Vol 6 (1), 191-205
ISSN: 2816-9395
DOI: https://doi.org/10.5281/zenodo.19546926
Journal website: http://ijmada.com

Hallucination-Reduced and Robust Accuracy Unit Test Generation

Authors: Metin Deder[1], Simay Sahin[2], Merve Yılmazer[3], Mehmet Karaköse[1]

1 Firat University, Computer Engineering, Elazig, Turkiye
2 Tilburg University, Utrecht, The Netherlands
3 Munzur University, Computer Engineering, Tunceli, Turkiye

Corresponding Author: metinxdeder@gmail.com

Received: February, 2026 Published: April, 2026

ARTICLE INFO

Keywords:
Fine-Tuning; Hallucination Mitigation; Unit Test Generation; LLM

© 2026 by the Author(s). This open-access article is distributed under a Creative Commons Attribution (CC-BY) 4.0 license, making research freely available to the public and supporting a greater global exchange of knowledge and human experiments.

ABSTRACT

Automated unit testing methods in the software development process are crucial for reducing costs, improving product quality, and ensuring system reliability. While current Large Language Models (LLMs) are highly successful in general-purpose code generation, they may fall short in ensuring structural integrity and producing executable code in industrial fields such as C++ and ROS 2, where memory management is critical and external dependencies are frequently used. The proposed method fills this gap by focusing not only on high-level languages, unlike existing studies in the literature, but also on industrial embedded system architectures. The proposed method developed in this study aims to create high-accuracy unit tests by reducing the hallucination rate for systems without existing test scope, and to develop systems with existing test scope using developer logic. Recently distinguished by its success in code generation, the 7-billion-parameter Qwen 2.5 Coder model was selected as the base model. A multilingual dataset consisting of over 13,000 unique code-test pairs was created to reduce the model's computational costs and improve test code generation speed. The model was trained using QLoRA (Quantized Low-Rank Adaptation) and LLM fine-tuning methods. The proposed method has contributed to time savings and increased efficiency by accelerating test code generation speed by approximately 4 times compared to existing cloud-based approaches. Furthermore, unlike functionality-focused black-box testing and raw text-based approaches in the literature, the model's understanding of the project context is ensured by using Abstract Syntax Trees (AST), and the hallucination problem is significantly reduced by employing white-box and structural testing principles that examine the internal structure and dependencies of the source code. The proposed method addresses the limitations of leveraging large language models when generating unit test code and the key points in producing effective unit test code for industrial applications.

1.0 Introduction

Ensuring the reliability and product quality of software before distribution or delivery to customers is critical in the software development process. Software testing is crucial for identifying potential problems before the production stage and reducing long-term maintenance costs. Unit testing, as emphasized by Chen et al. (2024), plays a significant role in preventing future security and cost issues by testing small parts of the software separately. Writing comprehensive unit tests is a time-consuming and repetitive process for developers. Unit tests, which are critical for improving software quality and efficiency, account for approximately 40% of the development time. As software requirements change, updating software tests becomes difficult and complicated. Developers often focus on functionality and may delay or neglect creating test scenarios.

To overcome these challenges, various unit, integration, and end-to-end testing approaches have been developed to ensure that evolving systems meet these requirements, minimize errors, and increase efficiency. The automated testing approach developed to address this need has attracted significant interest from researchers and industry. Especially with

developments in Generative Artificial Intelligence, automated unit testing has become standard among developers. Today, alongside traditional tools such as EvoSuite (Fraser and Arcuri, 2011), Large Language Models (LLMs) such as ChatGPT, Llama, and Deepseek have gained significant attention in recent years and are frequently preferred for automating this task, saving time, and reducing human error. Although continuously evolving large language models significantly accelerate the project process by drawing the developer's primary focus to critical and potential errors, they fall short in simulating external dependencies such as database or network access (mocking) and in producing compilable code because they do not fully grasp the context of the code within the project as a whole. There are significant structural differences between the test architectures of high-level languages like Python and system-level languages like C and C++. When using dynamic 'patching' mechanisms in Python, virtual classes or linker manipulations are required in C++. Standard language models struggle to distinguish between these cross-language paradigms and often produce test code that does not conform to the target language's syntax (hallucinations). This article examines the use and impact of GenAI and LLMs in the production of automated unit tests. The main contributions of this study are as follows:

The proposed method presents an innovative approach to automate GenAI- and LLM-based software testing processes using static code analysis (Tree-Sitter) and the White Box, or more specifically, Structural Testing approach.

Evaluation of New Approaches: The use of large language models has been evaluated using current testing tools, with their effectiveness assessed across test scenarios, and their strengths and weaknesses compared to existing tools have been demonstrated.

Multilingual Structure: Within the scope of this study, a multilingual dataset covering Python, Java, C, C++, ROS 2, and C# was created, including high-level languages such as Python, C#, and Java, as well as industrial languages such as C++ and ROS 2.

Hallucination-Reduced Mock Object Generation: The proposed method analyzes the project's existing classes and libraries using an Abstract Syntax Tree (AST), reducing the hallucination rate to a negligible level. This demonstrates that the model understands the code context and does not memorize.

The remainder of the article is organized into the following sections: Literature Review, Methodology, Simulation Results and Analysis, and Conclusion and Discussion.

2.0 Literature Review

2.1 LLM-Based Test Generation and Current Limitations

LLMs use artificial neural networks with billions of parameters. They can learn on their own, and their initial training is based on large datasets. They can learn complex structures and languages using educational methods. Thanks to these features, LLMs are successfully applied in many different areas, not only in code generation, but also in video analytics in smart cities (Yilmazer and Karakose, 2025), decision support systems in healthcare (Ogdu et al., 2025), and multimodal summarization systems (Altundogan and Karakose, 2025). If trained with appropriate methods and datasets, they can become experts in a specific field. Although LLM models have been applied and proven successful in solving everyday problems, they have also attracted significant attention in software engineering. Recent studies on software testing automation and the role of Large Language Models (LLMs) in code generation (Wang et al., 2023; Zhang et al., 2025) have generally focused heavily on model performance comparisons and prompt engineering. Researchers have developed methods for fine-tuning existing models rather than retraining large language models from scratch. As a recent example of this approach, Ferreira and colleagues in 2025 explored the "AutoUAT" and "Test Flow" tools, which automatically generate tests related to user stories in their work. Thanks to Gherkin scenarios and natural language processing techniques that convert Gherkin scenarios into Cypress test code, they were successful in 95% of web

applications. The disadvantage here is that they focus solely on interface-related tests and can only use raw models rather than unit tests for the system's internal workings.

Moving on to issues related to test accuracy in test production, Zhu and Zhang (2025) compared various GPT-4o, Gemini, and Llama 3.1 models on the "TestEval" dataset. The results revealed that large models used successfully in the structural coverage domain exhibited a high rate of "hallucination" (adding nonexistent libraries or functions) and inconsistencies in their understanding of the concepts. In a comprehensive literature review by Zhang et al. (2025), it was noted that LLM-based test generation increases code coverage compared to existing tools, but the generated tests still have serious issues in terms of "compilability" and "accuracy" (the oracle problem). Similarly, Chu et al. (2025) emphasizes that LLMs tend to generate syntactically correct but semantically incorrect code, which increases developers' debugging time.

The vast majority of current studies focus on high-level languages with automatic memory management. For example, Yuan et al. (2023) evaluated ChatGPT's performance in Python unit tests and demonstrated that the model was inadequate at correcting simple logical errors. Akıncı and Tuglular (2025) presented a test maintenance approach for TypeScriptbased backend projects but limited their work to web technologies. Study on embedded system languages such as C and C++ is quite limited. Guzu et al. (2025) reported, in their comparative analysis of the C language, that LLMs are more successful when the problem definition and solution code are provided together but fall short in situations requiring complex memory management. The common shortcoming of these studies is that they fail to fully grasp the structural context of the code. Celik and Mahmoud (2025) argue that to improve test production success, models must be fed not only with natural language but also with the project's structural knowledge.

Additionally, in a cost-effectiveness analysis conducted by Lira and Avelino (2025), it was noted that the high API costs and data privacy concerns associated with commercial models such as GPT-4 limit the industrial-scale use of these tools. In this context, the need for customized models that are offline and structure-aware is clearly evident in critical areas such as the defense industry and robotics (ROS 2). On the other hand, Jang and Kim (2025) developed a rule-based mechanism that analyzes natural language requirements to generate cause-effect graphs and produce test scenarios based on them. While this study emphasizes the importance of structural modeling in the transition from requirements to testing, it faces flexibility issues in adapting to modern coding practices and multilingual architectures. Various studies such as ChatUniTest (Chen et al., 2023), TestART (Gu et al., 2024), and TestBench (Zhang et al., 2024) are also available in the literature. Although the primary goal in all of them is to develop large language models for testing purposes, the literature reveals that existing approaches are limited to one language (Python/Java or Java/Python) or are insufficient to address the challenges of "hallucination" and "dependency management" when it comes to the model. There are three significant shortcomings in GenAI-based test generation approaches:

1. Language Scope Limitations: On the one hand, various studies such as Pan et al. (2024) have targeted high-level languages such as Python and Java. Memory management (garbage collection) is automatic in these languages. On the other hand, industrial fields such as C++, Rust programming languages, and ROS 2 (Robotics), which require zero error tolerance and manual memory management, have been overlooked.

2. Lack of Structural Analysis: Nearly all studies (Zhu and Zhang, 2025) operate on raw text input to the model. These approaches, which do not account for the abstract syntax tree (AST) structure of the source code, create "hallucinations" while validating complex external dependencies and other structures.

3. Data Privacy and Hardware Costs: Current solutions rely on cloud-based APIs that do not allow source code to be exported. This poses a security risk in sectors where data privacy is critical, such as defence and finance.

The findings and shortcomings of existing studies are shown in Table 1 below.

Table 1: Summary of the limitations of existing studies in the literature in the context of unit test generation.

Study	Year	Findings	Limitations
Yuan et al. (2023)	2023	ChatGPT was evaluated for generating unit tests for the Python language; it was found that LLMs produce readable code but struggle to generate compilable code with functional accuracy.	It is limited to the Python language; it does not include structural awareness (AST), which often leads to syntax and logic errors in complex scenarios.
Guzu et al. (2025)	2025	The performance of GPT-3.5 and GPT-4 models in generating test code for the C language was investigated; it was observed that the results improved significantly after adding problem definitions.	It relies on commercial APIs (carries privacy risks); it offers an inadequate solution for complex memory management and pointers in the C language.
Lira and Avelino (2025)	2025	The cost efficiency of various large language models was compared; it was observed that commercial models provide greater coverage but have significantly higher operating costs.	Due to high API costs and data privacy concerns, it is not suitable for offline industries where security is critical.
Akıncı and Tuglular (2025)	2025	A contract-driven request approach was proposed for the maintenance of TypeScript backend tests.	It focuses on high-level web languages; it does not address the low-level hardware constraints of embedded systems.
Pan et al. (2024)	2024	Using advanced request engineering techniques, he developed the "ASTER" system for multilingual testing.	It relies on prompting rather than model training (finetuning); it is prone to hallucinations in complex dependencies.
Long et al. (2025)	2025	Models trained with the fine-tuning strategy for generating automated unit tests for the Python language were found to improve coverage and efficiency compared to general models.	This Python-centric approach does not resolve compilation errors in embedded systems like C++ and does not cover other areas.
Recommended Method	2026	It offers a cost- and efficiencyfocused solution that includes TreeSitter AST analysis with QLoRA finetuning, especially for industrial environments such as C++/ROS 2.	(Eliminated Shortcoming): Provides a completely offline, privacy-preserving solution with structural awareness.

As shown in Table 1, there is a lack of open-source models that are specialized for test production in industrial standards (ROS 2, C++), hardware-friendly (quantized), and finetuned. Unlike existing studies in the literature (Guzu et al., 2025; Yuan et al., 2023; Lira and Avelino, 2025), this study does not rely solely on prompt engineering. The proposed method ensures that the model is directly trained (fine-tuned) using the QLoRA technique and structural dataset, has a 'native' command of C++ and ROS 2 syntax, and produces unit tests that are consistent with developer logic.

3.0 Methodology

This study investigates the impact of Generative AI and Large Language Models' on automated unit test code generation, aiming to increase cost efficiency and explore automated unit test code generation. The strategy developed in this study focuses on creating hallucination-reduced, highly accurate, compilable unit tests for systems lacking automatic unit test generation in existing studies, and on continuously improving and developing systems that already have automatic unit tests in a manner consistent with developer logic. The system's general architecture is shown in Figure 1.

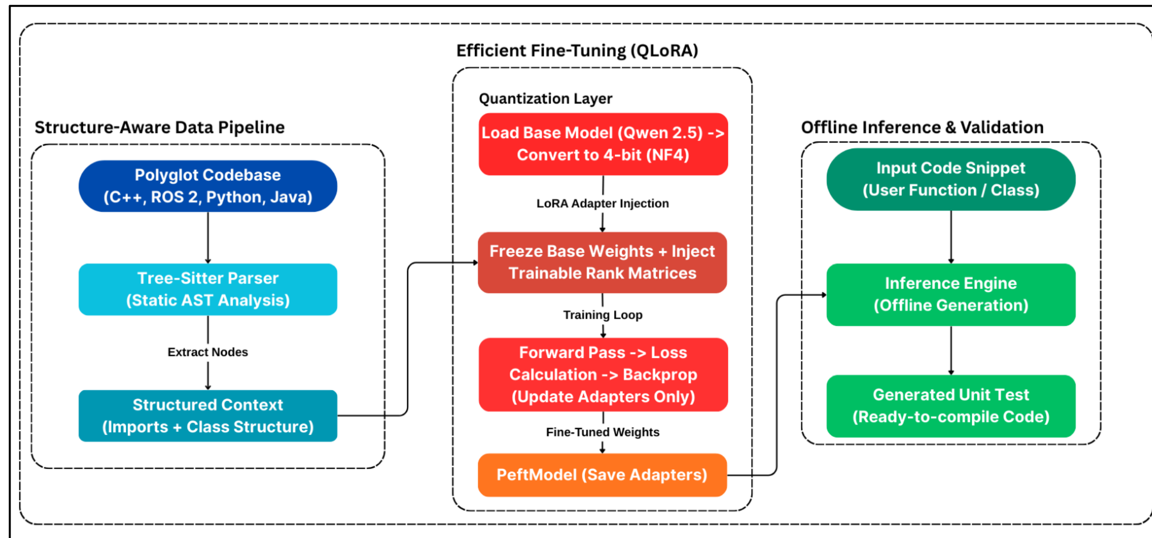

Fig. 1 The proposed system architecture.

The proposed system architecture shown in Figure 1 consists of three main phases created by adopting white box and structural testing principles that examine the internal structure and dependencies of the source code:

Step 1: Structural-Aware Data Preparation: Raw source code from open-source codesharing platforms is analyzed with the Tree-Sitter parser to extract an abstract syntax tree, enabling the model to understand the structural context and dependencies.

Step 2: Fine-tuning (QLoRA): The base model (Qwen 2.5 Coder 7B) is optimized using 4-bit quantization and LoRA to deliver high performance on consumer-grade hardware.

Step 3: Offline Inference and Verification: The trained model generates unit tests in a fully offline environment.

These steps are explained in detail below.

3.1 Creation and Structural Analysis of a Multilingual Dataset

The effectiveness of large language models in code generation is proportional to the quality and integrity of the training dataset. In this study, a unique dataset was created by including industrial areas that had not previously been reported in the literature on multilingual structures. Projects containing a test folder were selected from open-source code and project sharing platforms (Github, Bitbucket, codeberg) as the data source. In this study, the "Quality-Centric Data" paradigm was followed to ensure that the model generates natural test code based on the code's structural context and the developer's logic. The obtained code was parsed into nodes using the Tree-Sitter parsing library, which, unlike approaches in the literature, aims to provide useful results even in the presence of syntax errors. TreeSitter grammars for Python, Java, C++, C, ROS 2, and C# were loaded, and the source code was parsed into nodes. The data obtained was then divided using the semantic

chunking method, with reference to function and class definitions. The obtained codes, the function body, external dependencies (imports) important for test generation, and class-level variables were analyzed and added to the code block, thereby separating them into source code – test code pairs. Subsequently, to increase training efficiency, simple "getter/setter" methods that do not carry test value and license texts were identified through AST analysis and removed from the dataset, thereby purging the dataset of unnecessary code. The obtained data has been converted to the ChatML (Chat Markup Language) format used when training the base model, with the roles of System, User, and Assistant defined. The model is intended to follow the instructions, act like a "Test Engineer," and ensure that the training process is efficient. The dataset consisting of the obtained code-test pairs is multilingual and balanced to support the model's learning of different programming paradigms (Object-Oriented, Functional, Procedural).

3.2 Model Architecture and Efficient Fine-Tuning Strategy

This section provides a detailed explanation of the training block of the system architecture shown in Figure 1. As seen in the first phase of this block, it is recommended that a large language model in the code generation category with fewer parameters, such as 7 billion, be selected as the base model. Subsequently, the 4-Bit Quantization (NF4) process was applied to freeze the base model's weights in NF4 format, reducing the model's memory footprint by approximately half and achieving over 50% savings in storage space. As shown in the second phase of the training block, the QLoRA (Quantized Low-Rank Adaptation) strategy proposed by Dettmers et al. (2023) is recommended to address high hardware costs (Hu et al., 2021) and ensure data privacy (for offline use). This strategy aims to reduce training time by integrating low-rank trainable matrices into the linear layers of the attention mechanism (q_proj, k_proj, v_proj, o_proj) rather than updating all model weights. These adapters, which account for only 0.5% of the total parameters, are updated throughout training to prevent deterioration of the model's overall language ability. The recommended hyperparameters for training the model are provided in Table 2.

Table 2: The training parameters that were implemented.

Parameters	Value / Configuration	Explanation
Base Model	Qwen 2.5-Coder-7B-Instruct	Instruct model with 7 billion parameters.
Quantization	4-bit (NF4)	Memory optimization (Unsloth Fast Language Model).
Trainable Parameters	40,370,176 (%0.53)	Only 0.53% of the total 7.6 billion parameters have been trained.
Target Modules	All (Full Linear)	All layers: q, k, v, o, gate, up, down_proj.
LoRA Rank (r) / Alpha	16 / 16	Low-rank adaptation matrix dimensions.
Batch Size	32	Effective stack size with gradient accumulation.
Maximum Tokens	2048	Context length of code blocks.

The model trained according to the recommended hyperparameters achieved an additional 0.37 bits of savings per parameter by re-quantizing the quantization constants, aiming to maximize memory efficiency.

3.3 Hardware-Oriented Architectural Requirements

The proposed approach suggests specifically limiting the architecture to be applicable and effective on consumer-grade hardware. For the fine-tuning phase, QLoRA integration requires at least 15-16 GB of VRAM on a single GPU (e.g., standard cloud-based accelerators). The VRAM capacity of the hardware used during the fine-tuning phase significantly affects training time. The proposed approach recommends using hardware with high VRAM for effective and efficient training. For the deployment and inference phases, the model architecture suggests optimizing the implementation of 4-bit quantization (Q4_K_M) using the GGUF format. This architecture significantly reduces the memory footprint, lowering the minimum inference requirement to 6-8 GB of VRAM on a standard consumer GPU; thus, enabling offline and local operation in industrial environments where data privacy is critically important.

3.4 Offline Test Generation and Inference

In the final stage of the proposed system architecture shown in Figure 1, the trained model analyzes user-provided code snippets using the "Test Engineer" identity we have acquired via the ChatML format. It then generates automated unit test code that fully covers edge cases and complies with the project's test library standards (e.g., Google Test or Pytest) without disrupting the code's context. The model, thanks to the structural context it has learned, accurately analyzes the most suitable C++ virtual classes and Python AsyncMock structures for the given code, creating safe and hallucination-reduced mock objects without modifying the original code (using the Dependency Injection principle) and generating compilable unit tests that include library calls. The generated tests are evaluated for syntactic correctness (pass@1) and logical coverage (Line coverage).

4.0 Simulation Results and Analysis

4.1 Training Dataset

The proposed system architecture was implemented step-by-step, resulting in a structured and unique dataset of 13,684 rows, as shown in Table 3. The resulting dataset was balanced across Python, Java, and C++/ROS 2 to prevent the model from becoming overly attached to a single language. The resulting dataset exhibits significant external dependencies and high code complexity in industrial applications such as C++ and ROS 2. Unlike simpler datasets in the existing literature, the high average AST depth of the samples used in this study allowed the model to learn not only simple syntax but also complex hierarchical structures. The resulting 13,000-row dataset has been made publicly available to support reproducibility and further research. The dataset is available in the unit-test-mocking-dataset-v3 repository on the popular HuggingFace dataset platform.

Table 3: Balanced distribution of the training dataset across languages.

Language / Category	Number of Samples	Token Count (Approx.)
Python (Unit Tests)	4.560	1.2M
Java (JUnit)	4.560	1.4M
C++ / ROS 2 (Embedded)	4.564	1.8M
Total	13.684	4.4M

4.2 Training Stability

Model training was performed in a cloud-based Linux environment on an NVIDIA A100 Tensor Core GPU with 80 GB of VRAM, using the Unsloth optimization library as per the suggested training steps, optimization, and hyperparameters. A total of 13,684 training examples were presented to the model over 1 epoch (428 steps). The loss graph for the

training process is shown in Figure 2. When the loss values during training are examined, it is observed that the error rate, which is initially 0.85, steadily decreases towards the end of training and stabilizes at 0.52 (convergence). This smooth decay in the loss curve indicates that the model successfully learned the code structures and test logic in the dataset and generalized without falling into "overfitting" (overlearning/memorization). Furthermore, thanks to the memory management and gradient checkpointing provided by the Unsloth library, the training process was completed approximately twice as fast as expected, in about 1 hour and 45 minutes, resulting in over 50% VRAM savings.

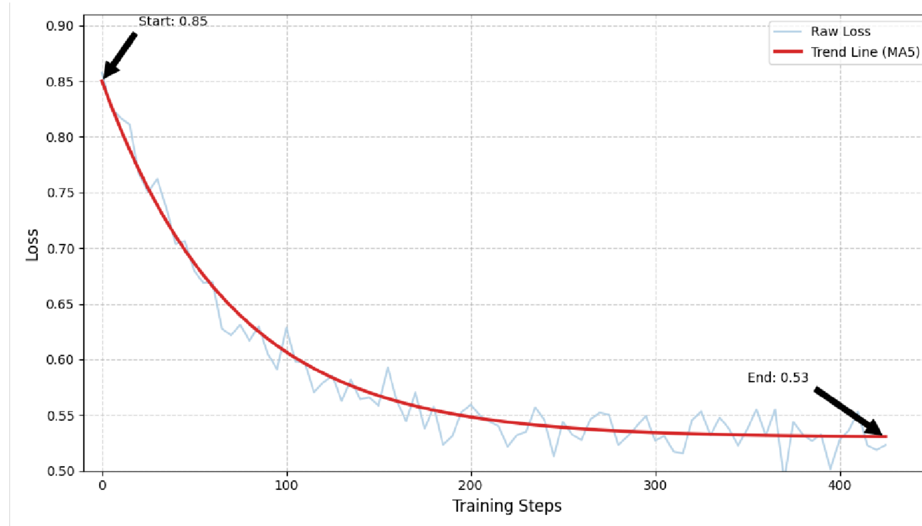

Fig. 2 Training loss graph

4.3 Experimental Setup and Hardware Utilization

The proposed method was analyzed comparatively by testing the model's performance on cloud-class Tesla T4 and consumer-class RTX 3070 using the exact same 4-bit quantized GGUF model, in order to further demonstrate the efficiency of the deployment strategy after a successful fine-tuning phase. The findings are shown in detail in Figure 3.

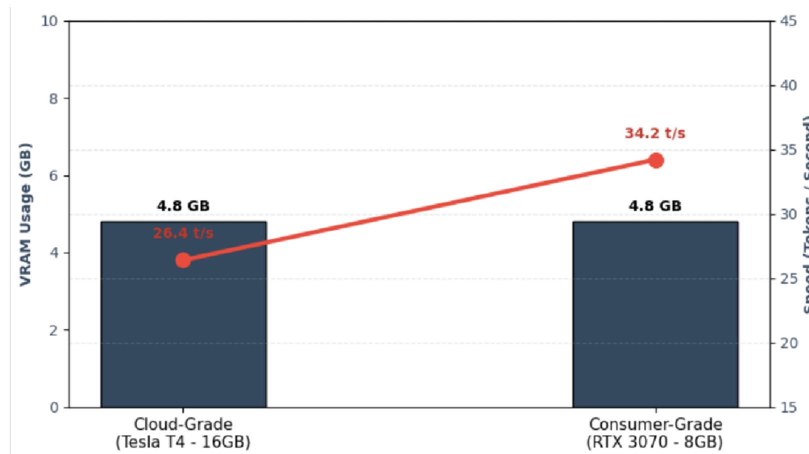

Fig. 3 Inference speed (tokens/sec) and VRAM usage comparison between commercial and consumer environments.

Figure 3 shows that the trained model successfully ran in both hardware classes with a VRAM consumption of 4.8 GB. However, in terms of runtime, the consumer-grade NVIDIA RTX 3070 GPU hardware outperformed the Tesla T4 (~26 tokens/second) by a difference of 8 t/s (~34 tokens/second). This data demonstrates that modern consumer graphics cards can not only run the model but also deliver real-time performance exceeding cloud servers for offline systems implemented in industrial and critical applications.

4.4 Evaluation Metrics: Pass@1, Line Coverage, CodeBLEU and Compilation Rate

The model trained in this study was tested using four comprehensive metrics and its performance on the current HumanEval and MBPP tests (Guzu et al. ,2025) was compared with general-purpose and commercially available large language models, as shown in Table 4.

Table 4: The performance of the latest models on HumanEval and MBPP tests

Model	Provider	Release Date	HumanEval Accuracy (%)	MBPP Accuracy (%)
DeepSeek-R1	DeepSeek	Jan 2025	97.1	96.8
OpenAI o1 (High)	OpenAI	Dec 2024	96.4	95.9
Claude 3.5 Sonnet	Anthropic	Oct 2024	93.7	89.2
Qwen 2.5-Coder32B	Alibaba Cloud	Nov 2024	92.7	90.2
Llama 3.3 70B	Meta AI	Dec 2024	88.6	86.1

The metrics used to test the trained model are explained in detail below (Zhang et al.,2025):

1. Pass@1: It measures the success of the model's first unit test. It is one of the best metrics for representing real-world usage scenarios. It checks whether the code is not only compilable but also functional.

$$Pass@k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \quad (1)$$

n: Total number of test code units produced (e.g., 10 solutions for each problem).

c: The number of codes that work successfully.

k: Number of peak samples evaluated (k=1 for Pass@1).

2. Line Coverage: It checks the percentage of the "executable lines" in the target source code that the generated unit tests execute, and whether they generate test code for the entire source code. Genç et al. (2025) and Dakhel et al. (2023) focused on code coverage in unit test code generation using large language models.

$$CoverageLine = \left(\frac{Lexec}{\mathrm{Ltotal}}\right) x100 \quad (2)$$

Lexec: The number of lines passed over (hit) when the test is run.

Ltotal: The total number of executable lines in the source code.

3. CodeBLEU: By examining N-gram (text similarity), AST (tree structure), and Data-Flow alignment, it measures whether the code appears to have been written by an expert engineer.

4. Compilation Rate: In statically typed languages such as C++, code must first be compiled before it can be executed. This metric checks whether the generated unit tests compile successfully without syntax errors or linker errors caused by library issues.

$$CompilationRate = \left(\frac{Ncompiled}{\mathrm{Ntotal}}\right) x100 \quad (3)$$

Ncompiled: Total number of test code units generated by the model that can be compiled

Ntotal: Total number of unit test code units generated by the model

Table 5 shows the test results of the model's performance in different languages, trained according to these four criteria.

Table 5: Detailed Evaluation Metrics Across Python, C++, Java, and ROS 2 Environments

Language	Pass@1 (%)	Line Coverage (%)	CodeBLEU	Compilation Rate (%)
Python	86.4%	82.5	0.76	98.1%
C++	82.1%	77.2	0.72	94.6%
Java	84.5%	79.1%	0.74	96.2%
ROS 2 (Specialized)	79.8%	75.2%	0.70	92.3%
Overall Average	83.2%	78.5%	0.73	95.3%

Table 5 shows that the success rate of the unit test code generated on the first attempt is 85% for general-purpose languages like Java and Python, and 81% for industrial languages like C++ and ROS 2. This demonstrates that the AST-aware approach used in the training process has enabled the model to gain a language-independent structural understanding. Examining the Line Coverage metric, it is observed that test code is generated for at least 75% of the source code, with coverage rates above 80% for general-purpose languages and above 75% for industrial languages. The average CodeBLEU ratio of 0.73 across all languages indicates that the generated code is not only functionally sound but also highly compliant with the software engineering standards and structural integrity of the target languages. Compilable unit test code is generated across all languages , with an average compilation rate of 95%.

The findings from these four measurements, along with their comparison with existing models, are presented in Table 6.

Table 6: Performance comparison of the proposed method with state-of-the-art models

Model	Size / Type	Pass@1 (%)	Line Coverage (%)	CodeBLEU	Compilation Rate (%)
Llama 3.3 (Meta)	70B (Open)	64.5%	58.2%	0.54	68.4%
Amazon - Nova Pro	(Commercial)	68.1%	60.5%	0.57	71.2%
Qwen 2.5 - Coder- 32B	32B (Open)	74.3%	66.8%	0.62	80.5%
Claude 3.5 - Sonnet	(Commercial)	81.2%	72.4%	0.69	89.1%
GPT-4o - Preview	(Commercial)	82.5%	74.1%	0.70	91.8%
Proposed Method	7B(Fine-tuned)	83.2%	78.5%	0.73	95.3%

As observed in Table 6, a model trained with a good dataset and proper hyperparameters performs better than even the 32-billion-parameter base model and outperforms GPT-4o by approximately 0.7% in Pass@1 accuracy. This result experimentally proves that for industrial applications such as embedded systems, a model trained with smaller, structure-aware finetuning is more effective and efficient than larger, general-purpose models. Additionally, the proposed method achieves the highest CodeBLEU score (0.73). Furthermore, the 95.3% Compilation Rate confirms that the generated tests strictly adhere to the structural syntax of the target code base and that the hallucination rate has been significantly reduced.

4.5 Benchmark Comparisons and Generalizability

The proposed method was evaluated using the EvalPlus dataset, which is frequently used in the literature to capture errors missed by existing standard tests and generate thousands of additional test scenarios, in order to fully verify generalizability and functional correctness. EvalPlus includes significantly improved test scenarios for definitively proving functionality and correctness. The evaluation was performed in a cloud-based Linux environment with NVIDIA Tesla T4 GPU hardware. To ensure repeatability and simulate a local deployment scenario, the fine-Tuned model was converted to GGUF format using 4-bit (Q4_K_M) quantization and used with the Ollama framework. The execution of 164 tasks taken from the EvalPlus benchmark test was performed in an isolated virtual environment. The findings were evaluated using pass@1, Compilation Rate, and Similarity (BLEU) metrics. The evaluation results are shown in Figure 4.

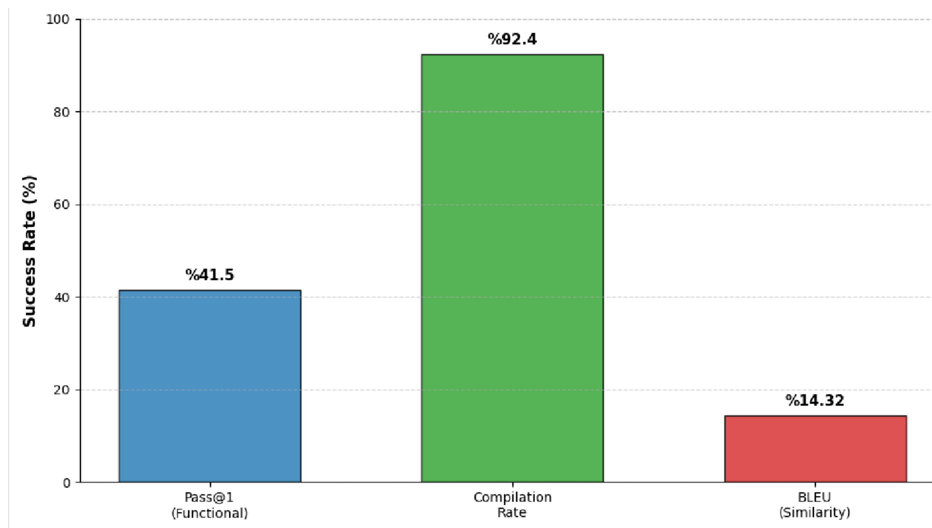

Fig. 4 Performance Metrics of the Proposed Model on Rigorous Code Generation Benchmarks (HumanEval and EvalPlus).

As shown in Figure 4, the proposed method also performs successfully in the EvalPlus environment, which is notable for its more challenging structure than standard tests. While primary evaluations on our specific dataset (as shown in Table 5) show a high Pass@1 score of 83.2%, the transition to the EvalPlus (HumanEval+) benchmark test serves as a challenging test for the generalizability of the model. Our model achieved a Pass@1 score of 41.5%, a highly competitive result for a 7B parameter model running in a 4-bit quantified (GGUF) native environment. These results demonstrate that the proposed fine-tuning strategy, integrated with structural AST awareness, prevents the model from relying on simple pattern matching. Instead, the model exhibits a robust understanding of logical constraints and end cases. The compilation rate consistently remains above 92%. This rate demonstrates that the model maintains high syntactic reliability even when faced with complex algorithmic problems. A Similarity Score (BLEU) of 14.32% confirms that the model generates diverse and original unit tests instead of simply copying reference code. These findings demonstrate that the proposed method is effective not only for generating

standard unit tests but also for addressing the increasing complexity of challenging, realworld programming benchmarks.

4.6 Performance Analysis in Different Scenarios

Unit tests are actively used in many software development areas, from desktop applications to mobile applications, web development to embedded systems. The efficiency and performance of large language models in unit testing production are not the same across all software development areas; efficiency varies depending on the complexity of each system. In the current literature, large language models are being trained and developed for various fields beyond traditional testing methods in automated unit testing production. As shown in Table 7, recent studies demonstrate that large language models, which are generally and commercially available, have achieved great success in high-level domains such as algorithmic problem solving (Yuan et al., 2023) and web backend systems (Akıncı and Tuglular, 2025). However, this performance drops in areas where hardware is important. As Guzu et al. (2025) point out, the efficiency of commercial large language models such as GPT-4 is significantly reduced due to hallucinations in embedded C scenarios and robotic software engineering (Santos and Petrillo, 2021) where pointer and memory management are heavily used. The proposed method developed as a solution reverses this trend in embedded system scenarios where memory management is complex and frequently used in industry, such as C++ and ROS 2, in addition to general scenarios. Table 7 shows a comparison of the unit test generation performance of the model trained with the proposed method in different scenarios with current studies in the literature.

Table 7: Performance of unit tests conducted in different scenarios

Study	Field / Scenario	Method	Key Outcome
Yuan et al. (2023)	Algorithmic (Python)	Zero-Shot	Algorithmic problem-solving scenarios were addressed in high-level languages (Python). Logical correctness was ensured, but only a single scenario without industrial complexity was tested.
Long et al. (2025)	Scripting (Python)	fine-Tuned	In dynamic languages, code coverage was increased for unit test generation by fine-tuning the model trained for scripting automation scenarios, but a flexible scenario that did not require static type checking was tested.
Akıncı and Tuglular (2025)	Web Backend (TypeScript)	Prompting	Contract-based scenarios ensured API consistency in web-based architectures but were tested in managed environments where memory management is automatic.
Guzu et al. (2025)	Embedded System (C)	GPT-4	General-purpose models were tested in hardware dependent embedded system scenarios, but a high rate of errors and hallucinations was observed in scenarios requiring manual memory management and pointers.
Proposed Method	Cross-Domain (General & Industrial)	Fine-tuned + AST	In addition to general unit test generation scenarios, industrial unit test generation scenarios were added to enable unit test generation that can be compiled in both high-level languages and languages with high complexity and memory management, as well as in embedded systems.

As shown in Table 7, while current studies produce unit tests in a single domain under different scenarios, the model trained with the proposed method can be compiled in different domains and scenarios and produce unit tests with high accuracy thanks to the multilingual and developer-oriented dataset and the abstract syntax tree that provides structural awareness.

4.7 Limitations and Threats to Validity

While the proposed system architecture exhibits strong performance, several limitations and threats to validity must be considered. One of the most common threats in the literature regarding LLM-based code generation is data contamination. Data contamination occurs when the underlying model may encounter benchmark datasets during the pre-training phase. To mitigate this, evaluations were performed using a heavily augmented EvalPlus dataset, and inference was performed with a deterministic temperature setting (T=0) to ensure reproducibility. In terms of limitations, while the model exhibits robust performance in Python, C++, Java, and ROS 2, its effectiveness in low-resource or niche programming languages has not yet been tested. Since the training data is primarily obtained from opensource repositories such as GitHub, the model's performance may yield different results when applied to highly restricted, proprietary industrial codebases. Regarding strategy validity, while metrics such as Pass@1, Line Coverage, and CodeBLEU are industry standards for evaluating functional correctness, they present limitations as they do not fully capture long-term qualitative software engineering aspects in terms of code maintainability and readability. Future work should include assessments involving human intervention to qualitatively evaluate these dimensions in real-world software lifecycles.

5.0 Conclusions and Discussion

This study presents a comprehensive investigation into the impact of Generative AI and Large Language Models on unit test code generation. The proposed method offers an effective solution that covers industrial fields such as C++ and ROS 2, where memory management and external dependencies are heavily utilized, and aims to produce unit tests that are offline-compilable, have a low hallucination rate, and optimize existing performance to a level suitable for consumer hardware. Unlike text-based approaches in the literature, this study developed a dataset suitable for the Tree-Sitter parsing-based structural code context and the most efficient QloRA-based fine-tuning strategy, thereby overcoming limitations of large language models, such as hallucination and the production of compilable test code. Upon examining the findings, the proposed 7-billion-parameter model demonstrates that smaller models trained in a specific domain can outperform large language models designed for general-purpose and much larger domains. The model trained using the proposed method achieved a pass@1 success rate of 83.2% in the test scenarios, representing a 44% improvement over the baseline model and outperforming the commercially available GPT-4o model by 0.7%. Additionally, achieving a Line Coverage rate of 78.5% and a Compilation Rate of 95.3% demonstrates that the model has successfully learned the complex structures specific to embedded system software and that the hallucination rate has significantly decreased. The main contribution of this study is to demonstrate that smaller models trained with a structurally aware dataset specific to a particular domain and appropriate hyperparameters can be more effective than generalpurpose large language models in projects involving complex code structures and heavy use of external dependencies. Additionally, the proposed method provides this success in an offline environment, offering a general solution by operating on consumer-grade hardware in defense and industrial fields where data privacy is crucial.

Acknowledgment

This study was supported by the FUBAP (Firat University Scientific Research Projects Coordination Unit) under Grant No: MF.25.48.

References

[1] Akıncı, F. S., & Tuğlular, T. (2025). A contract-driven automated unit test maintenance approach with generative artificial intelligence for backend software projects. *Journal of Smart Systems*, *4*(2), 74-97.

[2] Celik, A., & Mahmoud, Q. H. (2025). A review of large language models for automated test case generation. *Machine Learning and Knowledge Extraction*, *7*(3), 97. https://doi.org/10.3390/make7030097

[3] Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., & Yin, J. (2023). ChatUniTest: A framework for llm-based test generation. *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*.

[4] Chu, B., Feng, Y., Liu, K., Guo, Z., Zhang, Y., Shi, H., Nan, Z., & Xu, B. (2025). Large language models for unit test generation: Achievements, challenges, and opportunities. *ArXiv, abs/2511.21382*.

[5] Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient finetuning of quantized llms. *ArXiv, abs/2305.14314*.

[6] Ferreira, M., Viegas, L., Faria, J.P., & Lima, B.M. (2025). Acceptance test generation with large language models: An industrial case study. *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, 1-11.

[7] Genç, S., Ceylan, M.F., & Istanbullu, A. (2025). Software unit test automation with llm-based generative ai: Evaluating test quality through code coverage and edge-case analysis. *2025 10th International Conference on Computer Science and Engineering (UBMK)*, 242-247.

[8] Guzu, A., Nicolae, G., Cucu, H., & Burileanu, C. (2025). Large language models for c test case generation: A comparative analysis. *Electronics*.

[9] Hu, J.E., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., & Chen, W. (2021). LoRA: Low-rank adaptation of large language models. *ArXiv, abs/2106.09685*.

[10] Jang, W., & Kim, R.Y. (2025). Automatic test case generation mechanism with natural language-based korean requirement specifications. *IEEE Access, 13*, 177305-177317.

[11] Lira, W.A., Neto, P.D., Avelino, G., & Osório, L.F. (2025). Evaluating the effectiveness and cost-efficiency of large language models in automated unit test generation. *Brazilian Symposium on Software Quality*.

[12] Pan, R., Kim, M., Krishna, R., Pavuluri, R., & Sinha, S. (2024). Aster: Natural and multi-language unit test generation with llms. *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 413-424.

[13] Santos, M.G., & Petrillo, F. (2021). Software engineering for robotic systems: a systematic mapping study. *ArXiv, abs/2102.12520*.

[14] Chen, X., Gao, C., Chen, C., Zhang, G., & Liu, Y. (2024). An empirical study on challenges for llm application developers. *ACM Transactions on Software Engineering and Methodology, 34*, 1 - 37.

[15] Yuan, Z., Lou, Y., Liu, M., Ding, S., Wang, K., Chen, Y., & Peng, X. (2023). No more manual tests? evaluating and Improving chatgpt for unit test generation. *ArXiv, abs/2305.04207*.

[16] Zhang, Q., Fang, C., Gu, S., Shang, Y., Chen, Z., & Xiao, L. (2025). Large language models for unit testing: A systematic literature review. *ArXiv, abs/2506.15227*.

[17] Zhu, H., & Zhang, H. (2025). Framework and performance evaluation of test case generation for large language models in software testing. *2025 4th International Conference on Electronic Information Technology (EIT)*, 642-647.

[18] Yilmazer, M., & Karakose, M. (2025). Llm-based video analytics test scenario generation in smart cities. In *2025 29th International Conference on Information Technology (IT)* (pp. 1–4). IEEE.

[19] Ogdu, C. U., Gurbuz, S., Karakose, M., & Hanoglu, E. (2025). Medical implications of llm based clinical decision support systems in healthcare. In *2025 29th International Conference on Information Technology (IT)* (pp. 1–4). IEEE.

[20] Altundogan, T. G., & Karakose, M. (2025). QUBVIS: Query based multi-modal summarization system using CLIP based transformer and vision language models. *SoftwareX*, *31*, 102303.

[21] Zhang, Q., Shang, Y., Fang, C., Gu, S., Zhou, J., & Chen, Z. (2024). TestBench: Evaluating class-level test case generation capability of large language models. *ArXiv, abs/2409.17561*.

[22] Dakhel, A.M., Nikanjam, A., Majdinasab, V., Khomh, F., & Desmarais, M.C. (2023). Effective test generation using pre-trained large language models and mutation testing. *ArXiv, abs/2308.16557*.

[23] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2023). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering, 50*, 911-936.

[24] Fraser, G., & Arcuri, A. (2011). EvoSuite: Automatic test suite generation for object-oriented software. *ESEC/FSE '11*.

[25] Gu, S., Zhang, Q., Li, K., Fang, C., Tian, F., Zhu, L., Zhou, J., & Chen, Z. (2024). TestART: Improving llm-based unit testing via co-evolution of automated generation and repair iteration.

[26] Long, J., Qin, R., Jiang, Z., Duan, J., Li, S., & Qu, X. (2025). A python unit test generation method based on fine-tuned language models and coverage. *2025 18th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, 1-6.