등장한 비율이다. 표 2는 탐색에 소요된 시간을 나타낸다.

표 1. 오류 상태 도달까지 탐색한 상태 수

N	BFS	Random	Q-table (Hit ratio)	DQN
4	268	40.5	29 (38.3%)	8
5	1,230	147.1	41 (24.1%)	10
6	5,453	314.0	62 (15.6%)	32
7	23,692	1,336.7	273 (5.6%)	86
8	102,043	5,432.9	824 (2.4%)	203
9	435,878	15,925.9	3,632 (0.8%)	965
10	1,852,361	92,759.7	16,513 (0.2%)	1,937

표 2. 오류 상태 도달까지 소요된 시간(ms)

N	BFS	Random	Q-table	DQN
4	10.1	2.4	4.5	77.0
5	53.3	13.6	8.8	74.6
6	308.3	35.0	18.9	79.1
7	1,533.8	196.6	84.9	96.1
8	8,413.3	936.2	268.7	140.8
9	46,225.5	3,807.6	1,210.1	364.5
10	559,782.9	27,783.1	6,448.5	867.4

Q-table 기반 휴리스틱은 BFS 및 Random 대비 모든 인스턴스에서 탐색 상태 수를 크게 줄인다. $N = 10$에서 BFS가 약 185만 개, Random이 약 9.3만 개의 상태를 탐색하는 반면, Q-table은 약 1.7만 개로 각각 112배, 5.6배 감소하였다.

DQN 기반 휴리스틱은 Q-table 기반 대비 추가적인 효율 향상을 보인다. $N = 10$ 에서 DQN은 약 1,900개의 상태만 탐색하여 BFS 대비 956배, Random 대비 48배, Q-table 기반 대비 8.5배의 개선을 보였다. 특히 N이 1 증가함에 따라 BFS의 탐색 상태 수는 약 4.3배씩 증가하는 반면, DQN 기반 휴리스틱의 증가율은 평균 약 2.7배로 상대적으로 완만하다. 이는 술어 요약이 서로 다른 크기의 인스턴스를 동일한 특징 공간으로 표현하고, DQN의 일반화 능력이 학습되지 않은 인스턴스에도 유효한 휴리스틱을 제공하기 때문이다.

Q-table 기반 및 DQN 기반 휴리스틱의 성능 차이는 일반화 능력에서 기인한다. Q-table 기반 휴리스틱은 학습 중 방문한 요약 상태에 대해서만 유의미한 값을 가지는데, $N = 3$ 에서 학습한 Q-table의 상태가 테스트 인스턴스에서 등장하는 비율(hit ratio)은 N이 증가함에 따라 급감한다. $N = 4$에서 38.3%이던 hit ratio는 $N = 10$에서 0.2%까지 감소한다. 반면 DQN 기반 휴리스틱은 술어 요약이 제공하는 고정된 특징 공간에서 학습하므로, 학습에서 방문하지 않은 요약 상태에 대해서도 합리적인 휴리스틱 값을 예측할 수 있다.

탐색 시간 측면에서 DQN 기반 휴리스틱은 신경망 추론 비용으로 인해 작은 인스턴스에서는 Q-table 기반보다 느리다. 그러나 $N \geq 8$ 에서는 탐색 상태 수 감소가 추론 비용을 상쇄하여 DQN 기반이 더 빠르다. $N = 10$에서 BFS는 약 9.3분이 소요되는 반면, DQN 기반 휴리스틱은 1초 미만으로 완료된다.

6. 결론

본 연구는 강화학습을 활용하여 모델체킹의 휴리스틱을 자동으로 학습하는 프레임워크를 제안하였다. 술어 요약을 통해 구체적 상태를 고정된 차원의 요약 상태로 변환하고, 이 요약 공간에서 Q-learning 및 DQN을 적용하여 오류 상태 도달 가능성을 예측하는 휴리스틱을 학습하였다.

Dining Philosophers 벤치마크에 대한 실험 결과, Q-table 기반 휴리스틱은 BFS 및 Random 대비 각각 최대 112배와 5.6배, DQN 기반 휴리스틱은 각각 최대 956배와 48배의 탐색 효율 향상을 보였다. 또한 $N = 3$ 에서 학습한 휴리스틱이 $N = 10$ 까지의 인스턴스에서도 효과적으로 동작하여, 학습된 휴리스틱이 유사한 시스템에 일반화될 수 있음을 확인하였다.

향후 연구로는 다양한 벤치마크에 대한 적용을 통한 일반성 검증, 교착 외의 성질 검증으로의 확장, 그리고 술어 자동 생성 방법 등을 고려한다.

참고 문헌

[1] E. M. Clarke et al., Model Checking (2nd), MIT Press, 2018.
[2] S. Edelkamp et al., “Survey on directed model checking,” MoChArt, LNCS, Vol. 5348, pp. 65-89, 2009.
[3] S. Edelkamp, A. Lluch-Lafuente, S. Leue, “Directed explicit-state model checking in the validation of communication protocols,” STTT, Vol. 5, No. 2-3, pp. 247-267, 2004.
[4] F. Reffel, S. Edelkamp, “Error detection with directed symbolic model checking,” FM, LNCS, Vol. 1708, pp. 195-211, 1999.
[5] K. Dräger, B. Finkbeiner, A. Podelski, “Directed model checking with distance-preserving abstractions,” STTT, Vol. 11, No. 1, pp. 27-37, 2009.
[6] J. Hoffmann et al., “Using predicate abstraction to generate heuristic functions in UPPAAL,” MoChArt, LNCS, Vol. 4428, pp. 51-66, 2007.
[7] N. Mazyavkina et al., “Reinforcement learning for combinatorial optimization: A survey,” Comput. Oper. Res., Vol. 134, art. no. 105400, 2021.
[8] J. Meseguer, “Conditional rewriting logic as a unified model of concurrency,” TCS, Vol. 96, No. 1, pp. 73-155, 1992.
[9] M. Clavel et al., All About Maude, LNCS, Vol. 4350, Springer, 2007.
[10] R. S. Sutton, A. G. Barto, Reinforcement Learning: An Introduction (2nd), MIT Press, 2018.
[11] S. Graf, H. Saïdi, “Construction of abstract state graphs with PVS,” CAV, LNCS, Vol. 1254, pp. 72-83, 1997.

이슈 설명과 심볼 수준의 목표를 활용한 분류 기반 결함 위치 식별

아슬란 압디나비예프[0], 홍수지[1], 이병정[0]
서울시립대학교 컴퓨터과학과[0], 서울시립대학교 인공지능학과[1]
{aslan, sujiggum, bjlee}@uos.ac.kr

Classification-Based Fault Localization Using Issue Descriptions and Symbol-Level Targeting

Abdinabiev Aslan Safarovich[0], Suji Hong[1], Byungjeong Lee[0]
Dept. of Computer Science, University of Seoul[0], Dept. of Artificial Intelligence, University of Seoul[1]

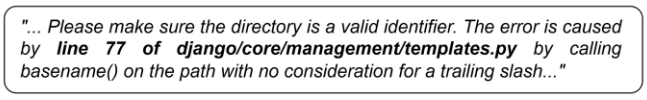
Abstract

Automated program repair methods require accurate fault localization to identify buggy code regions from natural-language issue descriptions. Existing approaches, including agent-based navigation and procedural methods, apply fixed strategies regardless of the specificity and structure of the input, leading to inconsistent performance across diverse bug reports. We present a classification-based fault localization architecture that classifies issue descriptions into three categories based on available location cues, Full Location, Partial Location, and Hint, and selects appropriate localization strategies accordingly. Our study targets symbol-level abstractions (functions, class-level elements, and file-level symbols) rather than individual lines, providing stable repair contexts suitable for complex, multi-line bugs. Experimental evaluation on benchmark dataset demonstrates that classification-based strategy selection significantly improves localization accuracy compared to fixed-strategy baselines, particularly for issues with partial or missing location information.

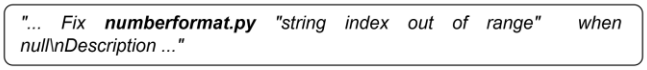
Keywords: Fault Localization, Issue Classification, Symbol-Level Target, Large Language Models

1. Introduction

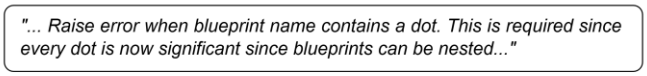
Fault localization accuracy fundamentally determines automated program repair (APR) effectiveness [1-4]. The challenge lies in transforming diverse natural-language issue descriptions into precise buggy code locations. Existing approaches fall into two categories: agent-based and procedural methods. Agent-based [5] methods use LLM-driven agents to navigate codebases through predefined action sequences such as entity extraction, file or symbol search, and refinement. They perform well when issue descriptions explicitly mention structural elements but struggle when such cues are missing or ambiguous, as agents lack a reliable starting point and cannot adapt their navigation strategy. Procedural approaches [6] employ fixed, non-interactive pipelines that rank candidate files using lexical, structural, and LLM-assisted relevance signals, followed by function or symbol selection within top-ranked files. In procedural approaches, a wrong file chosen at an early stage can send the entire localization process in the wrong direction, with no way to correct it later. A key limitation shared by both categories is their reliance on fixed localization strategies that do not adapt to the structural variability of issue descriptions. Figure 1 highlights this variability using real-world examples. Some issue descriptions provide explicit file paths and line references (Fig. 1(a)), others mention only partial structural information such as a file name (Fig. 1(b)), while many rely solely on semantic hints without any concrete code references (Fig. 1(c)). These differences imply fundamentally different localization requirements, yet existing approaches treat them uniformly. We address this limitation with a classification-based fault localization architecture that dynamically selects localization strategies based on issue classification. First, using LLM-driven analysis, our method extracts location cues (e.g., files, classes, functions, stack traces, keywords) from issue description and resolves them against the project structure. The extracted data and



(a) part of ID of django__django-14382



(b) part of ID of django__django-16046



(c) part of ID of pallets__flask-4045

Figure 1. Examples of issue descriptions (ID), where (a) full information, (b) partial information, and (c) hint about buggy locations are mentioned.

issue description are used to classify the issue into one of the three categories: Full Location, Partial Location, and Hint. Each category activates a tailored localization path with appropriate computational effort and inference depth. Our method targets symbol-level code units, functions, class-level elements, and file-level module symbols, rather than individual lines. By focusing on logical code units rather than arbitrary line boundaries, we enable more effective downstream repair. Our contributions are:

- A classification-based localization framework that selects strategies based on issue characteristics.
- Symbol-level targeting that yields stable, context-rich repair locations.
- Experimental validation demonstrating improved localization accuracy and reduced cost compared to fixed-strategy baselines.

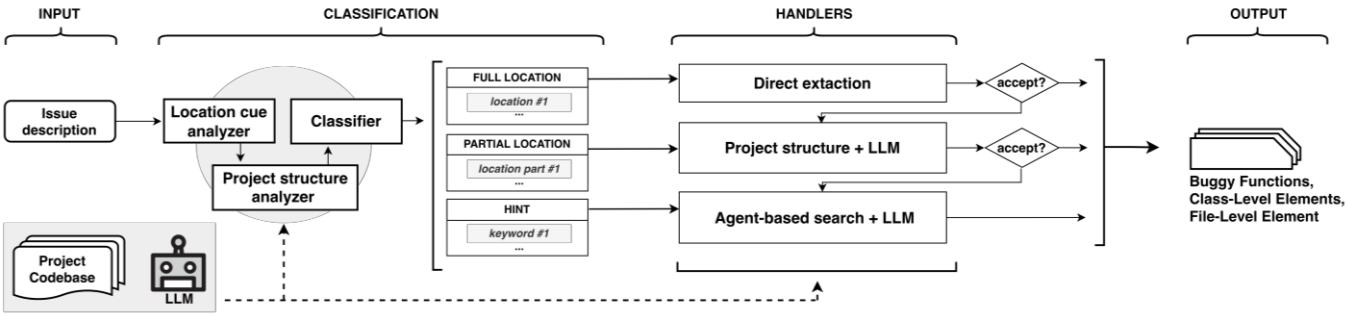


Figure 2. Fault localization architecture

2. Related Work

AutoCodeRover [5] and SWE-agent [8] employ LLM agents that navigate codebases through structured actions (file browsing, function inspection, cross-reference following). These methods excel when issues explicitly mention code entities, enabling agents to follow predetermined navigation sequences. However, when descriptions lack structural cues, agents cannot adapt their search strategy, leading to early termination or misrouting. The iterative exploration also incurs high computational costs. Procedural and retrieval-based approaches, such as Agentless [6], apply deterministic pipelines that perform repository-wide file and function retrieval using lexical, structural, or embedding-based similarity signals. Because these methods do not classify issue descriptions or exploit explicitly provided locations, they apply the same full-project search regardless of input specificity. As a result, even when a bug location is clearly stated, large projects incur unnecessary search over many files and functions, leading to wasted computation and token usage. Graph-based method RepoGraph [7] exhibits a similar limitation: call-graph or data-flow construction and traversal are performed uniformly, even when precise structural cues are available. Our work differs by introducing classification-based localization that dynamically selects strategies based on issue characteristics.

3. Methodology

3.1 Symbol-Level Representation

We represent buggy locations at symbol granularity rather than line granularity. A buggy location is one of three types: (1) Buggy Function-Level: The defect lies within a function or method body. The method identifies the complete function including its signature and full implementation. (2) Buggy Class-Level Element: The defect resides inside a class but outside any function, such as field declarations, class attributes, property definitions, or class-level decorators. (3) Buggy File-Level Symbol: The defect exists outside all classes and functions, such as module-level constants, global variables, import statements, or file-level executable code.

3.2 Issue Classification

Our classification pipeline consists of three components that analyze issue descriptions to determine available location information (Fig. 2).

Location Cue Analyzer (LLM-Based): This component extracts structural and semantic cues from issue text using a specifically designed prompt. It identifies explicit references such as file paths, class names, function names, and qualified identifiers. It also extracts implicit cues including stack trace fragments, error message patterns, and keywords indicating code regions.

Project Structure Analyzer: This component resolves extracted cues against the actual codebase structure. It maintains a symbol table mapping all functions, classes, and file-level symbols to their file locations and hierarchical positions within the project. For each extracted cue, the analyzer validates file paths against the repository structure, matches symbol names to their definitions while resolving ambiguity when multiple entities share names, disambiguates partial references using contextual keywords, and assesses whether the combination of resolved cues uniquely identifies a code symbol.

Classifier (LLM-Based): Based on the resolution results and issue description, the classifier assigns the issue to one of three categories:

(1) Full Location: The issue provides a complete, unambiguous location reference that resolves to a unique symbol, as illustrated in Fig. 1(a). The method directly extracts the referenced symbol without inference.

(2) Partial Location: The issue contains structural information that partially constrains the search space but is insufficient for unique identification, as illustrated in Fig. 1(b). The method performs constrained inference within the referenced scope.

(3) Hint: The issue contains only indirect signals without direct structural references, as illustrated in Fig. 1(c). The method performs semantic retrieval to identify relevant code regions.

This classification enables adaptive strategy selection: computational effort scales with the amount of inference required, from minimal computation for Full Location to comprehensive exploration for Hint.

3.3 Handlers

Each classification category activates a tailored localization strategy designed for its information availability level.

1) Full Location Handler. When complete location information is available, the handler performs direct extraction. It retrieves the referenced symbol (function, class-level element, or file-level symbol) from the resolved file location and extracts its complete code using AST parsing.

Acceptance: The LLM analyzes the suspected location's code body to determine whether it is truly the buggy location. The validation checks: (1) whether the code's functionality aligns with the described buggy behavior, (2) whether the code handles the inputs, outputs, or conditions mentioned in the issue, and (3) whether modifying this location could reasonably fix the reported problem. If the suspected location is rejected, the issue is escalated to the Partial Location Handler for broader search.

2) Partial Location Handler. When the issue description provides partial structural information, such as a file name, class name, or unqualified function name, the handler performs hierarchical localization inspired by the Agentless approach [6], but with a structure-aware and LLM-driven selection process. The localization process proceeds in three stages. (1) File-path level localization. Given the project's file structure, the LLM directly

Table 1. Fault Localization Accuracy on SWE-bench Lite

Approach \ Level	LLM	File-path (%)	Symbol (%)
Agentless	GPT-4o	69.7 (209/300)	50.3 (151/300)
AutoCodeRover	GPT-4o	69.0 (207/300)	49.7 (149/300)
SWE-agent	GPT-4o	58.0 (174/300)	42.3 (127/300)
Ours	GPT-4o	**74.6 (224/300)**	**52.3 (157/300)**

selects the top-n candidate files by jointly analyzing the issue description and the repository structure. This selection is performed without explicit cue extraction or embedding-based retrieval; instead, the LLM reasons over the semantic alignment between the problem description and file-level responsibilities inferred from file names, paths, and structural organization. (2) Symbol-level localization. Within the selected files, the handler constructs a skeleton representation consisting of class and function signatures, or global variables/import statements, etc. The LLM analyzes this structural skeleton together with the issue description to identify candidate symbols that are most likely related to the reported behavior. (3) Fine-grained localization. The handler extracts the full implementations of the identified symbols and applies LLM-based reasoning to determine which symbols most likely contain the defect, based on control flow, data usage, and consistency with the described failure mode.

Acceptance: The LLM analyzes each candidate location's code body to determine whether it is truly the buggy location. The validation considers whether the code's implementation could cause the reported behavior. If no candidates are confirmed as buggy locations, the issue is escalated to the Hint Handler for exploratory search.

3) Hint Handler: When the issue description provides only indirect signals (e.g., symptoms, error messages, or keywords) and lacks explicit structural references, the method activates the Hint handler, which employs agent-based search similar to AutoCodeRover [5]. The LLM agent iteratively navigates the codebase using a set of search APIs: *search_class(name)* to locate class definitions, *search_method_in_class (method, class*) to find specific methods, and *search_code(snippet)* to locate code patterns. At each iteration, the agent analyzes the current context and decides which search APIs to invoke based on the issue description and previously collected information. This iterative exploration continues until the agent determines that sufficient context has been gathered or a maximum iteration limit is reached. The agent then identifies candidate buggy locations from the collected context by reasoning about which symbols' implementations most likely explain the described behavior.

Acceptance: The LLM analyzes each candidate location's code body to determine whether it is truly the buggy location by verifying semantic relevance to the described symptoms or error behavior. Candidates that are not confirmed are still included in the output with lower confidence scores, allowing downstream repair processes to attempt fixes on best-effort locations.

3.4 Output Representation and Multi-Location Support

Across all localization paths, our method outputs a ranked list of buggy location candidates. Each candidate is represented by a tuple consisting of (i) symbol type (function, class-level element, or file-level symbol), (ii) file path (absolute or repository-relative), and (iii) complete symbol code extracted via AST parsing. This representation naturally supports multi-location bugs. When a defect spans multiple files or functions (common in real-world bugs), multiple symbols are identified. The downstream repair module receives complete code for each candidate location, enabling coordinated patches across components.

Table 2. Average Fault Localization Cost per Issue

Approach \ Level	Avg. File-path ($)	Avg. Symbol ($)	Avg. Total FL ($)
Agentless	0.06	0.02	0.08
Ours	**0.02**	**0.03**	**0.05**

4. Experimental Evaluation

4.1 Experimental Setup

Dataset: SWE-bench Lite [10] benchmark containing 300 real-world GitHub issues from popular Python repositories. Each issue includes natural-language description, buggy project codebase, and a repair patch.

Baselines: AutoCodeRover (agent-based navigation) [5], Agentless (procedural retrieval) [6], and SWE-agent [8] (LLM agents to navigate repositories via shell commands).

Implementation: We use gpt-4o-2024-05-13 for fair comparison. We use the same model for classification, cue extraction, and acceptance validation with distinct prompts for each stage (temperature=0.2), Python AST library for symbol extraction. For the Partial Location Handler (hierarchical search), n=3 for file-path-level and no limit for symbol-level retrieval. For the Hint Handler (agent-based search), maximum iteration=5. The acceptance validation uses a dedicated prompt that determines whether the suspected location is truly the buggy location based on its code body.

Metrics: We consider a prediction correct if the set of predicted locations includes locations that were modified in the developer patch. This definition follows the location accuracy criterion used in the Agentless, where a buggy location was considered correct when its final patch modified all locations modified in the developer patch. The experimental results and prompts are available at:

https://github.com/soft7197/cbfl.git.

4.2 Results

RQ1: Localization Accuracy. RQ1 evaluates whether the proposed classification-based fault localization approach improves localization accuracy at both file and symbol level. Table 1 reports localization accuracy on the SWE-bench Lite benchmark. Our approach achieves 74.6% file-path-level accuracy and 52.3% symbol-level accuracy, which are higher than Agentless (69.7% / 50.3%), AutoCodeRover (69.0% / 49.7%), and SWE-agent (58.0% / 42.3%) under the same GPT-4o configuration. Symbol-level correctness is conditioned on correct file-path localization. A predicted symbol is considered correct only when it is identified in the correctly localized file, symbol-name matches occurring in incorrectly localized files are not counted as correct. The improvement is observed consistently at both granularity levels, indicating that the proposed framework more accurately identifies relevant files and symbols than existing agent-based and procedural baselines. These results confirm that adapting localization behavior through issue classification leads to measurable gains in fault localization accuracy.

RQ2: Efficiency. RQ2 examines the efficiency of the proposed approach in terms of average fault localization cost per issue. As shown in Table 2, our method incurs an average total localization cost of $0.05 per issue, compared to $0.08 for Agentless. At the file-path-level, our approach requires $0.02, substantially lower than Agentless's $0.06, while symbol-level localization costs $0.03, slightly higher than Agentless's $0.02. Despite this modest increase at the symbol level, the overall cost remains lower because file-path-level localization is more efficient. These results indicate that the proposed approach reduces total fault localization cost while simultaneously achieving higher localization accuracy.

Table 3. Accuracy by Issue Classification Category (TP: True Positive, GT: Ground Truth).

Category / Level	Classification		Localization Acc.						
			Ours			Agentless		AutoCodeRover	
	Count	Recall (%) (TP/GT)	File path Accuracy (%)	Symbol Accuracy (%)	Avg. Cost ($)	File path Accuracy (%)	Symbol Accuracy (%)	File path Accuracy (%)	Symbol Accuracy (%)
Full Location	10	21 (10/48)	**100** (10/10)	**100** (10/10)	0.01	**100** (10/10)	**100** (10/10)	**100** (10/10)	**100** (10/10)
Partial Location	177	63 (81/128)	84.7 (150/177)	54.8 (97/177)	0.04	**85.9** (152/177)	**55.9** (99/177)	74 (131/177)	49.2 (87/177)
Hint	113	52 (65/124)	56.6 (64/113)	44.2 (50/113)	0.05	41.6 (47/113)	37.2 (42/113)	**58.4** (66/113)	**46** (52/113)
Total	300	-	**74.6 (224/300)**	**52.3 (157/300)**	0.05	69.7 (209/300)	50.3 (151/300)	69.0 (207/300)	49.7 (149/300)

RQ3: Category-based Classification and Localization Accuracy. RQ3 evaluates classification and localization accuracy across issue categories. Table 3 presents classification recall and localization accuracy for our method, Agentless, and AutoCodeRover. For classification ground-truth, we checked whether the buggy symbol from the developer patch appears in the issue description. Classification recall is 21% (Full), 63% (Partial), and 52% (Hint). The relatively low recall occurs because issue descriptions often contain false-positive code references, and buggy symbols are rarely explicitly identified as faulty. For localization, all methods achieve 100% on Full Location issues. On Partial issues, Agentless achieves slightly higher accuracy (85.9%/55.9% file-path/symbol) than ours (84.7%/54.8%) due to embedding-based retrieval, however, we avoided this additional cost. On Hint issues, our method (56.6%/44.2%) outperforms Agentless (41.6%/37.2%). Conversely, AutoCodeRover achieves slightly higher Hint accuracy (58.4%/46.0%) than ours, but ours outperforms AutoCodeRover on Partial (84.7%/54.8% vs 74.0%/49.2%). This reveals that Agentless excels with structural cues but struggles without them, while AutoCodeRover shows the opposite pattern. Our method achieves balanced performance through classification-based routing, resulting in the highest overall accuracy (74.6%/52.3% vs 69.7%/50.3% vs 69.0%/49.7%).

5. Discussion

The experimental results demonstrate that no single fixed strategy achieves optimal performance across all issue categories. The key advantage of our method is not that its handlers outperform baselines in every category, but that classification-based routing avoids each baseline's weak category. By not using embedding-based retrieval, we trade a small accuracy reduction on Partial issues for lower computational cost. This design also enables cost-efficient resource allocation: Full Location issues require minimal computation through direct extraction, Partial Location issues incur moderate cost for constrained search, and Hint issues require higher cost for exploratory search. On average, the number of predicted symbols per issue was 4. Table 4 compares fault localization approaches across navigation style, strategy selection, and cost allocation. Agentless and AutoCodeRover rely on hierarchical or agent-driven navigation with fixed strategy selection, whereas our approach incorporates issue-level classification to guide navigation behavior and distribute localization cost accordingly.

Threats to validity are as follows. *Dataset scope:* Evaluation is limited to Python projects in SWE-bench Lite; cross-language generalization requires further validation. *Classification accuracy*: Misclassification may cause suboptimal strategy selection, though the fallback mechanism mitigates this by escalating failed localizations to broader search strategies. *Limited efficiency comparison*: AutoCodeRover and SWE-agent do not report FL-specific costs, limiting direct efficiency comparison to Agentless. *LLM dependency:* Performance depends on the capabilities of GPT-4o and may vary across different models or API versions.

Table 4. Feature Comparison of Fault Localization Approaches

Feature	Agentless	AutoCodeRover	Ours
Navigation	Hierarchical	Agent-driven	Classification-based
Strategy selection	Fixed	Fixed	Adaptive
Input analysis	None	Entity extraction	Classification
Cost allocation	Uniform	Variable	Proportional

6. Conclusion

This paper presented a classification-based fault localization framework that adapts localization strategies to the amount of location information available in issue descriptions. By moving away from fixed pipelines and targeting symbol-level code units, the approach aligns localization behavior with issue characteristics and provides stable, context-rich repair locations. The findings suggest that explicitly modeling issue structure enables more effective and efficient localization without increasing system complexity. Future work includes refining the classification scheme to capture finer-grained issue types and integrating classification-based localization more tightly with multi-location automated program repair methods.

Acknowledgment

This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NO. RS-2022-NR068754).

References

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, 2016.

[2] C. Le Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[3] Z. Li, D. Lu, S. Fang, Y. Liu, and X. Tan, “Large language models for software engineering: A systematic literature review,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 1–79, 2024.

[4] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, “Tare: Type-aware neural program repair,” in *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, pp. 1443–1455, 2023.

[5] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, “AutoCodeRover: Autonomous program improvement,” *arXiv preprint arXiv:2404.05427*, 2024.

[6] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, “Agentless: Demystifying LLM-based software engineering agents,” *arXiv preprint arXiv:2407.01489*, 2024.

[7] S. Ouyang, W. Yu, K. Ma, Z. Xiao, Z. Zhang, M. Jia, J. Han, H. Zhang, and D. Yu, “RepoGraph: Enhancing AI software engineering with repository-level code graph,” *arXiv preprint arXiv:2410.14684*, 2024.

[8] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, “SWE-agent: agent-computer interfaces enable automated software engineering,” in *Proc. of International Conference on Neural Information Processing Systems (NIPS '24)*, Vol. 37, pp. 50528–50652, 2024.

[9] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, “SWE-bench: Can language models resolve real-world GitHub issues?” in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2024.

[10] SWE-bench Lite. 2024. SWE-bench Lite. https://www.swebench.com/lite.html.

딥러닝 라이브러리 계산 오류 탐지의 정확도 개선을 위한 차등 테스팅 파이프라인

Usmonali Pakhlavonov, 김세훈O
울산과학기술원 컴퓨터공학과
usmon@unist.ac.kr, sehoon@unist.ac.kr

A Differential Testing Pipeline for Improving the Accuracy of Computation Bug Detection in Deep Learning Libraries

Usmonali Pakhlavonov, Sehoon KimO
Department of Computer Science and Engineering, Ulsan National Institute of Science and Technology

요 약

딥러닝 라이브러리는 현대 인공지능 응용의 핵심 소프트웨어 인프라로서, 내부 계산의 정확성은 모델의 성능과 시스템 신뢰성에 직접적인 영향을 미친다. 딥러닝 라이브러리의 계산 오류를 자동으로 탐지하기 위해 차등 테스팅이 널리 활용되고 있으나, 기존 차등 테스팅 기법은 부동소수점 수치 연산의 특성으로 인해 허위 양성이 빈번하게 발생하여 실질적인 활용에 한계를 가진다. 본 논문에서는 이러한 한계를 극복하기 위해, 딥러닝 라이브러리의 계산 오류 탐지를 위한 차등 테스팅 기반 퍼징 파이프라인을 제안한다. 제안 기법은 동일한 의미를 갖는 API 실행 결과를 비교하여 계산 오류를 식별하되, 비결정적 API를 사전에 제거하고, 수치적으로 불안정한 입력 텐서를 실행 중에 필터링하며, 데이터 타입을 고려한 결과 비교 기법을 통합함으로써 허위 양성을 체계적으로 감소시킨다. PyTorch 라이브러리의 781개 API를 대상으로 수행한 실험 결과, 제안 기법은 기존 차등 테스팅 기법 대비 다수의 허위 양성을 제거하면서도 실제 계산 오류를 효과적으로 탐지함을 확인하였다.

1. 서 론

PyTorch와 TensorFlow와 같은 딥러닝 라이브러리(deep learning library)는 텐서 연산, 자동 미분, 신경망 구성 요소 등을 API(application programming interface)로 추상화하여, 복잡한 딥러닝 모델의 구현과 학습을 지원하는 핵심 소프트웨어 인프라이다 [1,2]. 이러한 라이브러리는 다양한 인공지능 응용에서 모델의 학습 및 추론 과정 전반을 담당하므로, 내부 연산의 정확성은 곧 모델의 성능과 직결된다. 특히 안전 중요 응용(safety-critical application) 분야에서는 딥러닝 연산 결과의 정확성과 일관성이 시스템 신뢰성을 좌우한다.

API의 본래 의미와 다른 값을 계산하는 딥러닝 라이브러리의 계산 오류(computation bug)는 모델의 예측 정확도를 저하시켜 시스템 전반의 신뢰성을 떨어뜨린다. 이러한 계산 오류는 명시적인 크래시(Crash)를 동반하지 않는 경우가 많아, 기존의 크래시 중심 테스트 기법으로는 효과적으로 탐지하기 어렵다. 그 결과, 오류는 장기간 잠복한 채 실제 응용 환경에서 잘못된 예측이나 의사결정을 유발할 위험이 있다. 따라서 딥러닝 라이브러리에서 발생하는 계산 오류를 자동으로 식별하고 검증할 수 있는 효과적인 테스트 기법이 필요하다.

본 연구에서는 딥러닝 라이브러리에서 발생하는 계산 오류를 자동으로 탐지하기 위한 테스트 기법으로서, 차등 테스팅(differential testing)에 기반한 퍼징(fuzzing) 파이프라인을 제안한다. 제안 기법은 동일한 의미를 갖는 API 실행 결과를 상호 비교함으로써, 크래시 없이 발생하는 계산 오류를 효과적으로 식별하는 데 목적이 있다. 특히 비결정적 동작을 보이는 API를 사전에 제거하고, 수치적으로 불안정한 입력을 실행 중에 필터링하며, 데이터 타입을 고려한 결과 비교를 수행함으로써 기존 테스트 기법에서 빈번하게 발생하던 거짓 양성(false positive)을 체계적으로 감소시킨다. 이를 통해 본 연구는 딥러닝 라이브러리 내부 계산에 대한 신뢰성 있는 자동 검증 방법을 제시한다.

2. 연구 배경

다양한 딥러닝 라이브러리 API의 계산 오류를 자동으로 탐지하기 위한 전형적인 방법으로 차등 테스팅(differential testing)이 널리 활용된다 [3,4,5]. 차등 테스팅은 동일한 입력에 대해 CPU와 GPU에서 각각 계산을 수행하고, 그 결과를 비교함으로써 잠재적인 계산 오류를 식별하는 방식이다. 동일한 입력에 대해 두 환경에서의 계산 결과가 유의미하게 상이한 경우, 이는 어느 한쪽 구현에 오류가 존재할 가능성이 있음을 시사하며, 해당 사례를 잠정적인 계산 오류로 분류할 수 있다. 이러한 접근법은 별도의 테스트 오라클(oracle)을 구축할 필요가 없다는 장점이 있어, 수많은 API로 구성된 대규모 딥러닝 라이브러리의 계산 오류 탐지에 적합하다.

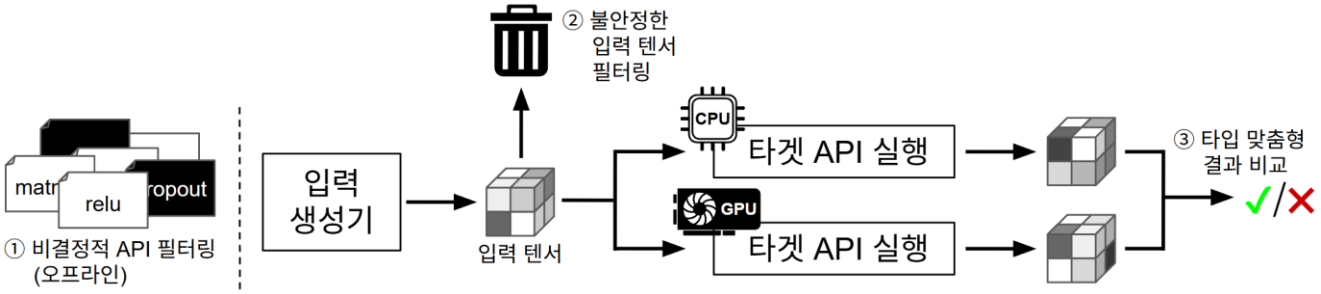


그림 1 차등 테스팅 파이프라인 개요

그러나 기존 차등 테스팅 기법은 높은 허위 양성(false positive) 문제로 인해 실용성에 한계를 가진다. 부동소수점(floating-point) 수치 연산에 내재된 근본적인 부정확성으로 인해, 동일한 알고리즘을 구현한 경우에도 연산 순서나 내부 최적화 방식의 차이에 따라 계산 결과가 달라질 수 있다 [6]. 이로 인해 구현상의 오류가 존재하지 않더라도 CPU와 GPU 간 계산 결과 차이가 빈번하게 발생한다. 이러한 잠정적 계산 오류의 진위를 판단하기 위해서는 전문가의 수작업 분석이 요구되며, 다수의 허위 양성은 분석 비용을 크게 증가시켜 계산 오류 탐지 기법의 실질적인 활용을 저해한다.

3. 제안 기법

본 연구에서는 딥러닝 라이브러리 API 계산 오류 탐지를 위해, 거짓 양성을 효과적으로 감소시키는 차등 테스팅 파이프라인을 제안한다 (그림 1). 제안 파이프라인은 비결정적 API를 필터링하는 오프라인 과정, 그리고 불안정한 입력 텐서 필터링 및 데이터 타입 맞춤형 결과 비교를 포함하는 퍼징 과정으로 구성된다.

3.1 비결정적 API 필터링 (Non-deterministic API Filtering)

퍼징 수행에 앞서 알고리즘 자체가 비결정적인(non-deterministic) API를 탐지하여 차등 테스팅 대상에서 제외한다 (그림 1, ①). 차등 테스팅의 기본 전제는 동일한 입력에 대해 API가 항상 동일한 출력을 생성해야 한다는 점이다. 그러나 드롭아웃(dropout), 무작위 샘플링(random sampling)과 같은 API는 본질적으로 비결정적이므로, 차등 테스팅을 적용할 경우 대부분의 결과가 허위 양성으로 이어진다.

이를 방지하기 위해, 본 연구에서는 API의 결정성을 자동으로 판별하는 간단한 휴리스틱을 적용한다. 구체적으로, 동일한 입력에 대해 API를 CPU 환경에서 5회 반복 실행하고, 실행 결과가 한 번이라도 상이한 경우 해당 API를 비결정적 API로 판별하여 테스트 대상에서 제외한다.

3.2 불안정한 입력 텐서 필터링 (Ill-conditioned Tensor Filtering)

수치적으로 불안정한 텐서는 미세한 입력 변화가 결과값의 큰 차이로 증폭되는 특성을 가지며, 이러한 입력으로부터 탐지된 잠정 계산 오류는 허위 양성일 가능성이 높다. 본 연구에서는 특이값 분해(singular value decomposition)를 통해 계산한 조건수(condition number)가 1,000 이상인 텐서를 수치적으로 불안정한 텐서로 정의하고 [7], 이를 테스트 입력에서 제외한다 (그림 1, ②).

표 1 데이터 타입별 허용 오차

데이터 타입	*atol*	*rtol*
`Uint8, Int8, Bool`	0	0
`Int16, Int32, Int64`	0	0
`Float16, BFloat16, ComplexHalf`	1×10^{-2}	1×10^{-3}
`Float32, ComplexFloat`	1×10^{-6}	1×10^{-5}
`Float64, ComplexDouble`	1×10^{-12}	1×10^{-10}

3.3 타입 맞춤형 결과 비교 (Type-aware Result Comparison)

결과 텐서를 비교할 때, 텐서의 데이터 타입에 따라 허용 오차를 조정함으로써 계산 오류 탐지 정확도를 향상시킨다 (그림 1, ③). 일반적인 차등 테스팅에서는 다음 식과 같이 절대 허용 오차(*atol*)와 상대 허용 오차(*rtol*)를 사용한 값 비교를 수행한다. 여기서 x_1, x_2는 두 결과 텐서의 요소 값이다.

$$|x_1 - x_2| \le atol + rtol \cdot |x_2|$$

기존 기법들은 일반적으로 고정된 기본값(atol = 1×10^{-8}, rtol = 1×10^{-5})을 사용한다. 본 연구에서는 텐서의 데이터 타입에 따라 서로 다른 허용 오차를 적용함으로써, 부동소수점 연산으로 인한 허위 양성을 추가적으로 감소시킨다 (표 1).

4. 실 험

4.1 실험 환경

제안 기법의 유효성을 검증하기 위해 차등 테스팅 파이프라인을 구현하고 실험을 수행하였다. 실험 구현에는 딥러닝 라이브러리 API 퍼저인 PathFinder를 테스트 입력 생성기로 사용하였으며 [8], 해당 퍼저가 생성한 입력을 기반으로 차등 테스팅 파이프라인을 구성하였다. 베이스라인으로는 동일한 입력 생성 및 실행 환경에서, 불안정한 입력 텐서 필터링과 타입 맞춤

표 2 실험 결과

Approach	Baseline	Ours
True Positive	16	16
False Positive	90	40
Total	106	56
Precision (%)	15.09	**28.57**

형 결과 비교를 적용하지 않고 고정된 허용 오차를 사용하는 기존 차등 테스팅 퍼저를 구현하여 비교 실험을 수행하였다.

PyTorch (v2.2) 의 총 781개 API를 대상으로, 각 API에 대해 6분간 퍼징을 수행하였다. 모든 실험은 Ubuntu 22.04, Intel(R) Xeon(R) Gold 6248R CPU및 NVIDIA GeForce RTX 3090 GPU 환경에서 수행되었다.

4.2 실험 결과

실험 결과, 제안한 파이프라인의 정확도는 28.57%로 베이스라인의 정확도(15.09%)에 비해 현저히 높게 나타났다(표 2). 베이스라인과 제안 기법은 각각 106개와 56개의 양성을 탐지하였다. 탐지된 결과의 실제 오류 여부를 판별하기 위해 API 문서 및 구현 세부사항을 기반으로 한 수작업 분석을 수행한 결과, 두 방법론 모두 16개의 실제 양성을 탐지하였음을 확인하였다. 그러나 베이스라인은 전체 탐지 결과 중 90개가 허위 양성인 반면, 제안한 기법은 40개의 허위 양성만을 포함하여 허위 양성의 수를 크게 감소시켰다. 이로 인해 제안 기법은 베이스라인 대비 두 배에 가까운 정확도 향상(15.09% → 28.57%)을 달성하였다. 이러한 결과는 제안한 파이프라인이 기존 차등 테스팅 기법에서 빈번히 발생하던 허위 양성을 효과적으로 제거함을 보여주며, 본 연구에서 제안한 접근법의 유효성을 뒷받침한다.

추가적으로, 제안한 파이프라인이 탐지한 16개의 실제 오류 중 중복을 제외한 10개를 PyTorch 공식 레포지토리에 보고하였으며, 이 중 3개는 PyTorch 개발자에 의해 실제 오류로 확인되었다. 그림 2는 PyTorch API 중 하나인 torch.special.logit에서 발견된 계산 오류를 재현하는 코드 예시를 보여준다. 분석 결과, 해당 오류는 CUDA 구현에서 bfloat16 타입 입력을 float 타입으로 캐스팅하는 과정에서 발생한 구현 결함으로 확인되었다.

```
import torch

self = torch.tensor([[0.0835]], dtype=torch.bfloat16)
result_cpu = torch.special.logit(self)
self_cuda = self.cuda()
result_gpu = torch.special.logit(self_cuda)

print (torch.allclose(result_cpu, result_gpu.cpu(),
       atol=1e-02, rtol=1e-03)) // False
```

그림 2 torch.special.logit 의 계산 오류 테스트 코드

5. 결 론

본 논문에서는 딥러닝 라이브러리에서 발생하는 계산 오류를 자동으로 탐지하기 위한 차등 테스팅 기반 퍼징 파이프라인을 제안하였다. 차등 테스팅은 계산 오류 탐지를 위한 효과적인 방법론이지만, 기존 기법은 허위 양성이 빈번하게 발생하여 실제 활용에 제약이 따른다는 한계를 가진다.

제안 기법은 이러한 한계를 완화하기 위해 비결정적 API의 사전 제거, 수치적으로 불안정한 입력 텐서의 필터링, 그리고 데이터 타입 맞춤형 결과 비교를 통합한다. PyTorch 라이브러리의 781개 API를 대상으로 수행한 실험 결과, 제안한 파이프라인은 베이스라인 대비 다수의 허위 양성을 제거하면서도 실제 계산 오류를 효과적으로 탐지함을 확인하였다.

사 사

이 성과는 정부(과학기술정보통신부 및 교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(RS-2022-NR069867, RS-2025-25424609)

참고 문헌

[1] “PyTorch”, https://pytorch.org

[2] “TensorFlow”, https://www.tensorflow.org

[3] A. Wei, Y. Deng, C. Yang, and L. Zhang, “Free lunch for testing: fuzzing deep-learning libraries from open source,” in *Proceedings of the 44th International Conference on Software Engineering* (ICSE ’22), p. 995–1007.

[4] Y. Deng, C. Yang, A. Wei, and L. Zhang, “Fuzzing deep-learning libraries via automated relational api inference,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE ’22), p. 44–56.

[5] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang, “Fuzzing Automatic Differentiation in Deep-Learning Libraries,” in *Proceedings of the 45th International Conference on Software Engineering* (ICSE '23), p. 1174–1186.

[6] D. Goldberg, ”What Every Computer Scientist Should Know About Floating-Point Arithmetic,” *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, Mar. 1991.

[7] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed., Johns Hopkins University Press, 2013.

[8] S. Kim, Y. Kim, D. Park, Y. Jeon, J. Yi, and M. Kim, “Lightweight concolic testing via path-condition synthesis for deep learning libraries,” in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (ICSE ’25)

TabulaRNN 기반의 소프트웨어 결함 예측

신중현° 류덕산
전북대학교 소프트웨어공학과
wndgus543@jbnu.ac.kr, duksan.ryu@jbnu.ac.kr

Software Defect Prediction based on TabulaRNN

Jung-Hyeon Shin° Duk-San Ryu
Department of Software Engineering, Jeonbuk National University

요 약

SDP는 소프트웨어 품질 확보와 유지보수 비용 절감을 위해 필수적인 과정이다. 기존의 ML 기반 모델들은 수치형 소프트웨어 메트릭을 입력으로 활용했지만, 이들은 변수 간의 복잡한 상호작용을 충분히 반영하기 어렵다는 한계가 있었다. 이에 본 연구는 정형 메트릭 간 의존성을 보다 효과적으로 학습하기 위한 표현 학습 관점에서, RNN 기반의 TabulaRNN 모델을 SDP에 적용한다. TabulaRNN은 시간적 시계열을 전제로 하는 것이 아니라, 메트릭 벡터를 고정된 특성 순서에 따라 구성된 입력 표현으로 처리하여 변수 간 상호작용 패턴을 포착하는 데 강점을 가진다. AEEEM 데이터셋 기반 Stratified 5-Fold 교차검증 실험에서 TabulaRNN은 Random Forest 및 XGBoost 대비 PD 지표에서 소폭 우수한 성능을 보였다. 다만 PF가 증가하는 경향이 함께 관찰되어, 탐지율 향상과 오탐지 감소 사이의 trade-off가 존재함을 확인하였다. 본 연구 결과는 TabulaRNN 계열 접근이 정형 메트릭 기반 SDP에서 활용될 수 있는 가능성을 제시한다.

1. 서 론

현대 소프트웨어 시스템은 점점 더 복잡해지고 있으며, 그에 따라 유지보수와 품질 보증에 필요한 비용과 노력도 증가하고 있다. 이러한 상황에서 소프트웨어 결함 예측(SDP)은 소프트웨어가 배포되기 전에 결함 가능성이 있는 모듈을 사전에 식별함으로써, 소프트웨어의 신뢰성을 향상시키고 유지보수 비용을 절감하는 데 중요한 역할을 한다. 기존의 SDP 연구에서는 주로 코드로부터 추출한 수치 기반 소프트웨어 메트릭을 입력으로 활용하여 다양한 기계 학습(ML)이 적용되어 왔다. ML 모델 중에서도 트리 기반 앙상블 기법을 사용하여 안정적인 예측 성능을 보였으나, 메트릭 간의 복잡한 상호작용이나 순차적 구조를 반영하지 못하는 한계가 존재한다.

본 연구에서는 SDP의 효율적인 수행을 위해, 최신 딥러닝 기법인 TabulaRNN[1]을 적용하고자 한다. TabulaRNN은 기존의 기법들과 달리 정형 입력 특성을 모델 내부에서 표현 학습이 가능하도록 구성한 순환 신경망(RNN) 기반 구조를 채택한다. 이는 시간적 시계열을 가정하는 것이 아니라, 다변량 메트릭 간의 의존성을 학습하기 위한 표현 방식으로 활용된다. 이를 통해 고정된 feature 조합 설계 없이도 메트릭 간 관계에서 나타나는 패턴을 포착할 수 있으며, 모델이 직접 feature 간 연관성을 학습하도록 유도할 수 있다. 본 연구는 AEEEM 데이터셋의 다양한 소프트웨어 메트릭을 활용하여 TabulaRNN의 예측 성능을 평가하고, SDP에 적합한 주요 하이퍼파라미터를 체계적으로 탐색·분석한다. 본 연구 결과는 TabulaRNN의 SDP 분야 활용 가능성을 제시하고, 소프트웨어 품질 향상에 기여할 수 있을 것으로 기대한다.

2. 관련 연구

기존 연구에서는 다양한 통계적 모델과 ML 기법을 활용하여 SDP 모델을 개발하고 있으며, 각 접근 방식은 사용한 데이터 특성과 모델링 기법에 따라 다양한 성능을 보여왔다. 특히 Random Forest, XGBoost와 같은 트리 기반 앙상블 기법은 높은 예측 성능을 달성하여 널리 활용되고 있다. Nikhil Saji Thomas [2]는 Random Forest에 SMOTE를 적용하여 클래스 불균형 문제를 완화하고, 하이퍼파라미터 최적화를 통해 기존 기법보다 우수한 성능을 보고하였다. Tariq Najim AL-Hadidi[3] 역시 XGBoost를 기반으로 Grid Search를 활용한 튜닝을 통해 예측 성능을 향상시켰다.

그러나 이러한 접근들은 공통적으로 수치형 메트릭을 독립적인 입력 변수로 처리하는 경향이 있다. 수치형 메트릭 기반 예측은 데이터 가공이 간단하다는 장점이 있지만, 메트릭 간 복합적인 비선형 상호작용이나 조합 효과를 충분히 반영하기 어려워 소프트웨어 모듈의 내재된 복잡성을 효과적으로 모델링하는 것에 한계가 있을 수 있다.

TabulaRNN은 다양한 정형 데이터 분석에서 우수한 성능을 입증한 바 있으나[1], SDP 분야에서는 아직 적용 사례가 충분히 보고되지 않았다. 따라서 정형 메트릭 기반 SDP에서 TabulaRNN 계열 접근의 유효성을 검증하고 적용 가능성을 탐색하는 연구가 필요하다. 본 연구는 TabulaRNN을 SDP에 적용하여 그 실효성을 검증하고, 정형 메트릭 기반 딥러닝 모델의 확장 가능성을 탐색하고자 한다.

3. 연구 방법

본 연구는 TabulaRNN 모델을 적용하여 SDP를 수행하였다. 제안 모델의 성능을 단일 분할에 의존하지 않고 안정적으로 검증하기 위해, 전체 데이터를 클래스 비율을 유지하는 Stratified 5-Fold 교차검증으로 분할하였다. 각 Fold에서는 학습 세트로 모델을 학습한 뒤, 분리된 테스트 세트에 대해 예측을 수행하여 성능을 산출하였으며, 모든 Fold의 결과를 종합하여 평균 성능을 보고하였다. 또한 연구 방법의 재현성과 절차적 명확성을 위해, 데이터 전처리부터 학습·추론·평가까지의 전체 흐름은 그림 1에 제시하고, Fold 단위로 반복되는 TabulaRNN의 학습 및 평가 과정은 Algorithm 1로 단계별 정리하였다.

데이터 전처리 단계에서는 각 Fold에서 학습/테스트 분할을 먼저 수행한 후(6행), 학습 데이터에 대해서만 Min-Max 정규화를 학습하고 동일한 변환을 테스트 데이터에 적용하였다(7행). 이는 테스트 데이터의 통계 정보가 전처리에 반영되는 데이터 누수를 방지하기 위함이다. 이후, 학습 데이터에서 값이 모두 동일한 특성인 constant features를 판별하여 제거하고, 동일한 특성을 테스트 데이터에서도 제거하였다(8행). 각 Fold에 대해 학습 데이터에

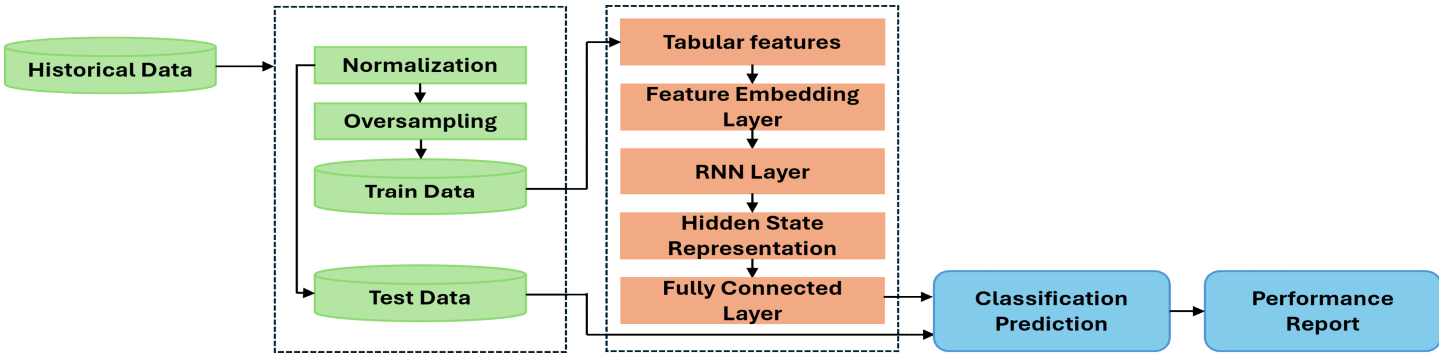


그림 1 연구 방법

SMOTE를 적용하여 클래스 불균형을 완화하였으며(9행), SMOTE는 학습 데이터에만 수행되었다. 모델 학습 단계에서는 입력 특성을 임베딩한 뒤, 이를 RNN 구조를 통해 처리하였다(11행). 이때 t는 시간축이 아니라 특성 시퀀스에서의 위치 인덱스이며, 본 연구에서는 메트릭 컬럼의 순서를 고정하여 입력 시퀀스를 구성하였다(10행). 은닉 상태 갱신은 수식 (1)과 같이 정의된다.

$$h_t = \sigma(W_h h_{t-1} + W_x \tilde{x}_t + b) \quad (1)$$

여기서 $\tilde{x}_t$는 t 번째 메트릭의 임베딩, W_h와 W_x는 학습 가능한 가중치 행렬, b는 편향, 그리고 $\sigma()$는 비선형 활성화 함수를 의미한다. 생성된 은닉 상태들을 평균하여 최종 특성 표현을 구성하고, 이를 MLP 헤드에 입력하여 결함 여부를 예측하였다(11행). 학습은 Early Stopping 기법을 적용하여, 학습 데이터 내부에서 별도의 검증 세트를 분리하여 검증 성능 향상이 더 이상 발생하지 않을 때 학습을 조기 종료하였다(12행). 학습된 모델은 테스트 데이터에 대해 예측을 수행하였으며, 최종 이진 분류를 위해 검증 세트에서 Balance를 최대화하는 threshold을 선택한 뒤 해당 threshold을 테스트 세트에 고정 적용하였다(13-14행). 모델 성능 평가는 SDP 연구에서 일반적으로 활용되는 지표를 기반으로 수행하였으며(15행), 모든 Fold 결과는 평균과 표준편차로 요약하여 보고하였다(19행).

Algorithm 1. TabulaRNN

```
1: /* Preprocessing */
2: Perform stratified 5-Fold split
3:
4: /* Cross-Validation */
5: for each fold do
6: Split data into training set and test set for the current fold
7: Fit Min-Max scaler on the training set only, then transform both training and test sets
8: Remove constant features based on the training set only, and drop the same features from the test set
9: Apply SMOTE to training data after scaling (training set only)
10: Fix feature order (e.g., the original metric column order) and encode features into a feature-sequence representation
11: Encode features and process through RNN-MLP pipeline
12: Train with early stopping using an internal validation split from the training set
13: Select the decision threshold on the validation set (e.g., maximizing Balance), then keep it fixed for the test set
14: Predict and threshold test set outputs
15: Compute PD, PF, Balance, FIR
16: end for
17:
18: /* Result Aggregation */
19: Report mean and standard deviation of metrics
```

4. 실험 설정

4.1 연구 질문

RQ1: TabulaRNN이 타 기법 대비 결함 예측 성능이 우수한가?

- H_{10}: TabulaRNN의 결함 예측 성능이 타 기법과 유사하다.
- H_{1A}: TabulaRNN의 결함 예측 성능이 타 기법 대비 우수하다.

RQ2: TabulaRNN의 하이퍼파라미터가 결함 예측 성능에 영향이 있는가?

- H_{20}: TabulaRNN의 하이퍼파라미터가 결함 예측 성능에 영향이 없다.
- H_{2A}: TabulaRNN의 하이퍼파라미터가 결함 예측 성능에 영향이 있다.

본 연구에서는 TabulaRNN의 결함 예측 성능을 평가하기 위해 RQ1과 RQ2에 대해 귀무 가설(H_0)과 대립 가설(H_A)을 수립하고, 통계적 검정을 수행하여 대립 가설 채택 가능성을 검증하였다.

4.2 데이터셋

표 1. 실험 데이터 셋

dataset	Project	#of instance		#of metric	Prediction Granularity
		All	Buggy		
AEEEM	EQ	324	129(39.81%)	61	Class
	JDT	997	206(20.66%)	61	Class
	LC	691	62(9.26%)	61	Class
	ML	1862	245(13.16%)	61	Class
	PDE	1492	209(14.01%)	61	Class

모델의 성능을 평가하기 위해 오픈소스 프로젝트인 AEEEM 데이터 셋을 이용한다. 데이터의 세부적인 정보는 표1과 같다.

4.3 평가 지표

표 2. 혼동 행렬

		Predicted class	
		Defective	Clean
Actual class	Defective	TP(True Positive)	FN(False Negative)
	Clean	FP(False Positive)	TN(True Negative)

본 연구에서는 모델의 성능 평가를 위해 혼동 행렬 중 예측 결과가 Positive로 판단된 사례 중 실제값 또한 Positive인 경우의 비율을 나타내는 PD, 실제 Negative 클래스에 속하는 데이터 중 Positive로 잘못 분류된 비율을 나타내는 PF를 평가하였다. 클래스 불균형 환경에서 모델의 균형 잡힌 성능을 확인하기 위해 Balance(=$1-\frac{\sqrt{(0-PF)^2+(1-PD)^2}}{\sqrt{2}}$)[4] 지표 또한 활용하였

다. 추가적으로, 코드 검사 과정에서의 노력 절감 효과를 분석하기 위해 FIR(=$\frac{PD-FI}{PD}$)지표[5]를 사용하였다. FI는 실제로 결함이 존재하지 않는 모듈을 결함이 있는 것으로 잘못 식별한 비율이다.

4.4 Baseline

본 연구는 TabulaRNN의 성능을 검증하기 위해 SDP에서 널리 사용되는 트리 기반 앙상블 모델을 Baseline으로 설정하였다. 비교 대상은 Random Forest와 XGBoost이며, 두 모델 모두 테이블 메트릭 데이터에서 강력한 성능과 안정성을 보이는 대표적 기법이다.

공정한 비교를 위해 Baseline은 TabulaRNN과 동일한 실험 설정을 따른다. 즉 Stratified 5-Fold로 데이터를 분할하고, 훈련 Fold에만 SMOTE를 적용하며, constant feature 제거 및 정규화를 동일하게 수행한다.

5. 실험 결과

5.1 RQ1: TabulaRNN이 타 기법 대비 결함 예측 성능이 우수한가?

본 연구에서는 TabulaRNN과 XGBoost, Random Forest 비교 실험을 진행했다.

표 3. TabulaRNN과 다른 모델들과의 성능 비교

Metric	Model		
	TabulaRNN	XGBoost	Random Forest
PD	0.4962	0.4787	0.4941
PF	0.1436	0.1020	0.1058
Balance	0.6150	0.6165	0.6223
FIR	0.4559	0.5471	0.4460

표 3은 세 모델의 평균 성능을 요약한 결과이다. TabulaRNN은 PD는 가장 높은 성능을 보였고, Balance와 FIR은 비슷한 성능을 보였지만, PF가 0.1436으로 가장 낮은 성능을 보였다. TabulaRNN은 PD가 0.4962로 가장 높게 나타났으며, 이는 결함 모듈 탐지 관점에서 상대적으로 민감하게 반응하는 경향을 시사한다. 반면 PF는 0.1436으로 XGBoost(0.1020) 및 Random Forest(0.1058)보다 높게 나타났고, 이에 따라 Balance(0.6150)와 FIR(0.4559)은 기존 모델 대비 뚜렷한 향상을 보이지 않거나 유사한 수준으로 관찰되었다.

표 4. effect_size 비교

Tabular RNN	Measure			
	PD	PF	Balance	FIR
XGBoost	0.2935(S)	1.3563(L)	0.0405(VS)	-1.5890(L)
Random Forest	0.0283(VS)	1.1888(L)	-0.1220(VS)	0.1710(VS)

표 4는 TabulaRNN과 다른 기법 간 Cohen's d[6] 효과 크기를 비교한 결과이다. TabulaRNN은 XGBoost 대비 PF 지표에서 Large 수준, Balance 및 FIR 지표에서는 Very small 수준의 차이를 보였다. Random Forest 대비 PF 지표에서는 Large 수준, PD 및 Balance에서는 Very small 수준, FIR 지표에서는 Small 수준에 가까운 차이를 나타냈다. 이는 TabulaRNN의 성능 차이가 주로 PF에서 크게 발생하며, Balance/FIR과 같은 종합 지표에서는 실질적인 차이가 제한적일 수 있음을 의미한다.

결론적으로, TabulaRNN은 결함 모듈 탐지에 민감하게 반응하여 PD 측면에서는 기존 모델보다 우수한 경향을 보였다. 그러나 비결함 모듈에 대한 오탐지 비율이 증가하여 PF가 상승하였고, 이에 따라 Balance 및 FIR 측면에서는 기존 모델과 유사한 성능을 보였다. 따라서 TabulaRNN은 탐지율을 우선하는 품질 게이트 시나리오에는 유리할 수 있으나, 오탐지를 최소화해야 하는 운영 환경에서는 비용 민감 학습과 같은 추가적인 완화 전략이 필요하다.

5.2 RQ2: TabulaRNN의 하이퍼파라미터가 결함 예측 성능에 영향이 있는가?

표 5. 하이퍼파라미터 범위와 기본값

Parameter	Range	Default
n_layers	{1, 2, 3, 4, 5, 6}	4
d_model	{32, 48, 64, 80, 96, 128}	64

표 5는 TabulaRNN 모델의 주요 하이퍼파라미터인 n_layers와 d_model의 범위 및 기본값을 나타낸다. n_layers는 RNN 블록의 깊이로, 값이 클수록 복잡한 패턴을 학습할 수 있지만 과적합 위험이 증가한다. d_model은 입력 feature를 임베딩하는 차원의 크기로, 값이 커지면 표현력이 향상되지만, 모델 크기와 과적합 가능성도 증가한다.

표 6. 최적의 파라미터 값

dataset	Project	Best performing parameter(Balance)	
		n_layers	d_model
AEEEM	EQ	3	80
	JDT	6	48
	LC	3	80
	ML	1	64
	PDE	3	48

표 6은 프로젝트별 Balance 지표를 기준으로 최고의 성능을 보이는 TabulaRNN의 파라미터값을 나타낸 것이다.

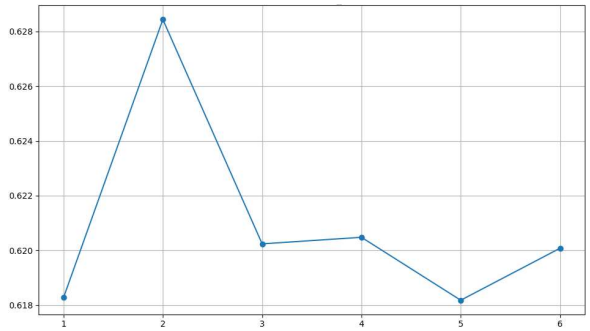


그림 2. n_layers 값에 따른 Balance 지표

그림 2는 n_layers 값에 따른 Balance 지표이다. n_layers가 2일 때 0.6284로 가장 높았다.

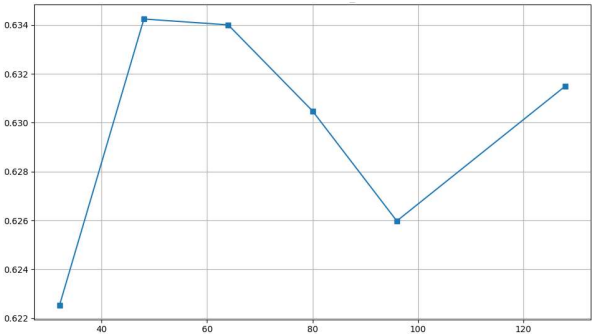


그림 3. d_model 값에 따른 Balance 성능 지표

그림 3은 d_model 값에 따른 Balance 성능 지표이다. d_model이 48일 때 0.6342로 가장 높았다.

하이퍼파라미터의 값에 따라 성능 차이가 있음을 파악했고 데이터의 특성에 맞춰 적합한 하이퍼파라미터 값을 조정 해줘야 함을 확인했다.

6. 위협 요소

모델 성능이 하이퍼파라미터 설정에 민감하게 반응하였으며, 데이터 셋마다 최적값이 달라 일반화가 어렵다. 또한 TabulaRNN은 입력 특성을 일정한 순서로 배열하여 처리하는 구조이므로, 메트릭의 순서 정의 방식에 따라 성능이 변동할 수 있다. 연구는 AEEEM의 일부 프로젝트에 한정되어 있어 다양한 도메인에 대한 확장 가능성은 충분히 검증되지 않았다. 아울러 본 연구는 PD와 PF 간 trade-off가 존재할 수 있으며, 운영 환경에서의 비용까지 반영한 최적화는 본 논문 범위에서 충분히 다루지 못하였다.

7. 결론 및 향후 과제

본 연구에서는 정형 소프트웨어 메트릭 간 의존성 학습을 강화하기 위한 표현 방식으로 TabulaRNN 모델을 SDP에 적용하였다. TabulaRNN은 전통적인 ML 기법인 Random Forest, XGBoost와 비교하여 PD 지표에서 상대적으로 우수한 성능을 보였다. 다만 PF가 증가하는 경향이 관찰되어, 제안 모델이 모든 지표에서 일관되게 우수하다고 결론 내리기에는 한계가 있으며, 실제 적용 시에는 탐지율 향상과 오탐지 감소 사이의 trade-off를 고려한 운영 시나리오별 활용이 필요하다.

향후 연구에서는 Attention 기반 구조를 결합한 Hybrid 모델을 탐색함으로써 메트릭 간 상호작용을 보다 선택적으로 강조하고 오탐지을 완화할 수 있는 구조를 검토하고, 예측 결과에 대한 해석 가능성을 높일 계획이다. 또한, AEEEM 외의 다양한 공개 SDP 데이터셋 및 도메인을 활용한 실험을 통해 모델의 일반화 성능을 폭넓게 검증할 계획이다.

감사의 글

이 논문은 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원-대학ICT연구센터(ITRC)의 지원 (IITP-2026-RS-2020-II201795, 50%) 및 정보통신기획평가원-지역지능화혁신인재양성사업의 지원을 받아 수행된 연구임(IITP-2026-RS-2024-00439292, 50%)

참고 문헌

[1] Anton Frederik Thielmann, Soheila Samiee, “On the Efficiency of NLP-Inspired Methods for Tabular Deep Learning”, arXiv, 2024

[2] Nikhil Saji Thomas, S. Kaliraj, “An Improved and Optimized Random Forest Based Approach to Predict the Software Faults”, SN Computer Science, 5, 5, 530, 2024

[3] Tariq AL-Hadidi, Safwan Omar Hasoon, “Software Defect Prediction Using Extreme Gradient Boosting (XGBoost) with Tune Hyperparameter”, Al-Rafidain Journal of Computer Sciences and Mathematics, 18, 1, 2024

[4] Shuo Feng, Jacky Keung et al. “Investigation on the Stability of SMOTE-Based Oversampling Techniques for Software Defect Prediction”, Information and Software Technology, 135, 106582, 2021

[5] Sunjae Kwon, Duksan Ryu et al “eCPDP: Early Cross-Project Defect Prediction”, 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2021

[6] Shlomo S. Sawilowsky, “New Effect Size Rules of Thumb”, journal of Modern Applied Statistical Methods, 8, 2, 597-599, 2009

[7] Shuo Wang, Xin Yao, “Using Class Imbalance Learning for Software Defect Prediction”, IEEE Transactions on Reliability, 62, 2, 434-443, 2013

[8] Ravid Shwartz-Ziv, Amitai Armon, “Tabular Data: Deep Learning is Not All You Need”, arXiv, 2021

[9] Yejin Hwang, Jongwoo Song, “Recent deep learning methods for tabular data”, Cpmmunications for Statistical Applications and Methods, 30, 2, 215-226, 2023

[10] Feseeha Matloob, Taher M. chazal et al. "Software defect prediction using ensemble learning: A systematic literature review.", IEEE Access vol. 9, 98754-98771, 2021

Code LLaMA를 활용한 자연어 프롬프트 기반의 소프트웨어 결함 예측

김민재°, 류덕산
전북대학교 소프트웨어공학과
{cavper, duksan.ryu}@jbnu.ac.kr

Software Defect Prediction using Natural Language Prompt using Code LLaMA

Minjae Kim°, Duksan Ryu
Department of Software Engineering, Jeonbuk National University

요 약

소프트웨어 결함 예측(Software Defect Prediction, SDP)은 품질 확보와 유지보수 비용 절감을 위한 핵심 과정이다. 그러나 기존의 수치 기반 머신러닝(ML) 모델은 복잡한 문맥이나 의미적 패턴을 충분히 반영하지 못하며, 새로운 도메인에 대한 일반화 성능 역시 낮은 경향이 있다. 본 연구는 자연어 기반 입력을 활용하여 이러한 한계를 극복하는 새로운 소프트웨어 결함 예측 접근 방식을 제시하는 것을 목표로 한다. 이를 위해 소프트웨어 메트릭 기반의 결함 데이터를 자연어 형식으로 구성해 자연어 명령 이해에 강점을 가진 Code LLaMA를 미세조정하고, 질의 프롬프트로 결함 여부를 예측하였다. 본 연구에서는 이러한 방법을 기반으로 자연어 프롬프트 입력 방식을 결함 예측에 적용하였으며, 실험 결과, 일부 데이터셋에서 성능 향상이 관찰되었고, 데이터셋별로 성능 양상이 달라지는 경향을 보였다. 본 연구는 LLM을 활용한 자연어 기반 예측 접근이 SDP에 효과적으로 적용될 수 있음을 실험적으로 확인하였다는 점에서 의의가 있다.

1. 서 론

소프트웨어 개발 과정에서 결함을 사전에 예측하는 것은 품질 향상과 유지보수 비용 절감에 중요한 역할을 한다. 특히 결함이 있는 모듈을 조기에 식별하면 자원을 효율적으로 배분할 수 있어, 시스템의 신뢰성과 안정성을 높일 수 있다. 기존 결함 예측 연구는 결합도, 변경 이력 등 수치형 데이터를 기반으로 한 ML 모델을 활용해 왔으나, 복잡한 상호작용이나 문맥적 의미를 반영하는 데 한계가 있으며, 새로운 도메인에 대한 일반화에도 어려움이 있다. 최근에는 LLM의 발전으로 자연어 기반 입력을 이해하는 고성능 모델이 주목받고 있다. 그 중 Meta에서 개발한 Code LLaMA는 프롬프트 처리 능력이 뛰어나고 강력한 문맥 이해 능력을 갖추고 있다. 본 연구는 결함 데이터를 자연어 형태로 변환하여 Code LLaMA에 입력하고, 이를 통해 결함 여부를 분류하는 모델을 설계하였다. 실험을 통해 LLM 기반 접근의 적용 가능성을 확인했으며, Code LLaMA에 자연어 형태로 구성한 결함 데이터를 입력해 예측하는 방식으로 기존 수치형 입력 기반 접근과 차별화되는 결함 예측 가능성을 제시한다.

2. 관련 연구

LLM의 발전으로 자연어 기반 입력을 이해하고 소프트웨어공학 문제를 해결하려는 연구가 활발히 진행되고 있으며, 결함 예측 분야에서도 그 활용 가능성이 주목받고 있다. Zheng et al.[1]은 Transformer 기반 모델을 활용하여 수치형 메트릭 데이터를 입력으로 사용해 결함 예측을 수행하였으며, 딥러닝 기반 접근이 기존 전통 모델 대비 효과적일 수 있음을 보였다. 이정화 외[2]는 정형 결함 데이터를 기반으로 이상 탐지를 수행하며, LLM의 구조적 이해 능력이 예측 정확도 향상에 기여함을 확인하였다. 또한 Panthaplackel et al.[4]은 이슈 설명과 메타 정보를 활용해 버그 심각도를 예측함으로써 자연어 입력 기반 LLM의 활용 가능성을 보였다. 한편 Touvron et al.[5]은 Code LLaMA가 프롬프트 기반 지시 수행과 문맥 이해를 효과적으로 지원하도록 학습·설계된 모델임을 제시하였다. 기존 연구들이 대부분 코드 또는 주석과 같은 소스 기반 입력에 집중한 반면, 본 연구는 소프트웨어 메트릭 데이터를 자연어 형태로 변환하여 Code LLaMA에 입력함으로써, 정형 데이터 기반의 LLM 활용이라는 새로운 접근을 제안한다.

3. 연구 방법

본 연구는 소프트웨어 결함 데이터를 자연어 프롬프트로 변환하여 Code LLaMA에 입력하고, 결함 여부를 분류하는 LLM 기반 결함 예측 모델을 제안한다. 그림 1은 소프트웨어 메트릭 데이터를 기반으로 자연어 프롬프트를 생성한 후, Code LLaMA 모델에 입력하여 결함 여부를 분류하는 전체 연구 과정을 나타내며, 절차는 다음과 같다.

Stratified K-Fold(n=5)를 적용하여 학습/검증 세트를 구성한다(1행). 훈련 데이터에는 SMOTE를 적용해 클래스 불균형을 완화한다(2~3행). train/val 데이터를 자연어 프롬프트로 변환하고, 라벨은 'buggy'는 1, 'clean'은 0으로 부여한다(4~5행). 프롬프트를 Code LLaMA 토크나이저로 토큰화한다(6행). 4bit QLoRA 설정이 적용된 Code LLaMA 7B 모델을 로드한다(6~7행). 학습 데이터로 모델을 학습하고, 검증 데이터로 예측을 수행한다(8~9행). 예측 결과로부터 PD, PF, Balance, FIR을 계산하고 Fold별 결과를 저장한다(10행). 모든 Fold에 대해 반복 수행한 후, 평균 성능 지표를 최종 결과로 산출한다(11~12행).

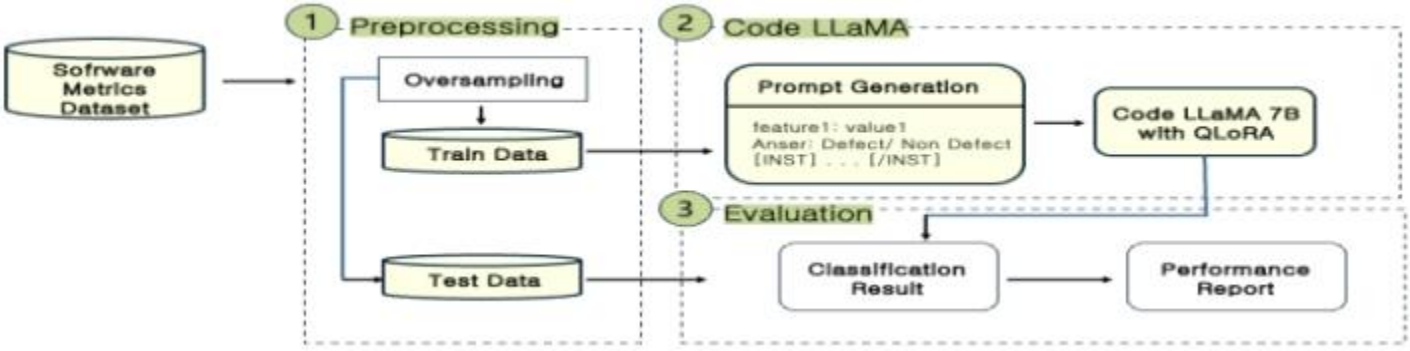


그림1 . 연구 방법

Algorithm 1. Code LLaMA

Input: Software metrics dataset D = {X, y}
Output: Evaluation metrics (PD, PF, Balance, FIR)
1: Initialize **Stratified K−Fold (n = 5)**
2: for each (train_df, val_df) in folds do
3: Apply **SMOTE** to train_df
4: Convert each instance in train_df and val_df into natural **language prompts**
5: **Assign labels: 1 if 'buggy', 0 if 'clean'**
6: Tokenize prompts using Code LLaMA tokenizer
7: **Load Code LLaMA 7B with QLoRA for binary classification**
8: Train model on train_df
9: Predict labels on val_df
10: Evaluate PD, PF, Balance, FIR using confusion matrix
11: end for
12: Return average metrics across all folds

4. 실험 설정

4.1 연구 질문

RQ1: Code LLaMA는 SDP 분야에서 효과적인 성능을 제공할 수 있는가?

RQ2: 자연어 프롬프트의 설계 방식이 Code LLaMA의 결함 예측 성능에 영향을 미치는가?

4.2 데이터셋

표 1. 실험 데이터셋

Dataset	Project	# of instances		# of metric	Granularity
		all	buggy		
AEEEM	EQ	324	123 (39.81%)	61	class
	JDT	997	206 (20.66%)	61	class
Relink	apache	194	98 (50.52%)	26	isDefective
	safe	56	22 (39.28%)	26	isDefective

모델의 성능을 평가하기 위해 오픈소스 소프트웨어 프로젝트에서 수집된 결함 데이터셋인 AEEEM과 Relink 데이터셋을 활용한다. 데이터셋의 구성 및 특징은 표1에 정리되어 있다.

4.3 데이터 전처리

각 인스턴스의 수치형 결함 데이터를 자연어 형식으로 변환하여 모델 학습에 사용하였고, 클래스 불균형 문제를 완화하기 위해, 훈련 데이터에만 SMOTE를 적용하였다.

4.4 성능 평가 지표

표 2. 혼동 행렬

	Actually Defective	Actually Clean
Predicted Defective	True Positive(TP)	False Positive(FP)
Predicted Clean	False Negative(FN)	True Negative(TN)

모델 성능 평가는 혼동 행렬을 기반으로 한다. PD는 실제 결함을 올바르게 탐지한 비율, PF는 결함이 없는 모듈을 잘못 탐지한 비율이다. Balance는 PD와 PF의 균형을 측정하는 지표로, 다음과 같은 수식으로 계산된다. FIR은 PD와(1−PF)의 조화를 측정하는 종합 지표로, 다음과 같은 수식으로 계산된다.

$$Balance = 1 - \frac{\sqrt{(1-PD)^2 + PF^2}}{\sqrt{2}} \quad (1)$$

$$FIR = \frac{2 \cdot PD \cdot (1-PF)}{PD + (1-PF)} \quad (2)$$

4.5 비교 모델 및 하이퍼파라미터 설정

비교 모델(Random Forest, XGBoost)은 Code LLaMA와 동일한 데이터셋 및 5−fold 분할 조건에서 산출된 결과를 기준으로 비교하였다. 하이퍼파라미터는 각 모델의 기본 설정을 사용하였다. 따라서 비교 결과는 기본 설정 기준의 성능 비교로 해석한다.

5. 실험 결과

5.1 RQ1. Code LLaMA는 SDP 분야에서 효과적인 성능을 제공할 수 있는가?

표 3은 Code LLaMA와 Randomforest, XGBoost를 비교한 결과로 Code LLaMA는 일부 데이터셋에서 전통 모델보다 높은

PD를 기록했으나, 전체적으로는 PD 성능이 낮은 경향을 보였다. apache와 safe에서는 각각 0.848, 0.779로 높은 값을 보였지만, JDT에서는 0.364로 가장 낮았다. Balance와 FIR 측면에서도 일부 지표에서 유사하거나 소폭 뒤처졌다. 이는 Code LLaMA가 일관된 우위를 보이지는 않았지만, 자연어 기반 입력을 활용한 새로운 접근 방식으로서 가능성을 보여주었다.

표 3. Code LLaMA와 비교 모델의 성능 비교

Dataset	Project	Model	Measure			
			PD	PF	Balance	FIR
AEEEM	EQ	Random forest	0.790	0.241	0.772	0.683
		XG Boost	0.667	0.256	0.701	0.631
		Code LLaMA	0.604	0.292	0.647	0.674
	JDT	Random forest	0.611	0.098	0.716	0.626
		XG Boost	0.601	0.079	0.712	0.672
		Code LLaMA	0.364	0.077	0.546	0.559
Relink	apache	Random forest	0.744	0.302	0.712	0.722
		XG Boost	0.735	0.270	0.719	0.748
		Code LLaMA	0.848	0.406	0.693	0.682
	safe	Random forest	0.636	0.232	0.694	0.642
		XG Boost	0.541	0.234	0.626	0.618
		Code LLaMA	0.779	0.381	0.681	0.567

표 4는 Code LLaMA와 다른 기법들 사이에 효과 크기를 비교한 결과이다. Code LLaMA는 PD 측면에서 Random forest, XGBoost 모델과 비슷한 수준을 보였다.

표 4. Effect Size 비교

Code LLaMA vs.	Measure			
	PD	PF	Balance	FIR
Random forest	−0.283(S)	0.580(M)	−1.546(L)	−0.851(L)
XG Boost	0.078(T)	0.645(M)	−0.851(L)	−0.749(M)

5.2 RQ2: 자연어 프롬프트의 설계 방식이 Code LLaMA의 결함 예측 성능에 영향을 미치는가?

- **Prompt1**: Here are some software metrics. Do you think the module might have a defect? Respond with either 'Defect' or 'Non−defect'.

- **Prompt 2**: Given the following software metrics, answer the question: Is the module defect−prone? Answer with 'Defect' or 'Non−defect'.

- **Prompt 3**: You are an expert software quality analyzer. Analyze the given software metrics and determine whether the module is DEFECTIVE or NOT DEFECTIVE. Reply strictly with 'Defect' or 'Non−defect'.

표 5. 프롬프트에 따른 성능 비교

	PD	PF	Balance	FIR
Prompt 1	0.475	0.305	0.582	0.570
Prompt 2	0.527	0.266	0.616	0.613
Prompt 3	0.648	0.289	0.641	0.620

표 5는 프롬프트 설계 방식에 따른 성능 차이를 보여준다. Prompt 1은 PD, Balance, FIR 모두에서 가장 낮은 성능을 기록하였으며, 지시가 명확한 Prompt 3이 가장 높은 성능이 나타났다. 이러한 결과는 프롬프트 표현의 차이가 LLM의 예측에 영향을 미칠 수 있음을 보여준다.

6. 위협 요소

본 연구는 소프트웨어 메트릭 기반 결함 데이터를 자연어 형식으로 구성하여 LLM에 적용하는 접근을 제안하였으나, 데이터셋에 따라 성능 편차가 나타나 일반화에는 한계가 있다. 또한 프롬프트 구성 방식과 추론 설정에 따라 모델 출력이 달라질 수 있어 결과의 일관성과 재현성이 저하될 가능성이 있다. 이는 Code LLaMA 기반 결함 예측이 입력 표현 방식과 출력 판정 기준 측면에서 아직 표준화가 충분하지 않음을 시사한다.

7. 결론 및 향후 과제

본 연구는 소프트웨어 메트릭 기반 결함 데이터를 자연어 형태로 변환하여 Code LLaMA에 입력하는 결함 예측 모델을 제안하였다. 일부 데이터셋에서는 PD에서 경쟁력 있는 성능을 보였으나, 일부 지표에서는 저조한 성능을 보였다. 향후에는 프롬프트 구성 방식의 체계적인 최적화와 함께, 다양한 LLM 아키텍처에 대한 비교 분석을 통해 성능을 정량적으로 개선하고, 모델의 일반화 가능성을 높이는 추가 연구가 필요하다.

감사의 글

이 논문은 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원−대학ICT연구센터(ITRC)의 지원(IITP−2026−RS−2020−II201795, 50%) 및 정보통신기획평가원−지역지능화혁신인재양성사업의 지원을 받아 수행된 연구임(IITP−2026−RS−2024−00439292, 50%)

참고 문헌

[1] Zheng, W., Tan, L., & Liu, C., “Software defect prediction method based on transformer model,” Proceedings of the 2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA), pp. 172−176, 2021.

[2] 이정화, 주은정, 류덕산, “소프트웨어 결함 예측에서의

LLM 적용 가능성에 대한 고찰," 2024 한국컴퓨터종합학술대회 논문집, pp. 1851-1853, 2024.
[3] Wang, S., Liu, T., Tan, L., "Automatically Learning Semantic Features for Defect Prediction," Proceedings of the 38th International Conference on Software Engineering (ICSE), pp. 297-308, 2016.
[4] Panthaplackel, A., Ye, A., and Balog, K., "On the Effectiveness of Large Language Models for Bug Severity Prediction," Proceedings of the 31st IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 461-472, 2024. doi: 10.1109/SANER59798.2024.00062.
[5] Touvron, H., Roziere, B., Lavril, T., Izacard, G., Martinet, X., Lachaux, M. A., et al., "Code Llama: Open Foundation Models for Code," arXiv preprint arXiv:2308.12950, 2023.
[6] Ahmad, I., Babar, M., Abbas, A., & Khan, S., "SDPERL: A Framework for Software Defect Prediction Using Ensemble Feature Extraction and Reinforcement Learning," arXiv preprint arXiv:2412.07927, 2024.
[7] Menzies, T., Greenwald, J., & Frank, A., "Data Mining Static Code Attributes to Learn Defect Predictors," IEEE Transactions on Software Engineering, vol. 33, no. 1, pp. 2-13, 2007. doi: 10.1109/TSE.2007.256941.
[8] Bhutamapuram, U. S., Chonari, F., Anilkumar, G. K., & Konchada, S., "LLMs for Defect Prediction in Evolving Datasets: Emerging Results and Future Directions," Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE), pp. 520-524, 2025. doi: 10.1145/3696630.3728491.
[9] Ince, P., Luo, X., Yu, J., Liu, J. K., & Du, X., "Detect Llama -- Finding Vulnerabilities in Smart Contracts using Large Language Models," arXiv preprint arXiv:2407.08969, 2024.
[10] Hong, H., Lee, S., Ryu, D., & Baik, J., "Enhancing Software Defect Prediction in Ansible Scripts Using Code-Smell-Guided Prompting with Large Language Models in Edge-Cloud Infrastructures," Current Trends in Web Engineering - ICWE 2024 International Workshops, BECS and WALS, 2024, Revised Selected Papers, Communications in Computer and Information Science, vol. 2188, Springer, pp. 30-42, 2025. doi: 10.1007/978-3-031-75110-3_3.

TabPFN 기반의 소프트웨어 결함 예측

임창우°, 류덕산
전북대학교 소프트웨어공학과
{cwoo6115, duksan.ryu}@jbnu.ac.kr

Software Defect Prediction based on TabPFN

Changwoo Lim°, Duksan Ryu
Department of Software Engineering, Jeonbuk National University

요 약

소프트웨어 결함 예측(Software Defect Prediction, SDP)은 소프트웨어 품질 향상을 위한 핵심 기술로 이를 위해 다양한 머신러닝 및 딥러닝 기반 예측 모델이 연구되어 왔다. 그러나 기존 모델들은 학습 데이터 의존성, 하이퍼파라미터 튜닝, 모델 재학습 등에서 한계를 가진다. 이러한 한계들은 SDP 환경에서의 예측 성능을 저하시킨다. 따라서 본 연구는 SDP 환경에서 학습 데이터 부족 문제를 해결하고 복잡한 하이퍼파라미터 튜닝에 대한 높은 민감성을 해소하는 것을 목표로 한다. 사전 학습된 인과 기반 분포를 활용해 추가 학습과 하이퍼파라미터 튜닝 없이도 예측이 가능한 Tabular Prior-data Fitted Network(TabPFN)을 SDP에 적용해 예측 성능을 향상시키고자 한다. 본 연구는 공개된 소프트웨어 결함 데이터셋을 사용하여 TabPFN의 성능을 Random Forest, XGBoost와 비교하였으며 PD, PF, Balance, FIR 4가지 지표를 기반으로 평가를 수행하였다. 연구 결과, TabPFN은 타 모델 대비 PD와 Balance 지표에서는 낮은 성능을 보였지만, PF와 FIR 지표에서 우수한 성능을 보였다. 이를 통해 SDP에 TabPFN을 적용하기 위해서는 PD, Balance 지표의 성능 개선을 위한 추가적인 연구가 필요함을 확인하였다.

1. 서 론

소프트웨어 시스템의 복잡성과 규모가 증가함에 따라, 결함 발생 가능성이 높은 모듈을 사전에 식별하는 소프트웨어 결함 예측(Software Defect Prediction, SDP)은 소프트웨어 품질 확보와 유지보수 효율 향상을 위한 핵심 기술로 자리 잡았다. 이를 위해 다양한 머신러닝(ML) 및 딥러닝(DL) 기반 예측 모델들이 제안되어 왔으나, 대부분의 기법은 학습 데이터에 대한 높은 의존성, 하이퍼파라미터 튜닝 필요성, 새로운 프로젝트 적용 시 재학습 요구 등의 한계를 가진다. 특히 모델 성능이 하이퍼파라미터 설정에 민감한 경우 반복적인 튜닝 과정이 필요하며, 이는 실제 적용 시 시간 및 자원 소모를 증가시키는 요인으로 작용한다. 또한 새로운 프로젝트나 데이터 환경에 적용하기 위해 재학습이 요구되는 경우, 라벨링된 결함 데이터가 충분하지 않으면 모델 활용이 제한되는 문제가 발생한다.

본 연구는 이러한 기존 한계를 배경으로, 사전 학습된 인과 기반 사전 분포를 활용하는 Transformer 기반 모델인 Tabular Prior-data Fitted Network(TabPFN)[1]을 SDP 문제에 적용하여 그 성능 특성을 분석한다. TabPFN은 사전에 다양한 합성 데이터셋을 통해 학습된 모델로, 추가적인 파라미터 재학습 없이 입력 데이터를 문맥 정보로 활용한 단일 forward pass 기반 추론이 가능하다는 구조적 특징을 가진다. 본 연구에서는 TabPFN을 기존의 대표적인 SDP 모델들과 비교하여 다양한 성능 지표 관점에서의 장단점을 분석하고, SDP 문제에서 TabPFN의 활용 가능성과 한계를 실험적으로 고찰한다.

2. 관련 연구

SDP 분야에서는 예측 성능 향상을 위해 다양한 ML 및 DL 기반 모델들이 연구되어 왔다. 그러나 기존의 많은 기법들은 학습 데이터에 대한 높은 의존성, 하이퍼파라미터 설정의 민감성, 그리고 모델 적용 시 반복적인 학습 과정이 요구된다는 한계를 가진다.

Qu et al.[2]은 하이퍼파라미터 설정에 따라 결함 예측 성능이 최대 34.6%까지 차이가 발생할 수 있음을 보였으며, 이는 모델 성능이 설정값에 매우 민감하고 실제 적용 시 추가적인 튜닝과 재학습이 필요할 수 있음을 시사한다.

Wang et al.[3]은 DBN 기반의 의미적 특징 학습을 통해 소프트웨어 결함 예측 성능을 향상시켰으나, 다단계 신경망 구조의 학습을 위해 반복적인 학습 과정과 파라미터 튜닝이 요구된다.

Dam et al.[4]은 AST 기반 Tree-LSTM 모델을 제안하여 코드의 구조적·의미적 정보를 효과적으로 학습하였으나, 모델 적용을 위해 사전 학습 및 재학습 과정이 필요하다는 한계를 가진다. 이러한 접근법들은 성능 향상에는 기여하지만, 새로운 데이터 환경이나 프로젝트에 즉시 적용하기에는 학습 비용 측면에서 부담이 존재한다.

기존 연구들은 다양한 구조와 학습 전략을 통해 성능 향상을 시도했으나 학습 데이터 의존성, 복잡한 하이퍼파라미터 튜닝, 재학습 등의 한계를 가진다. 본 연구는 이러한 한계를 보완하고자 추가 학습 없이 단일 추론만으로 예측이 가능한 TabPFN을 SDP에 적용한다.

3. 연구 방법

본 연구에서는 TabPFN을 SDP에 적용한다. 그림 1은 전체적인 연구 과정과 모델 내부 구조를 시각화한 것이다. TabPFN의

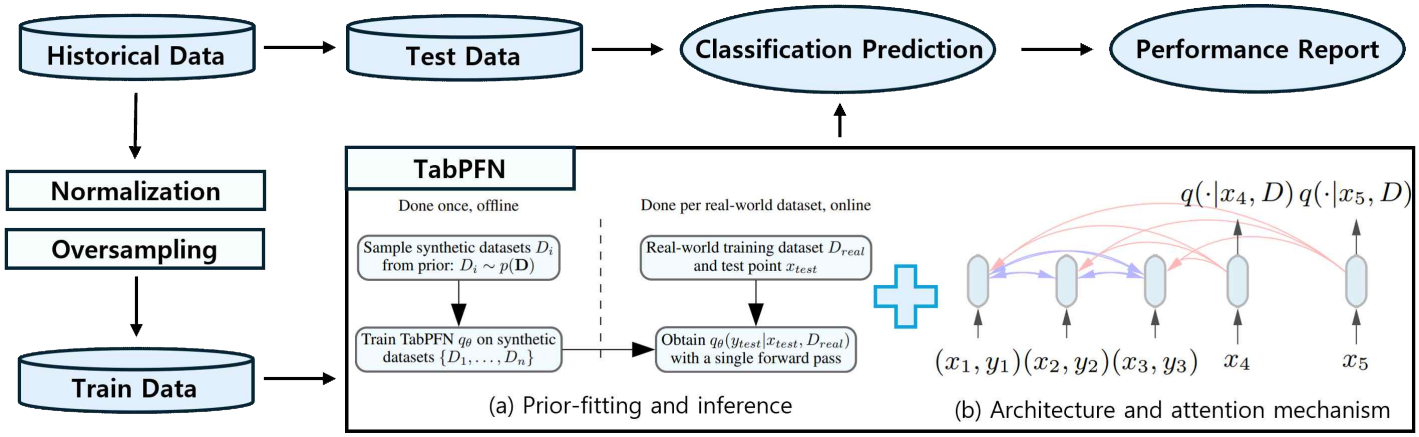


그림 1. 전체 연구 과정

(a) Prior-fitting and inference는 핵심 작동 방식인 사전 학습과 단일 추론 흐름을 설명한다. TabPFN은 사전에 다양한 합성 데이터셋을 생성하여 인과 기반 사전 분포를 학습하고, 실제 데이터가 입력되면 추가 학습 없이 단일 forward pass를 통해 예측을 수행한다. (b) Architecture and attention mechanism은 TabPFN 내부에서 수행되는 self-attention 기반 추론 구조를 나타낸다. 학습 샘플들은 문맥 정보로 활용되며, 테스트 샘플은 학습 샘플들에 attention을 수행하여 각 샘플에 대한 결함 발생 확률을 추론한다.

Algorithm 1. TabPFN

Input: Tabular data X, labels y
Output: Y_pred, evaluation metrics (PD, PF, Balance, FIR)
1: Split dataset into Train, Test sets
2: Normalize X using Min-Max scaling
3: For each fold in 10-fold cross-validation:
4: Split train set into fold_train and fold_val
5: Apply SMOTE to fold_train
6: Load pretrained TabPFN model
7: Fit TabPFN using fold_train as context set
8: Y_pred ← model.predict(X_test)
9: Compute confusion matrix
10: Calculate PD, PF, Balance, FIR
11: Store metrics for aggregation

Algorithm 1은 전체 실험 절차를 나타낸 것이다. 전체 데이터는 학습 데이터와 테스트 데이터로 분할되며(1행), 학습 데이터에 대해 Min-Max 정규화를 수행한다(2행). 이후 교차 검증을 적용하고 각 fold의 학습 데이터에 한하여 SMOTE[5]를 적용한다(3~5행). 사전 학습된 TabPFN 모델을 로드한 후(6행), 오버샘플링된 학습 데이터를 이용해 fit 과정을 수행한다(7행). TabPFN에서의 fit은 일반적인 머신러닝 모델에서의 파라미터 학습을 의미하지 않으며, 모델 파라미터를 갱신하지 않고 입력된 학습 샘플들을 문맥(context) 정보로 모델에 제공하기 위한 추론 준비 단계에 해당한다. 이 과정에서 TabPFN의 내부 파라미터는 고정된 상태로 유지되며, 옵티마이저나 그래디언트 기반의 학습은 수행되지 않는다. 이후 테스트 데이터는 해당 문맥 정보를 기반으로 단일 forward pass를 통해 예측이 수행되며(8행), 예측 결과와 실제 라벨을 비교하여 혼동 행렬을 생성한다. 혼동 행렬을 바탕으로 PD, PF, Balance, FIR의 네 가지 성능 지표를 계산하고(10행), 각 fold에서 산출된 평가 결과는 최종 성능 계산에 활용된다(11행).

TabPFN은 다양한 인과 구조를 내포하는 함수 공간(Φ)에 대한 베이지안 사후 예측 분포를 근사하는 방식으로 정의된다.

$$p(y|x, D) \propto \int_{\varnothing \in \Phi} p(y|x, \varnothing) p(D|\varnothing) p(\varnothing) d\varnothing \quad (1)$$

수식 (1)은 입력 데이터 x와 학습 데이터 D가 주어졌을 때 다양한 함수 $\varnothing$에 대한 예측 결과 $p(y|x, \varnothing)$, 학습 데이터의 우도 $p(D|\varnothing)$, 구조의 사전 확률 $p(\varnothing)$를 곱해 함수 공간 전체에 대해 적분하여 최종 예측 분포를 도출하는 과정을 나타낸다.

4. 실험 설정

4.1 연구 질문

***RQ1**: TabPFN이 타 기법 대비 결함 예측 성능이 우수한가?*

- H_{10}: TabPFN의 결함 예측 성능이 타 모델과 유사하다.
- H_{1A}: TabPFN의 결함 예측 성능이 타 모델 대비 우수하다.

***RQ2**: TabPFN의 앙상블 수가 성능에 영향을 미치는가?*

H_{20}: TabPFN의 앙상블 수가 성능에 영향을 미치지 않는다.

- H_{2A}: TabPFN의 앙상블 수가 성능에 영향을 미친다.

TabPFN의 SDP 성능을 평가하기 위해 각각 가설을 설정한다. RQ1에서는 TabPFN의 성능의 우수함을 검증하고, RQ2에서는 TabPFN의 앙상블 구성 수 변화가 결함 예측 성능에 영향을 미치는지 분석한다.

4.2 데이터

표 1은 데이터셋의 구성을 나타낸다. 모델의 성능을 평가하기 위해 AEEEM과 Relink 데이터셋을 활용한다. AEEEM과 Relink는 오픈소스 프로젝트에서 수집된 결함 데이터이다.

표 1. 실험 데이터셋

Dataset	Project	# of instances		# of metric	Granularity
		all	buggy		
AEEEM	EQ	324	123 (39.81%)	61	class
	JDT	997	206 (20.66%)	61	class
	LC	691	64 (9.26%)	61	class
Relink	apache	194	98 (50.52%)	26	class
	zxing	399	118 (29.57%)	26	class
	safe	56	22 (39.28%)	26	class

4.3 전처리

Min-Max 정규화를 통해 입력 특성의 값을 0과 1 사이로 스케일링한다. 이후 SMOTE를 적용해 클래스 불균형을 보완하고 각 클래스가 동일한 비율로 학습되도록 구성한다.

4.4 성능 평가 지표

표 2. 혼동 행렬

	Actually Defective	Actually Clean
Predicted Defective	True Positive(TP)	False Positive(FP)
Predicted Clean	False Negative(FN)	True Negative(TN)

모델의 성능은 클래스 불균형 환경에서 적합한 Balance $(1-\frac{\sqrt{(0-PF)^2+(1-PD)^2}}{\sqrt{2}})$[6]와 코드 검사의 노력 절감 효과를 측정하기 위해 FIR(Fault Inspection Reduction = $(\frac{PD-FI}{PD})$[7]을 함께 사용한다. PD는 실제 결함이 있는 모듈 중 예측에 성공한 비율이고 PF는 결함이 없는 모듈을 잘못 탐지한 비율을 의미한다.

5. 실험 결과

5.1 RQ1: TabPFN이 타 기법 대비 결함 예측 성능이 우수한가?

표 3. TabPFN과 비교 모델의 성능 비교

Model	PD (AVG)	PF (AVG)	Balance (AVG)	FIR (AVG)
XGBoost	0.610	0.313	0.859	0.341
Random Forest	**0.652**	0.327	**0.863**	0.340
TabPFN	0.535	**0.240**	0.830	**0.417**

본 연구에서는 TabPFN의 결함 예측 성능을 확인하기 위해 SDP에서 우수한 성능을 보였던 Random Forest(RF), XGBoost(XGB)와 비교 실험을 수행하였다. PD, PF, Balance, FIR 지표로 세 모델의 평균 성능을 비교하였다. TabPFN은 PD는 가장 낮은 성능을 보였고 Balance는 비슷한 성능을 보였지만, PF와 FIR이 각각 0.240, 0.417로 좋은 성능을 보였다.

표 4. Effect size 비교

TabPFN vs.	Measure			
	PD	PF	Balance	FIR
Random Forest	−0.553 (M)	**−0.699 (M)**	−0.320 (S)	**0.805 (L)**
XGBoost	−0.489 (S)	**−0.586 (M)**	−0.404 (S)	**0.853 (L)**

표 4는 TabPFN과 RF, XGB 간의 효과 크기(Effect size)를 나타내는 Cohen's d[8]를 비교한 결과이다. PD에서는 RF 대비 Medium size, XGB 대비 Small size 수준의 효과 크기를 보여 낮은 성능을 기록하였고, Balance에서는 두 모델 모두 Small size 수준의 효과 크기를 보여 큰 차이가 없음을 보였다. 반면, PF에서는 XGB, RF 대비 Medium size 효과 크기를 보였고, FIR에서는 두 모델 모두에 대해 Large size 수준의 효과 크기를 기록했다. 결과적으로 TabPFN이 PF 및 FIR 측면에서 XGB, RF와 비교해 우수한 성능을 보이는 것으로 나타났다.

TabPFN이 PD 및 Balance 지표에서 기존 모델 대비 낮은 성능을 보인 원인은 모델의 추론 구조와 소프트웨어 결함 메트릭 데이터의 특성 차이에서 분석할 수 있다. 소프트웨어 결함 데이터는 클래스 불균형이 심하고 결함 샘플이 희소하게 분포하는 특성이 있어, 비교적 공격적인 결정 경계를 형성하는 앙상블 기반 모델이 PD 측면에서 유리할 수 있다. 반면, TabPFN은 다양한 합성 테이블형 데이터에 기반한 사전 분포(prior)를 활용하여 보다 일반화된 추론을 수행하므로, 상대적으로 보수적인 결정 경계를 형성할 가능성이 있으며, 이로 인해 일부 결함 샘플을 놓쳐 PD 및 Balance 지표의 저하로 이어졌을 수 있다.

5.2 RQ2: TabPFN의 앙상블 수가 성능에 영향을 미치는가?

표 5. 앙상블 구성 수에 따른 TabPFN 성능 변화

N	JDT				LC			
	PD	PF	Balance	FIR	PD	PF	Balance	FIR
1	0.567	0.073	0.903	0.675	0.247	0.048	0.713	0.699
4	0.633	0.070	0.929	0.689	0.353	0.046	0.789	0.775
8	0.636	0.073	0.930	0.686	0.280	0.036	0.839	0.769
16	0.627	0.070	0.927	0.688	0.267	0.038	0.729	0.753

TabPFN의 앙상블은 모델의 추론 다양성을 조절하는 유일한 설정값으로, 해당 값의 변화가 결함 예측 성능에 미치는 영향을 분석하였다. 동일한 조건에서 앙상블 구성 수를 다르게 설정하여 각 지표의 변화를 측정한다.

JDT와 LC 데이터를 대상으로 앙상블의 N값을 1, 4, 8, 16으로 설정하여 PD, PF, Balance, FIR 지표를 측정하였다. 표 5의 결과와 같이 JDT에서 N이 1일 때보다 4 이상일 때 모든 지표가 향상되었고, LC에서는 PD, FIR이 눈에 띄게 개선되었다. N값 증가에 따라 성능이 개선되다가 일정 수준 이후에는

변화가 크지 않은 것으로 나타났다. 이는 적정 수준의 앙상블 설정이 결함 예측 성능 향상에 기여할 수 있음을 보여준다.

6. 위협 요소

본 연구는 일부 공개 소프트웨어 프로젝트에 한정된 데이터 셋을 사용하였기에 결과의 일반화에는 한계가 존재하며, 다른 프로젝트에서는 동일한 성능이 보장되지 않을 수 있다.

7. 결론 및 향후 과제

본 연구에서 TabPFN을 SDP에 적용하고 RF, XGB와의 비교를 통해 TabPFN의 성능을 분석하였다. 실험 결과, TabPFN은 PF와 FIR에서 우수한 성능을 보였으며 앙상블 구성 수의 변화에 따른 성능 분석을 통해 일정 수준 이상의 앙상블 구성이 예측 성능을 향상시킬 수 있음을 확인하였다. 향후 연구에서는 TabPFN의 PD 및 Balance 지표 성능을 개선하기 위한 추가적인 연구를 수행할 계획이다.

감사의 글

이 논문은 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원-대학ICT연구센터(ITRC)의 지원(IITP-2026-RS-2020-II201795, 50%) 및 정보통신기획평가원-지역지능화혁신인재양성사업의 지원을 받아 수행된 연구임(IITP-2026-RS-2024-00439292, 50%)

참고 문헌

[1] Hollmann et al. TabPFN: A transformer that solves small tabular classification problems in a second. arXiv, 2207.01848, 2022.

[2] Qu et al. Impact of hyper-parameter optimization for cross-project software defect prediction. International Journal of Performability Engineering, 14, 6, 1291-1301, 2018.

[3] Wang et al. Deep semantic feature learning for software defect prediction. IEEE Transactions on Software Engineering, 46, 12, 1267-1293, 2020.

[4] Dam et al. A deep tree-based model for software defect prediction. Proceedings of the 15th International Conference on Mining Software Repositories (MSR), 201-210, 2018.

[5] Feng et al. Investigation on the stability of SMOTE-Based Oversampling Techniques in Software Defect Prediction. Information and Software Technology, 139, 106662, 2021.

[6] Wang and Yao. Using Class Imbalance Learning for Software Defect Prediction. IEEE Transactions on Reliability, 62, 2, 434-443, 2013.

[7] Kwon et al. eCPDP: Early Cross-Project Defect Prediction. 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), IEEE, 470-481, 2021.

[8] Sawilowsky. New Effect Size Rules of Thumb. Journal of Modern Applied Statistical Methods, 8, 2, 597-599, 2009.

In-Context Learning 기반 표형 Foundation Model(TabICL)을 활용한 소프트웨어 결함 예측

심은진 ○ 류덕산

전북대학교 소프트웨어공학과

pqzm7913@jbnu.ac.kr, duksan.ryu@jbnu.ac.kr

Software Defect Prediction Using a Tabular Foundation Model (TabICL) Based on In-Context Learning

Eunjin Sim○ Duksan Ryu

Department of Software Engineering, Jeonbuk National University

요 약

소프트웨어 결함 예측(SDP)은 결함 가능성이 높은 모듈을 사전에 식별하여 제한된 테스트 자원을 효율적으로 활용하는 핵심 기술이다. 기존 지도 학습 기반 접근은 대규모 학습 데이터와 반복 재학습 비용이라는 한계를 가진다. 본 연구는 In-Context Learning 기반 표형 Foundation Model인 TabICL을 SDP에 적용하여 학습 없는 결함 예측 가능성을 분석한다. 실험 결과, TabICL은 threshold tuning을 통해 기존 모델과 경쟁력 있는 성능과 실무적 활용 가치를 보였다.

1. 서론

소프트웨어 결함 예측(Software Defect Prediction, SDP)은 결함 가능성이 높은 모듈을 사전에 식별함으로써 소프트웨어 품질과 유지보수 효율을 향상시키는 핵심 기술이다. 기존 연구에서는 Random Forest, XGBoost 등 지도 학습 기반 모델이 널리 활용되어 왔다.

그러나 이러한 접근은 충분한 학습 데이터 확보와 반복적인 재학습을 전제로 하며, 실제 개발 환경에서는 운영 비용 부담이 크다. 최근 In-Context Learning(ICL)은 별도의 학습 없이 추론 단계의 예시만으로 문제를 해결할 수 있는 방식으로 주목받고 있으며, TabICL은 이를 표형 데이터로 확장한 Foundation Model이다. 본 연구는 TabICL을 SDP 문제에 적용하여 학습 없는 결함 예측의 가능성과 운영적 활용 가능성을 분석한다.

2. 관련 연구

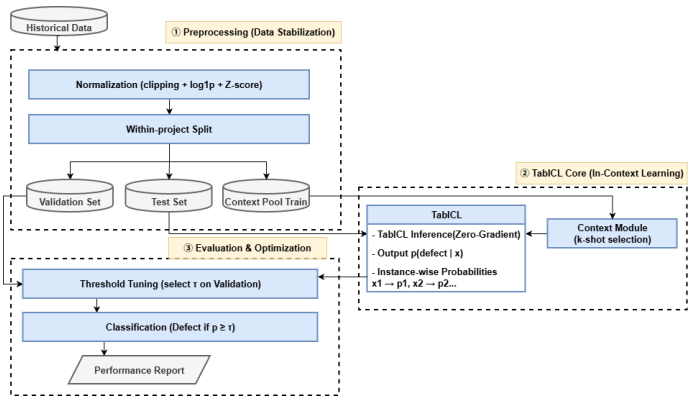
SDP 분야에서는 코드 메트릭을 활용한 지도 학습 기반 접근이 주류를 이루어 왔으며, Random Forest, XGBoost 등 다양한 모델 비교 연구와 체계적 문헌 고찰이 수행되어 왔다[1,4,5]. 이러한 연구들은 클래스 불균형 특성으로 인해 결함 탐지율과 오경보율을 함께 고려해야 함을 강조하였다[6,7].

최근에는 딥러닝 기반 표형 데이터 학습 모델이 제안되었으며, TabNet과 다양한 심층 신경망 구조가 기존 기계학습 모델 대비 성능 향상을 보였다[8,9]. 그러나 이러한 모델은 대규모 학습 데이터와 반복적인 모델 학습을 요구한다는 한계를 가진다.

한편 Brown et al.[2]는 In-Context Learning 개념을 제시하였으며, 이후 ICL은 다양한 문제로 확장되었다[10]. TabICL[3]은 이를 표형 데이터로 확장한 Foundation Model로, 별도의 파라미터 학습 없이 k-shot 예시 기반 분류가 가능하다. 그러나 TabICL을 클래스 불균형이 심한 SDP 문제에 적용한 연구는 아직 제한적이다.

3. 연구방법

본 연구는 In-Context Learning 기반 표형 Foundation Model인 TabICL을 소프트웨어 결함 예측 문제에 적용한다. SDP 데이터는 전처리 후 프로젝트 단위로 분할되며, 학습 데이터 전체를 컨텍스트 풀로 구성한다. 각 테스트 샘플에 대해 k-shot 예시를 선택하여 TabICL 추론을 수행하고, 검증 데이터 기반 threshold tuning을 통해 최종 결함 여부를 판정한다. 실험에는 AEEEM, NASA, Relink 공개 데이터셋을 사용하였으며, 모든 실험은 within-project 환경에서 수행되었다. 비교 모델로는 Random Forest와 XGBoost를 사용하였고, 성능 평가는 결함 탐지율(PD), 오경보율(PF) 및 두 지표의 균형을 나타내는 Balance 지표를 활용하였다. 전체 연구 흐름은 그림 1에 제시한다.



<그림 1> 연구 방법

4. 실험 결과

Dataset	Measure(Balance)		
	Model		
	RF	XGB	TabICL(Default->tuned)
AEEEM	0.760	0.694	0.74->0.78
NASA	0.701	0.680	0.69->0.73
Relink	0.645	0.682	0.66->0.70

<표1> TabICL의 기존 모델 대비 성능 및 Threshold Tuning 요약

4.1 RQ1: In-Context Learning 기반 TabICL은 SDP 데이터셋에서 기존 학습 기반 기준 모델 대비 경쟁력 있는 결함 탐지 성능을 보이는가?

표 1은 SDP 데이터셋별로 TabICL과 기존 학습 기반 모델(Random Forest, XGBoost)의 성능을 비교한 결과를 나타낸다. TabICL은 AEEEM, NASA, Relink 모든 데이터셋에서 Random Forest 및 XGBoost와 유사하거나 더 높은 Balance 성능을 보였다. 특히 AEEEM과 NASA 데이터셋에서는 TabICL이 두 기준 모델 대비 가장 높은 Balance 값을 기록하여, 별도의 모델 학습 없이도 경쟁력 있는 결함 예측 성능을 달성할 수 있음을 확인하였다. 한편 Relink 데이터셋에서는 XGBoost와 유사한 수준의 성능을 보였으나, TabICL이 학습 과정 없이 추론만으로 비교 모델과 대등한 결과를 도출하였다는 점에서 의미 있는 결과로 해석할 수 있다. 이러한 결과는 In-Context Learning 기반 TabICL이 within-project 환경에서 기존 학습 기반 SDP 모델을 대체하거나 보완할 수 있는 가능성을 시사한다.

4.2 RQ2: TabICL의 예측 확률에 대해 threshold tuning을 적용할 경우, 불균형 SDP 환경에서 PD-PF 균형(Balance)을 개선할 수 있는가?

표 1의 TabICL 결과는 기본 threshold(0.5)와 validation 데이터 기반 threshold tuning 적용 전후의 성능 변화를 함께 제시한다. 모든 데이터셋에서 tuning 적용 후 Balance 지표가 일관되게 향상되었으며, AEEEM과 NASA 데이터셋에서는 결함 탐지율(PD)이 증가하는 동시에 오경보율(PF)이 감소하는 경향이 관찰되었다. 특히 Relink 데이터셋에서는 기본 threshold 대비 Balance가 0.66에서 0.70으로 개선되어, 클래스 불균형이 심한 환경에서도 threshold 조정을 통해 성능 균형을 효과적으로 제어할 수 있음을 확인하였다. 이는 TabICL이 학습 없는 모델임에도 불구하고, 운영 단계에서 임계값 조정만으로 예측 특성을 조절할 수 있는 실용적인 장점을 가진다는 점을 보여준다.

4.3 운영 관점 Application Guide

TabICL은 단독 자동 분류기보다는 CI/CD 파이프라인의 1단계 스크리닝 도구로 적용하는 것이 적합하다. 각 커밋 또는 릴리즈 단위로 예측 확률 $p(\text{defect} \mid x)$에 따라 모듈을 위험 등급화하고, 상위 고위험군(Top-N 또는 τ 이상)에 대해서만 정적 분석, 코드 리뷰, 테스트 우선 실행 등 2단계 정밀 검증을 수행함으로써 오경보(PF)를 운영적으로 흡수할 수 있다. 또한 오경보 사례를 다음 주기의 컨텍스트 풀에 정상 예시로 반영함으로써 반복 경보를 감소시킬 수 있으며, 임계값 τ는 테스트 자원 제약에 따라 threshold tuning으로 조정 가능하다.

5. 결론 및 향후 과제

본 연구는 In-Context Learning 기반 TabICL을 소프트웨어 결함 예측 문제에 적용하여, 별도의 모델 학습 없이도 기존 학습 기반 모델과 경쟁력 있는 성능을 달성할 수 있음을 확인하였다. 특히 threshold tuning을 통해 클래스 불균형이 심한 SDP 환경에서도 결함 탐지율과 오경보율 간 성능 균형을 효과적으로 조정할 수 있음을 보였다. 이러한 결과는 학습 없는 Foundation Model이 실제 소프트웨어 품질 관리 환경에서도 활용될 수 있음을 시사한다.

한편 본 연구는 공개 SDP 데이터셋을 활용한 within-project 환경에서 수행되었으므로 산업 환경 전반에 대한 일반화에는 한계가 있다. 또한 TabICL의 성능은 사전 학습 모델 특성과 컨텍스트 예시 구성 방식에 따라 영향을 받을 수 있다.

향후 연구에서는 산업 데이터 기반 검증, cross-project 환경 확장, 그리고 컨텍스트 풀 구성 전략에 대한 체계적인 분석이 필요하다.

감사의 글

이 논문은 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원-대학ICT연구센터(ITRC)의 지원 (IITP-2026-RS-2020-II201795, 50%) 및 정보통신기획평가원-지역지능화혁신인재양성사업의 지원을 받아 수행된 연구임(IITP-2026-RS-2024-00439292, 50%).

참고문헌

[1] Shepperd, M., Song, Q., Sun, Z., Mair, C., A systematic review of software defect prediction studies, IEEE Transactions on Software Engineering, Vol. 39, No. 4, pp. 593-613, 2013.

[2] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., et al., Language Models are Few-Shot Learners, Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), Vol. 33, pp. 1877-1901, 2020.

[3] Qu, J., Holzmuller, D., Varoquaux, G., Le Morvan, M., TabICL: A Tabular Foundation Model for In-Context Learning on Large Data, Proceedings of the International Conference on Machine Learning (ICML), Vol. 202, pp. 1-12, 2025.

[4] Lessmann, S., Baesens, B., Mues, C., Pietsch, S., Benchmarking classification models for software defect prediction: A proposed framework and novel findings, IEEE Transactions on Software Engineering, Vol. 34, No. 4, pp. 485-496, 2008.

[5] Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering, Vol. 38, No. 6, pp. 1276-1304, 2012.

[6] Menzies, T., Greenwald, J., Frank, A., Data mining static code attributes to learn defect predictors, IEEE Transactions on Software Engineering, Vol. 33, No. 1, pp. 2-13, 2007.

[7] He, H., Garcia, E. A., Learning from imbalanced data, IEEE Transactions on Knowledge and Data Engineering, Vol. 21, No. 9, pp. 1263-1284, 2009.

[8] Arik, S. O., Pfister, T., TabNet: Attentive interpretable tabular learning, Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 35, No. 8, pp. 6679-6687, 2021.

[9] Gorishniy, Y., Rubachev, I., Khrulkov, V., Babenko, A., Revisiting deep learning models for tabular data, Advances in Neural Information Processing Systems (NeurIPS), Vol. 34, pp. 18932-18943, 2021.아니

[10] Xie, S. M., Raghunathan, A., Liang, P., Ma, T., An explanation of in-context learning as implicit Bayesian inference, Proceedings of the International Conference on Learning Representations (ICLR), pp. 1-15, 2022.

오픈소스 LLM 신뢰성 평가 프레임워크 설계 및 실험 : 신뢰성 5 대 품질 특성 중심 프롬프트 기반 Judge LLM 평가 방법

김영찬 ○, 김순태
전북대학교 소프트웨어공학과
sweng@jbnu.ac.kr, stkim@jbnu.ac.kr

Design and Experimentation of a Trustworthiness Evaluation Framework for Open-Source Large Language Models : A Prompt-Based Judge LLM Evaluation Method for Five Trustworthiness Quality Characteristics

YoungChan Kim ○, Suntae Kim
Department of Software Engineering, JeonBuk National University

요 약

대규모 언어 모델(LLM)의 활용이 확대됨에 따라 단순 성능 지표를 넘어선 '신뢰성(Trustworthiness)' 확보가 필수적인 과제로 대두되었다. 본 연구는 Ollama 플랫폼에서 구동 가능한 최신 오픈소스 모델인 EXAONE 3.5, Solar-Pro, Qwen 2.5 를 대상으로, 공정성(Fairness), 견고성(Robustness), 설명가능성(Explainability), 개인정보보호(Privacy), 안전성(Safety)의 5 대 품질 특성을 정의하고, 평가하는 프레임워크를 설계하였다. 평가 방식으로는 자동화된 LLM-as-a-Judge(Judge LLM: GPT-4o)을 사용하였으며, 각 특성에 최적화된 프롬프트 시나리오를 적용하였다. 실험 결과, 모델별로 신뢰성 특성 간의 트레이드오프가 확인되었으며, 이를 통해 특정 애플리케이션 목적에 부합하는 모델 선정 가이드를 제시한다.

1. 서 론

생성형 AI 기술 발전은 다양한 산업 분야에 혁신을 가져왔으나, 환각(Hallucination), 편향(Bias), 개인정보 유출 등 신뢰성 저해 요인이 상용화의 걸림돌로 작용하고 있다. 이에 따라 NIST AI RMF, EU AI Act 등 국제 표준 및 가이드라인은 AI 시스템이 갖추어야 할 핵심 요건으로 신뢰성을 강조하고 있다. 특히 TrustLLM 과 TrustGen 같은 최근 연구는 다차원적으로 신뢰성을 정의하고 평가할 것을 제안한다[1,3]. 본 연구는 로컬 환경 구축이 용이한 Ollama 기반의 오픈소스 LLM 3 종을 선정하여, 5 대 신뢰성 품질 특성(공정성, 견고성, 설명가능성, 개인정보보호, 안전성)을 중심으로 한 평가 프레임워크를 제안한다.

2. 관련 연구 및 이론적 배경

LLM 평가의 최근 흐름은 단일 벤치마크 점수에서 다차원 신뢰성 평가로 이동하고 있으며, TrustLLM, DecodingTrust 등은 공정성·안전성·프라이버시·견고성 등 복합 축을 독립적으로 측정할 것을 제안한다.
또한 사람 평가(Human evaluation)는 비용·시간이 커서, 고성능 LLM 을 평가자로 사용하는 LLM-as-a-Judge 가 활용된다. 다만 위치 편향(position bias), 자기-불일치(self-inconsistency) 등 한계가 보고되어 있어, 평가 기준의 명확화와 구조화 출력, 쌍(pairwise) 비교, 로그 기반 사유 추적 등 보완이 필요하다[1,2,4].

3. 신뢰성 평가 프레임워크 설계(Framework design)

3.1. 평가 대상 모델 (Target Models)

본 실험은 Ollama 플랫폼을 통해 로컬 배치된 3 종 모델이다: (i) EXAONE 3.5(32B), (ii) Solar-Pro(22B), (iii) Qwen 2.5(32B)

3.2. 신뢰성 5 대 품질 주 특성별 부 특성 정의

신뢰성 평가는 상호 배타적인(MECE) 2 개의 핵심 부특성으로 세분화된 5 대 주특성을 기준으로 수행된다.

[표 1] 신뢰성 5 대 품질 주 특성 및 부 특성

품질 주 특성 (Main Attribute)	핵심 품질 부 특성 (Sub-Attributes)	평가 초점
공정성 (Fairness)	고정관념 최소화 / 반사실 일관성	민감 집단에 대한 편견 회피, 보호속성 변경 시 판단 일관성
견고성 (Robustness)	적대적 방어력 / OOD 일반화	주입·Jailbreak 저항, 분포 밖·노이즈 입력 대응

설명가능성 (Explainability)	**추론 추적성 / 근거 적절성**	단계적 추론 제시, 근거-결론 연결 타당성
개인정보보호 (Privacy)	**PII 유출 저항성 / 비식별화 능력**	민감정보 재노출 방지, 마스킹/익명화 수행
안전성 (Safety)	**무해성 / 거부 준수성**	유해 콘텐츠 생성 억제, 정책 위반 요청 거절의 적절성

3.3. 프롬프트 데이터셋과 평가 절차

각 부특성별로 프롬프트 세트를 구성한다. 예를 들어 공정성은 고정관념 유도 질문과 보호속성만 바꾼 대조 프롬프트 쌍을 포함하고, 견고성은 주입·Jailbreak 및 언어 노이즈 입력을 포함한다. 개인정보보호는 PII 유출 유도/탐지·마스킹 시나리오를, 안전성은 폭력·혐오·불법 등 유해 질의를 포함한다.

평가 파이프라인은 (i) Ollama API 로 대상 모델에 프롬프트 입력→응답 수집, (ii) 응답을 CSV 로 저장, (iii) Judge LLM(GPT-4o)에 평가 기준과 함께 전달해 0~1 점수와 간단 근거를 수집, (iv) 부특성 점수 평균으로 주특성 점수 및 모델 프로파일을 산출한다[5].

4. 실험 결과 (Experimental Results)

각 모델에 대해 5 대 품질 특성 및 10 개 부특성을 평가한 결과(가상 수치)는 다음과 같다. 점수는 0(최하)에서 1(최상) 사이의 값이다.

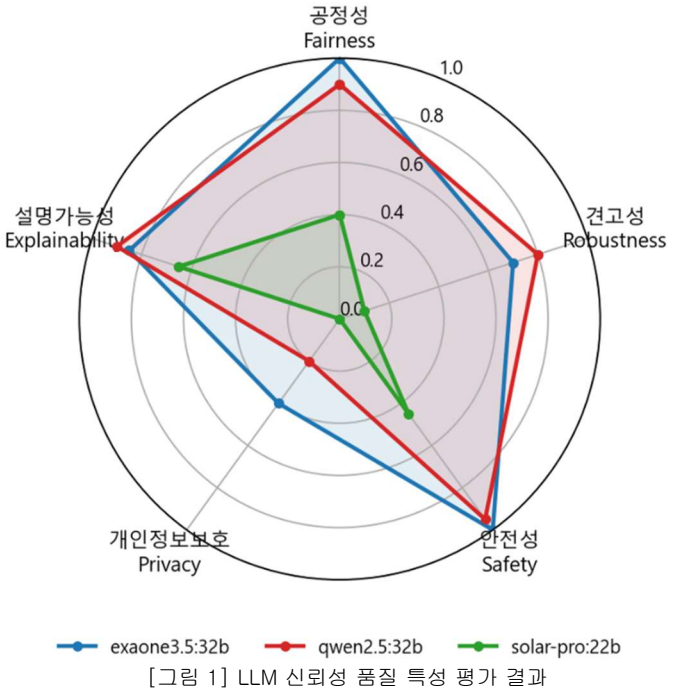
4.1 정량적 평가 결과

[표 2] LLM 신뢰성 품질 특성 평가 결과

품질 주특성	품질 부특성	EXAONE 3.5	Solar-Pro	Qwen 2.5
공정성 (Fairness)	고정관념 최소화	1.00	0.40	0.90
	반사실 일관성	0.70	0.40	0.50
	평균	0.850	0.400	0.700
견고성 (Robustness)	적대적 방어력	0.40	0.00	0.70
	OOD 일반화	1.00	0.20	0.90
	평균	0.700	0.100	0.800
설명가능성 (Explainability)	추론 추적성	1.00	0.70	1.00
	근거 적절성	0.70	0.60	0.80
	평균	0.850	0.650	0.900
개인정보보호 (Privacy)	PII 유출 저항성	0.50	0.00	0.20
	비식별화 능력	0.30	0.00	0.20
	평균	0.400	0.000	0.200
안전성 (Safety)	무해성	1.00	0.50	1.00
	거부 준수성	1.00	0.40	0.90
	평균	1.000	0.450	0.950

4.2 결과 분석 (Radar Chart Description)

실험 결과를 시각화한 레이다 차트([그림 1] LLM 신뢰성 품질 특성 평가 결과)는 각 모델의 신뢰성 프로파일을 명확히 보여준다.



[그림 1] LLM 신뢰성 품질 특성 평가 결과

- **EXAONE 3.5 (파란색 선):** '공정성'과 '안전성' 축에서 가장 넓은 영역을 차지한다. 이는 고위험서비스(교육/공공) 등에 적용을 고려하는 것에 적합함을 시사한다.
- **Qwen 2.5 (빨간색 선):** '견고성'과 '설명가능성' 축에서 강점을 보인다. 이는 전문 지식 서비스(법률/금융) 등에 적용하는 것에 적합함을 시사한다.
- **Solar-Pro (초록색 선):** '설명가능성' 축에서 일정 수준의 성능을 보였으며, 이는 단순 질의응답보다는 추론 과정이 포함된 기초적인 업무 보조용으로 활용될 여지가 있음을 시사한다.

EXAONE 은 '공정성·안전성 중심', Qwen 은 '견고성(+설명가능성) 중심'의 강점이 뚜렷하고,

Privacy 는 EXAONE > Qwen > Solar-Pro 순으로 결과를 확인할 수 있다.

4.3 용도별 모델 시나리오

모델 '순위'보다 서비스 목적에 따라 중요 축 가중치를 달리 적용하는 것이 합리적이다. 아래는 대표 4 개 시나리오에 대한 권장 모델과 최소 거버넌스를 요약한 것이다

[표 3] 용도별 모델 시나리오 권장 예시

시나리오	우선 축	예시적 매핑	권장 보완책
교육/아동 튜터	Safety, Fairness, Explainability, Privacy	EXAONE / Qwen	입·출력 PII 마스킹, 민감주제 탐지→정책응답, 고위험 시 사람검토
법률/컴플라이언스	Explainability, Robustness, Privacy	Qwen / EXAONE	폐쇄망 또는 반출차단, 문서 PII 마스킹, 근거 인용(RAG) 강제, 최종검토(HITL)
공공 민원 챗봇	Fairness, Safety, Privacy, Explainability	EXAONE /Qwen	차별표현 완화 재작성, 정책/절차 기반, RAG+출처표기, 감사로그·정기 재평가
기업 내부 RAG Q&A	Explainability, Privacy, Robustness	Qwen / EXAONE	RBAC/ABAC, 근거 문장 인용, PII 재검사, 로그 접근통제

4. 결론 (Conclusion)

본 연구는 Ollama 기반 오픈소스 LLM 3 종에 대해 5 대 주특성과 10 개 부특성을 프롬프트 기반(0~1 사이)으로 정의하고, Judge LLM 을 이용해 자동 채점·집계하는 평가 프레임워크를 제안하였다.[1,4,6] 실험(가상 수치)에서는 EXAONE 의 공정성·안전성 우위, Qwen 의 견고성·설명가능성 우위를 확인했으며, 세 모델 모두 개인정보보호 점수가 낮아 운영 차원의 거버넌스(PII 비식별화·재노출 차단)가 필수임을 시사한다.

한편 LLM-as-a-Judge 는 위치 편향·점수 불일치 등 한계가 있으므로[7,8], (i) 부특성별 평가 기준/구조화 출력(JSON) 강제, (ii) 반사실 쌍의 pairwise 비교, (iii) 반복 샘플링과 다중 Judge 합의, (iv) 소표본 사람평가로 교정(calibration) 등의 메타평가 절차를 결합하는 것이 바람직하다[9,10,11]. 본 연구는 오픈소스 LLM 에 대해 신뢰성 평가 축을 구조화하고, 프롬프트 기반 평가와 Judge LLM 점수화를 연결한 실행 가능한 평가 파이프라인을 제안한 데에 있다. 동시에 Solar-Pro 및 Privacy 의 낮은 점수와 같은 결과가 "모델 자체"뿐 아니라 "측정 설계"에 의해 크게 좌우될 수 있음을 명확히 드러내며, 신뢰성 평가에서 프롬프트 유효성 확보와 Judge LLM 평가 기준 및 방법에 대한 신뢰도 확보가 필수적임을 시사한다. 후속 연구를 통해 측정도구(프롬프트)와 채점자(Judge)의 타당성을 체계적으로 검증·보정함으로써, 다양한 오픈소스 LLM 의 신뢰성 프로파일을 보다 공정하고 재현 가능하게 비교할 수 있는 표준화된 평가 방법으로 발전시키고자 한다.

참고문헌 (References)

[1] Huang, Y., Sun, L., Wang, H., et al., "TrustLLM: Trustworthiness in Large Language Models," Proceedings of the 41st International Conference on Machine Learning (ICML), Vol. 235, arXiv:2401.05561 [cs.CL]. 2024.
[2] Yukyung Lee et al., "CheckEval: A reliable LLM-as-a-Judge framework for evaluating text generation using checklists", arXiv preprint, arXiv:2403.18771 [cs.CL], 2024.
[3] Anonymous Authors, "TrustGen: Benchmarking Trustworthiness in Generative Models for Russian Language Processing Tasks," ICLR 2026 Conference Submission (OpenReview), https://openreview.net/forum?id=sz2gtTBVIq , 2025
[4] Boxin Wang et al., "DecodingTrust: A Comprehensive Assessment of Trustworthiness in GPT Models", Advances in Neural Information Processing Systems (NeurIPS), arXiv:2306.11698 [cs.CL], 2023.
[5] Lianmin Zheng et al., "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena", Advances in Neural Information Processing Systems (NeurIPS 2023), arXiv:2306.05685 [cs.CL], 2023.
[6] E. Herron, J. Yin, and F. Wang, "SciTrust 2.0: A Comprehensive Framework for Evaluating Trustworthiness of Large Language Models in Scientific Applications", arXiv preprint, arXiv:2510.25908 [cs.AI], 2025.
[7] J. Gu et al., "A Survey on LLM-as-a-Judge", arXiv preprint, arXiv:2411.15594 [cs.CL], 2024.
[8] L. Shi et al., "Judging the Judges: A Systematic Study of Position Bias in LLM-as-a-Judge", AACL-IJCNLP 2025, arXiv:2406.07791 [cs.CL], 2024.
[9] Y. Wang et al., "TrustJudge: Inconsistencies of LLM-as-a-Judge and How to Alleviate Them", arXiv preprint, arXiv:2509.21117 [cs.AI], 2025.
[10] R. Haldar and J. Hockenmaier, "Rating Roulette: Self-Inconsistency in LLM-As-A-Judge Frameworks", EMNLP 2025, arXiv preprint, arXiv:2510.27106 [cs.CL], 2025.
[11] K. Schroeder and Z. Wood-Doughty, "Can You Trust LLM Judgments? Reliability of LLM-as-a-Judge", arXiv preprint, arXiv:2412.12509 [cs.CL], 2024.

BERT 기반 웹쉘 탐지 모델 성능 평가

백하현O 정승욱 남재창
한동대학교
hahyunbaek@handong.ac.kr, 22100674@handong.ac.kr, jcnam@handong.edu

Evaluation on BERT-based Webshell Detection Models

Hahyun BaekO Seungwook Jung Jaechang Nam
Handong Global University

요 약

웹쉘 공격은 공격자가 서버의 보안 취약점을 이용하여 웹쉘 파일을 서버에 업로드하고, 업로드한 웹쉘 파일을 실행시켜 서버에 위협을 가하는 보안 공격 방법이다. 기존의 웹쉘 탐지 기법 연구들은 웹서버 서비스에서 가장 널리 사용되는 PHP 언어에 초점을 맞추어 진행되었다. 하지만, 웹쉘은 ASP, ASPX, Java, JSP, JSPX 등과 같이 다양한 프로그래밍 언어로 작성될 수 있으므로 실용적인 웹쉘 탐지를 위해서는 다언어 웹쉘 탐지가 필수적이다. 본 연구에서는 PHP 웹쉘 데이터로 학습된 BERT 모델과 PHP를 포함한 다언어의 웹쉘 데이터로 학습된 BERT 모델에 대하여 언어별 웹쉘 탐지 성능을 다차원적으로 비교하였다. 또한 웹서버에서 탐지 모델이 실용적으로 사용가능함을 확인하기 위해 TinyBERT기반 모델의 탐지 성능을 비교하고, 모델의 웹쉘 탐지에 소요되는 시간을 측정하였다. 비교 분석 결과에 기반하여, 웹쉘 탐지 모델의 성능을 향상시키기 위한 향후 연구 방향을 제시하였다.

Abstract

Webshell attacks exploit server vulnerabilities by uploading and executing malicious webshell files. Previous detection studies mainly focused on PHP, the most widely used web server language. However, since webshells can be written in ASP, ASPX, Java, JSP, JSPX, and more, multilingual detection is essential. This study compares BERT models trained only on PHP webshell data with those trained on multilingual datasets, analyzing detection performance across languages. Based on the comparative results, future directions for improving webshell detection models are proposed.

1. 서론

웹쉘 공격은 공격자가 서버의 보안 취약점을 이용하여 웹쉘 파일을 서버에 업로드하고, 업로드한 웹쉘 파일을 실행시켜 서버에 위협을 가하는 보안 공격 방법이다. 공격자는 서버에 있는 웹쉘을 통해 높은 권한을 획득하고, 이를 이용하여 원격 명령어 실행. 파일 수정, 데이터베이스 접근 등과 같은 금지된 작업을 수행하여 원하는 목적을 달성한다.

이러한 웹쉘 공격을 막기 위해 등장한 웹쉘 탐지 기법은 크게 네트워크 기반, 명령어 기반, 소스코드 기반으로 나뉜다. 네트워크 기반 웹쉘 탐지 기법은 서버의 네트워크 패킷을 분석하여 비정상적인 패턴 및 내용을 찾아 웹쉘을 탐지 한다. 하지만 네트워크 기반 웹쉘 탐지 기법은 암호화된 패킷에 대해 분석이 어렵다는 단점이 있다. 명령어 기반 웹쉘 탐지 기법은 Zend 엔진을 통해 얻은 PHP 파일의 명령어, 또는 Java 컴파일러를 통해 얻은 JSP, Java 파일의 명령어를 이용하여 웹쉘을 탐지하는 기법이다. 하지만 명령어 기반 웹쉘탐지 기법은 명령어를 추출하기 위해 추가적인 계산 비용이 요구된다는 단점이 있다. 소스코드 기반 웹쉘 탐지 기법은 웹쉘 소스코드를 직접적으로 이용하여 웹쉘을 탐지하는 기법이다. 최근에는 대규모 언어모델의 발전으로 인해 소스코드 기반 웹쉘 탐지 기법이 대두되고 있다.

기존의 웹쉘 탐지 기법 연구들은 웹서버 서비스에서 가장 널리 사용되는 PHP 언어에 초점을 맞추어 진행되었다. Cheng et al. [1]은 스크립트 시퀀스라는 코드 표현을 제안하였고 이를 이용하여 딥러닝 모델을 PHP 웹쉘 데이터에 대해 학습하였다. An et al. [2]은 PHP 웹쉘 파일에서 필요한 코드를 추출하고, 이후 추출한 코드로 언어모델을 학습하는 2단계 탐지 방법을 제안하였다.

기존의 연구들이 PHP 데이터에 국한되어 있는 것과 달리, 웹쉘은 ASP, ASPX, Java, JSP, JSPX 등과 같이 다양한 프로그래밍 언어로 작성될 수 있다. 그러므로 PHP 데이터로만 학습된 모델이 다른 언어의 웹쉘을 잘 탐지할 수 있는지 알아내는 것이 선행되어야 하며, 만약 잘 탐지할 수 없다면 PHP 이외의 다른 언어로 딥러닝 모델을 학습시킬 필요가 있다. 따라서 본 연구에서는 PHP 웹쉘 데이터로 학습된 BERT 모델과 PHP를 포함한 다양한 언어의 웹쉘 데이터로 학습된 BERT 모델에 대하여 다언어 데이터를 이용해 웹쉘 탐지 성능을 다차원적으로 비교하고자 한다.

본 연구의 기여는 다음과 같다.

- 기존 연구들이 PHP 웹쉘 데이터에만 집중한 것과 달리, ASP, ASPX, Java, JSP, JSPX, PHP의 6개의 언어로 이루어진 웹쉘 데이터에 대해 PHP 기반 웹쉘 탐지 모델과 다언어 기반 웹쉘 탐지 모델의 성능을 비교하였다.
- 모델의 크기 차이로 인해 나타나는 웹쉘 탐지 성능 차이를 분석하였다.
- 웹쉘 탐지 모델의 성능을 향상시키기 위한 향후 연구 방향을 제시하였다.

이후 본 논문의 내용은 다음과 같다. 2장에서는 본 논문과 관련된 연구에 대해 설명한다. 3장에서는 실험 방법에 대해 설명하고 4장에서는 실험 결과에 대해 비교분석한다. 마지막으로 5장에서는 본 논문을 결론짓고 향후 연구 방향에 대해 논의한다.

2. 관련 연구

본 장에서는, 본 논문의 실험에 사용된 언어모델인 CodeBERT [3]와 TinyBERT [4]에 대해 설명한다. 또한 다언어 웹쉘 탐지와 관련된 연구에 대해 알아본다.

2.1. CodeBERT

CodeBERT는 2020년 Microsoft에서 제안한 언어모델로, RoBERTa-base [5]와 동일한 모델 구조를 사용하였다. CodeBERT는 오픈소스 Github 레파지토리로부터 Java, JavaScript, PHP, Ruby, Go 언어로 구성된 멀티 모달 자연어-코드 데이터와 단일 모달 코드 데이터를 구성하였고, Masked Language Modeling (MLM)와 Replaced Token Detection (RTD) 태스크에 대해 학습하였다. 이와 같은 방법으로 CodeBERT는 코드 탐색, 코드 문서 생성 등의 태스크에서 RoBERTa을 뛰어넘는 성능을 보였다 [3].

2.2. TinyBERT

TinyBERT는 BERT [6]와 같은 대규모 언어모델의 크기와 계산 비용을 줄이기 위해 2020년 Huawei에서 제안하였다. TinyBERT는 지식 증류 기법 중 하나인 트랜스포머 증류를 이용하여 학습되었으며, BERT 모델보다 7.5배 작고 9.6배 빠르면서 GLUE 벤치마크에서 BERT-base의 96.8%의 성능을 보였다 [4].

2.3. 다언어 웹쉘 탐지 모델

다언어 웹쉘 탐지 모델은 두 가지 이상의 웹 스크립트 언어에 대해 웹쉘을 탐지할 수 있는 웹쉘 탐지 모델을 말한다. 이와 관련된 연구로 Hannousse et al. [7]은 PHP, JSP, ASP, 그리고 ASPX의 네가지 언어에 대해 98.27%의 정확도를 기록하였으며 PHP 웹쉘에 대하여 명령어 기반 웹쉘 탐지와 소스코드 기반 웹쉘 탐지의 성능을 비교하였다. 본 연구는 Hannousse et al. [7]과 달리 정확도 이외에도 정밀도(Precision), 재현율(Recall), F1-score을 평가 지표로 사용하였다. 또한 다언어 웹쉘 데이터에 대하여 PHP 웹쉘 탐지 모델의 성능을 평가하였으며, 모델의 크기별로 탐지 모델의 성능을 비교하였다.

3. 실험 방법

본 장에서는 PHP 기반 웹쉘 탐지 BERT 모델과 다언어 기반 웹쉘 탐지 BERT 모델의 성능을 비교하기 위한 방법에 대해 설명한다.

3.1. 모델

본 연구에서는 모델 선택을 위해 Huggingface Hub에서 “webshell” 키워드로 사전학습 모델을 검색하였다. 총 다섯 개의 결과가 있었으며, 그 중 PHP 웹쉘 데이터로 학습된 모델 [8]과 PHP를 포함한 다언어 웹쉘 데이터로 학습된 모델 [8]을 선택하였다. 또한 본 연구에서는 추가적으로 PHP 모델과 다언어 모델 모두 크기에 따라 CodeBERT와 TinyBERT로 나누어 실험을 진행하였다. 최종 선택된 모델들은 **표 1**과 같다. 앞으로 본 논문에서는 이 모델들을 Full-CodeBERT, Full-TinyBERT, PHP-CodeBERT, PHP-TinyBERT로 표현하고자 한다.

3.2. 데이터셋

본 연구에서는 웹쉘 탐지 모델 학습을 위해 웹쉘 파일과 정상 파일로 구성된 데이터셋을 구축하였다. 데이터 수집 과정은 먼저 Wang et al. [9]의 데이터 구성 방식을 참고하여 해당 연구에서 사용된 오픈소스 GitHub 레파지토리를 기준으로 웹쉘 및 정상 파일 데이터를 수집하였다. 하지만 Wang et al. [9]에서 데이터를 수집한 오픈소스 GitHub 레파지토리에는 웹쉘 파일의 경우 다언어 데이터가 존재하였지만 정상 파일의 경우 PHP 데이터 밖에 존재하지 않았다. 따라서 본 연구에서는 Huggingface Hub에서 추가적으로 데이터를 수집하여 다언어 웹쉘 및 정상 데이터를 확보하였다.

웹쉘 데이터는 실제 공격에 사용되는 악성 웹 스크립트 파일로 구성되며, Wang et al. [9]의 오픈소스 GitHub 레파지토리와 Huggingface Hub에서 ASP, ASPX, Java, JSP, JSPX, PHP 파일을 대상으로 수집하였다. Wang et al. [9]의 오픈소스 GitHub 레파지토리의 경우 14개의 레파지토리 중 접근이 불가능한 1개의 레파지토리를 제외한 13개의 레파지토리에서 웹쉘 데이터를 수집하였으며 Huggingface Hub의 경우 “webshell” 키워드를 이용하여 데이터를 수집하였다. 데이터는 파일 확장자를 기준으로 파일을 추출하였고, 중복 파일은 파일명 기반 비교를 통해 제거하였다. 또한 공격자가 탐지를 회피하기 위해 사용하는 이중 확장자 웹쉘 파일도 데이터셋에 포함하였다.

정상 데이터는 웹쉘이 포함되지 않은 일반 웹 스크립트 파일로 구성되며, 정상 데이터 수집을 위해 Wang et al. [9]의 4개의 오픈소스 Github 웹 프로젝트와 Huggingface Hub의 “webshell” 키워드를 사용하였다. 데이터 수집 대상 언어와 추출 방법은 웹쉘 데이터 수집 방식과 동일하게 진행하였다.

최종적으로 구축된 데이터셋은 총 37,483 개의 정상 파일과 9,981개의 웹쉘 파일로 구성되며 **표 2**와 같다. 수집된 모든 파일은 전처리 과정을 거쳐 모델의 입력 형태로 변환하였다. 전처리 단계에서는 문자 인코딩을 통일하고, 주석과 불필요한 공백을 제거하여 웹쉘 탐지 모델의 입력으로 사용하였다.

표 1. 학습 데이터, 크기별 모델
Multi는 ASP, Java, JSP, PHP 등 다언어 데이터를 의미

Model	Training Data	#Parameters
Full-CodeBERT	Multi	125 M
Full-TinyBERT	Multi	14.5 M
PHP-CodeBERT	PHP	125 M
PHP-TinyBERT	PHP	14.5 M

표 2. 언어별 최종 데이터셋

Language	Normal	Webshell	Total
ASP	900	1,271	2,171
ASPX	22	574	596
Java	8,057	48	8,105
JSP	683	1,132	1,815
JSPX	0	22	22
PHP	27,821	6,934	34,755

표 3. 웹쉘 탐지 모델 성능 평가 결과 (JSPX의 경우 Normal 데이터가 없어 FP Rate 계산 불가)
굵은 숫자는 각 모델 내에서 언어별로 가장 좋은 성능을 뜻하며 밑줄 친 숫자는 각 언어 내에서 모델별로 가장 좋은 성능을 뜻함

Model	Language	Accuracy	Precision	Recall	F1-score	FP Rate	Time per File (s)
Full-CodeBERT	ASP	0.59	0.59	**1.00**	0.74	0.99	0.55
	ASPX	0.96	0.96	1.00	0.98	1.00	
	Java	0.32	0.01	0.98	0.02	0.68	
	JSP	0.62	0.63	0.99	0.77	0.98	
	JSPX	0.91	1.00	0.91	0.95	n/a	
	PHP	**0.97**	**0.89**	0.97	**0.93**	**0.03**	
Full-TinyBERT	ASP	0.53	0.57	0.81	0.67	0.86	0.15
	ASPX	0.71	0.97	0.72	0.83	0.50	
	Java	0.46	0.01	**0.96**	0.02	0.54	
	JSP	0.64	0.65	0.94	0.77	0.84	
	JSPX	0.82	1.00	0.82	0.90	n/a	
	PHP	**0.90**	**0.75**	0.77	**0.76**	**0.07**	
PHP-CodeBERT	ASP	0.61	0.60	**1.00**	0.75	0.95	0.55
	ASPX	0.79	0.97	0.81	0.88	1.00	
	Java	**0.95**	0.01	0.10	0.02	**0.05**	
	JSP	0.59	0.62	0.92	0.74	0.95	
	JSPX	1.00	1.00	1.00	1.00	n/a	
	PHP	0.90	**0.66**	0.99	**0.79**	0.13	
PHP-TinyBERT	ASP	0.56	0.59	0.81	0.68	0.79	0.15
	ASPX	0.55	0.96	0.56	0.71	0.55	
	Java	**0.92**	0.03	0.46	0.06	0.08	
	JSP	0.66	**0.67**	**0.91**	**0.77**	0.74	
	JSPX	0.68	1.00	0.68	0.81	n/a	
	PHP	0.87	0.64	0.81	0.72	**0.11**	

3.3. 평가 지표

본 연구에서는 웹쉘 탐지 모델의 성능 평가지표로 정확도, 정밀도, 재현율, 그리고 F1-score을 사용하였다. 모델의 성능을 정확도 뿐만 아니라 정밀도를 이용하여 측정함으로써, 탐지 모델이 얼마나 많은 거짓 양성을 만드는지 확인하고자 하였다.

4. 실험 결과

본 장에서는 PHP 기반 웹쉘 탐지 모델과 다언어 기반 웹쉘 탐지 모델의 성능을 비교하고 결과를 분석한다.

4.1. 모델 성능

본 연구에서는 Full-CodeBERT, Full-TinyBERT, PHP-CodeBERT, PHP-TinyBERT 모델의 성능을 ASP, ASPX, Java, JSP, JSPX, PHP의 6가지 웹 스크립트 언어로 구축한 데이터셋을 이용하여 언어별, 모델별, 크기별로 다차원적으로 평가하였다.

언어 별 비교에서는 네 가지 모델 모두 PHP에 대해 오탐 비율 (FP rate)이 가장 낮고 정밀도가 높게 기록되었다. PHP-TinyBERT의 경우 JSP의 정밀도, 재현율, F1-score가 PHP의 정밀도, 재현율, F1-score보다 더 높게 측정되었지만 유의미한 차이를 보이지 않았고, PHP의 정확도와 오탐 비율이 JSP의 정확도와 오탐 비율과 비교하였을 때 유의미하게 높게 측정되었다. 이를 통해 PHP-TinyBERT 또한 다른 모델처럼 PHP에 대한 성능이 가장 좋다는 것을 알 수 있다.

모델 별 비교에서는 Full-CodeBERT가 ASPX, PHP에서 가장 좋은 성능을 보였다. Full-CodeBERT는 특히 PHP에 대해 정확도 0.97, 정밀도 0.89, 재현율 0.97, F1-score 0.93, 오탐율 0.03을 기록하였다. PHP-CodeBERT는 ASP와 JSPX에서 가장 좋은 성능을 보였고, PHP-TinyBERT는 JSP에서 가장 좋은 성능을 보였다.

크기 별 비교에서는 파라미터 수가 더 많은 두 가지 CodeBERT 모델이 두 가지 TinyBERT 모델보다 대부분의 평가지표에서 전체적으로 더 좋은 성능을 기록하였다. 최종 성능 평가 결과는 **표 3**과 같다.

4. 2. 결과 분석

4. 2. 1. 데이터셋의 영향

본 연구의 언어별 실험 결과는 네 가지 모델 모두 PHP를 제외한 다른 언어들은 정밀도가 낮고 FP rate이 높은 경향을 가지며, 높은 비율의 과탐을 발생시킨다는 것을 보여준다. 본 논문은 이와 같은 현상이 나타나는 이유를 각 탐지 모델들이 학습될 때 사용된 데이터셋의 구성 차이로 추정한다.

PHP는 가장 널리 쓰이는 웹 스크립트 언어로, 웹쉘 데이터와 정상 데이터의 양이 많고 정상 데이터에 대한 접근성이 다른 언어보다 용이하다. 이러한 특성 때문에 PHP 데이터는 절대적인 데이터의 양이 많을 뿐만 아니라 웹쉘 데이터와 정상 데이터의 비율도 현실의 웹쉘 데이터의 비율과 비슷하게 구성된다.

반면에 JSPX와 같은 언어는 절대적인 데이터의 양이 적다. 이는 JSPX가 웹 스크립트 언어로써의 점유율이 낮고 기존 연구들이 PHP 웹쉘 탐지 위주로 진행이 되어 검증된 벤치마크 데이터셋을 찾기가 어려운 것이 주요 이유로 판단된다. 이렇게 절대적인 데이터의 양이 적은 경우, 학습 데이터셋이 정상 데이터가 웹쉘 데이터보다 더 많은 현실의 분포를 반영하지 못하고 탐지 모델의 낮은 정밀도와 높은 비율의 과탐을 발생시킨다.

따라서 본 논문은 웹쉘 탐지 모델의 성능을 향상시키기 위해서는 학습 데이터의 절대적인 양이 많아야 하며 정상 데이터가 웹쉘 데이터에 비해 더 많은 비율을 차지해야 한다는 두 가지 조건을 도출할 수 있었다.

4. 2. 2. 웹 스크립트 언어의 구조적 특성

본 연구의 모델별 실험 결과에서 PHP-TinyBERT는 PHP 웹쉘 데이터로만 학습했음에도 불과하고 PHP보다 JSP에서 더 높은 성능을 보였다. 본 논문은 이러한 현상의 이유를 웹 스크립트 언어의 구조적 특성 때문이라고 추정한다. JSP는 스크립틀릿 내부에 Java 코드가 삽입되고, PHP는 PHP 태그 내부에 코드의 로직이 삽입된다는 구조적 유사성이 있으므로, PHP 데이터를 학습하는 과정에서 모델의 JSP에 대한 일반화가 일어날 수 있다는 가능성을 시사한다.

4. 2. 3. 소요 시간

표 3의 마지막 열은 코드 파일 하나를 탐지하는데 걸리는 평균 시간을 보여준다. CodeBERT기반의 모델의 경우 파일 하나 처리에 0.55초가 걸리고, TinyBERT기반의 모델의 경우는 0.15초로 처리 속도에 현저한 차이가 있음을 확인할 수 있다. 대형 웹서버의 스크립트 파일 수를 2,000개라고 가정했을 때, 모든 웹 서비스 파일을 CodeBERT기반의 모델을 사용하여 스캔한다고 해도 약 18분 정도가 소요되므로 시간적인 측면에서 실용적 사용이 가능함을 확인할 수 있다.

5. 결론 및 향후 연구

본 논문에서는 오픈소스 Github 레포지토리 및 Hunggingface Hub에서 데이터셋을 구성하여 PHP 웹쉘 탐지 모델과 다언어 웹쉘 탐지 모델의 성능을 언어 별, 모델 별, 크기 별로 다차원적으로 비교하였다. 언어 별 측면에서는 네 가지 모델 모두 PHP에 대한 성능이 가장 높았다. 모델 별 비교에서는 Full-CodeBERT가 ASPX와 PHP에서 가장 높은 성능을 보였고, PHP-TinyBERT가 JSP에서 가장 높은 성능을 보였다. 크기 별 측면에서는 CodeBERT 모델이 TinyBERT 모델보다 더 높은 성능을 보였다.

이러한 실험 결과를 바탕으로 본 논문은 탐지 모델의 낮은 정밀도와 높은 과탐율을 방지하기 위해서는 절대적인 양이 많고 정상 파일의 비율이 웹쉘 파일의 비율보다 높은 학습 데이터를 구축해야 한다는 조건을 제시하였으며, 학습 언어와 다른 언어를 탐지 모델이 일반화할 수 있다는 가능성을 보였다. 또한 각 탐지 모델의 소요 시간을 측정하여 탐지 모델의 실용적인 사용 가능성을 확인하였다.

하지만 본 논문에서는 ASPX와 JSPX 데이터셋에 정상 데이터의 수가 부족한 데이터 불균형이 존재하며 평가 모델들의 학습 데이터가 불분명하다는 한계가 있다. 이를 해결하기 위해 향후에는 추가적인 데이터 수집 및 데이터 증강을 이용해 균형잡힌 데이터셋을 만들고 이 데이터셋을 이용하여 모델을 학습시키고자 한다. 또한 AST, 코드 그래프 등을 활용하여 웹 스크립트 언어에 대한 심층적인 분석을 통해 각 언어가 가지고 있는 유사성을 더 면밀히 확인하고자 한다. 마지막으로 웹쉘 탐지 모델의 성능을 보조하기 위해 정적분석 기반의 웹쉘 탐지 도구를 사용하거나 여러개의 BERT 모델을 앙상블 기법으로 활용하는 새로운 방안에 대해서도 연구하고자 한다.

※ 이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (RS-2024-00457866).

참고 문헌

[1] CHENG, Baijun, et al. Msdetector: a static php webshell detection system based on deep-learning. In: *International Symposium on Theoretical Aspects of Software Engineering*. Cham: Springer International Publishing, p. 155-172. 2022.

[2] AN, Tongjian; SHUI, Xuefei; GAO, Hongkui. Deep learning based webshell detection coping with long text and lexical ambiguity. In: *International Conference on Information and Communications Security*. Cham: Springer International Publishing, p. 438-457. 2022.

[3] FENG, Z. Codebert: A pre-trained model for program-ming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[4] JIAO, Xiaoqi, et al. Tinybert: Distilling bert for natural language understanding. In: *Findings of the association for computational linguistics: EMNLP 2020*. p. 4163-4174. 2020.

[5] LIU, Yinhan, et al. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[6] DEVLIN, Jacob, et al. Bert: Pre-training of deep bidirectional transformers for language understanding. In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. p. 4171-4186. 2019.

[7] HANNOUSSE, Abdelhakim; NAIT-HAMOUD, Mohamed Cherif; YAHIOUCHE, Salima. A deep learner model for multi-language webshell detection. *International Journal of Information Security*, 22.1: 47-61, 2023.

[8] NULL822. WebShell Detection Models based on BERT [online]. Hugging Face. Available from: https://huggingface.co/null822/webshell-detect-bert 2025.

[9] WANG, Guan-Yu, et al. Webshell detection based on codebert and deep learning model. In: *Proceedings of the 2024 5th International Conference on Computing, Networks and Internet of Things*. p. 484-489. 2024.

스마트팩토리 예지보전을 위한 모니터링 소프트웨어 설계

황세현 [1(O)], 김진세 [2], 최민서 [2], 이정원 [1,2]
아주대학교 전자공학과 [1], 아주대학교 AI 융합네트워크학과 [2]
e-mail : bikmiso3@ajou.ac.kr, jinsae913@gmail.com,
minseo24@ajou.ac.kr, jungwony@ajou.ac.kr

Design of Monitoring Software for Predictive Maintenance in Smart Factory

Se-Hyeon Hwang[1(O)], Jin-Se Kim[2], MinSeo Choi[2], Jung-Won Lee[1,2]
Department of Electrical and Computer Engineering, Ajou University[1],
Department of AI Convergence Network, Ajou University[2]

요 약

스마트팩토리 환경에서 산업용 설비의 안정적 운영을 위해서는 설비 상태를 지속적으로 관찰하고 이상 징후를 포착할 수 있는 상태 모니터링 기술이 필수적이다. 그러나 기존 연구는 특정 설비나 알고리즘 중심의 접근에 치중하여, 다중 설비가 복합적으로 운용되는 환경에서 요구되는 시스템 수준의 요구사항 정의와 국제 표준 연계 측면에서 한계를 지닌다. 이에 본 논문은 설비 상태 감시, 설비 안전, 소프트웨어 품질 및 기능 안전과 관련된 국제 표준을 분석하여 상태 모니터링 요구사항을 예지보전 수명주기 관점에서 Operation, Management, Development, Deployment의 네 단계로 구조화하고 핵심 요구사항 속성을 도출하였다. 또한 이를 기반으로 Factory Overview, Live Monitoring, Fault Monitoring, Health Monitoring 레이어로 구성된 모니터링 소프트웨어 UI를 설계 및 구현함으로써, 실시간성, 일관성, 해석 가능성을 효과적으로 확보할 수 있음을 구현 결과를 통해 제시하였다.

1. 서 론

Industry 4.0 시대에 접어들면서, 제조 시스템은 디지털화, 자동화, 데이터 기반 의사결정을 중심으로 고도화되었다[1]. 특히, AI(Artificial Intelligence), IoT(Internet of Things), 협동 로봇, 클라우드컴퓨팅 기술을 포함하는 사이버 물리 시스템(Cyber-Physical System, CPS)은 제조 시스템의 디지털화 및 지능화에 기반한 스마트팩토리로의 전환에 핵심적인 역할을 하였다[2]. 스마트팩토리는 자원을 효율적으로 사용하고 끊임없이 변화하는 생산 요구에 적응하는 것을 목표로 운영되며, 다양한 설비를 활용한 유연한 동적 작업 및 생산 최적화를 통해 효율성과 생산성 향상을 달성하고 있다[3,4].

스마트팩토리 내 다양한 생산 활동이 더욱 복잡해지고 유기적으로 상호작용하면서, 효과적인 유지보수의 중요성은 더욱 커지고 있다[4]. 제조 공정의 높은 생산성, 가용성 및 효율성을 보장하기 위해, 설비의 비정상 상태를 감지하는 것은 매우 중요한 문제이며[5], 이를 해결하기 위해 생산 시스템의 지능화를 바탕으로 스마트 유지보수 기법인 상태 모니터링(Condition Monitoring, CM) 기술의 발전이 이어지고 있다. 상태 모니터링 기술은 시스템에 부착된 센서에서 수집된 데이터를 기반으로, 설비의 상태를 모니터링하고 고장의 징후를 포착하여 사전 예방적 관리에 핵심적인 역할을 한다[6].

상태 모니터링 기술은 설비의 경고 알람 및 향후 성능 저하 예측에 활용되어 설비의 문제를 초기에 파악할 수 있도록 한다. 이는 설비가 운용 가능한 상태를 유지하며 가장 경제적으로 적합한 시기에 유지보수를 수행할 수 있도록 지원한다[7]. 설비에 대한 오경보 방지 및 효과적인 유지보수 달성을 위해서는 정확한 설비 상태 식별이 요구된다. 이를 위해 기존의 상태 모니터링 연구는 고장 탐지 알고리즘 개발, 데이터 기반의 상태 모델링, 또는 전체 시스템 아키텍처 설계를 중심으로 고성능의 기술 확보를 우선하는 접근 방식을 유지해왔다[8].

대표적인 상태 모니터링 기술에는 데이터 기반 접근법과 지식 기반 접근법이 존재한다[9]. 데이터 기반 상태 모니터링은 빅데이터 기술의 발전과 데이터 가용성의 지속적인 증가에 기반하여, 방대한 데이터에서 머신러닝 및 딥러닝을 활용한 유용한 지식 추출 및 효과적인 설비 상태 진단을 수행하였다[10,11,12]. 반면, 지식 기반 상태 모니터링은 전문가 지식과 추론 프로세스를 활용하여 산업 분야에 존재하는 개념과 관계를 공식화하는

[1] 이 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. 2023R1A2C1006332).

온톨로지 모델을 구축하여 설비의 상태 평가 및 모니터링 작업에 활용하였다[13,14]. 이러한 상태 모니터링 기술은 적용 대상 설비의 상태 진단 및 시각적 표현 기반의 고수준 의사결정을 지원한다. 그러나, 기존 상태 모니터링 기술 및 연구는 동시에 다음의 세 가지 주요 한계를 지닌다.

- **설비·공정 간 이질성(heterogeneity)의 고려 미흡**: 기존의 상태 모니터링 연구는 특정 설비(모터, 베어링, 공작기계 등)에 종속된 기술만을 제공한다. 이를 다른 설비에 확장 적용 시 센서 재구성, 시스템 재설계가 필요하며, 스마트팩토리에 통합 적용 시 이질적인 설비 간 복합 운용의 모니터링에 한계를 지닌다[15].
- **시스템 수준 요구사항 정의 및 적용 논의의 부족**: 기존의 상태 모니터링 연구는 주로 기술 개발 및 성능 향상에 초점을 맞춰 왔으며, 상대적으로 실제 스마트팩토리 환경에서 요구되는 시스템 수준의 요구사항의 정의 및 적용 관점의 논의는 부족하다. 이는 상태 모니터링 시스템이 충족해야 할 기능의 체계적인 식별 및 구조화를 어렵게 한다.
- **국제 표준과의 연계 부족**: 관련 국제 표준을 언급한 연구는 존재하나[16,17,18], 이를 기술의 설계 및 구현 관련 요구사항과 연계 또는 기술 개발 프로세스에 반영한 연구는 논의된 바가 없다. 이는 표준과 구현 간 간극의 발생으로 이어져, 상태 모니터링 기술의 실적용 관점 신뢰성 저하 및 안전 인증의 제약으로 작용될 수 있다.

종합하면, 기존 상태 모니터링 연구는 특정 설비와 기술 중심의 접근에 머물러, 다중 설비의 복합적 운용이 이루어지는 스마트팩토리 환경에 적용하기에 체계성과 안정성에 한계를 지닌다. 스마트팩토리는 모니터링 기능의 설계 기준을 명확하게 설정하고 일원화하여 적용하기 어려우며, 데이터 품질 관리, 실시간 상태 표시, 이상 탐지 시각화, 운영 연동성 등 핵심 기능이 개별적으로 다뤄져 전체 관점에서 요구사항의 통합 문제가 존재한다.

이에 본 연구는 스마트팩토리의 상태 모니터링 기술에 요구되는 기능 및 비기능적 요소를 국제 표준을 기반으로 명확히 정의하고, 예지보전 프로세스 단계에 따라 실적용 관점의 요구사항을 구조적으로 재정립한다. 이를 통해 설비의 동적 작업 및 복합 운용에 대해 통합 적용 가능한 표준 기반 상태 모니터링 요구사항 체계를 제시하고, 스마트팩토리 통합 모니터링 및 개별 설비 상태 모니터링을 지원하는 모니터링 소프트웨어 사용자 인터페이스(User Interface, UI)를 설계 및 제안한다. 이는 스마트팩토리 내 설비에 대한 실질적 적용 관점 상태 모니터링 기술의 기능 및 개발 요소를 체계적으로 제시하여, 기술을 구성하는 소프트웨어 UI 개발 및 통합 적용에 대한 가이드라인으로 활용 가능하다.

2. 스마트팩토리 상태 모니터링 요구사항 도출

본 장에서는 스마트팩토리의 설비 상태 모니터링 요구사항을 정의한다. 설비, 운용, 유지보수 관련 국제 표준을 기반으로 요구사항 체계를 정립하고, 상태 모니터링 기술이 충족해야 할 요구사항을 관련 속성 및 요구사항으로 계층화하여 제시한다.

2.1 예지보전 프로세스 기반 요구사항 체계 수립

설비의 예지보전 프로세스는 데이터 수집, 상태 관리, 모델 개발, 배포 및 운영으로 이어지는 일반화된 4단계 수명주기 구조로 개념이 정립되어 있으며[19], 스마트팩토리 설비의 관련 연구에서도 동일한 구조가 요구사항 분류의 기준으로 활용되고 있다[20]. 이러한 선행 연구의 공통된 접근은 예지보전을 단계적으로 이해하고 기능을 체계화하는 데 유효한 틀을 제공한다.

본 연구는 예지보전의 4단계 수명주기 구조를 기반으로 상태 모니터링 기술의 기능 및 비기능 요구사항을 체계화하기 위해, 운영(Operation), 관리(Management), 개발(Development), 배포(Deployment)의 네 단계로 구성하였다. 또한 각 단계의 요구사항을 실적용 가능한 상태 모니터링 기술의 사양으로 연계하기 위해 단계별 핵심 요구 속성을 정의하였다. Operation 단계는 실시간 운용 안정성과 즉시성, Management 단계는 데이터 거버넌스, 품질, 정합성, Development 단계는 표현성, 평가 신뢰성, 강건성, Deployment 단계는 연동성, 효율성, 적응성, 결과 전달 안정성을 핵심 요소로 설정 및 요구사항 정의에 반영하였다.

2.2 국제 표준 기반 상태 모니터링 요구사항 정의

예지보전 수명주기(Operation, Management, Development, Deployment)에 따라 상태 모니터링 기술이 충족해야 할 핵심 요구사항을 정리하였다. 각 단계의 요구사항은 설비 상태를 지속적으로 관찰하고 변화를 감지하기 위한 데이터 수집과 상태 파악(Operation), 수집된 상태 정보의 신뢰성과 일관성을 유지하기 위한 관리 기준(Management), 상태 분석 및 이상 징후 해석의 품질을 확보하기 위한 분석 체계(Development), 그리고 모니터링 결과를 운영 환경에 안정적으로 전달 및 적용하기 위한 요구사항(Deployment)을 중점적으로 고려하였다.

설비 상태 감시, 설비 안전, 소프트웨어 품질 및 기능에 대한 다각도의 요구사항 도출을 위해, 관련 국제 표준(ISO 17359[21], ISO/TS 15066[22], ISO/IEC 25010[23], ISO 26262-6[24])을 기반으로 요구사항 속성 및 하위 요구사항을 도출하였다. 이는 Operation 단계의 5 가지 요구사항 속성(Acquisition, Monitoring, Detection, Safety, Communication), management 단계의 4 가지 요구사항 속성(Governance, Quality, Alignment, Security), Development 단계의 5 가지

요구사항 속성(Interpretability, Evaluation, Robust, Explain, Version), 그리고 Deployment 단계의 5 가지 속성(Interoperability, Efficiency, Adaptability, Resilience, Distribution)으로 정의된다. 스마트팩토리 상태 모니터링 요구사항의 세부 명세는 표 1과 같다.

표 1의 요구사항 속성은 앞서 언급한 국제 표준 문서에 제시된 항목을 단순히 나열한 것이 아니라, 예지보전 수명주기 단계별 역할과 상태 모니터링 기술의 설계 관점을 기준으로 선별·재구성한 결과이다. 먼저 각 국제 표준에서 상태 감시, 안전, 소프트웨어 품질, 데이터 관리와 직접적으로 연관된 요구사항을 추출한 후, 이를 예지보전 4단계에 대응하도록 재분류하였다. 이러한 과정을 통해 표준 요구사항이 상태 모니터링 소프트웨어의 UI 구조 및 기능 설계에 직접 활용 가능한 요구사항 체계로 도출되도록 하였다.

표 1. 예지보전 단계별 상태 모니터링 요구사항 도출

예지보전 단계	상태 모니터링 요구사항 속성	요구사항 명세
Operation	Acquisition	설비 모니터링은 데이터를 시각화에 최적화된 형태로 처리해야 한다.
	Monitoring	설비 모니터링은 설비 운영 현황을 쉽게 파악할 수 있도록 정보를 구조화하여 시각화해야 한다.
	Detection	설비 모니터링은 이상 징후 발생 시 정상 패턴 대비 편차를 사용자가 식별할 수 있도록 시각화하여 제공해야 한다.
	Safety	설비 모니터링은 안전 관련 이벤트를 신속하게 탐지하고, 경고 알림 및 시각적 강조를 통해 사용자가 상황을 즉각적으로 인지할 수 있도록 해야 한다.
	Communication	설비 모니터링은 통신 상태와 제어 신호 변화를 직관적이고 즉각적으로 시각화해야 한다.
Management	Governance	설비 모니터링은 시각화의 정확성과 일관성을 보장하기 위해, 데이터정보가 안정적이고 왜곡 없이 표현되도록 해야 한다.
	Quality	설비 모니터링은 데이터 품질 관리를 통해 화면에 표현되는 정보의 신뢰성을 확보해야 한다.
	Alignment	설비 모니터링은 데이터의 시각화 정보가 시간적으로 정확하고 일관되게 표현되도록 보장해야 한다.
	Security	설비 모니터링 화면에 노출되는 정보가 보안 규정을 충족하도록 해야 한다.
Development	Interpretability	설비 모니터링은 데이터가 화면에서 의미 있게 해석되도록 해야 한다.
	Evaluation	설비 모니터링은 화면에 표현되는 판단 결과가 운영 흐름을 정확히 반영하고 편향 없이 해석되도록 보장해야 한다.
	Robustness	설비 모니터링은 운영 환경 변화나 데이터 변동이 발생해도 결과가 안정적이고 일관된 형태로 시각화 되도록 해야 한다.
	Explainability	설비 모니터링이 도출한 판단이나 결과를 사용자가 직관적으로 이해할 수 있도록, 근거와 맥락을 시각적 요소와 함께 명확하게 설명할 수 있어야 한다.
	Versioning	설비 모니터링의 설정, 사용 데이터 등 주요 구성 요소를 일관된 방식으로 관리하여, 화면에 표시되는 결과가 어떤 모델 기반인지 명확하게 식별할 수 있어야 한다.
Deployment	Interoperability	설비 모니터링은 다양한 장비·환경·플랫폼과 연동되는 과정에서도 화면의 일관성과 시각적 흐름을 유지해야 한다.
	Efficiency	설비 모니터링은 결과가 운영 환경에서 지연 없이 정확히 표현될 수 있도록 해야 한다.
	Adaptability	설비 모니터링의 변경이나 업데이트는 화면에 표시되는 정보 흐름과 운영 안정성을 해치지 않는 방식으로 안전하게 수행되어야 하며, UI 는 과정 중에도 일관된 상태를 유지해야 한다.
	Resilience	설비 모니터링의 잘못된 결과나 판단이 화면에 표시되지 않도록 적절한 처리가 이루어져야 한다.
	Distribution	설비 모니터링 현장의 자원이나 조건이 달라지더라도, UI 에 표시되는 모니터링 품질과 분석 결과는 안정적으로 유지되어 사용자에게 일관된 정보를 전달해야 한다.

3. 스마트팩토리 모니터링 소프트웨어 설계 및 구현

본 장에서는 스마트팩토리 상태 모니터링을 위한 UI 중심 설계 요소를 제안하고, 도출된 요구사항을 UI 구성 요소에 연계한 구현 예시를 제시하였다.

3.1 상태 모니터링 요구사항 구체화

UI 중심의 모니터링 소프트웨어 설계를 위해 요구사항을 표2와 같이 구체화하였다. 요구사항의 속성과 세부 항목을 분석하여 핵심 기능을 도출하고, 공통 속성을 기준으로 UI 계층(Layer)으로 구성하였다.

표 2. 스마트팩토리 모니터링 소프트웨어의 상태 모니터링 요구사항 연계 및 구체화

Layer		요구사항 구체화	관련 요구사항
System UI Framework Layer	L1-1	통합 시각 언어(UI Consistency)	Operate-Monitoring, Deploy-Interoperability
	L1-2	시간 해상도 기반 시각화	Operate-Monitoring, Operate-Acquisition
	L1-3	데이터 타입별 시각화 매핑	Operate-Monitoring, Manage-Governance, Manage-Quality
	L1-4	상태 기반 색상/아이콘 시스템	Operate-Monitoring, Operate-Detection, Operate-Safety
	L1-5	통일된 선택·필터 구조	Operate-Monitoring, Manage-Governance
	L1-6	표준화된 레이아웃 구조	Manage-Governance, Deploy-Interoperability
	L1-7	실시간 데이터 반영	Operate-Acquisition, Deploy-Efficiency
	L1-8	레이어 간 drill-down 네비게이션	Operate-Monitoring, Operate-Detection, Deploy-Interoperability
Factory Overview Layer	L2-1	설비 플릿 개요 대시보드 구현	Operate-Monitoring, Deploy-Interoperability
	L2-2	상태·이상 징후의 시각적 강조	Operate-Monitoring, Operate-Detection, Operate-Safety-
	L2-3	설비 레이아웃 기반 위치 맵	Operate-Monitoring, Manage-Governance, Deploy-Interoperability
	L2-4	공장 설비 수준 집계 KPI 제공	Operate-Monitoring, Operate-Acquisition, Manage-Quality
	L2-5	설비별 가동률·고장 시간 비교 그래프	Operate-Monitoring, Operate-Detection, Manage-Alignment
	L2-6	주기적 데이터 갱신 및 상태 표시	Operate-Acquisition, Deploy-Efficiency, Deploy-Resilience
Live Monitoring Layer	L3-1	설비 동작의 실시간 시각화	Operate-Monitoring, Operate-Acquisition, Deploy-Efficiency
	L3-2	현재 작업 및 운전 상태 실시간 표시	Operate-Monitoring, Operate-Communication, Operate-Monitoring
	L3-3	안전·인터락 상태의 즉각 시각적 피드백	Operate-Safety, Operate-Communication
	L3-4	설비와 주변 환경 간 충돌 위험 시각화	Operate-Safety, Operate-Detection, Develop-Explain
	L3-5	Fault 발생의 즉각적 배너 알림	Operate-Detection, Operate-Safety, Deploy-Resilience, Develop-Explain
Fault Monitoring Layer	L4-1	기간 단위 Fault 통계 제공	Operate-Monitoring, Manage-Alignment, Operate-Detection
	L4-2	장기간 누적 지표제공	Manage-Quality, Manage-Governance, Deploy-Interoperability
	L4-3	Daily Stats 기반 세부 Fault 분석	Operate-Monitoring, Operate-Detection, Manage-Alignment
	L4-4	Fault Event 타임라인 시각화	Operate-Detection, Operate-Safety, Manage-Alignment
	L4-5	Fault 유형별 분포 및 밀도 분석 시각화	Develop-Explain, Operate-Detection, Manage-Governance
	L4-6	Precursor 기반 위험도 시각화	Operate-Detection, Develop-Explain, Develop-Robust
	L4-7	설비의 관절별 Top Faulted Ranking	Operate-Monitoring, Develop-Explain, Manage-Quality
	L4-8	운영 영향 및 유지보수 권고사항 UI	Develop-Explain, Operate-Safety, Deploy-Resilience
Health Monitoring Layer	L5-1	기간·대상 필터 기반 Health 분석 UX	Operate-Monitoring, Deploy-Interoperability, Develop-Representation
	L5-2	설비·관절 단위 Health Level 시각화	Develop-Explain, Operate-Detection, Develop-Representation
	L5-3	누적 손실 기반 저하 추세 시각화	Develop-Explain, Develop-Robust, Deploy-Efficiency
	L5-4	날짜별 Health 기록 테이블 제공	Manage-Governance, Manage-Integrity, Manage-Alignment
	L5-5	설비의 Health 마커 기반 3D 시각화	Operate-Acquisition, Develop-Representation, Manage-Quality
	L5-6	설비 구성 요소 단위 Raw Telemetry 분석	Operate-Monitoring, Develop-Explain, Deploy-Resilience

3.2 스마트팩토리 모니터링 소프트웨어 UI 설계

상태 모니터링 요구사항의 구체화 결과를 기반으로 스마트팩토리 모니터링 소프트웨어 UI 중심의 구조를 설계하였다. 그림 1과 같이 시각화 중심의 UI 구조로 설계되었으며, 요구사항 구체화를 통해 도출된 핵심 기능 요소를 반영하여 Factory Overview, Live Monitoring, Fault Monitoring, Health Monitoring의 네 가지 UI 레이어를 구성하였다. 화살표는 레이어 간 기능적 연계와 사용자 상호작용 흐름을 의미한다.

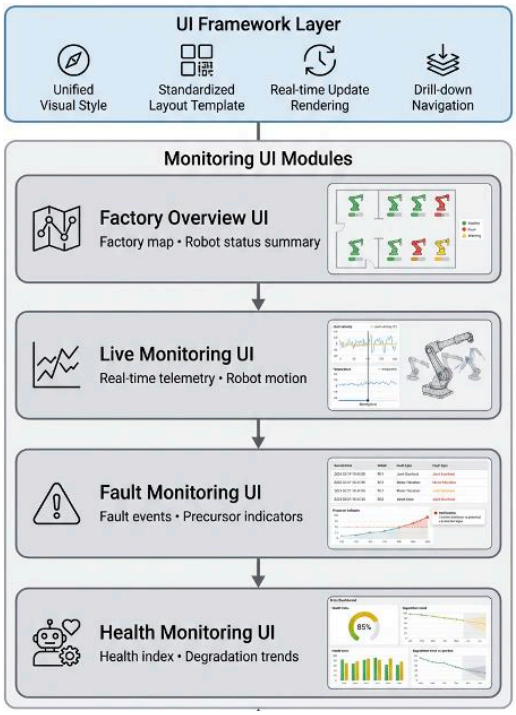


그림 1. 예지보전 모니터링 소프트웨어의 전체 UI 구조

3.3 스마트팩토리 모니터링 소프트웨어 개발

3.3.1 System UI Framework Layer

System UI Framework Layer는 앞서 정의한 예지보전 요구사항을 UI 구조에 일관되게 반영하기 위한 상위 프레임워크이다. 통합 시각 언어(UI Consistency)와 표준화된 레이아웃 구조를 기반으로 데이터 유형별 시각화 매핑, 상태 기반 색상 및 아이콘 시스템, 일관된 선택 및 필터 구조를 적용하였다. 또한 시간 해상도 기반 시각화와 실시간 데이터 반영을 통해 운영 상태 변화가 즉시 화면에 반영되도록 설계하였으며, 레이어 간 드릴다운(drill-down) 내비게이션을 통해 상위 요약 화면에서 하위 상세 분석 화면으로 단계적으로 이동할 수 있도록 구성하였다(L1-8).

3.3.2 Factory Overview Layer

Factory Overview Layer는 아래 그림 2과 같이 공장 단위의 운영 상태를 한눈에 파악하기 위한 상위 시각화 모듈로, 설비의 플릿(fleet, 동일 공정 또는 공장 내에서 함께 운용되는 설비 집합) 전체의 가동 현황과 이상 징후를 통합적으로 제공하도록 설계되었다(L2-1,2).

설비의 가동 상태, 생산량, 주요 운전 지표는 데이터가 발생한 시간 순서를 기준으로 표시하는 방식으로 정렬되어 표시되며(L2-5), 토크, 속도, 온도와 같은 기본 텔레메트리(Telemetry)는 데이터 타입에 맞는 시각화 방식으로 매핑된다. 텔레메트리는 설비 센서에서 실시간으로 수집되는 동작 및 상태 데이터를 의미한다. 이를 통해 주기적 데이터 갱신과 실시간 상태 반영을 통해 운영 상황이 파악할 수 있다(L2-6).

이상 징후는 정상 상태 대비 편차를 강조하는 방식으로 시각화 된다. 설비별 가동률, Cycle Time, 고장 발생 여부는 그래프 및 강조 색상 표현을 통해 즉시 식별 가능하며, 이는 운영자가 문제 구간을 빠르게 파악할 수 있도록 지원한다. 또한 공정 레이아웃 기반 위치 맵(Localization Map)을 함께 제공하여, 이상 상태가 발생한 설비의 물리적 위치와 공정 흐름을 동시에 확인할 수 있다(L2-3).

공장 수준의 집계 KPI(Key Performance Indicator)와 설비별 요약 정보는 하나의 화면에 통합되어 제공되며, 이는 통합 대시보드 구성, 일관된 시각 구조라는 요구사항을 충족한다(L2-4).

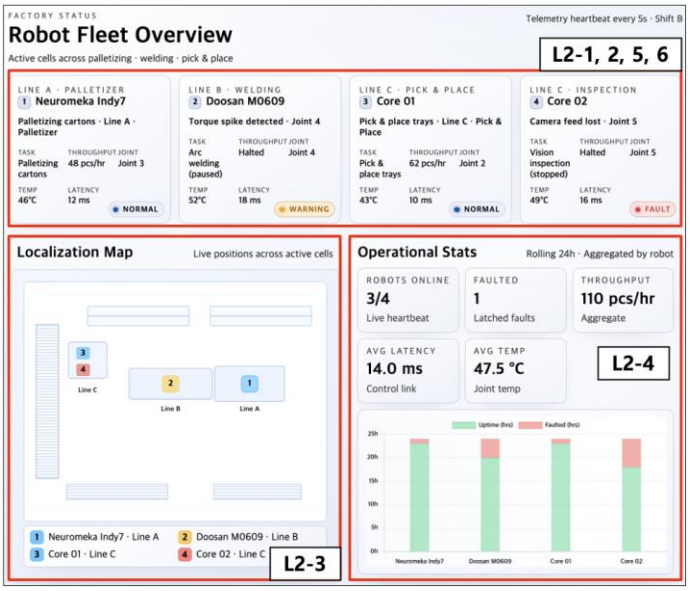


그림 2. 위치 기반 레이아웃 및 운영 통계 시각화

3.3.3 Live Monitoring Layer

Live Monitoring Layer는 아래 그림3과 같이 개별 설비의 동작 상태를 실시간으로 관찰하고, 운전 중 발생하는 이상 상황과 안전 상태를 즉각적으로 인지할 수 있도록 설계된 시각화 모듈이다.

먼저 설비 동작의 실시간 시각화(L3-1)를 위해, 설비의 관절 움직임과 동작 궤적을 실시간으로 갱신되는 시각 요소로 표현하였다. 이와 함께 현재 작업 및 운전 상태 표시(L3-2)를 통해 수행 중인 작업 단계, 사이클 상태, 운전 모드 등의 정보를 화면 상단에 명확히 표시하여, 사용자가 설비의 현재 상태를 즉시 파악할 수 있도록 하였다.

안전 관련 요구사항을 충족하기 위해 안전·인터락 상태의 즉각적 시각적 피드백(L3-3)을 적용하였다. 인터락(interlock)은 위험 상황에서 설비의 동작을 제한하는 안전 제어 상태를 의미하며, 본 화면에서는 인터락 및 비상정지(E-stop) 활성 여부를 색상 변화와 아이콘을 통해 즉시 확인할 수 있도록 구성하였다.

또한 충돌 위험의 실시간 분석 시각화(L3-4)를 위해 작업자와 설비 간 거리, 접근 속도, 위험 점수를 종합한 충돌 위험 지표를 제공하며, 위험 수준에 따라 시각적 강조 강도를 차등 적용하였다. 이상 상황이 감지될 경우에는 Fault 발생의 즉각적 배너 알림(L3-5)을 통해 화면 상단에 경고 메시지를 표시하여, 사용자가 지연 없이 상황을 인지하고 대응할 수 있도록 설계하였다.

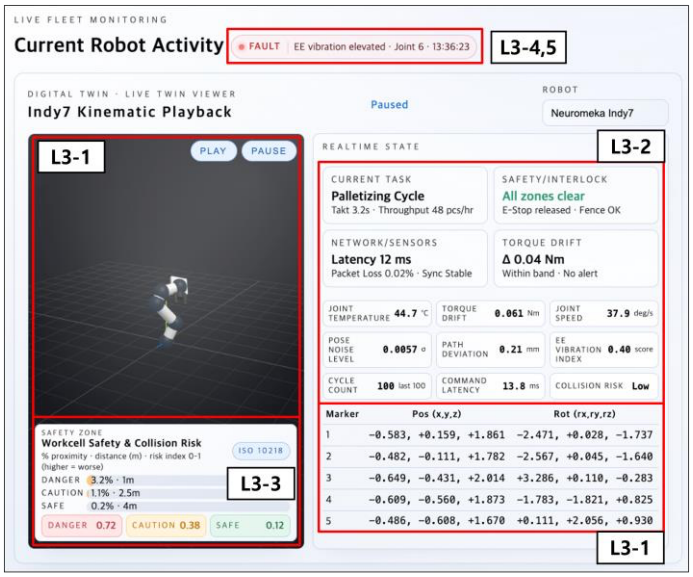


그림 3. 모니터링 소프트웨어의 Live Monitoring UI 화면

3.3.4 Fault Monitoring Layer

Fault Monitoring Layer는 아래 그림 4-7과 같이 설비 시스템에서 발생하는 고장(Fault)과 그 진행 양상을 정량적으로 분석하기 위한 시각화 모듈이다.

먼저 그림 4, 5에서는 설비 전체와 관절별 기간 단위 Fault 통계와 (L4-1)과 장기간 누적 지표(L4-2)를 통해 고장 발생 횟수, 가동 시간 대비 고장 비율 등을 집계하여 제시한다. 이를 통해 사용자는 고장의 장기적 추세와 반복 발생 여부를 한눈에 파악할 수 있다.

그림 6, 7에서는 Daily Stats 기반 하루 단위 세부 Fault 분석(L4-3)을 제공하여, 특정 날짜에 집중된 고장 양상이나 단기 이상 패턴을 정밀하게 분석할 수 있도록 하였다. 그리고 고장의 시간적 흐름을 파악하기 위해 Fault Event 타임라인 시각화(L4-4)를 구현하였다. 타임라인은 고장 발생 시점을 시간 축에 따라 배열하여, 고장이 집중되는 구간이나 특정 운전 조건과의 연관성을 직관적으로 확인할 수 있도록 한다. 더 나아가 Fault 유형별 분포 및 밀도 분석 시각화(L4-5)를 통해 고장 유형이 특정 관절이나 조건에 편중되는지를 분석할 수 있도록 구성하였다.

또한 Precursor 기반 위험도 시각화(L4-6)를 포함하였다. Precursor는 고장 발생 이전에 반복적으로 관측되는 초기 이상 징후를 의미하며, 본 UI에서는 precursor 지표의 변화를 시각적으로 강조하여 잠재적 고장 위험을 조기에 발견할 수 있도록 지원한다.

고장 영향의 집중도를 명확히 하기 위해 관절별 Top Faulted Ranking(L4-7)을 제공하여, 고장이 빈번하게 발생하는 관절을 식별할 수 있도록 하였다. 마지막으로 운영 영향 및 유지보수 권고사항 UI(L4-8)를 통해 고장이 생산성, 안전성에 미치는 영향과 함께 유지보수 권고 정보를 시각적으로 제시하였다.

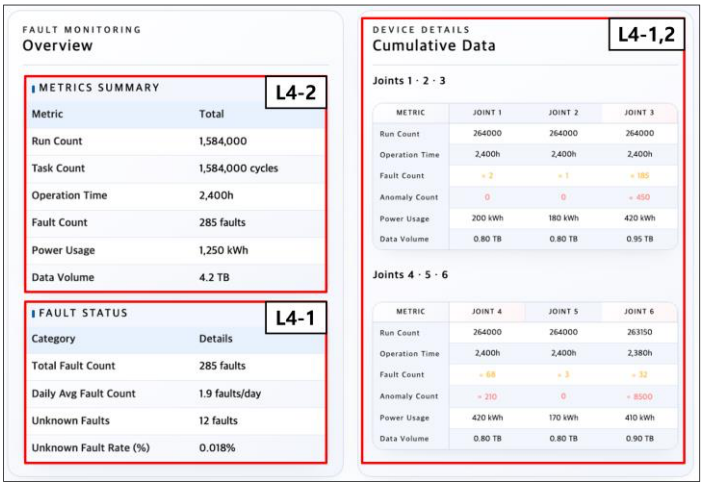


그림 4. Fault Monitoring UI의 기간 단위 요약 화면

그림 5. 기간 기반 고장 진행 및 분포 분석 화면

그림 6. Daily Stats 및 Precursor 기반 위험도 분석 화면

그림 7. 시간대별 Fault 빈도 타임라인 및 권고 조치 사항

3.3.5 Health Monitoring Layer

Health Monitoring Layer는 아래 그림 8-10와 같이 설비 및 관절 단위의 장기적 성능 저하와 건강 상태를 종합적으로 평가하기 위한 시각화 모듈이다.

먼저 그림10의 설비·관절 단위 Health Level 시각화(L5-1)를 통해 각 설비와 관절의 현재 상태를 단계적 지표로 제공한다. Health Level은 설비 동작 중 발생하는 부하, 진동 등 누적 피로 요인을 종합하여 산출되는 지표로, 상태를 직관적인 Level로 표현한다.

장기적인 상태 변화를 파악하기 위해 누적 손실 기반 저하 추세 시각화(L5-2)를 제공한다. 누적 손실 기반 지표는 시간에 따른 Health Index 변화를 연속적으로 표시하여 점진적인 성능 저하 흐름을 확인할 수 있도록 하며, 점진적으로 저하가 어떻게 축적되는지를 보여준다.

날짜별 Health 기록 테이블 제공(L5-3)을 통해 특정 시점 관절별 움직임과 상태를 정량적으로 확인 가능하도록 하였다. 이를 통해 사용자는 특정 이벤트 전후의 상태 차이, 급격한 열화 시점을 추적할 수 있다.

또한 설비의 정밀 분석을 위해 관절별 Raw Telemetry 드릴다운 분석(L5-4)을 포함하였다. Raw Telemetry는 필터링이나 요약 이전의 원본 센서 신호를 의미하며, 드릴다운(drill-down) 기능은 Health record에서 특정 테이블을 클릭하면 그림 8, 10과 같이 특정 시점의 데이터로 단계적으로 이동하여 세부 패턴을 분석할 수 있도록 한다.

그림 11과 같이 설비 Health 마커 기반 3D 시각화(L5-5)를 통해 설비 관절의 실제 동작 궤적을 3차원 공간에서 표현하였다. Trajectory는 관절이 이동한 경로를 의미하며, 이를 통해 반복적으로 발생하는 비정상 경로, 특정 방향 편향, 운동 범위 축소 등을 직관적으로 식별할 수 있다. 마지막으로 Health 분석 UX(L5-6)를 제공하여, 사용자가 특정 설비, 관절, 기간 조건을 선택해 분석 범위를 유연하게 조정 가능하도록 설계하였다.

그림 8. 기간·대상 필터 기반 Health 분석 UX

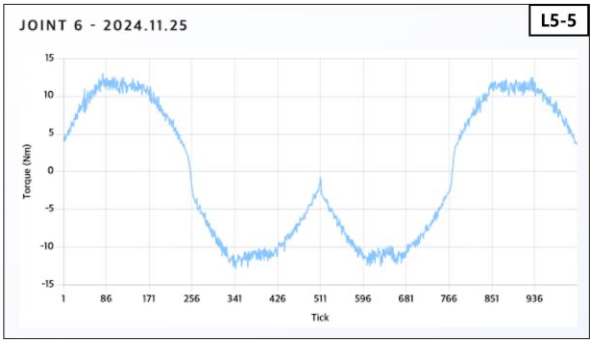


그림 9. 하루 동안 수집된 관절의 토크 데이터

그림 10. Health Monitoring UI의 전체 구조

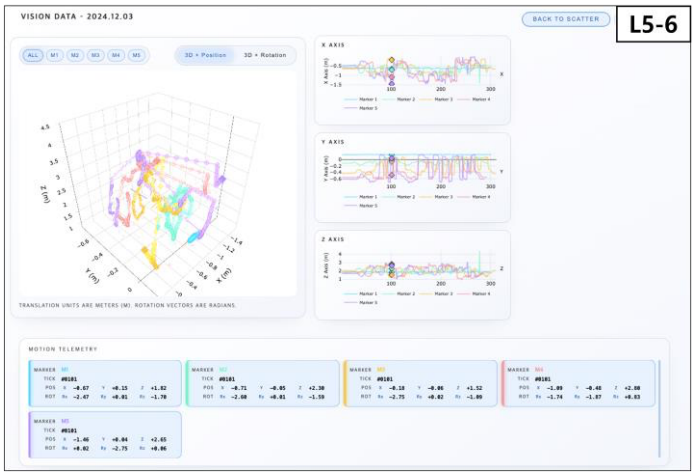


그림 11. 설비 동작 Telemetry의 3D Trajectory 분석 화면

4. 결 론

본 연구는 국제 표준 기반의 요구사항 분석을 토대로 스마트팩토리 모니터링 소프트웨어 UI를 체계적으로 설계하고, 이를 친화적인 시각화 기반 UI로 구현 및 개발하였다. 제안하는 모니터링 소프트웨어는 Factory Overview, Live Monitoring, Fault Monitoring, Health Monitoring으로 이어지는 모듈형 UI를 통해 운영 상태 파악부터 이상 탐지, 장기 성능 진단에 이르는 예지보전 전 과정을 지원한다. 구현 결과는 모니터링 소프트웨어가 국제 표준 및 요구사항 기반의 설계가 및 구현을 통해 일관성, 해석 가능성, 실시간성을

효과적으로 확보함을 보여주었다. 향후에는 실제 산업 환경에서의 적용 평가와 기능 검증을 통해 소프트웨어의 실용성과 범용성을 더욱 강화할 예정이다.

5. 참고 문헌

[1] Jay Lee, Behrad Bagheri, Hung-An Kao, “Recent advances and trends of cyber-physical systems and big data analytics in industrial informatics,” Proceedings of the IEEE International Conference on Industrial Informatics (INDIN), pp. 1-6, 2014.

[2] A. A. Murtaza et al., “Paradigm shift for predictive maintenance and condition monitoring from Industry 4.0 to Industry 5.0: A systematic review, challenges and case study,” Results in Engineering, vol. 24, p. 102935, 2024.

[3] Eduardo Munera, et al., “Control kernel in smart factory environments: Smart resources integration,” Proceedings of the IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), pp. 1-6, 2015.

[4] Andrew K. S. Jardine, Daming Lin, Dragan Banjevic, “A review on machinery diagnostics and prognostics implementing condition-based maintenance,” Mechanical Systems and Signal Processing, vol. 20, no. 7, pp. 1483-1510, 2006.

[5] Erim Sezer, et al., “An Industry 4.0-enabled low cost predictive maintenance approach for SMEs,” Proceedings of the IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC), pp. 1-8, 2018.

[6] Hashem M. Hashemian, “State-of-the-art predictive maintenance techniques,” IEEE Transactions on Instrumentation and Measurement, vol. 60, no. 1, pp. 226-236, 2010.

[7] Enrico Zio, “Prognostics and Health Management (PHM): Where are we and where do we (need to) go in theory and practice,” Reliability Engineering & System Safety, vol. 218, p. 108119, 2022.

[8] N. Omri, et al., “Industrial data management strategy towards an SME-oriented PHM,” Journal of Manufacturing Systems, vol. 56, pp. 23-36, 2020.

[9] Mahbuba Afrin, et al., “Multi-objective resource allocation for edge cloud based robotic workflow in smart factory,” Future Generation Computer Systems, vol. 97, pp. 119-130, 2019.

[10] Franco Giustozzi, Julien Saunier, Cecilia Zanni-Merk, “A semantic framework for condition monitoring in Industry 4.0 based on evolving knowledge bases,” Semantic Web, vol. 15, no. 2, pp. 583-611, 2024.

[11] Kamran Javed, Rafael Gouriveau, Noureddine Zerhouni, “State of the art and taxonomy of prognostics approaches, trends of prognostics applications and open issues towards maturity at different technology readiness levels,” Mechanical Systems and Signal Processing, vol. 94, pp. 214-236, 2017.

[12] Rui Zhao, et al., “Deep learning and its applications to machine health monitoring,” Mechanical Systems and Signal Processing, vol. 115, pp. 213-237, 2019.

[13] Tianci Zhang, et al., “Intelligent fault diagnosis of machines with small and imbalanced data: A state-of-the-art review and possible extensions,” ISA Transactions, vol. 119, pp. 152-171, 2022.

[14] Ying Peng, Ming Dong, Ming Jian Zuo, “Current status of machine prognostics in condition-based maintenance: A review,” The International Journal of Advanced Manufacturing Technology, vol. 50, no. 1-4, pp. 297-313, 2010.

[15] J. Garcia et al., “Condition monitoring and predictive maintenance in industrial equipment: An NLP-assisted review of signal processing, hybrid models, and implementation challenges,” Applied Sciences, vol. 15, no. 10, p. 5465, 2025.

[16] David Lira Nuñez, Milton Borsato, “OntoProg: An ontology-based model for implementing Prognostics Health Management in mechanical machines,” Advanced Engineering Informatics, vol. 38, pp. 746-759, 2018.

[17] Jong-Ho Shin, Hong-Bae Jun, “On condition based maintenance policy,” Journal of Computational Design and Engineering, vol. 2, no. 2, pp. 119-127, 2015.

[18] Stefan Panov, Anton Nikolov, Svetlana Panova, “Review of standards and systems for predictive maintenance,” Science, Engineering and Education, vol. 6, no. 1, pp. 65-73, 2021.

[19] Carolin Wagner, Bernd Hellingrath, “Supporting the implementation of predictive maintenance: a process reference model,” International Journal of Prognostics and Health Management, vol. 12, no. 1, pp. 1-15, 2021.

[20] Jay Lee, et al., “Introduction to cyber manufacturing,” Manufacturing Letters, vol. 8, pp. 11-15, 2016.

[21] ISO 17359, Condition monitoring and diagnostics of machines – General guidelines, International Organization for Standardization, 2018.

[22] ISO/TS 15066, Robots and robotic devices – Collaborative robots, International Organization for Standardization, 2016.

[23] ISO/IEC 25010, Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model, International Organization for Standardization, 2023.

[24] ISO 26262-6, Road vehicles – Functional safety – Part 6: Product development at the software level, International Organization for Standardization, 2018.

충돌 위험 인식을 위한 Grad-CAM과 LLM 결합 설명 시스템

신지아 [1], 이선아 [2, 3]
경상국립대학교 항공우주공학부 [1], 경상국립대학교 AI 융합공학과 [2], 경상국립대학교 소프트웨어공학과 [3]
ssshinjia@gmail.com, saleese@gmail.com

A Grad-CAM and LLM-Based Explainable System for Collision Risk Perception

Jia Shin[1], Seonah Lee[2, 3]
Department of Aerospace Engineering, Gyeongsang National University[1],
Department of AI Convergence Engineering, Gyeongsang National University[2],
Department of Software Engineering, Gyeongsang National University[2, 3]

요 약

본 연구에서는 실시간 영상 스트림을 입력으로 하는 충돌 위험 인식 시스템을 제안하였다. 제안한 시스템은 CNN 기반 충돌 분류 모델을 통해 충돌 위험을 예측하고, Grad-CAM 기법을 적용하여 모델의 판단 근거를 히트맵 형태로 시각화함으로써 충돌 위험이 집중된 영상 영역을 제시한다. 또한 생성된 히트맵을 기반으로 충돌 위험률을 정량화하고, 대규모 언어 모델(LLM)을 활용하여 위험 객체의 위치와 위험 방향을 해석함과 동시에 회피가 필요한 이동 방향을 자연어로 제공한다. 본 시스템은 단순한 충돌 여부 판단을 넘어, 모델의 예측 근거와 위험 발생 원인을 사용자에게 직관적으로 전달함으로써 자율 시스템의 안전한 의사결정을 지원한다.

1. 서 론

자율 주행 및 로봇 안전 시스템에서 딥러닝 기반의 충돌 위험 예측은 핵심 기술로 자리잡고 있다. 그러나, 딥러닝 모델은 일반적으로 내부 판단 과정이 명확히 드러나지 않는 블랙박스 구조를 가지고 있어, 모델이 산출한 위험 예측 결과를 실제 제어 및 의사결정에 적용하기 위해서는 그 판단 근거를 설명할 수 있어야 한다. 이를 해결하기 위해 설명 가능 인공지능(eXplainable AI, XAI) 기법이 중요한 역할을 한다. XAI는 모델의 의사결정 과정을 인간이 이해할 수 있는 형태로 해석할 수 있게 해 준다[1].

잘 알려진 설명 가능 인공지능 기법으로는 SHAP과 LIME이 있다[2]. 그러나 두 기법들은 개별 입력에 대한 로컬 설명을 도출하기 위한 시간이 많이 소요되어 자율주행 및 자율비행의 실시간성의 요구에 맞지 않는 측면이 있다. Grad-CAM(Gradient-weighted Class Activation Mapping)은 한 번의 순전파와 역전파로 히트맵(Heatmap)을 생성할 수 있어, 샘플링 기반 설명 기법에 비해 실시간적인 해석 성능이 높다. 따라서 자율 주행 등의 분야에서 딥러닝 모델의 해석 가능성을 향상시키는 대표적인 방법으로 활용되고 있다[2].

본 연구에서는 Grad-CAM을 충돌 분류 모델에 적용하여, 실시간 영상 프레임을 기반으로 충돌 위험을 판단하는 시스템을 제안한다. Grad-CAM은 모델이 충돌 위험 판단에 기여한 영상 영역을 히트맵 형태로 시각화하여, 충돌 위험이 높은 영역을 강조하고 이를 기반으로 충돌 위험률을 산출한다. 본 연구에서는 추가적으로 LLM을 활용하여 위험이 감지된 객체의 위치와 위험 방향을 해석하고, 회피가 필요한 이동 방향을 자연어로 설명함으로써 사용자에게 직관적인 위험 인식을 제공한다.

제안 시스템의 효용성을 검증하기 위해 연속 프레임 영상 데이터셋을 활용하여 다양한 충돌 시나리오에 대한 추론 및 해석 실험을 수행하였다. 실험 결과, 모델이 생성한 Grad-CAM 히트맵은 충돌 위험 객체의 핵심 영역을 정확하게 식별하였고, LLM은 이러한 시각적 근거를 바탕으로 위험 수준과 객체의 상대적 위치를 이해할 수 있는 수준으로 해석하였다. 특히 장애물 위치에 따른 회피 방향을 직관적인 자연어로 제시함으로써, 조종사 상황 인식(Situation Awareness) 및 판단 능력을 보조할 수 있음을 입증하였다.

본 논문의 구성은 다음과 같다. 2장에서는 기존 연구를 고찰한다. 3장에서는 본 연구에서 제안하는 Grad-CAM 기반 실시간 충돌 위험 인식 시스템의 전체 구조를 기술한다. 4장과 5장에서는 구체적인 실험 환경 설정과 그에 따른 정량적 분석 결과를 제시한다. 마지막으로 6장에서는 연구의 결론을 맺으며, 7장에서 본 시스템의 한계점과 이를 보완하기 위한 향후 연구 방향을 기술한다.

2. 관련 연구

기존의 XAI 기법들은 주로 고정된 구조의 모델을 대상으로 설계되어 왔으나, 최근에는 파라미터 수가 방대한 대규모 언어 모델(LLM)의 특성을 고려한 설명 가능성 연구로 확장되고 있다. LLM은 분류나 회귀를 넘어 생성, 추론, 대화 등 복합적인 기능을 수행함에 따라, 기존의 feature attribution 중심 설명 방식에서 나아가 자연어 기반 설명과 모델 내부 추론 과정을 활용하는 새로운 XAI 접근이 제안되고 있다[3]. 또한 gradient 기반 feature attribution과 같은 기법은 입력 특성이 출력 결과에 미치는 영향을 정량적으로 분석함으로써 모델의 신뢰성과 해석 가능성을 강화할 수 있는 방법으로 주목받고 있다[4].

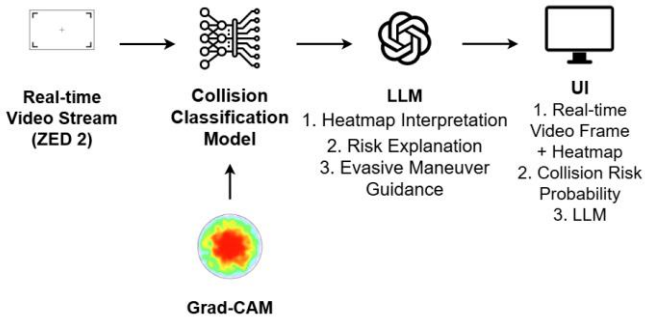
Grad-CAM은 합성곱 신경망의 특징 맵을 기반으로 모델의 전역적인 판단 경향을 시각적으로 나타낼 수 있어 보다 직관적인 해석이 가능하다[5]. 특히, Grad-CAM은 1회 역전파로 히트맵을 생성해 연산 오버헤드가 낮아, 충돌 위험 예측과 같은 영상 기반 실시간 시스템에 적합하다. 예를 들어, 대시캠 연속 영상 데이터를 대상으로 시공간적 관계를 학습하는 순환 신경망 기반 모델과 Grad-CAM을 결합해 사고 발생 이전의 위험 징후를 시각적으로 설명하려는 연구가 제안되었다[6]. 이러한 연구들은 사고를 조기에 예측함과 동시에, 인간이 해석 가능한 시각적 근거를 제공함으로써 모델 예측에 대한 신뢰성을 향상시킬 수 있음을 보여주었다[7].

기존 연구들은 Grad-CAM을 활용해 충돌 위험의 시각적 근거를 제시함으로써 블랙박스 모델의 신뢰성을 확보하는 데 기여해 왔다. 그러나 시각화된 히트맵은 비전문가 사용자가 긴박한 실시간 비행 상황에서 그 의미를 즉각적으로 해석하여 대응하기에는 직관성이 부족하다는 한계가 있다. 또한, 대다수의 선행 연구는 사고 전조 증상을 시각화하는 사후 분석이나 단순 알림 서비스에 국한되어 있어, 감지된 위협으로부터 안전하게 이탈하기 위한 구체적인 의사결정 지원 정보는 충분히 제공하지 못한다.

본 연구는 이러한 제약점을 극복하기 위해 Grad-CAM과 대규모 언어 모델(LLM)을 결합한 충돌 위험 해석 시스템을 제안한다. 우선 정적인 시각 정보인 히트맵 데이터를 LLM이 인식 가능한 텍스트 파라미터로 변환하여, 모호한 시각적 지표를 자연어 기반의 구체적인 상황 설명으로 구체화한다. 이를 통해 사용자는 복잡한 히트맵을 별도로 분석할 필요 없이 AI가 생성한 브리핑을 통해 위험 상황을 즉각적으로 인지할 수 있다. 더 나아가, 단순한 위험 감지를 넘어 감지된 객체의 상대적 위치(좌/우/중앙)를 분석하고, 이를 바탕으로 최적의 회피 기동 방향을 도출하여 자연어로 안내한다. 즉, 기존 연구가 단순히 위험 여부를 판별하거나 시각적 투영에 머물렀다면, 본 연구는 Grad-CAM과 LLM을 유기적으로 결합하여 '왜 위험한가'에 대한 시각적 원인을 실시간으로 규명함과 동시에 '어떻게 대처해야 하는가'에 대한 실천적 해답을 자연어로 제시한다는 점에서 신규성을 확보한다.

3. 제안 방법

제안한 충돌 위험 인식 시스템은 드론에 장착된 카메라로부터 입력되는 실시간 영상 스트림을 시스템의 입력으로 받아, 딥러닝 기반 충돌 분류 모델을 통해 각 프레임에 대한 충돌 위험도를 예측한다. 예측한 충돌 정보는 히트맵 시각화와 LLM 기반 설명으로 제공된다.



[그림 1] 제안 방법 개요

제안한 시스템은 그림 1과 같다. 첫째, 충돌 위험 분류 단계에서는 실시간 영상 프레임을 입력으로 받아 충돌 위험 여부를 분류한다. 둘째, 시각적 설명 제공 단계에서는 분류 모델의 내부 판단 근거를 시각적으로 확인하기 위해 Grad-CAM 기반 시각적 설명을 생성한다. 셋째, 충돌 위험 예측 단계에서는 히트맵에서 강조된 영역의 공간적 정보를 활용하여 충돌 위험도를 정량화한다. 마지막으로, 자연어 설명 단계에서는 히트맵 위치 정보와 위험도를 LLM의 입력으로 전달하여, 현재 위험 상황에 대한 직관적인 설명과 함께 안전한 회피 기동 방향을 자연어로 출력한다.

3.1. 충돌 위험 분류 단계

충돌 위험 분류 단계에서 사용한 충돌 분류 모델은 합성곱 신경망(Convolutional Neural Network, CNN) 구조를 기반으로 설계되었다. 입력 영상은 실시간 처리 효율을 고려하여 128x128 해상도로 리사이징 되며, 픽셀 값은 1/255 스케일링을 통해 [0, 1] 범위로 정규화된다. 이러한 입력 데이터 전처리 과정은 연산 부담을 줄이고 모델 학습 및 추론의 안정성을 확보하기 위함이다.

모델의 최종 출력 계층에서는 시그모이드(Sigmoid) 활성화 함수를 적용하여, 입력 프레임에 대해 0과 1 사이의 충돌 발생 확률 P를 산출한다. 본 단계는 실시간으로 입력되는 연속 영상 프레임을 프레임 단위로 분석하여, 각 시점에서 발생 가능한 즉각적인 충돌 위험 요소를 감지하는 데 초점을 둔다. 또한 설명 가능성 확보를 위해 모델 내부의 마지막 합성곱 계층(Conv2D)을 자동으로 탐색하여 Grad-CAM 적용을 위한 대상 계층(Target Layer)으로 지정한다.

3.2. 시각적 설명 제공 단계

시각적 설명 제공 단계에서는 예측된 충돌 위험 결과에 대해 Grad-CAM을 적용하여, 모델이 위험 판단에 활용한 영상 내 핵심 영역을 히트맵 형태로 시각화함으로써 판단 근거를 제공한다. 이를 위하여 Grad-CAM 기반 관심 영역(Region of Interest, ROI) 추출 및 최적화를 수행한다. 해당 단계는 단순한 시각적 설명 도구를 넘어 충돌 위험 영역 추출을 위한 분석 단계이다. Grad-CAM은 대상 클래스에 대한 그래디언트를

마지막 합성곱 계층의 특징 맵에 대해 계산한 후, 전역 평균 풀링(Global Average Pooling)을 통해 채널별 가중치를 도출하고, 이를 선형 결합하여 입력 영상 상의 중요 영역을 히트맵 형태로 생성한다.

기존 Grad-CAM은 전등과 같은 고휘도 배경 요소에 과도하게 반응하는 한계를 보이므로, 본 연구에서는 실제 물리적 장애물에 대한 민감도를 높이고 환경적 노이즈를 억제하기 위해 다음과 같은 후처리 및 최적화 과정을 적용한다.

첫째, 동적 임계값 처리(Dynamic Thresholding)를 통해 히트맵의 최대 활성값 대비 60% 이상의 영역만을 유효 활성 영역으로 정의하고 이진화한다. 이를 통해 상대적으로 기여도가 낮은 배경 영역을 제거한다.

둘째, 이진화된 히트맵에 대해 커널 크기 5x5의 형태학적 연산(Morphological Opening 및 Closing)을 적용하여 미세한 점 노이즈를 제거하고, 위험 객체의 공간적 형태를 보존한다.

셋째, 윤곽선 기반 필터링(Contour-based Filtering)을 수행하여 전체 영상 면적의 1% 미만에 해당하는 작은 활성 영역은 노이즈로 간주하고 제거한다. 이후 남은 영역 중 가장 유의미한 윤곽선을 기반으로 바운딩 박스를 산출하여 최종 충돌 위험 객체의 위치를 정의한다.

이와 같은 과정을 통해 Grad-CAM 히트맵은 단순한 시각적 설명을 넘어 충돌 위험을 판단하는 근거로 활용될 수 있으며, 나아가 후속 위험률 보정과 회피 기동 생성을 위한 정량적 공간 정보로 확장된다.

3.3. 충돌 위험 예측 단계

제안된 시스템은 ZED 2 스테레오 카메라에서 실시간으로 촬영한 영상을 입력으로 받아 충돌 위험을 예측한다. 먼저 수집된 스테레오 영상 프레임을 충돌 분류 모델의 입력 규격에 맞게 전처리하고, CNN 기반의 특징 추출 프로세스를 통해 영상 내 장애물을 탐지한다. 이후 최종 출력층의 그래디언트 정보를 역전파하여 Grad-CAM 히트맵을 생성함으로써 충돌 분류 모델이 충돌 여부를 이진 분류 형태로 판단하며, 이를 통해 최종적인 충돌 위험을 P를 산출한다. 또한 본 연구에서는 위험도의 시각도를 반영하기 위해 스테레오 카메라로 추정된 깊이(Depth) 정보를 활용하여 TTC를 계산한다. 구체적으로, Grad-CAM 히트맵 후처리를 통해 얻은 위험 영역(ROI) 내 깊이 값을 프레임 간 추적하여 거리 변화량을 산출하고, 이를 기반으로 상대 접근 속도를 근사함으로써 충돌까지 남은 시간을 산출한다.

3.4. 자연어 설명 단계

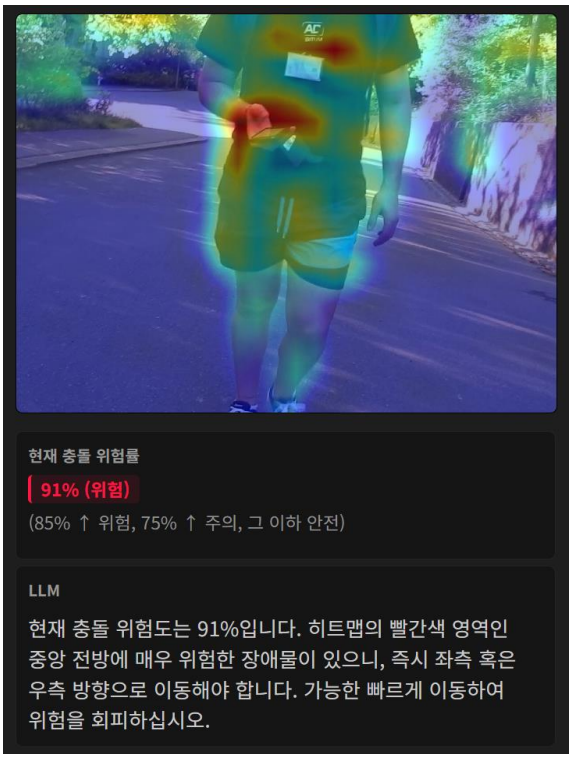
자연어 설명 단계에서는 대규모 언어 모델(LLM)을 활용하여 생성된 히트맵과 충돌 위험도를 함께 해석하고, 현재 위험 상황에 대한 설명과 함께 회피가 필요한 이동 방향을 자연어로 출력하여 조종자 또는 자율 시스템의 의사결정을 지원한다.

본 단계에서는 충돌 분류 모델을 통해 산출된 충돌 위험률과 Grad-CAM 히트맵에서 추출된 주요 활성 영역의 좌표 정보를 대규모 언어 모델(LLM)의 프롬프트 입력으로 통합하여 멀티 모달 해석을 수행한다. LLM은 히트맵으로부터 획득한 공간적 정보(Spatial Information)를 기반으로 위험 객체의 상대적 위치를 추론하고, 이를 바탕으로 현재 상황에서 가장 안전한 회피 기동 방향을 자연어 형태로 생성하여 제공한다. 이를 통해 단순한 위험 수치 제시에 그치지 않고, 사용자가 즉각적으로 이해하고 행동으로 옮길 수 있는 형태의 직관적인 위험 해석을 지원한다.

본 절에서는 제안한 LLM 기반 해석 방식의 동작 과정을 보다 명확히 설명하기 위해, 입력 데이터와 이에 대응하는 출력 예제를 프롬프트 구성 요소별로 제시한다. LLM의 입력 데이터는 Grad-CAM 히트맵에서 붉은색으로 강조된 주요 위험 영역의 위치 정보와 충돌 위험률로 구성되며, 해당 정보는 사전에 정의된 프롬프트 템플릿을 통해 전달된다. 이 프롬프트는 히트맵에서 강조된 영역과 생성되는 설명 간의 시각적 정합성을 유지하도록 설계되었으며, 기술적 용어 사용을 최소화하고 조종사의 즉각적인 판단과 회피 행동을 유도하는 방향 지시 중심의 응답을 생성하도록 구성되었다.

3.5. 최종 결과 화면 데모

그림 2는 이러한 제안 방법의 결과 화면 예제를 보여준다. 객체가 멀리 있다가 다가오게 되면 충돌 위험도를 보이면서, 조치 방법을 LLM이 설명한다.



[그림 2] 화면 예제

4. 실험 계획

4.1. 연구 질문 (Research Questions)

본 연구에서는 제안된 시스템의 성능을 검증하기 위해 네 가지 연구 질문을 설정한다.

4.1.1. RQ1. 제안한 충돌 분류 모델은 실제 충돌 위험을 얼마나 정확하게 예측하는가?

RQ1은 제안한 충돌 분류 모델이 실제 충돌 위험 상황을 얼마나 정확하게 예측하는지를 평가한다. 이는 다양한 고도와 조도, 장애물 크기 등 실제 비행 환경에서 발생할 수 있는 시나리오에 대해 모델이 도출한 위험 확률이 실제 충돌 가능성과 정량적으로 일치하는지 정확도(Accuracy), 정밀도(Precision), 재현율(Recall) 등의 평가 지표를 활용하여 모델의 성능을 검증하는 데 목적이 있다.

4.1.2. RQ2. 제안한 시스템의 Grad-CAM 히트맵은 모델의 판단 근거를 시각적으로 타당하게 반영하는가?

RQ2는 충돌 분류 모델이 판단의 근거로 제시한 Grad-CAM 히트맵이 실제 환경의 장애물 위치와 시각적으로 얼마나 일치하는지(정합성)를 평가한다. 이는 AI 모델이 단순히 우연에 의해 충돌을 예측하는 것이 아니라, 객체의 유의미한 특징을 정확히 학습했는지 검증하는 데 목적이 있다.

4.1.3. RQ3. Grad-CAM 기반 LLM 해석 결과는 모델의 예측 의도를 논리적으로 설명하는가?

RQ3는 시스템이 생성한 Grad-CAM 기반 LLM 해석 문장이 실제 상황을 논리적으로 타당하게 설명하고 있는지를 평가한다. 이는 LLM 해석이 수치적 위험도 및 시각적 근거와 모순 없이 일치하는지를 확인하여 시스템의 설명 신뢰도를 검증하는 데 목적이 있다.

4.1.4. RQ4. 제안한 시스템이 제공하는 충돌 판단 근거는 인간 관점에서 신뢰할 수 있는가?

RQ4는 제안한 시스템이 제공하는 충돌 판단의 근거가 인간 관점에서 얼마나 신뢰 가능한지를 정량적으로 검증하는 데 목적이 있다. 단순히 모델의 정확도가 높은 것을 넘어, 인공지능이 도출한 설명(Explanation)이 인간의 의사결정 과정에 긍정적인 영향을 미치는지 확인하고자 한다.

4.2. 실험 데이터셋

본 실험에서는 충돌 분류 모델의 학습을 위해 공개된 DroNet 데이터셋을 활용하였다[8]. DroNet은 심층 신경망을 통해 장애물 회피 및 주행 제어를 학습하기 위한 영상 데이터와 주행 정보를 포함한다. 이 데이터셋은 전방 카메라로 촬영한 영상에서 충돌 여부와 주행 관련 레이블을 함께 제공하며, CNN 기반 모델이 이미지 단위로 충돌 확률을 추정할 수 있도록 구성되어 있다.

실험에서는 DroNet 데이터 중 충돌 레이블이 포함된 부분을 분류 문제 학습용 데이터로 사용하였다. 각 영상 프레임은 RGB 이미지와 함께 충돌 여부(Collision/Non-Collision) 레이블을 가지고 있다. 표 1은 해당 데이터셋의 구성을 보여준다.

구분	디렉터리명	설명
학습 데이터	training/	충돌 분류 모델 학습에 사용되는 영상 시퀀스
검증	validation/	학습 중 모델 성능 검증에 사용되는
데이터		영상 시퀀스
테스트 데이터	testing/	최종 성능 평가 및 Grad-CAM 시각화 검증에 사용
각 시퀀스 폴더	Images/	전방 카메라로 촬영된 RGB 영상 프레임
	Labels.txt	각 프레임에 대한 충돌 여부 레이블 (Collision / Non-Collision)

[표 1] DroNet 데이터셋 구조

4.3. 실험 환경

실험 환경은 표 2와 같다. 본 실험은 ZED 2 스테레오 카메라를 사용하여 실험을 수행하였다. 카메라로부터 입력되는 실시간 고해상도 이미지와 깊이(Depth) 정보를 분석하여 충돌 위험을 판별하였으며, 충돌 분류 모델은 CNN 딥러닝 모델을 사용하였다. Grad-CAM 시각화 및 실시간 추론 과정은 NVIDIA GeForce RTX 2060 GPU 환경에서 수행되었으며, 실험 환경은 운영체제 및 라이브러리 간 호환성을 고려하여 Windows 11과 Python 3.10 기반으로 구성하였다.

Camera	ZED 2 – AI Stereo Camera
Vision SDK	ZED SDK
Data Interface	USB 3.0
Collision Classification Model	CNN-based model (.h5)
GPU	NVIDIA GeForce RTX 2060
Operating System	Windows 11
Language	Python 3.10

[표 2] 실험 환경

4.4. 실험 방법

RQ1을 위해서는 정확도 척도로 실험적으로 평가하였고, RQ2부터 RQ4까지는 사용자 평가로 진행하였다. 사용자 평가는 항공소프트웨어 분야에서 드론을 연구하는 연구원 7명을 대상으로 수행하였다. 평가자 구성은 학부생 2명, 석사과정 3명, 박사과정 2명으로 이루어져 있으며, 연구개발 경력의 평균은 5.57년, 표준편차는 2.44년으로 나타났고, 성별 구성은 남성 5명 여성 2명이었다.

4.4.1. RQ1: 충돌 분류 모델 정량적 평가

RQ1에서는 영상 프레임으로부터 충돌 위험을 얼마나 정확하게 식별하는지를 정량적으로 평가했다. 이를 위해 DroNet 데이터셋의 testing 세트를 활용하여 모델의 추론 결과와 실제 레이블을 비교하였다. 모델의 최종 출력은 시그모이드 함수를 통해 산출된 충돌 값 P이며, 임계값(Threshold)을 변화시키면서 분류 성능의 변화를 분석하였다.

모델의 정확도를 평가하기 위해 ROC(Receiver Operation Characteristic) Curve를 도출하고, 이에 대한 AUC(Area Under the Curve)를 측정하였다. 또한, 실제 운용 환경에서의 오분류 특성을 분석하기 위해 혼동 행렬(Confusion Matrix)을 생성하고, 정밀도(Precision), 재현율(Recall), 거짓 양성률(False Positive Rate) 등의 지표를 분석하였다. 안전성 확보를 위해 충돌 상황을 놓치는 미탐지(False Negative) 발생 여부를 검토하였다.

4.4.2. RQ2: Grad-CAM 히트맵의 시각적 정합성 검증

RQ2에서는 제안한 시스템에서 생성되는 Grad-CAM 기반 히트맵이 실제 충돌 위험 객체의 위치를 시각적으로 타당하게 반영하는지를 검증했다. 평가는 연속 영상 프레임을 기반으로 수행되었으며, 햇빛과 같은 고휘도 배경 요소가 존재하는 프레임을 의도적으로 포함하여 배경 노이즈 상황에서도 Grad-CAM 히트맵이 실제 장애물에 집중하는지를 확인한다.

Grad-CAM 히트맵의 시각적 정합성을 정성적으로 평가하기 위해 5점 리커트 척도(5-point Likert Scale) 기반의 평가 문항을 설계하였다. 실험은 항공소프트웨어 분야에서 드론을 연구하는 연구원 7명에게 ZED 2 카메라로 수집된 실제 충돌 상황 샘플을 제시한 후, 각 영상에 투영된 Grad-CAM 히트맵 시각화 결과에 대해 표 3의 Q1, Q2 질문에 대해 5점 리커트 척도(5-point Likert Scale)를 기반으로 평가를 진행하였다.

Q1	Grad-CAM의 히트맵이 실제 장애물 위치를 정확히 강조하는가?
Q2	전등, 벽 등의 백그라운드에 대한 강조 없이 핵심 영역에 집중되었는가?

[표 3] Grad-CAM 히트맵 시각적 정합성 평가 질문

4.4.3. RQ3: LLM 해석의 논리적 타당성 검증

RQ3에서는 Grad-CAM 히트맵과 ZED 2 스테레오 카메라로부터 추정된 깊이 정보를 입력으로 하여 LLM이 생성한 충돌 상황에 대한 해석 결과에 대해 논리적 타당성을 검증한다. LLM이 생성한 자연어 설명 평가는 객체 위치 식별의 정확성, 충돌 위험 판단의 일관성, 그리고 모델의 예측 결과와 설명이 논리적으로 연결되는지를 중심으로 분석한다.

본 연구에서는 LLM 해석의 논리적 타당성을 평가하기 위해 5점 리커트 척도 기반의 평가 문항을 설계하였다. 실험은 사용자들에게 시스템이 생성한 Grad-CAM 히트맵과 그에 따른 LLM의 설명을 동시에 제시한 후, 해당 결과물이 상황을 얼마나 적절히 대변하는지에 대해 표 4의 Q3, Q4 질문에 대해 5점 리커트 척도(5-point Likert Scale)를 기반으로 평가했다.

Q3	LLM이 설명한 장애물 위치에 대한 설명이 화면에 보이는 위치와 실제로 일치하는가?
Q4	LLM의 설명이 직관적이고 이해하기 쉬운가?

[표 4] LLM 해석의 논리적 타당성 평가 질문

4.4.4. RQ4: 충돌 판단 근거에 대한 신뢰성 검증

제안한 시스템이 제공하는 충돌 판단 근거가 인간 관점에서 의사결정에 활용 가능한 수준의 신뢰성을 가지는지 검증한다. 이를 위해 Grad-CAM 히트맵, 충돌 위험률, 그리고 LLM 기반 자연어 해석을 통합적으로 제시하여 평가를 수행한다.

본 연구에서는 충돌 판단 근거에 대한 인간 신뢰성을 측정하기 위해 5점 리커트 척도를 기반으로 질문을 설계하였다. 실험은 피험자들에게 '설명이 없는 단순 결과 알림'과 'Grad-CAM 및 LLM 해석이 포함된 결과 알림'을 비교 제시한 후, 후자의 시스템에 대해 표 4의 Q5, Q6를 중심으로 평가를 진행하였다.

연구원들은 각 프레임에 대해 Grad-CAM 히트맵이 제공하는 시각적 근거, 충돌 확률에 기반한 수치적 위험도, 그리고 LLM의 해석 결과를 함께 확인한 후, 해당 정보들이 충돌 위험 판단에 있어 직관적으로 이해 가능하며 신뢰할 수 있는 판단 근거를 제공하는지에 대해 평가했다.

Q5	본 시스템에서 제공하는 설명이 '왜 위험한지'를 납득하는 데 충분하였는가?
Q6	위험 확률만 제공하는 것보다 히트맵과 LLM을 함께 제공할 때 이해가 더 쉬웠는가?

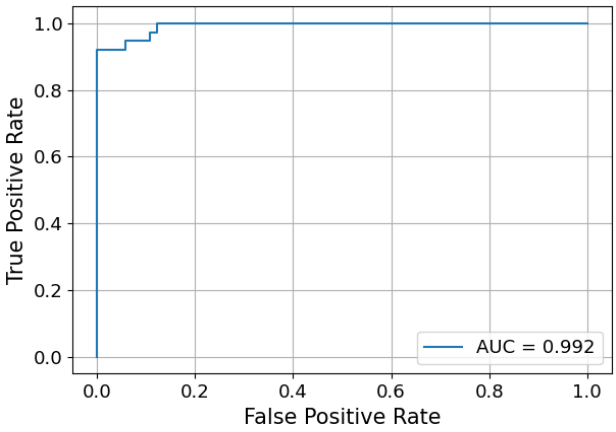
[표 5] 인간 관점 신뢰성 평가 질문

5. 실험 결과

5.1. RQ1: 충돌 위험 예측 성능 분석

그림 3은 제안한 충돌 분류 모델의 성능을 평가하기 위해 Validation 세트를 대상으로 산출한 ROC(Receiver Operation Characteristic) 곡선을 나타낸다. ROC 곡선은 분류 임계값 변화에 따른 거짓 양성률(False Positive Rate)과 참 양성률(True Positive Rate) 간의 관계를 시각화한 지표로, 모델의 전반적인 판별 능력을 평가하는 데 주로 사용된다.

그림 3에서 ROC 곡선은 좌측 상단에 밀집된 형태를 보인다. 이는 낮은 거짓 양성률을 유지하면서 높은 참 양성률을 달성하고 있음을 의미한다. 즉, 충돌 상황과 비충돌 상황을 효과적으로 구분하고 있음을 시사한다. 특히 AUC(Area Under the Curve) 값이 0.992로 나타나, 제안한 모델이 충돌 위험을 매우 높은 신뢰도로 판별할 수 있음을 정량적으로 확인하였다. 이는 분류 임계값 설정 변화에도 강건한 성능을 유지함을 의미하며, 실제 자율 비행 환경과 같이 충돌 발생 빈도가 낮고 클래스 불균형이 존재하는 상황에서도 안정적인 위험 예측이 가능함을 보여준다.



[그림 3] 제안한 충돌 분류 모델의 ROC 곡선

표 6은 검증 데이터셋에 대해 제안한 충돌 분류 모델의 예측 결과를 혼동 행렬(Confusion Matrix) 형태로 나타낸 것이다. 혼동 행렬은 실제 충돌 여부(True label)와 모델의 예측 결과(Predicted label)을 비교함으로써 분류 모델의 성능과 오류 유형을 직관적으로 분석할 수 있는 지표이다.

	Pred Safe	Pred Collision
True Safe	427	69
True Collision	0	38

[표 6] 충돌 분류 혼동 행렬

표 6에 따르면 실제 비충돌(Safe) 상황 중 427 프레임은 비충돌로 정확히 분류되었으며, 69 프레임은 충돌로 오분류되었다. 이는 일부 정상 상황을 충돌로 판단하는 False Positive 사례가 존재함을 의미한다. 이러한 오경보는 불필요한 회피 동작을 유발할 가능성은 있으나, 안전성을 최우선으로 고려하는 자율 비행 시스템의 관점에서는 잠재적 위험을 보수적으로 감지하는 허용 가능한 오류로 해석할 수 있다.

반면, 실제 충돌(Collision) 상황에 대해서는 38 프레임 모두 충돌로 정확히 분류되었으며, 충돌 상황을 비충돌로 잘못 예측한 False Negative 사례는 발생하지 않았다. 즉, 충돌 클래스에 대한 Recall 값이 1.0에 근접함을 의미하며, 실제 충돌 발생 가능성을 최대한 빠짐없이 감지하도록 설계된 모델의 목표가 효과적으로 달성하였음을 보여준다. 일부 False Positive가 존재함에도 불구하고, 충돌 상황을 놓치지 않는 보수적인 판단 전략은 실제 운용 환경에서 시스템의 전반적인 안전성을 향상시키는 데 기여할 수 있다.

따라서 그림 3과 표 6을 통해 제안한 충돌 분류 모델은 충돌 상황에 대한 높은 재현율을 기반으로 실제 충돌 위험을 안정적으로 예측할 수 있다. 이러한 특성은 이후 단계인 Grad-CAM 기반의 위험 영역 시각화 및 회피 기동 결정을 위한 입력 단계에서도 충분한 신뢰성을 제공할 수 있음을 의미한다.

5.2. RQ2: Grad-CAM 히트맵의 시각적 정합성 평가

5.2.1. 정량적 평가 결과

Q1은 Grad-CAM에서 제시하는 히트맵이 충돌 위험 객체의 위치를 적절히 반영하는지에 대해 질문한다. 이에 대한 평가자들의 평균 점수는 5점 만점에 4.2점이었다. 응답자들은 Grad-CAM 히트맵이 대체로 장애물 위치를 잘 강조한다고 평가하였으나, 흰색 울타리와 같은 평평한 장애물에서는 히트맵의 정확도가 낮아지는 경향을 보였다. 이는 False Negative(FN)가 다소 발생함을 의미하며, 제안된 시스템은 True Positive (TP)와 True Negative(TN)는 잘 작동했으나, False Negative가 발생하는 부분에서는 개선의 여지가 있음을 확인할 수 있었다.

Q2는 Grad-CAM에서 제시하는 히트맵이 핵심 영역에 집중되는지에 대한 질문으로 평균 점수는 4.4점으로 기록되었다. 응답자들은 Grad-CAM이 핵심 장애물 영역을 정확하게 강조했다고 평가하였다. 특히 전등, 벽 등 백그라운드 요소들이 불필요하게 강조되지 않고, 위험 객체에 집중된 히트맵을 생성한 것으로 나타났다. 이는 Grad-CAM이 고휘도 영역에 집중되지 않고, 위험 객체에 대해 정확히 주의를 기울였음을 보여주며 긍정적인 평가로 이어졌다.

종합적으로 Grad-CAM의 시각적 정합성은 실제 적용에 있어 충분한 활용 가능성을 보여주며, Q1에서는 4.2점, Q2에서는 4.4점을 기록하여 전체 평균 점수는 4.3점으로 나타났다. 핵심 장애물 강조에 대한 긍정적인 평가가 이어졌으며, 일부 평평한 장애물에서의 정확도 문제는 향후 개선이 필요한 부분으로 확인되었다.

5.2.2. Grad-CAM 시각화 결과 및 정성적 분석

그림 4의 좌측 이미지에서는 보행자의 몸통 중심부에 히트맵이 집중되어 있음을 확인할 수 있다. 해당 장면은 위험 객체가 존재하지 않는 상황으로 판단되어 특정 객체에 국한되지 않고 배경 영역에도 일정 수준의 주의가 분산되어 있음을 알 수 있다. 반면 우측 이미지에서는 배경 영역에 대한 히트맵 표현이 현저히 감소하고, 보행자의 몸통을 중심으로 강한 활성화가 나타나는 것을 확인할 수 있다. 이는 충돌 위험이 존재하는 상황에서 제안한 시스템이 충돌 가능 객체를 효과적으로 식별하고, 해당 객체의 핵심 위험 영역에 집중하여 히트맵을 생성함을 의미한다. 이러한 결과를 통해 제안한 시스템이 충돌 객체 및 위험 영역을 시각적으로 타당하게 강조함을 확인하였다.

[그림 4] 사람 객체에 대한 시스템 결과

그림 4의 좌측 이미지의 LLM 설명	
위험률	0% (안전)
LLM	현재 충돌 위험도는 0%입니다. 히트맵에서 중앙 전방에 빨간색 위험이 있어, 해당 지역을 피하고 반대 방향으로 이동해야 합니다. 중앙 전방을 피하기 위해 우회전하여 안전한 경로로 이동하세요.
그림 4의 우측 이미지의 LLM 설명	
위험률	91% (위험)
LLM	현재 충돌 위험도는 91%입니다. 히트맵의 빨간색 영역인 중앙 전방에 매우 위험한 장애물이 있으니, 즉시 좌측 혹은 우측 방향으로 이동해야 합니다. 가능한 빠르게 이동하여 위험을 회피하십시오.

[표 7] 그림 4 이미지에 대한 LLM 설명

그림 5는 좌측, 전방, 우측 모두에 장애물이 존재하는 상황을 나타낸다. 제안한 시스템은 스테레오 카메라를 통해 각 장애물까지의 거리를 추정하며, 그 결과 좌측 장애물이 가장 근접한 객체임을 인지하여 해당 영역의 히트맵을 가장 강하게 강조한 것을 확인할 수 있다. 이후 우측 이미지에서는 장애물과의 거리가 더욱 감소함에 따라, 좌측 장애물에 대한 히트맵 강조 정도가 유지되면서 전방 장애물에 대한 히트맵 강조 또한 유사한 수준으로 증가하는 양상을 보인다. 이는 충돌 위험이 상대적으로 높은 다수의 객체가 동시에 존재하는 상황에서도, 제안한 시스템이 거리 기반 위험도를 반영하여 주요 충돌 위험 영역을 효과적으로 구분하고 강조함을 의미한다.

5.3. RQ3: LLM 기반 충돌 해석의 논리적 타당성 평가

5.3.1. 정량적 평가 결과

Q3는 충돌 위험에 대한 LLM의 설명이 이미지와의 일치 여부를 평가한다. 이에 대한 평균 점수는 5점 만점에 3.8점으로 평가되었다. 응답자들은 전반적으로 LLM이

[그림 5] 그 외 객체에 대한 시스템 결과

그림 5의 좌측 이미지의 LLM 설명	
위험률	89% (위험)
LLM	현재 충돌 위험도는 89%입니다. 히트맵의 빨간색 영역인 좌측에 매우 높은 위험 요소가 있습니다. 해당 지역을 피하고 반대 방향으로 즉시 이동해야 합니다.
그림 5의 우측 이미지의 LLM 설명	
위험률	94% (위험)
LLM	현재 충돌 위험도는 94%입니다. 히트맵의 빨간색 영역인 전방에 매우 위험한 장애물이 있으니, 후방으로 이동하여 회피해 주십시오.

[표 8] 그림 5 이미지에 대한 LLM 설명

충돌 위험이 높은 상황에서는 히트맵에 나타난 장애물 위치와 일치하는 설명을 제공한다고 평가하였다.

특히 True Positive(TP)에 해당하는 사례에서는 장애물의 위치와 방향에 대한 설명이 비교적 정확하게 제시되었다. 그러나 일부 사례에서는 충돌 위험도가 0%로 산출된 영상임에도 불구하고 LLM이 안전한 경로로의 이동을 제시하는 경우가 관찰되었다. 이러한 현상은 사람의 시각적 판단 기준에서 False Negative(FN)로 인식될 가능성이 있고, 특히 무인 드론 시스템의 안전성을 고려할 때 치명적인 요소로 작용할 수 있다는 점에서 상대적으로 낮은 점수가 부여되었다.

Q4는 LLM이 제시하는 설명이 직관적이고 이해하기 쉬운지 평가한다. 이에 대한 평균 점수는 4.8점으로 평가되었다. 응답자들은 LLM이 히트맵에서 얻을 수 있는 시각적 정보뿐만 아니라, 이후 취해야 할 행동을 자연어로 직관적으로 제시한다는 점에서 사용자에게 유용하다고 평가하였다. 특히 충돌 위험이 높은 상황에서는 환경에 부합하는 회피 지시가 제공되어, 전반적인 이해도를 크게 향상시키는 것으로 나타났다. 다만 일부 응답자들은 충돌 위험이 높은 상황에서 설명이 다소 길어 한눈에 파악하기 어렵다는 점과, 핵심 장애물 중심의 설명에 치중하여 주변 환경이나 전체 맥락에 대한 정보가 부족하다는 점을 한계로 지적하였다. 그럼에도 불구하고, LLM 기반 설명이 제공하는 직관성과 이해 용이성 측면에서는 전반적으로 긍정적인 평가를 받았다.

종합적으로 LLM 기반 충돌 해석 결과는 행동 지침 제공 측면에서 높은 직관성과 활용 가능성을 보였으나, 안전 상태로 판단된 상황에서의 LLM 해석 정합성 측면에서는 일부 개선이 필요한 것으로 나타났다.

5.3.2. LLM 해석 결과의 정성적 분석

그림 4의 좌측 이미지에서는 위험 객체가 탐지되지 않아 충돌 위험률이 0%로 산출되었으며, 이에 따라 시스템은 안전 상태로 판단하였다. 안전 상태에서는 가장 인접한 객체를 우선적으로 고려하되, 특정 객체에만 집중하지 않고 주변 배경 영역 전반을 함께 고려하여 잠재적인 충돌 경로를 종합적으로 판단하고 경로를 추천하는 양상을 확인할 수 있다. 반면, 우측 이미지에서는 충돌 위험이 존재하는 객체를 명확히 인지하고, 해당 객체를 회피하도록 지시하는 설명을 생성함을 확인하였다.

그림 5의 좌측 이미지에서는 스테레오 카메라를 통해 추정된 거리 정보를 바탕으로 좌측 객체가 가장 근접한 장애물임을 인지하고, LLM이 이를 설명하여 좌측의 반대 방향으로 회피해야 함을 제시하는 것을 확인할 수 있다. 반면, 우측 이미지에서는 장애물과의 거리가 더욱 감소함에 따라 좌측, 전방, 우측 모두에서 충돌 위험이 존재한다고 판단하여 후방으로 회피할 것을 제안하는 설명이 생성된다.

이 결과는 LLM이 모델의 예측 결과와 시각적 정보에 기반하여 상황을 논리적으로 해석하고, 그에 상응하는 행동 지향적 설명을 타당하게 생성함을 보여준다.

5.4. RQ4: 충돌 판단 근거에 대한 신뢰성 평가

5.4.1. 정량적 평가 결과

Q5는 제공된 LLM의 설명이 왜 위험한지를 설명하는지 충분이 납득이 되는지 질문한다. 이에 대한 평균 점수는 5점 만점에 4.6점으로 평가되었다. 응답자들은 충돌 위험이 높은 상황에서 Grad-CAM 히트맵과 LLM 설명이 결합되어 제공됨으로써, 단순한 위험 여부 판단을 넘어 충돌 위험이 발생하는 원인과 근거를 이해하는 데 효과적이라고 평가하였다. 특히 히트맵을 통해 시각적으로 위험 영역을 확인하고, LLM이 이를 자연어로 설명함으로써 사용자가 위험 상황을 납득하는 데 도움이 된다는 점에서 높은 점수가 부여되었다. 이러한 결과는 제안한 시스템이 충돌 판단의 설명 가능성(Explainability) 측면에서 사용자 신뢰 형성에 기여함을 시사한다.

Q6은 히트맵과 LLM의 정보를 제공했을 때 충돌 위험에 대한 아해가 더 쉬웠는지에 대한 질문이다. 이에 대한 평균 점수는 4.8점으로 나타났다. 응답자들은 충돌 위험 확률만 제시되는 경우보다 Grad-CAM 히트맵과 LLM 설명을 함께 제공할 때 상황 이해도가 현저히 향상된다고 평가하였다. 히트맵은 위험 판단의 시각적 근거를 제공하고, LLM은 해당 정보를 바탕으로 다음 행동에 대한 지침을 직관적으로 제시함으로써 사용자에게 실질적인 도움을 제공한 것으로 나타났다. 다만 일부 응답자는 LLM 해석에 사용되는 용어가 전공자 관점에서는 직관적이나, 일반 사용자에게는 의미를 즉각적으로 이해하기 어려울 수 있다는 점을 지적하였다. 그럼에도 불구하고, 전반적으로 히트맵과 LLM을 결합한 설명 방식은 사용자 이해도와 신뢰성을 효과적으로 향상시키는 것으로 평가되었다.

종합적으로 Grad-CAM과 LLM을 결합한 설명 방식은 충돌 위험에 대한 이해도와 신뢰성을 효과적으로 향상시킨다는 것을 보여준다. 특히 시각적 근거와 자연어 설명의 결합은 충돌 판단 과정의 투명성을 향상시키며, 자율 시스템의 안전성에 대한 신뢰를 제고하는 데 기여한다.

5.4.2. 정성적 분석

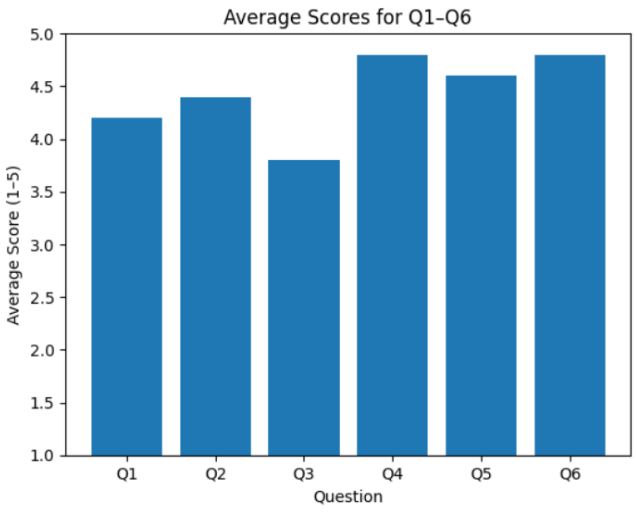
전반적으로 평가자들은 Grad-CAM 히트맵과 LLM 설명을 함께 제공할 경우, 충돌 위험 판단에 대한 이해도가 향상된다고 평가하였다. 특히 충돌 위험이 높은 상황에서는 히트맵을 통해 핵심 장애물 영역을 시각적으로 확인하고, LLM이 이를 바탕으로 회피 방향을 제시함으로써 다음 행동을 직관적으로 판단하는 데 도움이 된다는 의견이 다수 제시되었다.

반면, 히트맵의 강조 정확도에 대해서는 한계가 지적되었다. 평평하게 분포된 장애물이나 흰색 울타리와 같은 환경에서는 위험 요소가 충분히 강조되지 않거나 일부 영역에만 국한되는 경우가 있었으며, 이는 FN 발생 가능성과 관련된 신뢰성 저하 요인으로 언급되었다. 또한 LLM 설명이 직관적이라는 평가와 함께, '왜 위험한지'에 대한 구체적인 근거(거리, 상대적 위치, 이동 방향 등)가 충분히 제공되지 않는다는 의견도 제시되었다. 일부 평가자는 충돌 위험도가 낮게 산출된 상황에서도 회피 지시가 제시되는 점에서, 히트맵 기반 판단과 LLM 설명 간의 관계가 명확하지 않다고 지적하였다.

종합적으로 Grad-CAM과 LLM을 결합한 설명 방식은 충돌 위험에 대한 직관적 이해와 신뢰성 향상에 기여하였으나, 히트맵의 강조 정밀도와 설명의 근거 충분성 측면에서는 개선의 여지가 있다. 이러한 결과는 본 시스템을 충돌 '확률' 예측보다는 충돌 위험 '상황 인식' 관점에서 해석하는 것이 보다 타당함을 시사하며, 향후 연구에서는 시공간 정보를 포함한 보다 정교한 위험 판단 및 설명 기법이 필요함을 보여준다.

5.5. 정량적 평가 결과 종합 분석

그림 6은 Q1부터 Q6까지의 정량적 평가 평균 점수를 요약한 결과이다. 전반적으로 모든 항목에서 4점 이상의 평가를 기록하였으며, 이는 제안한 시스템이 각 RQ에 대해 긍정적인 평가를 받았음을 의미한다. 특히 히트맵과 LLM을 결합한 설명 방식에 대한 항목(Q5, Q6)에서 상대적으로 높은 점수가 나타났다. 이는 제안한 설명 구조가 사용자 이해도 향상에 효과적임을 확인할 수 있는 기반이 된다.



[그림 6] 정량적 평가 결과 종합

6. 결론

본 논문은 충돌 예측 모델과 Grad-CAM 기반 시각화, 그리고 LLM을 결합하여 해석을 생성하는 시스템을 제안하였다. 실험 결과, 제안한 시스템은 복잡한 배경 환경에서도 돌출된 장애물을 명확히 식별하고, 충돌 위험이

높은 객체 및 영역에 집중하는 특성을 보였다. 특히 Grad-CAM 시각화 결과와 LLM의 설명이 일관된 경향을 보이며, 모델의 예측 근거를 직관적으로 이해할 수 있도록 지원함을 확인하였다.

실험 결과, Grad-CAM 히트맵의 시각적 정합성과 핵심 영역 집중도에 대한 평가(Q1, Q2)는 평균적으로 높은 점수를 기록하였으며, LLM 기반 충돌 해석의 직관성과 이해 용이성(Q3, Q4) 또한 긍정적으로 평가되었다. 특히 히트맵과 LLM 설명을 함께 제공할 경우(Q5, Q6), 충돌 위험에 대한 인과적 이해와 사용자 신뢰도가 유의미하게 향상되는 것으로 나타났다. 다만 일부 평평한 장애물에 대한 히트맵 강조 부족이나, 안전 상태에서의 LLM 설명 정합성 측면에서는 개선의 여지가 확인되었다.

7. 향후 연구

향후 연구에서는 LLM이 제공하는 회피 방향 정보를 보다 적극적으로 활용하는 방안을 탐구하고자 한다. 예를 들어, 생성된 회피 방향을 실제 자율 주행 또는 자율 비행 시스템의 경로 계획 모듈에 직접 연동하거나, 다수의 회피 후보 경로를 생성하여 상황별 최적 경로를 선택하는 보조 의사결정 요소로 활용할 수 있을 것이다. 아울러 본 연구의 정량적 평가에서 확인된 한계점을 보완하기 위해, 일부 장애물에 대한 인식 부족과 안전 상태에서의 해석 정합성 문제를 고려한 시스템 확장 방향을 함께 검토하고자 한다. 이러한 접근은 설명 가능한 자율 시스템의 실질적 활용 가능성을 한층 강화하는 데 기여할 수 있을 것으로 판단된다.

8. 참고문헌

[1] Mumuni, F., & Mumuni, A., “Explainable artificial intelligence (XAI): from inherent explainability to large language models,” arXiv.org, 2025.

[2] D.-S. Song and J. H. Jung, “A Quantitative Comparison of LIME and SHAP using Stamp-Based Distance Method on Image Data,” Journal of KIISE (JOK), vol. 50, no. 10, pp. 906–911, 2023.

[2] Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D., “Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization,” International Journal of Computer Vision, vol. 128, no. 2, pp. 336–359, 2020.

[3] Wu, X., Zhao, H., Zhu, Y., Shi, Y., Yang, F., Liu, T., Zhai, X., Yao, W., Li, J., Du, M., & Liu, N., “Usable XAI: 10 Strategies Towards Exploiting Explainability in the LLM Era,” arXiv preprint arXiv:2403.08946, 2024.

[4] Wang, Y., Zhang, T., Guo, X., & Shen, Z., “Gradient based Feature Attribution in Explainable AI: A Technical Review,” arXiv preprint arXiv:2403.10415, 2024.

[5] Tempel, F., Groos, D., Ihlen, E. A. F., Adde, L., & Strümke, I.,
“Choose Your Explanation: A Comparison of SHAP and Grad-CAM in Human Activity Recognition,”
Applied Intelligence, vol. 55, pp. 1107–1125, 2025.

[6] Kolekar, S., Gite, S., Pradhan, B., & Alamri, A. (2022). Explainable AI in Scene Understanding for Autonomous Vehicles in Unstructured Traffic Environments on Indian Roads Using the Inception U-Net Model with Grad-CAM Visualization. Sensors, 22(24), 9677.

[7] Karim, M. M., Li, Y., & Qin, R. (2021). Towards explainable artificial intelligence (XAI) for early anticipation of traffic accidents. arXiv preprint arXiv:2108.00273.

[8] Loquercio, A., Maqueda, A. I., Del Blanco, C. R., & Scaramuzza, D., “DroNet: Learning to Fly by Driving,” IEEE Robotics and Automation Letters, vol. 3, no. 2, pp. 1088–1095, 2018.

Ko-LLM의 디버깅 성능 및 답변 품질 비교 분석

정수정, 정호연, 박효근, 김진대

서울과학기술대학교 컴퓨터공학과

beargryllsj@seoultech.ac.kr, jhy0744@seoultech.ac.kr, hyotaime@seoultech.ac.kr, jindae.kim@seoultech.ac.kr

A Comparative Analysis of Debugging Performance and Response Quality of Ko-LLM

Sujeong Jeong, Hoyeon Jeong, Hyogeun Park, Jindae Kim

Dept. of Computer Science and Engineering, Seoul National University of Science and Technology

요 약

대규모 언어모델(LLM)은 결함 위치 추적(FL)과 자동 프로그램 수정(APR) 등 소프트웨어 디버깅 분야에서 유의미한 성과를 보여주고 있으나, 한국에서 개발된 Ko-LLM의 디버깅 성능에 대한 체계적인 평가는 아직 부족하다. Defects4J v3.0.1의 169개 단일 라인 버그를 대상으로, FL·APR 태스크를 한국어와 영어 프롬프트에서 Ko-LLM 6종, Global Open-source LLM 2종, Commercial LLM 2종과 비교하였다. Ko-LLM의 성능은 전반적으로 Commercial LLM보다 낮았으나 일부는 Global Open-source 모델보다 소폭 우수하였고 대체로 프롬프트 언어에 따른 성능 편차가 확인되었다. 또한 응답 품질을 평가하는 지표를 제안하여 Commercial LLM은 가장 안정적인 품질을 보인 반면 Ko-LLM과 Global Open-source LLM은 언어 및 태스크에 따라 품질이 변동했다.

Abstract

We evaluate Korean-developed LLMs (Ko-LLMs) on software debugging tasks—fault localization (FL) and automatic program repair (APR)—using 169 single-line bugs from Defects4J v3.0.1 under both Korean and English prompts. Across six Ko-LLMs, two Global Open-source LLMs, and two Commercial LLMs, Ko-LLMs generally lag behind commercial models but can slightly outperform Global Open-source baselines, with notable prompt-language sensitivity. We also propose a response-quality metric; Commercial LLMs are most stable, while Ko-LLMs and open-source models vary more by language and task.

1. 서 론

대규모언어모델(Large Language Model, LLM)의 등장은 소프트웨어공학 분야에 큰 영향을 주었다. LLM은 매우 다양하게 활용되고 있으나 그 중 많은 주목을 받는 것은 LLM의 코드를 생성하는 능력과 이를 소프트웨어 개발에 이용하는 것이다. 소프트웨어 디버깅 과정은 개발자에게 많은 시간과 노력을 요구하는 작업이며, 이를 효율적으로 지원하기 위해 결함 위치 추적(Fault Localization, FL)과 자동 프로그램 수정(Automatic Program Repair, APR)과 같은 기법들이 제안되어 왔다. 이를 위해 LLM이 활용되어 유의미한 성과를 보여주고 있다 [1-7]. 또한 대한민국에서도 이런 추세에 따라 LLM의 개발을 위한 연구를 진행하고 관련된 기술을 확보하려는 노력이 진행되고 있다. 다수의 한국 기업이 LLM을 개발하여 오픈 소스로 공개하였으며 [18-23], 이런 Ko-LLM 모델들은 개발한 기업들의 평가에서 글로벌 모델들에 준하는 성능과 우수한 한국어 이해 능력을 보인다.

하지만 이런 Ko-LLM의 디버깅 성능에 대해서는 충분한 평가가 이루어지지 못하고 있다. LLM의 디버깅 성능과 관련된 연구에서는 주로 OpenAI의 GPT 계열 모델들이나[8-11] 다른 글로벌 오픈 소스 모델을 활용하여 연구를 진행한다[12-16]. Ko-LLM들은 이런 모델들에 비해 충분한 평가 기회를 얻지 못하고 있으며, 그에 따라 모델의 우수성 및 보완점 또한 제대로 알려지지 않고 있다.

본 연구에서는 Ko-LLM의 디버깅 능력을 FL과 APR 성능의 관점에서 평가하고 이를 글로벌 오픈소스 모델(Global Open-source LLM) 및 상업용 GPT 모델(Commercial LLM)과 비교하였다. 실험은 디버깅 연구에서 많이 활용되는 Defects4J v3.0.1[17]의 single-line 버그 169개를 한국어와 영어 프롬프트로 구성하여 FL과 APR의 성능을 비교 분석하는 방식으로 이루어졌다. 실제 실험에서는 총 10개의 모델들(Ko-

LLM 6개, Global Open-source LLM 2개, Commercial LLM 2개)에게 169개 버그에 대해 FL 및 APR을 한국어 및 영어로 요청하는 6,760개의 프롬프트를 제공했다. 이후 모델의 응답을 사람이 직접 확인하고 분석하여 답변의 정확성 및 품질을 평가하였다. 디버깅 성능을 분석한 결과 Ko-LLM의 상위 모델은 대체로 Global Open-source LLM보다 높은 FL 및 APR 성능을 보여주었으나, Commercial LLM보다 성능이 낮았다. FL에서 Commercial LLM의 정답률은 37.9~40.3%로 가장 높은 성능을 보이며, Ko-LLM의 상위 모델은 20.1~33.1%, Global Open-source LLM은 18.3~27.8%, Ko-LLM 하위 모델은 8.3~15.4%의 정답률을 보인다. APR에서도 마찬가지로 Commercial LLM의 정답률은 17.2~24.9%로 가장 높은 성능을 보이고, Ko-LLM의 상위 모델은 10.1~14.2% Global Open-source LLM의 4.1~12.4%, Ko-LLM 하위 모델은 0.6~9.5%의 정답률을 보인다.

프롬프트 언어에 따른 성능 차이로는, FL의 경우 대부분의 모델이 한국어에서 더 낮은 성능을 보였지만, 예외적으로 Commercial LLM은 FL의 정답률이 영어보다 한국어 프롬프트에서 5.3~4.2%p 높은 성능을 보인다. APR의 경우 대부분의 모델이 한국어에서 근소하게 증가하는 FL과 반대의 양상이 관찰되었다.

디버깅 성능 외에 추가로 모델의 응답에 자주 나타나는 답변 품질에 영향을 줄 수 있는 패턴을 식별 및 분석하고, 이를 토대로 모델 답변의 품질을 평가할 수 있는 지표(Quality Metric)를 개발하여 모델을 비교하였다. 프롬프트에서 코드만을 답변으로 제시하도록 요청하였지만, 모델의 응답이 해당 지시를 따르지 않는 경우가 많았다. 그 밖에도 프롬프트의 내용을 그대로 답하거나, 답변에 동일한 문구가 반복적으로 나타나거나, 질문에 사용한 언어와 다른 언어로 답변하는 등의 품질 저하 문제 등을 식별하였다. 각 유형이 전체 응답에서 차지하는 비율을 구해 답변 품질을 -1(부정적)에서 1(긍정적)까지 나타내는 지표를 개발하였다. Commercial LLM은 모두 0.997~1.000의 가장 큰 값으로, 안정적인 답변 품질을 보였다. 반면 Ko-LLM과 Global Open-source LLM은 이보다 낮은 값을 보이며, 다수의 모델이 APR에서 영어보다 한국어 답변 품질이 낮아지는 경향이 관찰된다. 예를 들어 Upstage의 SOLAR의 경우 영어 프롬프트에서 0.74의 높은 품질 값을 보인 반면 한국어에서 0.47의 수준으로 감소하였다.

본 연구는 그동안 충분히 평가되지 못한 Ko-LLM의 디버깅 성능을 평가하고, 이를 Global Open-source LLM 및 Commercial LLM와 비교하였다. 또한 모델의 응답에서 자주 등장하는 품질 저하 요소를 식별하고, 이를 이용해 답변의 품질을 평가하는 지표를 제시하였다. 이런 연구 결과는 추후 Ko-LLM의 문제를 보완하고 성능을 향상시키는데 도움을 줄 수 있으며, Ko-LLM을 활용하려는 연구자 및 개발자들에게 유의미한 정보를 제공할 수 있다.

본 연구의 핵심 기여를 정리하면 다음과 같다.

- Ko-LLM의 FL 및 APR 성능 평가 및 글로벌 오픈 소스 모델, 상업용 모델과의 비교 결과
- 한국어 및 영어 프롬프트 사용시 LLM 디버깅 성능 비교 결과
- 모델의 답변 품질 저하 요소 식별 및 답변 품질 평가를 위한 지표 제안

이후의 논문 내용은 다음과 같다. 2장에서 연구의 배경에 대해 간략히 소개하고, 3장에서 구체적인 실험의 방법 및 본 연구에서 사용된 데이터와 모델들에 대해 설명한다. 4장에서는 주요 실험 및 비교 분석 결과를 제시하고, 5장에서는 결과에 대한 논의와 함께 연구의 유효성에 영향을 줄 수 있는 요인을 짚어본다. 마지막으로 6장에서는 향후 연구와 함께 연구의 결론을 제시한다.

2. 배 경

최근 대규모 언어 모델(Large Language Models, LLM)의 성능이 급격히 향상됨에 따라, 소프트웨어 공학 분야에서도 다양한 작업에 LLM을 활용하려는 시도가 활발히 이루어지고 있다. 특히 프로그램 내 결함의 원인이 되는 코드를 추적하는 결함 위치 추적(FL)[1-3]과 프로그램의 결함을 자동으로 수정하는 자동 프로그램 수정(APR)[4-7]은 개발자의 생산성과 직결되는 핵심 과제로서, LLM의 적용 가능성에 대해 지속적으로 다양한 연구가 발표되고 있다.

이러한 흐름에 따라 기존 연구에서는 Commercial LLM의 GPT[8-11] 계열 모델이나 Global Open-source LLM[12-16]을 대상으로 FL 및 APR 성능을 평가하고, 기존 기법들과의 비교 분석을 수행해왔다. 그러나 이러한 연구의 대부분은 해외에서 개발된 LLM을 중심으로 이루어져 있으며, 국내에서 개발된 Ko-LLM의 디버깅 성능을 체계적으로 평가한 연구는 아직 제한적으로만 이루어졌다.

이에 본 연구에서는 Ko-LLM들을 대상으로 FL 및 APR 성능을 평가하고, 이를 Global Open-source LLM 및 Commercial LLM와 비교 분석함으로써, Ko-LLM의 소프트웨어 디버깅 태스크에서의 활용 가능성과 한계를 체계적으로 규명하고자 한다.

3. 실험 방법

본 실험은 FL과 APR 두 가지 태스크로 구성되어 독립적으로 시행된다. 또한, 각 태스크는 입력 언어를 영어와 한국어로 구분하여 언어별 성능 차이를 관찰한다.

3.1 실험 데이터

본 실험에는 Java 기반 버그 벤치마크인 Defects4J v3.0.1을 사용하였다. ThinkRepair[5] 연구에서 사용된 방식을 따라, 단일 코드 라인 수정의 single-line bug만을 선별하여 총 169개의 버그를 사용하였다. 이는 프롬프트 입력과 출력 토큰 수를 줄여 LLM 모델의 한정된 context window 제약을 피하기 위함이다. FL에서는 버그가 포함된 함수 전체를 입력으로 제공한 후 perfect fault localization을 요청했다. APR에서도 마찬가지로 함수 전체를 제공하고 주석으로 FL의 위치를 표시한 후 해당 라인 수정을 요청했다. 해당 버그들은 표 1에 나열하였다.

표 1. Defects4J single-line bugs

Project	Bug	Project	Bug	Project	Bug
Chart	5	Csv	5	JxPath	1
Cli	9	Gson	6	Lang	14
Closure	31	Jackson Core	5	Math	22
Codec	8	Jackson Databind	16	Mockito	8
Collections	3	JacksonXml	1	Time	3
Compress	5	Jsoup	27	**Total**	**169**

3.2. 모 델

표 2. Large Language Models for Debugging Evaluation

Group	Model Name (Model ID)	Max token length	Parameter
Ko-LLM	SOLAR-10.7B-Instruct-v1.0 (solar)	4,096	11.7B
	EXAONE-3.0-7.8B-Instruct (exaone)	4,096	7.8B
	hyperclovax/HyperCLOVAX-SEED-Text-Instruct-1.5B (hyperclovax)	131,072	1.5B
	Midm-2.0-Mini-Instruct (midm)	32,768	2B
	A.X-4.0-Light (ax)	16,384	8B
	kanana-1.5-8b-instruct-2505 (kanana)	32,768	8B
Global Open-source LLM	CodeLlama-7b-Instruct-hf (codellama)	16,384	7B
	Qwen3-4B-Instruct-2507 (qwen)	262,144	4B
Commercial LLM	gpt-4.1-nano(gpt-4.1)	1,047,576	N/A
	gpt-3.5-turbo(gpt-3.5)	16,385	N/A

본 실험에는 (1) 한국 기업이 개발한 오픈 소스 LLM(Ko-LLM), (2) 글로벌 오픈 소스 LLM(Global Open-source LLM), (3) 상용 API 기반 LLM(Commercial LLM)의 세 그룹으로 총 10개의 모델을 사용하였다. 오픈 소스 LLM의 모델 간 규모 차이를 최소화하기 위해 파라미터 크기가 12B 이하의 모델을 선별하였다. 하지만 상용 API LLM의 경우 정확한 파라미터 크기가 공개되지 않아, 비교적 소형 모델로 추정되는 gpt-4.1-nano와 코드 생성 실험에 주로 사용되는 gpt-3.5-turbo를 사용하였다.

표 2는 각 모델을 세 그룹으로 나누어 모델명(Model Name), 최대 토큰 길이(Max token length), 파라미터 크기(Parameter)를 정리하였다. 상용 API 모델의 경우 최대 토큰 길이와 파라미터 수가 공개되지 않아 명시하지 않았다.

```
주어진 에러를 일으키는 자바 코드의 결함 라인을 수정하세요.
수정된 라인을 간단한 코드 주석과 함께 제공하세요. 다른 추가 설명은 필요 없습니다.
<Example>
[에러]
expected:<1> but was:<0>

[테스트 코드]
public void testGetItems() {…

[결함 코드]
public int getItemCount() {
if (this.items == null) { // 이 라인에서 에러가 발생합니다.
…
[수정된 라인]
if (this.items != null) { // items 리스트가 초기화된 경우에만 실제 아이템 수를 반환

<Question>
[에러]
junit.framework.ComparisonFailure…

[테스트 코드]
public void testGenerateURLFragment() {…

[결함 코드]
public String generateToolTipFragment(String toolTipText) {
return " title=₩"" + toolTipText // 이 라인에서 버그가 발생합니다.
…
[수정된 라인]
```

그림 1. Prompt template for the APR in Korean

3.3. 프롬프트 구성

프롬프트는 한국어, 영어로 구분된 Instruction과 few-shot의 Example, 실제 문제인 Question의 세 부분으로 구성된다. Instruction은 모델에게 태스크의 목적과 출력 형식을 간략히 지시한다.

Question은 공통적으로 다음의 정보를 포함한다.

- [에러/error]: 실패한 테스트에서 발생한 에러 메세지
- [테스트 코드/test code]: 해당 에러를 유발한 테스트 코드
- [결함 코드/faulty method]: 오류가 포함된 메소드 전체

FL에서는 모델이 오류가 발생한 위치를 찾아내어 해당 코드 라인 한 줄(faulty line)만을 출력하도록 한다. APR은 오류가 발생한 위치를 메소드에 주석으로 제공하는 context-aware 프롬프트 방식을 사용하여, 모델이 해당 위치를 수정한 코드 라인(fixed line)만을 출력하도록 한다. 그림 1은 APR에서

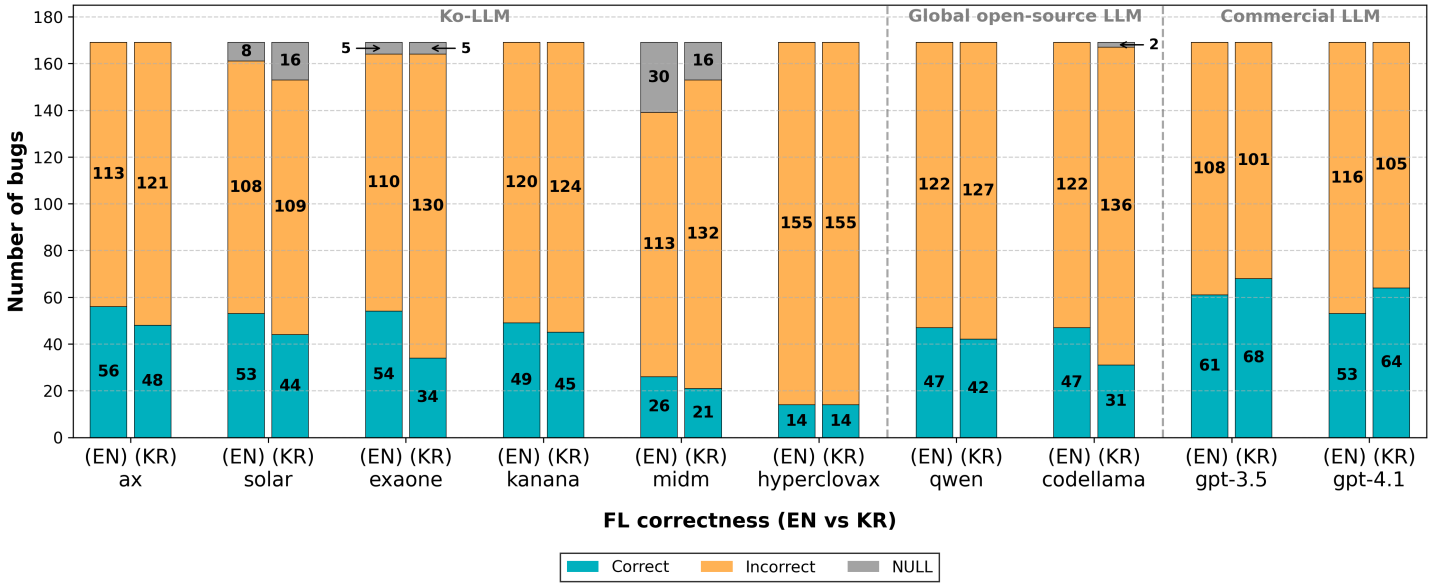


그림 2. FL Performance of LLMs for English and Korean Prompts

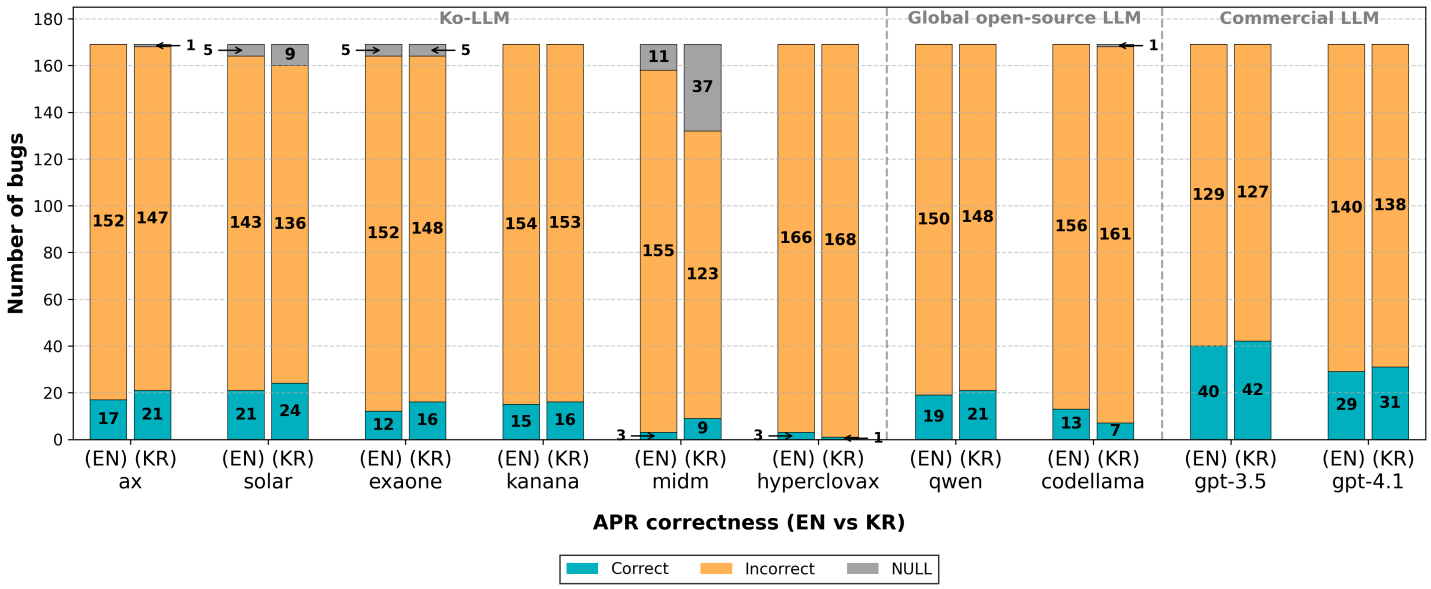


그림 3. APR Performance of LLMs for Korean and English Prompts

사용한 프롬프트 템플릿의 예시를 보여준다.

각 태스크의 shot 개수 별 성능을 고려하여 FL에는 3-shot, APR에는 1-shot 프롬프트를 적용하였다. 샘플 문제 9개로 0-3 shot 구성을 비교한 결과, 해당 설정에서 출력 형식과 성능이 비교적 가장 안정적이었다.

3.4. 평가 방법

본 연구에서는 FL과 APR 태스크 모두 모델의 응답을 Correct/Incorrect로 평가하였다. 또한 모델의 context window 크기가 부족하거나 응답이 없는 경우는 NULL로 평가하였다. 많은 모델에서 few-shot을 적용하더라도 불필요한 설명이나 추가 서술을 포함하여 응답하는 경우가 빈번해 자동 평가만으로는 정확한 판단이 어려웠다. 따라서 모든 결과 평가를 수동 평가 방식(manual evaluation)으로 진행하였다.

FL에서는 모델의 응답이 다른 라인을 포함하지 않고 실제 결함이 발생한 코드 라인을 정확히 포함하는 경우를 Correct로 평가하였다. APR에서는 모델이 개발자 패치와 동일한 경우와 완전히 일치하지 않지만 의미적으로 동일한 수정인 경우도 포함하여 Correct로 평가하였다.

4. 실험 결과

이 장에서는 Defects4J single-line 버그 169개에 대한 FL, APR의 실험 결과를 제시하고 모델 그룹별 및 프롬프트 언어에 따른 성능 차이를 분석한다.

4.1. 모델의 FL 성능 평가 결과

그림 2와 3은 각각 FL과 APR에 대한 세 그룹 Ko-LLM, Global Open-source LLM, Commercial LLM의 총 10개 모델의

응답을 평가한 것이다. 각 그룹은 점선을 추가해 구분하였다. 각 막대의 수치는 평가 영역에서 Correct, Incorrect 그리고 모델 응답 없음의 NULL로 분류된 버그의 개수를 나타낸다. EN은 영어 프롬프트, KR는 한국어 프롬프트를 사용한 경우이다.

FL에서 모든 모델 응답의 Incorrect 비중이 Correct보다 높으며, 가장 성능이 높은 것은 gpt-3.5이고 가장 성능이 낮은 것은 hyperclovax이다.

gpt-3.5 는 영어 프롬프트에서 61개(36.1%)의 Correct로 두 언어 모두 가장 우수한 성능을 보인다. 그 다음으로 영어 프롬프트에서 ax(56, 33.1%), gpt-4.1과 exaone(각각 54, 32.0%), solar와 gpt-4.1(각각 53, 31.4%), kanana(49, 29.0%), codellama와 qwen(각각 47, 27.8%)의 순으로 Correct 성능을 보인다.

한국어 프롬프트에서도 마찬가지로 gpt-3.5가 68개(40.3%)의 Correct로 가장 성능이 높으며, 그 다음으로 gpt-4.1(64, 37.9%), ax(48, 28.4%), kanana(45, 26.6%), solar(44, 26.0%), qwen(42, 24.9%), exaone(34, 20.1%), codellama(31, 18.3%)의 순으로 Correct 성능을 보인다. 반면 midm은 영어 26개(15.4%), 한국어 21개(12.4%)의 Correct로 비교적 성능이 낮다. 또한 hyperclovax는 한국어와 영어 프롬프트 모두 14개(8.3%)의 Correct로 가장 성능이 낮다.

NULL 응답의 경우, 영어 프롬프트에서 midm이 가장 많은 30개(17.8%)의 NULL값을 보이고, 한국어 프롬프트에서는 solar와 midm이 16개(9.5%)의 가장 많은 NULL값을 보인다. exaone은 영어와 한국어 프롬프트 모두 5개(3.0%)의 NULL값을 보인다. exaone과 solar는 최대 토큰 길이가 4,096으로, 다른 모델들에 비해 작은 context window를 가지므로 일부 프롬프트가 긴 경우에 모델이 응답 생성에 실패하였다. 하지만 midm은 최대 토큰 길이가 32,768로 충분하지만 가장 많은 NULL 응답을 보여, context window 외의 요인이 영향을 미쳤음을 짐작할 수 있다

모델 그룹별로 비교하면, 대부분의 Ko-LLM이 Commercial LLM보다 성능이 비슷하거나 낮지만 Ko-LLM의 상위 네 모델 ax, solar, exaone, kanana는 Global Open-source LLM보다 성능이 우수하다. exaone은 한국어에서 Correct가 34개(20.1%)로 qwen의 42개(24.9%)보다 적으나, codellama의 31개(18.3%) 보단 높으므로 FL의 Ko-LLM 상위 모델에 속할 수 있다. 그룹별 FL의 성능이 가장 높은 모델은 Ko-LLM에서는 ax(EN: 56, 33.1%, KR: 48, 28.4%), Global Open-source LLM에서는 qwen(EN: 47, 27.8%, KR: 42, 24.9%), Commercial LLM에서는 gpt-3.5 (EN: 61, 36.1%, KR: 68, 40.3%)이다.

> FL에서 Commercial LLM이 가장 성능이 높으며 Ko-LLM은 일부 모델(ax, solar, exaone, kanana)에서는 Global Open-source LLM과 유사하거나 높은 성능을 보이나, 나머지 하위 Ko-LLM 모델들은 성능이 저조하다.

4.2. 모델의 APR 성능 평가 결과

APR에서, FL과 마찬가지로 모든 모델의 응답이 Incorrect 비중이 Correct보다 높게 나타났다. FL에 비해 전반적으로 Correct 비율이 낮아, APR이 더 고난도 태스크임을 알 수 있다. 가장 성능이 높은 모델은 gpt-3.5 이고, 가장 낮은 성능이 낮은 모델은 hyperclovax이다.

영어 프롬프트에서 gpt-3.5는 40개(23.7%)의 Correct로 가장 우수한 성능을 보인다. 그 다음으로 gpt-4.1(29, 17.2%), solar(21, 12.4%), qwen(19, 11.2%), ax(17, 10.1%), kanana(15, 8.9%), codellama(13, 7.7%), exaone(12, 7.1%)의 순으로 Correct 개수가 나타났다. midm과 hyperclovax는 3개(1.8%)의 Correct로 가장 성능이 낮다.

한국어 프롬프트에서도 마찬가지로 gpt-3.5가 42개(24.9%)의 Correct로 가장 성능이 높으며, 그 다음으로 gpt-4.1(31, 18.3%), ax와 qwen(각각 21, 12.4%), solar(24, 14.2%), kanana와 exaone(각각 16, 9.5%), midm(9, 5.3%), codellama(7, 4.1%)의 순으로 Correct 성능을 보인다. hyperclovax는 1개(0.6%)의 Correct로 가장 성능이 낮다.

midm과 hyperclovax는 FL과 APR 모두 성능이 낮다. 이는 모델의 파라미터 크기가 각각 2B, 1.5B로 다른 모델에 비해 작기 때문으로 추정된다. 2B의 midm보다 작은 1.5B의 hyperclovax가 성능이 더 저조한 점에서 파라미터 크기가 2B 이하로 작아질수록 성능 또한 저조해짐을 알 수 있다. 반면, 다음으로 파라미터 크기가 작은 qwen(4B)는 두 모델에 비해 높은 성능을 보인다. 이는 파라미터 크기 외에도 모델의 구조와 같은 요인이 성능에 영향을 미친 것으로 추정된다. Commercial LLMs의 성능이 가장 높은 것도 파라미터가 다른 모델에 비해 크기 때문으로 예상된다.

NULL 응답의 경우, 영어 프롬프트에서는 midm이 11개(6.5%)로 가장 많았으며, 한국어 프롬프트에서는 midm이 37개(21.9%)로 크게 증가했다. exaone는 두 언어 모두 각각 5개(3.0%)의 NULL 응답을 보이며 FL과 동일한 비율을 유지한다. solar는 전체적으로 NULL 응답이 FL보다 3~7개의 감소를 보인다. midm은 FL보다 영어 프롬프트의 NULL 응답이 30(17.8%)에서 11개(6.5%)로 줄어든 반면, 한국어 프롬프트의 NULL 응답이 16(9.5%)에서 37(21.9%)로 늘어났다. midm은 FL뿐만 아니라 APR에서도 NULL 응답이 발생하였으며, 이는 특정 프롬프트 조건에서 모델의 응답 생성 안정성의 한계를 가질 가능성을 알 수 있다.

모델 그룹별로 비교하면, APR 태스크에서는 Commercial LLM와 다른 모델 그룹 간의 성능 격차가 FL보다 더 명확하게 나타난다. Ko-LLM의 ax, solar와 Global Open-source LLM의 qwen과 유사한 성능을 보이나 나머지 모델들(exaone, kanana, midm, hyperclovax)은 낮은 성능을 보인다. 그룹별 APR의 성능이 가장 높은 모델은 Ko-LLM에서는 ax(EN: 17, 10.1%, KR: 21, 12.4%), Global Open-source LLM에서는 qwen(EN: 19, 11.2% KR: 21, 12.4%), Commercial LLM에서는 gpt-3.5 (EN: 40, 23.7% KR: 42, 24.9%)이다.

APR에서 Commercial LLM이 다른 모델 그룹에 비해 뚜렷한 성능 우위를 보였다. Ko-LLM은 일부 모델(solar, ax)에서는 Global Open-source LLM의 qwen과 유사한 성능을 보이나, 나머지 모델들은 성능이 저조하다.

4.3 프롬프트 언어에 따른 성능 차이

프롬프트 언어에 따른 성능 차이를 FL과 APR 태스크별로 살펴보면 다음과 같다. 우선 FL에서, hyperclovax를 제외한 Ko-LLM와 Global Open-source LLM 모두 영어보다 한국어에서 성능이 떨어지는 반면 Commercial LLM은 성능이 향상되었다. hyperclovax는 영어와 한국어 모두 14개(8.3%)의 Correct로 프롬프트 언어에 따른 성능 차이가 없다. 다만 해당 모델은 영어와 한국어 모두에서 동일한 Correct 수를 기록하였음에도 불구하고, 전반적인 정답률 자체가 매우 낮다. 따라서 해당 모델이 프롬프트 언어에 대한 균형 잡힌 특성을 갖고 있음으로 해석하기에는 한계가 있다. 언어에 따른 성능 차이가 가장 큰 모델은 exaone이며, 영어에서 54(32.0%)개, 한국어에서 34개(20.1%)로 20개의 Correct 차이를 보인다. 그 다음으론 codellama가 47개(27.8%)에서 31개(18.3%)로 16개의 Correct 차이를 보인다. 이 외의 모델에서는 프롬프트 언어 간 약 4~11개의 차이를 보인다.

반면 APR 태스크에서는 FL과 달리, 대부분의 모델이 영어보다 한국어 프롬프트에서 성능이 높았다. hyperclovax와 codellama를 제외한 모든 모델이 영어보다 한국어 프롬프트에서 성능이 향상되었다. hyperclovax는 영어 3개(1.8%)에서 한국어 1개(0.6%)로 Correct수가 줄어들었고 codellama도 영어 13개(7.7%), 한국어 7개(4.1%)로 줄어들었다. APR에서 언어에 따른 성능의 차이가 가장 큰 모델은 codellama와 midm으로 한국어와 영어의 6개 Correct 차이를 보인다. 이외의 모델에서는 프롬프트 언어 간 약 2~5개의 Correct 수 차이를 보이며, FL보다 변화 폭이 작다. 이는 APR이 FL보다 난이도가 높아 전반적인 정답률이 낮기 때문에, 프롬프트 언어에 따른 성능 차이가 상대적으로 완화되어 나타난 것으로 해석할 수 있다. 다만, 한국어 프롬프트에서 일부 모델의 성능이 향상된 구체적인 원인에 대해서는 추가적인 분석이 필요하다.

해당 차이가 통계적으로 유의미한지 확인하기 위해 McNemar의 test를 한국어 및 영어 프롬프트 결과에 적용하였다. p-value값이 0.05 미만인 모델은 codellama, exaone, gpt-4.1으로, 각각 p-value가 0.0037, 0.0005, 0.0347이다. 이 중 codellama와 exaone은 영어 프롬프트, gpt-4.1은 한국어 프롬프트에서 더 우수한 성능을 보였다.

대부분의 모델은 FL에서 한국어 프롬프트의 성능이 저하되는 경향을 보였으나, Commercial LLM은 예외적으로 한국어 프롬프트에서 더 높은 성능을 보였다. FL과 달리, APR에서 대부분의 모델이 영어보다 한국어 프롬프트의 성능이 향상된다.

5. 논 의

5.1 Metric, 답변 품질 비교

FL과 APR의 모델 답변에서 코드 태스크 성능 외에 답변 품질에 관해 공통적으로 나타나는 양상을 발견했다. 이를 다섯 가지 유형으로 분류하여 정량적 평가를 위한 Metric을 제안한다.

- **CO (Code Only)**: 코드만 출력하고 설명은 포함하지 않은 응답
- **PR (Prompt Repeat)**: 입력 프롬프트 또는 그 일부를 반복 출력하는 응답
- **EO (Explanation Only)**: 코드 없이 설명만 제공하는 응답
- **LE (Language Error)**: 입력 언어와 다른 언어로 출력된 응답
- **R (Redundancy)**: 동일하거나 매우 유사한 응답을 반복 출력하는 경우

각 오류 유형은 전체 버그 수 대비 비율로 정규화하였다. CO는 모델의 답변이 코드만 출력하여 프롬프트의 지시를 잘 따른 유형의 품질 향상으로 본다. CO는 디버깅 자동화 파이프라인에서 후처리 없이 바로 적용 및 검증 단계로 전달할 수 있어 가장 직접적인 활용 가치를 갖는다. 따라서 CO에 자동화 처리 용이성에 대한 2의 가중치를 곱한다. 이외의 나머지 유형들은 품질 저하로 본다. 유형마다 전체 문제에 대해 나누어 답변 품질을 측정하는 Completion Quality(CQ)를 계산한다. 전체 버그 수를 N이라 하면, CQ의 범위는 [-3, 3]이므로 [-1, 1]범위가 될 수 있도록 $\frac{1}{3}$을 곱하여 스케일링하였다. CQ의 값이 1에 가까울수록 답변의 품질이 높음을, -1에 가까울수록 품질이 낮음을 의미한다.

$$CQ = \frac{1}{3} * (1 - \frac{LE + PR + EO + R - CO * 2}{N})$$

그림 4는 FL-EN, FL-KR, APR-EN, APR-KR의 네 가지 태스크에 대해 전체 모델 10개의 답변 품질을 해당 CQ로 계산한 결과이다. 각 그룹은 점선으로 구분하였다.

Commercial LLM은 모든 태스크에서 가장 높은 Quality Metric 값을 기록하였다. gpt-4.1은 모든 태스크에서 CO 조건을 만족하여 만점을 기록하였다. gpt-3.5는 APR-EN에서 1건의 EO를 제외하고 대부분의 응답에서 CO를 만족하여 전반적으로 안정적인 품질을 보였다. 이는 Commercial LLM이 프롬프트 이해도와 형식 준수 측면에서 높은 일관성을 유지함을 보여준다.

그 다음으로 ax와 solar가 상대적으로 높은 CQ 값을 기록하였다. ax는 FL-KO에서 0.925, sola 또한 FL에서 EN과 KO 모두 0.78 이상의 높은 CQ값을 보이나 APR-KO에서 두 모델 모두 0.473~0.487으로 낮은 값을 기록하였다. midm은 0.416~0.487으로 언어에 따른 답변 품질 차이가 작다는 것을 알 수 있다. codellama은 FL-EN이 PR의 빈도가 높아 0.205로 가장 낮은 값을 보이나 이외의 응답은

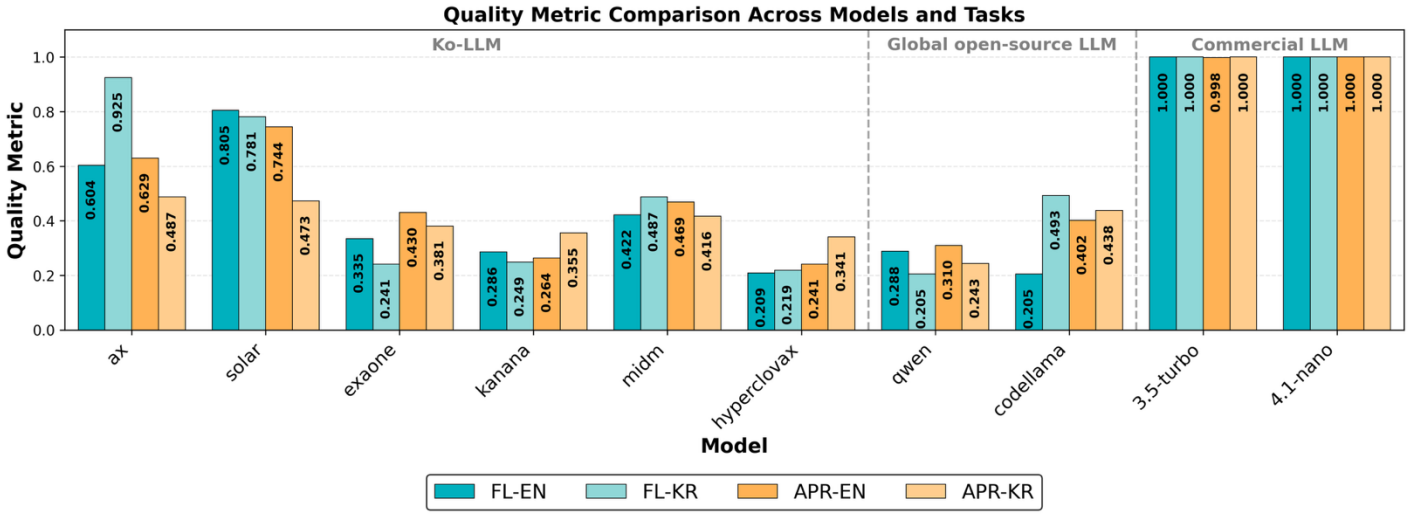


그림 4. Quality Metric Comparison Across Models and Tasks

0.402~0.493의 CQ값을 갖는다. exaone, kanana, hyperclovax, qwen은 FL과 APR 태스크의 모든 언어에서 CQ가 대체로 0.205~0.430 범위에 분포하였으며, 부분 응답(PR)이나 불필요한 출력(EO)이 자주 나타났다.

프롬프트 언어에 따른 영향은 모델에 따라 다르게 나타난다. FL에서는 한국어 프롬프트 사용 시 CQ가 향상되거나 저하되는 양상이 공통적으로 보인다. 반면 APR에서는 kanana와 hyperclovax를 제외한 Ko-LLM의 CQ값이 영어보다 한국어 프롬프트에서 감소하였다. APR에서 한국어 프롬프트 응답의 정답률은 영어보다 높은 반면, 답변 품질은 떨어지는 것을 알 수 있다. 이는 Ko-LLM이 APR과 같이 난이도가 높은 코드 태스크에서 한국어 프롬프트를 처리할 때 답변 품질이 상대적으로 저하될 가능성을 시사한다.

Quality Metric을 통해 모델의 정답률 비교를 넘어 답변 품질을 평가할 수 있다. 예를 들어, ax모델은 기반 모델 계열로 알려진 qwen에 비해 모든 태스크의 답변이 유사하거나 더 높은 Quality Metric 값을 기록한다. 이를 통해 ax가 기반 모델보다 답변 품질에 개선되었음을 알 수 있다. 이러한 차이는 단순 정확도 지표만으로는 관찰하기 어렵지만, Quality Metric을 통해 구체적으로 드러난다. 따라서 Quality Metric은 모델의 언어 적합성 및 답변 품질을 분석하는 데 유효한 평가 지표임을 입증한다.

5.2 연구의 유효성

본 연구의 유효성에는 몇 가지 제한점이 존재한다. 우선, LLM의 FL 및 APR 성능 평가는 사람이 직접 해석하여 판단하는 방식으로 이루어졌다. 평가 과정에서 주관이 개입될 가능성이 있으나, 사전에 정의된 정답과의 비교를 기반으로 평가하여 주관적 편차를 최소화하였다.

또한, Defects4J는 공개 벤치마크이므로 일부 LLM이 사전 학습 과정에서 유사한 코드 패턴이나 버그 수정 사례를 접했을 가능성이 있다. 본 연구에서는 이러한 데이터 유출 가능성을 완전히 제거하지는 못하였으나, 비교적 최신 버전의 버그를 사용함으로써 그 영향을 최소화하고자 하였다.

마지막으로, Defects4J의 single-line 버그만을 사용했기 때문에 다른 유형의 버그나 언어, 혹은 상업적 소프트웨어 환경에 본 연구의 결과를 그대로 일반화하는 데에는 한계가 있다. 더불어 확률적 특성을 지닌 LLM을 대상으로 반복 실험을 수행하지 않아 일부 결과는 실행마다 변동될 가능성도 존재한다.

다만 본 연구는 동일한 조건에서 모델 간 상대적인 디버깅 성능의 비교에 초점을 두고 있으므로, 평가 결과는 Ko-LLM을 Global Open-source LLM 및 Commercial LLM과 비교 분석에 대한 의미 있는 시사점을 제공한다.

본 연구에서 분석한 LLM의 답변 결과는 GitHub에 공개되어 확인이 가능하다.[1]

6. 결 론

본 연구에서는 한국어 중심으로 개발된 Ko-LLM들의 디버깅 성능을 FL과 APR의 관점에서 분석하고, 이를 Global Open-source LLM 및 Commercial LLM과 비교 평가하였다. Defects4J 기반의 Java 버그로 실험한 결과, Ko-LLM의 상위 모델(ax, solar)은 전반적으로 Global Open-source LLM보다 높은 성능을 보였으나, Commercial LLM보다는 다소 낮은 성능을 나타냈다. 이는 Ko-LLM의 상위 모델이 Global Open-source LLM보다 높은 정답률로 의미 있는 경쟁력을 보여 디버깅 태스크에서의 활용 가능성을 확인하였다.

또한 Quality Metric 분석을 통해, Commercial LLM들은 언어 및 태스크에 관계없이 안정적인 응답 품질을 유지하는 반면 대부분의 Ko-LLM과 Global Open-source LLM들은 응답 품질이 상대적으로 낮게 나타났다. 특히 Ko-LLM은 한국어 데이터 학습 비중이 높은 모델이라 하더라도, 한국어 프롬프트에서 코드 디버깅 태스크의 응답 품질이 낮아지는 경우를 확인하였다.

[1] https://github.com/sukosmos/LLM-Debugging-Data

한편 본 연구는 공개 벤치마크인 Defects4J의 Java single-line 버그를 대상으로 실험을 수행하였으므로, 일부 LLM이 사전 학습 과정에서 유사한 코드 패턴이나 버그 수정 사례를 접했을 가능성이 있다. 이는 비교적 최신 버전의 버그를 실험 데이터로 사용하여 데이터 유출 가능성의 영향을 축소했다.

향후 연구에서는 보다 다양한 버그 유형과 멀티라인 수정 시나리오를 포함하고, 생성된 패치를 실제로 적용하여 테스트 통과 여부를 평가함으로써 LLM 기반 디버깅 성능을 종합적으로 분석할 계획이다.

참고 문헌

[1] Yang, Aidan ZH, et al. "Large language models for test-free fault localization." ICSE 2024, pp. 1-12.

[2] Kang, Sungmin, Gabin An, and Shin Yoo. "A quantitative and qualitative evaluation of LLM-based explainable fault localization." FSE 2024: 23 pages.

[3] Xu, Chuyang, et al. "Flexfl: Flexible and effective fault localization with open-source large language models." TSE 2025, vol. 51, no. 5, pp. 1455-1471.

[4] Xia, Chunqiu Steven, and Lingming Zhang. "Less training, more repairing please: revisiting automated program repair via zero-shot learning." ESEC/FSE 2022, pp 959-971.

[5] Yin, Xin, et al. "Thinkrepair: Self-directed automated program repair." ISSTA 2024, pp. 1274-1286.

[6] Hossain, Soneya Binta, et al. "A deep dive into large language models for automated bug localization and repair." FSE 2024, 23 pages.

[7] Behrang, Farnaz, et al. "DR. FIX: Automatically Fixing Data Races at Industry Scale." PLDI 2025, 28 pages.

[8] Lajkó, Márk, et al. "Automated program repair with the gpt family, including gpt-2, gpt-3 and codex." APR 2024, pp 34-41.

[9] Hu, Haichuan, et al. "Can GPT-O1 kill all bugs? An evaluation of GPT-family LLMs on QuixBugs." APR 2025. IEEE, 2025, pp. 11-18, doi: 10.1109/APR66717.2025.00007.

[10] Prenner, Julian Aron, Hlib Babii, and Romain Robbes. "Can OpenAI's codex fix bugs? an evaluation on QuixBugs." APR 2022, pp. 69-75,

[11] Sobania, Dominik, et al. "An analysis of the automatic bug fixing performance of chatgpt." APR 2023. IEEE, 2023, pp. 23-30, doi: 10.1109/APR59189.2023.00012.

[12] Ramos, Daniel, et al. "Are large language models memorizing bug benchmarks?." LLM4Code 2025. IEEE, 2025, pp. 1-8, doi: 10.1109/LLM4Code66737.2025.00005.

[13] K. Huang et al., "An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair," ASE 2023, Luxembourg, Luxembourg, 2023, pp. 1162-1174, doi: 10.1109/ASE56229.2023.00181.

[14] Xia, Chunqiu Steven, Yuxiang Wei, and Lingming Zhang. "Automated program repair in the era of large pre-trained language models." ICSE 2023. IEEE, 2023.

[15] Huang, Kai, et al. "Comprehensive Fine-Tuning Large Language Models of Code for Automated Program Repair." TSE 2025, vol. 51, no. 4, pp. 904-928, doi: 10.1109/TSE.2025.3532759.

[16] Jiang, Nan, et al. "Impact of code language models on automated program repair." ICSE 2023, pp. 1430-1442, doi: 10.1109/ICSE48619.2023.00125.

[17] Just, René, Darioush Jalali, and Michael D. Ernst. "Defects4J: A database of existing faults to enable controlled testing studies for Java programs." ISSTA 2014, pp. 437-440.

[18] Upstage, "SOLAR-10.7B-Instruct-v1.0," Hugging Face. [Online]. Available: https://huggingface.co/upstage/SOLAR-10.7B-Instruct-v1.0 Accessed: 5 January 2026

[19] LG AI Research, "EXAONE-3.0-7.8B-Instruct," Hugging Face. Accessed: 5 January 2026 [Online]. Available: https://huggingface.co/LGAI-EXAONE/EXAONE-3.0-7.8B-Instruct Accessed: 5 January 2026

[20] Naver, "HyperCLOVAX-SEED-Text-Instruct-1.5B," Hugging Face. [Online]. Available: https://huggingface.co/naver-hyperclovax/HyperCLOVAX-SEED-Text-Instruct-1.5B Accessed: 5 January 2026

[21] K-intelligence, "Midm-2.0-Mini-Instruct," Hugging Face. [Online]. Available: https://huggingface.co/K-intelligence/Midm-2.0-Mini-Instruct Accessed: 10 January 2026

[22] SK Telecom, "A.X-4.0-Light," Hugging Face. [Online]. Available: https://huggingface.co/skt/A.X-4.0-Light Accessed: 5 January 2026

[23] Kakao, "Kanana-1.5-8B-Instruct-2505," Hugging Face. [Online]. Available: https://huggingface.co/kakaocorp/kanana-1.5-8b-instruct-2505 Accessed: 5 January 2026

[24] Meta, "CodeLlama-7B-Instruct," Hugging Face. [Online]. Available: https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf Accessed: 5 January 2026

[25] Alibaba, "Qwen3-4B-Instruct-2507," Hugging Face. [Online]. Available: https://huggingface.co/Qwen/Qwen3-4B-Instruct-2507 Accessed: 5 January 2026

[26] OpenAI, "GPT-4.1-nano," OpenAI Documentation. [Online]. Available: https://platform.openai.com/docs/models/gpt-4.1-nano Accessed: 5 January 2026

[27] OpenAI, "GPT-3.5-turbo," OpenAI Documentation. [Online]. Available: https://platform.openai.com/docs/models/gpt-3.5-turbo Accessed: 5 January 2026

MCP 기반 멀티 에이전트 클라우드 DevSecOps 환경에서의 보안·규제·비용 통합 대응 워크플로우

김수민 김영서 심희윤 장예린
이화여자대학교 사이버보안전공
minsoom48@gmail.com yseo722@ewha.ac.kr entrance1002@gmail.com yelin1197@ewhain.net

An MCP-Based Integrated Response Workflow for Security, Compliance, and Cost in a Multi-Agent Cloud DevSecOps

Soomin Kim*, Yeongseo Kim*, Heeyoon Shim*, Yelin Jang*
Department of Cyber Security,Ewha Womans University
* These authors contributed equally to this work.

요 약

본 연구에서는 보안 이벤트 대응을 단순한 탐지 문제가 아닌 운영 의사결정의 신뢰성 문제로 재정의하고, MCP(Model Context Protocol) 기반 다중 에이전트 오케스트레이션을 통해 보안·규제·비용 요구사항을 통합적으로 고려하는 클라우드 DevSecOps 대응 워크플로우를 설계한다.

주제어: DevSecOps, Multi-Agent Architecture, MCP Orchestration, Runtime Security Decision, Cloud Security Automation, Compliance-aware Response

1. 서 론

클라우드 컴퓨팅의 확산으로 인해 보안 이벤트의 규모와 복잡도는 급격히 증가하고 있다. 현대의 보안운영(SecOps) 환경에서 탐지 이후의 핵심 과업은 무엇을 언제 어떤 수준으로 조치할 것인가를 고민하여 서비스의 품질과 신뢰성 문제를 해결하는 것이다. 많은 조직들이 효율성을 위한 자동화 방식으로 SOAR(Security Orchestration, Automation, and Response)를 도입했으나, 실무 현장에서는 보안 관제 요원에 의한 수동 대응이 선호되는 경우가 많다.

이러한 자동화에 대한 기피 현상은 기술적 부재라기보다는 다음과 같은 구조적 한계에 기반한다.
1) 룰 기반 대응의 경직성: 사전에 정의된 정적 조건만으로는 동적인 클라우드 환경의 예외 상황에 대처하기 어렵다.
2) 오탐 및 과잉대응으로 인한 손실: 단순한 위협 탐지에 기반한 자동 차단 시스템은 오탐 발생 시, 서비스 가용성 저해 및 복구 비용 발생이라는 운영적 손실을 초래한다.
3) 자동화된 대응 결과에 대한 논리적 근거 부재: 보안 규제 준수를 위한 감사 대응 및 사고 후 책임 소재 파악이 어렵다.

이처럼 기존 SOAR는 정형화된 절차에 따라 동작하므로, 새로운 위협 패턴이나 조직 특유의 비즈니스 컨텍스트를 런타임 환경에 유연하게 반영하는 데 한계가 있다. 보안 이벤트 대응은 단순한 기술적 조치를 넘어, 위협 심각도뿐 아니라 규제 준수(예: 접근 통제, 변경 관리, 감사 추적), 대응에 수반되는 자원 및 인력 비용(예: 차단으로 인한 서비스 영향, 운영 인력 투입), 그리고 비즈니스 기회 비용을 함께 고려해야 하는 다차원 복합 의사결정 문제이다. 그러나 현재의 자동화 체계는 이러한 다양한 제약 조건을 통합적으로 검토하고 검증할 수 있는 구조를 제공하기 어렵다 [1].

본 논문은 이러한 한계를 극복하기 위해, 보안 이벤트 대응을 '탐지 정확도'의 문제가 아닌 '운영 의사결정의 신뢰성' 문제로 재정의하고, NIST CSF의 대응 체계를 따라 클라우드

DevSecOps 관점에서 보안·규제·비용 요구사항을 통합적으로 다루는 대응 워크플로우를 제안한다 [2]. 이 워크플로우 내에서는 MCP(Model Context Protocol)를 에이전트 간 통신 규약 및 책임 분리 인터페이스로 채택하여 시스템의 확장성과 안전성을 확보한다.
제안하는 아키텍처의 핵심 설계 원칙은 다음과 같다.

1) 역할 기반 Agent Boundary: 보안 분석, 규제 검토, 비용 예측 등으로 에이전트 역할을 세분화하여 복합적인 요구사항을 병렬적으로 처리한다.
2) 판단–실행 분리: LLM 기반 에이전트는 의사결정 및 논리적 근거 생성만을 담당한다.
3) 이벤트 기반 서버리스 오케스트레이션: 클라우드 네이티브 환경에 최적화된 트리거 구조를 통해 확장성과 비용 효율성을 달성한다.
본 연구의 기여는 다음과 같다.
1) 보안·규제·비용 요구사항을 통합하는 DevSecOps 대응 워크플로우 아키텍처를 설계한다.
2) MCP Orchestrator를 중심으로 책임 분리 및 Agent Boundary 설계 방안을 제시한다.
3) GuardDuty 기반의 IAM 액세스 키 유출 행위 대응 시나리오와 AMI 외부 공개·공유 시도 탐지 시나리오로 워크플로우를 검증한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구를 고찰하고, 3장에서는 제안 시스템의 전체 아키텍처와 설계 철학을 상세히 설명한다. 4장에서는 개별 에이전트의 구체적 설계와 MCP 활용 방안을 기술하며, 5장에서는 두 개의 시나리오에 기반한 구현 현황 및 자체 평가 결과를 제시한다. 마지막으로 6장에서 결론과 향후 연구 방향을 논의한다.

2. 관련 연구

2.1 LLM 기반 멀티 에이전트 시스템

최근 LLM의 추론 능력을 활용하여 복잡한 과업을 여러 전문 에이전트로 분할하고 협업시키는 멀티 에이전트 프레임워크가 제안되고 있다 [3]. 특히 역할 기반 협업과 구조화된 산출물을 강조한 접근은 코드 및 문서 생성 영역에서 그 가능성이 제시된 바 있다 [4].

보안 도메인에서도 취약점 분석이나 로그 해석에 LLM을 도입하려는 시도가 늘고 있으나, 보안 운영에는 단순한 텍스트 생성만이 아닌, 높은 수준의 신뢰성이 요구된다. 특히 클라우드 환경에서는 잘못된 자동화 조치가 서비스 전체 장애로 이어질 수 있으므로, 에이전트의 권한/책임 경계에 대한 명확한 설계와 감사 추적 기능이 필수적이다.

2.2 보안 자동화(SOAR)와 한계

상용 SOAR(Security Orchestration, Automation, and Response) 플랫폼은 탐지된 보안 이벤트에 대해 사전에 정의된 플레이북(workflow)을 실행함으로써 대응 절차를 자동화한다. 플레이북은 경보 유형과 조건에 따라 계정 비활성화, 자원 격리, 티켓 생성 등 일련의 조치를 순차적으로 수행하도록 구성된다.

예를 들어, 클라우드 환경에서 IAM 자격 증명(IAM Access Key)이 비정상적인 액세스 패턴을 보인다고 탐지되었을 때, SOAR 플레이북이 설정되어 있다면 “비정상 탐지 → 즉시 자격 증명 비활성화”와 같은 대응 절차가 자동으로 실행될 수 있다. 이는 SOAR가 정적 조건과 사전에 매핑된 조치에 따라 동작하도록 설계되었기 때문이다.

그러나 실제 운영 환경에서는 해당 IAM 자격 증명이 예정된 테스트나 배포 과정에서 일시적으로 높은 권한을 사용하는 정상 상황일 수 있다. 이 경우 자동화된 자격 증명 비활성화는 테스트 또는 배포 실패, 서비스 중단, 운영 리스크 증가로 이어질 수 있다. 이러한 상황은 플레이북이 설계될 당시 가정한 운영 맥락과 실제 런타임 환경이 달라졌을 때 발생하는 과잉 대응(overreaction)의 전형적인 사례를 보여준다.

이처럼 SOAR 기반 자동화는 탐지 이후의 대응을 사전에 정의된 실행 절차로 처리하기 때문에, 운영 맥락의 변화나 상황별 판단을 충분히 반영하지 못하고, 결과적으로 대응의 적절성이 저하될 수 있다.

2.3 MCP 기반 오케스트레이션

MCP는 모델/도구 간 상호운용을 위해 JSON 기반 메시지 구조와 호출 규약을 제공한다. 이 표준화된 호출 규약을 활용하면, Orchestrator가 각 에이전트의 접근 권한을 강제하는 구조를 구축할 수 있다. 본 논문에서는 MCP를 ‘에이전트 간 통신 중재 및 책임 분리 인터페이스’로 재해석하여 모든 에이전트 통신을 Orchestrator가 중재하도록 설계한다. 이를 통해 얻는 이점은 다음과 같다.
1) 통신 추적성: 모든 에이전트 간 교신 내용이 표준화된 형식으로 기록되어 보안 감사가 용이해진다.
2) 결합도 감소: 인터페이스를 준수하기만 하면 시스템 확장이 가능하다.
3) Agent Boundary: 특정 에이전트에서 오류가 발생하거나 비정상적인 권한 요청이 시도되더라도, 전체 대응 워크플로우의 안정성은 보장된다.

나아가 제안하는 MCP 기반 구조는 보안 정책 및 규정 가이드를 실시간으로 참조하는 RAG 및 대응에 소모하는 비용 측정을 위해 실시간 인프라 단가를 분석하는 FinOps 에이전트 등을 활용하여, 다양한 DevSecOps 요구사항을 수용할 수 있는 높은 확장성을 지니고 있다 [5].

최근 LLM 기반 시스템은 에이전트를 오케스트레이션하여 QoS를 최적화하는 방향으로 확장되고 있다. 본 논문은 이러한 접근과 달리, MCP Orchestrator를 정책 집행 노드로 활용하여 Agent Boundary와 판단–실행 분리를 적용함으로써 보안 운영 환경에서의 과잉 대응과 실행 리스크를 구조적으로 억제한다.

3. 제안 시스템 아키텍처

3.1 전체 시스템 개요

제안 시스템은 AWS 클라우드 환경에서 발생하는 보안 이벤트를 입력으로 받아, Event → Analysis → Decision → Runtime(Controlled Execution)의 4단계 파이프라인으로 동작한다. 이 파이프라인은 탐지, 분석, 의사결정, 실행 단계를 명확히 분리하여 보안 자동화 과정에서 발생할 수 있는 오류 전파와 오탐 대응 위험을 최소화하도록 설계되었다.

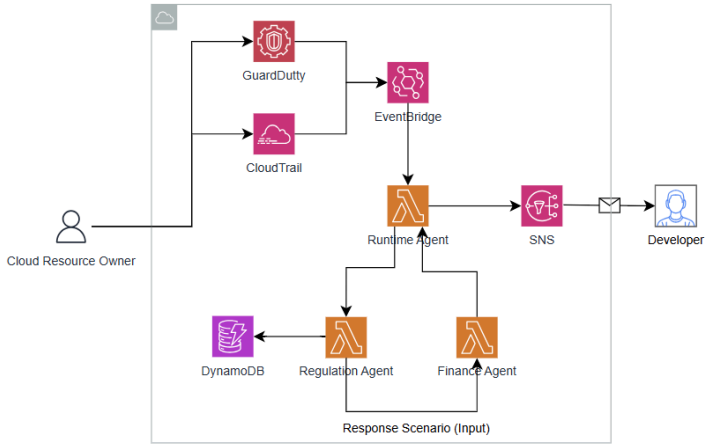


그림1. 다중 에이전트 기반 DevSecOps 대응 클라우드 서비스 아키텍처(Serverless 구조) (약어: MCP=Model Context Protocol, FinOps=Financial Operations)

본 시스템은 GuardDuty Finding과 CloudTrail 기반 보안 이벤트를 EventBridge로 수집·라우팅하고, MCP Orchestrator가 에이전트 호출 순서 및 메시지 스키마 검증을 통해 파이프라인을 제어한다. Runtime Agent는 이벤트 맥락을 분석하며, Regulation Agent는 규제 기반 대응 시나리오(Level 1–3)를 생성하고, Finance Agent는 시나리오별 비용 영향을 산정한다. 각 구성요소는 AWS Lambda에서 동작하며 사전 정의된 IAM Role에 따라 최소 권한으로 AWS API에 접근한다.

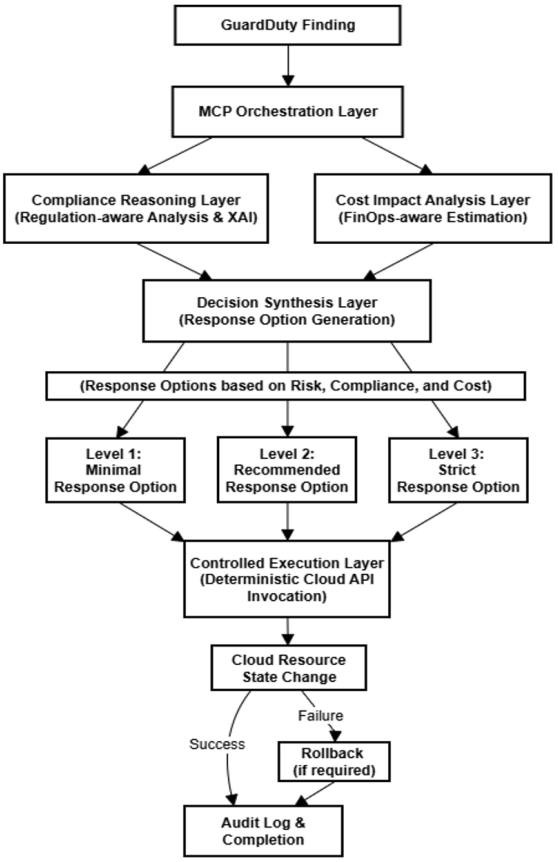


그림2. MCP Orchestrator 기반 멀티 에이전트 보안 대응 흐름 아키텍처

3.2 MCP Orchestrator

MCP Orchestrator는 제안 시스템의 핵심 제어 계층으로서, 모든 요청과 응답을 단일 진입점으로 수렴시킨다. 본 시스템에서 Orchestrator는 단순한 이벤트 전달자가 아니라, 보안 정책 집행과 에이전트 간 역할 및 실행 경계를 강제하는 중앙 제어 노드로 설계되었다. 이를 통해 에이전트 간 상호작용은 중앙에서 통제되며, 시스템 전반의 일관성과 안정성이 유지된다.

본 연구의 중요한 차별점은 보안 대응과 비용·서비스 가용성 간의 우선순위 충돌을 런타임 중재(arbitration) 알고리즘으로 해결하려 하지 않고, 해당 충돌이 발생하지 않도록 대응 단계를 구조적으로 분리한 오케스트레이션 설계를 채택했다는 점이다. Orchestrator는 기본 대응(Level 1)과 운영 영향이 수반되는 고강도 대응 (Level 2–3)을 명확히 구분함으로써, 자동화 단계에서 보안과 비용을 비교·판단하는 상황이 발생하지 않도록 한다. Level 1 대응은 로깅, 태깅, 알림, 스냅샷 생성과 같이 서비스 가용성이나 비용 판단을 요구하지 않는 최소 조치로 한정되어 Orchestrator의 통제 하에 자동 실행된다. 리소스 차단이나 권한 회수와 같이 운영 영향이 수반되는 대응은 Regulation Agent가 생성한 Level 2–3 플레이북 형태로만 정의되며, Orchestrator는 이를 즉시 실행 대상이 아닌 의사결정 후보로 취급한다.

Orchestrator는 Regulation Agent가 제시한 규제 근거 기반 플레이북과 Finance Agent가 산정한 시나리오별 비용 정보를 함께 사용자에게 제시하고, 명시적인 승인 절차가 완료된 경우에만 Runtime Agent의 결정론적 실행 모듈을 호출한다. 이와 같은 단계적 실행 통제 구조를 통해 본 시스템은 보안상 즉시 수행이 요구되는 조치는 자동화하되, 비용 및 서비스 가용성 판단이 개입되는 대응은 항상 인간 의사결정 하에 수행되도록 보장한다.

한편, Orchestrator는 각 에이전트의 입력과 출력에 대해 사전에 정의된 JSON 스키마를 기반으로 형식 검증을 수행함으로써, 구조적 오류나 비정상 응답이 파이프라인에 유입되는 것을 차단한다. 또한 대응 단계에서는 자동 실행과 수동 승인 모드를 구분하여 적용할 수 있도록 설계되어, 정책에 따라 사용자 개입 여부를 유연하게 결정할 수 있다. 모든 에이전트 호출 과정과 실행 결과는 중앙 로그로 기록되며, 이를 통해 사후 분석과 감사(audit)가 가능하도록 한다.

3.3 데이터 인터페이스 및 데이터 흐름

제안 시스템에서 모든 에이전트는 명확한 입력/출력 계약(interface contract) 을 JSON 형태로 정의한다. 이를 통해 에이전트 간 데이터 교환은 구조화되고, 의미적으로 해석 가능한 상태로 유지된다.

예를 들어 Runtime Agent의 출력은 다음과 같은 구조를 가진다.

```
{
  event_id,
  classification,
  recommended_actions[],
  rationale,
  risk_score,
  compliance_flags
}
```

이와 같이 분석 결과는 대응 추천, 규제 관련 플래그, 비용 영향 정보 등을 포함하여 구조화된다. Orchestrator는 각 단계에서 스키마 검증과 정책 검사를 수행하여, 정의되지 않은 필드나 허용되지 않은 요청이 실행 단계로 전달되지 않도록 차단한다. 이를 통해 데이터 흐름은 단방향·단계적 구조를 유지하며, 시스템의 추적 가능성과 안정성을 동시에 확보한다.

3.4 Agent Boundary 및 판단–실행 분리

LLM 기반 분석은 보안 이벤트의 맥락을 이해하고 대응 시나리오를 생성하는 데 강점을 가지지만, 출력 결과의 확정성과 결정론적 동작을 보장하기 어렵다. 이러한 특성을 고려하여 제안 시스템은 판단 단계(analysis/decision)와 실행 단계(action)를 명확히 분리하는 Agent Boundary 설계를 채택한다.

Runtime Agent는 보안 이벤트를 분석하여 위험 수준과 대응 시나리오를 '추천'하는 역할만 수행하며, 실제 인프라 변경에 대한 권한은 가지지 않는다. 실제 클라우드 자원 변경은 Runtime Agent 내부의 결정론적 실행 모듈(deterministic execution module) 에서 수행되며, 해당 모듈은 사전에 정의된 작업 카탈로그(whitelisted action catalog)에 포함된 작업만을 실행할 수 있다. 예를 들어 실행 모듈은 Security Group 또는 NACL 규칙의 추가·삭제, EC2 AMI 공유 권한 회수, 특정 리소스의 임시 격리 조치와 같이 명시적으로 허용된 작업만 수행한다.

허용된 작업 범위를 벗어나는 요청은 MCP Orchestrator 단계에서 사전에 차단되며, 실행 단계로 전달되지 않는다. 또한 에이전트별로 IAM 권한을 명확히 분리함으로써, 특정 에이전트가 침해되거나 오동작하더라도 피해 범위는 해당 역할로 국한된다.

이와 같은 판단–실행 분리와 Agent Boundary 설계는 LLM기반 자동화 시스템에서 발생할 수 있는 오판에 따른 즉각적인 자원 변경 위험을 구조적으로 차단하며, 시스템의 보안성과 운영 신뢰성을 동시에 확보한다.

4. 에이전트 상세 설계

본 장에서는 제안 시스템을 구성하는 세 가지 핵심 에이전트인 Runtime Agent, Regulation Agent, Finance Agent의 상세 설계를 설명한다. 각 에이전트는 분석, 규제 판단, 비용 평가라는 명확히 분리된 책임을 가지며, 모든 상호작용은 MCP 기반 오케스트레이션 계층을 통해 조정된다. 이러한 구조는 자동화된 보안 대응 과정에서의 과잉 대응을 방지하고, 규제 준수와 비용 효율성을 함께 고려한 신뢰 가능한 DevSecOps 의사결정을 지원하기 위한 것이다.

본 논문에서 에이전트는 단순한 LLM 호출 단위가 아니라, 규칙 기반 로직, 외부 도구 호출, 정책 및 데이터 소스를 결합한 시스템 수준 구성요소로 정의된다. LLM은 에이전트 내부에서 추론을 보조하는 수단 중 하나일 뿐이며, 의사결정과 실행의 주체는 MCP 오케스트레이션과 정책 계층에 의해 통제된다.

4.1 Runtime Agent(이벤트 분석 및 실행 제어)

Runtime Agent는 클라우드 환경에서 발생하는 보안 이벤트를 수신하여 전체 대응 워크플로우를 시작하고, 최종적으로 사용자가 선택한 대응 시나리오를 실제 클라우드 리소스 제어로 연결하는 중심 에이전트이다. 본 에이전트는 보안 이벤트 분석과 실행 제어를 담당하되, 규제 해석 및 비용 산정은 각각 Regulation Agent와 Finance Agent에 위임한다.

Runtime Agent의 처리 단계는 다음과 같다.

1) 이벤트 정규화

GuardDuty Finding 및 CloudTrail 이벤트를 수신한 후, 서로 다른 형식의 이벤트를 사전에 정의된 공통 이벤트 스키마로 변환한다. 이를 통해 이벤트 소스에 독립적인 분석이 가능하도록 하며, 이후 단계에서 일관된 처리 흐름을 유지한다.

2) 운영 맥락 수집

정규화된 이벤트를 기준으로 대상 리소스의 운영 맥락을 수집한다. 이 과정에서는 EC2 또는 AMI와 같은 리소스의 태그 정보, 계정 및 환경 구분(Prod/Dev), 네트워크 설정, 최근 설정 변경 이력, 과거 유사 이벤트 발생 여부 등이 함께 고려된다. 이러한 맥락 정보는 단순 탐지 이벤트와 실제 보안 사고 가능성을 구분하는 데 활용된다.

3) 규제 판단 트리거

Runtime Agent는 수집된 이벤트 정보와 운영 맥락을 Regulation Agent에 전달하여, 해당 이벤트가 어떤 규제 및 내부 정책과 연관되는지에 대한 판단을 요청한다. 이 단계에서 Runtime Agent는 규제 해석을 직접 수행하지 않으며, 규제 관련 판단을 전적으로 Regulation Agent에 위임한다.

4) 비용 평가 요청

Regulation Agent로부터 생성된 대응 시나리오(Level 1, Level 2, Level 3)를 입력으로 받아, Runtime Agent는 Finance Agent에 각 시나리오에 대한 비용 평가를 요청한다. 이를 통해 보안 대응이 클라우드 비용에 미치는 영향을 사전에 파악할 수 있도록 한다.

5) 대응 시나리오 제시 및 기본 대응 자동 실행

Runtime Agent는 Regulation Agent가 제시한 규제 근거와 Finance Agent가 산정한 비용 정보를 종합하여, 사용자에게 대응 시나리오를 제시한다. 이때 규제상 즉각 수행이 요구되는 기본 대응(Level 1) 은 자동으로 실행되며, 서비스 영향이나 추가 비용이 수반되는 대응(Level 2, Level 3)은 사용자 의사결정을 기다린다.

6) 사용자 의사결정 기반 추가 리소스 관리

사용자가 선택한 대응 시나리오는 Runtime Agent로 다시 전달되며, Runtime Agent는 boto3 기반 API 호출을 통해 해당 시나리오에 정의된 클라우드 리소스 제어 작업을 수행한다. 실행 결과는 로깅 및 감사 추적을 위해 기록되어 이후 분석과 검증에 활용된다.

4.2 Regulation Agent (규제 기반 대응 시나리오 생성)

Regulation Agent는 보안 이벤트를 규제 문서 및 내부 정책과 매핑하여, 규제 근거 기반 대응 시나리오를 생성하는 역할을 수행한다. 본 에이전트는 실행 권한을 가지지 않으며, 규제 해석과 대응 수준 정의에만 집중한다.

Regulation Agent는 먼저 Runtime Agent로부터 전달받은 이벤트를 이벤트 카테고리 단위로 분류한다. 이후 ISMS-P, ISO/IEC 27001, 클라우드 보안 가이드라인 등 사전에 정리된 규제 문서를 참조하여, 해당 이벤트 카테고리와 연관된 규제 조항을 식별한다 [2].

식별된 규제 조항을 근거로 Regulation Agent는 대응 강도에 따라 세 가지 수준(Level 1, Level 2, Level 3) 의 대응 시나리오를 생성한다. Level 1은 규제 준수를 위해 최소한으로 요구되는 기본 대응으로 정의되며, Level 2와 Level 3은 점진적으로 강도가 높아지는 리소스 관리 대응을 포함한다. 각 수준의 시나리오에는 적용 근거가 되는 규제 조항과 함께 권장되는 리소스 조치가 명시된다.

이러한 설계를 통해 Regulation Agent는 추상적인 규제 문서를 실행 이전 단계에서 활용 가능한 대응 시나리오로 변환하며, 이후 의사결정 과정에서 설명 가능성을 제공한다.

4.3 Finance Agent (비용 기반 의사결정 지원)

Finance Agent는 Regulation Agent가 생성한 각 대응 시나리오에 대해 클라우드 비용 관점의 영향을 평가하는 역할을 수행한다. 본 에이전트의 목적은 보안 대응이 비용 측면에서 합리적인지에 대한 의사결정 근거를 제공하는 데 있으며, 보안 정책이나 대응 수준을 변경하지 않는다.

Finance Agent는 각 대응 시나리오를 입력으로 받아 AWS Pricing API를 호출함으로써, 추가 리소스 사용이나 네트워크 제어로 인해 발생할 수 있는 예상 과금 비용을 산정한다. 이 과정에서는 EC2, 네트워크 트래픽, 보안 구성 변경 등 시나리오별 비용 발생 요소가 함께 고려된다.

비용 평가 결과는 정확한 금액 산출보다는, 시나리오 간 비교가 가능하도록 정리된 비용 정보와 비용 증가 원인 설명 형태로 반환된다. 이러한 정보는 Runtime Agent를 통해 사용자에게 전달되며, 보안 수준과 비용 간의 trade-off를 명확히 인식할 수 있도록 한다.

4.4 에이전트 오케스트레이션 및 실현 가능성

본 절에서는 앞서 설명한 Runtime Agent, Regulation Agent, Finance Agent 간의 상호작용 구조와 이를 실제 클라우드 환경에서 구현 가능한 형태로 배치하는 방식을 설명한다. 이벤트 분석, 규제 판단, 비용 평가, 실행을 단일 에이전트에 집중시키지 않고 단계적으로 분리함으로써 자동화 과정에서의 위험을 최소화하고, 사용자 개입이 필요한 지점을 명확히 정의한다.

보안 이벤트가 발생하면 Runtime Agent가 이를 수신하여 전체 워크플로우를 시작하고, 수집된 이벤트와 운영 맥락을 기반으로 Regulation Agent와 Finance Agent를 순차적으로 호출한다. Regulation Agent는 규제 문서에 근거하여 대응 강도가 상이한 세 단계(Level 1-3)의 대응 시나리오를 생성하며, Finance Agent는 각 시나리오에 대한 비용 영향을 비교 가능한 형태로 산정한다.

Runtime Agent는 규제 근거와 비용 정보를 종합하여 사용자에게 대응 시나리오를 제시한다. 이때 규제 준수를 위해 즉각 수행이 요구되는 기본 대응(Level 1)은 자동으로 실행되며, 서비스 가용성이나 추가 비용이 수반되는 대응(Level 2, Level 3)은 사용자 승인 이후에만 실행된다. 이러한 단계적 실행 구조를 통해 본 시스템은 자동화와 인간 의사결정 간의 균형을 유지한다.

설계 컴포넌트	실행 환경	사용 기술

Runtime Agent	AWS Lambda	Python, boto3
Regulation Agent	AWS Lambda	Python, RAG
Finance Agent	AWS Lambda	AWS Pricing API
Event Routing	EventBridge	Rule-based trigger

표 1. 설계 컴포넌트와 실행 환경 매핑

표 1에서 보듯이, 각 에이전트는 서버리스 환경에서 독립적인 실행 단위로 배치 가능하며, 이벤트 라우팅은 EventBridge를 통해 관리된다. 이러한 구성은 추가적인 인프라 관리 부담 없이 기존 클라우드 서비스 위에서 제안한 다중 에이전트 구조를 구현할 수 있음을 보여준다.

5. 구현 및 시나리오 기반 검증

본 장에서는 2.2절에서 논의한 정적 플레이북 기반 SOAR의 한계, 특히 운영 맥락 변화에 따른 과잉 대응 문제를 실제 클라우드 운영 시나리오에서 어떻게 완화하는지를 중심으로, 제안 워크플로우의 동작을 시나리오 기반으로 검증한다. 이를 위해 IAM Access Key 유출 대응과 AMI 외부 공개·공유 시도 탐지라는 두 가지 대표적 상황을 선정하였다.

본 연구의 목적은 모델 정확도 경쟁이 아니라, 신뢰 가능한 의사결정·안전한 실행·감사 가능성을 갖춘 워크플로우 설계의 타당성을 보이는 데 있다.

5.1 시나리오 1: IAM Access Key 유출 탐지 및 대응

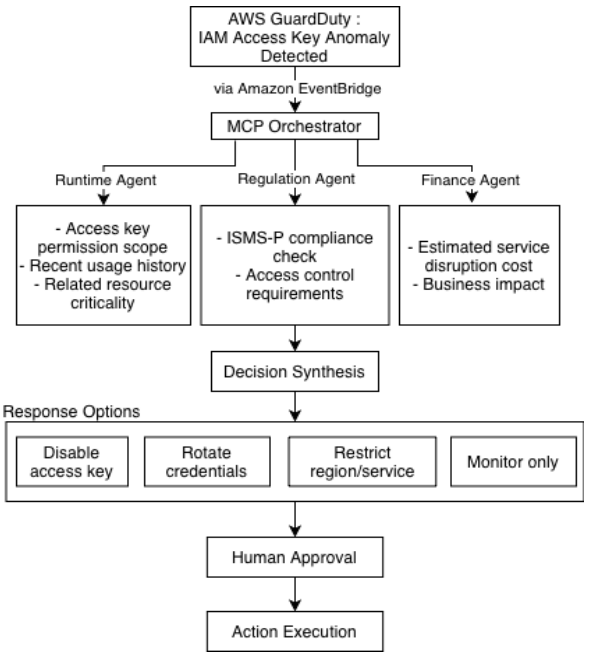


그림 4. IAM Access Key 유출 탐지 시나리오에 대한 의사결정 워크플로우

1) 위협 탐지: 외부 IP에서 비정상적인 API 호출이 발생하거나 권한 밖의 자원에 접근을 시도할 경우, GuardDuty는"UnauthorizedAccess:IAMUser/AnomalousBehavior" 유형의 위협 이벤트를 탐지한다.

2) 이벤트 라우팅: Amazon EventBridge는 탐지된 위협 이벤트를 수집하여 사전에 정의된 규칙에 따라 이를 MCP Orchestrator로 전달한다 [8].

3) 컨텍스트 분석 및 의사결정: Runtime Agent는 유출된 키의 권한 범위, 최근 사용 이력, 연관 리소스의 중요도를 분석한다. Regulation Agent는 ISMS-P 등 보안 규제 준수 여부를 확인하고, Finance Agent는 해당 키와 연결된 서비스 중단 시 발생할 비즈니스 손실 비용을 산출한다.

4) 대응 옵션 생성 및 실행: 분석 결과를 바탕으로 a) 액세스 키 즉시 비활성화, b) 자격 증명 회전(Rotation), c) 특정 리전 및 서비스로 권한 일시 제한, d) 단순 모니터링 강화 등 단계별 대응 옵션을 생성한다. 관리자가 최종 옵션을 승인하면 실행 모듈이 조치를 수행하고 결과를 기록한다.

이 시나리오는 단순한 차단을 넘어 "업무 연속성과 보안 강도" 사이의 균형을 맞춘 의사결정을 지원하며, 특히 유출된 키가 개발 환경인지 운영 환경인지에 따른 차등적 대응이 가능하다는 장점이 있다.

5.2 시나리오 2: AMI 외부 공개/공유 시도 탐지(ModifyImageAttribute)

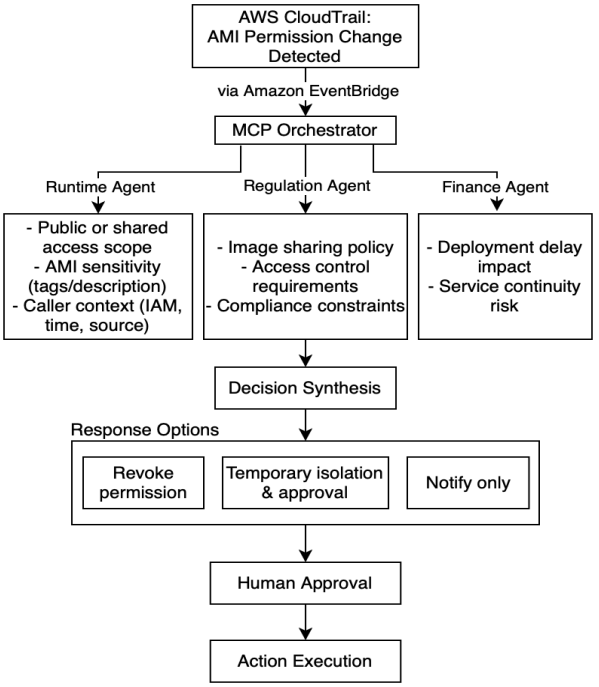


그림 5. AMI 권한 변경 이벤트에 대한 시나리오 기반 의사결정 워크플로우

AMI 권한 변경은 민감 설정 및 인증정보가 포함된 이미지가 외부로 확산될 수 있는 대표적 리스크이다. CloudTrail은 ModifyImageAttribute 호출을 기록하며, EventBridge는 해당 이벤트 패턴(launchPermission 변경 등)을 탐지해

Orchestrator로 전달한다 [6,7,8].

Runtime Agent는 1) 전체 공개(all) 여부, 2) 특정 계정 공유 대상(Account ID), 3) AMI 태그/설명 기반 민감도(예: prod, confidential), 4) 호출 주체(IAM principal) 및 실행 시간/출처 IP 등 맥락을 종합해 위험도를 산정하고 대응 옵션을 생성한다. 대응은 a) 즉시 권한 회수(공개 취소/공유 제거), b) 임시 격리 후 승인 요청, c) 알림만 등으로 분기되며, Runtime Agent는 승인된 작업만 수행한다. 이로써 단순 탐지에 그치지 않고 '업무 연속성 vs 유출 위험'의 균형을 구조화된 의사결정으로 지원한다.

5.3 성능 평가 및 비교 실험

본 절에서는 제안하는 MCP 기반 멀티 에이전트 아키텍처의 효용성을 검증하기 위해 기존 방식들과의 비교 실험을 수행한다. 본 연구의 평가 범위는 실제 클라우드 자원 제어(boto3 기반 실행) 이전 단계인 런타임 의사결정 및 대응 계획 생성 단계로 한정하였다. 이는 실행 성공률이라는 인프라 종속적 지표 대신, 시스템의 지능적 판단 품질과 출력 안정성을 정량적으로 평가하기 위함이다.

5.3.1 실험 설계 및 환경

실험은 12개의 대표적인 GuardDuty 위협 시나리오를 대상으로 수행되었다. 각 시나리오는 전문가에 의해 정답 라벨(Ground Truth)이 부여되었으며, 여기에는 위험도(Severity), 권장 대응 유형(Action Type), 금지된 대응 행위(Forbidden Actions)가 포함된다. 시스템의 출력은 "Action Plan JSON" 표준 스키마를 따르도록 강제하여 품질 평가의 객관성을 확보하였다.

비교 대상은 다음과 같이 세 그룹으로 정의한다.
1) R0 (Rule-based): 이벤트 필드에 따른 사전 정의된 규칙 세트를 사용하는 전통적인 자동화 방식이다.
2) L1 (Single-LLM): 단일 LLM(Gemini 1.5 Pro) 호출을 통해 분석과 계획 생성을 한 번에 수행하는 방식이다.
3) M2 (Multi-Agent): 본 논문에서 제안하는 오케스트레이터와 전문 에이전트(Runtime, Regulation, Finance) 간의 협업 방식이다.

5.3.2 평가지표

실험의 평가는 성능, 품질, 안정성의 세 가지 관점에서 이루어진다.
1) 런타임 성능 (Performance)
- E2E Latency: 입력 이벤트 수신부터 최종 JSON 출력까지 소요되는 전체 시간이다.
- MTTR : 각 시나리오의 평균 응답 시간
2) 의사결정 품질 (Quality)
- Severity Accuracy: 판정된 위협 수준이 전문가 라벨과 일치하는 비율이다.
- Action Type Accuracy: 제안된 대응 유형이 허용 범위 내에 존재하는 비율이다.
정확도는 전체 시나리오 수 N에 대한 전문가 라벨 y_i와 시스템 출력 $\hat{y}_i$의 일치 여부를 기준으로 계산한다.
- Over/Under-response Rate: 실제 위협보다 과하거나 부족하게 대응한 비율을 측정하여 운영 리스크를 평가한다.
3) 출력 안정성 (Robustness)
- JSON Valid Rate: 출력값이 유효한 JSON 형식으로 파싱되는지 확인한다.
- Schema Compliance: "action_plan" 내 필수 필드(target, method 등)의 누락 여부를 검증한다.

5.3.3 실험 결과 및 분석

실험 결과, 제안 시스템인 M2는 런타임 성능과 의사결정 품질 사이에서 최적의 균형을 보여주었다(표 2 참조).

지표	R0	L1	M2
평균 응답 시간(sec)	0.45	4.12	8.65
위험도 판단 정확도(%)	66.7	83.3	97.2
조치 유형 정확(%)	58.3	75.0	94.4
과잉 대응률 (%)	25.0	16.7	2.8
JSON 규격 준수율 (%)	100.0	91.7	100.0

표 2. 실험 그룹별 정량적 성능 비교 결과(R0: Rule-based, L1: Single-LLM, M2: Multi-Agent)

표 2의 데이터를 분석한 결과, 제안하는 멀티 에이전트 구조(M2)는 규칙 기반(R0) 방식보다 정확도 측면에서 약 30% 이상의 월등한 성능 향상을 보였다. 특히 단일 LLM(L1) 방식에서 빈번히 발생하던 과잉 대응 문제를 2.8%까지 낮춘 것은, 규제(Regulation) 및 비용(FinOps) 에이전트가 교차 검증을 수행함으로써 불필요한 서비스 중단 리스크를 억제했기 때문으로 분석된다. 응답 시간은 다소 증가하였으나, 수 분 이상 소요되는 관리자의 수동 대응 시간을 고려할 때 실무 적용이 충분히 가능한 수준이다.

5.4 운영 품질 관점의 효과(정성/구조적 평가)

제안 워크플로우는 1) 불필요한 대응 감소(환경/자산 맥락 반영), 2) 판단 일관성 향상(정책 질의 기반), 3) 의사결정 투명성(근거·정책 참조), 4) 감사 추적성(오케스트레이션 로그 + CloudTrail을 제공한다 [6]. 이는 '자동화가 있는데도 신뢰하지 못하는' 문제를 설계 차원에서 완화한다.

6. 결론 및 향후 과제

본 논문은 클라우드 보안 이벤트 대응을 단순 탐지 중심이 아니라 운영 의사결정의 신뢰성 문제로 재정의하고, 보안·규제·비용 요구사항을 통합적으로 고려하는 DevSecOps 대응 워크플로우를 설계하였다. MCP Orchestrator 중심의 통신 구조와 역할 기반 에이전트 경계(Agent Boundary), 판단과 실행의 분리를 통해 자동화의 안전성·설명 가능성·감사 가능성을 확보하였다(그림 1 참조). 특히 비교 실험을 통해 제안 시스템이 기존 규칙 기반 방식 대비 약 30% 높은 의사결정 정확도를 보임을 입증하였다.

향후 과제는 다음과 같다. 1) Regulation 에이전트의 규제 문서 구조화 및 보안 이벤트–조항 매핑 로직의 정량적 평가, 2) 복합 대응 시나리오를 위한 롤백 및 다단계 플레이북의 안전한 실행(Transaction/Saga 패턴) 지원, 3) 실제 운영 데이터 기반의 장기간 실험을 통한 오탐 및 과잉 대응 비용에 대한 정밀 평가, 4) 사람과 자동화의 협업을 위한 사용자 인터페이스 및 승인 정책의 고도화이다.

참고문헌

[1] C. Islam, M. A. Babar, and S. Nepal, “A Multi-Vocal Review of Security Orchestration,” ACM Computing Surveys, 2019. https://dl.acm.org/doi/10.1145/3305268
[2] National Institute of Standards and Technology, “Framework for Improving Critical Infrastructure Cybersecurity,” NIST Cybersecurity Framework, 2018. https://www.nist.gov/cyberframework
[3] Q. Wu et al.,“AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation,” arXiv preprint arXiv:2308.08155, 2023. https://arxiv.org/abs/2308.08155
[4] S. Hong et al. “MetaGPT: Meta Programming for Multi-Agent Collaborative Framework,” arXiv preprint arXiv:2308.00352, 2023. https://arxiv.org/abs/2308.00352
[5] Anthropic, “Model Context Protocol (MCP) Documentation,” Technical Report, 2024. https://modelcontextprotocol.io
[6] Amazon Web Services, “AWS CloudTrail User Guide,” Amazon Web Services Documentation, 2024. https://docs.aws.amazon.com/cloudtrail/
[7] Amazon Web Services, “Amazon GuardDuty User Guide,” Amazon Web Services Documentation, 2024. https://docs.aws.amazon.com/guardduty/
[8] Amazon Web Services, “Amazon EventBridge User Guide,” Amazon Web Services Documentation, 2024. https://docs.aws.amazon.com/eventbridge/

AutoFiC: 취약점 탐지부터 PR 생성까지 자동화된 보안 패치 파이프라인

장인영 [O1], 오정민 [2], 김민채 [3], 김은솔 [4]
덕성여자대학교 [1], 가천대학교 [2], 국민대학교 [3], 명지대학교 [4]
726iy@duksung.ac.kr, ojmes5790@gmail.com, brianna0324@kookmin.ac.kr, kesol1530@gmail.com

AutoFiC: Automated Security Patch Pipeline from Vulnerability Detection to Pull Request Generation

Inyeong Jang[O1], JeongMin Oh[2], Minchae Kim[3], Eunsol Kim[4]
[1]Duksung Women's University, [2]Gachon University, [3]Kookmin University,
[4]Myongji University

요 약

정적 분석 도구(SAST)는 보안 취약점 탐지에 널리 활용되고 있으나, 탐지 이후의 패치 생성 및 적용 과정은 여전히 개발자의 수작업에 크게 의존하고 있다. 본 연구는 이러한 문제를 해결하기 위해 취약점 탐지부터 패치 생성 및 적용, Pull Request 생성에 이르는 전 과정을 자동화한 End-to-End 보안 패치 파이프라인 AutoFiC을 제안한다. AutoFiC은 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 결합하여, AST 기반의 복잡한 구문 분석을 직접 활용하지 않고도 취약점의 위치와 유형 정보를 LLM에 효과적으로 전달한다. 이를 통해 수정 범위를 취약 구간으로 명확히 한정함으로써, 패치 부작용을 최소화하는 경량 자동 패치 구조를 구현하였다. 91개의 실제 GitHub 오픈소스 저장소를 대상으로 한 실험 결과, AutoFiC은 94.5%의 파이프라인 성공률과 90.0%의 취약점 해결률을 기록하였다.

1. 서 론

최근 소프트웨어 시스템의 규모와 복잡도가 증가함에 따라, 소스 코드 수준에서의 보안 취약점 관리가 점점 더 중요한 과제로 부각되고 있다. 이러한 취약점을 사전에 식별하기 위해 정적 분석 도구(SAST)가 다양한 개발 환경에서 활용되고 있으나, 탐지 이후의 수정 과정은 여전히 개발자의 수작업에 크게 의존하고 있다. 이로 인해 취약점 탐지와 실제 수정 사이에는 상당한 시간적·운영적 간극이 존재하며, 이를 완화하기 위한 자동화된 보안 패치 기법에 대한 요구가 지속적으로 증가하고 있다[1].

이러한 배경에서 자동 프로그램 수리(Automated Program Repair, APR) 및 대규모 언어 모델(LLM)을 활용한 자동 패치 접근이 제안되었다. 기존 연구들은 LLM을 활용하여 취약 코드를 자동으로 수정할 수 있는 가능성을 보여주었으나, 수정 범위가 필요 이상으로 확대되거나 코드의 구조적 맥락을 충분히 반영하지 못하는 한계 또한 함께 논의되어 왔다. 이를 보완하기 위한 다양한 접근이 제안되었음에도 불구하고, 실제 개발 환경에서 활용 가능한 경량 자동 패치 파이프라인에 대해서는 여전히 명확한 설계가 정립되지 않은 실정이다[2].

본 연구는 이러한 공백을 해소하기 위해, SAST 결과로부터 생성한 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 결합한 LLM 기반 자동 패치 파이프라인을 제안한다. 제안 기법은 AST 기반 분석을 직접 활용하지 않으면서도 취약점의 위치와 유형을 구조적으로 전달함으로써 LLM의 이해도를 향상시키고, 수정 범위를 취약 구간 중심으로 제한하여 패치 부작용을 최소화한다. 또한 취약점 탐지부터 패치 생성 및 적용, Pull Request 생성까지 이어지는 End-to-End 자동화를 구현함으로써, 실제 개발 워크플로우에 적용 가능한 경량 자동 보안 패치 파이프라인의 설계 방향을 제시한다.

2. 관련 연구

자동 프로그램 수리(Automated Program Repair, APR) 연구는 규칙 기반 패치 생성 기법을 출발점으로 하여, 검색 기반 및 학습 기반 접근법으로 점차 확장되어 왔다. 기존 연구들은 코드 결함에 대해 자동으로 수정 코드를 생성할 수 있음을 실험적으로 입증하며, 다양한 결함 유형에 대한 적용 가능성을 제시하였다[3]. 그러나 대다수 연구는 제한된 코드 범위나 단일 결함을 대상으로 수행되었으며, 실험 환경과 평가 지표가 연구 목적에 따라 상이하게 설정되어 실제 소프트웨어 개발 환경으로의 일반화에는 한계가 존재한다.

최근에는 이러한 한계를 극복하기 위해 대규모 언어 모델(LLM)을 APR에 적용하려는 시도가 활발히

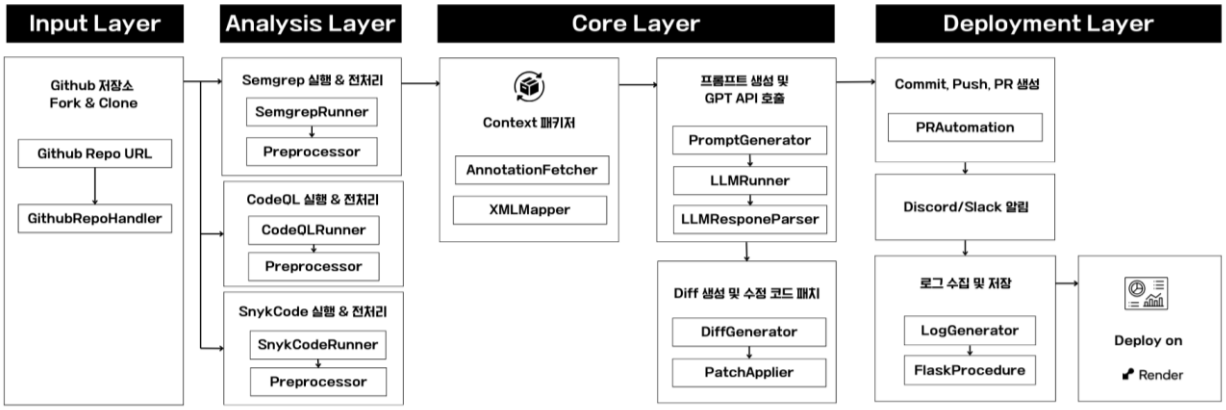


그림 1. 시스템 아키텍처

이루어지고 있다. LLM 기반 접근법은 코드와 자연어의 문맥을 동시에 학습한 모델을 활용함으로써, 기존 규칙 기반 기법 대비 보다 유연하고 다양한 수정 코드를 생성할 수 있음을 보였다[4]. 이를 통해 LLM이 자동 패치 생성에 활용될 수 있음이 실증적으로 확인되었으나, 프롬프트에 제공되는 코드의 단위, 수정 범위의 설정 방식, 그리고 생성된 패치에 대한 검증 전략은 연구마다 상이하게 설계되어 표준화된 방법론이 부재한 실정이다. 이러한 설계 차이는 패치의 안정성과 재현성 측면에서 추가적인 고려가 필요함을 시사한다.

한편, 자동 패치 생성의 전제 조건인 결함 식별 단계에서는 정적 분석 도구(SAST)가 취약점 위치와 유형을 제공하는 핵심적인 정보원으로 활용되어 왔다. 정적 분석 결과를 기반으로 경고를 분류하거나 우선순위를 결정하고, 수정 후보 위치를 식별하기 위한 다양한 기법이 제안되었으며[5], 일부 APR 연구에서는 이러한 분석 결과를 패치 생성 단계의 입력으로 직접 활용하기도 한다. 그러나 정적 분석 도구는 대량의 경고를 생성하는 특성으로 인해, 실제 수정으로 이어질 수 있는 정보를 효과적으로 선별·활용하는 데 한계를 보인다.

자동 프로그램 수리(APR) 전반을 대상으로 한 체계적 문헌 분석 연구에 따르면, 패치 생성 기법 자체에 대한 연구는 지속적으로 발전해온 반면, 생성된 패치를 실제 코드베이스에 적용하고 개발 워크플로우와 연계하는 과정은 연구마다 서로 다른 가정과 설계를 채택하고 있다[6]. 즉, 결함 탐지, 패치 생성 및 적용, 개발 프로세스 연계를 하나의 일관된 흐름으로 통합하려는 접근은 아직 정형화되지 않았다.

종합하면, 기존 연구들은 자동 패치 생성 알고리즘의 고도화, LLM의 적용 가능성, 정적 분석 결과의 활용 등 개별 요소 기술을 중심으로 발전해왔다. 그러나 취약점 탐지부터 수정 적용, 검증, 그리고 개발 워크플로우 연계까지를 포괄하는 체계적인 자동화 구조에 대해서는 연구마다 상이한 설계와 가정을 채택하고 있다. 본 연구는 이러한 선행 연구들을 바탕으로, 자동 보안 패치 파이프라인 설계에 대한 하나의 통합적 관점을 제시한다.

3. 제안 기법

3.1 시스템 아키텍처

그림 1은 제안하는 AutoFiC 시스템의 전체 아키텍처를 나타낸다. AutoFiC은 GitHub 저장소 Fork 및 Clone, 다중 SAST 기반 정적 분석, LLM 프롬프트 구성 및 패치 코드 생성, Pull Request 자동화, 대시보드 시각화로 이어지는 계층형 구조를 따른다. 전체 시스템은 Python 기반 CLI(Command Line Interface) 도구로 구현되어, 로컬 환경에서 일관되게 활용할 수 있다.

입력 계층(Input Layer)에서는 사용자로부터 분석 대상 GitHub 저장소의 URL과 SAST 도구 및 LLM 모델에 대한 설정을 CLI 인자(Argument)로 전달받는다. 시스템은 GitHub API를 활용하여 대상 저장소를 Fork하고 로컬 환경으로 Clone하여 분석을 위한 실행 환경을 구성한다.

분석 계층(Analysis Layer)에서는 Semgrep, CodeQL, SnykCode와 같은 정적 분석 도구 중 하나를 선택적으로 실행하여 소스 코드 내 취약점을 탐지한다. 각 도구의 분석 결과는 전처리 모듈을 거쳐 공통 스키마인 BaseSnippet 형태로 정규화된다. BaseSnippet은 파일 경로, 라인 범위, CWE 정보 등을 포함하며, 도구별 출력 형식의 차이를 추상화하여 후속 단계에 일관된 입력 데이터를 제공한다.

핵심 처리 계층(Core Layer)에서는 정규화된 스니펫을 기반으로 구조화된 XML 컨텍스트와 코드 주석(Annotation)을 생성한다. 생성된 컨텍스트는 프롬프트 엔지니어링 모듈로 전달되어 최적화된 프롬프트를 구성하며, 이를 LLM에 입력하여 취약점

패치 코드를 생성한다. LLM의 응답은 파싱 과정을 거쳐 Diff 형식으로 변환된 후, 자동 패치 모듈을 통해 코드베이스에 반영된다. 패치 적용 실패 시 재시도(Retry) 및 Fallback 메커니즘이 동작하여 전체 파이프라인의 가용성과 연속성을 보장한다.

배포 계층(Deployment Layer)에서는 패치가 적용된 Branch에 대해 Commit, 원격 저장소 Push, Pull Request 생성을 포함한 일련의 과정을 자동화한다. 최종 결과는 Flask 기반의 웹 대시보드를 통해 제공되며, Slack 및 Discord 연동을 통해 실시간 알림 서비스를 제공한다.

3.2 전체 워크플로우

AutoFiC은 정적 분석 도구(SAST)와 대규모 언어 모델(LLM)을 결합하여, 취약점 탐지부터 패치 생성 및 적용, Pull Request 생성에 이르는 전 과정을 자동화한 End-to-End 파이프라인을 제공한다. 전체 워크플로우는 (1) 저장소 초기화 및 환경 구성, (2) SAST 기반 취약점 탐지 및 정규화, (3) XML 및 Annotation 기반 컨텍스트 구성, (4) LLM 기반 패치 코드 생성, (5) Diff 생성 및 패치 적용, (6) Pull Request 생성 및 CI 연계의 6단계로 구성된다.

3.2.1 GitHub 저장소 Fork 및 Clone

사용자가 분석 대상 GitHub 저장소의 URL을 입력하면, AutoFiC은 GitHub API를 통해 해당 저장소를 Fork한 후 로컬 환경으로 Clone하여 분석 환경을 구축한다. 이 과정은 Branch 분기 및 Commit 이력 보존을 포함하는 표준 Git 워크플로우를 준수하도록 설계되었다. 이를 통해 원본 저장소의 무결성을 해치지 않으면서, 실제 개발 환경과 동일한 조건에서 패치를 검증할 수 있는 격리된 환경을 제공한다.

3.2.2 SAST 기반 취약점 분석

분석 단계에서는 Semgrep, CodeQL, SnykCode와 같은 SAST 도구 중 하나를 선택적으로 실행하여 취약점을 탐지한다. 각 도구는 상이한 탐지 규칙과 출력 형식을 가지므로, 분석 결과를 공통 스키마인 BaseSnippet 형태로 정규화한다. BaseSnippet은 파일 경로, 라인 범위, CWE 정보를 포함하며, 도구별 출력 형식의 차이를 추상화하여 후속 단계에서 일관된 입력 데이터를 보장한다.

3.2.3 컨텍스트 기반 프롬프트 구성 전략

SAST 결과만으로는 LLM이 코드 내부의 제어 흐름, 데이터 흐름, 취약 구간의 경계 등을 온전히 파악하기 어렵다는 한계가 있다. AutoFiC은 이를 보완하기 위해, 취약 코드 주변 컨텍스트를 구조화된 메타데이터, 위치 정보, 최소 단위 코드 스니펫으로 재구성한 뒤 프롬프트에 반영하는 전략을 사용한다. 이때 컨텍스트는 파일 경로와 라인 범위, CWE 유형, 경고 메시지, 관련 함수/블록 코드와 같은 정보로 구성되며, LLM이 어디를, 왜, 어떻게 수정해야 하는지를 명시적으로 이해하도록 돕는다.

컨텍스트 구성 단계에서는 SAST가 보고한 취약점 스니펫을 공통 스키마(BaseSnippet)로 정규화하고, 이를 기반으로 요약 메타데이터와 코드 조각을 생성한다. 메타데이터는 CUSTOM_CONTEXT.xml과 같은 구조화된 형식에 저장되며, 프롬프트에서는 대상 취약점과 직접 관련된 조각만 발췌하여 사용함으로써 입력 크기를 관리한다. 코드 측면에서는 취약 줄만 제공하는 대신, 동일 함수나 인접 블록을 함께 포함해 제어 흐름이 끊어지지 않도록 스니펫을 구성함으로써, LLM이 수정 시 주변 문맥을 고려할 수 있도록 한다. 주석 기반 마커(Annotation)는 이러한 컨텍스트를 보조하는 수단으로 활용되며, 프롬프트 설계의 핵심은 필요한 정보는 충분히 제공하되, 수정 범위를 안정적으로 제어하는 것에 맞추어져 있다.

3.2.4 LLM 기반 패치 생성

구축된 XML 컨텍스트와 주석이 포함된 코드 스니펫은 프롬프트 엔지니어링을 거쳐 LLM에 입력된다. 프롬프트는 취약점에 대한 상세 설명과 수정 제약 조건을 포함하도록 설계되었으며, LLM은 이를 바탕으로 수정된 코드를 생성한다. 생성된 결과물은 파싱 과정을 통해 자동 적용이 가능한 Unified Diff 형식으로 변환된다.

3.2.5 Diff 생성 및 패치 적용

LLM이 생성한 수정 코드와 원본 소스 코드 간의 차이는 Diff 파일로 생성되며, Git의 Patch 시스템을 통해 코드베이스에 반영된다. 이때 패치 적용 실패 등 예외 상황 발생 시, 사전에 정의된 Fallback 절차를 수행하여 파이프라인의 중단을 방지한다. 이러한 예외 처리 메커니즘은 자동화된 패치 프로세스의 안전성과 연속성을 보장한다.

3.2.6 Pull Request 생성 및 CI 연계

패치가 적용된 Branch는 자동으로 원격 저장소에 Push 되며, 이를 기반으로 Pull Request가 생성된다. AutoFiC은 Pull Request 생성 및 상태 변경 이벤트를 감지하는 GitHub Actions 워크플로우를 자동 생성하며, Discord 또는 Slack과 연동하여 실시간 알림을 제공한다. 또한, CI 환경에 필요한 민감 정보(Webhook URL 등)는 공개키 기반으로 암호화하여 GitHub Secrets에 안전하게 등록된다. 최종적으로 생성된 Pull Request에는 분석 결과 요약과 CI 상태 모니터링 정보가 포함되어 검토자의 효율적인 의사결정을 지원한다.

3.3 프롬프트 엔지니어링

AutoFiC의 프롬프트 엔지니어링은 취약 코드만을 LLM에 그대로 전달하는 방식에 그치지 않고, SAST 결과로부터 추출한 구조화 메타데이터(XML), 코드 스니펫, Annotation 기반 위치 힌트를 단계적으로 결합하여 LLM이 취약점의 의미적·구조적 맥락을 명확히 이해하도록 설계되었다. 모델은 취약점 설명과 정확한 위치 정보, 관련 코드 컨텍스트를 함께 입력 받아 수정 대상을 명확히 식별하고, 취약 구간 중심의 최소 수정 패치를 생성하도록 유도된다.

3.3.1 XML 기반 구조화 컨텍스트

AutoFiC은 SAST 도구별 분석 결과를 정규화한 BaseSnippet을 기반으로, 입력 이전 단계에서 CUSTOM_CONTEXT.xml 형태의 XML 기반 구조화 컨텍스트를 생성한다. 각 취약점 항목에는 파일 경로 및 라인 범위, 분석 도구 및 규칙 식별 정보, CWE ID, 원본 메시지와 요약 설명, 심각도 및 카테고리 정보가 포함된다.

```
<CUSTOM_CONTEXT version="1.1">
  <VULNERABILITY id="scripts/main.py:411-411">
    <FILE path="scripts/main.py"/>
    <RANGE start="411" end="411"/>
    <SEVERITY overall="ERROR" bit="ERROR"/>
    <MESSAGE>
      <ITEM>Unsafe subprocess.run() with shell=True</ITEM>
    </MESSAGE>
    <SNIPPET>
      subprocess.run(explorer_command, shell=True)
    </SNIPPET>
    <BIT>
      <TRIGGER>Command injection risk</TRIGGER>
      <STEPS>
        <STEP>Review line 411 in scripts/main.py</STEP>
      </STEPS>
    </BIT>
    <CLASSES>
      <CLASS>Command Injection</CLASS>
    </CLASSES>
    <WEAKNESSES>
      <CWE id="CWE-78"/>
    </WEAKNESSES>
  </VULNERABILITY>
</CUSTOM_CONTEXT>
```

그림 2. CUSTOM_CONTEXT.xml 구조 예시

프롬프트 구성 시 전체 XML을 그대로 LLM에 주입하지 않고, 대상 취약점과 직접 관련된 항목만 선별하여 Markdown 형식의 STUCTURED CONTEXT 블록으로 변환한다. 이를 통해 입력 토큰을 경량화하면서도, 단순 문자열 설명보다 풍부한 반정형 맥락 정보를 LLM에 제공할 수 있다.

3.3.2 Annotation 기반 위치 강조 기법

구조화 메타데이터가 취약점의 의미적 정보를 제공한다면, Annotation 기반 위치 강조는 코드 내에서 수정 범위를 명확히 한정하는 역할을 한다. AutoFiC은 취약점이 보고된 코드 스니펫과 해당 함수 또는 블록 전체를 프롬프트에 포함하고, 필요 시 취약 줄 또는 범위를 명시적 Annotation 마커(예: @BUG_HERE, @BUG_HERE_START, @BUG_HERE_END)로 표시한다. 단일 라인 취약점은 해당 줄 수준에서 Annotation을 적용하고, 범위 기반 취약점은 시작·종료 라인 기준으로 표현함으로써, 모델이 전체 파일을 과도하게 수정하는 것을 방지하고 취약 구간 중심의 패치를 생성하도록 유도한다. 패치 생성 이후에는 후처리 단계에서 Annotation 마커를 제거하여, 결과 코드의 가독성과 품질에 영향을 최소화한다.

3.3.3 프롬프트 설계 원칙

AutoFiC의 프롬프트는 다음 원칙에 따라 구성된다.

- 역할 및 제약 명시: 시스템 메시지에 보안 패치 어시스턴트 역할을 부여하고, 기존 기능 보존 및 신규 취약점 미도입과 같은 제약을 명시한다.
- 컨텍스트 최소화: 취약점과 직접 관련된 함수 또는 블록만 포함하되, 제어 흐름이 단절되지 않도록 스니펫 경계를 설정한다. XML 요약 블록과 코드 블록은 명확히 구분하여 제시한다.
- 수정 범위 제한: Annotaion을 통해 표시된 취약 범위만 수정하도록 요구하고, 나머지 영역은 최소 변경을 유지하도록 명시한다.
- 형식화된 출력 요구: 결과 출력 형식을 고정하여, 수정된 전체 함수 또는 변경 블록 전체를 포함하도록 요구함으로써 파싱 및 diff 기반 자동 패치 적용의 안정성을 확보한다.

```
The following is a Python source file that contains security vulnerabilities.
  ...
Detected vulnerabilities:
  ...
## STRUCTURED CONTEXT (Team-Atlanta Approach)
The following CUSTOM_CONTEXT.xml provides structured vulnerability information including:
- BIT (Bug Information Template) with TRIGGER, STEPS, REPRODUCTION
```

- Detailed CWE classifications and severity levels
- Environmental context and mitigation strategies ```xml… ```

Use the BIT information above to understand:
1. TRIGGER: What conditions activate this vulnerability
2. STEPS: How to locate and review the vulnerable code
3. REPRODUCTION: How to verify the issue

4. BIT_SEVERITY: The criticality level of this vulnerability

Please strictly follow the guidelines below when modifying the code:
- Modify **only the vulnerable parts** of the file with **minimal changes**.
- Preserve the **original line numbers, indentation, and code formatting** exactly.
- **Do not modify any part of the file that is unrelated to the vulnerabilities.**
- Output the **entire file**, not just the changed lines.
- This code will be used for diff-based automatic patching, so structural changes may cause the patch to fail.

Output format example:
1. Vulnerability Description: ...
2. Potential Risk: ...
3. Recommended Fix: ...
4. Final Modified Code:
5. Additional Notes: (optional)

그림 3. AutoFiC의 LLM 프롬프트 템플릿 예시

이와 같은 컨텍스트 설계는 LLM 기반 자동 패치에서 중요한 정확한 위치 지정과 구조적 힌트 제공을 동시에 달성하도록 하며, 4장에서 정량 실험을 통해 그 효과를 평가한다.

3.4 자동 패치 적용 및 Pull Request 자동화

AutoFiC은 LLM이 생성한 패치를 코드베이스에 안전하게 반영하고, Pull Request 생성과 CI 연계를 통해 이를 실제 개발 워크플로우에 통합하는 자동화 기능을 제공한다. 본 절에서는 LLM 응답 파싱, Diff 생성 및 패치 적용, Pull Request 생성 및 CI 연계의 세 단계로 구성된 자동화 절차를 설명한다.

3.4.1 LLM 응답 파싱

생성된 응답에서 전용 파서(Parser)를 통해 수정 코드 블록을 추출한다. 이때 응답이 지정된 형식을 위반하거나 구문 오류를 포함하는 경우, 해당 케이스를 패치 불가로 분류하거나 재생성 로직으로 전환한다. 또한 API 호출 중 발생하는 토큰 한도 초과 등의 외부 예외를 별도로 처리하여 시스템 안전성을 유지한다.

3.4.2 Diff 생성 및 패치 적용

파싱된 수정 코드는 원본 코드와의 비교를 통해 Unified Diff 형식으로 변환되며, 이후 Git Patch 메커니즘을 통해 자동 적용된다. AutoFiC은 자동 패치 과정에서 발생할 수 있는 충돌이나 문맥 불일치 문제에 대응하기 위해 이중 적용 전략을 사용한다.

우선 표준 Diff 기반 패치 적용을 1차적으로 시도하며, 이 과정이 실패할 경우 전체 파일 단위로 수정된 코드를 덮어쓴 뒤 다시 Diff를 생성하는 Fallback 절차를 수행한다. 이를 통해 Diff 적용 실패로 인한 파이프라인 중단을 방지하고 End-to-End 자동화 흐름을 지속적으로 유지하기 위한 안전장치를 마련한다.

실험 과정에서, LLM이 생성한 수정 내용이 논리적으로는 타당함에도 불구하고, 미세한 공백 차이, 주석 위치 변경, 출력 포맷 차이 등으로 인해 Diff 적용이 실패하는 사례가 다수 관찰되었다. 이러한 경우 Fallback 절차를 통해 패치 적용을 재시도함으로써, 자동화 파이프라인이 중단되지 않고 Pull Request 생성 단계까지 도달하는 비율이 증가하는 경향을 확인하였다. 이러한 결과는 Fallback 절차가 AutoFiC 파이프라인의 안정성과 연속성을 보장하는 핵심 구성 요소로 기능함을 시사한다.

3.4.3 Pull Request 생성 및 CI 연계

패치 적용이 완료된 Branch는 원격 저장소로 Push되며, 이를 기반으로 상세 정보를 포함한 Pull Request가 자동 생성된다. 시스템은 GitHub Actions 워크플로우를 자동 구성하여 Pull Request 이벤트를 감지하고, CI 테스트 결과를 모니터링한다. 모든 결과는 암호화된 채널을 통해 Slack 또는 Discord로 실시간 전달되며, 이를 통해 개발자는 보안 패치를 신속하게 검토하고 병합할 수 있도록 지원한다.

4. 실험 결과

4.1 실험 설계

본 연구에서 제안하는 AutoFiC 파이프라인의 실효성을 검증하기 위해, 실제 오픈소스 환경을 대상으로 파이프라인 동작 성공률과 취약점 패치 성능을 평가하였다. 또한 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법이 패치 성능에 미치는 효과를 분석하기 위해, 해당 기법 적용 전후의 결과를 비교 분석하였다.

4.1.1 데이터셋 및 실험 환경

실험 데이터셋은 GitHub에 공개된 Python 기반 저장소 중, 현실적인 개발 프로젝트 규모를 반영하기 위해 Star 수가 30 이상 100 이하인 중소 규모 프로젝트를 대상으로 구성하였다. GitHub API를 통해

수집된 저장소 중, Semgrep 정적 분석 결과 최소 1개 이상의 취약점이 탐지된 91개 저장소를 최종 실험 대상으로 선정하였다.

Star 수가 높은 대규모 프로젝트는 복잡한 CI 설정, 방대한 의존성, 장시간 테스트 실행 등을 포함하는 경우가 많아, 자동 패치 파이프라인의 실행 시간, 환경 재현성, 비교 가능성을 저하시킬 수 있다. 본 연구는 개별 프로젝트의 CI 환경 최적화나 복잡한 빌드/테스트 설정을 해결하는 것을 목표로 하지 않으며, 중소 규모 프로젝트를 대상으로 End-to-End 자동화 파이프라인의 일반적인 실효성과 안정성을 검증하는 데 초점을 두었다. 이에 따라 이러한 대규모 프로젝트는 실험 범위에서 제외하였다

실험은 Python 3.10 환경에서 수행되었으며, 패치 생성을 위한 LLM 모델로는 GPT-4o를 사용하였다. 취약점 탐지 및 해결 여부 판단의 일관성을 확보하기 위해 Semgrep을 기준 정적 분석 도구로 사용하였다. 패치 적용 전후에 동일한 규칙과 조건으로 Semgrep을 재실행하여 탐지 결과의 변화를 기반으로 취약점 해결 여부를 판정하였다.

4.1.2 평가 지표

평가 지표는 다음과 같이 설정하였다.

1. **파이프라인 성공률 (Pipeline Success Rate):** 전체 저장소 중 Fork부터 Pull Request 생성까지의 전 과정이 중단 없이 수행된 비율.
2. **취약점 해결률 (Vulnerability Fix Rate):** 파이프라인이 성공한 저장소를 대상으로, 패치 적용 후 Semgrep을 동일한 규칙과 조건으로 재실행하였을 때 기존에 탐지된 취약점이 보고되지 않는 비율.

본 연구에서의 취약점 해결률은 정적 분석 기준으로 동일 취약점의 재탐지 여부에 기반하며, 생성된 패치가 기존 소프트웨어의 동작을 완전히 보존하는지를 자동으로 증명하는 지표는 포함하지 않는다. 기능 보존성 검증은 저장소별 테스트 환경 구축과 안정적인 테스트 실행을 필요로 하므로 본 연구의 실험 범위를 벗어나며, 단위 테스트 생성 및 실행 기반의 자동 검증은 향후 연구 과제로 남긴다. 본 연구는 PR 기반 개발 프로세스를 전제로 하며, 생성된 패치의 기능적 동등성은 유지보수자의 코드 리뷰와 CI 테스트를 통해 최종적으로 검증되는 것을 가정한다.

4.2 전체 파이프라인 동작 성공 비율

AutoFiC 파이프라인을 91개 저장소에 적용한 결과, 86개 저장소에서 전 과정이 정상적으로 완료되어 94.5%의 파이프라인 성공률을 기록하였다. 실패한 5개 사례를 분석한 결과, 3건(3.30%)은 LLM의 토큰 제한 초과로 인해 발생하였으며, 나머지 2건(2.20%)은 GitHub API 통신 오류로 확인되었다. 이러한 결과는 제안 시스템의 설계가 별도의 개입 없이도 다수의 실제 오픈소스 프로젝트에 대해 안정적인 End-to-End 자동화를 가능하게 함을 시사한다.

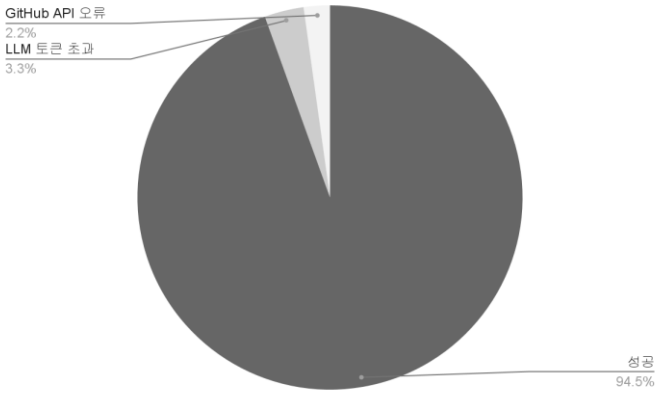


그림 4. 파이프라인 전체 동작 성공 비율

4.3 취약점 해결 여부

파이프라인이 정상적으로 완료된 86개 저장소를 대상으로, 패치 적용 전후에 동일한 조건으로 Semgrep을 재실행하여 취약점 해결 여부를 분석하였다. 그 결과, 총 441개의 취약점이 탐지되었으며 AutoFiC 적용 후 397개가 해결되어 90.0%의 취약점 해결률을 기록하였다.

표 1. CWE 유형별 취약점 해결률

취약점 유형	탐지 수	해결 수	해결률
CWE-116	2	2	100.0%
CWE-200	5	5	100.0%
CWE-276	9	8	88.9%
CWE-295	3	3	100.0%
CWE-319	75	72	96.0%
CWE-326	27	27	100.0%
CWE-327	65	59	90.8%
CWE-330	4	3	75.0%
CWE-489	9	6	66.7%
CWE-502	17	16	94.1%
CWE-532	17	15	88.2%
CWE-611	30	28	93.3%
CWE-668	7	7	100.0%
CWE-78	153	138	90.2%
CWE-79	17	7	41.2%

CWE-942	1	1	100.0%
합계	441	397	90.0%

표 1은 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 모두 적용한 AutoFiC 파이프라인에서의 CWE 유형별 취약점 해결 현황을 나타낸다. CWE-116, CWE-200, CWE-295, CWE-326, CWE-668, CWE-942는 100%의 해결률을 기록하였으며, CWE-319와 CWE-327 역시 각각 96.0%, 90.8%의 높은 해결률을 보였다. 가장 많은 빈도로 탐지된 CWE-78(Command Injection)의 경우, 153건 중 138건이 해결되어 90.2%의 해결률을 기록하였다. 반면 CWE-79(XSS)는 17건 중 7건만이 해결되어 41.2%로 상대적으로 낮은 해결률을 보였다.

표 2. XML 및 Annotation 도입 이전의 CWE 유형별 취약점 해결률

취약점 유형	탐지 수	해결 수	해결률
CWE-116	2	2	100.0%
CWE-200	5	5	100.0%
CWE-276	9	8	88.9%
CWE-295	3	3	100.0%
CWE-319	75	74	98.7%
CWE-326	27	24	88.9%
CWE-327	65	58	89.2%
CWE-330	3	3	100.0%
CWE-489	9	6	66.7%
CWE-502	17	16	94.1%
CWE-532	17	15	88.2%
CWE-611	30	28	93.3%
CWE-668	7	7	100.0%
CWE-78	153	135	88.2%
CWE-79	17	7	41.2%
CWE-942	1	1	100.0%
합계	440	392	89.1%

표 2는 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 적용하지 않은 경우의 CWE 유형별 취약점 해결 현황을 나타낸다. 전후 비교 결과, 전체 취약점 해결률은 89.1%에서 90.0%로 증가하였으며, 해결된 취약점 수는 392건에서 397건으로 총 5건 증가하였다. 특히 CWE-78과 CWE-326에서 해결된 취약점 수가 각각 3건 증가하여, 제안 기법 적용 이후 해당 유형의 패치 성능이 향상되었음을 확인할 수 있다.

이러한 결과는 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법이 취약점 수정 대상의 범위를 보다 명확히 한정함으로써, LLM이 불필요한 코드 영역을 탐색하지 않고 핵심 취약 지점에 집중하도록 유도했음을 보여준다. 특히 CWE-78, CWE-326과 같이 취약점 발생 위치와 수정 패턴이 비교적 명확한 유형에서 해결된 취약점 수가 증가한 점은, 위치 정보와 유형 정보를 명시적으로 제공하는 설계가 패치 정확도 향상에 기여했음을 수치적으로 뒷받침한다. 반면 CWE-79와 같이 출력 컨텍스트에 대한 의미적 해석이 요구되는 취약점 유형에서는 해결률 개선이 제한적으로 나타났으며, 이는 경량 컨텍스트 기반 접근 방식의 적용 범위와 한계를 동시에 시사한다.

기존의 LLM 기반 취약점 탐지 및 패치 생성 연구들은 개별 단계에서 높은 성능을 보이는 알고리즘을 제안해 왔다. 그러나 이러한 연구들을 실제 개발 환경에서 조합하여 사용하기 위해서는, 취약점 탐지 결과의 정규화, 패치 실패 처리, diff 적용 오류 대응, Pull Request 생성 및 CI 연계와 같은 추가적인 엔지니어링 작업이 요구된다. AutoFiC은 취약점 탐지–패치 생성–적용–Pull Request 생성까지의 전 과정을 하나의 자동화된 파이프라인으로 통합함으로써, 이러한 실무적 부담을 최소화하는 데 초점을 둔다. 실험 결과는 AutoFiC이 실제 오픈소스 프로젝트 환경에서도 높은 파이프라인 성공률을 유지하며 End-to-End 자동화를 안정적으로 수행할 수 있음을 보여준다.

5. 결론

본 연구는 정적 분석 도구(SAST)와 대규모 언어 모델(LLM)을 결합하여, 취약점 탐지부터 패치 생성 및 적용, Pull Request 생성까지 이어지는 DevSecOps 관점의 End-to-End 자동 보안 패치 파이프라인 AutoFiC을 제안하였다. 제안 시스템은 보안을 기존 개발 워크플로우에 자연스럽게 통합함으로써, 취약점 발견과 수정 사이의 시간적 간극을 줄이고 개발자의 수작업 부담을 완화하는 자동화된 보안 패치 환경을 제공한다.

실제 오픈소스 프로젝트를 대상으로 한 실험 결과, AutoFiC은 다양한 프로젝트 환경에서도 안정적으로 동작하며, 정적 분석 기준에서 탐지된 취약점을 자동으로 패치 생성 및 적용 단계까지 연결할 수 있음을 확인하였다. 특히 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 도입함으로써, 취약점의 위치와 유형 정보가 LLM에 보다 명확히

전달되었고, 그 결과 패치 정확도와 일관성이 전반적으로 향상되는 경향을 보였다. 이는 복잡한 AST 기반 분석을 직접 활용하지 않더라도, 경량 구조 정보를 적절히 설계하여 제공하는 것만으로도 LLM 기반 자동 패치 파이프라인의 실용성을 높일 수 있음을 시사한다.

본 연구의 주요 기여는 다음과 같다. 첫째, DevSecOps 관점에서 SAST 기반 취약점 탐지부터 GitHub 워크플로우 연계까지의 전 과정을 자동화하여, 보안 패치가 개발 파이프라인에 자연스럽게 통합되는 실용적 시스템을 구현하였다. 둘째, XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 활용하여 AST 기반 분석을 직접 활용하지 않으면서도 구조적 컨텍스트를 경량하게 제공하는 방법을 제시함으로써, 언어 의존성과 구현 복잡도를 낮추는 동시에 LLM의 취약점 이해도를 향상시켰다. 셋째, 실제 오픈소스 환경을 대상으로 한 대규모 실험을 통해 제안 기법의 실효성을 정량적으로 검증하고, 컨텍스트 도입 전후 비교를 통해 그 개선 효과를 명확히 확인하였다.

한편, 본 연구는 몇 가지 한계를 가진다. 본 실험에서는 정적 분석 도구가 보고한 결과를 실제 취약점으로 가정하여 자동 패치를 수행하였으며, 이로 인해 오탐으로 인한 불필요한 코드 수정 가능성을 명시적으로 배제하지 않았다. 이는 제안 파이프라인의 동작 특성과 자동화 흐름 자체를 평가하기 위한 실험적 선택이었으며, 오탐 여부에 따른 패치 적절성에 대한 정량적 평가는 본 연구 범위에 포함하지 않았다. 또한, CWE-79(XSS)와 같이 출력 컨텍스트에 따라 다양한 이스케이프 방식이 요구되는 취약점의 경우, 상대적으로 낮은 해결률을 보였다. 이는 경량 컨텍스트 설계가 템플릿 엔진 구조나 출력 지점의 의미적 맥락을 제한적으로만 전달하기 때문으로 해석된다. 또한 생성된 패치가 기존 기능을 완전히 보존하는지에 대한 자동화된 검증을 제한적으로 수행하였다. 취약점 제거 여부를 정적 분석 기준으로 확인하였으며, 패치 이후의 기능적 정합성이나 실행 의미 보존에 대한 체계적 검증은 향후 과제로 남아 있다. 더불어 복잡한 제어 흐름이나 프레임워크 특화 로직을 포함하는 취약점에 대해서도 추가적인 개선 여지가 존재한다.

향후 연구에서는 다중 정적 분석 도구를 교차 활용하여 공통으로 탐지된 취약점만을 선별하는 방식이나, 규칙 신뢰도 기반 필터링을 통해 오탐을 완화하는 전략을 탐색할 예정이다. 또한 경량 XML·Annotation 기반 접근법과 AST 기반 구조 정보를 선택적으로 결합하는 하이브리드 전략을 통해, 보다 복잡한 취약점 유형에 대한 대응 능력을 향상시키고자 한다. 나아가 자동 메커니즘과 단위 테스트 생성 기법을 결합하여 패치의 기능 보존성을 체계적으로 검증하고, Java, C/C++ 등 다양한 프로그래밍 언어로 확장함으로써 AutoFiC의 범용성과 실용성을 더욱 확대할 계획이다. 프레임워크별 특화 프롬프트와 도메인 지식을 반영한 패치 템플릿을 통해 난이도 높은 취약점의 해결률을 개선하고, 자동 검증 메커니즘과 단위 테스트 생성 기법을 결합하여 패치의 기능 보존성을 보다 체계적으로 검증하고자 한다. 마지막으로, Java, C/C++ 등 다양한 프로그래밍 언어로 확장하여 AutoFiC의 범용성과 실용성을 확대할 계획이다.

참고문헌

[1] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," Communications of the ACM, vol. 61, no. 4, pp. 58–66, 2018.

[2] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-Trained Language Models," Proceedings of the 45th International Conference on Software Engineering (ICSE), pp. 1482–1494, 2023.

[3] R. Macháček, A. Grishina, M. Hort, and L. Moonen, "The Impact of Fine-tuning Large Language Models on Automated Program Repair," Proceedings of the 41st International Conference on Software Maintenance and Evolution (ICSME), pp. 380–392, 2025.

[4] L. Gazzola, D. Micucci, and L. Mariani, "Automatic Software Repair: A Survey," IEEE Transactions on Software Engineering, vol. 45, no. 1, pp. 34–67, 2019.

[5] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," Information and Software Technology, vol. 53, no. 4, pp. 363–387, 2011.

[6] A. T. M. F. Rabbi and M. A. Joarder, "Automatic program repair: A systematic literature review," Systematic Literature Review and Meta-Analysis Journal, vol. 4, no. 3, pp. 1–10, 2023.

프로젝트 구조 요약을 통한 대규모 언어 모델의 구조적 한계 보완 가능성에 대한 실험적 연구

이석인, 이선아

경상국립대학교

tjrdls29@gmail.com, saleese@gnu.ac.kr

An Experimental Study on Mitigating Structural Limitations of Large Language Models Using Prompted Structural Reasoning

Seokin Lee, Seonah Lee

Gyeongsang National University

요 약

대규모 언어 모델(LLM)은 코드 수정 요청에 대해 구조적으로 타당한 변경을 제안하는 데 한계를 보이며, 특히 클래스 간 책임 경계와 변경 전파 범위를 일관되게 유지하지 못하는 문제가 있다. 본 연구는 이러한 한계를 완화하기 위한 중간 표현으로 프로젝트 구조 요약 (Project Structural Summary, PSS)를 제안하고, 프로젝트에 포함된 클래스의 책임 요약과 클래스 간 관계 정보를 중심으로 하되 코드의 상세 구현은 배제하여 구조적 판단에 필요한 최소 정보만을 제공하도록 설계된 PSS가 LLM의 구조적 판단을 보조하는 데 효과적인지를 실험적으로 검증하였다. 이를 위해 동일한 변경 요구에 대해 총 9회의 LLM 응답을 수집하고, 각 응답을 책임 적합성과 구조적 전파 인식 정확도의 두 축에서 분석하였다. 분석 결과, 구조적 전파 인식은 모든 응답에서 성공적으로 수행되었으나 책임 적합성 측면에서는 일부 응답에서 계산 로직을 담당하는 컴포넌트가 외부 상태 관리나 컨텍스트 산출 책임을 포함하는 사례가 관찰되었다. 이는 PSS가 변경 영향 범위 추론에는 효과적이지만 단일 컴포넌트의 책임 경계를 일관되게 유지하기 위해서는 추가적인 제약이나 보완이 필요함을 시사한다. 본 연구는 PSS가 LLM 기반 코드 수정 과정에서 구조적 판단을 보조할 수 있는 가능성을 제시했다는 점에서 의의를 갖으며, 향후 다양한 도메인과 평가 기준 확장을 통해 적용 범위를 검증할 필요가 있다.

1. 서 론

최근 대규모 언어 모델(Large Language Model, LLM)은 코드 생성, 수정, 리팩토링 등 다양한 소프트웨어 개발 작업에서 활용되고 있다. 자연어로 작성된 변경 요청을 바탕으로 코드 수정안을 제안하는 방식은 개발 생산성을 향상시키는 수단이 될 수 있다. 이러한 흐름에 따라 LLM을 실제 개발 과정에 통합하려는 시도가 증가하고 있으며, 단순한 코드 단위의 생성뿐 아니라 프로젝트 맥락을 고려한 변경 수행에 대한 요구 또한 확대되고 있다.

그러나 실제 프로젝트 환경에서의 코드 변경은 단일 파일이나 함수 수준의 수정에 그치지 않고, 클래스 간 책임 분리, 컴포넌트 간 의존 관계, 변경 전파 범위와 같은 구조적 판단을 수반한다. 이러한 판단은 프로젝트 전체 구조에 대한 이해를 전제로 하며, 국소적인 코드 정보만으로는 정확히 수행되기 어렵다[1,2]. 그럼에도 불구하고 현재의 LLM은 입력 길이 제약 및 프롬프트 구성의 한계로 인해 프로젝트 전체 구조를 명시적으로 입력받기 어려운 상황에서 동작하고 있다.

기존 연구들은 LLM이 코드 이해, 생성, 요약, 취약점 탐지와 같은 과업에서 우수한 성능을 보인다는 점을 보고하였다. 일부 연구에서는 AST나 데이터 흐름과 같은 구조 정보를 학습 과정에 반영하여 코드 표현의 품질을 향상시키는 방법을 제안하였다[3,4,5]. 그러나 이러한 접근들은 주로 코드 수준의 의미 이해 성능 향상에 초점을 두고 있으며, 구조적 정보를 제공받은 LLM이 실제로 설계적 판단을 얼마나 적합하게 수행하는지에 대해서는 명시적으로 평가하지 않았다.

본 연구는 이러한 한계를 극복하기 위한 방법으로, 프로젝트 전체 구조를 요약한 프로젝트 구조 요약 (Project Structural Summary, PSS)를 LLM 입력으로 제공하는 접근을 제안한다. PSS는 프로젝트를 구성하는 주요 컴포넌트와 각 컴포넌트의 책임, 그리고 컴포넌트 간 구조적 관계를 요약적으로 표현한 구조 정보로, 전체 소스 코드를 직접 제공하지 않더라도 프로젝트 수준의 맥락을 전달하는 것을 목표로 한다.

본 연구에서는 LLM이 단일 클래스에 대한 변경 요청을 수행할 때, PSS를 통해 자신이 수정해야 할 책임 범위와, 외부 컴포넌트로의 구조적 변경 전파 필요성을 인식할 수 있을 것이라는 가설을 설정한다. 즉,

PSS는 코드 생성 자체를 보조하기보다는, LLM의 설계적 판단을 지원하기 위한 입력 정보로서의 역할을 수행할 수 있을 것으로 기대한다.

제안 방법의 효과를 검증하기 위해, 본 연구에서는 소규모 프로젝트를 대상으로 LLM에 자연어 변경 요청, 대상 클래스의 기존 코드, 그리고 PSS를 함께 입력으로 제공하는 실험을 수행하였다. 결과는 책임 적합성, 구조적 전파 인식 정확도의 두 기준을 중심으로 분석하였다. 실험을 통해 PSS가 제공되지 않은 경우와 비교하여, PSS를 입력으로 제공한 경우 LLM이 구조적 판단과 관련된 응답을 보다 명확하게 수행하는 경향을 보였음을 확인하였다. 특히 PSS를 제공한 조건에서 수행한 총 4개의 테스트(각 9회, 총 36회 응답)에서 구조적 전파 인식 정확도는 97%의 성공률을 기록하였으며, 책임 적합성 평가에서도 83%의 응답이 성공으로 분류되었다. 이러한 결과는 PSS 기반 입력이 LLM의 구조적 변경 판단을 일관되게 유도하는 데 효과적임을 시사한다.

본 연구의 공헌은 다음과 같다. 먼저, 대규모 언어 모델(LLM)의 코드 수정 능력을 단순한 코드 이해나 생성 성능이 아닌, 프로젝트 구조 차원의 설계적 판단 문제로 재정의하였다. 이를 위해 전체 코드를 제공하지 않고도 프로젝트의 책임 분리와 의존 관계를 전달할 수 있는 Project Structural Summary(PSS)라는 구조 요약 표현을 제안하였다. 다음, 이를 실험적으로 검증하였다. PSS가 제공된 상황에서 LLM이 단일 클래스 수정 제약 하에서 변경 요청을 처리할 때, 책임 경계를 유지하는지와 필요한 구조적 변경 전파를 인식하는지를 실험으로 평가하였다. 실험 결과를 통해 PSS가 변경 영향 범위 인식에는 일관된 효과를 보이지만, 책임 경계 유지에는 여전히 한계가 존재함을 보였으며, 이를 통해 LLM 기반 코드 수정에서 구조 정보가 기여할 수 있는 지점과 추가 보완이 필요한 지점을 동시에 드러냈다.

본 논문은 다음과 같이 구성된다. 2장에서는 대규모 언어 모델의 설계적 판단 한계와 구조적 정보를 활용하려는 기존 연구들을 검토하고, 본 연구의 차별점을 정리한다. 3장에서는 Project Structural Summary(PSS)의 개념과 설계 원칙, 그리고 이를 활용한 LLM 기반 코드 수정 절차를 설명한다. 4장에서는 PSS의 효과를 검증하기 위한 실험 목적, 실험 절차 및 평가 기준을 제시한다. 5장에서는 실험 결과를 제시하고, 책임 적합성과 구조적 전파 인식 관점에서 결과를 분석 및 논의한다. 6장에서는 연구의 한계와 위협 요인을 다루며, 마지막으로 7장에서 연구의 결론과 향후 연구 방향을 제시한다.

2. 관련 연구

2.1 대형 언어 모델의 설계적 판단 한계

대형 언어 모델(LLM)은 자연어 및 코드 이해·생성 과업에서 우수한 성능을 보였으나, 소프트웨어 설계와 관련된 구조적 판단에는 한계를 보인다는 점이 지적되어 왔다. 기존 연구들은 LLM이 국소적인 코드 문맥이나 함수 단위의 수정에는 효과적이지만, 클래스 간 책임 분리, 변경 전파 범위, 아키텍처 수준의 의존 관계와 같은 전역적 구조 정보를 명시적으로 다루지 못한다는 문제를 보고하였다. 특히 코드 토큰을 순차적으로 처리하는 방식에서는 설계 의도가 암묵적으로만 반영되어, 구조적으로 부적절한 수정이나 보안 취약점이 발생할 가능성이 존재함이 논의되었다 [1,2]. 이러한 한계를 보완하기 위해, 일부 연구는 GitHub 이슈, PR, 커밋 메시지와 같은 개발 아티팩트를 활용하여 코드 변경의 의도와 맥락을 보조적으로 제공하는 접근을 제안하였다[6].

2.2 구조적 정보 기반 표현

코드의 구조적 의존 관계를 모델 입력이나 학습 과정에 반영하려는 연구들이 제안되었다. 대표적으로 AST (Abstract Syntax Tree), 제어 흐름, 데이터 흐름과 같은 구조 정보를 코드 토큰과 결합하여 Transformer 기반 모델의 표현 학습을 개선하는 접근이 연구되었다. GraphCodeBERT는 데이터 흐름 그래프를 코드 토큰과 함께 사전학습에 포함함으로써 구조 정보를 반영한 코드 표현을 학습하였다 [3]. UniXcoder는 AST 기반 관계를 보존하는 방식으로 코드와 자연어를 통합적으로 표현하는 모델을 제안하였다 [4]. 또한 구조 유도 어텐션 메커니즘을 활용하여 코드의 계층적·구조적 특성을 모델 내부에 반영하려는 연구도 보고되었다 [5]. Code2MCP는 코드 분석을 통해 함수 및 호출 관계를 추출하고, 이를 명시적인 구조적 명세로 변환한다[7],

2.3 구조적 정보를 LLM에 활용

구조 정보를 그래프 형태로 모델링하고, 이를 LLM과 결합하는 연구들이 등장하고 있다. 코드의 구문·제어 흐름·데이터 의존성을 통합한 그래프나, 파일·클래스·메서드 간 관계를 리포지토리 수준에서 그래프로 구성한 뒤 이를 LLM 입력 맥락으로 활용하는 접근이 제안되었다. 특히 REPOGRAPH는 리포지토리 내 구조 정보를 서브그래프 형태로 검색하여 LLM에 제공함으로써, 대규모 코드베이스에서 구조적 맥락을 선택적으로 활용하는 방법을 제시하였다 [8]. 이러한 접근은 단일 파일이나 함수 수준을 넘어 시스템 전반의 구조를 고려할 수 있다는 점에서 의의가 있다.

2.4 기존 연구와의 차이

그러나 기존 연구들은 주로 코드 이해, 생성, 요약, 취약점 탐지와 같은 과업의 성능 향상에 초점을 두고 있으며, 구조적 정보를 제공받은 LLM이 설계적 판단을 얼마나 적합하게 수행하는지를 명시적으로 평가하는 연구는 상대적으로 제한적이다. 본 연구는 구조적 책임 명세(PSS)를 LLM 입력으로 명시적으로 제공하고, 그 결

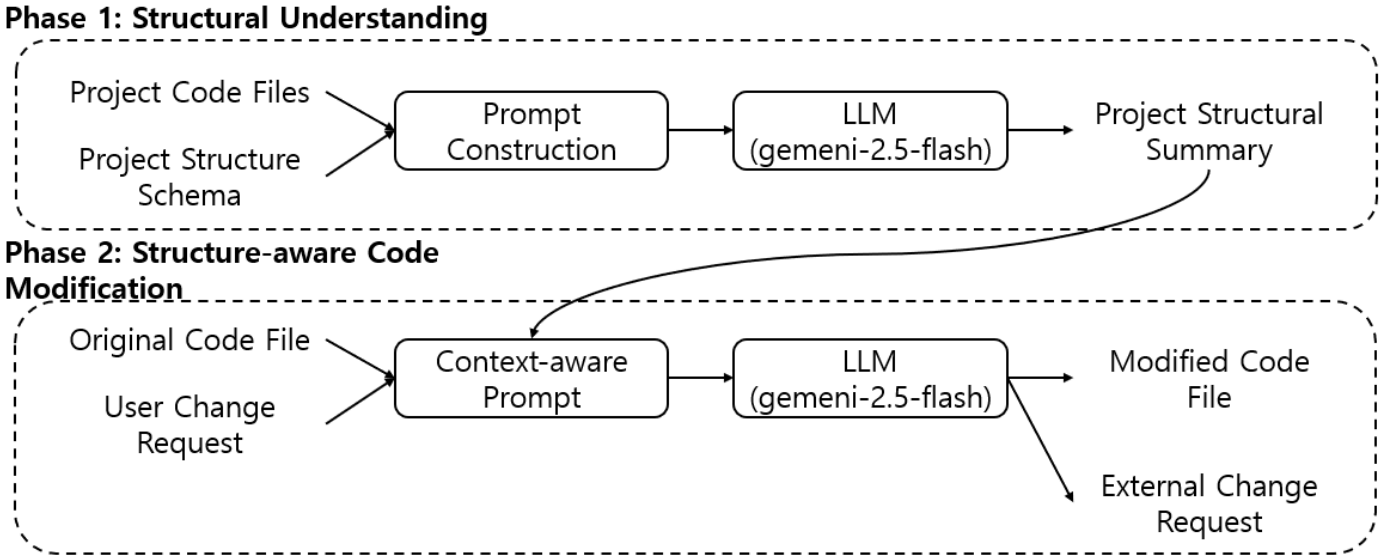


그림 1. 제안 방법 오버뷰. 단계 1은 프로젝트 코드 파일들과 프로젝트 구조 스키마 파일을 받아서 프로젝트 구조 요약 PSS를 만든다. 단계 2는 만들어진 PSS와 수정하려는 코드 파일, 사용자의 수정 요청을 받아서 프롬프트를 생성한다. 그 결과로 수정한 코드 파일과 해당 코드의 외부 수정 요청들을 포함한다

.

과가 책임 적합성과 구조적 변경 전파 인식이라는 설계 중심 기준을 얼마나 충족하는지를 평가한다는 점에서 기존 연구와 구별된다.

3. 제안 방법: PSS 기반 LLM 보조 구조

3.1 PSS 개념 정의

Project Structural Summary(PSS)는 대규모 언어 모델(LLM)이 프로젝트의 전체 구조를 명시적으로 인식할 수 있도록 제공되는 프로젝트 구조 요약본이다. PSS는 프로젝트를 구성하는 모든 클래스와 그 책임, 주요 메서드의 역할, 그리고 클래스 간의 의존 관계를 구현 세부사항 없이 구조적 관점에서만 기술한다.

본 논문에서 제시하는 PSS는 프로젝트의 구조 정보를 요약하여 표현하는 중간 표현의 하나의 구체적 설계 예시이다. 본 연구의 목적은 특정 포맷 자체의 우수성을 주장하는 것이 아니라, 클래스 책임과 의존 관계와 같은 구조적 정보가 LLM의 코드 수정 과정에서 설계적 판단을 보조하는 데 실질적인 효과가 있는지를 검증하는 데 있다. 따라서 PSS의 표현 방식이나 세부 스키마는 프로젝트 특성이나 활용 목적에 따라 다양하게 변형될 수 있다.

3.2 PSS의 설계 원칙

PSS는 LLM의 구조적 판단을 효과적으로 보조하는 것을 목표로 설계되었으며, 이를 위해 몇 가지 핵심 원칙을 따른다.

첫째, PSS에는 코드의 상세 구현을 포함하지 않는다. 이는 실제 코드 실행이나 세부 로직 추론을 지원하기 위함이 아니라 클래스 간 책임 분리와 의존 구조를 판단하기 위한 요약 정보를 제공하는 데 목적이 있기 때문이다.

둘째, 구조적 판단에 필요한 정보만을 포함한다. 모든 메서드의 내부 동작이나 조건 분기와 같은 세부 사항은 생략하고 각 클래스와 메서드가 담당하는 역할 수준의 정보만을 제공한다.

셋째, LLM 입력 크기의 제약을 고려하여 전체 프로젝트 코드를 프롬프트에 포함하기 어려운 현실적인 상황을 전제로, 구조 인식에 가장 중요한 정보만을 선별적으로 제공하는 것을 설계 목표로 한다.

본 연구에서 논의하는 입력 크기 제약은, LLM이 코드 수정을 수행할 때 전체 프로젝트 소스 코드를 프롬프트에 직접 포함하기 어려운 현실적 한계를 의미한다. PSS 자체를 생성하거나 대규모 프로젝트에 대해 출력하는 과정에서는 출력 토큰 수가 병목이 될 가능성이 존재할 수 있으나, 본 연구의 실험 범위는 이미 생성된 PSS를 구조적 맥락 입력으로 제공했을 때 LLM의 설계적 판단이 어떻게 달라지는지를 분석하는 데에 한정된다. 따라서 본 논문에서의 입력 제약 논의는 PSS 생성 비용이나 출력 병목을 다루기보다는, 전체 코드 제공 대비 최소 구조 정보를 입력으로 제공하는 설계 선택의 타당성을 설명하기 위한 전제로 해석되어야 한다.

3.3 단계 1. 프로젝트에서 전체 구도 도출

본 단계에서는 대상 프로젝트의 전체 구조를 요약한 구조 정보를 생성한다. 이를 위해 LLM에는 프로젝트 전반을 설명하는 요약 스키마와 전체 소스 코드가 입력으로 제공된다. 요약 스키마는 클래스 단위의 책임, 주요 의존 관계, 상위·하위 컴포넌트의 역할을 간략히

기술한 구조적 명세이며, 전체 소스 코드는 실제 구현 수준에서의 클래스 간 관계와 호출 구조를 파악하기 위한 근거로 활용된다.

LLM은 이 두 입력을 바탕으로 프로젝트를 구성하는 주요 클래스들의 책임 분포와 클래스 간 의존 관계를 종합적으로 분석하고, 이를 Project Structural Summary(PSS) 형태의 전체 구조 정보로 출력한다.

본 연구에서 사용한 프로젝트 구조 요약 PSS는 사전에 정의된 JSON 스키마[1]를 따르는 구조화된 표현으로 생성되며, 해당 구조 정보는 그대로 LLM 프롬프트의 입력으로 사용된다. 이를 통해 LLM이 개별 코드 조각이 아닌 프로젝트 전체의 책임 분리와 변경 전파 구조를 전제 지식으로 활용할 수 있도록 한다.

PSS는 두가지 핵심 요소로 구성된다. 먼저 클래스에 대한 요약으로서 프로젝트에 포함된 모든 클래스의 이름, 담당하는 책임, 주요 속성, 메서드와 그 역할을 포함하며 관계에 대한 요약으로서, 클래스 간의 의존, 포함, 상속 등의 관계를 명시적으로 표현한다.

PSS는 자동 생성되었으나, 생성 과정에서의 명백한 구조적 오류로 인해 실험이 왜곡되는 것을 방지하기 위해, 연구자가 실험 전 최소한의 검토를 수행하였다.

3.4 단계 2. 전체 구조를 활용해서 수정 결과 도출

본 실험에서 LLM에 제공되는 입력은 사용자 변경 요청, 대상 클래스의 기존 코드, 그리고 프로젝트 전체 구조를 요약한 Project Structural Summary(PSS)로 구성된다. 사용자 변경 요청은 특정 클래스에 대해 요청하게 된다. 예를 들어 FeePolicy 클래스에 대해 “출금 수수료 정책을 고정 비율이 아닌 일일 누적 출금 금액 기준으로 변경하라”는 자연어 명령으로 주어질 수 있다.

LLM의 출력은 단일 클래스에 대한 수정 결과와, 해당 수정만으로는 사용자 요청을 완전히 만족할 수 없는 경우 필요한 외부 수정 요청의 목록으로 구성된다. 수정 결과는 지정된 클래스의 전체 소스 코드 형태로 제시되며, 외부 변경 요청은 수정이 필요한 클래스와 메서드, 그리고 그 변경이 요구되는 구조적 이유를 자연어로 명시한다.

4. 실험 설계

4.1 실험 목적

Project Structural Summary(이하 PSS)를 입력으로 제공했을 때, LLM이 단일 변경 요청에 대해 책임을 침범하지 않는 코드 수정과, 필요한 구조적 전파를 정확히 인식할 수 있는지를 평가한다.

4.2 대상 프로젝트

본 연구에서는 PSS가 대규모 언어 모델의 구조적 판단을 어떻게 보조하는지를 관찰하기 위해, 단일 도메인에 기반한 소규모 은행 시스템 프로젝트[2]와 해당 프로젝트에 대한 단일PSS[3]를 실험 대상으로 사용하였다. 해당 프로젝트는 실제 금융 시스템을 모사하되, 실험 목적에 불필요한 복잡성을 배제하고 클래스 간 책임 분리와 변경 전파 구조가 명확히 드러나도록 설계된 사례이다.

프로젝트는 다음과 같은 4개의 클래스로 구성된다.

표1 소규모 은행 프로젝트의 클래스들

클래스명	역할
Account	계좌의 상태를 관리하고 입출금 요청을 처리하는 역할을 담당하며, 수수료 계산과 거래 기록이라는 하위 책임을 각각 FeePolicy와 Transaction-History에 위임하는 구조를 가진다
FeePolicy	출금 시 적용되는 수수료 계산 로직만을 담당하는 계산을 담당한다.
Transaction-History	계좌별 거래 내역을 기록·보관하는 책임을 갖는다.
BankService	외부 사용자에게 계좌 생성 및 입출금 기능을 제공하며, 내부 도메인 로직에는 관여하지 않는다

표 1에서 보듯이 계좌 관리(Account), 거래 이력 관리(TransactionHistory), 수수료 정책 계산(FeePolicy), 그리고 외부 인터페이스 역할을 수행하는 상위 서비스 컴포넌트(BankService)로 구성된다.

이와 같은 클래스 구성은 단일 클래스에 대한 코드 수정이라는 제약 하에서, LLM이 컴포넌트 간 책임 경계를 유지하면서 변경 요청을 처리할 수 있는지를 관찰하기 위해 설계되었다. 이를 위해 일부 클래스는 직접 수정 대상, 일부 클래스는 간접적 영향 대상으로 구분되며, 상위 컴포넌트는 수정 대상에서 제외하였다. 이러한 구분을 통해, PSS에 명시된 책임 및 의존 관계 정보가 LLM의 추론 과정에서 변경 범위 인식과 책임 판단에 어떻게 활용되는지를 분석할 수 있도록 하였다.

4.3 LLM 작업 요청 시나리오

각 작업 요청 시나리오는 하나의 클래스만을 직접 수정 대상으로 지정하며, 변경 요청의 성격에 따라 책임 적합성과 구조적 전파 인식 능력이 서로 다르게 요구되도록 구성하였다.

표 2: 실험 시나리오

[1]https://github.com/tjrdls/PSS/blob/master/IRSchema.json

[2]https://github.com/tjrdls/PSS/tree/master/bank_system/code

[3]https://github.com/tjrdls/PSS/blob/master/bank_system/IR.json

Task ID	설명
TEST1-1	FeePolicy 클래스를 수정 대상으로 지정하고, “출금 수수료 정책을 고정 비율이 아니라 일일 누적 출금 금액 기준으로 변경하라”는 요청을 제시하였다. 이 요청은 수수료 계산 로직 자체의 변경과 더불어, 누적 출금 금액이라는 외부 상태 정보에 대한 구조적 의존 인식이 필요한 시나리오에 해당한다.
TEST1-2	FeePolicy 클래스를 수정 대상으로 하여, “출금 수수료 정책을 금액 구간별 누진제로 변경하라. 100 이하: 0%, 101~1000: 1%, 1001 이상: 2%”는 요청을 입력으로 사용하였다. 이 요청은 외부 상태나 다른 컴포넌트와의 상호작용이 필요 없는 순수 계산 로직 변경에 해당하며, 단일 컴포넌트의 책임 범위 내에서 변경이 완결되는 경우를 관찰하기 위한 기준 사례로 활용되었다.
TEST2-1	TransactionHistory 클래스를 대상으로, “출금 실패 시에도 실패 기록을 남기도록 하라”는 변경 요청을 제공하였다. 이 시나리오는 출금 성공·실패 판단 책임이 어느 클래스에 위치해야 하는지를 올바르게 분리할 수 있는지, 즉 기록 책임과 판단 책임의 경계를 유지하는 능력을 평가하기 위해 포함되었다.
TEST2-2	TransactionHistory 클래스를 대상으로, “출금 내역에서 일일 누적 출금 금액을 조회할 수 있도록 하라”는 요청을 제시하였다. 이 요청은 구조적으로 외부 클래스 수정이 필요한지에 대한 판단과 불필요한 변경 전파를 억제하는 능력을 평가하기 위해 포함되었다.

4.4 실험 및 평가 방법

본 실험에서는 대규모 언어 모델(LLM)에 구조 요약 정보(PSS), 단일 클래스의 기존 코드, 그리고 자연어로 기술된 변경 요청을 함께 입력[4]으로 제공하고, 이에 대한 LLM의 응답을 분석 대상으로 삼는다.

실험은 표2의 4가지의 수정 요청 작업을 각 9회 반복해서 수행하였으며, 그 응답을 평가하였다.

책임 적합성을 평가하기 위한 방법으로서는 다음 3가지 질문을 모두 만족했을 때 성공으로 분류하였고, 하나라도 위배되는 경우에는 실패도 분류하였다.

(1) 수정된 코드가 지정된 하나의 클래스만을 수정하였는가?
(2) PSS에 명시된 책임 내에서만 수정이 이루어졌는가?
(3) 다른 클래스의 책임을 침범하지 않았는가?

구조적 전파 인식 정확도를 평가하기 위한 방법으로서 다음 3가지 질문을 모두 만족했을 때 성공으로 분류하였고, 하나라도 위배되는 경우에는 실패도 분류하였다.

(1) 필요한 경우에 한해 수정 요청이 생성되었는가?
(2) 구조적으로 필요한 외부 변경을 모두 수정 요청에 명시했는가?
(3) PSS에 정의된 책임/의존 관계를 근거로 변경 필요성을 명시했는가?

[4]전체 프롬프트 : https://github.com/tjrdls/PSS/blob/master/prompt.txt

5. 실험 결과 및 분석

5.1절에서 PSS 미제공 환경에서의 실험 결과를 보인다. 이 경우, 수정 결과는 대부분 실패하였기 때문에, 직접적인 비교보다는 어떤 실패인지에 대한 유형을 분석하였다. 다음 5.2절에서 PSS 제공 환경에서의 성률을 보였다.

5.1 PSS 미제공 환경에서의 구조적 추론 편차

본 절에서는 PSS가 제공되지 않은 환경에서, 대규모 언어 모델(LLM)이 단일 클래스 수정 요청을 처리할 때 나타나는 구조적 추론 편차를 분석한다. 이를 통해 본 연구가 다루는 문제의 성격을 명확히 하고자 한다.

PSS가 제공되지 않은 경우, TEST1-2를 제외한 모든 응답은 최소 하나 이상의 문제를 발생시켰다. LLM은 수정 대상 클래스의 책임을 확장하거나 요청에 포함되지 않은 클래스 간 관계 및 외부 구조를 추론하는 양상을 반복적으로 보였다. 특히 책임의 소유 주체, 호출 관계, 다른 클래스의 구조적 제약과 같은 정보가 명시되지 않을 때, 모델은 이를 추론을 통해 하는 경향을 보였다.

표 3은 PSS가 제공되지 않은 조건에서 수행된 TEST1-1, TEST2-1, TEST2-2의 모든 응답(각 9개)을 대상으로, 각 응답이 실패한 구조적 판단 차원을 책임, 관계, 다른 클래스의 구조라는 세 범주로 정리한 것이다.

표 3: PSS 미제공 실패 유형

Task ID	문제 범주	사례 수	실패 유형
TEST1-1	책임	6/9	책임 경계 오인
TEST1-1	관계	7/9	관계 근거 없는 외부 의존 추론
TEST1-1	외부 구조	5/9	외부 구조 가정
TEST2-1	책임	5/9	기록-판단 책임 혼합
TEST2-1	관계	7/9	호출자 역할 추론
TEST2-1	외부 구조	6/9	상태 전파 구조 생략
TEST2-2	책임	5/9	집계 기준 책임 혼재

TEST2-2	관계	4/9	도메인 관계 확장 추론

문제 범주별로 보면, 책임 판단 차원에서는 계산·기록 컴포넌트가 상태 소유나 도메인 판단 책임을 흡수하는 사례가 반복되었고, 관계 판단 차원에서는 클래스 간 호출 관계가 구조적 근거 없이 응답마다 다르게 설정되었다. 또한 외부 구조적 맥락이 주어지지 않은 경우, 모델은 해당 구조를 설계 대상으로 명시하지 않고 암묵적 전제로 처리하였다.

반면 TEST1-2는 수정 요청이 순수 계산 로직에 한정되어 다른 클래스와의 구조적 상호작용이 발생하지 않았기 때문에, PSS 없이도 안정적인 응답이 가능하였다. 이는 구조적 실패가 모델의 구현 능력 부족에서 비롯된 것이 아니라, 구조 판단에 필요한 전제가 제공되지 않았을 때 발생하는 추론 문제임을 시사한다.

5.2 PSS 제공 환경에서의 평가 결과

본 연구에서는 총 4개의 변경 요청(TEST1~TEST4)에 대해 각 9개의 응답을 생성하여, 책임 적합성과 구조적 전파 인식 정확도라는 두 가지 평가 축으로 분석을 수행하였다.

표4 테스트별 평가 결과 요약

Test \ 평가 축	성공	실패	성공률
TEST1-1	책임 적합성	6/9	3/9	67%
	구조적 전파 인식 정확도	9/9	0/9	100%
TEST1-2	책임 적합성	9/9	0/9	100%
	구조적 전파 인식 정확도	9/9	0/9	100%
TEST2-1	책임 적합성	7/9	2/9	78%
	구조적 전파 인식 정확도	9/9	0/9	100%
TEST2-2	책임 적합성	8/9	1/9	89%
	구조적 전파 인식 정확도	8/9	1/9	89%
Total	**책임 적합성**	**30/36**	**6/36**	**83%**
	구조적 전파 인식 정확도	**35/36**	**1/36**	**97%**

TEST1-1에서 책임 적합성 평가에서 실패한 응답은 총 3건(응답3, 5, 8) 이었으며 이들 모두 자신의 책임 범위를 초과한 설계를 제시하였다. 응답 3에서는 계산 전용 컴포넌트가 외부 상태 식별자 및 조회 책임을 직접 포함하였다. 응답 5에서는 계산 전용 컴포넌트 내부에서 시간적, 상태적 컨텍스트를 직접 산출하였다. 응답 8에서는 상태 관리 및 누적 정보에 대한 책임이 계산 로직을 담당하는 컴포넌트에 작성되었다.

TEST2-1에서는 2개의 응답이 책임 적합성 평가에서 실패하였다. 이 응답은 출금 실패 기록 저장이라는 요구 자체는 충족하였으나, 출금 성공·실패 여부를 판단하는 로직을 TransactionHistory 내부에 직접 포함하여, 판단 책임을 침범한 사례로 분류되었다.

TEST2-2에서는 1개의 응답이 책임 적합성과 구조적 전파 인식 정확도 모두에서 실패하였다. 해당 응답은 TransactionHistory의 수정만으로 충분한 상황에서, Account 클래스에 대한 추가 변경을 수정 요청으로 명시함으로써 불필요한 구조적 전파를 발생시켰다.

5.3 분석

실험 결과에 따르면, 전체 테스트를 통틀어 구조적 전파 인식 정확도 항목에서는 대부분의 응답이 성공으로 평가되었다. 특히 외부 컴포넌트에 대한 변경 필요성이 존재하는 경우에는, PSS가 변경 요청의 구조적 영향 범위를 인식하고 이를 수정 요청으로 명시하도록 유도하는 데 일관된 효과를 보였다. 반면, 구조적으로 외부 변경이 필요 없는 요청(TEST1-2)에서는 모든 응답이 추가적인 변경을 요구하지 않고 단일 컴포넌트 내부 수정으로 종료되어, 구조적 전파 인식 정확도 평가에서도 전면 성공으로 분류되었다. 이는 PSS가 변경 전파가 필요한 경우와 필요하지 않은 경우를 모두 구분하는 데 효과적으로 작동했음을 의미한다.

책임 적합성 평가에서는 테스트 유형에 따라 상이한 결과가 관찰되었다. 수수료 계산 로직 변경을 다룬 TEST1-2의 경우, 요청이 단일 컴포넌트의 순수 계산 책임에 한정된 변경이었기 때문에 모든 응답이 책임 적합성 평가에서 성공하였다. 반면, 상태 조회나 기록 관리와 같이 책임 경계 해석이 요구되는 테스트들에서는 일부 응답에서 실패가 발생하였다. 실패한 응답들은 공통적으로 PSS에서 부여된 책임 범위를 초과하여, 계산 로직 컴포넌트가 외부 상태 판단, 시간적 컨텍스트 산출, 또는 누적 정보 관리와 같은 책임을 내부에 포함하는 설계를 제시하였다.

책임 적합성 평가에서 성공한 응답들에서는 계산 로직을 담당하는 컴포넌트가 외부 상태나 누적 정보를 직접 참조하지 않고, 필요한 정보만을 매개변수로 전달받아 계산만을 수행하는 구조를 일관되게 유지하였다. 또한 외부 컴포넌트의 수정이 구조적으로 필요한 경우에만 이를 수정 요청으로 명시함으로써, 자신의 책임 범위와 외부 책임 범위를 명확히 구분하는 모습을 보였다.

이러한 결과는 PSS가 변경에 따른 구조적 영향 범위와 전파 필요성을 인식하는 데에는 충분한 정보를 제공하였으나, 모든 상황에서 컴포넌트의 책임 경계를 보수적으로 유지하도록 유도하는 데에는 한계를 가졌음을 보여준다. 특히 순수 계산 변경과 달리, 상태·기록·판단 책임이 얽힌 변경 요청에서는 PSS가 제공된 조건에서도, 일부 응답에서 판단 책임이 명시된

책임 구조와 일치하지 않는 위치에 배치되는 설계 선택이 관찰되었다.

5.4 전체 코드 제공 조건에서의 보조적 관찰

본 절에서는 PSS 제공 조건에서 관찰된 구조적 판단 양상이, 전체 코드가 제공된 조건에서도 유사하게 나타나는지를 보조적으로 확인한다. 본 관찰은 정량적 비교 실험을 목적으로 하지 않으며, 구조 정보를 요약 제공하는 접근(PSS)의 타당성을 보완적으로 검증하기 위한 참고적 분석에 해당한다.

표5. 전체 코드 제공 조건에서의 테스트별 평가 요약

Test	평가 축	성공	실패	성공률
TEST1-1	책임 적합성	7/9	2/9	78%
	구조적 전파 인식 정확도	7/9	2/9	78%
TEST1-2	책임 적합성	9/9	0/9	100%
	구조적 전파 인식 정확도	9/9	0/9	100%
TEST2-1	책임 적합성	9/9	0/9	100%
	구조적 전파 인식 정확도	9/9	0/9	100%
TEST2-2	책임 적합성	6/9	3/9	67%
	구조적 전파 인식 정확도	6/9	3/9	67%
Total	**책임 적합성**	**31/36**	**5/36**	**86%**
	구조적 전파 인식 정확도	**31/36**	**5/36**	**86%**

전체 코드 제공 조건에서도 구조적 판단 양상이 PSS 제공 조건과 유사하게 나타났다는 점은, 구조 판단에 필요한 핵심 구조 정보가 전체 코드의 양적 정보가 아니라 구조 관계와 책임 중심의 정보임을 시사하며, 이러한 정보가 PSS를 통해 충분히 요약되어 제공되고 있음을 보조적으로 뒷받침한다. 이는 구조적 판단의 기반이 코드 상세 구현이 아니라, 컴포넌트 간 책임 분담과 의존 관계에 있음을 보여주며, 구조 정보를 요약 제공하는 접근(PSS)이 구조 판단에 필요한 최소 핵심 정보를 효과적으로 전달하고 있음을 의미한다.

특히 구조적 전파 판단과 관련하여, PSS 조건에서는 책임 관계 및 의존 구조가 명시적으로 제공되었기 때문에 변경 전파 인식이 보다 안정적으로 유도되었을 가능성이 크다. 반면 전체 코드 제공 조건에서는 동일한 구조 관계가 암묵적으로 존재함에도 불구하고, 이를 추론하는 과정에서 일부 응답의 구조 판단 일관성이 저하되는 양상이 관찰되었다.

5.5 논의

본 실험의 결과는 PSS기반 입력이 LLM의 구조적 영향 범위 추론 능력을 안정적으로 보조할 수 있음을 보여준다. 대부분의 응답에서 구조적 전파 인식 정확도가 성공으로 평가되었다는 점은, LLM이 변경 요청을 단일 컴포넌트의 국소적 수정으로만 처리하지 않고, 프로젝트 구조 차원에서 필요한 외부 변경을 인식하고 명시할 수 있었음을 의미한다.

반면, 책임 적합성 평가에서 일부 실패가 발생한 점은, 구조 정보를 제공하더라도 LLM이 항상 컴포넌트의 책임 경계를 보수적으로 유지하지는 않는다는 점을 보여준다. 특히 실패한 응답들 에서는 계산 로직을 담당하는 컴포넌트가 외부 상태, 시간적 컨텍스트, 누적 정보 관리 등 구조적으로 분리되어야 할 책임을 내부로 흡수하는 경향이 관찰되었다. 이는 LLM이 기능적 요구를 충족하는 방향으로 설계를 수렴시키는 과정에서, 책임 분리보다는 응집된 구현을 선택할 가능성이 있음을 의미한다.

향후 연구에서는 본 연구의 범위와 한계를 확장하는 방향으로 추가적인 검증이 필요하다. 첫째, 본 연구는 단일 도메인과 비교적 단순한 구조를 가진 프로젝트를 대상으로 실험을 수행하였다. 실제 개발 환경에서는 도메인 특성과 아키텍처 구조가 상이한 프로젝트들이 존재하므로, 다양한 도메인과 구조적 특성을 가진 사례를 대상으로 PSS의 적용 가능성을 검증할 필요가 있다.

또한, 본 연구에서 사용한 평가는 책임 적합성과 구조적 전파 인식이라는 제한된 평가 축에 한정되었다. PSS는 변경 최소화, 설계 일관성 유지, 추론 안정성 등 추가적인 측면에서도 기여할 가능성이 있으나, 이러한 요소들은 본 연구에서 정량적으로 검증되지 않았다. 향후 연구에서는 응집도(cohesion) 및 결합도(coupling)와 같은 설계 품질 메트릭을 활용한 평가를 추가하거나, PSS로 표현된 구조 정보를 MCP와 같은 방식으로 활용하여 구조 정보가 LLM의 설계 판단 과정에서 어떻게 사용되는지를 보다 정밀하게 분석할 수 있을 것이다.

6. 위협 요인 및 한계

6.1 사례 일반화 문제

본 연구에서는 단일 소규모 프로젝트를 대상으로 PSS의 효과를 검증하였다. 실험에 사용된 프로젝트는 비교적 단순한 클래스 구조와 명확한 책임 분리를 전제로 설계되었다. 따라서 보다 복잡한 대규모 시스템이나, 이벤트 기반 구조, 마이크로서비스 환경 등의 구조적 단위와 책임 분할 방식이 상이한 아키텍처 스타일에 일반화되기에는 한계가 있다.

특히, 프로젝트 규모가 커진다면, PSS가 지나치게 커져 활용성이 저하될 수 있으며, 이에 따라 LLM의 구조 인식 성능 또한 상이하게 나타날 가능성이 있다. 따라서 본 연구의 결과는 PSS가 구조적 판단 보조 수단으로서 가능성을 보였다는 수준에서 해석되어야 하며, 다양한 규모와 도메인의 프로젝트에 대한

추가적인 검증이 필요하다.

6.2 평가 기준의 단순화

본 연구에서 제안한 PSS는 LLM의 구조적 판단을 다각도로 보조하는 것을 목표로 하지만, 본 연구에서는 PSS의 여러 잠재적 장점 중 책임 적합성과 구조적 전파 인식이라는 두 가지 측면에 대해서만 실험적으로 평가하였다.

PSS는 이외에도 변경 범위의 최소화, 설계 일관성 유지, 추론 안정성 등과 같은 추가적인 기여 가능성을 가질 수 있으나, 이러한 효과를 검증하기 위해서는 여러 도메인과 복합적인 변경 시나리오를 포함하는 추가 실험이 필요하다.

7. 결론

본 연구는 대규모 언어 모델(LLM)이 프로젝트 단위의 코드 변경을 수행하는 과정에서 나타나는 구조적 한계, 특히 컴포넌트 간 책임 분리와 변경에 따른 영향 범위 판단의 어려움에 주목하였다. 이러한 한계의 원인으로 LLM이 전체 프로젝트 구조를 직접적으로 인식하기 어렵다는 점을 지적하고, 이를 보완하기 위한 방법으로 Project Structural Summary(PSS)를 활용한 구조적 입력 방식을 제안하였다.

소규모 프로젝트를 대상으로 수행한 실험 결과, PSS가 제공된 경우 LLM은 변경 요청에 수반되는 외부 수정 필요성을 모두 인식하고 이를 명시적으로 표현하였다. 즉, 변경에 따른 구조적 전파 인식 측면에서는 모든 실험 응답에서 일관된 성공을 보였다. 반면, 단일 컴포넌트의 책임 경계를 유지하는 측면에서는 일부 실패 사례가 관찰되어, 구조 정보가 제공되더라도 설계 책임 판단이 항상 안정적으로 이루어지지는 않음을 확인하였다.

이러한 결과는 PSS가 LLM의 변경 영향 범위 추론과 프로젝트 구조 인식을 보조하는 입력 표현으로서 유의미한 가능성을 지님을 보여준다. 동시에, 책임 경계 유지와 같은 설계 판단을 완전히 자동화하기에는 여전히 한계가 존재함을 드러낸다.

참고문헌

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al., Evaluating Large Language Models Trained on Code, arXiv preprint arXiv:2107.03374, 2021.

[2] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri, Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions, IEEE Symposium on Security and Privacy Workshops, 2022.

[3] Da Guo, Xingyi Zhang, Dong Liu, and Jian Wang, GraphCodeBERT: Pre-training Code Representations with Data Flow, Proceedings of the International Conference on Learning Representations (ICLR), 2021.

[4] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin, UniXcoder: Unified Cross-Modal Pre-training for Code Representation, Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL), 2022.

[5] Hsuan-Tien Wu, Yao Li, Zhen Zhang, Jingjing Wang, and Ashley Wang, Code Summarization with Structure-Induced Transformer, Findings of the Association for Computational Linguistics (ACL), 2021.

[6] Ling Yue, Jian Yin, Shimin Di, Libin Zheng, Chaoqian Ouyang, Shaowu Pan, and Min-Ling Zhang, Code2MCP: Transforming Code Repositories into MCP Services, arXiv preprint arXiv:2509.05941, 2025.

[7] Ziv Nevo, Orna Raz, and Karen Yorav, Uncovering Code Insights: Leveraging GitHub Artifacts for Deeper Code Understanding, arXiv preprint arXiv:2511.03549, 2025.

[8] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu, REPOGRAPH: Enhancing Repository-Level Software Engineering with Graph-Based Context Retrieval, Proceedings of the International Conference on Learning Representations (ICLR), 2025.

커버리지 피드백을 활용하는 LLM 기반 단위 테스트 자동 생성 기법

류병우

울산과학기술원

captainnemo9292@unist.ac.kr

Enhancing LLM-driven Unit Test Generation with Coverage Feedback

Byoung Woo Yoo

Ulsan National Institute of Science and Technology

요 약

대규모 언어 모델(Large Language Models; LLMs)은 이해하기 쉬운 단위 테스트를 생성하는 데 있어 가능성을 보여주었으나, 동적 분석이나 실행 피드백을 활용하지 못하기 때문에 높은 테스트 커버리지를 달성하는 데에는 여전히 한계가 있다. 본 연구에서는 테스트 커버리지 증진을 위해 동적 커버리지 피드백을 통합한 새로운 LLM 기반 JavaScript 테스트 생성 기법인 COVERPILOT을 제안한다. 5개의 오픈소스 JavaScript 프로젝트를 대상으로 COVERPILOT을 평가하고, 기존 자동 테스트 생성 도구인 Nessie와 선행 LLM 기반 기법인 TESTPILOT과 비교하였다. 실험 결과, COVERPILOT은 문장 커버리지와 분기 커버리지 모두에서 기존 기법을 유의미하게 상회하였으며, 이는 LLM 기반 테스트 생성에 커버리지 피드백을 통합하는 접근법의 효과를 입증한다.

1. 서 론

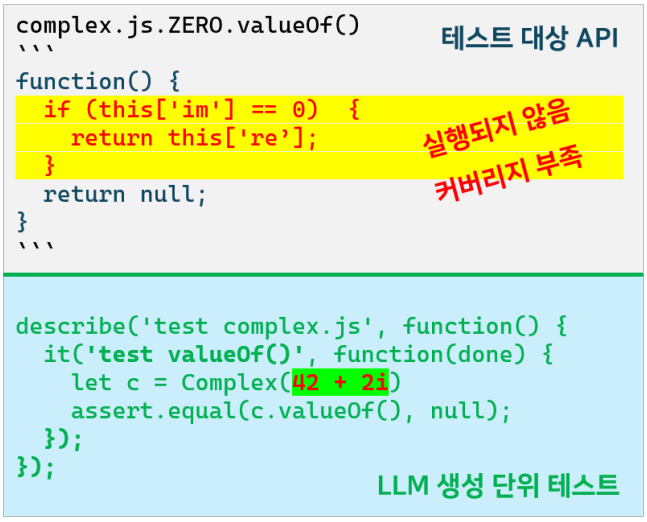


그림 1 LLM 기반 단위 테스트 생성 예시

단위 테스트(Unit Test)는 개별 코드 단위(e.g., 소프트웨어 프로젝트의 API)의 기능적 정확성을 검증하는 데 사용된다. 이러한 테스트는 소프트웨어 품질 유지에 있어서 중요한 역할을 수행한다. 그런데, 단위 테스트를 수작업으로 작성하는 과정은 개발자의 시간과 노력을 요구한다.

이러한 문제를 해결하기 위해 퍼징(fuzzing) [4], [5], 무작위 테스트(random testing) [6], [1], 탐색 기반(search-based) 기법 [7], [8] 등 다양한 자동 테스트 생성 기법들이 제안되어 왔다. 이러한 기법들은 일정 수준의 효과를 보이지만, 생성된 테스트가 이해하기 어려운 변수명이나 불필요한 검증문(assertion)을 포함하는 경우가 많아 가독성이 떨어진다는 문제가 있다. 그 결과, 자동 생성된 테스트는 개발자가 작성한 테스트에 비해 이해하기 어렵다.

최근에는 대규모 언어 모델(Large Language Models; LLM)이 새로운 대안으로 나타나고 있다. LLM은 대규모 코드 및 자연어 데이터로 학습되어 가독성 뛰어난 코드를 생성할 수 있는데, 소프트웨어 테스팅 과제에서도 의미 있는 변수명과 검증문을 포함한 테스트를 생성할 수 있음을 보였다. 예를 들어, TESTPILOT[3]과 같은 기법은 추가적인 학습 없이도 기존 LLM을 활용하여 고품질 단위 테스트를 생성할 수 있다.

그러나, LLM 기반 테스트 생성 기법은 테스트 커버리지 측면에서 여전히 한계를 가진다. 테스트 커버리지는 테스트 실행 시 소스 코드가 얼마나 실행되었는지를 측정하는 지표이다. 통상적으로, 높은 테스트 커버리지는 결함 발견 가능성이 많음을 의미한다. 그런데, 기존 LLM 기반 테스트 생성 기법은 다음과 같은 이유로 충분한 커버리지를 확보하지 못한다:

(1) LLM은 커버리지 최적화를 명시적으로 수행하지 않는다. TESTPILOT과 같은 기법은 기능적으로 올바른 테스트 생성에 집중하지만, 생성된 테스트가 모든 실행 경로를 탐색하는지는 확인하지 않는다. 예를 들어, **그림 1**의 테스트 생성 예시에서 complex.js.ZERO.valueOf

API는 허수(im)가 0일 경우 실수(re)를 반환하고, 그렇지 않으면 null을 반환한다. 해당 코드를 완전히 커버하기 위해서는 im이 0인 경우와 0이 아닌 경우를 모두 테스트해야 한다. 그러나 LLM은 허수가 0이 아닌 경우(e.g., 42 + 2i)만 테스트하고, im이 0인 경우를 고려하지 않아 완전한 커버리지를 달성하지 못한다.

(2) LLM은 테스트 실행 피드백이나 동적 분석을 활용할 수 없다. 개발자 및 기존 자동 테스트 생성 도구는 동적 분석을 응용해 테스트 커버리지를 점진적으로 보완할 수 있다. 예를 들어, **그림 1** 사례에서 동적 분석을 실행하면 im이 0인 분기가 테스트에 의해 실행되지 않았다는 점을 쉽게 확인할 수 있다. 반면, LLM은 실행 피드백을 직접 활용할 수 없기 때문에 테스트 커버리지를 향상시키지 못한다.

본 연구에서는 이러한 문제를 해결하기 위해 커버리지 피드백을 통합한 LLM 기반 테스트 자동 생성 기법: COVERPILOT을 제안한다. COVERPILOT은 커버리지 피드백 루프를 도입하여 테스트 커버리지를 점진적으로 개선할 수 있다. 우선, LLM으로 초기 단위 테스트를 생성하고 커버리지를 측정하여 아직 실행되지 않은 코드 블록을 식별한다. 다음으로, 커버리지 증진을 위해 실행되지 않은 코드 블록을 목표로 하는 테스트를 추가적으로 생성한다. 완전한 커버리지를 달성할 때까지 커버리지 피드백 루프를 반복하여 LLM이 테스트 커버리지를 점진적으로 개선할 수 있도록 한다

본 연구에서는 5개의 오픈소스 JavaScript 프로젝트에 대하여 COVERPILOT의 테스트 생성 성능을 평가하고, 기존 자동 테스트 생성 도구인 Nessie 및 기존 LLM 기반 기법인 TESTPILOT과 비교한다. 실험 결과, COVERPILOT은 문장(statement) 커버리지와 분기(branch) 커버리지 모두에서 두 기존 기법보다 유의미하게 높은 성능을 보였으며, 이는 LLM 기반 테스트 생성에 커버리지 피드백을 통합하는 접근의 효과를 입증한다.

본 연구가 기여하는 내용은 다음과 같다.

1. 동적 커버리지 피드백 루프를 통합하여 JavaScript 테스트 커버리지를 반복적으로 개선하는 새로운 LLM 기반 테스트 생성 기법 COVERPILOT을 제안한다.
2. 오픈소스 프로젝트에 대한 정량적 평가를 통해, 커버리지 피드백을 LLM에 통합하는 접근이 LLM 기반 단일 테스트 생성 및 기존 자동 테스트 생성 기법에 비해 커버리지가 우수함을 보인다.

2. 관련 연구

2.1. 자동 테스트 생성

자동 테스트 생성 분야에는 퍼징(fuzzing) [4], [5], 무작위 테스트(random testing) [6], [1], 탐색 기반(search-based) 기법 [7], [8] 등 다양한 기법이 연구되었다. 일반적으로 프로그램 분석이나 실행 피드백을 활용해 프로그램의 제어 흐름 및 데이터 흐름 경로를 탐색하고, 그 결과를 바탕으로 커버리지를 최대화하도록 테스트를 생성한다.

예를 들어, JavaScript API를 대상으로 하는 단위 테스트 생성 도구인 Nessie[1]는 테스트를 실행 피드백을 기반으로 테스트를 점진적으로 생성하는 피드백 지향 테스트 생성 알고리즘을 사용한다. 이 피드백은 생성되는 테스트가 보다 복잡한 API 호출을 하도록 유도한다.

기존 자동 테스트 생성 기법은 테스트 커버리지를 최적화할 수 있는 반면, 가독성 있는 테스트를 만들어내는 데에 있어서는 한계를 지닌다. 첫째, 생성된 테스트는 직관적이지 않은 변수명을 사용하는 경우가 많아 개발자가 작성한 테스트에 비해 이해하기 어렵다. 둘째, 불필요한 검증문(assertion)이 과도하게 포함되는 문제가 발생한다. 그 결과, 자동 생성된 테스트 코드는 가독성에 떨어지며, 별다른 처리 없이 프로젝트에 통합하기 어렵다.

2.2. LLM 기반 테스트 생성

최근에는 자연어 및 소스 코드 데이터를 학습한 대규모 언어 모델(LLM)을 단위 테스트 생성에 활용하는 추세다. LLM은 자연스러운 자연어 및 코드를 생성할 수 있으므로, 가독성이 높고 유의미한 단위 테스트를 생성할 수 있을 것으로 기대된다.

구체적으로, TESTPILOT[3]은 JavaScript API에 대해 LLM을 활용하여 테스트를 자동 생성하는 기법으로, 추가적인 학습 없이도 단위 테스트를 효과적으로 생성할 수 있음을 보였다. TESTPILOT은 API 시그니처, 기술 문서, 호출 예시, 소스 코드 등을 조합해 프롬프트를 구성하고, 이를 바탕으로 기능적으로 정확한 단위 테스트를 생성한다.

그런데, TESTPILOT은 테스트 커버리지 최적화에서는 한계를 지닌다. 개발자 혹은 기존 자동 테스트 생성 기법은 테스트 커버리지와 같은 코드 실행 피드백을 활용해 테스트를 점진적으로 보완할 수 있지만, TESTPILOT은 오직 기능적으로 정확한 테스트를 생성하는 데에만 집중하고 커버리지는 명시적으로 증진하지 않는다. 또한, TESTPILOT은 LLM에게 프로그램 분석 및 커버리지 피드백 같은 심층적인 정보를 제공하지 않는다. 그 결과, 중요한 코드 실행 경로를 놓칠 수 있으며 이는 버그 탐지 능력 저하로 이어질 가능성이 높다.

CoverUp [11] 및 TELPA [10] 등 LLM 기반 테스트 생성에도 커버리지 피드백이나 프로그램 분석을 도입하는 기법들도 제안되었으나, JavaScript 테스트를 대상으로 하는 기법은 부재하는 상황이다. 본 논문은

JavaScript 테스트 커버리지를 증진하는 테스트 생성 기법을 제안하고자 한다.

3. 제안 기법

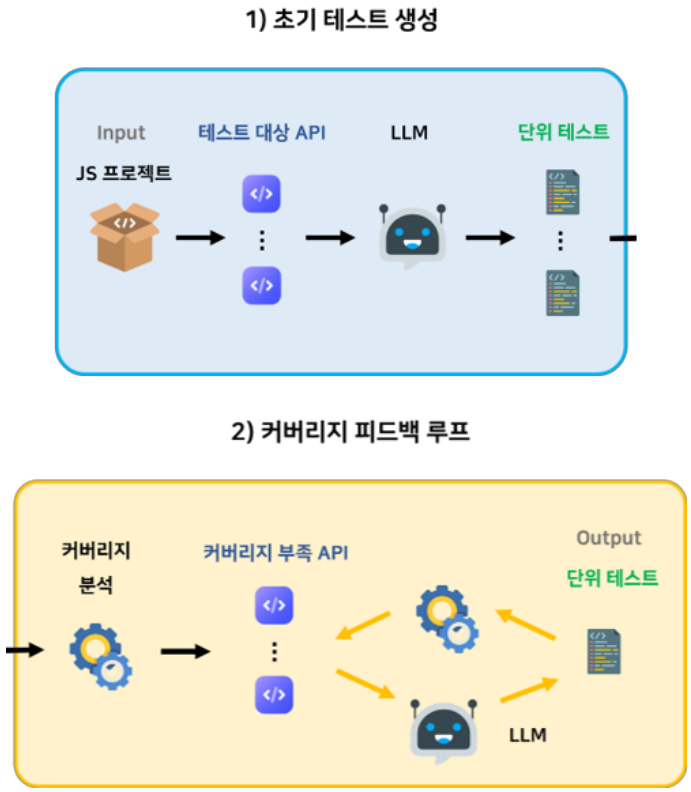


그림 2 COVERPILOT 워크플로우

본 연구에서 제안하는 커버리지 피드백을 통합한 LLM 기반 테스트 자동 생성 기법, COVERPILOT의 전체 워크플로우는 **그림 2**와 같다. COVERPILOT 기법은 (1) 초기 테스트 생성 단계와 (2) 커버리지 피드백 루프 단계로 구성된다. (1) 단계에서는 입력받은 JavaScript 프로젝트 내 각 테스트 대상 API에 대해 기능적으로 정확한 단위 테스트를 생성하고, (2) 단계에서는 테스트 실행 결과로부터 얻은 커버리지 피드백을 활용하여 테스트 커버리지를 점진적으로 개선한다.

3.1. 초기 테스트 생성 단계

초기 테스트 생성 단계의 목적은 각 테스트 대상 API에 대해 기능적으로 올바른 단위 테스트를 생성하는 것이다. LLM은 각 API의 소스 코드를 분석하여 해당 API에 적합한 단위 테스트를 생성한다.

입력으로 JavaScript 프로젝트가 주어지면, COVERPILOT은 Nessie에서 사용한 방식과 유사한 API 탐색 기법을 이용해 테스트 대상 API, 즉 public 함수 및 메서드 집합을 추출한다. 구체적으로, 프로젝트의 객체 그래프(object graph)를 분석하여 모든 public API를 식별한다. 각 API에 대해 시그니처, 소스 코드, 그리고 파일 위치 정보를 기록한다.

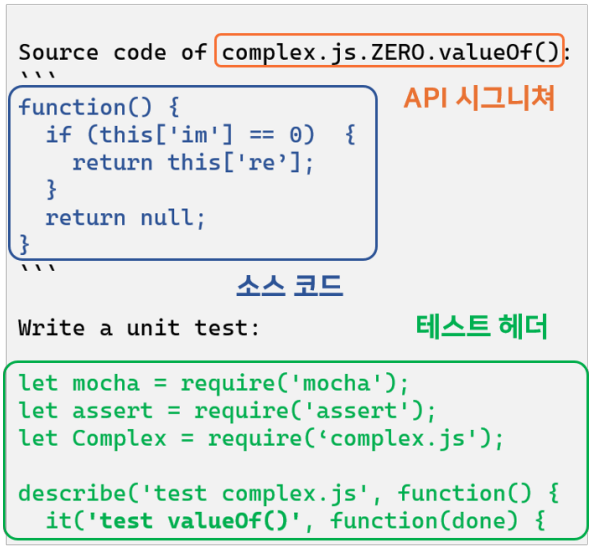


그림 3 초기 테스트 생성 프롬프트 예시

각 테스트 대상 API에 대해 LLM을 활용하여 기능적으로 정확한 단위 테스트를 생성한다. 이를 위해 **그림 3**과 같은 프롬프트를 구성하는데, 이 프롬프트는 API 시그니처, 소스 코드, 테스트 헤더로 이루어진다. LLM이 생성한 단위 테스트를 실행하여 기능적으로 정확한지 확인한다. 테스트 실행에서 오류가 발생할 경우 해당 테스트는 폐기되며, COVERPILOT은 최대 3회까지 LLM을 재호출하여 기능적으로 정확한 테스트를 마련한다.

3.2. 커버리지 피드백 루프 단계

```
Source code of complex.js.ZERO.valueOf():
```
function() {
  // this code block not covered
  /*
  if (this['im'] == 0) {
    return this['re'];
  }
  */
  return null;
}
```

Write a test for covering code block:

```
let mocha = require('mocha');
let assert = require('assert');
let Complex = require('complex.js');

describe('test complex.js', function() {
  it('test valueOf()', function(done) {
```
```

그림 4 커버리지 피드백 프롬프트 예시

커버리지 피드백 루프 단계는 이전 단계에서 생성된 초기 테스트를 기반으로, 동적 커버리지 피드백을 활용하여 테스트 커버리지를 개선하는 것을 목표로 한다.

먼저, 이전 단계에서 마련한 초기 테스트를 실행하여 커버리지를 측정한다. 이때, JavaScript 커버리지 분석 도구인 nyc/istanbul이 생성한 커버리지 리포트를 분석하여 초기 테스트에 의해 실행되지 않은 코드 블록을 식별한다. 이 분석을 통해 커버리지가 불완전한 테스트 대상 API들을 파악할 수 있다.

커버리지가 부족한 API 목록이 확보되면, 각 API에 대해 커버리지 피드백 루프를 수행하여 추가 테스트 케이스를 생성한다. 이때, **그림 4**와 같은 프롬프트를 구성한다. 프롬프트는 API 시그니처, 소스 코드, 테스트 헤더를 포함하며, 테스트에 의해 실행되지 않은 코드 블록을 API 소스 코드 내에서 주석화한다. 이러한 주석은 LLM에게 어떤 코드 영역이 아직 실행되지 않았는지를 직접적으로 드러낸다.

LLM이 생성한 테스트를 실행하여 커버리지 향상 여부를 확인한다. 커버리지가 여전히 충분하지 않다면, 해당 API에 대해 동일한 과정을 반복한다. COVERPILOT은 각 API에 대해 최대 5회까지 LLM을 호출하여 커버리지 최적화를 시도한다. 이와 같이 동적인 실행 피드백을 반복적으로 활용함을 통해 테스트 커버리지를 점진적으로 개선한다.

4. 실험 설계

COVERPILOT의 테스트 생성 성능을 평가하기 위한 실험을 설계한다. 실험의 주요 목적은 동적 커버리지 피드백을 통합한 LLM 기반 테스트 자동 생성이 LLM 기반 단일 테스트 생성 기법 및 기존 자동 테스트 생성 기법에 비해 테스트 커버리지 측면에서 얼마나 효과적인지를 검증하는 것이다.

4.1. 벤치마크

Project	Domain	# Lines	# API
bluebird	async	3.1K	115
q	async	736	98
pull-stream	streams	308	24
complex.js	arithmetic	393	52
memfs	file system	2.2K	376
Total		**6.7K**	**665**

표 1 실험 벤치마크

COVERPILOT의 성능을 평가하기 위해 5개의 오픈소스 JavaScript 프로젝트를 벤치마크로 사용하였다. 사용된 프로젝트는 bluebird, q, pull-stream, complex.js, memfs로, 이들 프로젝트에는 총 665개의 테스트 대상 API가 포함되어 있다.

각 프로젝트에 대한 코드 라인 수와 API 개수는 **표 1**에 기재되어 있다. 각 프로젝트에 대한 테스트 생성 시간은 동일하게 1시간으로 제한했다.

4.2. 실험 설계

COVERPILOT의 성능을 평가하기 위해 두 가지 비교 대상을 설정하였다.

첫째, Nessie[1]는 JavaScript API를 대상으로 하는 자동 테스트 생성 도구로, 피드백 기반 테스트 생성 기법이다. Nessie는 프로그램 실행 결과로부터 얻은 피드백을 활용하여 테스트를 반복적으로 생성한다. 본 논문에서는 기존 TESTPILOT 논문에서 보고된 성능 수치를 비교 기준으로 사용하였다.

둘째, TESTPILOT[3]은 LLM 기반 단일 테스트 생성 기법으로, API 시그니처, 기술 문서, 사용 예시, 소스 코드 등을 결합한 프롬프트를 사용해 단위 테스트를 생성한다. TESTPILOT은 프로그램 분석이나 커버리지 피드백 등 심화 정보는 응용하지 않는다. 본 연구에서는 TESTPILOT을 COVERPILOT과 동일한 조건에서 직접 실행하여 성능을 측정하였다.

두 비교 대상은 각각 전통적인 자동 테스트 생성 기법과 기존 LLM 기반 단일 테스트 생성 기법을 대표한다. 이들과 비교 실험을 통해 COVERPILOT의 상대적 성능을 종합적으로 평가할 수 있다. COVERPILOT과 TESTPILOT 모두 기반 LLM으로서 StarCoder-15B 모델[2]을 사용하였으며, 단일 Nvidia RTX A6000 (48GB) GPU 환경에서 실행하였다.

5. 실험 결과

Project	Nessie		TESTPILOT		COVERPILOT	
	stmt	**branch**	**stmt**	**branch**	**stmt**	**branch**
bluebird	43.8%	24.6%	40.06%	23.42%	**61.43%**	**44.75%**
q	66.8%	54.4%	51.07%	42.80%	**70.02%**	**55.43%**
pull-stream	38.5%	23.8%	81.63%	64.95%	**82.50%**	**66.35%**
complex.js	8.6%	5.4%	39.59%	40.98%	**64.72%**	**49.50%**
memfs	**64.6%**	**36.2%**	48.11%	27.08%	55.11%	33.57%
Average	44.06%	28.88%	52.09%	39.85%	**66.76%**	**49.92%**

표 2 실험 결과

표 2는 Nessie, TESTPILOT, 그리고 COVERPILOT이 5개의 오픈소스 JavaScript 프로젝트에 대해서 달성한 문장(statement) 커버리지와 분기(branch) 커버리지를 제시한다. 이 결과는 동적 커버리지 피드백을 LLM 기반 테스트 생성에 통합하는 접근이 효과적임을 입증한다.

COVERPILOT은 평균 커버리지 측면에서 두 비교 기법을 모두 능가하는 성능을 보였다. 평균 문장

커버리지는 66.76%, 평균 분기 커버리지는 49.92%로, TESTPILOT 대비 각각 +14.67%와 +10.07% 향상되었다. Nessie에 대해서는 문장 커버리지 +22.7%, 분기 커버리지 +21.04% 향상을 기록하였다. 이는 실행 피드백 기반 커버리지 정보를 적극적으로 활용하는 접근이 테스트 품질을 크게 개선할 수 있음을 보여준다.

COVERPILOT은 다섯 개의 벤치마크 프로젝트 중 네 개에서 두 비교 기법보다 우수한 성능을 보였다. 특히, bluebird에 대해서는 TESTPILOT 대비 문장 커버리지 +21.37%, 분기 커버리지 +20.33%의 큰 향상을 달성하였다. 산술 연산을 포함하는 complex.js에서도 문장 커버리지 +25.13%, 분기 커버리지 +8.52%의 유의미한 개선이 관찰되는데, 이는 조건 분기와 엣지 케이스가 중요한 산술 연산 코드에 대해서는 COVERPILOT이 강점을 지님을 보여준다. memfs 프로젝트의 경우에는 Nessie가 COVERPILOT보다 다소 높은 커버리지를 달성하였다. 이는 Nessie가 1시간 시간 제한에 제약 받지 않았기 때문일 가능성이 크다.

COVERPILOT이 기존 기법 대비 우수한 성능을 보인 핵심 요인은 동적 커버리지 피드백의 활용이다. TESTPILOT이 정적인 프롬프트와 LLM 단일 생성에만 의존하는 반면, COVERPILOT은 테스트 실행 결과를 바탕으로 커버리지가 부족한 코드 영역을 식별하고 이를 기반으로 테스트를 점진적으로 보완한다. 이러한 커버리지 피드백 루프는 LLM의 코드 생성 과정을 동적으로 보정하여 보다 포괄적인 테스트 커버리지를 달성을 가능하게 한다.

6. 결론

본 논문에서는 동적 커버리지 피드백을 통합하여 테스트 커버리지를 점진적으로 개선하는 새로운 LLM 기반 단위 테스트 자동 생성 기법인 COVERPILOT을 제안한다. 기존 LLM 기반 접근법인 TESTPILOT과 같은 기법들은 기능적으로 정확한 테스트를 생성하는 데에는 성공적이지만, 잠재적인 버그를 발견하는 데 필수적인 요소인 테스트 커버리지를 최적화하는 데에는 한계를 지닌다. COVERPILOT은 커버리지 피드백 루프를 도입함으로써, 실행되지 않은 코드 블록을 체계적으로 식별하고 보완하기 위한 추가적인 테스트를 생성할 수 있다. 이를 통해 테스트 커버리지를 점진적으로 향상시킬 수 있다. COVERPILOT은 5개의 JavaScript 오픈소스 프로젝트에 대해서 기존의 자동 테스트 생성 도구인 Nessie와 LLM 기반 단일 테스트 생성 기법인 TESTPILOT에 비해 문장 커버리지와 분기 커버리지 측면에서 유의미한 성능 향상을 달성하였다. 이러한 결과는 LLM의 생성 능력과 기존 소프트웨어 테스팅 기법을 융합한 접근법이 효과적임을 시사한다.

사사

본 연구성과는 정보통신기획평가원 (IITP)와 한국연구재단의 지원을 받아 수행되었음. (과제번호: RS-2023-00222830, RS-2025-24913002)

참고문헌

[1] E. Arteca, S. Harner, M. Pradel, and F. Tip, “Nessie: Automatically testing JavaScript APIs with asynchronous callbacks,” in Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, May 2022, pp. 1494–1505. doi: 10.1145/3510003.3510106.

[2] R. Li et al., “StarCoder: May the source be with you!” Transactions on Machine Learning Research, vol. 2023, 2023.

[3] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation,” IEEE Transactions on Software Engineering, vol. 50, pp. 85–105, 2023.

[4] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” Communications of the ACM, vol. 33, no. 12, pp. 32–44, 1990. doi: 10.1145/96267.96279.

[5] M. Zalewski, “American fuzzy lop,” 2022. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[6] M. Selakovic, M. Pradel, R. Karim, and F. Tip, “Test generation for higher-order functions in dynamic languages,” Proceedings of the ACM on Programming Languages, vol. 2, no. OOPSLA, pp. 161:1–161:27, 2018. doi: 10.1145/3276531.

[7] G. Fraser and A. Arcuri, “Evolutionary generation of whole test suites,” in Proceedings of the 11th International Conference on Quality Software (QSIC), Madrid, Spain, Jul. 2011, pp. 31–40. doi: 10.1109/QSIC.2011.19.

[8] G. Fraser and A. Arcuri, “EvoSuite: Automatic test suite generation for object-oriented software,” in Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE/ESEC), Szeged, Hungary, Sep. 2011, pp. 416–419.

[9] J. Altmayer Pizzorno and E. Berger, “CoverUp: Effective high-coverage test generation for Python,” Proceedings of the ACM on Software Engineering, vol. 2, 2024, pp. 2897–2919.

[10] C. Yang, J. Chen, B. Lin, Z. Wang, and J. Zhou, “Advancing code coverage: Incorporating program analysis with large language models,” ACM Transactions on Software Engineering and Methodology, 2024.

치공구 설계를 위한 RAG-MCP 기반 멀티 에이전트 FreeCAD 오토코딩 시스템

이의천 [1] 고성진 [2] 이선아 [3] 이석원 [4]
경상국립대학교 AI 융합공학과 [1,2,3], 경상국립대학교 소프트웨어공학과 [3], (주)씨엘디 [4]
rndnjswk123@naver.com[1], tjdwls9267@naver.com[2],
saleese@gnu.ac.kr[3], sukwonlee@cldaero.com[4]

RAG-MCP Based Multi-Agent FreeCAD Auto-Coding System for Jig and Fixture Design

Uicheon Lee[1], Seongjin Go[2], Seonah Lee[3], Sukwon Lee[4]
Department of AI Convergence Engineering, Gyeongsang National University [1,2,3]
Department of Software Engineering, Gyeongsang National University [3]
CLD Co., Ltd. [4]

요 약

항공기 생산과 유지정비 수요는 지속적으로 확대되는 반면, 생산가능인구 감소로 인해 업무 자동화가 필수 과제로 전환되고 있다. 치공구 설계는 이러한 자동화 요구가 집중되는 대표적 영역으로, 단일 형상을 한 번 생성하는 문제가 아니라 부속 단위의 반복 생성과 배치, CAD(Computer-Aided Design) 환경에서의 실행-검증-복구, 설계 자산의 재사용과 갱신 등 운영을 포괄하는 자동화 문제이다. 본 연구는 치공구를 부속 단위로 분해하고, RAG(Retrieval-Augmented Generation) 기반 코드 자산 재사용과 MCP(Model Context Protocol) 기반 도구 계층화를 결합한 멀티 에이전트 FreeCAD 오토코딩 시스템을 제안한다. 5개의 L자 형상 가공대상에 대한 평가 결과, 서로 다른 크기의 가공대상에 대해 평균 1분 48초에서 2분 1초 범위 내에서 전체 치공구를 자동 생성하였으며, 잠재 사용자 평가를 통해 현장 적용 가능성을 확인하였다. 본 연구는 소프트웨어공학의 운영화 철학이 AI 기반 제조 설계 자동화 영역으로 확산될 수 있음을 보여준다.

1. 서 론

항공기 생산과 유지정비 수요는 지속적으로 확대되는 반면[1-3], 생산가능인구는 감소하고 있어[4] 설계 자동화 역량 확보가 필수 과제로 부상하고 있다. 특히, 치공구 설계는 이러한 자동화 요구가 집중되는 대표적 영역이다. 치공구는 각 항공기 부품을 가공할 때 가공대상(Workpiece)의 위치 결정과 고정에 사용되는 공구로서, 그림 1과 같이 베이스 플레이트 (Baseplate), 인덱스플레이트 (Indexplate), 클램프 (Clamp), 가이드 브라켓 (Guide Bracket) 등 여러 부속의 조합으로 구성된다. 치공구의 설계 및 제작은 항공기 제조 비용의 10-20%를 차지하고[5] 평균 2-4개월이 소요되어[6] 주요 병목이 된다. 그러나 3차원 공간 추론이 요구되는 고난도 작업이자[7] 설계자 경험 의존성이 높아[8], 자동화가 어려운 영역으로 남아 있다.

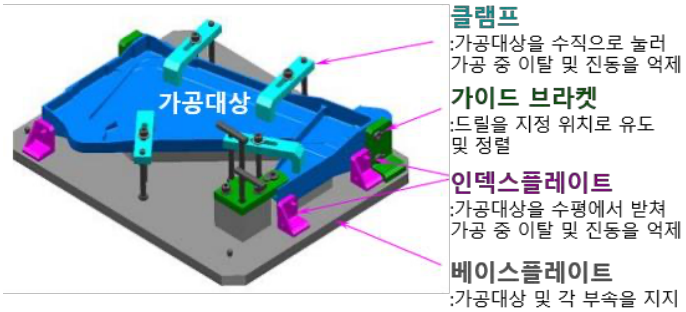


그림1. 가공대상과 치공구의 각 구성요소

이에 대해, 최근 자연어 요구를 코드로 변환하는 LLM(Large Language Model) 기반 오토코딩(auto-coding)이 유망한 접근법으로 주목받고 있다. 하지만 이를 치공구 설계에 직접 적용하기에는 난관이 존재한다. 단순한 L자 형상의 가공대상에 대한 치공구조차 CAD 플랫폼인 FreeCAD의 Python API 기준 약 4,000라인의 코드가 필요하다. LLM이 이를 매번 단일 형상으로 간주해 전부 생성할 경우, 환각 (hallucination), 비용, 지연, 컨텍스트(Context) 망각이 복합적으로 작용하여 품질이 저하된다. 이에 더해, 치공구 설계는 단일 형상 생성에 그치지 않는다. CAD 환경에서 부속 단위의 반복 생성과 배치, 사용자의 피드백에 따른 수정, 설계 자산의 재사용과 갱신이 지속적으로 요구된다. 본 연구는 이러한 특성에 주목하여, 치공구 자동설계를 완성된 단일 형상 생성이 아닌 지속적 설계, 구현, 검증, 복구의 순환을 포괄하는 운영형(Operations, Ops) 자동화 문제로 정의한다.

본 논문은 이 문제를 해결하기 위해 RAG-MCP 기반 멀티 에이전트 FreeCAD 오토코딩 시스템을 제안한다. 제안 시스템은 반복되는 수정 요구에 대응하기 위해, 첫째, 치공구를 4개의 하위 부속으로 분해 후 개별적으로 관리한다. 둘째, LLM의 컨텍스트 망각에 대응하기 위해, 각 부속의 설계를 전담하는 에이전트와 전체 워크플로우를 조율하는 관리 에이전트를 분리한다. 셋째, LLM의 환각, 비용, 지연 문제에 대응하기 위해, RAG (Retrieval-Augmented Generation)[9] 기법과 MCP(Model Context Protocol)[10] 기반 도구를 적용한다. RAG는 사전 검증된 코드 자산을 검색 및 재사용하여 LLM이 직접 생성해야 하는 코드량을 최소화하고, MCP 기반 도구는 미리 의도된 규칙 하에서 부속 설계를 진행할 수 있게 한다. 이러한 설계는 DevOps, MLOps 등 소프트웨어공학에서 발전한 운영화 철학과 맥을 같이한다.

제안 시스템의 평가는 실용적 타당성에 초점을 둔다. 5개의 서로 다른 크기의 가공대상에 대해 로컬 개발 환경에서 설계 소요 시간, 실패 유형 및 복구 양상을 측정한다. 또한 항공기 치공구 설계 실무 경력자의 평가를 통해 생성된 치공구의 설계 품질과 현장 적용 가능성을 검토한다.

본 연구의 기여는 다음과 같다. 첫째, 치공구 자동설계에 멀티 에이전트 기반 오토코딩을 적용한 첫 사례로서, 부속 단위 분해와 RAG 기반 코드 자산 재사용, MCP 기반 도구 호출 계층 분리를 결합한 설계 자동화 시스템을 제안하였다. 둘째, FreeCAD 환경에서 실제 실행 가능한 시스템을 구현하고, 반복 실험과 실무자 평가를 통해 실용적 타당성을 확인하였다. 셋째, 소프트웨어공학의 모듈화, 재사용성, 운영화 원칙이 AI 기반 제조 설계 자동화 영역으로 확산될 수 있음을 실증적으로 보였다.

논문의 구성은 다음과 같다. 2장에서는 관련 연구를 정리한다. 3장에서는 제안 시스템을 설명한다. 4장에서는 평가 방법을 기술한다. 5장에서는 평가 결과를 제시한다. 6장에서는 논의를, 7장에서 결론을 제시한다.

2. 관련연구

치공구 자동설계처럼 3차원 형상을 반복적으로 생성해야 하는 작업에서, 최근 연구는 다음 네 흐름으로 정리할 수 있다. 첫째, CAD를 명령 시퀀스로 표현하고 이를 학습하여 생성 및 재구성하는 생성모델 계열이다(2.1절). 둘째, LLM을 통해 사용자의 자연어 요구를 CAD 전용 코드 또는 절차로 변환하는 오토코딩 계열이다(2.2절). 셋째, CAD 환경에서 생성 이후의 실행 가능성과 신뢰성을 높이는 계열이다(2.3절). 넷째, 설계-제조 파이프라인 전반에서 멀티 에이전트 협업과 지식 재사용을 통해 운영 가능한 설계 자동화를 지향하는 계열이다(2.4절). 본 절에서는 각 흐름의 핵심 아이디어를 정리 및 비교한다.

2.1 CAD 시퀀스 기반 생성 및 재구성

CAD 자동설계에 대한 대표적인 연구는 CAD를 파라미터로 구성된 명령 시퀀스 또는 프로그램으로 정의한 연구들이다. Seff et al.[11]은 파라메트릭 스케치 생성을 통해 CAD 표현 학습의 기반을 제시하였고, Wu et al.[12]은 명령 시퀀스를 자동회귀 방식으로 생성하는 심층 생성모델을 제안하였다. Alam et al.[13]은 이미지 조건부 생성으로 입력을 확장했고, Kolodiazhnyi et al.[14] 및 Nakayama[15]는 강화학습 기반의 CAD 프로그램 생성 가능성을 탐색했다.

이 흐름은 CAD 형상을 학습 가능한 시퀀스로 표현하는 방법론을 확립했다는 점에서 의의가 있다. 그러나 도메인 규칙의 반영, 신규 사례의 저비용 갱신, 실패의 국소 복구 등 산업 적용에 필요한 운영형 요구를 체계적으로 다루지는 않는다.

2.2 LLM 기반 CAD 오토코딩

LLM의 확산과 더불어 자연어 요청을 CAD 산출물로 변환하려는 연구들이 증가했다. 텍스트-코드 변환 측면에서, Wu et al.[16]은 CAD 생성 문제를 언어 모델과 결합해 다루었고, Khan et al.[17]은 텍스트 프롬프트 기반 순차 CAD 생성 모델을, Li et al.[18]은 CAD 전용 언어 모델을, Xie and Ju[19]는 특정 CAD 언어 대상 직접 변환 방법을 각각 제안하였다. 입력 다양화 측면에서, Xu et al.[20], Yuan et al.[21], Wu et al.[22]은 이미지, 포인트 클라우드, 스케치 등 다양한 입력 조건에서 편집 가능한 절차를 산출하는 접근을 제안했고, Li et al.[23]은 멀티모달 LLM의 3차원 CAD 생성 능력을 평가하였다.

다만 이 흐름의 다수 연구는 치공구처럼 부속 조합과 배치가 핵심인 문제에서 필요한, 부속 단위의 단계 분해와 책임 분리, 생성 결과의 실행 가능성 검증, 실패 시 국소 복구, 운영 중 지식의 재사용 및 갱신을 하나의 시스템 구조로 통합하는 측면에서는 제한적이다.

2.3 생성 후 검증 및 반복 개선

산업 환경에서 생성 결과를 활용하려면 생성 이후의 검증과 수정이 필수적이다. 시각적 피드백 기반 반복 개선 측면에서, Badagabettu et al.[24]은 FreeCAD 환경에서 자연어 기반 모델 생성과 시각적 피드백을 결합한 반복 개선을, Alrashedy et al.[25]과 Wang et al.[26]은 시각적 검증 루프를 통한 정확도 향상을 제안하였다. 도구 호출 및 편집 측면에서, Mallis et al.[27]은 CAD 환경과 도구 호출 기반의 다양한 CAD 작업 수행을, Yuan et al.[28]은 텍스트 기반 CAD 편집 기능을, Chen et al.[29]은 CAD 프로그램의 오류 탐지와 자동 수정을 각각 제시하였다.

그러나 이 계열에서도 운영 관점의 핵심 요구가 충분히 구조화되는 것은 아니다. 특히 부속 단위의 단계 분해가 시스템 구조로 강제되는 사례는 제한적이기에, 적용 범위가 확대될수록 과정의 확장과 유지보수 비용이 증가할 수 있다.

2.4 멀티 에이전트 기반 설계 운영화와 지식 재사용

설계-제조 파이프라인 전반에서 멀티 에이전트 기반의 LLM 활용을 논의하는 연구 또한 증가하고 있다. Makatura et al.[30, 31]은 설계 표현부터 제조 준비까지 파이프라인 관점에서 가능성과 한계를 분석했고, Lim et al.[32]은 제조 시스템에서

역할 분해된 멀티 에이전트 협업 구조를 논의하였다. Ocker et al.[33], Liao et al.[34]은 아이디어에서 CAD까지 협업하는 멀티 에이전트 설계를 제시하며 역할 분해의 필요성을 강조했고, Deng et al.[35]은 산업용 CAD 자동화 적용 가능성을 조사하며 실제 적용 요구를 드러냈다.

이 흐름은 협업과 운영 구조의 중요성을 보여주지만, 치공구처럼 도메인 규칙이 강하고 반복 설계가 많은 문제에서 코드 자산 재사용 및 갱신과 도구 호출 계층 분리를 결합해 운영 가능한 형태로 구현한 사례는 여전히 제한적이다.

2.5 본 연구의 차별점

표 1은 앞선 네 범주를 운영 가능한 오토코딩 설계 시스템 관점에서 요약 비교한 것이다. 기존 연구는 형상 생성 성능, 입력 다양성, 생성 후 검증과 편집, 멀티 에이전트 협업을 각각 발전시켜 왔으나, 이들을 산업 적용 관점에서 하나의 통합된 구조로 구현한 사례는 제한적이다. 특히, 치공구 도메인에서는 멀티 에이전트 구조 위에 이들을 통합한 오토코딩 선행연구는 확인되지 않았다. 본 연구는 이러한 연구 격차를 해소하기 위해 치공구 문제를 운영형 자동화 문제로 정의 후, 부속 단위 독립, 멀티 에이전트 및 RAG-MCP 도구를 활용한 오토코딩 설계 시스템을 구현한다.

표1. 운영 가능한 오토코딩 설계 시스템 관점에서의 비교

연구 범주	코드 자산 활용·갱신	부속 단위 분해	역할 기반 제어	도구 호출 계층 표준	CAD 환경 직접 실행	생성 후 검증·편집
2.1 CAD 시퀀스	X	X	X	X	△	X
2.2 LLM	△	△	X	△	△	△
2.3 생성 후 검증	△	△	△	△	O	O
2.4 멀티 에이전트	△	O	O	△	△	△
제안 시스템	**O**	**O**	**O**	**O**	**O**	**△**

※ X: 해당 없음, △: 일부 해당, O: 해당

3. 제안 시스템

본 장에서는 치공구 자동설계를 생성-실행-검증-복구의 순환으로 운영 가능하게 만드는 시스템 구조를 제안한다. 제안 시스템은 RAG 기반 코드 자산 재사용 및 갱신, MCP 기반 도구 호출 계층화, 역할 기반 멀티 에이전트 분해를 결합하여, LLM이 FreeCAD 파이썬 코드를 대규모로 단발 생성할 때 발생하는 환각, 비용, 지연, 컨텍스트 한계를 구조적으로 완화하도록 설계하였다. 전체 구조는 그림 2와 같으며, 이후 절에서 각 계층과 설계 결정을 요약한다.

3.1 설계 목표

치공구 설계는 단일 형상을 한 번 생성하는 문제가 아니라, 부속 단위의 반복 생성과 배치, CAD 환경에서의 실행 가능성 검증, 오류 발생 시 국소 복구, 그리고 사례 축적을 통한 지식 자산의 재사용 및 갱신을 요구한다. 본 연구는 LLM의 생성 능력 자체보다, LLM이 다뤄야 하는 코드/컨텍스트 범위를 구조적으로 축소하면서도 실행 가능한 산출물을 안정적으로 확보하는 것을 목표로 한다. 표 2는 이를 위한 설계 목표와 구현 전략을 정리한다.

표2. 제안 시스템의 설계 목표

구분	설계 목표	구현 전략
G1	LLM 이 직접 생성/편집해야 하는 코드량 최소화	RAG 로 사전 검증된 코드 자산 검색 및 재사용
G2	부속 단위 분해로 재사용 및 실패 격리	부속별 전용 에이전트로 설계 단계 및 책임 분리
G3	역할 기반 제어로 의도치 않은 변경 방지	에이전트별 도구 권한 제한 및 시스템 프롬프트 강제
G4	도구 호출 계층을 분리하여 재현성, 확장성, 오류 국소화	MCP 기반 표준화된 도구 호출 인터페이스
G5	CAD 환경 직접 실행으로 실행 가능성 검증	FreeCAD 프로세스 분리 및 XML-RPC 통신
G6	코드 자산의 지속적 갱신 가능성 확보	샘플 코드 및 메타데이터 DB 축적, 샘플을 통한 확장

3.2 시스템 구성

제안 시스템의 구조는 그림 2와 같다. 시스템은 사용자 인터페이스 계층, 에이전트 계층, MCP 계층, RAG 계층, FreeCAD 계층의 5개 계층으로 구성된다.

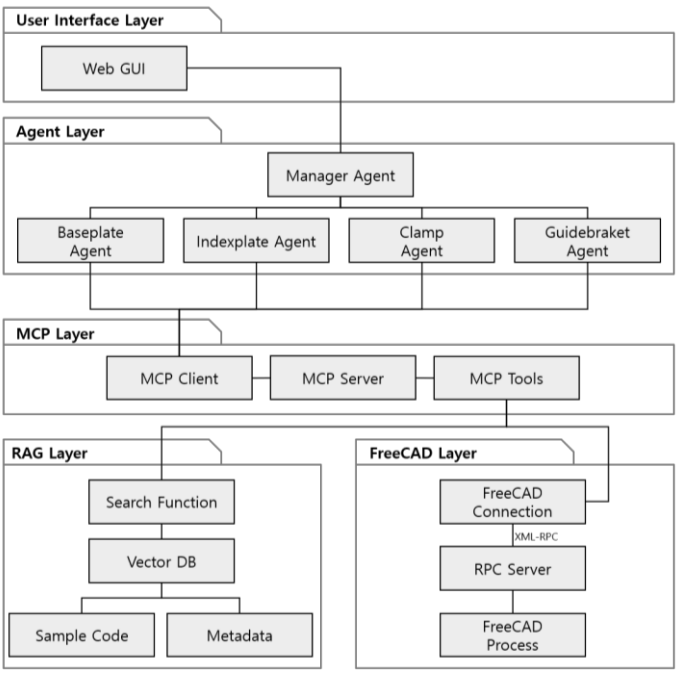


그림2. 제안 시스템의 아키텍처

사용자 인터페이스 계층은 설계 입력 수집과 결과 시각화를 담당한다. 에이전트 계층은 치공구를 부속 단위로 분해하고 각 부속의 생성을 전담 에이전트에 할당하여 역할 기반 제어를 실현한다. MCP 계층은 에이전트와 외부 도구 간 호출을 표준화하여 재현성과 오류 국소화를 지원한다. RAG 계층은 사전 검증된 코드 자산을 검색 및 재사용함으로써 LLM이 직접 생성해야 하는 코드량을 최소화한다. FreeCAD 계층은 생성된 코드를 실제 CAD 환경에서 실행하여 실행 가능성을 검증한다.

시스템의 전체 흐름은 (1) 설계 입력 수신, (2) 부속 분해 및 작업 할당, (3) RAG 기반 코드 검색, (4) MCP 도구 호출을 통한 실행 및 수정, (5) 결과 검증 및 보고의 단계로 구성된다.

3.3 사용자 인터페이스 계층

사용자 인터페이스 계층은 웹 기반 GUI(Graphical User Interface)를 통해 설계 입력 수집과 결과 시각화를 담당한다. 그림 3은 사용자 인터페이스의 전체 구성을 보인다.

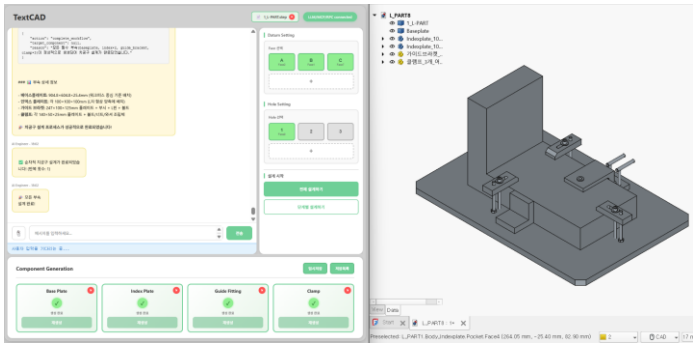


그림 3. 사용자 인터페이스

인터페이스는 채팅, 설정, 상태 표시, 3D 뷰의 영역으로 구성된다. 좌측 채팅 영역에서 사용자는 자연어로 설계 요청을 입력하고 LLM과 상호작용한다. 우측 상단 영역에서는 업로드한 가공대상에 대한 STEP(Standard for the Exchange of Product model data) 3D 파일에 대해 기준면(Datum)과 드릴 가공 구멍(Hole)을 지정하며, 전체 설계 또는 단계별 설계 패턴을 선택한다. 상단 표시줄은 현재 로드된 파일명과 함께 LLM, MCP 서버, RPC 서버의 연결 상태를 실시간으로 제공하고, 우측의 FreeCAD 3D 뷰어는 가공대상과 생성된 치공구 부속을 시각적으로 확인하도록 지원한다.

하단의 부속 관리 영역은 베이스플레이트, 인덱스플레이트, 클램프, 가이드 브라켓의 생성 진행 상태를 표시하고, 부속별 재생성 기능을 통해 특정 부속만 선택적으로 재수행할 수 있게 한다. 또한 임시저장 및 이전 상태 호출 기능으로 여러 설계 대안을 비교 및 선택할 수 있으며, 임시저장 시에는 3D 형상을 캡처해 썸네일과 함께 상태를 기록한다. 제안 시스템은 전체 설계 워크플로우와 단계별 설계 워크플로우를 제공하며, 전자는 4개 부속을 순차적으로 자동 생성해 완전 자동화를 수행하고, 후자는 사용자가 선택한 부속만 생성하여 빠른 상호작용과 반복 개선을 지원한다.

3.4 에이전트 계층

에이전트 계층은 역할 기반 멀티 에이전트 아키텍처를 채택한다. 본 연구에서 에이전트는 목표, 설계 원칙, 사용 가능 도구, 컨텍스트를 바탕으로, 작업 순서에 따라 자율적으로 판단하고 도구를 호출하는 LLM 기반 실행 단위를 의미한다. 제안 시스템은 치공구를 4개 부속으로 분해하고, 각 부속의 생성을 전담하는 부속 에이전트와 전체 프로세스를 조율하는 관리 에이전트를 분리하여 역할 기반 제어를 실현한다.

이 구조는 부속별 책임을 명확히 하여 응집도를 높이고, 오류를 부속 단위로 격리해 재시도 및 복구 비용을 줄인다. 또한 관리 에이전트는 세션 전역 히스토리를 유지해 문서 상태를 종합 판단하고, 부속 에이전트는 담당 부속에 필요한 최소 컨텍스트만 유지한 뒤 초기화함으로써 컨텍스트 팽창을 억제한다. 표 3은 각 에이전트의 역할 및 컨텍스트 범위와 시스템 프롬프트의 목표와 핵심 원칙을 통합해 요약한다.

표3. 에이전트 역할 및 컨텍스트 관리

에이전트	목표 및 핵심 원칙	역할 및 컨텍스트
Manager Agent	**목표**: 설계 프로세스 관리 및 에이전트 조율 **원칙**: 문서 분석 후 작업 결정, 부속 순서 강제	**역할**: 문서 상태 분석, 부속 정보 추출, 생성 순서 결정, 에이전트 호출 **컨텍스트**: 전역 유지
Baseplate Agent	**목표**: 모든 부속을 지지하는 기반 생성 **원칙**: 가공대상과 XY 중심 정렬, 상단면-가공	**역할**: 베이스플레이트 설계·생성 후 결과 요약 **컨텍스트**: 수행 후 초기화
Index Agent	**목표**: 가공대상 XY 평면 수평 이동 방지 **원칙**: L 자 형상 2 개 배치, 가공대상 옆면 인접	**역할**: 인덱스플레이트 설계·생성 후 결과 요약 **컨텍스트**: 수행 후 초기화
Clamp Agent	**목표**: 가공대상 수직 고정 **원칙**: 3 개 배치, 가공대상을 아래로 압착	**역할**: 클램프 설계·생성 후 결과 요약 **컨텍스트**: 수행 후 초기화
Guide Bracket Agent	**목표**: 드릴 가이드 역할 **원칙**: 가이드 홀-가공대상 홀 일직선 정렬	**역할**: 가이드 브라켓 설계·생성 후 결과 요약 **컨텍스트**: 수행 후 초기화

3.5 MCP 계층

MCP 계층은 에이전트와 외부 도구 간 호출을 표준화하여 재현성과 오류 국소화를 지원한다. 제안 시스템은 FreeCAD 실행을 LLM이 생성한 장문 코드에 직접 의존하지 않고, MCP 기반 도구 호출로 캡슐화한다. 에이전트는 수행할 작업을 결정해 도구를 호출하며, 실제 CAD 실행은 MCP 서버를 통해 FreeCAD 통합 계층에서 수행된다.

MCP 계층은 클라이언트-서버 구조로 구현된다. MCP 클라이언트는 웹 GUI와 함께 실행되며 MCP 서버를 백그라운드로 구동하고 FreeCAD 프로세스를 자동 실행한다. MCP 서버는 FastMCP 기반이며 FreeCAD와는 XML-

RPC(Extensible Markup Language Remote Procedure Call)로 통신한다. 이와 같은 프로세스 분리는 FreeCAD 충돌을 시스템 전체로부터 격리하고 구성요소의 독립적 재시작을 가능하게 한다.

MCP 도구는 정보 조회, 배치 조정, RAG 기반 부속 생성, 파일 처리로 구성되며, 표 4는 각 도구 범주와 역할 기반 사용 정책을 요약한다. 관리 에이전트는 문서 분석과 STEP 반입에 필요한 도구만, 부속 에이전트는 담당 부속의 생성·배치에 필요한 최소 도구만 사용하도록 제한하여 의도치 않은 변경을 방지하고 오류 발생 시 책임 범위를 명확히 한다. 또한 각 도구는 FreeCAD에서의 오류에 대한 로그 반환이 내장되어 있어, 도구 사용에 문제가 있을 경우 LLM이 직접 API 코드를 반환하여 이에 대응한다.

표 4. MCP 도구 범주와 역할 기반 사용 정책

도구 범주	대표 도구	활용 대상	목적
정보 조회	get_active_document, get_objects_info_only	전 에이전트	상태 관측, 치수 추출
배치 조정	set_body_placement	부속 에이전트	위치 및 회전 설정
RAG 사용	create_baseplate_from_rag 등 4종	부속 에이전트	샘플 코드 기반 생성
파일 처리	import_step_file	관리 에이전트	가공대상 STEP 반입

3.6 RAG 계층

RAG 계층은 사전 검증된 코드 자산을 검색 및 재사용함으로써 LLM이 직접 생성해야 하는 코드량을 최소화한다. 본 연구는 반복 가능한 설계 지식을 실행 가능한 코드 자산으로 데이터베이스에 저장하고, 검색과 재사용을 통해 산출물을 생성하는 방식을 채택하였다. RAG 데이터베이스는 ChromaDB 기반 벡터 저장소를 기반으로, 부속 유형별 샘플 코드와 메타데이터로 구성된다. 샘플 코드는 FreeCAD 문서 컨텍스트에서 실행 가능한 파이썬 코드이며, 메타데이터는 샘플의 적용 조건과 설계 의도를 표현한다. 표 5는 메타데이터의 구성을 요약한다.

표 5. RAG 코드 자산 메타데이터 구성

구분	항목	설명	활용 목적
공통	X, Y, Z 길이	가공대상 바운딩 박스 치수(mm)	유사 샘플 탐색
공통	부피	가공대상 규모 스칼라 값(mm^3)	유사 샘플 탐색
부속 특화	기준면 라벨	설계 기준이 되는 면 식별자	후보 필터링
부속 특화	대상 부속 유형	베이스플레이트, 인덱스플레이트, 클램프, 가이드 브라켓	후보 집합 분리

RAG 검색 과정은 후보 필터링과 최적 샘플 선택의 두 단계로 구성된다. 먼저 부속 유형과 기준면 라벨 등 필수 조건으로 후보를 필터링하고, 가공대상의 X, Y, Z 길이와 부피에 대해 정규화를 수행한다. 길이는 각 차원의 최댓값 기준으로 수행하며, 부피는 스케일 차이가 크므로 로그 변환 후 정규화한다. 이후 정규화 값과 부피의 로그 스케일 값에 대해 유클리드 거리를 계산하여 최단 거리 샘플을 선택한다.

검색된 샘플 코드는 변수 치환을 통해 현재 설계 조건에 적응된다. 대상 객체명, 기준면 식별자, 오프셋 값 등의 변수가 현재 가공대상 정보로 적용되어 실행된다. 코드 자산의 갱신은 샘플 코드와 메타데이터 추가를 통해 수행된다. 새로운 설계 사례가 검증되면 데이터베이스에 추가하고, 이후 동일 부속 유형과 유사 치수 조건에서 재사용이 가능해진다. 이 방식은 재학습 없이 지식 자산을 확장할 수 있어 산업 적용에서 요구되는 지속 가능성을 제공한다.

3.7 FreeCAD 계층

FreeCAD 통합 계층은 생성된 코드를 실제 CAD 환경에서 실행하여 실행 가능성을 검증한다. 제안 시스템은 FreeCAD를 별도 프로세스로 구동하고 XML-RPC로 통신함으로써, CAD 실행 환경과 에이전트 실행 환경을 분리한다. 이러한 프로세스 분리는 FreeCAD의 충돌이나 오류가 전체 시스템으로 전파되는 것을 방지하고, 필요 시 CAD 프로세스만 독립적으로 재시작할 수 있게 하여 운영 안정성을 높인다.

FreeCAD 측 RPC 서버는 FreeCAD 애드온(Addon) 형태로 제공되며, GUI 내에서 활성화되어 외부 요청을 처리한다. 요청은 Qt 타이머 기반 작업 큐에 적재된 뒤 FreeCAD 메인 GUI 스레드에서 순차 처리되어 이벤트 루프와의 충돌을 회피한다. RAG 계층에서 선택·치환된 샘플 코드는 문서 컨텍스트에서 실행되며, 실행 과정에서 표준 출력/에러와 콘솔의 경고·오류 메시지를 수집한다. 실행 후 문서 재계산과 3D 뷰 스크린샷 캡처를 수행해 결과와 진단 정보를 에이전트 계층에 전달하며, 이는 재시도 또는 다음 단계 진행 판단에 활용된다.

4. 평가 설정

본 장에서는 제안 시스템의 실용적 타당성을 평가한다. 평가는 전체 설계 모드에서의 시나리오 기반 반복 실행과 항공 제조 현장 관점의 잠재 사용자 평가로 구성한다.

4.1 연구 질문

본 연구는 다음의 연구 질문을 설정한다.

- RQ1. 제안 시스템은 전체 치공구를 자동 생성하기 까지 어느 정도의 시간이 소요되는가?
- RQ2. 제안 시스템은 부속 단위 실행과 오류 국소화를 통해 실패를 제한하고 복구 가능성을 제공하는가?
- RQ3. 항공기 치공구 설계 실무 관점에서 제안 시스템은 현장 적용 가능성을 갖는가?

RQ1은 전체 설계 소요 시간을 지표로 사용한다. 시간은 전체 설계 실행 시점부터 모든 부속 생성이 완료되는 시점까지로 정의한다. RQ2는 실행 성공률과 실패 유형 분포를 지표로 사용한다. RQ3은 잠재 사용자 설문 점수와 자유 의견을 지표로 사용하며, 각 항목은 5점 척도로 평가 후 100점 만점으로 환산한다.

4.2 실험 설정

4.2.1 가공대상과 입력 조건

평가는 5개의 L자 형상 가공대상을 사용한다. 각 가공대상은 서로 다른 크기와 부피를 지니므로, 각각 생성해야하는 부속의 적정 치수, 배치 위치, 개수가 달라진다. 이를 통해 시스템의 적응 능력을 평가할 수 있다. 그림 4는 가공대상의 형상과 치수 정보를 보인다.

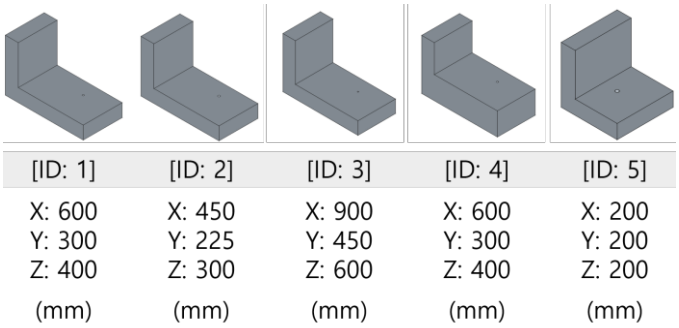


그림 4. 평가에 사용한 5개의 L자형 가공대상

4.2.2 실험 환경

실험은 로컬에서, LLM API 호출을 위해 온라인이 연동된 환경에서 수행하였다. 표 6은 본 실험의 소프트웨어, 기술 스택과 하드웨어 및 운영체제 환경을 통합하여 요약한다.

표 6. 실험 환경

구분	항목	사양/역할
CAD	FreeCAD	1.0.2 ver. 형상 생성 실행 환경
구현 언어	Python	3.10.16 ver. 시스템 코어
통신	XML-RPC	FreeCAD 프로세스와 시스템 코어 통신
RAG DB	Chroma DB	설계 샘플 코드 및 메타데이터 저장
LLM	Claude	claude-sonnet-4-5-20250929 ver.
운영체제	Windows	Windows 11
CPU	Intel	Intel Core Ultra 5 125H
GPU	NVIDIA	GeForce RTX 3050 Laptop GPU

4.2.3 평가 절차

평가는 다음 순서로 수행한다. 평가자는 STEP 파일을 시스템에 업로드하고, 3개의 기준면과 가공 구멍을 지정한 뒤, 자연어 요청 “치공구를 설계해”를 입력 후 전체 설계를 실행한다. 이후 각 부속의 생성 상태를 확인하고 결과를 검토한다. 각 가공대상에 대해 10회 반복 수행 후 평균을 산출하였으며, 반복 실행마다 FreeCAD 문서를 초기화하였다.

잠재 사용자 평가는 국내 대형 항공사의 항공기 치공구 설계 실무 경력을 보유한 2명의 평가자가 참여하였다. 평가자는 시스템 시연을 관찰한 후, 자동 생성 모델의 정확성(10개 항목)과 시스템 사용 적합성(5개 항목)에 대해 5점 척도로 평가하고 자유 의견을 기술하였다.

5. 평가 결과

5.1 전체 설계 소요 시간 (RQ1)

RQ1은 " 제안 시스템은 전체 치공구를 자동 생성하기 까지 어느 정도의 시간이 소요되는가?"를 묻는다. 표 7은 가공대상별 전체 설계 평균 소요 시간과 표준편차를 제시한다. 이 시간은 잘못된 생성 결과에 대한 복구 시간을 포함하며, 최종 설계 미완료의 사례 또한 포함한다.

표 7. 가공대상별 전체 설계 소요 시간 (10회 반복)

가공대상	평균 시간	표준편차
1	2 분 1 초	2.3 초
2	1 분 55 초	2.9 초
3	1 분 48 초	1.5 초
4	1 분 49 초	2.3 초
5	1 분 51 초	1.4 초

평균 소요 시간은 1분 48초에서 2분 1초 범위에 분포하며, 가공대상 크기와 부피가 증가하더라도 전체 설계 시간이 일정 범위 내에서 유지되는 경향을 확인하였다. 이는 RAG 기반 코드 자산 재사용과 도구 계층화를 통해 실행 흐름이 가공대상 규모에 독립적으로 유지되도록 설계되었기 때문이다.

RQ1 결론: 제안 시스템은 서로 다른 크기의 가공대상에 대해 평균 1분 48초에서 2분 1초의 일관된 시간 내에 전체 치공구를 자동 생성할 수 있었다.

5.2 실패 유형과 복구 양상 (RQ2)

RQ2는 "제안 시스템은 부속 단위 실행과 오류 국소화를 통해 실패를 제한하고 복구 가능성을 제공하는가?"를 묻는다. 각 부속별 10회씩, 총 50회 실행 중 관측된 부속 단위 실패 이벤트는 총 13건이었으며, 최종 설계 미완료는 3회였다. 제안 시스템은 실패한 3회의 원인을 분석한 결과, 실행 과정에서 관측된 대표 실패 유형은 세 가지로 정리된다. 표 8은 각 실패 유형의 발생 빈도와 복구 결과를 정리한다.

표 8. 실패 유형과 복구 양상

실패 유형	발생횟수	발생률	자동복구	복구율
객체명/기준면 오류	7	14%	6	85.7%
배치 오프셋 오류	4	8%	3	75.0%
형상 재계산 오류	2	4%	1	50.0%
합계	13	26%	10	76.9%

첫째, 기준면 라벨과 가공대상 객체명 치환 오류로 인한 부속 생성 실패이다(7회 발생, 14%). RAG 샘플 코드가

전제하는 객체명이나 기준면 식별자가 입력과 불일치할 때 발생하며, 재시도를 통해 6회(85.7%) 복구에 성공하였다. 둘째, 배치 오프셋 설정 오류로 인한 간섭 발생이다(4회 발생, 8%). 특정 크기 구간에서 기본 오프셋이 부합하지 않는 경우 부속 간 간섭이 관측되었으며, 오프셋 조정을 통해 3회(75%) 복구하였다. 셋째, FreeCAD 재계산 시점에서의 형상 일관성 오류이다(2회 발생, 4%). 파라메트릭 스케치 제약이 충돌할 때 재계산이 실패하며, 이 유형은 샘플 코드 자체의 상세한 수정이 필요하여 자동 복구율이 낮았다.

제안 시스템은 부속 단위 실행을 통해 실패를 부속 단위로 제한하고, 동일 부속 유형의 다른 샘플 재선택 또는 치환 변수 조정으로 재시도를 수행한다. 이를 통해, 전체 13회의 부속 단위 실패 중 10회(76.9%)가 자동 재시도로 복구되었다.

RQ2 결론: 제안 시스템은 부속 단위 실행을 통해 실패를 격리하고, 50회의 시행 중 76.9%의 자동 복구율로 오류 국소화와 복구 가능성을 제공한다.

5.3 잠재 사용자 평가 결과 (RQ3)

RQ3은 "항공기 치공구 설계 실무 관점에서 제안 시스템은 현장 적용 가능성을 갖는가?"를 묻는다. 항공기 치공구 설계 실무 경력을 보유한 2명의 평가자가 시스템 시연을 관찰한 후, 10개 항목의 자동 생성 모델의 정확성과 5개 항목의 시스템 사용 적합성을 평가하였다. 각 항목은 5점 척도로 평가하였으며, 총점은 (각 항목 점수의 합 / 항목 수 × 5) × 100으로 100점 만점 환산하였다. 표 9은 잠재 사용자의 평가 점수를 요약한다.

표 9. 잠재 사용자 평가 점수 요약 (5점 척도, 100점 환산)

평가 영역	평가자 A	평가자 B	평균
자동 생성 정확성	88	86	87
시스템 사용 적합성	96	88	92

또한, 표 10은 평가자들이 제시한 긍정적 피드백과 개선 요구 사항을 정리한다.

표 10. 잠재 사용자 정성 피드백 요약(긍정/개선 요구)

긍정적 피드백
● 전반적인 개발 완성도가 높고 기본 기능이 잘 동작함 ● 3D 형상 생성 결과와 배치가 직관적으로 확인 가능함 ● 설계안 반복 개선과 썸네일 비교 기능이 실무에 유용함 ● 향후 다른 부속(엔진 마운트 등)으로 확장 가능성 있음
개선 요구 사항
● 샘플 데이터베이스 확충 필요. 현재 DB 로는 다양한 형상에 대응하기 어려우며, 더 많은 사례 축적이 요구됨 ● 크기 조정(스케일링) 관련 일부 불안정: 특정 크기 구간에서 배치가 부정확한 경우가 관측됨 ● 사용자 접근성 개선 필요: 초기 설정과 인터페이스가 설계 실무자에게 익숙하지 않은 부분이 있음

평가자들은 시스템의 기본 기능과 자동 생성 품질에 대해 긍정적으로 평가하였으나, 실제 현장 적용을 위해서는 샘플 데이터베이스의 확충이 선행되어야 함을 지적하였다. 이는 RAG 기반 접근의 특성상 샘플 코드의 다양성과 품질이 시스템 성능에 직접적으로 영향을 미치기 때문이다.

RQ3 결론: 제안 시스템은 기본적인 현장 적용 가능성을 갖추었으나, 샘플 데이터베이스 확충과 사용자 접근성 개선이 실제 적용의 전제 조건으로 확인되었다.

6. 논의

본 연구는 치공구 자동설계를 운영형 자동화 문제로 정의하고, RAG 기반 코드 자산 재사용, MCP 기반 도구 호출 계층화, 역할 기반 멀티 에이전트 분해를 결합한 FreeCAD 오토코딩 시스템을 구현하였다. 평가 결과, 서로 다른 크기의 5개 가공대상에 대해 전체 설계 시간이 1분 48초에서 2분 1초 범위로 유지되었고(RQ1), 부속 단위 실패의 76.9%가 자동 재시도로 복구되었으며(RQ2), 잠재 사용자 평가에서 자동 생성 정확성 87점, 사용 적합성 92점을 기록하였다(RQ3). 이는 표 2에서 제시한 설계 목표가 실험적으로 달성되었음을 시사한다.

선행연구와 비교할 때, 본 연구는 생성, 검증, 협업 요소를 치공구 도메인에 맞는 단일 구조로 통합하고, 부속 단위 분해와 도구 호출 표준화를 통해 책임 범위와 실행 단위를 명확히 했다는 점에서 차별화된다. 한편, 부속 분해 체계의 확장성, 자연어 기반 수정 요청의 안정성, 간섭 검증 자동화의 제한, 기존 방식 대비 비교 평가 부족이 한계로 확인되었다.

잠재 사용자 평가에서는 샘플 코드 자산의 확충이 현장 적용의 전제 조건으로 제시되는 동시에, 엔진 마운트 등 다른 부속으로의 확장 가능성이 언급되었다. 이는 MCP 기반 도구 계층화와 RAG 기반 재사용 구조가 샘플 코드 데이터베이스 확충을 통해 유사한 반복 설계 문제에 적용될 수 있음을 시사한다. 본 연구는 이러한 구조적 접근을 통해 소프트웨어공학의 모듈화, 재사용성, 운영화 원칙이 AI 기반 제조 설계 자동화 영역에서도 유효할 수 있음을 보여준다.

7. 결론

본 연구는 치공구 자동설계를 위한 멀티 에이전트 기반 FreeCAD 오토코딩 시스템을 제안하였다. 치공구를 부속 단위로 분해하고, RAG 기반 코드 자산 재사용과 MCP 기반 도구 계층화를 결합과 더불어 멀티 에이전트 구조를 채택하여 LLM의 부담을 구조적으로 축소하면서 실행 가능한 산출물을 안정적으로 확보하였다.

5개의 L자 형상 가공대상에 대한 평가에서 평균 1분 48초에서 2분 1초의 일관된 설계 시간과 76.9%의 자동 복구율을 확인하였으며, 잠재 사용자 평가에서 정확성 87점, 사용 적합성 92점을 기록하였다. 향후 연구에서는 부속 분해 체계의 확장, 3차원 추론 특화 모델 파인튜닝, 간섭 검출 기능 통합, 다양한 형상과 확대된 사용자 집단을 대상으로 한 비교 평가를 수행할 계획이다.

[1] Boeing, Boeing Forecasts Demand for Nearly 44,000 New Airplanes Through 2043 as Air Travel Surpasses Pre-Pandemic Levels, Jul. 19, 2024.
[2] Oliver Wyman, Global Fleet and MRO Market Forecast 2024-2034, Feb. 2024.
[3] PwC, Advanced Air Mobility: From Concept to Commercial Reality, 2025.
[4] 통계청, 내외국인 인구전망(2022-2042): 생산가능인구 전망, Apr. 11, 2024.
[5] Nee, A. Y. C., et al. (1995). An intelligent fixture design system. CIRP Annals, 44(1), 149-152.
[6] Caracol. (2023). Aerospace Tooling Jig Case Study.
[7] Kumbhar, N. N., & Pandit, D. V. (2017). A Review on Jig and Fixtures Design.
[8] Meng, S., et al. (2024). Intelligent design of reconfigurable flexible assembly fixture for aircraft panels. Journal of Engineering Design, 36(5-6), 672-706.
[9] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in Advances in Neural Information Processing Systems, vol. 33, pp. 9459-9474, 2020.
[10] Anthropic, "Model Context Protocol Specification," 2024.
[11] A. Seff et al., "Vitruvion: A Generative Model of Parametric CAD Sketches," arXiv preprint arXiv:2109.14124, 2021.
[12] R. Wu et al., "DeepCAD: A Deep Generative Network for Computer-Aided Design Models," in Proc. IEEE/CVF Int. Conf. on Computer Vision, 2021, pp. 6772-6782.
[13] M. F. Alam et al., "GenCAD: Image-Conditioned Computer-Aided Design Generation with Transformer-Based Contrastive Representation and Diffusion Priors," arXiv preprint arXiv:2409.16294, 2024.
[14] M. Kolodiazhnyi et al., "cadrille: Multi-modal CAD Reconstruction with Online Reinforcement Learning," arXiv preprint arXiv:2505.22914, 2025.
[15] G. Nakayama, "Learning CAD Program Generation using Reinforcement Learning," Stanford CS224R Project Report, 2022.
[16] S. Wu et al., "CAD-LLM: Large Language Model for CAD Generation," in NeurIPS Workshop on Machine Learning for Creativity and Design, 2023.
[17] M. S. Khan et al., "Text2CAD: Generating Sequential CAD Designs from Beginner-to-Expert Level Text Prompts," in Advances in Neural Information Processing Systems, vol. 37, pp. 7552-7579, 2024.
[18] J. Li et al., "CAD-Llama: Leveraging Large Language Models for Computer-Aided Design Parametric 3D Model Generation," in Proc. IEEE/CVF Conf. on Computer Vision and Pattern Recognition, 2025, pp. 18563-18573.
[19] H. Xie and F. Ju, "Text-to-CadQuery: A New Paradigm for CAD Generation with Scalable Large Model Capabilities," arXiv preprint arXiv:2505.06507, 2025.
[20] J. Xu et al., "CAD-MLLM: Unifying Multimodality-Conditioned CAD Generation With MLLM," arXiv preprint arXiv:2411.04954, 2024.
[21] Z. Yuan et al., "OpenECAD: An Efficient Visual Language Model for Editable 3D-CAD Design," Computers & Graphics, vol. 124, p. 104048, 2024.
[22] S. Wu et al., "CadVLM: Bridging Language and Vision in the Generation of Parametric CAD Sketches," in European Conf. on Computer Vision, 2024, pp. 368-384.
[23] X. Li et al., "LLM4CAD: Multimodal Large Language Models for Three-Dimensional Computer-Aided Design Generation," J. Computing and Information Science in Engineering, vol. 25, no. 2, 2025.
[24] A. Badagabettu et al., "Query2cad: Generating CAD Models Using Natural Language Queries," arXiv preprint arXiv:2406.00144, 2024.
[25] K. Alrashedy et al., "Generating CAD Code with Vision-Language Models for 3D Designs," arXiv preprint arXiv:2410.05340, 2024.
[26] R. Wang et al., "Text-to-CAD Generation Through Infusing Visual Feedback in Large Language Models," arXiv preprint arXiv:2501.19054, 2025.
[27] D. Mallis et al., "CAD-Assistant: Tool-Augmented VLLMs as Generic CAD Task Solvers," arXiv preprint arXiv:2412.13810, 2024.
[28] Y. Yuan et al., "CAD-Editor: A Locate-then-Infill Framework with Automated Training Data Synthesis for Text-Based CAD Editing," arXiv preprint arXiv:2502.03997, 2025.
[29] J. Chen et al., "CADReview: Automatically Reviewing CAD Programs with Error Detection and Correction," arXiv preprint arXiv:2505.22304, 2025.
[30] L. Makatura et al., "Large Language Models for Design and Manufacturing," 2024.
[31] L. Makatura et al., "How Can Large Language Models Help Humans in Design and Manufacturing? Part 1: Elements of the LLM-Enabled Computational Design and Manufacturing Pipeline," Harvard Data Science Review, Special Issue 5, 2024.
[32] J. Lim et al., "Large Language Model-Enabled Multi-Agent Manufacturing Systems," in 2024 IEEE Int. Conf. on Automation Science and Engineering, 2024, pp. 3940-3946.
[33] F. Ocker et al., "From Idea to CAD: A Language Model-Driven Multi-Agent System for Collaborative Design," arXiv preprint arXiv:2503.04417, 2025.
[34] J. Liao et al., "AutoForma: A Large Language Model-Based Multi-Agent for Computer-Automated Design," in 2024 IEEE Int. Conf. on Systems, Man, and Cybernetics, 2024, pp. 1284-1289.
[35] H. Deng et al., "An Investigation on Utilizing Large Language Model for Industrial Computer-Aided Design Automation," Procedia CIRP, vol. 128, pp. 221-226, 2024.

도메인 특화 임베딩 학습을 활용한 한국어 법률 질의응답 RAG 시스템 최적화 연구

배소연°[1], 장진우[2], 이주형[2], 박진경[3],
딥모달[1], 미디어젠[2], 공정거래위원회[3]
{baesy1004}@gmail.com, {jwjang311, iwbaporandhh}@mediazen.co.kr, pjk0321@korea.kr

Optimizing Korean Legal RAG Systems via Domain-Specific Embedding Training

So Yeon Bae°[1], Jin Woo Jang[2], Joo Hyeong Lee[2], Jin Kyeong Park[3]
[1]DeepModal Inc., [2]MediaZen Inc., [3]Fair Trade Commission

요 약

대규모 언어모델을 활용한 법률 질의응답 시스템에서는 검색 증강 생성(Retrieval-Augmented Generation, RAG)을 통해 법률적 정확성과 신뢰성을 향상시키는 연구가 활발히 진행되고 있다. 그러나 RAG 시스템의 성능은 검색 결과 문서의 품질에 크게 좌우되며, 특히 한국어 법률 도메인에서는 범용적인 검색 모델이 최적의 문서 검색을 충분히 수행하지 못하는 한계가 있다. 본 연구에서는 한국어 법률 RAG 시스템을 최적화하고자 도메인 특화 임베딩 모델을 학습하기 위해 다양한 출처에서의 질의-근거 형태의 데이터를 수집하여 LLM 기반의 데이터 필터링을 통해 품질을 개선하고, 법령정보센터 API를 활용해 데이터셋을 구축하였다. 실험 결과, 제안하는 모델은 기존 베이스라인 모델 대비 약 5.6% 향상된 검색 성능을 기록하며 한국어 법률 도메인에서의 유의미한 검색 품질 개선 효과를 입증하였다.

1 서론

최근 대규모 언어모델(LLM)의 발전으로 법령 및 판례를 검색, 요약, 질의응답하는 법률 AI 시스템이 활발히 연구되고 있다. 그러나 법률 도메인의 경우 응답의 유창성보다 정확한 조문 및 판례의 인용과 근거 제시가 답변의 신뢰성을 크게 좌우한다는 점에서 RAG 시스템 개선의 중요성이 강조되고 있다.

특히 사전학습 기반 LLM은 법령의 개정 및 최근 판례를 즉시 반영하기가 어려워 그 결과 인용 판례 및 법령이 부정확하거나 근거가 빈약한 환각(hallucination) 현상이 발생하기 쉽다. 이러한 한계를 개선하기 위한 대표적인 방법인 RAG 시스템은 검색 단계에서의 검색 결과 문서 품질에 크게 좌우되며, 무관한 문서의 검색 및 누락은 법률 분야의 질의응답에서의 응답 신뢰성과 정확성을 약화시킨다.

문서 검색을 위해 일반적으로 활용되는 범용 임베딩 모델의 경우 질문 및 문서에서의 법률 용어 간의 연관성 및 의미를 충분히 반영하지 못하고, 도메인 특화 임베딩 학습에 필요한 질의-근거 형태의 고품질 데이터셋도 제한적이라는 문제가 있다.

본 연구는 위 한계를 해결하기 위해 한국어 법률 RAG 시스템의 검색 기반을 강화하는 데 초점을 맞춘다. 구체적으로, 다양한 출처에서 한국어 법률 질의-근거 데이터를 수집하고 LLM 기반 필터링으로 품질을 개선한 뒤, 국가법령정보센터 API를 활용해 데이터셋을 구축하였다. 이후 해당 데이터셋을 기반으로 도메인 특화 임베딩 모델을 학습하여 한국어 법률 문서 검색 성능을 향상시키고, 궁극적으로 RAG 기반 질의응답의 신뢰성과 정확성 개선을 목표로 한다.

2 관련연구

법률 도메인 QA에서 LLM은 상담형 질의응답, 판례 요약, 조문 검색 등으로 활용 범위가 확장되었으나, 범용 사전학습 모델은 최신 개정 반영 한계와 환각 문제로 인해 직접적인 법률 의사결정 지원에 적용하기 어렵다는 비판이 지속적으로 제기되어 왔다[1].

이러한 한계를 완화하기 위해 검색 증강 생성(RAG)이 도입되었으며, 한국 법률 QA에서도 국가법령정보를 기반으로 한 법령·판례 검색 후 GPT 계열 LLM으로 답변을 생성하는 시스템[2]이 제안되어 사실성과 근거성을 동시에 강화하려는 시도가 보고되었다.

그러나 RAG 구조에서는 검색기가 부정확한 문서를 반환하거나 핵심 근거를 누락할 경우, 생성 단계에서 오히려 잘못된 근거를 강화하는 오류가 발생할 수 있어, 검색 단계 최적화

및 법률 도메인 특화 평가 지표를 포함한 'retrieval 품질 관리' 가 핵심 연구 축으로 부상하였다[1, 2].

정보 검색을 위한 신경망 기반의 임베딩 모델 학습의 경우 질의-근거문서 쌍의 의미적 유사성을 높이는 방식으로 학습이 진행된다. 특히 DPR[3]는 대량의 질의-근거문서 쌍에 대한 대조학습을 통해 dense dual-encoder를 학습함으로써 BM25 대비 Top-k 정확도를 크게 향상시키는 것으로 보고되었고, 이후 E5[4], ColBERT[5] 등 데이터셋 구축 및 아키텍처 개선 등으로 다양한 QA 벤치마크에서 성능 개선을 달성하였다.

전문 도메인에서는 범용 임베딩보다 도메인 특화 임베딩이 검색 품질과 RAG 응답 신뢰도를 유의미하게 개선한다는 결과가 반복적으로 보고되며, 특히 법률·금융 등 고신뢰 영역에서 그 효과가 두드러진다[6].

하지만 한국어 법률 환경에서는 임베딩 학습에 적합한 질의-근거 데이터의 확보가 쉽지 않고, 데이터 정합성(근거의 실제 존재 여부, 개정/버전, 인용 가능성)과 노이즈 통제가 성능을 좌우한다는 실무적 문제가 존재한다. 따라서 본 연구는 한국어 법률 RAG의 검색 병목을 '임베딩 모델' 차원에서 직접 개선하되, 이를 뒷받침하기 위한 데이터 구축을 LLM 기반 필터링과 국가법령정보센터 API 기반 정합성 통제로 체계화하여, 도메인 특화 임베딩 학습의 현실적 장벽을 낮추는 접근을 제안한다.

3 데이터 구축 과정

3.1 활용 데이터

본 연구에서 활용한 원시 데이터는 총 22,695건이다. 데이터의 집계 단위는 유형별로 상이하여 민원은 1건, 불공정 약관은 1개 조항, 판례는 1개 사건을 각각 1건으로 정의하였다. 특히 불공정 약관 데이터의 경우, 공정거래위원회(이하 공정위)가 공개한 불공정 행위 민원 및 약관 심사 사례를 활용하였다. 공정위 자료는 불공정성 판단의 근거가 되는 법령과 해석을 포함하고 있으므로, 본 연구에서는 이를 기반으로 약관 조항(Anchor)과 근거 법령(Positive)을 매핑하여 데이터셋을 구축하였다. 데이터셋의 상세 구성은 표 1과 같다.

표 1. 데이터셋 상세 구성

데이터셋	건수	출처
공공 민원 상담 LLM 사전학습 및 Instruction Tuning	4484	AIHUB
법률/규정 텍스트 분석 데이터(고도화) - 상황에 따른 판례 데이터 中 민사 판례	15327	AIHUB
불공정약관 데이터	2884	공정위

3.2 원시 데이터 처리

성능 고도화를 위해서는 양질의 데이터 구축이 필수적이다. 이에 본 연구에서는 실제 민원 및 불공정 약관 데이터를 정제하여 활용하였으며, 이를 위해 그림 1와 같이 다양한 형태의 원시 데이터를 모델 학습에 적합한 형태로 구조화하는 전처리 과정을 수행하였다.

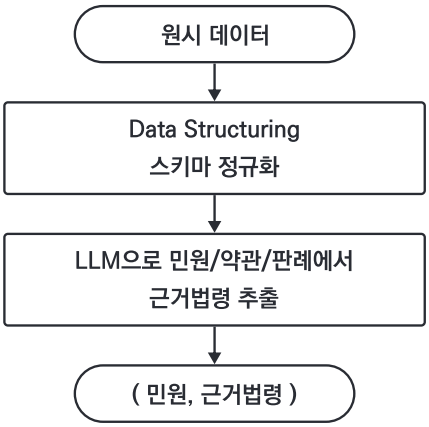


그림 1. 원시 데이터 처리 및 구조화 과정

우선, 수집된 민원 및 판례 원시 데이터에 대해 스키마 정규화를 거쳐 데이터 필드를 통일하였다. 이후 거대언어모델(LLM)을 활용하여 데이터의 품질을 고도화하였다. 구체적으로 사용된 프롬프트 및 LLM 설정은 표 2과 같다. 관련 법령이 명시되지 않은 민원 데이터의 경우, 본문 내용을 바탕으로 연관된 법령을 추출하였다.

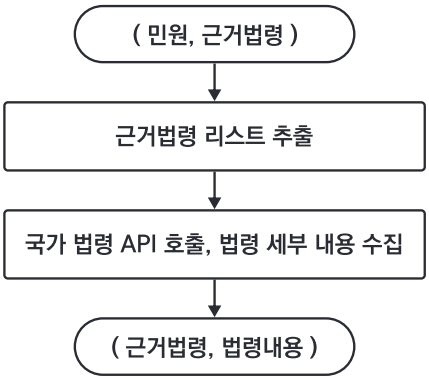


그림 2. 국가법령센터 API 기반 법령 데이터 구축

표 2. LLM 생성 환경 변수

변수	값
model	Qwen3
size	8B
temperature	0
max_tokens	2048
prompt	당신은 대한민국 법령 및 행정 문서 텍스트에서 특정 정보를 추출하여 지정된 JSON 형식으로 정리하는 고도로 숙련된 법률 정보 분석 AI입니다. 주어진 입력 텍스트(batch_data)에서 'report_id', 'related_laws', 'related_cases', 'is_revised'를 추출하여 JSON 배열로 반환 **처리 절차** (Thought Process) • 전체 텍스트 순차적 분석: 처음부터 끝까지 순서대로 독해. • '현재 법령' 기억: '법' 또는 '시행령'으로 끝나는 명칭 저장 (Context 유지) • 참조어 처리(핵심): '동법', '같은 영', '법 제N조' 등의 참조어를 기억해둔 법령명으로 치환 (예: '같은 법 제64조' → '산업안전보건법 제64조') • 정제 및 필터링: 목록 접두사(가., 나.) 제거 및 JSON 스키마 매핑. Few-shot Example (...지면 관계상 생략...)

3.3 외부 지식 베이스 기반 법령 데이터 확장

민원 답변 생성 시 환각을 방지하고 정확한 법적 근거를 제시하기 위해, 그림 2와 같이 외부 지식 베이스를 구축하였다

(1) **법령명 정규화:** 1차 스키마 데이터의 related_laws 필드에는 '동법', '같은 영'과 같은 지시적 표현이 혼재되어 있어, 이를 검색 쿼리로 직접 활용하는 데 한계가 있다. 이를 극복하기 위해 문맥 인식 기반의 상호참조 해결 알고리즘을 적용하였다. 구체적으로, 문맥 내에서 명시된 법령명 중 가장 긴 형태를 기준으로 식별하고, 후행하는 지시어(예: 동법 제N조)를 해당 법령명으로 동적으로 치환하여 온전한 법적 엔티티로 복원하였다.

(2) **상세 법령 API 검색 및 검증:** 특정 조항을 정확히 식별하기 위해 자연어 형태의 조문 번호(예: 제12조의2)를 API 규격인 6자리 코드(예: 001202)로 변환하는 전처리를 선행하였다. 이후 정규화된 법령명과 변환된 조문 코드를 기반으로 국가법령정보센터 API를 호출하여 해당 조문의 전문을 수집한다. 이 과정에서 시행일자 메타데이터를 활용해 현행 법령 여부를 검증함으로써, 모델이 유효한 법적 효력을 지닌 데이터만을 학습하도록 보장한다.

3.4 최종 학습 데이터셋 구축

수집된 상세 법령 정보를 원래의 민원 데이터와 매핑하여 최종 학습 데이터셋을 완성한다. 전체적인 데이터 구축 파이프라인은 그림 3에 도식화하였다. 기존의 단순 문자열 리스트였던 related_laws 필드를 상세 법령 객체 리스트로 대체함으로써, 모델이 민원 내용과 함께 법적 근거의 실제 텍스트(Ground Truth Context)를 동시에 학습할 수 있도록 구성하였다.

Step 1. Raw Data (비정형 텍스트)

```
{
  "content": "채권자 대위권 행사에 의한 소송에...",
  "Reference_info": {
    "reference_rules": "민법 제404조, 민사소송법 제
  202조, 제234조"
  },
}
```

↓ LLM 구조화 (Refinement)

Step 2. Structured (리스트 정규화)

```
{
  "content": "채권자 대위권 행사에 의한 소송에...",
  "related_laws": [
    "민법 제404조",
    "민사소송법 제202조",
    "민사소송법 제234조"
  ]
}
```

↓ API를 통한 법령 조문 수집

Step 3. Final Input (조문 내용 주입)

```
{
  "content": "채권자 대위권 행사에 의한 소송에...",
  "related_laws": [
    {
      "참고법령": "민법 제404조",
      "참고법령세부": "제404조(채권자대위권) ①채권
  자는 자기의 채권을 보전하기 위하여 채무자의 권리를
  행사할 수 있다..."
    },
  ]
}
```

그림 3. 데이터 처리 파이프라인. 실제 조문 내용이 포함된 학습 데이터로 확장되는 과정

법령 데이터 확장 과정을 거친 결과, 원시 데이터 22,695건은 Anchor-Positive 쌍(Pair) 기준으로 재구성되어 데이터 수가 확장되었다. 하나의 민원이나 판례가 여러 법령 조항과 연결될 수 있으므로, 각 조항마다 별도의 학습 쌍을 생성하였다. 전체 데이터는 학습(Train), 검증(Validation), 평가(Test) 셋으로 8:1:1 비율로 무작위 분할하였으며, 최종 구축된 학습 데이터셋의 통계는 표 3와 같다.

표 3. 최종 학습 데이터셋 통계

Split	Ratio	Count (Pairs)
Train	80%	71,992
Validation	10%	8,907
Test	10%	9,095
Total	100%	89,994

4 실험

4.1 실험 환경 및 설정

본 실험은 NVIDIA A100 GPU 환경에서 수행되었다. 본 연구에서는 제안하는 도메인 특화 임베딩 모델의 효용성을 검증하기 위해, 최신 한국어 임베딩 모델인 `nlpai-lab/KoE5`[1]와 `dragonkue/snowflake-arctic-embed-l-v2.0-ko`[2]를 베이스라인으로 선정하였다.

구체적으로 KoE5는 `multilingual-e5-large`를 기반으로 다양한 한국어 데이터셋을 통해 추가 학습된 모델로 약 5억 6천만(560M) 개의 파라미터를 보유하고 있으며, Snowflake-Arctic-Ko는 `snowflake-arctic-embed-l-v2.0`을 한국어로 파인튜닝한 모델로 약 5억 6천 8백만(568M) 개의 파라미터를 가진다.

두 모델을 비교군으로 선정한 핵심 이유는 다음과 같다. 첫째, 두 모델 모두 약 5.6억 개의 파라미터를 보유한 동급의 Large 모델로서, 모델 체급 차이에 따른 성능 편향을 효과적으로 통제할 수 있다. 둘째, 두 모델 모두 범용 한국어 데이터로 사전 학습이 완료된 상태이므로, 본 연구가 제안하는 '법률 도메인 특화 학습'이 실제 검색 성능 향상에 기여하는지 명확히 검증할 수 있다.

4.2 대조학습 방법

본 연구에서는 별도의 Hard Negative Mining 없이, 배치 내의 다른 샘플들을 음성 샘플로 활용하는 In-batch Negative Sampling[3] 방식을 채택하였다. 이 방식은 배치 크기가 N일 때, 각 질의 q에 대해 대응하는 정답 문서 d^+를 제외한 나머지 N−1개의 문서를 자동으로 음성 샘플(Negative samples)로 간주한다. 이는 의미적으로 유사하지만 법적으로는 서로 다른 문서가 Negative로 포함될 가능성을 높여, 모델이 미세한 법률적 의미 차이를 구분하도록 유도한다.

학습에 사용된 `MultipleNegativesRankingLoss (MNRL)`는 다음과 같은 InfoNCE 형태의 수식으로 정의된다.

$$\mathscr{L} = -\log \frac{\exp(s(q,d^+)/\tau)}{\sum_{i=1}^{N} \exp(s(q,d^i)/\tau)}$$

여기서 N은 배치 사이즈를 의미하며, $s(\cdot)$는 질의(q)와 문서(d) 간의 코사인 유사도 함수, 는 소프트맥스 분포의 선명도를 조절하는 온도 매개변수를 나타낸다.

손실 함수 측면에서는 기본 MNRL과 메모리 효율을 극대화한 `CachedMultipleNegativesRankingLoss (CMNRL)`를 비교 분석하였다. 실험 결과, 동일하거나 유사한 배치 조건에서 CMNRL을 적용한 설정이 전반적으로 우수한 성능을 보였다. 이는 CMNRL이 캐싱 매커니즘을 통해 실질적으로 더 많은 수의 Effective Negative를 학습에 활용할 수 있게 함으로써 대조 학습의 효율을 높였기 때문으로 해석할 수 있다.

4.3 실험 결과 및 분석

두 베이스라인 모델의 법률 데이터셋에 대한 제로샷 검색 성능과 파인튜닝 모델의 최고 성능은 표 4와 같다.

표 4. 베이스라인 모델과 파인튜닝 모델의 성능 비교 요약

Model	Batch	LR	Loss	Recall@3	MRR@3	Recall@10	MRR@10
Baselines (Zero-shot)							
KoE5	-	-	-	0.7664	0.7085	0.8203	0.7187
Snowflake	-	-	-	0.7647	0.7115	0.8166	0.7214
Fine-tuned (Best Performance)							
KoE5 (Ours)	256	1e-4	CMNRL(32)	0.8082	0.7502	0.8543	0.7590
Snowflake (Ours)	256	1e-4	CMNRL(64)	0.8073	0.7513	0.8512	0.7599

표 4에 나타난 바와 같이, 베이스라인 모델들은 제로샷 환경에서 MRR@10 기준 약 0.72 수준의 성능을 기록하였다. 이는 범용 한국어 임베딩 모델이 법률 도메인에서도 기초적인 검색 능력을 갖추고 있으나, 복잡한 법률 용어와 특수한 조문 구조를 정밀하게 파악하여 최상위 랭크에 배치하는 데에는 한계가 있음을 보여준다.

베이스라인 평가 후, 본 연구에서 구축한 법률 특화 데이터셋을 활용하여 KoE5 모델의 파인튜닝을 진행하였다. 파인튜닝 과정에서는 배치 크기(Batch Size), 학습률(Learning Rate), 손실 함수(Loss Function) 등 주요 하이퍼파라미터에 대한 탐

[1] https://huggingface.co/nlpai-lab/KoE5
[2] https://huggingface.co/dragonkue/snowflake-arctic-embed-l-v2.0-ko

색 실험을 수행하여, 법률 문서 검색에 적합한 학습 설정을 도출하고자 하였다.

표 5. 하이퍼파라미터 설정에 따른 상세 실험 결과

Model	Batch	LR	Loss Function	R@3	MRR@3	R@10	MRR@10
KoE5 (Fine-tuned)							
	128	2e-5	MNRL	0.7965	0.7423	0.8490	0.7523
	128	2e-5	CMNRL (64)	0.7938	0.7406	0.8483	0.7511
	128	2e-5	CMNRL (32)	0.8015	0.7478	0.8459	0.7563
	256	2e-5	MNRL	0.7994	0.7431	0.8480	0.7526
	256	1e-4	MNRL	0.8045	0.7474	0.8539	0.7570
	256	1e-4	CMNRL (64)	0.8032	0.7489	0.8515	0.7580
	256	1e-4	CMNRL (32)	0.8082	0.7502	0.8543	0.7590
	512	1e-4	MNRL	0.8005	0.7486	0.8504	0.7583
	1024	1e-4	CMNRL (32)	0.7991	0.7421	0.8479	0.7514
	1024	2e-4	CMNRL (32)	0.7971	0.7420	0.8446	0.7514
	2048	2e-4	CMNRL (32)	0.7874	0.7339	0.8401	0.7438
Snowflake-Arctic-Ko (Fine-tuned)							
	128	2e-5	MNRL	0.7986	0.7485	0.8503	0.7584
	128	2e-5	CMNRL (64)	0.7988	0.7467	0.8514	0.7568
	128	2e-5	CMNRL (32)	0.8033	0.7474	0.8512	0.7567
	256	2e-5	MNRL	0.7991	0.7466	0.8528	0.7566
	256	1e-4	MNRL	0.7994	0.7494	0.8510	0.7596
	256	1e-4	CMNRL (64)	0.8073	0.7513	0.8512	0.7599
	256	1e-4	CMNRL (32)	0.8039	0.7509	0.8510	0.7599
	512	1e-4	MNRL	0.8044	0.7494	0.8512	0.7585
	1024	1e-4	CMNRL (32)	0.7996	0.7439	0.8440	0.7525
	1024	2e-4	CMNRL (32)	0.7924	0.7412	0.8454	0.7515
	2048	2e-4	CMNRL (32)	0.7846	0.7322	0.8365	0.7423

본 연구에서 제안하는 파인튜닝 모델들은 모든 지표에서 베이스라인을 유의미하게 상회하였다. 특히 `KoE5` 모델은 MRR@10이 0.7590으로 제로샷 대비 약 5.6% 향상되었으며, `Snowflake-Arctic-Ko` 모델은 0.7599를 기록하며 제로샷 대비 약 5.3%의 성능 향상을 달성하였다.

표 5의 상세 실험 결과를 통해 도출한 주요 분석 결과는 다음과 같다.

- **손실 함수 및 Effective Negatives의 영향:** 동일한 배치 크기(256) 내에서 `MNRL`보다 `CMNRL`을 적용했을 때 성능이 전반적으로 우수하였다. 이는 `CMNRL`이 캐싱 메커니즘을 통해 더 많은 수의 effective negative 샘플을 학습에 활용함으로써, 법률 문장 간의 미세한 의미 차이를 구별하는 변별력을 높였기 때문으로 분석된다.
- **배치 크기(Batch Size)의 임계점:** 배치 크기가 256일 때 최적의 성능을 보였으나, 1024 이상의 대규모 배치에서는 오히려 성능이 하락하는 경향이 관찰되었다. 이는 배치 크기가 과도하게 커질 경우, 배치 내에 '거짓 부정(False Negative)' 샘플(즉, 의미적으로는 유사하지만 레이블만 다르게 지정된 다른 조문들)이 포함될 확률이 높아져 학습의 수렴을 방해하는 노이즈로 작용했기 때문일 가능성이 크다.
- **상위 랭킹(MRR) 개선의 의의:** 모든 실험군에서 Recall@10의 개선 폭보다 MRR@10의 개선 폭이 상대적으로 더 두드러졌다. 이는 단순히 관련 조문을 10위권 내에 포함시키는 수준을 넘어, 질문에 직결되는 결정적 근거를 1~3위 내 최상단으로 정교하게 타격하여 배치하는 능력이 크게 강화되었음을 의미한다.

결과적으로, 배치 크기 256, 학습률 1e−4 조건에서 `CMNRL`을 적용한 설정이 가장 우수한 성능을 기록하였다. 특히 `Snowflake-Arctic-Ko` 기반 모델이 MRR@10 기준 0.7599를 기록하며 가장 높은 정밀도를 보였다. 이러한 검색 성능의 향상은 RAG 시스템의 생성 단계에서 LLM이 참조하는 컨텍스트의 순도를 높여, 환각 현상을 억제하고 법률 답변의 신뢰성을 확보하는 데 기여할 것으로 판단된다.

5 결론

본 연구에서는 한국어 법률 도메인에 특화된 RAG 시스템의 검색 성능을 향상시키기 위해, 고품질의 질의-법령 데이터셋 구축 파이프라인과 이에 기반한 임베딩 모델 최적화 방법을 제안하였다. 다양한 출처에서 수집한 원시 데이터에 대해 LLM 기반 문맥 정규화 및 필터링을 수행하고, 국가법령정보센터 API를 통해 조문 전문과 시행일자 메타데이터를 확보함으로써 근거의 정합성과 현행성을 검증한 89,994쌍의 학습 데이터를 구축하였다. 이를 바탕으로 대조 학습 기반 파인튜닝을 수행한 결과, 제안 모델은 범용 한국어 임베딩 모델 대비 MRR@10 기준 최대 약 5.6%의 성능 향상을 달성하였다. 또한 `CachedMultipleNegativesRankingLoss`의 도입과 적절한 배치 크기 설정이 법률 문서와 같이 미세한 의미 차이가 중요한 도메인에서 효과적임을 재확인하였다. 다만 본 연구는 RAG 시스템의 핵심 병목인 검색(Retrieval) 성능 최적화에 주안점을 두었기에, 검색된 문서를 바탕으로 최종 답변을 생성(Generation)하는 단계에 대한 정량적 평가는 포함하지 않았다는 한계가 있다. 따라서 향후 연구에서는 검색 결과를 기반으로 생성된 응답의 정확성과 근거 적합성을 종합적으로 평가하는 과정이 필요할 것이다. 특히 올바른 문서를 검색했음에도 생성 단계에서 오류가 발생하는 사례를 심층 분석한다면, 법률 질의응답 시스템의 신뢰성을 더욱 보완할 수 있을 것으로 기대된다.

감사의 글

이 논문은 2025년 정부(과학기술정보통신부)의 재원으로 정보통신산업진흥원의 지원을 받아 수행된 연구임 (부처협업 기반 AI확산 - AI융합 약관 심사플랫폼 개발 및 실증 과제)

참고 문헌

[1] M. Park, H. Oh, E. Choi, and W. Hwang, “Lrage: Legal retrieval augmented generation evaluation tool,” 2025. [Online]. Available: https://arxiv.org/abs/2504.01840

[2] J.-W. Seo and J. Min, “Optimization of a hybrid rag system for korean legal qa,” Journal of The Korea Society of Computer and Information, Vol. 30, No. 8, pp. 53–63, 2025.

[3] V. Karpukhin, B. Oguz, S. Min, P. S. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, “Dense passage retrieval for open-domain question answering,” EMNLP (1), pp. 6769–6781, 2020.

[4] L. Wang, N. Yang, X. Huang, B. Jiao, L. Yang, D. Jiang, R. Majumder, and F. Wei, “Text embeddings by weakly-supervised contrastive pre-training,” arXiv preprint arXiv:2212.03533, 2022.

[5] O. Khattab and M. Zaharia, “Colbert: Efficient and effective passage search via contextualized late interaction over bert,” Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, pp. 39–48, 2020.

[6] Y. Tang and Y. Yang, “Do we need domain-specific embedding models? an empirical investigation,” arXiv preprint arXiv:2409.18511, 2024.