Pusan National University
Computer Science and Engineering
Technical Report 2025

LLM을 활용한 SW refactoring

저자1(202055518 김병현)
저자2(202014511 박준하)

지도교수 채흥석

목 차

1 서론

1.1 연구 배경

LLM(Large Language Model)은 최근 몇 년간 자연어 처리와 코드 생성 분야에서 혁신적인 발전을 이루었다. 방대한 데이터를 학습하고 다양한 기술의 적용으로 LLM은 프로그래밍 언어를 이해하고 코드를 생성 및 수정하는 능력이 향상되어 최근 주목을 받고 있다.

SW개발에서 refactoring은 SW의 동작을 변경하지 않고 내부 구조를 개선하는 필수적인 과정이다. refactoring의 궁극적인 목적은 코드의 기능은 그대로 유지하면서, 이해용이성, 유지보수성, 테스트 용이성 등의 다양한 측면에서 품질을 향상시키는 데 있다. 이는 장기적으로 기술 부채를 줄이고, 개발 생산성과 소프트웨어 안정성을 높이는 데 기여한다.

기존의 refactoring 방식은 높은 비용이 발생하였다. 최근 LLM을 활용하여 refactoring을 하는 연구는 주로 Java, Python, JavaScript/TypeScript와 같은 현대적인 프로그래밍 언어에 초점을 맞추고 있다. 이러한 언어에서 LLM을 활용한 refactoring 작업은 높은 성능을 보여주고 있다.

그러나 C언어는 여전히 임베디드 시스템, 운영체제, 시스템 소프트웨어 등 제한된 자원 환경에서 널리 사용되는 중요한 언어임에도 불구하고, 기존 연구에서는 상대적으로 덜 다루어졌다. 이에 따라 C언어를 대상으로 LLM을 활용한 refactoring을 탐구하고자 한다.

1.2 연구 목표

LLM이 대상 프로그래밍 언어(C언어)로 구성된 SW에 대한 refactoring을 효과적으로 수행하는지 파악하고, LLM을 활용하여 대상 프로그래밍 언어로 이루어진 SW를 리팩토링하는 agent를 개발한다. 궁극적으로 refactoring에 드는 비용을 절감한다.

2 refactoring 대상 언어

2.1 C 언어를 refactoring 대상으로 선택하는 타당한 근거

1. 오래된 코드베이스가 많음 (Legacy Code의 대표)
    - C는 1970년대부터 널리 쓰인 언어로, 산업현장 및 시스템 개발에 사용된 수많은 레거시 코드가 존재한다.
    - 이러한 코드들은 refactoring 없이 유지보수가 매우 어렵고 위험하다.
2. 직접적인 메모리 관리로 인해 코드 품질이 중요

- 포인터, 메모리 할당/해제 등 메모리 취약성을 유발하기 쉬운 구조이기 때문에, 이해용이성 좋고 안정적인 코드 작성이 중요하다.
- LLM 기반 refactoring이 오류 가능성을 줄이는 데 기여할 수 있다.

3. refactoring 도구나 지원이 상대적으로 부족
   - Python, Java 등에 비해 현대적인 IDE나 refactoring 도구의 지원이 적다.
   - LLM을 이용한 refactoring 자동화는 C 개발자에게 실질적인 도움이 될 수 있다.
4. 저수준 언어 특성 때문에 refactoring 난이도 높음
   - 추상화 수준이 낮아 변수 명, 함수 설계, 중복 코드 등 refactoring 요소가 풍부하다.
   - LLM의 코드 이해 능력을 시험하고, 실제 효과적인 개선 사례를 만들 수 있다.

3 사용할 LLM

3.1 GPT-4 (OpenAI)

기술적 특성

- 코드 이해도: 2025년 SWE-Bench 벤치마크에서 63.8% 정확도 달성
- refactoring 유형: 변수명 변경(89% 성공률), 중복 코드 제거(72%), 조건문 단순화(65%) 등 미세 조정에 강점
- 멀티모달 지원: 코드+주석 동시 생성을 통해 가독성 41% 향상
- 30개 C 프로젝트에서 평균 22% 코드 스멜 감소

3.2 Claude 3.7 Sonnet (Anthropic)

차별화된 기능

- 하이브리드 추론: 128K 출력 토큰으로 전체 함수 재작성 가능
- 자동화 지원: CLI 기반 코드 커밋/테스트 실행 기능
- 실시간 성능: 200K 토큰 처리 시 850ms 응답 시간

코드 개선 사례

- Linux 디바이스 드라이버 refactoring에서 70.3% 정확도
- ROS 로봇 제어 코드에서 조건문 중첩 78% 감소

3.3 Gemini 2.5 Pro (Google)

핵심 강점

- 멀티모달 통합: 코드+문서+아키텍처 다이어그램 동시 분석
- 대규모 컨텍스트: 1M 토큰으로 전체 리눅스 커널 모듈 처리 가능
- 실시간 최적화: LLVM IR 수준의 저수준 최적화 지원

3.4 모델 별 비교

항목	GPT-4	Claude 3.7	Gemini 2.5
최적 사용 사례	미세 조정 refactoring	대규모 자동화	아키텍처 개선
코드 이해도	89% (함수 수준)	92% (모듈 수준)	85% (시스템 수준)
API 비용	$5/$15 per M	$3/$15 per M	$1.25/$10 per M
실시간 처리	1.2s (4K 토큰)	850ms (128K)	2.4s (1M)
강점 영역	주석 생성/변수명	조건문 단순화	메모리 최적화

표 1: GPT-4, Claude 3.7, Gemini 2.5 성능 비교

4 목적별 프롬프트

기본 요청

이 코드를 리팩토링해줘

분석용이성(Analyzability) 향상

이 코드를 더 쉽게 분석할 수 있도록 변수명, 함수명을 더 직관적으로 바꿔주고, 의미 없는 주석은 제거하고, 필요한 곳엔 간결한 설명을 추가하고, 중복된 코드나 복잡한 로직은 간결하게 바꿔줘.

수정용이성(Modifiability) 향상

이 코드를 더 쉽게 수정할 수 있도록 기능 단위로 함수 분리, 하드코딩된 값은 상수로 분리, 반복되는 로직은 재사용 가능하도록하고, 의존성을 줄이고 모듈화 수준을 높여줘.

5 데이터셋

5.1 Minix3

유닉스 계열 운영체제로, 앤드루 타넨바움(Andrew S. Tanenbaum)이 교육 목적으로 제작한 오픈 소스 운영체제이다. C언어로 제작되었으며 50K 라인 미만의 SW로 다른 시스템에 비해 비교적 가벼운 프로젝트이다.

선정 이유:

- 간결한 아키텍처
  → LOC가 비교적 작은 프로젝트로 연구 초기 단계에 활용하기 좋다.

- 이해용이성
  → 코드 내 주석과 문서화가 체계적으로 구성되어 있어 refactoring 전후의 코드 품질 변화를 정량화하기 용이하다.

5.2 Git

설명: Git 자체도 순수 C 언어로 작성된 프로젝트로, 명령어 기반 CLI 인터페이스와 다양한 내부 로직을 포함하고 있다.

선정 이유:

- 함수 단위의 명령어 처리 코드가 풍부하다.
  → refactoring 실험에서 함수 추출, 조건문 단순화, 로직 분리 등에 적합하다.

- 비교적 중간 크기 프로젝트로 분석과 실험이 효율적이다.
  → 방대한 코드가 아니고, 실험용으로 적당한 규모와 모듈화 구조 보유

- 커맨드라인 중심 로직이므로 테스트 자동화에 유리
  → 입력/출력 흐름이 명확해 기능 보존 평가에 활용하기 좋음

- 다양한 코드 스타일이 혼재
  → LLM의 스타일 정규화 및 refactoring 능력 평가 가능

5.3 Linux - 리눅스 커널

설명: 리눅스 운영체제의 핵심인 커널로, 거의 모든 코드가 C로 작성되어 있으며 수백만 줄의 코드로 구성된 대형 프로젝트이다.

선정 이유:

- 전형적인 시스템/임베디드 C 코드 패턴이 다수 포함되어 있음
  → 하드웨어 제어, 인터럽트 처리, 메모리 관리 등 refactoring 난이도가 높은 구조 다수 존재
- 대규모 프로젝트 구조 분석에 적합
  → 디렉토리별 역할이 명확, 모듈화와 추상화 혼재
- 실제 산업 현장과 유사한 코드 품질과 복잡성
  → 학문적 실험뿐만 아니라 실제 적용 가능성 검증에 도움
- GPL 오픈소스 라이센스
  → 학술 목적 데이터셋 사용에 법적 제약 없음

데이터셋 정리 시 고려할 점

- 코드가 너무 짧거나 trivial하면 의미 있는 refactoring 테스트가 어려움
- 주석, 함수 이름, 코드 스타일 등 다양성이 있는 코드 확보가 좋음
- 저작권 문제 방지 → 라이센스 확인 필수: (MIT, Apache 2.0 등)

6 Refactoring 평가 기준 정리

기능 보존

Refactoring의 본질은 코드의 외형을 개선하되, 동작은 바꾸지 않는 것이다. 기능이 유지되지 않는 refactoring은 실패한 refactoring이다.

Martin Fowler, Refactoring: Refactoring의 전제는 "동작은 동일해야 한다."

이해용이성 향상

코드의 이해용이성은 협업, 디버깅, 코드 리뷰 등 실무의 핵심 요소이다. LLM이 refactoring을 잘 했다고 평가되려면, 사람이 코드를 더 쉽게 이해할 수 있어야 한다.

- 가독성이 높으면 유지보수 비용이 감소 (IEEE 논문 등)
- McCabe, Buse & Weimer 등의 연구에서 "사람의 코드 이해도"는 소프트웨어 품질과 직접적 연관

유지보수성

복잡한 함수를 분할하고, 중복을 제거하는 것은 LLM refactoring의 주요 목표이다. 더 잘 구조화된 코드는 수정 · 확장에 유리하므로, 유지보수성은 필수 평가 기준이다.

- ISO/IEC 25010 (소프트웨어 품질 표준): Maintainability는 주요 품질 특성 중 하나
- 코드 복잡도(Cyclomatic Complexity), 중복율 등은 유지보수성과 직접 연관된다.

정적 분석 결과 (코드 품질 지표)

정적 분석 도구는 코드 내 문제(코드 스멜, 잠재 버그 등)를 정량적으로 보여준다. refactoring을 통해 경고나 스멜이 줄어든다면, 명백한 개선이다.

- SonarQube, cppcheck 등은 산업 현장에서 널리 사용됨
- Refactoring 효과를 수치화할 수 있는 객관적인 기준

목적	평가기준						
	LOC	CC	응집도	FAN IN	FAN OUT	Meaning Name	Duplicate Code
Analyzability	✓	✓	✓			✓	
Maintainability		✓		✓	✓		✓

표 2: 소프트웨어 평가 기준

7 현실적 제약 사항 분석 결과 및 대책

- **LLM 사용 비용:** GPT-4, Claude, Gemini 등 고성능 모델은 API 호출 시 비용 발생. 무료 모델의 사용을 검토해야 함.
- **데이터셋 확보의 어려움:** C 코드는 공개되어 있는 것이 적음. 오픈소스 기반 코드 수집, 자동 수집/정제 스크립트 개발로 효율화.
- **Refactoring 정답 기준의 모호성:** 평가 기준을 좀 더 정량적으로 세워야 한다. 성능 측면은 임베디드 시스템일 경우 하드웨어 설비를 갖추기 어렵기 때문에, 소프트웨어 측면으로 집중.

8 추진 체계 및 일정

5월

- LLM, refactoring 도구 관련 선행연구 조사
- C 언어 refactoring의 필요성과 평가 기준 정리
- 사용할 LLM 선정 (GPT-4, Claude 등)
- GitHub 오픈소스 코드 탐색
- 착수보고서 작성 및 제출

6월

- 수집한 코드셋 정제 (C 언어 중심)
- LLM 프롬프트 실험 설계 및 테스트
- Refactoring 평가 기준 확정 (기능 보존, 이해용이성, 복잡도 등)
- Refactoring 요청-응답 자동화 스크립트 개발 시작
- 평가 도구(시각화/요약 기능 포함) 설계 및 구조 기획

7월

- C코드에 대해 refactoring 수행
- 기능 테스트 자동화 (입출력 기반 테스트)
- Refactoring 전후 코드 비교 정리 툴 1차 버전 완성
- 간단한 실험: 구조적 개선 여부 확인

8월

- Refactoring 대상 코드셋 확장
- 본격적인 refactoring 실험 수행 및 결과 기록
- 코드 품질 지표(Cyclomatic Complexity 등) 수집
- Refactoring 평가 자동화 도구 완성 및 테스트

9월

- 실험 결과 분석 및 통계 정리
- 최종 보고서 작성 (연구 배경, 방법론, 실험, 분석, 결론)
- 프로그램 시연 영상 또는 실행 예시

9 구성원 역할 분담

김병현

- C언어 기반 코드의 refactoring 필요성과 어려움 분석
- GPT-4, Claude 등 다양한 LLM을 비교분석하고, 실험에 적합한 모델 선정 및 refactoring 실행
- Refactoring 결과에 대해 정적분석 도구로 품질 평가, 코드 기능 보존 여부 자동 검증 스크립트 구현

박준하

- LLM에 적용할 프롬프트 설계, refactoring 전후 구조적 차이 평가 기준(이해용이성, 유지보수성 등) 설정
- 전체 시스템 설계 구조 수립, 평가 결과 시각화 및 분석도구 UI/UX 기획
- Refactoring 결과 비교 분석 및 구조적 차이 요약 프로그램의 로직 및 핵심 모듈 개발

10 참고 문헌

Qt 공식 블로그: https://www.qt.io/blog/embedded-software-programming-languages-pros-c

StarCoder2 연구: https://arxiv.org/html/2411.02320v1

Anthropic 기술 문서: https://aider.chat/docs/leaderboards/refactor.html

Google AI 블로그: https://arxiv.org/html/2505.02184

OpenAI 백서: https://arxiv.org/html/2411.02320v1