The Transactions of the Korea Information Processing Society
TKIPS Vol.15, No.3, pp.230~237
eISSN: 3022-7011
https://doi.org/10.3745/TKIPS.2026.15.3.230

A Hybrid Prompting-based Code Generation on Software Development Life Cycle

Yejin Jin† · Janghwan Kim†† · Kidu Kim††† · R. Young Chul Kim††††

ABSTRACT

The advancement of large language models (LLMs) is transforming software development. However, LLMs struggle to accurately interpret user intent when insufficient context and requirements are provided. As a result, developers must invest significant effort in advanced prompt engineering to achieve the desired outcomes. To solve this, we propose a hybrid code generation mechanism that takes natural language requirements, design information, and template code as input. By providing a predefined template that represents the customer's intent and the system's structure, the method enables LLMs to understand specific requirements more accurately. This approach is expected to improve the accuracy and reliability of code generation compared to natural language-only methods. This study objectively evaluates its performance in comparison to existing LLM approaches through a case study. By presenting a new paradigm for LLM-based software development, the proposed method is expected to offer both academic and practical contributions to the field.

Keywords : Generative AI, Software Life Cycle, Code Generation, Natural Language Processing

소프트웨어 개발 생명주기 상에서 하이브리드 프롬프팅 기반 코드 생성 메커니즘

진 예 진† · 김 장 환†† · 김 기 두††† · 김 영 철††††

요 약

거대 언어 모델의 발전은 소프트웨어 개발 방식을 변화시키고 있다. 그러나 LLM은 충분한 맥락과 요구사항을 제공받지 못할 때, 사용자의 의도를 정확히 파악하기 어렵다. 이러한 한계는 개발자가 원하는 결과를 얻기 위해 정교한 프롬프트 엔지니어링에 많은 노력이 필요하다. 이를 해결하기 위해, 새로운 접근 방식인 자연어 요구사항, 설계 및 템플릿 코드를 입력하는 복합적인 프롬프팅 기반 코드 생성 메커니즘을 제안한다. 사용자의 의도와 시스템의 구조를 미리 정의된 템플릿 형태로 제공함으로써, LLM이 구체적인 요구사항을 더 정확하게 이해하도록 지원한다. 이러한 방식이 단순한 자연어 기반 방식보다 코드 생성의 정확성과 신뢰성을 향상시킬 것으로 기대한다. 본 연구는 사례 연구를 통해 기존 LLM과의 성능을 객관적으로 평가한다. LLM을 활용한 소프트웨어 개발 패러다임을 제시함으로써, LLM 기반 소프트웨어 개발 분야에 학문적 및 실용적으로 기여할 것으로 기대된다.

키워드 : 생성형 AI, 소프트웨어 개발 생명주기, 코드 생성, 자연어 처리

1. 서 론

거대 언어 모델(Large Language Models, LLMs)의 발전은 소프트웨어 개발 분야에 큰 영향을 미치고 있다. 특히 자연어 기반 코드 생성(Natural Language to Code Generation) 기술은 개발자의 생산성을 향상시키는 접근 방식으로 사용되고 있다[1]. 현재 LLM은 단순한 명령어 해석을 넘어 복잡한 로직을 처리하고 코드를 생성하는 능력을 가진다. 그러나 현재의 LLM 기반 코드 생성 방식은 단순 자연어 질의에 의존한다. 이는 사용자의 복잡한 요구사항이나 구체적인 설계 의도를 정확하게 전달하는 데 한계를 갖는다. 사용자의 모호하거나 불완

※ 이 논문은 2025년 KCSE 2025의 우수논문으로 “Software Process 내에서 AI 지향 요구 공학 적용한 사례”의 제목으로 발표된 논문을 확장한 것임.
※ 이 논문은 한국연구재단의 4단계 두뇌한국21사업(과제명: 초분산 자율 컴퓨팅 서비스 기술 연구팀, 과제번호: 202003520005)의 지원을 받아 수행된 연구임.
† 준 회 원 : 홍익대학교 소프트웨어융합학과 박사과정
†† 준 회 원 : 홍익대학교 소프트웨어융합학과 박사수료
††† 준 회 원 : 한국정보통신기술협회 AI디지털융합단 팀장
†††† 정 회 원 : 홍익대학교 소프트웨어융합학과 교수
Manuscript Received : October 1, 2025
First Revision : November 18, 2025
Second Revision : January 6, 2026
Accepted : January 29, 2026
* Corresponding Author : R. Young Chul Kim(bob@hongik.ac.kr)

※ This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

전한 자연어 입력은 LLM이 충분한 구조적 맥락이나 상세 요구사항을 이해하지 못하게 한다[2]. 이는 기능적으로 부정확하거나 비효율적인 코드를 생성하는 주요 원인이며 신뢰성 및 효율성 저하로 직결된다. 결과적으로 개발자는 원하는 코드를 얻기 위해 과도하고 반복적인 프롬프트 엔지니어링에 노력을 투입해야 하는 비효율성을 초래한다.

본 연구는 언어학적 분석을 기반으로 요구사항을 체계적으로 추출하고, UML 설계로 구조화하며, 모델 기반 개발(MDD, Model-Driven Development)을 통해 스켈레톤 코드를 생성하는 절차적 소프트웨어 개발 사이클을 제시한다. 생성된 산출물을 LLM에 입력하는 하이브리드 프롬프팅 전략을 활용함으로써, 모호한 자연어 요구사항이 주어지더라도 LLM이 시스템의 구조적 맥락과 구체적 구현 제약을 정확히 이해하도록 한다. 또한, 기능적 정확성과 신뢰성이 높은 코드를 생성할 수 있도록 한다. 본 연구는 LLM을 활용한 고품질 소프트웨어 개발을 위한 실용적 프롬프팅 전략과 새로운 개발 패러다임을 제시하며, LLM 기반 소프트웨어 공학 분야에 학문적, 실용적 기여를 기대한다. 2장에서는 소프트웨어의 자동 코드 생성에 대한 관련 연구를 언급한다. 3장에서는 제안하는 메커니즘을 제시하고, 4장에서는 사례 연구를 통해 정량적 지표를 통해 성능 평가를 수행한다. 마지막으로 결론을 언급한다.

2. 관련 연구

LLM을 활용한 코드 생성 연구에서는 자연어 요구사항으로부터 개발 코드를 자동화하는 시도가 활발히 이루어지고 있다. 대부분의 연구에서는 LLM을 프롬프트 기반으로 사용하며, 요구사항 입력으로부터 코드를 생성할 수 있도록 한다. Ye의 연구에서는 자동 프롬프트 정제 기법 (Prochemy)을 통해 LLM의 코드 생성 성능을 기준으로 프롬프트를 최적화하는 접근을 제안했다[3]. 이를 통해 평균 4~5%의 코드 생성 정확도를 향상했으며 LLM 기반 코드 생성의 신뢰성과 효율성을 높였다. Li의 연구에서는 기존 CoT(Chain-of-Thought) 방식의 한계를 극복하기 위해 SCoT(Structured Chain-of-Thought)를 도입했다[4]. 순차, 분기, 반복 구조를 중간 추론 단계에 포함해 LLM의 프로그래밍 사고를 활성화한다. 이를 통해 최대 13.79% 성능 향상과 코드 품질 및 가독성 개선 효과를 보여준다. Morales의 연구에서는 Impromptu라는 모델 주도 프롬프트 엔지니어링 프레임워크를 제시하였다[5]. 플랫폼 독립적인 도메인 특화 언어(DSL)를 활용하여 프롬프트 정의, 모듈화, 체이닝, 버전 관리를 할 수 있다. 요구사항을 만족하는지 자동 검증이 가능하도록 설계되어 LLM 기반 코드 생성 과정에서 프롬프트 관리의 효율성 및 신뢰성을 동시에 확보하였다. Su의 연구는 LLM의 코드 생성 능력에 대해 정확성, 견고성, 효율성, 유지보수성을 포함한 평가 방법론을 제시하였다[6]. GPT-3, GPT-4, Llama 2 등을 대상으로 한 실험에서 고성능 모델일수록 코드의 정확도는 높지만, 구조적 복잡도 또한 증가하는 경향을 보였다. 이는 실제 소프트웨어 현장에 적용 가능한 수준의 고품질 코드를 보장하기 어려움을 보여준다.

제시된 연구들은 LLM을 활용한 코드 생성의 효율성과 정확도를 높이는 데 기여한다. 그러나, 소프트웨어 개발의 생애주기 관점에서 한계를 보인다. 해당 연구들은 생성된 코드를 실행하거나 검증하는 단계에 의존한다. 이는 LLM의 오류가 요구사항에 대한 이해 문제보다는 코드 자체의 문제일 가능성이 높다고 해석된다. 또한, 연구의 초점이 요구사항에서 함수 수준의 실행 가능한 코드를 생성하는 데 맞추어져 있다. Su의 연구 결과에 따르면, 작동하는 코드가 좋은 코드를 의미하지 않는다. LLM을 활용한 코드 생성은 모듈화, 시스템 아키텍처와 같은 고수준 설계 문서 또는 프로세스가 명시적으로 포함되지 않는다. 이로 인해 대규모 시스템 개발에 필수적인 해석용이성, 재사용성, 확장성이 저해될 수 있다.

본 연구는 소프트웨어 설계 단계의 부재를 해결하고자 한다. 기존 연구들은 요구사항으로부터 코드를 생성하는 단일 단계로 구성된다. 본 연구는 요구사항 해석과 코드 생성 사이에 명시적인 설계 단계를 추가한다. 구체적으로, LLM 기반의 프롬프트 엔지니어링을 활용하되, 이를 두 번의 단계에 나눠 적용한다. 첫 번째 단계에서는 프롬프트를 사용하여 요구사항을 해석한다. 요구사항 정보를 바탕으로 시스템 구조 및 모듈화 정보를 담고 있는 UML 설계를 생성한다. 두 번째 단계에서는 생성된 UML 설계와 원래의 요구사항을 결합한 새로운 프롬프트를 구성하여, 이를 기반으로 최종 개발 코드를 생성한다. 이러한 단계적 구조를 통해, 코드 생성 이전에 시스템 설계의 완전성 및 일관성을 확보한다. 또한, LLM이 기능 중심의 코드를 구현하는 것 뿐 만 아니라 시스템 아키텍처를 반영한 코드를 생성하도록 유도한다.

3. 소프트웨어 개발 생명주기에서의 코드 생성

본 연구는 자연어 요구사항으로부터 실행이 가능한 스켈레톤 코드를 발생시킨다. 소프트웨어 생명 주기에 따라 요구사항 분석, 설계, 구현의 단계를 수행한다. 자연어 요구사항에 대해 자연어 전처리와 촘스키의 구문 구조 분석 이론[7], 필모어의 시맨틱 롤 이론[8] 기반의 분석을 수행한다. 자연어 전처리와 필모어의 시맨틱 롤 이론 기반 분석을 자동화하는 시도로 LLM을 사용한다. 분석된 요구사항 결과를 바탕으로 UML 다이어그램을 생성한다. 스크립트를 통해 UML 이미지를 보여주기 위해 PlantUML을 사용한다. UML 설계는 코드 생성의 기반이 된다. 모델 기반의 코드 생성을 위해 메타모델링을 사용하여 코드의 메타모델, UML 다이어그램의 메타모델을 생성한다. 메타모델 변환 엔진을 통해 데이터 변환을 수행하여 스켈레톤 코드를 생성한다. 스켈레톤 코드를 완전한 코드로 생성하기 위해 LLM을 사용한다. 요구사항과 분석을 통해 생

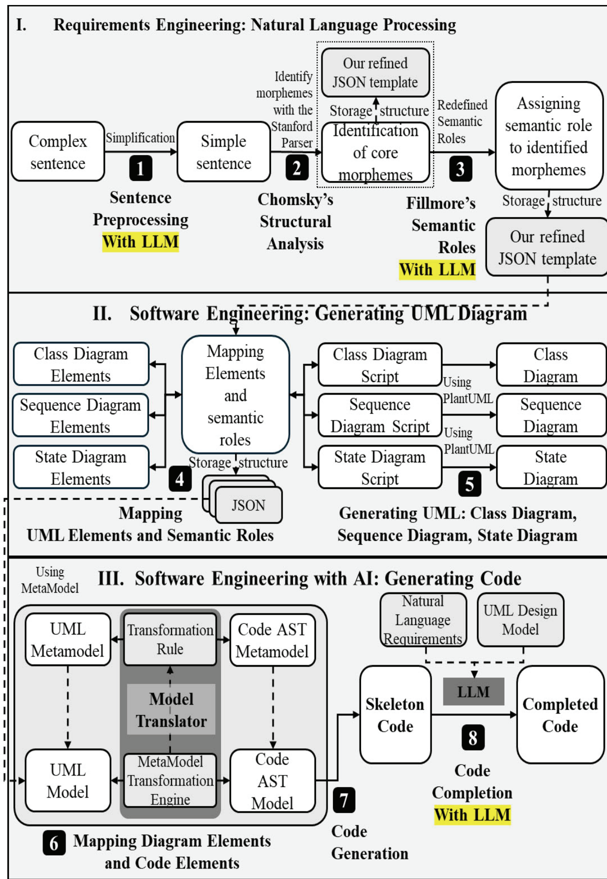

Fig. 1. Code Generation Process on the Software Development Lifecycle [9]

성된 설계, 그리고 스켈레톤 코드를 LLM의 입력으로 사용한다. Fig. 1은 본 연구의 메커니즘을 보여준다.

기존 LLM은 주로 자연어 요구사항만으로 코드를 생성한다. 자연어 요구사항은 모호성과 복잡성을 지닌다. 따라서 정확한 결과를 얻기 위해서는 구조화된 고급 프롬프트가 필요하다. 본 연구에서는 요구사항으로부터 일관성 있고 체계적인 분석을 통해 도출된 설계 정보, 스켈레톤 코드를 일관된 형태로 제공한다. LLM이 다양한 프로그래밍 언어로 완전한 코드를 생성하도록 한다. 본 메커니즘은 자연어 분석과 코드 생성에 AI를 부분적으로 사용한다.

3.1 생성형 AI를 적용한 요구공학

자연어 요구사항을 체계적으로 분석하기 위해 일관된 문장 형태로 전처리한다. 영어 문장으로 작성된 요구사항을 대상으로 연구를 수행한다. 영어 문장은 크게 세 가지의 문장 형태로 나뉜다. 주어와 동사가 하나씩 존재하는 문장을 단문, 여러 개의 동사나 주어로 이루어진 문장은 복문 혹은 중문이라고 한다. 중문은 등위 접속사로 두 개의 절을 연결한 문장이다. 복문은 종속 접속사로 종속절과 주절로 연결된 문장이다. 중문과 복문을 단문의 형태로 변형하기 위해 접속사를 기준으로 절을 분리한다. 분리된 절을 새로운 문장으로 처리한다. 문장 내에 대명사가 있는 경우에는 분리되기 이전의 문장 요소로 대체한다. 또한 누락된 문장의 요소는 기존의 문장 요소로 사용한다. 전처리된 문장은 촘스키의 구문 구조 분석 이론을 바탕으로 구조 분석을 한다 [7]. 구문 구조 분석 이론은 문장의 형태소를 구분하고, 각 형태소에 맞는 품사를 부여한다. 문장은 문장, 구, 절, 단어, 형태소로 나뉜다. 이를 통해 문장 내의 구성 요소를 트리 형태로 시각화할 수 있다. 대표적으로 Stanford Parser, Berkeley Parser가 촘스키의 이론에 따라 문장을 분석할 수 있다. 본 연구에서는 형태소와 그에 따른 품사의 쌍이 중요하다. 따라서, 자연어 처리에서 많이 사용되는 NLTK 라이브러리를 사용하여 문장을 분석한다[10]. 그리고, 기존 문장과 형태소, 품사를 구조화하여 저장할 수 있도록 JSON 구조를 정의한다. Fig. 2는 동일한 문장에 대한 분석 결과를 보여준다.

문장의 의미를 파악하기 위해 필모어의 시맨틱 롤 이론을 사용한다. 필모어의 이론은 동사를 기준으로 각 명사의 관계를 분석한다[8]. 필모어의 시맨틱 롤 이론은 다양한 도메인에서 의미 축소 및 확장되었다. 따라서 UML 생성을 위한 역할들로 기존 역할을 재정의한다. Table 1은 재정의된 시맨틱 롤을 보여준다.

필모어의 시맨틱 롤 이론은 명사와 동사의 관계를 파악하여 명사에 의미를 부여하는 이론이다. 이는 형태소의 품사가 기준이 된다. 따라서, 구문 구조 분석 이론으로 식별된 형태소를 바탕으로 의미 분석을 수행한다. 문장 내의 동사와 명사 간의 역할을 식별해야 하므로 동사가 행위 동사인지, 상태 동사인지 구분해야 한다. 행위 동사는 행위, 행동, 일시적으로 변

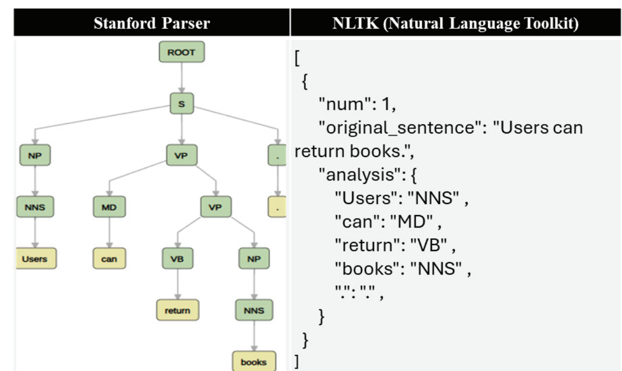

Fig. 2. Syntactic analysis using the Stanford Parser and NLTK libraries

Table 1. Refined Fillmore's Semantic Roles

Roles	Definition
Actor	The entity that is the main subject of the event.
Object	The entity affected by the event.
Instrument	The tool used to perform the action.
Source	The entity performing the action.
Target	The entity receiving the action.

화하는 상태를 나타내는 동사이다. 상태 동사는 감각이나 관계, 생각 등을 나타내는 동사이다. 따라서 행위 동사의 주체는 Actor나 Source가 될 수 있고, 그 행위 동사를 받는 객체는 Object나 Target으로 분류될 수 있다. 그리고 행위 동사에 사용되는 도구는 Instrument로 분류된다. 상태 동사는 어떠한 감정을 받는 객체로서 Object가 될 수 있다. 이를 자동화하기 위해 LLM의 입력에 시맨틱 롤 이론을 제공하고 퓨샷 프롬프팅 기법을 통해 결과를 얻는다. LLM의 결과는 문장 번호, 원본 문장, 동사와 동사 종류 쌍, 명사와 시맨틱 롤 쌍으로 저장된다. 특히 동사의 경우, 표제어로 변형한다. 표제어란 여러 변형된 단어들을 대표하는 기본형 단어를 의미한다. 예를 들어 'went'라는 동사는 'go'가 표제어이다. 이를 통해 동사의 의미를 더욱 명확히 표현한다.

3.2 요구사항 데이터 기반 UML 설계 생성

분석된 요구사항 정보로 UML 다이어그램을 생성한다. 클래스, 시퀀스, 상태 다이어그램을 생성하며 각 다이어그램의 필수 요소와 필모어의 시맨틱 롤 이론, 동사, 형용사를 대응시킨다. Actor, Object는 클래스 다이어그램의 클래스로 대응시킨다. 시퀀스 다이어그램은 Actor 또는 Source가 메시지를 보내는 객체, Object 또는 Target이 메시지를 받는 객체로 대응시킨다. 상태 다이어그램은 시스템의 상태를 나타내므로, Object가 시스템 객체인 경우에만 상태 다이어그램과 대응하여 사용한다. 클래스 다이어그램의 함수, 시퀀스 다이어그램의 메시지, 상태 다이어그램의 이벤트는 행위 동사와 대응되고, 그 동사가 수행될 때 같이 사용되는 도구로서 의미를 지니는 Instrument는 매개변수가 된다. Fig. 3은 UML 중 Class Diagram의 대응 관계를 예시로 보여준다.

대응된 데이터는 하나의 JSON 형식을 가진다. 저장된 데이터를 바탕으로 PlantUML의 문법을 사용하여 UML 생성 스크립트를 생성한다.

3.3 메타모델링 기법을 통한 스켈레톤 코드 생성

생성된 3가지의 UML을 바탕으로 모델 기반 스켈레톤 코드를 생성한다. 모델 변환을 위해 메타모델링 기법을 수행한다.

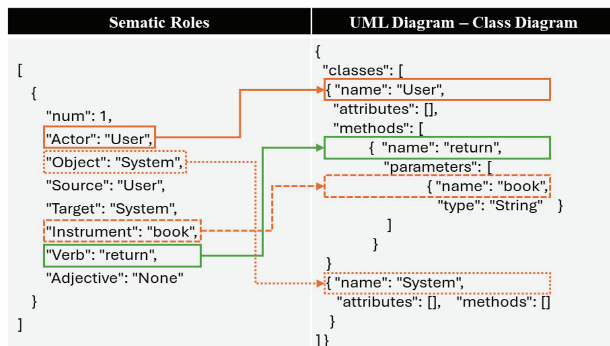

Fig. 3. Example of Mapping Class Diagram and Semantic Roles Elements

Eclipse Modeling Framework(EMF)를 사용하여 메타모델을 설계한다. EMF는 표준화된 방식으로 메타모델을 정의하고, 모델 요소 간의 관계를 명확히 표현할 수 있는 도구이다[11]. 이를 활용하여 UML 메타모델과 코드 메타모델을 정의하고, XMI(XML Metadata Interchange) 형식으로 데이터를 저장한다. 이때 저장된 데이터는 메타모델의 구조를 따르며, UML 메타모델을 코드 메타모델로 변환하기 위해 별도의 모델 변환 규칙을 설계한다. 이러한 과정을 통해 UML 기반 설계 정보를 코드 생성에 직접 활용할 수 있도록 한다.

UML 설계를 기반으로 생성된 스켈레톤 코드는 기본적인 클래스, 메서드 등을 포함하고, 실행할 수 있다. 그러나 이는 단순히 코드의 구조적 틀을 제공할 뿐, 실제 세부 로직 구현은 개발자가 직접 작성해야 한다. 따라서 완전히 자동화된 구현에는 한계가 있다. 이를 보완하기 위해 LLM을 활용하여 스켈레톤 코드 내부의 구체적 로직을 생성함으로써 완전한 코드를 도출한다. 최근 개발자들은 이와 유사하게 생성형 AI를 바이브 코딩(Vibe coding) 방식으로 활용하여 생산성을 높이고 있다[12]. 단순한 요구사항 문장만을 입력으로 제공하지 않고, 제안한 메커니즘의 산출물인 설계 정보와 스켈레톤 코드를 함께 사용한다.

4. 사례 연구

사례 연구를 통해 제안하는 메커니즘과 기존의 자연어 요구사항만을 입력으로 활용한 코드 생성 방식을 비교한다. 사례 연구 대상으로는 도서관 관리 시스템을 선정하였으며, 자연어 요구사항은 다음과 같다.

[1] Users should be able to register new books in the system.
[2] Users should be able to search for and borrow books.
[3] Users should be able to return books.
[4] Administrators should be able to edit or delete book information.
[5] The system should manage the status of books currently on loan.
[6] Users should be able to check their loan history.
[7] The system should properly handle books that have been borrowed for a long time.

도서관 관리 시스템은 책 등록, 대출, 반납 등의 기능이 필요하다. 본 연구에서는 후카츠 프롬프트 기법을 활용하여 다음과 같이 LLM에 입력을 제공한다. 후카츠 프롬프팅은 단일 입력만으로 모델이 정확하고 구조화된 출력을 생성하도록 지원하는 방법이다[13]. 입력에 대한 설명 및 참조 자료를 추가하여 모델의 이해도를 높이는 전략을 포함한다. 비교 평가를 위해 GPT 5.1과 Gemini 3을 활용한다. LLM은 주어진 초기 요구사항을 넘어 누락된 기능 및 제약 조건을 스스로 보완한다. 이로써 코드의 완성도 및 실용성을 높인다. 그러나 모델이 추가하는 기능과 코드의 복잡성을 개발자가 완전히 제어하기

어렵다. 또한, 불필요한 구조로 인한 복잡성 문제를 야기하고, 코드의 유지보수를 어렵게 만든다. 초기 개발 속도를 높이는 데에는 효과적이지만, 생성된 코드에 대한 검토 및 재구조화 작업이 필요하다.

> **#지침:** 당신은 최고의 파이썬 개발자입니다. 사용자의 요구사항을 정확히 이해하고, 최적화되고 효율적인 코드를 작성하는 데 탁월한 능력을 가졌습니다.
> **#목표:** 요구사항과 제약조건을 기반으로, 실행 가능하고, 명확하며, 주석이 잘 정리된 파이썬 코드를 작성합니다.
> **#제약조건:**
> - Python 3.13.X를 사용하세요.
> - 불필요한 라이브러리 사용을 지양하고, 표준 라이브러리를 우선적으로 사용하세요.
> - 코드의 가독성을 최우선으로 고려하세요.
>
> **#요구사항:** 1)~7)
> **#출력 형태:** 최종 파이썬 코드 블록을 제공하세요. 추가적인 설명이나 대화는 생략합니다.

본 연구의 메커니즘은 자연어 요구사항뿐만 아니라 UML 설계 정보와 스켈레톤 코드를 함께 입력으로 제공한다. 이를 통해 LLM은 구조적으로 일관된 맥락을 바탕으로 코드 생성을 수행할 수 있으며, 결과적으로 요구사항을 충실히 반영한 완전한 코드를 도출한다. LLM에 입력하는 방식은 후카츠 프롬프팅 기법으로 동일하지만, 목표에 설계와 스켈레톤 코드를 함께 고려하도록 수정하였다.

> **#목표:** 요구사항, 제약조건, 설계, 그리고 스켈레톤 코드를 바탕으로 실행 가능하고, 명확하며, 주석이 잘 정리된 파이썬 코드를 작성합니다.

본 연구를 통해 생성된 코드는 Book, User, Administrator, System, 네 가지 클래스로 설계된다. 시스템 관리 클래스인 System이 모든 핵심 로직을 중앙에서 처리한다. System은 도서를 등록 및 검색하고, 대출 시, 도서의 상태를 OnLoan으로 변경하며 사용자에게 반납 예정일을 안내하고, 반납 시 상태를 Available로 되돌린다. Administrator 클래스는 System을 통해 도서 정보를 수정하거나 삭제하고, 사용자는 자신의 대출 기록을 언제든 확인할 수 있도록 한다. 추가 기능으로 사용자에게 ID가 부여되었다. 대출 시, 상세한 대출/반납 기록을 관리하는 기능이 추가되었다. Fig. 4는 자연어 입력만을 받은 LLM(ChatGPT, Gemini)와 본 연구의 방법을 통해 발생한 설계와 코드의 일부를 보여준다. 본 연구에서는 절차적 분석을 통해 요구사항 중심의 간결하고 핵심적인 설계를 도출한다. 이를 바탕으로 간결하고 명확하게 코드를 생성한다.

LLM은 UML 설계 이미지를 생성할 수 없기 때문에, 스크립트 형태로 출력한다. 자연어 입력으로부터 UML 설계를 생성할 수 있으나, 그 추상화 수준이 지나치게 낮아 결과적으로 불필요하게 복잡한 UML을 생성하는 한계가 있다. Fig. 4의 코드는 '관리자가 도서 정보를 수정할 수 있다.'라는 요구사항 4번을 구현한 코드이다. LLM은 도서 정보를 수정할 때 도서 ID를 기반으

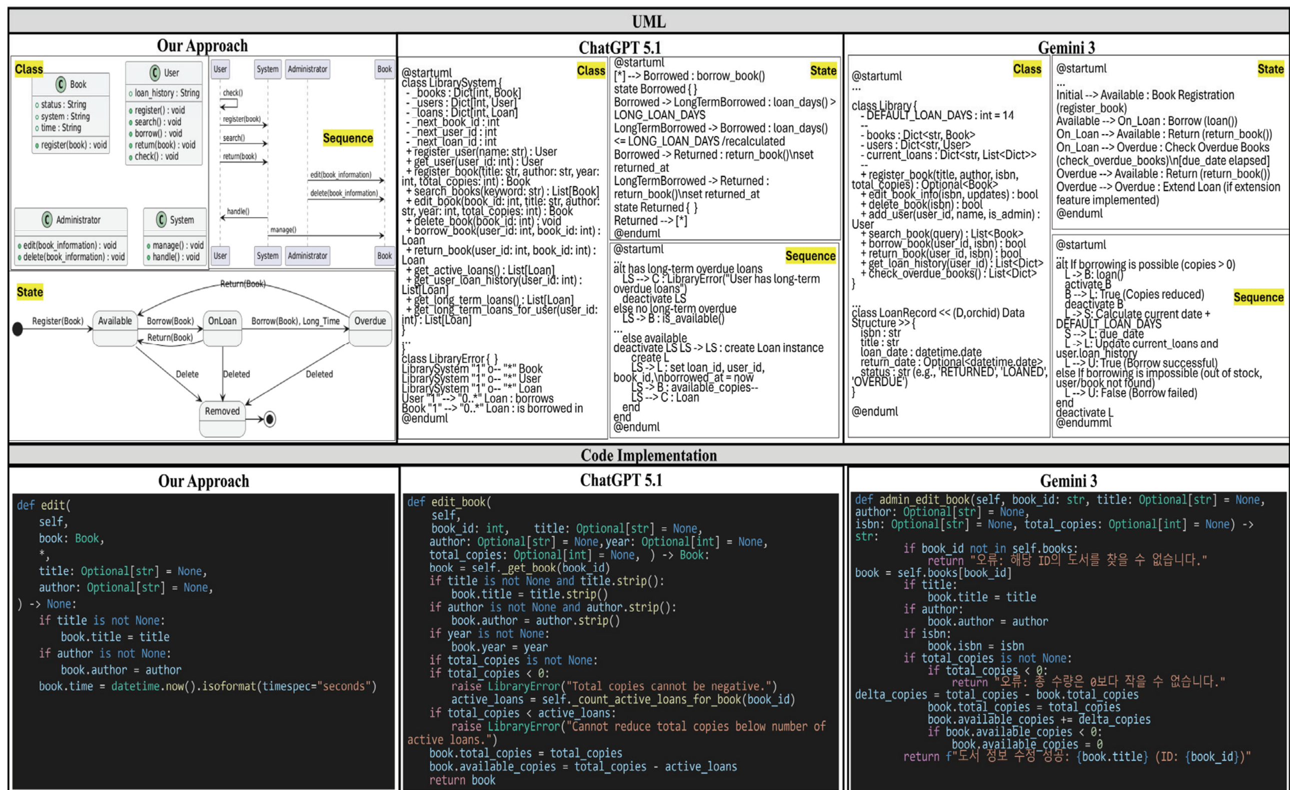

Fig. 4. Each Design and Code Generated by Our Research, GPT, and Gemini

로 재고 수량 및 대출 현황의 무결성 검증 등 실제 운영 환경에서의 다양한 예외 상황을 고려한다. 이러한 접근은 불필요한 조건문 증가로 인해 순환 복잡도(Cyclomatic Complexity)[14]를 높이고 코드 유지보수성을 저하시킨다. 반면 본 연구에서는 주어진 도서 객체의 책 제목과 저자를 단순히 갱신하고, 수정 시각을 타임스탬프로 기록하는 메타데이터만 업데이트하는 방식으로 구현하였다. 이러한 접근은 요구사항 범위 내에서 최소한의 변경만 수행한다. 이를 정량적으로 검증하기 위해 Table 2에서 LLM의 코드와 본 연구의 코드를 비교한다.

설계와 코드 수준에서 비교하였을 때 더 우수한 결과를 보여주는 값을 진하게 표시하였다. 이를 바탕으로 확인해보면, 본 연구의 평가가 대체로 우수함을 볼 수 있다.

설계 수준에서는 UML 스크립트의 길이를 비교한다. 길이가 길수록 세부 요소까지 포함되어 설계의 가독성 및 활용성을 떨어트린다. LLM만을 사용했을 때에는 UML 설계 생성 과정에서 추상화 수준을 낮추는 한계가 있다. 반면, 본 연구에서는 요구사항을 해석하는 방식을 통해 실제 시스템 동작을 설명하는 데 필요한 설계 요소를 추출하였다. 따라서 불필요한 세부 항목을 배제되어 더 짧은 스크립트 길이를 가진다. 설계의 복잡성을 최소화하며 일관성과 이해 가능성을 높인다.

코드 수준에서 Gemini의 코드는 요구사항에서 명시되지 않은 기능을 다수 추가하는 경향이 확인된다. 이러한 과도한 기능 확장은 코드 해석을 어렵게 만들고, 개발자에게 불필요한 이해 부담을 초래한다. 그러나, 모든 사례에서 공통으로 추가된 기능도 존재했는데, 중복 대출 방지 기능이다. 이 기능이 추가된 방식에서 두 접근법의 근본적인 차이가 드러난다. 자연어 입력에만 의존한 LLM은 자체적인 추론을 통해 해당 기능을 코드에 구현했다. 반면, 본 연구의 경우 절차에 따라 생성된 상태 다이어그램의 설계 정보가 LLM 입력에 포함되었다. 따라서 도서의 상태에 따라 대출이 불가능하다는 명확한 논리적 흐름이 체계적으로 구현된 것이다. 이는 단순한 추론에 의존하기보다 구조화된 설계 정보를 활용하여 시스템 개발에 더 안정적이고 예측 가능한 결과를 가져온다. 본 연구의 방식은 요구사항에 없는 기능을 추가하더라도 그 과정이 신뢰성

Table 2. Comparative Evaluation of LLMs and Our Approach

Criteria		LLM		Our Research	
		ChatGPT 5.1	Gemini 3	ChatGPT 5.1	Gemini 3
Design Level	LOC of Class Diagram Script	52	51	24	32
	LOC of Sequence Diagram Script	34	27	14	12
	LOC of State Diagram Script	12	13	11	11
Code Level	Unspecified Feature #	4	5	3	3
	Class #	5	3	5	4
	Method #	22	15	32	10
	Cyclomatic Complexity	60	35	28	32
	LOC	352	101	258	104

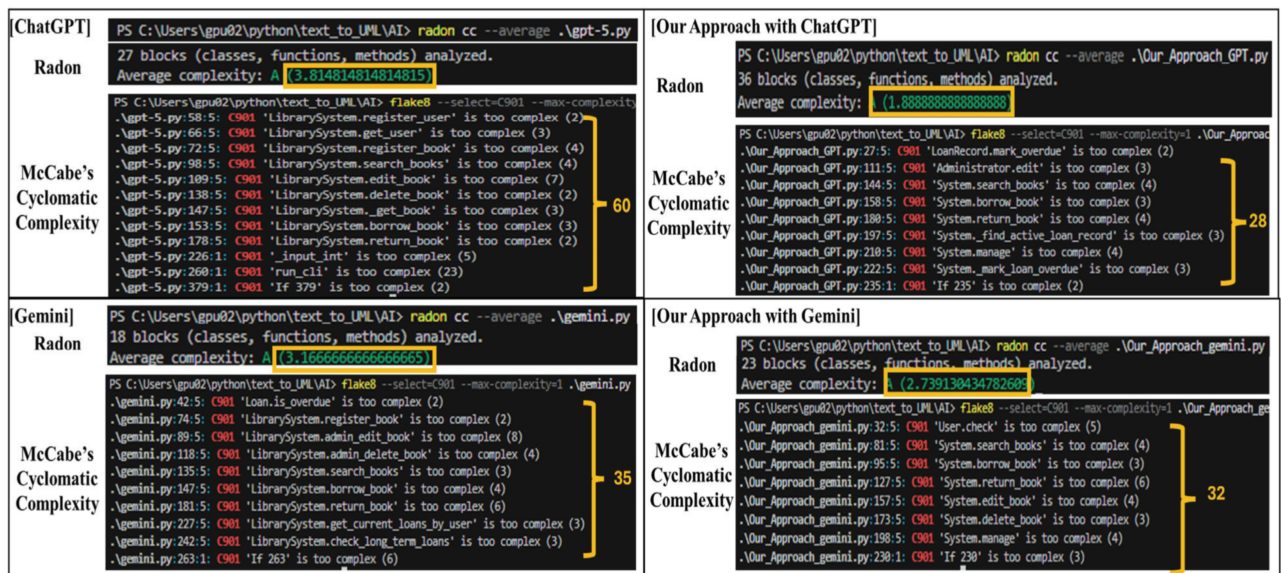

Fig. 5. Comparing the Cyclomatic Complexity of Our Research, GPT, and Gemini

있는 설계 정보에 기반하여 이루어졌기 때문에, 불필요한 복잡성을 방지하고 코드의 품질을 유지할 수 있다. 순환 복잡도의 경우, 두 개의 정적 분석기를 활용하여 분석하였다. Radon은 블록별 복잡도의 평균값을 보여주고, McCabe의 경우 블록별 복잡도의 개수를 보여준다[15].

Fig. 5는 순환 복잡도 측정 결과를 보여준다. 요구사항으로부터 생성된 Gemini의 코드는 평균 복잡도 3.17, 전체 복잡도 35이다. 본 연구 프로세스를 통해 Gemini를 사용한 코드는 평균 복잡도 2.74, 전체 복잡도는 32이다. 따라서 평균 복잡도는 13.5%, 전체 복잡도는 8.57% 낮다. 또한, 요구사항으로부터 바로 생성된 GPT의 코드는 평균 복잡도 3.8, 전체 복잡도 60이다. 본 연구 프로세스를 따라 GPT를 사용하여 생성된 코드는 평균 복잡도 1.89, 전체 복잡도 28이다. 즉, 평균 복잡도는 50.39%, 전체 복잡도는 53.3% 더 낮다. 전체적으로 LLM을 사용했을 때와 본 연구의 프로세스를 사용했을 때, 코드는 더 단순하고 유지보수성이 높은 코드를 생성한다.

5. 결 론

본 연구는 언어학 이론에 기반한 자연어 요구사항 분석, UML 설계 생성, 모델 기반 스켈레톤 코드 생성, 그리고 LLM을 활용한 코드 완성의 절차로 구성된다. 이러한 절차를 통해 자연어 요구사항의 모호성과 불완전성을 보완하고, 이전보다 신뢰성 있는 코드를 생성하는 메커니즘을 제안하였다. 사례연구로 도서관 관리 시스템을 대상으로 실험을 수행하였으며, 제안 메커니즘과 자연어 요구사항만을 입력으로 한 기존 LLM 기반 코드 생성 방식과 비교하였다. 비교 평가 결과, 제안 메커니즘을 활용한 경우, 코드 복잡도가 낮게 나타나, 구조화된 입력이 코드 품질 향상에 기여함을 확인할 수 있었다. 비교 사례는 300줄 정도의 간단한 예시를 사용하였다. 더 크고 복잡한 시스템일수록 이 차이가 명확히 보일 것으로 기대한다.

향후 연구에서는 제안된 메커니즘의 적용 범위를 다양한 도메인과 복잡한 시스템으로 확장하고, 정량적・정성적 품질 평가를 더욱 체계화할 계획이다. 또한, 자동화된 모델 변환 및 프롬프트 최적화 기법을 통합하여 코드 생성 과정의 효율성을 높이고자 한다. 현재 규칙 기반 절차에 의존하는 메커니즘을 각 단계별 AI 학습을 통해 고도화하여 성능을 향상시킬 것으로 기대한다. 궁극적으로, 본 연구는 AI 기반 코드 생성 기술이 단순히 코드를 제공하는 수준을 넘어, 개발 생산성을 높이고 코드 품질을 보장하며 소프트웨어 개발에 대한 신뢰성을 확보하는 데 기여할 것이다.

References

[1] Stack Exchange Inc., Section 3 (AI) of "2024 Developer Survey," Stack Exchange Inc., 2024, [Internet], https://survey.stackoverflow.co/2024/ai

[2] A. Keluskar, A. Bhattacharjee, and H. Liu, "Do LLMs understand ambiguity in text? a case study in open-world question answering," *2024 IEEE International Conference on Big Data (BigData)*, pp.7485-7490, 2024.

[3] S. Ye et al., "Prompt alchemy: Automatic prompt refinement for enhancing code generation," arXiv preprint arXiv:2503.11085, 2025.

[4] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM Transactions on Software Engineering and Methodology*, Vol.34, No.2, pp.1-23, 2025.

[5] S. Morales, R. Clariso, and J. Cabot, "Impromptu: A framework for model-driven prompt engineering," *Software and Systems Modeling*, pp.1-19, 2025.

[6] H. Su, J. Ai, D. Yu, and H. Zhang, "An Evaluation Method for Large Language Models' Code Generation Capability," *2023 10th International Conference on Dependable Systems and Their Applications (DSA)*, 2023.

[7] N. Chomsky, "Syntactic structures," Walter de Gruyter, USA, 2002.

[8] C. J. Fillmore, "The Case for Case," HR&W, NewYork, 1968.

[9] Y. Jin, C. Seo, J. Kong, K. Kim, and R. Y. C. Kim, "Best Practices on AI driven Requirement Engineering in Software Development Life Cycle", in *Proceedings of the 27th Korea Conference on Software Engineering*, Vol.27, No.1, pp.289-292, 2025.

[10] S. Bird, E. Klein, and E. Loper, "Natural language processing with Python: analyzing text with the natural language toolkit," O'Reilly Media, Inc., 2009.

[11] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, "EMF: eclipse modeling framework," Pearson Education, 2008.

[12] A. Sarkar and I. Drosos, "Vibe coding: programming through conversation with artificial intelligence," arXiv preprint arXiv:2506.23253, 2025.

[13] A. Goulston, "AI Prompting Methods from Japan: Fukatsu and Shunsuke Methods," MacroLingo, [Internet], https://macrolingo.com/japanese-fukatsu-shunsuke-ai-prompting-methods/

[14] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol.SE-2, No.4, pp.308-320, 1976.

[15] M. Lacchia, Radon 4.1.0 documentation [Internet], https://radon.readthedocs.io/en/latest/

진 예 진

https://orcid.org/0009-0000-4996-2436
e-mail : yejin_jin@g.hongik.ac.kr

2023년 홍익대학교 소프트웨어융합학과 (학사)
2025년 홍익대학교 소프트웨어융합학과 (석사)
2025년~현 재 홍익대학교 소프트웨어융합학과 (박사과정)
관심분야 : Software Engineering, Natural Language-based Code Generation, Text-to-Emotion Recognition

김 장 환

https://orcid.org/0000-0003-0185-4406
e-mail : lentoconstante@hongik.ac.kr

2019년 아이다호주립대학교 컴퓨터과학과 (학사)
2022년 홍익대학교 소프트웨어융합학과 (석사)
2022년~현 재 홍익대학교 소프트웨어융합학과 박사
관심분야 : AI Software Testing, Software Engineering, Reinforcement Learning Software Validation

김 기 두

https://orcid.org/0009-0003-7726-1970
e-mail : kdkim@tta.or.kr

2014년 홍익대학교 컴퓨터정보통신공학과 (박사)
2005년~현 재 한국정보통신기술협회 AI디지털융합단 팀장
관심분야 : Test Maturity Model, Software Testing, AI Software Testing

김 영 철

https://orcid.org/0000-0002-2147-5713
e-mail : bob@hongik.ac.kr

2000년 일리노이공대(IIT) 소프트웨어공학과 (박사)
2001년~현 재 홍익대학교 소프트웨어융합학과 교수
관심분야 : Scenario Based Validation for AI Reinforcement Learning, Metaverse(VR/AR), Software Engineering