Preprints.org

Article

Not peer-reviewed version

# Integrating Large Language Models into Automated Software Testing

Yanet Sáez Iznaga [*,†] , Luís Rato [*] , Pedro Salgueiro , Javier Lamar León [†]

Posted Date: 18 September 2025

doi: 10.20944/preprints202509.1433.v1

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

*Article*

# Integrating Large Language Models into Automated Software Testing

**Yanet Sáez Iznaga [1,2,*,†], 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 Évora, Portugal
[2] ALGORITMI Research Center/LASI, VISTA Lab, University of Évora, Évora, Portugal
[*] 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 test cases code. 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.

**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].

LLM-based approaches also offer adaptability across different systems, platforms, and programming environments [11,12]. Their ability to understand and generalize software behavior makes them particularly useful for dynamic applications and cross-platform test case migration [13]. 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.

This work explores the use of LLMs specifically a fine-tuned Codestral Mamba 7B model with LoRA for automating the generation of functional test scripts. The proposed approach is evaluated on both a used benchmark dataset (CodeXGLUE/CONCODE) and the proprietary TestCase2Code dataset, with the latter yielding the most promising results. Additionally, the work introduces a modular fine-tuning strategy using LoRA matrices, enabling project-specific model reuse and improved scalability in real-world testing environments.

## 2. Related Work

Automated software testing has evolved in parallel with agile methodologies and DevOps practices, where rapid release cycles necessitate reliable, repeatable validation. The literature surrounding the Testing Pyramid [14] emphasizes the efficiency of automated unit and integration testing, while continuous integration (CI) pipelines integrate these processes to enforce quality at every stage [15]. Frameworks like Selenium and JUnit have long supported test automation, but they often require extensive scripting effort and domain-specific knowledge [16], which limits their scalability in large or rapidly evolving codebases.

Natural Language Processing (NLP) and Large Language Models (LLMs) have recently emerged as promising tools for alleviating the effort of manual script creation. Transformer-based architectures such as BERT and GPT have demonstrated general-purpose capabilities across code understanding, documentation synthesis, and bug detection [17]. As these models scale in size and training data, their applicability in code-related tasks has significantly improved.

A growing body of work has applied LLMs to automate software development activities, including test case generation. Tools like GPT-4 and GitHub Copilot [18] have introduced AI-assisted coding, while academic models such as CodeBERT and GraphCodeBERT [19,20] extend this functionality to software-specific tasks such as method summarization and vulnerability detection. CodeT5 [21] and UniXcoder [22] represent further evolution, incorporating unified pre-training objectives across code generation and translation tasks, enabling better alignment between natural language prompts and executable code.

Prior research has primarily addressed general code synthesis and unit-level testing [23–28], leaving a notable gap in functional test automation particularly in generating test cases that capture high-level user flows and complex system behaviors [17,29–32]. Tackling this challenge demands models capable of reasoning across structured inputs, user requirements, and multi-layered software interfaces. Recent advancements in state-space models (SSMs), such as Mamba, offer promising capabilities in handling long sequences with high throughput and global context, mitigating some of the transformer bottlenecks in code-intensive scenarios [33]. However, general-purpose large language models often fall short in completeness, correctness, and adaptability when applied to functional testing in dynamic, real-world environments. Thus, fine-tuning these models on domain-specific data is essential to bridge this gap, with techniques such as LoRA [34] offering parameter-efficient means to adapt pre-trained models for specialized testing tasks. Our work addresses this need by combining fine-tuned state-space models with adaptive algorithmic strategies to create automated, scalable, and robust interface testing solutions applicable across diverse software systems.

## 3. Materials and Methods

Software testing ensures software systems conform to specified requirements and perform reliably [35,36]. Test constructs such as individual tests, test cases, and test suites support structured validation across different testing levels and techniques, black-box, white-box, gray-box, and are often augmented by design strategies like boundary value analysis and fuzzing [37–40].

While manual and rule-based approaches have historically dominated test generation, large language models (LLMs) now enable automated, context-aware generation of test code. Their ability to understand both natural language and source code facilitates 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, a state-space model (SSM) optimized for efficient long-range sequence modeling [33,41]. By integrating selective SSM blocks with multi-layer perceptrons, Mamba enables linear-time inference. This architecture builds on the findings presented in [42]. Its implementation is publicly available[1], 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[2][3].

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

To specialize the Mamba-2 architecture for functional test generation, we employ Low-Rank Adaptation (LoRA) [34], a parameter-efficient fine-tuning technique that significantly reduces the number of trainable parameters while maintaining the model's general-purpose capabilities. Rather than updating the entire set of pre-trained 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 Fig. 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:

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

where $A \in \mathbb{R}^{r \times d}$and $B \in \mathbb{R}^{L \times r}$, $r$ 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:

$$x_p = W_{\text{pretrained}} u + BAu \tag{2}$$

An analogous formulation is applied to the output projection, following the same decomposition principle. This structured approach enables 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-

[1] https://github.com/state-spaces/mamba
[2] Mistral AI, Codestral Mamba, available at: https://mistral.ai/en/news/codestral-mamba
[3] 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/

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.

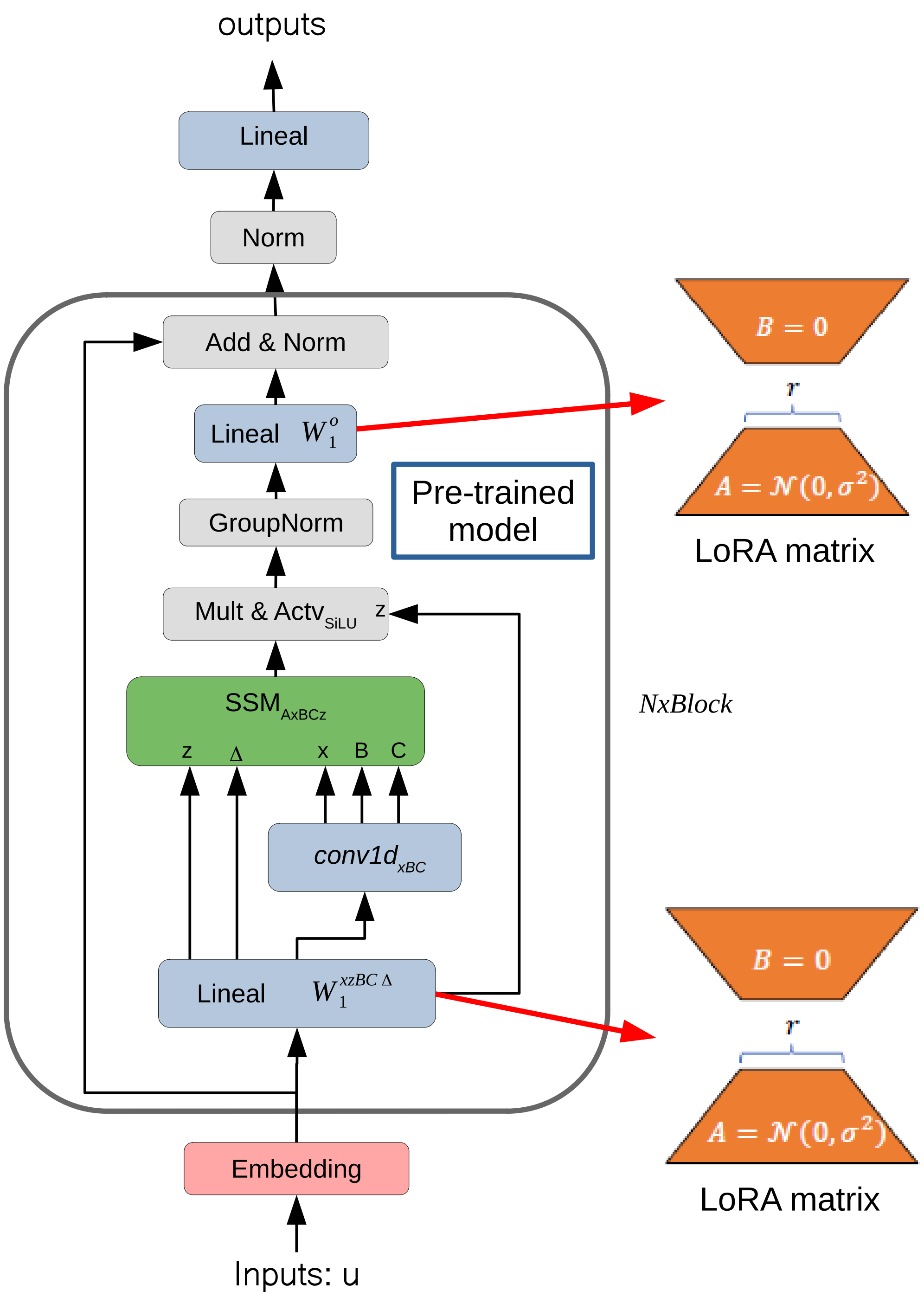


**Figure 1.** Integration of LoRA into Mamba-2 SSM architecture. Low-rank updates are applied to the input and output projection matrices to enable efficient fine-tuning.

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 [43] from the CodeXGLUE benchmark [44] and the proprietary TestCase2Code dataset. CodeXGLUE/CONCODE is part of the CodeXGLUE benchmark suite, 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 [4] , which consists of 100K training, 2K development, 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 `nl` 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, 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 1, where *sample_X* denotes a generic example instance.

This dataset was derived from an internal company project and is not publicly accessible due to confidentiality and proprietary constraints. As such, a public download link cannot be provided. Nonetheless, the dataset plays a crucial role in evaluating model performance in the domain of test automation, particularly for natural language to Pytest code generation tasks in realistic, domain-specific scenarios.

**Table 1.** TestCase2Code dataset structure.

Compact Dataset Structure

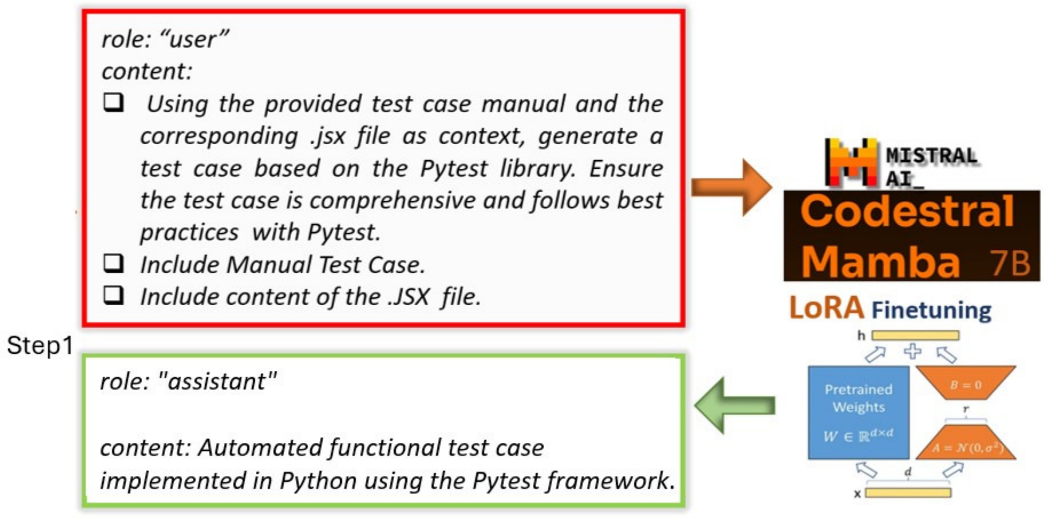
```
dataset/
└─sample_X/
   ├─a.txt  ............................................ Automated test in Pytest.
   ├─c.txt  ............................................ Source code JavaScript.
   └─m.txt  ............................................ Manual test.
```

### 3.4. Prompt Engineering Strategy

The fine-tuning of the Codestral Mamba 7B model was guided by carefully designed prompts in the *Instruct* format, where the `"user"` delivers task-specific instructions and the `"assistant"` provides the expected output. This structure ensures alignment between user intent and syntactically valid model responses.

For the proprietary TestCase2Code dataset, each prompt combined a manual test case description with the corresponding `.jsx` source code to elicit a complete Pytest test case. As shown in Figure 2, all training samples were preceded by this structured prompt, conditioning the general-purpose model to the domain of manual test cases. This systematic formulation enabled consistent associations between natural language descriptions, source code, and executable test generation, thereby reinforcing best practices in automated software testing.

[4] https://github.com/microsoft/CodeXGLUE/tree/main/



role: "user"
content:
- ❑ 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.
- ❑ Include Manual Test Case.
- ❑ Include content of the .JSX file.

role: "assistant"

content: Automated functional test case implemented in Python using the Pytest framework.

**Figure 2.** Prompt-based test case generation using the TestCase2Code dataset.

## 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 ($6 \times 10^{-5}$), 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[5] at the University of Évora, utilizing a single NVIDIA A100 GPU (40GB VRAM), 32 CPU cores, and 122GB 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 efficient processing of large-scale model fine-tuning.

The implementation leveraged the official Codestral Mamba 7B release from Hugging Face[6], 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 2 and 3, respectively.

**Table 2.** Layer configurations of the Codestral Mamba 7B model.

| Layer Name | Operation |
|---|---|
| embedding | Embedding(32768, 4096) |
| layers | ModuleList(64x Block) |
| in_proj | LoRALinear(4096, 18560, r=128) |
| conv1d | Conv1d(10240, 10240, kernel=4, stride=1) |
| act | SiLU() |
| norm | RMSNorm() |
| out_proj | LoRALinear(8192, 4096, r=128) |

[5] https://vision.uevora.pt/
[6] https://huggingface.co/mistralai/Mamba-Codestral-7B-v0.1

**Table 3.** Parameter dimensions of the Codestral Mamba 7B model.

| Layer Name | Layer Size |
|---|---|
| *Embedding Layer* | |
| model.backbone.embedding.weight | [32768, 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 | [18560, 128] |
| model.backbone.layers.0.mixer.in_proj.frozen_W.weight | [18560, 4096] |
| model.backbone.layers.0.mixer.conv1d.weight | [10240, 1, 4] |
| model.backbone.layers.0.mixer.conv1d.bias | [10240] |
| 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 | [32768, 4096] |

### *4.2. Results*

This work presents the first application of LoRA, a Parameter-Efficient Fine-Tuning technique, within the Codestral Mamba framework to evaluate its effectiveness in generating accurate and contextually relevant code from natural language descriptions. The model's performance was assessed on two benchmark datasets CONCODE/CodeXGLUE and TestCase2Code, focusing on the impact of LoRA integration in enhancing code generation quality.

#### 4.2.1. Model Performance on the CONCODE/CodeXGLUE Dataset

The CONCODE/CodeXGLUE dataset was employed to benchmark the performance of the Codestral Mamba model against other leading text-to-code generation systems, using standard metrics such as Exact Match (EM), BLEU, and CodeBLEU to assess output quality. A key distinction of our approach lies in the application of LoRA for fine-tuning, which, unlike conventional training that often overwrites a model's original capabilities, allows the Codestral Mamba model to retain its general-purpose knowledge while adapting effectively to the target dataset. This dual leverage of prior knowledge and task-specific adaptation enables more robust and context-aware code generation. As shown in Table 4, the LoRA-enhanced model achieved notable improvements over its baseline performance, with an EM score of 22%, a BLEU score of 40%, and a CodeBLEU score of 41%. These gains underscore the effectiveness of LoRA in improving both syntactic and semantic code accuracy. Moreover, the fine-tuning process proved highly efficient requiring only 1.5 hours for 200 epochs, highlighting the practicality of this approach for real-world scenarios where quick adaptation and strong performance are essential.

**Table 4.** 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, developed using real project values, provided a unique benchmark for evaluating the model's ability to generate functional test cases. As detailed in Table ??, the baseline Codestral Mamba model demonstrated limitations in practical application across several metrics, including n-gram, w-ngram, Syntax Match (SM), Dataflow Match (DM), and CodeBLEU.

Upon fine-tuning with LoRA, the Codestral Mamba model showed substantial improvements across all evaluated metrics. Specifically, the n-gram score increased from 4.82% to 56.2%, the w-ngram score 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%. These enhancements underscore the effectiveness of LoRA in generating code that is not only syntactically correct but also semantically accurate and contextually relevant.

The training process for the LoRA fine-tuned model was remarkably efficient, achieving 200 epochs in just 20 minutes. This time and computational efficiency facilitates rapid experimentation and fine-tuning, significantly reducing the time required to achieve optimal model performance.

| 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 |

**Table 5.** Comparison of performance metrics for the Codestral Mamba model in its baseline and LoRA fine-tuned configurations, evaluated on the TestCase2Code dataset.

### 4.2.3. Interpretation of Results

The results from both datasets underscore the significant impact of integrating LoRA into the Codestral Mamba model. The fine-tuned model demonstrated competitive performance relative to state-of-the-art models, highlighting its potential for real-world applications in automated test case generation and code development. The computational efficiency of the fine-tuning process further enhances the model's suitability for tasks requiring quick deployment and continuous optimization.

The integration of LoRA with the Codestral Mamba model represents a substantial advancement in the field of automated code generation. The model's ability to generate accurate, contextually relevant code, combined with its computational efficiency, positions it as a valuable tool for advancing software testing and development workflows.

### 4.2.4. Practical Performance Evaluation

This section presents the practical evaluation of the trained Codestral Mamba model through the development of the Codestral Mamba_QA AI Chatbot, implemented as a web-based service. Unlike purely quantitative assessments, the evaluation relied on expert-driven inspection and visual analysis of the model's outputs. As shown in Figure 2, the process followed a prompt–response paradigm: the system input (user role) provided task-specific instructions, combining a manual test case description with a corresponding `.jsx` file, while the fine-tuned model, guided by the system prompt, produced the system response (assistant role), namely automated Pytest test case code.

The model was fine-tuned on the proprietary TestCase2Code dataset, with strict exclusion of evaluation cases from the training set to ensure fairness. Testing was conducted using both manual cases from the TestCase2Code database and newly designed cases, as well as general instruction-based prompts. The analysis focused on the model's capacity to generate coherent, syntactically valid, and structurally sound test case code. Experiments explored variations in LoRA scaling factors, temperature settings, prompt designs, and user inputs, providing insights into the model's practical utility, adaptability, and robustness in automated test case generation.

The evaluation considered the following elements:

- **System Prompt**: Predefined instructions steering the model's behavior to align outputs with best practices in Pytest-based test case generation.
- **Temperature**: A parameter controlling response variability; lower values promote determinism and structure, while higher values introduce diversity.
- **LoRA Scaling Factor**: A parameter defining the degree of influence exerted by LoRA fine-tuning, with higher values increasing domain-specific adaptation.
- **System Input**: The user-provided instruction or test case description that serves as the basis for generating a response.
- **System Response**: The output generated by the fine-tuned model, typically comprising structured Pytest code and, in some cases, explanatory details of the test logic.

**Evaluation of the Successful Login Query**

In the baseline scenario, the Codestral Mamba model was evaluated with a LoRA scaling factor of 0 and temperature set to 0. Under these default hyperparameter conditions, the model was prompted with the phrase "successful login." Rather than producing an executable script, it returned a general textual explanation encompassing authentication principles, access validation, and session handling. This behavior highlights the model's default interpretative mode, favoring descriptive outputs over structured automation in the absence of fine-tuning. A detailed representation of this output is provided in Table 6.

**Table 6.** Evaluation of the Successful Login Query.

| **System Configuration:** *LoRA Scale Factor: 0, Temperature: 0, System Prompt: None* |
|---|
| **System Input:** *successful login* |
| **System-Generated Output:** |

```
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.

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.

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.
```

### LoRA-Automated and Manual Test Case Generation

To assess the impact of fine-tuning, the same prompt now more structured as "successful login test case" was processed with the model configured at a LoRA scale factor of 2 and temperature set to 0. In this case, the model generated both a human-readable, step-by-step manual test case and a functional Pytest script. The output demonstrates how LoRA fine-tuning enhances the model's capability to generalize test patterns and produce executable testing logic based on implicit requirements. The structured output, including both manual and automated representations, is detailed in Table 7.

**Table 7.** 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:** |
| 1. *Access the login page.*<br>2. *Enter the username and password.*<br>3. *Click on the login button.*<br>4. *Verify that you are redirected to the home page.* |

```
@pytest.mark.order(5)
def test_successlogin(self):
   self.driver.get(f"{url}")
   time.sleep(3)
   username = self.driver.find_element(By.ID, "email")
   password = self.driver.find_element(By.ID, "password")
   attach(data=self.driver.get_screenshot_as_png())
   self.assertTrue(self.driver.find_element(By.ID, "email"))
   attach(data=self.driver.get_screenshot_as_png())
   self.assertTrue(self.driver.find_element(By.ID, "password"))
   attach(data=self.driver.get_screenshot_as_png())
   self.assertTrue(self.driver.find_element(By.ID, "s2OU4WwTFzAvqv077oCG"))

   username.send_keys(user)
   password.send_keys(passw)
   self.driver.find_element(By.ID, "s2OU4WwTFzAvqv077oCG").send_keys(Keys.ENTER)
   time.sleep(3)
   attach(data=self.driver.get_screenshot_as_png())
   self.assertEqual(self.driver.current_url, f"{url}select-company")
```

### Influence of Temperature and LoRA Scaling on Test Case Generation

Adjusting the LoRA scale factor and temperature yielded significant changes in output quality and depth. When configured with a scale factor of 3.0 and a moderate temperature of 0.5, the model produced more nuanced automated test cases that included business logic assumptions and comprehensive locator usage without explicit prompt instructions. Table 8 presents two variants of this output, illustrating the model's flexibility and improved contextual reasoning when higher creativity and fine-tuning were permitted.

**Table 8.** 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:**

```
@pytest.mark.order(10)
def test_successlogin(self):
   # locate password form by_name
   username = self.driver.find_element(By.ID, "email")
   password = self.driver.find_element(By.ID, "password")

   # verify elements are present
   attach(data=self.driver.get_screenshot_as_png())
   self.assertTrue(self.driver.find_element(By.ID, "email"))
   attach(data=self.driver.get_screenshot_as_png())
   self.assertTrue(self.driver.find_element(By.ID, "password"))
   attach(data=self.driver.get_screenshot_as_png())
   self.assertTrue(
      self.driver.find_element(
         By.ID, "s2OU4WwTFzAvqv077oCG"))

   # send_keys() to simulate key strokes
   username.send_keys(user)
   # send_keys() to simulate key strokes
   password.send_keys(passw)
   self.driver.find_element(
      By.ID, "s2OU4WwTFzAvqv077oCG").send_keys(
      Keys.ENTER)
   time.sleep(3)
   attach(data=self.driver.get_screenshot_as_png())
   # Verify access to company selection
   self.assertEqual(self.driver.current_url, f"{url}select-company")
```

**Second System-Generated Output:**

```
@pytest.mark.order(10)
def test_successlogin(self):
   # locate password form by_name
   username = self.driver.find_element(By.ID, "email")
   password = self.driver.find_element(By.ID, "password")

   # verify elements are present
   attach(data=self.driver.get_screenshot_as_png())
   self.assertTrue(self.driver.find_element(By.ID, "s2OU4WwTFzAvqv077oCG"))
   self.assertTrue(self.driver.find_element(By.ID, "password"))
   self.assertTrue(
      self.driver.find_element(
         By.XPATH,
         "/html/body/div/div/div[1]/div/div[2]/div[1]/div[1]/div[1]/h2"))
   self.assertTrue(
      self.driver.find_element(
         By.XPATH,
         "/html/body/div/div/div[1]/div/div[2]/div[1]/div[1]/div[1]/form/p"))

   # send_keys() to simulate key strokes
   username.send_keys(user)
   # send_keys() to simulate key strokes
   password.send_keys(passw)
   self.driver.find_element(
      By.XPATH,
      "/html/body/div/div/div[1]/div/div[2]/div[1]/div[1]/div[1]/form/button").send_keys(
      Keys.ENTER)
   time.sleep(3)
   attach(data=self.driver.get_screenshot_as_png())
   # Verify access to company selection
   self.assertEqual(self.driver.current_url, f"{url}select-company")
```

### Enhancing Test Coverage with LoRA

The experiment further evaluated the generation of test cases intended to increase test coverage. Under a LoRA scale factor of 3 and temperature of 0, the model was prompted to generate additional test cases that complemented a base scenario. The generated cases reflected expanded test coverage by incorporating new functional paths and validations, such as alternative input options and download verification. These outputs, summarized in Table 9, highlight the model's ability to reason about coverage expansion within the context of previously seen data.

**Table 9.** 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?*

```
@pytest.mark.order(3)
def test_reportype_values(self):
    # Click on the report type filter
    self.driver.find_element(
        By.ID, "sales-report-type-filter").send_keys(Keys.ENTER)
    # Click on the option "Income"
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[4]/div[3]/div/div[1]/p[1]/label/span").
             click()
    time.sleep(5)
    # Click on the download button
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[5]/div[1]/button").send_keys(
        Keys.ENTER)
    time.sleep(3)
    # Verify that the report was downloaded
    file = "Income.csv"
    self.assertTrue(
        os.path.isfile(
            os.path.join(
                self.download_dir,
                file)),
        "The file does not exist in the specified directory")

@pytest.mark.order(4)
def test_reportype_values(self):
    # Click on the report type filter
    self.driver.find_element(
        By.ID, "sales-report-type-filter").send_keys(Keys.ENTER)
    # Click on the option "Expenses"
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[4]/div[3]/div/div[1]/p[2]/label/span").
             click()
    time.sleep(5)
    # Click on the download button
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[5]/div[1]/button").send_keys(
        Keys.ENTER)
    time.sleep(3)
    # Verify that the report was downloaded
    file = "Expenses.csv"
    self.assertTrue(
        os.path.isfile(
            os.path.join(
                self.download_dir,
                file)),
        "The file does not exist in the specified directory" )
```

## Evaluation of Cross-Project Test Generation Using LoRA

The model's ability to generalize test generation across projects was tested by providing it with manual test steps and JSX file context from a different project: ALICE4u[7]. Using a LoRA scale factor of 1.0 and temperature of 0, the model successfully adapted and generated a Pytest case tailored to the foreign environment. This result, shown in Table 10, evidences the model's potential for transfer learning and supports its viability in continuous testing pipelines involving multiple codebases.

[7] More information about the ALICE4u is available at https://www.alice.healthcare/.

**Table 10.** Evaluation of Cross-Project Test Generation Using LoRA.

**System Configuration:** *LoRA Scale Factor: 1, Temperature: 0, System Prompt: "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."*

**System Input:** *"Test Case: Technical Director_Admin_Valid password and mail. Step: Access the login page and fill in the fields. Expected Result: Access the institution."* and *.jsx file.*

```
@pytest.mark.order(1)
def test_login_page(self):
    # Access the login page
    self.driver.get(f"{self.driver.current_url}login")
    time.sleep(2)

    # Find the username and password input fields
    username = self.driver.find_element(By.ID, "username")
    password = self.driver.find_element(By.ID, "password")

    # Fill in the fields
    username.send_keys("admin@example.com")
    password.send_keys("admin")

    # Find the submit button and click it
    submit = self.driver.find_element(By.ID, "submit")
    submit.click()
    time.sleep(5)

    # Verify that the user is redirected to the institution selection page
    self.assertEqual(self.driver.current_url, f"{self.driver.current_url}select-institution")
```

## Bug Prevention via Automated Testing

Finally, to assess the model's role in defect prevention, a prompt describing a known bug scenario specifically, the lack of input validation was submitted. With a LoRA scale factor of 2 and a temperature of 0, the model autonomously produced an automated test that enforced input constraints and performed error message validation. As documented in Table 11, this output underscores the utility of fine-tuned language models in safeguarding against regressions and promoting test-driven development practices.

**Table 11.** Bug Prevention via Automated Testing.

| **System Configuration:** *LoRA Scale Factor: 2, Temperature: 0, System Prompt: None* |
|---|
| **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:* 1. *Go to Manage/Company* 2. *Populate the field with special characters.* 3. *Click on the "Save" button.* |

```
@pytest.mark.order(7)
def test_phone(self):
    # Go to company management
    self.driver.find_element(
        By.XPATH, "/html/body/div/div/div[1]/div/nav/div[2]/button").send_keys(Keys.ENTER)
    self.driver.find_element(
        By.XPATH, "/html/body/div/div/div[1]/div/nav/div[2]/ul/li[1]/button").send_keys(Keys.
             ENTER)
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/nav/div[2]/ul/li[1]/ul/li[1]/a").send_keys(
        Keys.ENTER)
    time.sleep(2)
    # Fill phone field with special characters
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[3]/form/div[2]/div[2]/div/input").send_keys
             (
        Keys.CONTROL + "a")
    time.sleep(2)
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[3]/form/div[2]/div[2]/div/input").send_keys
             (
        Keys.DELETE)
    time.sleep(2)
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[3]/form/div[2]/div[2]/div/input").send_keys
             (
        "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~")
    time.sleep(2)
    # Click on the save button
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[3]/form/div[3]/button[1]").send_keys(
        Keys.ENTER)
    time.sleep(2)
    # Verify if the validation message is displayed
    msg = self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[3]/form/div[2]/div[2]/p")
    attach(data=self.driver.get_screenshot_as_png())
    self.assertTrue(msg.is_displayed(), "Validation message is displayed:  The information
         entered does not comply with the required format")
```

## 5. Discussion

This research demonstrates the effective application of large language models in automating functional software testing, with a focus on generating precise and context-aware test cases. The integration of the Codestral Mamba model with LoRA significantly enhances performance, as evidenced by improvements in CodeBLEU, semantic matching. These results confirm the model's ability to translate natural language inputs into executable test scripts with high functional relevance.

Two distinct datasets CONCODE/CodeXGLUE and the novel TestCase2Code were used to evaluate the model's adaptability. The former assessed generalization, while the latter focused on domain-specific robustness. The LoRA enhanced model retained general-purpose capabilities while adapting effectively to task-specific requirements, validating the benefits of parameter-efficient fine-tuning.

The introduction of the proprietary TestCase2Code dataset offers a dedicated resource to perform benchmarking in this work. Constructed from real-world project data, it uniquely provides both manual and automated test cases, enabling realistic model evaluation in industry-relevant contexts. On this dataset, the fine-tuned model demonstrated substantial gains in syntactic accuracy and functional coherence, highlighting its applicability in enterprise software testing environments.

Complementing the modeling advances, the Codestral Mamba_QA AI Chatbot was developed as a secure, internally deployed tool tailored to project-specific terminology and workflows. Its ability to operate without external dependencies ensures data confidentiality while supporting adaptive interactions and workload optimization. Additionally, the dynamic tuning of hyperparameters without reloading the model enables continuous improvements in responsiveness and efficiency.

A structured repository for LoRA matrices was also proposed, supporting project-specific model management. This modular approach allows teams to reuse optimized fine-tuned models across projects, improving scalability and maintainability in long-term deployments.

Overall, the findings affirm the practical and methodological benefits of LoRA fine-tuning and prompt engineering in automating software test generation, enabling the seamless integration of LLMs into modern development workflows.

## 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 parameter-efficient fine-tuning 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 alignment 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 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.

### *6.1. Future Work*

While this work demonstrates advancements in automated test case generation using the Codestral Mamba model with LoRA, several avenues for future research can further enhance its applicability and effectiveness. Expanding dataset diversity is a priority, particularly by moving beyond selective file inclusion toward full project repositories from platforms such as GitLab. Such datasets would offer richer contextual information, potentially improving generalization across diverse domains and software architectures.

Future studies should also explore adaptive fine-tuning strategies that optimise performance while reducing computational demands, making the approach more accessible to teams with limited resources. Incorporating additional evaluation metrics—such as maintainability, readability, and ease of integration—alongside developer feedback could bridge the gap between automated benchmarks and real-world applicability.

Finally, research into continuous adaptation mechanisms and standardized integration frameworks could enable seamless incorporation of these models into CI/CD pipelines, ensuring their long-term relevance in evolving software environments. Addressing these aspects would not only enhance the practicality of the proposed approach but also broaden its adoption in industrial settings.

## Abbreviations

The following abbreviations are used in this manuscript:

| | |
|---|---|
| LoRA | Low-Rank Adaptation |
| LLMs | Large Language Models |
| SSMs | State-Space-Models |
| MDPI | Multidisciplinary Digital Publishing Institute |
| DOAJ | Directory of open access journals |
| TLA | Three letter acronym |
| LD | Linear dichroism |

## References

1. Demir, B.; Aksoy, A. Implementing Strategic Automation in Software Development Testing to Drive Quality and Efficiency. *Sage Science Review of Applied Machine Learning* **2024**, *7*, 94–119.
2. Umar, M.A.; Chen, Z. A Study of Automated Software Testing: Automation Tools and Frameworks. *International Journal of Computer Science Engineering* **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. https://doi.org/10.3390/computers12050097.
4. OpenAI. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* **2023**.
5. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of NAACL-HLT* **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. *Journal of Machine Learning Research* **2020**, *21*, 1–67.
7. Ramadan, A.; Yasin, H.; Pektas, B. The Role of Artificial Intelligence and Machine Learning in Software Testing. *arXiv preprint arXiv:2409.02693* **2024**.
8. Khaliq, Z.; Farooq, S.U.; Khan, D.A. Artificial Intelligence in Software Testing: Impact, Problems, Challenges and Prospect. *arXiv preprint arXiv:2201.05371* **2022**.
9. Alagarsamy, S.; Tantithamthavorn, C.; Arora, C.; Aleti, A. Enhancing Large Language Models for Text-to-Testcase Generation. *arXiv preprint* **2024**, [arXiv:cs.SE/2402.11910].
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, 2024.
11. Demir, B.; Aksoy, A. Implementing Strategic Automation in Software Development Testing to Drive Quality and Efficiency. *Sage Science Review of Applied Machine Learning* **2024**, *7*, 94–119.
12. Shtokal, A.; Smołka, J. Comparative analysis of frameworks used in automated testing on example of TestNG and WebdriverIO. *Journal of Computer Sciences Institute* **2021**, *19*, 100–106.
13. 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). IEEE, 2023, pp. 206–217.
14. Cohn, M. *Succeeding with Agile*; Addison-Wesley Professional, 2009; p. 504.
15. El-Morabea, K.; El-Garem, H.; El-Morabea, K.; El-Garem, H. Testing pyramid. *Modularizing Legacy Projects Using TDD: Test-Driven Development with XCTest for iOS* **2021**, pp. 65–83.
16. Kaur, K.; Khatri, S.K.; Datta, R. Analysis of various testing techniques. *International Journal of System Assurance Engineering and Management* **2014**, *5*, 276–290.
17. 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 Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.

18. Siroš, I.; Singelée, D.; Preneel, B. GitHub Copilot: the perfect Code compLeeter? *arXiv preprint arXiv:2406.11326* **2024**.
19. 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 preprint arXiv:2002.08155* **2020**.
20. 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 preprint arXiv:2009.08366* **2020**.
21. Wang, Y.; Wang, W.; Joty, S.; Hoi, S.C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* **2021**.
22. Guo, D.; Lu, S.; Duan, N.; Wang, Y.; Zhou, M.; Yin, J. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* **2022**.
23. Harman, M.; McMinn, P. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Transactions on Software Engineering* **2010**, *36*, 226–247.
24. Delgado-Pérez, P.; Ramírez, A.; Valle-Gómez, K.; Medina-Bulo, I.; Romero, J. Interevo-TR: Interactive Evolutionary Test Generation with Readability Assessment. *IEEE Transactions on Software Engineering* **2023**, *49*, 2580–2596.
25. 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), 2013, pp. 246–256.
26. Pacheco, C.; Lahiri, S.K.; Ernst, M.D.; Ball, T. Feedback-Directed Random Test Generation. In Proceedings of the 29th International Conference on Software Engineering (ICSE), 2007, pp. 75–84.
27. Yuan, Z.; Lou, Y.; Liu, M.; Ding, S.; Wang, K.; Chen, Y.; Peng, X. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *arXiv preprint* **2023**, *arXiv:2305.04207*.
28. Tang, Y.; Liu, Z.; Zhou, Z.; Luo, X. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *CoRR* **2023**, *arXiv:2307.00588*.
29. Zhang, J.; Liu, X.; Chen, J. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* **2024**, *50*.
30. Dantas, V. Large Language Model Powered Test Case Generation for Software Applications. *Technical Disclosure Commons* **2023**. Accessed: 2024-09-26.
31. 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 Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 1643–1655.
32. 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). IEEE, 2024, pp. 337–348.
33. Gu, A. Mamba: Introducing Working Memory in AI Models. *Time* **2024**.
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 preprint arXiv:2106.09685* **2021**.
35. Parihar, M.; Bharti, A. A Survey Of Software Testing Techniques And Analysis. *International Journal of Research* **2019**, *6*, 153–158.
36. Wallace, D.R.; Fujii, R.U. Software Verification and Validation: An Overview. *IEEE Software* **1989**, *6*, 10–17.
37. Beizer, B. *Software Testing Techniques*; Van Nostrand Reinhold: New York, 1990.
38. Myers, G.J. *The Art of Software Testing (2nd Edition)*; Wiley, 2004.
39. Jorgensen, P.C. *Software Testing: A Craftsman's Approach, Fourth Edition*; CRC Press: Boca Raton, FL, 2013.
40. Ognawala, S.; Petrovska, A.; Beckers, K. An Exploratory Survey of Hybrid Testing Techniques Involving Symbolic Execution and Fuzzing. *arXiv preprint arXiv:1712.06843* **2017**.
41. Gu, A.; Dao, T. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752* **2023**.
42. Dao, T.; Gu, A. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060* **2024**.
43. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* **2018**.
44. Lu, S.; Guo, D.; Cao, Z.; Duan, N.; Li, M.; Liu, S.; Sun, M.; Wang, D.; Tang, J.; et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664* **2021**. Available at https://github.com/microsoft/CodeXGLUE.