ADBA Computer Science
Cutting-Edge Scientific Solutions

e-ISSN: 3023-8544
RESEARCH ARTICLE
Vol.3 / No.2 / 2026 / pp.72-80
https://doi.org/10.69882/adba.cs.2026071

Multi-Agent LLM Pipeline for Code Writing: An Experimental Study of Writer-Reviewer Architecture

Temel Kaan Ekiz [*,1] **and Mehmet Zeki Konyar** [α,2]
*Department of Software Engineering, Biruni University, Istanbul, Türkiye, αDepartment of Software Engineering, Kocaeli University, Kocaeli, Türkiye.

ABSTRACT In this paper, we develop a multi-agent pipeline-based approach for solving competitive programming problems via Large Language Models (LLMs). Specifically, we analyze the Writer-Reviewer pipeline where the Writer Agent produces Python solutions and the Reviewer Agent gives static natural language feedback. Experiments on a 96-problem AtCoder subset of LiveCodeBench involve comparing twelve pipeline setups which employ three different models (GPT-OSS-20B, Qwen3-Coder-30B-A3B-Instruct, and Qwen2.5-Coder-7B-Instruct) in various agent roles. Within this experimental setup, the review process increases the performance of GPT-OSS-20B from 87.5% to 91.7% Pass@1, with the bootstrap 95% confidence intervals overlapping, while the performance of Qwen3-Coder-30B-A3B-Instruct does not improve and the performance of Qwen2.5-Coder-7B-Instruct improves marginally from 0% to 1.0%. This may indicate that the static iterative feedback mechanism helps to further improve the performance of a strong Writer but not the performance of a weak Writer. A diagnostic audit suggests that the very low Qwen2.5-Coder-7B-Instruct scores mainly reflect structured-output compliance failures in our setup rather than standalone coding capability. Moreover, performance varies more when changing the Writer Agent than when changing the Reviewer Agent, with differences of up to 88.6 percentage points across Writers and up to 17.7 percentage points across Reviewers.

KEYWORDS
Large language models
Multi-Agent systems
Code generation
Code evaluation
LiveCodeBench
Competitive programming

INTRODUCTION

The process of writing code has evolved significantly over the past decades. Recent progress in Large Language Models (LLMs) has introduced a new form of code-generation assistance: models can now translate natural language problem descriptions into executable programs, often acting as automated collaborators within a broader human- or system-controlled workflow. This shift motivates careful evaluation of how LLMs should be organized when they are used for programming tasks.

Modern LLMs are typically built on the transformer architecture (Vaswani *et al.* 2017) and trained on large-scale text and code corpora. They have achieved strong results in general language tasks such as few-shot text generation and language understanding (Brown *et al.* 2020). They have also been investigated in specialized settings, including medical question answering and clinical-knowledge evaluation (Singhal *et al.* 2022), legal-assistant systems (Cui *et al.* 2024), educational support (Kasneci *et al.* 2023), and scientific workflows where their benefits must be balanced against reliability and misuse risks (Birhane *et al.* 2023).

Manuscript received: 2 May 2026,
Revised: 29 June 2026,
Accepted: 3 July 2026.

[1]tekiz@biruni.edu.tr (**Corresponding author**)
[2]mzeki.konyar@kocaeli.edu.tr

This broad range of applications has led to the development of LLMs specialized for different domains. While general-purpose models such as GPT-4 (OpenAI *et al.* 2024), Claude (Anthropic 2024), and Gemini (Gemini Team *et al.* 2023) demonstrate high performance across a wide variety of tasks, specialized models focusing on specific domains have also emerged. For code generation, CodeLlama (Rozière *et al.* 2024), StarCoder (Li *et al.* 2023), and DeepSeek-Coder (Guo *et al.* 2024); for mathematical reasoning, Llemma (Azerbayev *et al.* 2024) and DeepSeekMath (Shao *et al.* 2024); for medical applications, Med-PaLM (Singhal *et al.* 2022) and BioMistral (Labrak *et al.* 2024) exemplify these specialized models. This divergence raises the question of whether domain-specialized models or general-purpose models are more effective for specific tasks.

In the context of code generation, models trained or further tuned on large code corpora can transform natural language descriptions into executable code. The initial Codex model achieved 28.8% Pass@1 on the HumanEval benchmark (Chen *et al.* 2021), and subsequent systems have reported much higher scores on several coding benchmarks. In competitive programming, DeepMind's AlphaCode system reached an average ranking better than approximately 54% of participants in simulated Codeforces contests (Li *et al.* 2022). Developer adoption is also increasing: Stack Overflow's 2024 Developer Survey reports that 76% of all respon-

dents were using or planning to use AI tools in their development process (Stack Overflow 2024).

Various benchmarks have been developed to evaluate the code generation capabilities of LLMs. HumanEval (Chen *et al.* 2021), containing 164 hand-written programming problems and using the Pass@k metric, became an early standard benchmark. MBPP (Mostly Basic Programming Problems) (Austin *et al.* 2021) tests models' abilities to generate code from natural language with 974 entry-level Python tasks. For more challenging problems, APPS (Hendrycks *et al.* 2021) offers 10,000 competition-level programming problems. However, static benchmarks face important risks of data contamination and overfitting: benchmark problems may appear in training data, causing reported capabilities to appear higher than true generalization performance (Jain *et al.* 2024). LiveCodeBench (Jain *et al.* 2024) addresses this concern by continuously collecting new problems from LeetCode, AtCoder, and Codeforces and labeling each problem with its publication date. This design enables contamination-aware evaluation when model training cutoffs are known and respected. CodeElo (Quan *et al.* 2025) offers Elo ratings directly comparable to human participants on Codeforces problems.

Since this field is quite new, how to use LLMs most efficiently and accurately remains an active research topic. In single-model approaches, Self-Debugging (Chen *et al.* 2023) enables models to detect and correct their own errors, while Self-Refine (Madaan *et al.* 2023) and Reflexion (Shinn *et al.* 2023) offer iterative improvement mechanisms. Multi-agent systems have achieved particularly notable results in this area. ChatDev (Qian *et al.* 2024) and MetaGPT (Hong *et al.* 2024) have accomplished end-to-end software production by simulating the software development process with agents in different roles such as CEO, CTO, Programmer, and Tester. MapCoder (Islam *et al.* 2024), focused on competitive programming, achieved 93.9% Pass@1 success on HumanEval by mimicking the human programming cycle with retrieval, planning, coding, and debugging agents. AgentCoder (Huang *et al.* 2024a) reached 96.3% on HumanEval using an iterative testing and optimization approach with programmer, test designer, and test executor agents. Significant progress has also been made in debugging-focused studies. LDB (Zhong *et al.* 2024) performs error detection by verifying runtime execution step by step, while RGD (Jin *et al.* 2024) provides multi-stage improvement with guide, debug, and feedback agents. Self-Organized Agents (Ishibashi and Nishimura 2024) enables large-scale code generation with a hierarchical agent structure.

Although existing multi-agent studies have achieved impressive results, several practical questions remain insufficiently characterized. First, the relative importance of different agent roles (code writer versus reviewer) is often not isolated. Second, the relationship between the effectiveness of review mechanisms and the underlying model capability remains unclear—can a strong reviewer compensate for a weak writer? Third, many studies emphasize benchmarks such as HumanEval and MBPP, where high scores can make it difficult to distinguish system-level effects. Finally, the comparative behavior of general-purpose and code-specialized models in different multi-agent roles remains underexplored.

In this study, we investigate these questions by solving competitive programming problems using a Writer-Reviewer architecture. We compare a general-purpose model (GPT-OSS-20B) and two code-specialized models (Qwen3-Coder-30B-A3B-Instruct and Qwen2.5-Coder-7B-Instruct) in twelve different pipeline configurations on 96 AtCoder problems from LiveCodeBench. Our contributions are as follows:

- We present a simple Writer-Reviewer pipeline architecture and report that its best configuration achieves 91.7% Pass@1 on the evaluated LiveCodeBench AtCoder subset.
- We provide empirical evidence, within this limited setting, that Writer choice has a larger observed association with performance than Reviewer choice.
- We observe that static review provides uneven benefits across models: it improves GPT-OSS-20B in our experiments but provides little or no self-review benefit for the two Qwen configurations.
- We systematically compare the individual performances and cross-combinations of one general-purpose and two code-specialized models across twelve configurations, while explicitly noting that model scale, prompting, and provider-specific implementation details confound broad conclusions about model families.

RELATED WORK

Code Generation with Large Language Models

The field of LLMs code generation has witnessed remarkable progress since the introduction of Codex (Chen *et al.* 2021), which demonstrated that language models trained in code repositories could generate functional programs from natural language descriptions. Codex achieved 28.8% Pass@1 on the HumanEval benchmark, establishing both a baseline and a standardized evaluation methodology for the field.

Subsequent models have dramatically improved upon these initial results. CodeLlama (Rozière *et al.* 2024), built on the Llama foundation model and further trained on code-specific data, introduced specialized variants for different programming tasks, including completion and follow-up of instructions. StarCoder (Li *et al.* 2023) and its successor StarCoder2 (Lozhkov *et al.* 2024) demonstrated that training on carefully curated code datasets with appropriate licensing could achieve competitive performance while addressing data governance concerns. DeepSeek-Coder (Guo *et al.* 2024) pushed the boundaries further with models ranging from 1B to 33B parameters, achieving state-of-the-art results on multiple benchmarks through a combination of pre-training on 2 trillion tokens of code and fill-in-the-middle training objectives.

The emergence of reasoning-enhanced models has particularly impacted code generation. OpenAI's o1 model (OpenAI 2024) and DeepSeek-R1 (DeepSeek-AI 2025) incorporate chain-of-thought reasoning directly into the generation process, enabling more sophisticated problem decomposition and algorithm design. These models have shown especially strong performance on competitive programming tasks that require multi-step reasoning.

Benchmarks for Code Generation Evaluation

The evaluation of code generation capabilities has evolved alongside model development. HumanEval (Chen *et al.* 2021) introduced the Pass@k metric and provided 164 hand-crafted Python programming problems with unit tests. While influential, its limited size made it susceptible to overfitting and memorization. MBPP (Mostly Basic Programming Problems) (Austin *et al.* 2021) expanded the problem set to 974 entry-level Python tasks, providing broader coverage but maintaining a relatively low difficulty ceiling.

For evaluating advanced algorithmic capabilities, APPS (Hendrycks *et al.* 2021) compiled 10,000 problems from competitive

programming platforms spanning introductory to competition-level difficulty. CodeContests (Li *et al.* 2022) further contributed problems from Codeforces with comprehensive test suites designed to minimize false positives.

However, these static benchmarks face the fundamental challenge of data contamination. As LLMs are trained on increasingly large web corpora, the probability that benchmark problems appear in training data grows substantially, inflating reported performance metrics. LiveCodeBench (Jain *et al.* 2024) addresses this concern through continuous collection of new problems from LeetCode, AtCoder, and Codeforces, with each problem timestamped to enable contamination-aware evaluation. This temporal labeling allows researchers to evaluate models only on problems published after their training data cutoff. CodeElo (Quan *et al.* 2025) takes a different approach by providing Elo ratings directly comparable to human competitors on Codeforces, enabling more intuitive assessment of model capabilities relative to the human programmer population.

Self-Improvement and Iterative Refinement

Single-model approaches to improving code generation have explored various self-correction mechanisms. Self-Debugging (Chen *et al.* 2023) demonstrated that models could identify and fix errors in their own code when provided with execution feedback, achieving substantial improvements over single-pass generation. The key insight was that error messages and failed test cases provide valuable signals that models can interpret and act upon.

Self-Refine (Madaan *et al.* 2023) generalized this concept beyond code to arbitrary generation tasks, showing that iterative refinement based on self-generated feedback improves output quality across domains. Reflexion (Shinn *et al.* 2023) extended this further by maintaining an episodic memory of past attempts and their outcomes, enabling learning from failure within a single problem-solving session.

These approaches share a common limitation: they rely on a single model to both generate and evaluate solutions. This creates a potential ceiling effect where a model's evaluation capability is bounded by the same knowledge that constrains its generation capability, and prior work has cautioned that self-correction is not guaranteed across reasoning tasks (Huang *et al.* 2024b). Our work explores whether separating these roles across potentially different models can mitigate this limitation in code-generation pipelines.

Multi-Agent Systems for Software Development

Multi-agent architectures have emerged as a promising paradigm for complex software development tasks. ChatDev (Qian *et al.* 2024) pioneered the simulation of a software company with agents assuming roles such as CEO, CTO, programmer, reviewer, and tester. By decomposing the development process into phases, designing, coding, testing, and documenting, and assigning specialized agents to each phase, ChatDev demonstrated that role-based decomposition could produce complete software applications. MetaGPT (Hong *et al.* 2024) refined this approach by introducing Standardized Operating Procedures (SOPs) that structure agent interactions and outputs. By requiring agents to produce structured artifacts rather than free-form text, MetaGPT reduced miscommunication and cascading errors that plagued earlier multi-agent systems.

For competitive programming specifically, MapCoder (Islam *et al.* 2024) designed a four-agent pipeline mimicking the human programming cycle: retrieval of relevant examples, planning the algorithmic approach, coding the solution, and debugging based on test results. This architecture achieved 93.9% Pass@1 on HumanEval by leveraging the complementary strengths of each specialized agent. AgentCoder (Huang *et al.* 2024a) focused on the interplay between code generation and testing, employing separate agents for programming, test design, and test execution. The iterative refinement based on execution results pushed HumanEval performance to 96.3%, though this required significantly more inference iterations than simpler approaches.

Debugging and Error Correction Agents

Specialized debugging agents represent another strand of multi-agent research. LDB (Large Language Model Debugger) (Zhong *et al.* 2024) introduced block-level debugging that verifies runtime execution state at intermediate program points, enabling more precise localization of errors than end-to-end testing alone. RGD (Rubber Duck Debugging) (Jin *et al.* 2024) implemented a multi-stage pipeline with guide, debug, and feedback agents that collaborate to identify and resolve code issues. The guide agent provides high-level direction while the debug agent performs detailed code analysis, achieving improvements on complex debugging scenarios. Self-Organized Agents (Ishibashi and Nishimura 2024) scaled multi-agent systems to larger codebases through hierarchical organization, with high-level agents decomposing tasks and delegating to specialized sub-agents for implementation.

Research Gaps

Despite substantial progress, several questions remain underexplored in the literature. First, the relative importance of different agent roles has not been systematically quantified – is the code generator or the reviewer more critical to system performance? Second, most multi-agent studies focus on benchmarks like HumanEval where even simple approaches achieve high accuracy, leaving uncertain how these systems perform on more challenging competitive programming tasks. Third, the interaction between review mechanisms and base model capability has not been investigated, do weaker models benefit from review to the same degree as stronger models? Finally, while cross-model configurations have been explored informally, systematic comparison of general-purpose versus code-specialized models of varying scales in different pipeline roles is lacking.

Our work addresses these gaps through controlled experiments that vary Writer and Reviewer roles, evaluate on an AtCoder subset of LiveCodeBench, and compare model combinations across twelve pipeline configurations using three models of varying scales and specializations. The design isolates role choices within our implementation, but it does not eliminate all confounds, including model training cutoffs, provider-specific serving details, and prompt-template sensitivity.

METHODOLOGY

This section presents our multi-agent pipeline architecture for competitive programming, covering the system design, pipeline configurations, and experimental setup.

System Architecture and Agent Design

Our system implements a modular multi-agent architecture built on top of LangGraph, a framework for constructing stateful workflows and agents (LangChain 2025). The architecture follows a Writer-Reviewer paradigm where specialized agents collaborate through a shared state to iteratively improve code solutions. Figure 1 illustrates the high-level system architecture for single and multi-agent systems.

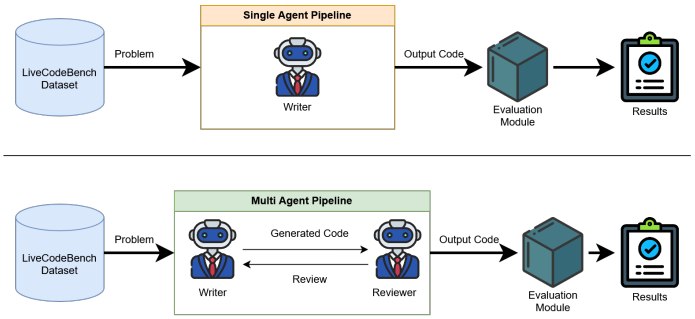

Figure 1 Overview of the single-agent and multi-agent pipeline architectures. In the single-agent configuration, the Writer directly generates a solution from the problem input. In the multi-agent configuration, the Writer and Reviewer engage in an iterative refinement loop before producing the final output. Both pipelines share the same input source and evaluation module.

The system consists of three main components: (1) a dataset loader that fetches problems from LiveCodeBench with temporal filtering capabilities, (2) a configurable multi-agent pipeline that processes each problem through a sequence of specialized agents, and (3) an evaluation module that executes generated code against public and private test cases.

All agents inherit from a shared abstraction that provides unified interfaces for prompt construction, LLM invocation, and response processing. Each agent maintains its own LLM client configured with model-specific parameters. A structured state dictionary flows through the pipeline and accumulates outputs from each processing stage. This state contains the following fields: a unique question identifier, the full problem statement text, the generated code produced by the Writer Agent, review feedback from the Reviewer Agent, debug information and optimization notes for future extensibility, an iteration counter tracking the current review cycle, and a history list storing the complete conversation log across iterations. This design allows each agent to access all previous agents' outputs and contribute its own findings to the shared context.

The **Writer Agent** generates Python code solutions from problem descriptions. It receives the problem statement and, in iterative configurations, feedback from previous review cycles. The agent's prompt (Figure 2) instructs it to produce complete, efficient Python code handling all edge cases, with specific formatting requirements for stdin-based and function-based problems. When feedback is available, it is injected into the prompt with instructions to address the identified issues. To reduce failures caused by ambiguous markdown formatting or mixed natural-language responses, the Writer Agent is instructed to return a single valid JSON object with a `code` field. The evaluation pipeline first attempts to parse the model response as JSON and extracts the generated program from this field. This structured-output strategy replaces reliance on markdown code-block extraction as the primary mechanism. If JSON parsing fails, the pipeline falls back to the previous markdown code-block extractor; if both strategies fail, the response is counted as an extraction failure.

The **Reviewer Agent** analyzes generated code for correctness, edge case handling, efficiency, and code quality through static analysis, without executing the code. Its prompt (Figure 3) instructs the model to evaluate the solution along four dimensions (correctness, edge cases, efficiency, code quality) and either issue an approval signal or list specific, actionable issues. The approval decision is

```
You are an expert Python programmer.
Write a complete Python solution for the following problem.

Problem:
{problem_statement}
{feedback_section}

Instructions:
- Output ONLY one valid JSON object.
- Do not include explanations or markdown.
- The JSON object must have exactly this structure:
  {"code": "<your complete Python solution code>"}
- The code field must contain ONLY Python code.
- Handle all edge cases mentioned.
- Ensure the code is efficient and correct.
- For stdin problems: implement a main() function and call it.
- For function problems: implement the Solution class with required methods.
```

Figure 2 Writer Agent prompt template. The {feedback_section} is populated with Reviewer feedback during iterative refinement cycles; it remains empty in writer-only configurations.

determined by checking for a predefined keyword in the response, which directly controls the loop termination condition. This simple matching approach makes the workflow deterministic, but it is also a limitation: keyword-based approval can be brittle, and future implementations should use structured outputs or schema validation to reduce false approvals and false rejections.

```
You are an expert code reviewer.
Review the following Python solution for
the given problem.

Problem:
{problem_statement}

Generated Code:
```python
{generated_code}
```

Review the code and provide feedback on:
1. Correctness: Does it solve the problem
   correctly?
2. Edge Cases: Are all edge cases handled?
3. Efficiency: Is the algorithm efficient
   enough?
4. Code Quality: Is the code clean and
   readable?

If the code looks good, respond with:
"APPROVED: Code is correct and ready."

If there are issues, list them clearly
with specific suggestions.
```

Figure 3 Reviewer Agent prompt template. Approval by the agent triggers loop termination.

Pipeline Configurations and Models

We implement a flexible pipeline factory supporting multiple configurations through a declarative specification structure. Each configuration defines the agent sequence, an optional description, whether the pipeline includes a review loop, which agent to loop back to, the agent responsible for the loop condition, the maximum iteration count, and a mapping of agent roles to specific models. Table 1 summarizes the twelve configurations evaluated in this study, spanning three writer-only baselines and nine review loop variants with different model combinations.

We evaluate three language models accessed through the Nebius API infrastructure. **GPT-OSS-20B** is an open-weight Mixture-of-Experts model with approximately 20.9B total parameters and 3.6B active parameters per forward pass (OpenAI 2025). **Qwen3-Coder-30B-A3B-Instruct** (hereafter **Qwen3-Coder**) is a code-specialized model from Alibaba's Qwen family with 30.5B

total parameters and 3.3B activated parameters (Qwen Team 2025). **Qwen2.5-Coder-7B-Instruct** (hereafter **Qwen2.5-Coder**) is a smaller code-specialized model from the Qwen family with 7.61B parameters (Qwen Team 2024; Hui *et al.* 2024). All three models are configured with identical inference parameters (Table 2): temperature 0.1 to encourage deterministic outputs appropriate for competitive programming, maximum token limit of 8,192, and a request timeout of 180 seconds. Using identical parameters improves experimental control, but these settings may not be optimal for every model and should be considered a design choice rather than a claim of model-level optimality.

Table 1 Pipeline configurations with writer and reviewer agents

Pipeline Name	Writer Agent	Reviewer Agent
`writer_oss`	GPT-OSS-20B	–
`writer_qwen3`	Qwen3-Coder	–
`writer_qwen2.5`	Qwen2.5-Coder	–
`review_loop_oss`	GPT-OSS-20B	GPT-OSS-20B
`review_loop_qwen3`	Qwen3-Coder	Qwen3-Coder
`review_loop_qwen2.5`	Qwen2.5-Coder	Qwen2.5-Coder
`review_loop_exp1`	Qwen3-Coder	GPT-OSS-20B
`review_loop_exp2`	GPT-OSS-20B	Qwen3-Coder
`review_loop_exp3`	Qwen2.5-Coder	GPT-OSS-20B
`review_loop_exp4`	GPT-OSS-20B	Qwen2.5-Coder
`review_loop_exp5`	Qwen2.5-Coder	Qwen3-Coder
`review_loop_exp6`	Qwen3-Coder	Qwen2.5-Coder

Table 2 Model inference parameters

Parameter	Value
Temperature	0.1
Max Tokens	8,192
Request Timeout	180 seconds

The review loop pipeline implements an iterative refinement process (Figure 4). After the Writer generates initial code, the Reviewer evaluates it. If issues are identified, feedback is passed back to the Writer for revision. This cycle continues until the Reviewer approves the code or the maximum iteration limit is reached. The termination logic is implemented as a conditional edge in the LangGraph workflow: the loop ends when the iteration count reaches the maximum or the Reviewer issues an approval signal; otherwise, execution returns to the Writer Agent with the accumulated feedback. We set the limit to 2 in order to balance improvement potential against computational cost, as additional iterations are expected to yield diminishing returns given the Writer Agent's bounded capability to implement feedback.

Dataset, Evaluation, and Experimental Procedure

We use the LiveCodeBench code generation lite dataset (release v5). LiveCodeBench is designed to support contamination-aware evaluation through continuous collection of new problems from competitive programming platforms and release-date filtering (Jain *et al.* 2024). We filter for AtCoder problems published between November 1, 2023 and February 1, 2025. After filtering and excluding 16 blacklisted problems with known test case formatting issues, our evaluation set consists of 96 problems. Because GPT-OSS-20B has a reported knowledge cutoff of June 2024 (OpenAI 2025) and the exact training cutoffs of the two Qwen models are not fully documented in the model cards used here, we do not claim that this experimental subset is strictly contamination-free for all evaluated models. Instead, we treat the dataset as a temporally labeled and contamination-aware benchmark subset.

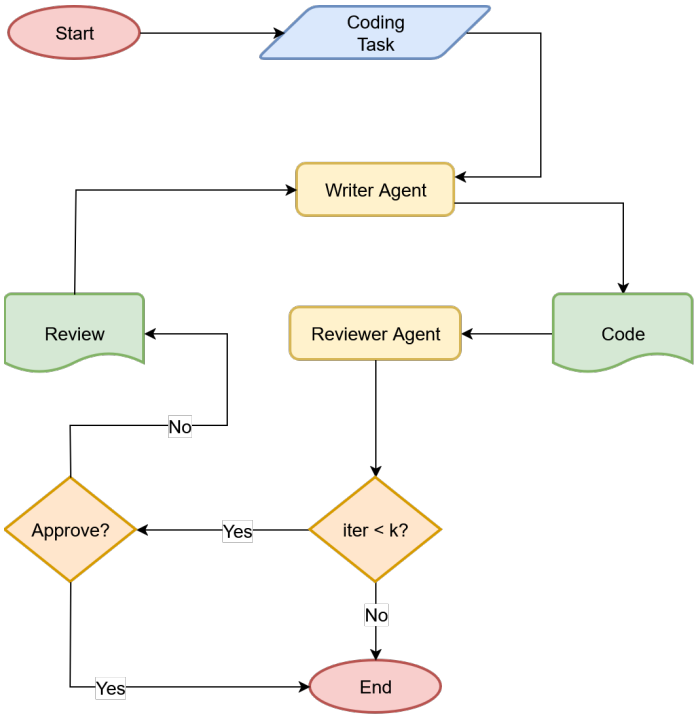

Figure 4 Review loop pipeline flow diagram. The Writer generates code, the Reviewer provides feedback or approval. The loop continues until approval or maximum iterations is reached. State information flows through the shared state structure.

Generated code is executed in a sandboxed subprocess environment with a 6 second timeout per test case. The implementation pre-imports standard library modules, which reduces failures due only to missing imports but differs from a strict online-judge submission environment. For each problem, evaluation follows a fail-fast approach consistent with competitive programming judge behavior: the code is executed with test input provided by the dataset, the output is compared against expected output after whitespace normalization, and execution stops at the first failed test case. We report Pass@1, defined here as the percentage of problems for which one run of a pipeline configuration produces code that passes all test cases:

$$\text{Pass@1} = \frac{\left|\{p \in \text{Problems} : \text{all\_tests\_pass}(s_p)\}\right|}{|\text{Problems}|} \times 100\% \quad (1)$$

This strict metric reflects the binary nature of competitive programming evaluation, where partial solutions receive no credit. Since each configuration is evaluated on 96 problems without repeated stochastic runs, small percentage-point differences should be interpreted cautiously. To quantify uncertainty due to the finite size of the evaluation subset, we also compute 95% non-parametric bootstrap confidence intervals for Pass@1. For each pipeline configuration, we resample the 96 binary problem-level correctness outcomes with replacement for 10,000 bootstrap iterations and report the 2.5th and 97.5th percentiles of the resulting Pass@1 distribution.

The bootstrap random seed is fixed to 12345 for reproducibility. These intervals estimate uncertainty due to the sampled problem set; they do not capture stochastic variation across repeated model generations because each configuration was evaluated with a single run. Therefore, we report confidence intervals as descriptive uncertainty estimates rather than as pairwise statistical significance tests.

For each pipeline configuration, we load and filter the dataset, initialize the agent state for each problem, execute the pipeline, extract the generated code from the final state, and log execution details including prompts, responses, and timing. Transient API or infrastructure failures are retried up to 2 times. This retry policy is intended to handle request-level failures rather than to sample multiple independent solutions; model-generated incorrect code is evaluated as produced.

RESULTS

Overall Performance and Effect of Review Loop

Figure 5 and Table 3 present Pass@1 scores across all twelve pipeline configurations. The best observed performance of 91.7% is achieved by two GPT-OSS-20B-as-Writer configurations: GPT-OSS-20B self-review and GPT-OSS-20B paired with Qwen3-Coder as Reviewer, each correctly solving 88 out of 96 problems. GPT-OSS-20B-as-Writer configurations occupy the top four positions. Because the evaluation set contains 96 problems, a 1.0 percentage-point change corresponds to roughly one problem, and the 4.2 percentage-point improvement of GPT-OSS-20B with self-review corresponds to four additional solved problems.

Table 3 Pass@1 Results with 95% Bootstrap Confidence Intervals

Pipeline	Solved	Pass@1 % (95% CI)
`writer_oss`	84/96	87.5 [80.2, 93.8]
`writer_qwen3`	56/96	58.3 [49.0, 68.8]
`writer_qwen2.5`	0/96	0.0 [0.0, 0.0]
`review_loop_oss`	88/96	91.7 [85.4, 96.9]
`review_loop_qwen3`	56/96	58.3 [49.0, 67.7]
`review_loop_qwen2.5`	1/96	1.0 [0.0, 3.1]
`review_loop_exp1`	72/96	75.0 [66.7, 83.3]
`review_loop_exp2`	88/96	91.7 [85.4, 96.9]
`review_loop_exp3`	13/96	13.5 [7.3, 20.8]
`review_loop_exp4`	86/96	89.6 [83.3, 95.8]
`review_loop_exp5`	5/96	5.2 [1.0, 10.4]
`review_loop_exp6`	55/96	57.3 [47.9, 67.7]

Note on Qwen2.5-Coder results. To better understand the unusually low Qwen2.5-Coder scores, we performed a diagnostic audit of its writer-only outputs. Table 4 summarizes the first generated output for each of the 96 evaluated problems. Among these 96 first outputs, 94 returned the literal placeholder text in the `code` field, 2 echoed the prompt or problem statement, and none produced executable Python-like code. Therefore, the very low Qwen2.5-Coder scores should be interpreted primarily as failures under our specific prompt, serving stack, and structured extraction protocol, rather than as a general measure of the model's standalone coding capability.

The review mechanism produces an asymmetric effect depending on the base model. For GPT-OSS-20B, adding a Reviewer Agent improves Pass@1 from 87.5% to 91.7%, a gain of 4.2 percentage points. However, the corresponding 95% bootstrap confidence

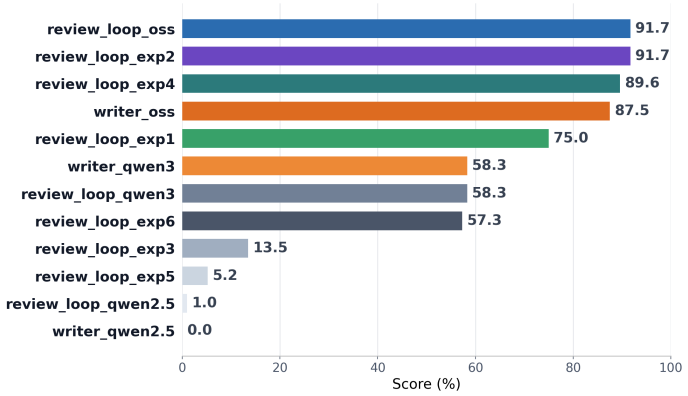

Figure 5 Pass@1 performance across all twelve pipeline configurations, sorted by score.

Table 4 Diagnostic Output-Format audit for Qwen2.5-Coder writer-Only outputs

Output category	Count	Percentage
Placeholder returned in `code` field	94/96	97.9%
Prompt/problem statement echo	2/96	2.1%
Executable Python-like code obtained	0/96	0.0%

intervals overlap, so this improvement should be interpreted as a descriptive trend rather than as evidence of statistical significance. In contrast, Qwen3-Coder shows no observed self-review benefit: performance remains at 58.3% with and without the review loop. Qwen2.5-Coder improves only marginally from 0% to 1.0% with self-review. Pairing Qwen2.5-Coder as Writer with GPT-OSS-20B as Reviewer yields 13.5%, which is higher than Qwen2.5-Coder alone but still far below configurations with stronger Writers. This suggests that static review feedback may require a minimum level of Writer capability to be useful; however, without a failure taxonomy, we cannot fully separate model capability limits from prompt-format, code-extraction, or serving-template effects.

Writer Agent vs. Reviewer Agent Quality

Table 5 presents the full Writer-Reviewer interaction matrix, enabling direct comparison across all combinations. Holding the Reviewer constant and varying the Writer reveals dramatic performance gaps. With GPT-OSS-20B as Reviewer, the gap between the best Writer (GPT-OSS-20B, 91.7%) and the worst Writer (Qwen2.5-Coder, 13.5%) is 78.2 pp. With Qwen2.5-Coder as Reviewer, the gap spans from 89.6% to 1.0%, a difference of 88.6 pp. With Qwen3-Coder as Reviewer, the gap ranges from 91.7% to 5.2%, yielding 86.5 pp. In the opposite direction—holding the Writer constant and varying the Reviewer—the gaps are consistently smaller: only 2.1 pp when GPT-OSS-20B writes (ranging from 89.6% to 91.7%), 17.7 pp when Qwen3-Coder writes (ranging from 57.3% to 75.0%), and 12.5 pp when Qwen2.5-Coder writes (ranging from 1.0% to 13.5%).

These observations indicate that, in this implementation and dataset subset, generation capability is more strongly associated with performance than review capability. A strong Writer paired with the weakest Reviewer (GPT-OSS-20B + Qwen2.5-Coder, 89.6%) substantially outperforms the reverse configuration

Table 5 Pass@1 (%) Matrix: Writer Agent vs. Reviewer Agent

Writer Agent	Reviewer Agent			
	None	GPT-OSS-20B	Qwen3-Coder	Qwen2.5-Coder
GPT-OSS-20B	87.5	91.7	91.7	89.6
Qwen3-Coder	58.3	75.0	58.3	57.3
Qwen2.5-Coder	0.0	13.5	5.2	1.0

(Qwen2.5-Coder + GPT-OSS-20B, 13.5%). The writer-only baselines further support this interpretation: GPT-OSS-20B achieves 87.5%, Qwen3-Coder 58.3%, and Qwen2.5-Coder 0%. However, these results should not be interpreted as a general proof that general-purpose models outperform code-specialized models, since our model selection is limited and the comparison is confounded by model scale, training data, serving stack, and prompt compatibility.

DISCUSSION

This section interprets our experimental findings and explores their implications for multi-agent system design.

Why Does Review Help Strong Models but Not Weak Ones?

We hypothesize that this asymmetry may stem from the nature of errors each model produces. GPT-OSS-20B, with its stronger baseline, may make more "correctable" errors—for example, off-by-one mistakes, missing edge cases, or suboptimal but fundamentally sound algorithmic approaches. In such cases, a Reviewer can identify issues and the Writer may possess sufficient capability to address them.

In contrast, the lower baselines of Qwen3-Coder and Qwen2.5-Coder suggest that some failures may involve more fundamental algorithmic or formatting errors. When a solution's core approach is wrong, incremental feedback such as "check edge cases" may not be sufficient: the Writer would need to reconceptualize the entire approach, which natural language feedback alone may not effectively guide. This interpretation is plausible but not proven by the current aggregate results. The very low Qwen2.5-Coder writer-only score also warrants particular caution. A diagnostic audit indicates that Qwen2.5-Coder frequently failed to follow the required structured-output instruction under our prompt and serving setup. Therefore, this result should be interpreted primarily as reflecting prompt-following and output-format compliance problems in our pipeline, rather than as a standalone benchmark of the model's coding ability.

Our cross-model experiments are consistent with this interpretation: when GPT-OSS-20B serves as Reviewer for Qwen3-Coder, performance improves from 58.3% to 75.0%, and for Qwen2.5-Coder, from 0% writer-only to 13.5%. Better feedback quality appears to help in some pairings, but the final result remains strongly constrained by the Writer.

Practical Implications for System Design

The Writer choice is associated with up to 88.6 pp of observed performance variation, exceeding the largest observed Reviewer-side range of 17.7 pp. With a fixed computational budget, our results suggest that allocating the strongest available model to the Writer role is likely to yield higher returns than allocating it only to review.

In these experiments, self-review benefits appear only for the strongest baseline model. Qwen2.5-Coder (0% baseline) and Qwen3-Coder (58.3% baseline) show negligible or no improvement from self-review. This pattern suggests that static review loops should be applied selectively, but the present study does not establish a universal numerical capability threshold.

A strong Writer paired with a heterogeneous Reviewer can match or approach the best observed results in this study: GPT-OSS-20B with Qwen3-Coder as Reviewer reaches 91.7%, while GPT-OSS-20B with Qwen2.5-Coder as Reviewer reaches 89.6%. This suggests a possible cost-performance tradeoff, although we did not measure token usage, latency, or API cost directly.

The maximum iteration limit of the Reviewer agent is 2 in our implementation. This value was chosen to control computational cost; the present experiments do not test larger iteration budgets, so we cannot conclude that two iterations are generally optimal.

In our experiments, the general-purpose GPT-OSS-20B outperformed both code-specialized models, Qwen3-Coder by 29.2 pp and Qwen2.5-Coder by 87.5 pp in writer-only mode. However, this performance gap may stem from differences in base reasoning capacity, model scale, serving configuration, or prompt compatibility rather than from the general-purpose versus code-specialized distinction itself. Stronger code-specialized models such as larger Qwen-Coder or DeepSeek-Coder variants might narrow or reverse this gap. We therefore refrain from concluding that general-purpose models are inherently superior for algorithmic tasks.

LIMITATIONS AND FUTURE WORK

Several limitations affect the generalizability of our findings. We evaluated only three models (GPT-OSS-20B, Qwen3-Coder, and Qwen2.5-Coder); results may differ for dense architectures, frontier proprietary systems, or other model families such as Claude, Gemini, DeepSeek, or Llama. In particular, our comparison between general-purpose and code-specialized models is confounded by differences in model scale, training data, release date, serving stack, and base reasoning capability. Including stronger code-specialized models would be necessary to disentangle these factors. Our evaluation covered 96 AtCoder problems, which emphasize algorithmic reasoning and may not represent all competitive programming domains. Because our evaluation uses a limited model set, a single platform subset, and descriptive bootstrap uncertainty estimates rather than pairwise significance tests, the conclusions should be interpreted as empirical trends rather than universal claims about all multi-agent code-generation systems. We report 95% bootstrap confidence intervals over the 96 problem-level correctness outcomes to quantify uncertainty due to the finite evaluation subset, but these intervals do not establish that small differences, such as the 4.2 pp GPT-OSS-20B self-review gain, are statistically significant. While GPT-OSS-20B has an official knowledge cutoff of June 2024, Qwen3-Coder's and Qwen2.5-Coder's exact training cutoffs are not fully documented in the model cards used here, introducing uncertainty about potential data contamination.

These limitations suggest several promising research directions. Expanding to additional models and scaling to larger problem sets across LeetCode, Codeforces, and AtCoder would improve both model coverage and statistical reliability. Difficulty-stratified analysis using AtCoder's A–F ratings could reveal whether the review mechanism's effectiveness varies with problem complexity. Systematic investigation of iteration counts and adaptive stopping criteria would help identify optimal review budgets. Integrating execution-based feedback into the review loop could substantially improve feedback quality, and exploring additional agent roles—such as Planner, Debugger, or Optimizer—would test whether more complex architectures provide benefits beyond our two-agent

design. Finally, cost-performance analysis across configurations, failure-mode analysis, and systematic prompt optimization remain unexplored dimensions that could further improve practical deployment.

SECURITY, ETHICS, AND BROADER IMPACT

The proposed pipeline executes model-generated code, so safe deployment requires sandboxing, resource limits, and restrictions on file-system and network access. In this study, code was executed in a controlled subprocess environment, but the paper does not evaluate adversarial prompts, malicious code generation, or insecure programming practices. Future work should include security-oriented evaluation, including whether generated solutions use unsafe operations or depend on hidden environmental assumptions. Multi-agent review loops also increase inference cost and energy use relative to single-pass generation; a complete practical evaluation should therefore report token usage, latency, and cost. Finally, because code models may be trained on public repositories, future deployments should consider licensing, attribution, and privacy risks when generated code resembles training examples.

CONCLUSION

We presented a Writer-Reviewer multi-agent pipeline for competitive programming whose best observed configuration achieves 91.7% Pass@1 on 96 LiveCodeBench AtCoder problems. Through evaluation of twelve configurations combining GPT-OSS-20B, Qwen3-Coder, and Qwen2.5-Coder in different agent roles, we report four main observations.

First, the effectiveness of the review mechanism appears to depend on the capability and output-format compliance of the base Writer model in this setup. GPT-OSS-20B improves by 4.2 percentage points with review, while Qwen3-Coder shows no improvement and Qwen2.5-Coder improves only from 0% to 1.0%. These results suggest that static iterative refinement may amplify existing capabilities rather than reliably compensating for weak or non-compliant Writer outputs. In particular, the Qwen2.5-Coder results should be interpreted with caution because the diagnostic audit indicates substantial structured-output compliance failures. Therefore, these results should not be read as a standalone measure of the model's general coding capability, but rather as evidence that the pipeline is sensitive to prompt compliance and structured-output behavior.

Second, Writer Agent choice shows the largest observed association with system performance, with gaps of up to 88.6 pp across Writers under a fixed Reviewer—larger than the 17.7 pp maximum observed range across Reviewers under a fixed Writer. A strong Writer with the weakest Reviewer (89.6%) substantially outperforms the reverse configuration (13.5%). Third, heterogeneous model configurations may offer practical value. Pairing a strong Writer with a cheaper Reviewer achieves near-best observed performance, suggesting a potential cost-performance tradeoff, although actual cost and latency were not measured. Fourth, in our experiments GPT-OSS-20B outperformed both code-specialized models. This observation should not be generalized to a claim that general-purpose models are superior to code-specialized models, because the comparison is based on a limited model selection and is confounded by scale, release date, serving configuration, and prompt compatibility.

These findings provide preliminary guidance for practitioners: prioritize Writer Agent capability, apply static review mechanisms conditionally based on baseline performance, and consider heterogeneous configurations when balancing performance and cost.

Acknowledgments

This study was supported by the Kocaeli University Scientific Research Projects Coordination Unit under project number 2025-4476. This study was conducted at the Kocaeli University Software Engineering Department, Software Technologies Research Laboratory (STAR Lab).

The authors would like to thank the developers of LiveCodeBench for providing the evaluation framework used to assess the code generation capabilities of the language models in this study. The authors also gratefully acknowledge Nebius for providing API access to the language models evaluated in this work. In addition, the authors thank Freepik for the icons used in some of the figures.

Availability of data and material

The benchmark data used in this study are available from the cited LiveCodeBench resources. Implementation artifacts, prompts, and execution logs may be made available by the authors upon reasonable request, subject to provider terms and storage constraints.

Conflicts of interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Ethical standard

The authors have no relevant financial or non-financial interests to disclose.

LITERATURE CITED

Anthropic, 2024 The claude 3 model family: A new standard for intelligence. Technical report, Anthropic.

Austin, J., A. Odena, M. Nye, M. Bosma, H. Michalewski, *et al.*, 2021 Program synthesis with large language models.

Azerbayev, Z., H. Schoelkopf, K. Paster, M. D. Santos, S. McAleer, *et al.*, 2024 Llemma: An open language model for mathematics.

Birhane, A., A. Kasirzadeh, D. Leslie, and S. Wachter, 2023 Science in the age of large language models. Nature Reviews Physics **5**.

Brown, T. B., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, *et al.*, 2020 Language models are few-shot learners.

Chen, M., J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, *et al.*, 2021 Evaluating large language models trained on code.

Chen, X., M. Lin, N. Schärli, and D. Zhou, 2023 Teaching large language models to self-debug.

Cui, J., M. Ning, Z. Li, B. Chen, Y. Yan, *et al.*, 2024 Chatlaw: A multi-agent collaborative legal assistant with knowledge graph enhanced mixture-of-experts large language model.

DeepSeek-AI, 2025 Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

Gemini Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, *et al.*, 2023 Gemini: A family of highly capable multimodal models.

Guo, D., Q. Zhu, D. Yang, Z. Xie, K. Dong, *et al.*, 2024 Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

Hendrycks, D., S. Basart, S. Kadavath, M. Mazeika, A. Arora, *et al.*, 2021 Measuring coding challenge competence with apps.

Hong, S., M. Zhuge, J. Chen, X. Zheng, Y. Cheng, *et al.*, 2024 Metagpt: Meta programming for a multi-agent collaborative framework.

Huang, D., J. M. Zhang, M. Luck, Q. Bu, Y. Qing, *et al.*, 2024a Agentcoder: Multi-agent-based code generation with iterative testing and optimisation.
Huang, J., X. Chen, S. Mishra, H. S. Zheng, A. W. Yu, *et al.*, 2024b Large language models cannot self-correct reasoning yet.
Hui, B., J. Yang, Z. Cui, J. Yang, D. Liu, *et al.*, 2024 Qwen2.5-coder technical report.
Ishibashi, Y. and Y. Nishimura, 2024 Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization.
Islam, M. A., M. E. Ali, and M. R. Parvez, 2024 Mapcoder: Multi-agent code generation for competitive problem solving.
Jain, N., K. Han, A. Gu, W.-D. Li, F. Yan, *et al.*, 2024 Livecodebench: Holistic and contamination free evaluation of large language models for code.
Jin, H., Z. Sun, and H. Chen, 2024 Rgd: Multi-llm based agent debugger via refinement and generation guidance.
Kasneci, E., K. Sessler, S. Küchemann, M. Bannert, D. Dementieva, *et al.*, 2023 Chatgpt for good? on opportunities and challenges of large language models for education. Learning and Individual Differences **103**: 102274.
Labrak, Y., A. Bazoge, E. Morin, P.-A. Gourraud, M. Rouvier, *et al.*, 2024 Biomistral: A collection of open-source pretrained large language models for medical domains.
LangChain, 2025 Langgraph documentation. Online, Accessed: 2026-04-30.
Li, R., L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, *et al.*, 2023 Starcoder: may the source be with you!
Li, Y., D. Choi, J. Chung, N. Kushman, J. Schrittwieser, *et al.*, 2022 Competition-level code generation with alphacode. Science **378**: 1092–1097.
Lozhkov, A., R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, *et al.*, 2024 Starcoder 2 and the stack v2: The next generation. arXiv preprint arXiv:2402.19173 .
Madaan, A., N. Tandon, P. Gupta, S. Hallinan, L. Gao, *et al.*, 2023 Self-refine: Iterative refinement with self-feedback.
OpenAI, 2024 Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/, Accessed: 2026-04-30.
OpenAI, 2025 gpt-oss-120b & gpt-oss-20b model card. Includes gpt-oss-20b parameter counts and June 2024 knowledge cutoff.
OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, *et al.*, 2024 Gpt-4 technical report.
Qian, C., W. Liu, H. Liu, N. Chen, Y. Dang, *et al.*, 2024 Chatdev: Communicative agents for software development.
Quan, S., J. Yang, B. Yu, B. Zheng, D. Liu, *et al.*, 2025 Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings.
Qwen Team, 2024 Qwen2.5-coder-7b-instruct model card. Online, Accessed: 2026-04-30.
Qwen Team, 2025 Qwen3-coder-30b-a3b-instruct model card. Online, Accessed: 2026-04-30.
Rozière, B., J. Gehring, F. Gloeckle, S. Sootla, I. Gat, *et al.*, 2024 Code llama: Open foundation models for code.
Shao, Z., P. Wang, Q. Zhu, R. Xu, J. Song, *et al.*, 2024 Deepseekmath: Pushing the limits of mathematical reasoning in open language models.
Shinn, N., F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, *et al.*, 2023 Reflexion: Language agents with verbal reinforcement learning.
Singhal, K., S. Azizi, T. Tu, S. S. Mahdavi, J. Wei, *et al.*, 2022 Large language models encode clinical knowledge.
Stack Overflow, 2024 Stack overflow developer survey 2024. https://survey.stackoverflow.co/2024, Accessed: 2026-04-30.
Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, *et al.*, 2017 Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30.
Zhong, L., Z. Wang, and J. Shang, 2024 Debug like a human: A large language model debugger via verifying runtime execution step-by-step.

How to cite this article: Ekiz, T. K., and Konyar, M. Z. Multi-Agent LLM Pipeline for Code Writing: An Experimental Study of Writer-Reviewer Architecture. *ADBA Computer Science*, 3(2), 72–80, 2026.