<PROJECT 최종보고서**>**

LLM을 활용한 코드 자동 생성 및 변환 프로그램 개발

2019-14593 박현

2019-14536 배준익

엄현상 교수님

계정민 담당자님 **(**인스웨이브**,** 개발팀**)**

Table of Contents

1. Abstract

인스웨이브 사의 Java 프레임워크인 **Proworks 4**에서 **Proworks 5**로의 마이그레이션 과정은 현재 수작업에 의존하고 있어 작업 효율이 낮고, 코드 오류 발생 가능성이 높다는 문제를 안고 있다. 본 프로젝트는 이러한 문제를 해결하기 위해 **LLM** 기반의 코드 변환 자동화 도구를 개발하는 것을 목표로 한다. LLM의 코드 이해 및 생성 능력을 활용하여 Proworks 버전 간의 문법적·구조적 차이를 자동으로 변환함으로써, 마이그레이션 시간과 비용을 절감하고 코드 품질을 향상시키는 것을 목표로 한다.

이를 위해 세 가지 핵심 모듈을 설계하고 구현하였다. 첫째, **Dependency Analysis** 모듈은 Proworks 5에서 요구하는 데이터 클래스와 스켈레톤 코드를 자동으로 생성하며, 정적 분석 기반의 Call Graph Parsing을 통해 구현된다. 이를 통해 기존 코드베이스의 함수 및 객체 간 의존 관계를 분석하고, 변환 준비 단계에서의 자동화를 달성하였다. 둘째, **Code Conversion** 모듈은 LLM을 활용하여 레거시 코드를 신규 API에 맞게 자동 변환한다. 특히 **RAG(Retrieval-Augmented Generation)** 기법을 통해 Proworks 5의 공식 문서나 사내 코드 레퍼런스를 실시간으로 참조함으로써 변환 정확도를 높였다. 셋째, **Self-Refinement** 모듈은 초기 변환 결과물에 포함된 오류를 LLM 스스로 수정하도록 하는 Agentic 접근을 따른다. Reflexion 기법을 기반으로 LLM이 자신의 출력을 분석·피드백하고, 다단계 과정을 통해 점진적으로 코드의 완성도를 높일 수 있도록 하였다.

이러한 세 모듈의 통합을 통해 수작업 대비 높은 자동화율과 코드 품질을 확보할 수 있었음을 실험을 통해 보였다. 무엇보다도, 기존 룰 기반의 변환보다 20% 이상의 성능 향상을 보여주어 전체적인 마이그레이션 리소스를 크게 줄일 수 있었음을 보였다. 더불어 향후 다른 프레임워크 간 변환이나 버전 업그레이드 시에도 활용 가능한 범용적인 코드 변환 시스템의 가능성을 보여주었으며, 실제 산업 현장에서의 확장성과 응용 가능성 또한 입증하였다.

2. Introduction

기업용 애플리케이션 개발에 특화된 인스웨이브 사의 Java 기반 프레임워크인 Proworks는, 안정적인 백엔드 구축과 운영을 위해 다양한 기능을 제공하고 있다. 최근 출시된 Proworks 5는 이전 버전인 Proworks 4에 비해 API 확장, 인터페이스 정비, 성능 개선, 유지보수 편의성 등 다방면에서 향상된 기능을 포함하고 있으며, 기업 시스템의 현대화를 위한 중요한 업그레이드로 평가받고 있다. 그러나 이러한 기능 향상은 코드 구조 및 사용 방식의 근본적인 변화로 이어져, 기존 시스템과의 호환성 문제를 야기하고 있으며, 그로 인해 마이그레이션 과정에서의 높은 기술적·관리적 부담이 발생하고 있다.

우리 R조는 이러한 마이그레이션 문제를 면밀히 분석한 결과, 현업에서 다음과 같은 세 가지 주요 Pain Point를 발견하였다.

첫 번째 **Pain Point**는 규칙 기반(Rule-based) 코드 변환 방식의 구조적 한계이다. 기존

마이그레이션 작업은 대부분 정해진 치환 규칙을 기반으로 이루어지며, 단순 패턴 매칭에는 효과적일 수 있으나 실무에 존재하는 다양한 코드 스타일과 예외 상황을 포괄하지 못한다. 특히 실환경의 코드는 다수의 개발자가 축적한 비정형적 코드, 중첩 로직, 복합 조건문 등으로 구성되어 있으며, 이러한 코드 특성은 단순한 규칙으로 설명할 수 없어 수작업 개입이 불가피해진다. 결과적으로 개발자의 반복 작업과 휴먼 에러가 빈번하게 발생하며, 전체 생산성이 저하되고 코드 품질도 보장되지 않는다.

두 번째 **Pain Point**는 문맥 기반 구조적 변환의 비효율성이다. 기존 규칙 기반 시스템은 대부분 1:1 치환에 최적화되어 있어, 예를 들어 복수의 MAP 객체를 하나의 VO 객체로 통합하거나, 조건문 내 API 호출을 상황에 맞게 조정하는 등의 고차원적 변환에는 대응하지 못한다. 이러한 변환은 코드의 의미적 이해와 문맥 분석이 수반되어야 하므로, 단순 치환이 아닌 이해 기반의 생성적 변환이 필요하다.

세 번째 **Pain Point**는 마이그레이션 전 과정에 걸쳐 수작업이 과도하게 발생하며, 전체 일정과 품질 관리에 부담이 크다는 점이다. 변환 실패 시 복구 자동화가 불가능하고, 모든 실패 구간은 개발자의 수작업으로 보완해야 한다. 또한 변환 이후에도 검수 작업이 필수이며, 이 역시 개발자의 시간과 리소스를 소모하게 된다. 이로 인해 마이그레이션 일정은 반복적으로 지연되고, 이는 곧 시스템 안정성과 운영 효율성 저하로 직결된다. 이러한 문제들을 해결하기 위해 우리 R조는 최근 급속히 발전 중인 대규모 언어 모델(LLM, Large Language Model) 기술에 주목하였다. LLM은 기존의 정적 규칙을 넘어 코드의 문맥과 구조를 이해하고, 예외 상황까지 유연하게 처리할 수 있는 능력을 갖추고 있다. 이를 활용하면 기존 방식으로는 처리할 수 없었던 구조적·논리적 코드 변환을 자동화할 수 있으며, 반복 작업을 줄이고 오류 가능성을 크게 낮출 수 있다.

본 프로젝트의 목표는 LLM을 기반으로 한 Proworks 4에서 5로의 자동 마이그레이션 도구를 개발하는 것이다. 이를 위해 다음의 세 가지 핵심 모듈을 구현하였다. 첫째, Dependency Analysis 모듈은 기존 시스템의 함수 호출 관계를 분석하여 Proworks 5에 적합한 스켈레톤 코드와 데이터 클래스를 자동으로 생성한다. 이는 call graph parsing 기반으로 구현된다. 둘째, Code Conversion 모듈은 LLM을 통해 레거시 함수와 Proworks 5 API를 매핑하며, RAG(Retrieval-Augmented Generation) 기법을 활용하여 외부 문서나 코드 레퍼런스를 실시간으로 참조함으로써 변환 정확도를 높인다. 셋째, Self-Refinement 모듈은 LLM의 초기 출력이 불완전하거나 오류를 포함할 경우, 이를 LLM이 스스로 수정하도록 유도하는 Agentic 방식을 적용하였다. 이 과정에서는 Reflexion 방법론을 활용하여 LLM이 자기 출력을 재검토하고 개선하도록 설계하였다. 이러한 세 모듈을 기반으로 본 시스템은 기존 개발자의 수작업을 대체하고, 서버 개발자가 검토와 고도화 작업에 집중할 수 있도록 구조를 재편하였다. 이를 통해 마이그레이션의 효율성과 안정성을 동시에 확보할 수 있었으며, 결과적으로 전체 일정 단축, 비용 절감, 코드 품질 향상이라는 성과를 달성하였다. 나아가 본 프로젝트는 향후 다양한 프레임워크 전환과 코드 리팩토링 분야에 적용 가능한 확장성과 범용성을 지니며, 기업의 기술 경쟁력을 강화하는 핵심 도구로 자리매김할 수 있을 것으로 기대된다.

3. Background Study

A. 관련 접근방법/기술 장단점 분석

기존의 마이그레이션 자동화 방식은 주로 규칙 기반(rule-based) 접근을 따르고 있다. 이 방식은 일정한 패턴이 반복되는 단순한 변환 작업에서는 높은 정확도와 안정성을 제공할 수 있으나, 문맥이나 구조를 고려한 복잡한 변환에서는 명확한 한계를 드러낸다. 특히 실무 환경에서는 다양한 예외 상황과 중첩된 조건문, 일관되지 않은 코드 스타일이 빈번하게 존재하며, 이러한 경우를 모두 포괄하기 위해서는 규칙의 수가 기하급수적으로 증가하게 된다. 이로 인해 유지보수 비용이 높아지고, 규칙 누락에 따른 변환 실패가 발생했을 때는 반드시 수작업 보완이 수반되어야 하는 구조적인 제약이 존재한다.

이에 반해 최근 주목받고 있는 LLM(Large Language Model) 기반 접근 방식은 코드의 문법적 구조뿐 아니라 의미와 문맥까지 파악할 수 있는 장점을 지닌다. LLM은 복잡한 조건문, 비정형 구조, 예외 상황에 대응하는 유연한 코드 변환 능력을 갖추고 있으며, 반복적인 수작업의 부담을 줄이고 개발 리소스를 절감할 수 있다. 그러나 LLM의 출력은 항상 일관되지는 않으며, 도메인 지식이 부족한 경우 예기치 않은 오류나 중요한 내용 누락이 발생할 수 있다. 이러한 점에서 완전한 대체보다는 상호 보완적인 활용이 현실적인 대안으로 제시된다.

이에 따라 본 프로젝트에서는 규칙 기반 방식과 LLM 기반 방식을 결합한 하이브리드 전략을 채택하였다. 전처리 및 후처리 영역은 비교적 안정적인 규칙 기반으로 처리하고, 핵심적인 구조 변환은 LLM에 위임하는 방식을 통해 양쪽의 장점을 극대화하고자 하였다. 다만 두 방식 간 역할 분담과 상호 연결 체계는 정교하게 설계되어야 하며, 이에 따른 구현 복잡도와 유지관리 이슈는 별도로 고려해야 할 과제로 남는다.

이와 같은 기술 선택 배경은 관련 선행연구로부터도 영향을 받았다. 예를 들어, LLM 기반 코드 변환의 정확성과 일관성을 검증한 Verified Code Transpilation with LLMs[1] 연구와, ChatGPT를 활용한 코드 개선 가능성을 분석한 Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study[2] 등은 본 프로젝트의 평가 기준 설정에 있어 참고 자료로 활용되었다. 특히 두 연구는 Exact Match와 BLEU score 같은 정량 지표를 활용하여 코드 품질을 객관적으로 측정하였으며, 이는 본 프로젝트의 평가 설계와 정합성을 높이는 데 기여하였다. 또한, Shin et al.의 Reflexion: Language Agents with Verbal Reinforcement Learning과 Franzen et al., Product of Experts with LLMs: Boosting Performance on ARC Is a Matter of Perspective은 각각 self-refinement 모듈과 programmable converter의 아이디어를 얻는 데에 도움을 주었다.

---

[1] Bhatia, Sahil, et al. "Verified code transpilation with LLMs." *Advances in Neural Information Processing Systems* 37 (2024): 41394-41424.

[2] Guo, Qi, et al. "Exploring the potential of chatgpt in automated code refinement: An empirical study."*Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 2024.

B. 프로젝트 개발환경

본 프로젝트는 LLM 기반 코드 변환 자동화 도구의 개발을 목표로 하며, 전체 시스템은 Python 언어를 기반으로 구현되었다. Python은 자연어 처리와 머신러닝 라이브러리와의 연동성이 뛰어나고, 빠른 프로토타이핑과 유연한 통합 설계가 가능하여 본 프로젝트의 요구사항에 부합하는 언어로 선택되었다. 변환 대상은 Java 1.8 기반의 Spring 프레임워크로 구성된 Proworks 4 및 Proworks 5 코드이며, 실제 환경에서는 기업 보안 정책으로 인해 외부 클라우드 LLM API의 사용이 제한된다. 이에 따라 로컬 환경에서 모델을 실행하는 구조로 설계하였다. 개발 초기의 실험 단계에서는 모델 로딩과 테스트의 유연성을 확보하기 위해 Ollama를 활용하였으며, 고성능 GPU 상에서의 빠른 응답성과 메모리 효율성을 확보하였다.

문맥 기반 코드 변환 정확도를 향상시키기 위해 RAG(Retrieval-Augmented Generation) 구조를 도입하였고, 검색 인덱싱에는 Faiss를 활용하여 코드 문서, 스펙, 예제 등을 벡터화하고, 관련 정보를 실시간으로 LLM에 제공할 수 있도록 하였다. 코드 분석 단계에서는 Java의 정적 구조 분석을 위해 Tree-sitter Java 파서를 사용하였다. Tree-sitter는 정밀한 구문 트리 생성을 통해 코드 구조 이해도를 높이는 데 기여하였다. LLM 입력을 정제하고 구조화하는 데 효과적으로 활용되었다.

전체 시스템은 FastAPI 기반의 RESTful 서버로 구성되었으며, LLM 서빙과 클라이언트 요청 간의 통신을 효율적으로 처리하도록 설계되었다. 사용자 인터페이스는 VS Code 확장 기능 형태로 구현되었으며, 이를 위해 기존의 llm-vscode 오픈소스 프로젝트를 참고하여 프로젝트 요구에 맞는 사용자 경험(UI/UX)을 설계하고 자체적으로 기능을 구성하였다.

프로젝트는 2인 팀으로 구성되어 진행되었으며, Git을 기반으로 한 협업 및 버전 관리 체계를 운영하였다. GitHub를 원격 저장소로 사용하여 브랜치 전략, 이슈 관리, Pull Request 기반 코드 리뷰 등을 병행하였고, 이를 통해 협업 효율성과 코드 품질을 함께 확보할 수 있었다.

4. Goal/Problem & Requirements

본 프로젝트의 목표는 대규모 언어 모델(LLM)을 활용하여, 기존 Java 기반의 Proworks 4 코드를 Proworks 5 구조로 자동 변환하는 도구를 개발하는 것이다. 단순한 정규표현식 기반의 치환을 넘어서, 다대다 매핑이 포함된 고차원적인 구조 변환을 자동화하는 것을 지향한다. 이를 통해 코드 변환 개발자의 반복적 개입을 최소화하고, 서버 개발자는 검토 및 고도화 작업에 집중할 수 있는 작업 흐름을 구축하는 것이 최종 목표이다. 본 프로젝트에서는 전체 대상 코드 중 50% 이상의 자동 변환 정확도(Exact Match Score) 달성을 정량적 목표로 설정하였다.

현행 Proworks 4에서 Proworks 5로의 마이그레이션 과정에서 주요 장애 요인은 문맥 및 구조 복잡성이 높은 코드 영역에서 주로 발생한다. 특히, Map 기반의 유연한 파라미터 전달 방식을, 명시적 타입 정의와 필드 기반 구조를 요구하는 Value Object(VO) 패턴으로 전환하는 작업은 대표적인 어려움 중 하나다. 이때 단순히 SQL 매핑만 분석해서는 충분하지 않으며, 해당 로직에 포함된 조건문, 내부 분기, 비즈니스 규칙 등의 맥락까지 해석해야 한다. 따라서 이 문제는

정형화되지 않은 실무 코드에 대해 의미 기반 추론이 가능한 수준의 코드 분석 및 생성 능력을 요구한다.

이러한 문제 해결을 위해 본 시스템은 다음과 같은 기능적 요구사항을 충족해야 한다. 첫째, 입력으로는 Proworks 4의 Java 코드 및 SQL 매핑 XML 파일을 받아야 하며, 출력은 Proworks 5의 디자인 패턴에 부합하는 형식의 코드로 자동 생성되어야 한다. 둘째, Map 기반 → VO 기반 전환, 다대다 매핑 구조 대응, 중첩 조건문 내부의 API 변환, 명세 기반 클래스 필드 생성 등의 고차원 구조 변환을 지원해야 한다. 셋째, 변환이 불완전하거나 실패한 경우에는 해당 위치에 명시적인 주석 또는 로그를 자동 삽입하여, 사용자의 사후 보완 작업을 용이하게 해야 한다.

비기능적 요구사항으로는, GPU VRAM 40GB 이하의 로컬 환경에서 안정적으로 실행 가능해야 하며, 고객사의 보안 정책을 고려해 인터넷 연결 없이 폐쇄망에서도 완전한 작동이 가능해야 한다. 또한, 모든 코드와 모델은 상업적 라이선스로 사용 가능한 범위 내에서만 구성되어야 하며, 외부 API 호출 없이 독립적으로 실행 가능한 구조여야 한다.

본 프로젝트는 이러한 기능적 및 비기능적 요구사항을 충족함으로써, 기업 실무 환경에서도 적용 가능한 안정적이고 확장 가능한 코드 자동 변환 도구로서의 가능성을 검증하고자 한다.

5. Approach

본 프로젝트는 단순한 문자열 치환 방식이 아닌, 구조적·문맥적 이해를 바탕으로 한 고차원적인 코드 생성 메커니즘을 목표로 하며, Proworks 5의 설계 패턴 및 아키텍처 특성을 고려한 일련의 처리 과정을 설계하였다.

우선, 변환 대상 시스템의 구조적 특성을 고려하여, 기존의 Map 기반 동적 데이터 전달 방식을 Value Object(VO) 기반의 정적 구조로 전환한다. 이를 통해 명시적 타입 정보에 기반한 안정성과 유지보수성을 확보할 수 있으며, 자동 생성된 코드의 품질과 신뢰성 또한 향상된다.

전체 코드 변환은 일괄적으로 처리하는 방식이 아니라, 계획 기반 접근 방식**(Planning Approach)**을 채택하여 순차적으로 진행된다. 구체적으로는 먼저 Callgraph 분석을 통해 함수 간의 호출 관계를 파악하고, 이를 기반으로 상위 수준의 스켈레톤 코드를 자동 생성한 후, LLM을 이용하여 각 함수의 내부 로직을 채워나가는 방식이다. 이러한 절차적 분할은 LLM의 환각(hallucination) 문제를 줄이는 데 효과적이며, 실제 코드 의미와 구조의 일치도를 높이는 데 유리하다고 판단하였다.

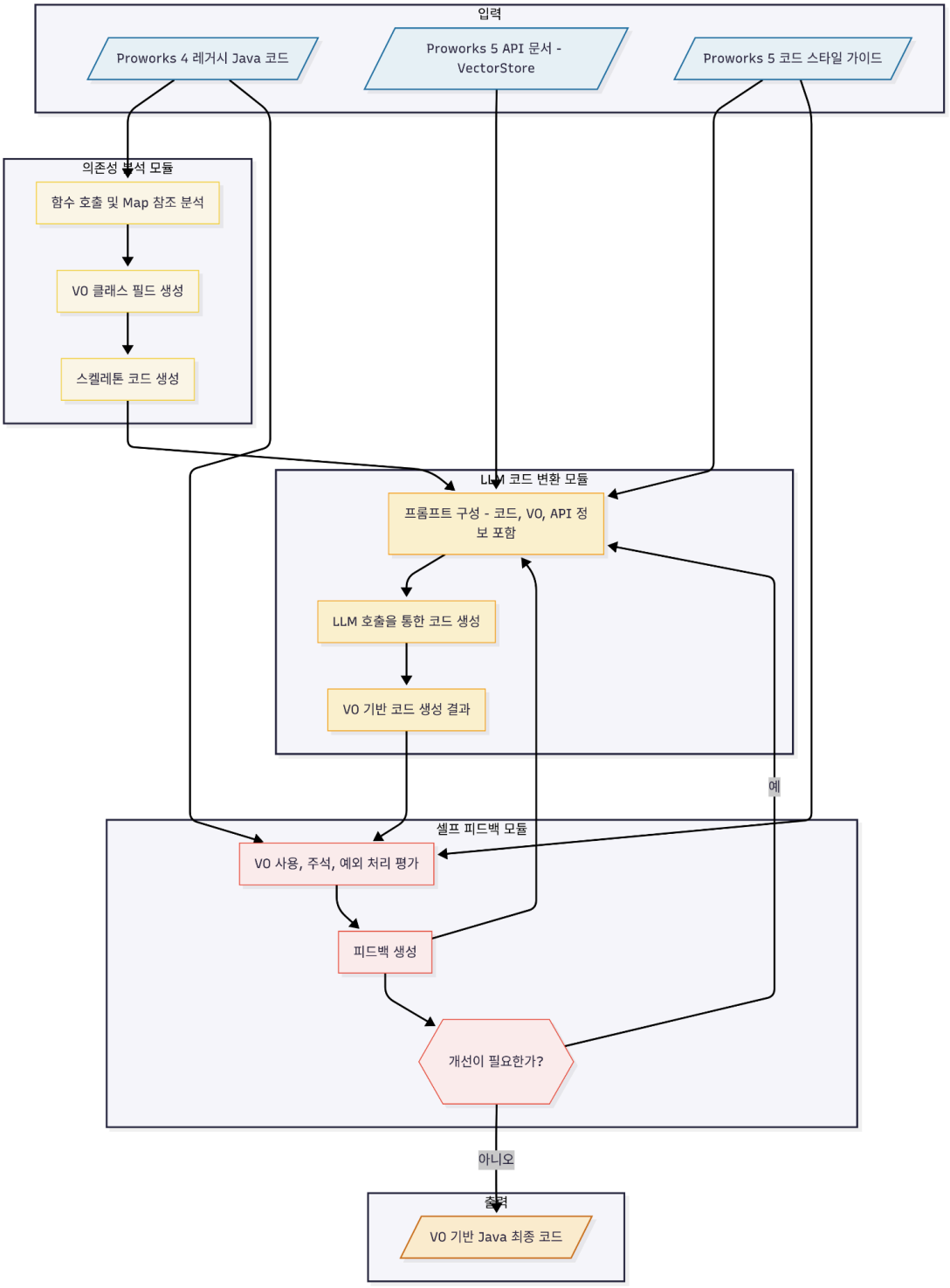
또한, LLM의 출력 품질 향상을 위해 RAG(Retrieval-Augmented Generation) 구조를 도입하고, 내부 코드 레퍼런스 및 문서를 검색하여 실시간으로 반영할 수 있도록 하였다. 여기에 더해, LLM이 자신의 출력을 스스로 평가하고 개선하는 Self-feedback 기법을 함께 적용하여, 반복적인 검토와 수정을 통한 코드의 정확도 향상을 도모하고 있다.

이러한 접근은 구조적 일관성을 유지하면서도 유연한 대응이 가능하다는 점에서, 실무 현장의

다양한 코드 형태와 요구사항에 효과적으로 대응할 수 있을 것으로 기대된다.

6. Project Architecture

A. Architecture Diagram



B. Architecture Description

본 프로젝트의 아키텍처는 기존 레거시 Java 코드에서 Proworks 5의 VO(Value Object) 기반 구조로 자동 변환을 수행하기 위한 전체 시스템의 처리 흐름을 중심으로 설계되었다. 이 시스템은 함수 단위의 구조적 분석과 프롬프트 생성, 코드 생성, 피드백 및 반복 개선으로 구성되며, 각 단계는 신뢰성과 정밀도를 확보하는 데 중점을 두고 있다.

우선, 입력으로 주어지는 레거시 Java 코드는 함수 시그니처와 본문을 기준으로 파싱되며, 각 함수 간의 호출 관계(Call Graph)를 분석하여 전체적인 호출 흐름과 의존 구조를 추출한다. 이 과정에서 특정 함수가 호출하는 다른 함수, 사용되는 매개변수, 반환값 등의 정보를 정적으로 수집하고, 함수 간 연관성을 구조화된 형태로 표현한다. 최종적으로 Map을 사용하는 변수를 추출하고 필드명을 파악하여 **VO** 데이터 클래스를 생성한다.

이후, 분석된 호출 관계 및 코드 구조 정보를 바탕으로 **LLM** 입력을 위한 프롬프트가 구성된다. 프롬프트에는 해당 함수의 시그니처, 기존 구현 내용, 자동 생성된 데이터 클래스(VO), 관련 API 스펙, 호출된 함수의 컨텍스트, 과거 생성 결과에 대한 피드백 정보 등이 포함된다. 이와 같은 풍부한 맥락 정보를 바탕으로, LLM은 VO 기반의 새로운 함수 구현을 생성한다.

생성된 코드는 바로 사용되지 않고, **Feedback LLM**에 전달되어 검토 과정을 거친다. 이 과정에서는 동작 보존 여부, 주석 유지 여부, VO 패턴 준수, 런타임 오류 가능성 등의 기준에 따라 피드백이 제공된다. 피드백 결과는 다시 초기 프롬프트에 반영되어, 반복적인 개선 루프를 통해 코드의 품질과 신뢰성을 점진적으로 향상시킨다.

또한, 코드 생성 시 정확도와 일관성을 높이기 위해 시스템은 Proworks 문서에서 추출한 API 정보를 사전에 벡터화하여 VectorStore에 저장하고 있으며, 이를 기반으로 **RAG(Retrieval-Augmented Generation)** 기법을 적용한다. 이를 통해 생성 과정에서 필요한 API 사용 예시나 형식을 실시간으로 참조할 수 있게 되어, LLM이 보다 정확한 변환을 수행할 수 있도록 지원한다.

결과적으로, 이 아키텍처는 정적인 구조 분석, 풍부한 프롬프트 구성, LLM 기반 생성, 2차 검토 및 반복 개선, 외부 문서 기반 보조 정보를 유기적으로 결합함으로써, 신뢰도 높고 구조적으로 일관된 코드 전환을 함수 단위로 점진적으로 수행하는 체계를 제공한다.

7. Implementation Spec

A. Input/Output Interface

1) Input

시스템의 주요 입력은 크게 세 가지로 구성된다. 첫째, 변환 대상이 되는 Proworks 4 기반의 레거시 Java 코드이다. 주로 Map 객체를 중심으로 동적 데이터 처리를 수행하는 구조를 갖는다. 둘째, Proworks 5에 해당하는 공식 API 문서 파일이다. 해당 문서는 코드 생성 시 RAG(Retrieval-Augmented Generation)의 지식 기반으로 활용되며, 필요한 API 예시 및 사용 방식에 대한 정보를 LLM에 제공한다. 셋째, Proworks 5의 디자인 패턴 및 코드 스타일 가이드 문서이다. 해당 문서는 하드코딩된 형태로 시스템 내에 내장되어 있으며, 변환 결과가 해당 스타일 가이드에 부합하도록 LLM의 출력 형식을 정규화하는 기준으로 사용된다. 이 문서는 본 프로젝트 진행 중에는 변경되지 않으며, 기준점으로 고정되어 있다.

입력으로 제공되는 레거시 코드는 실험 및 평가 목적에 따라 두 가지 규모로 구분된다. 일반 평가용 코드는 약 100줄 내외의 파일 4~5개로 구성되어 있으며, 전형적인 Map 기반 데이터 전달, 조건문 분기, 단일 쿼리 중심의 컨트롤러/서비스 구조를 포함한다. 고급 평가용 코드(Advanced Set)는 약 1000줄 분량으로, 다수의 중첩된 조건문, 복잡한 함수 호출, 다대다 Map 구조 등이 포함되어 있어 구조적·문맥적 변환의 정확성을 평가하는 데 사용된다.

2) Output

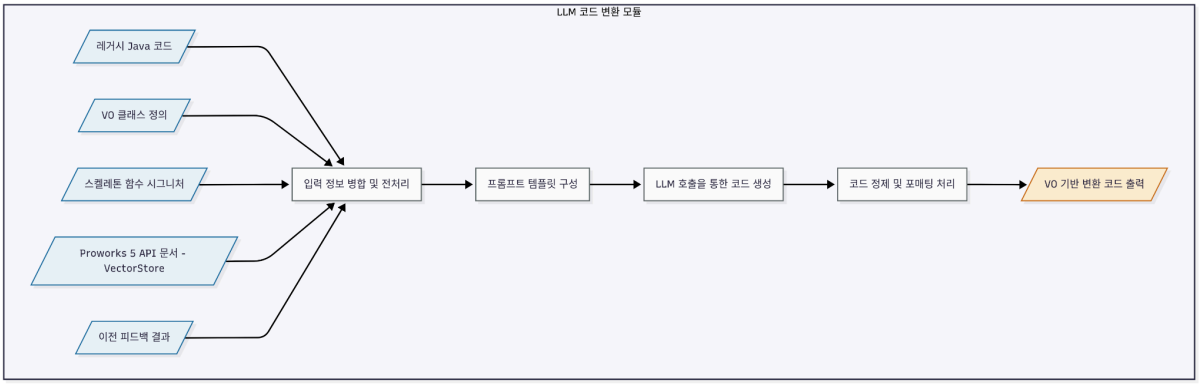
시스템의 출력은 Proworks 5의 구조에 부합하는 Java 코드이다. 이 코드는 기존 Proworks 4 기반의 Map 중심 로직을 VO(Value Object) 기반의 정적 구조로 재작성한 함수 단위 코드로, Proworks 5에서 요구하는 표준 코드 패턴을 준수한다. 변환 결과는 LLM에 의해 자동 생성되며, 평가 및 디버깅을 돕기 위해 일부 코드에 자동 주석이 삽입될 수 있다. 생성된 출력은 중간 결과로 간주되며, 이후 Feedback LLM을 통해 자가 피드백 과정을 거친다. 이 과정을 통해 더 이상의 개선점이 없다는 판단이 내려진 경우, 해당 코드가 최종 결과물로 확정된다.

B. Modules

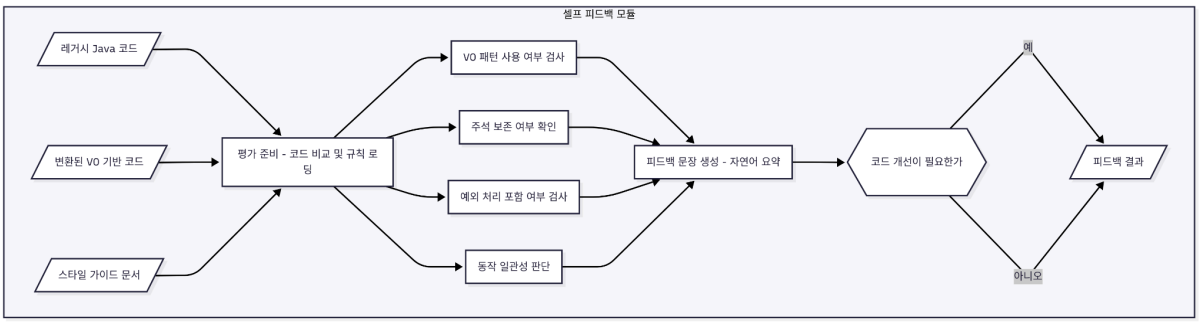
본 프로젝트는 총 세 개의 핵심 모듈로 구성되며, 각각은 Proworks 4 레거시 코드에서 Proworks 5의 정적 타입 기반 VO(Value Object) 구조로의 전환을 목표로 하는 변환 시스템 내 주요 기능을 담당한다. 모듈은 순차적으로 실행되며, 각 단계에서의 출력은 다음 단계의 입력으로 활용된다. 전체 시스템은 Dependency Analysis, LLM Code Conversion, Self-Feedback의 세 가지 모듈로 구성된다.

첫 번째 모듈인 Dependency Analysis는 자바 코드 내 함수 간 호출 관계(Call Graph)를 기반으로 의존성을 분석한다. 이를 위해 javalang 라이브러리를 활용하여 자바 소스 코드를 파싱하고, 각 함수 간의 호출 관계 및 데이터 흐름을 추출한다. 특히 Map 객체를 통해 데이터베이스로부터 값을 조회하는 로직을 중심으로 분석하며, 그 결과 함수 내에서 key-value 형태로 사용되는 필드를 식별한다. 분석된 필드 정보는 이후 생성될 VO 클래스의 필드로 전환되며, 이를 기반으로 Proworks 5 스타일의 정적 데이터 클래스를 자동 생성한다. 이후 이러한 DataClass를 활용할 수 있도록 기존

함수 시그니처를 수정하고, 전체 변환 대상 함수들의 스켈레톤 코드를 구성한다. 이로써 후속 단계에서 LLM이 기능적 문맥을 파악하고 안정적으로 코드를 생성할 수 있는 기반이 마련된다.



두 번째 모듈인 LLM Code Conversion은 앞선 분석 모듈에서 생성된 정보들을 입력으로 받아 실제 코드 변환을 수행하는 단계이다. 입력으로는 Proworks 4 레거시 코드, Dependency Analysis 단계에서 생성된 스켈레톤 코드(converted schema), 자동 생성된 DataClass가 포함된다. 변환 대상 함수의 의미를 보다 명확히 전달하기 위해 해당 함수가 호출하는 callee 함수들의 기존 구현도 프롬프트에 포함된다. 이외에도 RAG(Retrieval-Augmented Generation) 기능이 활성화된 경우, 변환 대상 코드를 query로 사용하여 Proworks 5의 API 문서에서 관련 내용을 검색하고 해당 정보를 프롬프트 컨텍스트에 삽입한다. Self-Feedback 모듈이 활성화된 상태에서는 이전에 생성된 코드와 이에 대한 피드백 내용도 프롬프트에 함께 포함된다. 이렇게 구성된 풍부한 컨텍스트를 기반으로 프롬프트를 생성하여 LLM에 입력하면, Proworks 5 구조에 맞는 VO 기반 코드가 출력된다.



세 번째 모듈인 Self-Feedback은 변환된 코드에 대해 자동 피드백을 생성하고, 해당 피드백을 바탕으로 반복적으로 개선하는 역할을 수행한다. 이 모듈은 Shinn et al.의 Reflexion: Language Agents with Verbal Reinforcement Learning 기법을 변형하여 설계되었으며, LLM 기반의 평가 메커니즘을 통해 품질 진단 및 개선 방향을 제시한다. 입력으로는 Proworks 4 코드, 변환된 Proworks 5 코드, 그리고 명시적으로 정의된 Proworks 5의 디자인 패턴 규칙이 함께 사용된다. 피드백 LLM은 VO 패턴 준수 여부, 주석 보존, 동작 일관성, 예외 처리 등의 항목을 기준으로 코드의 품질을 평가하며, 자연어 형식으로 평가 결과를 출력한다. 만약 추가적인 개선이 필요하다는 판단이 내려지면, 해당 피드백은 LLM Code Conversion 모듈로 다시 전달되어 다음 반복에서 개선된 코드

생성을 유도하게 된다. 더 이상 개선점이 없다는 피드백이 출력될 경우, 최종 코드가 확정되며 변환 과정을 종료한다.

C. Inter-Module Communication Interface

본 프로젝트의 시스템은 세 가지 주요 모듈 간의 명확한 인터페이스를 바탕으로 구성되어 있으며, 각 모듈은 입력과 출력을 통해 상호작용한다.

먼저, Dependency Analysis Module은 시스템에서 가장 처음 실행되는 모듈로서, 입력으로는 Proworks 4의 레거시 코드를 받는다. 이 코드를 기반으로 함수 간 호출 관계와 Map 기반 데이터 참조를 분석한 후, 출력으로 Proworks 5의 구조에 맞는 스켈레톤 코드(converted schema)와 변환에 활용될 데이터 클래스(Value Object)를 생성한다.

이후 생성된 스켈레톤 코드와 데이터 클래스는 LLM Code Conversion Module의 입력으로 전달된다. 해당 모듈은 Proworks 4 레거시 코드, 변환된 스켈레톤 코드, 데이터 클래스 외에도 RAG(Retrieval-Augmented Generation) 방식을 사용하는 경우, Proworks 5의 API 문서로부터 벡터화된 정보를 담은 VectorStore를 추가 입력으로 받는다. 또한, Self-Feedback Module을 활용하는 경우에는 이전 단계에서 생성된 피드백 결과 역시 입력으로 함께 사용된다. 이 모듈의 출력은 최종적으로 변환된 Proworks 5의 Java 코드이다.

마지막으로, Self-Feedback Module은 LLM이 생성한 변환 코드의 품질을 점검하는 역할을 수행한다. 입력으로는 Proworks 4의 레거시 코드와 변환된 Proworks 5 코드, 그리고 Proworks 5에서 요구되는 디자인 패턴에 대한 명세를 함께 받는다. 이를 바탕으로 LLM이 작성한 코드에 대한 피드백을 출력하며, 해당 피드백은 다시 LLM Code Conversion Module의 입력으로 순환되어 반복적인 개선에 활용될 수 있다.

이처럼 각 모듈은 명확한 입력/출력 구조를 기반으로 구성되며, 데이터와 피드백의 흐름을 통해 점진적이고 안정적인 코드 전환이 가능하도록 설계되었다.

8. Solution

A. Implementation Details

1) Dependency Analysis Module

- ElementLevelDependencyAnalyzer: Java 소스 코드에서 “요소 단위(element-level)” 의존성을 분석하는 Python 클래스이다..
  - extract_element_level_dependencies : 입력된 Proworks 4 자바 코드에서 코드 요소를 추출하여 함수 간 호출 관계(Call Graph)를 추출하는 기능으로, 이후 의존성 분석 및 리팩토링 단위 결정의 기초 자료로 활용된다.
  - _extract_*_elements, _extract_*_dependency : 함수들은 Tree-sitter 기반 자바

코드 분석에서 구조적 요소(클래스, 메서드, 필드 등) 와 의존성 관계(메서드 호출, 객체 생성 등) 를 추출하는 역할을 한다.

- _analyze_node_dependencies : 각 요소의 AST를 순회하면서 Map.get("key")와 같은 field_access, method_invocation 노드를 분석한다..

- MapToVOConverter: Java 프로젝트 전체에서 Map 사용을 분석하고, 이를 기반으로 VO 클래스)를 생성한 뒤, 기존 코드를 VO 기반으로 자동 변환해주는 클래스이다.

  - _analyze_line : Map 메서드 호출에서 “key” 문자열을 추출한다..

  - _record_key : _analyze_line에서 추출된 문자열에서 사용 정보 (key, 사용 횟수, 파일 위치 등)를 축적한다.

  - _generate_vo_structure : 위에서 수집된 key들을 기반으로 VO(Value Object) 클래스의 스켈레톤(schema)을 생성한다. key의 타입 유추가 어려운 경우는 기본적으로 String으로 설정한다.

2) LLM Code Conversion Module

- convert : 위에서 생성된 VO 클래스 정보 및 Map 기반 레거시 구현을 기반으로, 지정된 LLM Agent를 통해 Map 기반 구현을 VO 기반으로 변환한다. 주어진 변환 규칙에 따라 함수 시그니처, 주석, 로직을 유지하며 Map 사용 부분만 VO로 치환한다.

3) Self-Refinement Module

- feedback : 변환된 VO 기반 코드에 대해 LLM 기반의 정적 피드백을 수행한다. 각 규칙에 대한 충족 여부를 확인하며, 규칙을 위반한 경우 간단한 설명을 첨부한다.
- conversion_feedback_loop : 변환과 피드백 과정을 반복적으로 수행하며, 일정 횟수 내에서 개선된 코드가 도출될 때까지 refinement 루프를 진행한다. 이를 통해 안정적인 코드 품질 확보가 가능하다.

4) RAG Module

- create_vectorstore : Proworks 5의 문서 자료들을 벡터로 임베딩하고, FAISS를 활용하여 검색 가능한 VectorStore를 생성한다.
- retrieve_context_for_conversion : 레거시 코드 또는 질의 내용을 기반으로 VectorStore에서 관련 문서를 검색한 후, 이를 바탕으로 LLM이 변환 맥락을 이해할 수 있도록 지원한다. RAG를 통해 보다 문맥에 맞는 변환 결과를 유도한다.

B. Implementation Issues

1) LLM의 일관되지 않은 응답 형식 문제

LLM을 통해 함수 단위로 코드를 변환하는 과정에서, 동일한 명령을 주더라도 응답 형식이 일관되지

않게 출력되는 문제가 있었다. 예를 들어 한 응답에서는 코드 전체가 ```java 태그로 감싸져 출력되었지만, 다른 응답에서는 태그 없이 일반 문자열로 출력되어 파싱 단계에서 실패하는 경우가 발생했다. 이는 LLM 응답이 확률적 샘플링 기반으로 생성되기 때문에 발생하는 구조적 한계이다. 이를 해결하기 위해 ```java 태그를 프롬프트의 마지막에 명시적으로 삽입하여, 코드 출력의 시작을 유도하는 방식으로 일관성을 확보하였다. 이를 통해 코드 블록 추출 실패율을 현저히 낮출 수 있었다. 추가로, 파싱 실패에 대비해 정규식 패턴을 보완하여 예외적인 응답 형식에도 일정 수준의 대응이 가능하도록 하였다.

2) 변환 규칙 미준수 사례

LLM에게 명시적으로 “기존 주석을 유지하고, 메서드 명/파라미터명을 변경하지 말 것” 등의 규칙을 전달했음에도 불구하고, 일부 응답에서는 주석을 누락하거나 함수 내부 로직을 변경하는 등 규칙을 위반하는 결과가 출력되었다. 이는 LLM이 긴 문맥이나 세부 규칙을 간과할 가능성이 있는 점에서 기인한 것으로 보인다. 이를 보완하기 위해 Feedback LLM에 해당 항목을 추가하여 활용하고, 위반 항목에 대해 Self-Refinement 과정을 반복 수행하도록 하였다. 추가로, LLM이 자동으로 코드의 indentation이나 spacing, 주석 정렬 등을 일괄적으로 정리해버리는 현상이 있었는데, 이는 의도하지 않은 불필요한 수정으로 간주되어야 한다. 변환 시스템은 반드시 변경이 필요한 부분에만 변화를 적용해야 하며, 이를 위해 코드 변환 전후의 주석과 공백 정보를 별도로 보존한 뒤, 변환이 완료된 후 이를 다시 복원하는 방식의 전처리 및 후처리 과정을 도입하여 문제를 해결하였다.

3) RAG 응답의 맥락 부적합 이슈

Proworks5 문서에서 관련 정보를 RAG 방식으로 가져오는 과정에서, 간혹 문맥상 맞지 않거나 핵심적인 규칙이 누락된 문서 내용을 기반으로 코드 변환이 이뤄지는 문제가 발생했다. 특히 VectorStore 구성 시 문서 단위(chunk)가 너무 작거나 분할 기준이 불균형한 경우, 의미 단위로 적절한 문장을 추출하지 못해 관련 없는 정보가 검색되는 경우가 있었다. 이에 따라 문서의 **chunk** 크기 및 분할 기준을 조정하고, 질의 문장에 클리닝을 적용하여 검색 정확도를 높이고 정답률을 개선할 수 있었다. 또한, **VectorStore** 검색 결과 중 유사도 점수가 **0.9** 미만인 문서는 **LLM** 프롬프트에 포함하지 않도록 필터링하여, 부정확한 맥락 주입을 방지하고 응답의 신뢰도를 향상시켰다.

4) 과도한 코드 생성 및 추론 지연 문제 개선

LLM 추론 과정에서 종종 개발자의 의도와 다른 코드가 생성되는 현상이 발생하였으며, 이는 단순한 정확도 문제뿐 아니라 전체 추론 시간의 지연으로도 이어졌다. 특히 LLM이 무의미한 코드를 과도하게 생성하는 사례가 있었고, 이는 길이 제한이 없는 상태에서 자주 나타나는 문제였다. 이를 해결하기 위해 기존 코드의 라인 수를 기준으로, 생성될 코드의 예상 라인 수를 실시간으로 모니터링하고 상한선을 설정하였다. 해당 방식은 추론 시간을 크게 단축시켰으며, 특히 극단적으로 오래 걸리던 일부 사례에 대해 효과적인 성능 최적화를 달성할 수 있었다.

9. Future Enhancements & Experimental Directions

A. Motivation and Expected Impact

변환 파이프라인이 정식으로 동작한 이후, 실제 코드베이스에 대한 변환 작업이 점차 누적됨에 따라 사람 검수를 통해 변환이 완료된 고품질 샘플 데이터(input-output pair)가 축적된다. 이 데이터가 단순한 결과물만이 아닌 해당 고품질 샘플들을 적극적으로 활용할 수 있는 가능성이 제기되었다.

기존 변환 과정은 입력 함수에 대한 분석 → 변환 규칙 적용 → LLM 기반 생성이라는 일괄 처리 방식으로 구성되어 있으며, 이때 LLM은 주어진 입력 함수만을 기반으로 추론을 수행한다. 그러나 동일한 유형의 코드나 유사한 컨텍스트를 가진 입력이 반복적으로 등장할 경우, 이미 처리된 기존 샘플을 참조하여 성능을 향상시킬 수 있는 여지가 존재한다. 즉, 사람이 직접 검수한 정답 페어는 단순한 로그나 결과물이 아니라, 후속 변환의 정확도를 높이는 학습 기반으로 작용할 수 있다.

이러한 맥락에서, 본 프로젝트는 완료된 고품질 변환 샘플을 단순히 저장하는 것을 넘어서, 이를 후속 변환 과정에서 활용 가능한 자산으로 전환하려는 시도를 시작하였다. 특히 다음과 같은 시나리오에서 그 활용 가능성이 주목되었다:

- 동일한 API를 사용하는 함수가 반복적으로 등장할 때, 이전 변환 결과를 참조함으로써 정확도를 보장할 수 있음
- 동일한 디자인 패턴 또는 구조를 따르는 함수에 대해, 반복된 재해석을 줄이고 일관된 결과를 보장할 수 있음
- 특정 함수명이나 도메인 전용 용어가 자주 등장할 경우, 이전 번역 결과의 어휘 선택을 유지할 수 있음

기대 효과로는 우선, 검수된 ground truth 데이터가 예시 또는 context로 활용됨으로써 LLM의 모호한 판단을 줄이고 보다 정밀한 변환 결과를 도출할 수 있다는 점이 있다. 또한, 이전에 성공적으로 처리된 변환 패턴을 유사한 입력에 적응적으로 적용함으로써 일관된 품질을 유지하고 추론 효율을 높일 수 있다. 이러한 접근은 단순한 변환 정확도 향상을 넘어서, 지속적으로 학습하고 적응하는 코드 변환 시스템으로의 진화를 가능하게 한다. 나아가 축적된 변환 데이터를 적절히 구조화하여 변환 흐름에 통합하면, 전체 시스템의 정밀도와 효율성을 높일 수 있으며, 실무 적용 시 반복성과 재사용성 기반의 비용 절감 효과도 기대할 수 있다. 결국, 변환 결과물의 누적은 유의미한 자산으로 기능하며, 이를 적극적으로 활용하는 것이 LLM 기반 시스템의 “지속 가능한 진화”를 실현하는 핵심 요소가 될 것이다.

B. Preliminary Trials and Approaches

1) In-Context Learning with Retrieval-Augmented Generation

코드 변환 과정에서 사람이 직접 검수를 한 고품질 input-output 변환 페어는, 이후 유사한 변환 작업에서 LLM의 성능을 개선할 수 있는 중요한 기반 자원으로 작용한다. 다만 이러한 사례를

무작위로 LLM 입력에 포함하는 방식은 비효율적이며, 오히려 모델의 주의 집중을 분산시켜 부정확한 결과를 유도할 수 있다. 이에 본 프로젝트는 ICL-RAG[3] 방식을 도입하여, 현재 입력과 유사한 사례만을 선택적으로 불러와 in-context 예시로 활용하는 구조를 설계하였다. RAG는 사전에 임베딩된 변환 샘플 중 가장 유사한 사례를 검색하여 제공함으로써, LLM이 전체 데이터를 탐색하지 않고도 필요한 맥락만을 바탕으로 추론할 수 있도록 돕는다.

RAG에서 임베딩의 주요 대상은 세 가지로 나뉜다. 첫째, 전 코드(Before)는 함수명, 구조, 파라미터, 주석 등 다양한 정적 요소를 포함하며 검색의 핵심 기준이 된다. 둘째, 후 코드(After)는 정답 예시로 제공되어 출력 형태나 스타일을 직관적으로 제시한다. 셋째, 주석이나 요약 정보는 필요한 경우에만 포함되며, 변환 조건이나 반환 규칙 등의 부가 정보를 명시하는 데 사용된다. 단, 이러한 요약은 본래의 문맥을 충분히 전달하지 못하거나 혼란을 유발할 수 있으므로, 적용 여부에 있어 신중한 판단이 요구된다.

추출 조건은 '양보다 질'의 원칙을 따른다. 많은 사례를 제공하기보다, 현재 입력과 구조적으로 근접한 한두 개의 사례만을 선택하는 것이 훨씬 효과적이었다. 특히 함수 유형, 사용 API, 반환 타입, 주석 구조 등 다양한 정적 속성을 종합적으로 고려하여 유사도를 판단하고, 이를 기준으로 사례를 필터링하였다. 결과적으로 이러한 방식은 변환 정확도와 일관성을 동시에 향상시키는 데 기여했으며, RAG 기반의 context 선택 전략이 성능 최적화의 핵심 요소로 작용함을 확인할 수 있었다.

2) Programmable converter

기존 LLM 기반 코드 변환 방식은 추론 결과가 "왜 맞았는지, 왜 틀렸는지"에 대한 명확한 해석이 불가능하다는 근본적인 한계를 가진다. 변환 성공 여부는 확인할 수 있지만, 그 원인을 규명하거나 이를 일반화된 규칙으로 환원하기는 어렵다. 이러한 한계를 극복하기 위해, 본 프로젝트에서는 사람이 검수한 고품질 input-output 페어를 기반으로 특정 패턴을 완벽히 처리할 수 있는 함수를 직접 생성하는 프로그래머블 변환기(Programmable Converter) 를 설계하였다. 이는 변환기의 자동화와 일반화를 동시에 추구하는 방식이다.

핵심 아이디어는, 검수 완료된 변환 샘플을 바탕으로 해당 규칙을 내포하는 변환 함수를 작성하고, 이를 통해 유사한 구조 또는 API 호출 패턴을 가진 입력에 대해 동일한 출력을 안정적으로 생성하는 것이다. 이 과정은 단발성 생성이 아니라 반복 사용 가능한 일반화된 함수 생성을 목표로 한다.

모든 샘플을 한 번에 학습시켜 단일 함수를 만드는 것은 비현실적이므로, 일정 수의 샘플(예: 10개)을 기준으로 나누어 최적화 과정을 반복 수행하였다. 특히 변환 실패 사례를 중심으로 샘플을 구성함으로써, 기존 함수의 한계를 빠르게 보완할 수 있도록 하였고, 매 사이클마다 함수가 전체 샘플을 정확히 처리하는지를 점검해 일반화 수준을 지속적으로 평가하였다. 또한, 단일 샘플에 과도하게 특화되는 오버피팅을 방지하기 위해 LLM 프롬프트 설계에도 신중을 가했다. 예를 들어,

[3] Liu, Xueguang, et al. "Lost in the Middle: How Language Models Use Long Contexts." *Advances in Neural Information Processing Systems* 36 (2023): 38453–38469.

특정 상수가 하드코딩되는 현상을 방지하기 위해 “정규표현식을 사용하라”, “입력 패턴을 일반화하라”, “조건 분기를 활용하라” 등의 명시적 지시를 포함시켰다. 이를 통해 보다 범용적인 변환 로직 생성을 유도하고 다양한 입력에 유연하게 대응할 수 있도록 하였다.

물론 새로운 변환 패턴이나 함수 일반화 실패로 인해 오답이 발생할 수 있으나, 이 또한 검수를 통해 새로운 샘플로 축적되고, 이후 이를 만족하는 새로운 함수가 추가됨으로써 전체 시스템의 변환 성능이 점진적으로 향상된다. 이는 결국 사람이 일일이 작성하던 규칙 기반 변환기를 LLM이 자동 생성하고 개선하는 구조로, 기존 룰베이스 방식의 한계를 크게 보완한다.

이러한 programmable converter는 다음과 같은 이점을 제공한다. 첫째, 변환 규칙 생성의 자동화로 인적 리소스를 절감할 수 있다. 둘째, 각 최적화 주기마다 새로운 패턴을 반영함으로써 반복 학습 없이도 빠르게 정확도를 높일 수 있다. 셋째, 고정된 룰이 아닌 샘플 기반 함수 생성을 통해 새로운 유형의 입력에도 적응할 수 있는 유연성을 확보한다.

이러한 접근은 최근 ARC-AGI 문제 해결을 위한 방식으로 주목받는 “Product of Experts” 접근과도 유사한 철학을 따른다. Franzen et al. (2024)[4]은 특정 문제에 특화된 서브 시스템을 구성하고, 이들 간의 역할 분담과 결합을 통해 전체 성능을 향상시키는 구조를 제안하였다. 본 프로젝트의 programmable converter 역시 각 변환 샘플을 하나의 전문가(expert)로 보고, 이들의 집합을 통해 시스템 전반의 변환 능력을 끌어올리는 구조라는 점에서 동일한 방향성을 공유한다.

10. Results

C. Experiments

본 프로젝트에서는 Proworks 4에서 Proworks 5로의 코드 변환을 자동화하기 위해 LLM 기반 변환 시스템을 설계하였다. 이 과정에서 주요 모듈인 Dependency Analysis (VO Generation),, Code Conversion, Self-Refinement, RAG Module의 성능을 검증하기 위한 다양한 실험을 수행하였다.

Evaluation dataset은 총 4가지 범주로 구성되어 있으며, 각각 **VO** 클래스 생성**, API Call** 변환, **getter/setter** 대체, **Map** 타입 자체의 **VO** 타입 변경, 엣지 케이스를 포함한다. 각 범주는 실제 현업 코드에서 자주 등장하는 변환 유형을 반영하도록 구성하였고,, 범주별로 10~100개의 예제를 수집/작성하였다. 특히, “엣지 케이스(hard cases)” 카테고리는 중첩된 Map 구조, 코드 내 주석, 비표준 포맷팅 등의 케이스를 포함시켜 모델의 견고성을 평가할 수 있도록 하였다. 정답은 수작업으로 리팩토링된 Java 코드와 그에 대응되는 유닛 테스트로 구성되며, 평가는 각 예제에 대한 테스트를 통과하는지를 기준으로 이진 정확도를 계산한다.

[4] Franzen, Daniel, Jan Disselhoff, and David Hartmann. “Product of Experts with LLMs: Boosting Performance on ARC Is a Matter of Perspective.” *arXiv preprint arXiv:2505.07859* (2025).

D. Results and Discussion

	VO Generation (Acc)
LLM (GPT-4.1)	0.5
LLM (devstral:24b)	0.0
Ours (Dependency analysis)	1.0

다음은 평가 결과이다.

먼저, **Dependency Analysis (VO Generation) Module**을 통해 함수 간 호출 관계 및 Map 기반 데이터 접근 패턴을 정확히 추출할 수 있었다. LLM으로 VO 클래스를 생성할 때의 생성된 필드의 정확도는 0.5에 불과하지만, 제시한 Dependency Analysis를 통해 VO를 생성하는 경우 모든 테스트 케이스에 대해 VO를 정확히 생성해낸 것을 볼 수 있다. 이로 인해 LLM이 VO(Value Object) 기반으로 변환해야 할 함수들을 선별적으로 학습할 수 있는 기반이 마련되었으며, 실제 변환 효율성을 크게 향상시킬 수 있다..

	Avg	API calls	VO get/set	Type Changes	Edge cases (Bad spacing, nested maps, comments etc)
Base	0.54	0.02	0.75	0.78	0.60
Base + RAG	0.91	0.98	0.94	0.97	0.75
Base + Reflexion	0.69	0.09	0.92	0.81	0.94
Base + RAG + Reflexion	**0.99**	**1.00**	**1.00**	**1.00**	**0.98**

*Reflexion: Our implementation of (Shinn et al., NIPS 2024).

다음으로, RAG 및 Reflection Module의 결과이다. Base 모델로는 Devstral:24b를 사용하였으며, RAG와 Reflexion 없이 프롬프팅만으로 변환을 시도한 파이프라인의 평균 성능이 0.54으로 도출되었다.

RAG을 추가하였을 때 모든 카테고리에 대한 성능이 향상된 것을 볼 수 있다. API Call에 대한 성능이 극적으로 (0.02->0.98) 향상된 것은 ProWorks 5 Documentation에 대한 지식을 취득했기 때문으로 해석할 수 있다. 또한 다른 카테고리에 대한 성능이 향상된 이유는 RAG에서 검색하는 Documentation이 다양한 케이스의 코드 변환 예시를 포함하기 때문으로 분석된다.

다만 경미한 타입 누락 혹은 VO 필드명 불일치가 여전히 발생하였는데, 이러한 한계를 보완하기 위해 **Self-Refinement Module**을 추가하여 실험하였다. 그 결과 초기 변환 코드의 평균 정확도는 0.54였으나 Self-Feedback 적용 이후 0.69까지 향상되었다.

더불어, RAG와 Self-Refinement Module을 함께 추가하였을 때 성능 향상이 가장 크게 이루어진

것을 확인할 수 있었다.

	Avg	API calls	VO get/set	Type Changes	Edge cases
Inswave-base (Rule-based conversion)	**0.79**	0.32	0.99	0.98	0.87

이는 인스웨이브측에서 제공한 rule-based converter의 정확도를 크게 개선하는 결과로 본 프로젝트에 목표에 부합하는 파이프라인이 도출되었다고 볼 수 있다.

Full pipeline (Base + RAG + Reflexion)	Avg Score
llama3.2	0.67
llama3.1:8b	0.94
phi-4	0.96
qwen2.5-coder:14b	0.96
qwen2.5-coder:7b	**0.98**
gemma3:12b	0.79
devstral:24b	**0.98**
mistral:7b	0.96
codellama:7b	0.72
dolphin3:8b	0.89

또한, 변환 모델과 변환 성능의 의존도를 파악하기 위해 다양한 모델로 평가를 진행하였다. 그 결과 **qwen2.5-coder:7b**와 **devstral:24b**의 성능이 가장 뛰어난 것을 확인할 수 있었다. 이 결과를 바탕으로 데모 및 최종 산출물 또한 devstral:24b를 이용하도록 결정하였다.

정리하자면, 본 프로젝트의 결과는 사전 분석 + LLM + 피드백 반복 + 문서 기반 context injection의 조합이 전통적인 코드 마이그레이션 작업에 있어 높은 효율성과 정확성을 달성할 수 있음을 시사한다. 추가적으로, Self-Feedback과 RAG 기술의 결합은 향후 자동화된 코드 변환 시스템의 정밀도를 높이는 데 핵심적 전략이 될 수 있음을 확인하였다.

11. Division & Assignment of Work

항목	담당자
변환 케이스 분석 및 방법 검토	박현, 배준익
문헌 조사	박현, 배준익
의존성 분석 모듈 구현	박현

RAG 구현	배준익
구현 모듈 테스팅	박현, 배준익
Reflexion 모듈 구현	배준익
IDE 플러그인 포팅 작업 (배포)	박현
추가적인 성능 개선 방법 구현	박현, 배준익

12. Conclusion

본 프로젝트를 통해 Proworks 4 기반의 레거시 자바 코드를 Proworks 5 기반의 VO(Value Object) 방식으로 자동 변환하는 LLM 기반 시스템을 구축할 수 있었다. 우선 Dependency Analysis Module, Code Conversion Module, Self-Refinement Module, RAG Module로 구성된 아키텍처는 코드 분석부터 변환, 피드백 기반의 품질 향상, 외부 문서 참조를 통한 정확도 향상까지 유기적으로 연결되도록 설계되었다.

실험을 통해 Dependency Analysis (VO Generation),, Code Conversion, Self-Refinement, RAG Module의 성능 향상을 검증하였으며, 특히 rule-based 베이스라인에 비해 20%p 성능 향상을 관찰할 수 있었다. 이는 본 변환 시스템은 실제 자바 코드베이스에서 높은 수준의 정확도와 일관된 스타일을 유지하며 동작하는 것을 의미하며, 추후 다양한 레거시 마이그레이션 사례에 범용적으로 활용될 수 있는 가능성을 확인할 수 있었다.

◆ [Appendix] User Manual

프로젝트의 대표 사이트는 다음과 같다..

https://nemodleo.github.io/llm-code-converter

1. 개요

LLM 기반 코드 변환 VS Code Extension을 통해 코드 변환 개발자들이 실제 Proworks 4 프로젝트를 Proworks 5로 마이그레이션하는 과정을 자동화하는 데 이용할 수 있다. 해당 익스텐션은 복잡한 Map 기반 로직을 Value Object (VO) 기반의 정적 구조로 변환하고 변환 정확도를 높이는 로직이 다수 포함되어 있다.

2. 시스템 요구 사항

운영체제: Windows, macOS, Linux

VS Code: 최신 버전 설치

Python 환경: Python 3.8 이상 (백엔드 서버 구동용)

GPU: 40GB 이하 VRAM을 가진 GPU (LLM 로컬 실행을 위함. Ollama 등 사용 시 권장)

네트워크: 고객사 보안 정책에 따라 폐쇄망 환경에서도 작동 가능 (외부 LLM API 사용 제한)

3. 설치 및 설정

다음을 통해 프로젝트를 다운로드받을 수 있다.

git clone https://github.com/nemodleo/llm-code-converter.git

현재 이 Extension은 공개 마켓플레이스에 등록되어 있지는 않으므로, .vsix 파일을 통해 수동으로 설치해야 한다.

Extension 설치:

VS Code 왼쪽 사이드바에서 Extensions 뷰 (Ctrl+Shift+X 또는 Cmd+Shift+X)를 연다.

Extensions 뷰 상단의 ... (점 3개) 메뉴를 클릭한 후, Install from VSIX...를 선택한다.

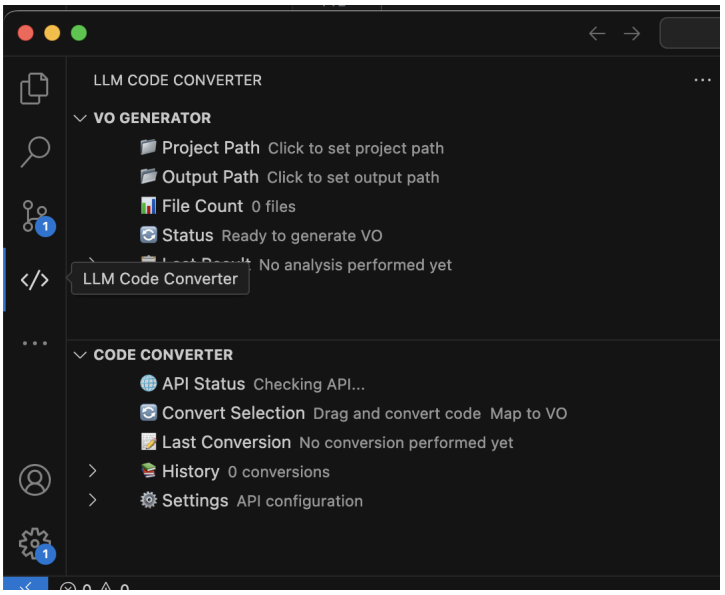
다운로드받은 llm-code-converter-extension.vsix 파일을 선택하고 Install을 클릭한다.

설치가 완료되면 VS Code를 다시 시작하여 Extension을 활성화한다.

extension side panel에 LLM Code Converter가 존재하면 설치가 완료된다.

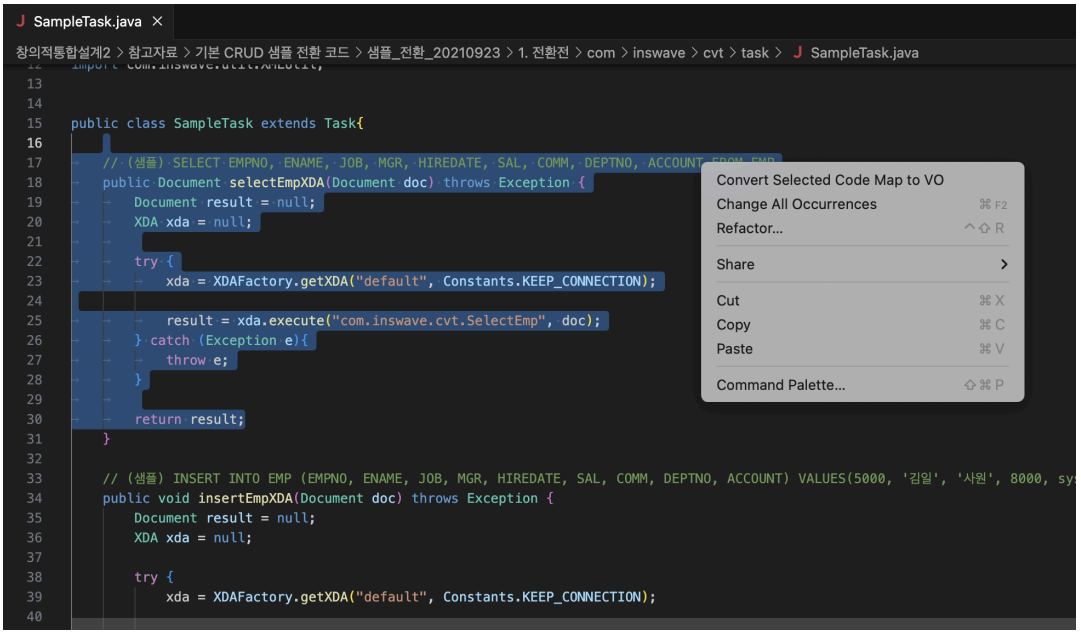
Extension 사용시 직전에 FastAPI 서버 구동을 python server.py을 통해 진행할 수 있다.

4. **Extension** 사용 방법

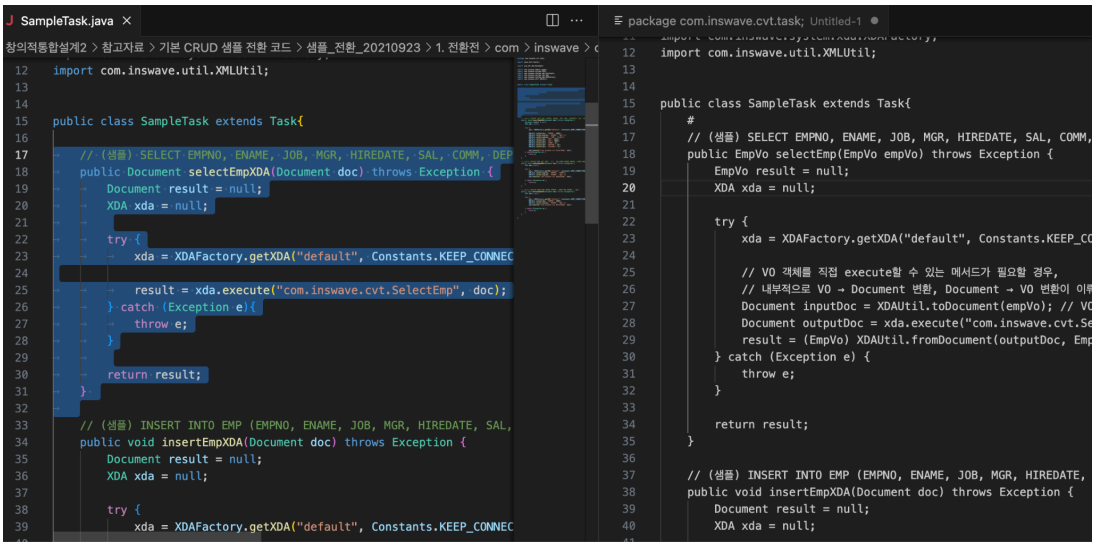


LLM 기반 코드 변환 VS Code Extension은 사이드 패널을 통해 직관적인 코드 변환 기능을 제공한다. VS Code 좌측 사이드바에 추가된 LLM Code Converter 아이콘을 클릭하면, "VO GENERATOR"와 "CODE CONVERTER" 두 가지 세션이 나타난다.

Extension은 두 가지 주요 코드 변환 작업을 지원한다. 첫째, **VO** 생성 기능은 Proworks 4 프로젝트의 Map 기반 데이터 구조를 Proworks 5의 VO 기반 구조로 자동 변환하기 위한 첫 단계이다. 이 기능을 사용하려면 Extension 사이드 패널에서 **"VO GENERATOR"** 세션을 선택한 후, 변환 대상 Java 프로젝트의 루트 경로를 **Project Path**에, 생성된 VO 클래스 파일이 저장될 경로를 **Output Path**에 입력한다. 경로 입력 후 해당 세션 내의 "Generate VO" 버튼을 클릭하면, 프로젝트 내의 Map 사용 패턴을 분석하여 Proworks 5 형식의 VO 클래스들이 지정된 Output Path에 생성된다.



둘째, Map to VO 코드 변환 기능은 기존 Java 코드 내의 Map 기반 로직을 생성된 VO 클래스를 활용하여 변환한다. 이 기능을 사용하려면 Extension 사이드 패널에서 **"CODE CONVERTER"** 세션을 선택한 후, 변환하고자 하는 Java 파일에서 코드의 일부 또는 전체를 마우스로 드래그하여 선택한다. 선택된 코드가 없으면 현재 파일 전체가 변환 대상으로 간주될 수 있다. 선택된 코드 위에서 마우스 우클릭한 후, 컨텍스트 메뉴에서 **"Convert Selected Code Map to VO"**를 선택한다. 변환 실행 시 팝업되는 입력창에 "변수명은 camelCase로"와 같이 추가적인 변환 지시(사용자 프롬프트)를 입력할 수 있다. 이는 LLM이 변환 결과를 사용자의 특정 요구사항에 맞춰 최적화하도록 돕는다. 변환된 결과 코드는 보통 새로운 파일로 생성되거나, 현재 파일의 적절한 위치에 삽입된다.



변환이 완료되면, Extension은 원본 코드와 변환된 코드를 나란히 비교할 수 있는 변환 결과 비교(Before vs After) 기능을 제공하여 변경 사항을 직관적으로 확인하고 검수할 수 있다. 또한, Extension 사이드 패널의 하단이나 VS Code의 출력(Output) 패널을 통해 변환 과정의 로그와 현재 상태를 확인할 수 있다. 만약 코드 변환 중 오류가 발생하면, 해당 출력 패널에 상세한 로그와 함께 예외 메시지가 표시되어 문제 해결에 도움을 준다.