A Hybrid Neuro-Symbolic BDI Multi-Agent Architecture for LLM-Based Unit Test Generation

Minel Üstgel[1]*, Esi Aşkın[1], Barış Tekin Tezel[1], and Moharram Challenger[2]

[1] Department of Computer Science, Dokuz Eylül University, Izmir, Turkey
m.ustgel@ogr.deu.edu.tr, askin.esi@ogr.deu.edu.tr, baris.tezel@deu.edu.tr
[2] Department of Computer Science, University of Antwerp and Flanders Make, Antwerp, Belgium
moharram.challenger@uantwerpen.be

Abstract. Automating unit test generation with Large Language Models (LLMs) has shown great potential, yet standard generative approaches often struggle with systematic path exploration and the effective integration of deterministic execution feedback. This paper introduces a hybrid neuro-symbolic multi-agent architecture, developed within the JaCaMo framework, where agents follow a Belief–Desire–Intention (BDI) deliberation cycle to perform goal-directed test generation, that re-frames unit test generation as a goal-directed, agentic search process. By combining the creative reasoning of LLMs with the formal precision of symbolic analysis, the architecture utilizes a society of role-specialized agents and artifacts to transform coverage gaps into symbolic "Logic Hints". This creates an iterative neuro-symbolic feedback loop capable of resolving complex branch conditions and navigating deep logic paths that typically stagnate one-shot prompting methods. Evaluation across seven benchmarks shows that our architecture achieves a 100% success rate in reaching the targeted coverage threshold, consistently outperforming all baselines. Statistical tests (Friedman, Wilcoxon) confirm superior reliability and search efficiency with large effect sizes ($A_{12} > 0.8$), suggesting that structured agentic autonomy effectively bridges the gap between LLM reasoning and formal software testing requirements.

Keywords: Multi-Agent Systems · BDI Agent · Large Language Models · Unit Test Generation · Neuro-Symbolic AI

1 Introduction

Software testing is a crucial undertaking that serves as a cornerstone for ensuring the quality and reliability of software products [18]. The precision of software is largely ensured by unit testing; however, writing unit tests manually is a time-consuming process [8]. This inefficiency necessitates a transition toward the paradigm of "Automation Analysis" to streamline development [8].

* Corresponding author.

Large Language Models (LLMs) have achieved remarkable success in various domains, yet they exhibit numerous limitations such as a lack of autonomy and self-improvement [10]. Empirical studies on Java programs reveal that even advanced models struggle to achieve significant coverage on complex, real-world benchmarks [16]. Furthermore, the often-criticized "hallucination" of LLMs remains a significant barrier to their reliable application in industrial-scale testing environments [6, 1].

The engineering of Multi-Agent Systems (MAS) provides a robust foundation for addressing these limitations through core properties such as autonomy and reactivity [5]. Recent research indicates that multi-agent architectures enhance performance across various software testing tasks, including test case generation and debugging [17]. These agents typically operate within a framework comprising modules for perception, memory, and action to optimize complex exploration tasks [19]. Emerging systems such as CoverNexus have demonstrated that multi-agent configurations can significantly outperform single-agent baselines in code coverage enhancement [2]. Moreover, treating testing as a separable problem where agents collaborate via semantic dependency structures allows for more effective navigation of the search space [11]. While traditional tools often target lower-level adequacy metrics, there is an increasing demand for semantic, intent-driven automation to validate system behavior [20]. In this study, the analyzed targets are standard method-centric Java programs, not MAS programs themselves. Classical automated test generation also includes symbolic and concolic execution, as in DART and CUTE [9, 15], feedback-directed random testing and its Java tool Randoop [13, 12], and search-based Java test-suite generation as in EvoSuite [7]. We do not position the current evaluation as replacing these mature tools; rather, we study whether MAS coordination can make LLM-based generation more stable under executable feedback.

In this paper, we propose a hybrid multi-agent architecture implemented via the JaCaMo framework [3], an integrated multi-agent programming platform that brings together Jason [4] for Belief–Desire–Intention (BDI)-based deliberative agent reasoning, CArtAgO [14] for artifact-based environment modeling in order to maximize structural line coverage through a neuro-symbolic feedback loop.Beyond conventional LLM pipelines, our architecture is interpreted through the lens of Agentic AI [5], each component operates as a goal-directed entity whose behavior is structured through a BDI deliberation cycle, equipped with specialized perception channels, reasoning capabilities, and bounded decision authority. This framing effectively shifts automated test generation from a standard prompt–response paradigm to a coordinated, goal-oriented search process grounded in executable feedback. By integrating neuro-symbolic reasoning into a disciplined MAS engineering process, our approach coordinates four specialized agents through CArtAgO artifacts to iteratively resolve uncovered code segments. Ultimately, by combining the generative power of LLMs with structured MAS coordination, this framework provides a deterministic path toward exhaustive unit testing and semantically sound assertion reliability. The main contributions of this paper are:

- **Agentic Interpretation:** An agentic interpretation of LLM-based unit test generation as a cooperative, goal-driven search process rather than a linear pipeline.
- **BDI-based Deliberative Control:** The use of a Belief–Desire–Intention reasoning model to structure goal commitment, coverage-driven intention revision, and coordinated agent deliberation.
- **Neuro-symbolic Coordination:** A hybrid neuro-symbolic coordination loop that enables agents to iteratively refine test generation hypotheses using deterministic execution feedback.
- **Artifact-mediated Environment:** A modular, artifact-mediated environment that grounds generative reasoning in symbolic evidence, ensuring that agent beliefs (in the BDI sense) are systematically updated and anchored in the source code's structure..
- **Empirical Validation:** Empirical evidence demonstrating that structured agent coordination significantly improves both structural coverage completeness and semantic correctness in algorithmic testing scenarios.

To systematically assess these contributions, we evaluate the proposed architecture along three dimensions: (i) coverage reliability across diverse benchmarks, (ii) search efficiency under iterative refinement, and (iii) semantic correctness of generated test oracles.

The remainder of this paper is organized as follows. Section 2 presents the proposed hybrid neuro-symbolic multi-agent architecture, including the agent roles and the artifact-mediated feedback loop that converts coverage misses into symbolic Logic Hints. Section 3 describes the evaluation methodology, research questions, baselines, and experimental setup, and reports quantitative results across the benchmark subjects. Section 4 discusses the implications of agentic structuring, the observed coverage–correctness trade-off, and the main limitations of the approach. Finally, Section 5 concludes the paper and outlines directions for future work.

2 The Proposed Architecture

The proposed framework follows a hybrid neuro-symbolic multi-agent design, where the generative reasoning of Large Language Models (LLMs) is disciplined by deterministic symbolic analysis. In our implementation, all LLM-mediated input generation and oracle synthesis calls are routed through an OpenAI `gpt-4o-mini` interface. The architecture is implemented using the JaCaMo platform, leveraging Jason for agent reasoning and CArtAgO for environment abstraction through specialized artifacts. As illustrated in Fig. 1, the system coordinates four autonomous agents to achieve maximum structural line coverage through an iterative feedback loop.

2.1 Agentic Roles and Responsibilities

The distribution of labor is designed to manage the complexity of the test generation lifecycle:

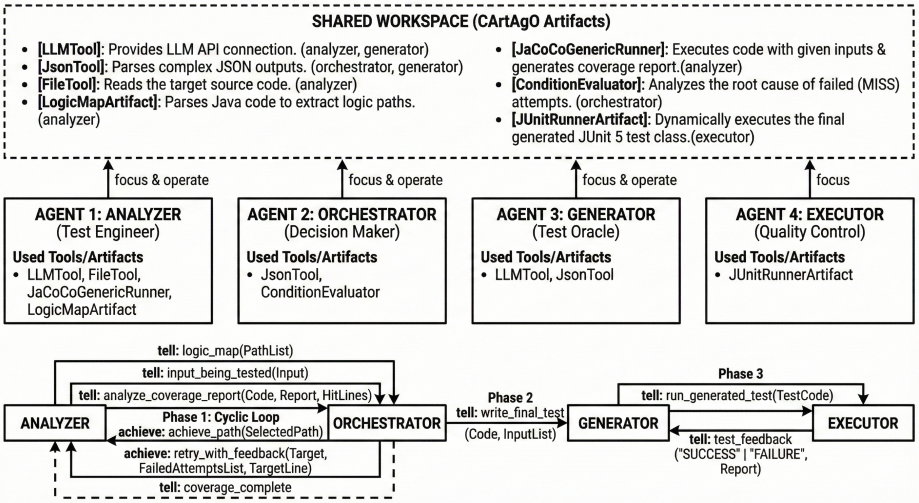

Fig. 1. Overview of the proposed hybrid multi-agent architecture for LLM-based unit test generation and logic coverage enhancement

- **Analyzer Agent (Path Mapping & Data Generation):** This agent acts as the primary data provider. It utilizes the `LogicMapArtifact` to perform static analysis via JavaParser, extracting deterministic execution paths and branch conditions as formal beliefs. Its core responsibility is to manage the "Current Input List" and generate unique candidate test data. When a specifically targeted logical path is missed, the Analyzer utilizes feedback to refine its generation strategy.
- **Orchestrator Agent (Strategic Loop Control):** Serving as the central controller, the Orchestrator manages the global coverage state and the "Reflexion" budget. It monitors real-time feedback from the dynamic execution environment. When a specifically targeted logical path is identified as a "MISS" in the coverage report, the Orchestrator invokes the `Condition EvaluatorArtifact`. It then communicates the resulting symbolic analysis to the Analyzer to guide the next iteration. This loop is fully automated, with no manual prompt rewriting or human intervention during execution.
- **Generator Agent (Semantic Test Oracle):** Designated as the semantic expert, the Generator operates after coverage targets are met. It analyzes the source code to infer the developer's original intent. It then synthesizes a complete JUnit 5 test suite, utilizing verified input lists to produce precise assertions that validate the method's behavior against its semantic purpose. This oracle-synthesis step is downstream from coverage exploration: the Generator uses verified inputs collected during exploration and performs source-code-based intent inference to derive expected outputs and assertions.
- **Executor Agent (Quality Control):** The Executor handles the dynamic verification of the synthesized test suite. It utilizes the `JUnitRunnerArtifact` to compile and run the tests in an isolated in-memory environment. It re-

ports the final success or failure status back to the Generator, ensuring that the produced code is syntactically correct and execution-ready.

2.2 Symbolic Environment and Artifact Interaction

The core innovation of our framework lies in the environment-agent interaction facilitated by specialized CArtAgO artifacts, which serve as a bridge between high-level neural reasoning (LLMs) and the deterministic structure of the source code. These artifacts provide a "ground truth" for the agents, ensuring that the generative process is anchored in symbolic evidence rather than probabilistic guessing.

LogicMapArtifact: Static Decomposition and Sub-logic Expansion The `LogicMapArtifact` acts as the system's static analysis engine, utilizing the JavaParser library to construct a comprehensive symbolic map of the target method. Its input is the source code of the Java class under test; the current prototype focuses on the target method and its internal control-flow paths rather than interprocedural caller–callee trajectories. Its functionality extends beyond conventional path extraction through two primary mechanisms:

- **Symbolic Path Extraction:** It decomposes the method into all reachable execution trajectories, including conditional branches (`if/else`), loops (`while/for`), and `return` statements. Each path is injected into the agents' belief base as a formal target, effectively mapping the entire logical search space.
- **Dependency Tracing and Sub-logic Generation:** The artifact recursively traces the dependencies of variables within predicates back to the initial method parameters. By tracking variable assignments and transformations throughout the method body, it expands complex predicates into a symbolic sub-logic template (e.g., converting `if (diff > 0)` into `[SUB_LOGIC: diff = (a + b) - (a * b)]`). This algebraic representation allows the Analyzer agent to perceive the internal computation logic, providing the LLM with a clear roadmap of how input manipulations influence specific branch outcomes.

ConditionEvaluatorArtifact: Predicate Satisfaction Analysis The `ConditionEvaluatorArtifact` is a lightweight heuristic predicate evaluator designed to resolve coverage gaps; it does not perform full symbolic execution. Unlike standard debugging tools that analyze runtime execution errors, this artifact focuses on predicate satisfaction. When the Orchestrator identifies a "MISS" for a specifically targeted logical path, this artifact evaluates the branch requirements using the latest concrete input values and the sub-logic expressions extracted by `LogicMapArtifact`:

- **Heuristic Predicate Evaluation:** It decomposes boolean expressions to identify which specific component of a predicate caused a branch failure. By

comparing current input values against required symbolic states, it identifies the logical discrepancy.
- **Logic Hint Generation:** Instead of providing raw stack traces or error messages, the artifact produces a Logic Hint (e.g., "Current value: 5, Required: > 10 to satisfy this branch"). This transformational feedback significantly reduces the search space for the LLM, enabling the system to satisfy complex edge cases and boundary conditions that standard iterative prompting often fails to resolve.

JaCoCoGenericRunner: Dynamic Feedback and Deterministic Transitions This artifact acts as the system's dynamic observer, providing real-time, line-level coverage data through bytecode instrumentation. The resulting coverage reports serve as deterministic triggers for the Orchestrator, enabling a precise transition from the exploration phase (managed by the Analyzer) to the final synthesis phase (managed by the Generator). This ensures that a complete test suite is only synthesized once maximum line coverage targets have been verified.

Supporting Artifacts and Environmental Infrastructure The operational continuity of the framework is supported by a suite of specialized artifacts that abstract low-level system operations, allowing the BDI agents to remain focused on high-level strategic reasoning. These artifacts standardize the interaction between the multi-agent society and external resources such as the file system, the compiler, and the LLM APIs.

- **LLMTool (The Neural Gateway):** This artifact serves as the primary interface between the MAS and the Large Language Model. It encapsulates the complexities of API communication, managing request-response cycles and connection resilience. By treating neural reasoning as an environmental service, it enables agents to delegate cognitive tasks—such as data generation or semantic analysis—through a standardized communicative channel.
- **FileTool (The Data Ingestion Artifact):** To ensure that agent beliefs are grounded in the actual source code, the `FileTool` provides secure access to the local file system. Through the `readSourceCode` operation, it ingests Java source files and injects their raw content into the agents' belief base. This establishes the initial knowledge state required for subsequent static and dynamic analysis.
- **JsonTool (The Semantic Structuring Artifact):** A persistent challenge in LLM-based systems is the transition from unstructured text to structured logic. The `JsonTool` facilitates this by performing syntactic sanitization on LLM responses. Through the `sanitizeLLMResponse` operation, it removes Markdown artifacts and incompatible characters, ensuring that the data is ready for the unification process within the agents' BDI reasoning cycles.
- **JUnitRunnerArtifact (The Isolated Execution Engine):** Dynamic verification is performed in a non-persistent, isolated environment via this artifact. Utilizing a `MemoryJavaFileManager`, it compiles synthesized JUnit

test classes in-memory without disk I/O overhead. By executing tests within an isolated classloader, it provides a safe sandbox to validate the correctness of generated code before it is finalized.

2.3 Agentic Interpretation of the Architecture

From an Agentic AI perspective, the framework forms a society of autonomous agents driven by explicit goals and grounded environmental perception [5]. The Orchestrator encodes the global objective as a measurable maximum line coverage target and bounds the search space via the Reflexion budget, avoiding the pitfalls of unconstrained generative prompting.

Agency is realized through a closed perception–reasoning–action loop mediated by CArtAgO artifacts, creating a neuro-symbolic feedback system. The LogicMapArtifact provides a symbolic map of trajectories and sub-logic templates for perception; the JaCoCoGenericRunner reports deterministic line-level coverage signals; and the ConditionEvaluatorArtifact converts coverage misses into actionable Logic Hints. These hints serve as a formal self-reflection signal that updates agent beliefs and strategically guides the Analyzer toward uncovered branches.

Unlike conventional pipelines with passive modules, each agent retains local decision authority within a coordinated BDI (Belief-Desire-Intention) cycle. The Analyzer proposes unique inputs based on logic maps, the Orchestrator prioritizes targets using coverage feedback, and the Generator-Executor pair synthesizes and verifies an execution-ready JUnit suite once coverage criteria are met. Consequently, the framework operationalizes agentic principles—such as goal-directed behavior, environment grounding, and iterative self-correction—within a concrete and disciplined software engineering task.

2.4 Workflow and Logic Satisfaction

The operationalization of the proposed framework is formalized in Algorithm 1, representing the procedural manifestation of the agentic society. The workflow is structured into three distinct phases that transition from static symbolic decomposition to dynamic coverage refinement and semantic synthesis. Here, L_{total} is derived from the path set extracted by `LogicMapArtifact`; $N_{\max}$ is a global retry budget for LLM-guided reflection, while a separate per-target miss limit prevents repeated attempts on a single unreachable path.

3 Evaluation

3.1 Research Questions and Metrics

We evaluate whether the proposed hybrid neuro-symbolic multi-agent architecture improves the reliability and completeness of LLM-based unit test generation. Our evaluation is guided by three research questions:

Algorithm 1 Hybrid Neuro-Symbolic Unit Test Generation

Require: Java source code S, max. global reflection budget N_{max}
Ensure: Executable JUnit 5 test suite T_{final}

Phase I: Static symbolic mapping
Initialize agents: *Analyzer, Orchestrator, Generator, Executor*
Analyzer parses S via `LogicMapArtifact`
Build symbolic path set $L = \{p_1, p_2, \ldots, p_n\}$
Set $L_{total} \leftarrow |L|$ and $B_{global} \leftarrow 0$
Orchestrator stores L as global targets
Phase II: Strategic coverage exploration
while $(CoveredPaths + BlockedPaths) < L_{total}$ **do**
  Orchestrator selects $p_{target} \in L$ and delegates to *Analyzer*
  Analyzer generates input I_{curr} via `LLMTool`
  if p_{target} contains sub-logic **then**
    Add symbolic logic hints to the prompt (e.g., $diff = (a + b) - (a * b)$)
  end if
  Execute I_{curr} via `JaCoCoGenericRunner`
  if p_{target} is HIT **then**
    Mark p_{target} covered; reset per-target miss counter M
  else
    Orchestrator analyzes miss via `ConditionEvaluatorArtifact`
    Produce symbolic feedback; retry using failure history
    $M \leftarrow M + 1$; $B_{global} \leftarrow B_{global} + 1$
    if $M \geq 5$ or $B_{global} \geq N_{max}$ **then**
      Mark p_{target} blocked
    end if
  end if
end while
Phase III: Semantic synthesis
Let $I_{verified}$ be the inputs confirmed to cover at least one target
Generator infers intent and synthesizes T_{final} using $I_{verified}$
Executor validates T_{final} via `JUnitRunnerArtifact`
return T_{final}

RQ1 (Coverage Reliability): How reliably does the proposed architecture achieve high line coverage across diverse algorithmic scenarios compared to non-agentic baselines?

- *Metric 1: Line Coverage (LineCov):* Measured via JaCoCo bytecode instrumentation.
- *Metric 2: Success Rate per Run:* The percentage of individual executions exceeding a 90% coverage threshold.
- *Metric 3: Scenario Achievement:* The percentage of subjects where the 90% threshold was reached at least once across all runs.

RQ2 (Search Efficiency): To what extent do symbolic Logic Hints reduce redundant test attempts compared to standard iterative prompting?

- *Metric: Test Efficiency:* Coverage achieved per generated test, computed as $LineCov/\#Tests$; aggregate tables report the mean of this value over

successful runs ($\bar{E}_{success}$). This metric captures compactness of coverage-oriented search rather than wall-clock time or API cost.

RQ3 (Semantic Cohesion and Correctness): How does the decoupling of path exploration from test synthesis affect the semantic alignment of test oracles?

- *Metric: Oracle Correctness (%):* The ratio of generated tests where the expected values accurately reflect the program's intended logic.

3.2 Experimental Setup

Subjects. We evaluate on seven controlled Java benchmark classes from the project repository: *ArrayStatAnalyzer*, *BankAccount*, *InsurancePremiumCalculator*, *LeapYearChecker*, *OrderProcessor*, *PayrollCalculator*, and *TriangleClassifier*. These subjects are representative of introductory algorithmic tasks, featuring diverse control-flow and programming constructs such as nested `if/else` conditions, boundary-heavy predicates, `for/while` loops (including skip/enter cases), and simple object creation with field assignments and method calls. They provide a controlled setting where missed branches and boundary conditions can be explicitly observed via coverage feedback.

Procedure. For each subject, we run each approach 5 times under identical constraints (same coverage target, maximum reflection budget $N_{\max} = 100$, and stopping criteria). Each run produces a JUnit 5 test suite. All retry and reflection steps were triggered automatically in every run. Repeated runs account for LLM sampling effects and the Orchestrator's random selection among currently uncovered target paths. We record line coverage (JaCoCo), executability (compile-and-run success), and the number of generated tests.

Stopping criteria. A run terminates when the global reflection budget is exhausted, or in the agentic configurations, all extracted targets are marked as covered or blocked. In addition, a target path is blocked after five consecutive MISS outcomes.

3.3 Baselines

We compare four configurations under the same execution harness. These configurations are intended as a controlled ablation-style comparison of the proposed MAS design rather than as an exhaustive comparison against all unit-test-generation tools. Mature non-LLM generators such as symbolic/concolic tools, Randoop, and EvoSuite are treated as reference points for future direct comparison rather than empirical baselines in this study:

Pure/Zero-shot LLM. A single prompt requests a comprehensive JUnit 5 test class aiming at full coverage. No structured iteration is performed beyond code extraction/normalization.

Self-Reflection. The system iteratively re-prompts the LLM using natural-language feedback derived from the previous run (e.g., observed failures or uncovered branches), without symbolic sub-logic templates or artifact-generated Logic Hints. This captures a common "LLM + feedback" baseline.

Generator-Critic LLMs. A reduced multi-agent pipeline where the Orchestrator/Analyzer loop exists but *without* symbolic sub-logic expansion and *without* Logic Hint generation for missed targets, resulting in a weaker form of guided exploration.

Proposed MAS. The full architecture with artifact-mediated path mapping (`LogicMapArtifact`), deterministic coverage observation (`JaCoCoGenericRunner`), and miss-to-hint transformation via predicate satisfaction analysis (`Condition EvaluatorArtifact` producing Logic Hints).

3.4 Quantitative Analysis

This section provides a high-level overview of the experimental findings across seven benchmarks, focusing on aggregate performance and statistical significance. Detailed per-subject results are presented in Section 3.5.

RQ1 (Coverage Reliability): The aggregate line coverage performance is summarized in the boxplot (**Fig. 2**) and Table 2. The Proposed MAS demonstrates superior reliability, maintaining a **100% Success Rate per Run** (SR_{run}) and **100% Scenario Achievement** ($SA_{\%}$), effectively reaching the 90% coverage threshold in every execution. In contrast, baselines exhibit significant variance, with the **Pure LLM** and **Reflect** strategies failing to achieve the target in roughly 40–50% of the runs. To confirm the validity of these trends, we performed a **Friedman test**, which indicated a statistically significant difference between the strategies ($\chi^2 = 31.21, p < 0.001$). Post-hoc **Wilcoxon signed-rank tests** further revealed that Proposed MAS significantly outperforms Pure LLM ($p < 0.001, A_{12} = 0.84$) and Reflect ($p < 0.001, A_{12} = 0.76$) with large effect sizes, as well as the Generator-Critic baseline ($p = 0.004, A_{12} = 0.58$).

RQ2 (Search Efficiency): The proposed approach is most efficient under both aggregate views: $\bar{E}_{success} = 9.84$ in Table 2 and average efficiency = 9.82 in Table 3. Generator-Critic is lower in both tables ($\bar{E}_{success} = 6.46$; average efficiency = 5.95), despite sharing the agentic structure, because it requires more iterations to resolve complex conditions. Although Pure LLM occasionally shows competitive efficiency in simpler tasks, this is often a result of a "coverage plateau" where the model fails to explore deeper paths, thus generating fewer but less effective tests. The Proposed MAS effectively narrows the search space by transforming execution gaps into structured Logic Hints, ensuring high coverage is reached with a stabilized number of tests.

RQ3 (Correctness): Oracle correctness is maintained as a foundational requirement. All strategies achieved average accuracy scores between 43% and 53% (Table 3). While the Proposed MAS prioritized structural exploration, it maintained an average accuracy of 47.30%, which is comparable to the Pure LLM baseline (48.68%). Such performance should be interpreted as a limitation rather than a claim of oracle superiority, since oracle generation is not the main strength of the proposed system. These results confirm that the decoupled architecture of the proposed framework successfully reaches difficult logic branches without compromising the semantic integrity of the final test suite.

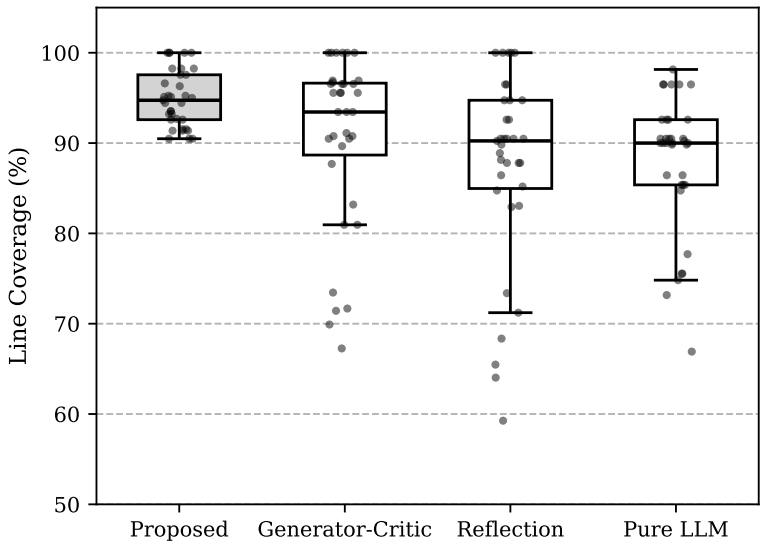

Fig. 2. Distribution of Line Coverage (%) across the proposed multi-agent architecture and baseline approaches

Table 1. Descriptions, column names, and symbols for performance metrics.

Description	Column Name	Abbreviation / Symbol
Percentage of runs where coverage exceeds 90%	Success Rate per Run (%)	SR_{run}
Total number of runs where coverage exceeds 90%	Success Count per Run	$N_{success}$
Mean coverage-per-test over successful runs	Mean Efficiency (at Success)	$\bar{E}_{success}$
Mean coverage of successful iterations	Mean Coverage (at Success)	$\bar{C}_{success}$
Percentage of test scenarios achieving > 90% coverage in at least one instance	Scenario Achievement (%)	$SA_{\%}$

3.5 Individual Subject Evaluation

Results on ArrayStatAnalyzer Findings. Table 4 summarizes outcomes over repeated runs on *ArrayStatAnalyzer* (mean with min–max in parentheses). Overall, the proposed configuration achieves higher line coverage with fewer generated tests than the alternatives. On average, it reaches 97.90% line coverage with 9.0 tests, whereas Generator-Critic attains 95.41% coverage but requires 21.0 tests to do so. Reflection and Pure LLM are competitive in coverage (typically around 95–96%) with roughly 12 tests on average, but they do not match the proposed method's combination of higher coverage and lower test count. Oracle correctness is also highest for the proposed method across runs (42.57% on average), compared to 14.37% for Generator-Critic and approximately 24–25% for Reflection and Pure LLM, indicating that the coverage and test-count advantages do not coincide with degraded oracle quality on this subject.

Table 2. Overall performance comparison of the proposed architecture against baselines.

Approach	SR_{run} (%)	$N_{success}$	$\bar{E}_{success}$	$\bar{C}_{success}$	$SA_{\%}$ (%)
Proposed	100	35	9.84	94.91	100
Generator-Critic	71.42	25	6.46	95.71	85.71
Reflect	51.42	18	8.10	94.72	71.47
Pure LLM	60	25	8.01	92.55	71.47

Table 3. Average coverage, efficiency, and accuracy across all evaluated scenarios.

Approach	Avg. Cov. (%)	Avg. Eff.	Avg. Acc. (%)
Proposed	95.06	9.82	47.30
Generator-Critic	90.54	5.95	43.84
Reflect	87.41	7.28	52.90
Pure LLM	88.02	7.64	48.68

Results on BankAccount Findings. Table 5 summarizes outcomes over repeated runs on *BankAccount* (mean with min–max in parentheses). Overall, the proposed configuration achieves the highest average line coverage (96.61%) while generating substantially fewer tests than Generator-Critic (10.4 vs. 25.2 on average). Reflection attains moderate coverage (87.31%) with a stable test count (14–15), but remains below the proposed method in coverage while requiring more tests on average. Pure LLM exhibits the largest variance across runs (73.17–90.24% coverage) and fluctuates strongly in test count (6–13), making its overall performance less consistent. Oracle correctness is highest on average for Generator-Critic (46.85%) but varies substantially across runs (10.71–73.91%), whereas the proposed method provides a more stable oracle-correctness range (33.34–58.33%) alongside strong and consistent coverage on this subject.

Results on InsurancePremiumCalculator Findings. Table 6 summarizes outcomes over repeated runs on *InsurancePremiumCalculator* (mean with min–max in parentheses). The proposed configuration achieves high coverage with fewer tests on average than Generator-Critic (10.8 vs. 17.6), while maintaining comparable coverage levels (94.07% vs. 95.17%). Pure LLM typically requires the largest number of tests (20.4 on average) and, although it reaches high best-case coverage (98.15%), it does so with consistently higher test counts. Reflection shows the widest variability across runs, with coverage ranging from 59.26% to 92.59% and oracle correctness ranging from 14.23% to 91.30%, indicating inconsistent behavior under this subject. We also observed loop behavior in

Table 4. ArrayStatAnalyzer results over 5 runs (mean with min–max in parentheses).

Approach	**LineCov (%)**	**#Tests**	**OracleCorrect (%)**
Proposed	97.90 (94.74–100)	9.0 (7–10)	42.57 (30.00–55.55)
Generator-Critic	95.41 (93.44–100)	21.0 (19–23)	14.37 (13.04–20.00)
Reflection	95.44 (94.74–96.49)	12.2 (12–13)	24.23 (23.07–25.00)
Pure LLM	96.49 (96.49–96.49)	12.2 (11–14)	24.81 (21.42–27.27)

Table 5. BankAccount results over 5 runs (mean with min–max in parentheses).

Approach	LineCov (%)	#Tests	OracleCorrect (%)
Proposed	96.61 (92.68–97.56)	10.4 (9–12)	43.61 (33.34–58.33)
Generator-Critic	94.67 (91.11–95.56)	25.2 (19–32)	46.85 (10.71–73.91)
Reflection	87.31 (82.93–90.24)	14.2 (14–15)	36.66 (33.34–42.85)
Pure LLM	83.90 (73.17–90.24)	11.0 (6–13)	41.12 (30.76–66.67)

Table 6. InsurancePremiumCalculator results over 5 runs (mean with min–max in parentheses).

Approach	LineCov (%)	#Tests	OracleCorrect (%)
Proposed	94.07 (92.59–96.30)	10.8 (9–13)	24.43 (20.00–33.34)
Generator-Critic	95.17 (89.66–96.55)	17.6 (15–23)	25.42 (17.65–37.50)
Reflection	83.70 (59.26–92.59)	12.6 (6–23)	50.10 (14.23–91.30)
Pure LLM	93.70 (92.59–98.15)	20.4 (16–24)	32.17 (25.00–40.00)

some Generator-Critic runs (marked as "loop" in run logs), which coincides with increased test counts on this subject.

Table 7. LeapYear results over 5 runs (mean with min–max in parentheses).

Approach	LineCov (%)	#Tests	OracleCorrect (%)
Proposed	99 (95–100)	7.0 (7–7)	100 (100–100)
Generator-Critic	100 (100–100)	8.2 (8–9)	100 (100–100)
Reflection	100 (100–100)	15.2 (10–18)	100 (100–100)
Pure LLM	90 (90–90)	13.2 (6–15)	100 (100–100)

Results on LeapYear Findings. Table 7 summarizes outcomes over 5 runs (mean with min–max in parentheses). Generator-Critic, Reflection, and Proposed all reach near-perfect coverage on this subject, with Generator-Critic and Reflection achieving 100% line coverage on average, while Proposed attains 99% on average (95–100). In terms of test count, Proposed is the most compact and stable configuration (7 tests in all runs), whereas Baseline requires slightly more tests on average (8.2; 8–9). Reflection maintains full coverage but at a substantially higher test cost and variability (15.2 tests on average; 10–18). Pure LLM underperforms in coverage (90% in all runs) and shows large variability in the number of generated tests (6–15). Oracle correctness is 100% for all approaches across all runs, indicating that the observed differences are driven primarily by coverage and test-count behavior rather than oracle quality.

Results on OrderProcessor Findings. Table 8 summarizes outcomes over 5 runs on *OrderProcessor* (mean with min–max in parentheses). Overall, the proposed configuration achieves the highest average line coverage (92.23%) but requires more generated tests than Reflection and Pure LLM (22.0 vs. 15.4 and 15.2 on average). Generator-Critic attains notably lower and less stable coverage (73.09%; 67.26–83.19) while also producing the largest number of tests on average (27.6; 22–32). Reflection generates relatively few tests (15.4; 14–17) but remains below the proposed method in coverage (68.49% on average); however, it yields the highest oracle correctness (37.46%; 33.00–50.00). Pure LLM reaches

Table 8. OrderProcessor results over 5 runs (mean with min–max in parentheses).

Approach	LineCov (%)	#Tests	OracleCorrect (%)
Proposed	92.23 (91.37–93.53)	22.0 (19–25)	11.67 (5.26–15.00)
Generator-Critic	73.09 (67.26–83.19)	27.6 (22–32)	13.39 (10.00–20.83)
Reflection	68.49 (64.03–73.38)	15.4 (14–17)	37.46 (33.00–50.00)
Pure LLM	74.10 (66.91–77.70)	15.2 (13–18)	23.68 (0.22–43.75)

moderate coverage (74.10%; 66.91–77.70) with a similarly low test count (15.2; 13–18), but its oracle correctness is highly variable (0.22–43.75), suggesting inconsistent oracle quality across runs.

Table 9. PayrollCalculator results over 5 runs (mean with min–max in parentheses).

Approach	LineCov (%)	#Tests	OracleCorrect (%)
Proposed	93.22 (91.53–96.61)	13.6 (11–15)	30.74 (26.66–35.71)
Generator-Critic	92.61 (87.69–96.92)	20.0 (15–26)	27.71 (12.63–56.25)
Reflection	86.44 (83.05–89.83)	11.2 (10–12)	46.84 (41.66–60.00)
Pure LLM	87.45 (84.75–89.83)	10.0 (9–11)	43.95 (40.00–45.45)

Results on PayrollCalculator Findings. Table 9 summarizes outcomes over 5 runs on *PayrollCalculator* (mean with min–max in parentheses). Overall, the proposed configuration achieves the highest average line coverage (93.22%; 91.53–96.61) but requires more tests than Reflection and Pure LLM (13.6 vs. 11.2 and 10.0 on average). Generator-Critic reaches a similar average coverage (92.61%; 87.69–96.92) but with a higher test count (20.0; 15–26) and includes runs marked as "loop" in the logs, indicating unstable exploration behavior. Reflection produces a relatively small and stable number of tests (11.2; 10–12) and attains lower coverage (86.44%), yet it yields the highest oracle correctness on average (46.84%; 41.66–60.00). Pure LLM achieves comparable coverage to Reflection (87.45%; 84.75–89.83) with the fewest tests (10.0; 9–11) and shows a relatively tight oracle-correctness range (40.00–45.45).

Table 10. Triangular Classifier results over 5 runs (mean with min–max in parentheses).

Approach	LineCov (%)	#Tests	OracleCorrect (%)
Proposed	92.38 (90.48–95.24)	6.4 (6–7)	78.09 (66.66–85.71)
Generator-Critic	82.85 (71.43–90.48)	10.4 (8–12)	79.13 (55.55–100)
Reflection	90.48 (90.48–90.48)	8.0 (8–8)	75.00 (75.00–75.00)
Pure LLM	90.48 (90.48–90.48)	8.0 (8–8)	75.00 (75.00–75.00)

Results on Triangular Classifier Findings. Table 10 summarizes outcomes over 5 runs on *TriangleClassifier* (mean with min–max in parentheses). Overall, the proposed configuration achieves the highest average line coverage (92.38%; 90.48–95.24) while also generating the fewest tests on average (6.4; 6–7). Reflection and Pure LLM reach stable coverage (90.48% in all runs) with a fixed test count (8), but they remain slightly below the proposed method in coverage and require more tests. Generator-Critic is the least stable approach, with coverage

ranging from 71.43% to 90.48% and a higher average test count (10.4; 8–12). Oracle correctness is comparable across methods on average, but Generator-Critic varies widely (55.55–100), whereas the proposed method is more consistent (66.66–85.71) and Reflection/Pure LLM remain constant at 75%.

Validity considerations: LLM-driven components are non-deterministic; we therefore fix prompts/constraints and report best/worst/average results over repeated runs. Moreover, we use JaCoCo line coverage as the primary metric, which does not directly imply fault detection. Finally, our subjects are small algorithmic programs, so generalization to large industrial systems remains limited.

Artifact Availability. To facilitate reproducibility and support further research, the source code of the proposed multi-agent architecture, including the JaCaMo configuration files and the Java code for the evaluation scenarios, is publicly available at: https://github.com/Venuin/llm-bdi-testgen.

4 Discussion

Why Agentic Autonomy Matters: Stability and Determinism. The experimental findings demonstrate that the agentic formulation is not merely a conceptual wrapper but a critical driver of reliability and completeness in LLM-based unit test generation. Our results indicate that while baseline approaches—such as Pure LLM and Reflection—exhibit significant stochastic volatility and often fail to reach targets in 40–50% of executions, the Proposed MAS achieves a consistent 100% Success Rate per Run (SR_{run}) and Scenario Achievement ($SA_{\%}$) across all evaluated benchmarks. This deterministic performance is a direct consequence of the Orchestrator's goal-directed search process, which maintains an explicit global objective and enforces a bounded BDI (Belief-Desire-Intention) reasoning cycle grounded in executable feedback rather than unconstrained generative prompting.

Single-pass prompting and loosely guided iterative prompting tend to waste effort on redundant trials and often fail to systematically reach deep or boundary-heavy branches. In contrast, our architecture justifies each iteration by a concrete uncovered path rather than free-form prompt exploration. Moreover, while simpler configurations may exhibit non-terminating exploration behavior— characterized by repeated iterations without coverage progress—our proposed version prevents such stagnation through progress-aware target transitions and a strict reflection budget. The high statistical significance observed in the Friedman test ($p < 0.001$) and the large effect sizes yielded by the Wilcoxon post-hoc analysis ($A_{12} > 0.8$) further support that this stability is inherent to the system's design rather than a result of non-deterministic fluctuations. This goal-conditioned control effectively transforms automated testing from a stochastic generative task into a disciplined engineering process capable of reliable convergence on complex logic paths.

From Opaque Generation to Explainable Causal Grounding. By replacing speculative revisions with concrete symbolic requirements, our architecture ef-

fectively addresses the "black-box" nature of standard LLM-based testing. While typical generative approaches produce test inputs through opaque probabilistic patterns, our neuro-symbolic feedback loop ensures that each generated input is causally linked to a specific, identified predicate requirement. This provides a transparent and observable relationship between the source code's logic and the resulting test data, making the input space's causality explicitly traceable through Logic Hints.

Efficiency and the "Coverage Plateau" Phenomenon. The Proposed MAS demonstrated significantly higher average efficiency ($\bar{E}_{success} = 9.84$) compared to the Generator-Critic baseline (6.46), highlighting a "coverage plateau" in unguided prompting where models fail to resolve deep branches despite repeated iterations. While baselines often stagnate in redundant feedback loops, our architecture utilizes the `LogicMapArtifact` to expose the symbolic structure of predicates. The large effect size ($A_{12} > 0.8$) confirms that transforming coverage misses into deterministic Logic Hints effectively narrows the search space, allowing the agents to resolve complex branches with minimal trial-and-error.

Strategic Separation of Concerns: Balancing Coverage and Correctness. Our experimental observations reveal a fundamental engineering trade-off between structural coverage attainment and test oracle correctness (semantic cohesion). Pure LLM and Reflection-based approaches tend to co-generate test inputs and expected outputs within a unified reasoning chain. This "co-generation advantage" allows the model to exhibit high semantic consistency in simple scenarios of its own making; however, this success is typically limited to a shallow search space and results in lower overall coverage.

In contrast, our proposed MAS architecture deliberately decouples the exploration and synthesis processes. While the *Analyzer* and *Orchestrator* agents focus on resolving complex predicates and deep logic paths using symbolic Logic Hints, the *Generator* agent constructs the final test suite grounded in these verified inputs. This decoupling is the primary driver enabling our system to reach deep logic branches that remain inaccessible to baselines. Although the "semantic distance" between input discovery and oracle synthesis may lead to a minor accuracy trade-off compared to purely generative models, this represents a strategic engineering decision. In software testing, identifying an uncovered logic branch (exploration) is a significantly more challenging problem than refining the expected output of an existing input. Consequently, our architecture adopts this staged design to prioritize high-coverage test generation grounded in deterministic execution feedback—validated by the *Executor*—rather than speculative generation.

Limitations and future work. This study is subject to several limitations, primarily its focus on small-scale algorithmic Java subjects and its reliance on line coverage, which—while standard—is weaker than criteria such as branch coverage and mutation adequacy. The evaluation also does not yet include direct empirical comparisons with mature non-LLM generators such as symbolic/concolic execution tools, Randoop, or EvoSuite. While the proposed staged architecture significantly enhances exploration depth, it introduces a minor semantic trade-off

in oracle accuracy due to the decoupling of path exploration and test synthesis. Future research will focus on extending the framework with stronger adequacy metrics, such as branch coverage and mutation analysis, and validating the approach on complex industrial systems and established test-generation baselines. Furthermore, we intend to strengthen semantic integrity by augmenting the intent inference process with lightweight specifications (e.g., contracts) and developing adaptive reflection budgeting strategies to optimize agent performance.

5 Conclusion

This paper proposed a hybrid neuro-symbolic multi-agent architecture for LLM-based unit test generation on JaCaMo, where artifact-mediated feedback turns coverage misses into symbolic Logic Hints. By treating test generation as an agentic, goal-directed search process, the approach reduces redundant trial-and-error compared to prompt-centric baselines and reliably reaches the target line coverage on small Java subjects, while preserving executability as a sanity check.

More broadly, this study suggests that integrating deliberative BDI-based agent control with neuro-symbolic feedback mechanisms provides a principled pathway for transforming LLMs from probabilistic text generators into structured reasoning components within dependable software engineering workflows.

Future work includes evaluating on larger codebases with stronger adequacy measures (branch coverage, mutation score), reporting cost metrics, and strengthening oracles via lightweight specifications and adaptive reflection budgets.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Alshahwan, N., et al.: Automated unit test improvement using large language models at meta. In: FSE (2024)
2. Ba, T., et al.: Covernexus: Multi-agent llm system for automated code coverage enhancement. In: ICT. Springer (2025)
3. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Jacamo: A programming framework for multi-agent systems. Software: Practice and Experience **43**(5), 615–641 (2013). https://doi.org/10.1002/spe.2108
4. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. Wiley Series in Agent Technology, John Wiley & Sons (2007)
5. Botti, V.: Agentic ai and multiagentic: Are we reinventing the wheel? In: VRAIN (2025)
6. Feldt, R., Kang, S., Yoon, J., Yoo, S.: Towards autonomous testing agents via conversational large language models (2023), arXiv preprint arXiv:2306.05152v2
7. Fraser, G., Arcuri, A.: 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. pp. 416–419. ACM (2011). https://doi.org/10.1145/2025113.2025179

8. Garlapati, A., Parmesh, M.N.V.S.S.M.: Ai-powered multi-agent framework for automated unit test case generation. In: GCAT (2024)
9. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation. pp. 213–223. ACM (2005). https://doi.org/10.1145/1064978.1065036
10. Jin, H., Huang, L., Cai, H., Yan, J., Li, B., Chen, H.: From llms to llm-based agents for software engineering: A survey (2025), arXiv preprint arXiv:2408.02479v2
11. Kim, M., et al.: A multi-agent approach for rest api testing with semantic graphs and llm-driven inputs. In: ICSE (2025)
12. Pacheco, C., Ernst, M.D.: Randoop: Feedback-directed random testing for Java. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion. pp. 815–816. ACM (2007). https://doi.org/10.1145/1297846.1297902
13. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proceedings of the 29th International Conference on Software Engineering. pp. 75–84. IEEE Computer Society (2007). https://doi.org/10.1109/ICSE.2007.37
14. Ricci, A., Omicini, A., Viroli, M.: Cartago: A framework for prototyping artifact-based environments in mas. In: International Workshop on Environments for Multi-Agent Systems (E4MAS). pp. 67–86. Springer (2006)
15. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 263–272. ACM (2005). https://doi.org/10.1145/1081706.1081750
16. Siddiq, M., et al.: Using large language models to generate junit tests: An empirical study (2024), arXiv preprint arXiv:2305.00418v4
17. Tyynelä, J.: Llm-based agents in software testing: A systematic mapping study (2025)
18. Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., Wang, Q.: Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering (2024)
19. Wang, Y., Zhong, W., Huang, Y., Shi, E., Yang, M., Chen, J., Li, H., Ma, Y., Wang, Q., Zheng, Z.: Agents in software engineering: Survey, landscape, and vision (2024), arXiv preprint arXiv:2409.09030v2
20. Yoon, J., Feldt, R., Yoo, S.: Intent-driven mobile gui testing with autonomous large language model agents. In: ICST (2024)