Code in Harmony: Evaluating Multi-Agent Frameworks

Anonymous authors
Paper under double-blind review

Abstract

Multi-agent coding frameworks based on Large Language Models (LLMs) have become a promising approach to improving automated code generation. By simulating collaborative software development teams, these systems coordinate multiple specialized LLM agents, such as coders, testers, and planners, to divide work, check results, and refine code through multiple steps. This survey offers a detailed evaluation of leading multi-agent coding frameworks, including AgentCoder, CodeSIM, CodeCoR, and others, with a focus on their architecture, collaboration methods, and performance on common benchmarks such as HumanEval and MBPP. We identify important design features including role specialization, feedback mechanisms, and structured workflows that separate high-performing systems from weaker ones. Our analysis explores the balance between scalability, speed, and code quality, while also pointing out continuing challenges like communication overhead, reliability, and the ability to adapt to different programming domains. We end by discussing future directions for building multi-agent coding systems that are more efficient, better integrated with tools, and more flexible across tasks. This work aims to support the development of the next generation of AI-assisted programming systems by reviewing the current progress and highlighting practical lessons from recent frameworks.

1 Introduction

The success of Large Language Models (LLMs) in generating code has sparked interest in collaborative AI coding agents that can work together to tackle programming tasks beyond the capability of any single model. Single-agent LLM systems have demonstrated impressive code generation abilities, yet they often struggle with complex or open-ended software problems that require planning, error-checking, and iterative refinement.

In human software development, complex tasks are typically solved by a team with different experts, like designers, programmers, testers, contributing their specialized knowledge. This observation motivates the idea of multi-agent LLM coders, where multiple LLMs or multiple instances of an LLM adopt specialized roles and collaborate to generate, verify, and improve code.

Recent frameworks like MetaGPT Hong et al. (2023) and ChatDev Qian et al. (2024) pioneered this approach by simulating a team of AI developer agents working via natural language communication. However, early multi-agent systems faced notable limitations that spurred further research. For example, limited feedback and excessive agent counts hampered some initial frameworks: MetaGPT employed 5 agents and ChatDev 7 agents, yet their feedback loops were weak: MetaGPT's generated tests were only around 80% accurate on HumanEval Chen et al. (2021). The large number of agents also incurred huge token costs for inter-agent communication.

These issues highlighted that simply adding agents is not enough; effective collaboration mechanisms and efficient designs are crucial.

So in this survey, we focus on the multi-agent LLM frameworks for code generation, with an emphasis on performance-based comparison and design dissection. We review how these systems improve code correctness and analyze the architectural choices that lead to their success or failure. Standard programming benchmarks such as HumanEval Chen et al. (2021) and MBPP Austin et al. (2021) serve as our main yardsticks for performance. We also consider the frameworks' scalability and their generality towards larger software development tasks.

Contributions: This survey provides an academic overview and comparative analysis of multi-agent LLM coders. The key contributions are:

- **Classification by Performance:** We ranked multi-agent coding frameworks based on their performance on different benchmarks, and classified them into excellent vs. non-excellent categories. This performance-based classification highlights which approaches have truly advanced the state-of-the-art.
- **Design Feature Analysis:** We identify common features and design principles among the high-performing frameworks. By contrasting these with the typical traits of lower-performing systems, we pinpoint which design choices correlate with better outcomes.
- **Extraction of Unique Innovations:** We delve into the top-tier systems, notably AgentCoder Islam et al. (2024), CodeSIM Islam et al. (2025), and CodeCoR Pan et al. (2025), to explicate the novel techniques each introduces. These include innovative mechanisms that set them apart from prior work.

Through this survey, we aim to synthesize current knowledge on multi-agent LLM coding systems and provide insights that can drive the next generation of collaborative AI developers.

2 Background

2.1 LLMs for Code Generation: From Single to Multi-Agent

LLMs in code generation: In recent years, LLMs pre-trained on code and text have achieved remarkable results in automatically generating source code from natural language descriptions. Models like OpenAI's CodexOpenAI (2021) and Meta's CodeLlamaMeta (2023) have demonstrated the ability to produce correct solutions for programming problems in one or a few attemptsPan et al. (2025). These models leverage vast knowledge of syntax, libraries, and programming idioms learned from large code corpora. Despite their power, single LLM-based coders exhibit limitations on complex tasks: they may produce syntactically or logically incorrect code, miss edge casesChen et al. (2021), or require extensive prompt engineering to guide their reasoningKojima et al. (2022). To enhance single LLM performance, researchers devised prompting strategies like Chain-of-Thought (CoT) reasoningWei et al. (2022), self-debugging loopsChen et al. (2023), and tool-assisted refinementSchick et al. (2023). These techniques, while helpful, still involve a single model trying to handle all aspects of the task serially.

Rise of multi-agent paradigms: Motivated by the collaborative nature of human programming, researchers began exploring multi-agent frameworks where multiple LLMs interact to generate code. Instead of one model doing everything, different agents can take on distinct subtasks. For example, one agent writes an initial solution, another agent writes tests or checks the code, and a third agent suggests fixes. Early examples include ChatDev, which simulates a chat-based software company with agents taking roles like CEO, CTO, programmer, and tester, and MetaGPT, which encodes software engineering SOPs (standard operating procedures) into multi-agent task prompts. By dividing labor, multi-agent approaches aimed to overcome individual LLM limits through collaboration. Notably, even self-collaboration has been explored: Dong et al. (2024b) has one ChatGPT instance adopt multiple role personas and "talk to itself" following a software development methodology.

3 Design Principles in Multi-Agent LLM Coders

3.1 Agent Domain Specialization

In multi-agent LLM coding frameworks we usually witness frameworks utilizing different agents foe specialized roles. These agents are considered to be the domain experts and try to mimic a human software development team. Each agent focuses on a distinct aspect of automated code generation process - for example, MapCoder deploys four separate LLM agents devoted to retrieval of information, high-level planning, code generation and debugging. this structure helps in evaluating the full cycle of programming Islam et al. (2024). Similarly, AgentCoder introduces a team of specialized agents – a programmer, a unit test designer, and a test executor. These agents collaborate to produce correct code Huang et al. (2024). Other frameworks adopt a role-based division of labor. For example, ChatDev coordinates multiple specialized "software agents"—such as a reviewer focused on identifying bugs and a coder responsible for implementation—who collaborate to complete tasks efficiently Qian et al. (2024). These intentional specializations make use of each agent's unique strengths and have been shown to enhance the clarity and organization of the resulting code Hong et al. (2023). In fact, multi-agent models with well-chosen roles often outperform single-agent LLMs on standard coding benchmarks like HumanEval and MBPP, highlighting the advantages of distributing tasks among specialized agents Islam et al. (2024). Even though, working with multi-agents is an effective approach but, how do we calculate the optimal number of agents required for best performance. This suggests that while domain specialization is a valuable approach, further research is needed to determine the optimal "team composition" of LLM agents for coding tasks.

3.2 Communication Models

The way these specialized agents communicate plays a crucial role in the overall system design. Many frameworks employ natural language dialogues or messaging between agents to coordinate their actions. For example, ChatDev uses an instructor–assistant dual-agent dialogue: in each subtask, one agent gives instructions and the other responds, and they engage in multi-turn discussion until reaching a consensus on the solution Qian et al. (2024). Other systems pass information through structured channels – in AgentCoder, the programmer agent receives feedback from the test executor agent in the form of test results and error messages, rather than an open-ended chat Huang et al. (2024). A major challenge lies in maintaining clear and effective communication between agents. Simply chaining LLM outputs without structure can result in errors or misinterpretations due to hallucinations. Without clear constraints, agents can easily lose direction. ChatDev, for instance, has reported issues such as role confusion, duplicated efforts, and repetitive or unproductive exchanges, all of which can hinder progress. To overcome this, ChatDev uses an inception prompting strategy at the beginning of each dialogue to reinforce each agent's role, goals, and permissible interactions. This helps prevent role confusion and keeps the conversation focused on the task Qian et al. (2024). Similarly, MetaGPT encodes standard operating procedures (SOPs) into the prompts to structure agent interactions and verify intermediate results Hong et al. (2023). Good communication between agents can look different—sometimes it's open conversation, other times it's more structured. The goal is to find a balance: open chats can lead to new ideas, but simpler setups are easier to keep on track. Recent research agrees that setting clear communication rules and using turn-taking helps multi-agent systems work better without things getting messy or off track.

3.3 Workflow Structures

Beyond agent specialization and communication, the structure of the multi-agent workflow impacts the overall performance of the framework. Sequential pipelines inspired by real world software development projects are common. ChatDev implements a waterfall-like pipeline with distinct phases – design, coding, testing which are executed in order Qian et al. (2024). This kind of phase-structured workflow enforces a logical progression from requirements to final code and ensures that earlier steps (design decisions, etc.) inform later ones. On the other hand, some frameworks opt for iterative loops that repeatedly refine the solution. For instance, MapCoder performs iterative planning and debugging cycles. The planning agent outlines a solution, the coding agent produces code, and the debugging agent then verifies and prompts corrections, potentially looping back for re-planning if needed Islam et al. (2024). Such iterative workflows allow error correction and incremental improvement, closely mirroring how a human might debug a program in multiple passes. Moreover, they improve reliability by allowing errors to be caught and corrected early, often resulting in final answers that pass all tests on the first try, as shown by high pass@1 rates. However, there is evidence that beyond a certain point, additional self-debugging iterations yield diminishing returns or even regressions, partly due to accumulating noise in the feedback. PyCapsule deliberately uses only a two-agent pipeline (generator + executor) with a few specialized sub-modules, arguing that adding more agents can "overcrowd" the process for marginal gains Adnan et al. (2025). Thus, a key design question is how to structure the agent workflow to balance thoroughness and efficiency. In contrast, others like MetaGPT involve many agents in an assembly-line fashion to tackle complex tasks comprehensively. In practice, the optimal workflow likely depends on the task complexity. Current research is actively exploring this design space, and it remains an open problem how best to organize multi-agent coding workflows for both small-scale and large-scale programming tasks Rasheed et al. (2024).

3.4 Evaluation Dimensions

The effectiveness of multi-agent coder systems is mostly judged using coding benchmarks, where functional correctness is the key measure of success. Benchmarks like HumanEval and MBPP provide sets of programming problems with unit tests, and the goal is to generate code that passes all tests (usually measured by pass@1 or pass@k rates). Multi-agent frameworks are known to outperform single-agent frameworks on these metrics. MapCoder achieved new state-of-the-art pass@1 scores of 93.9% on HumanEval and 83.1% on MBPP, and AgentCoder pushed this further to 96.3% on HumanEval and 91.8% on MBPP using GPT-4 as the LLM Islam et al. (2024) Huang et al. (2024). These results significantly outperform earlier single-model methods, showing that collaborative and iterative problem-solving among multiple agents can more reliably produce correct solutions for challenging coding tasks. Besides raw accuracy, efficiency and cost have emerged as important evaluation dimensions. Multi-agent methods often require more prompt tokens and compute time due to their step-by-step deliberation. Early multi-agent solutions to HumanEval consumed over 138K tokens per problem to reach 90% accuracy, but AgentCoder cut this overhead by more than half (to ∼57K) while improving accuracy to over 96% Huang et al. (2024). Studies test multi-agent coders across diverse problem sets, benchmarks such as APPS Hendrycks et al. (2021) and CodeContests Li et al. (2022) (for competitive programming), HumanEval-ET and MBPP-ET (extended versions of the HumanEval and MBPP with additional tests)Dong et al. (2024a), and BigCodeBench have been used to assess performance on different languages and more complex or unseen scenarios Islam et al. (2024). A good multi-agent design should maintain high success rates across these varied tasks, indicating strong general problem-solving ability and not just overfitting to one benchmark. Finally, researchers consider failure cases as part of evaluation: examining where the multi-agent system falls short can reveal gaps in the design. Common failure modes include cases where agents misunderstand the problem or each other's outputs, or where the iterative loop gets stuck oscillating between a small set of errors. By analyzing these, recent papers identify directions for improvement – e.g., better error feedback mechanisms or more advanced planning to avoid blind spots Adnan et al. (2025) Zhong et al. (2024). Scaling these frameworks to more complex, real world programming tasks, and evaluating code quality remains a key challenge for future work.

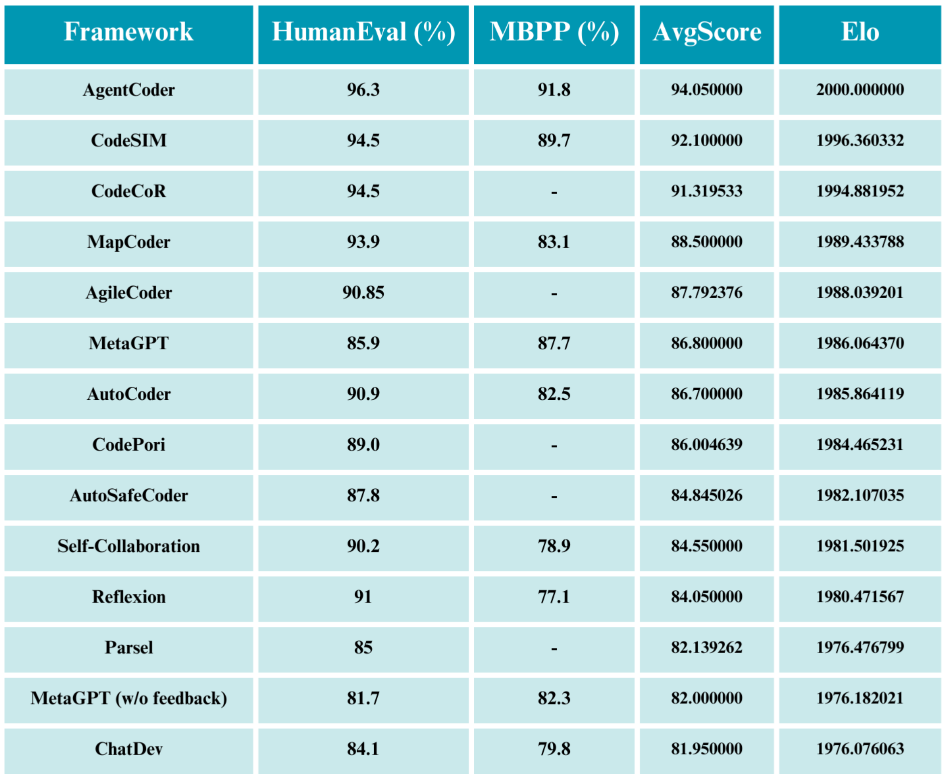
4 Ranking of Multi-Agent Frameworks

To systematically evaluate and compare multi-agent LLM coding frameworks, we adopt a pairwise Elo Rating approachWikipedia contributors (2025) based on performance on two standard benchmarks: HumanEval and MBPP. We choose these two benchmarks because we find that many recent studies still rely on the traditional benchmarks to evaluate code generation models. Also, we use Elo Rating in this context because it is particularly effective for handling incomplete benchmark coverage: all frameworks report results on HumanEval, but some do not provide scores on MBPP. Unlike averaging or threshold-based comparisons, Elo can accommodate such missing data gracefully by evaluating performance only on overlapping subsets.

Using this approach, we evaluate a total of 14 multi-agent code generation frameworks that have been tested with the same base LLM (GPT-4). The cumulative Elo scores across HumanEval and MBPP then yield a global ranking of all frameworks. Based on their aggregated Elo scores, we identify the top 3 performing systems and label them as "excellent frameworks". The remaining frameworks are classified as baseline or "non-excellent", serving as comparative anchors in subsequent analysis.

5 Comparative Feature Analysis

In the following, we analyze common characteristics that distinguish the highest performers from the rest, and discuss their effectiveness and limitations in the context of code-generation benchmarks.



Framework	HumanEval (%)	MBPP (%)	AvgScore	Elo
AgentCoder	96.3	91.8	94.050000	2000.000000
CodeSIM	94.5	89.7	92.100000	1996.360332
CodeCoR	94.5	-	91.319533	1994.881952
MapCoder	93.9	83.1	88.500000	1989.433788
AgileCoder	90.85	-	87.792376	1988.039201
MetaGPT	85.9	87.7	86.800000	1986.064370
AutoCoder	90.9	82.5	86.700000	1985.864119
CodePori	89.0	-	86.004639	1984.465231
AutoSafeCoder	87.8	-	84.845026	1982.107035
Self-Collaboration	90.2	78.9	84.550000	1981.501925
Reflexion	91	77.1	84.050000	1980.471567
Parsel	85	-	82.139262	1976.476799
MetaGPT (w/o feedback)	81.7	82.3	82.000000	1976.182021
ChatDev	84.1	79.8	81.950000	1976.076063

Figure 1: Illustrates the relative Elo scores of 14 multi-agent coding frameworks on the HumanEval and MBPP benchmarks, based on ChatGPT-4.0 model.

5.1 Common Features Among "Excellent" Frameworks

- The best-performing frameworks tend to have two things in common: clear role specialization and strong feedback loops. Together, these help them consistently score higher on benchmarks like HumanEval and MBPP, especially in terms of pass@1 accuracy. Nearly all top-tier methods incorporate an explicit testing or self-correction (debugging) loop to catch and fix errors. For example, AgentCoder employs a cyclic workflow where a "test executor" agent runs generated code and provides failure feedback to the "programmer" agent for refinement. This kind of automated debugging loop enables AgentCoder(GPT-4) achieves 96.3% and 91.8% pass@1 in HumanEval and MBPP datasets Huang et al. (2024). Similarly, CodeCoR and CodeSIM both generate new test cases to validate code and trigger repairs. CodeCoR's multi-agent self-reflection workflow produces multiple code solutions and prunes those that fail its generated tests, yielding a high average pass@1 score of 77.8% across HumanEval/MBPP variants on GPT-3.5 turbo Pan et al. (2025). CodeSIM takes this a step further by simulating how the code would run on example inputs. First, a planning agent lays out a step-by-step solution, then a debugging agent tests it using input-output simulations. This helps catch issues early and fine-tune the logic before generating the final code. This strategy achieved 94.5% HumanEval and 89.7% MBPP Pass@1 scores for ChatGPT-4 Islam et al. (2025).

- Another commonality is the use of chain-of-thought (CoT) reasoning and modular sub-tasks, often via domain specific agents. Top frameworks break the coding problem into parts handled by different

Framework	**Roles/Agents**	**Feedback Loop**	**Notable Techniques**
AgentCoder	3 (Coder, Test Designer, Test Executor)	Yes (iterative test execution)	Programmer refines code based on runtime feedback from test agent; specialized test generation to uncover errors Huang et al. (2024).
CodeSIM	3 (Planning, Coding, Debugging)	Yes (internal simulation & multi-turn debug)	Step-by-step plan verification via I/O simulation; chain-of-thought style planning before coding; self-debugging without external toolsIslam et al. (2025).
CodeCoR	4 (Prompt, Coding, Testing, Repair)	Yes (iterative repair cycle)	Generates multiple code outputs and tests them; prunes low-quality outputs; failed tests trigger automatic repair advice and code revision Pan et al. (2025).
MapCoder	4 (Retrieval, Planning, Coding, Debugging)	Yes (iterative debugging)	Retrieves past examples to guide solution; plans solution steps; executes and debugs code iteratively until tests pass Islam et al. (2024).
AgileCoder	3+ (Agile Team: e.g. Analyst, Coder, Tester)	Yes (multi-sprint refinement)	Adapts Agile methodology with iterative sprints; persistent code state via a dynamic code graph; static analysis of dependencies to assist refinement Nguyen et al. (2024).
MetaGPT	4 (Manager, Architect, Coder, Tester)	Limited (linear hand-off)	Encodes human SOPs in prompts; assembly-line waterfall workflow (design $\rightarrow$ code $\rightarrow$ review); domain-specific roles verify intermediate outputs qualitatively Hong et al. (2023).
AutoCoder	1 (LLM with built-in Interpreter)	Yes (self-check via execution)	Single-model trained with agent-generated dialogues and execution-verified data; can internally run code (including installing packages) to validate outputs Lei et al. (2024).
CodePori	5 (Design, Development, Review, Verification, Testing)	Yes (iterative until criteria met)	Full software development pipeline; each agent addresses a phase (including security checks); integrates results to produce running multi-file code Rasheed et al. (2024).
AutoSafeCoder	3 (Coding, Static Analyzer, Fuzz Tester)	Yes (security-driven iteration)	Iteratively generates code and scans for vulnerabilities; fuzz-testing agent executes mutated inputs to catch runtime errors and fix insecure code Nunez et al. (2024).
Self-Collab.	3 (Analyst, Coder, Tester)	Yes (multi-turn Q&A)	Single ChatGPT playing all roles in a virtual team; roles collaborate via prompts to analyze requirements, write code, and generate tests Dong et al. (2024b).
Reflexion	1 (Self-Reflective Agent)	Yes (trial-and-error learning)	Single agent with an episodic memory of mistakes; appends a self-critique after failures and iterates to avoid repeated errors Shinn et al. (2023).
Parsel	1 (Hierarchical Decomposer)	Yes (hierarchical testing)	Decomposes a complex problem into sub-functions; tests and integrates them into a final solution; uses auto-generated tests for each part Zelikman et al. (2023).
MetaGPT (w/o fb)	4 (Manager, Architect, Coder, Tester)	No (one-pass waterfall)	Original MetaGPT without feedback loops; agents interact in a fixed sequence without iteration, making it prone to cascading errors Hong et al. (2023).
ChatDev	3+ (e.g. CEO, CTO, Programmer)	No (one-pass waterfall)	Chat-based simulation of a software startup; natural language dialogue across roles; follows a fixed waterfall process without re-testing Qian et al. (2024).

Table 1: Summarizes key features of multi-agent LLM coding frameworks. Frameworks with strong HumanEval/MBPP performance tend to incorporate rigorous feedback/testing loops and clear role specialization.

"expert" roles, mirroring how human teams delegate tasks. For instance, MapCoder separates the concerns of planning, coding, and bug-fixing into distinct agents and then employs a retrieval agent to supply relevant code examples from a knowledge base. CodeSIM's planning agent and Parsel's algorithmic decomposition, both force the model to articulate a structured approach which reduces logical errors downstream Zelikman et al. (2023). In general, excellent frameworks tend to "look ahead" and "look back" during generation: they anticipate solution structure (via planning or examples) and later reflect on errors (via testing or self-critique), leading to much higher reliability on the benchmarks.

- Another shared feature of top frameworks is comprehensive role coverage of the software development cycle, ensuring no aspect of the task is neglected. The best-performing agents typically include at least a "coder" and a "tester" role and often additional roles for planning or reviewing. Self-Collaboration uses an analyst, coder, and tester in concert, to surpass its own single-agent performance by roughly 30–47% on pass@1 Dong et al. (2024b).

5.2 Features Typical of "Non-Excellent" Frameworks

In contrast, the lower-ranked frameworks usually lack one or more of the above strengths.

- Many "non-excellent" approaches follow a static, single-pass workflow (often a linear sequence of roles) without returning to fix mistakes detected post-hoc. MetaGPT (w/o feedback) and the original ChatDev both adopt a classic waterfall software development process where each agent hands off to the next stage without iteration. If a bug is introduced in an early phase, it persists to the end since there is no mechanism to revisit earlier decisions. For instance, MetaGPT using GPT-3.5 in a one-pass manner achieved only about 62.8% HumanEval pass@1 (and $\sim$74.7% on MBPP) which is significantly below frameworks that incorporate iterative debugging Hong et al. (2023). Similarly, ChatDev's agents lack automated test execution; they can discuss and jointly write code, but can not verify it against running test cases Qian et al. (2024).
- Furthermore, lower-ranked frameworks suffer from inefficiencies or coordination problems that indirectly harm their effectiveness. Having multiple agents communicate freely in natural language without a clear plan can lead to confusion and make it unclear who's responsible for what. Without a clear error-correction protocol, additional agents can even confuse the main coder (the LLM might receive conflicting signals). Non-excellent frameworks often did not perform thorough ablation to optimize each agent's role, so the extra layers end up adding complexity without much payoff. Oversight of such factors often leads to code that looks clean and has good comments, but it still misses the mark on tricky edge cases—which shows up in their lower Elo scores on benchmarks like HumanEval and MBPP 1.

5.3 Discussion

The comparative analysis reveals that the presence of feedback-based refinement is the single biggest differentiator between top-performers and others. Frameworks that systematically test and refine their code (either via generated unit tests, fuzzing, or self-reflection) achieve substantially higher reliability on HumanEval and MBPP. In contrast, one-pass methods often leave bugs in the solution, lowering their pass@1 scores.

Trade-off between specialization and generality: Multi-agent frameworks take inspiration from human software teams, but an LLM agent team is not a perfect analogy. In a human team, different members bring different skills and perspectives and by contrast, in an LLM-based team, all agents are backed by the same model (often with the same weights) and differ only by their prompts. Without clear structure, this can lead to duplicated effort or inconsistencies. The most effective frameworks address this by clearly defining roles, enabling agents to cross-check each other's outputs, and ensuring that feedback leads to meaningful changes. For example, Reflexion incorporates self-critiques into future attempts, and CodeCoR uses repair suggestions to guide code revisions Pan et al. (2025) Shinn et al. (2023).

Even the best frameworks have their limits. They often struggle with complex or domain-specific bugs that need deeper reasoning. Benchmarks usually involve straightforward tests, so if a problem requires a tricky algorithm or advanced math, feedback loops won't help much if the model just doesn't know how to solve it. Planning agents like those in CodeSIM or MapCoder can help to an extent, but tougher problems still pose a challenge. Another issue is that too many agents or iterations can lead to diminishing returns. Reflexion and AgentCoder both found that most of the progress happens in the first few tries, and then it starts to level off Huang et al. (2024). That's why many systems stop after a set number of rounds or once the tests pass. It's a practical fix, even if it's not a perfect one.

6 Unique Innovations in Top-Performing Frameworks

6.1 AgentCoder

AgentCoder Huang et al. (2024) presents a streamlined and effective multi-agent framework for code generation, with a strong focus on simplicity, efficiency, and reliability. Its main innovation lies in its minimal agent setup: rather than using a large number of agents with complex communication patterns, AgentCoder employs just three agents—a programmer agent, a test designer agent, and a test executor agent. This clear division of roles avoids unnecessary duplication and greatly reduces token usage, allowing the system to scale well while still achieving high accuracy.

A particularly notable feature of AgentCoder is its method for generating test cases. The test designer agent works independently of the programmer agent and does not have access to the code being tested. This separation ensures objective testing and improves the variety and quality of the test cases. By avoiding overlap between code generation and test creation, the system reduces confirmation bias and is better able to catch subtle errors that might be missed in more tightly coupled designs.

AgentCoder also includes a feedback loop grounded in real code execution. The test executor agent runs the code in a local environment and collects actual error messages, which are then used to help the programmer agent refine its output. This method relies on concrete runtime behavior rather than assumptions or heuristic filtering, making the revision process more precise and dependable.

Another strength of the framework is its ability to find correct solutions with relatively few iterations. Unlike systems that depend on long planning phases or repeated cycles of revision, AgentCoder often reaches a working solution within three rounds of interaction. This is made possible by its clear communication structure, which focuses on factual test results instead of abstract scores or evaluations.

The framework's design is also supported by strong empirical results. Ablation studies Huang et al. (2024) show that separating the roles of code generation and test creation leads to better test coverage and higher success rates. In addition, the modular architecture improves the system's transparency and makes it easier to adapt or extend in real-world applications.

In summary, AgentCoder introduces several key improvements—minimal agent roles, independent test generation, execution-based feedback, and efficient convergence—that together offer a simpler and more effective approach to multi-agent code generation. By focusing on clarity and grounded evaluation, it achieves strong performance without the high cost and complexity seen in many recent frameworks.

6.2 CodeSIM

CodeSIMIslam et al. (2025) introduces several unique techniques that distinguish it from other multi-agent code generation frameworks such as CodeCoR and AgentCoder. Architecturally, CodeSIM adopts a streamlined three-agent design composed of a Planning Agent, Coding Agent, and Debugging Agent. Unlike CodeCoR's four-agent setup and AgentCoder's test-centric configuration, CodeSIM's Planning Agent uniquely employs an exemplar-based approach, retrieving and adapting a similar previously solved problem to guide the current task. This enables CodeSIM to ground its solution process in a concrete and relevant reference, rather than relying on abstract prompt decomposition or separate test generation.

In terms of debugging, CodeSIM features an internal LLM-based mechanism that closely mimics human problem-solving behavior. Instead of relying on external test feedback as in CodeCoR or iterative test execution as in AgentCoder, CodeSIM's Debugging Agent performs step-by-step simulation of the code on failing inputs to identify and resolve errors. This self-contained debugging process avoids the overhead of generating new test cases and reduces reliance on tool-based runtime feedback, offering a more integrated and introspective error correction workflow.

Planning in CodeSIM is also uniquely simulation-driven. Before any code is generated, the Planning Agent verifies its plan by simulating I/O behavior to ensure logical correctness. If discrepancies arise, the plan is revised and revalidated. This contrasts with CodeCoR and AgentCoder, which do not incorporate an explicit plan validation step and instead begin code generation based on initial prompts or heuristics. As a result, CodeSIM reduces the risk of propagating flawed reasoning into later stages of code synthesis.

Finally, CodeSIM demonstrates improved efficiency and reduced token overhead by avoiding redundant generations and pruning processes. Its reliance on single-exemplar planning and internal debugging minimizes the number of LLM calls and avoids costly test generation and repair iterations seen in CodeCoR and AgentCoder. Empirical results in the source document confirm that CodeSIM achieves competitive or superior accuracy with fewer tokens, highlighting its effectiveness in both performance and computational efficiency.

6.3 CodeCoR

Code Collaboration and **R**epair is a self-reflective multi-agent framework which follows an innovative architecture. It explicitly evaluates the effectiveness and efficacy of each agent and their contribution in the code generation process. It employs four specialized agents for prompt analysis, code synthesis, test case generation, and repair advice. Unlike Other frameworks, each agent in CodeCoR produces multiple candidate outputs and prunes low-quality results. By introducing multiple solutions at each step and refining them via feedback (re-testing and repair) loop, CodeCoR makes the overall workflow more robust to any single agent's failure.

This design contrasts with AgentCoder's lean three-agent setup, which emphasizes independent test generation with minimal communication overhead. AgentCoder focuses on generating unbiased test cases separately from the code to catch bugs iteratively,whereas CodeCoR further boosts reliability by adding a dedicated repair agent and selecting the best among multiple solutions at each stage. CodeCoR also departs from CodeSIM's simulation-driven strategy. CodeSIM uses a planning agent and step-by-step input/output simulation to verify algorithm correctness and debug internally, while CodeCoR relies on executing generated test cases and using repair feedback to concretely validate the code.

Among the top three frameworks, CodeCoR is the only one that explicitly evaluates and compares the computational cost of its multi-agent setup against both single-agent and other multi-agent baselines, while maintaining strong performance. The authors quantitatively measure token usage, noting that CodeCoR, despite involving multiple agents and multiple output generations at each step, consumes fewer tokens on average. CodeCoR uniquely analyzes the tradefoff between accuracy and cost Pan et al. (2025).

7 Current Challenges and Future Directions

High Token Cost and Latency. Large agent groups such as MetaGPT and ChatDev introduce high communication costs, often exceeding $10 per HumanEval task, due to many serial messages being billed and processed Hong et al. (2023); Qian et al. (2024). Smaller teams, like the three-agent setup in AgentCoder, reduce token usage but may compromise on role specialization Huang et al. (2024). *Research direction:* design adaptive agent managers that activate roles and dialogue steps only when the task is predicted to be complex Du et al. (2024). Additionally, hybrid training methods, such as those used in AutoCoder, can embed reasoning from multiple roles into a single model Lei et al. (2024).

Error Propagation and Incomplete Test Coverage. If a weak tester or reviewer allows incorrect code to pass, later stages in the workflow may amplify the mistake Pan et al. (2025). Tests generated purely by language models often fail to cover unusual or edge cases Dong et al. (2024b). *Research direction:* combine the creativity of LLMs with traditional verification tools. For example, AutoSafeCoder uses GPT-4 alongside static analysis and fuzz testing Nunez et al. (2024), while CodeCoR generates formal specifications that can be verified through symbolic execution Pan et al. (2025).

Coordination Complexity. Unstructured conversations between agents can lead to conflicting results, while overly strict workflows may reduce flexibility and originality Hong et al. (2023). *Research direction:* simplify coordination by either fine-tuning a single model with integrated role behaviors Lei et al. (2024), or by designing streamlined hybrid protocols that combine MetaGPT-style role templates with feedback mechanisms similar to Reflexion Islam et al. (2024); Pan et al. (2025).

Scaling to Complex, Multi-File Codebases. Scalability becomes a challenge when working with large projects that span many files, classes, or API definitions as shown in CodePori Rasheed et al. (2024). *Research direction:* introduce persistent memory systems such as vector stores or knowledge graphs that store design choices, file relationships, and past decisions. These systems can help agents retrieve only the most relevant information for each task while keeping prompt lengths manageable.

Evaluation Limitations. Standard metrics like Pass@1 focus on whether code runs correctly for a single input but overlook broader software qualities like readability, performance, and security. *Research direction:* adopt project-level benchmarks such as ProjectEval, which assess entire codebases on documentation, code complexity, and resource usage Ishibashi & Nishimura (2024); Liu et al. (2024); Chen et al. (2021). Such metrics better reflect how well agents perform in real-world software engineering tasks.

8 Conclusion

Multi-agent LLM coders represent a significant evolution in AI-assisted software development. This survey reviewed their architectures, strengths, and weaknesses. Techniques such as iterative error correction, role decomposition, and feedback loops have advanced the state of the art, boosting pass@1 scores on HumanEval to over 90

Top systems like AgentCoder, CodeSIM, and CodeCoR demonstrate shared design principles including modular roles, strong feedback loops, and integrated verification. Yet challenges remain: high token and compute costs, instability in iteration, and difficulty generalizing to complex software projects.

The path forward includes self-optimizing agents, scalable hierarchies, deeper integration with software engineering tools, and robust verification systems. As these techniques mature, multi-agent LLM coders could evolve from sandbox problem solvers into full software engineering collaborators. In this future, human-AI teams may jointly design, develop, and maintain large-scale systems, leveraging the consistency and speed of AI with the creativity and oversight of human developers.

The groundwork has been laid; the coming years will determine how far multi-agent LLM coding systems can go in reshaping the landscape of automated software engineering.

References

Muntasir Adnan, Zhiwei Xu, and Carlos CN Kuhn. Large language model guided self-debugging code generation. *arXiv preprint arXiv:2502.02928*, 2025.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL `https://arxiv.org/abs/2108.07732`.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Xinyun Chen, Maxwell Lin Zhang, Bailin Wu, Mingyu Wang, Kenton Lee, Mengzhou Zhang, and Dale Schuurmans. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution, 2024a. URL https://arxiv.org/abs/2301.09043.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt, 2024b. URL https://arxiv.org/abs/2304.07590.

Zhuoyun Du, Chen Qian, Wei Liu, Zihao Xie, Yifei Wang, Yufan Dang, Weize Chen, and Cheng Yang. Multi-agent software development through cross-team collaboration, 2024. URL https://arxiv.org/abs/2406.08979.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021. URL https://arxiv.org/abs/2105.09938.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024. URL https://arxiv.org/abs/2312.13010.

Yoichi Ishibashi and Yoshimasa Nishimura. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization. *arXiv preprint arXiv:2404.02183*, 2024.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving, 2024. URL https://arxiv.org/abs/2405.11403.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Codesim: Multi-agent code generation and problem solving through simulation-driven planning and debugging, 2025. URL https://arxiv.org/abs/2502.05664.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.

Bin Lei, Yi Zhang, Shan Zuo, and Caiwen Ding. Autocoder: Enhancing code large language model with aiev-instruct. 2024.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey, 2024. URL https://arxiv.org/abs/2409.02977.

Meta. Codellama: Open foundation models for code. https://ai.facebook.com/blog/code-llama-large-language-model-coding/, 2023. Accessed: 2024-04-16.

Minh Huynh Nguyen, Thang Phan Chau, Phong X. Nguyen, and Nghi D. Q. Bui. Agilecoder: Dynamic collaborative agents for software development based on agile methodology, 2024. URL `https://arxiv.org/abs/2406.11912`.

Ana Nunez, Nafis Tanveer Islam, Sumit Kumar Jha, and Peyman Najafirad. Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing. *arXiv preprint arXiv:2409.10737*, 2024.

OpenAI. Codex. `https://openai.com/blog/openai-codex`, 2021. Accessed: 2024-04-16.

Ruwei Pan, Hongyu Zhang, and Chao Liu. Codecor: An llm-based self-reflective multi-agent framework for code generation. *arXiv preprint arXiv:2501.07811*, 2025.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development, 2024. URL `https://arxiv.org/abs/2307.07924`.

Zeeshan Rasheed, Malik Abdul Sami, Kai-Kristian Kemell, Muhammad Waseem, Mika Saari, Kari Systä, and Pekka Abrahamsson. Codepori: Large-scale system for autonomous software development using multi-agent technology, 2024. URL `https://arxiv.org/abs/2402.01411`.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36: 8634–8652, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Wikipedia contributors. Elo rating system — Wikipedia, the free encyclopedia, 2025. URL `https://en.wikipedia.org/w/index.php?title=Elo_rating_system&oldid=1282954176`. [Online; accessed 17-April-2025].

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. Parsel: Algorithmic reasoning with language models by composing decompositions. *Advances in Neural Information Processing Systems*, 36:31466–31523, 2023.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*, 2024.