정적 분석 기반 LLM 자동 프로그램 수리의 품질 저하 사례 분석과 프롬프트 기반 완화 전략

(An Analysis of Quality Deterioration Cases in Static Analysis - Based LLM-Driven Automated Program Repair and Prompt-Based Mitigation Strategies)

손새봄*, 송지영**

(Saebom Son, Jiyoung Song)

요 약

정적 분석 기반 자동 프로그램 수리는 LLM을 활용한 코드 수정 과정에서 평균적으로 높은 품질 향상을 보이지만, 일부 사례에서는 새로운 오류나 경고가 도입되는 품질 저하(quality deterioration) 위험이 발생한다. 이러한 품질 저하는 비율상 소수에 해당하더라도, 세이프티 크리티컬 환경에서는 잠재적으로 심각한 위험 요소가 될 수 있음에도 불구하고, 기존 연구에서는 주로 평균적인 성능 개선에 초점이 맞추어져 체계적으로 분석되지 않았다. 본 연구는 LLM 기반 정적 분석 자동 프로그램 수리 과정에서 발생하는 품질 저하 현상을 주요 분석 대상으로 설정하고, 그 구조적·의미적 원인을 체계적으로 분석하였다. 대규모 C/C++ 코드에 대해 LLaMA 3.1 기반 코드 수리를 수행하고 Cppcheck 결과를 분석한 결과, 반복적으로 관측된 품질 저하 사례를 식별하고 코드 변형 패턴에 기반한 10개의 품질 저하 유형 라벨 체계를 정의하였다. 또한 정의된 품질 저하 유형 라벨을 프롬프트 제약 조건으로 통합하여 재수리 실험을 수행함으로써, 일부 품질 저하 사례에서 정적 분석 기준의 품질 저하가 완화되거나 개선으로 전환될 수 있음을 확인하였다. 이는 LLM 기반 코드 수리 과정에서 발생하는 품질 저하를 단순한 실패 사례가 아닌, 위험 요소를 유형화하고 관리 가능한 문제로 다룰 수 있음을 보여주며, 향후 LLM 기반 정적 분석 자동 프로그램 수리에서 품질 저하 인식 기반 프롬프트 설계 및 제약 조건 설정의 중요성을 시사한다.

■ 중심어 : 대형 언어 모델 ; 자동 프로그램 수리 ; 정적 분석 ; 품질 저하 ; 프롬프트 설계

Abstract

Static-analysis-based automated program repair using large language models (LLMs) generally achieves substantial quality improvements; however, in some cases, it introduces quality deterioration in the form of new errors or warnings. Although such deterioration occurs infrequently, it can pose serious risks in safety-critical environments, yet prior studies have largely focused on average performance improvements and have not systematically examined these cases. This study takes quality deterioration in LLM-based static-analysis-driven automated program repair as a primary analysis target and investigates its structural and semantic causes. Using LLaMA 3.1-based code repair on a large collection of C/C++ code and analyzing quality changes with Cppcheck, we identify recurring deterioration cases and define a taxonomy of ten quality deterioration types based on code transformation patterns. Furthermore, by incorporating this taxonomy into prompt-level constraints and conducting re-repair experiments, we show that quality deterioration can be mitigated or converted into quality improvements in a subset of cases. These results demonstrate that quality deterioration should be treated as a manageable and categorizable risk factor, highlighting the importance of quality-deterioration-aware prompt design for reliable static-analysis-based automated program repair.

■ keywords : Large Language Models ; Automated Program Repair ; Static Analysis ; Quality Deterioration ; Prompt Design

Ⅰ. 서 론

대형 언어 모델(LLM)의 발전은 코드 생성, 요약, 리팩터링, 변환 등 다양한 소프트웨어 공학 작업에서 기존 규칙 기반·통계 기반 기법을 넘어서는 성능을 보여주고 있다. 최근에는 단순한 코드 생성 자동

* 준회원, 한남대학교 컴퓨터공학과
** 종신회원, 한남대학교 컴퓨터공학과
이 논문은 2025학년도 한남대학교 학술연구비 지원에 의하여 연구되었음

접수일자 : 2025년 12월 30일
수정일자 : 2026년 01월 20일
게재확정일 : 2026년 02월 11일
교신저자 : 송지영 e-mail : jysong@hnu.kr

화를 넘어, 대규모 코드의 품질 향상과 유지보수 비용 절감을 목표로 기존 코드의 구조적·품질적 개선을 자동화하려는 연구로 확장되고 있다.

전통적인 자동 프로그램 수리(Automatic Program Repair, APR)는 실패 테스트 케이스를 기반으로 패치의 정당성을 판단한다. 그러나 레거시 시스템이나 임베디드·산업 제어 소프트웨어에서는 충분한 테스트 스위트가 없거나 구축 비용이 매우 높아 테스트 기반 APR의 적용에 한계가 있다.

이러한 한계를 극복하기 위해 최근에는 정적 분석 도구를 품질 오라클로 활용하는 정적 분석 기반 코드 개선 접근이 주목받고 있다. 이 방법은 명시적 버그 정의나 테스트 없이도 코드 품질을 향상시킬 수 있으며, 메모리 안전성이나 경곗값 처리 등 세이프티 크리티컬 속성을 직접적으로 다룰 수 있다. 특히 C/C++과 같은 언어에서는 정적 분석 결과가 코드 안전성 평가의 핵심 지표로 활용된다.

본 논문에서는 LLM을 활용하여 정적 분석 기준에서 품질 저하로 판정된 코드를 수정하는 과정을 코드 수리(code repair)로 정의한다. 이는 코드의 구조적 개선이나 가독성 향상을 목적으로 하는 일반적인 코드 개선(code improvement)과 구분되며, 실패 테스트 케이스를 기반으로 패치의 정당성을 판단하는 기존 자동 프로그램 수리(Automatic Program Repair, APR) 연구와는 품질 오라클의 측면에서 차이를 가진다. 이후 본 논문에서는 제안 방법과 실험을 지칭할 때 “코드 수리”라는 용어를 일관되게 사용한다.

기존 연구들은 LLM 기반 정적 분석 코드 개선 기법이 평균적으로 코드 품질을 향상시킬 수 있음을 보고하고 있다. 그러나 이러한 평균적인 성능 지표만으로는 일부 사례에서 발생하는 새로운 오류 도입이나 정적 분석 기준의 품질 저하 위험을 충분히 설명하기 어렵다.

특히 세이프티 크리티컬 환경에서는 소수의 품질 저하 사례라도 잠재적인 위험 요소가 될 수 있으나, 어떤 코드 변환 특성이 이러한 품질 저하로 이어지는지에 대한 체계적인 분석은 아직 제한적이다. 이에 본 연구는 LLM 기반 정적 분석 코드 개선 과정에서 발생하는 품질 저하 사례를 분석 대상으로 삼아, 그 구조적·의미적 특성을 규명하고, 이를 반영한 프롬프트 설계를 통해 품질 저하 완화 가능성을 검증하는 것을 목적으로 한다.

Ⅱ. 관련 연구

1. 자동 프로그램 수리 (Automatic Program Repair)

자동 프로그램 수리(APR)는 결함이 포함된 프로그램을 자동으로 수정하여 올바른 동작을 수행하도록 만드는 기술로, 대표적으로 GenProg, SPR, Prophet과 같은 접근이 제안되어 왔다. Weimer 등은 유전 알고리즘을 활용하여 테스트 스위트를 통과하는 패치를 탐색하는 GenProg를 제안하였으며, 이는 테스트 기반 APR 연구의 출발점으로 평가된다 [1]. 이후 Long과 Rinard는 조건 기반 패치 탐색을 수행하는 SPR을 통해 패치 탐색 공간을 효율화하였고 [2], Kim 등은 학습 기반 패치 우선순위 모델인 Prophet을 제안하여 패치 품질을 향상시켰다 [3].

APR 연구들은 공통적으로 테스트 스위트를 패치 검증의 핵심 오라클로 활용한다. 그러나 테스트 기반 접근은 테스트의 품질과 커버리지에 크게 의존하며, 충분한 테스트가 존재하지 않는 대규모 산업 코드 베이스나 레거시 시스템에는 적용이 어렵다는 한계를 가진다. 이로 인해 테스트 오라클이 없는 환경에서도 코드 품질을 평가·개선할 수 있는 대안적 접근이 요구되어 왔다.

2. 대형 언어 모델 기반 코드 수리 및 코드 개선 연구

최근에는 대형 언어 모델(LLM)을 활용하여 코드 수리 및 개선을 조건부 코드 생성 문제로 다루는 연구가 활발히 진행되고 있다. Chen 등은 대규모 코드 데이터로 학습된 Codex를 통해 자연어 설명과 코드 컨텍스트를 기반으로 다양한 코드 생성 및 수

정 작업이 가능함을 보였다 [4]. Rozière 등은 Code LLaMA를 제안하여 코드 특화 사전학습을 통해 코드 생성 및 수정 성능을 향상시켰다 [5].

이후 연구들은 범용 LLM을 코드 수리 및 개선 작업에 직접 적용하는 방향으로 확장되었다. LLaMA 계열 모델은 공개 가중치 기반의 대형 언어 모델로서, 코드 및 자연어를 동시에 다룰 수 있는 범용성을 바탕으로 코드 수정, 리팩터링, 결함 수정과 같은 작업에 활용되고 있다 [6]. 또한, DeepSeek 계열 모델은 코드 생성과 추론 중심의 학습 전략을 통해 복잡한 코드 수정 작업에서 안정적인 생성 성능을 보이는 것으로 보고되었다 [7].

이러한 LLM 기반 접근은 기존 APR 기법에 비해 높은 유연성과 범용성을 제공하지만, 과도한 코드 수정, 의미 왜곡, 불필요한 구조 변경과 같은 부작용을 유발할 수 있음이 지적되고 있다. 특히 평균적인 성능 지표만으로는 일부 사례에서 발생하는 품질 저하 현상을 충분히 설명하기 어렵다는 한계가 존재하며, 이는 LLM 기반 코드 개선 기법의 실제 적용 가능성을 검토할 때 고려해야 할 잠재적인 안전성 이슈로 논의되고 있다.

3. 정적 분석 기반 코드 품질 평가와 한계

정적 분석은 프로그램을 실행하지 않고 코드의 잠재적 결함을 탐지하는 기술로, C/C++ 환경에서는 Cppcheck, Clang-Tidy, Coverity 등이 널리 사용된다. Johnson 등은 개발자들이 정적 분석 도구를 사용하는 데 있어 신뢰성과 경고 품질이 중요한 요소임을 분석하였으며 [8], Sadowski 등은 Google의 대규모 코드 베이스에서 정적 분석을 실제로 운영한 경험을 통해 정적 분석의 확장성과 실효성을 보고하였다 [9].

정적 분석 결과를 코드 품질의 지표로 활용한 자동 코드 개선 연구 또한 제안된 바 있으나, 대부분의 연구는 규칙 위반 감소와 같은 평균적인 개선 효과에 초점을 맞추고 있다. 특히 LLM을 활용한 코드 변환 과정에서 새롭게 도입되는 정적 분석 이슈나 정적 분석 기준의 품질 저하 사례를 구조적으로 분석하고 유형화한 연구는 매우 제한적이다.

정적 분석 도구는 테스트 실행 없이도 코드의 잠재적 결함을 탐지할 수 있다는 특성으로 인해 자동 코드 개선 및 프로그램 수리 연구에서 품질 평가 오라클로 활용되어 왔다. 그러나 이러한 평가는 규칙 정의와 분석 보수성에 크게 의존하며, 일부 코드 변환은 기능적으로 문제가 없더라도 새로운 경고를 유발할 수 있다. 이로 인해 평균적인 정적 분석 이슈 감소 지표만으로는 자동 코드 수정 과정에서 발생하는 품질 저하 위험을 충분히 설명하기 어렵다.

Ⅲ. 본 론

1. 실험 개요

본 연구는 정적 분석 기반 코드 수리 환경에서 LLM이 코드 품질에 미치는 영향을 체계적으로 분석하는 것을 목표로 한다. 특히 LLM 기반 코드 수리가 평균적인 품질 향상뿐 아니라 일부 사례에서 품질 저하를 유발할 수 있다는 문제의식에서 출발하여, 이러한 현상이 어떠한 조건과 특성 하에서 발생하는지를 단계적으로 분석하고자 한다.

이를 위해 본 연구는 (1) 대규모 경향 분석, (2) 품질 저하 사례의 원인 유형화, (3) 유형을 반영한 프롬프트 기반 재수리 실험으로 구성된 3단계 실험 프레임워크를 설계하였다. 그림 1은 본 연구에서 사용한 정적 분석 기반 LLM 코드 수리 실험의 전체 파이프라인을 요약하여 보여준다.

1차 실험에서는 대규모 C/C++ 함수 집합을 대상으로 LLM 기반 코드 수리를 수행하고, 수정 전·후 코드에 대해 동일한 정적 분석 도구를 적용하여 코드 품질 변화 양상을 측정하도록 실험을 구성하였다. 이 단계의 목적은 대규모 코드 집합에서 LLM 기반 코드 수리가 전반적으로 보이는 품질 변화 경향을 파악하는 데 있다.

2차 실험에서는 1차 실험 결과에서 품질 저하 사례로 판정된 코드 전체를 분석 대상으로 삼아, 원본 코드와 LLM이 생성한 수정 코드 간의 구조적·의미적 차이를 비교하였다. 이를 통해 코드 수정 과정에

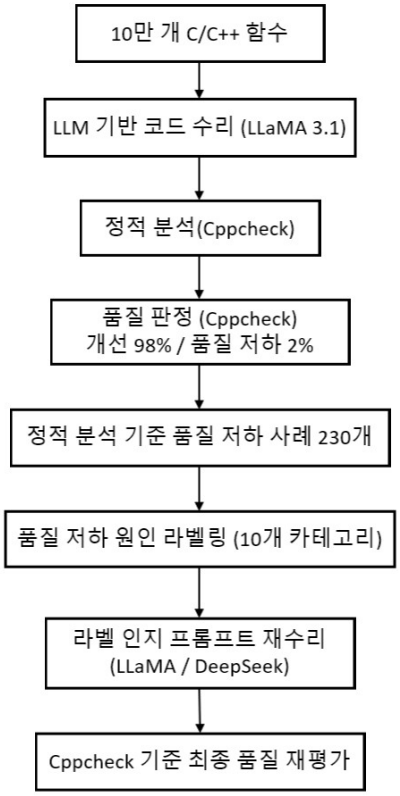

그림 1. 정적 분석 기반 LLM 코드 수리 실험의 전체 파이프라인

서 나타나는 변화를 분석하고, 품질 저하와 연관된 주요 원인을 유형화하기 위한 라벨링을 수행하였다.

3차 실험에서는 2차 실험에서 도출된 품질 저하 유형을 프롬프트 제약 조건으로 반영한 전략적 프롬프트를 설계하고, 동일한 코드 집합에 대해 재수리 실험을 수행하도록 구성하였다. 또한 기본 모델이 수정 코드를 생성하지 못한 일부 사례에 대해서는 보조적으로 다른 LLM 계열 모델을 적용하여, 동일한 프롬프트와 평가 파이프라인 하에서 모델 교체에 따른 생성 가능성과 품질 변화 양상을 분석하도록 하였다. 해당 보조 실험의 세부 설정과 분석 범위는 이후 3.4절에서 상세히 기술한다.

2. 실험 대상 및 환경

본 연구의 실험 대상은 C/C++ 함수 단위 코드이며, 각 함수는 독립적인 분석 단위로 취급하였다. 이는 대규모 코드 베이스 환경에서 정적 분석 기반 코드 수리를 적용하는 현실적인 시나리오를 반영하기 위한 설정이다. 모든 LLM 입력과 출력은 동일한 함수 범위 내에서 이루어졌다.

기본 코드 생성 모델로는 LLaMA 3.1을 사용하였으며, LLaMA가 유의미한 수정 코드를 생성하지 못한 일부 사례에 한해 DeepSeek 계열 모델을 보조적으로 적용하였다. 이때 프롬프트 구성과 평가 파이프라인은 동일하게 유지하고 모델만 교체하여 실험을 수행하였다.

정적 분석 도구로는 Cppcheck v2.18.0(64-bit)을 사용하였으며, --enable=all --inconclusive 옵션을 적용하여 가능한 많은 결함 유형을 탐지하였다. 모든 실험 단계에서 동일한 설정을 유지하였다.

코드 품질 평가는 단일 오류의 존재 여부가 아니라, 원본 코드와 수정 코드에서 탐지된 정적 분석 이슈 집합의 변화에 기반하였다. 즉, 정적 분석 이슈의 수, 유형, 심각도 분포가 수정 전·후 어떻게 변화했는지를 기준으로 분석하였으며, 이 평가는 1차 대규모 실험과 3차 재수리 및 보조 모델 검증 단계에 동일하게 적용되었다.

본 연구에서는 정적 분석 도구 간의 정적 분석 이슈 검출 정확도를 비교하는 것을 연구 목표로 삼지 않았다. LLM 기반 코드 변환 과정에서 기존 코드에 존재하지 않던 정적 분석 경고가 새롭게 도입되거나, 상위 심각도(error, warning)의 정적 분석 이슈가 증가하는지를 중점적으로 분석하고자 하였다.

이러한 분석 관점은 안전 필수 환경에서 평균적인 품질 향상 여부보다, 소수의 코드에서라도 품질 저하가 발생하는 현상이 실제 적용 가능성에 더 큰 영향을 미칠 수 있다는 문제의식에 기반한 것이다.

이러한 목적에 따라 본 연구에서는 단일 정적 분석 도구인 Cppcheck를 일관된 기준으로 사용하고, 정적 분석 기준의 품질 저하 사례 개수를 LLM 기반 코드 수리의 안정성을 평가하기 위한 핵심 지표로 활용하였다.

3. 정적 분석 기반 품질 판정 기준

(1) 품질 판정 기준

LLM이 생성한 코드 수정 결과의 품질은 원본 코드와 수정 코드에 대해 수행한 Cppcheck 결과를 비

교하여 판정하였다. 본 연구는 severity의 상대적 중요도를 반영하여 error > warning > style / performance / portability / information의 우선순위를 설정하고, 상위 심각도(error, warning)의 변화 여부를 중심으로 품질을 판정하였다.

이에 따라 코드 수정 결과는 이슈 제거(Zero-issue), 개선(Improved), 품질 저하(Deteriorated), 동일(Same), 코드 생성 실패(Failure)의 다섯 가지로 분류하였다. 이슈 제거는 수정 코드에서 Cppcheck 이슈가 전혀 탐지되지 않은 경우로, 명확성을 위해 개선과 구분하였다. 개선은 error 또는 warning의 감소나 주요 정적 분석 규칙의 제거가 관측된 경우로 정의하였으며, 하위 심각도 범주의 경미한 증가는 허용하였다. 반대로 품질 저하는 새로운 error 또는 warning의 도입이나 기존 상위 심각도 정적 분석 이슈의 증가가 발생한 경우로 정의하였다. 동일은 error 및 warning 수준에서 변화가 없는 경우를 의미하며, 코드 생성 실패는 비교 가능한 수정 코드가 생성되지 않은 경우를 포함한다.

이 판정 기준은 1차 실험에서 품질 저하 사례를 식별하는 데 사용되었으며, 3차 재수리 및 보조 모델 검증 단계에서도 동일하게 적용되었다. 반면, 2차 실험에서는 정적 분석 도구를 사용하지 않고 코드 수정의 구조적·의미적 변화에 대한 정성적 분석만을 수행하였다.

(2) 품질 저하 유형 라벨 정의

품질 저하로 분류된 코드들에 대해서는 LLM 기반 코드 수정 과정에서 나타난 구조적·의미적 변화를 기준으로 총 10개의 품질 저하 유형 라벨을 정의하였다. 각 라벨은 단순한 현상 기술이 아니라, 정적 분석 기준의 품질 저하와 직접적으로 연관된 코드 변환 특성을 판정 규칙으로 삼아 구성되었다.

예를 들어, 불필요한 조건문 삽입(Unnecessary Conditional Insertion) 라벨은 기존 코드에는 존재하지 않던 조건 분기가 새롭게 추가되면서, 해당 분기 내 변수 초기화 누락이나 경계 조건 오류로 인해 새로운 error 또는 warning이 발생한 경우에 부여하였다. 의미 변경에 따른 경고 유발(Semantic Change Inducing Warning) 라벨은 변수 타입, 연산 순서, 반환 조건 등의 변경으로 인해 프로그램 의미가 달라지면서 정적 분석 경고가 새롭게 탐지된 사례에 적용하였다.

이와 같이 각 품질 저하 유형 라벨은 (1) 코드 변환의 형태적 특징과, (2) 그로 인해 유발된 정적 분석 경고의 성격을 함께 고려하여 정의되었으며, 대표적인 사례를 통해 라벨 의미와 적용 기준의 명확성을 확보하였다.

(3) 품질 저하 유형 라벨링 절차 및 신뢰도

품질 저하 유형 라벨링은 2차 실험에서 식별된 품질 저하 사례를 대상으로 수행되었다. 라벨링 과정의 주관성을 최소화하고 신뢰도를 확보하기 위해, 연구자는 사전에 정의된 라벨 설명과 판정 규칙을 기반으로 각 코드 사례에 대해 품질 저하 유형 라벨을 부여하였다. 라벨 부여 과정에서는 정적 분석 결과와 원본 코드 대비 수정 코드의 구조적·의미적 변화를 함께 고려하였다.

라벨링 결과에 대해서는 지도교수의 검토를 통해 라벨 정의의 타당성과 적용의 일관성을 점검하였다. 특히 해석의 여지가 있는 경계 사례에 대해서는 코드 수정의 의도와 실제 변화 양상, 그리고 해당 변경으로 인해 유발된 정적 분석 경고의 성격을 중심으로 논의를 거쳐 라벨을 조정하거나 확정하였다. 명확한 판단이 어려운 경우에는 보다 보수적인 기준을 적용하여 라벨을 부여하였다.

이와 같은 검토 및 합의 절차를 통해, 품질 저하 유형 라벨 체계가 단일 연구자의 주관적 판단에 과도하게 의존하지 않도록 하였으며, 이후 수행된 라벨 기반 프롬프트 설계 및 재수리 실험에서 활용 가능한 신뢰성 있는 분석 결과를 확보하였다.

4. 라벨 기반 프롬프트 설계

3차 실험에서는 2차 실험에서 도출된 품질 저하 유형 라벨을 기반으로, LLM이 코드 수정 과정에서

품질 저하를 유발한 변환 패턴을 회피하도록 유도하는 라벨 기반 프롬프트를 설계하였다. 본 연구의 목적은 단순히 정적 분석 경고를 제거하는 것이 아니라, 기존 코드의 외부 동작과 구조적 안정성을 보존하면서 품질 저하 위험을 최소화하는 방향으로 코드 수리를 수행하는 데 있다.

이를 위해 프롬프트는 모든 실험에서 공통적으로 적용되는 기본 제약과, 품질 저하 유형 라벨에 따라 선택적으로 적용되는 라벨 기반 제약으로 구성하였다. 기본 제약은 외부 동작 및 인터페이스 보존, 함수 시그니처 유지, 최소 수정 원칙, 컴파일 가능성과 안전성 우선, 그리고 코드만 출력하도록 하는 출력 형식 제약을 포함한다. 이러한 기본 제약은 모델이 과도한 구조 변경이나 불필요한 리팩터링을 수행하지 않도록 제어하는 역할을 한다.

라벨 기반 제약은 2차 실험에서 관측된 주요 품질 저하 유형을 프롬프트 수준의 수정 회피 규칙으로 변환하여 구성하였다. 예를 들어, 함수 시그니처 변경으로 인한 품질 저하 유형의 경우, 함수 이름, 매개변수, 반환 타입을 변경하지 않도록 명시적인 제약을 부여하였다. 제어 흐름 왜곡 유형에 대해서는 모든 실행 경로에서 정의된 동작을 유지하도록 요구하며, 불필요한 분기 추가나 기존 제어 흐름의 재구성을 제한하였다. 또한 의미 누락으로 인한 품질 저하 유형에 대해서는 기존 코드의 의도를 유지하면서 최소 범위 내에서 누락된 로직만을 보완하도록 유도하였다.

여러 라벨이 동시에 적용되는 경우에는 사전에 정의된 우선순위에 따라 제약을 결합하였다. 이를 통해 함수 시그니처 보존과 같은 상위 위험 요인이 경미한 스타일 수정 요구보다 우선적으로 고려되도록 하였으며, 상충 가능성이 있는 제약 간의 충돌을 방지하였다. 이러한 제약 결합 방식은 모든 실험 모델에서 동일하게 유지하였으며, 모델별 비교 실험에서는 프롬프트의 구조와 제약 내용은 고정한 상태에서 모델별 입력 형식에 맞게 문장 표현만을 최소한으로 조정하였다. 이를 통해 프롬프트 설계 자체의 효과와 모델 의존성을 분리하여 분석하고자 하였다.

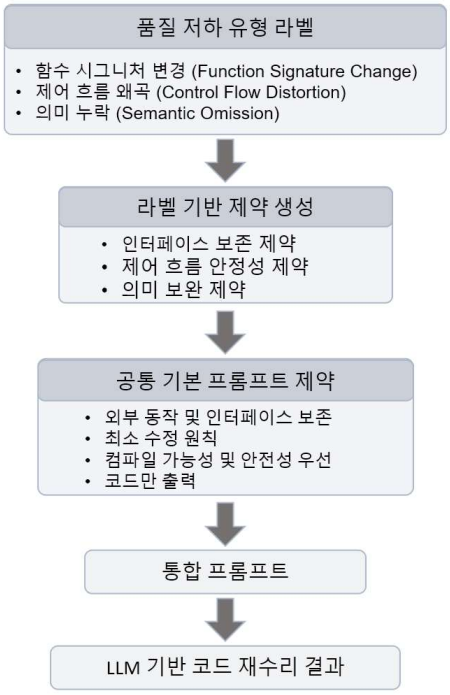

그림 2. 라벨 기반 프롬프트 설계의 전체 구조

그림 2는 라벨 기반 프롬프트 설계의 전체 구조를 예시적으로 나타낸다. 품질 저하 유형 라벨은 대응되는 라벨 기반 제약으로 변환되며, 이러한 제약은 공통 기본 제약과 결합되어 통합 프롬프트를 구성한다. 생성된 통합 프롬프트는 LLM 기반 코드 재수리 단계의 입력으로 사용된다.

5. 품질 저하 사례 분석 및 보조 모델 검증

2차 실험에서는 1차 실험 결과에서 정적 분석 기준 품질 저하로 판정된 코드 전체를 대상으로, 원본 코드와 LLaMA 3.1이 생성한 수정 코드 간의 구조적·의미적 차이를 분석하였다. 이 단계의 목적은 품질 저하 여부를 재판정하는 것이 아니라, LLM 기반 코드 수정 과정에서 품질 저하를 유발한 원인을 규명하고 반복적으로 관측되는 변환 패턴을 유형화하는 데 있다. 이에 따라 본 단계에서는 정적 분석 도구를 사용하지 않고, 코드 수정의 구조적·의미적 변화에 대한 정성적 분석과 라벨링을 수행하였다.

본 연구에서 정의한 품질 저하 유형 라벨은 과도한 코드 확장, 구현 삭제 또는 의도 왜곡, 제어 흐름 변경과 같은 구조적 변화와 문법·의존성·타입 안전성 훼손과 같은 의미적 문제를 중심으로 구성되었

다. 예를 들어, function_overly_long은 불필요한 코드 확장으로 인해 함수 길이가 비정상적으로 증가한 경우를 의미하며, function_removed_or_intent_distorted는 기존 구현이 삭제되거나 코드의 의도가 왜곡된 사례를 나타낸다. language_mismatch는 C/C++ 문법과 호환되지 않는 표현이 삽입된 경우를 지칭하며, header_or_dependency_issue는 필수 헤더 또는 의존성이 누락되었거나 잘못 변경된 사례를 포함한다. macro_or_build_config_issue는 매크로나 빌드 구성 요소가 손상된 경우를 의미하고, api_signature_change는 함수 서명(매개변수·반환 타입 등)이 불필요하게 변경된 사례를 포괄한다. control_flow_rewiring은 조건문이나 반복문의 구조가 재구성되거나 흐름이 왜곡된 사례를 지칭하며, init_or_type_safety_regress는 변수 초기화 누락 또는 타입 안전성과 관련된 문제가 도입된 경우를 뜻한다. format_or_style_only는 의미적 변화 없이 형식 또는 스타일만 불필요하게 수정된 사례를 의미하고, 마지막으로 other는 상기 기준에 속하지 않는 기타 품질 저하 패턴을 포괄한다.

품질 저하 원인은 코드 구조 및 의미 변화 관점에서 분석되었으며, 반복적으로 관측된 수정 패턴에 기반해 총 10개 유형의 라벨 체계를 설계하였다. 라벨 체계는 품질 저하 사례에 대한 탐색적 분석을 통해 연구자가 수작업으로 정의한 것으로, 품질 저하 현상을 설명하기 위한 해석적 분류 기준으로 사용되었다. 각 사례에는 가장 지배적인 원인에 해당하는 하나의 라벨이 할당되었으며(다중 라벨링은 적용하지 않음), 분류는 코드 수정 자체의 구조적·의미적 특성에 따라 수행되었다.

특히 과도한 코드 삭제 또는 의도 왜곡에 해당하는 라벨 범주에서는, LLM이 실제 구현을 제거하고 해당 부분을 주석(예: “이하 동일”, “기존 로직 유지”)으로 대체하는 comment elision 패턴이 다수 관측되었다. 본 연구는 이러한 패턴을 실질적인 개선이 아닌 수리 회피에 해당하는 실패 하위 유형으로 간주하였으며, 독립적인 라벨로 분리하지 않고 기존 라벨 범주의 하위 패턴으로 처리하였다.

LLaMA가 비교 가능한 수정 코드를 생성하지 못한 일부 사례에 대해서는 DeepSeek 계열 모델을 보조적으로 적용하였다. 본 보조 실험은 모델 간 성능 비교가 아니라, 동일한 프롬프트와 평가 파이프라인 하에서 모델 교체 시 생성 가능성과 정적 분석 결과의 변화를 확인하기 위한 검증 단계로 수행되었으며, 결과 해석은 제한된 사례를 대상으로 이루어졌다.

Ⅳ. 실험 결과

본 장에서는 1차 실험에서 관측된 정적 분석 기준 코드 품질 변화의 전체적인 경향을 먼저 제시한 후, 2차 실험을 통해 식별된 품질 저하 사례의 라벨 분포를 분석한다. 이후 3차 실험에서 적용한 전략적 프롬프트 및 보조 모델 결과를 비교·해석한다.

1. 1차 실험 결과: 정적 분석 기준 품질 변화

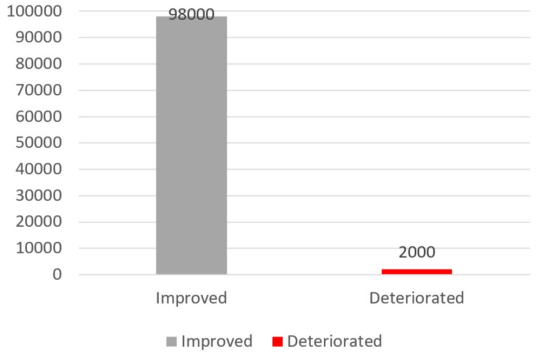

그림 3. 1차 실험의 정적 분석 기준 코드 품질 변화 분포

그림 3은 1차 실험에서 LLM 기반 코드 수리 수행 후, 정적 분석 기준에 따라 관측된 코드 품질 변화의 전체 분포를 나타낸다. 총 10만 개의 C/C++ 함수에 대해 실험을 수행한 결과, 약 98%에 해당하는 코드에서 정적 분석 기준의 품질 개선이 관측되었으며, 약 2%에 해당하는 코드에서는 새로운 정적 분석 이슈 도입 또는 상위 심각도 경고 증가로 인해 품질 저하가 발생하였다.

이 결과는 LLM 기반 코드 수리가 평균적인 관점

에서는 정적 분석 기준의 코드 품질을 효과적으로 향상시킬 수 있음을 보여준다. 그러나 전체 비율로는 소수에 해당하더라도, 약 2%의 코드에서 관측된 품질 저하 사례는 세이프티 크리티컬 환경이나 대규모 자동 적용 시 잠재적인 안전성 위험으로 이어질 수 있다. 이는 단순한 평균 성능 지표만으로는 LLM 기반 코드 수리의 안정성을 충분히 설명하기 어렵다는 점을 시사한다.

이에 본 연구에서는 이후 2차 실험에서 해당 품질 저하 사례를 집중적으로 분석하여, 코드 수정 과정에서 나타난 구조적·의미적 변화 패턴을 유형화하고, 3차 실험에서는 이를 반영한 라벨 기반 프롬프트 설계를 통해 품질 저하 완화 가능성을 검증하고자 한다.

2. 품질 저하 사례 라벨 분포 분석

2차 실험에서는 Cppcheck 기준으로 식별된 총 230개의 품질 저하 사례를 대상으로, LLM 기반 코드 수정 과정에서 나타난 변경 내용을 분석하였다. 이를 위해 코드 수정 과정에서 관측되는 구조적·의미적 변화 유형을 포괄적으로 반영할 수 있도록, 총 10개의 품질 저하 유형 라벨 체계를 사전에 정의하고 각 사례를 이 중 하나의 라벨로 분류하였다.

그림 4는 정적 분석 기준으로 품질 저하로 판정된 230개 사례에 대해, 코드 수정 과정에서 관측된 구조적·의미적 변화에 기반하여 분류한 품질 저하 유형 라벨의 분포를 나타낸다. 분포를 살펴보면, function_overly_long, function_removed_or_intent_distorted, control_flow_rewiring, init_or_type_safety_regress와 같은 일부 유형에 사례가 집중되어 있음을 확인할 수 있다. 이는 LLM 기반 코드 수정 과정에서 발생하는 품질 저하가 무작위적인 오류라기보다는, 특정 코드 변환 패턴과 밀접하게 연관되어 나타나는 경향이 있음을 시사한다.

한편, macro_or_build_config_issue, format_or_style_only와 같이 상대적으로 사례 수가 적은 라벨도 관측되었는데, 이는 해당 유형의 품질 저하가 발생 빈도는 낮으나 실제 코드 수정 과정에서 배제할 수 없는 오류 유형임을 의미한다. 본 연구에서는 이러한 저빈도 유형까지 포함하여 라벨 체계를 구성함으로써, 품질 저하 현상을 보다 포괄적으로 분석하고자 하였다. 모든 라벨의 사례 수를 합산한 총 분석 대상은 230개로, 표 1 및 그림 4의 분포는 동일한 데이터를 기반으로 한다.

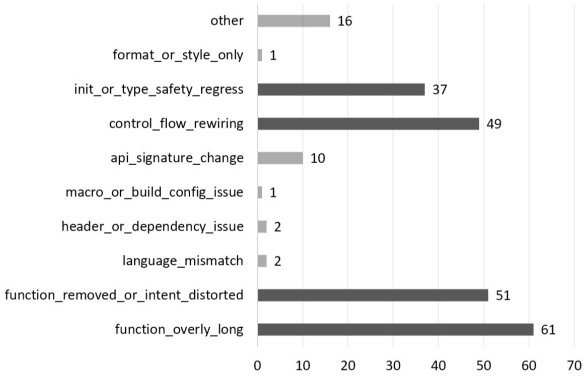

그림 4. 2차 실험에서 식별된 230개 품질 저하 사례의 라벨 분포

표 1은 230개 품질 저하 사례에 대해 적용된 라벨별 분포를 개수 및 비율 기준으로 정리한 결과를 나타낸다. 분포 결과를 살펴보면, 품질 저하 사례는 다양한 유형에 균등하게 분포되기보다는 일부 특정 유형에 집중되는 경향을 보인다. 이 중 function_overly_long(26.52%), function_removed_or_intent_distorted(22.17%), control_flow_rewiring(21.30%), init_or_type_safety_regress(16.09%)의 네 가지 라벨이 전체 사례의 상당 부분을 차지함을 확인할 수 있다.

한편, function_removed_or_intent_distorted 범주에서는 기존 구현을 주석 처리로 대체하는 형태의 변경(comment elision)이 23건 관찰되었다. 본 연구에서는 해당 패턴을 독립적인 라벨로 분리하지 않고, 의미 보존 실패라는 공통 특성을 고려하여 해당 라벨의 하위 패턴으로 포함하여 분석하였다. 이러한 라벨 분포 결과는 이후 3차 실험에서 수행된 전략적 프롬프트 설계 과정에서, 품질 저하 유형별 대응 전략을 구성하기 위한 근거로 활용되었다.

표 1. 2차 실험에서 식별된 230개 품질 저하 사례의 라벨 분포

ID	Label name	Count	Ratio
1	function_overly_long	61	26.52%
2	function_removed_or_intent_distorted	51	22.17%
3	language_mismatch	2	0.87%
4	header_or_dependency_issue	2	0.87%
5	macro_or_build_config_issue	1	0.43%
6	api_signature_change	10	4.35%
7	control_flow_rewiring	49	21.30%
8	init_or_type_safety_regress	37	16.09%
9	format_or_style_only	1	0.43%
10	other	16	6.96%

3. 코드 생성 성공 여부 분석

3차 실험에서는 Cppcheck 기준으로 품질 저하 사례로 식별된 230개 코드를 대상으로, 전략적 프롬프트를 적용하여 LLaMA 3.1 기반 코드 생성을 수행하였다. 본 절에서는 코드 생성 성공 여부를, 수정된 코드 산출물이 원본 코드와 비교 가능한 형태로 생성되었는지를 기준으로 판단하였다.

그 결과, LLaMA 3.1은 230개 사례 중 194개에 대해 비교 가능한 수정 코드를 생성하였으며, 나머지 36개의 사례에서는 이러한 산출물을 생성하지 못하였다. 표 2는 LLaMA 3.1의 코드 생성 성공 및 실패 사례 수를 요약하여 제시한다.

표 2. 3차 실험에서 LLaMA 3.1 코드 생성 결과 (품질 저하 사례 230개 대상)

구분	개수
코드 생성 성공	194
코드 생성 실패	36

LLaMA 3.1에서 코드 생성에 실패한 36개 사례에 대해, 동일한 프롬프트와 평가 파이프라인을 유지한 상태에서 DeepSeek 모델을 사용하여 코드 생성을 재시도하였다. 표 3은 DeepSeek을 이용한 재시도 결과를 요약한다. 그 결과, DeepSeek은 36개 사례 중 8개에서 비교 가능한 수정 코드를 생성하였으며, 나머지 28개 사례는 생성 실패로 분류되었다.

표 3. 3차 실험에서 DeepSeek 코드 생성 결과 (LLaMA 3.1 생성 실패 사례 36개 대상)

구분	개수
코드 생성 성공	8
코드 생성 실패	28

표 2와 표 3의 결과를 종합하면, DeepSeek 적용은 LLaMA 3.1에서 코드 생성에 실패한 일부 사례에 대해 비교 가능한 수정 코드 산출 가능성을 보완할 수 있음을 확인하였다.

다만 코드 생성 성공은 분석 기준상 품질 개선으로 직결되지 않으므로, 생성 결과의 품질 향상 여부는 별도의 정적 분석 절차를 통해 판단할 필요가 있다. 이에 따라 다음 절에서는 4.1절에서 제시한 품질 저하 유형 분포를 기반으로, Cppcheck 기준에 따른 최종 품질 판정 결과를 모델별로 비교·분석한다.

4. Cppcheck 기반 품질 판정 결과

본 절에서는 원본 코드와 수정 코드의 Cppcheck 분석 결과를 비교하여, 이슈 제거(Zero-issue), 개선, 동일, 품질 저하, 코드 생성 실패의 다섯 가지 범주로 구분한 최종 품질 평가 결과를 제시한다. 품질 판정은 Cppcheck 이슈의 총 개수와 심각도(error > warning > 기타) 변화에 기반하여 수행되었다.

여기서 “코드 생성 실패”는 비교 가능한 수정 코드가 생성되지 않은 경우를 의미하며(예: 빈 출력 또는 원본과 비교 불가능한 손상된 출력 등), “이슈 제거(Zero-issue)”는 수정 코드에서 Cppcheck 이슈가 관측되지 않은 경우를 의미한다. 또한 “동일”은 수정 코드와 원본 코드 간 Cppcheck 이슈 수 및 심각도 변화가 관측되지 않은 경우를 의미한다. 한편, DeepSeek 결과는

LLaMA 3.1 코드 생성 실패 36개 사례에 대한 제한된 서브셋 기반의 보조 실험 결과로 해석되어야 한다.

(1) LLaMA 3.1 생성 코드 품질 판정 결과 (230개, Cppcheck 기반)

표 4는 품질 저하 사례 230개에 대해 LLaMA 3.1이 생성한 수정 코드의 최종 품질 판정 결과를 요약한다. LLaMA 3.1은 전체 230개 사례 중 57개를 개선으로, 1개를 이슈 제거(Zero-issue) 상태로 생성하였으며, 4개는 동일로 분류되었다. 반면, 132개의 사례에서는 Cppcheck 기준 품질 저하가 관측되었고, 36개 사례는 비교 가능한 수정 코드가 생성되지 않아 코드 생성 실패로 분류되었다.

표 4. LLaMA 3.1 재수리 결과 및 품질 판정 결과 (230개 대상, Cppcheck 기반)

결과	개수
이슈 제거(Zero-issue)	1
개선	57
동일	4
품질 저하	132
코드 생성 실패	36

표 4의 결과는 코드 생성 성공과 정적 분석 기준의 품질 개선이 동일한 개념이 아님을 명확히 보여준다. 즉, 수정 코드가 생성된 경우(194개)에도 불구하고, 그 중 상당수는 정적 분석 기준에서 품질 저하로 판정되었으며, 이는 LLM 기반 코드 수정 과정에서 품질 저하를 제어하기 위한 추가적인 제약 조건 및 전략적 프롬프트 설계의 필요성을 시사한다.

표 4의 전체 품질 판정 결과를 라벨별로 분석한 결과, 품질 개선 또는 완화가 관측되는 양상은 품질 저하 유형 라벨에 따라 상이하게 나타났다. function_overly_long 및 function_removed_or_intent_distorted와 같은 라벨에서는 일부 사례에서 개선 또는 이슈 제거(Zero-issue)가 관측되었는데, 이는 함수 길이 조정이나 불필요한 코드 제거와 같이 비교적 국소적인 수정이 정적 분석 기준의 품질 개선으로 이어질 가능성이 있음을 시사한다. 특히 해당 라벨들은 코드 구조의 전면적인 변경을 요구하지 않는 경우가 많아, LLaMA 3.1 기반 코드 수정 과정에서 제한적인 범위 내의 수정이 상대적으로 안정적으로 수행된 것으로 해석할 수 있다.

반면, control_flow_rewiring 및 init_or_type_safety_regress와 같이 제어 흐름 변경이나 타입·초기화 관련 변형을 포함하는 라벨에서는 품질 저하로 판정된 사례의 비중이 상대적으로 높게 나타났다. 이러한 라벨들은 코드 전반의 실행 흐름이나 안전성과 직접적으로 연관되어 있어, LLM 기반 코드 생성 과정에서 미세한 오류가 정적 분석 기준의 품질 저하로 이어질 가능성이 높은 유형으로 볼 수 있다.

본 결과는 품질 저하 완화 가능성이 라벨 유형에 따라 차이를 보임을 보여주며, 향후 전략적 프롬프트 설계 및 제약 조건 설정 시 라벨 특성을 고려한 접근이 필요함을 시사한다.

(2) DeepSeek 보조 모델 품질 판정 결과 (LLaMA 3.1 실패 36개)

표 5는 LLaMA 3.1 생성 실패 36개 사례에 대해, 동일한 프롬프트와 평가 파이프라인을 유지한 상태에서 DeepSeek 모델을 적용하여 코드 생성을 수행한 결과의 품질 판정 분포를 제시한다. 해당 실험은 LLaMA 3.1 단일 모델 실험에서 관측된 생성 실패를 부분적으로 보완할 수 있는지를 확인하기 위한 보조적 시도로 수행되었다.

그 결과, DeepSeek은 전체 36개 사례 중 3개를 개선으로 분류하였으나, 5개는 품질 저하로 판정되었고, 28개는 비교 가능한 수정 코드가 생성되지 않아 코드 생성 실패로 분류되었다. 이는 DeepSeek 적용이 일부 사례에서 생성 실패를 보완할 수 있음을 보여주는 한편, 정적 분석 기준의 개선이 일관되게 관측되지는 않음을 시사한다.

표 5. DeepSeek 보조 모델 품질 판정 결과 (LLaMA 3.1 실패 36개 대상, Cppcheck 기반)

결과	개수
개선	3
품질 저하	5
코드 생성 실패	28

표 5의 결과는 DeepSeek이 일부 사례에서 비교 가능한 수정 코드 생성에 성공하였음에도 불구하고, 해당 성공 사례가 곧바로 정적 분석 기준의 품질 개선으로 이어지지는 않음을 보여준다. 또한 본 결과는 LLaMA 3.1 생성 실패 사례라는 제한된 서브 셋에서 측정된 값이므로, DeepSeek의 절대적인 성능을 일반화하기보다는 보조 검증 관점에서 해석하는 것이 적절하다.

V. 논 의

본 연구의 실험 결과는 LLM 기반 코드 수정에서 코드 생성의 성공 여부와 정적 분석 기준의 품질 개선이 반드시 일치하지 않음을 보여준다. LLaMA 3.1과 DeepSeek 모두 일정 수준의 코드 생성에는 성공하였으나, 상당수 사례에서 Cppcheck 기준의 품질 저하 또는 코드 생성 실패가 반복적으로 관측되었으며, 이는 단순한 생성 성공률만으로 재수리 성능을 평가하기 어렵다는 점을 시사한다.

특히, 품질 판정 결과를 품질 저하 유형 라벨 관점에서 분석한 결과, 품질 개선 또는 완화 가능성은 라벨 유형에 따라 뚜렷한 차이를 보였다. function_overly_long이나 function_removed_or_intent_distorted와 같이 함수 길이 조정이나 불필요한 코드 제거가 중심이 되는 라벨에서는 일부 사례에서 품질 개선 또는 이슈 제거가 관측되었는데, 이는 비교적 국소적인 수정이 정적 분석 기준의 개선으로 이어질 가능성이 있음을 보여준다. 반면, control_flow_rewiring이나 init_or_type_safety_regress와 같이 제어 흐름 변경이나 타입·초기화 관련 변형을 포함하는 라벨에서는 품질 저하가 지속적으로 관측되어, 해당 유형의 수정이 LLM 기반 코드 생성 과정에서 특히 어려운 문제임을 시사한다.

이러한 경향은 본 연구에서 함수 단위 코드만을 입력으로 사용한 실험 설정과도 밀접하게 연관된다. 상위 컨텍스트 정보(예: #include 의존성, 타입 정의, 매크로)가 제거된 상태에서 코드 수정이 이루어짐에 따라, 문법적으로는 타당하더라도 정적 분석 관점에서는 타입 불일치, 의존성 누락, 제어 흐름 왜곡과 같은 새로운 이슈가 발생할 가능성이 높아진다. 실제로 품질 저하로 분류된 사례들에서도 이러한 구조적·의미적 변화가 주요 원인으로 확인되었다.

또한 정적 분석 도구를 품질 오라클로 사용하는 접근의 특성상, Cppcheck의 보수적인 판정 기준 역시 결과 해석에 영향을 미친다. 코드 구조 단순화나 스타일 변경과 같은 수정이 기능적으로는 문제가 없더라도, 정적 분석 규칙에 따라 품질 저하로 판정될 수 있으며, 이는 테스트 기반 APR과 구별되는 정적 분석 기반 접근의 한계로 볼 수 있다.

DeepSeek 보조 모델 실험은 단일 LLaMA 3.1 모델이 수정 코드를 생성하지 못한 일부 사례에서 생성 가능성을 보완할 수 있음을 보여주었으나, 모델 교체만으로 정적 분석 기준의 품질 개선이 일관되게 달성되지는 않았다. 품질 저하 문제가 모델 성능 자체보다는, 생성 과정에서의 제약 부재와 품질 저하 인식 부족과 더 밀접하게 관련됨을 시사한다.

LLM 기반 정적 분석 APR은 평균적인 개선 성능이나 단순한 성공률 지표만으로는 세이프티 크리티컬 환경에 직접 적용하기 어렵다. 그러나 품질 저하가 무작위적으로 발생하는 것이 아니라 특정 품질 저하 유형 라벨에 집중되어 나타남을 확인함으로써, 향후 품질 저하 인식 기반 프롬프트 설계나 라벨 특성을 반영한 다단계 검증 파이프라인과 결합할 경우 조건부 실용 가능성이 있음을 보여준다.

VI. 결 론

본 논문은 정적 분석 도구를 품질 오라클로 활용한 LLM 기반 코드 수정 접근을 대상으로, 대규모 C/C++ 코드 실험을 통해 품질 저하 현상을 체계적으로 분석하였다. 실험 결과 전반적으로 정적 분석 기준의 품질 개선이 관측되었으나, 일부 코드에서는 지

속적인 품질 저하가 발생함을 확인하였다. 이는 코드 생성의 성공 여부와 정적 분석 기준 품질 개선이 반드시 일치하지 않음을 의미하며, 특히 세이프티 크리티컬 환경에서 평균 성능 지표만으로 LLM 기반 코드 수리를 적용하는 데 한계가 있음을 시사한다. 나아가 품질 저하 사례를 단순 예외가 아닌 구조적·의미적 코드 변환 패턴에 기반한 유형 라벨 체계로 정형화하고, 이를 재수리 프롬프트 설계 및 평가에 연계함으로써 품질 저하가 특정 변환 유형에 집중됨을 실험적으로 규명하였다.

또한 품질 저하 유형에 따라 재수리 난이도와 개선 가능성이 상이함을 확인하였으며, 함수 길이 조정이나 불필요 코드 제거와 같은 국소 수정은 비교적 안정적인 개선으로 이어지는 반면, 제어 흐름 변경이나 타입·초기화 변형은 품질 개선이 어려운 유형으로 나타났다. 이는 향후 LLM 기반 코드 수리 시스템 설계 시 품질 저하 유형별 차별화된 프롬프트 전략과 제약 조건 적용의 필요성을 시사한다. 본 연구는 LLM 기반 정적 분석 APR의 한계를 실증적으로 규명하고, 품질 저하를 관리 가능한 문제로 다룰 수 있는 분석 틀을 제시한다. 향후에는 풍부한 코드 컨텍스트 활용, 다중 정적 분석 도구 결합, 정적 분석 결과를 생성 과정에 직접 반영하는 제약 기반 코드 생성 기법을 통해 신뢰성과 실용성 향상이 기대된다.

REFERENCES

[1] W. Weimer, T. Nguyen, C. Le Goues, S. Forrest, “Automatically finding patches using genetic programming,” *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 364 - 374, 2009.

[2] F. Long, M. Rinard, “Staged program repair with condition synthesis,” *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 166 - 178, 2015.

[3] D. Kim, J. Nam, J. Song, S. Kim, “Automatic patch generation learned from human-written patches,” *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 802 - 811, 2013.

[4] M. Chen et al., “Evaluating large language models trained on code,” arXiv Report, arXiv:2107.03374, 2021.

[5] B. Rozière et al., “Code LLaMA: Open foundation models for code,” arXiv Report, arXiv:2308.12950, 2023.

[6] T. Touvron et al., “LLaMA: Open and Efficient Foundation Language Models,” arXiv:2302.13971, 2023.

[7] DeepSeek-AI, “DeepSeek-Coder: When the Large Language Model Meets Programming,” arXiv:2310.08739, 2023.

[8] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why Don’t Software Developers Use Static Analysis Tools to Find Bugs?,” *Proc. 35th International Conference on Software Engineering (ICSE)*, pp. 672 - 681, 2013.

[9] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, C. Jaspan, “Lessons from Building Static Analysis Tools at Google,” *Communications of the ACM*, vol. 61, no. 4, pp. 58 - 66, 2018.

저 자 소 개

손새봄(준회원)

2026년 한남대학교 컴퓨터공학과 학사 졸업 예정.

<주관심분야 : 소프트웨어 테스팅, 소프트웨어 품질보증, 소프트웨어 테스트 자동화>

송지영(종신회원)

2014년 이화여자대학교 컴퓨터공학과 학사 졸업.
2016년 한국과학기술원 전산학부 석사 졸업.
2022년 한국과학기술원 전산학부 학과 박사 졸업.

<주관심분야 : 소프트웨어공학, 소프트웨어 테스팅, 소프트웨어 모델 검증, 프로젝트 관리, 시스템 오브 시스템즈, IoT, CPS>