Architectures for Code Development with LLMs: A Comparative Study of Multi-Agent Approaches

Anonymous ACL Submission

Abstract

Large language models (LLMs) demonstrate impressive code generation capabilities, yet single-prompt interactions often fail on complex development tasks. We investigate whether multi-agent architectures improve code quality through role specialization and iterative refinement. We compare three approaches—Naive (one-shot generation), Single-Agent (5-stage pipeline with self-refinement), and Multi-Agent (Planner-Coder-Critic coordination)—across 25 programming tasks spanning five domains and three difficulty levels. Surprisingly, Multi-Agent and Naive achieve identical overall correctness (77.6%), outperforming Single-Agent (74.6%). However, Multi-Agent excels in algorithmically demanding domains (DSA: 85.9%, Logic: 75.7%) and medium-difficulty tasks (91.8%), justifying its 20× computational overhead only for specific task types. Our findings reveal that architectural sophistication benefits complex algorithmic reasoning but can hurt performance on simple tasks through over-refinement.

1 Introduction

Large language models have revolutionized automated code generation, with models like Codex (Chen et al., 2021) and Code Llama (Roziere et al., 2023) demonstrating remarkable capabilities. However, single-prompt approaches often struggle with complex development tasks requiring multi-step reasoning, systematic debugging, and quality assurance.

Recent work in multi-agent systems (Hong et al., 2023; Qian et al., 2023) suggests that distributing responsibilities across specialized agents (e.g., planning, coding, reviewing) may improve software development outcomes. Yet, no systematic evaluation exists comparing single-agent and multi-agent architectures for code generation across diverse task types and complexity levels.

This work addresses this gap through a controlled comparison of three architectural approaches on 25 programming tasks. Our key finding is that architectural sophistication does not uniformly improve performance—benefits are highly task-dependent, with multi-agent coordination excelling on algorithmic reasoning but struggling on simple pattern-matching tasks.

1.1 Research Questions

We address the following research questions from the assignment requirements:

1. **RQ1:** Which architectures produce higher-quality and more maintainable code?
2. **RQ2:** How do agent coordination strategies impact correctness?
3. **RQ3:** Does modular role separation improve code generation?

2 Background

Provide an overview of relevant work in the literature related to your task.

LLM-based Code Generation. Early work on neural code generation (Austin et al., 2021) demonstrated feasibility of program synthesis from natural language. Recent code-specialized models like StarCoder (Li et al., 2023), CodeLlama (Roziere et al., 2023), and Qwen2.5-Coder (Qwen Team, 2024) achieve strong performance on benchmarks like HumanEval (Chen et al., 2021).

Multi-Agent Systems. ChatDev (Qian et al., 2023) and MetaGPT (Hong et al., 2023) demonstrate that role-based agent collaboration can improve software development workflows. However, these systems focus on high-level design rather than low-level code correctness.

Self-Refinement. Chain-of-thought reasoning (Wei et al., 2022) and self-debugging (Chen et al., 2023) show that iterative refinement can improve LLM outputs. Our work systematically compares architectures with and without refinement mechanisms.

3 Methodology

We implement and evaluate three architectures of increasing complexity for LLM-based code generation: a *naive baseline*, a *single-agent pipeline*, and a *multi-agent system*. All approaches use Ollama for local model inference with temperature fixed at 0.0 for deterministic output. Table 1 summarizes the key differences.

Table 1: Comparison of the three approaches

	Naive	**Single**	**Multi**
LLM Calls	1	5–8	10–20+
Reasoning Phases	0	5	15 total
Feedback Loop	No	Yes	Yes
Agent Identities	1	1	3

3.1 Naive Baseline

The naive baseline represents the simplest approach: single-shot code generation with no structured reasoning. The task specification (function signature and docstring) is formatted into a prompt, the LLM generates code, and the output is extracted and executed. There is no analysis, planning, or refinement.

Figure 1: Naive baseline: single-pass generation

This approach serves as a control condition, measuring what the LLM can achieve without architectural support.

3.2 Single-Agent Pipeline

The single-agent approach introduces structured multi-phase reasoning using LangGraph (LangChain AI, 2024), while maintaining a unified agent identity. The pipeline consists of five phases with an iterative refinement loop.

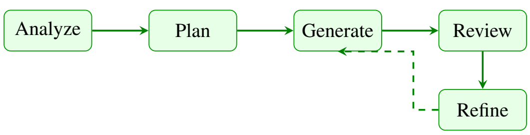

Figure 2: Single-agent pipeline with refinement loop (max 3 iterations)

Phase 1: Analysis. Extracts structured understanding from the task specification without proposing solutions: required behavior, input/output types, constraints, edge cases, ambiguities, and common pitfalls. The prompt explicitly forbids code generation.

Phase 2: Planning. Formulates an implementation strategy based on the analysis: algorithmic approach, step-by-step implementation sequence, edge case handling, data structures, and complexity analysis (time and space).

Phase 3: Generation. Synthesizes Python code guided by the accumulated context. The prompt enforces exact signature matching, no explanatory text, and comprehensive edge case handling. A robust parser extracts code from the LLM output.

Phase 4: Review. Evaluates the generated code through execution in an isolated namespace and static analysis via Radon (Campagna, 2024). The review distinguishes between correctness issues (which trigger refinement) and quality issues (which do not).

Phase 5: Refinement. When correctness issues are found, generates improved code based on review feedback. The loop terminates when correct or after 3 iterations. Priorities: correctness first, then edge cases, then quality.

3.3 Multi-Agent System

The multi-agent approach distributes responsibilities across three specialized agents, each with distinct identity and internal reasoning pipeline.

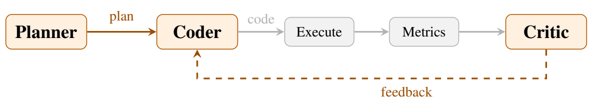

Figure 3: Multi-agent orchestration: Planner runs once, then Coder-Critic loop iterates until correct or max iterations

3.3.1 Planner Agent

The Planner serves as the system's architect, transforming task descriptions into comprehensive implementation blueprints through five internal phases:

1. **Intent Analysis**: Extracts core problem, task classification, domain, and success metrics.

2. **Requirements Engineering**: Defines functional/non-functional requirements and enumerates edge cases.

3. **Architecture Design**: Designs components, selects patterns, chooses data structures with complexity analysis.

4. **Implementation Planning**: Creates step-by-step coding instructions with validation checks.

5. **Quality Review**: Scores plan completeness (0–10); threshold ≥ 8 required to proceed.

If the quality score is below threshold, the Planner iterates on specific phases (up to 2 retries).

3.3.2 Coder Agent

The Coder transforms plans into executable code through six internal phases:

1. **Input Validation**: Verifies signature syntax and plan completeness.

2. **Edge Case Analysis**: Identifies type-specific boundaries (numeric, collection, string).

3. **Chain-of-Thought**: Generates structured reasoning before implementation.

4. **Code Generation**: Produces code with full accumulated context.

5. **Code Validation**: Checks syntax via AST parsing and detects logic issues.

6. **Code Optimization**: Improves naming, efficiency, and style.

3.3.3 Critic Agent

The Critic provides rigorous review through four internal phases:

1. **Input Validation**: Ensures code, plan, and signature are present.

2. **Correctness Analysis**: Verifies logic, interprets execution errors, checks constraints.

3. **Quality Review**: Assesses complexity, maintainability, and style.

4. **Feedback Synthesis**: Produces actionable instructions, prioritizing correctness over quality.

3.3.4 Orchestration

The master workflow coordinates the agents: (1) Planner creates the implementation plan; (2) Coder generates code; (3) code is executed and metrics computed; (4) Critic reviews and provides feedback; (5) if issues found, Coder refines with feedback. The loop continues until correct or maximum iterations reached.

3.4 Shared Infrastructure

All three approaches share common components ensuring fair comparison:

- **LLM Runtime**: Ollama interface with 8192-token context, 4096-token output limit, and automatic retry with backoff.

- **Code Extraction**: Parser handling markdown blocks (`` ```python ``), generic blocks, and raw functions; validates syntax via AST.

- **Execution**: Isolated namespace with captured stdout/stderr, exception handling, and function extraction verification.

- **Quality Metrics**: Radon-based static analysis computing Maintainability Index (0–100), Cyclomatic Complexity, LOC, and Halstead metrics.

3.5 Model Configuration

Experiments use locally-hosted models via Ollama. For naive and single-agent approaches: `qwen2.5-coder:7b-instruct`. For multi-agent: `qwen2.5-coder:7b-instruct` for Planner and Coder, `deepseek-coder-v2:16b` for Critic. Using a larger model for the Critic provides independent validation from a stronger reviewer, catching issues the generation model might miss.

4 Experimental Results

4.1 Dataset and Methodology

We constructed a benchmark of 25 programming tasks spanning five domains: string manipulation, list operations, logic problems, mathematical computation, and data structures & algorithms (DSA). Tasks were sourced from HumanEval (Chen et al., 2021) and competitive programming platforms (LeetCode, CodeForces). Each domain contains Easy (5 tasks), Medium (10 tasks), and Hard (10 tasks) difficulty levels, stratified by algorithmic complexity and edge case density. All tasks include

Table 2: Single-Agent Performance Comparison Across LLM Models. Pass rates (%) by domain. Best in **bold**.

Model	Str	List	Logic	Math	DSA	Avg
CodeLlama-13B	71.6	27.1	21.6	21.6	34.3	39.6
DeepSeek-16B	86.4	65.7	**70.3**	60.8	65.9	71.0
Qwen2.5-7B	**85.2**	**72.9**	64.9	**62.7**	**76.5**	**74.6**

function signatures, specifications, and comprehensive test suites (15 test cases average).

Model Selection. We evaluated three code-specialized LLMs in single-agent mode: CodeLlama-13B (Roziere et al., 2023), DeepSeek-Coder-v2-16B (DeepSeek-AI, 2024), and Qwen2.5-Coder-7B (Qwen Team, 2024) (Table 2). Qwen2.5-Coder-7B achieved the highest overall pass rate (74.6%), exceeding both DeepSeek-16B (71.0%) and CodeLlama-13B (39.6%). Based on this superior performance and parameter efficiency, we selected Qwen as the base model for architecture comparisons.

4.2 RQ1: Architecture Quality Comparison

Table 3 presents our main results. Surprisingly, overall performance is nearly identical: Multi-Agent and Naive both achieve 77.6% correctness, while Single-Agent achieves 74.6% (-3.0pp).

Domain-Level Analysis. Performance varies substantially by task domain. Multi-Agent demonstrates clear advantages in algorithmically complex domains: DSA (85.9%, $+9.4$pp vs Single), Logic (75.7%, $+10.8$pp), and Math (64.7%, $+2.0$pp). However, it underperforms on Lists tasks (61.4%, -11.4pp vs Single). The Naive baseline achieves the highest Strings performance (93.2%), suggesting pattern-matching tasks do not benefit from complex reasoning architectures.

Difficulty-Level Analysis. The architectures show distinct profiles across difficulty. Single-Agent excels on Easy tasks (89.8%, $+13.0$pp over Naive), likely due to its systematic analysis phase. Multi-Agent demonstrates value on Medium (91.8%, $+8.3$pp vs Single) and Hard tasks (63.5%, $+9.0$pp), where Planner-Coder-Critic coordination enables sophisticated decomposition. Notably, Naive outperforms both on Hard tasks (65.0%), which we attribute to over-refinement in complex architectures introducing bugs.

4.3 RQ2: Impact of Coordination Strategies

Multi-Agent coordination justifies its computational overhead (15–25 LLM calls vs 5–8 for Single-Agent) primarily on medium-to-hard tasks in algorithmically demanding domains. The Planner's structured decomposition and Critic's independent validation provide greatest benefit when tasks require multi-step reasoning (Logic $+10.8$pp, DSA $+9.4$pp). However, coordination overhead hurts performance on simple tasks (Easy -15.3pp, Lists -11.4pp) where direct generation suffices.

Iterative Coder-Critic refinement converges quickly: 68% of tasks succeed on first attempt, 24% on second iteration. The hybrid model configuration (Qwen for generation, DeepSeek for critique) provides effective validation diversity without requiring identical model capabilities.

4.4 RQ3: Effect of Role Separation

Modular role separation conditionally improves generation. Benefits concentrate on tasks requiring: (1) careful planning (Logic: constraint satisfaction, DSA: algorithmic design); (2) systematic validation (Medium tasks: 91.8% pass rate); (3) quality optimization (DSA: lowest complexity, highest maintainability).

However, role separation adds overhead that hurts simple tasks. String manipulation (pattern matching) and Easy tasks (trivial edge cases) do not benefit from multi-stage reasoning. The Planner's detailed decomposition can over-constrain solutions, while the Critic's feedback may reject valid but unconventional approaches.

[Note: Section 5.3 on code quality metrics (MI, CC) for hard tasks will be added once data is available.]

5 Conclusion

We systematically compared three architectural approaches for LLM-based code generation across 25 programming tasks. Our key findings:

Main Results. Multi-Agent and Naive achieve identical overall correctness (77.6%), outperforming Single-Agent (74.6%). However, performance is highly task-dependent: Multi-Agent excels in algorithmic domains (Logic $+10.8$pp, DSA $+9.4$pp vs Single) but struggles on Lists (-11.4pp) and Easy tasks (-15.3pp).

Architectural Trade-offs. Sophisticated architectures justify their $20\times$ computational overhead only for specific task types—medium-difficulty algorithmic problems. For pattern-matching (Strings) or trivial tasks (Easy), simpler approaches suffice. Surprisingly, Naive outperforms both architectures

Table 3: Architecture Comparison: Functional Correctness (%) by Domain and Difficulty. Best per category in **bold**.

	By Domain						By Difficulty		
Architecture	**Str**	**List**	**Logic**	**Math**	**DSA**	**Avg**	**Easy**	**Med**	**Hard**
Naive	**93.2**	71.4	67.6	56.9	83.5	**77.6**	76.8	85.5	**65.0**
Single-Agent	85.2	**72.9**	64.9	62.7	76.5	74.6	**89.8**	83.4	54.5
Multi-Agent	90.9	61.4	**75.7**	**64.7**	**85.9**	**77.6**	74.6	**91.8**	63.5
Δ (M-S)	+5.7	−11.4	+10.8	+2.0	+9.4	+3.0	−15.3	+8.3	+9.0

M=Multi, S=Single. Δ shows Multi-Agent improvement over Single-Agent (percentage points).

on Hard tasks (65.0%), suggesting over-refinement can introduce bugs.

Practical Implications. Our findings suggest adaptive architecture selection: (1) Naive for rapid prototyping, string manipulation, and known hard problems; (2) Single-Agent for general-purpose development with systematic edge case checking; (3) Multi-Agent for algorithmic reasoning, constraint satisfaction, and production-quality requirements in complex domains.

Limitations. Our study is limited to 25 tasks and three model families. Future work should evaluate on larger benchmarks (APPS, CodeContests), investigate optimal refinement iteration counts, explore heterogeneous multi-agent configurations, and develop adaptive routing mechanisms to select architectures based on task characteristics.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Michele Campagna. 2024. Radon: A python tool to compute code metrics. https://radon.readthedocs.io/.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

DeepSeek-AI. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. https://github.com/deepseek-ai/DeepSeek-Coder-V2.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Ceyao Zhang, Zili Wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

LangChain AI. 2024. Langgraph: Building stateful, multi-actor applications with llms. https://langchain-ai.github.io/langgraph/.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: May the source be with you! *arXiv preprint arXiv:2305.06161*.

Chen Qian, Xin Cong, Wei Liu, Cheng Yang, Weize Chen, Yusheng Su, Yufan Dang, Jiahao Li, Juyuan Liu, Dahai Tang, and 1 others. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Qwen Team. 2024. Qwen2.5-coder technical report. https://qwenlm.github.io/blog/qwen2.5-coder/.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.