Automated Black-Box Testing: A Comparative Study of LLM Agent Architectures and Prompt Engineering

Automated Black-Box Testing: A Comparative Study of LLM Agent Architectures and Prompt Engineering

Anna Arnaudo
Department of Control and Computer Engineering (DAUIN)
Politecnico di Torino
Torino, IT
anna.arnaudo@polito.it

Riccardo Coppola
DAUIN
Politecnico di Torino
Torino, IT

Flavio Giobergia
DAUIN
Politecnico di Torino
Torino, IT

Maurizio Morisio
DAUIN
Politecnico di Torino
Torino, IT

Van-Thanh Nguyen
Politecnico di Torino
Torino, IT

Enrico Chen
Politecnico di Torino
Torino, IT

Xiaoning Ma
Politecnico di Torino
Torino, IT

XiaoQuan Ji
Politecnico di Torino
Torino, IT

Minh-Thai Mai
Politecnico di Torino
Torino, IT

***Abstract*—Automated black-box test case generation remains challenging due to diverse implementation behaviours and complex boundary conditions. This work investigates the effectiveness of *multi-agent* Large Language Model (LLM) architectures employing collaborative and competitive interaction patterns, compared against a single-agent baseline, within the context of function-level black-box testing. Experiments involving the HumanEval benchmark show that prompt engineering is the primary driver of performance, with *rule-augmented few-shot* prompting yielding improvements up to $20 - 30\%$ in both coverage and Execution Success Rate (ESR) against all the other tested strategies — i.e., *zero-shot* and conventional *few-shot* prompting. Although multi-agent architectures achieve superior test coverage — peaking at 99.54% — they yield an ESR comparable to single-agent frameworks (96.98% versus 96.89%). Crucially, this comes at the expense of a threefold to fourfold increase in token expenditure. In conclusion, the *single-agent* configuration paired with optimised *rule-augmented few-shot* prompting provides the most effective balance between accuracy, coverage, and computational efficiency.**

***Index Terms*—Software testing, Black-box Testing, Large Language Models, Prompt Engineering, Software Engineering**

I. Introduction

In software engineering, testing is essential to ensure that systems are reliable and work as intended. It also supports teamwork by making it easier to check code and stop old bugs from reappearing. While some tests need to be done manually — especially when performed on complex visual interfaces — automated testing is more scalable, since scripts can run automatically every time it is needed. However, creating automated tests is challenging, as many different scenarios needs to be covered to truly check the code.

Software testing often requires collaboration among developers, testers, domain experts, and automated tools, each contributing complementary perspectives. Developers focus on implementation choices, testers on edge cases and failure scenarios, and domain experts on realistic usage constraints. Effective testing depends on integrating these viewpoints.

The term 'black-box testing' refers to a software testing method in which the internal implementation of the Code Under Test (CUT) is not considered during the definition of the test cases. It concentrates only on the input and output of the system, offering an unbiased measure of the system's robustness and reliability. In the case of unit tests — which are the focus of the present work — the test developers know only the function signatures and the proposed objectives of the CUT. Most automated test generation tools rely on a single reasoning strategy, limiting their ability to capture diverse behaviours, edge cases, and domain-specific knowledge. This limitation affects not only large software systems but also individual functions, where subtle bugs can arise from boundary conditions or uncommon execution paths. Consequently, the effectiveness of function-level testing strongly depends on the diversity and coverage of the generated test cases.

Recent advances in Large Language Models (LLMs) enable new approaches to automated test generation. Specifically, in *multi-agent* architectures multiple LLMs can collaborate, each playing a different role — such as developer, tester, or domain expert. By combining complementary viewpoints, multi-agent systems have the potential to produce more comprehensive and diverse test sets compared to single-agent approaches.

In this work, we build upon the QA Agent framework proposed by Deo et al. [1], a multi-agent system for unit test generation. We extend this framework by modifying the prompting strategies and proposing new agent interaction mechanisms. Like in the original work, our evaluation focuses on function-level test generation using the HumanEval benchmark under black-box testing conditions.

We evaluate the quality of the generated suites by executing the tests against reference code implementations available on the HumanEval dataset, and by measuring line coverage and

tests Execution Success Rate (ESR).

We compare our *multi-agent* configurations — employing either collaborative or competitive interaction patterns — with a *single-agent* baseline, aiming to determine whether the improvements in ESR and code coverage offered by multi-agent patterns justify their increased computational overhead. Furthermore, we quantify the extent to which advanced prompt engineering can bridge the performance gap between these configurations.

The contributions of this work can be summarised as follows:

- We propose and assess multiple *multi-agent* architectures and collaboration patterns;
- We demonstrate the importance of the prompting strategy across all the tested architectures;
- We improve the performance metrics achieved by the reference work [1].

The implementation of the proposed system is publicly available on Zenodo.[1]

II. Related Work

Being trained on massive amounts of data, Large Language Models (LLMs) show remarkable capabilities in understanding, reasoning, and generating language. They can be used in a huge variety of *Software Engineering* tasks like code generation and test case generation. Being real-world problems inherently complex and requiring experts in multiple domains, performance of a *single LLM-based agent* may be limited. Therefore, the introduction of *multi-agent* systems marks an important evolution in the field: by collaborating with each other and having distinct roles and responsibilities, agents can solve complex tasks [2] with improved factuality [3]. *Multi-agent* systems can exploit different collaboration patterns [4]. For example, they can be *Cooperative* — in which agents interact with each other to reach a common goal — or *Competitive* — in which agents compete to reach the same goal, with eventual debate [5].

In the last years, substantial research has been conducted on multi-agent systems applied to test case generation. In the present work, we aim to expand the research conducted by Deo et al. [1], which introduces a *multi-agent* system — namely, QA Agent — where a *code architect agent* generates the function implementation plan in natural language and pseudocode, and a *test generator agent* generates the test cases. Our system extends QA Agent by broadening the evaluation of prompting strategies and introducing parallel execution flows in place of a linear pipeline.

Hong et al. [6] simulates an entire software company where different agents cooperate. Specifically, there are 5 different roles: the *Product manager* creates user stories given user requirements, the *Architect* creates software components, the *Project manager* distributes the tasks, the *Engineer* develops code, and the *Quality Assurance (QA) engineer* develops tests. In contrast, this work focuses only on test generation under black-box conditions.

Qian et al. [7] proposed a *multi-agent* system used for the design, coding, and testing phases of software development. Each task is divided into subtasks with the respective agents — i.e., the *instructor* and the *assistant* — working in a collaborative setup. Specifically, the *assistant* asks clarifications to the *instructor* in order to solve problems of factual correctness due to hallucinations [8]. In addition to the study conducted by Qian et al. [7], we assess *competitive* collaboration paradigms and compare them directly with different collaborative approaches.

Huang et al. [9] introduces a cooperative system with 3 agents: a *programmer*, a *test designer*, and a *test executor*. The first is an LLM that creates functions according to human specifications using the Chain-Of-Thought (CoT) technique. The second is an LLM with the task of generating test cases. Finally, the *test executor* is not an LLM, and tests the given code snippets using the relative test cases. If the execution generates some failures, then the information is given back to the *programmer* to refine the code. In contrast, we focus only on test generation. Moreover, we avoided integrating feedback loops because we are focusing on black-box testing. Including them would have forced the system to look at the internal code, which defeats our goal.

Other works propose an architecture based on a feedback loop mechanism. Pan et al. [10] use the results of static analysis to guide LLMs in generating test cases. Another work in which LLMs are guided by the measures from the previous iteration is that presented by Altmayer et al. [11]: in this case, the prompt contains coverage information. Finally, Wang et al. [12] incorporates mutation feedback into the prompt, resulting in improved performance over standard prompt-based techniques.

III. Methodology

This section details the architectures evaluated in our study, spanning from a baseline single-agent configuration to sophisticated multi-agent collaborative and competitive frameworks.

A. Research Questions

In order to evaluate the capabilities of multi-agent LLM architectures in generating high-quality test cases, we address the following research questions (RQs):

- **RQ 1**: How accurate and efficient are LLM agents in generating comprehensive and accurate function-level test cases in a black-box setting?
- **RQ 2**: Does collaboration among multiple LLM agents improve function-level black-box test case generation compared to single-agent?
- **RQ 3**: What is the effectiveness of the prompting strategy adopted for function-level black-box test case generation?

B. Single-agent Baseline

The single-agent approach serves as our control, allowing us to isolate the impact of agent collaboration in subsequent

[1]https://zenodo.org/records/19524022

experiments. In this architecture, a single model instance is prompted to generate a comprehensive test suite without intermediary planning, as described below:

Problem Description → *Test Generation* → *Output Test Cases* → *Evaluation*

We evaluated this simple pipeline by considering either zero-shot, few-shot or rule-augmented few-shot as prompting strategies.

C. Multi-agent System

This section details the *multi-agent LLM* system designed for automated test case generation. The architecture leverages two distinct agents to decompose test generation into two sub-tasks: planning and execution. This mimics a real-world software development workflow in which a senior architect designs the solution (pseudocode/plan) and a developer (tester) implements the specific test cases based on that design. It has been inspired by the original QA Agent [1] — already described in Section II — by introducing two additional *Architect-Generator* pipelines, whose outputs are subsequently processed either in a *Collaborative* or *Competitive* pattern. Furthermore, we explore the impact of a new rule-augmented prompt.

The execution flow can be schematised as follows:

Problem Description → *Code Architect* → *Output Plan* → *Test Generator* → *Output Test Cases* → *Evaluation*

Where:

1) **Problem Description** is a coding problem definition (e.g., docstrings from HumanEval);
2) *Code Architect* is the first agent. It analyses the problem and produces a natural language plan or pseudocode (the output format depends on the prompting strategy, as detailed below);
3) **Test Generator** is the second agent, which uses the original problem and the generated plan to write executable Python unit tests;
4) **Evaluation** involves executing the generated tests on the CUT, and collecting performance metrics (described in Section IV-B).

In the next sections, we introduce new approaches that extend this pipeline by introducing collaborative and competitive interaction between agents. Specifically, the proposed architecture includes three pipelines, each utilising a different prompting strategy for the **Code Architect** — i.e., Chain-of-Thought (CoT), ReAct, or few-shot, as described in Section III-D. Observations indicate that, contingent upon the strategy, the *Code Architect* prioritises either architectural planning in natural language or pseudocode.

The outputs from the *Code Architects* inform a different *Test Generator*, which synthesises the corresponding test suite. The *Test Generator* agents across the three branches are configured using a uniform prompting strategy, as detailed in Section III-D.

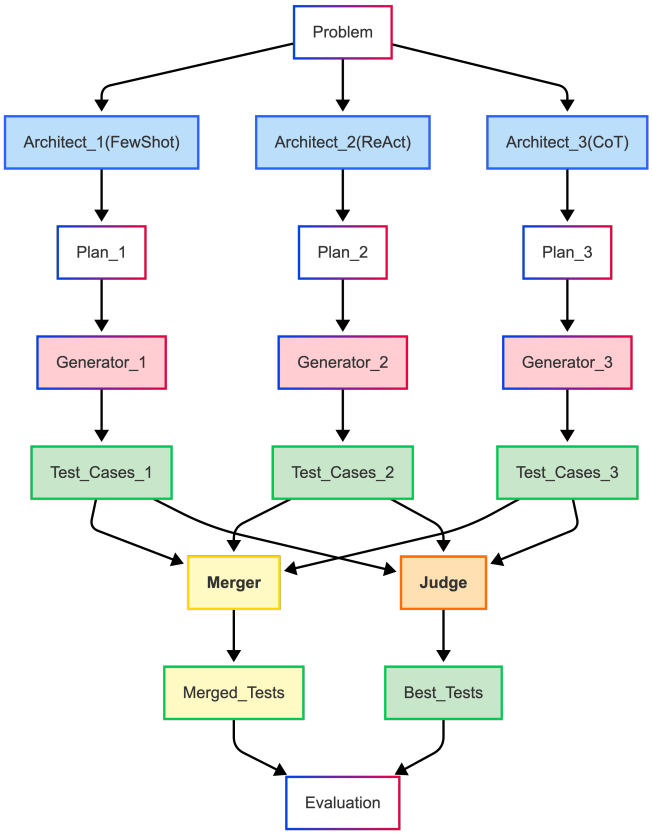

Fig. 1. Schema of the proposed multi-agent architecture

1) Multi-agent Competitive Architecture: In the competitive multi-agent paradigm, the system leverages diversity by executing test generation pipelines in parallel, each operating autonomously to produce a distinct test suite. The final output is determined by a winner-take-all selection process. This workflow is illustrated in Figure 1. Specifically, the outcome of this approach is the one finally outputted by the *Judge* node. The **Judge** agent evaluates the suites produced by the three parallel flows, and selects the best for the final system's output. This strategy allows to select the most robust candidate while avoiding the logical conflicts or redundancies that can arise from merging disparate codebases.

To better investigate the effectiveness of the *Judge*, we explore two different implementations of this agent:

- **Competitive strategy with LLM Selector**: the three test suites are given altogether as input to the *Judge*, which can directly compare them and indicate the best one;
- **Competitive strategy with LLM Scorer**: the *Judge* gives a score to each candidate independently, and the highest score determines the final output.

In both cases, the *Judge* is provided with a list of rules — reported in Table II — which have been empirically derived and serve as guidelines for the evaluation. Future work should assess more systematically the effect of these rules on the final system's output.

2) Multi-agent Collaborative Architecture: The collaborative strategy adopts a different interaction paradigm, based on a centralised reconciliation phase rather than selection. The implementation is schematised in Figure 1. A *Merger* node has been inserted into the architecture to produce the output of the collaborative approach.

We evaluate three distinct strategies for reconciling the test cases produced by the three parallel pipelines:

- **Concatenation**: This strategy aggregates all generated test cases into a single suite. It prioritises the volume and diversity of the resulting set.
- **Accuracy-Based**: This approach prioritises logical consistency, filtering tests with syntax errors, wrong function names, or duplicates — which are detected via Abstract Syntax Tree (AST) comparison.[2]
- **LLM-based**: A *Merger* agent is prompted to perform the merge of the test suites produced by the three precedent parallel flows. Specifically, the prompt includes a set of rules — reported in Table II — and two *few-shot* prompts.

D. Prompt Engineering

We utilised *Role-prompting* to delineate responsibilities within the multi-agent collaboration. Furthermore, we compared various role-specific prompting configurations as detailed below. Some examples are reported in Table II, while the specific roles are described in Section III-C.

For the **Code Architect (or Planner)**, we consider three prompting styles:

- Basic Chain-of-Thought (CoT)[3] without examples;
- CoT anchored with few-shot examples;
- ReAct-style workflow[4] with few-shot examples.

This enables the evaluation and section between diverse plans, thus increasing the possibility of finding the optimal solution [15]. We observed variability across planners, including differences in verbosity and structure. For instance, given the task of *extracting the decimal part from a positive float* — the HumanEval problem task #2 — we obtained the plans reported in Table I.

TABLE I
COMPARISON OF FLOAT DECIMAL EXTRACTION PLANS.

Prompting Strategy	Generated Plan
Few-Shot	*(1) Extract integer part by casting to* `int`*; (2) Subtract from original to get fractional part; (3) Return result.*
ReAct	*Thought: Truncate to find the integer part.* *Act: Subtract it from the original number and return the decimal part.*
CoT	*(1) Identify integer part (e.g.,* `int()` *or* `math.floor`*); (2) Subtract from original; (3) Return result as decimal component.*

[2]Abstract Syntax Tree (AST) comparison for software functions involves parsing source code into tree structures and analysing the differences between those trees rather than the raw text [13].

[3]Chain of Thought (CoT) is a a prompting technique that encourages the LLM to break down a complex problem into a sequence of intermediate logical steps.

[4]In prompt engineering, the ReAct-style workflow enforces an explicit 'Thought-Act-Observe' procedure, making the LLM reason before acting [14].

For the **Test Generator Agent**, we consider two prompting strategies:

- The *Standard few-shot* configuration includes two examples of related tasks, which have been sourced from the work of Deo et al. [1]. Including the examples in the prompt helps to teach the LLM how to generate assert-based test cases from given function descriptions;
- The *rule-augmented few-shot* configuration further extends the previous approach by incorporating a list of detailed guidelines — such as rules for numeric precision, boundary logic, and deterministic outputs — along with few-shot examples.

IV. EXPERIMENTAL SETUP

This section describes the experimental environment used to evaluate and compare the different approaches listed in Section III.

A. Datasets and Tasks

The framework is evaluated using the HumanEval benchmark dataset[5], which comprises 164 Python programming tasks. Each item includes a function signature, a descriptive docstring, a reference (canonical) Python implementation, and the associated unit tests. Notably, we do not use the provided unit tests; instead, the canonical implementation is designated as the Code Under Test (CUT). By utilising only the signatures and docstrings as input, the proposed architecture generates tests within a black-box paradigm: the CUT remains decoupled from the system until the final evaluation stage.

To ensure both depth and breadth in our evaluation, we employ two experimental configurations. First, we use a *curated* subset of 20 HumanEval problems, spanning from basic arithmetic to advanced logic validation. These tasks are categorised as follows: Basic (4 tasks), Intermediate (6 tasks), Complex (6 tasks), and Advanced (4 tasks). The code used to classify the tasks based on their difficulty is available in the online repository. It leverages the counting of the Lines Of Code (LOC), the loop structures and the if-branches. Each task is executed 10 times per strategy, improving statistical robustness and minimising the impact of variance in model outputs. This curated set allows us to test our system on a dataset with a balanced distribution of task difficulties.

The second experimental setup is based on the full set of 164 HumanEval problems, executed 10 times per strategy. This allows us to evaluate the generalisability of our approach across a wider range of programming tasks.

Table III reports the distribution of the difficulties within the two versions of the HumanEval dataset according to the adopted classification criteria.

B. Evaluation Metrics

We assess the quality and efficiency of the generated test suites using three metrics:

[5]https://github.com/openai/human-eval

TABLE II
EXAMPLES OF PROMPTS USED WITH DIFFERENT AGENTS OF OUR ARCHITECTURE. THE FULL SET CAN BE FOUND IN THE ONLINE REPOSITORY

Code Architect - ReAct prompting

```
You are a software programmer. You are required to plan out how you will write the function in pseudocode.

To ensure accuracy and logical flow, you must use the **ReAct paradigm**:
1. **Thought**: First, reason about the problem. Analyze the input/output, edge cases, and
the algorithmic logic required.
2. **Act**: Then, generate the actual plan using the specific pseudocode format below.
```

Merger agent for the Collaborative architecture

```
You are a software tester. Your task is to merge multiple test sets into a single, high-quality test set.

First, identify and remove any test that has a syntax error:
- Missing parentheses, brackets, or quotation marks
- Invalid Python syntax
- If a test cannot be parsed, remove it

Next, remove tests that use incorrect function names:
- Use ONLY the correct function name as specified by the Entry Point
- Remove any tests calling functions with different names

**Important Rules:**
- Do NOT add new test cases that are not present in any input test set
- Ensure the final output contains only valid, executable Python assert statements
```

Selector (Judge) agent for the Competitive architecture

```
You are an expert in evaluating black-box unit test suites. You will receive one programming problem and multiple
candidate test suites. Your task is to compare all candidates and select the best one.

PREFER suites with strong execution reliability:
- tests are syntactically valid and directly executable as assertions,
- assertions are precise, unambiguous, and deterministic,
- expected outputs are clearly justified by the specification,
- tests cover distinct behaviors, including meaningful edge and boundary scenarios,
- each test adds meaningful signal with minimal redundancy.
PENALIZE suites containing suspicious or fragile tests, including:
- unclear or weak assertions,
- brittle assumptions not grounded in the specification,
- contradictory expectations,
- likely-to-fail syntax/formatting patterns,
- noisy or low-value tests that reduce confidence in reliable execution.
```

TABLE III
DISTRIBUTION OF TASK DIFFICULTIES IN THE CURATED AND THE FULL VERSION OF THE HUMANEVAL DATASET.

Dataset	Basic	Intermediate	Complex	Advanced
Curated HumanEval	20.00%	30.00%	30.00%	20.00%
Full HumanEval	31.10%	25.61%	23.78%	19.51%

1) *Code Coverage*: percentage of the CUT's lines executed during the test, serving as a measure of test thoroughness;
2) *Execution Success Rate (ESR)*: the percentage of tests that run successfully within the generated suite. As the CUT comprises consolidated solutions to programming tasks, its correctness is assumed; consequently, any test failure is interpreted as an error within the test generation procedure;
3) *Token Consumption*: the average total tokens (input and output) processed per task, providing a measure to compare the computational efficiency and cost of each strategy.

The first two have been selected as complementary metrics to evaluate the overall quality of the test code. While *Mutation Testing* would offer an alternative evaluation, it was omitted, as we argue that *Code Coverage* sufficiently quantifies the suite's ability to navigate code nuances within a black-box setting [16].

C. Model Selection

We utilise the *nvidia/nemotron-3-nano-30b-a3b* model, a Mixture-of-Experts (MoE) architecture with $3.5B$ active and $30B$ total parameters.[6] It has been selected for its efficiency and because it has undergone a specialised fine-tuning to generate code and structured outputs. In our implementation, the model is run at a fixed temperature of 0 to improve the determinism and reproducibility of our experiments. By maintaining a single, consistent model across all configurations, we isolate the effects of our prompting and agent coordination strategies, ensuring that performance variations are attributable solely to our methodological variants.

[6]https://build.nvidia.com/nvidia/nemotron-3-nano-30b-a3b/modelcard

TABLE IV

EVALUATION RESULTS ON THE HUMANEVAL CURATED SUBSET (20 PROBLEMS, DESCRIBED IN SECTION IV-A), SORTED BY PROMPT AND STRATEGY. RESULTS REPRESENT THE AVERAGE PERFORMANCE ACROSS 10 INDEPENDENT RUNS. FOR EACH PROMPTING STRATEGY AND METRIC, THE PEAK VALUES ARE IN BOLD.

Strategy	Prompt	Execution Success Rate (%)	Coverage (%)	Tokens/Problem
Single Agent	Zero Shot (Baseline)	**61.55**	**68.03**	**2,372.58**
Single Agent	Standard few-shot	88.38	96.57	**2,287.66**
Multi-agent Competitive (LLM scorer)	Standard few-shot	**93.94**	98.12	19,047.33
Multi-agent Competitive (LLM selector)	Standard few-shot	93.07	98.04	17,783.04
Multi-agent Merge (Accuracy)	Standard few-shot	86.59	91.78	12,278.32
Multi-agent Merge (Concat)	Standard few-shot	90.09	97.05	12,183.05
Multi-agent Merge (LLM)	Standard few-shot	90.86	**98.76**	21,144.74
Single Agent	Rule-augmented few-shot	98	98.44	**4,143.58**
Multi-agent Competitive (LLM scorer)	Rule-augmented few-shot	**98.15**	98.96	23,382.68
Multi-agent Competitive (LLM selector)	Rule-augmented few-shot	97.55	99.11	22,155.49
Multi-agent Merge (Accuracy)	Rule-augmented few-shot	96.57	99.23	17,386.96
Multi-agent Merge (Concat)	Rule-augmented few-shot	96.96	**99.34**	17,666.72
Multi-agent Merge (LLM)	Rule-augmented few-shot	96.50	99.19	24,080.56

TABLE V

EVALUATION RESULTS ON THE FULL HUMANEVAL DATASET (164 PROBLEMS), SORTED BY PROMPT AND STRATEGY. RESULTS REPRESENT THE AVERAGE PERFORMANCE ACROSS 10 INDEPENDENT RUNS. FOR EACH PROMPTING STRATEGY AND METRIC, THE PEAK VALUES ARE IN BOLD.

Strategy	Prompt	Execution Success Rate (%)	Coverage (%)	Tokens/Problem
Single Agent	Zero Shot (Baseline)	**62.17**	**68.23**	**2,149.84**
Single Agent	Standard few-shot	91.11	96.15	**2,400.29**
Multi-agent Competitive (LLM scorer)	Standard few-shot	**93.02**	97.83	17,584.13
Multi-agent Competitive (LLM selector)	Standard few-shot	92.51	**98.39**	16,968.47
Multi-agent Merge (Accuracy)	Standard few-shot	91.50	96.99	11,672.22
Multi-agent Merge (Concat)	Standard few-shot	89.78	96.29	11,609.57
Multi-agent Merge (LLM)	Standard few-shot	91.96	97.65	20,432.18
Single Agent	Rule-augmented few-shot	**96.98**	99.30	**3,843.29**
Multi-agent Competitive (LLM scorer)	Rule-augmented few-shot	96.89	99.25	21,635.40
Multi-agent Competitive (LLM selector)	Rule-augmented few-shot	96.70	99.37	21,154.44
Multi-agent Merge (Accuracy)	Rule-augmented few-shot	96.28	**99.54**	16,279.48
Multi-agent Merge (Concat)	Rule-augmented few-shot	96.41	**99.54**	16,197.11
Multi-agent Merge (LLM)	Rule-augmented few-shot	96.12	99.46	23,078.21

V. EXPERIMENTAL RESULTS

The results of the evaluation on the HumanEval benchmark are summarised in Table IV and Table V — representing the performance on the curated and the full dataset, respectively. Discrepancies between these results arise from differing task difficulty distributions, as reported in Table III, with the curated version being composed by a higher percentage of difficult tasks. However — as the keen reader will have noted — the performances measured on the curated dataset are slightly higher. This may be due to a greater affinity between the selected tasks and the few-shot examples, which may help the model produce more accurate outputs. Future work may repeat the assessment with different subsets of the original dataset.

Nevertheless, we are still able to make inter-table comparisons to illustrate the performance trade-off inherent in various prompting styles and multi-agent strategies.

A. The Primacy of Prompt Engineering

Across all architectural strategies, the *rule-augmented few-shot* prompt consistently outperforms the *standard few-shot* and *zero-shot* configurations. Notably, the shift from *zero-shot* to *rule-augmented few-shot* yields improvements up to $20-30\%$ in both coverage and execution success rates, demonstrating that prompt refinement is critical to performance.

The *single-agent (rule-augmented few-shot)* configuration proves to be the most cost-effective for general-purpose applications, achieving the highest ESR at 96.98% and an impressive code coverage on the full dataset at 99.30%, all while minimising token usage. These results can be attributed to the prompting strategy adopted. As reported in Table V, the same pipeline obtained an ESR of 62.17% and a coverage of 68.23% with zero-shot prompting.

B. Comparison Between Single-agent and Multi-agent Approaches

While the *multi-agent* architectures generally outperformed *single-agent* in terms of coverage, and achieved comparable ESR metrics — both on the curated (Table IV) and the complete dataset (Table V) — they imply a substantial computational overhead. Multi-agent architectures require approximately $3-4$ times the token volume of the single-agent baseline. This identifies a clear trade-off, where marginal gains

in accuracy come at a non-proportional increase in computational cost. Notably, the performance enhancements derived from inter-agent collaboration are most evident in the results related to *standard few-shot*. When comparing single-agent performance against the proposed multi-agent architectures using the same prompting strategy, the ESR increased by up to 5.56% on the curated dataset and 2.91% on the full version, while the highest gain in coverage was 2.19% on the curated and 3.21% for the full dataset.

C. Comparison with the Reference Work

As established in Section III, our system builds upon the one proposed by Deo et al. [1]. It is important to emphasise that their GPT-4 Turbo baseline with few-shot prompting reached an ESR of 76.20% and a coverage of 79.20% — metrics that are markedly superior to the baseline performance recorded in the present study, that employs a different base model. The metrics reported in Table VI demonstrate that the best of our proposed approaches outstrips the reference implementation, thereby advancing beyond existing research. Crucially, these results underscore the methodology's efficacy, since it yields superior final performance notwithstanding the initial handicap of a markedly inferior baseline.

D. Discussion

Based on our experiments, we provide the following answers to our initial research questions:

RQ 1: How accurate and efficient are LLM agents in generating comprehensive and accurate function-level test cases in a black-box setting? Our findings indicate that LLM agents can achieve nearly-perfect performance when generating diverse test cases, provided that the input prompt is optimised. Indeed, the *rule-augmented few-shot* prompting strategy consistently outperforms *zero-shot* and *standard few-shot* configurations. Since the model *nvidia/nemotron-3-nano-30b-a3b* is specifically designed for coding tasks, the *rule-augmented few-shot* prompt already enables a single agent to generate complete and accurate test cases.

However, the bigger limitation affecting our approach is the consideration of a single benchmark dataset, which could negatively affect the external validity of our study.

RQ 2: Does collaboration among multiple LLM agents improve function-level black-box test case generation compared to single-agent? When *standard few-shot* is applied, collaboration demonstrates clear superiority over the baseline. This effect is less evident if we consider *rule-augmented few-shot*, as the prompting technique seems to play a more important role. Moreover, we observe that the performance gains introduced by multi-agent architectures on the coverage metric are more significant than the ones on the ESR.

On the **curated dataset** — having a higher percentage of complex tasks — multi-agent collaboration always enables superior performances compared to the single-agent counterpart.

Specifically, the *Competitive (LLM scorer)* exploits the comparison between different solutions to obtain the highest percentage of passing test; at the same time, the *Collaborative (Concat)* solution, via the aggregation of different contributions, was able to reach the best coverage.

The results are different on the **full dataset** — having a task difficulty distribution shifted towards simplicity. In this case, the engineering of the prompt is enough for the *single-agent* architecture to achieve the maximum ESR — which is however comparable to the values measured when assessing the multi-agent *Competitive* approaches. On this simpler benchmark, the *Merge (Concat)* strategy still obtains the primacy for the coverage metric — accompanied by the *Merge (Accuracy)*, which confirms to be a good design choice when prioritising coverage.

RQ 3: What is the effectiveness of the prompting strategy adopted for function-level black-box test case generation? Prompt engineering alone yielded improvements exceeding 30 points across measured performance metrics. Specifically, adding *few-shot* examples resulted in gains of over 25 percentage points — with both ESR and coverage surpassing 90%. By further incorporating clear rules — in the approach we named *rule-augmented few-shot* — the system reached nearly perfect scores. Future work may investigate in more depth the isolate effect of the inclusion of the rules without shot examples.

VI. Conclusion

In this work, we explore the effectiveness of different architectures in automated black-box test generation. Our analysis indicates a clear hierarchy of needs. We observed that the advantages delivered by agents collaboration tend to be more evident especially for the coverage metric. However, the addition of structured rules into the prompts seems to play a more crucial role in improving performance, as the single-agent architecture with *rule-augmented few-shot* prompting already achieves nearly-perfect metrics — and even outperforms multi-agent in one case. Regarding token consumption, where prompt engineering is utilised as in the present study, the single-agent architecture represents the optimal equilibrium between accuracy and computational cost. This may suggest that priority should be given to prompt engineering when optimising architectures for black-box test generation. However, using multiple agents becomes necessary if the specific few-shot examples or rules used in this study do not fit other contexts of application. Indeed, we found that, when applying only few-shot in place of rule-augmented few-shot prompting, the performance gains introduced by agent collaboration over single-agent are more evident.

VII. Future Work

We acknowledge that the current work presents several limitations that need to be addressed in the future.

Expanding the evaluation to datasets such as *MBPP* [17] will allow a broader assessment of the generalisability of the approach. Similarly, a deeper investigation on how performance varies across task difficulties could complete our assessment.

TABLE VI

A COMPARISON IS PRESENTED BETWEEN THE BASELINE OF THE REFERENCE WORK PROPOSED BY DEO [1] (EMPLOYING GPT-4 TURBO) AND ITS PEAK RESULTS, ALONGSIDE THE IMPROVEMENTS ACHIEVED BY APPLYING OUR PROPOSED PROMPTING STRATEGIES TO THAT FRAMEWORK. THIS IS CONTRASTED WITH THE BASELINE OF OUR STUDY (LEVERAGING NVIDIA/NEMOTRON-3-NANO-30B-A3B) AND THE PEAK PERFORMANCE ATTAINED BY OUR EXPERIMENTAL ARCHITECTURES. ALL THESE RUNS INVOLVED THE FULL HUMANEVAL DATASET. THE RESULTS DEMONSTRATE THAT, DESPITE STARTING FROM A LOWER BASELINE, OUR APPROACH OUTPERFORMS THE ORIGINAL QA AGENT. AGENTCODER [9] IS INCLUDED FOR COMPLETENESS.

	Base model	**Execution Success Rate (%)**	**Coverage** (%)
Single-agent with zero-shot (baseline of the reference work)	GPT-4 Turbo	76.20	79.20
QA Agent	GPT-4 Turbo	88.60	96.60
AgentCoder	GPT-4 Turbo	87.8	87.5
Single-agent with zero-shot (baseline of this study)	Nvidia/Nemotron-3-nano-30b-a3b	62.17	68.23
Single-agent with rule-augmented few-shot	Nvidia/Nemotron-3-nano-30b-a3b	**96.98**	99.30
Multi-agent Merge (Concat) with rule-augmented few-shot	Nvidia/Nemotron-3-nano-30b-a3b	96.41	**99.54**
QA Agent with Standard few-shot	Nvidia/Nemotron-3-nano-30b-a3b	88.65	96.12
QA Agent with rule-augmented few-shot	Nvidia/Nemotron-3-nano-30b-a3b	96.42	98.98

Moreover, the architecture could evolve by introducing additional agents and an iterative feedback loop, where a critic agent generates constructive critiques to refine the tests.

Finally, we seek to identify which specific prompting strategy applied to the *Code Architect* (Section III-D) most frequently yielded the 'winner' code within the *Competitive* strategy (Section III-C1); such analysis will enable further refinement of the prompting methodology applied to the *Code Architect* agent.

REFERENCES

[1] A. Deo, "Qaagent: a multiagent system for unit test generation via natural language pseudocode (student abstract)," in *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'25/IAAI'25/EAAI'25, AAAI Press, 2025.

[2] J. He, C. Treude, and D. Lo, "LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, pp. 124:1–124:30, 2025.

[3] Y. Du, S. Li, A. Torralba, J. B. Tenenbaum, and I. Mordatch, "Improving factuality and reasoning in language models through multiagent debate," in *Proceedings of the 41st International Conference on Machine Learning*, vol. 235 of *ICML'24*, (Vienna, Austria), pp. 11733–11763, JMLR.org, 2024.

[4] K.-T. Tran, D. Dao, M.-D. Nguyen, Q.-V. Pham, B. O'Sullivan, and H. D. Nguyen, "Multi-Agent Collaboration Mechanisms: A Survey of LLMs," *CoRR*, vol. abs/2501.06322, 2025. arXiv: 2501.06322.

[5] T. Liang, Z. He, W. Jiao, X. Wang, Y. Wang, R. Wang, Y. Yang, S. Shi, and Z. Tu, "Encouraging Divergent Thinking in Large Language Models through Multi-Agent Debate," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, (Miami, Florida, USA), pp. 17889–17904, Association for Computational Linguistics, 2024.

[6] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "MetaGPT: Meta programming for a multi-agent collaborative framework," in *The Twelfth International Conference on Learning Representations*, 2024.

[7] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "ChatDev: Communicative Agents for Software Development," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (L.-W. Ku, A. Martins, and V. Srikumar, eds.), (Bangkok, Thailand), pp. 15174–15186, Association for Computational Linguistics, Aug. 2024.

[8] Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen, L. Wang, A. T. Luu, W. Bi, F. Shi, and S. Shi, "Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models," *Computational Linguistics*, vol. 51, pp. 1373–1418, Dec. 2025.

[9] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, "AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation," *CoRR*, vol. abs/2312.13010, 2023. arXiv: 2312.13010.

[10] R. Pan, M. Kim, R. Krishna, R. Pavuluri, and S. Sinha, "ASTER: Natural and Multi-Language Unit Test Generation with LLMs," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, (Ottawa, ON, Canada), pp. 413–424, IEEE, Apr. 2025.

[11] J. Altmayer Pizzorno and E. D. Berger, "Coverup: Effective high coverage test generation for python," *Proceedings of the ACM on Software Engineering*, vol. 2, p. 2897–2919, June 2025.

[12] M. Harman, J. Ritchey, I. Harper, S. Sengupta, K. Mao, A. Gulati, C. Foster, and H. Robert, "Mutation-Guided LLM-based Test Generation at Meta," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pp. 180–191, ACM, June 2025.

[13] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[14] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "REACT: Synergy Reasoning and Acting in Language Models," in *Proceedings of the 11th International Conference on Learning Representations, ICLR 2023*, 2023.

[15] H. Zhang, W. Cheng, Y. Wu, and W. Hu, "A Pair Programming Framework for Code Generation via Multi-Plan Exploration and Feedback-Driven Refinement," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, (New York, NY, USA), pp. 1319–1331, Association for Computing Machinery, 2024.

[16] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 72–82, Association for Computing Machinery, 2014.

[17] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program Synthesis with Large Language Models," *CoRR*, vol. abs/2108.07732, 2021. arXiv: 2108.07732.