Computers and Artificial Intelligence; Vol.3, No.4; 2026
E-ISSN: 3006-3728 P-ISSN: 3006-371X
DOI: https://doi.org/10.70267/cai.v3n4.127145
Published by: Zeus Press

A Review of Research on the Development of Artificial Intelligence Programming Agent-Assisted Software

Yuzhe Li*

School of Microelectronics, Hefei University of Technology, Hefei 230601, China

**Corresponding author: Yuzhe Li.*

Abstract

The development of large language models has enabled AI-assisted programming to gradually expand from early code completion to requirements understanding, code generation, testing, debugging, and project maintenance. The programming intelligence agent formed on this basis is no longer just returning a piece of code based on user input, but can read project files, break down tasks, call terminals and testing tools, and repeatedly modify the program based on the running results. Based on 60 publicly searchable representative papers, this paper adopts a structured narrative review method to review the research progress from the aspects of code-based models, agent control mechanisms, repository-level systems, evaluation benchmarks, and human-computer collaboration evidence. It also compares the research scope and analysis focus with existing reviews on software engineering agents. This paper further utilizes method classification, performance statistics, and mechanism diagrams to analyze different technical approaches, focusing on issues such as reliability, contextual understanding in large-scale projects, security and privacy, evaluation and reproducibility, operating costs, and accountability and governance. Research shows that programming agents can reduce the cost of repetitive coding and help users lacking professional programming experience complete prototype development. However, its efficiency gains are significantly affected by task size, project familiarity, quality standards, and manual review costs, making it difficult to consistently understand requirements and guarantee overall quality in complex projects. Future research should shift from simply pursuing the quantity of code generated to building verifiable, explainable, traceable, and collaborative software development processes.

Keywords

artificial intelligence programming, large language models, programming agents, software engineering, code generation

1. Introduction

Software development is a complex activity that relies on formal knowledge, engineering experience, and teamwork. Developers not only need to master programming languages and development frameworks, but also need to complete the design, coding, testing, deployment and maintenance under conditions where requirements are not fully clear, project structure is constantly evolving and quality goals are mutually constrained. Syntax checking, static analysis, and code completion in traditional integrated development environments can reduce typing errors and repetitive work, but these tools mainly work based on predefined

rules or local contexts and have difficulty understanding cross-file dependencies, business constraints, and the developer's high-level goals. Therefore, early AI programming tools primarily served as “assistive input” tools and did not substantially change the human-centered software development process.

The emergence of large language models has changed the way people interact with programming tools. The models can map natural language requirements into code, and can also interpret programs, generate tests, analyze errors, and suggest modifications. Users thus do not need to pre-convert all their intentions into precise program statements, but can explain their goals step by step through continuous dialogue. Studies exemplified by Codex and subsequent large-scale code models have shown that when a model is pre-trained on large-scale natural language and source code data, it can demonstrate strong transfer capabilities in tasks such as function generation, code completion, code translation, and program repair [1-3]. This capability lowers the operational threshold for program development and enables artificial intelligence to expand from local code suggestions to multiple stages of the software lifecycle.

However, conversational code generation is still primarily a “suggestion-adoption” model: the model provides text, and the user decides which file to modify, how to execute the program, and how to handle failures. When the model is further connected to file systems, compilers, testing frameworks, browsers, and version control tools, and has the ability to plan tasks, remember states, and provide feedback corrections, AI-assisted programming evolves into a programming intelligent agent. Compared with ordinary dialogue models, the key difference of programming agents is not only that the output content is longer, but also that it forms an execution closed loop of perceiving the environment–selecting actions–receiving feedback–continuing decision-making [4-6]. It can search code repositories, modify multiple files, run tests, and relocate the problem after failure, based on the problem description. This ability to act brings them closer to collaborators who can use development tools.

Programmable intelligent agents have also changed the research objects of software engineering. Traditional code generation research often treats individual functions as independent samples and measures results by the pass rate of test cases. Real-world projects contain a large number of files, dependencies, configurations, and implicit specifications. Modifying a local module may affect the entire system. Repository-level benchmarks such as SWE-bench combine real-world problem descriptions with specific versions of open-source code repositories, requiring the system to complete defect localization, patch generation, and test verification [7]. This means that the evaluation focus has shifted from “whether the model can be used to write a piece of code” to “whether the system can make verifiable changes in a constrained engineering environment.” Whether the code retrieval is accurate, whether the plan is stable, whether the tool calls are effective, and whether the scope of modification is controllable are all important factors affecting the success rate of the task.

At the same time, a higher degree of automation does not necessarily lead to stable efficiency gains. While programming intelligent agents can generate and modify code quickly, developers still need to undertake tasks such as clarifying requirements, judging solutions, reviewing results, and controlling risks. When the task boundaries are clear and the results are easy to test automatically, the model can significantly reduce boilerplate code and information retrieval costs. When projects are large, quality standards are high, or business knowledge is difficult to express explicitly, checking for errors and correcting errors may offset the time saved [8,9]. Furthermore, intelligent agents can execute commands and access project data, and their errors are no longer limited to a piece of unworkable code, but can also spread to configuration corruption, sensitive information leakage, or supply chain security issues.

Existing reviews have discussed the application of large language models in software engineering, the structure of intelligent agents based on large language models, and the development of code generation models [3-6,10]. However, from the perspective of actual software development, it is still necessary to put model capabilities, agent methods, engineering tools, and human-machine collaboration in the same analysis framework: On the one hand, it is necessary to explain the specific mechanisms through which various methods improve code understanding, planning, retrieval, or verification; on the other hand, it is also necessary to identify the differences between benchmark results, experimental productivity, and real engineering value. Simply listing system names and task results can easily overlook the technical assumptions and applicable boundaries of different methods.

Table 1 further provides the positioning differences between this article and representative reviews. Existing work has established a relatively complete knowledge base, but most studies focus on software

engineering task coverage or general intelligent agent architecture. This article takes "method mechanism–warehouse-level execution–empirical effect–engineering governance" as a continuous analysis chain, and presents evidence through literature distribution, method comparison and performance statistics, with particular attention to whether high benchmark results can be transformed into maintainable and auditable real engineering results.

Table 1: Comparison of Research Positioning between This Paper and Representative Reviews

Study	Main Coverage	Analytical Framework or Key Contributions	Relationship to This Paper
Hou et al. [3]	Applications of large language models in the software engineering lifecycle	Systematically summarizes evidence on tasks such as requirements, development, testing, and maintenance	Broad coverage; agent execution closed-loop is not the sole focus
Jin et al. [4]	Evolution from large language models to software engineering agents	Distinguishes models from agents and organizes research by six software engineering topics	Emphasizes lifecycle migration and provides a perspective on agentic transformation
Liu et al. [5]	Software engineering agents and their collaboration patterns	Classifies from dual perspectives of SE tasks and agent capabilities, and discusses multi-agent systems and human-computer interaction	Comprehensive system, focusing on the research landscape and architecture categories
Wang et al. [6]	Technical landscape and vision of software engineering agents	Explains agent architecture with perception, memory, and action as the core	Provides a general capability framework for analyzing operational mechanisms
This paper	Programming agents oriented to real development workflows	Connects methodological mechanisms, repository-level execution, performance and cost evidence, security governance, and human-machine responsibility	Emphasizes evidence comparability, applicability boundaries, and verifiable collaborative processes

1.1 Research Questions and Literature Analysis Methods

Based on the aforementioned research gaps, this paper sets out four interrelated research questions: RQ1, what key technological changes has occurred in the evolution of AI programming from code models to programming intelligent agents, and through what mechanisms do various methods function? RQ2, what working patterns have programming intelligent agents formed at different stages of software development, and under what conditions do their efficiency gains hold true? RQ3, what reliability, security, cost, and human-machine collaboration issues have existing benchmark and empirical studies revealed? RQ4, for the implementation of trusted engineering, what capabilities and evidence should future research prioritize in addition to? These four questions correspond to the technical mechanisms, application effects, practical challenges, and future agenda of this paper, enabling the paper to not only describe "what systems exist," but also analyze "why they are effective, when they fail, and how to verify them."

Methodologically, this paper adopts a structured narrative review rather than an exhaustive systematic review. The literature primarily covers peer-reviewed papers and preprints published or stably searchable between 2020 and 2025, prioritizing the following studies: those directly discussing the technical mechanisms of large-scale code models or programming agents; proposing methods for using repository-level systems or tools; establishing widely adopted benchmarks for code generation, bug fixing, and security evaluation; and measuring developer behavior and productivity through user studies or controlled experiments. Product promotional materials, data lacking methodological explanations and unverifiable data, and duplicate versions of the same work are not considered core evidence. For important, rapidly evolving but not yet formally published results, this paper retains publicly traceable preprints and clearly defines their evidentiary boundaries in the discussion.

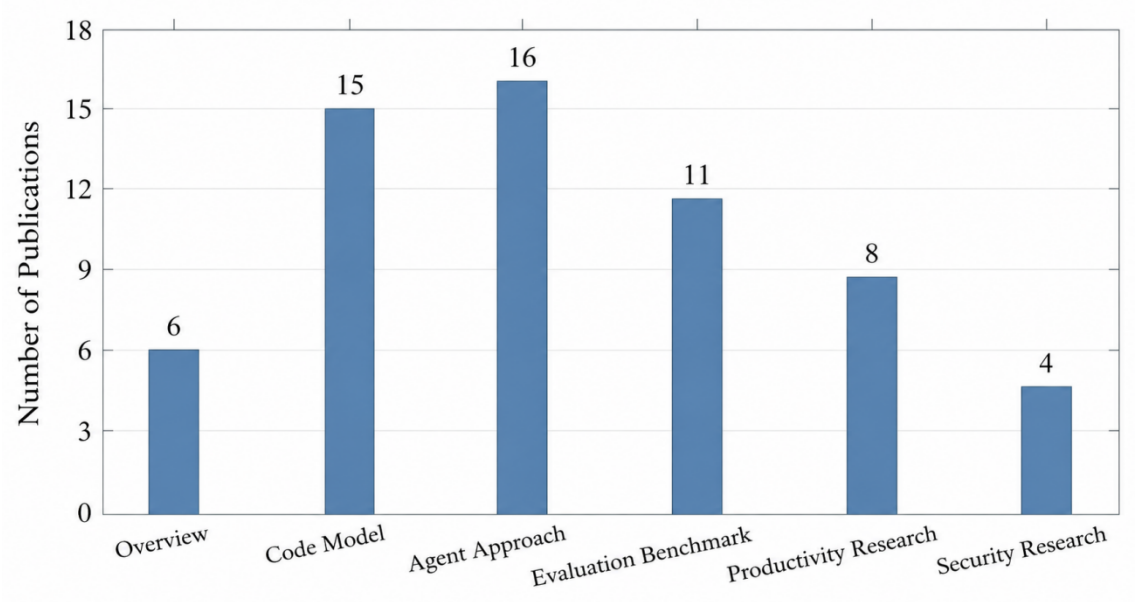
Document coding revolves around six pieces of information: research tasks, model or agent mechanisms, tools and environments used, evaluation benchmarks and indicators, human participation methods, and limitations reported by the authors. The 60 documents were then classified into six categories: review research, code-based models, agent methods and systems, evaluation benchmarks, developers and productivity research, and security research. This process does not attempt to simply merge heterogeneous experimental results, but conducts evidence triangulation through mechanism comparisons, performance intervals, and study design

differences, thereby avoiding treating values under different data sets, model versions, and tool configurations as a unified scale that can be directly ranked.

Based on the aforementioned problems and methods, this paper provides a review across three levels. First of all, from a technological evolution perspective, it outlines code pre-training models, inference and tool invocation methods, and programming intelligent agent systems, explaining their core mechanisms and interrelationships. Second, from a software lifecycle perspective, it analyzes the application of intelligent agents in requirements gathering, coding, testing, debugging, and maintenance, comparing advisory, collaborative, and relatively autonomous working modes. Finally, from an engineering implementation perspective, it discusses issues such as reliability, repository context, security and privacy, evaluation and reproducibility, operating costs, and accountability governance. Building on this foundation, this paper further argues that the development focus of programming intelligent agents should shift from expanding the scale of code generation to establishing verifiable, explainable, and traceable human-machine collaborative processes.

In order to avoid excessive concentration of literature selection on a single technical route, this article divides the 60 cited documents into six categories: review research, code-based models, agent methods and systems, evaluation benchmarks, developer and productivity research, and security research. Figure 1 shows that code models and agent methods together account for more than half of the total literature, constituting the main body of technical analysis. At the same time, evaluation, productivity, and safety research occupy a relatively stable proportion and are used to evaluate programming agents solely from engineering results rather than model capabilities. This distribution also shows that existing research pays significantly more attention to "how to build a system" than to issues such as long-term maintenance, organizational governance, and responsibility tracking.

Figure 1: Thematic Distribution of the 60 References in this Article



2. Development and Operation Mechanism of Artificial Intelligence-Assisted Programming

2.1 From Code Completion to Programming Agents

The development of artificial intelligence-assisted programming can be roughly divided into three stages. The first stage is code completion based on rules and local context. The tool predicts the next possible statement based on variable names, function structures, and the code already entered by the developer. These types of tools primarily reduce repetitive input, but they cannot independently understand more complex development tasks.

The second stage is conversational code generation. Users can use natural language to ask the model to explain code, generate functions, convert programming languages, add comments, or locate errors. Compared

to traditional code completion tools, conversational models lower the barrier to human-computer interaction, allowing users unfamiliar with specific syntax to participate in program development. However, the model at this stage primarily outputs text; whether the generated content is adopted, how it runs, and how to correct errors are usually still decided by the user.

The third stage is agent-based programming. At this stage, the model no longer simply generates text, but can create task plans, search project files, invoke compilers, testing frameworks, browsers, and version control tools, and complete multiple rounds of modifications based on the results returned by the tools. Some systems also save historical information, record the reasons for previous failed modifications, and adjust subsequent execution plans. Related research typically summarizes its core capabilities as environmental awareness, task planning, tool invocation, memory, and feedback, and further distinguishes between single-agent and multi-agent systems [10,11]. However, the ability of an agent to perform operations does not necessarily mean that it possesses complete software engineering judgment.

From a model perspective, the development of programming intelligent agents is based on the continuous evolution of code representation learning and generative models. CodeBERT employs a Transformer encoder to jointly learn natural language and programming language representations, and utilizes "code-document" paired data and monomodal code data through masked language modeling and substitution lexical detection. Its advantages are mainly reflected in understanding tasks such as code search and document generation, but the encoder structure itself is not suitable for directly completing the autoregressive generation of long programs [12]. GraphCodeBERT addresses the problem that pure word sequences are difficult to express program semantics by introducing data flow relationships between variables and designing structure edge prediction, code and structure alignment, and graph-guided attention, enabling the model to learn "where variable values are generated from and where they flow to" [13]. Compared with directly using a deep abstract syntax tree, data flow emphasizes semantic dependencies and is suitable for code search, clone detection, translation, and refinement, but its effectiveness depends on whether static analysis can correctly extract the structure.

PLBART and CodeT5 further adopt an encoder-decoder structure to unify code understanding and code generation into a sequence-to-sequence framework. PLBART performs denoising pre-training on Java, Python codes and natural language technical texts, and learns bidirectional representation and generation capabilities by restoring obscured, deleted or scrambled inputs, which can be used for code summarization, code generation and program translation [14]. CodeT5 emphasizes the special role of identifiers in programs: it first identifies identifiers such as variable names and function names, and then learns program semantics through identifier-aware masking and recovery tasks, while supporting understanding tasks and generation tasks [15]. Both illustrate that code models not only need to learn the co-occurrence relationships of grammatical tokens, but also explicitly utilize program structure and naming information.

In terms of large-scale generation models, Codex further trained the GPT-style autoregressive language model on public code data and proposed HumanEval, which evaluates the functional correctness of function generation by executing tests rather than text similarity [1]. Related research has also constructed tasks such as MBPP that are composed of natural language descriptions, reference codes and test cases, promoting the evaluation from "the output looks similar" to "can the program execute and meet the specifications" [2]. CodeGen not only releases code models of different scales and data stages, but also describes multiple rounds of program synthesis as an interactive process of gradual clarification and generation, emphasizing the importance of continuous communication between users and models [16]. InCoder uses causal mask training to enable the model to be generated from left to right and fill in the missing areas in the middle of the code based on the context, making it more suitable for local insertion and reconstruction in real editing scenarios [17].

AlphaCode is geared towards competitive programming tasks. Its core is not to generate a single answer at a time, but to sample candidate programs on a large scale and then narrow down the candidate set through compilation, sample testing, clustering, and filtering [18]. This method shows that the quality of code generation can be improved through the "generate–execute–select" process, but the computational cost of large-scale sampling is high, and the explicit input and output of competition problems are still significantly different from the requirements of real software. Subsequently, CodeT5+ balanced code understanding, generation, and retrieval by mixing pre-trained objectives and model components [19]. Code Llama performs

code-specific training on the basis of a general language model and provides code completion, instruction following, and intermediate padding variants [20]. StarCoder uses a large-scale code corpus that has been governed and padding training to support multilingual code generation [21].

Subsequent work on the open code model further emphasizes data governance and project-level context. SantaCoder uses relatively strictly screened multilingual code data to study open training solutions for smaller-scale models [22]. StarCoder2 improves code data source tracking, deduplication and training recipes based on The Stackv2, and releases models of multiple parameter sizes [23]. DeepSeek-Coder enhances project-level completion capabilities through training corpus with a high code proportion, long context and warehouse-level data organization [24]. These models themselves are not equivalent to programming agents, but provide the agents with basic capabilities such as code semantic representation, local editing, long context generation and execution candidates.

2.2 Basic Operating Mechanism of Programmable Intelligent Agents

From a system architecture perspective, programming agents typically consist of modules such as a large language model, task planning, context management, tool invocation, and result verification. The large language model is responsible for understanding user requirements and generating analysis, plans, or code. The task planning module decomposes complex goals into several executable steps. The context management module selects information relevant to the current task from the code repository, project documents, and history. The tool interface enables the agent to operate file systems, compilers, testing frameworks, and version control systems. The verification module determines the effectiveness of modifications based on compilation results, test reports, or static analysis results [4,5].

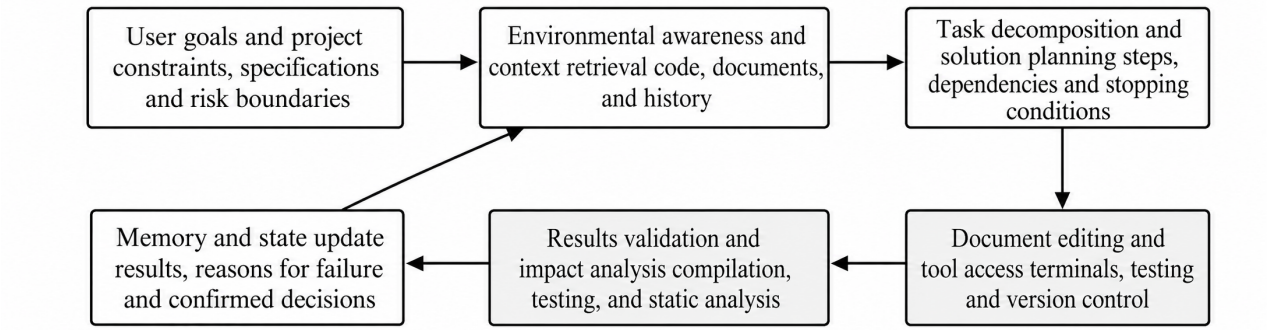
A typical process can be summarized as "understand – plan – execute – observe – correct." The agent first reads the user description and project structure to determine which modules may need modification. It then formulates a preliminary plan, such as locating relevant interfaces, modifying business logic, and supplementing tests. Next, it uses search and file reading tools to obtain context and implements the modifications. After the program runs, if compilation errors or test failures occur, the agent adds the information returned by the tools back into the context, analyzes the causes, and attempts the next round of corrections. This closed loop allows the model to correct errors based on actual runtime results, making it closer to the actual development process than one-time code generation.

However, closed-loop execution does not guarantee the correctness of the final result. An agent's plan may be based on flawed understanding from the outset, and compilers and testing tools can only check explicitly stated rules. They cannot automatically determine whether functionality meets implicit, unstated user requirements. For example, a user might request the development of a "simple user management system" without specifying role divisions, permission scopes, or data storage strategies. The agent might adopt a common design. While technically feasible, this design may not be suitable for the specific scenario. Therefore, the agent's ability to pause execution and proactively inquire when multiple reasonable interpretations of a requirement are possible is a crucial criterion for measuring its maturity.

Context management also directly affects actual performance. Large projects cannot be completely placed into the model context at once, and the system needs to select relevant information through file names, keywords, code dependencies, or vector retrieval. If key documents are missed during the retrieval phase, subsequent inferences, even if complete in form, may be based on insufficient information. Therefore, programming agents not only need the ability to generate code, but also need the ability to accurately locate the code and be able to identify whether they lack necessary information.

Figure 2 summarizes a typical closed loop of a programmed agent. User and project constraints together define task boundaries. The model does not directly treat text output as the final result, but changes the project state through planning, context retrieval and tool execution, and then produces observable feedback through testing, compilation and static analysis. The memory module saves task history, failure reasons, and confirmed decisions, while humans intervene at requirements ambiguities, architectural choices, and high-risk operations. This structure shows that the reliability of the agent depends on the joint action of models, tools, verification and manual control, and distortion in any link may destroy the closed loop.

Figure 2: Typical Closed-loop Architecture for Programming Agent-assisted Software Development



Manual control points: clarification of requirement ambiguity, confirmation of scheme and architecture, authorization of high-risk operations, final quality acceptance

3. Application of Programmable Intelligent Agents in Software Development

3.1 Main Applications in the Software Development Process

During the requirements and design phase, programming agents can organize natural language descriptions into feature lists, interface specifications, and page structures, helping users quickly create software prototypes. For those lacking programming experience, this interactive approach lowers the barrier from idea to runnable program. Users don't need to master the complete syntax from the start. Instead, they can first define the required functionality and then gradually adjust the requirements based on the running results.

During the coding phase, intelligent agents can generate boilerplate code, supplement interfaces, refactor repetitive logic, migrate programming languages, and interpret the implementation methods of existing projects. When a project contains multiple files, some programming agents can search for relevant modules and jointly modify the front-end, back-end, and configuration files. They can also supplement comments according to the existing style of the project, generate interface documentation, or complete maintenance tasks with relatively clear rules, such as dependency version upgrades.

During the testing and debugging phase, the agent can generate test cases based on the function behavior, read the error information after running the program, and then locate the code that may have problems. For software maintenance tasks, it can also analyze defect descriptions, search for related modules, and form patches. SWE-bench combines the problem descriptions and code repositories in real open source projects, requiring the model to directly modify multi-file projects. Its initial version contains 2294 real GitHub issues, reflecting that the evaluation object has shifted from "whether a function can be written" to "whether a real engineering task can be completed" [7].

3.2 Typical Working Mode

Based on the degree of autonomy of the intelligent agent in development, current applications can be broadly categorized into three modes: advisory, collaborative, and relatively autonomous. In the advisory mode, the developer primarily controls the operational process, with the intelligent agent responsible for interpreting code, proposing modifications, or generating local code snippets, but without directly altering project files. This approach facilitates item-by-item review by the developer, carries relatively low risk, and is suitable for projects with high security requirements, but its degree of automation is limited.

The collaborative model allows agents to read project data and modify files, but developers are responsible for verifying task plans, checking code differences, and deciding whether to accept modifications. Agents can run tests and fix simple bugs, while developers are responsible for architecture design and final acceptance. This approach strikes a balance between human control and execution efficiency, making it a more realistic application at present.

The relatively autonomous mode requires the agent to independently search code, modify multiple files, run tests, and generate a patch that can be submitted based on the problem description. This mode is suitable for dependency upgrades, standardized formatting, test supplementation, and well-structured defect repairs.

However, for tasks with ambiguous requirements, complex business rules, or involving important data, fully autonomous execution still carries significant risks. If the agent misunderstands the task early on, the more steps it automatically executes subsequently, the greater the potential impact of the error.

Multi-agent systems attempt to complete development through role division, such as setting separate roles for requirements analysis, code generation, testing, and security review [5,11]. Role division can lead to mutual checks, but multi-agent systems are not necessarily superior to single-agent systems. If agents use similar models and share incomplete contexts, they may produce the same biases. Without clear decision rules, mutual rebuttals, redundant modifications, or unclear responsibilities may also occur. Therefore, relevant research should analyze communication costs, conflict resolution methods, and error propagation paths, in addition to comparing task success rates.

In general agent approaches, ReAct organizes language reasoning and environmental action alternately into a "think–act–observe" trajectory. The model first explains the current judgment in natural language, then invokes search or interaction actions, and uses the results returned by the environment as the basis for the next round of reasoning [25]. This design links internal planning with external evidence, which can reduce factual errors caused by relying solely on closed reasoning and also makes it easier for developers to check the decision-making process. However, if the trajectory selects the wrong tool or search direction early on, subsequent observations may continue to reinforce the wrong assumptions.

Reflexion focuses on how agents learn from failures. It does not update model parameters through gradients, but generates textual "reflections" based on test results, environmental rewards or self-evaluation after the task is over, and saves them in episodic memory for reference in the next attempt [26]. In programming tasks, reflection can record why a certain patch failed the test and what boundary conditions were missed, thereby reducing repeated mistakes. However, the reflection content is still generated by the model. If the cause of failure is misjudged, memory may carry the bias into the next round. Tree of Thoughts expands the single-path thinking chain into a tree search: the system generates multiple candidate "thinking units" at each stage, scores the candidate status, and allows look-ahead, pruning, and backtracking [27]. This method is suitable for planning tasks that require solution comparison, but multi-branch generation and evaluation will significantly increase the cost of reasoning, and reliable state representation and stop conditions are also required in the code base.

Tool learning methods address the question of "when the model calls which tools and how to organize parameters." Toolformer inserts candidate API calls into ordinary text through self-supervision and selects training samples based on whether the call results can reduce subsequent prediction losses, enabling the model to learn to call tools such as calculators, search engines, and translators when needed [28]. TALM combines language models with external tools, allowing the model to continue completing tasks with the information returned by the tools, emphasizing that tool enhancement can compensate for the limitations of parameter knowledge [29]. ToolLLM builds tool usage data for large-scale real APIs and trains the model to handle API selection, parameter generation, and multi-step calls through instruction construction and tool path search [30]. These works laid the foundation for programming agents to call compilers, testers, and code search tools, but successful calls to general APIs do not mean that the model can understand the implicit engineering constraints in the software repository.

AgentBench tests the capabilities of language models as agents in various interactive environments, including operating systems, databases, knowledge graphs, card games, and online shopping. The evaluation focus shifts from static question answering to whether multi-step actions truly change the environment and achieve the goal [31]. It reveals the gap between language ability and action ability: a model may be able to correctly describe operational steps, but fails in long trajectories due to state tracking errors, formatting mistakes, or accumulated biases. This conclusion also applies to programmed agents, because "knowing how to fix" and "performing a verifiable fix in the correct file" are not the same problem.

Multi-agent frameworks attempt to organize complex tasks through role division and communication protocols. AutoGen abstracts agents as participants who can converse, be configured, and connect to tools or humans, allowing developers to define collaborative processes using dialogue relationships [32]. MetaGPT encodes standard operating procedures in software companies into multi-agent workflows, allowing roles such as product managers, architects, and engineers to sequentially generate structured intermediate products such as requirements, designs, and code, thereby reducing information loss in pure chat collaboration [33]. ChatDev

adopts a waterfall-like software phase division and completes design, coding, testing, and documentation work through "chat chains" between roles [34]. CAMEL uses role-playing and initial prompts to enable two agents to collaborate continuously under conditions of clear identities and goals [35]. These methods demonstrate the potential of process division, but if intermediate products are not externally validated, multiple roles may simply continue to propagate the same model's errors within the process.

Table 2 compares representative methods from three dimensions: control structure, feedback sources, and main limitations. It can be seen that method evolution is not simply about increasing autonomy: ReAct and Reflexion strengthen single-agent closed-loop control, Agentless actively limits autonomy in exchange for stability and low cost, while MetaGPT and ChatDev reduce the organizational difficulty of complex tasks through process division of labor. Therefore, when selecting a method, a suitable control structure should be determined based on task verifiability, tool risk, and collaboration costs.

3.3 Differences in Applicability for DIFFERENT Tasks

The performance of programming agents is closely related to task characteristics. For tasks with clearly defined requirements, fixed output formats, and easily automated results verification, agents typically perform well, such as generating data transformation scripts, supplementing simple interfaces, writing unit tests, standardizing code formatting, and updating repetitive configurations. These tasks have clear inputs and outputs, allowing developers to quickly assess results through testing or manual review. However, for complex tasks spanning multiple modules, agent performance is more inconsistent. For example, modifying a permission system might involve database structure, backend interfaces, frontend pages, and deployment configurations simultaneously. Any oversight in any of these areas could cause problems. Software architecture design also includes long-term factors such as performance, scalability, team capabilities, and long-term maintenance, which are difficult to fully incorporate into a single prompt.

Different project phases also affect the effectiveness of use. In the prototype development phase, the goal is usually to quickly validate ideas, and some code may need to be rewritten in the future, so a certain degree of imperfection can be tolerated. Once in the production environment, the system needs to meet stability, security, and compatibility requirements; any changes may affect real users. At this point, the code generated by the agent must undergo a more rigorous testing, review, and release process. Therefore, evaluating programming agents cannot be done without considering the specific task conditions and discussing a uniform percentage of efficiency improvement.

Table 2: Mechanism Comparison of Representative Programming Agent Methods

Method	Core Control Structure	Main Feedback or Intermediate Artifacts	Main Advantages and Limitations
ReAct	Alternating execution of Thought, Action, and Observation	Search results, tool returns, and environment states	Tight coupling between reasoning and evidence; early errors tend to accumulate in long trajectories
Reflexion	Generates textual reflection after failure and writes it into episodic memory	Test results, reward signals, and self-evaluations	No need to update model parameters; erroneous reflections may contaminate subsequent decisions
Tree of Thoughts	Multi-branch candidate generation, scoring, pruning, and backtracking	Candidate state scores and lookahead results	Supports solution comparison; high reasoning and evaluation costs
SWE-agent	Autonomous repository operations via a dedicated agent–computer interface	File contents, command outputs, and test reports	Well-adapted tool interface; long interaction trajectories with unstable costs
Agentless	Fixed three-stage pipeline of localization, repair, and verification	Candidate locations, candidate patches, and verification results	Simple and reproducible process; limited dynamic exploration capability
AutoCode-Rover	Iterative retrieval and patch generation based on program structure	AST structures, fault localization, and test results	Strong localization targeting; dependent on project structure and test quality
MetaGPT/ ChatDev	Role specialization and staged software development process	Requirements, design, code, tests, and documentation	Clear process organization; multiple roles may propagate the same model biases

3.4 Changes in Actual Efficiency

Artificial intelligence programming tools can reduce the time spent looking up syntax, writing repetitive code, and building infrastructure. A controlled experiment on a defined programming task found that participants using GitHubCopilot completed the task 55.8% faster [8]. This result shows that when the task boundaries are clear, the code size is small, and the results are easy to verify, artificial intelligence-assisted tools can improve development efficiency.

However, this conclusion cannot be directly generalized to all development scenarios. A randomized controlled experiment conducted in 2025 on senior open source developers found that in mature projects that developers were familiar with for a long time, allowing the use of existing artificial intelligence tools actually increased the task completion time by 19% [9]. Participants believed that artificial intelligence could save time before and after the experiment, but the actual records showed that waiting for model generation, checking suggestions and correcting errors took up a lot of time.

The results of the two studies are not contradictory: the former has a more clearly defined task scope, making the generated code easier to inspect; the latter involves complex codebases, higher quality standards, and a large amount of implicit context. Therefore, evaluating programming agents cannot simply be based on the amount of code generated. It is also necessary to consider task type, developer experience, human review time, number of defects, and subsequent maintenance costs.

4. Major Challenges

4.1 Reliability of the Generated Results

Large language models generate code based on statistical patterns in training data, which may result in non-existent interfaces, incorrect dependencies, missed boundary conditions, or only passing some tests. While programming agents can repeatedly modify programs using compilation and testing results, passing tests does not necessarily mean that user requirements have been fully implemented.

If the test cases themselves are insufficiently covered, the agent may produce “superficially correct” results. For example, a function might be able to handle existing input but fail to process abnormal data; a modification might resolve a current error but break other modules. Real-world software development also involves runtime performance, platform compatibility, code readability, and long-term maintenance costs, which are difficult to fully evaluate based on a single test result.

In addition, the agent may change the original code structure during the continuous modification process, generating new duplicate codes or irrelevant modifications. When the number of modifications increases, it is difficult for developers to quickly judge whether each step is necessary. Therefore, the agent needs to provide clear modification instructions and difference records, rather than just showing the final code.

4.2 Contextual Understanding of Large-scale Projects

Complex software projects typically contain a large number of source files, third-party dependencies, configuration files, database structures, and internal team specifications. Programming agents, limited by context length and retrieval capabilities, may only see localized code and overlook the relationships between other modules.

Early SWE-bench research showed that real-world defect fixing often requires understanding multiple functions, classes, and files simultaneously, which is significantly more difficult than traditional code generation tasks [7]. Although programming agents have improved their performance on relevant benchmarks in recent years, code localization, long-process planning, and cross-module consistency remain major bottlenecks. When project documentation is incomplete, variable naming is chaotic, or historical code lacks testing, it is more difficult for agents to accurately determine the developer’s original design intent. Future systems cannot rely solely on general model knowledge. They also need to combine project documentation, historical commit records, and team specifications.

For real-world warehouse tasks, SWE-agent first emphasizes the design value of an “agent-computer interface.” Instead of simply exposing a general terminal to the model, it provides file browsing, code editing,

searching, and testing commands suitable for language models, and controls the form of information returned for each observation, enabling the model to navigate the warehouse within a limited context [36]. Its contribution demonstrates that agent performance depends not only on the underlying model, but also on the clarity of the action space and the compactness of tool feedback. However, interactive execution results in a longer trajectory, and erroneous operations and ineffective searches still accumulate costs.

Agentless offers a different approach to complex autonomous loops. This method eliminates the open agent process where the model freely decides the next action, instead decomposing the task into three fixed stages: fault location, patch generation, and patch verification, generating and filtering a limited number of candidates at each stage [37]. This structure reduces tool calls and unpredictable trajectories, making the process easier to reproduce and explain. It also demonstrates that, given current model capabilities, a well-designed phased pipeline may outperform a highly autonomous agent. Its limitation lies in the fact that the fixed process makes strong assumptions about the task type, resulting in insufficient flexibility when facing problems requiring dynamic environmental exploration or repeated clarification of requirements.

AutoCodeRover combines problem description with program structure, gradually collecting code context related to defects by searching classes, methods, and call relationships, and then generating patches from the model and verifying them with test feedback [38]. Compared to directly putting a large number of files into the hints, this structured retrieval helps reduce irrelevant context, but the effect is still affected by the quality of the initial localization. OpenHands focuses more on scalable platform construction, providing common components such as code editing, terminal execution, web page interaction, and event flow, enabling different models and agent strategies to be implemented and compared in a unified environment [39]. Platformization lowers the threshold for system development, but also raises new engineering requirements such as environment isolation, access control, and repeatable execution.

Repository-level benchmarks characterize contextual issues from different perspectives. RepoBench breaks down capabilities into three tasks: cross-file code retrieval, code completion using given context, and an end-to-end pipeline connecting retrieval and completion. This allows it to distinguish whether failure stems from "not finding the relevant code" or "finding it but not knowing how to use it" [40]. CrossCodeEval uses static analysis to filter samples that require information from other files to complete correctly, and covers Python, Java, TypeScript, and C#. It measures a model's ability to utilize cross-file definitions and call relationships [41]. These two types of benchmarks are closer to the engineering environment than single-file function generation, but they still primarily focus on local completion rather than complete defect repair.

RepoCoder employs a search and generation iteration for repository-level completion: the system first queries the repository fragment based on the current code, generates preliminary results, and then uses the generated content to update the query and context, thereby alleviating the problem of missing key information in a single search [42]. LongCoder, on the other hand, is geared towards long-distance code dependencies, improving the completion capability across distant locations through long context modeling and filling tasks suitable for the code structure [43]. The former emphasizes "choosing the right context," while the latter emphasizes "accommodating and modeling longer contexts," which correspond to search enhancement and long context routes, respectively. Real-world projects usually require a combination of both: expanding the window cannot replace accurate retrieval, and retrieval cannot completely eliminate the need for cross-module consistency analysis.

Figure 3 and Table 3 summarize the paper reports of representative systems on SWE-bench Lite. The benchmark contains 300 real GitHub issues, and the metric is the percentage of tasks that are successfully solved and validated. The original AutoCodeRover paper reported 18% for SWE-agent, 19% for AutoCodeRover single attempt, and 22% for the version combined with spectrum-based fault location in the unified table [38]. Agentless subsequently reported 32% for the three-stage process [37]. The results show that structured location and controlled workflow can improve the resolution rate and reduce the call cost, but these figures are affected by model version, hints, number of runs and evaluation time, and should not be interpreted as permanent system rankings.

4.3 Security, Privacy and Intellectual Property

AI-generated code may reproduce insecure patterns present in the training data. Early security research on GitHub Copilot generated 1689 programs across various high-risk scenarios, of which approximately 40%

were found to contain vulnerabilities [44,45,46,47]. These results, from an earlier version of the tool, cannot directly represent all current systems, but they demonstrate that automatically generated code still requires security checks.

Figure 3: Solving Rates of Representative Methods on the Paper Reporting Task in SWE-bench Lite [38, 37]

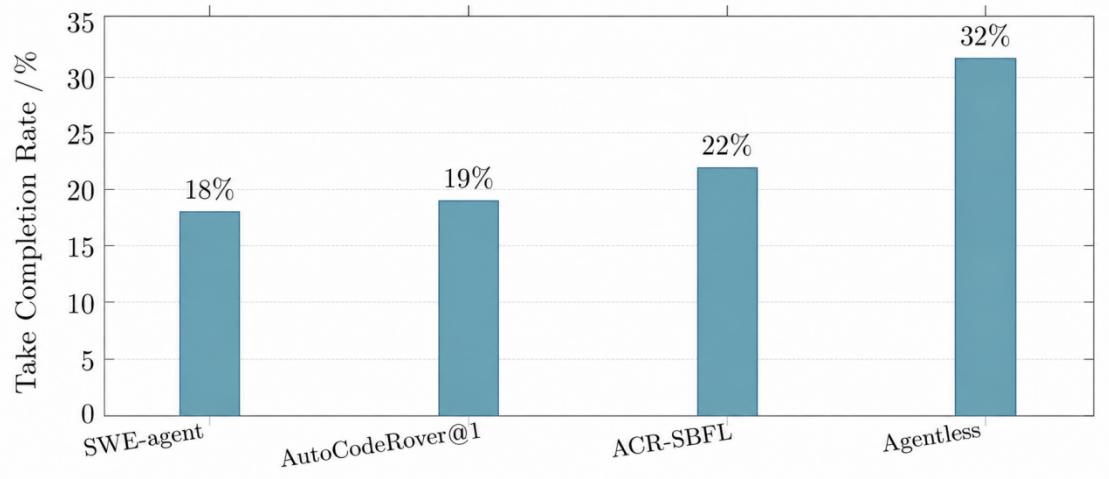


Table 3: Performance and Cost Statistics of Representative Methods on SWE-bench Lite

Method	Evaluation Setting	Number of Solved Tasks	Solve Rate	Average Cost Reported in the Paper
SWE-agent	pass@1	54/300	18.00%	≈ $2.51
AutoCodeRover	pass@1	57/300	19.00%	≈ $0.43
ACR-SBFL	pass@1 + fault localization	66/300	22.00%	≈ $0.47
Agentless	Multi-stage candidate filtering	96/300	32.00%	≈ $0.70

Note: The data are taken from the corresponding paper reports [38, 37]. The models, prompts, running budgets, and evaluation times used in different studies are not completely consistent; the costs are the values reported in the papers and are for reference only, and do not constitute a strict under the same conditions comparison.

Enterprise projects may also contain unpublished source code, access keys, user information, and business data. If developers directly send this content to external models, there is a risk of data breaches and the leakage of trade secrets. The risk further increases once programming agents can access terminals and modify files: erroneous operations could delete files, change configurations, or execute insecure commands. Therefore, the system needs to restrict access according to the principle of least privilege and require manual confirmation for high-risk operations.

Furthermore, there is still a lack of fully unified rules governing the relationship between training data sources, generated code, and open-source licenses. When the generated results are highly similar to existing open-source code, determining copyright and usage liability requires joint solutions from technology platforms, businesses, and legal systems.

4.4 Evaluation System and Experiment Reproduction

Traditional code generation research typically uses unit test pass rates to evaluate results, but programming agents also involve task planning, file retrieval, tool invocation, and human-computer interaction. Simply looking at whether the final program passes tests cannot determine whether the agent relies on a large number of accidental attempts, nor can it evaluate whether its modifications are concise, safe, and easy to maintain.

In addition to whether the final task is completed, the execution process should also be evaluated. One agent might accidentally pass a test after multiple undirected attempts, while another might only require minor code modifications through precise localization. Although both may ultimately score the same, the latter typically has lower costs and better maintainability. Evaluation metrics can further include the number of files read, the number of tool calls, the proportion of irrelevant modifications, the number of test executions, the number of human interventions, and the time and computing resources consumed to complete the task.

Evaluation data may also suffer from training contamination. If the model's training data includes code, problem descriptions, or solutions from benchmark projects, test scores may not fully represent its ability to solve new problems. Real-world open-source projects are constantly updated, and their dependencies change, making it difficult to rerun older tasks. Researchers need to record code repository versions, dependency versions, model versions, cue words, and runtime parameters, and publish execution logs as publicly as possible. Due to the randomness of model output, experiments should be run multiple times independently, reporting the average success rate and cost distribution, rather than simply selecting the best result.

Automated testing is suitable for determining whether a function meets specific conditions, but code readability, architectural rationality, and interpretation quality still require evaluation by professionals. A more reasonable system should combine automated testing, static analysis, manual code review, and user experience surveys.

Existing code capability benchmarks have gradually expanded from short function generation to data science, multi-language, and complex tool calls. HumanEval includes Python tasks with function signatures and natural language descriptions, and uses hidden unit tests to count pass@k. Its advantage is that the execution evaluation is clear, but the task size is small and the context is self-contained [1]. MBPP consists of more basic natural language programming problems, reference implementations, and tests, and is more suitable for examining common program structures [2]. APPS collects programming competition and online evaluation problems, covering complete program generation from beginner to competition difficulty, requiring the model to handle standard input and output, algorithm selection, and longer problem descriptions [48]. The three have jointly promoted execution-oriented evaluation, but still fall short of tasks requiring multi-file maintenance.

DS-1000 constructs tasks involving libraries such as NumPy, Pandas, Matplotlib, and Scikit-learn from real-world data science problems, and reduces the likelihood of hard-coded solutions passing tests through semantic testing and surface constraints [49]. MultiPL-E systematically translates the same set of natural language problems and tests into multiple programming languages to differentiate the algorithmic capabilities of models from the advantages of training data in specific languages [50]. EvalPlus addresses the insufficient test coverage of HumanEval and MBPP by automatically generating a large number of additional tests in combination with manual testing, revealing hidden errors in some programs that "pass the original tests" [51]. These studies demonstrate that the difficulty of a benchmark depends not only on the problem itself but also on whether the test is sufficient to identify false positives.

HumanEvalPack organizes code generation, repair, and interpretation tasks across multiple natural and programming languages to examine the model's cross-language generalization [52]. LiveCodeBench continuously builds time-controlled dynamic evaluations from newly released competition problems to reduce training data pollution and incorporates code generation, self-repair, execution prediction, and test generation into a unified framework [53]. BigCodeBench requires models to combine function calls across 139 libraries and multiple application domains to test their ability to understand complex instructions and correctly use tool interfaces [54]. However, even though these benchmarks are getting closer to real-world programming, they still struggle to cover long-term maintenance, team standards, and requirements negotiation, and therefore cannot completely replace manual acceptance testing in real-world projects.

4.5 Operating Costs, Environmental Differences, and Maintenance Burden

Programming agents consume computational resources during multi-round planning and tool calls. Complex tasks may require the model to repeatedly read large amounts of files, generate long contexts, and run tests multiple times, which is significantly more costly than ordinary code completion. If the test environment starts slowly or the project has many dependencies, waiting for the tool to return will also increase the completion time. Therefore, whether efficiency can be improved should not only be compared with manual coding time, but also the model call cost, test resource cost, and developer review time should be calculated.

Differences in runtime environments are also a common problem in practical applications. The same piece of code might run normally in the test environment provided by the agent, but fail in the developer's local or production environment. Operating system, dependency versions, hardware architecture, network permissions, and environment variables can all affect the execution result. If the agent modifies dependency files without proper documentation, it may cause other members of the project to be unable to run the program correctly.

The code generated by the AI agent can also create a maintenance burden. While rapidly generating large amounts of code in the short term can speed up prototype development, if the code lacks a consistent structure, comments, and tests, maintainers will need to invest more time in understanding and refactoring it, creating "AI-generated technical debt." Therefore, the scope of modifications should be controlled, the AI agent should be required to adhere to existing coding standards, and necessary testing and documentation should be considered prerequisites for task completion.

4.6 Human Capabilities, Organizational Governance, and Boundaries of Responsibility

Programming agents lower the barrier to entry for writing code, but they don't eliminate the need to understand software. Beginners may obtain working projects through natural language but lack understanding of code structure. When errors occur or features need to be added, they may still be unable to maintain the program independently. Professional developers, if overly reliant on model suggestions, may also reduce their scrutiny of code details. Especially when the model output is long and has a seemingly complete structure, developers are prone to trust bias, mistaking "written like correct code" for "validated."

Research on developer experience suggests that the value of AI programming tools cannot be measured solely by code acceptance rates. Observations, interviews, and experimental studies on GitHub Copilot have found that developers switch between different usage methods, such as "accelerators" and "exploration tools," while also undertaking the additional work of understanding, validating, and revising suggestions [55-60]. This echoes the contextual differences observed in productivity experiments [8,9].

Once programming agents are integrated into the enterprise development process, issues arise including who can use the agents, what data they can access, and how to trace responsibility after an incident. Organizations need to set permissions based on the sensitivity of the project. For routine tasks, agents can be allowed to read code and run local tests, while access to production databases, modification of deployment configurations, or release versions should be strictly restricted. The system should also save records of task objectives, plan changes, tool return results, code differences, and manual confirmations to enable subsequent audits to reconstruct the execution process.

A more realistic approach is not to allow artificial intelligence to completely replace programmers, but rather for humans to be responsible for defining goals, system architecture, risk boundaries, and final acceptance, while programming agents handle decomposable tasks such as information retrieval, initial implementation, test generation, and documentation. The agent can generate suggestions and execute actions, but the ultimate responsibility for software quality and security cannot be simply transferred to the model.

5. Future Development Trends

Programming agents will be further integrated with the complete development toolchain. Future systems will not only be able to invoke text search and terminals, but will also connect to requirements documents, code repositories, compilers, testing platforms, and deployment environments, enabling cross-tracking of requirements, code, tests, and modification records. When integrated with continuous integration processes, each modification can automatically execute unit tests, integration tests, code style checks, dependency and vulnerability scans, and performance tests, with the results uniformly fed back to the agent.

Enhanced retrieval and long-term memory will become crucial foundations for handling large-scale projects. Source code is not plain text. There are calls, inheritance, data flows, and dependencies between modules. Future systems can combine abstract syntax trees, call graphs, dependency graphs, and version history to construct project knowledge representations, enabling agents to retrieve code from the perspective of engineering structure rather than simple keywords. Enterprise-oriented systems should also establish hierarchical access control, local knowledge bases, and sensitive information filtering mechanisms to reduce data leakage.

The importance of code verification will gradually surpass that of simply generating code. Static analysis, unit testing, integration testing, formal verification, and manual code review can collectively form a multi-layered inspection mechanism. Agents should explain what was modified, why it was modified, and which modules might be affected, and request user confirmation before performing high-risk operations such as

deleting files, changing databases, or deploying code. The system also needs to set resource limits and stopping conditions to prevent agents from getting bogged down in repetitive modifications.

Future intelligent agents will need to express uncertainty more accurately. Current models often offer solutions in a definitive tone, even when the information is incomplete. A trustworthy agent should distinguish which conclusions come from project documents and which are general empirical inferences, and proactively inquire when key conditions are missing. The system can dynamically adjust its autonomy based on risk: low-risk format modifications and test execution can be automated, while high-risk data changes and production deployments must be approved manually.

Multi-agent collaboration is also a possible development direction. Different agents can be responsible for requirements analysis, system design, coding, testing, and security review, reducing the probability of a single model missing problems through role division. However, increasing the number of agents also brings communication costs and may lead to multiple agents accepting each other's incorrect conclusions. Therefore, multi-agent systems still require unified task management, conflict resolution, and result verification mechanisms.

Finally, software engineering education and industry governance need to be adjusted in tandem. Basic syntax and programming concepts remain important, but students should also learn to accurately describe requirements, read AI-generated code, design test cases, identify security issues, and use version control tools. Industry standards should cover training data documentation, code source citations, security testing, privacy protection, operation logging, and accountability. In the future, developers' core competency may gradually shift from "independently writing every line of code" to "organizing humans and intelligent agents to collaboratively complete reliable software systems."

5.1 Research Agenda and Evidence Gaps

For the aforementioned trends to translate into stable software engineering capabilities, it is still necessary to translate the macro-level direction into verifiable problems. Table 4 summarizes current common practices, evidence gaps, and suggested research paths for six key issues. As can be seen, the main obstacle at this stage is not just the insufficient model size, but also the lack of a complete chain of evidence among requirements, code, testing, human decision-making, and operation logs. Future research should report on both task success rate and process quality, and explain how the system allocates autonomy at different risk levels.

Table 4: Research Agenda and Key Evidence Gaps for Programming Agents

Key Issue	Current Mainstream Practices	Missing Key Evidence	Suggested Research Path
Requirements Understanding	Forming implementation plans through prompts, dialogue clarification, and task decomposition	Insufficient evaluation of ambiguous, conflicting, and continuously changing requirements	Construct interactive requirements benchmarks to evaluate clarification quality and requirements-to-code traceability
Repository Context	Using vector retrieval, keyword search, static analysis, or long-context methods to locate files	Limited evidence on the impact of cross-language, configuration, dependencies, and historical decisions	Integrate call graphs, data flows, and commit history, and separately report localization recall and patch correctness
Code Verification	Relying on unit tests, compilation feedback, and model self-reflection for iterative repair	Gaps remain between test passage and requirements satisfaction, security, and regression risks	Combine mutation testing, property-based testing, integration testing, static analysis, and necessary formal verification
Human-Machine Collaboration	Statistics on suggestion acceptance rates, task time, or subjective satisfaction	Lack of longitudinal evidence on long-term maintenance, team collaboration, review burden, and trust bias	Conduct cross-project longitudinal controlled studies recording types of human intervention, cognitive load, and subsequent defects
Security Governance	Adopting sandboxes, permission control, confirmation of sensitive operations, and logging	Lack of unified standards for threat models, log granularity, and responsibility allocation	Establish protocols for least privilege, provenance tracking, risk-tiered approval, and replayable auditing
Evaluation & Reproducibility	Reporting single-run experimental results using pass rates or task resolution rates	Model updates, randomness, environmental differences, and inference costs undermine comparability	Disclose versions, prompts, trajectories, and environments; run multiple times and report confidence intervals, costs, and failure types

5.2 Limitations

This paper also has several limitations. First, programming agents iterate very quickly, and model versions, tool permissions, and benchmark leaderboards will continue to change. The performance values presented in this paper should be understood as snapshots under specific time and experimental configurations. In addition, different studies use different task difficulty, testing environment, success criteria and cost standards. Therefore, the horizontal comparisons in this paper are mainly used to reveal the magnitude, trend and boundary, rather than to give an absolute ranking. Furthermore, the training data, prompting strategies, and operational trajectories of some commercial systems are not fully disclosed, making it difficult for published papers to adequately explain performance differences. Finally, this paper selects 60 representative and searchable articles for structured narrative analysis, rather than an exhaustive systematic review covering all relevant results. Several preprints with timely value are retained, and the conclusions still need to be verified by subsequent peer review and independent replication experiments. In the future, this study can be expanded into a retrievable dynamic review through continuously updated open literature repositories, unified coding tables, and executable experimental environments.

6. Conclusion

AI-assisted programming is evolving from code completion to programming agents that can understand tasks, invoke tools, and modify projects. Existing research demonstrates that these tools have significant value in software prototyping, repetitive code generation, testing, and defect localization. However, its efficiency gains are affected by the task size, familiarity with the project, quality requirements, and the cost of manual review.

Reliability, contextual understanding of large projects, security and privacy, evaluation criteria, operating costs, and human control remain the main limitations in current applications. The fact that a programming agent can generate code does not mean that it can independently assume responsibility for software quality and security. Real-world engineering requires not only that the program can run, but also that the system is maintainable, auditable, and conforms to organizational standards.

References

[1] Chen, M., Tworek, J., Jun, H., et al. (2021). Evaluating large language models trained on code. arXiv. https://arxiv.org/abs/2107.03374.

[2] Austin, J., Odena, A., Nye, M., et al. (2021). Program synthesis with large language models. arXiv. https://arxiv.org/abs/2108.07732.

[3] Hou, X., Zhao, Y., Liu, Y., et al. (2024). Large language models for software engineering: A systematic literature review. ACM Transactions on Software Engineering and Methodology, 33(8), 1–79.

[4] Jin, H., Huang, L., Cai, H., et al. (2024). From LLMs to LLM-based agents for software engineering: A survey of current, challenges and future. arXiv. https://arxiv.org/abs/2408.02479.

[5] Liu, J., Wang, K., Chen, Y., et al. (2024). Large language model-based agents for software engineering: A survey. arXiv. https://arxiv.org/abs/2409.02977.

[6] Wang, Y., Zhong, W., Huang, Y., et al. (2024). Agents in software engineering: Survey, landscape, and vision. arXiv. https://arxiv.org/abs/2409.09030.

[7] Jimenez, C. E., Yang, J., Wettig, A., et al. (2024). SWE-bench: Can language models resolve real-world GitHub issues? In International Conference on Learning Representations (ICLR).

[8] Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv. https://arxiv.org/abs/2302.06590.

[9] Becker, J., Rush, N., Barnes, E., & Rein, D. (2025). Measuring the impact of early-2025 AI on experienced open-source developer productivity. arXiv. https://arxiv.org/abs/2507.09089.

[10] Dong, Y., Jiang, X., Jin, Z., et al. (2025). A survey on code generation with LLM-based agents. arXiv. https://arxiv.org/abs/2508.00083.

[11] Wang, H., et al. (2025). AI agentic programming: A survey of techniques, challenges, and opportunities. arXiv. https://arxiv.org/abs/2508.11126.

[12] Feng, Z., Guo, D., Tang, D., et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. In Findings of the Association for Computational Linguistics: EMNLP 2020 (pp. 1536–1547).

[13] Guo, D., Ren, S., Lu, S., et al. (2021). GraphCodeBERT: Pre-training code representations with data flow. In International Conference on Learning Representations (ICLR).

[14] Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. (2021). Unified pre-training for program understanding and generation. In Proceedings of NAACL-HLT (pp. 2655–2668).

[15] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Proceedings of EMNLP (pp. 8696–8708).

[16] Nijkamp, E., Pang, B., Hayashi, H., et al. (2023). CodeGen: An open large language model for code with multi-turn program synthesis. In International Conference on Learning Representations (ICLR).

[17] Fried, D., Aghajanyan, A., Lin, J., et al. (2023). InCoder: A generative model for code infilling. In International Conference on Learning Representations (ICLR).

[18] Li, Y., Choi, D., Chung, J., et al. (2022). Competition-level code generation with AlphaCode. Science, 378(6624), 1092–1097.

[19] Wang, Y., Le, H., Gotmare, A., et al. (2023). CodeT5+: Open code large language models for code understanding and generation. In Proceedings of EMNLP (pp. 1069–1088).

[20] Rozière, B., Gehring, J., Goyal, F., et al. (2023). Code Llama: Open foundation models for code. arXiv. https://arxiv.org/abs/2308.12950.

[21] Li, R., Allal, L. B., Zi, Y., et al. (2023). StarCoder: May the source be with you! arXiv. https://arxiv.org/abs/2305.06161.

[22] Allal, L. B., Muennighoff, N., Kocetkov, D., et al. (2023). SantaCoder: Don't reach for the stars! arXiv. https://arxiv.org/abs/2301.03988.

[23] Lozhkov, A., Li, R., Allal, L. B., et al. (2024). StarCoder 2 and The Stack v2: The next generation. arXiv. https://arxiv.org/abs/2402.19173.

[24] Guo, D., Zhu, Q., Yang, D., et al. (2024). DeepSeek-Coder: When the large language model meets programming—The rise of code intelligence. arXiv. https://arxiv.org/abs/2401.14196.

[25] Yao, S., Zhao, J., Yu, D., et al. (2023). ReAct: Synergizing reasoning and acting in language models. In International Conference on Learning Representations (ICLR).

[26] Shinn, N., Cassano, F., Gopinath, A., et al. (2023). Reflexion: Language agents with verbal reinforcement learning. In Advances in Neural Information Processing Systems (Vol. 36, pp. 8634–8652).

[27] Yao, S., Yu, D., Zhao, J., et al. (2023). Tree of thoughts: Deliberate problem solving with large language models. In Advances in Neural Information Processing Systems (Vol. 36, pp. 11809–11822).

[28] Schick, T., Dwivedi-Yu, J., Dessì, R., et al. (2023). Toolformer: Language models can teach themselves to use tools. In Advances in Neural Information Processing Systems (Vol. 36, pp. 68539–68551).

[29] Parisi, A., Zhao, Y., & Fiedel, N. (2022). TALM: Tool augmented language models. arXiv. https://arxiv.org/abs/2205.12255.

[30] Qin, Y., Liang, S., Ye, Y., et al. (2024). ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In International Conference on Learning Representations (ICLR).

[31] Liu, X., Yu, H., Zhang, H., et al. (2024). AgentBench: Evaluating LLMs as agents. In International Conference on Learning Representations (ICLR).

[32] Wu, Q., Bansal, G., Zhang, J., et al. (2023). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv. https://arxiv.org/abs/2308.08155.

[33] Hong, S., Zhuge, M., Chen, J., et al. (2024). MetaGPT: Meta programming for multi-agent collaborative framework. In International Conference on Learning Representations (ICLR).

[34] Qian, C., Cong, X., Yang, C., et al. (2024). ChatDev: Communicative agents for software development. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (pp. 15174–15186).

[35] Li, G., Hammoud, H. A. A. K., Itani, H., et al. (2023). CAMEL: Communicative agents for mind exploration of large scale language model society. In Advances in Neural Information Processing Systems (Vol. 36, pp. 51991–52008).

[36] Yang, J., Jimenez, C. E., Wettig, A., et al. (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. arXiv. https://arxiv.org/abs/2405.15793.

[37] Xia, C. S., Deng, Y., Dunn, S., & Zhang, L. (2024). Agentless: Demystifying LLM-based software engineering agents. arXiv. https://arxiv.org/abs/2407.01489.

[38]Zhang, Y., Raghothaman, M., Liu, Y., et al. (2024). AutoCodeRover: Autonomous program improvement. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 1592–1604).

[39] Wang, X., Li, B., Song, Y., et al. (2024). OpenHands: An open platform for AI software developers as generalist agents. arXiv. https://arxiv.org/abs/2407.16741.

[40] Liu, T., Xu, C., & McAuley, J. (2023). RepoBench: Benchmarking repository-level code auto-completion systems. arXiv. https://arxiv.org/abs/2306.03091.

[41] Ding, Y., Wang, Z., Ahmad, W. U., et al. (2023). CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. arXiv. https://arxiv.org/abs/2310.11248.

[42] Zhang, F., Chen, B., Zhang, Y., et al. (2023). RepoCoder: Repository-level code completion with iterative retrieval and generation. In Proceedings of EMNLP (pp. 2471–2484).

[43] Guo, D., Xu, C., Du, N., et al. (2023). LongCoder: A long-range pre-trained language model for code completion. In Proceedings of the 40th International Conference on Machine Learning (pp. 12098–12107).

[44] Pearce, H., Ahmad, B., Tan, B., et al. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In 2022 IEEE Symposium on Security and Privacy (pp. 754–768).

[45] Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (pp. 2785–2799).

[46]Khoury, R., Avila, A. R., Brunelle, J., & Camara, B. M. (2023). How secure is code generated by ChatGPT? In 2023 IEEE International Conference on Systems, Man, and Cybernetics (pp. 2445–2451).

[47] Sandoval, G., Pearce, H., Noor, N., et al. (2023). Lost at C: A user study on the security implications of large language model code assistants. In 32nd USENIX Security Symposium (pp. 2205–2222).

[48]Hendrycks, D., Basart, S., Kadavath, S., et al. (2021). Measuring coding challenge competence with APPS. In Advances in Neural Information Processing Systems (Vol. 34, pp. 12674–12687).

[49] Lai, Y., Li, C., Wang, Y., et al. (2023). DS-1000: A natural and reliable benchmark for data science code generation. In Proceedings of the 40th International Conference on Machine Learning (pp. 18319–18345).

[50] Cassano, F., Gou, A., Nguyen, M., et al. (2023). MultiPL-E: A scalable and extensible approach to benchmarking neural code generation. IEEE Transactions on Software Engineering, 49(7), 3675–3691.

[51] Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In Advances in Neural Information Processing Systems (Vol. 36, pp. 21558–21572).

[52] Muennighoff, N., Liu, Q., Zhuo, T. Y., et al. (2023). OctoPack: Instruction tuning code large language models. arXiv. https://arxiv.org/abs/2308.07124.

[53] Jain, N., Han, K., Gu, A., et al. (2024). LiveCodeBench: Holistic and contamination-free evaluation of large language models for code. arXiv. https://arxiv.org/abs/2403.07974.

[54] Zhuo, T. Y., Vu, M. C., Chim, J., et al. (2024). BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. arXiv. https://arxiv.org/abs/2406.15877.

[55] Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In CHI Conference on Human Factors in Computing Systems Extended Abstracts (pp. 1–7).

[56] Barke, S., James, M. B., & Polikarpova, N. (2023). Grounded Copilot: How programmers interact with code-generating models. Proceedings of the ACM on Human-Computer Interaction, 7(CSCW1), 1–27.

[57] Ziegler, A., Kalliamvakou, E., Li, X. A., et al. (2022). Productivity assessment of neural code completion. arXiv. https://arxiv.org/abs/2205.06537.

[58] Nguyen, N., & Niu, S. (2022). An empirical study of GitHub Copilot's code suggestions. In Proceedings of the 19th International Conference on Mining Software Repositories (pp. 1–5).

[59] Dakhel, A. M., Majdinasab, V., Nikanjam, A., et al. (2023). GitHub Copilot AI pair programmer: Asset or liability? Journal of Systems and Software, 203, 111734.

[60] Yetiştiren, B., Özsoy, I., & Tüzün, E. (2022). Assessing the quality of GitHub Copilot's code generation. In Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering (pp. 62–71).

Funding

This research received no external funding.

Conflicts of Interest

The authors declare no conflict of interest.

Acknowledgment

This paper is an output of the science project.

Copyrights