피지컬 AI 시대의 AI 기반 코드 인텔리전스 연구 동향

Research Trends in AI-Based Code Intelligence in the Era of Physical AI

전형국 (H.K. Jun, hkjun@etri.re.kr) 온디바이스AI모델연구실 책임연구원
이기호 (K.H. Lee, kiho@etri.re.kr) 온디바이스AI모델연구실 연구원
김상철 (S.C. Kim, sheart@etri.re.kr) 온디바이스AI모델연구실 책임연구원
우예린 (E.Y. Woo, yrwoo@etri.re.kr) 온디바이스AI모델연구실 연구원
한호재 (H.J. Han, hojae.han@etri.re.kr) 온디바이스AI모델연구실 연구원
마유승 (Y.S. Ma, ysma@etri.re.kr) 온디바이스AI모델연구실 책임연구원

ABSTRACT

Physical artificial intelligence (AI) is expanding the role of AI from digital information processing to systems that perceive, reason, and act in physical environments, thereby increasing the complexity of software development and validation. In parallel, code intelligence, which is the capability of AI systems to understand, generate, transform, and verify source code, is reshaping modern software development. The field has evolved from code completion to agentic coding, in which AI agents plan, implement, execute, and validate software based on natural language requirements. This paper reviews recent advances in AI-based code intelligence technologies in the era of physical AI. It covers recent developments in code datasets and models, code generation pipelines, and the evolving ecosystem of agentic coding tools. It also reviews evaluation benchmarks and discusses key challenges in code quality, security, and developer productivity. Finally, it introduces the Pipeline for Unified LLM Software Engineering (PULSE), a modular framework developed at ETRI that supports requirements analysis, code generation, testing, and execution-based verification. These advances suggest that code intelligence can serve as an important software-engineering foundation for increasingly complex AI systems in the era of physical AI.

KEYWORDS Agentic Coding, Code Generation, Code Intelligence, Language Models, Physical AI

* DOI: https://doi.org/10.22648/ETRI.2026.J.410601
* This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2022-0-00995, Automated reliable source code generation from natural language descriptions).

본 저작물은 공공누리 제4유형
출처표시+상업적이용금지+변경금지 조건에 따라 이용할 수 있습니다.
©2026 한국전자통신연구원

I. 서론

최근 인공지능은 디지털 공간에서 정보를 생성·분석하는 수준을 넘어, 센서와 디바이스를 통해 물리 환경을 인지하고 판단하며 동작하는 피지컬 AI(Physical AI)로 적용 범위가 확대되고 있다. 이러한 변화는 AI 모델 자체의 성능뿐만 아니라 모델, 소프트웨어, 실행 환경, 다양한 도구를 통합하여 복잡한 시스템을 개발하고 검증하는 소프트웨어 엔지니어링 역량의 중요성을 함께 높이고 있다. 특히 피지컬 AI 시스템은 실제 실행 환경과의 연계와 반복적인 검증이 중요하므로, 요구사항을 이해하고 코드를 생성한 뒤 실행 결과를 바탕으로 수정과 검증을 반복할 수 있는 AI 기반 개발 자동화 기술이 향후 핵심 기반 기술 중 하나가 될 수 있다.

소프트웨어가 산업 전반의 핵심 경쟁력으로 자리잡으면서 다양한 분야에서 소프트웨어 개발 수요가 지속적으로 증가하고 있다. 그러나 전문 개발 인력의 공급과 개발 생산성은 이러한 수요 증가를 충분히 뒷받침하지 못하고 있으며, 개발 효율성을 높이기 위한 자동화 기술의 중요성도 더욱 커지고 있다. 이에 따라 프로그램 합성(Program Synthesis), 모델 기반 개발(Model-Driven Engineering) 등 소프트웨어 개발 과정을 자동화하기 위한 다양한 연구가 이루어져 왔다. 그러나 이러한 기술은 개발 전에 형식 명세(Formal Specification)나 도메인 모델을 미리 작성해야 하거나 적용할 수 있는 범위가 제한적이어서 다양한 소프트웨어 개발에 활용하는 데는 한계가 있었다.

이러한 한계는 대규모 언어모델(LLM: Large Language Model)의 등장으로 크게 변화하기 시작하였다. 2021년 공개된 Codex와 이를 기반으로 한 GitHub Copilot은 자연어와 소스 코드를 함께 학습하여 실용적인 수준의 코드 생성이 가능함을 보여주었다[1]. 이후 GPT, Claude, Gemini와 같은 범용 초거대 언어모델의 코드 생성 성능이 빠르게 향상되었으며, Code Llama, DeepSeek-Coder, Qwen-Coder 등 코드 생성에 특화된 공개 모델도 잇달아 등장하였다[2-4]. 또한, AI 코드 어시스턴트가 개발 생산성과 소프트웨어 품질을 향상시킨다는 연구 결과가 보고되면서[5], 국내외 주요 기업들도 이를 실제 개발 환경에 적극 도입하고 있다.

본고에서는 코드의 이해, 생성, 변환, 검증을 지원하는 AI 기술을 코드 인텔리전스(Code Intelligence)로 정의하고, 관련 기술의 국내외 연구 및 산업 동향을

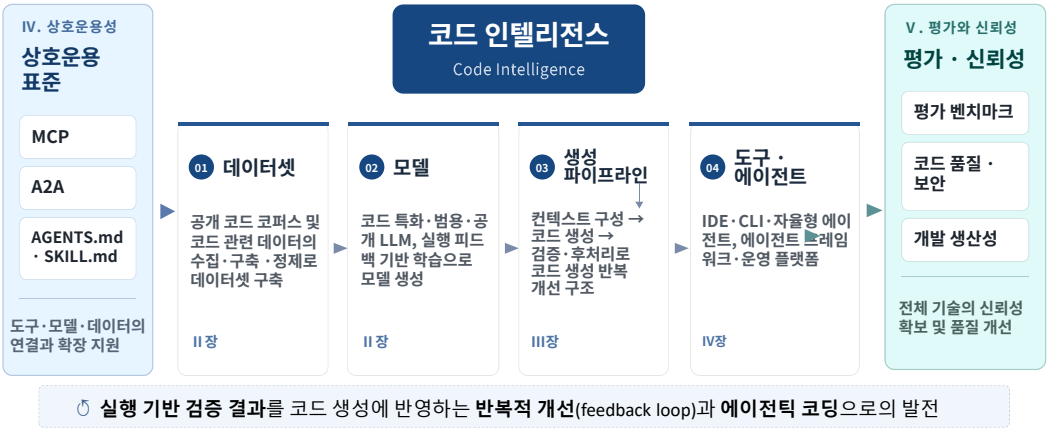

그림 1 코드 인텔리전스 기술 구성

살펴본다(그림 1). 특히 코드 자동완성을 지원하는 수준을 넘어, 자연어 요구사항을 이해하고 계획 수립부터 코드 생성, 실행, 검증까지 수행하는 에이전틱 코딩(Agentic Coding)으로의 발전에 주목한다. 이러한 발전은 소프트웨어가 실제 실행 환경과 지속적으로 상호작용을 해야 하는 피지컬 AI 시대에 더욱 중요해질 수 있다. 본고는 피지컬 AI 자체의 모델 · 로봇 · 제어 기술을 직접 다루기보다는 복잡한 AI 시스템 개발을 지원하는 소프트웨어 엔지니어링 기반 기술이라는 관점에서 코드 인텔리전스의 발전을 조망한다. 이를 위해 코드 데이터셋과 모델, 코드 생성 파이프라인, 개발 도구 생태계, 평가와 신뢰성, 그리고 국내 기술 동향을 차례로 살펴보고, 앞으로의 발전 방향을 함께 논의한다.

II. 코드 생성 기반 기술 동향

최근 코드 생성 기술은 단순히 대규모 언어모델의 성능 향상만으로 발전한 것이 아니라, 대규모 코드 데이터셋, 코드 특화 모델, 생성 파이프라인, 그리고 실행 기반 학습 기술이 함께 발전하면서 빠르게 고도화되고 있다. 특히 학습 데이터는 공개 코드 중심에서 실행 가능한 개발 환경과 에이전트 수행 과정까지 포함하는 형태로 확장되고 있으며, 모델 또한 단순한 코드 생성기를 넘어 저장소를 이해하고 도구를 활용하는 에이전틱(Agentic) 모델로 발전하고 있다. 이 장에서는 이러한 코드 생성 기술의 기반을 이루는 코드 데이터셋과 코드 기반 모델의 발전 동향을 중심으로 살펴본다.

1. 코드 LLM 학습 데이터

1.1 사전학습 데이터

코드 LLM(Code Large Language Model)의 성능은 사전학습 데이터의 규모와 품질에 크게 좌우된다. 2019년 공개된 CodeSearchNet은 약 200만 건의 함수-주석 쌍을 제공하여 코드 검색과 코드 이해 연구의 기반을 마련하였다[38]. 이후 BigCode 프로젝트의 The Stack(2022)은 허가적(Permissive) 라이선스를 가진 오픈소스 코드만을 수집한 약 3TB 규모의 데이터셋을 공개하였으며, StarCoder 계열 모델의 사전학습 데이터로 활용되었다[6]. 후속 버전인 The Stack v2(2024)는 600여 개 프로그래밍 언어와 약 67TB 규모로 확장되었으며, 라이선스 관리와 데이터 제외 요청 절차를 도입하여 공개 데이터셋의 신뢰성과 활용성을 높였다[7].

최근에는 코드 데이터셋 경쟁의 중심이 규모(Scale)에서 품질(Quality)로 빠르게 이동하고 있다. 첫째, Stack-Edu(2025)[8]와 RefineCode(2024)[9]는 오류와 중복을 줄이는 데이터 정제(Data Curation) 기술을 적용하여 학습 데이터의 품질을 높였으며, SwallowCode(2025)는 저품질 코드를 LLM으로 다시 작성해 활용하는 데이터 재가공(Data Refinement) 방식을 제안하였다[10]. 둘째, 학습 대상이 개별 파일(File)에서 저장소(Repository) 전체로 확대되면서 파일 간 의존성, 프로젝트 구조, 모듈 관계를 함께 학습하는 방식이 보편화되고 있다[3]. 셋째, 커밋(Commit), 이슈(Issue), 풀 리퀘스트(Pull Request), 코드 리뷰(Code Review) 등 개발 과정에서 생성되는 정보까지 함께 활용하면서, 코드뿐만 아니라 실제 소프트웨어 개발 과정을 학습하는 방향으로 발전하고 있다.

1.2 사후학습 데이터와 합성 데이터

사전학습된 모델이 개발자의 의도에 맞는 코드를 생성하기 위해서는 지시 미세조정(Instruction Tuning)이 필요하다. 그러나 사람이 직접 대규모 지시-코드 데이터를 구축하는 데는 큰 비용과 시간이 소요된다. 이에 따라 최근에는 LLM을 활용하여 학습 데이

터를 자동으로 생성하는 합성 데이터(Synthetic Data)가 사후학습(Post-Training)의 핵심 데이터로 자리 잡고 있다.

대표적으로 WizardCoder의 Evol-Instruct는 단순한 지시문을 점차 복잡한 문제로 확장하는 방법을 제안하였으며[11], Magicoder의 OSS-Instruct는 실제 오픈소스 코드로부터 자연어 지시문을 자동 생성하는 방식을 도입하였다[12]. 최근에는 생성된 코드가 실제로 실행되어 테스트를 통과한 경우에만 학습 데이터로 활용하는 실행 검증(Execution Verification) 기반 데이터셋이 새로운 품질 기준으로 주목받고 있다. 대표적인 예인 OpenCodeInstruct(2025)는 약 500만 건의 지시-코드 쌍과 함께 실행 결과와 테스트 정보를 제공하여 코드 생성 모델의 신뢰성을 높이고 있다[13].

최근에는 학습 대상도 코드 자체를 넘어 실제 소프트웨어 개발 과정으로 확대되고 있다. SWE-Gym(2024)은 실제 저장소와 실행 가능한 테스트 환경을 포함한 개발 환경을 제공하였으며[14], SWE-smith(2025)는 다양한 저장소에서 개발 문제와 해결 과정을 자동으로 생성하였다[15]. 또한, SWE-RL(2025)은 GitHub의 실제 Pull Request 해결 과정을 강화학습(Reinforcement Learning)에 활용하여 에이전트의 문제 해결 능력을 향상시키고자 하였다[16].

이처럼 코드 데이터셋은 단순히 대규모 소스 코드를 수집하는 단계를 넘어, 실행 가능한 개발 환경과 실제 문제 해결 과정을 함께 학습하는 방향으로 발전하고 있다. 앞으로는 코드뿐만 아니라 개발자의 의사결정과 도구 활용 과정까지 함께 학습하는 에이전트 중심 데이터셋(Agent-centric Dataset)이 코드 인텔리전스의 핵심 데이터로 자리 잡을 것으로 전망된다(표 1).

표 1 코드 데이터셋의 발전 방향

발전 방향	대표 데이터셋	핵심 특징
공개 코드 코퍼스	CodeSearchNet, The Stack, The Stack v2	공개 코드 기반 사전학습 데이터 구축
데이터 품질 향상	Stack-Edu, RefineCode, SwallowCode	품질 정제 및 LLM 기반 데이터 재가공
지시 미세조정	Magicoder, OpenCodeInstruct	합성 데이터 및 실행 검증 기반 학습
에이전트 학습	SWE-Gym, SWE-RL	실행 가능한 개발 환경과 개발 과정 학습

2. 코드 파운데이션 모델

코드 LLM은 대규모 공개 저장소의 소스 코드를 학습한 언어모델로, 코드 이해, 생성, 수정, 테스트, 검증 등 다양한 소프트웨어 개발 작업을 지원한다. 소스 코드는 자연어와 달리 엄격한 문법 구조와 심볼(Symbol) 간의 의존성, 그리고 실행 가능성을 가지므로 일반 언어모델과는 다른 학습 방법이 필요하다. 사전학습 단계에서는 코드 앞뒤 문맥을 활용해 중간 코드를 생성하는 FIM(Fill-in-the-Middle)과 저장소 단위의 코드를 함께 학습하는 저장소 수준 학습(Repository-level Learning)이 널리 활용되었다. 이후 모델을 실제 개발 작업에 맞게 학습시키는 사후학습 단계에서는 지시 미세조정과 실행 결과를 활용한 실행 피드백 기반 학습(Execution Feedback Learning)이 성능 향상의 핵심 요소로 자리 잡고 있다.

최근에는 모델 간 성능 차이가 사전학습보다 사후학습 단계에서 더욱 크게 나타나고 있다. 첫째, 실행 결과를 보상으로 활용하는 실행 피드백 기반 강화학습(RLEF: Reinforcement Learning from Execution Feedback)은 테스트를 통과한 코드를 더 많이 생성하도록 학습함으로써 성능을 향상시키고 있다[17]. 둘째, 단계적으로 문제를 해결하는 추론학습(Reasoning Training)이 도입되면서 알고리즘 문제 해결과 디버깅

능력이 크게 개선되었다[18]. 셋째, 저장소 탐색, 도구 활용, 테스트와 반복적인 수정 과정을 함께 학습하는 에이전틱 사후학습(Agentic Post-Training)이 확대되면서, 모델도 단순한 코드 생성기를 넘어 실제 개발 작업을 수행하는 AI 에이전트로 발전하고 있다.

주요 코드 LLM의 발전 과정을 표 2에 정리하였다. 2023년까지는 StarCoder와 Code Llama 등 코드 특화 공개 모델이 기술 발전을 주도하였다[2,6]. 이후 GPT, Claude, Gemini와 같은 범용 초거대 언어 모델의 코드 생성 성능이 빠르게 향상되면서 코드 특화 모델과 범용 모델의 경계는 점차 희미해지고 있다[3,4]. 특히 2025년 이후에는 DeepSeek, Qwen, GLM, Kimi 등 공개(Open-weights) 모델이 희소 활성화 구조(MoE: Mixture-of-Experts)와 최대 100만 토큰 수준의 긴 문맥 처리 능력을 바탕으로 빠르게 발전하였다. 이들 모델은 2026년 상반기 기준 SWE-bench와 같은 에이전틱 코딩 벤치마크에서 70~80% 수준의 문제 해결률을 기록하며 상용 모델과의 성능 격차를 빠르게 좁혀가고 있다[18,19]. 이에 따라 공개 모델은 온프레미스(On-Premises) 환경에서 코드 인텔리전스를 구현하기 위한 핵심 기반 기술로 자리 잡고 있으며, 모델 경쟁의 중심도 단순한 코드 생성 능력에서 저장소를 이해하고 다양한 개발 도구를 활용하며 복잡한 개발 문제를 해결하는 능력으로 이동하고 있다.

표 2 주요 코드 LLM의 발전 단계

발전 단계	대표 모델	핵심 특징
코드 특화 모델	Codex, StarCoder, Code Llama	코드 생성을 위한 전용 LLM
코드 이해 강화	DeepSeek-Coder, Qwen-Coder	저장소 수준 학습과 FIM 등 코드 특화 학습
범용 LLM 확장	GPT, Claude, Gemini	코드 생성과 추론, 에이전틱 코딩 지원
공개 모델	DeepSeek, Qwen, GLM, Kimi	MoE 기반 고성능 공개 모델

III. 코드 생성 파이프라인 기술

동일한 코드 LLM을 사용하더라도 입력 정보와 생성 전략, 검증 방식에 따라 생성 결과의 품질은 크게 달라진다. 최근 코드 생성 시스템은 컨텍스트 구성(Context Engineering), 코드 생성(Code Generation), 검증 및 후처리(Verification & Post-Processing)의 세 단계로 이루어진 파이프라인 형태로 발전하고 있으며, 각 단계는 이전 단계의 결과를 바탕으로 반복적으로 개선되는 구조를 갖는다.

1. 컨텍스트 구성

실제 소프트웨어 개발은 기존 저장소(Repository)를 수정하거나 확장하는 경우가 대부분이므로, 모델이 개발 맥락(Context)을 얼마나 정확하게 이해하는지가 생성 성능에 큰 영향을 미친다. 가장 널리 사용되는 방법은 저장소에서 관련 함수와 클래스를 검색하여 프롬프트에 함께 제공하는 검색 증강 생성(RAG: Retrieval-Augmented Generation)이다[20]. 최근에는 코드의 문법 구조를 고려한 검색과 코드 전용 임베딩 모델을 활용하여 필요한 코드를 더욱 정확하게 찾을 수 있게 되었다. 또한, 추상 구문 트리(AST: Abstract Syntax Tree), 함수 호출 관계, 타입 의존성 등을 분석하여 수정 대상과 관련된 코드만 선별하는 코드 그래프(Code Graph) 기반 기법도 활용되고 있다.

최근에는 단순히 코드를 검색하는 것을 넘어, AI 에이전트가 파일 검색, 심볼 탐색 등 다양한 도구를 이용하여 필요한 정보를 스스로 수집하는 에이전틱 검색(Agentic Retrieval)도 확산되고 있다. 또한, 프로젝트 규약 문서(AGENTS.md), 이전 작업 기록, 대화 내

용 등을 함께 활용하여 개발 상황을 보다 정확하게 이해하는 방향으로 발전하고 있다.

2. 코드 생성

코드 생성 단계에서는 한 번 생성한 결과를 그대로 사용하는 것이 아니라 계획을 세우고 여러 번 수정하면서 품질을 높이는 방식이 일반화되고 있다.

첫 번째는 계획 기반 생성(Planning)으로, 요구사항을 분석한 뒤 구현 계획을 먼저 수립하고 이를 바탕으로 코드를 생성하는 방식이며, Self-Planning이 대표적인 연구이다[21]. 최근에는 요구사항과 설계 문서, 작업 계획을 먼저 작성한 후 이를 기준으로 구현하는 사양 주도 개발(Spec-Driven Development)로 발전하고 있으며, AWS Kiro와 GitHub Spec Kit이 대표적인 사례이다.

두 번째는 여러 개의 후보 코드를 생성한 뒤 가장 적합한 결과를 선택하는 다중 후보 생성(Multi-Sampling) 방식이다. AlphaCode는 대규모 후보 코드 생성을 통해 경쟁 프로그래밍에서 높은 성능을 달성하였고[22], CodeT는 생성된 테스트를 이용해 후보 코드를 평가하고 최적의 결과를 선택하였다[23]. 최근에는 여러 AI 에이전트가 서로 다른 해결 방법을 제안하고, 이를 비교 · 검증하여 최종 결과를 선택하는 방식으로 발전하고 있다.

세 번째는 자기 수정(Self-Refinement)으로, 컴파일 오류나 테스트 실패 결과를 다시 모델에 입력하여 코드를 반복적으로 수정하는 방식이며, Self-Debugging[24] 이후 대부분의 상용 도구에서 기본 기능으로 제공되고 있다.

3. 코드 검증과 후처리

최근 가장 빠르게 발전하는 영역은 코드 검증과 후처리이다. 실제 서비스에서는 생성된 코드를 바로 사용하지 않고 여러 단계의 검증을 거쳐 품질과 신뢰성을 확보한다. 먼저 컴파일러를 이용해 문법 오류를 확인하고, 타입 검사기(Type Checker)로 타입의 일관성을 검증하며, 린터(Linter)로 코딩 규칙과 코드 스타일을 점검한다. 이후 자동 생성된 테스트를 실행하여 코드가 올바르게 동작하는지 확인하고, 오류가 발견되면 자동 프로그램 수정(APR: Automated Program Repair)을 통해 반복적으로 개선한다. 또한, 정적 분석, 시크릿 검출, 라이선스 검증 등을 통해 보안과 규정 준수 여부를 확인하고, AI 기반 코드 리뷰를 통해 설계 품질과 요구사항 충족 여부도 함께 검토한다.

이러한 검증은 한 번만 수행되는 것이 아니라 코드 생성과 반복적으로 연결된다. 검증 과정에서 발견된 오류와 테스트 결과는 다시 코드 생성 단계로 전달되어 수정과 재생성을 반복하며, 이러한 피드백 구조가 최근 코드 생성 파이프라인의 핵심 요소로 자리 잡고 있다.

4. SDLC 전반으로의 코드 생성 확장

코드 인텔리전스는 새로운 코드를 생성하는 기술을 넘어 소프트웨어 개발 생명주기(SDLC: Software Development Life Cycle) 전반을 지원하는 기술로 확대되고 있다. 저장소 검색, 코드 요약과 설명 등 코드 이해 기능을 비롯하여 단위 테스트 생성, 오류 분석, 레거시 코드 문서화, 대규모 리팩터링 등 다양한 개발 작업에 활용되고 있다. 또한, AI 코드 리뷰(CodeRabbit, Greptile, GitHub Copilot Review)와 취약한 코드를 자동으로 분석하고 수정하는 보안 분석 기술도 독립적인 서비스로 발전하고 있다.

이처럼 코드 인텔리전스는 코드 생성뿐만 아니라 코드 이해, 검증, 유지보수까지 지원하는 핵심 기술

로 자리 잡고 있으며, 이러한 기술을 바탕으로 다음 장에서 살펴볼 에이전틱 코딩 생태계가 빠르게 확산하고 있다. 피지컬 AI 관점에서 이러한 변화는 코드 인텔리전스의 적용 범위가 저장소 내부의 코드 작성 · 수정에서 실제 실행 환경을 고려하는 개발 자동화로 확장될 가능성을 보여준다. 향후에는 시뮬레이션 결과, 디바이스 상태, 센서 입력, 실행 로그와 같은 환경 정보를 개발 컨텍스트로 활용하고, 생성된 소프트웨어를 실행 · 검증한 결과를 다시 에이전트의 수정 과정에 반영하는 폐루프(Closed-Loop) 개발 방식이 중요해질 것으로 예상된다. 이는 다음 장에서 살펴보는 에이전틱 코딩의 계획 · 도구 활용 · 실행 · 검증 구조와도 밀접하게 연결된다.

IV. 에이전틱 코딩 생태계

최근 에이전틱 코딩은 단순한 코드 생성 도구를 넘어 IDE(Integrated Development Environment), CLI(Command-Line Interface), 에이전트 프레임워크, 클라우드 서비스를 포함하는 하나의 생태계로 발전하고 있다(표 3). 또한, 하나의 AI 에이전트를 IDE, CLI, 원격 실행 환경 등 다양한 개발 환경에서 사용할 수 있는 방식이 확산하고 있다. 이에 따라 개발자는 작업 환경에 맞게 동일한 AI 에이전트를 활용하며 개발을 수행할 수 있게 되었다.

1. 개발 도구: IDE와 CLI

IDE 기반 에이전트는 개발자에게 가장 익숙한 개발 환경을 중심으로 발전하고 있다. GitHub Copilot이 코드 자동완성을 대중화한 이후, Cursor, Windsurf, Zed, JetBrains AI Assistant 등은 저장소 검색, 코드 생성, 테스트, 코드 리뷰 기능을 통합하며 에이전트 중심 IDE로 진화하고 있다. 최근에는 여러 에이전트를 동시에 실행 · 비교하고 로컬과 원격 에이전트를 함께 활용하는 개발 환경으로 발전하는 추세이다.

표 3 에이전틱 코딩 도구의 발전 단계

구분	대표 사례	주요 특징
IDE 기반 에이전트	GitHub Copilot, Cursor, Windsurf, Zed, JetBrains AI Assistant	편집기 기반 코드 생성·수정·테스트·리뷰 지원, 멀티 에이전트 개발 환경으로 발전
CLI 기반 에이전트	Claude Code, Codex CLI, Gemini CLI, OpenCode, Aider, Cline	터미널 기반 자율 작업, 서버·CI/CD 자동화 지원, 벤더 네이티브와 오픈소스 생태계 공존
자율형 에이전트	Devin, Codex(Cloud), Google Jules, Replit Agent	원격 실행 환경에서 비동기적으로 작업 수행, PR 기반 개발 지원
에이전트 구축 프레임워크	LangGraph, CrewAI, Microsoft Agent Framework, OpenAI Agents SDK, Claude Agent SDK, Google ADK	에이전트 워크플로 구성, 상태 관리, 도구 호출, 병렬 실행 지원
에이전트 운영 플랫폼	Cursor Agents Window, Warp, Conductor, Vibe Kanban	다중 에이전트세션, 작업 이력(Diff), 정책·비용·감사 로그 관리

CLI 기반 에이전트는 터미널 환경에서 파일 탐색, 코드 수정, 명령 실행, 테스트, 커밋을 하나의 작업 흐름으로 수행하며, GUI(Graphical User Interface) 없이도 동작하므로 서버, 컨테이너, CI/CD(Continuous Integration/Continuous Deployment) 환경과 쉽게 연동된다. Claude Code, Codex CLI, Gemini CLI와 같은 상용 도구와 OpenCode, Cline, Aider 등 오픈소스 도구가 함께 발전하고 있으며, 특히 오픈소스 도구는 로컬 모델을 구동하거나 폐쇄망 환경에서도 활용할 수 있다. 또한, 최근에는 파일 편집, 명령 실행, Git 작업 등 다양한 개발 도구를 AI 에이전트와 연동하는 기술이 발전하고 있다. 이에 따라 동일한 언어모

델이라도 어떤 도구를 함께 사용하고 어떤 환경에서 실행하는지에 따라 성능 차이가 나타나는 것으로 보고되고 있다[25].

2. 자율형 에이전트와 플랫폼

자율형 에이전트(Autonomous Agent)는 개발자가 작업을 위임하면 원격 실행 환경에서 저장소를 분석하고 파일을 수정한 뒤 테스트와 검증을 수행하고, Pull Request(PR)를 생성하여 개발자에게 수정 결과를 제안한다. Devin이 이러한 방식을 대표적으로 제시하였으며, OpenAI Codex와 Google Jules 등도 장시간 개발 작업을 원격에서 수행하는 비동기 에이전트를 제공하고 있다. 최근에는 로컬 IDE에서 작업을 시작한 뒤 장시간 작업은 원격 에이전트에 맡기고, 결과를 다시 IDE에서 검토하는 하이브리드 방식이 확산하고 있다. 또한, Replit Agent와 Lovable처럼 자연어만으로 응용 프로그램을 생성하는 서비스도 등장하면서 비개발자의 활용 범위도 점차 확대되고 있다.

조직이 자체 AI 에이전트를 구축하기 위한 프레임워크도 빠르게 발전하고 있다. MetaGPT[26], ChatDev 등이 다중 에이전트 협업 방식을 제시한 이후, LangGraph[27], CrewAI와 같은 프레임워크와 OpenAI, Anthropic, Google 등이 제공하는 SDK(Software Development Kit)는 상태 관리, 메모리, 도구 호출 등 공통 기능을 제공하며 에이전트 개발을 지원하고 있다. 또한, 여러 에이전트를 함께 운영하는 환경에서는 작업 이력과 실행 결과, 비용 등을 통합 관리하는 플랫폼의 중요성도 커지고 있다.

3. 에이전틱 개발 환경의 진화

최근 코드 생성 도구는 표 3과 같이 IDE 기반 코드 보조에서 CLI 기반 에이전트, 자율형 에이전트, 에이전트 구축 프레임워크, 에이전트 운영 플랫폼으로 빠르게 발전하고 있다. 초기에는 개발자의 코딩 작업을 보조하는 수준에 머물렀으나, 최근에는 요구사항 분석부터 코드 생성, 테스트, 리뷰, 배포까지 소프트웨어 개발 전 과정을 여러 에이전트가 협력하여 수행하는 에이전틱 개발 환경으로 진화하고 있다. 에이전트 생태계가 확대되면서 상호운용성과 표준화가 중요한 기술 요소로 부상하고 있다. MCP(Model Context Protocol)는 에이전트가 외부 도구와 데이터에 접근하는 방식을 표준화하였으며[28], A2A(Agent2Agent)는 에이전트 간 협업을 지원하는 표준 프로토콜로 발전하고 있다. 또한, AGENTS.md, SKILL.md와 같은 에이전트 설정 파일도 여러 도구에서 공통으로 활용되며 에이전트의 이식성을 높이고 있다. 다만, 이러한 표준을 통해 에이전트가 접근할 수 있는 자원이 늘어나면서 악성 MCP 서버, 악성 스킬, 프롬프트 주입(Prompt Injection)과 같은 새로운 보안 위협도 함께 증가하고 있다.

이와 함께 개발자의 역할도 직접 코드를 작성하는 것에서 AI 에이전트에게 작업 목표와 요구사항을 제시하고 생성된 결과를 검토하는 방향으로 변화하고 있다. 그러나 AI 도구가 항상 개발 생산성을 높이는 것은 아니다. 개발자 서베이에서는 AI 도구 사용률이 80~90% 수준에 이르렀지만, 생성 결과에 대한 신뢰는 오히려 낮아지고 있으며, 숙련 개발자가 익숙한 코드베이스에서 AI 도구를 사용할 때 작업 시간이 오히려 증가했다는 연구 결과[29]도 보고되었다.

이에 따라 최근에는 단순히 코드를 생성하는 능력보다 복잡한 개발 작업을 안정적으로 수행하고 생성 결과를 신뢰할 수 있는지가 더욱 중요해지고 있으며, 이는 다음 장에서 살펴볼 코드 품질과 신뢰성 문제로 이어진다.

V. 코드 인텔리전스의 평가와 신뢰성

코드 생성 모델과 에이전틱 코딩 도구가 빠르게 발전하면서 생성 결과를 어떻게 평가하고 신뢰성을 확보할 것인지가 중요한 과제로 부상하고 있다. 이 장에서는 평가 벤치마크의 발전 과정과 생성 코드 및 에이전트의 신뢰성, 그리고 실제 개발 환경에서의 생산성 효과를 차례로 살펴본다.

1. 평가 벤치마크의 진화

초기 벤치마크는 하나의 함수를 얼마나 정확하게 생성하는지를 평가하는 데 초점을 두었으나, 최근에는 저장소를 이해하고 실제 소프트웨어 개발 문제를 해결하는 능력을 평가하는 방향으로 발전하고 있다. 본고에서는 이러한 발전 과정을 평가 대상에 따라 4세대로 구분하였다(표 4). 1세대는 HumanEval[1], MBPP와 같이 단일 함수를 생성한 뒤 테스트를 통해 정확성을 평가하는 방식이다. 이후 EvalPlus[30]는 테스트 케이스를 보강하여 평가의 신뢰성을 높였으며, LiveCodeBench[31]는 새로운 문제를 지속적으로 추가하여 기존 문제를 미리 학습하는 데이터 오염(Data Contamination) 문제를 줄이고자 하였다. 2세대는 RepoBench, CrossCodeEval, ClassEval 등 저장소 문맥을 고려하여 코드 생성 능력을 평가한다.

표 4 코드 인텔리전스 평가 벤치마크의 발전

세대	대표 벤치마크	특징
1세대	HumanEval, MBPP, EvalPlus, LiveCodeBench	함수 생성 및 실행 정확도 평가 (독립 함수)
2세대	RepoBench, CrossCodeEval, ClassEval	저장소 이해 및 코드 완성 평가 (저장소 문맥)
3세대	SWE-bench, SWE-bench Verified	실제 GitHub 이슈 해결 능력 평가 (이슈 해결)
4세대	SWE-bench Pro, SWE-Lancer, Terminal-Bench	장기·복합 개발 작업 수행 능력 평가 (장기 개발)

3세대는 SWE-bench[32]의 등장과 함께 본격화되었다. 실제 GitHub 이슈와 저장소를 기반으로 수정 패치를 생성하도록 하여 실제 개발 문제를 해결하는 능력을 평가하며, SWE-bench Verified는 현재 에이전트 성능을 평가하는 대표적인 벤치마크로 활용되고 있다. 초기에는 최고 해결률이 2%에도 미치지 못했으나, 2026년 상반기에는 상위 모델 기준 80% 후반에서 90% 수준까지 향상되었다. 4세대는 장시간에 걸쳐 수행되는 복잡한 개발 작업을 평가한다. SWE-Bench Pro(2025)는 수 시간에서 수일이 소요되는 다중 파일 수정 작업을 포함하며[33], SWE-Lancer와 Terminal-Bench는 실제 개발 업무와 유사한 전문가 수준의 작업까지 평가 대상으로 확대하고 있다.

한편, 벤치마크 자체의 신뢰성 문제도 제기되고 있다. 오래 공개된 문제는 학습 데이터에 포함되어 평가 결과가 실제 성능보다 높게 나타날 수 있으며, 일부 에이전트는 Git 이력과 같은 외부 정보를 활용하여 문제를 해결하기도 한다. 이에 따라 지속적으로 새로운 문제를 추가하는 Live 벤치마크와 비공개 저장소를 활용한 평가 등 더욱 신뢰성 있는 평가 방법에 관한 연구가 활발히 이루어지고 있다.

2. 생성 코드와 에이전트의 신뢰성

생성 코드에서는 존재하지 않는 API나 패키지를 사용하는 코드 환각(Code Hallucination), 보안 취약점이 포함된 코드 생성, 저작권 및 라이선스 위반 등이 주요 문제로 지적되고 있다[34]. 이러한 문제를 줄이기 위해 생성된 코드는 실행 검증, 정적 분석, 보안 검사, 라이선스 검증 등을 거쳐 품질과 안전성을

확인하는 과정이 일반적으로 적용되고 있다.

최근에는 생성 코드뿐만 아니라 에이전트 자체의 신뢰성도 중요한 이슈로 부상하고 있다. 에이전트는 셸 명령 실행, 파일 수정, 외부 서비스 호출 등 다양한 작업을 수행하므로 프롬프트 주입이나 악성 MCP 서버와 같은 새로운 보안 위협에 노출될 수 있다. 이에 따라 샌드박스 환경에서 에이전트를 실행하거나 필요한 권한만 부여하는 최소 권한 원칙, 그리고 중요한 작업은 사람이 최종 확인하는 인간 승인(Human-in-the-Loop) 방식이 널리 활용되고 있다. 이처럼 최근에는 생성된 코드뿐만 아니라 에이전트가 실행되는 환경까지 함께 보호하는 것이 중요한 과제로 인식되고 있다.

3. 코드 생산성 효과와 실제 활용

AI 코드 생성 도구의 생산성 효과는 개발 환경과 작업의 특성에 따라 다르게 나타난다. 초기 GitHub Copilot 실험에서는 개발 시간이 크게 단축되는 효과가 보고되었으며[5], 반복적인 코드 작성이나 테스트 생성과 같은 작업에서는 생산성 향상이 꾸준히 확인되고 있다[35]. 반면, METR의 2025년 무작위 대조 실험에서는 숙련 개발자가 익숙한 대규모 오픈소스 저장소에서 AI 도구를 사용할 경우 실제 작업 시간이 약 19% 증가한 것으로 나타났다[29]. 이는 생성된 코드를 그대로 사용하는 것이 아니라, 검토하고 수정하는 과정에 추가 시간이 필요하기 때문으로 분석되었다.

따라서 AI 코드 생성 도구는 신규 기능 구현이나 반복적인 작업에서는 높은 생산성을 제공하지만, 복잡한 저장소를 이해하거나 설계 판단이 필요한 작업에서는 오히려 검토와 수정에 더 많은 시간이 필요할 수 있다. 최근에는 코드 생성 속도뿐만 아니라 생성 코드의 품질과 신뢰성, 토큰 사용량, 실행 비용까지 함께 고려하는 비용 대비 효과(ROI: Return On Investment) 중심의 평가가 이루어지고 있다. 앞으로 코드 인텔리전스 기술은 단순히 코드를 빠르게 생성하는 것을 넘어, 신뢰할 수 있는 결과를 경제적으로 제공하는 방향으로 발전할 것으로 전망된다.

VI. 국내 코드 인텔리전스 동향

1. 국내 산업 및 연구 동향

국내에서도 2025년 이후 생성형 AI 기반 개발 도구의 도입이 빠르게 확대되고 있다. 삼성전자는 DX 부문을 중심으로 ChatGPT, Gemini, Claude 등 상용 생성형 AI를 개발 환경에 도입하고 있으며[36], OpenAI와 협력하여 Codex 기반의 엔터프라이즈 코딩 환경을 구축하고 있다. LG그룹은 LG CNS를 중심으로 상용 AI 서비스와 함께 자체 LLM인 EXAONE 기반의 개발 지원 환경을 확대하고 있으며, 네이버와 카카오 역시 자체 LLM과 외부 AI 서비스를 함께 활용하고 있다. 이처럼 국내 산업계는 상용 AI 서비스를 활용하는 동시에 자체 AI 모델을 개발하는 방향으로 코드 인텔리전스 기술을 도입하고 있다.

한편, 공공기관과 연구기관에서는 해외 상용 서비스 의존도를 줄이고 폐쇄망 환경에서도 활용할 수 있는 코드 인텔리전스 기술을 개발하기 위한 연구가 활발히 진행되고 있다. 공개 모델의 성능 향상으로 온프레미스 기반 코드 생성 환경의 구축이 가능해졌지만, 실제 개발 환경에 적용하기 위해서는 에이전트 오케스트레이션, 실행 기반 검증, 저장소 수준의 컨텍스트 관리와 같은 기반 기술이 함께 필요하다. 이에 따라 과학기술정보통신부와 정보통신기획평가원(IITP)을 중심으로 관련 연구개발이 추진되고 있으며, ETRI의 PULSE 프레임워크도 이러한 연구의 대표적인 사례이다.

2. PULSE 프레임워크와 ArchCode

ETRI는 IITP 연구개발 과제를 통해 자연어 요구사항으로부터 전문 개발자 수준의 코드를 자동 생성하는 기술을 연구하고 있으며, 이를 위해 PULSE(Pipeline for Unified LLM Software Engineering) 프레임워크를 개발하였다. PULSE는 다양한 LLM을 활용하여 요구사항 분석부터 코드 생성, 테스트, 실행 검증까지의 과정을 하나의 파이프라인으로 구성하고 실행할 수 있는 코드 생성 프레임워크이다.

PULSE는 각 기능을 독립적인 모듈로 구성하여 필요한 기능만 조합해 사용할 수 있도록 설계되었다. 요구사항 분석, 구현 계획 수립, 코드 생성, 테스트 생성, 실행 검증 등을 하나의 파이프라인으로 연결하며, OpenAI API, vLLM, Ollama 등 다양한 LLM 실행 환경도 동일한 방식으로 활용할 수 있다. 또한, 새로운 검증 기능이나 도메인 특화 모듈을 쉽게 추가할 수 있어 다양한 개발 환경에 적용할 수 있다(그림 2).

ArchCode는 PULSE를 이용해 구현한 요구사항 기반 코드 생성 시스템이다. 요구사항 분석, 구현 계획 수립, 코드와 테스트 생성, 실행 기반 검증, 결과 선별의 다섯 단계로 구성되며, 기능 요구사항뿐만 아니라 예외 처리와 성능 등 비기능 요구사항(NFR: Non-Functional Requirements)까지 함께 고려하여 테스트를 수행한다. 공개 벤치마크에서는 일반적인 프롬프팅 방식보다 높은 Pass@k 성능을 보였으며, HumanEval-NFR에서도 우수한 성능을 확인하였다[37]. 이를 통해 요구사항 분석부터 실행 기반 검증까지의 전 과정을 하나의 파이프라인으로 통합하여 코드 생성의 품질과 신뢰성을 향상시킬 수 있음을 확인하였다.

3. 의의와 향후 발전 방향

PULSE는 요구사항 분석부터 코드 생성, 테스트, 실행 검증까지의 과정을 하나의 파이프라인으로 지원하는 국내 코드 인텔리전스 프레임워크이다. 또한, 상용 모델과 공개 모델을 함께 활용할 수 있어 폐쇄망을 포함한 다양한 개발 환경에 적용할 수 있다. 앞으로는 저장소 단위의 코드 생성과 에이전틱

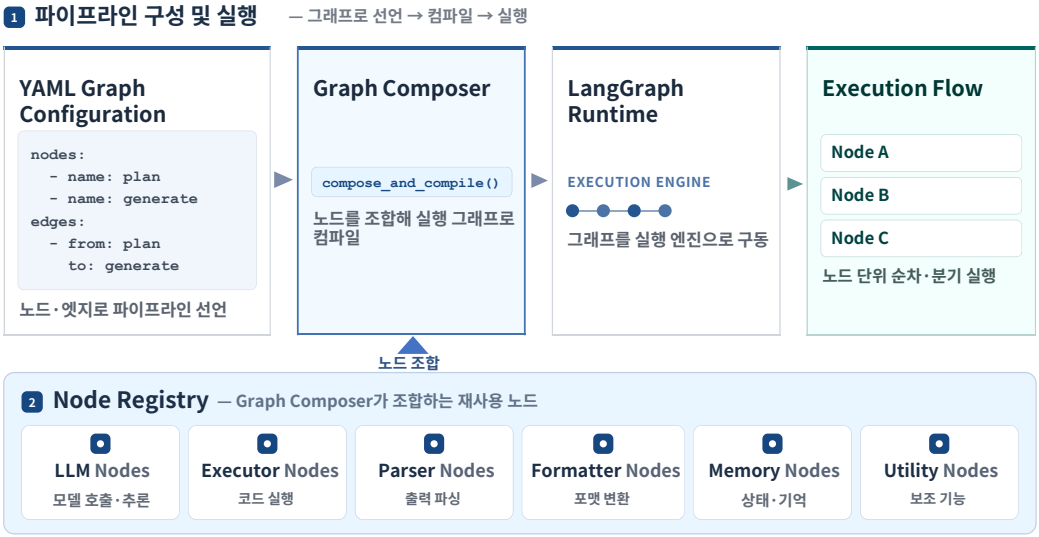

그림 2 PULSE 프레임워크 기술 구성

코딩을 지원하기 위해 저장소 컨텍스트 관리와 멀티에이전트 협업 기능을 강화하고, 또한 다양한 개발 도구와의 연계, AI 코드 리뷰, 보안 검증 기능 등을 추가하여 국내 개발 환경에 적합한 코드 인텔리전스 프레임워크로 발전시켜 나갈 예정이다. 이러한 실행 · 검증 중심의 구조는 향후 시뮬레이터, 디바이스, 로봇 등 실제 실행 환경과 연계되는 피지컬 AI 소프트웨어 개발로 적용 범위를 확장하기 위한 기반으로도 활용할 수 있다.

VII. 결론

본고에서는 피지컬 AI 시대의 소프트웨어 개발 환경 변화라는 관점에서 코드 인텔리전스 기술의 발전 동향을 데이터셋과 모델, 코드 생성 파이프라인, 에이전틱 코딩 생태계, 평가와 신뢰성, 그리고 국내 기술 동향을 중심으로 살펴보았다. 최근 코드 인텔리전스는 단순한 코드 자동완성을 넘어 자연어 요구사항을 이해하고 코드를 생성한 뒤 실행을 통해 검증하는 방향으로 발전하고 있다. 또한, 코드 생성 과정은 실행 기반 검증과 반복적인 개선을 포함하는 형태로 발전하고 있으며, 앞으로 코드 인텔리전스 기술의 경쟁력은 얼마나 많은 코드를 생성하는가보다 생성된 코드를 얼마나 신뢰하고 실제 개발에 활용할 수 있는가에 달려 있다. 이를 위해 실행 기반 검증, 신뢰할 수 있는 평가 체계, 안전한 에이전트 실행 환경이 더욱 중요해질 것으로 예상된다. 특히 피지컬 AI 시스템은 소프트웨어가 시뮬레이터, 디바이스 및 물리 환경과 연계되어 동작해야 하므로, 저장소 수준의 코드 생성만으로는 충분하지 않으며, 실제 실행 환경의 피드백을 개발 과정에 통합하는 방향으로 코드 인텔리전스가 확장될 필요가 있다. ETRI의 PULSE는 이러한 흐름에 맞추어 요구사항 분석부터 코드 생성과 실행 검증까지를 하나의 파이프라인으로 지원하는 프레임워크이며, 앞으로 다양한 개발 환경과 도구를 지원하고 실제 실행 환경과의 연계를 강화함으로써 피지컬 AI 시대의 소프트웨어 개발을 지원하는 코드 인텔리전스 기반 기술로 발전할 수 있을 것으로 기대된다.

용어해설

코드 인텔리전스(Code Intelligence) 소스 코드의 이해, 생성, 변환, 검증을 수행하는 AI 기술의 총칭으로, 코드 자동완성, 버그 수정, 테스트 생성 등에 활용됨

에이전틱 코딩(Agentic Coding) AI 에이전트가 계획 수립, 파일 수정, 명령 실행, 테스트를 반복하며 개발 작업을 자율적으로 수행하는 개발 방식

사양 주도 개발(Spec-Driven Development) 요구사항과 설계를 먼저 문서화하고, 이를 기준으로 AI 에이전트가 구현과 검증을 수행하는 개발 방식

참고문헌

[1] M. Chen et al., “Evaluating Large Language Models Trained on Code,” arXiv preprint, 2021, doi: 10.48550/arXiv.2107.03374

[2] B. Rozi re et al., “Code Llama: Open Foundation Models for Code,” arXiv preprint, 2023, doi: 10.48550/arXiv.2308.12950

[3] D. Guo et al., “DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence,” arXiv preprint, 2024, doi: 10.48550/arXiv.2401.14196

[4] B. Hui et al., “Qwen2.5-Coder Technical Report,” arXiv preprint, 2024, doi: 10.48550/arXiv.2409.12186

[5] S. Peng et al., “The Impact of AI on Developer Productivity: Evidence from GitHub Copilot,” arXiv preprint, 2023, doi: 10.48550/arXiv.2302.06590

[6] R. Li et al., “StarCoder: May the Source Be with You!,” Trans. Mach. Learn. Res., 2023.

[7] A. Lozhkov et al., “StarCoder 2 and The Stack v2: The Next Generation,” arXiv preprint, 2024, doi: 10.48550/arXiv.2402.19173

[8] L.B. Allal et al., “SmolLM2: When Smol Goes Big—Data-Centric Training of a Small Language Model,” arXiv preprint, 2025, doi: 10.48550/arXiv.2502.02737

[9] S. Huang et al., “OpenCoder: The Open Cookbook for Top-Tier Code Large Language Models,” arXiv preprint, 2024, doi: 10.48550/arXiv.2411.04905

[10] K. Fujii et al., “Rewriting Pre-Training Data Boosts LLM Performance in Math and Code,” arXiv preprint, 2025, doi: 10.48550/arXiv.2505.02881

[11] Z. Luo et al., “WizardCoder: Empowering Code Large Language Models with Evol-Instruct,” in Proc. Int. Conf. Learn. Represent., (Vienna, Austria), 2024.

[12] Y. Wei et al., “Magicoder: Empowering Code Generation with OSS-Instruct,” in Proc. Int. Conf. Mach. Learn., (Vienna, Austria), 2024, pp. 52632-52657.

[13] W.U. Ahmad et al., “OpenCodeInstruct: A Large-Scale Instruction Tuning Dataset for Code LLMs,” arXiv preprint, 2025, doi: 10.48550/arXiv.2504.04030

[14] J. Pan et al., “Training Software Engineering Agents and Verifiers with SWE-Gym,” in Proc. Int. Conf. Mach. Learn., (Vancouver, Canada), 2025, pp. 47717-47737.

[15] J. Yang et al., “SWE-smith: Scaling Data for Software Engineering Agents,” in Proc. 39th Conf. Neural Inf. Process. Syst., Datasets and Benchmarks Track, (San Diego, CA, USA), 2025.

[16] Y. Wei et al., “SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution,” in Proc. 39th Conf. Neural Inf. Process. Syst., (San Diego, CA, USA), 2025.

[17] J. Gehring et al., “RLEF: Grounding Code LLMs in Execution Feedback with Reinforcement Learning,” arXiv preprint, 2024, doi: 10.48550/arXiv.2410.02089

[18] D. Guo et al., “DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning,” Nature, vol. 645, no. 8081, 2025, pp. 633-638.

[19] X. Hou et al., “Large Language Models for Software Engineering: A Systematic Literature Review,” ACM Trans. Softw. Eng. Methodol., vol. 33, no. 8, 2024, pp. 1-79.

[20] F. Zhang et al., “RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation,” in Proc. Conf. Empir. Methods Nat. Lang. Process., (Singapore), 2023, pp. 2471-2484.

[21] X. Jiang et al., “Self-Planning Code Generation with Large Language Models,” ACM Trans. Softw. Eng. Methodol., vol. 33, no. 7, 2024, pp. 1-30.

[22] Y. Li et al., “Competition-Level Code Generation with AlphaCode,” Science, vol. 378, no. 6624, 2022, pp. 1092-1097.

[23] B. Chen et al., “CodeT: Code Generation with Generated Tests,” in Proc. Int. Conf. Learn. Represent., (Kigali, Rwanda), 2023.

[24] X. Chen et al., “Teaching Large Language Models to Self-Debug,” in Proc. Int. Conf. Learn. Represent., (Vienna, Austria), 2024.

[25] J. Yang et al., “SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering,” in Proc. 38th Conf. Neural Inf. Process. Syst., (Vancouver, Canada), 2024.

[26] S. Hong et al., “MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework,” in Proc. Int. Conf. Learn. Represent., (Vienna, Austria), 2024.

[27] LangChain, “LangGraph,” GitHub. https://github.com/langchain-ai/langgraph [Accessed July 9, 2026.]

[28] Anthropic, “Introducing the Model Context Protocol,” Anthropic, 2024. 11. 25. https://www.anthropic.com/news/model-context-protocol

[29] J. Becker et al., “Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity,” arXiv preprint, 2025, doi: 10.48550/arXiv.2507.09089

[30] J. Liu et al., “Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation,” in Proc. 37th Conf. Neural Inf. Process. Syst., (New Orleans, LA, USA), 2023.

[31] N. Jain et al., “LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code,” in Proc. Int. Conf. Learn. Represent., (Singapore), 2025.

[32] C.E. Jimenez et al., “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?,” in Proc. Int. Conf. Learn. Represent., (Vienna, Austria), 2024.

[33] X. Deng et al., “SWE-bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?,” arXiv preprint, 2025, doi: 10.48550/arXiv.2509.16941

[34] H. Pearce et al., “Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions,” in Proc. IEEE Symp. Secur. Priv., (San Francisco, CA, USA), 2022, pp. 754-768.

[35] Anthropic, “2026 Agentic Coding Trends Report: How Coding Agents Are Reshaping Software Development,” 2026. https://resources.anthropic.com/2026-agentic-coding-trends-report

[36] 서유덕, “삼성전자, DX부문 챗지피티·제미나이·클로드 도입,” 정보통신신문, 2026. 6. 11. https://www.koit.co.kr/news/articleView.html?idxno=207524

[37] H. Han et al., “ArchCode: Incorporating Software Requirements in Code Generation with Large Language Models,” in Proc. 62nd Annu. Meet. Assoc. Comput. Linguist., (Bangkok, Thailand), 2024, pp. 13520-13552.

[38] H. Husain et al., “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search,” arXiv preprint, 2019, doi: 10.48550/arXiv.1909.09436