LLM Agents for Collaborative Test Case Generation*

Davide Randino s346257
Marco Roman s343781
Erik Scolaro s343553
Luigi Aceto s343860
Isabella Ilaria Favero s349429

Abstract

Single-turn LLM interactions often fail to produce comprehensive test suites for complex code logic. We investigate the effectiveness of Multi-Agent Systems (MAS) for Python unit test generation, comparing a Single-Agent baseline with Collaborative and Competitive architectures. Evaluating these on algorithmic and stateful problems using Line Coverage and Mutation Score, we show that MAS, particularly the Competitive approach, improves reliability and achieves near-perfect Line Coverage. However, our study reveals a steep cost-performance trade-off: MAS consumes up to 20x more tokens with negligible gains in Mutation Score. While agentic loops maximize structural coverage, reasoning about deep logic remains an expensive challenge.

1 Introduction

Automated software testing is a cornerstone of modern software engineering, essential for ensuring code reliability and preventing regression errors. While the advent of Large Language Models (LLMs) has democratized code generation, relying on single-turn interactions often proves insufficient for robust test generation. A simple LLM chain frequently suffers from hallucinations or the "illusion of coverage", producing test suites that execute without errors but fail to assert critical edge cases or complex business logic effectively.

To overcome these limitations, recent research has shifted towards Multi-Agent Systems (MAS). By decomposing the testing task into specialized roles, such as planning, coding, and debugging, agents can mimic the iterative workflow of human developers. This collaborative paradigm allows for self-correction loops where generated tests are refined based on execution feedback and coverage reports, rather than accepted at face value.

*Code available at: https://github.com/daviderandino/LLM-Agents-for-Collaborative-Test-Case.git

In this work, we present a comparative analysis of agentic architectures for Python unit test generation. We investigate whether the increased complexity of multi-agent interactions translates into tangible improvements in test quality. Specifically, we propose and evaluate three distinct architectures:

- **Single-Agent (baseline)**, representing standard zero-shot LLM usage.
- **Collaborative-Agents**, employing a Planner-Developer-Executor loop with feedback-driven refinement.
- **Competitive-Agents**, which utilizes a tournament-style selection mechanism to optimize for coverage and robustness.

We evaluate these approaches on a diverse dataset comprising both algorithmic data structures and stateful business logic modules. Performance is measured not only by standard Line Coverage but also by Mutation Score, providing the tools to measure and compare the agents' capabilities of writing comprehensive and complete tests.

Research Questions. To guide our investigation, we address these research questions:

- **RQ1 (Effectiveness):** How effective are LLM agents in generating comprehensive and diverse test cases?
- **RQ2 (Collaboration vs. single model):** Does agent collaboration outperform single-model test generation?
- **RQ3 (Collaboration patterns):** What patterns of collaboration lead to higher-quality test cases?

2 Background

2.1 Evolution of Automated Testing

Historically, automated test generation relied on search-based techniques like EvoSuite [2], which often produced effective but unreadable code. The advent of Large Language Models (LLMs) trained on code [1] revolutionized this field, enabling the generation of human-readable test cases. However, single-turn LLM interactions frequently suffer from "hallucinations" and the "illusion of coverage," where generated tests execute code without meaningfully asserting its logic [7].

2.2 Multi-Agent Systems (MAS)

To address the limitations of single models, research has shifted towards Multi-Agent Systems. Seminal works like ChatDev [3] and MetaGPT [4] demonstrated that decomposing complex software tasks into specialized roles (e.g., Planner, Developer, Reviewer) significantly improves code quality and reduces errors compared to zero-shot generation. In this paper, we apply this paradigm specifically to the domain of unit testing.

2.3 Mutation Testing as a Gold Standard

Traditional metrics like line coverage are often insufficient indicators of test suite quality. Mutation testing, which evaluates tests based on their ability to detect semantic faults (mutants) [5], provides a more robust assessment. Empirical studies confirm that mutation score is a strong proxy for real fault detection capability [6], making it the ideal metric to benchmark the reasoning capabilities of our agents.

3 System Overview

3.1 Architectures and Agent Roles

We developed three distinct architectures to investigate the impact of agent interaction patterns on code coverage and mutation scores.

Single Agent Baseline. The Single Agent architecture (`SingleAgent`) serves as a representative of the usual and common approach used to generate tests with an LLM.

Multi-Agent Collaborative Graph. The Collaborative architecture (fig.1) utilizes a cyclic graph where agents work sequentially to improve a shared state. The workflow consists of three specialized nodes:

- **Planner Node**: acting as a "Coverage Specialist", this agent analyzes the source code and existing tests. It produces a structured JSON test plan identifying missing logical paths, boundary values, and edge cases.
- **Developer Node**: acting as a "Pytest Engineer", this agent receives the plan and implements the specific test cases. It supports an "Append Mode" to add new tests without rewriting successful existing ones, preventing regression.
- **Executor Node**: runs the test suite and compute the coverage. If tests fail or syntax errors occur, the workflow routes back to the *Developer* for debugging. If coverage is below 100%, it routes back to the *Planner* to generate a new plan for the missing lines.

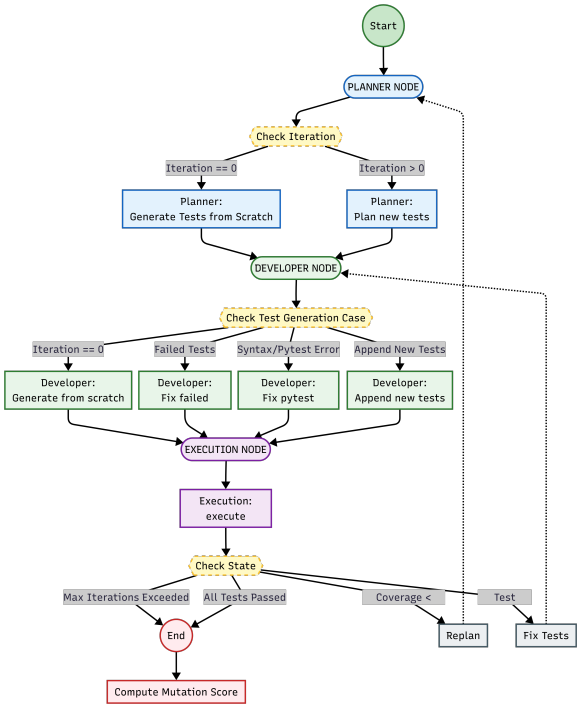

Figure 1: Collaborative Agent Flowchart

Multi-Agent Competitive Graph. The Competitive architecture (fig. 5) introduces a tournament-style selection mechanism to maximize test quality.

- **Competitive Generation**: A central *Planner* creates the test strategy. Subsequently, two distinct *Developer* agents (Developer 1 and Developer 2) independently generate candidate test suites based on the same plan.

- **Execution & Selection**: The *Executor* node runs both candidate suites. A selection algorithm determines the "Winner" based on a strict hierarchy:
    1. Validity (Syntax/Runtime correctness).
    2. Code Coverage (Higher percentage wins).
    3. Robustness (Fewer failed tests in tie-break scenarios).

    The winning test suite becomes the new state, and the loser is discarded. This approach forces the system to converge on the locally optimal solution at every step. Competition is mediated through a shared evolving state rather than inter-agent communication, with only the winning solution propagated to subsequent interactions.

Iterative Refinement. Both multi-agent architectures operate within a defined loop (default `max_iterations= 5` to `10`). The system maintains a persistent `AgentState` containing the source code, the current test suite, the latest test plan, and execution metrics (coverage, passed/failed count). The workflow creates a feedback loop where runtime errors or coverage gaps trigger specific prompt strategies (e.g., "Fixing Failed Tests" or "Gap Filling").

Prompt Engineering Techniques. We utilized several prompt engineering techniques to guide the LLMs:

- **Role Playing**: Agents are assigned specific personas (e.g., "Senior Pytest Debugger", "Coverage Specialist") to narrow the generation scope.
- **Few-Shot Prompting**: The prompts include concrete examples of Input-Code to Output-JSON (for planning) or Plan-to-Code (for generation) to enforce strict output formats.
- **Structured Outputs**: The Planner is constrained to output strictly valid JSON arrays containing test rationale and inputs, preventing hallucination of non-existent code paths.
- **Feedback-Driven Repair**: Instead of blind regeneration, our system employs a context-aware debugging strategy. During repair iterations, the prompt is augmented with three key components: the Source Code, the Current Test Code, and the Pytest Failure Report. This context enables the agent to perform "surgical" fixes, correcting only the specific assertions or logic that failed rather than hallucinating new tests or rewriting valid code.

3.2 Model Selection

To isolate the impact of model capacity on agentic reasoning, we selected four models categorized into two size classes:

- **Big Models:** `Llama-3-70B` and `GPT-Oss-120B`.
- **Small Models:** `Llama-4-scout-17B` and `GPT-Oss-20B`.

This setup allows for ablation studies, such as pairing a "Strong Planner" (e.g., GPT-Oss-120b) with a "Weak Developer" (e.g., Llama-4-scout-17B) to determine which role is more critical for coverage generation. We use the **Groq** API for all LLM calls to access top-tier models (SOTA) without needing expensive, high-end hardware locally. This choice provides the speed (high throughput) necessary to run the architecture's multiple loops in a short time.

3.3 Experimental Setup

Dataset. To evaluate the generalization capabilities of our agents, we utilized a benchmark dataset of Python modules covering distinct programming tasks. The problems are categorized into three domains:

- **Data Structures:** Modules implementing classical algorithmic structures - `Stack` and `LinkedList`. These problems test the agents' ability to handle object references, edge cases (e.g., popping from an empty stack) while maintaining internal state consistency.
- **Business Logic Simulations:** Stateful classes that simulate real-world scenarios, including `BankAccount`, `HotelReservation`, and `LibrarySystem`. These modules require the test suite to track internal state changes and verify logical flows (e.g., preventing a reservation overlap or an overdraft).
- **Complex Logic & Integration Traps:** The `complex_logic` module introduces non-deterministic scenarios and intricate state dependencies, specifically designed to challenge the agents' reasoning capabilities beyond standard algorithmic problems.

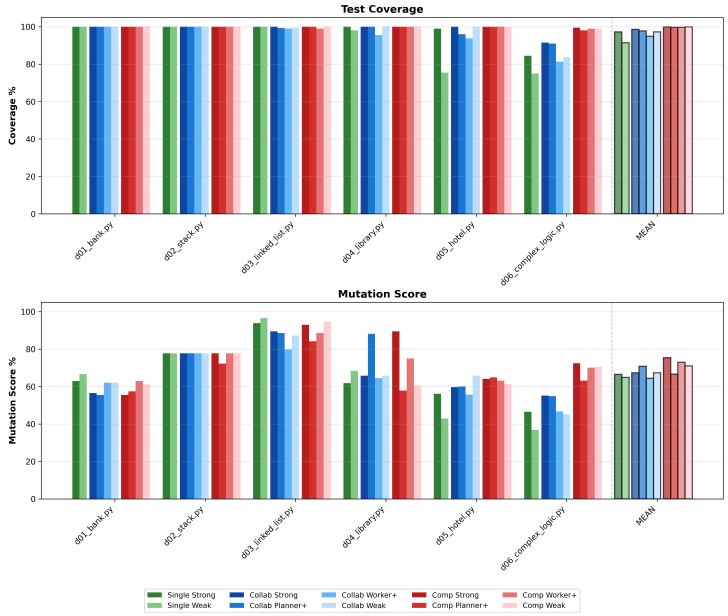

Figure 2: Comparison of Coverage and Mutation Score across different agent configurations for different files.

Evaluation. We evaluate the generated tests using the following criteria:

- **Line Coverage (pytest-cov):** We utilize the `pytest-cov` plugin to calculate line coverage percentage during the generation loop. This real-time metric drives the conditional routing in the Collaborative graph, acting as a threshold to trigger refinement iterations.

- **Mutation Score (mutmut):** As a post-processing validation step, we employ `mutmut` to assess the semantic robustness of the generated tests. This tool injects artificial faults (mutations) into the source code; the ratio of detected faults ("killed mutants") to total valid mutations provides a rigorous measure of test quality beyond simple execution. In our implementation the mutations will be run only on lines of code that were covered by the generated tests, as all the mutations executed on uncovered lines won't ever be detected. This is justified by the fact that we don't want our mutation score to quantify in its value only test robustness and not also uncovered lines.

- **Token Usage:** To evaluate the efficiency and economic viability of the agents, we track the total count of input and output tokens consumed via the API. This metric allows us to analyze the trade-off between architectural complexity and operational cost.

Hyperparameters. We fixed the sampling temperature to $T = 0.2$. This setting balances consistency and flexibility. It ensures our results are reliable and easy to repeat, while giving the agent just enough randomness to avoid getting stuck in repetitive mistakes.

Sanity Check. To ensure the reliability of our evaluation, we implemented a strict post-generation validity check: any generated test case that fails against the original, correct target code is automatically discarded before computing evaluation metrics (coverage and mutation score). This prevents invalid tests (i.e., tests with incorrect assertions) from artificially inflating the mutation score through spurious kills.

4 Experimental results

To evaluate the proposed architectures, we conducted a comprehensive experimental campaign involving multiple model configurations. Before discussing the research questions, we define the experimental configurations derived from the combination of **Big Models** and **Small Models**. We grouped these configurations into three categories to analyze the impact of model capability on the agentic workflow:

- **Strong Configuration**: All agents in the architecture (Planner, Developer, and eventually the second Developer) are instantiated using large size models. This setup represents the

upper bound of reasoning capability.

- **Weak Configuration**: All agents utilize smaller, cost-effective models. This setup tests whether the agentic architecture alone can compensate for the limited reasoning power of smaller models.
- **Planner+ Configuration**: This hybrid setup makes models of different sizes interface each other in our architectures. We will have a Big Model as a planner and a small model as developer.
- **Worker+ Configuration**: The exact opposite of Planner +, with a small planner and big developer.

Each configuration defines 4 experiments for the Collaborative architecture and 2 experiments for the Competitive Architecture. This is due to the fact that we decided to put always models from different families in the developer node of the Competitive architecture.

The overall results comparing Line Coverage and Mutation Score across these configurations are illustrated in Figure 2, where each bar is the mean of 2 experiments for the single agent, 4 experiments for the Collaborative agent and 2 experiments for the Competitive agent.

The following subsections address our research questions by analyzing the Line Coverage, Mutation Score, and Token Usage metrics collected across the experiments.

4.1 RQ1: Effectiveness of Agentic Test Generation

Our first research question assesses the general capability of LLM agents to generate high-quality tests. Looking at the aggregate results, both Multi-Agent architectures demonstrated a high degree of effectiveness. As shown in our data, the *Competitive* architecture achieved a mean Line Coverage of nearly 100% across most configurations (e.g., *GPT-Oss-120B* setups).

While coverage remains consistently high, mutation scores are substantially lower and vary across tasks, particularly for modules involving more complex control flow and semantic constraints. As illustrated in Figure 2, even configurations with near-perfect coverage often fail to exceed mid-range mutation scores. This gap indicates that, although agentic approaches are highly effective at generating syntactically complete and well-covered test suites, their effectiveness at capturing subtle logical errors remains limited.

4.2 RQ2: Agent Collaboration vs. Single Model

This question investigates whether the complexity of Multi-Agent Systems (MAS) yields tangible benefits over a simple Single-Agent baseline. Comparing the *Single-Agent* baseline against the *Collaborative* and *Competitive* architectures, we observe two distinct trends:

1. **Stability and Reliability**: The Single-Agent configurations show noticeably higher variability across tasks, with performance dropping sharply on more complex modules such as *d05_hotel.py* and *d06_complex_logic.py*. While Single-Agent runs can reach high coverage on simpler files, their mean coverage is consistently lower and less uniform than that of Multi-Agent systems. In contrast, both Collaborative and Competitive Multi-Agent architectures achieve near-perfect coverage across almost all files. A similar pattern emerges for mutation score: Single-Agent approaches fluctuate widely and degrade on complex logic, whereas Multi-Agent setups maintain higher and more consistent mutation scores.
2. **The Cost-Quality Trade-off**: While MAS architectures outperform the Single Agent in Mutation Score (e.g., $\sim$76% vs $\sim$68% for the best configurations), the gap is not exponentially large. The "Collaborative" and "Competitive" agents do not outperform the Single Agent by an order of magnitude in terms of test quality, but they do so in terms of cost. The token usage analysis, illustrated in Figure 3 reveals that Multi-Agent systems consume significantly more resources—averaging between 20k and 48k tokens for Competitive runs, compared to just $\sim$2k tokens for Single Agent runs. This suggests that while collaboration ensures reliability and marginally better quality, it comes at a steep operational cost.

Qualitative Analysis. While quantitative metrics indicate a gap in Mutation Score, a manual inspection reveals the root cause. We observed a recurrent pattern of "Shallow Testing" in the baseline, where tests satisfy coverage requirements via execution paths without verifying state transitions. Figure 4

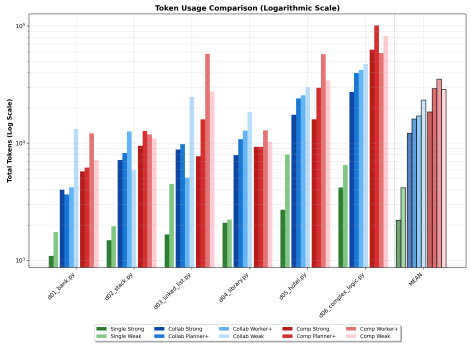

Figure 3: Token usage comparison on logarithmic scale.

contrasts a representative case: the Single-Agent test relies on loose string matching of the standard output, failing to check if the book's status actually changed. Conversely, the Competitive Agent explicitly asserts the internal state (`is_available is False`). This deeper verification explains why the Multi-Agent architecture achieves a higher Mutation Score, effectively detecting bugs that the baseline ignores.

(a) Single Agent: Output Check Only

```
def test_borrow_success(capsys):
    manager = LibraryManager()
    manager.add_book("Dune", "F.H.", "123")

    manager.borrow_book("123")

    # WEAK: Checks stdout only
    cap = capsys.readouterr()
    assert cap.out.endswith("Success...\n")
```

(b) Competitive Agent: State Verification

```
def test_borrow_success(fresh, capsys):
    fresh.add_book("Dune", "F.H.", "123")

    fresh.borrow_book("123")

    # STRONG: Checks internal state
    cap = capsys.readouterr()
    assert cap.out == "Success...\n"
    assert fresh.inventory["123"].is_available is False
```

Figure 4: Qualitative impact on **Mutation Score**.

4.3 RQ3: Collaboration Patterns

Our analysis reveals distinct advantages in how different architectural configurations handle task complexity.

The Collaborative architecture does not achieve steady performance through the experiments. During our experiments, we observed that this architecture is not particularly reliable, as repeated runs of the same experiment exhibit noticeable variance in performance. As shown in Figure 2, the differences in mean coverage between the configurations are negligible. The mutation score appears to be largely driven by stochastic effect, as even the challenging d05 file was cleared by the weakest collaborative configuration with the highest mutation score.

The *Competitive* architecture leads in Line Coverage because it effectively leverages the **heterogeneity** of parallel generation. By instantiating two distinct Developer agents, we introduce reasoning diversity: since different models (or instances) possess unique training biases and "blind spots," the probability of both agents failing on the same logical hurdle is drastically reduced. This decorrelation of errors acts as a robust fail-safe mechanism: if one Developer hallucinates or produces a syntax error, the other often provides a valid alternative. Consequently, the Executor almost always receives a high-quality candidate, allowing the architecture to saturate coverage (100%) even without relying exclusively on top-tier models. The mean mutation scores of this architecture indicate that, a strong developer is more effective than a strong planner. While this architecture performs well on challenging files, it is less effective on short and simple ones.

5 Conclusion

In this study, we evaluated Single-Agent, Collaborative, and Competitive architectures for Python test generation. Our results demonstrate that Multi-Agent Systems effectively solve the reliability and variance issues of single-turn LLMs. Specifically, the **Competitive** architecture proved to be a robust "coverage maximizer," consistently achieving near-perfect line coverage through parallel generation and selection.

However, this stability comes with a significant trade-off. We observed diminishing returns regarding the Mutation Score: while agents excel at traversing code paths, they incur a high operational cost (up to 20x more tokens) without a proportional increase in semantic fault detection. The agents appear more proficient at satisfying structural constraints than at conceiving deep logical assertions.

Future research could address the efficiency gap by exploring adaptive routing strategies, triggering multi-agent loops only when single-shot generation fails. Additionally, integrating a specialized "Reviewer" agent could help bridge the gap between simple line coverage and true semantic robustness, improving the quality of assertions without relying solely on costly parallel execution.

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[2] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419.

[3] Chen Qian, Xin Cong, Wei Yang, et al. 2023. ChatDev: Communicative Agents for Software Development. *arXiv preprint arXiv:2307.07924*.

[4] Sirui Hong, Mingchen Zhuge, Jonathan Chen, et al. 2024. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

[5] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678.

[6] René Just, Darioush Jalali, Laura Inozemtseva, et al. 2014. Are mutants a valid substitute for real faults? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665.

[7] M. Siddiq and J. Santos. 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*.

A Supplemental Materials

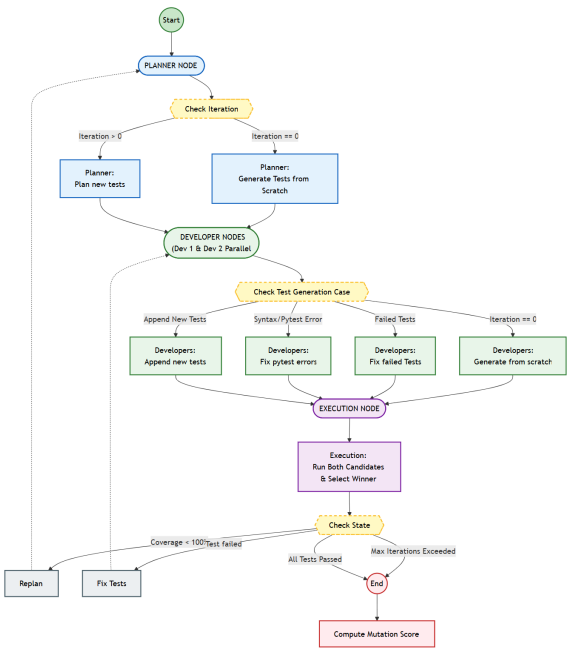

Figure 5: Competitive Agent Flowchart