사물인터넷융복합논문지 Vol. 12, No. 4, pp. 33-44, 2026 https://doi.org/10.20465/KIOTS.2026.12.4.033

구조 선이해 기반 다중 에이전트를 활용한 DirectX 기반 MMORPG 레거시 시스템 버그 수정 및 성능 평가

류지수[1], 정순기[2*]
[1]경북대학교 컴퓨터학부 대학원 박사과정, [2]경북대학교 IT대학 컴퓨터학부 교수

Bug Fixing and Performance Evaluation in DirectX-Based MMORPG Legacy Systems Using a Structure-Aware Multi-Agent Approach

Ji Soo Ryu[1], Soon Ki Jung[2*]
[1]Ph.D. student, Graduate School of Computer Science and Engineering, Kyungpook National University
[2]Professor, School of Computer Science and Engineering, College of IT, Kyungpook National University

요 약 본 논문은 대규모 DirectX 기반 MMORPG 레거시 시스템을 대상으로, 구조 선이해(Structure-aware) 다중 에이전트 접근법을 활용한 버그 수정 방법론을 제안하고 그 성능을 평가한다. 본 논문은 AI에게 즉각적인 버그 수정을 지시할 경우 전체 프로젝트 구조와 상충하는 국소적 수정에 그치고 마는 생성형 AI 기반 바이브코딩(Vibe coding)의 한계점을 다룬다. 이러한 문제는 모듈 간 의존성, 호출 체인, 스프라이트 기반 프레임 및 모션 처리 구조가 복합적으로 얽혀 있는 MMORPG 환경에서 더욱 두드러진다. 제안하는 방법론의 성능을 평가하기 위해, 단일 에이전트 기반의 직접 지시 방식과 시스템 구조 선이해를 기반으로 한 다중 에이전트 방식을 비교 실험하였다. 동일한 코드베이스와 과제를 기준으로 버그 해결 성공률, 추가 버그 발생률, 요구사항 일치도, 반복 수정 횟수 및 구조 해석 정확도를 측정하였다. 실험 결과, 코드 구현 이전에 시스템 구조와 호출 체인을 선행적으로 분석하는 방식이 버그 해결률과 요구사항 일치도를 크게 향상시키는 동시에, 추가적인 오류 발생과 반복 수정 횟수를 대폭 감소시키는 것으로 나타났다. 결론적으로, 구조 선이해 다중 에이전트 시스템을 활용하는 것은 대규모 레거시 게임 프로젝트에서 즉시 구현 중심의 프롬프트 방식보다 훨씬 더 효과적이고 안정적인 버그 수정 프로세스를 제공한다.

주제어 : 다중 에이전트, 생성형 AI, 바이브코딩, 레거시 시스템, MMORPG, 구조 선이해 기반 코드 생성, 버그 수정 자동화, 성능 평가

Abstract This paper proposes and evaluates a bug-fixing methodology using a structure-aware multi-agent approach for DirectX-based MMORPG legacy systems. This paper addresses the limitations of generative AI-based vibe coding, where immediately instructing AI to fix bugs often leads to localized fixes that conflict with the overall project structure. These issues are especially prominent in MMORPG environments, where module dependencies, call chains, and sprite-based frame/motion processing are closely interconnected. To evaluate performance, we compared a single-agent direct instruction baseline with our proposed multi-agent approach based on prior structural understanding. Using the same codebase and tasks, we measured bug resolution success rate, additional bug occurrence rate, requirement alignment, revision iterations, and structural interpretation accuracy. The results showed that analyzing the system structure and call chains before implementation significantly improved resolution rates and requirement alignment while drastically reducing additional errors and revision iterations. In conclusion, utilizing a structure-aware multi-agent system provides a highly effective and stable bug-fixing process in large-scale legacy game projects, outperforming immediate implementation-focused prompts.

Key Words : Multi-Agent System, Generative AI, Vibe Coding, Legacy System, MMORPG, Structure-Aware Code Generation, Automated Bug Fixing, Performance Evaluation

본 논문은 과학기술정보통신부 및 정보통신산업진흥원의 지역산업 SW인재양성기반조성사업 지원으로 연구되었음.
*교신저자 : 정순기(skjung@knu.ac.kr)
접수일 2026년 06월 02일 수정일 2026년 06월 29일 심사완료일 2026년 08월 06일

1. 서론

최근 생성형 인공지능의 발전은 소프트웨어 개발 방식에 큰 변화를 가져오고 있다. 특히 Transformer 기반 대규모 언어모델은 자연어 이해와 생성 능력을 바탕으로 다양한 작업에 활용되고 있으며[1], 대규모 언어모델의 few-shot 학습 능력은 자연어 기반 개발 보조 가능성을 확대하였다[2]. 또한 Claude와 같은 대규모 언어모델 기반 AI는 코드 생성, 리팩터링, 버그 수정, 문서화 등 다양한 개발 보조 작업에서 활용되고 있으며, 코드 특화 언어모델은 자연어 설명으로부터 프로그램을 생성하는 가능성을 보였다[3]. 이러한 흐름 속에서 개발자가 세부 구현보다 의도와 방향을 중심으로 AI에게 작업을 위임하는 바이브코딩(vibe coding) 방식이 새로운 개발 패러다임으로 등장하고 있다.

그러나 실제 산업 환경, 특히 대규모 레거시 시스템에서는 바이브코딩이 항상 안정적인 결과를 보장하지 않는다[4]. DirectX 기반 MMORPG와 같은 장기 개발 프로젝트는 수년간 누적된 방대한 코드와 복잡한 구조를 가지며, 클라이언트, 서버, 렌더링, 네트워크, UI, 데이터 처리 로직이 서로 강하게 결합되어 있다. 레거시 시스템의 유지보수와 진화에서는 구조 이해, 변경 영향 분석, 설계 복원 과정이 중요한 요소로 다루어진다[5][6]. 이러한 환경에서는 단순한 코드 수정 능력뿐 아니라, 전체 구조와 호출 흐름을 고려한 변경이 요구된다.

생성형 AI 기반 바이브코딩의 가장 큰 한계는 국소적 수정에 머무를 가능성이 높다는 점이다. 개발자가 AI에게 “이 버그를 고쳐달라”거나 “이 기능을 구현해달라”고 직접 지시할 경우, AI는 입력된 정보 범위 안에서 즉시 해결책을 생성하려 한다. 이 과정에서 전체 호출 체인, 데이터 흐름, 모듈 간 의존성을 충분히 고려하지 못하면 표면적인 오류는 일부 해결되더라도 근본 원인이 남거나, 다른 기능에서 새로운 부작용이 발생할 수 있다. 즉, 바이브코딩은 빠른 수정에는 유리하지만, 복잡한 레거시 시스템에서는 잘못된 수정 위치를 선택하거나 요구사항과 맞지 않는 코드를 생성할 위험이 있다.

이러한 한계를 극복하기 위해서는 AI에게 즉시 코드를 생성하도록 요청하는 방식이 아니라, 문제 해결 이전에 시스템 구조를 먼저 파악하도록 하는 절차가 필요하다. 대규모 MMORPG 프로젝트에서 하나의 오류는 단일 함수 내부에서만 발생하지 않고, 입력 처리, 상태 변경, 애니메이션 갱신, 렌더링, 충돌 판정, 서버 통신 등 여러 계층의 호출 체인과 연결될 수 있다. 따라서 AI가 올바른 수정 위치를 찾기 위해서는 관련 함수 목록을 단순히 확인하는 수준을 넘어, 호출 흐름이 어디에서 시작되어 어느 계층까지 이어지는지, 수정이 다른 기능에 어떤 영향을 줄 수 있는지를 선행적으로 분석해야 한다.

특히 2D 기반 MMORPG에서 널리 사용되는 스프라이트(sprite) 그래픽 시스템은 이러한 문제를 더욱 부각시킨다. 실시간 그래픽 시스템에서는 이미지 리소스, 프레임 인덱스, 좌표계, 렌더링 순서가 함께 작동하며[7], 컴퓨터 그래픽스에서는 좌표 변환과 렌더링 구조가 시각적 결과에 직접적인 영향을 미치는 요소로 설명된다[8]. 스프라이트 기반 렌더링은 이미지 리소스, 좌표값, 프레임 구성, 기준점(anchor), 입력 영역(rect) 등이 복합적으로 연결된 구조를 가지므로, AI가 단편적인 코드 정보만으로는 실제 동작을 정확히 이해하기 어렵다. 그 결과 스킬 모션 속도 조정이나 UI 좌표 보정과 같은 문제에서 캐릭터 이동 속도, 전역 애니메이션 속도 등 잘못된 대상을 수정할 가능성이 있다.

이에 본 연구는 구조 선이해 기반 다중 에이전트 협업 모델을 제안한다. 제안 방식은 하나의 AI가 즉시 코드를 생성하는 방식이 아니라, 문제 분석, 코드 생성, 검증, 평가 역할을 분리하여 단계적으로 수행한다. 이러한 접근은 LLM을 단순 응답 생성 도구로 활용하는 것을 넘어, 추론과 행동을 결합하거나[9], 피드백을 기반으로 결과를 반복 개선하며[10], 코드 생성·테스트·평가 역할을 분리한 다중 에이전트 방식[11] 및 복수 에이전트 간 협업 프레임워크[12]와도 연결된다.

본 연구에서 제안하는 다중 에이전트 구조는 Planner, Code Generator, QA, Evaluator의 네 가지 역할로 구성된다. Planner는 문제와 관련된 호출 체인, 데이터 흐름, 수정 위치, 영향 범위를 분석하고, Code Generator는 Planner가 제시한 범위 안에서 최소 수정 원칙에 따라 코드를 생성한다. QA는 수정 결과가 기존 구조와 충돌하지 않는지, 부작용이나 회귀 위험이 없는지를 검토하며, Evaluator는 요구사항 일치도와 구조 적합성을 평가하여 재수행 여부를 판단한다. 이를 통해 생성형 AI를 단순 코드 생성기가 아니라 구조 분석과 검증을 포함한 협업형 개발 도구로 활용하고자 한다.

본 논문의 주요 기여도는 다음과 같다.

첫째, 대규모 DirectX 기반 MMORPG 레거시 프로젝트에서 생성형 AI 기반 바이브코딩이 국소적 수정, 오수정, 부작용을 유발할 수 있음을 분석하였다.

둘째, 단일 에이전트 직접 지시 방식의 한계를 보완하기 위해 Planner, Code Generator, QA, Evaluator로

구성된 구조 선이해 기반 다중 에이전트 협업 모델을 제안하였다.

셋째, 스프라이트 기반 스킬 모션 수정, UI anchor/pivot 보정, 선택적 기능 변경 시나리오를 통해 직접 지시 방식과 구조 선이해 방식을 정량적으로 비교하였다.

넷째, 실험 결과를 바탕으로 대규모 레거시 소프트웨어 환경에서 생성형 AI를 구조 분석과 검증을 포함한 협업형 개발 도구로 활용해야 함을 제시하였다.

2. 이론적 배경

2.1 생성형 AI와 코드 생성

최근 생성형 인공지능(Generative AI)은 자연어 처리 뿐만 아니라 소프트웨어 개발 영역에서도 중요한 역할을 수행하고 있다. 특히 대규모 언어모델(Large Language Model, LLM)은 자연어 기반의 요구사항을 해석하여 코드 생성, 버그 수정, 리팩터링, 문서화 등을 지원할 수 있으며, 개발자의 생산성을 향상시키는 도구로 활용되고 있다. 이러한 AI 기반 코드 생성은 기존의 정형화된 자동화 도구와 달리, 비정형 요구사항에 대해서도 유연하게 대응할 수 있다는 장점을 가진다.

그러나 LLM 기반 코드 생성은 입력으로 제공된 문맥에 크게 의존하며, 프로젝트 전체 구조를 완전하게 이해하지 못한 상태에서 부분적인 정보만을 기반으로 추론을 수행한다는 한계를 가진다. 특히 대규모 코드베이스에서는 전체 시스템의 흐름과 데이터 구조, 모듈 간 관계를 충분히 반영하지 못할 경우, 생성된 코드가 실제 시스템과 정합성을 가지지 못하는 문제가 발생할 수 있다[4].

2.2 바이브코딩(Vibe Coding)의 개념

바이브코딩은 개발자가 세부 구현보다는 기능의 의도와 방향을 중심으로 AI에게 작업을 위임하는 개발 방식으로, 최근 생성형 AI의 발전과 함께 주목받고 있는 새로운 패러다임이다. 이 방식에서는 개발자가 직접 코드를 작성하기보다, 자연어 기반으로 요구사항을 전달하고 AI가 이를 코드로 변환하는 과정을 반복적으로 수행한다.

바이브코딩은 반복 작업을 줄이고 빠른 프로토타이핑을 가능하게 한다는 장점이 있으나, 명확한 구조 정의 없이 즉흥적으로 활용될 경우 문제를 야기할 수 있다. 특히 복잡한 시스템에서는 AI가 전체 구조를 고려하지 못한 채 국소적인 코드 생성에 집중하게 되며, 이는 결과적으로 유지보수성과 안정성을 저하시킬 수 있다. 따라서 바이브코딩의 효과를 극대화하기 위해서는 단순한 명령 중심 접근이 아닌, 구조 기반의 단계적 활용 방식이 요구된다.

2.3 레거시 시스템과 MMORPG 구조의 특성

레거시 시스템은 장기간에 걸쳐 개발 및 유지보수된 소프트웨어로, 다양한 기능이 누적되어 복잡한 구조를 가지는 특징이 있다[5].

MMORPG와 같은 대규모 게임 시스템은 클라이언트와 서버 간의 실시간 상호작용, 데이터 동기화, 상태 관리, UI 렌더링, 네트워크 통신 등 다양한 요소가 결합된 구조를 가진다. 이러한 시스템에서는 하나의 기능 변경이 다른 모듈에 영향을 미칠 가능성이 높으며, 코드 간 의존성이 복잡하게 얽혀 있는 경우가 많다. 또한 과거 개발 과정에서 형성된 암묵적 규칙과 예외 처리 방식이 존재하여, 단순한 코드 분석만으로 전체 구조를 이해하기 어렵다. 이로 인해 외부 도구나 AI가 부분적인 정보만으로 정확한 수정이나 확장을 수행하기에는 한계가 존재한다[6].

2.4 스프라이트 기반 그래픽 시스템과 좌표 처리 문제

2D 기반 게임에서는 스프라이트(sprite) 그래픽 시스템이 널리 사용되며, 하나의 이미지 파일 내에 여러 프레임을 저장하고 좌표를 기준으로 이를 분리하여 렌더링하는 방식이 일반적이다[7][8]. 이 과정에서 스프라이트의 출력 위치, 프레임 인덱스, 기준점(anchor), 입력 영역(rect) 등 다양한 요소가 함께 고려되어야 한다.

그러나 이러한 정보는 코드뿐만 아니라 외부 리소스 파일, 데이터 테이블, 경험적 보정값 등에 분산되어 있는 경우가 많다. 따라서 단순히 코드 일부만을 기반으로 스프라이트의 실제 동작을 정확히 이해하기 어렵다. 생성형 AI 역시 이러한 구조적·시각적 맥락을 완전히 파악하지 못할 경우, 좌표 보정이나 UI 정렬과 같은 작업에서 부정확한 결과를 생성할 가능성이 높다.

2.5 구조 기반 AI 활용의 필요성

앞서 살펴본 바와 같이, 생성형 AI 기반 코드 생성은 입력 문맥과 지시 방식에 크게 영향을 받는다. 특히 대규모 레거시 시스템에서는 코드 일부만으로 전체 동작을 판단하기 어렵기 때문에, AI가 문제의 실제 발생 위치나 수정 범위를 잘못 해석할 가능성이 있다. 이러한 한계는 단순한 코드 생성 능력의 문제가 아니라, AI가 문제 해결 이전에 시스템 구조와 맥락을 충분히 파악하지 못한 상

태에서 응답을 생성하기 때문에 발생한다.

최근 LLM 기반 연구에서는 이러한 한계를 보완하기 위해 다양한 에이전트 활용 방식이 제안되고 있다. ReAct는 언어모델이 추론(reasoning)과 행동(action)을 결합하여 문제를 단계적으로 해결하도록 하는 접근 방식이다[9]. 이 방식은 LLM이 단순히 최종 답변을 생성하는 것이 아니라, 중간 추론 과정을 통해 필요한 행동을 선택하고 그 결과를 다시 반영한다는 점에서 의미가 있다. 그러나 ReAct의 핵심은 일반적인 문제 해결 과정에서 추론과 행동을 연결하는 데 있으며, 특정 소프트웨어 시스템의 호출 체인, 모듈 의존성, 수정 영향 범위를 사전에 분석하도록 설계된 것은 아니다.

Reflexion은 LLM이 이전 시도에서 발생한 실패나 피드백을 언어적으로 반영하여 다음 응답을 개선하도록 하는 방식이다[10]. 이는 반복 수정이나 자기 개선이 필요한 코드 생성 작업과 관련성이 높다. 그러나 Reflexion은 실패 이후의 개선 과정에 중점을 두기 때문에, 최초 수정 전에 어떤 함수와 데이터 구조를 먼저 확인해야 하는지, 수정 대상과 제외 대상을 어떻게 구분해야 하는지에 대한 구조 분석 절차는 상대적으로 약하다. 따라서 초기 구조 해석이 잘못된 경우, 이후의 반복 개선도 잘못된 수정 방향 안에서 수행될 가능성이 있다.

AgentCoder는 코드 생성, 테스트, 평가 역할을 분리한 다중 에이전트 기반 코드 생성 방식을 제안한다[11]. 이 연구는 단일 LLM이 모든 작업을 수행하는 방식보다, 역할을 분리한 에이전트 간 협력이 코드 생성 품질을 높일 수 있음을 보여준다. 그러나 AgentCoder는 주로 코드 생성 결과와 테스트 통과 여부에 초점을 두며, 레거시 프로젝트에서 수정 전에 호출 흐름을 추적하고, 실제 수정 위치가 어느 계층에 존재하는지 판단하며, 수정하면 안 되는 전역 로직이나 공통 모듈을 배제하는 절차까지 명시적으로 다루지는 않는다.

AutoGen은 여러 LLM 에이전트가 대화 기반으로 협업할 수 있는 범용 프레임워크를 제안한다[12]. AutoGen은 다양한 역할의 에이전트를 구성하고 상호작용시킬 수 있다는 점에서 다중 에이전트 시스템 구현의 기반을 제공한다. 그러나 AutoGen 자체는 특정 도메인에 종속된 구조 분석 방법론이 아니라, 에이전트 간 대화와 협업을 가능하게 하는 일반 프레임워크에 가깝다. 따라서 대규모 게임 레거시 프로젝트에서 어떤 구조 정보를 먼저 분석해야 하는지, 어떤 기준으로 수정 범위를 제한해야 하는지, 어떤 방식으로 부작용 가능성을 평가해야 하는지는 별도의 연구 설계가 필요하다.

이상의 선행 연구들은 LLM이 추론, 피드백, 역할 분리, 에이전트 협업을 통해 코드 생성 및 문제 해결 능력을 향상시킬 수 있음을 보여준다. 그러나 이들 연구는 주로 LLM의 문제 해결 절차, 반복 개선, 역할 분리, 범용 협업 구조에 초점을 둔다. 반면 대규모 레거시 시스템의 유지보수에서는 코드 생성 이전에 “무엇을 수정해야 하는가”뿐 아니라 “어디를 수정해서는 안 되는가”를 판단하는 과정이 중요하다. 즉, 문제 해결의 핵심은 코드 생성 자체가 아니라, 수정 전 구조 해석과 수정 범위 제한에 있다.

구조 선이해 기반 에이전트는 이러한 점에서 기존 프레임워크들과 차별성을 가진다. ReAct가 추론과 행동의 결합을 강조하고, Reflexion이 실패 피드백을 통한 자기 개선을 강조하며, AgentCoder가 코드 생성·테스트·평가의 역할 분리를 제안하고, AutoGen이 범용 다중 에이전트 협업 구조를 제공한다면, 본 연구는 기존 에이전트 연구의 개념을 대규모 DirectX 기반 MMORPG 레거시 프로젝트의 코드 수정 문제에 적용하고, 수정 전 구조 해석과 수정 범위 제한을 핵심 절차로 설정한다. 특히 수정 전에 호출 체인, 데이터 흐름, 핵심 변수, 수정 위치, 수정 제외 범위, 부작용 가능성을 먼저 분석하고, 이 분석 결과를 기준으로 코드 생성과 검증을 수행한다는 점에서 차별적이다.

예를 들어 스프라이트 기반 스킬 모션 수정 문제에서 단순 직접 지시 방식은 “속도를 빠르게 하라”는 요구를 캐릭터 이동 속도나 전역 애니메이션 속도 변경으로 잘못 해석할 수 있다. 반면 구조 선이해 기반 에이전트는 먼저 스킬 입력 함수, 스킬 이펙트 호출 함수, 스프라이트 프레임 설정 함수로 이어지는 호출 체인을 분석하고, 실제 수정 대상이 캐릭터 속도가 아니라 특정 이펙트의 프레임 유지 시간임을 식별한다. 이처럼 본 연구의 핵심은 에이전트 협업 자체가 아니라, 에이전트 협업을 통해 레거시 시스템의 구조적 맥락을 먼저 파악하고 수정 범위를 제어하는 데 있다.

3. 제안 방법

3.1 연구 개요 및 목적

본 연구는 생성형 AI를 활용한 바이브코딩 과정에서, 문제 해결 방식에 따라 결과 품질이 어떻게 달라지는지를 실증적으로 분석하는 것을 목적으로 한다. 최근 생성형 AI는 코드 생성, 버그 수정, 기능 구현 등 다양한 개발

업무에 활용되고 있으나, 대규모 레거시 프로젝트에서는 단순 지시만으로 안정적인 결과를 얻기 어려운 한계가 존재한다. 특히 구조가 복잡하고 장기간 누적 개발된 게임 프로젝트에서는, AI가 문제의 증상만을 기준으로 국소적인 수정에 머무르거나 예상하지 못한 부작용을 발생시키는 사례가 나타날 수 있다. 특히 DirectX 기반 MMORPG는 실시간 렌더링, 프레임 기반 애니메이션, 네트워크 동기화, UI 이벤트 처리 등이 동시에 결합되어 있어 일반 업무 소프트웨어보다 구조적 복잡성이 높다.

이에 본 연구는 동일한 AI 모델을 기반으로 서로 다른 작업 방식에 따른 성능 차이를 비교하고자 한다. 실험에는 Claude Opus 4.6 모델을 사용하였으며[13], 단일 에이전트 기반의 직접 지시 방식(Baseline)과 다중 에이전트 기반의 구조 선이해 방식(Proposed Method)을 비교 대상으로 설정하였다. 직접 지시 방식은 개발자가 “이 버그를 수정하라”, “이 기능을 구현하라”와 같이 결과 중심으로 명령하는 방식이며, 구조 선이해 방식은 문제 해결 이전에 시스템 구조, 데이터 흐름, 의존 관계뿐 아니라 호출 체인(call chain)이 어느 함수에서 시작되어 어느 계층까지 이어지는지를 먼저 분석한 후 수정 전략을 수립하도록 설계된 방식이다.

본 연구에서 강조하는 구조적 차이는 단순히 관련 파일을 많이 제공하는 것에 있지 않다. 핵심은 특정 버그나 기능 요구가 어떤 함수 호출 흐름 안에서 발생하며, 그 호출 체인이 UI, 렌더링, 애니메이션, 네트워크, 서버 처리, 데이터 저장 계층 중 어디까지 연결되는가를 파악하는 데 있다. 예를 들어 클라이언트에서 스킬 모션이 느리게 보이는 문제는 단순히 화면 출력 함수의 문제가 아니라, 스킬 상태 변경 함수, 애니메이션 프레임 갱신 함수, 스프라이트 렌더링 함수, 타격 판정 함수와 연속적으로 연결될 수 있다. 따라서 AI가 호출 체인을 충분히 파악하지 못하면, 실제 수정해야 할 frameDelay나 프레임 테이블이 아니라 캐릭터 이동 속도 또는 전체 애니메이션 속도를 수정하는 잘못된 판단을 내릴 수 있다.

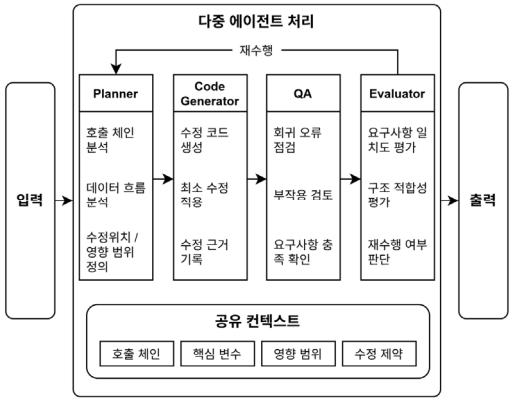
본 연구의 핵심 가설은 다음과 같다. 첫째, 구조 분석 없이 즉시 문제 해결을 요청하는 방식은 문제의 근본 원인보다 증상 중심 수정에 그칠 가능성이 높다. 둘째, 시스템 구조와 호출 체인을 선행적으로 파악한 후 단계적으로 작업을 수행하는 방식은 버그 해결 정확도와 요구사항 일치도를 향상 시킨다. 셋째, 역할이 분리된 다중 에이전트 기반 구조적 접근 방식은 단일 에이전트 방식보다 안정적인 결과를 도출할 가능성이 높다. 이러한 가설을 검증하기 위한 실험 환경, 시나리오, 평가 지표 및 결과는 4장에서 제시한다.

이를 검증하기 위해 본 연구는 DirectX 기반 MMORPG 레거시 프로젝트를 실험 대상으로 선정하였다. 해당 프로젝트는 수십만 라인 이상의 C++ 코드로 구성되어 있으며, 클라이언트와 서버 구조, UI 처리, 네트워크 로직, 데이터 처리, 스프라이트 기반 그래픽 시스템 등 다양한 모듈이 상호 연결된 형태를 가진다.

실험은 동일한 코드베이스와 동일한 문제 집합을 대상으로 수행되었으며, 빌드 성공 여부, 기능 정상 동작 여부, 추가 버그 발생 여부, 반복 수정 횟수, 요구사항 일치도 등을 기준으로 결과를 측정하였다. 특히 스프라이트 프레임 기반 캐릭터 스킬 모션 속도 조정 사례를 통해, 자연어 요구를 직접 해석하는 방식과 구조 및 호출 체인을 먼저 분석한 뒤 수정하는 방식 간의 차이를 정량적으로 비교하였다.

3.2 제안 방법의 전체 처리 흐름

제안 방법의 전체 흐름은 다음과 같은 순서로 진행된다.



1. 문제 입력 및 작업 유형 분류
2. 관련 구조 및 의존성 분석
3. 원인 후보 도출 및 영향 범위 추정
4. 수정 또는 구현 전략 수립
5. 코드 생성 또는 수정 수행
6. 정적·논리적 검증 수행
7. 결과 평가 및 점수화
8. 기준 미달 시 재계획 및 재생성
9. 최종 결과 확정 및 변경 내용 문서화

이러한 순환 구조는 단순한 1회성 코드 생성과 달리, 여러 단계의 검증과 재수행을 포함한다. 따라서 AI가 초기

응답에서 완전한 정답을 내지 못하더라도, 반복적 개선 과정을 통해 점진적으로 문제 해결 정확도를 높일 수 있다.

제안 시스템은 역할 분리 기반 소프트웨어 엔지니어링 에이전트 연구의 구조를 참고하여 다음과 같은 형태로 구성하였다[15].

3.2.1 에이전트 기반 역할 분리 구조

본 연구에서는 Claude Opus 4.6을 기반으로 다중 에이전트 구조를 구성하였다. 기본적으로는 Planner, Code Generator, QA, Evaluator의 4개 에이전트를 사용하지만, 실제 성능 향상을 위해 각 역할 안에 필요한 기능을 세분화하여 다음과 같이 정의할 수 있다.

(1) Planner 에이전트: 구조 분석 및 해결 전략 수립

Planner는 전체 시스템의 구조를 이해하고, 현재 문제를 어느 계층에서 다루어야 하는지를 판단하는 역할을 담당한다. 이 에이전트는 단순히 관련 함수 몇 개를 보는 것이 아니라, 문제와 관련된 모듈, 클래스, 구조체, 데이터 흐름, 호출 흐름, 의존 관계를 먼저 파악한다.

Planner의 주요 기능은 다음과 같다.

- 문제 유형 분류: 버그 수정, 기능 구현, 그래픽 좌표 수정 등
- 관련 파일 및 모듈 식별
- 구조체 및 클래스 관계 분석
- 호출 흐름과 데이터 흐름 정리
- 원인 후보 도출
- 수정 시 영향받을 가능성이 있는 영역 예측
- 변경 전략 초안 작성

예를 들어 UI 클릭 오류가 발생한 경우, Planner는 단순히 draw 함수만 보는 것이 아니라, draw rect, click rect, UI 이벤트 처리, 해상도 스케일링, 스프라이트 anchor 규칙, 관련 상수 정의까지 함께 조사하도록 설계한다. 문제를 증상 중심이 아니라 구조 중심으로 해석하도록 유도한다. 이러한 구조 기반 추론 방식은 ReAct의 단계적 추론 접근과도 관련된다[9].

(2) Code Generator 에이전트: 제한된 범위 내 코드 생성 및 수정

Code Generator는 Planner가 정의한 수정 전략을 기반으로 실제 코드 변경안을 생성한다[11][12]. 이 에이전트의 가장 중요한 원칙은 “최소 수정 범위(minimal change principle)” 이다. 즉, 대규모 레거시 프로젝트에서 불필요한 리팩터링이나 광범위한 변경을 지양하고, Planner가 지정한 범위 내에서 필요한 부분만 수정하도록 한다.

Code Generator의 주요 기능은 다음과 같다.

- 수정 대상 파일 및 함수 한정
- 기존 인터페이스 유지
- 기존 코드 스타일 최대 유지
- 필요한 로그 및 방어 코드 추가
- 코드 diff 형태 산출
- 수정 사유와 예상 영향 범위 설명

특히 본 연구에서는 단순 코드 생성보다 “설계 기반 생성”을 중요하게 본다. 즉, Generator는 임의로 정답을 추측하지 않고, Planner가 정리한 구조적 해석을 바탕으로 코드 생성에 들어간다.

(3) QA 에이전트: 논리 검증 및 회귀 위험 탐지

- 수정된 로직이 원인에 맞는가?
- 기존 데이터 흐름과 충돌하지 않는가?
- 상태값 일관성이 유지되는가?
- null, 범위, 초기화, 순서 문제는 없는가?
- 다른 모듈과의 인터페이스를 깨뜨리지 않는가?
- draw rect와 click rect가 동시에 고려되었는가?
- 스프라이트 좌표 수정 시 anchor/pivot이 반영되었는가?
- Git 협업 관점에서 과도한 변경을 일으키지 않는가?

즉, QA 에이전트는 테스트 코드 수준의 검증뿐 아니라, 구조적 회귀 위험을 판단하는 역할을 한다[10].

(4) Evaluator 에이전트: 결과 점수화 및 재수행 판단

Evaluator는 최종 결과가 실제 작업 목표에 얼마나 부합하는지를 평가한다. 단순히 “코드가 작성되었다”로 끝내지 않고, 문제 해결 성공도, 부작용 가능성, 요구사항 일치도, 수정 범위 적절성 등을 종합 점수로 산정한다. 이 점수가 기준 이하이면 Planner 단계로 되돌아가 다시 전략을 세우게 된다.

Evaluator의 평가 요소는 다음과 같다.

- 문제 해결 가능성
- 요구사항 일치도
- 구조 적합성
- 수정 범위의 적절성
- 회귀 가능성
- 검증 통과 여부
- 문서화 가능성

이 구조를 통해 제안 시스템은 1회성 응답이 아니라 반복 개선 가능한 자기검증 루프를 가진다.

3.3 비교 대상 방법 정의

본 연구에서는 다음 두 가지 방식 간의 비교를 수행한다.

3.3.1 직접 지시 방식

초기 실험에서는 단일 AI(Claude Opus 4.6)를 사용하여, 컴파일 과정에서 발생한 오류나 동작하지 않는 기능에 대해 직접적으로 수정 요청을 수행하였다. 예를 들어, “이 버그를 고쳐달라” 또는 “이 기능이 동작하지 않는데 수정해달라”와 같은 방식으로 지시하였다. 이 방식은 구현 속도는 빠르지만, 다음과 같은 문제를 확인할 수 있었다.

- 버그가 완전히 해결되지 않거나 반복적으로 발생함
- 특정 부분을 수정하면 다른 영역에서 새로운 버그가 발생함
- 구조를 고려하지 않은 국소적 수정이 이루어짐

3.3.2 구조 선이해 기반 다중 에이전트 방식

개선된 실험에서는 동일한 AI 모델을 활용하되, 역할을 분리한 다중 에이전트 구조를 적용하였다. 특히, 문제를 바로 수정하도록 지시하는 것이 아니라, 먼저 AI에게 시스템 구조와 관련 흐름을 분석하도록 요청하였다.

이후 다음과 같은 단계로 작업을 수행하였다.

1. 시스템 구조 및 관련 모듈 분석
2. 문제 발생 원인에 대한 구조적 해석
3. 수정 설계 수립
4. 코드 수정 및 구현
5. QA 검증
6. 평가 및 재수행 판단

이 과정에서 AI는 단순한 코드 생성기가 아닌, 구조 기반 문제 해결 도구로 활용되었다.또한 역할 분리와 협업 기반 처리 구조는 AutoGen 계열 다중 에이전트 프레임워크와도 연결된다[12].

4. 실험 및 성능 평가

4.1 실험 대상 시스템 및 환경

본 연구는 생성형 AI 기반 바이브코딩의 효과를 검증하기 위해, 실제 개발 환경과 유사한 조건의 DirectX 기반 MMORPG 레거시 프로젝트를 대상으로 실험을 수행하였다. 해당 시스템은 약 5년 이상 개발된 2D MMORPG로, 클라이언트와 서버 구조를 포함하며 UI, 네트워크, 데이터 처리 등 다양한 모듈이 결합된 형태를 가진다.

코드베이스는 수십만 라인 이상의 C++로 구성되어 있으며, 모듈 간 의존성이 높고 일부 로직은 명시적 문서 없이 암묵적 규칙에 의존하는 특징을 가진다. 특히 스프라이트 기반 그래픽 시스템을 사용함에 따라 좌표, 프레임, 입력 영역 등이 코드와 외부 데이터에 분산되어 있어 구조적 복잡도가 높다.

개발 환경은 버전 관리 기반의 협업 구조로 운영되며, 코드 변경 시 최소 수정 범위와 구조적 정합성이 중요한 요소로 작용한다.

실험에는 Claude Opus 4.6 모델을 사용하였으며[13], 단일 에이전트 기반 직접 지시 방식과 다중 에이전트 기반 구조 선이해 방식을 비교하였다. 모든 실험은 동일한 코드베이스와 문제 집합을 대상으로 수행되었으며, 빌드 성공 여부와 기능 정상 동작, 추가 버그 발생 여부를 기준으로 결과를 평가하였다.

4.2 실험 조건 및 반복 방식

본 연구에서는 직접 지시 방식과 구조 선이해 기반 다중 에이전트 방식의 성능 차이를 비교하기 위해 실제 AI 응답 수집 실험을 수행하였다. 실험은 세 가지 시나리오를 대상으로 하였으며, 각 시나리오에 대해 직접 지시 방식 20회, 구조 선이해 방식 20회를 각각 수행하였다.(3개 시나리오×2개 방식×20회=총 120회)

모든 실험은 동일한 코드 범위, 오류 설명, 요구사항을 기반으로 독립 세션에서 수행하였다. 직접 지시 방식은 문제 증상과 요구사항을 제시한 뒤 즉시 수정을 요청하였고, 구조 선이해 방식은 수정 전에 관련 구조, 호출 체인, 수정 위치, 영향 범위를 먼저 분석하도록 하였다.

프롬프트 구성은 반복 횟수와 별도로 관리하였으며, 두 방식의 지시 형식을 구분하여 작성하였다. 본문에 제시한 코드는 실제 프로젝트 구조를 기반으로 단순화한 의사 코드이나, 실험 결과 수치는 동일한 조건에서 실제 AI 응답을 반복 수집하여 평가한 결과이다.

4.3 실험 시나리오

본 연구에서는 세 가지 실험 시나리오를 구성하였다.

본 시나리오의 코드는 실제 프로젝트의 스킬 처리 흐름을 기반으로 연구 목적에 맞게 단순화한 의사 코드이다.

시나리오 A는 스프라이트 기반 스킬 모션 속도 수정 문제로, 특정 스킬 이펙트의 프레임 유지 시간을 조정하는 과제이다. 시나리오 B는 UI anchor/pivot 보정 문제로, 렌더링 좌표와 클릭 판정 좌표가 어긋나는 상황을 대상으로 한다. 시나리오 C는 특정 스킬 또는 조건에만 변경이 적용되어야 하는 선택적 기능 변경 문제로 설정하였다.

이를 통해 AI가 자연어 요구를 단순히 표면적으로 해석하는지, 또는 호출 체인과 수정 위치, 핵심 변수, 영향 범위를 구조적으로 파악하여 안정적인 수정안을 생성할 수 있는지를 비교하였다.

(1) 시나리오 A: 스킬 모션 속도 수정

실험 대상은 스프라이트 프레임 기반 애니메이션 구조를 사용하는 캐릭터 스킬 이펙트 처리 시스템이다. 해당 구조에서 스킬 모션은 캐릭터 이동 속도 변수가 아니라, 스킬 발동 이후 생성되는 이펙트의 프레임 유지 시간에 의해 제어된다.

주요 호출 흐름은 다음과 같다.

```
CPlayer_Intp::AttackAction()
 {
    int nSkillNum = 0;
    AttackActionControl();
    WizardAttackAction(&nSkillNum);

    AttackSendMessage(nSkillNum);
    SkillAttackEffect(nSkillNum);
 }
```

SkillAttackEffect()에서는 스킬 번호에 따라 개별 이펙트 함수가 호출된다.

```
void CPlayer_Intp::SkillAttackEffect(int nSkillNum)
 {
    switch (nSkillNum)
    {
     case 42:
       cNewEffect.WizardStaffSwing(
         m_nWorldX, m_nWorldY,
        nSkillNum, nSkillLevel, weaponType
      );
      break;
    }
 }
```

이펙트 함수 내부에서는 빈 슬롯을 할당하고, 해당 이펙트의 계산 함수와 유지 시간을 설정한다.

```
int i = GetNewEffect(_nWorldX, _nWorldY);
if (i <= -1)
    return;

pST_Effect[i]->m_nEffectCalNum = 42;
pST_Effect[i]->m_usSkillNum = _SkillNum;
```

스프라이트 프레임 정보는 SetImageData()를 통해 설정된다. 여기서 aniFrm은 한 프레임이 몇 tick 동안 유지되는지를 의미하므로, 스킬 모션 속도를 조정할 때 핵심 수정 대상이 된다.

```
// 기존:총 6프레임, 한 프레임당 4 tick 유지
pST_Effect[i]->m_nEffectDestroyTime = 25;

pST_Effect[i]->SetImageData(
    E_DPT_CHARACTER_POS,
    6, 4, 3,
    0, 0, 156, 162
);
```

실험 요구사항은 다음과 같이 설정하였다. 특정 스킬 이펙트의 스프라이트 재생 속도를 약 2배 빠르게 조정한다. 단, 캐릭터 이동 속도, 공격 속도, 다른 스킬 이펙트, 전역 애니메이션 처리에는 영향을 주지 않아야 한다. 올바른 수정 방향은 특정 스킬 이펙트 함수 내부에서 aniFrm 값만 조정하는 것이다.

```
// 수정 후: 한 프레임당 2 tick 유지
pST_Effect[i]->m_nEffectDestroyTime = 13;

pST_Effect[i]->SetImageData(
    E_DPT_CHARACTER_POS,
    6, 2, 3,
    0, 0, 156, 162
);
```

반면 다음과 같은 수정은 실패 사례로 정의하였다.

```
m_fMoveSpeed *= 2.0f;
// 실패: 캐릭터 이동 속도 변경
g_nGlobalAnimationDelay /= 2;
// 실패: 전역 애니메이션 속도 변경
g_fEffectTimeScale = 2.0f;
// 실패: 모든 이펙트 재생 속도에 영향
```

또한 스킬 구조에서는 SetClashData()를 통해 충돌 판정 시간창이 설정되며, 특정 tick 구간에서만 타격 판정이 적용된다. 따라서 모션 속도 변경 시 타격 판정 구간과 시각적 모션이 과도하게 어긋나지 않는지도 검토 항목에 포함하였다.

(2) 시나리오 B: UI anchor/pivot 보정

시나리오 B는 스프라이트 기반 UI에서 렌더링 좌표와 클릭 판정 좌표가 어긋나는 상황을 대상으로 하였다. 직접 지시 방식은 화면에 보이는 위치만 기준으로 좌표값 자체를 변경할 가능성이 있으며, 이 경우 다른 UI 정렬이나 클릭 영역에 부작용이 발생할 수 있다. 반면 구조 선이해 방식은 렌더링 좌표, anchor/pivot, 클릭 판정 영역의 관계를 먼저 분석한 뒤 수정하도록 설계하였다.

(3) 시나리오 C: 선택적 기능 변경

시나리오 C는 특정 스킬 또는 특정 조건에만 변경이 적용되어야 하는 상황을 대상으로 하였다. 예를 들어 특정 스킬의 후딜레이만 조정해야 하는데, AI가 이를 전체 공격 속도, 쿨타임, 전역 지연값 변경으로 확장하는지를 확인하였다. 이 시나리오는 AI가 요구사항의 적용 범위를 얼마나 정확하게 제한하는지를 평가하기 위해 사용하였다.

4.4 평가 지표 및 산정 기준

실험 결과의 객관적 비교를 위해 여섯 가지 지표를 사용하였다. 다만 수식은 핵심 지표인 1차 해결률, 부작용률, 구조 해석 정확도에 대해서만 정의하였다.

(1) 1차 해결 성공률

1차 해결 성공률은 AI가 최초 응답에서 요구사항을 충족한 비율을 의미한다.

$$FPSR = \frac{Nfirst}{Ntotal} \times 100 \quad (1)$$

여기서 Nfirst는 최초 시도에서 성공한 실험 수, Ntotal은 전체 반복 실험 수이다.

(2) 최종 해결률

최종 해결률은 반복 수정 요청을 포함하여 최종적으로 요구사항을 충족한 비율이다. 최초 응답이 실패하더라도 추가 지시 후 올바른 수정에 도달한 경우 최종 성공으로 판단하였다.

(3) 부작용 발생률

부작용 발생률은 AI가 제안한 수정 이후 다른 기능, 모션, 전역 로직에 의도하지 않은 오류가 발생한 비율이다.

$$SER = \frac{Nside}{Ntotal} \times 100 \quad (2)$$

여기서 Nside는 부작용이 발생한 실험 수, Ntotal은 전체 반복 실험 수이다.

(4) 평균 반복 횟수

평균 반복 횟수는 요구사항을 충족하기까지 필요한 평균 수정 요청 횟수이다. 이 지표는 개발자의 재요청 부담과 작업 효율성을 평가하기 위해 사용하였다.

(5) 요구사항 일치도

요구사항 일치도는 5점 척도로 평가하였으며 점수별 기준은 〈Table 1〉과 같다.

〈Table 1〉 Evaluation Criteria for Requirement Alignment

Score	Criteria
1	The modification is unrelated to the requirement.
2	The modification is partially related, but the main target is incorrect.
3	The requirement is partially satisfied, but side effects may occur.
4	The requirement is mostly satisfied, with minor issues remaining.
5	The requirement is fully satisfied without unintended side effects.

평가는 C++ 게임 클라이언트 개발 경험자, 게임 로직 개발 경험자, UI/스프라이트 처리 경험자로 구성된 3인이 독립적으로 수행하고, 평균값을 사용하였다. 평가자 간 판단이 크게 다른 경우에는 다수결 또는 협의를 통해

최종 판정을 확정하였다.

(6) 구조 해석 정확도

구조 해석 정확도는 AI가 문제 해결에 필요한 구조적 요소를 얼마나 정확히 파악했는지를 나타낸다.

$$SUA = \frac{\text{충족 항목 수}}{\text{전체 평가 항목 수}} \times 100 \quad (3)$$

시나리오 A의 구조 해석 정확도는 다음 5개 항목을 기준으로 산정하였다. 예를 들어 5개 항목 중 4개를 충족하면 구조 해석 정확도는 80%로 계산하였다. 시나리오 B와 C도 동일한 방식으로 호출 흐름, 수정 위치, 핵심 변수, 잘못된 수정 계층 배제, 부작용 검토의 5개 기준을 적용하였다.

4.5 실험 결과

본 연구는 직접 지시 방식과 구조 선이해 기반 다중 에이전트 방식의 차이를 비교하기 위해 세 가지 시나리오를 대상으로 실험을 수행하였다. 각 시나리오는 방식별 20회씩 반복하였으며, 총 120회의 결과를 수집하였다. 시나리오별 실험 결과는 〈Table 2〉와 같다.

〈Table 2〉 Experimental Results by Scenario

Scenario	Method	1st Success	Final Success	Error Rate	No. of Iterations	Req. Understanding	Structural Accuracy
A. Skill Motion Correction	Direct Instruction	35%	65%	40%	3.6 times	2.7/5	30%
	Structured Understanding	85%	100%	10%	1.5 times	4.8 / 5	90%
B. UI Anchor /Pivot Adjustment	Direct Instruction	40%	70%	35%	3.1 times	3.0 / 5	35%
	Structured Understanding	90%	100%	5%	1.4 times	4.7 / 5	92%
C. Selection Function Modification	Direct Instruction	30%	60%	45%	3.8 times	2.5 / 5	28%
	Structured Understanding	80%	95%	10%	1.7 times	4.6 / 5	88%

아래 〈Table 3〉 은 세 가지 시나리오의 평균값을 비교한 결과이다.

〈Table 3〉 Overall Average Comparison

Evaluation Metric	Direct Instruction	Structured Understanding
1st Success	35.0%	85.0%
Final Success	65.0%	98.3%
Error Rate	40.0%	8.3%
No. of Iterations	3.5 times	1.5 times
Req. Understanding	2.7 / 5	4.7 / 5
Structural Accuracy	31.0%	90.0%

실험 결과, 구조 선이해 방식은 모든 시나리오에서 직접 지시 방식보다 높은 해결률과 요구사항 일치도를 보였다. 특히 전체 평균 기준 1차 해결률은 35.0%에서 85.0%로 증가하였고, 부작용률은 40.0%에서 8.3%로 감소하였다. 평균 반복 횟수 역시 3.5회에서 1.5회로 줄어, 구조 분석을 선행하는 방식이 전체 문제 해결 비용을 낮추는 데 효과적임을 확인하였다.

또한 요구사항 일치도는 직접 지시 방식 평균 2.7점에서 구조 선이해 방식 평균 4.7점으로 향상되었으며, 구조 해석 정확도 역시 31.0%에서 90.0%로 크게 증가하였다. 이는 구조와 호출 체인을 먼저 분석한 후 수정 작업을 수행하는 방식이 단순 직접 지시 방식보다 안정적이고 일관된 결과를 제공함을 보여준다.

4.6 결과 분석

실험 결과, 직접 지시 방식은 자연어 요구를 표면적으로 해석하여 잘못된 수정 계층을 선택하는 경향이 나타났다. 특히 시나리오 A에서 “스킬 모션을 빠르게 하라”는 요구를 특정 스킬 이펙트의 프레임 유지 시간 조정으로 해석하지 못하고, 캐릭터 이동 속도, 공격 속도, 전역 애니메이션 속도 변경으로 확장하는 사례가 확인되었다.

반면 구조 선이해 기반 방식은 AttackAction()에서 SkillAttackEffect()를 거쳐 특정 cNewEffect 함수로 이어지는 호출 체인을 먼저 분석하였다. 이후 SetImageData()의 aniFrm 값이 스프라이트 프레임 유지 시간을 결정한다는 점을 식별하고, 캐릭터 속도나 전역 애니메이션 로직이 아닌 특정 스킬 이펙트 내부의 aniFrm 값만 조정하였다.

이러한 차이는 정량 결과에서도 확인된다. 직접 지시 방식은 평균 1차 해결률 35.0%, 부작용률 40.0%, 평균 반복 횟수 3.5회를 보인 반면, 구조 선이해 방식은 평균

1차 해결률 85.0%, 부작용률 8.3%, 평균 반복 횟수 1.5회를 기록하였다. 즉, 구조 분석을 선행하면 초기 분석 절차는 추가되지만, 잘못된 수정 방향과 반복 요청을 줄여 전체 문제 해결 효율을 높일 수 있다.

시나리오 B와 C에서도 동일한 경향이 나타났다. UI anchor/pivot 보정에서는 구조 선이해 방식이 렌더링 좌표와 클릭 판정 영역을 구분하여 부작용을 줄였고, 선택적 기능 변경에서는 수정 대상 변수와 적용 조건을 분리하여 특정 기능에만 변경을 적용하였다.

결과적으로 구조 선이해 기반 접근은 문제를 단순한 “속도 조정”이나 “좌표 수정”으로 처리하지 않고, 호출 체인, 수정 위치, 핵심 변수, 영향 범위를 기준으로 재정의하였다. 이는 생성형 AI 기반 바이브코딩에서 구조 분석이 버그 해결 정확도와 안정성을 높이는 핵심 요소임을 보여주며, 실제 소프트웨어 이슈 해결과 레거시 시스템 유지보수에서 구조적 맥락 이해가 중요하다는 기존 연구 흐름과도 일치한다[4][5][6][14][15].

5. 결론

본 연구는 대규모 DirectX 기반 MMORPG 레거시 프로젝트에서 생성형 AI를 활용한 바이브코딩의 한계를 분석하고, 이를 개선하기 위한 구조 선이해 기반 다중 에이전트 접근 방식을 제안하였다. 기존의 직접 지시 방식은 빠른 코드 생성에는 유용하지만, 시스템의 호출 체인과 데이터 구조를 충분히 반영하지 못할 경우 국소적 수정에 머무르거나 다른 기능에 부작용을 발생시킬 수 있음을 확인하였다.

제안 방법은 Claude Opus 4.6을 기반으로 Planner, Code Generator, QA, Evaluator의 역할을 분리하고, 구조 분석, 수정 전략 수립, 코드 생성, 검증 및 재평가를 단계적으로 수행하도록 구성하였다. 성능 평가 결과, 구조 선이해 기반 다중 에이전트 방식은 단일 에이전트 직접 지시 방식 대비 1차 해결 성공률을 평균 35.0%에서 85.0%로 향상시켰으며, 최종 해결률 역시 65.0%에서 98.3%로 증가하였다. 반면 부작용 발생률은 40.0%에서 8.3%로 감소하였고, 평균 반복 수정 횟수는 3.5회에서 1.5회로 줄어 제안 방식이 문제 해결 정확도와 안정성 측면에서 우수함을 확인하였다.

또한 요구사항 일치도는 2.7점에서 4.7점으로 향상되었으며, 구조 해석 정확도는 31.0%에서 90.0%로 증가하였다. 이는 제안 방식이 단순히 코드를 생성하는 데 그치지 않고, 호출 체인과 데이터 흐름을 먼저 분석함으로써 수정 대상과 영향 범위를 보다 정확히 식별할 수 있음을 보여준다. 특히 스프라이트 기반 스킬 모션 수정 사례에서는 AttackAction()에서 SkillAttackEffect()를 거쳐 특정 cNewEffect 함수로 이어지는 호출 체인을 분석한 뒤, SetImageData()의 aniFrm 값을 선택적으로 수정함으로써 캐릭터 이동 속도나 전역 애니메이션 속도에 영향을 주지 않는 안정적인 결과를 도출하였다.

이러한 결과는 생성형 AI의 문제 해결 성능이 모델 자체의 성능뿐 아니라, AI에게 문제를 어떤 절차로 이해시키고 지시하는가에 크게 의존함을 보여준다. 복잡한 레거시 시스템에서는 결과를 즉시 요구하는 방식보다, 구조와 호출 흐름을 먼저 파악하고 이를 기반으로 단계적으로 수정하는 방식이 더 효과적이다. 따라서 생성형 AI는 단순한 코드 생성 도구가 아니라, 구조 분석과 검증 과정을 포함한 협업형 개발 도구로 활용될 필요가 있다.

본 연구는 대규모 레거시 게임 프로젝트에서 생성형 AI 기반 바이브코딩의 한계를 정량적으로 확인하고, 이를 개선하기 위한 구조 선이해 기반 다중 에이전트 접근의 가능성을 제시했다는 점에서 의의를 가진다. 향후 연구에서는 다양한 게임 시스템, AI 모델, 버그 유형을 대상으로 확장 실험을 수행하고, 실제 개발 환경에서의 생산성 향상 효과와 장기적인 유지보수 안정성을 추가적으로 검증할 필요가 있다.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention Is All You Need,” Advances in Neural Information Processing Systems, vol. 30, pp. 5998–6008, 2017.

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language Models are Few-Shot Learners,” Advances in Neural Information Processing Systems, vol. 33, pp. 1877–1901, 2020.

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., “Evaluating Large Language Models Trained on Code,” arXiv preprint arXiv:2107.03374, 2021.

[4] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?," in Proceedings of the International Conference on Learning Representations (ICLR), 2024.

[5] K. H. Bennett and V. T. Rajlich, "Software Maintenance and Evolution: A Roadmap," in Proceedings of the Conference on the Future of Software Engineering, pp. 73-87, 2000.

[6] E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7, no. 1, pp. 13-17, 1990.

[7] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, Real-Time Rendering, 4th ed. Boca Raton, FL, USA: CRC Press, 2018.

[8] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, Computer Graphics: Principles and Practice, 2nd ed. Reading, MA, USA: Addison-Wesley, 1996.

[9] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," in Proceedings of the International Conference on Learning Representations (ICLR), 2023.

[10] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. R. Narasimhan, and S. Yao, "Reflexion: Language Agents with Verbal Reinforcement Learning," Advances in Neural Information Processing Systems, vol. 36, 2023.

[11] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, "AgentCoder: Multi-Agent Code Generation with Effective Testing and Self-optimisation," arXiv preprint arXiv:2312.13010, 2023.

[12] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," in Proceedings of the Conference on Language Modeling (COLM), 2024.

[13] Anthropic, "Introducing Claude Opus 4.6," Anthropic News, Feb. 5, 2026.

[14] C. S. Xia and L. Zhang, "Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT," in Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 819-831, 2024.

[15] J. Yang, C. Jin, R. Tang, X. Han, M. Feng, Z. Jiang, B. Yin, and X. Hu, "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," arXiv preprint arXiv:2405.15793, 2024.

류 지 수(Ji Soo Ryu) [정회원]

- 2024년 3월 ~ 현재 : 경북대학교 컴퓨터학부 대학원 박사수료
- 2023년 3월 ~ 현재 : 대구가톨릭대학교 컴퓨터학부 겸임교수
- 2018년 3월 ~ 현재 : ㈜드림아이디어소프트 이사

〈관심분야〉
인공지능, LLM, Game Engine, 자연어처리, 정보통신

정 순 기(Soon Ki Jung) [정회원]

- 1997년 2월 : KAIST 전산학과 박사
- 1998년 ~ 현재 : 경북대학교 컴퓨터학부 교수

〈관심분야〉
3DComputerVision, ComputerGraphics, Visualization, Human-Computer Interaction (HCI), Intelligent Vision Systems, VR/AR Systems