



Article

Integrating Large Language Models into Automated Software Testing

Yanet Sáez Iznaga ^{1,*}, Luís Rato ^{2,*}, Pedro Salgueiro ² and Javier Lamar León ^{2,†}¹ Dectech, Rua Circular Norte do Parque Industrial e Tecnológico de Évora, Lote 2, 7005-841 Evora, Portugal² VISTA Lab, ALGORITMI Research Center/LASI, University of Évora, 7000-671 Evora, Portugal; pds@uevora.pt (P.S.); jlamarleon@uevora.pt (J.L.L.)

* Correspondence: yanet.saez@gmail.com (Y.S.I.); lmr@uevora.pt (L.R.)

† These authors contributed equally to this work.

Abstract

This work investigates the use of LLMs to enhance automation in software testing, with a particular focus on generating high-quality, context-aware test scripts from natural language descriptions, while addressing both text-to-code and text+code-to-code generation tasks. The Codestral Mamba model was fine-tuned by proposing a way to integrate LoRA matrices into its architecture, enabling efficient domain-specific adaptation and positioning Mamba as a viable alternative to Transformer-based models. The model was trained and evaluated on two benchmark datasets: CONCODE/CodeXGLUE and the proprietary TestCase2Code dataset. Through structured prompt engineering, the system was optimized to generate syntactically valid and semantically meaningful code for test cases. Experimental results demonstrate that the proposed methodology successfully enables the automatic generation of code-based test cases using large language models. In addition, this work reports secondary benefits, including improvements in test coverage, automation efficiency, and defect detection when compared to traditional manual approaches. The integration of LLMs into the software testing pipeline also showed potential for reducing time and cost while enhancing developer productivity and software quality. The findings suggest that LLM-driven approaches can be effectively aligned with continuous integration and deployment workflows. This work contributes to the growing body of research on AI-assisted software engineering and offers practical insights into the capabilities and limitations of current LLM technologies for testing automation.



Academic Editor: Paolo Bellavista

Received: 13 September 2025

Revised: 10 October 2025

Accepted: 15 October 2025

Published: 18 October 2025

Citation: Iznaga, Y.S.; Rato, L.; Salgueiro, P.; León, J.L. Integrating Large Language Models into Automated Software Testing. *Future Internet* **2025**, *17*, 476. <https://doi.org/10.3390/fi17100476>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: automated software testing; large language models; test case generation; low-rank adaptation codestral mamba model

1. Introduction

Ensuring the quality and reliability of software systems is a foundational concern in modern software engineering. As applications become more complex and integrated into critical sectors, rigorous testing is essential to validate correctness and performance prior to deployment. Traditionally, testing processes have relied heavily on manual test case generation and execution—a practice that, while flexible, is time consuming, error-prone, and difficult to scale in fast-paced development environments [1,2].

To address these limitations, automated testing has emerged as a standard practice, offering efficiency, repeatability, and broader test coverage. A wide range of automation frameworks now supports various testing needs, from low-level unit testing to high-level

system and interface validations [3]. However, these tools often require programming expertise and ongoing maintenance to adapt to evolving software features, limiting their accessibility and long-term scalability [1].

Recent advancements in artificial intelligence (AI), particularly in machine learning (ML) and natural language processing (NLP), present promising alternatives. Large language models (LLMs) such as GPT-4 [4], BERT [5], and T5 [6] demonstrate strong capabilities in understanding context and generating human-like language, which can be leveraged to automate test script creation [7,8]. These models enable users to generate test scripts from natural language inputs, reducing the need for programming knowledge and expediting the testing process [9,10]. Recent studies also highlight the growing role of LLM-based agents and Parameter-Efficient Fine-Tuning (PEFT) methods in improving adaptability and scalability in software engineering contexts [11,12].

LLM-based approaches also offer adaptability across different systems, platforms, and programming environments [13]. Their ability to understand and generalize software behavior makes them particularly useful for dynamic applications and cross-platform test case migration [14]. This evolution in testing automation indicates a move from rigid, code-centric tools toward more intuitive, context-aware systems that align with modern development methodologies while remaining potentially complementary to traditional approaches.

Despite these advancements, a significant gap persists in the automation of functional test case generation, particularly in creating test cases that capture high-level user flows and complex system behaviors [15–19]. Addressing this challenge requires models that can reason across structured inputs, user requirements, and multi-layered software interfaces. While state-space models (SSMs), such as Mamba, offer promising capabilities in handling long sequences with high throughput and global context [20,21], general-purpose LLMs often fall short in completeness, correctness, and adaptability when applied to functional testing in dynamic, real-world environments.

1.1. Key Contributions

This work introduces a novel approach to automate software test case generation by leveraging the Codestral Mamba 7B model, which was fine-tuned with Low-Rank Adaptation (LoRA). Unlike traditional Transformer-based models, which rely on self-attention mechanisms and often face challenges in scalability, computational efficiency, and long-sequence modeling, our method exploits the unique advantages of SSMs. Specifically, Mamba's linear-time inference and global context modeling capabilities enable more efficient and scalable test case generation particularly for complex, real-world software systems.

There are five primary contributions of this paper:

- **Novel Integration of LoRA with a State-Space Model:** We propose and evaluate a method for embedding LoRA matrices into the Codestral Mamba 7B architecture, enabling efficient, domain-specific adaptation. This approach contrasts with existing Transformer-based fine-tuning techniques, which often require full-model updates or lack the linear scalability of SSMs. By focusing on LoRA, we achieve PEFT while preserving Mamba's inherent advantages in handling long sequences and maintaining computational efficiency.
- **Dual-Dataset Evaluation with Emphasis on Real-World Applicability:** Our method is evaluated on both the CodeXGLUE/CONCODE benchmark and a proprietary Test-Case2Code dataset, which comprises real-world manual-to-Pytest conversions. This dual evaluation demonstrates superior performance in syntactic accuracy, semantic alignment, and test coverage compared to baseline Transformer models. Notably, our approach achieves these improvements without the computational overhead typically associated with fine-tuning large Transformer architectures.

- **Prototype Development (Codestral Mamba_QA):** We present a functional chatbot prototype that integrates the fine-tuned Mamba model into a user-friendly interface, showcasing its potential for seamless integration into real-world software testing pipelines. This prototype highlights the practical applicability of our method, offering a scalable and efficient alternative to existing tools like GitHub Copilot or GPT-4-based solutions.
- **Computational Efficiency and Scalability:** The LoRA-based fine-tuning approach significantly reduces the number of trainable parameters, enabling rapid adaptation (e.g., 200 epochs in just 20 min for the proprietary TestCase2Code dataset). This efficiency is particularly advantageous for industrial applications, where quick iteration and low resource consumption are critical. In contrast, Transformer-based models often require extensive computational resources for fine-tuning, limiting their scalability in resource-constrained environments.
- **Compatibility with CI/CD Workflows:** The results demonstrate that LoRA-enhanced state-space models can be effectively integrated into Continuous Integration and Continuous Delivery (CI/CD) workflows. This alignment presents a strong alternative to Transformer-based methods, which often face limitations when dealing with the dynamic and rapidly evolving nature of modern software pipelines. By enhancing both efficiency and adaptability, the proposed approach promotes more sustainable and scalable test automation practices.

1.2. Why This Matters

While Transformer-based models (e.g., CodeT5 [22], GitHub Copilot [23], GPT-4 [4]) have dominated the field of automated test generation, their reliance on self-attention mechanisms introduces computational bottlenecks, particularly for long-sequence tasks and large-scale deployments. In contrast, state-space models like Mamba offer linear-time inference and global context modeling, making them inherently more scalable and efficient for real-world applications. By combining Mamba with LoRA, we not only retain these advantages but also introduce a PEFT strategy that outperforms traditional approaches in both performance and resource utilization.

This work thus bridges a critical gap in the literature, demonstrating that state-space models when fine-tuned with LoRA can surpass Transformer-based methods in automated test generation, particularly for complex, dynamic software environments.

1.3. Paper Organization

The remainder of this paper is structured as follows:

- **Section 2 (Related Work):** reviews the state-of-the-art in automated software testing and the application of LLMs for test case generation.
- **Section 3 (Materials and Methods):** details the model architecture, datasets, and prompt engineering strategy.
- **Section 4 (Experiments and Results):** presents the experimental setup, evaluation metrics, and analysis of results.
- **Section 5 (Discussion):** interprets the findings of the present work and critically examines its limitations.
- **Section 6 (Conclusions):** concludes the study and outlines directions for future research.
- **Appendices A–C:** Appendix A provides two non-sensitive dataset samples illustrating data structure and content to ensure transparency and reproducibility. Appendix B presents a representative prompt–response pair from the proprietary TestCase2Code dataset used for fine-tuning the Codestral Mamba 7B model, clarifying the prompt

engineering process. Appendix C includes representative prompt–response examples of model-generated Pytest code (Tables A1–A6), demonstrating the model’s behavior under varying configurations and its ability to produce precise, reusable, and semantically coherent test cases.

2. Related Work

2.1. Automated Software Testing: From Traditional Frameworks to AI-Assisted Approaches

The evolution of automated software testing has been closely tied to the adoption of agile and DevOps practices, where rapid iteration cycles demand efficient and reliable validation mechanisms. Traditional frameworks such as Selenium and JUnit [24] have laid the groundwork for test automation, yet the need for extensive scripting and domain expertise often constrains their effectiveness. While these tools excel in structured environments, they struggle to scale in dynamic codebases where requirements evolve continuously [25]. The Testing Pyramid [26] advocates for a balanced distribution of unit, integration, and end-to-end tests, but manual implementation remains a bottleneck, particularly for complex system-level validations.

2.2. Transformer-Based LLMs in Test Automation: Strengths and Limitations

The advent of Transformer-based LLMs such as GPT-4 [4], CodeBERT [27], and CodeT5 [22] has revolutionized AI-assisted software development by enabling natural language-to-code generation. These models have demonstrated proficiency in tasks ranging from code synthesis [28] to vulnerability detection [29] with tools like GitHub Copilot [23] and UniXcoder [30] further refining the alignment between human intent and executable outputs. However, their application to functional test automation, where test cases must capture high-level user flows and cross-component interactions, reveals critical limitations:

- **Computational Inefficiency:** Transformer architectures rely on self-attention mechanisms with quadratic complexity ($O(n^2)$), which becomes prohibitive for long-sequence tasks such as end-to-end test suites [16].
- **Context Window Constraints:** Limited token windows (e.g., 8K for GPT-4) restrict the models’ ability to process extensive test scenarios or maintain global context across interconnected software modules [14].
- **Unit-Level Focus:** Most prior work [31–33] has concentrated on unit-level testing, leaving a gap in generating test cases for system-level behaviors [15,17].
- **Fine-Tuning Overhead:** Adapting large Transformer models (e.g., CodeT5’s 220M parameters) to domain-specific tasks requires full fine-tuning, which is resource-intensive and impractical for many industrial settings [22].

2.3. State-Space Models: A Paradigm Shift for Long-Sequence Tasks

To address these challenges, SSMs such as Mamba [21] have emerged as a compelling alternative. Unlike Transformers, SSMs employ linear-time inference and global context modeling, making them inherently more scalable for tasks requiring long-range dependencies. Key advantages include the following:

- **Linear Scalability:** Selective SSM blocks process sequences in $O(n)$ time, enabling the efficient handling of large test suites without the computational bottlenecks of self-attention [20].
- **Extended Context Retention:** SSMs maintain state across entire sequences, improving coherence in generated test cases that span multiple software layers [18].

Despite these strengths, SSMs remain underexplored in test automation with existing research primarily focusing on general code synthesis rather than functional validation [19].

2.4. PEFT: Bridging the Gap for Functional Testing

Although SSMs offer architectural advantages, their adaptation to specialized domains, such as functional test generation, requires fine-tuning strategies that balance performance and computational cost. LoRA [34] addresses this need by injecting trainable low-rank matrices into pretrained models, reducing the number of updated parameters by orders of magnitude. Recent studies on PEFT techniques further highlight their value in optimizing large models for domain-specific applications without prohibitive resource costs [12,35,36]. Prior applications of LoRA have focused on Transformer-based models [34], but its integration with SSMs remains largely unexplored. This gap presents an opportunity to combine the scalability of SSMs with the efficiency of PEFT, which is a synergy that our work exploits to advance automated test generation.

2.5. Positioning Our Work in the Landscape

This study distinguishes itself from existing research in the following aspects:

- **First Integration of LoRA with SSMs for Testing:** Unlike prior work that applies LoRA to Transformers (e.g., CodeT5), we leverage it to fine-tune Codestral Mamba 7B, demonstrating that SSMs can surpass Transformer-based models in both performance and resource efficiency for functional test automation.
- **Focus on Real-World Functional Testing:** While datasets like CodeXGLUE/CON-CODE [37] evaluate general code synthesis, our proprietary TestCase2Code dataset targets end-to-end test scripts, providing a more realistic benchmark for industrial applications.
- **Prototype-Driven Validation:** We develop Codestral Mamba_QA, a chatbot prototype that showcases the practical deployment of LoRA-fine-tuned SSMs in CI/CD pipelines, a capability lacking in Transformer-based tools like Copilot, which rely on external APIs and lack project-specific adaptability.

2.6. Summary of Research Gaps

Table 1 summarizes key limitations in prior work and how our approach addresses them.

Table 1. Research gaps in automated test generation and our contributions.

Approach	Limitations	Our Solution
Transformer LLMs (e.g., GPT-4, CodeT5)	High quadratic cost, short context, limited to unit tests.	Linear-time SSM (Mamba) with LoRA for scalable functional test generation.
Conventional SSMs (e.g., Mamba)	No domain tuning; weak real-world validation.	LoRA fine-tuning with proprietary TestCase2Code dataset.
LoRA on Transformers	Inherits Transformer limits (e.g., context length).	LoRA on SSMs enabling efficient global context handling.
Public Datasets	Unit-level only; lack functional/system tests.	TestCase2Code: manual-to-Pytest conversions for system-level validation.

3. Materials and Methods

Software testing ensures software systems conform to specified requirements and perform reliably [38,39]. Test constructs such as individual tests, test cases, and test suites support structured validation across different testing levels and techniques, including

black-box, white-box, and gray-box, and they are often augmented by design strategies like boundary value analysis and fuzzing [40–43].

While manual and rule-based approaches have historically dominated test generation, large language models (LLMs) now enable an automated, context-aware generation of test code. Their ability to understand both natural language and source code facilitates the scalable, dynamic creation of test suites aligned with functional and exploratory testing goals.

3.1. LLM Architecture Grounded in State-Space Models

This work employs the Codestral Mamba 7B model, which is a state-space model (SSM) optimized for efficient long-range sequence modeling [20,21]. By integrating selective SSM blocks with multi-layer perceptrons, Mamba enables linear-time inference. This architecture builds on the findings presented in [44]. Its implementation is publicly available at <https://github.com/state-spaces/mamba> (accessed on 9 October 2025), promoting reproducibility and encouraging further exploration. The model's extended context capacity supports the effective retrieval and utilization of information from large inputs such as codebases and documentation, thereby enhancing performance in tasks like debugging and test generation (Mistral AI, Codestral Mamba, available at: <https://mistral.ai/en/news/codestral-mamba>, accessed on 9 October 2025. NVIDIA Developer Blog, Revolutionizing Code Completion with Codestral Mamba: The Next-Gen Coding LLM, available at: <https://developer.nvidia.com/blog/revolutionizing-code-completion-with-codestral-mamba-the-next-gen-coding-llm/>, accessed on 9 October 2025).

3.2. Integrating LoRA Fine-Tuning into the Mamba-2 Architecture

To specialize the Mamba-2 architecture for functional test generation, we employ LoRA [34], which is a PEFT technique that significantly reduces the number of trainable parameters while maintaining the model's general-purpose capabilities. Rather than updating the entire set of pretrained weights, LoRA injects trainable low-rank matrices into selected projection layers, allowing for efficient task-specific adaptation.

In the context of Mamba-2, LoRA is seamlessly integrated into the **input projection matrices** ($W^{(xzBC\Delta)}$) and the **output projection matrix** ($W^{(o)}$) (see Figure 1), which are key components in the Selective State-Space Model (SSSM) blocks. The original projection weights can be expressed as a combination of the pretrained weights and a low-rank update. This decomposition is formalized in Equation (1):

$$W = W_{\text{pretrained}} + \Delta W, \quad \Delta W = BA \quad (1)$$

where $A \in \mathbb{R}^{r \times d}$ and $B \in \mathbb{R}^{L \times r}$, and r is the rank of the low-rank decomposition with $r \ll \min(d, (h \cdot d_{\text{head}}))$. This decomposition introduces only a small number of trainable parameters compared to the Mamba model's total parameter count, resulting in a lightweight adaptation mechanism that preserves its computational efficiency. The modified input projection, referred to as the LoRA-adjusted input transformation, is formally defined in Equation (2):

$$x_p = W_{\text{pretrained}}u + BAu \quad (2)$$

An analogous formulation is applied to the output projection, following the same decomposition principle. This structured approach enables the targeted adaptation of the model to the specific domain of software testing while preserving the integrity of the original pretrained weights and avoiding the computational cost of full-model fine-tuning.

By leveraging LoRA within Mamba-2's architecture, we achieve an efficient and modular specialization pipeline, where general-purpose language capabilities are retained, and only task-relevant components are fine-tuned. This is especially advantageous in the

domain of automated test generation, where scalability, low resource consumption, and contextual understanding are critical. As shown in Figure 1, this architecture supports fine-grained control over adaptation, enabling precise alignment between software artifacts (e.g., function definitions, comments) and the resulting test case generation.

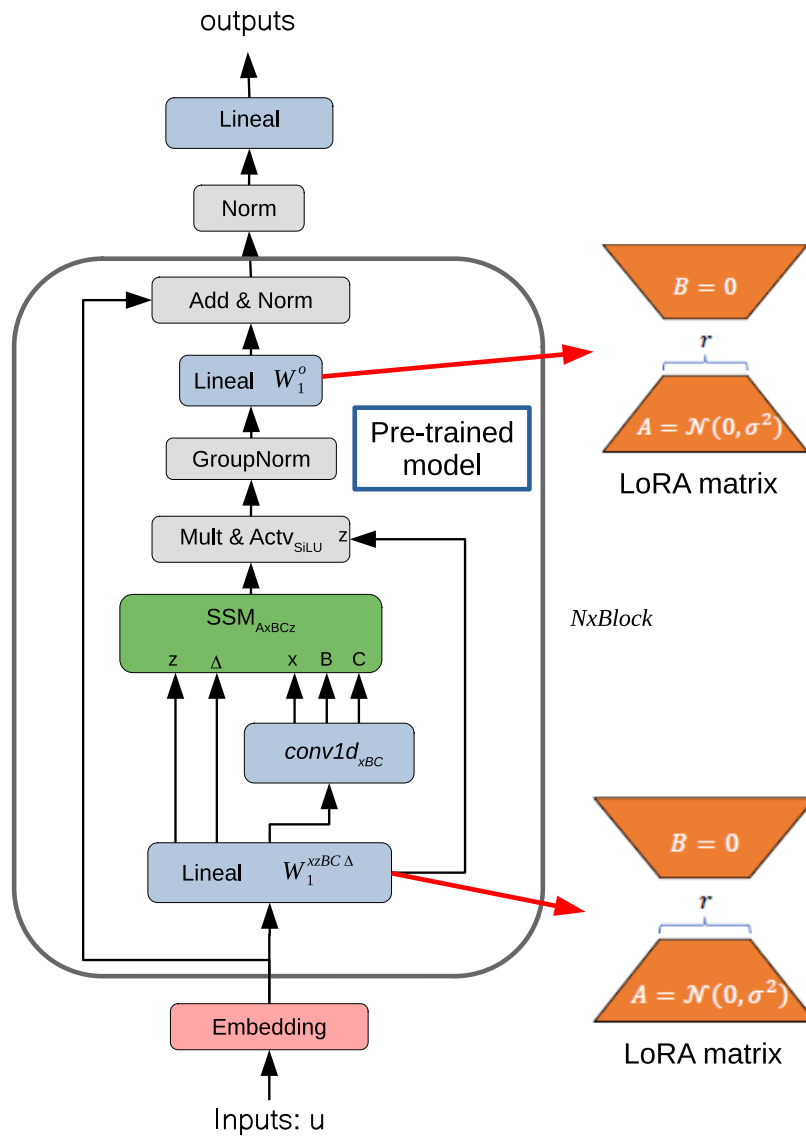


Figure 1. Integration of LoRA into the Mamba-2 SSM architecture. Low-rank updates are applied to the input and output projection matrices to enable efficient fine-tuning. The red arrows indicate the layers where the LoRA matrices are inserted, corresponding to the input and output linear projections. The black arrows represent the data flow between sequential components within each SSM block.

In summary, the integration of LoRA into Mamba-2 establishes a flexible and domain-adaptive framework for automated test case generation. This combination brings together the long-context modeling and scalability of SSMs with the efficiency and modularity of low-rank fine-tuning, enabling the generation of high-quality, context-aware functional tests directly from source code.

3.3. Datasets

This work employs two different datasets to support model training and evaluation: The CONCODE dataset [45] from the CodeXGLUE benchmark [37] and the proprietary

TestCase2Code dataset. CodeXGLUE/CONCODE is part of the CodeXGLUE benchmark suite, which is a comprehensive platform for evaluating models on diverse code intelligence tasks. For our experiments, we use the text-to-code generation task, specifically the CONCODE dataset (<https://github.com/microsoft/CodeXGLUE/tree/main/>, accessed on 9 October 2025), which consists of 100K training samples, 2K development samples, and 2K testing samples. Each example includes a natural language description, contextual class information, and the corresponding Java code snippet. The structured format, with fields like `n1` and `code`, facilitates context-aware code generation, making it well suited for benchmarking large language models in source code synthesis from textual prompts.

The proprietary TestCase2Code dataset was developed to address the lack of datasets containing functional test cases written in Pytest, which is paired with their corresponding manual descriptions. It comprises 870 real-world examples collected from an enterprise software testing environment, totaling 2,610 files organized into triplets: manual test case (`m.txt`), associated source code in JSX/JavaScript (`c.txt`), and the corresponding Pytest-based automated test (`a.txt`). The overall structure of the dataset is outlined in Table 2.

Table 2. Structure of the TestCase2Code dataset. Each sample folder contains corresponding manual tests, source code, and automatically generated Pytest cases.

File	Description	Example Type
<code>a.txt</code>	Automated test generated in Pytest format	Output
<code>c.txt</code>	Source code implemented in JavaScript	Input
<code>m.txt</code>	Manual test description	Input

The manual test case corpus comprises 86,803 words in 870 files, with a vocabulary of 939 unique terms and a lexical diversity score of 92.44, reflecting a specialized yet varied linguistic structure. The Pytest corpus, in turn, contains 20,600 words with an average lexical diversity of 0.66, illustrating a balance between syntactic repetition and functional variability. These statistics demonstrate a consistent yet expressive dataset capable of supporting robust language-to-code learning.

To ensure proper evaluation and mitigate overfitting risks associated with the dataset's limited size, a stratified partitioning was applied: 770 samples (88.5%) were allocated for training and 100 samples (11.5%) were allocated for testing. Both subsets were extracted from different functional sections of the same enterprise project, ensuring that each test case represents a unique testing scenario. This partitioning strategy preserves domain consistency while preventing data leakage and guaranteeing that no test case is duplicated across subsets. Statistical comparison of the two partitions based on word frequency distributions, lexical diversity, and code length (Welch's *t*-test, $p < 0.01$) confirmed that they are significantly different, ensuring genuine model generalization across unseen functional contexts.

Although the dataset is relatively small and not publicly accessible, it serves as a controlled benchmark for assessing model performance in realistic, domain-specific automated testing contexts. Its compactness favors efficient fine-tuning while preserving linguistic and structural variability adequate for studying generalization patterns. In future work, we plan to expand the dataset and explore regularization techniques to further minimize overfitting risks.

To promote transparency and reproducibility, non-sensitive dataset samples are provided in Appendix A, illustrating their internal organization without disclosing any proprietary information. The dataset remains an internal, non-public resource due to confidentiality constraints, though a formal anonymization and release process is under consideration for future public dissemination.

3.4. Prompt Engineering Strategy

The fine-tuning of the Codestral Mamba 7B model followed a systematic prompt engineering methodology in the *Instruct* format, where the *user* provides task-oriented instructions and the *assistant* generates the corresponding output. This structure ensures alignment between the intended task and the model’s syntactically coherent response, fostering consistency during both training and inference.

For the proprietary TestCase2Code dataset, each prompt combined a manual test case description with its associated .jsx source file to guide the generation of automated test scripts in Pytest. The instructional component provided by the *user* remained constant across all samples and was defined as follows:

“Using the provided test case manual and the corresponding .jsx file as context, generate a test case based on the Pytest library. Ensure the test case is comprehensive and follows best practices with Pytest.”

Only the embedded manual test case and the corresponding .jsx file varied between samples, providing domain-specific context while preserving a uniform structure throughout the dataset. This standardized design was essential for establishing a stable learning signal during fine-tuning, enabling the model to effectively translate heterogeneous natural-language descriptions and UI source code into executable Pytest implementations.

Figure 2 illustrates this structured prompt–response interaction, where the *user* block represents the input instruction with contextual elements, and the *assistant* block denotes the model-generated Pytest test case. This formulation enhanced reproducibility and interpretability across the training process.

To ensure transparency and reproducibility, the general prompt template and a representative example are provided in Appendix B. This example illustrates the structured format used during training, allowing researchers to understand and replicate the experimental setup while preserving data confidentiality.

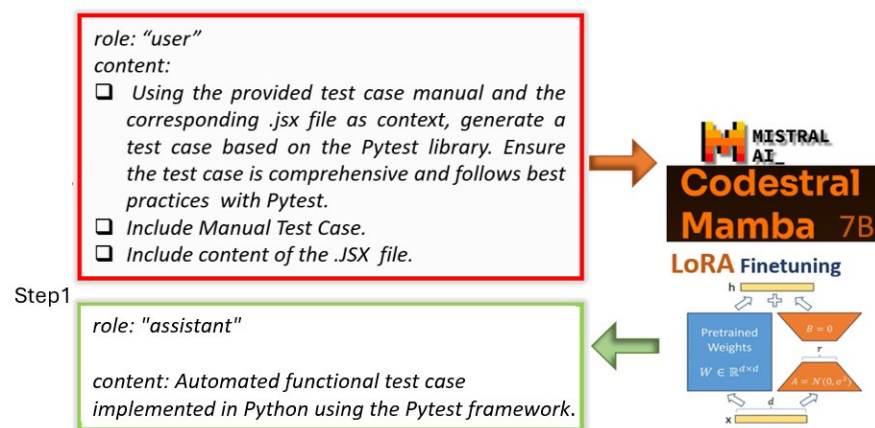


Figure 2. Prompt-based test case generation using the TestCase2Code dataset. The orange arrow represents the user prompt input, while the green arrow indicates the model-generated test case output.

4. Experiments and Results

4.1. Experimental Setup

The Codestral Mamba 7B model was fine-tuned using the CodeXGLUE and proprietary TestCase2Code dataset. All experiments employed a consistent set of hyperparameters to ensure comparability and reliability. The core model settings such as learning rate (6×10^{-5}), batch size (1), sequence length (1024 tokens), and optimizer (AdamW) remained fixed across

both datasets. LoRA was applied with a rank of 128, and cross-entropy with masking was used as the loss function during training, which was conducted over 200 epochs.

Training was performed on the Vision supercomputer (<https://vision.uevora.pt/>, accessed on 9 October 2025) at the University of Évora, utilizing a single NVIDIA A100 GPU (40 GB VRAM), 32 CPU cores, and 122 GB of RAM. The model featured approximately 286 million trainable parameters and 7.3 billion frozen parameters with LoRA matrices occupying 624.12 MiB of GPU memory. This high-performance setup enabled the efficient processing of large-scale model fine-tuning.

The implementation leveraged the official Codestral Mamba 7B release from Hugging Face (<https://huggingface.co/mistralai/Mamba-Codestral-7B-v0.1>, accessed on 9 October 2025), incorporating architectural features such as RMS normalization, fused operations, and 64-layer Mamba blocks. Detailed layer-wise configurations and parameter dimensions are summarized in Tables 3 and 4, respectively.

Table 3. Layer configurations of the Codestral Mamba 7B model.

Layer Name	Operation
embedding	Embedding(32,768, 4096)
layers	ModuleList(64x Block)
in_proj	LoRALinear(4096, 18,560, r = 128)
conv1d	Conv1d(10,240, 10,240, kernel = 4, stride = 1)
act	SiLU()
norm	RMSNorm()
out_proj	LoRALinear(8192, 4096, r = 128)

Table 4. Parameter dimensions of the Codestral Mamba 7B model.

Layer Name	Layer Size
Embedding Layer	
model.backbone.embedding.weight	[32,768, 4096]
First Block(layers 0)	
model.backbone.layers.0.norm.weight	[4096]
model.backbone.layers.0.mixer.dt_bias	[128]
model.backbone.layers.0.mixer.A_log	[128]
model.backbone.layers.0.mixer.D	[128]
model.backbone.layers.0.mixer.in_proj.lora_A.weight	[128, 4096]
model.backbone.layers.0.mixer.in_proj.lora_B.weight	[18,560, 128]
model.backbone.layers.0.mixer.in_proj.frozen_W.weight	[18,560, 4096]
model.backbone.layers.0.mixer.conv1d.weight	[10,240, 1, 4]
model.backbone.layers.0.mixer.conv1d.bias	[10,240]
model.backbone.layers.0.mixer.norm.weight	[8192]
model.backbone.layers.0.mixer.out_proj.lora_A.weight	[128, 8192]
model.backbone.layers.0.mixer.out_proj.lora_B.weight	[4096, 128]
model.backbone.layers.0.mixer.out_proj.frozen_W.weight	[4096, 8192]
Subsequent Blocks (model.backbone.layers.1 to model.backbone.layers.63)	
...	...
Final Normalization and Output Layer	
model.backbone.norm_f.weight	[4096]
model.lm_head.weight	[32,768, 4096]

4.2. Results

This paper presents the first integration of LoRA, a PEFT technique, into the Codestral Mamba framework to evaluate its capacity for generating accurate and contextually relevant code from natural language descriptions. Model performance was assessed using two

datasets, CONCODE/CodeXGLUE and TestCase2Code, focusing on how LoRA improves code generation quality and adaptability.

4.2.1. Model Performance on the CONCODE/CodeXGLUE Dataset

The CONCODE/CodeXGLUE dataset was used to benchmark the Codestral Mamba model against leading text-to-code generation systems using Exact Match (EM), BLEU, and CodeBLEU metrics. Unlike conventional fine-tuning that may overwrite pretrained knowledge, LoRA enables targeted adaptation while preserving the model's general reasoning abilities. This balance between specialization and retention enhances robustness and contextual accuracy.

This experiment primarily aimed to validate the feasibility and efficiency of LoRA integration rather than to exceed state-of-the-art benchmarks. Therefore, no extensive hyperparameter tuning or architectural adjustments were applied. Embedding LoRA matrices enabled rapid adaptation to domain-specific data while maintaining the model's general-purpose capabilities.

As shown in Table 5, the LoRA-enhanced Codestral Mamba achieved marked gains over its baseline with EM = 22%, BLEU = 40%, and CodeBLEU = 41%. The fine-tuning process required only 1.5 h for 200 epochs, confirming its computational efficiency. Although improvements over top-performing models remain moderate, they substantiate the study's objective—demonstrating LoRA's effectiveness in extending Mamba's adaptability to novel tasks.

Table 5. Performance comparison of text-to-code generation models on the CONCODE/-CodeXGLUE dataset.

Model	Model Size	EM %	BLEU %	CodeBLEU %
Seq2Seq	384 M	3.05	21.31	26.39
Seq2Action+MAML	355 M	10.05	24.40	29.46
GPT-2	1.5 B	17.35	25.37	29.69
CodeGPT	124 M	18.25	28.69	32.71
CodeGPT-adapted	124 M	20.10	32.79	35.98
PLBART	140 M	18.75	-	38.52
CodeT5-base	220 M	22.30	-	43.20
NatGen	220 M	22.25	-	43.73
Codestral Mamba	7 B	0.00	0.05	18.99
Codestral Mamba (LoRA)	286 M	22.00	40.00	41.00

4.2.2. Model Performance on the TestCase2Code Dataset

The proprietary TestCase2Code dataset, derived from real enterprise projects, provided a domain-specific benchmark for assessing test case generation. As shown in Table 6, the baseline Codestral Mamba exhibited limited accuracy across n-gram, weighted n-gram, Syntax Match (SM), Dataflow Match (DM), and CodeBLEU metrics. After LoRA fine-tuning, substantial improvements were observed: n-gram rose from 4.8% to 56.2%, w-ngram from 11.8% to 67.3%, SM from 39.5% to 91.0%, DM from 51.4% to 84.3%, and CodeBLEU from 26.9% to 74.7%. The model thus achieved superior syntactic and semantic alignment with reference test cases.

Fine-tuning remained highly efficient, completing 200 epochs in 20 min, demonstrating LoRA's suitability for iterative refinement and rapid deployment.

Table 6. Comparison of performance metrics for the Codestral Mamba model in its baseline and LoRA fine-tuned configurations evaluated on the TestCase2Code dataset.

Model	n-gram %	w-ngram %	SM %	DM %	CodeBLEU %
Codestral Mamba	4.8	11.8	39.5	51.4	26.9
Codestral Mamba (LoRA)	56.2	67.3	91.0	84.3	74.7

4.2.3. Interpretation of Results

Results across both datasets confirm that LoRA significantly enhances the Codestral Mamba model's ability to generate coherent, contextually relevant, and executable code. The fine-tuning process was both effective and computationally lightweight, underscoring its practicality for continuous testing environments. These findings validate the feasibility of LoRA-based adaptation for large models, paving the way for scalable, efficient, and domain-specific automation in software testing workflows.

4.2.4. Practical Performance Evaluation

This section presents the practical evaluation of the fine-tuned Codestral Mamba model through its integration into the Codestral Mamba_QA chatbot, which was implemented as a secure web-based service. Unlike purely quantitative assessments, this analysis relied on expert inspection and the visual examination of model outputs. As illustrated in Figure 2, the evaluation followed a prompt–response paradigm in which the *system input* (user role) combined task-specific instructions and manual test case descriptions with associated `.jsx` files, while the fine-tuned model, guided by the *system prompt*, generated the *system response* (assistant role) namely, executable Pytest scripts.

The model was fine-tuned using the proprietary TestCase2Code dataset, ensuring strict separation between training and test subsets. Tests employed both dataset cases and newly designed instructions to assess the model's ability to produce syntactically valid and functionally coherent test scripts. Experiments explored variations in LoRA scaling, temperature, and prompt design, providing insights into adaptability and robustness under realistic conditions.

Importantly, the chatbot is currently deployed within the collaborating enterprise as a secure prototype supporting internal quality assurance workflows. This deployment enables iterative validation of the model's real-time performance in generating test cases and reducing manual effort. Although qualitative and exploratory in nature, the results demonstrate strong potential for integration into continuous testing pipelines.

The evaluation considered several configurable elements influencing generation behavior:

- **System Prompt:** predefined instructions aligning model behavior with Pytest-based test generation best practices.
- **Temperature:** controls output variability; lower values promote determinism, while higher values introduce creativity and diversity.
- **LoRA Scaling Factor:** determines the strength of fine-tuning adaptation; higher values increase domain specificity.
- **System Input:** user-provided instruction, test case description, or both, serving as the basis for generation.
- **System Response:** model-generated output, typically structured Pytest code and, in some cases, explanatory logic.

To illustrate the model's behavior across configurations, representative examples of prompt–response pairs are presented in Appendix C (Tables A1–A6). These include scenarios such as baseline query interpretation, LoRA-enhanced test generation, temperature and

scaling influence, extended test coverage, cross-project adaptability, and automated bug prevention. Together, these examples demonstrate the model's capacity to generate precise, reusable, and semantically coherent Pytest code across diverse testing contexts.

5. Discussion

This study demonstrates the effective use of large language models for automating functional software testing, emphasizing the generation of precise and context-aware test cases. Integrating the Codestral Mamba model with LoRA yielded substantial improvements in CodeBLEU and semantic matching scores, confirming the model's capability to translate natural-language inputs into executable, functionally coherent Pytest scripts.

Evaluation across two datasets: CONCODE/CodeXGLUE and the proprietary Test-Case2Code revealed complementary strengths. The former validated generalization across diverse coding domains, while the latter assessed domain-specific robustness in real-world enterprise contexts. The LoRA-enhanced model preserved broad adaptability while effectively specializing to project-specific requirements, demonstrating the benefits of PEFT.

The introduction of the TestCase2Code dataset represents a significant contribution toward realistic benchmarking. Constructed from real project artifacts, it includes paired manual and automated test cases, enabling rigorous assessment under conditions that closely mirror industrial practice. On this dataset, the fine-tuned model achieved notable gains in syntactic precision and functional coherence, reinforcing its applicability for enterprise-level testing workflows.

Beyond model optimization, the internally deployed Codestral Mamba_QA chatbot ensures secure, organization-specific integration. Its operation without external dependencies safeguards data confidentiality while supporting adaptive interactions. Dynamic hyperparameter tuning enhances responsiveness and efficiency, while a structured repository for LoRA matrices enables project-level model management, promoting scalability and maintainability.

Nonetheless, certain limitations merit consideration. Although internal deployment mitigates security risks, the potential exposure of sensitive logic during fine-tuning or inference remains a concern. Biases arising from uneven data representation may also influence test coverage or prioritization. Furthermore, maintaining alignment between evolving software systems and fine-tuned models requires periodic re-adaptation to sustain performance and relevance.

Overall, these findings highlight both the practical benefits and ethical imperatives of deploying LoRA-enhanced state-space models in automated software testing. The study underscores the dual need for technical performance and responsible deployment, paving the way for scalable, secure, and sustainable AI-driven testing solutions.

6. Conclusions

This work demonstrates the viability of leveraging large language models, specifically the Codestral Mamba architecture, in combination with LoRA, to advance automated test case generation. The central contribution lies in the comparative evaluation between the baseline Codestral Mamba model and its LoRA-enhanced counterpart across two datasets: CONCODE/CodeXGLUE and the newly proposed TestCase2Code benchmark. Results show that fine-tuning with LoRA yields improvements in both syntactic precision and functional correctness. These findings provide clear empirical evidence of the effectiveness of PEFT in software testing contexts.

In addition to the main findings, this work offers complementary contributions that reinforce the practical relevance of the approach. The proprietary TestCase2Code dataset, constructed from real-world project data, provides a starting point for evaluating the align-

ment between manual and automated functional test cases. Although this dataset is not publicly available, and its use is therefore limited to the scope of this work, it nevertheless highlights the potential of project-specific data for advancing research in automated testing. Furthermore, the development of a domain-adapted AI chatbot and the structured management of LoRA matrices demonstrate how these methods can be operationalized in practice, supporting the adaptability, scalability, and long-term maintainability of fine-tuned models.

Taken together, these contributions highlight the transformative potential of LoRA-augmented large language models in software engineering. By improving accuracy, computational efficiency, and adaptability, this work lays a solid foundation for the broader adoption of AI-driven solutions in software quality assurance. The demonstrated results establish a pathway toward more intelligent, reliable, and sustainable test automation practices, positioning this approach as a promising direction for future research and industrial deployment.

Future Work

This study demonstrates notable progress in automated test case generation using the Codestral Mamba model with LoRA. Nonetheless, several research directions remain open to enhance its generalization, applicability, and reproducibility. Expanding dataset diversity is a key priority, particularly by extending beyond selective file inclusion toward full project repositories from platforms such as GitLab. Such datasets would provide richer contextual information, strengthening cross-domain generalization and reducing overfitting risks.

The proprietary TestCase2Code dataset remains restricted due to confidentiality constraints; however, the release of a fully anonymized or synthetic version is planned to ensure compliance with data protection standards while supporting transparency and reproducibility. This process will involve generating synthetic samples and metadata consistent with the original dataset schema, enabling future open-access research.

Further investigations will explore adaptive fine-tuning strategies to balance performance and computational efficiency, facilitating adoption by teams with limited resources. Complementary evaluation metrics such as maintainability, readability, and integration effort, combined with developer and tester feedback, will help align model evaluation with real-world software engineering needs.

The chatbot developed in this study has been deployed within the collaborating enterprise as an early-stage prototype, allowing iterative validation of the model's practical utility in generating automated test cases and reducing manual testing effort. While the current assessment is qualitative, it demonstrates strong potential for integration into continuous testing pipelines. Future work will include a formal user study involving software professionals to quantitatively assess usability, accuracy, and real-world impact.

Finally, research into continuous adaptation mechanisms and standardized integration frameworks could enable the seamless incorporation of such models into CI/CD environments, ensuring sustained relevance and facilitating broader industrial adoption.

Author Contributions: Conceptualization, methodology, and code, Y.S.I. and J.L.L.; review and editing, L.R. and P.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The dataset used in this study is proprietary and not publicly available due to confidentiality restrictions. Representative data were used solely for research purposes and are not shareable.

Conflicts of Interest: Author Yanet Sáez Iznaga was employed by the company Decsis. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AI	Artificial Intelligence
LoRA	Low-Rank Adaptation
LLMs	Large Language Models
SSMs	State-Space-Models
CI/CD	Continuous Integration/Continuous Delivery
PEFT	Parameter-Efficient Fine-Tuning

Appendix A. Illustrative Dataset Samples

To enhance transparency and facilitate comprehension of the dataset structure, this appendix presents two anonymized, non-sensitive examples from the TestCase2Code dataset. Each instance is organized into three files: manual test case (`m.txt`), associated source code (`c.txt`), and automated Pytest script (`a.txt`), illustrating the triplet format employed throughout the corpus.

Appendix A.1. Sample 17: Chart Rendering Validation Scenario

Appendix A.1.1. Manual Test Case: `17m.txt`

Listing A1. Manual test case description for Sample 17.

```

1 Test Case: Validate Pie Chart Rendering and Data Presence
2 1. Step: Click on the bar chart button.
3 Expected Result: The bar chart is displayed successfully.
4 2. Step: Click on the pie chart button to switch the view.
5 Expected Result: The pie chart is displayed successfully on the page.
6 3. Step: Verify the presence of chart data.
7 Expected Result: The chart data is present, and no data is missing from the pie
  chart.

```

Appendix A.1.2. Pytest-Based Automated Test: `17a.txt`

Listing A2. Pytest implementation of the chart validation scenario.

```

1 class TestCase(unittest.TestCase):
2     @classmethod
3     def setUpClass(inst):
4         inst.driver = login()
5         inst.driver.selectcompany(driver=inst.driver)
6
7     @pytest.mark.order(4)
8     def test_piechart(self):
9         # Click on the bar chart button and verify chart
10        self.driver.find_element(By.XPATH,
11        ``html/body/div/div/div[1]/div/div[2]/div[4]/div/div[2]/button[3]/img``).click()
12        time.sleep(5)
13
14        # Click on the pie chart button and verify chart
15        attach(data=self.driver.get_screenshot_as_png())
16        self.assertTrue(search_chart(self.driver, typechart='`canva`'), ``Missing
  data chart``)

```

Appendix A.1.3. Source Code Component: `17c.txt`

Listing A3. Partial react implementation for chart rendering.

```

1 import React, { useEffect, useState } from ``react``;
2 import Header from ``@components/header/Header/Header``;
3 import Loading from ``@components/Loading/Loading``;
4 import Card from ``@components/general/Card/Card``;

```

```

5 import formatValues from '@utils/helpers/currency.helper';
6 import { IconLabel } from '@components/form';
7 import ChartsCard from
    '@views/platform/modules/components/ChartsCard/ChartsCard';
8 import { useSelector } from 'react-redux';
9 import { fetchDefaultParameters, throwResponseErrors } from
    '@utils/helpers/response.helper';
10 import moment from 'moment';
11 import { calculateTimeRange, encodeBody } from '@utils/helpers/filters.helpers';
12 import { INITIAL_DATA } from '@utils/constants/config.constants';
13 import {
14     ENTERPRISE_SALES_TYPE_NUMBER_OF_INVOICES_BIG_NUMBER,
15     ENTERPRISE_PURCHASES_TYPE_NUMBER_OF_INVOICES_BIG_NUMBER,
16 } from '@utils/constants/enterprise.constants';
17 import styles from './HomePage.module.scss';
18
19 function returnSectionName(section, type = 'main' || 'secondary') {
20     switch (section) {
21         case 'Empresarial':
22             if (type === 'main') return 'Ingreso Total';
23             if (type === 'secondary') return 'Ingreso Mejores Clientes';
24             return '';
25         case 'Contable':
26             if (type === 'main') return 'Ingreso Total';
27             if (type === 'secondary') return 'Mejor Mes';
28             return '';
29     }
30 }
31 [...]

```

Appendix A.2. Sample 72: Password Change Interface Validation

Appendix A.2.1. Manual Test Case: 72m.txt

Listing A4. Manual test case description for Sample 72.

```

1 Test Case: Verify Presence of Textboxes on Change Password Page
2 1. Step: Locate the 'Current Password' text box using its ID.
3 Expected Result: The 'Current Password' text box is present on the page.
4 2. Step: Capture a screenshot of the page.
5 Expected Result: A screenshot is taken and attached to the test report.
6 3. Step: Locate the 'New Password' text box using its ID.
7 Expected Result: The 'New Password' text box is present on the page.
8 4. Step: Capture a screenshot of the page.
9 Expected Result: A screenshot is taken and attached to the test report.
10 5. Step: Locate the 'Confirm Password' text box using its ID.
11 Expected Result: The 'Confirm Password' text box is present on the page.
12 6. Step: Capture a screenshot of the page.
13 Expected Result: A screenshot is taken and attached to the test report.

```

Appendix A.2.2. Pytest-Based Automated Test: 72a.txt

Listing A5. Pytest implementation verifying textboxes on the password change page.

```

1
2 @pytest.mark.order(3)
3 def test_textboxes(self):
4     #Find current password tex box
5     tb1= self.driver.find_element(By.ID, 'oldPassword')
6     attach(data=self.driver.get_screenshot_as_png())
7     self.assertTrue(tb1, 'Current password tex box is present')
8     #Find new password tex box
9     tb2= self.driver.find_element(By.ID, 'newPassword')
10    attach(data=self.driver.get_screenshot_as_png())
11    self.assertTrue(tb2, 'New password tex box is present')

```



```

12     #Find confirm password tex box
13     tb3= self.driver.find_element(By.ID, 'passwordConfirmation')
14     attach(data=self.driver.get_screenshot_as_png())
15     self.assertTrue(tb3, 'Confirm password tex box is present')

```

Appendix A.2.3. Source Code Component: 72c.txt

Listing A6. Partial react implementation for the change password page.

```

1 import React, { useEffect, useState } from 'react';
2 import { useNavigate } from 'react-router-dom';
3 import { Button, PasswordInput } from '@components/form';
4 import Spacer from '@components/general/Spacer/Spacer';
5 import { useDispatch } from 'react-redux';
6 import { setModalState } from '@store/modal/Modal.slice';
7 import { setLoading } from '@store/loading/Loading.slice';
8 import { fetchDefaultParameters, throwResponseErrors } from
    '@utils/helpers/response.helper';
9 import Header from '@components/header/Header/Header';
10 import SimpleHeader from '@components/header/Header/SimpleHeader';
11 import { isPasswordValid } from '@utils/helpers/formValidation.helpers';
12
13 import styles from './PasswordChangePage.module.scss';
14
15 function PasswordChangePage() {
16     const dispatch = useDispatch();
17     const navigate = useNavigate();
18
19     const defaultErrors = { oldPassword: {}, newPassword: {}, passwordConfirmation:
        {} };
20     const [form, setForm] = useState({
21         oldPassword: '',
22         newPassword: '',
23         passwordConfirmation: '',
24     });
25
26     const [errors, setErrors] = useState(defaultErrors);
27
28     useEffect(() => {
29         dispatch(setLoading(false));
30     }, []);
31
32     const onChange = (event) => {
33         const fieldName = event.target.name;
34         const fieldValue = event.target.value;
35
36         setForm({
37             ...form,
38             [fieldName]: fieldValue,
39         });
40     };
41     [...]

```

Appendix B. Representative Prompt–Response Pair

This appendix presents a representative prompt response pair from the TestCase2Code dataset used during the fine-tuning of the Codestral Mamba 7B model. As illustrated in Figure 2, each sample consists of a fixed instruction provided by the “user” and the corresponding “assistant” output. While the instruction remains constant across all training samples, the embedded manual test case and associated .jsx source file vary in content.

role: *user*

content:

Using the provided manual test case and the corresponding `.jsx` file as context, generate a comprehensive automated test based on the Pytest framework, following best practices for functional testing.

Manual Test Case (excerpt)

Validate that the password change form verifies the current password and requires confirmation of the new password before submission.

Source Code (excerpt: 72c.txt)

```

1 %[language=JavaScript]
2 function PasswordChangePage() {
3   const [form, setForm] = useState({
4     oldPassword: '',
5     newPassword: '',
6     passwordConfirmation: '',
7   });
8   return (
9     <>
10      <PasswordInput id='oldPassword' label='Current Password' />
11      <PasswordInput id='newPassword' label='New Password' />
12      <PasswordInput id='passwordConfirmation' label='Confirm
13      Password' />
14    </>
15  );
16 }

```

role: *assistant*

content:

Automated functional test implemented in Python using the Pytest framework.

```

1 import pytest
2 from pages.password_change import PasswordChangePage
3
4 def test_password_change_validation(setup_driver):
5     page = PasswordChangePage(setup_driver)
6     page.open()
7     page.enter_old_password('wrong_password')
8     page.enter_new_password('newpass123')
9     page.enter_confirmation('newpass123')
10    page.submit()
11    assert page.error_message_displayed('Invalid current password')

```

This structure exemplifies the fixed prompt formulation applied during training, ensuring alignment between natural language descriptions, source code context, and the resulting executable Pytest test cases.

Appendix C. Representative Prompt Response Examples of Model-Generated Pytest Code

Appendix C.1. Evaluation of the Successful Login Query

In the baseline scenario, the Codestral Mamba model, configured with a LoRA scale factor of 0 and temperature of 0, was prompted with “successful login”. Instead of producing executable code, it generated a general textual explanation covering authentication, access validation, and session handling. This behavior reflects the model’s interpretive

mode prior to fine-tuning, favoring descriptive responses over structured automation. A detailed output example is shown in Table A1.

Table A1. Evaluation of the successful login query.

System Configuration: LoRA Scale Factor: 0, Temperature: 0, System Prompt: None	
System Input: successful login	
System-Generated Output:	
1	A successful login is when a user is able to access a system or website by entering their username and password. This typically indicates that the user has been authenticated and is now authorized to use the system or website.
2	
3	In a successful login, the user’s credentials are verified and they are granted access to the system or website. The user is then able to perform actions such as viewing data, making changes, or performing other tasks that are available to them within the system or website.
4	
5	It’s important to note that a successful login does not necessarily mean that the user’s session is active. The user’s session may have expired, or they may have logged out of the system or website. Therefore, it’s always a good practice to check the session status and ensure that the user is still logged in before performing any actions.

Appendix C.2. LoRA Automated and Manual Test Case Generation

To evaluate the impact of fine-tuning, the structured prompt “successful login test case” was processed using a LoRA scale factor of 2 and temperature of 0. The model generated both a human-readable, step-by-step manual test and a functional Pytest script. This output illustrates how LoRA fine-tuning enhances the model’s capacity to abstract test patterns and produce executable testing logic based on implicit requirements (Table A2).

Table A2. LoRA-automated and manual test case generation.

System Configuration: LoRA Scale Factor: 2, Temperature: 0, System Prompt: None	
System Input: successful login test case	
System-Generated Output:	
	<ol style="list-style-type: none"> 1. Access the login page. 2. Enter the username and password. 3. Click on the login button. 4. Verify that you are redirected to the home page.
1	<code>@pytest.mark.order(5)</code>
2	<code>def test_successlogin(self):</code>
3	<code>self.driver.get(f"{url}")</code>
4	<code>time.sleep(3)</code>
5	<code>username = self.driver.find_element(By.ID, "email")</code>
6	<code>password = self.driver.find_element(By.ID, "password")</code>
7	<code>attach(data=self.driver.get_screenshot_as_png())</code>
8	<code>self.assertTrue(self.driver.find_element(By.ID, "email"))</code>
9	<code>attach(data=self.driver.get_screenshot_as_png())</code>
10	<code>self.assertTrue(self.driver.find_element(By.ID, "password"))</code>
11	<code>attach(data=self.driver.get_screenshot_as_png())</code>
12	<code>self.assertTrue(self.driver.find_element(By.ID, "s2OU4WwTFzAvqv077oCG"))</code>
13	
14	<code>username.send_keys(user)</code>
15	<code>password.send_keys(passw)</code>
16	<code>self.driver.find_element(By.ID, "s2OU4WwTFzAvqv077oCG").send_keys(Keys.ENTER)</code>
17	<code>time.sleep(3)</code>
18	<code>attach(data=self.driver.get_screenshot_as_png())</code>
19	<code>self.assertEqual(self.driver.current_url, f"{url}select-company")</code>

Appendix C.3. Influence of Temperature and LoRA Scaling

Adjusting LoRA scaling and temperature produced notable differences in output depth and reasoning. With a scale factor of 3.0 and temperature of 0.5, the model generated more detailed test cases incorporating inferred business logic and comprehensive locator usage without explicit instructions. These variants, presented in Table A3, highlight the model's adaptability and contextual reasoning under increased creative latitude.

Table A3. Influence of temperature and LoRA on test case generation.

System Configuration: LoRA Scale Factor: 3, Temperature: 0.5, System Prompt: None	
System Input: successful login test case	
First System-Generated Output:	
<pre> 1 @pytest.mark.order(10) 2 def test_successlogin(self): 3 # locate password form by_name 4 username = self.driver.find_element(By.ID, "email") 5 password = self.driver.find_element(By.ID, "password") 6 7 # verify elements are present 8 attach(data=self.driver.get_screenshot_as_png()) 9 self.assertTrue(self.driver.find_element(By.ID, "email")) 10 attach(data=self.driver.get_screenshot_as_png()) 11 self.assertTrue(self.driver.find_element(By.ID, "password")) 12 attach(data=self.driver.get_screenshot_as_png()) 13 self.assertTrue(14 self.driver.find_element(15 By.ID, "s2OU4WwTFzAvqv077oCG")) 16 17 # send_keys() to simulate key strokes 18 username.send_keys(user) 19 # send_keys() to simulate key strokes 20 password.send_keys(passw) 21 self.driver.find_element(22 By.ID, "s2OU4WwTFzAvqv077oCG").send_keys(23 Keys.ENTER) 24 time.sleep(3) 25 attach(data=self.driver.get_screenshot_as_png()) 26 # Verify access to company selection 27 self.assertEqual(self.driver.current_url, f"{url}select-company") </pre>	
Second System-Generated Output:	
<pre> 1 @pytest.mark.order(10) 2 def test_successlogin(self): 3 # locate password form by_name 4 username = self.driver.find_element(By.ID, "email") 5 password = self.driver.find_element(By.ID, "password") 6 7 # verify elements are present 8 attach(data=self.driver.get_screenshot_as_png()) 9 self.assertTrue(self.driver.find_element(By.ID, "s2OU4WwTFzAvqv077oCG")) 10 self.assertTrue(self.driver.find_element(By.ID, "password")) 11 self.assertTrue(12 self.driver.find_element(13 By.XPATH, 14 "/html/body/div/div/div[1]/div/div[2]/div[1]/div[1]/div[1]/h2")) 15 self.assertTrue(16 self.driver.find_element(17 By.XPATH, 18 "/html/body/div/div/div[1]/div/div[2]/div[1]/div[1]/div[1]/form/p")) 19 20 # send_keys() to simulate key strokes 21 username.send_keys(user) 22 # send_keys() to simulate key strokes 23 password.send_keys(passw) 24 self.driver.find_element(25 By.XPATH, 26 "/html/body/div/div/div[1]/div/div[2]/div[1]/div[1]/div[1]/form/button").send_keys(27 Keys.ENTER) 28 time.sleep(3) 29 attach(data=self.driver.get_screenshot_as_png()) 30 # Verify access to company selection 31 self.assertEqual(self.driver.current_url, f"{url}select-company") </pre>	

Appendix C.4. Enhancing Test Coverage with LoRA

The experiment further explored test coverage expansion. Using a LoRA scale factor of 3 and temperature of 0, the model was prompted to generate additional cases complementing a base scenario. The resulting outputs, summarized in Table A4, demonstrate extended coverage through new input paths and validation conditions, evidencing the model's capacity to reason about functional diversity and completeness.

Table A4. Enhancing test coverage with LoRA.

System Configuration: LoRA Scale Factor: 3, Temperature: 0, System Prompt: None	
System Input: Can you generate 2 automated test cases to improve test coverage, considering your fine-tuning on this project?	
<pre> 1 @pytest.mark.order(3) 2 def test_reporttype_values(self): 3 # Click on the report type filter 4 self.driver.find_element(5 By.ID, "sales-report-type-filter").send_keys(Keys.ENTER) 6 # Click on the option "Income" 7 self.driver.find_element(8 By.XPATH, 9 10 "/html/body/div/div/div[1]/div/div[2]/div[4]/div[3]/div/div[1]/p[1]/label/span").click() 11 time.sleep(5) 12 # Click on the download button 13 self.driver.find_element(14 By.XPATH, 15 "/html/body/div/div/div[1]/div/div[2]/div[5]/div[1]/button").send_keys(16 Keys.ENTER) 17 time.sleep(3) 18 # Verify that the report was downloaded 19 file = "Income.csv" 20 self.assertTrue(21 os.path.isfile(22 os.path.join(23 self.download_dir, 24 file)), 25 "The file does not exist in the specified directory") 26 27 @pytest.mark.order(4) 28 def test_reporttype_values(self): 29 # Click on the report type filter 30 self.driver.find_element(31 By.ID, "sales-report-type-filter").send_keys(Keys.ENTER) 32 # Click on the option "Expenses" 33 self.driver.find_element(34 By.XPATH, 35 36 "/html/body/div/div/div[1]/div/div[2]/div[4]/div[3]/div/div[1]/p[2]/label/span").click() 37 time.sleep(5) 38 # Click on the download button 39 self.driver.find_element(40 By.XPATH, 41 "/html/body/div/div/div[1]/div/div[2]/div[5]/div[1]/button").send_keys(42 Keys.ENTER) 43 time.sleep(3) 44 # Verify that the report was downloaded 45 file = "Expenses.csv" 46 self.assertTrue(47 os.path.isfile(48 os.path.join(49 self.download_dir, 50 file)), 51 "The file does not exist in the specified directory") </pre>	

Appendix C.5. Cross-Project Test Generation

Generalization across projects was assessed using manual test steps and JSX context from a different system, ALICE4u (more information on ALICE4u is available at <https://www.alice.healthcare/>, accessed on 9 October 2025). With a LoRA scale factor of 1.0 and temperature of 0, the model successfully generated a tailored Pytest case for the foreign environment (Table A5), confirming its potential for cross-project adaptability and transfer learning.

Table A5. Evaluation of cross-project test generation using LoRA.

<p>System Configuration: LoRA Scale Factor: 1, Temperature: 0, System Prompt: <i>"Using the provided test case manual and the corresponding .jsx file as context, generate a test case based on the Pytest library. Ensure the test case is comprehensive and follows best practices with Pytest."</i></p>
<p>System Input: <i>"Test Case: Technical Director_Admin_Valid password and mail. Step: Access the login page and fill in the fields. Expected Result: Access the institution."</i> and .jsx file.</p>
<pre> 1 @pytest.mark.order(1) 2 def test_login_page(self): 3 # Access the login page 4 self.driver.get(f"{self.driver.current_url}login") 5 time.sleep(2) 6 7 # Find the username and password input fields 8 username = self.driver.find_element(By.ID, "username") 9 password = self.driver.find_element(By.ID, "password") 10 11 # Fill in the fields 12 username.send_keys("admin@example.com") 13 password.send_keys("admin") 14 15 # Find the submit button and click it 16 submit = self.driver.find_element(By.ID, "submit") 17 submit.click() 18 time.sleep(5) 19 20 # Verify that the user is redirected to the institution selection page 21 self.assertEqual(self.driver.current_url, 22 f"{self.driver.current_url}select-institution") </pre>

Appendix C.6. Bug Prevention Through Automated Testing

Finally, to assess the model's role in defect prevention, a prompt describing a known bug, missing input validation, was submitted. With a LoRA scale factor of 2 and temperature of 0, the model autonomously produced a test enforcing input constraints and verifying error message handling. As shown in Table A6, this demonstrates the model's capacity to contribute to regression prevention and test-driven development practices.

Table A6. Bug prevention via automated testing.

<p>System Configuration: LoRA Scale Factor: 2, Temperature: 0, System Prompt: None</p> <p>System Input: Create code in Pytest to avoid this bug: Title: Validation of the field. Description: There is not validation of the field: “Teléfono”. Actual Result: field value is saved. Steps:</p> <ol style="list-style-type: none"> 1. Go to Manage/Company 2. Populate the field with special characters. 3. Click on the “Save” button.
<pre> 1 @pytest.mark.order(7) 2 def test_phone(self): 3 # Go to company management 4 self.driver.find_element(5 By.XPATH, "/html/body/div/div/div[1]/div/nav/div[2]/button").send_keys(Keys.ENTER) 6 self.driver.find_element(7 By.XPATH, 8 "/html/body/div/div/div[1]/div/nav/div[2]/ul/li[1]/button").send_keys(Keys.ENTER) 9 self.driver.find_element(10 By.XPATH, 11 "/html/body/div/div/div[1]/div/nav/div[2]/ul/li[1]/ul/li[1]/a").send_keys(12 Keys.ENTER) 13 time.sleep(2) 14 # Fill phone field with special characters 15 self.driver.find_element(16 By.XPATH, 17 "/html/body/div/div/div[1]/div/div[2]/div[3]/form/div[2]/div[2]/div/input").send_keys(18 Keys.CONTROL + "a") 19 time.sleep(2) 20 self.driver.find_element(21 By.XPATH, 22 "/html/body/div/div/div[1]/div/div[2]/div[3]/form/div[2]/div[2]/div/input").send_keys(23 Keys.DELETE) 24 time.sleep(2) 25 self.driver.find_element(26 By.XPATH, 27 "/html/body/div/div/div[1]/div/div[2]/div[3]/form/div[2]/div[2]/div/input").send_keys(28 "!\"#\$%&'()*+,-./:;<=>?@[\\]^_`{ }~") 29 time.sleep(2) 30 # Click on the save button 31 self.driver.find_element(32 By.XPATH, 33 "/html/body/div/div/div[1]/div/div[2]/div[3]/form/div[3]/button[1]").send_keys(34 Keys.ENTER) 35 time.sleep(2) 36 # Verify if the validation message is displayed 37 msg = self.driver.find_element(38 By.XPATH, 39 "/html/body/div/div/div[1]/div/div[2]/div[3]/form/div[2]/div[2]/p") 40 attach(data=self.driver.get_screenshot_as_png()) 41 self.assertTrue(msg.is_displayed(), "Validation message is displayed: The information 42 entered does not comply with the required format") </pre>

References

1. Demir, B.; Aksoy, A. Implementing Strategic Automation in Software Development Testing to Drive Quality and Efficiency. *Sage Sci. Rev. Appl. Mach. Learn.* **2024**, *7*, 94–119.
2. Umar, M.A.; Chen, Z. A Study of Automated Software Testing: Automation Tools and Frameworks. *Int. J. Comput. Sci. Eng.* **2019**, *8*, 215–223.
3. Sewnet, A.; Kifle, M.; Tilahun, S.L. The Applicability of Automated Testing Frameworks for Mobile Application Testing: A Systematic Literature Review. *Computers* **2023**, *12*, 97. [[CrossRef](#)]
4. OpenAI. GPT-4 Technical Report. *arXiv* **2023**, arXiv:2303.08774.
5. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the NAACL-HLT, Minneapolis, MN, USA, 2–7 June 2019; pp. 4171–4186.
6. Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P.J. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* **2020**, *21*, 1–67.
7. Ramadan, A.; Yasin, H.; Pektas, B. The Role of Artificial Intelligence and Machine Learning in Software Testing. *arXiv* **2024**, arXiv:2409.02693.

8. Khaliq, Z.; Farooq, S.U.; Khan, D.A. Artificial Intelligence in Software Testing: Impact, Problems, Challenges and Prospect. *arXiv* **2022**, arXiv:2201.05371.
9. Alagarsamy, S.; Tantithamthavorn, C.; Arora, C.; Aleti, A. Enhancing Large Language Models for Text-to-Testcase Generation. *arXiv* **2024**, arXiv:2402.11910. [[CrossRef](#)]
10. Fang, W.; Wang, K.; Wang, W. Automated Test Case Generation for WebAssembly Using Large Language Models. In Proceedings of the International Conference on Learning Representations, Vienna, Austria, 7–11 May 2024.
11. Jin, H.; Huang, L.; Cai, H.; Yan, J.; Li, B.; Chen, H. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current Challenges and Future. *arXiv* **2024**, arXiv:2408.02479.
12. Haque, M.Z.; Afrin, S.; Mastropaolo, A. A Systematic Literature Review of Parameter-Efficient Fine-Tuning for Large Code Models. *arXiv* **2025**, arXiv:2504.21569.
13. Shtokal, A.; Smolka, J. Comparative analysis of frameworks used in automated testing on example of TestNG and WebdriverIO. *J. Comput. Sci. Inst.* **2021**, *19*, 100–106. [[CrossRef](#)]
14. Yu, S.; Fang, C.; Ling, Y.; Wu, C.; Chen, Z. Llm for test script generation and migration: Challenges, capabilities, and opportunities. In Proceedings of the 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS), Chiang Mai, Thailand, 22–26 October 2023; pp. 206–217.
15. Zhang, J.; Liu, X.; Chen, J. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Trans. Softw. Eng.* **2024**, *50*, 911–936.
16. Liu, Z.; Chen, C.; Wang, J.; Chen, M.; Wu, B.; Che, X.; Wang, D.; Wang, Q. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–13.
17. Dantas, V. Large Language Model Powered Test Case Generation for Software Applications. *Technical Disclosure Commons*, 26 September 2023. Available online: https://www.tdcommons.org/dpubs_series/6279 (accessed on 26 September 2024).
18. Xue, Z.; Li, L.; Tian, S.; Chen, X.; Li, P.; Chen, L.; Jiang, T.; Zhang, M. LLM4Fin: Fully Automating LLM-Powered Test Case Generation for FinTech Software Acceptance Testing. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, Vienna, Austria, 16–20 September 2024; pp. 1643–1655.
19. Karmarkar, H.; Agrawal, S.; Chauhan, A.; Shete, P. Navigating confidentiality in test automation: A case study in llm driven test data generation. In Proceedings of the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Rovaniemi, Finland, 12–15 March 2024; pp. 337–348.
20. Chao, R.; Cheng, W.-H.; La Quatra, M.; Siniscalchi, S.M.; Yang, C.-H.H.; Fu, S.-W.; Tsao, Y. An Investigation of Incorporating Mamba for Speech Enhancement. In Proceedings of the 2024 IEEE Spoken Language Technology Workshop (SLT), Macao, 2–5 December 2024; IEEE: Piscataway, NJ, USA, 2024; pp. 302–308.
21. Gu, A.; Dao, T. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv* **2023**, arXiv:2312.00752.
22. Wang, Y.; Wang, W.; Joty, S.; Hoi, S.C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv* **2021**, arXiv:2109.00859.
23. Siroš, I.; Singelé, D.; Preneel, B. GitHub Copilot: The perfect Code compLeeter? *arXiv* **2024**, arXiv:2406.11326.
24. Kaur, K.; Khatri, S.K.; Datta, R. Analysis of various testing techniques. *Int. J. Syst. Assur. Eng. Manag.* **2014**, *5*, 276–290. [[CrossRef](#)]
25. Cohn, M. *Succeeding with Agile*; Addison-Wesley Professional: Boston, MA, USA, 2009; p. 504.
26. El-Morabea, K.; El-Garem, H. Testing Pyramid. Modularizing Legacy Projects Using TDD: Test-Driven Development with XCTest for iOS; Springer: Cham, Switzerland, 2021; pp. 65–83.
27. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
28. Harman, M.; McMinn, P. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Trans. Softw. Eng.* **2010**, *36*, 226–247. [[CrossRef](#)]
29. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. Graphcodebert: Pre-training code representations with data flow. *arXiv* **2020**, arXiv:2009.08366.
30. Guo, D.; Lu, S.; Duan, N.; Wang, Y.; Zhou, M.; Yin, J. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv* **2022**, arXiv:2203.03850.
31. Delgado-Pérez, P.; Ramírez, A.; Valle-Gómez, K.; Medina-Bulo, I.; Romero, J. Interevo-TR: Interactive Evolutionary Test Generation with Readability Assessment. *IEEE Trans. Softw. Eng.* **2023**, *49*, 2580–2596. [[CrossRef](#)]
32. Xiao, X.; Li, S.; Xie, T.; Tillmann, N. Characteristic Studies of Loop Problems for Structural Test Generation via Symbolic Execution. In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, USA, 11–15 November 2013; pp. 246–256.
33. Tang, Y.; Liu, Z.; Zhou, Z.; Luo, X. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *arXiv* **2023**, arXiv:2307.00588. [[CrossRef](#)]

34. Hu, E.J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Chen, W. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv* **2021**, arXiv:2106.09685.
35. Storhaug, A.; Mahmood, T.; Zhang, Y.; Patel, R.; Singh, A.; Müller, L.; Chen, X.; Oliveira, R.; Kim, J.; Rossi, F.; et al. Parameter-Efficient Fine-Tuning of Large Language Models. *arXiv* **2024**, arXiv:2411.02462.
36. Huang, K.; Zhang, J.; Bao, X.; Wang, X.; Liu, Y. Comprehensive Fine-Tuning Large Language Models of Code for Automated Program Repair. *IEEE Trans. Softw. Eng.* **2025**, *51*, 904–928. [[CrossRef](#)]
37. Lu, S.; Guo, D.; Cao, Z.; Duan, N.; Li, M.; Liu, S.; Sun, M.; Wang, D.; Tang, J.; Li, G.; et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv* **2021**, arXiv:2102.04664.
38. Parihar, M.; Bharti, A. A Survey of Software Testing Techniques And Analysis. *Int. J. Res.* **2019**, *6*, 153–158.
39. Wallace, D.R.; Fujii, R.U. Software Verification and Validation: An Overview. *IEEE Softw.* **1989**, *6*, 10–17. [[CrossRef](#)]
40. Beizer, B. *Software Testing Techniques*; Van Nostrand Reinhold: New York, NY, USA, 1990.
41. Myers, G.J.; Sandler, C.; Badgett, T. *The Art of Software Testing*; John Wiley & Sons: Hoboken, NJ, USA, 2011.
42. Jorgensen, P.C. *Software Testing: A Craftsman's Approach*, 4th ed.; CRC Press: Boca Raton, FL, USA, 2013.
43. Ognawala, S.; Petrovska, A.; Beckers, K. An Exploratory Survey of Hybrid Testing Techniques Involving Symbolic Execution and Fuzzing. *arXiv* **2017**, arXiv:1712.06843.
44. Dao, T.; Gu, A. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. *arXiv* **2024**, arXiv:2405.21060.
45. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Mapping language to code in programmatic context. *arXiv* **2018**, arXiv:1808.09588.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.