Pusan National University

Computer Science and Engineering

Technical Report 2025-09

LLM을 활용한 SW Refactoring

저자 1 202055518 김병현

저자 2 202014511 박준하

지도교수 채흥석

목 차

1. 서론 4

1.1. 연구 배경 4

1.1.1. LLM 서비스의 발전과 소프트웨어 개발 패러다임의 변화 4
1.1.2. 유지보수성 개선 4

1.2. Research Questions 5

2. 연구 배경 6

2.1. LLM 서비스의 발전과 소프트웨어 개발 패러다임의 변화 6

2.2. 유지보수성 개선 6

3. 실험 설계 8

3.1. 실험 대상 LLM 서비스 8

3.2. 데이터셋 8

3.2.1. 데이터셋 요약 9
3.2.2. 데이터셋 특성 9

3.3. 프롬프트 11

3.3.1. RQ 1, 2 프롬프트 11
3.3.2. RQ3 프롬프트 12

3.4. LLM 서비스의 유지보수성 관점의 코드 개선 능력 평가 13

3.4.1. 유지보수성 개선율 평가 방법 14
3.4.2. 유지보수성 이슈 14

4. 실험 결과 17

4.1. RQ 1: 유지보수성 개선 능력 17

4.1.1. 데이터셋 전체 개선 능력 분석 17
4.1.2. 파일 내 일부 개선 능력 분석 18
4.1.3. 파일 내 전체 개선 능력 분석 19

4.2. RQ 2: 코드 특성과 유지보수성 개선 능력과의 관계 20

4.2.1. 규모와 코드 개선율과의 관계 20
4.2.2. 복잡도와 코드 개선율과의 관계 24
4.2.3. 상관 관계 분석 28

4.3. RQ 3: 연속 유지보수성 개선 능력과의 관계 29

4.3.1. 연속 개선에 따른 주요 이슈 추이 29
4.3.2. 코드 특성과 연속 리팩토링과의 관계 분석 31

5. 결론 및 향후 연구 방향 34

5.1. 결론 34

5.2. 향후 연구 방향 35

6. 참고 문헌 36

1. 서론

1.1. 연구 배경

1.1.1. LLM 서비스의 발전과 소프트웨어 개발 패러다임의 변화

거대 언어 모델(Large Language Models, LLM)은 최근 몇 년간 전례 없는 발전을 이루었다. OpenAI의 GPT 시리즈, Anthropic의 Claude, Google의 Gemini 와 같은 모델들은 인간의 언어를 이해하고 생성하는 능력을 바탕으로 다양한 산업 분야에서 혁신을 주도하고 있다.[i]

이러한 기술적 성과는 더 이상 학문적 연구에만 머무르지 않고, API(Application Programming Interface)를 통해 누구나 쉽게 접근하고 활용할 수 있는 LLM 서비스의 형태로 빠르게 확산되었다. 특히 소프트웨어 개발 분야에서 LLM 서비스의 등장은 코딩의 본질적인 방식을 바꾸는 패러다임의 전환을 이끌고 있다. GitHub Copilot 과 같은 도구는 단순한 코드 자동 완성을 넘어, 개발자의 의도를 파악하여 함수 전체나 복잡한 알고리즘을 생성하는 수준에 이르렀다.

이처럼 LLM 서비스는 코드 생성, 디버깅, 문서화 등 개발 생명주기의 여러 단계에 깊숙이 통합되고 있으며, 최근에는 기존 코드의 구조적 문제점을 파악하고 개선하는 코드 리팩토링(Refactoring)과 같은 고차원적인 작업에까지 그 활용 가능성이 확장되고 있다. 이는 LLM이 단순한 보조 도구를 넘어, 개발 생산성과 코드 품질을 동시에 향상시킬 수 있는 핵심 기술로 자리 잡고 있음을 시사한다.

1.1.2. 유지보수성 개선

소프트웨어의 전체 생명주기에서 가장 많은 비용과 시간을 차지하는 단계는 개발이 아닌 유지보수이다. 전체 비용의 50~80% 부분이 소프트웨어가 최초로 배포된 이루의 수정, 개선, 적응 과정에서 발생하는 것으로 알려져 있다.[ii] 따라서 소프트웨어를 얼마나 쉽게 이해하고, 수정하며, 확장할 수 있는지를 나타내는 유지보수성은 프로젝트의 장기적인 성공을 좌우하는 핵심적인 품질 지표이다.

낮은 유지보수성을 가진 코드는 흔히 기술 부채(Technical Debt)를 유발한다. 복잡하게 얽힌 코드(높은 결합도), 불분명한 구조, 문서의 부재 등은 당장의 기능 구현에는 문제가 없을지라도, 시간이 지남에 따라 사소한 변경에도 예기치 않은 버그를 발생시키고 개발 속도를 저하시킨다.

프로젝트의 규모가 커지고 요구사항이 복잡해질수록, 개발자들은 새로운 기능을 추가하는 시간보다 기존 코드를 분석하고 이해하는 데 더 많은 시간을 할애하게 된다. 이처럼 높은 복잡도를 가진 레거시 시스템의 유지보수성 개선은 과제로 남아있다.

특히 C언어는 여전히 임베디드 시스템, 운영체제, 시스템 소프트웨어 등 제한된 자원 환경에서 널리 사용되는 중요한 언어임에도 불구하고, 기존 LLM 리팩토링 연구는 주로 Java, Python, JavaScript/TypeScript와 같은 현대적인 프로그래밍 언어에 집중되어 있어 C언어에 대한 체계적인 연구가 부족한 상황이다.

이러한 배경 속에서, LLM 서비스의 능력을 활용하여 소프트웨어 공학의 고질적인 문제인 유지보수성 저하를 해결하고자 한다.

1.2. Research Questions

본 연구는 정적분석 기반 지표를 활용해 LLM을 활용한 소프트웨어 리팩토링이 코드의 유지보수성을 유의미하게 개선하는지 검증하고, 코드 특성 및 반복 적용 전략과의 관계를 체계적으로 분석하는 것을 목표로 한다.

RQ1. 유지보수성 개선 능력: LLM을 활용한 리팩토링을 통해 유지보수성 관련 지표를 유의미하게 개선하는가?

a. 전체 이슈 분석
b. 일부 개선 능력 분석
c. 전체 개선 능력 분석

RQ2. 코드 특성과 유지보수성 개선 능력과의 관계: 코드 규모(LOC), 복잡도(CC) 등 코드 특성이 LLM의 유지보수성 개선 효과에 어떤 영향을 미치는가?

d. 규모와 코드 개선율과의 관계
e. 복잡도와 코드 개선율과의 관계

RQ3. 연속 유지보수성 개선 능력과의 관계: 동일 코드베이스에 대한 반복 리팩토링이 유지보수성 개선을 안정적으로 향상시키는지, 혹은 수렴, 퇴행 패턴이 존재하는가?

2. 연구 배경

2.1. LLM 서비스의 발전과 소프트웨어 개발 패러다임의 변화

2020년대에 들어 인공지능 기술, 특히 거대 언어 모델(Large Language Models, LLM)은 전례 없는 발전을 이루었다. OpenAI의 GPT 시리즈, Anthropic의 Claude, Google의 Gemini 와 같은 모델들은 인간의 언어를 이해하고 생성하는 능력을 바탕으로 다양한 산업 분야에서 혁신을 주도하고 있다.[iii]

이러한 기술적 성과는 더 이상 학문적 연구에만 머무르지 않고, API(Application Programming Interface)를 통해 누구나 쉽게 접근하고 활용할 수 있는 LLM 서비스의 형태로 빠르게 확산되었다. 특히 소프트웨어 개발 분야에서 LLM 서비스의 등장은 코딩의 본질적인 방식을 바꾸는 패러다임의 전환을 이끌고 있다. GitHub Copilot 과 같은 도구는 단순한 코드 자동 완성을 넘어, 개발자의 의도를 파악하여 함수 전체나 복잡한 알고리즘을 생성하는 수준에 이르렀다.

이처럼 LLM 서비스는 코드 생성, 디버깅, 문서화 등 개발 생명주기의 여러 단계에 깊숙이 통합되고 있으며, 최근에는 기존 코드의 구조적 문제점을 파악하고 개선하는 코드 리팩토링(Refactoring)과 같은 고차원적인 작업에까지 그 활용 가능성이 확장되고 있다. 이는 LLM이 단순한 보조 도구를 넘어, 개발 생산성과 코드 품질을 동시에 향상시킬 수 있는 핵심 기술로 자리 잡고 있음을 시사한다.

2.2. 유지보수성 개선

소프트웨어의 전체 생명주기에서 가장 많은 비용과 시간을 차지하는 단계는 개발이 아닌 유지보수이다. 전체 비용의 50~80% 부분이 소프트웨어가 최초로 배포된 이루의 수정, 개선, 적응 과정에서 발생하는 것으로 알려져 있다.[iv] 따라서 소프트웨어를 얼마나 쉽게 이해하고, 수정하며, 확장할 수 있는지를 나타내는 유지보수성은 프로젝트의 장기적인 성공을 좌우하는 핵심적인 품질 지표이다.

낮은 유지보수성을 가진 코드는 흔히 기술 부채(Technical Debt)를 유발한다. 복잡하게 얽힌 코드(높은 결합도), 불분명한 구조, 문서의 부재 등은 당장의 기능 구현에는 문제가 없을지라도, 시간이 지남에 따라 사소한 변경에도 예기치 않은 버그를 발생시키고 개발 속도를 현저히 저하시킨

다. 이를 측정하기 위해 순환 복잡도(Cyclomatic Complexity, CC) 와 같은 정량적인 지표가 사용되며, 높은 복잡도는 곧 낮은 유지보수성을 의미한다.

프로젝트의 규모가 커지고 요구사항이 복잡해질수록, 개발자들은 새로운 기능을 추가하는 시간보다 기존 코드를 분석하고 이해하는 데 더 많은 시간을 할애하게 된다. 이처럼 높은 복잡도를 가진 레거시 시스템의 유지보수성 개선은 과제로 남아있다.

이러한 배경 속에서, LLM 서비스의 능력을 활용하여 소프트웨어 공학의 고질적인 문제인 유지보수성 저하를 해결하고자 한다.

3. 실험 설계

3.1. 실험 대상 LLM 서비스

본 졸업과제에서는 다음과 같은 3개의 대표적인 LLM 서비스를 대상으로 Refactoring 능력을 평가하고자 한다. GPT, Gemini, Claude는 모두 업계에서 코딩 능력이 우수한 LLM으로 알려져 있다. 본 졸업 과제에서는 각 LLM 서비스의 최신 모델을 대상으로 실험을 하고자 한다.

LLM 서비스	특징
Gemini-2.5-pro	• 100만 토큰 컨텍스트 윈도우(향후 200만 예정) • 강력한 논리/과학/수학 추론 능력 • 다중 모달(텍스트, 이미지, 오디오, 비디오) 지원 • 고급 코드 생성과 분석 능력 • 벤치마크 점수 GPT-4.5 대비 대폭 향상[v]
GPT-5	• 뛰어난 복합 추론 능력 및 빠른 응답 속도 • 멀티모달 통합 모델(텍스트, 이미지 등) • 한 세션에서 여러 도구 사용 가능 • 오류율 20% 이상 감소 • 개인화된 대화 경험과 멀티 턴 대화 지원[vi]
Claude-Opus-4-1	• 심층 다단계 추론 및 복잡한 문제 해결 능력 • 최대 7시간 자율 작업 가능 AI 에이전트 • 뛰어난 코딩 및 알고리즘 생성 능력[vii]

3.2. 데이터셋

본 졸업과제에서는 오픈소스 SW를 대상으로 앞에서 나열한 3개 대표적인 LLM 서비스의 리팩토링 능력을 확인하고자 한다. 실험 대상 오픈소스 SW를 선정한 기준은 다음과 같다.

- 본 졸업과제는 C 프로그램에 대한 리팩토링 능력을 평가하는 것을 목적으로 하고 있다. 따라서 C 언어로 작성된 오픈소스 SW를 데이터셋으로 선정한다.
- 신뢰할만한 그리고 실제로 많이 사용되는 오픈소스를 대상으로 한다. 예를 들어 MINIX, SourceWare에서 관리되는 유용한 SW를 대상으로 한다.

3.2.1. 데이터셋 요약

데이터셋	설명	출처
MINIX	MINIX는 작은 마이크로커널과 분리된 서버 구조, 강력한 신뢰성과 내결함성, POSIX 호환성 등 운영체제 설계의 교과서적 사례로 활용되는 OS이다.	https://github.com/Stichting-MINIX-Research-Foundation/minix
Coreutils (GNU Core Utilities)	리눅스/유닉스 커맨드라인 기본 유틸리티들의 모음으로 대부분 C 언어로 작성되어 있다	git://git.sv.gnu.org/coreutils
glibc (GNU C 라이브러리)	리눅스 환경에서 표준 C 라이브러리로 널리 쓰이는 핵심 라이브러리이며, C로 작성되어 있다. 시스템과 프로그램 간의 인터페이스 역할을 하며 다른 GNU 프로젝트와 함께 사용된다.	https://sourceware.org/git/glibc.git
Lua	Lua는 C 언어로 구현되어 가볍고 빠르며, C/C++ 프로그램에 쉽게 내장(임베딩)할 수 있도록 설계되었다. 덕분에 게임, 웹, IoT, 임베디드 시스템 등 다양한 분야에서 임베디드 스크립트 언어로 활발히 쓰인다.	https://github.com/lua/lua
binutils-gdb	binutils-gdb는 GNU에서 제공하는 바이너리 유틸리티 툴 모음(GNU Binutils)과 디버거(GDB, GNU Debugger)를 통합해서 관리하는 오픈소스 소프트웨어 패키지이다.	https://sourceware.org/git/binutils-gdb.git

Lua 프로젝트는 RQ3 실험에서 사용한다.

3.2.2. 데이터셋 특성

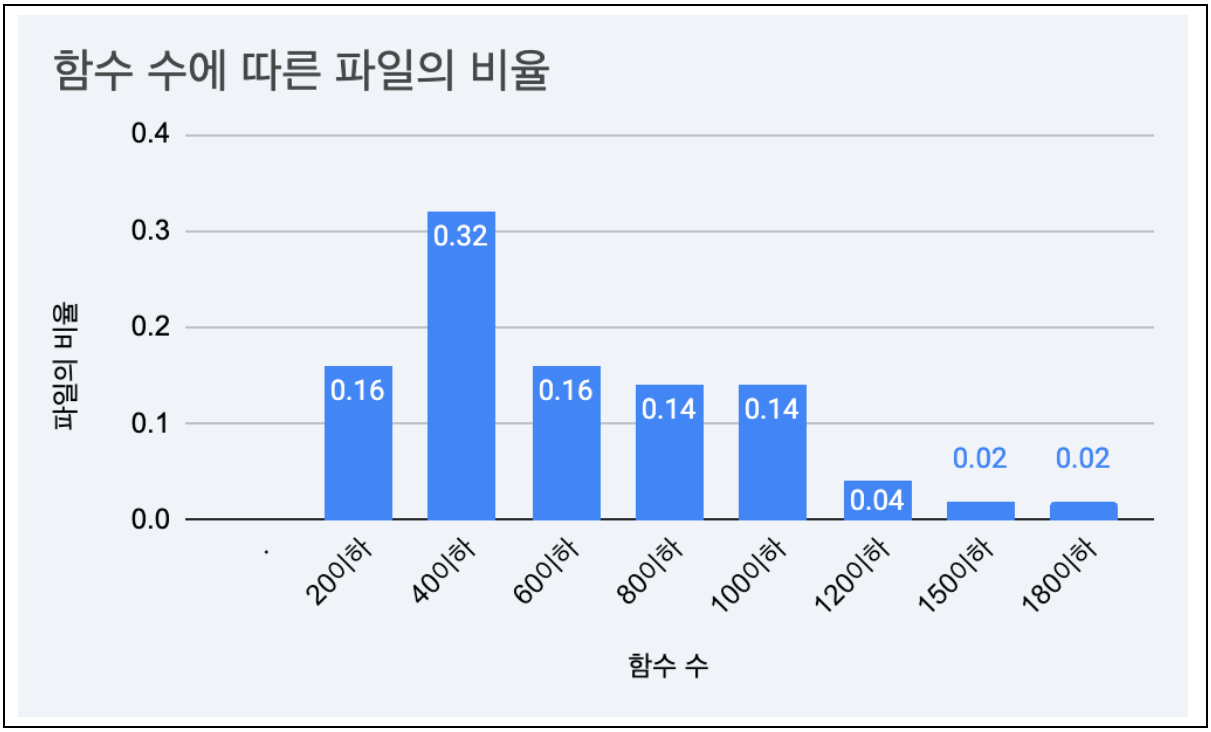
데이터셋의 구조적 특징을 정량적으로 파악하기 위해, 본 졸업과제에서는 오픈소스 정적 분석 도구인 lizard를 사용했다. Lizard는 C, C++, Python 등 다양한 언어의 소스코드를 대상으로 복잡도, 코드 라인 수 등 핵심 소프트웨어 메트릭을 빠르게 산출할 수 있는 명령줄 기반 도구다.

이해를 돕기 위해 소수점은 첫째자리에서 반올림하여 기록했다.

데이터셋	규모			복잡도
	총 파일 수	파일 당 평균 함수 수	파일 당 평균 LOC	함수당 평균 CC
MINIX	20	31	958	8
Coreutils	4	75	2,295	9
glibc	3	73	2,847	12
binutils-gdb	23	69	3,141	12

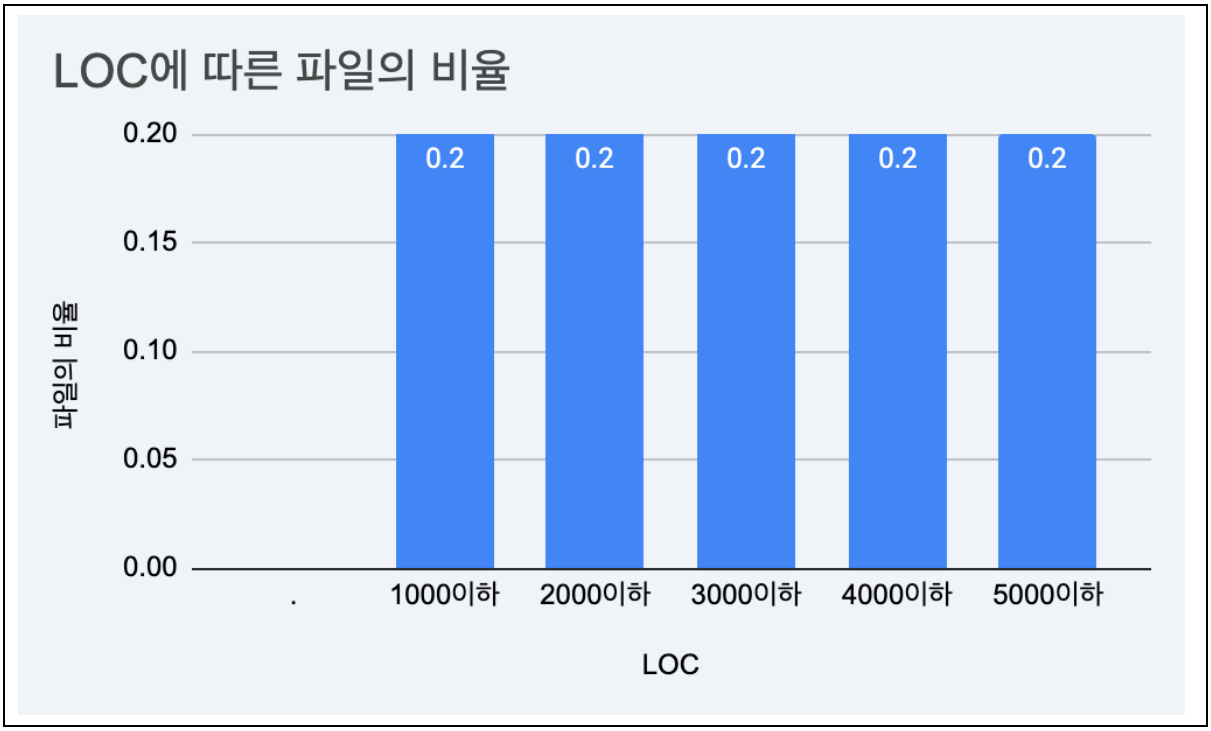
파일 기준 함수 수 분포

- X 축: 함수 수
- Y 축: 해당 함수 수를 가지는 파일의 비율



파일 기준 LOC 분포

- X 축: LOC: 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000이상
- Y 축: 해당 LOC를 가지는 파일의 비율



LOC 구간 별로 리팩토링 성능을 비교하기 위해 균등하게 데이터셋을 수집했다.

3.3. 프롬프트

3.3.1. RQ 1, 2 프롬프트

```
PROMPT = """You are an expert C programmer specializing in improving legacy code for SonarCloud analysis.
Refactor the following C code to improve its Maintainability without altering its external functionality.

Your refactoring must focus on these key SonarQube Maintainability rules:

- **Reduce Cognitive Complexity:** Avoid deep nesting of loops and conditionals (`if`, `else`, `switch`,
```

`for`, `while`). Break down complex logic into smaller, self-explanatory functions.
- **Eliminate Code Duplication:** Strictly follow the DRY (Don't Repeat Yourself) principle. Identify and refactor repeated code blocks into reusable functions.
- **Address Code Smells:** Decompose long functions into shorter ones that perform a single task. Replace "magic numbers" with named constants (`#define` or `const`).

Return ONLY the refactored C code, without any explanation, comments, or markdown formatting like ```c.
"""

역할 부여 (Role Assignment): 프롬프트의 시작 부분에서 You are an expert C developer tasked with refactoring code... 와 같이 LLM에게 'C 언어 리팩토링 전문가'라는 명확한 역할을 부여했다. 이는 LLM이 후속 요청을 처리할 때 특정 분야의 전문 지식을 최대한 활용하도록 만드는 중요한 단계이다.[viii]

명확한 목표 제시 (Clear Objective): SonarQube Issues to Fix 섹션을 통해 LLM이 해결해야 할 문제점을 명확하게 제시했다. 이는 SonarQube Cloud 정적 분석 도구를 통해 객관적으로 검증된 이슈 목록으로, '코드를 더 좋게 만들어줘'와 같은 모호한 요청 대신 구체적이고 측정 가능한 목표를 제공하여 리팩토링의 방향성을 명확히 한다.

3.3.2. RQ3 프롬프트

이슈 기반 리팩토링을 위한 프롬프트는 아래와 같이 구성했다.

You are an expert C developer.
Refactor the code to fix the listed SonarCloud issues without changing behavior.
Project context (related signatures/types):
<context>
{context}
</context>
File: {filepath}
Original C code:
```c{code}```
```

```
Issues to fix (file-scoped}:
<issues>
{issues}
</issues>
```

이슈 기반 리팩토링 이후 빌드 오류 해결을 위한 보완 프롬프트도 함께 사용하여 연속 적용의 안정성을 확보한다.

```
You are an expert C developer.
Your mission is to correct the provided C code so that it complies successfully.
Analyze the build errors from the ‘make’ log to identify the problems, and use the project context to apply the correct fixes.
The goal is to resolve the errors while preserving the original intended functionality.

Project context (related signatures/types):
<context>
{context}
</context>
File: {filepath}
Original C code:
```c{code}```

Build Errors (make log):
<log>
{log}
</log>
```

3.4. LLM 서비스의 유지보수성 관점의 코드 개선 능력 평가

초기 연구 계획에서는 유지보수성 인덱스(MI)를 핵심 지표로 선정하여 다음과 같은 공식을 적용하고자 했다.

$$\text{MI} = 171 - 5.2 \times \ln(\text{Halstead Volume}) - 0.23 \times (\text{Cyclomatic Complexity}) - 16.2 \times \ln(\text{Lines of Code})$$

그러나 연구 진행 과정에서 다음과 같은 문제점들이 확인되었다:

1. 단일 지표의 한계: MI는 여러 지표를 종합한 단일 수치이지만, 개별 문제점을 구체적으로
```

식별하고 개선 방향을 제시하기에는 부족했다.

2. 실무적 활용성 부족: MI 점수의 개선이 실제로 어떤 코드 품질 측면이 향상되었는지 명확하게 파악하기 어려웠다.
3. 도구 가용성 문제: C언어에 대한 정확한 MI 계산 도구의 확보가 어려웠고, 계산 결과의 일관성 확보에 어려움이 있었다.

따라서, LLM을 통해 리팩토링된 코드의 품질 개선 효과를 객관적으로 측정하기 위해, 산업 표준 정적 분석 도구인 SonarQube Cloud를 평가 시스템으로 도입했다. SonarQube Cloud는 코드의 잠재적 버그, 보안 취약점, 유지보수성 이슈 등을 다각적으로 분석한다.

3.4.1. 유지보수성 개선율 평가 방법

본 졸업과제는 유지보수성 측면의 코드 리팩토링에 초점을 둔다. 따라서, SonarQube Cloud가 검출하는 유지보수성 이슈를 활용하여 LLM 서비스의 유지보수성 개선 능력을 평가하고자 한다. 유지보수성 개선율은 다음과 같이 정의된다.

- SRC : 리팩토링 전의 소스코드
- SRC_{llm}: LLM서비스 llm (예: GPT-5) 을 통해서 리팩토링된 소스코드
- A : SRC에서 검출된 유지보수성 이슈 수
- B_{llm} : SRC_{llm}에서 검출된 유지보수성 이슈 수
- 유지보수성 개선율(llm) = (A – B_{llm}) / A * 100

3.4.2. 유지보수성 이슈

SonarQube Cloud는 소스코드에 존재하는 유지보수성 관련 이슈를 검출한다. 예를 들어 다음은 대표적인 유지보수성 이슈를 보여 준다.

유지보수성 이슈	설명
This goto statement must be replaced by a standard iteration statement.	goto 문은 입구와 출구가 명확한 블록 단위로 코드를 구성하는 구조적 프로그래밍의 핵심 원칙을 위배한다. 표준 반복문(for, while)이나 조건문(if, switch)을 사용하여 코드를 더 논리적이고 체계적인 블록으로 구성하도록 권장한다.

Refactor this code to not nest more than 3 if|for|do|while|switch statements.	코드의 중첩(nesting) 깊이가 너무 깊다는 것을 지적이다. 즉, if문 안에 또 다른 if문이 있고, 그 안에 for문이 있는 것처럼 제어문이 3~4단계 이상으로 겹겹이 쌓여있는 구조를 개선하도록 권장한다.
This function has 8 parameters, which is greater than the 7 authorized.	함수에 전달되는 파라미터(인자)의 개수가 너무 많다는 것을 지적한다. 일반적으로 파라미터 개수는 4~5개, 최대 7개를 넘지 않도록 권장한다.
Define each identifier in a dedicated statement.	하나의 선언문에는 변수 하나만 선언하라는 의미이다. 즉, 콤마(,)를 사용하여 한 줄에 여러 변수를 선언하는 것을 지양하라는 권고이다.

또한, SonarQube Cloud는 유지보수성 이슈를 그 심각도를 기준으로 Blocker, High, Medium, Low로 분류한다. 즉 Blocker, High, Medium, Low의 순으로 보다 심각한 유지보수성 문제를 유발할 수 있는 이슈를 의미한다. 다음은 각 심각도 수준 별 유지보수성 이슈를 예시한다.

이슈 심각도	예시
Blocker	- This goto statement must be replaced by a standard iteration statement. - Remove this misleading “common_error” label (in switch block)
High	- Refactor this code to not nest more than 3 if|for|do|while|switch statements. - Refactor this function to reduce its Cognitive Complexity from 32 to the 25 allowed.
Medium	- This function has 8 parameters, which is greater than the 7 authorized. - Move all #include directives to the very top of the file, before any code.
Low	Define each identifier in a dedicated statement. Declare the variable “i” inside the loop.

본 졸업과제에서는 전체 이슈 (Blocker, High, Medium, Low) 뿐만 아니라 주요 이슈 (Blocker, High) 도 기준으로 삼는다.

전체 이슈 대비 주요 이슈에도 연구하는 전략을 채택한 근거는 다음과 같다.

- 실무적 타당성: 실무 환경에서는 모든 이슈를 동시에 해결하는 것이 현실적으로 불가능하며, 심각도와 영향도를 고려한 우선순위 기반 접근이 필요하다. 스프린트 기반 개발과 기술 부

채 관리 관점에서, 주요 이슈에 집중하는 것이 ROI(투자 대비 효과)를 극대화한다.

- 측정 정밀도 향상: 전체 이슈를 대상으로 하면 대부분이 Medium/Low 수준으로 희석되어 모델 간 차이가 통계적으로 유의하지 않을 가능성이 높다. 주요 이슈만 추출하면 실질적 개선 효과를 민감하게 감지할 수 있다.

4. 실험 결과

4.1. RQ 1: 유지보수성 개선 능력

여러 조건에서 LLM 서비스의 리팩토링 성능을 관찰하기 위해, 3가지의 실험을 진행했다.

- 데이터 셋 전체의 개선 능력
- 파일 내 일부 개선 능력
- 파일 내 전체 개선 능력

이번 실험에서 코드 이슈의 심각도 조건에 따른 실험 결과를 수집하기 위해 SonarQube Cloud 에서 검출하는 유지보수성 이슈의 심각도를 기준으로 전체 이슈 (Blocker, High, Medium, Low), 주요 이슈 (Blocker, High) 로 분류하여 실험을 진행하였다.

4.1.1. 데이터셋 전체 개선 능력 분석

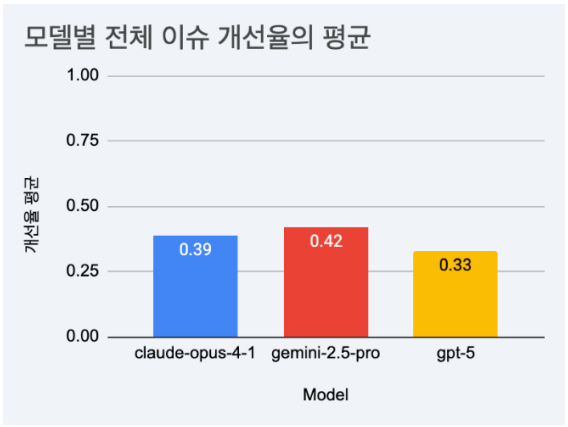
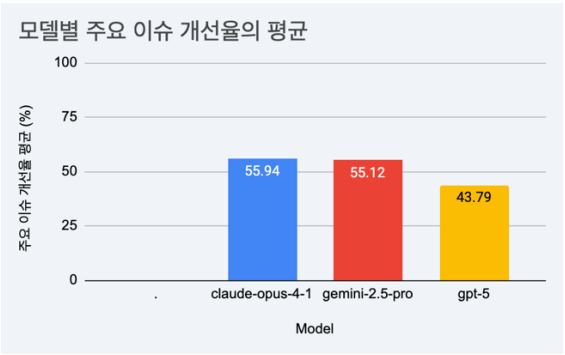
아래는 데이터셋의 전체 이슈와 주요 이슈 수를 각 LLM서비스의 리팩토링 전과 후로 보여준다.

데이터셋	리팩토링 전 이슈 수		리팩토링 후 이슈 수					
			GPT-5		Gemini-2.5-pro		Claude-opus-4-1	
	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue
MINIX	648	152	552	111	517	123	518	68
Coreutils	426	322	399	251	356	200	390	237
glibc	132	177	118	109	207	96	264	104
binutils-gdb	2,207	1,423	1,252	621	1,238	313	993	357

X 축: LLM

Y 축: 모델 별 개선율의 평균 (개선율은 위에서 언급한 공식을 사용한다.)

각 LLM 서비스 별 유지보수성 개선율을 막대 그래프로 제시한다.

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>(a) 코드 개선율: 전체 이슈</td>
<td>(b) 코드 개선율: 주요 이슈</td>
</tr>
</table>

LLM 서비스를 활용한 자동화된 코드 리팩토링이 실질적인 코드 품질 개선으로 이어진다는 것을 정량적으로 확인할 수 있다.

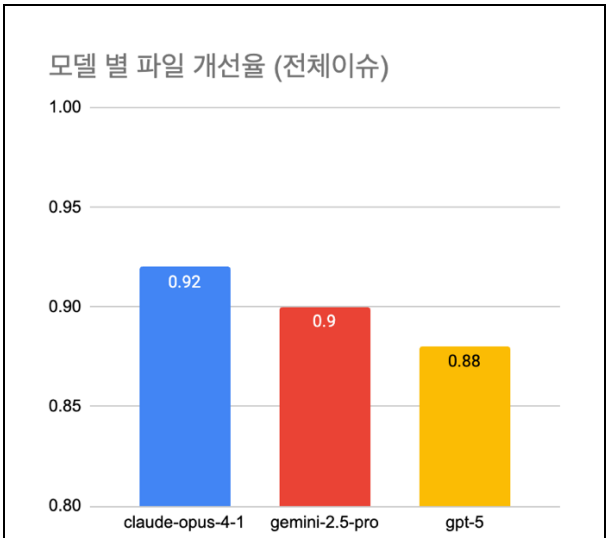
4.1.2. 파일 내 일부 개선 능력 분석

LLM 서비스가 데이터 셋에서 얼마나 많은 파일들을 리팩토링을 통해 개선하는지 알아본다.

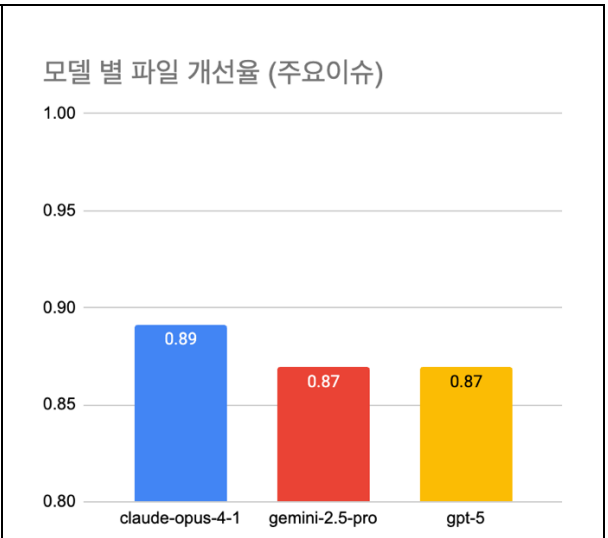
데이터 셋	리팩토링 전 이슈 보유한 파일 수		리팩토링 후 이슈 감소한 파일 수					
			GPT-5		Gemini-2.5-pro		Claude-opus-4-1	
	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue
MINIX	20	16	15	12	17	12	17	13
Coreutils	4	4	4	4	4	4	4	4
glibc	3	3	3	3	3	3	3	3
binutils-gdb	23	23	22	21	21	21	22	21

X 축: LLM

Y 축: 리팩토링 후 이슈 감소 파일 수 / 리팩토링 전 이슈 보유 파일 수



(a) 코드 개선율: 전체 이슈



(b) 코드 개선율: 주요 이슈

세 가지 LLM 서비스 모두 SonarQube Cloud가 탐지한 이슈를 해결하고 있음을 정량적으로 확인했다.

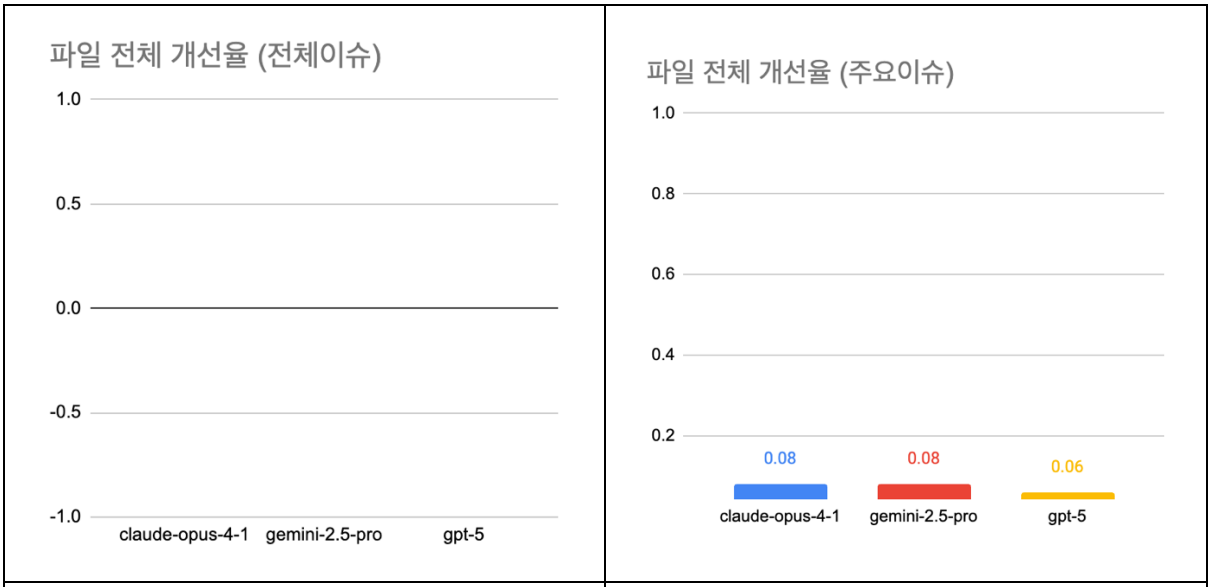
4.1.3. 파일 내 전체 개선 능력 분석

LLM 서비스가 리팩토링을 하면서 한 파일의 이슈를 모두 개선하는지 확인한다.

데이터 셋	리팩토링 전 이슈 보유 파일 수		리팩토링 후 모든 이슈 제거 파일 수					
			GPT-5		Gemini-2.5-pro		Claude-opus-4-1	
	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue
MINIX	20	16	0	3	0	4	0	4
Coreutils	4	4	0	0	0	0	0	0
glibc	3	3	0	0	0	0	0	0
binutils-gdb	23	23	0	0	0	0	0	0

X 축: LLM

Y 축: 리팩토링 후 이슈 제거 파일 수 / 전체 이슈 보유 파일 수



(a) 코드 개선율: 전체 이슈 | (b) 코드 개선율: 주요 이슈

LLM 서비스가 한 번의 리팩토링으로 파일 내의 모든 이슈를 처리하는 능력이 부족한 모습을 볼 수 있다.

4.2. RQ 2: 코드 특성과 유지보수성 개선 능력과의 관계

다른 연구[ix]를 참고하여 LLM 서비스의 리팩토링 능력에 영향을 주는 요소로 파일 규모(LOC) 와 함수 복잡도(CC)를 선정하고 실험을 진행했다.

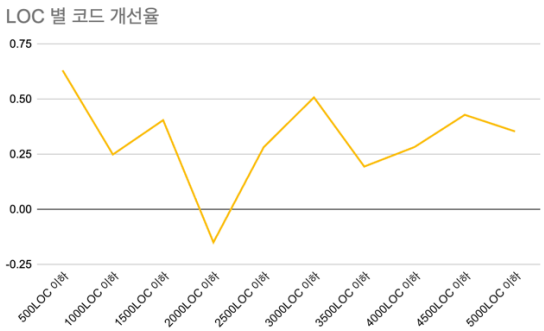
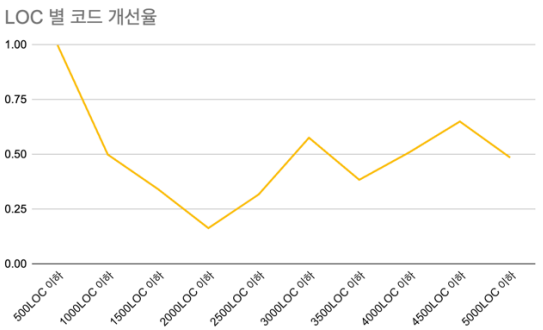
4.2.1. 규모와 코드 개선율과의 관계

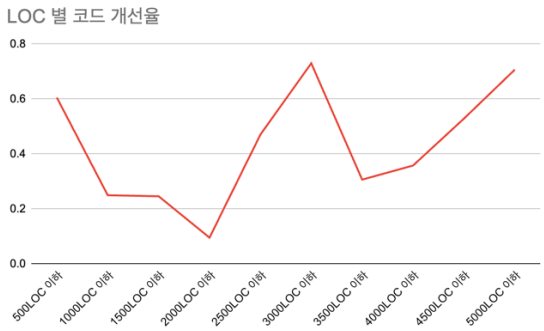
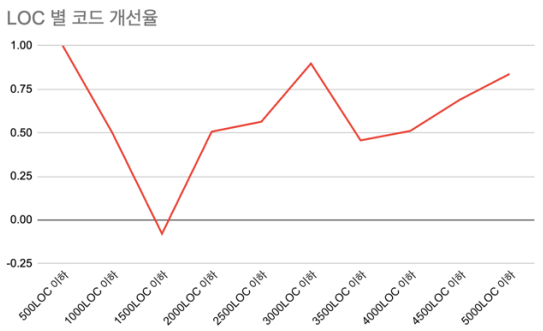
파일 규모	리팩토링 전 이슈 수		리팩토링 후 이슈 수					
			GPT-5		Gemini-2.5-pro		Claude-opus-4-1	
	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue
500LOC 이하	38	4	14	0	15	0	19	2
1000LOC 이하	68	10	51	5	51	5	56	2
1500LOC 이하	207	76	123	50	156	82	111	37
2000LOC 이하	303	73	348	61	274	36	312	27
2500LOC 이하	325	220	233	150	172	96	232	148
3000LOC 이하	330	215	162	91	89	22	103	37

3500LOC 이하	466	278	375	171	323	151	410	163
4000LOC 이하	458	274	328	134	294	134	286	112
4500LOC 이하	607	381	346	133	286	118	313	114
5000LOC 이하	792	543	511	279	232	88	323	124

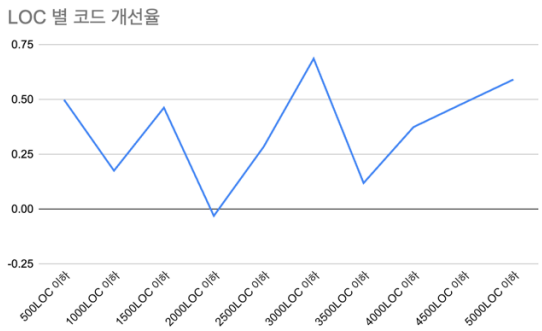
X 축: LOC, Y 축: 유지보수성 개선율

아래와 같이 각 LLM 서비스 별로 표현

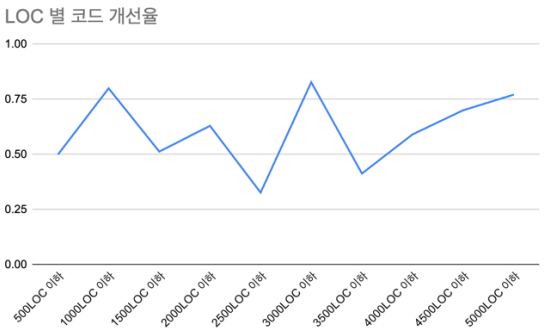
<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>(a) 코드 개선율: 전체 이슈</td><td>(b) 코드 개선율: 주요 이슈</td></tr>
<tr><td colspan="2">GPT-5</td></tr>
</table>

<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>(a) 코드 개선율: 전체 이슈</td><td>(b) 코드 개선율: 주요 이슈</td></tr>
</table>

gemini-2.5-pro



(a) 코드 개선율: 전체 이슈

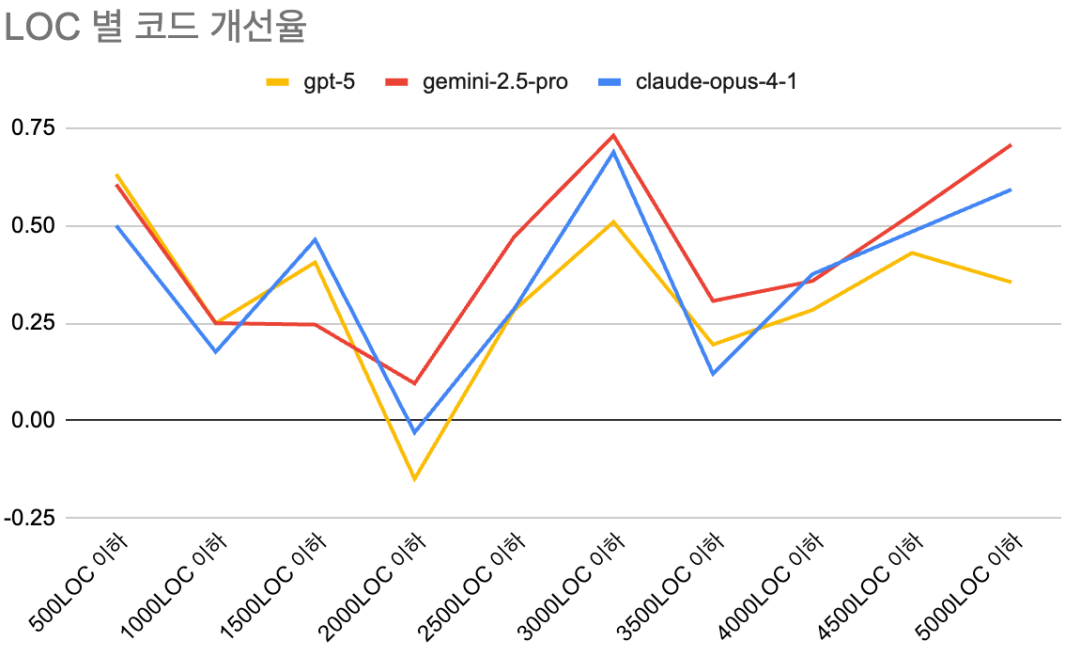


(b) 코드 개선율: 주요 이슈

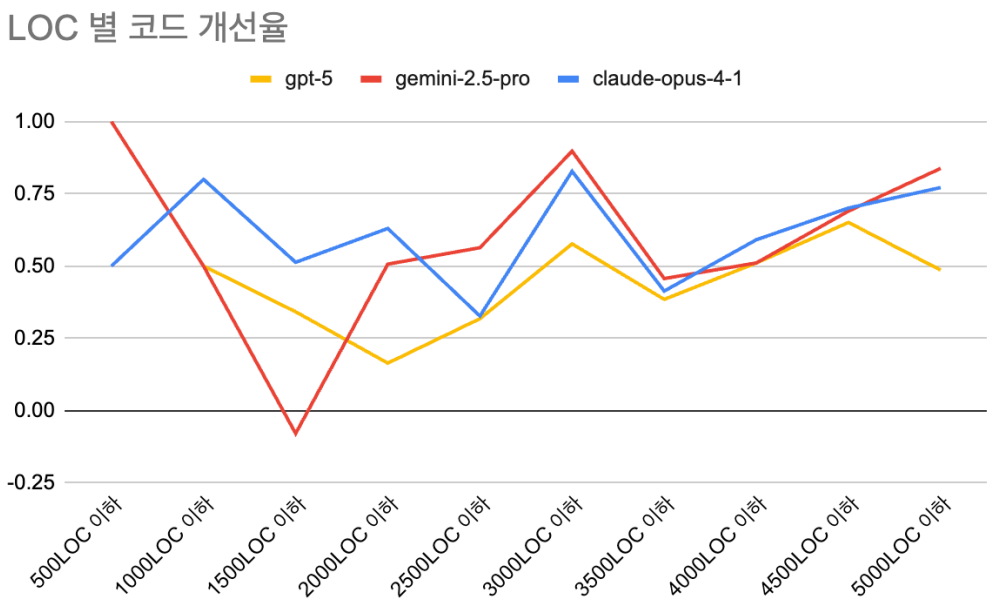
claude-opus-4-1

3개 LLM 서비스 모두 표현

특정 규모에서 어떤 LLM서비스가 보다 우수한지를 확인하고자 한다.



(a) 코드 개선율: 전체 이슈



(a) 코드 개선율: 주요 이슈

두 그래프는 세 가지 LLM 모델(GPT-5, Gemini-2.5-pro, Claude-opus-4-1)이 파일 크기별로 어떤 리팩토링 성능을 보이는지 비교한다. 전반적인 추세는 유사하지만, 모델별로 성능의 변동성과 강점을 보이는 구간이 다르게 나타난다.

- **초기 성능 (500 LOC 이하):** Gemini-2.5-pro 와 GPT-5 모델은 500 LOC 이하의 작은 파일에서 가장 높은 개선율을 보이며, 규모가 작은 코드에 대한 수정 능력이 뛰어남을 보여준다.
- **성능 저하 구간 (1500 ~ 2500 LOC):** 세 모델 모두 1500 LOC 에서 2500 LOC 사이 구간에서 눈에 띄는 성능 저하를 경험한다. 특히 두 번째 그래프에서 Gemini-2.5-pro 모델은 개선율이 음수(-)로 떨어지는데, 이는 해당 구간의 코드에 대해 리팩토링이 오히려 품질을 악화시켰음을 의미한다. 이는 중간 크기의 복잡도가 LLM 의 일관된 추론을 방해하는 '인지적 과부하' 구간일 가능성을 시사한다.
- **성능 회복 및 후반부 성능 (3000 LOC 이상):** 성능 저하 구간을 지난 후, 모든 모델은 다시 개선율을 회복하며 파일 크기가 커져도 안정적인 성능을 유지하는 경향을 보인다. 특히 Gemini-2.5-pro 와 Claude-opus-4-1 모델은 3000 LOC 이상의 대규모 파일에서도 높은 개선율을 기록하며, 복잡하고 긴 코드에 대한 이해 및 수정 능력이 우수함을 나타낸다.

4.2.2. 복잡도와 코드 개선율과의 관계

함수 복잡도(CC) 가 리팩토링 성능에 미치는 영향이 있는지 확인한다.

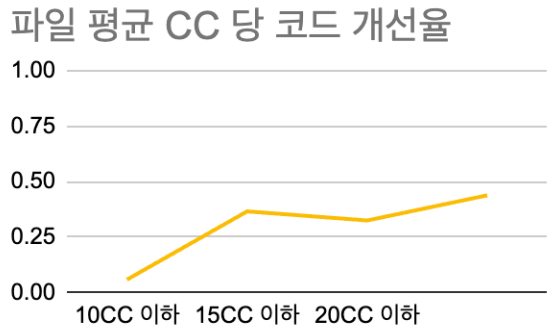
파일 평균 함수 복잡도	리팩토링 전 이슈 수		리팩토링 후 이슈 수					
			GPT-5		Gemini-2.5-pro		Claude-Opus-4-1	
	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue
5CC이하	85	22	80	13	85	10	90	11
10CC 이하	429	276	272	158	212	104	268	160
15CC 이하	594	339	401	201	364	177	406	160
20CC 이하	518	346	291	181	121	23	147	44

파일 평균 함수 복잡도 = 파일의 총 CC / 파일의 총 Function 수

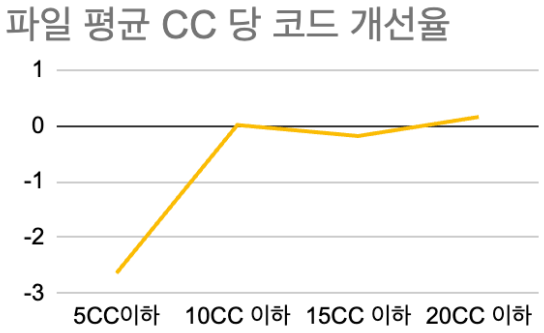
파일 평균 함수 복잡도	리팩토링 전 이슈 수		리팩토링 후 이슈 수					
			GPT-5		Gemini-2.5-pro		Claude-Opus-4-1	
	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue	전체 #issue	주요 #issue
5CC이하	85	22	80	13	85	10	90	11
10CC 이하	429	276	272	158	212	104	268	160
15CC 이하	594	339	401	201	364	177	406	160
20CC 이하	518	346	291	181	121	23	147	44

X 축: 파일 평균 함수 복잡도, Y 축: 유지보수성 개선율

아래와 같이 각 LLM 서비스 별로 표현

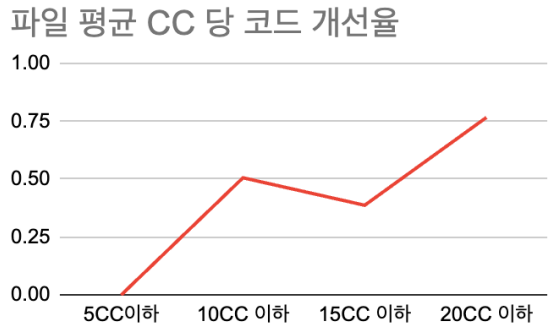


(a) 코드 개선율: 전체 이슈

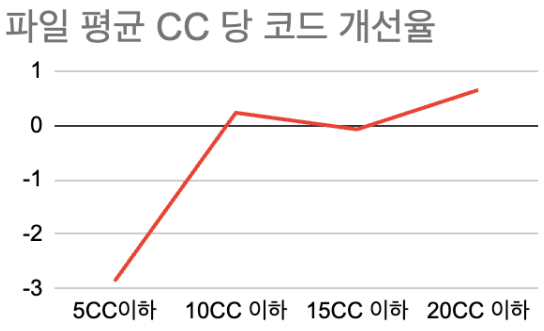


(b) 코드 개선율: 주요 이슈

GPT-5

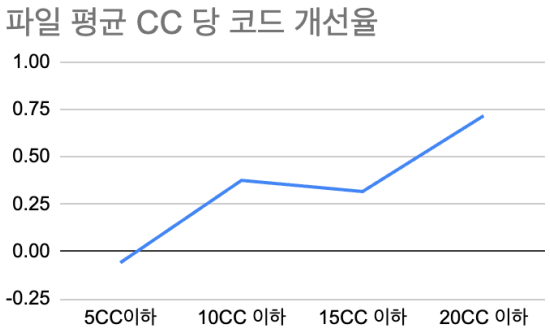


(a) 코드 개선율: 전체 이슈

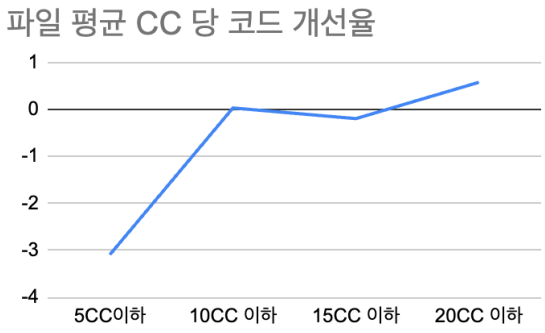


(b) 코드 개선율: 주요 이슈

Gemini 2.5 pro



(a) 코드 개선율: 전체 이슈

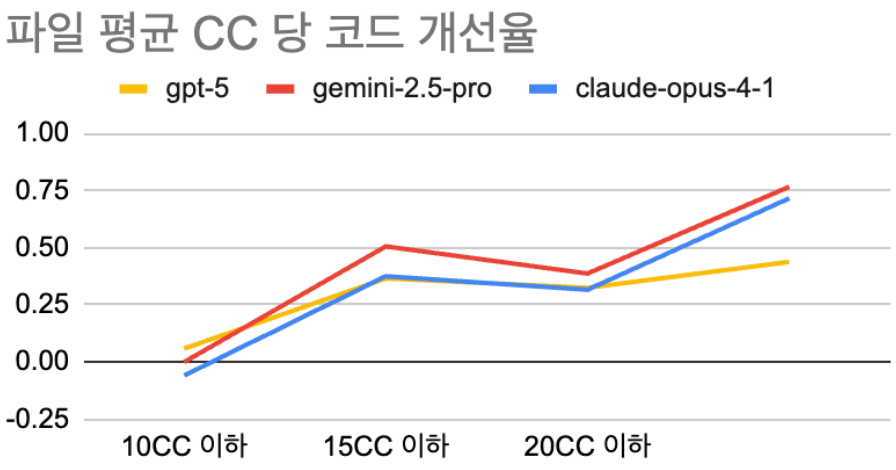


(b) 코드 개선율: 주요 이슈

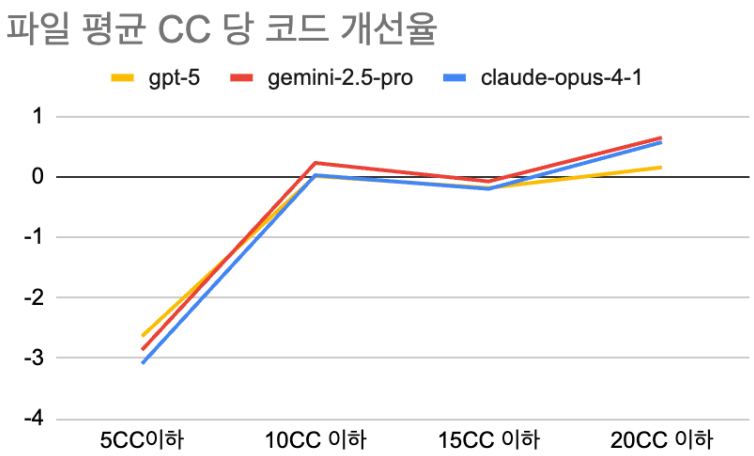
Claude opus 4-1

3개 LLM 서비스 모두 표현

특정 규모에서 어떤 LLM서비스가 보다 우수한지를 확인하고자 한다.



(a) 코드 개선율: 전체 이슈



(a) 코드 개선율: 주요 이슈

5CC 이하 구간에서 모든 모델의 개선율이 큰 폭의 음수 값(-3.0 ~ -2.5) 을 기록했다. 이는 LLM 이 이미 충분히 단순하고 명확한 코드에 대해서는 불필요한 변경을 시도하여 오히려 코드의 품질을 심각하게 악화시켰음을 의미한다.
10CC 이상의 코드에 대해서 모든 모델이 긍정적인 개선율을 보였으며, 특히 gemini-2.5-pro 모델이 대부분의 구간에서 가장 높은 성능을 나타냈다. 복잡도가 높은 20CC 이상 구간에서는 모든 모델의 개선율이 큰 폭으로 상승하며 가장 뛰어난 효과를 보였다.

위 실험을 통해 얻은 결론은 아래와 같다.

- 단순한 코드(Low-CC)는 리팩토링 대상에서 제외하는 것이 바람직하다. LLM 의 불필요한 개입이 오히려 코드 품질을 해칠 위험이 있다.
- 복잡한 코드(High-CC)일수록 LLM 의 개선 효과가 극대화된다. 따라서 LLM 을 활용한 자동화된 유지보수 전략은 기술 부채가 많이 쌓인 복잡한 모듈을 우선적인 대상으로 삼을 때 가장 큰 효율을 얻을 수 있다.

4.2.3. 상관 관계 분석

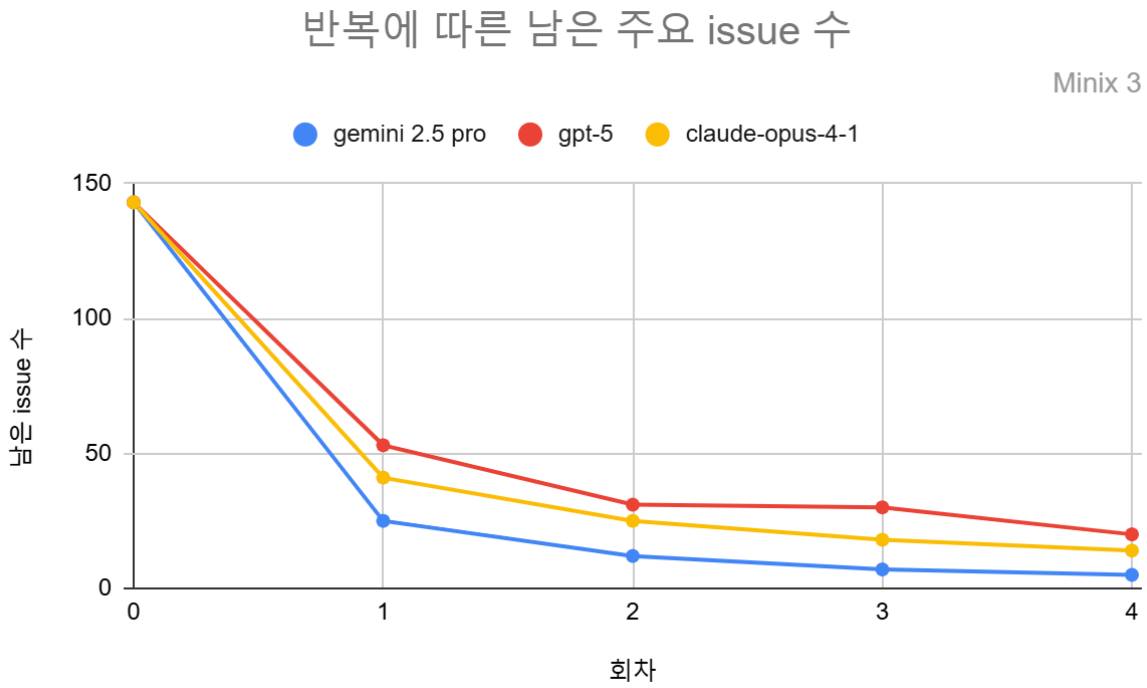
LLM 서비스	규모		복잡도	
	전체 #issue	주요 #issue	전체 #issue	주요 #issue
GPT-5	-0.036	0.020	-0.142	-0.213
Gemini-2.5-pro	0.172	0.177	0.144	0.225
Claude-opus-4.1	0.114	0.095	0.185	0.054

LLM 서비스 별 성능을 심층 분석한다.

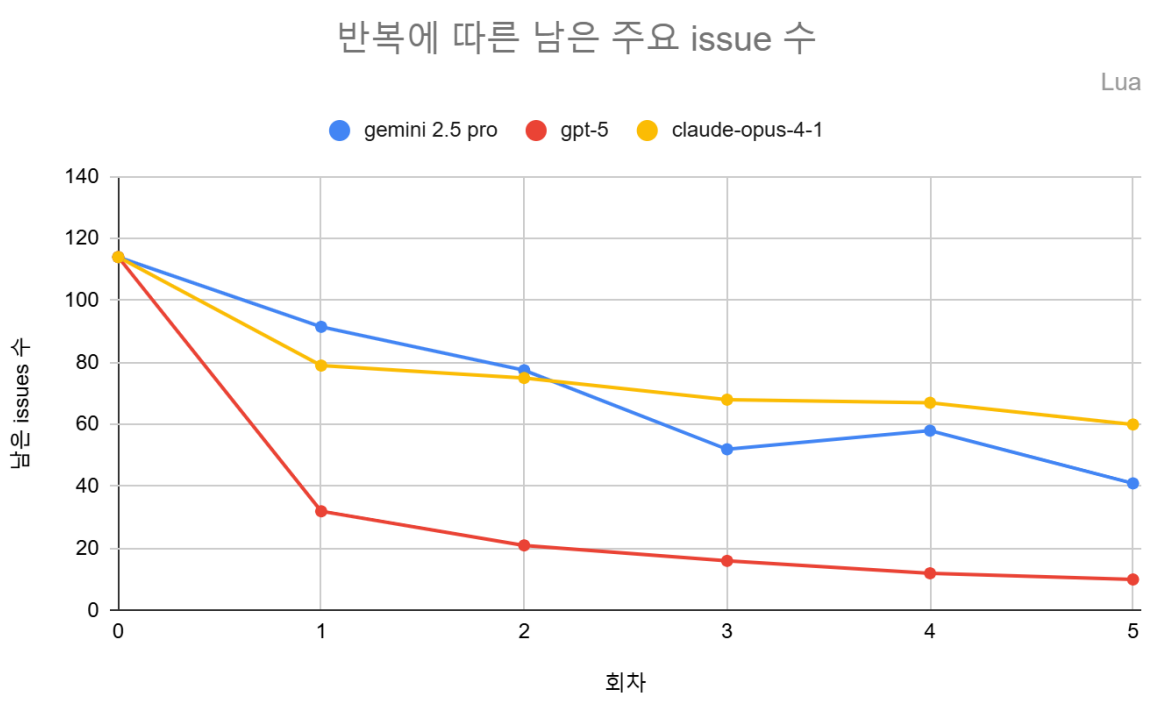
- Gemini-2.5-pro:
    - 모든 4 개 지표에서 일관되게 긍정적인 개선율을 기록한 유일한 모델입니다.
    - 특히 주요 이슈 해결 능력(규모: 0.177, 복잡도: 0.225)에 긍정적인 모습을 보이며, 심각한 버그나 구조적 결함을 효과적으로 수정하는 데 가장 강점을 보였다. 이는 코드의 안정성을 높이는 데 가장 적합한 모델임을 시사한다.
- Claude-opus-4-1:
    - 복잡도 관련 전체 이슈(0.185) 개선율에서 세 모델 중 가장 높은 점수를 기록했다. 이는 코드를 더 읽기 쉽고 단순한 구조로 만드는 일반적인 리팩토링 작업에 매우 효과적임을 의미한다.
    - 규모와 관련된 주요 이슈(0.095) 개선율은 Gemini-2.5-pro 에 비해 다소 낮아, 치명적인 문제 해결보다는 전반적인 코드 클린업에 더 특화되어 있을 가능성을 보여준다.
- GPT-5:
    - 가장 주목할 만한 결과로, GPT-5 는 3 개 지표에서 개선율이 음수로 나타났다. 이는 리팩토링 시도가 오히려 새로운 이슈를 만들거나 기존 구조를 더 나쁘게 만들어 기술 부채를 증가시켰음을 의미한다.
    - 특히 복잡도 관련 주요 이슈(-0.213) 에서 큰 폭의 마이너스 점수를 기록한 것은, 복잡한 로직을 수정하려다 더 심각한 구조적 문제를 야기할 수 있는 잠재적 위험성을 내포하고 있음을 보여준다.

4.3. RQ 3: 연속 유지보수성 개선 능력과의 관계

4.3.1. 연속 개선에 따른 주요 이슈 추이



반복에 따른 Minix 3의 남은 주요 issue 수



반복에 따른 Lua의 남은 주요 issue 수

자체 설계한 에이전트를 사용해 동일 코드베이스에 대해 반복적으로 이슈를 전달하는 리팩토링을 수행하고, 각 회차 종료 시점의 주요 이슈의 남은 수를 계측하였다. 동일 평가 절차와 공통 프롬프트 운용 원칙을 유지하여 회차 간 추세 비교의 일관성을 확보하였다.

1차 리팩토링에서 주요 이슈가 급감하고, 2회차 이후로는 완만한 감소 구간으로 진입하는 패턴이 전 모델에서 공통적으로 관찰되었다. 반복 초기에 대다수의 교정이 비교적 쉬운 이슈가 제거되고 아직 LLM이 기술적으로 제거하기 힘든 이슈가 잔존해 있는 것으로 보인다.

잔존 주요 이슈 중 상당 부분이 "Refactor this code to not nest more than 3 if | for | do | while | switch statements." 유형으로, 중첩 제어문을 초과하는 구조로 집중되는 양상이 나타났다. 이때, 리팩토링을 수행하지 못한 파일에 대해 agent는 사유를 함께 출력하도록 하여 확인해보니 "The code contains complex nested structures and high cognitive complexity that cannot be refactored without potentially altering the behavior. The functions are tightly coupled with specific logic that is difficult to simplify without a deeper understanding of the entire system and its dependencies. Therefore, it is safer to leave the code as is." 라는 사유를 출력하였다. 이 사유는 복잡한 중첩 구조, 높은 인지 복잡성이 자동화된 보수적 리팩토링의 한계로 작용했음을 설명한다.

4.3.2. 코드 특성과 연속 리팩토링과의 관계 분석

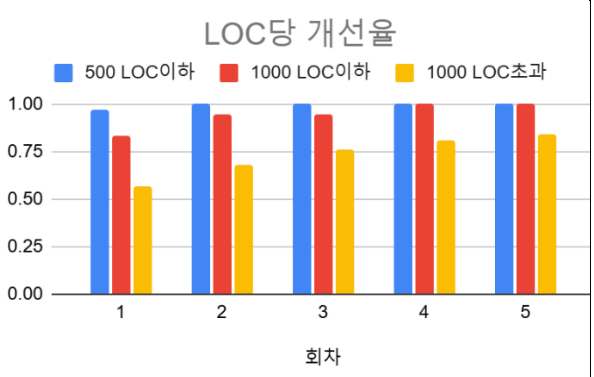
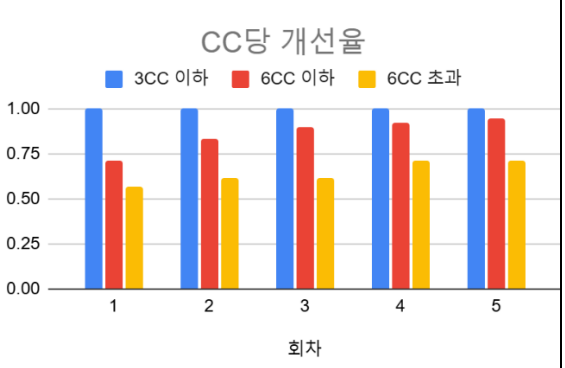
아래는 Lua 데이터셋을 LOC 별로 그룹화하여 전체 이슈와 주요 이슈의 추이를 비교한 표이다.

LLM 서비스	LOC	개선 전 이슈 수 (전체/주요)	연속 개선 후 이슈 수 (전체/주요)				
			1차	2차	3차	4차	5차
GPT-5	500이하	167/32	130/1	130/0	130/0	130/0	130/0
	1000이하	188/18	163/3	163/1	165/1	169/0	169/0
	1000초과	269/63	231/27	231/20	223/15	219/12	216/10
Gemini 2.5 pro	500이하	167/32	139/10	132/3	130/1	129/5	129/0
	1000이하	188/18	172/7	172/6	167/3	167/3	168/3
	1000초과	269/63	226/35	203/17	238/43	205/42	192/7
Claude Opus 4.1	500이하	167/32	144/9	136/4	168/6	133/1	134/1
	1000이하	188/18	170/10	173/11	198/25	173/6	173/6
	1000초과	269/63	267/61	269/63	212/43	268/60	257/53

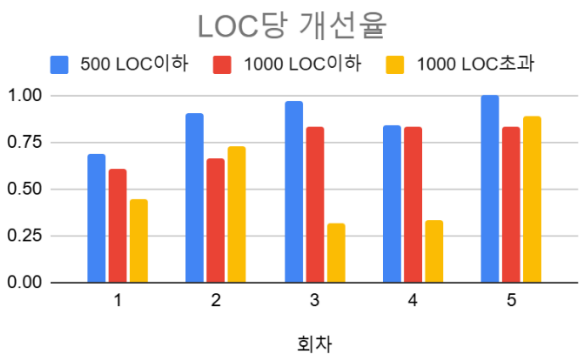
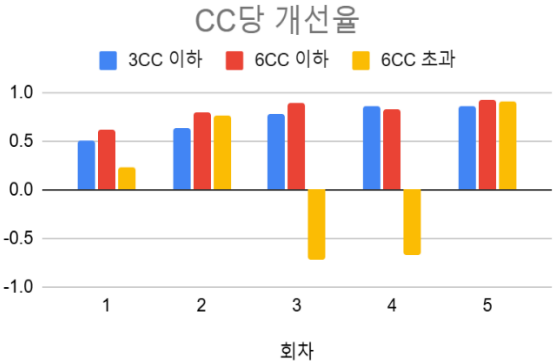
아래는 Lua 데이터셋을 CC 별로 그룹화하여 전체 이슈와 주요 이슈의 추이를 비교한 표이다.

LLM 서비스	CC	개선 전 이슈 수 (전체/주요)	연속 개선 후 이슈 수 (전체/주요)				
			1차	2차	3차	4차	5차
GPT-5	3 CC이하	150/14	163/0	163/0	163/0	163/0	163/0
	6 CC이하	348/77	259/22	259/13	251/8	253/6	251/4
	6 CC초과	92/21	100/9	100/8	102/8	100/6	99/6
Gemini 2.5 pro	3 CC이하	150/14	164/7	185/5	184/3	159/2	153/2
	6 CC이하	348/77	272/29	256/16	246/8	246/13	236/6
	6 CC초과	92/21	99/16	64/5	103/36	94/35	65/2
Claude Opus 4.1	3 CC이하	150/14	178/12	173/11	173/8	200/12	171/13
	6 CC이하	348/77	293/47	295/46	295/44	287/35	289/28
	6 CC초과	92/21	110/21	108/21	108/22	85/20	69/19

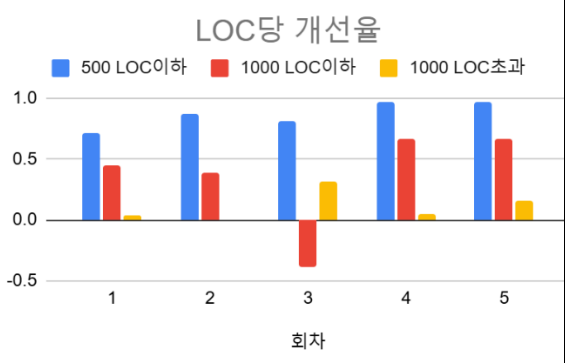
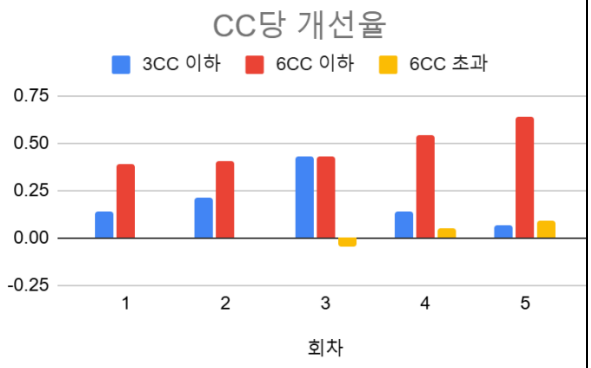
여기서 주요 이슈의 개선율을 시각화 하면 아래 차트와 같다.

	
(a) LOC당 개선율	(b) CC당 개선율
GPT-5	

- **LOC 당 개선율:** GPT-5 는 모든 회차와 모든 LOC 구간(500 이하, 1000 이하, 1000 초과)에서 **안정적인 양(+)의 개선율**을 보였다. 특히 코드 규모가 작을수록(500 LOC 이하) 개선율이 가장 높은 일관된 패턴을 나타냈다. 이는 GPT-5 가 파일 크기와 관계없이 코드 품질을 악화시키지 않는 보수적이면서도 신뢰도 높은 리팩토링을 수행함을 의미한다.
- **CC 당 개선율:** 복잡도 측면에서도 GPT-5 는 모든 구간에서 꾸준히 긍정적인 개선율을 보였다. 복잡도가 낮은 코드(3CC 이하)에 대한 개선율이 가장 높게 나타나, 상대적으로 단순한 구조의 코드에서 이슈를 확실하게 제거하는 데 강점을 보인다.

	
(a) LOC당 개선율	(b) CC당 개선율
Gemini 2.5 pro	

- **LOC 당 개선율:** Gemini 2.5 pro 는 대부분의 구간에서 높은 개선율을 보였지만, 눈에 띄는 변동성을 나타냈다. 특히 1000 LOC 초과 파일에 대해 3, 4 회차에서 개선율이 급격히 낮아지는 등, 대규모 파일에 대한 연속 리팩토링 시 성능의 일관성이 다소 부족한 모습을 보였다.
- **CC 당 개선율:** 가장 주목할 만한 결과로, **복잡도가 높은(6CC 초과) 코드에 대해 3, 4 회차에서 개선율이 큰 폭의 음수(-)** 를 기록했다. 이는 Gemini 2.5 pro 가 복잡한 코드를 연속적으로 수정하는 과정에서 오히려 새로운 이슈를 유발하여 **코드 품질을 악화시킬 수 있는 잠재적 위험**을 내포하고 있음을 시사한다. 하지만 5 회차에서는 다시 높은 양의 개선율로 회복하여, 특정 조건에서 성능 편차가 크게 나타나는 특성을 보였다.

<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>(a) LOC당 개선율</td><td>(b) CC당 개선율</td></tr>
<tr><td colspan="2">Claude Opus 4.1</td></tr>
</table>

- **LOC 당 개선율:** Claude Opus 4.1 은 1000 LOC 이하의 중간 크기 파일에 대한 리팩토링 3 회차에서 개선율이 음수(-) 로 떨어지는 현상을 보였다. 이는 특정 크기의 파일에 대해 반복적인 수정을 시도할 때, 오히려 코드 구조를 해칠 수 있음을 의미한다. 반면, 500 LOC 이하의 작은 파일에 대해서는 꾸준히 높은 성능을 유지했다.
- **CC 당 개선율:** 복잡도 측면에서는 비교적 안정적이었으나, 복잡도가 높은(6CC 초과) 코드에 대한 개선율이 다른 구간에 비해 현저히 낮게 나타났다. 이는 Claude Opus 4.1 이 매우 복잡한 논리 구조를 가진 코드의 연속 개선에는 상대적으로 약점을 가질 수 있음을 보여준다.

5. 결론 및 향후 연구 방향

5.1. 결론

본 졸업과제는 LLM 을 활용하여 C 언어 기반의 대규모 오픈소스 소프트웨어의 유지보수성을 개선하는 자동화된 파이프라인을 구축하고, 그 성능을 정량적으로 분석하였다. 실험을 통해 도출된 핵심 결론은 다음과 같다.

첫째,

LLM 은 코드의 유지보수성을 유의미하게 개선할 수 있으나, 그 효과는 이슈의 종류에 따라 다르게 나타났다. 세 가지 LLM 모델 모두 심각도가 높은 주요 이슈(Blocker, High)를 제거하는 데에는 뚜렷한 성능을 보였으며, 이는 LLM 이 코드의 안정성과 신뢰도를 높이는 데 실질적으로 기여할 수 있음을 증명한다. 하지만 모든 종류의 이슈를 포함한 전체 개선율은 0 에 수렴하는 결과를 보였다. 이는 LLM 이 주요 이슈를 해결하는 과정에서 새로운 부차적인 이슈를 발생시키는 '개선 효과의 트레이드오프(Trade-off)'가 존재함을 시사한다.

둘째,

코드의 특성은 LLM 의 리팩토링 성능에 비선형적인 영향을 미친다. 코드 라인 수(LOC)의 경우, 약 2000 LOC 근방의 파일에서 성능이 급격히 저하되며 오히려 코드 품질을 악화시키는 '임계점'이 관찰되었다. 또한, 순환 복잡도(CC)의 경우, 5CC 이하의 매우 단순한 코드는 불필요한 수정으로 품질이 악화될 수 있으며, 반대로 10CC 이상의 복잡한 코드일수록 LLM 의 개선 효과가 극대화되는 경향을 확인했다. 이는 LLM 을 활용한 리팩토링 전략 수립 시, 코드의 특성을 고려한 선별적 접근이 중요함을 보여준다.

셋째,

반복적인 리팩토링은 초기에 가장 효과적이며 점차 수렴하는 패턴을 보인다. 자체 설계한 에이전트를 통한 연속 개선 실험에서, 모든 모델은 첫 번째 리팩토링 시도에서 가장 많은 수의 주요 이슈를 제거했으며, 회차가 거듭될수록 개선 효과는 점차 감소했다. 특히, 깊은 중첩 구조와 같이 인지 복잡도가 매우 높은 코드는 LLM 이 잠재적인 동작 변경의 위험을 인지하고 수정을 거부하는 경향을 보여, 현재 기술의 명확한 한계를 확인할 수 있었다.

종합적으로, 본 졸업과제는 LLM 이 C 언어 레거시 코드의 유지보수성을 개선하는 강력한 도구가 될 수 있음을 입증하였다. 특히, 복잡하고 위험성이 높은 코드를 선별하여 개선하는 데 가장 큰 가치를 제공하며, gemini-2.5-pro 와 claude-opus-4-1 모델이 안정적이고 뛰어난 성능을 보였다.

5.2. 향후 연구 방향

본 연구의 발견을 바탕으로 다음과 같은 후속 연구를 제안한다.

1. 기능적 정확성 검증 시스템 통합: 현재 연구는 SonarQube Cloud 의 정적 분석 지표에만 의존하였다. 리팩토링된 코드가 원본 코드와 기능적으로 동일하게 동작하는지를 보장하기 위해, 자동화된 단위 테스트(Unit Test) 및 회귀 테스트(Regression Test)를 파이프라인에 통합하여 의미론적 동등성을 검증하는 연구가 필수적이다.
2. 에이전트 기반의 다단계 리팩토링 전략: 단일 프롬프트의 한계를 넘어, 자율적으로 판단하고 행동하는 AI 에이전트(AI Agent)를 고도화하는 연구가 필요하다. 예를 들어, 1 단계에서 코드를 분석하여 리팩토링 전략(예: 함수 분리, 패턴 적용)을 수립하고, 2 단계에서 전략에 따라 코드를 수정한 뒤, 3 단계에서 자체적으로 빌드 및 테스트를 수행하여 결과를 검증하는 다단계 워크플로우를 구현할 수 있다. 이는 현재 LLM 이 수정을 포기하는 고복잡도 코드 문제에 대한 해결책이 될 수 있다.
3. 인간-AI 협업 모델(Human-in-the-Loop) 연구: 완전 자동화를 넘어, LLM 이 여러 리팩토링 후보안을 제시하고 인간 개발자가 최종적으로 최적의 안을 선택하거나 수정하는 협업 모델을 연구할 수 있다. 이는 LLM 의 창의적인 제안과 인간의 깊은 도메인 지식을 결합하여, 더 안전하고 효과적인 코드 개선을 가능하게 할 것이다.
4. 아키텍처 수준의 리팩토링으로 확장: 현재 연구는 파일 단위의 코드 개선에 머물러 있다. 향후에는 여러 파일과 모듈 간의 의존성을 분석하여, 결합도를 낮추고 응집도를 높이는 등 소프트웨어 아키텍처 수준의 거시적인 리팩토링을 수행하는 LLM 의 능력에 대한 연구로 확장할 수 있다.

6. 참고 문헌

[i] Language Models are Few-Shot Learners

, https://arxiv.org/abs/2005.14165, 2020년 7월 22일

[ii] The true cost equation: Software development and maintenance costs explained,

https://idealink.tech/blog/software-development-maintenance-true-cost-equation, 2025년 5월 7일

[iii] Language Models are Few-Shot Learners, https://arxiv.org/abs/2005.14165, 2020년 5월 28일

[iv] The true cost equation: Software development and maintenance costs explained,

https://idealink.tech/blog/software-development-maintenance-true-cost-equation, 2025년 5월 7일

[v] Gemini 2.5 Pro vs GPT-4.5, https://www.edenai.co/post/gemini-2-5-pro-vs-gpt-4-5

[vi] GPT-5를 소개합니다, https://openai.com/ko-KR/index/introducing-gpt-5/, 2025년 8월 7일

[vii] 클로드 Opus 4 & Sonnet 4 출시: AI 지능 및 성능의 새로운 시대, https://apidog.com/kr/blog/claude-opus-4-sonnet-4-kr/, 2025년 8월 6일

[viii] Unleashing the potential of prompt engineering for large language models, https://www.sciencedirect.com/science/article/pii/S2666389925001084, 2025년 6월 13일

[ix] Evaluating the Dependency Between Cyclomatic Complexity and Response For Class

, https://arxiv.org/html/2410.06416v1, 2024년 10월 8일