Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Inteligência Artificial e Ciência de Dados

Dissertação

# Enhancing Software Testing Automation through Large Language Models

Yanet Sáez Iznaga

Orientador(es) | Luís Rato
Javier Lamar León
Pedro Salgueiro

Évora 2025

**Universidade de Évora - Escola de Ciências e Tecnologia**

Mestrado em Inteligência Artificial e Ciência de Dados

Dissertação

# Enhancing Software Testing Automation through Large Language Models

Yanet Sáez Iznaga

Orientador(es) | Luís Rato
Javier Lamar León
Pedro Salgueiro

Évora 2025

A dissertação foi objeto de apreciação e discussão pública pelo seguinte júri nomeado pelo Diretor da Escola de Ciências e Tecnologia:

| | |
|---|---|
| Presidente | Teresa Gonçalves (Universidade de Évora) |
| Vogais | Pedro Salgueiro (Universidade de Évora) (Orientador)<br>Vitor Beires Nogueira (Universidade de Évora) (Arguente) |

Évora 2025

*To the curious minds that question the status quo and drive innovations that transform industries.*

# Acknowledgments

The journey of completing this thesis has been both challenging and rewarding, and I am deeply grateful for the support and encouragement I have received from various individuals who have contributed to its successful completion.

First and foremost, I would like to express my heartfelt gratitude to my tutor and husband, Dr. Javier Lamar, whose unwavering support, guidance, and expertise have been instrumental in shaping this thesis. His insights and dedication have been a constant source of motivation and inspiration.

I also extend my appreciation to my other tutors, Dr. Luís Rato and Dr. Pedro Salgueiro, for their time, their valuable suggestions, and for being part of this academic journey.

I extend my sincere thanks to my employer Dr. Artur Romão and colleagues at Decsis, who have provided me with the opportunity and resources to pursue this master's degree. Their support has been crucial in balancing my academic pursuits with professional responsibilities.

Lastly, I am profoundly grateful to my family for their unwavering belief in me. To my mother, Damiana, brothers, Juan and Alexander, my daughter, Damiley, my husband, Javier and to my father Pedro, who is no longer with us, but whose values and faith in me continue to inspire me. Thank you for your love, encouragement, and unwavering support throughout this journey. Your faith in me has been a driving force behind my perseverance and success. The principles lovingly given by my family have been fundamental in forging my path, with pride and gratitude. To all who have contributed to this dissertation in any way, I express my deepest appreciation.

To all who have contributed to this dissertation in any way, I express my deepest appreciation.

# Contents

# List of Figures

# List of Tables

# Acronyms

**GUI** Graphical User Interface

**AI** Artificial Intelligence

**ML** Machine Learning

**NLP** Natural Language Processing

**LLMs** Large Language Models

**SSMs** State Space Models

**MLP** Multi-Layer Perceptron

**SSD** State Space Duality

**LoRA** Low-Rank Adaptation

**GVA** Grouped-Value Attention

**EM** Exact Match

**SM** Syntax Match

**DM** Data Flow Match

**CB** CodeBLEU

**ASTs** Abstract Syntax Trees

**SUT** System Under Test

**TMAP** Test Management Approach

**LTI** Linear Time-Invariant

**SSSMs** Selective State Space Models

**CNNs** Convolutional Neural Networks

# Abstract

This dissertation explores the application of advanced language models for automated software testing, focusing on generating high quality, context aware test scripts. It leverages the *Codestral Mamba* model using *Low-Rank Adaptation* technique to enhance test case generation. The model was fine-tuned on both the TestCase2Code dataset and CONCODE/CodeXGLUE to evaluate its capability to produce syntactically and semantically accurate automated code testing cases from natural language descriptions. The findings highlight the model's robustness, improving test coverage, software quality, and developer productivity. This research addresses key challenges in software testing and underscores the potential of integrating advanced language models into modern testing workflows.

**Keywords:** Automated Software Testing, Large Language Models, Test Case Generation, Low-Rank Adaptation, Codestral Mamba Model.

# Sumário


## Melhorando a Automação de Testes de Software através de Modelo de Linguagem de Grande Escala

Esta dissertação explora a aplicação de Modelo de Linguagem de Grande Escala para testes automatizados de software, focando na geração de scripts de teste de alta qualidade e sensíveis ao contexto. Utiliza o modelo *Codestral Mamba*, utilizando a técnica *Low-Rank Adaptation* para melhorar a geração de casos de teste. O modelo foi ajustado tanto no conjunto de dados TestCase2Code como no CONCODE/CodeXGLUE para avaliar a sua capacidade de produzir casos de teste automatizados precisos do ponto de vista sintático e semântico a partir de descrições em linguagem natural. Os resultados destacam a robustez do modelo, melhorando a cobertura de testes, a qualidade do software e a produtividade dos desenvolvedores. Esta pesquisa aborda desafios cruciais nos testes de software e sublinha o potencial de integrar modelos de linguagem avançados em fluxos de trabalho de testes modernos.



**Palavras chave:** Testes de Software Automatizados, Modelos de Linguagem de Grande Escala, Geração de Casos de Teste, Adaptação de Baixa Ordem, Modelo Codestral Mamba.

# Chapter 1

# Introduction

Ensuring the dependability of software systems is a fundamental aspect of modern development practices. As applications become increasingly intricate, verifying their correctness and compliance with specifications is crucial. Software testing serves as a key mechanism for identifying defects and validating expected behavior before deployment.

Historically, verification processes relied heavily on manual execution, demanding substantial human intervention and expertise. Nevertheless, this approach is often inefficient, susceptible to oversight, and difficult to scale. The emergence of automated testing has addressed many of these challenges, offering a systematic way to execute assessments with greater precision and repeatability [Demir and Aksoy, 2024a]. By leveraging predefined test scripts, automated solutions streamline validation procedures and support comprehensive coverage across different software environments [Umar and Chen, 2019].

A variety of automation frameworks facilitate test execution, addressing different testing needs. While some are designed for low-level validation, such as unit testing, others focus on higher-level system interactions, including those involving Graphical User Interface (GUI) evaluation [Sewnet et al., 2023]. It is important to distinguish between frameworks tailored for unit testing and those intended for interface testing, as not all automation frameworks rely on GUI interaction. Additionally, automation tools often require technical proficiency and continuous maintenance to remain effective as software evolves [Demir and Aksoy, 2024a].

## 1.1 Background and Motivation

The quality and reliability of software are essential elements for the success of any development project. Software testing plays a crucial role in the software development lifecycle by validating that an application functions correctly and meets its specified requirements. Given the ever-growing complexity of modern software systems, testing has become a key step in ensuring the overall quality of an application before its release. Traditionally, manual testing has been employed to ensure proper functionality. However, manual testing is not only time consuming but also error prone, requiring extensive human effort and domain expertise to properly execute test cases across different scenarios.

In recent years, the demand for automation in software testing has increased considerably, driven by the need to enhance efficiency and reliability in the testing process [Demir and Aksoy, 2024a]. Automated testing has proven to be an effective approach for reducing the manual effort required to execute repetitive tests while ensuring consistency and repeatability across different environments and software versions [Umar and Chen, 2019]. Central to this process are test scripts sets of predefined instructions that specify the steps and expected outcomes of each test case. Various test automation frameworks facilitate the execution of these scripts, enabling developers and quality engineers to perform tests more efficiently [Eldrandaly et al., 2019]. While some frameworks are designed for unit testing, focusing on the verification of individual code components, others are tailored for interface testing, often relying on GUI interactions [Sewnet et al., 2023]. However, many automation frameworks require programming expertise and frequent maintenance to accommodate changes in application features, posing challenges for long-term scalability [Demir and Aksoy, 2024a].

With the advent of Artificial Intelligence (AI), particularly in areas such as Machine Learning (ML) and Natural Language Processing (NLP), new possibilities for test automation have arisen [Ramadan et al., 2024]. These AI-driven technologies offer the potential to automate testing processes with greater efficiency, minimizing human intervention. In this context, these models are emerging as a powerful tool for test script generation. Models, such as OpenAI's *GPT-4* [OpenAI, 2023] , *Google's BERT* [Devlin et al., 2019], and *T5* [Raffel et al., 2020], are capable of processing large amounts of textual data and generating human-like language output. This has opened the door to automating test script creation, making it possible to generate scripts dynamically based on contextual understanding of software applications [Khaliq et al., 2022].

Large Language Models (LLMs) are particularly promising for automated testing, as they enable test case generation without requiring explicit programming expertise. This accessibility lowers the barrier to entry, allowing a broader range of users to leverage automation in software testing [Fang et al., 2024; Alagarsamy et al., 2024]. By leveraging these models, it is possible to generate test scripts based on user inputs, specifications, or natural language descriptions of test cases [Demir and Aksoy, 2024b]. This approach can reduce the time required to create and update test scripts, even in highly dynamic environments where application features are regularly updated. Furthermore, they can adapt to diverse systems and platforms, making them versatile tools for generating test scripts that can handle the specific behaviors and constraints of different operating systems or devices [Shtokal and Smołka, 2021].

The use of LLMs in test script generation also holds potential for cross-platform and cross-application migration. For instance, test scripts generated for one application could be adapted for use in a different software environment, thus further enhancing efficiency and scalability in the software testing process [Yu et al., 2023]. By exploring how these models perform in these diverse and evolving scenarios, this research aims to assess their viability as a tool for automating the generation of test scripts, improving the overall effectiveness and quality of the software testing process.

In summary, as software systems grow increasingly complex and critical to everyday life, automating the test script generation process through language models can be a game changer. This work will explore the application of LLMs in this domain.

## 1.2 Problem Statement

Software testing is a critical aspect of ensuring the quality and reliability of applications. However, manual test script generation presents several significant challenges that impact both the efficiency and effectiveness of quality assurance processes. One major issue is the large volume of data involved. Modern applications, especially those in sectors like big data, healthcare, and e-commerce, handle vast amounts of information. Testing these systems requires generating and managing extensive datasets, simulating a wide range of user interactions, and validating performance under heavy loads. This process is time-consuming and labor-intensive, demanding advanced automation tools to manage the complexity and scale of the data.

Another pressing concern is test coverage. Ensuring that every part of the software, from its core functionality to edge cases, is thoroughly tested is vital. Functional test cases play a crucial role in this process, as they help validate whether the system behaves as expected under various conditions. Incomplete coverage leaves the risk of undetected bugs or system failures that could surface after deployment. Given the time and resource constraints often faced by testing teams, achieving comprehensive coverage is a constant challenge. While automated testing tools can mitigate this to some extent, they still require detailed planning and precise execution to ensure the system is thoroughly evaluated. Manual methods, in particular, can be inefficient and prone to human error, raising the likelihood that critical issues may be overlooked.

Time and cost constraints further intensify the challenges of manual testing. By its nature, manual testing is time-intensive and costly. It requires skilled personnel to write, execute, and update test cases, alongside the infrastructure and oversight necessary to ensure test accuracy and coverage. As software complexity increases and project timelines compress, the workload on both manual testers and automation specialists intensifies considerably. While test automation offers long-term cost savings, it requires significant upfront investment for initial setup and ongoing maintenance. Automation scripts must be frequently updated to align with changes in the application, requiring continual involvement from developers and testers to keep everything in sync.

Moreover, dynamic application environments present a significant challenge to traditional testing processes. With the rise of Agile and DevOps methodologies, software updates and releases are happening at an increasingly rapid pace. This requires frequent adjustments to test scripts to reflect evolving code, new functionalities, and configuration changes. In such a fast moving landscape, manual testing struggles to keep up, as it lacks the scalability and flexibility needed to adapt to constant changes. As software evolves continuously, testing processes must be adaptive and capable of responding to these changes swiftly, something manual testing is unable to achieve efficiently on a large scale.

A fundamental challenge in software testing is the predominant focus on unit test automation in both research and industry, while higher-level tests, such as functional or end-to-end tests, receive comparatively less attention. Although the test pyramid, which will be discussed in detail in Subsection 1.3.2, emphasizes that tests at these upper levels require greater effort, time, and maintenance, their automation is essential. Automating functional tests not only enhances software quality but also enables development teams to manage the fast-paced evolution of software more effectively, mitigating the limitations of manual testing, which is often labor-intensive and prone to errors.

## 1.3 Literature Review on Software Testing

In the development of high-quality software, testing plays a critical role in ensuring that applications meet both functional and non-functional requirements. Two key approaches underpin this effort: *manual testing and automated testing* [Khant et al., 2016]. Both methods contribute to the overall reliability and robustness of software but differ significantly in their execution and focus [Banik and Dandyala, 2019]. Manual testing involves human testers actively interacting with the application to identify potential issues. It is particularly useful in scenarios where subjective human judgment is necessary, such as assessing user interfaces or evaluating the overall user experience. However, manual testing can be time-consuming and prone to human error, making it less scalable for large or repetitive testing scenarios [Kaur et al., 2014].

On the other hand, automated testing leverages software tools to execute prescripted tests repeatedly and accurately, without human intervention. Automated tests excel in areas where large volumes of test cases need to be executed quickly and consistently, such as regression testing or performance testing. By automating routine or repetitive tasks, development teams can focus on more complex testing activities and reduce the risk of oversight. Nonetheless, automated testing requires an upfront investment in time and resources to develop the necessary scripts and infrastructure, making it more suitable for projects that require long-term testing stability and scalability [Nidhra and Dondeti, 2012].

A crucial dimension in both manual and automated testing is whether the tests are conducted with knowledge of the internal workings of the system, which brings us to the concepts of *black-box testing* and white-box testing. Black-box testing involves evaluating the system solely based on its inputs and expected outputs, without considering how the internal logic of the software operates. In this approach, testers focus on what the system is supposed to do rather than how it achieves that functionality. This makes black-box testing an ideal candidate for both manual and automated functional testing, as it aligns closely with user-oriented test scenarios, where the emphasis is on ensuring the software behaves as required by its specifications [Maspupah, 2023].

In contrast, *white-box testing* involves a deep understanding of the internal logic and structure of the software. This testing approach requires knowledge of the code and algorithms that drive the application, allowing testers to verify the accuracy and efficiency of the underlying implementation. White-box testing is typically automated, as it involves running specific tests on individual pieces of code (such as unit tests) to ensure they behave correctly under various conditions. By examining the internal workings of the software, white-box testing provides more granular insights into potential issues that may not be visible from a purely external perspective, such as hidden bugs or inefficiencies in the code [Kaur et al., 2014].

This balanced approach between manual and automated testing, as well as between black-box and white-box testing, ensures that software is tested from multiple perspectives, both internally and externally. It provides comprehensive coverage, enabling the detection of both functional errors that impact the user and deeper technical issues that could affect performance or stability. This premise lays the groundwork for exploring the different types of software tests in more detail, where certain tests, such as unit or regression tests, are more likely to be automated due to their repetitive nature, while others, like acceptance or exploratory tests, may require manual intervention to assess the softwares usability and alignment with business requirements. A visual representation of the classification of software testing approaches is provided in Figure 1.1, illustrating the relationship between manual and automated testing as well as the categorization of black-box and white-box testing.

### 1.3.1 Integration of Automated Testing Techniques in Practice

Software quality teams must apply a variety of tests throughout the software lifecycle to ensure a high standard of quality. In practice, these tests are often merged when planning automated testing, depending on the stage of development, to achieve the shared goal of delivering reliable software. This integration not only enhances efficiency but also broadens test coverage, ensuring that the software is tested comprehensively from multiple perspectives.

For example, automated performance tests commonly include fuzz testing and system test input generation, ensuring the system not only meets performance standards but is also resilient to unexpected inputs [Miller et al., 1990; Wang et al.,

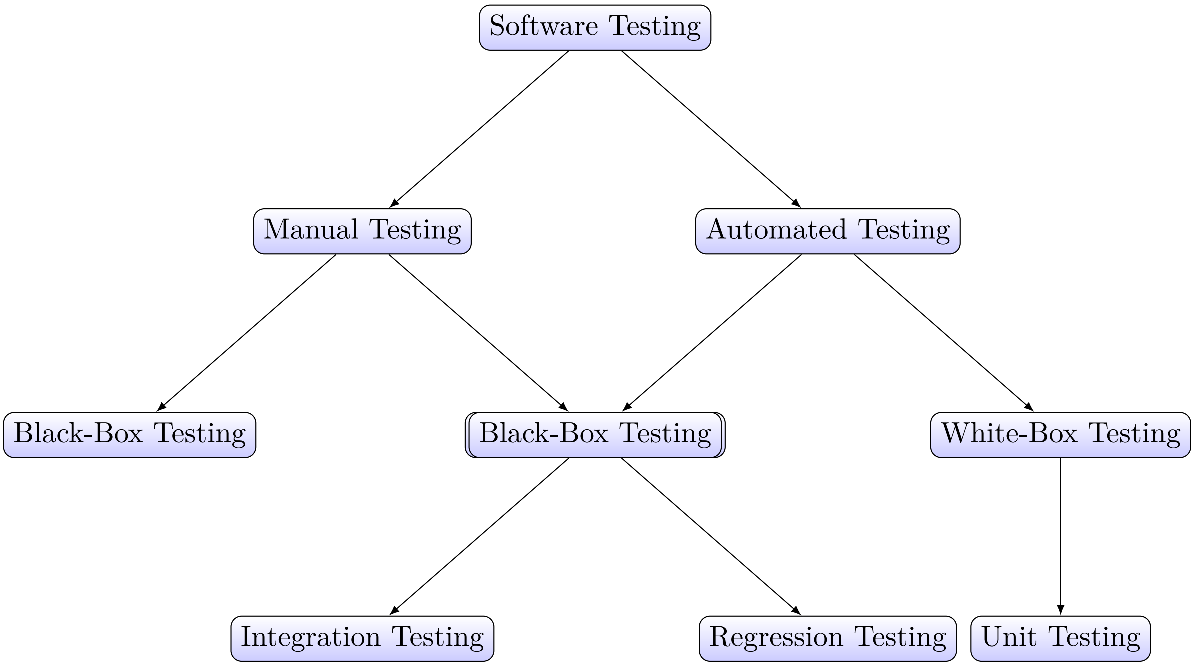



Figure 1.1: Classification of Software Testing Approaches

2017]. This combination allows teams to test both efficiency and robustness under varied conditions.

Similarly, functional, regression, and fuzz testing are frequently integrated. Functional tests verify the software against its requirements, while regression tests ensure new changes do not affect existing functionality. Incorporating fuzz testing enhances these by exposing edge cases or vulnerabilities, resulting in a more resilient system.

By merging these techniques, teams can automate more efficiently, reduce manual effort, and ensure thorough testing across various scenarios, increasing confidence in the softwares reliability and quality throughout its lifecycle.

### 1.3.2 Testing Pyramid and Test Automation

The Testing Pyramid (Figure 1.2), introduced by Mike Cohn [Cohn, 2009], is a framework that categorizes tests into three layers based on their scope and frequency: unit tests (at the base), integration tests or service tests (in the middle), and end-to-end or UI tests (at the top) [Contan et al., 2018; El-Morabea et al., 2021]. The pyramid illustrates that unit tests should be the most numerous, as they are fast, inexpensive, and require less effort. Moving up the pyramid, integration tests and end-to-end tests become fewer due to their higher cost, longer execution time, and increased complexity [Cohn, 2009].

While unit tests are heavily automated, there are fewer efforts to automate functional tests or end-to-end tests, which simulate real-world scenarios and user behavior. Automating these functional tests is critical for improving both software quality

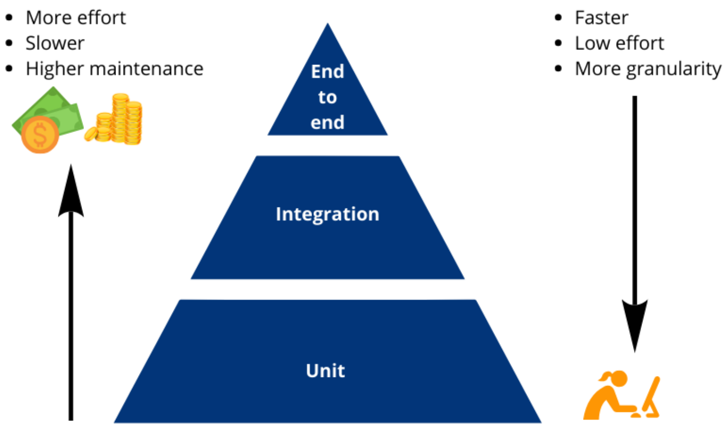



Figure 1.2: Testing Pyramid

and development speed. Automated functional tests help identify issues from the user's perspective, catching problems that unit or integration tests might miss, and they reduce the manual effort required for regression testing, speeding up the release process.

The importance of software testing has been widely acknowledged by both the academic and industrial sectors, solidifying its position as a cornerstone in software engineering research. This field continues to thrive, as evidenced by the significant number of conferences, workshops, and symposiums where software testing consistently ranks among the most submitted and published topics. Its prominence underscores the critical role testing plays in ensuring software reliability and quality.

Despite its advancements, several persistent challenges in software testing remain unresolved. One prominent issue is the automation of unit test case generation. Techniques such as search-based [Harman and McMinn, 2010; Delgado-Pérez et al., 2023], constraint-based [Xiao et al., 2013], and random-based approaches [Pacheco et al., 2007] have been developed to address this need. However, these methods often fail to deliver adequate test coverage or produce meaningful tests, as observed in recent studies [Yuan et al., 2023; Tang et al., 2023].

Similarly, in mobile GUI testing, traditional methodsincluding rule-based [Android Developers, 2012; Li et al., 2017], model-based [Su et al., 2017; Dong et al., 2020], and learning-based techniques [Pan et al., 2020]struggle to interpret the semantic information of GUI elements effectively, leading to incomplete test coverage and missed defects [Liu et al., 2023; Su et al., 2021].

To address these limitations, researchers are exploring innovative approaches to enhance the efficacy of testing. Among the most promising advancements are LLMs, which offer the potential to automate complex testing tasks and improve coverage by understanding both the code structure and user interaction semantics.

The integration of the language models into software testing has sparked significant advancements, yet various challenges persist in realizing their full potential. A comprehensive review by Zhang et al. (2024) highlights the diverse applications of LLMs in software testing tasks such as test case generation and program repair, but also emphasizes key challenges. One of the primary issues is the complexity of prompt engineering and ensuring that the generated tests are representative, raising concerns about the effectiveness of LLM-based approaches in diverse software testing environments [Zhang et al., 2024].

Similarly, the deployment of machine learning in testing workflows, as demonstrated by the ML Software Tester, brings the challenge of integrating accurate prediction models into existing pipelines, especially for complex projects like Flask. While machine learning offers improved test result predictions, its accuracy and seamless integration into current workflows remain significant hurdles [Doe, 2018].

Moreover, the landscape of automated mobile GUI testing has seen promising advancements through systems like *GPTDroid*, which leverages LLMs for human-like interactions in testing. However, it faces challenges such as low coverage and limited generalization capabilities due to training data constraints. Although it has improved bug detection and activity coverage, there is still room to expand its applicability and ensure broader testing [Liu et al., 2024].

In the realm of deep learning library fuzzing, *TITANFUZZ* illustrates how LLMs can be effectively applied to detect bugs in deep learning libraries like TensorFlow and PyTorch. While traditional fuzzing techniques struggle with the syntax and API constraints of deep learning systems, *TITANFUZZ* addresses these through LLM-driven program generation. Nonetheless, automating and generalizing these fuzzing techniques to other domains remains a challenge [Deng et al., 2023].

Another critical area where large language models are being explored is in cross-platform test script generation. They show potential in automating test scripts across various devices and platforms, but issues arise in ensuring compatibility and capturing accurate test behaviors across diverse environments. This remains a major challenge in realizing robust, cross-platform automation [Yu et al., 2023].

The literature on large language models in software testing highlights several challenges and innovative solutions to automate the generation of test cases, improve the precision of the testing, and improve the coverage of the code. A common issue across many of these studies is the difficulty in automating the generation of comprehensive and accurate test cases due to the complexity of natural language and the inherent limitations of existing tools.

For instance, [Dantas, 2023] discusses how current tools rely on manual or randomized user actions to build test cases, posing challenges in simulating real user behavior. Language models, however, can learn patterns of user interactions, enabling automatic test case generation during the software development lifecycle, thus offering a potential solution to this challenge. Similarly, [Watson et al., 2020] explore the generation of meaningful assert statements through the *Atlas* approach, a Neural Machine Translation (NMT)-based model designed to enhance the quality of assertions in automated testing. Their study demonstrates that Atlas can effectively predict assert statements that accurately assess the correctness of the focal method. Their work shows that incorporating machine learning models can enhance the generation of precise assertions, which are key to verifying the correctness of test methods.

On the FinTech side, [Xue et al., 2024] offers a breakthrough by automating acceptance testing of software against business rules, reducing human intervention and improving test coverage. It demonstrates the efficiency of fine-tuned large language models in generating comprehensive scenarios, reducing the time for testing from minutes to seconds, and outperforming state-of-the-art models like ChatGPT. This addresses a significant problem in financial software testing, where timely deployment and regulatory adherence are critical.

A recurring issue is the inability of LLM-based systems to produce valid or comprehensive test inputs, as pointed out by [Karmarkar et al., 2024]. This approach addresses two major problems in test data generation unsound and incomplete data. By employing an iterative refinement process, *TestRefineGen* ensures that the LLM-generated data is both valid and comprehensive, especially in confidentiality-restricted environments. Meanwhile, [Xia et al., 2024] take this further in the fuzzing domain with *Fuzz4All*, a universal fuzzer that targets multiple languages and system types. By leveraging language models, *Fuzz4All* generates more diverse and realistic inputs than traditional fuzzers, revealing previously unknown bugs across different programming environments.

Transfer learning techniques also play a significant role in enhancing the effectiveness of the language models in code-related tasks. Studies like T5 and GPT-3 [Mathur et al., 2023] propose using these models to automate test case generation for complex systems, reducing the reliance on manual input from developers. This aligns with the work by Xia [Zhang et al., 2023] on LLM-based fuzz driver generation, which, while promising, still faces challenges in filtering ineffective drivers and improving correctness for complex APIs.

The literature review reveals a significant imbalance in research focus, with considerably more attention devoted to automating unit tests compared to functional and system testing. While unit tests are vital for verifying individual components, functional tests are crucial for identifying issues from a user-centric perspective, often uncovering problems that unit or integration tests cannot detect. These tests play a pivotal role in ensuring that software behaves as intended in real-world scenarios. Automating functional tests not only reduces manual effort in regression testing but also accelerates development cycles and enhances overall software quality.

Despite their importance, functional and system tests remain underexplored due to the unique challenges they present. Simulating complex user workflows requires a deep understanding of dynamic interactions, which is far more intricate than generating unit tests. Ensuring comprehensive test coverage for functional scenarios is another significant hurdle, as it involves identifying and addressing untested paths and interactions. Furthermore, the scarcity of large, diverse datasets representing real-world functional testing scenarios limits the ability to train models effectively, reducing their generalizability across different application domains. The computational costs associated with generating and executing functional tests, particularly for dynamic user interfaces or large systems, further complicate their automation. These challenges have contributed to the limited focus on functional and system test automation in existing research, leaving a critical gap in the field.

*Codestral Mamba 7B* [Gu and Dao, 2023; Zuo et al., 2024] stands out as a superior choice over Llama and other smaller, free models. While Llama excels as a general-purpose language model, its broader scope comes at the cost of task-specific optimization, making it less effective for generating highly structured outputs like code. Codestral Mamba 7B, in contrast, has demonstrated significant improvements in contextual understanding and efficiency, thanks to architectural refinements and a modular mixture-of-experts framework that allows it to adapt to specialized tasks [Jiang et al., 2023; Thakkar and Manimaran, 2023]. Furthermore, Mistral outperforms Llama in key areas like reasoning, factuality, and computational efficiency, making it a more suitable foundation for code generation [Nadeau et al., 2024].

Building on the advancements of models like Mistral, Mamba 7B incorporates innovative features, such as linear-time sequence modeling with selective state spaces, which enhances its ability to handle structured, logical sequences like those found in code [Gu and Dao, 2023]. Additionally, hybrid approaches like Jamba demonstrate the potential of Mambas architecture to outperform traditional transformer-based models in specific use cases by leveraging its unique attention-free mechanism for faster and more accurate outputs [Lieber et al., 2024]. This makes Mamba 7B especially adept at translating textual prompts into precise, syntactically valid pytest code.

Moreover, Mamba's ability to efficiently utilize computational resources [Huang et al., 2024] makes it a practical choice for research projects requiring frequent experimentation without incurring high computational costs. Overall, Mamba 7B combines the domain-specific adaptability of Mistral with its own architectural innovations, making it a superior model for generating code from text in this thesis context.

While these models hold significant promise in automating various aspects of software testing, challenges remain in ensuring the completeness, correctness, and efficiency of the test cases they generate. Existing solutions often fall short in achieving comprehensive test coverage and adapting to dynamic, real-world environments. Many studies rely on controlled experimental settings, limiting their validation in more complex scenarios. This underscores the pressing need for testing frameworks that integrate real-world databases and dynamic environments to fully realize the potential of such models. *Our proposed work aims to fill this gap by developing model-driven automated interface testing solutions* that are practical, robust, and scalable. By refining the capabilities of these models and employing adaptive algorithmic techniques, we aspire to create testing solutions that are accurate, versatile, and applicable across diverse software domains.

## 1.4 Contributions

This dissertation makes significant contributions to the field of software testing, particularly in the automation of test case generation through the integration of LLMs. By leveraging the Codestral Mamba model with Low-Rank Adaptation (LoRA) fine-tuning, this research not only enhances the efficiency and reliability of software testing processes but also demonstrates the practical applicability of these advancements in real-world industrial settings. The following subsections detail both the methodological advancements and the practical impact of this study.

### 1.4.1 Advancements in Automated Software Testing

This research advances the state of automated software testing by introducing an innovative methodology for test case generation using LLMs. The key contributions in this regard are:

- **Improved Test Case Generation Efficiency:** The integration of the Codestral Mamba model with LoRA fine-tuning enables the automated generation of high-quality test cases, reducing reliance on manual scripting and enhancing test consistency across different software projects.
- **Enhanced Test Coverage and Defect Detection:** The model's ability to generate diverse and extensive test scenarios ensures broader coverage of software functionalities, facilitating the early detection of defects and contributing to overall software quality improvements.
- **Optimization of Software Quality Engineering Workflows:** By automating test case creation, software testing teams can focus more on strategic quality assurance activities, such as refining testing frameworks and analyzing results, leading to increased productivity and more streamlined development cycles.
- **Scalability and Adaptability to Real-World Applications:** The proposed approach is designed to be adaptable to different programming languages, testing frameworks, and continuous integration/continuous deployment (CI/CD) environments, underscoring its practical applicability across a wide range of software development contexts.
- **Foundation for a Modular Fine-Tuning Repository:** This research envisions a structured repository where LoRA matrices are encapsulated on a per-project basis. Such a repository would enable organizations to maintain fine-tuned models tailored to specific project requirements, facilitating reusability, efficient model updates, and sustained test relevance over time.

### 1.4.2 Acknowledgments and Practical Contributions

Beyond its theoretical advancements, this dissertation has been developed with a strong emphasis on practical validation and real-world applicability. The methodologies proposed were not only conceptualized in an academic setting but also tested and refined in collaboration with the software company ***Decsis***[1], which provided essential data for the creation of the TestCase2Code dataset.

The partnership with ***Decsis*** allowed for an empirical assessment of the proposed automated testing approach in an industrial environment, ensuring that the findings are aligned with real-world software engineering challenges. The data-driven validation process confirmed the feasibility of integrating LLM-based automated test generation into existing testing workflows, demonstrating tangible benefits such as increased efficiency, improved defect detection, and reduced manual effort.

Additionally, the insights gained from working with industry professionals helped refine the fine-tuning strategy and model adaptation process, ensuring that the approach remains practical, scalable, and adaptable to diverse software development needs. The collaboration underscores the importance of bridging the gap between academic research and industry practice, paving the way for future explorations into the role of LLMs in software quality engineering.

This dissertation, therefore, stands as a contribution not only to the academic discourse on automated software testing but also as a practical framework that can be leveraged by industry professionals seeking to enhance their testing processes through the use of advanced AI-driven methodologies.

## 1.5 Proposed Solution and Approach

To address the challenges outlined in the problem statement, we propose the integration of advanced language models into the software testing process. These models provide an innovative way to automate test script generation, reducing much of the effort and challenges tied to both manual and traditional automated testing. By leveraging their advanced natural language processing capabilities, such models can analyze application requirements, user stories, and existing documentation to generate comprehensive test cases, including functional test cases that validate core system behaviors and edge cases. This automation not only reduces the time and effort involved in creating test scripts but also enhances test coverage, minimizing the risk of undetected issues.

A critical advantage of employing these models is the ability to fine-tune them to meet the specific needs of an organization. By training a model on proprietary data, companies can ensure that it is attuned to their unique software ecosystem, capturing domain-specific knowledge and testing requirements. This tailored approach not

[1] https://www.decsis.eu/

only enhances the relevance and accuracy of generated tests but also reinforces data privacy. Organizations maintain ownership of the data used for training, which mitigates concerns surrounding sensitive information while aligning the models outputs with internal standards and practices.

Furthermore, fine-tuning allows companies to exert control over their models and their versions, fostering stability in quality and performance. By developing a solution specifically designed to address the testing demands of their applications, organizations can achieve a reliable and consistent testing framework. This control is essential in dynamic environments, where rapid software updates require frequent adjustments to testing strategies. With these models capable of adapting to code changes and evolving requirements, organizations can significantly enhance their agility and responsiveness to new features and configurations.

In addition to generating test cases, advanced language models can assist in the ongoing maintenance of test scripts. As software applications undergo continuous development, these models can analyze changes in code and automatically update the associated tests. This capability not only reduces the manual workload on testing teams but also ensures that tests remain relevant and effective in validating software functionality. By incorporating such models into the testing pipeline, organizations can streamline their quality assurance processes, ultimately improving software reliability while reducing reliance on labor-intensive manual methods.

By embracing advanced language models as a cornerstone of their testing strategy, organizations can foster a more efficient, effective, and adaptive quality assurance environment. This approach not only addresses the current challenges of manual testing but also positions organizations to thrive in the rapidly evolving landscape of software development, ensuring the delivery of high-quality applications in a timely and cost-effective manner.

## 1.6 Structure of the dissertation

This work is structured to provide a comprehensive examination of automated software testing through the application of LLMs. It begins with Chapter 1, the Introduction, which sets the stage by presenting the background and motivation for the research, highlighting the crucial role of effective software testing in contemporary software development. This chapter also reviews comprehensive software testing approaches and strategies, detailing various testing techniques and discussing the integration of automated testing methods in practice, alongside an exploration of the testing pyramid and its relationship with test automation. The problem statement identifies key challenges in the field, while the proposed solution and approach offer insights into the research's innovative aspects. Finally, a literature review situates the study within existing research, identifying gaps that this dissertation aims to address.

Chapter 2, Theoretical Foundations, delves deeper into the core concepts of software testing, starting with an analysis of test automation. This chapter also introduces large language models, elaborating on their capabilities and relevance to automated testing.

In Chapter 3, the Methodology section outlines the research design focused on test automation through fine-tuning advanced language models. It describes the model architecture and provides an overview of the pre-training process, followed by a detailed explanation of the fine-tuning techniques employed to adapt these models for generating test scripts.

Chapter 4 presents the Experimental Procedure and Results, detailing the steps taken for data collection, model development, and the training process, including hyperparameters used in the experiments. It further discusses the generation of test scripts and evaluates the model's performance, providing a clear picture of the outcomes achieved.

Finally, Chapter 5 encompasses the Discussion and Conclusions. This chapter interprets the results, discusses their practical implications, and reflects on the limitations of the study. It concludes by summarizing key findings and proposing directions for future research, reinforcing the significance of the work in advancing automated software testing practices. Through this structured approach, the work aims to contribute meaningfully to both academic and practical fields in software testing.

# Chapter 2

# Theoretical Foundations

Software testing is a critical and structured process that ensures the functionality, reliability, and overall quality of software products. This chapter delves into the theoretical foundations and definitions necessary for understanding various testing levels and techniques, with a particular focus on generating automated test cases using LLMs. By exploring these foundational concepts, this chapter aims to provide a comprehensive scope and context for the methods selected, emphasizing their role in enhancing software quality assurance. The integration of LLMs in this process represents a significant advancement, offering the potential to streamline test case generation and improve the efficiency and adaptability of automated testing strategies.

## 2.1 Software Testing

Software testing is a vital structured process that focuses on verifying and validating software products to ensure their functionality, reliability, and overall quality [Parihar and Bharti, 2019]. Beyond its foundational purpose, this dissertation focuses on specific testing levels and techniques that will form the basis for generating automated test cases with the aid of LLMs. The theoretical foundations and definitions necessary for understanding these approaches will be discussed in detail throughout this chapter, providing the scope and context for the methods selected.

In practice, software testing involves a disciplined assessment of an applications behavior and performance against specified requirements. This is achieved through a combination of testing levels such as functional, integration, and system testing, and techniques like regression and fuzz testing. These are often integrated to enhance test coverage and efficiency, ensuring that the software is comprehensively validated at every stage of development [Jamil et al., 2016].

The integration of automated testing techniques plays a central role in this discussion. Drawing on practical experience, the selection of levels and methods focuses on strategies that improve test automation efficiency, broaden coverage, and address real-world challenges in software quality assurance. Techniques such as functional testing verify compliance with requirements, regression testing ensures that changes do not impact existing functionality, and fuzz testing exposes vulnerabilities by testing system resilience under unexpected inputs. By systematically combining these methods, testing becomes not only more effective but also more adaptable to diverse scenarios [AbuSalim et al., 2021].

This dissertation builds upon established principles of software testing to investigate the potential of LLMs in automating test case generation, specifically targeting a range of test types, including functional, fuzz, exploratory, smoke, system, regression, as well as black-box and gray-box testing. By grounding this exploration in well-defined levels and techniques, it aims to demonstrate how advanced AI tools can enhance software quality assurance while addressing the challenges posed by increasingly complex software systems.

### 2.1.1 Definition and Role of Tests, Test Cases, and Test Suites

To fully understand the integration of automated testing techniques and the generation of test cases using large language models, it is essential to define and clarify the roles of tests, test cases, and test suites in software testing. These components form the foundation of a structured testing process and directly influence the quality and coverage of the validation efforts.

#### Tests

A test is a fundamental activity within the software testing process aimed at evaluating a specific aspect of a software application. Tests verify that the system behaves as expected under predefined conditions, ensuring compliance with requirements and specifications. They may focus on functionality, performance, security, or any other key attribute of the software.

#### Test Cases

A test case is a detailed, specific set of instructions and conditions designed to verify a particular functionality or feature of the software. It includes:

- **Input**: Data or actions provided to the system under test.
- **Execution Conditions**: The environmental setup or system state in which the test is performed.
- **Expected Results**: The anticipated outcome or behavior of the system based on the inputs provided.

Test cases serve as the building blocks of software testing. They ensure repeatability and consistency, making it possible to validate a system's behavior systematically. The effectiveness of test cases directly impacts the ability to detect defects and ensure the system meets its objectives [Beizer, 1990a].

#### Test Suites

A test suite is a collection of related test cases grouped together to evaluate a specific aspect of the software or to ensure comprehensive testing of a module, feature, or system. Test suites often target a particular level of testing, such as functional, integration, or system testing, and may be organized by features, components, or scenarios.

In automated testing, test suites are critical for executing tests efficiently, as they allow for the grouping and parallel execution of related test cases. This increases coverage while minimizing redundancy [Myers, 2004].

#### Role in Automated Testing and LLM-Generated Test Cases

In the context of this dissertation, tests, test cases, and test suites play pivotal roles in the automation process:

- **Tests**: Establish the objectives for validation, guiding the generation and execution of automated scenarios [Jorgensen, 2013].
- **Test Cases**: Provide the granular detail necessary for language models to generate meaningful and relevant automated scenarios, ensuring alignment with the intended requirements [Beizer, 1990b].
- **Test Suites**: Enable the structured organization and execution of LLM-generated test cases, facilitating integration with existing software development workflows [Crispin and Gregory, 2009].

By leveraging LLMs, the process of defining and generating test cases can be significantly streamlined, leading to enhanced test coverage and adaptability. The precise definition and integration of these components form the backbone of the automated testing strategies discussed in this dissertation.

### 2.1.2 Definition and Classification of Different Types of Software Testing

In software engineering, ensuring that a software product is both technically sound and meets end-user expectations is essential. To achieve this, structured testing processes such as Verification and Validation (V&V), diverse Testing Levels, a range of

Testing Techniques, and precise Test Case Design contribute to a robust framework for software quality assurance. Each element plays a specific role in identifying and resolving potential issues, ensuring the software is ready for production.

#### Verification and Validation (V&V)

Verification and Validation, commonly referred to as V&V, are complementary processes designed to confirm that a software product meets its requirements and is suitable for its intended purpose.

- **Verification**: Is an internal process focused on confirming that the software development process and its products are built correctly, in accordance with defined requirements and design specifications. Verification essentially answers the question: *Are we building the product correctly?* This process includes various checks at each stage of development to reduce discrepancies from initial specifications and helps identify errors early on [Wallace and Fujii, 1989].

- **Validation**: Is concerned with ensuring that the final software product fulfills user needs and intended use cases. This process addresses the question: *Are we building the right product?* Validation often occurs toward the end of the development cycle and includes user feedback and real-world testing to assess whether the software aligns with end-user expectations. Together, Verification and Validation serve as foundational pillars, ensuring the accuracy, relevance, and value of the final software product [Wallace and Fujii, 1989].

#### Testing Levels

To ensure comprehensive coverage, software testing is divided into various levels, each with a specific focus. These levels progressively assess the software, starting from individual components and culminating in system-wide evaluations.

- **Unit Testing**: The most granular level of testing, unit testing involves testing individual components or functions independently. Its purpose is to verify that each part of the software performs as expected in isolation. Unit tests are typically conducted early in the development cycle, enabling developers to catch errors at their source and facilitating smoother integration of components [Jorgensen, 2013].

- **Integration Testing**: This level of testing combines individual units and tests their interactions to detect issues that may arise when modules are integrated. Integration testing aims to ensure that components work together seamlessly and that any defects related to module interconnections are identified [Jorgensen, 2013].

- **System Testing**: System testing evaluates the fully integrated software application, checking its performance against both functional and non-functional requirements. This level simulates real-world scenarios to assess the software's behavior as a complete system, ensuring that all components function as a cohesive whole [Jorgensen, 2013].

- **Acceptance Testing**: The final level, acceptance testing, involves stakeholders and end-users who verify that the software meets business requirements and user expectations. Acceptance testing confirms that the software is ready for deployment, ensuring that it is suitable for release [Jorgensen, 2013].

### Testing Techniques

A range of Testing Techniques provides varied perspectives on software behavior, addressing both functional and structural attributes. Three primary techniques include:

- **Black-Box Testing**: This technique focuses on the softwares functionality without reference to its internal code. By examining inputs and outputs, black-box testing simulates end-user interactions and validates that the software performs as expected across a range of scenarios [Myers et al., 2011].

- **White-Box Testing**: White-box testing analyzes the softwares internal structure, focusing on code paths, logic, and conditions. It is especially effective for identifying hidden errors within the code and optimizing performance. White-box testing ensures comprehensive code coverage by allowing testers to examine the softwares inner workings [Myers et al., 2011].

- **Gray-Box Testing**: Combining elements of black-box and white-box testing, gray-box testing allows partial visibility of the internal code while maintaining a user-centered perspective. This approach is particularly useful in integration and system testing, as it provides insights into the softwares internal and external behavior [Myers et al., 2011].

### Functional and Non-Functional Testing

Comprehensive software testing includes both Functional and Non-Functional Testing, which assess the softwares operation and its overall qualities.

- **Functional Testing**: Ensures that each feature operates as specified by user requirements. It encompasses various forms, including:
  - **Smoke Testing**: A preliminary check to ensure that the critical functionalities of the software are working correctly.

  - **Regression Testing**: Ensures that updates or changes do not affect existing features.

- **Non-Functional Testing**: Evaluates attributes that impact user experience and system reliability. It includes:

  - **Performance Testing**: Assesses how the software operates under load conditions.
  - **Usability Testing**: Examines the ease of use and user experience.
  - **Security Testing**: Identifies potential vulnerabilities within the software.
  - **Compatibility Testing**: Ensures that the software functions across different devices and platforms.
  - **Reliability Testing**: Verifies the stability of the software over extended periods.

### Specialized Testing Techniques

Beyond conventional methods, Specialized Testing Techniques such as fuzzing and exploratory testing provide additional layers of validation for complex or unique requirements.

- **Fuzzing**: Is a security-focused technique that subjects the software to random or malformed inputs to identify vulnerabilities, particularly valuable for exposing security flaws [Ognawala et al., 2017].

- **Exploratory Testing**: Is an unscripted, intuitive method allowing testers to investigate the software freely, often uncovering bugs that scripted tests might overlook. This technique is particularly useful for discovering complex, context-dependent issues.

### Test Case Design

Effective Test Case Design is crucial for a successful testing strategy. Several design techniques maximize test coverage while minimizing redundancy:

- **Equivalence Partitioning**: This technique organizes input data into groups expected to produce similar outcomes, allowing testers to represent multiple scenarios efficiently and reduce the number of individual test cases.

- **Boundary Value Analysis**: This technique emphasizes testing edge cases at the boundaries of input ranges, which are often sources of defects.

- **Decision Table Testing**: Useful for software with complex decision-making, this method organizes conditions and actions into a table to ensure all possible input combinations are evaluated. Decision table testing is highly effective for verifying complex business logic.

By integrating Verification and Validation, structured Testing Levels, a diverse range of Testing Techniques, targeted Functional and Non-Functional Testing, and specialized approaches like fuzzing and exploratory testing, a comprehensive software testing framework emerges. This layered approach to software quality assurance ensures compliance with technical specifications, security, user satisfaction, and operational reliability. This comprehensive framework supports the development of high-quality software, aligning with both business objectives and user needs.

### 2.1.3 Importance and Role of Testing Throughout the Software Development Process

Testing plays a vital role in the software development process, ensuring that the final product is both technically sound and user-centered. Throughout each phase of development, testing serves as a quality control measure, enabling teams to identify and address issues before they escalate into costly or complex problems. Early in development, testing activities focus on **Verification and Validation (V&V)**, where verification ensures that each component aligns with design specifications, and validation confirms that the product will meet user needs and expectations. By detecting discrepancies at this stage, verification minimizes costly late-stage errors, while validation verifies that the evolving software product remains relevant to end-users. As the development progresses, **Testing Levels** like unit testing, integration testing, system testing, and acceptance testing offer structured checkpoints, each with a unique focus. Unit testing, for instance, verifies that individual functions or components perform as expected, while integration testing assesses how well these units work together, uncovering issues that may arise from interdependencies. Later, system testing evaluates the entire application as a whole, ensuring that it meets all functional and non-functional requirements, while acceptance testing allows stakeholders to determine if the product is ready for deployment.

In addition to testing levels, a range of **Testing Techniques** provides various perspectives for evaluating software. Techniques such as **black-box**, **white-box**, and **gray-box testing** cover different facets of software functionality. Black-box testing evaluates the software's outputs based on inputs, closely mimicking user interactions to ensure correct functionality, while white-box testing provides an internal view of the code, allowing for optimization and logical accuracy. Gray-box testing combines these two approaches, enabling a balanced assessment that is both user-centered and code-aware. Both **Functional** and **Non-Functional Testing** are essential to create a comprehensive assessment of the software. Functional testing verifies that each feature operates as required, while non-functional testing assesses qualities like performance, security, and usability, which impact user experience and reliability. Specialized techniques such as **fuzzing** and **exploratory testing** also enhance quality by addressing specific areas such as security and discovering complex, context-dependent bugs that scripted testing might miss.

**Test Case Design** is crucial for ensuring that testing is both effective and efficient, covering diverse scenarios with minimal redundancy. Techniques like **Equivalence Partitioning** reduce the number of test cases by grouping inputs expected to yield similar results, while **Boundary Value Analysis** focuses on edge cases where errors are more likely. **Decision Table Testing** helps ensure that software with complex decision-making logic behaves correctly under various conditions. Overall, the testing process is indispensable for achieving a high-quality, reliable product that meets both technical standards and user expectations. From early-stage verification to final acceptance testing, comprehensive testing supports each layer of development, reducing risks, improving quality, and fostering user satisfaction. By thoroughly evaluating software at every stage, testing ensures that the final product is robust, secure, and ready for deployment, building user trust and laying the foundation for successful software release.

### 2.1.4 Test Automation

Test automation is an essential aspect of modern software development, as it reduces testing time, effort, and costs while maintaining high standards of quality and reliability. Notably, software testing can account for up to 50–70% of a project's total cost, making automation a crucial strategy for cost reduction [Sullca, 2023]. This is particularly true in complex or critical systems, where exhaustive manual testing is impractical. Automation streamlines the testing process and ensures thorough testing across various system components.

Test automation relies on a few foundational concepts to guide effective testing and improve the chances of finding defects.

#### Test Cases and Components

A test case is a core element in automated testing, consisting of:

- Initial conditions: Setup requirements to ensure a reproducible environment.
- Test data: Input values that drive test parameters.
- Operations and Oracles: Actions performed on the system under test System Under Test (SUT) and an oracle, which verifies the results against expected outcomes.

#### Test Data Generation Techniques

Effective test data generation is critical to ensure coverage and detect potential defects. Common methods include:

- Input space partitioning: Dividing input data into partitions based on expected behaviors and selecting representative values.
- Boundary value analysis: Testing values at the boundaries of input partitions where errors are more likely.
- Error guessing: Using intuition and experience to choose inputs that are likely to reveal faults.

Various strategies are available to manage test data combinations:

- All combinations: Creating test cases for all possible input combinations.
- Pairwise testing: Ensuring that each pair of input values is tested at least once, providing an efficient trade-off between coverage and test count [Go et al., 2016].
- N-wise testing: Extending pairwise testing to cover combinations of $n$ parameters, which increases coverage without exhaustive testing.

### Coverage Criteria and Mutation Testing

To assess how well the SUT is tested, coverage criteria serve as essential metrics:

- Statement coverage: Ensures all code lines are executed.
- Decision coverage: Ensures all decision points (e.g., if-else statements) have been evaluated in true and false scenarios.
- Condition coverage: Focuses on each condition within decision statements.
- Modified Condition/Decision Coverage (MC/DC) further enhances testing by ensuring that each condition in a decision statement can independently affect the decision outcome, which is particularly valuable in complex systems.
- Mutation Testing is another technique to evaluate the effectiveness of a test suite by introducing small changes (mutations) into the code and checking if the test cases detect these deliberate faults. A high mutation score implies strong defect-detection capability, increasing confidence in software quality.

### Testing Levels and the V-Model

The V-Model is a software development model that illustrates the relationship between each phase of development and its corresponding testing level. It is structured in a V shape, where the left side represents stages of development, and the right side represents corresponding stages of testing. Each testing phase verifies or validates the outcome of its counterpart development phase, ensuring a systematic approach to quality assurance [Pressman, 2005]. Figure 2.1 shows the V-Model, which includes the most common testing levels.

The V-Model is particularly relevant in this discussion due to its structured approach, which clearly defines the direct correlation between development and testing phases. This characteristic is especially beneficial for projects requiring rigorous verification and validation, such as those in the health and finance sectors, where reliability, compliance, and risk mitigation are critical. While other models, such as Agile or Spiral, offer flexibility and iterative refinement, they may not always provide the same level of traceability between development and testing stages. Agile methodologies, for instance, prioritize continuous feedback and adaptability, which can be advantageous for rapidly evolving requirements but may introduce challenges in maintaining structured verification processes. Similarly, the Spiral model emphasizes iterative risk analysis, making it suitable for complex and high-risk projects but potentially less efficient in cases where requirements are well-defined from the outset. Given these considerations, the V-Model is chosen here for its clarity, systematic integration of testing activities, and suitability for projects where stability, regulatory compliance, and risk management are paramount, as is often the case in health and finance software development. In the context of test automation, the V-Model supports applying automation at each testing level:

- **Unit Testing**: At the base of the V-Model, unit testing is conducted to test individual components or modules in isolation. This phase focuses on validating the smallest parts of the application, often directly after they are implemented. Automated unit tests are typically written by developers to verify that each unit functions as expected before integration [Beizer, 1990a].

- **Integration Testing**: Moving up the V-Model, integration testing verifies the interactions between integrated components or units. Here, test automation ensures that different parts of the system work together as intended. Automated integration tests can simulate real-world interactions between components, identifying issues that may arise when units are combined [Beizer, 1990a].

- **System Testing**: Near the top of the V-Model, system testing evaluates the complete and integrated system to verify that it meets both functional and non-functional requirements. Automation at this stage focuses on end-to-end testing of workflows and features to ensure that the entire system functions correctly under realistic conditions [Beizer, 1990a].

- **User Acceptance Testing (UAT)**: At the highest level of the V-Model, UAT is conducted to ensure that the system meets the end-users expectations and business requirements. Automation in UAT can streamline testing repetitive processes and validate usability and functionality from an end-user perspective, though manual testing is often used here as well to capture subjective insights [Pressman, 2005].

- **Release Testing**: This final phase, often considered an extension of UAT, tests the system in a production-like environment to ensure it performs well

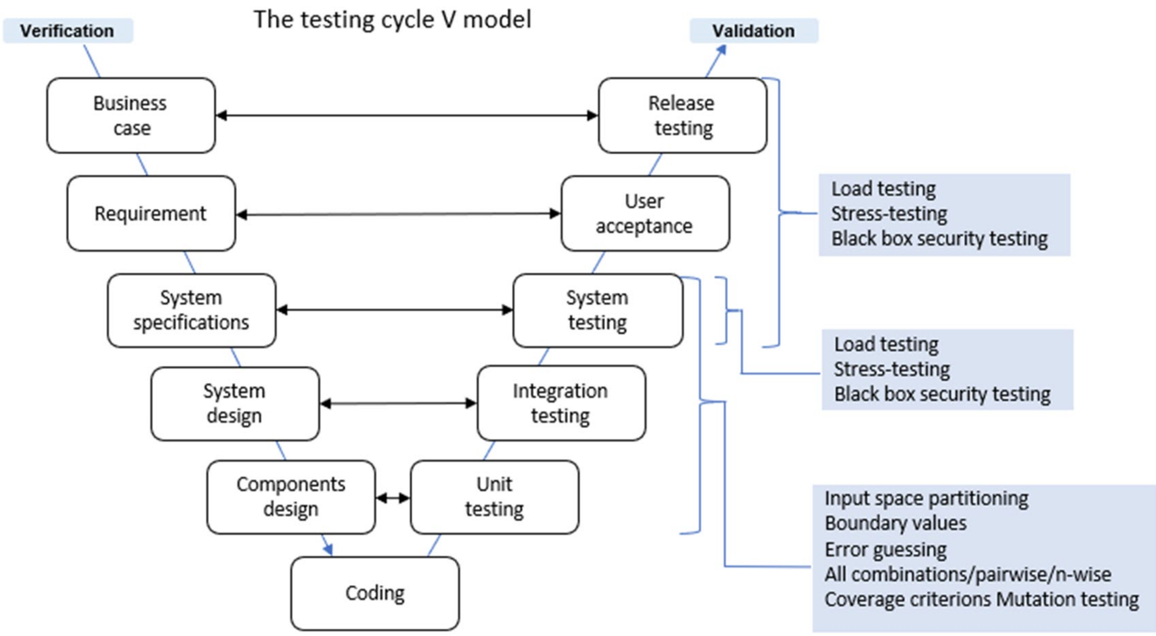



Figure 2.1: The V model and testing techniques.

under real-world conditions. Automated release testing can focus on aspects such as performance, security, and scalability, verifying that the system is robust and ready for deployment [Beizer, 1990a].

The V-Model promotes a proactive approach to testing by aligning each development stage with a specific testing activity, enabling early detection of defects and continuous quality validation throughout the development process.

At higher levels, additional methods such as keyword-driven and exploratory testing may be incorporated, especially for non-functional requirements like security and performance.

### 2.1.5 Exploration of Existing Automation Tools and Techniques

The landscape of test automation tools is vast, offering a range of capabilities to address different testing requirements across platforms and languages.

**Script-Based**: Tools like Selenium and Appium are popular for web and mobile UI automation, enabling testers to create detailed scripts that provide comprehensive coverage of application functionality. Selenium is widely adopted for web applications, while Appium supports mobile applications across various operating systems [Axelrod, 2018].

**Data-Driven and Keyword-Driven Tools**: These approaches allow separation of test scripts from test data, enabling reuse of scripts with different datasets. Keyword-driven tools simplify automation by replacing complex code with descriptive keywords, making it easier for non-technical users to contribute to test case development [Axelrod, 2018].

**Behavior-Driven Development (BDD)**: Tools such as Cucumber and SpecFlow align testing with business requirements by writing tests in plain language scenarios. This approach improves collaboration between technical and non-technical stakeholders, ensuring that tests remain relevant to business needs [Axelrod, 2018].

**Model-Based Testing (MBT)**: This technique generates test cases based on models that define system behavior, which is valuable for testing complex systems where manually designing test cases is impractical. MBT automates test generation and provides coverage for intricate system interactions [Axelrod, 2018].

**Continuous Testing in CI/CD Pipelines**: Continuous integration and continuous delivery (CI/CD) tools, like Jenkins and CircleCI, support automated test execution on every code commit, reducing the feedback cycle and enabling rapid identification of defects. This approach integrates testing directly into the development workflow, promoting frequent, incremental testing [Axelrod, 2018].

**Automation Methodologies**: A structured approach, like the Test Management Approach (TMAP), emphasizes test preparation, specification, and execution. TMAP promotes risk-driven testing, ensuring that test automation targets high-risk areas first, thereby optimizing resources and enhancing overall quality [Axelrod, 2018].

Test automation integrates well-defined test case design, rigorous coverage criteria, and powerful tools and methodologies. The strategic application of these elements significantly enhances testing efficiency, reduces manual effort, and improves the robustness of software applications.

## 2.2 Large Language Models

Large language models are advanced artificial intelligence systems designed to understand, generate, and engage with human language. Using powerful machine learning techniques, particularly deep learning, LLMs perform a wide range of NLP tasks, such as text generation, translation, question answering, and summarization. This versatility enables them to be applied in diverse fields, from customer support to scientific research [Jurafsky and Martin, 2025; Zhao et al., 2023].

These models undergo a multi-phase training process that enables them to learn language patterns, grammar, and contextual relationships from vast amounts of text data. The initial phase, known as pretraining, is typically based on self-supervised learning, a form of unsupervised learning where the model predicts the next word or token in a sequence using large-scale textual corpora. Following this, a fine-tuning phase is often applied, where the model is further trained on specific tasks using supervised learning to refine its capabilities. Finally, reinforcement learning with human feedback is employed to align the models responses with desired outputs, ensuring greater coherence, relevance, and ethical considerations. Through these stages, the language model progressively captures complex linguistic structures, contextual dependencies, and semantic meanings [Jurafsky and Martin, 2025; Zhao et al., 2023].

LLMs operate using deep learning, a subset of machine learning that enables models to learn intricate relationships and patterns within data. Recent advancements in deep learning have introduced new architectures that challenge the dominance of the Transformer model. For instance, Meta's Megalodon model extends the context window to millions of tokens without requiring extensive memory, outperforming Transformer models of similar size in processing large texts [Research, 2024]. Similarly, the Mamba architecture incorporates a form of working memory, enhancing speed and computational efficiency, particularly in non-language domains like audio and genetics [Gu, 2024]. Additionally, Liquid Neural Networks, inspired by biological systems, offer more efficient and transparent AI models, capable of ongoing learning and improved visual data processing [Researchers, 2024]. One such architecture is the Mamba 2 [Gu and Dao, 2023; Dao and Gu, 2024], a model from the Mistral family, which builds upon the principles of State Space Models (SSMs), as described in Section 2.3. Mamba 2 introduces significant advancements in sequence modeling by employing linear-time complexity methods through selective state spaces. These innovations optimize the efficiency and scalability of large-scale language models, enabling them to process vast amounts of data more effectively.

As part of the Cosdetral framework, Mamba 2 demonstrates improved practicality for real-world applications, such as natural language processing, scientific research, and customer support systems. Its ability to handle complex language patterns and long sequences efficiently makes it a valuable development in the evolution of modern language models.

## 2.3 Discrete State Space Model

SSMs provide a unified framework for describing the evolution of a hidden state that interacts with inputs and outputs over time. Depending on whether time is treated continuously or discretely, the equations differ in form but share the same conceptual structure.

In continuous time, the dynamics are captured by a system of differential equations, but for practical implementation, these dynamics are reformulated in discrete time. In the discrete setting, the structured SSM is expressed by recurrence relations that efficiently compute hidden state updates and output mappings at each time step

For many practical machine learning and signal processing tasks, data is sampled at fixed intervals, making the discrete-time formulation more suitable. The hidden state $h_t$ evolves in discrete steps:

$$h_t = A\, h_{t-1} + B\, x_t, \quad y_t = C\, h_t, \tag{2.1}$$

where:

- $h_t \in \mathbb{R}^N$ is the hidden state at time step $t$,
- $x_t \in \mathbb{R}^d$ is the discrete-time input,
- $y_t \in \mathbb{R}^d$ is the discrete-time output,
- $A$, $B$, and $C$ play analogous roles to those in the continuous-time formulation but are interpreted at discrete intervals.

Instead of learning the transition matrices $A$ and $B$ directly, they are derived from underlying continuous parameters $\mathring{A}$ and $\mathring{B}$, along with a parameterized step size $\Delta$. This transformation follows fixed discretization rules:

$$A = f_A(\Delta, \mathring{A}), \quad B = f_B(\Delta, \mathring{B}) \tag{2.2}$$

where the pair $(f_A, f_B)$ defines the specific discretization method applied (see [Gu and Dao, 2023]) for details).

Equation 2.1 is a special case where the system dynamics remain constant over time, referred to as a Linear Time-Invariant (LTI) SSMs. In this scenario, the model becomes mathematically equivalent to convolutions. Unlike standard Convolutional Neural Networks (CNNs), LTI SSMs implicitly parameterize convolution kernels through $(A, B, C)$, allowing them to capture global dependencies. Conversely, classical signal processing ensures that any well-behaved convolution can be represented as an SSMs.

While the standard LTI formulation provides a powerful framework for modeling sequential data, it imposes a fixed evolution of the hidden state over time. To introduce greater flexibility, Selective State Space Models (SSSMs) extend this formulation by allowing the parameters $(A, B, C)$ to vary dynamically at each timestep (see Equation 2.3). This selective mechanism enables the model to modulate its state updates, selectively attending to or ignoring inputs based on contextual relevance, as illustrated in Figure 2.2.

$$h_t = A_t\, h_{t-1} + B_t\, x_t, \quad y_t = C_t\, h_t, \tag{2.3}$$

The introduction of SSSMs has significantly improved performance on complex, information dense tasks, such as language modeling, particularly when the state

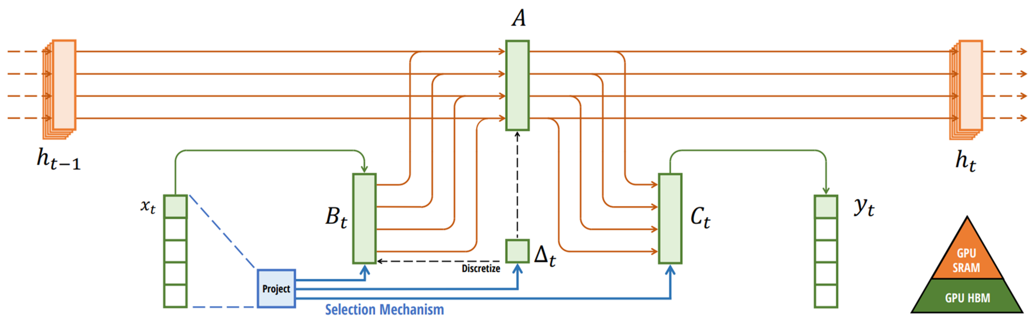



Figure 2.2: Selective State Space Model Architecture [Gu and Dao, 2023].

dimension $N$ is increased. However, the recurrent nature of SSSMs presents computational challenges. Unlike convolutional architectures or attention-based models, which can efficiently leverage modern accelerators like GPUs and TPUs through matrix multiplications, selective SSMs require specialized, hardware-aware implementations. This constraint makes them less efficient than parallelizable architectures such as CNNs and Transformers.

To address these limitations, [Dao and Gu, 2024] propose **Mamba-2**, a novel approach that builds on the strengths of selective SSMs while significantly improving their computational efficiency. By drawing deeper connections between SSSMs and attention mechanisms, Mamba-2 achieves faster training times and enables the use of substantially larger state sizes $N$. This advancement bridges the gap between structured sequence models and modern deep learning paradigms, making state space models more competitive with contemporary architectures in large-scale machine learning applications. Details of the Mamba 2 structure will be introduced in the section 3.3.

### Application in Large Language Models

In the context of LLMs, SSMs can be used to map an input sequence $x_t$ to an output sequence $y_t$ via a latent state sequence $h_t$. The goal is to find a state representation $h_t$ that allows the model to transition from an input sequence to an output sequence effectively.

The state equation describes how the state changes based on the input and the internal dynamics of the system. The output equation describes how the state is translated into the output sequence. By solving these equations, the model aims to uncover the statistical principles that govern the system's behavior, allowing it to make predictions based on observed data.

### Step-by-Step Learning Process

To gain insight into how these matrices influence the learning process, please refer to Figure 2.2. The following steps outline the key considerations:

1. The input signal $x_t$ is multiplied by matrix $B_t$, which describes how the inputs influence the system. This step incorporates new information into the model, updating the state based on the current input.

2. The updated state $h_t$ is a latent space that contains the core knowledge of the environment. This latent space encapsulates the system's internal dynamics and contextual information.

3. The state is then multiplied by matrix $A_t$, which describes how the internal states are connected and represent the underlying dynamics of the system. This step propagates the state forward in time, predicting future states based on the current state.

4. Matrix $C_t$ is used to describe how the state is translated into the output. This step maps the internal state to the observable output, generating predictions or responses based on the current state.

In conclusion, state-space models serve as a powerful framework for modeling dynamic systems, providing a systematic and structured approach to representing and predicting system behavior. Within the domain of large language models (LLMs), SSMs play a pivotal role in enabling effective capture of contextual information, which is essential to generate coherent and contextually relevant predictions. This capability underscores their significance as a valuable tool for tackling complex sequence modeling tasks.

### 2.3.1 Code Generation Using LLM

Language models have become essential tools in software development, significantly enhancing the efficiency and accuracy of the coding process. With capabilities that extend beyond automation, they streamline various stages of development by interpreting high-level instructions, completing code, and assisting in debugging. Developers benefit from the natural language processing capabilities, which allow these models to understand and generate contextually relevant code.

Through contextual code completions and intelligent suggestions, these tools assist developers in expanding and refining their code. When a developer writes an initial portion of a function or script, the model can predict and suggest further code that aligns with best practices, even filling in boilerplate code, optimizing loops, or suggesting data structures that improve efficiency. This guidance helps developers maintain high-quality code while focusing on more complex aspects of their projects.

Debugging is another area where these systems excel. By analyzing code context and identifying errors, they can help developers resolve issues quickly, reducing the downtime caused by debugging. For example, if a developer encounters a specific error or a complex stack trace, the model can provide an interpretation of the issue, suggest possible fixes, and explain steps to troubleshoot. This assistance enhances productivity and allows developers to resolve issues without needing to search through extensive documentation.

The most impactful role of LLMs in software development, however, lies in code generation. By processing natural language instructions, these systems can generate complete functions, classes, or even larger script structures, tailored to meet specific requirements. Developers can describe the functionality they need in simple terms, and the model generates code that performs data handling, integrates APIs, processes files, or even accomplishes machine learning tasks. This automated code generation reduces manual effort and allows for rapid prototyping, especially useful in fast-paced development environments.

When combined with the capabilities of automated testing, the code generation abilities of these tools enable a highly efficient workflow, transforming the development process from ideation to testing and deployment. This cohesive pipeline ultimately ensures reliable and optimized code while accelerating project timelines [Lops et al., 2024; Steenhoek et al., 2024; Schäfer et al., 2023].

### 2.3.2 Fine-Tuning Techniques

Fine-tuning is a critical process in the development and optimization of large language models such as GPT, Mistral or BERT. By further training a pre-trained model on specialized datasets, fine-tuning enhances the models ability to perform domain-specific tasks or address particular user needs. The primary purposes of fine-tuning are summarized below.

Fine-tuning serves several key purposes. It enables domain adaptation by customizing the model to understand the vocabulary, style, and content of a specific field, such as medicine, law, or technology. For instance, fine-tuning a general model like GPT on legal documents creates a system adept at answering legal queries. Additionally, fine-tuning allows for task specialization, where the model is tailored to perform tasks such as sentiment analysis, summarization, translation, or question answering. For example, fine-tuning BERT on a dataset of product reviews can develop a highly effective sentiment analysis tool.

Another major advantage of fine-tuning is efficiency in performance, as it eliminates the need to train a model from scratch, saving computational resources and time. It ensures that the model achieves higher accuracy and relevance for desired applications by adapting its pre-trained knowledge to the new context. Organizations also use fine-tuning for customization, applying proprietary data to align the models with internal terminology, branding, or user-specific requirements.

Fine-tuning techniques are diverse, each designed to optimize large language models for specific needs while balancing resource efficiency and performance. These methods vary in complexity and scope, from fully updating the model's parameters to fine-tuning only a small subset. By selecting the appropriate technique, developers can tailor a model's capabilities to specific tasks or domains without compromising its general-purpose utility.

- **Full Fine-Tuning**: In full fine-tuning, all the weights of the pre-trained model are updated during training on the new dataset. This method is suitable when the target dataset is large and diverse, requiring significant adaptation. While it offers maximum flexibility, it is computationally expensive and risks overfitting on smaller datasets.
- **Adapter Fine-Tuning**: This technique involves adding small trainable neural layers (adapters) to the frozen pre-trained model, with only these layers being updated during training. Adapter fine-tuning is computationally efficient and modular, allowing reusability across tasks. However, it may deliver slightly less performance improvement for highly complex tasks.
- **Low-Rank Adaptation**: LoRA introduces low-rank matrices to the model's existing weight matrices, reducing the number of parameters that need training. This method is ideal for constrained computational resources, balancing performance and efficiency.
- **Prompt Tuning**: Prompt tuning learns a small set of continuous prompt embeddings while keeping the entire model frozen. It is highly lightweight and computationally inexpensive, though it is less flexible for adapting the deeper layers of the model.
- **Prefix Tuning**: Prefix tuning prepends a learned prefix to the model's input representation, with only the prefix parameters being trained. This technique is often applied in text generation tasks, preserving pre-trained knowledge while allowing task-specific adaptation.
- **Instruction Tuning**: Instruction tuning involves fine-tuning the model on datasets with instruction-response pairs, enhancing its ability to follow explicit instructions. It improves usability and generalization to unseen instructions but depends on high-quality datasets.
- **Reinforcement Learning with Human Feedback**: Combine fine-tuning with reinforcement learning to align the model's outputs with human preferences. This technique is used in conversational agents like ChatGPT to improve response alignment with user expectations. However, it is resource-intensive and relies on curated human input.

Among the various fine-tuning techniques, LoRA emerges as a promising approach due to its efficiency and scalability. By introducing low-rank updates, this method significantly reduces the number of trainable parameters, enabling lightweight fine-tuning of pre-trained models [Hu et al., 2021]. Such an approach is particularly appealing for structured and syntax-heavy domains like programming languages, as it allows for targeted adaptation without demanding extensive computational resources [Xu et al., 2023]. Its ability to optimize models with minimal overhead makes it especially suitable for resource-constrained environments or scenarios where computational efficiency is a priority. While not necessarily the optimal solution for all tasks, this technique strikes a balance between performance and efficiency, making it a strong candidate for adapting large language models to code generation.

### 2.3.3 Prompt Engineering

In the context of utilizing LLMs to enhance software testing automation, Prompt Engineering emerges as a pivotal technique. It focuses on constructing precise and context-aware inputs to maximize the effectiveness of pre-trained models, guiding them to produce responses that align with specific software testing needs. This approach enables users to harness the full potential of language models without altering their underlying architecture, making it especially practical for rapid deployment in dynamic testing environments.

Prompt Engineering revolves around the strategic design of inputs, or prompts, that instruct the model on how to approach a given task. For example, when automating the generation of test cases in Python, a prompt can explicitly define the desired functionality, constraints, and output format. This ensures the generated code is both syntactically correct and contextually relevant, addressing specific testing scenarios with precision.

The role of Prompt Engineering is twofold: it acts as a bridge between the models general-purpose capabilities and the domain-specific requirements of software testing, and it provides a cost-effective alternative to fine-tuning. By leveraging well-crafted prompts, software testers can dynamically adapt the language models to tasks such as test case generation, bug detection, and test data creation without the need for extensive computational resources or additional training datasets.

Key principles of effective Prompt Engineering include:

- **Clarity and Specificity:** Prompts must be clear, unambiguous, and specific, providing explicit instructions that leave little room for misinterpretation by the model.
- **Contextual Framing:** Including relevant context, such as the system under test or the specific type of test case required, helps ensure outputs align with testing goals.
- **Examples and Templates:** Incorporating examples or templates into the prompt can guide the model toward producing outputs consistent with desired formats or patterns.
- **Iterative Refinement:** Testing and refining prompts iteratively can significantly improve output quality, particularly for complex or nuanced testing

tasks.

- **Domain-Specific Terminology:** Including testing-related terminology in the prompt enables the model to better understand and respond to the specialized requirements of software testing.

In the field of software testing automation, Prompt Engineering is particularly advantageous for tasks requiring adaptability, such as dynamic test case generation and exploratory testing. By crafting tailored prompts, testers can effectively utilize advanced language models to handle complex scenarios, reduce manual effort, and accelerate the testing process. This makes Prompt Engineering a key component of methodologies that integrate AI-driven solutions into software testing workflows.

# Chapter 3

# Methodology: Mamba LLM and Fine-Tuning

This chapter delves into the methodology of fine-tuning Large Language Models (LLMs) for test automation, focusing on the innovative Mamba architecture. The advent of LLMs has revolutionized code generation, transforming the landscape of software development and testing. This chapter explores the architecture and pre-training of Mamba architecture, its advantages, and the integration of LoRA to enhance its performance in automated testing frameworks like Pytest. By understanding these components, developers and researchers can leverage the full potential of LLMs to streamline and improve the software testing process.

## 3.1 Applications and Impact of LLMs

Language models have transformed numerous industries by leveraging natural language processing to understand, interpret, and generate human language. These models excel in a wide array of natural language processing tasks, including text classification, sentiment analysis, named entity recognition, summarization, and more. Their powerful language comprehension capabilities make them invaluable assets across fields like healthcare, finance, legal services, and customer support. One of their most impactful contributions is the ability to process large volumes of unstructured data, enabling organizations to automate workflows, enhance decision making, and improve communication. subsubsectionApplications in Software Development In software development, these models are particularly transformative. They streamline workflows by generating code, assisting with debugging, and automating documentation. For developers, they provide contextual code suggestions, help complete functions, and even anticipate potential issues within code before they manifest, which reduces time spent on debugging and facilitates a more efficient software development lifecycle. Additionally, language models assist in creating comprehensive documentation and usage examples, making it easier to maintain high standards across teams and to onboard new developers quickly.

subsubsectionAdvancements in Software Testing In the specialized field of software testing, advanced language models offer significant improvements in automation and quality assurance. Traditionally, software testing has required substantial manual effort to create test cases, execute tests, and identify defects. These models streamline the process by automating various tasks, such as test case generation and anomaly detection. By analyzing requirements or functional specifications expressed in natural language, they can produce relevant test cases that cover a wide range of scenarios, reducing the time needed for test creation and ensuring more thorough test coverage.

Furthermore, they contribute to automated anomaly detection by identifying potential issues or inconsistencies in test results. By comparing expected outputs with actual outcomes, these models can detect discrepancies indicating bugs or performance problems. This enables testers to address critical defects promptly, thus improving software reliability. Their predictive capabilities also help prioritize testing efforts by focusing on parts of the software that are most prone to failure or have recently undergone changes. This prioritization optimizes testing resources, reduces redundant testing, and increases the robustness of the final product.

Language models further support software testing through intelligent documentation generation. By summarizing test outcomes, creating detailed reports, and tracking test coverage, they improve communication among teams and maintain clear documentation, which is essential for continuous integration and deployment (CI/CD) workflows. This automation of documentation significantly reduces the time teams spend on these tasks, freeing up resources for more strategic activities and enhancing transparency and accountability in the testing process.

Language models have expanded the scope of natural language processing beyond conventional text analysis, driving significant advancements in software testing. By improving automation, streamlining test case generation, accelerating anomaly detection, and simplifying documentation, they help software development teams enhance the quality, speed, and scalability of their testing workflows. These AI-powered tools support the delivery of higher quality software that meets stringent standards while adapting to the fast pace of modern development cycles.

In the context of software testing, language models offer substantial improvements in automation and quality assurance. Traditionally, testing required substantial manual effort to design test cases, execute tests, and identify defects. However, these models automate numerous aspects of the testing process, from generating diverse and relevant test cases to detecting anomalies. By analyzing requirements or functional specifications written in natural language, they create comprehensive test scenarios efficiently, reducing manual effort, and ensuring broader test coverage.

## 3.2 Low-Rank Adaptation

LoRA is a sophisticated fine-tuning technique that we will employ for automated testing with Pytest. This method was initially proposed in the seminal paper by Hu et al. (2021), titled "LoRA: Low-Rank Adaptation of Large Language Models" [Hu et al., 2021].

### 3.2.1 Introduction to LoRA

#### Definition and Concept

LoRA is designed to adapt large-scale, pre-trained language models to multiple downstream applications efficiently. Traditional fine-tuning methods update all parameters of the pre-trained model, leading to significant storage and computational overhead. It mitigates this issue by adapting only a subset of parameters or learning external modules for new tasks. This approach retains the pre-trained weights frozen while training smaller, task-specific matrices, significantly reducing computational and storage requirements. By leveraging this strategy, it enables efficient task adaptation without compromising model quality.

#### Historical Context and Development

The development of LoRA addresses the limitations of existing fine-tuning techniques, which often introduce inference latency or reduce the models usable sequence length. These methods frequently fail to match the performance of full fine-tuning baselines, creating a trade-off between efficiency and model quality. LoRA overcomes these challenges by optimizing rank decomposition matrices of the dense layers changes during adaptation, while keeping the pre-trained weights frozen.

### 3.2.2 Mechanism of LoRA

#### Mathematical Formulation

LoRA's mechanism involves training some dense layers in a neural network indirectly by optimizing rank decomposition matrices of the dense layers changes during adaptation. For a pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$, the update is constrained by representing it with a low-rank decomposition:

$$W_0 + \Delta W = W_0 + BA$$

where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, and the rank $r \ll \min(d, k)$. During training, $W_0$ is frozen and does not receive gradient updates, while $A$ and $B$ contain trainable parameters. The modified forward pass is given by:

$$h = W_0x + \Delta Wx = W_0x + (BAx)\alpha$$

the factor $\alpha$ is a scalar that controls the influence of the Low-Rank Adaptation update on the weight matrix. Heres what each term represents:

- $W_0$: The original weight matrix of the model.
- $x$: The input vector.
- $\Delta W$: The LoRA-induced weight adjustment.
- $\Delta W = \alpha BA$: The LoRA decomposition, where:
  - $B$ and $A$ are low-rank matrices (often with rank significantly lower than the full weight matrix).
  - $\alpha$ scales the contribution of these matrices.
  - $A$ is initialized with a random Gaussian distribution, and $B$ is initialized to zero, ensuring $\Delta W = BA$ is zero at the beginning of training.

- Interpretation of $\alpha$
  - Higher $\alpha$ values: Increase the contribution of the LoRA update, giving more importance to the learned adaptation.
  - Lower $\alpha$ values: Reduce the influence of the LoRA update, making the model rely more on the original weights $W_0$.

Effectively, $\alpha$ acts as a scaling factor to balance the trade-off between the pretrained weights and the fine-tuned adaptation introduced by Low-Rank Adaptation. It can be tuned to control how much the model deviates from its original parameters during fine-tuning.

In practice, LoRA can be applied to any subset of weight matrices in a neural network to reduce the number of trainable parameters. The pre-trained model can be shared and used to build many small LoRA modules for different tasks. By freezing the shared model and efficiently switching tasks by replacing the matrices $A$ and $B$, it significantly reduces storage requirements and task-switching overhead. This implementation detail is illustrated in Figure 3.1.

### 3.2.3 Advantages of LoRA

#### Computational Efficiency

LoRA enhances training efficiency and lowers the hardware barrier to entry by up to three times when using adaptive optimizers. This is achieved by optimizing only

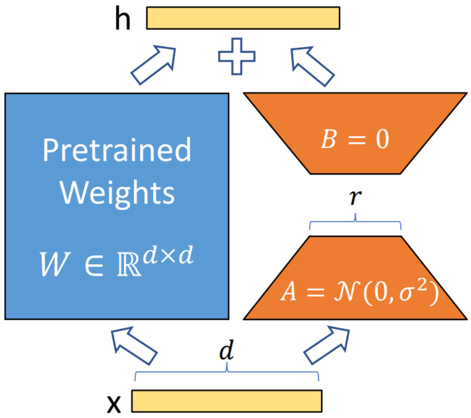



Figure 3.1: Mechanism of Low-Rank Adaptation (LoRA)

the injected, much smaller low-rank matrices, rather than calculating the gradients or maintaining the optimizer states for most parameters.

#### Memory Utilization

By freezing the pre-trained weights and optimizing only the low-rank matrices, LoRA significantly reduces memory utilization. This allows for the efficient deployment of multiple task-specific models without the need for extensive storage resources.

#### Scalability

LoRA's design allows for seamless integration with various prior methods, such as prefix-tuning. Its simple linear design enables the merging of trainable matrices with the frozen weights during deployment, introducing no inference latency compared to a fully fine-tuned model. This scalability makes it a versatile tool for a wide range of applications.

### 3.2.4 Applications of LoRA in LLMs

#### Task-Specific Adaptation

LoRA's ability to adapt pre-trained models to specific tasks without requiring extensive computational resources makes it ideal for task-specific adaptation. By fine-tuning only the low-rank matrices, it enables efficient and effective adaptation to new tasks, enhancing the model's performance on task-specific data.

#### Domain-Specific Fine-Tuning

LoRA enables domain-specific adaptations of LLMs by allowing targeted fine-tuning without modifying the core pre-trained model. This approach is particularly valuable in automated testing, where lightweight, specialized modules can be trained to generate and validate test cases efficiently. For instance, in the context of test automation with Pytest, LoRA facilitates the creation of AI-assisted testing frameworks that enhance test generation and execution while preserving computational efficiency.

Pytest is a widely used testing framework in Python, known for its simplicity, scalability, and extensive plugin ecosystem. It supports test discovery, parameterization, and rich reporting, making it suitable for both small-scale unit tests and large, complex test suites. Moreover, Pytest can be seamlessly integrated with the Selenium Library to automate regression testing for web applications. By installing and configuring both libraries, testers can leverage Pytests flexibility to manage and execute Selenium-based tests, benefiting from its structured test execution and powerful assertion mechanisms [Axelrod, 2018].

By incorporating LoRA into Pytest-based testing pipelines, organizations can further optimize test automation, enabling adaptive and intelligent test case generation tailored to specific industry requirements. This combination enhances the reliability of software validation while reducing manual effort and computational overhead.

## 3.3 Mamba Architecture and Pre-training Overview

This research focuses on the generation of code from textual descriptions using Mamba. Crafting accurate code has traditionally been a laborious process, necessitating programming expertise and the ability to troubleshoot bugs effectively. However, the advent of LLMs like ChatGPT has revolutionized this landscape. Now, developers and automated software quality specialists can simply input their manual test cases and receive swift, precise code for automated testing.

In May 2024, Mistral AI introduced a groundbreaking model, *Codestral*, specifically tailored for code generation. This model quickly gained recognition for its superior performance on benchmark evaluations. However, its size spanning 22 billion parameters posed practical challenges, particularly in terms of computational demands and storage requirements, which rendered it less accessible for individual developers and smaller teams.

To address these limitations, Mistral AI subsequently launched a more compact variant, *Codestral Mamba*. Despite its reduced size of 7 billion parameters, Codestral Mamba delivers competitive performance, often surpassing the larger Codestral model in certain coding benchmarks. This balance of efficiency and performance has made Codestral Mamba an attractive choice for developers seeking a lightweight yet powerful solution for automated code generation. This capability is largely due to its utilization of the SSM structure, which enhances contextual understanding, improves memory retention for handling large codebases, and ensures high-quality code generation with remarkable efficiency

### 3.3.1 Advantages of Codestral Mamba

The Codestral Mamba model offers a range of features that set it apart from general-purpose LLMs and competing code-generation models such as ChatGPT and Gemini. These features include:

**Specialization in Code Generation** Unlike general-purpose LLMs, Codestral Mamba has been fine-tuned on an extensive dataset encompassing programming languages, algorithms, and best practices in software development. This targeted specialization enables the model to produce high-quality, context-aware code snippets and deliver accurate solutions for complex coding problems. Whether debugging, refactoring, or writing new code, Codestral Mamba consistently demonstrates superior precision and adaptability.

**Benchmark Performance** As shown in the benchmark performance comparison presented in Table 4.3 in Chapter 4, the Codestral Mamba model demonstrates superior performance compared to other models. This performance is consistent across different test scenarios, validating its effectiveness in test automation tasks. Remarkably, it also rivals larger models like Codestral (22B) and CodeLlama (34B) in specific benchmarks, showcasing its advanced coding capabilities within a compact architecture.

**Efficient Inference** One of the defining attributes of Codestral Mamba is its efficient inference mechanism. By employing linear-time inference and sequence modeling techniques, the model can process extensive input sequences without significant performance degradation. Linear-time inference ensures response times scale predictably with input length, while the ability to handle sequences of effectively infinite length allows for detailed comprehension of large codebases. These features significantly enhance productivity in code generation and debugging tasks.

**In-Context Retrieval Capabilities** With an in-context token capacity of up to 256,000 tokens, Codestral Mamba excels in managing and retrieving relevant information from large and complex inputs. This capability enables the model to generate precise, contextually relevant outputs tailored to specific programming scenarios. For example, when debugging a function, the model can seamlessly integrate contextual details such as variable names and logic structures to provide targeted solutions[1][2].

**Flexible Deployment Options** Codestral Mamba supports multiple deployment configurations, including the mistral-inference SDK and TensorRT-LLM for optimized performance. Additionally, the model weights are available on HuggingFace, with upcoming support for llama.cpp expected to further simplify local inference. These deployment options, combined with licensing flexibility under Apache 2.0, ensure the model can be integrated seamlessly into diverse development environments.

**Accessibility and Licensing** For ease of testing, Codestral Mamba is available on *la Plateforme* (codestral-mamba-2407) alongside its larger counterpart, Codestral 22B. While Codestral Mamba is offered under an open-source Apache 2.0 license, the larger model is available for self-deployment under a commercial license or a community license for evaluation purposes. This broad accessibility ensures that developers can leverage the models capabilities in a manner that best suits their project requirements.

In summary, the Codestral Mamba model combines state-of-the-art performance, accessibility, and efficient inference to establish itself as a robust tool for automated code generation. Its tailored architecture, contextual understanding, and deployment flexibility make it an invaluable asset for advancing the automation of software testing and development workflows.

### 3.3.2 Architecture of Mamba

Mamba represents a novel neural network architecture specifically designed to efficiently model long sequences by leveraging selective state-space models. By integrating the core principles of earlier SSM frameworks with the Multi-Layer Perceptron (MLP) block from Transformers, Mamba offers a unified and streamlined architecture. This integration simplifies previous deep sequence modeling paradigms, resulting in a computationally efficient and performance-driven model optimized for real-world applications.

#### Core Components

The core of the Mamba architecture is built around a refined version of the selective SSM, which is both faster and competitive with Transformers in language modeling tasks. This refinement is achieved through the State Space Duality (SSD) framework, leading to the development of Mamba-2, an advanced iteration of the original Mamba model.

[1] Mistral AI, Codestral Mamba, available at: `https://mistral.ai/en/news/codestral-mamba`

[2] 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/`

#### Key Properties

The Mamba architecture incorporates several key properties that make it a robust choice for sequence modeling tasks across various domains:

1. **High Quality**: The selectivity mechanism in Mamba ensures strong performance across dense modalities such as language and genomics. This selectivity allows the model to focus on relevant information, enhancing its predictive capabilities.

2. **Efficient Training and Inference**: Mamba's design enables linear scaling of computation and memory requirements with sequence length during training. Furthermore, during inference, the model employs autoregressive unrolling with constant time per step, eliminating the need for caching previous elements and significantly enhancing efficiency.

3. **Long-Context Capability**: The architecture's ability to handle extended contexts, combined with its computational efficiency, ensures robust performance on real-world data. This makes Mamba an ideal solution for tasks requiring the processing of long sequences, such as document analysis and genomic data modeling.

#### Architectural Design

The architecture of Mamba is intentionally simple and homogeneous, comprising two primary components:

- **Selective SSM Block**: The central component of the Mamba architecture, this block effectively models sequences by combining the strengths of Selective SSMs and the selective scan algorithm. This integration facilitates dynamic input handling while maintaining computational efficiency.

- **MLP Block**: Inspired by Transformer architectures, the MLP block enhances the model's capacity to capture complex patterns and relationships within the data. When paired with the modified Selective SSM block, it creates a comprehensive and flexible modeling framework. The combination of the modified parallel Mamba block, together with using Structured SSD as the inner state space models layer, results in the Mamba-2 architecture.

This modular design allows Mamba to achieve an optimal trade-off between simplicity and effectiveness, making it a versatile tool for a wide range of sequence modeling applications.

### Evolution of Architecture

The image depicted in Figure 3.2 illustrates the evolution of the Mamba architecture from earlier models such as H3 and Gated MLP. This progression underscores the integration of key components essential for efficient sequence modeling.

1. **H3 Architecture**:
   - Employs a sequence transformation mechanism followed by a convolutional (Conv) layer.
   - Integrates a Selective State Space Model (SSM) block for state space modeling.
   - Utilizes linear projections and nonlinear activations to form the MLP block.
2. **Gated MLP**:
   - Simplifies the architecture by introducing gated mechanisms within the MLP block.
   - Retains essential components, including sequence transformation and linear projections.
3. **Mamba Architecture**:
   - Combines the strengths of H3 and Gated MLP models.
   - Features a refined Selective SSM block optimized for sequence modeling.
   - Includes convolutional layers and linear projections to enhance representational capacity.
   - Integrates nonlinearities within the MLP block for capturing intricate patterns in data.
4. **Mamba-2 Architecture**:
   - Incorporates insights from the paper [Dao and Gu, 2024] to make slight modifications to the Mamba block.
   - Introduces a Grouped-Value Attention (GVA) head structure and repositions all data-dependent projections to occur in parallel at the beginning of the block.
   - Allows for the implementation of tensor parallelism.
   - Utilizes SSD as the inner SSM layer, resulting in the Mamba-2 architecture.

This architectural evolution, as illustrated in Figure 3.2, highlights the thoughtful design progression that underpins Mamba's ability to achieve state-of-the-art performance in sequence modeling tasks.

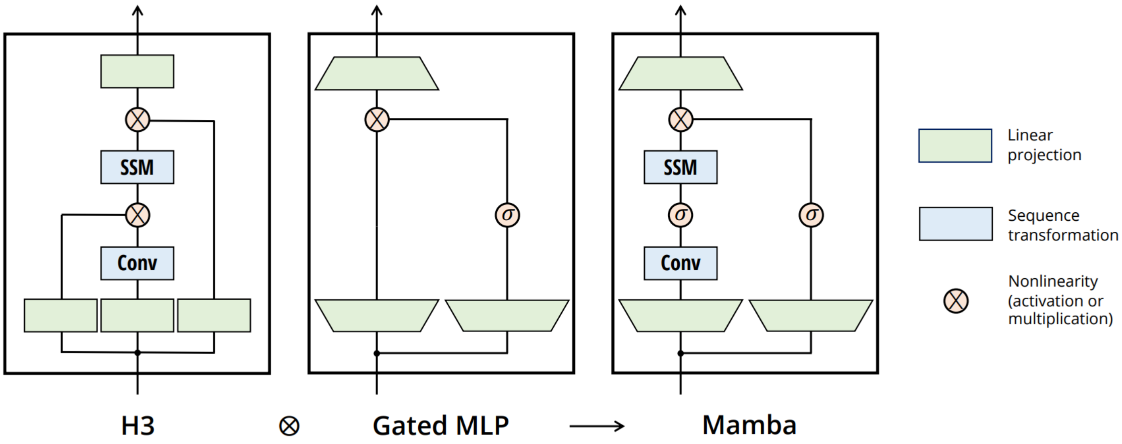



Figure 3.2: Evolution of the Mamba Architecture from H3 and Gated MLP [Gu and Dao, 2023].

This modular design allows Mamba to achieve an optimal trade-off between simplicity and effectiveness, making it a versatile tool for a wide range of sequence modeling applications.

Finally, building upon the results presented in [Dao and Gu, 2024] and leveraging its implementation available on GitHub[3], we present the Mamba-2 structure in Figure 3.3. This figure provides a detailed visualization of the primary components and their interconnections, highlighting the geometric configuration and structural innovations underpinning the design.

The mathematical formalism underlying the Mamba-2 structure *Block*, as depicted in Figure 3.3, is captured by the equations detailed in the subsequent section. These equations define the core computational pipeline, outlining the sequence of operations applied to the input data. For simplicity, we assume a batch size of 1, which is not explicitly considered in the formulation. Additionally, Python-style operations, such as *split* and *rearrange*, are used throughout the equations to enhance readability and understanding for those familiar with programming notations.

The process begins by transforming the input $u$ (an input sequence of shape $\mathbb{R}^{L\times d}$, where $L$ is the maximum sequence length and $d$ is the model dimensionality) using the weight matrices $W^{(zxBC\Delta)}$ (a projection matrix of shape $\mathbb{R}^{d\times dm}$, where $dm = 2\cdot e\cdot d + 2\cdot \text{ngroups}\cdot d_{\text{state}} + \text{nheads}$), which generate the intermediate representations $x_p$:

$$x_p = uW^{(zxBC\Delta)} \in \mathbb{R}^{L\times d} \tag{3.1}$$

The block then employs a split mechanism to obtain $z, S_{xBC}, \Delta$, where $S_{xBC}$ is a matrix formed from the variables $x$, $B$, and $C$. Here, $z$ has shape $\mathbb{R}^{L\times nheads\times m}$

[3] https://github.com/state-spaces/mamba

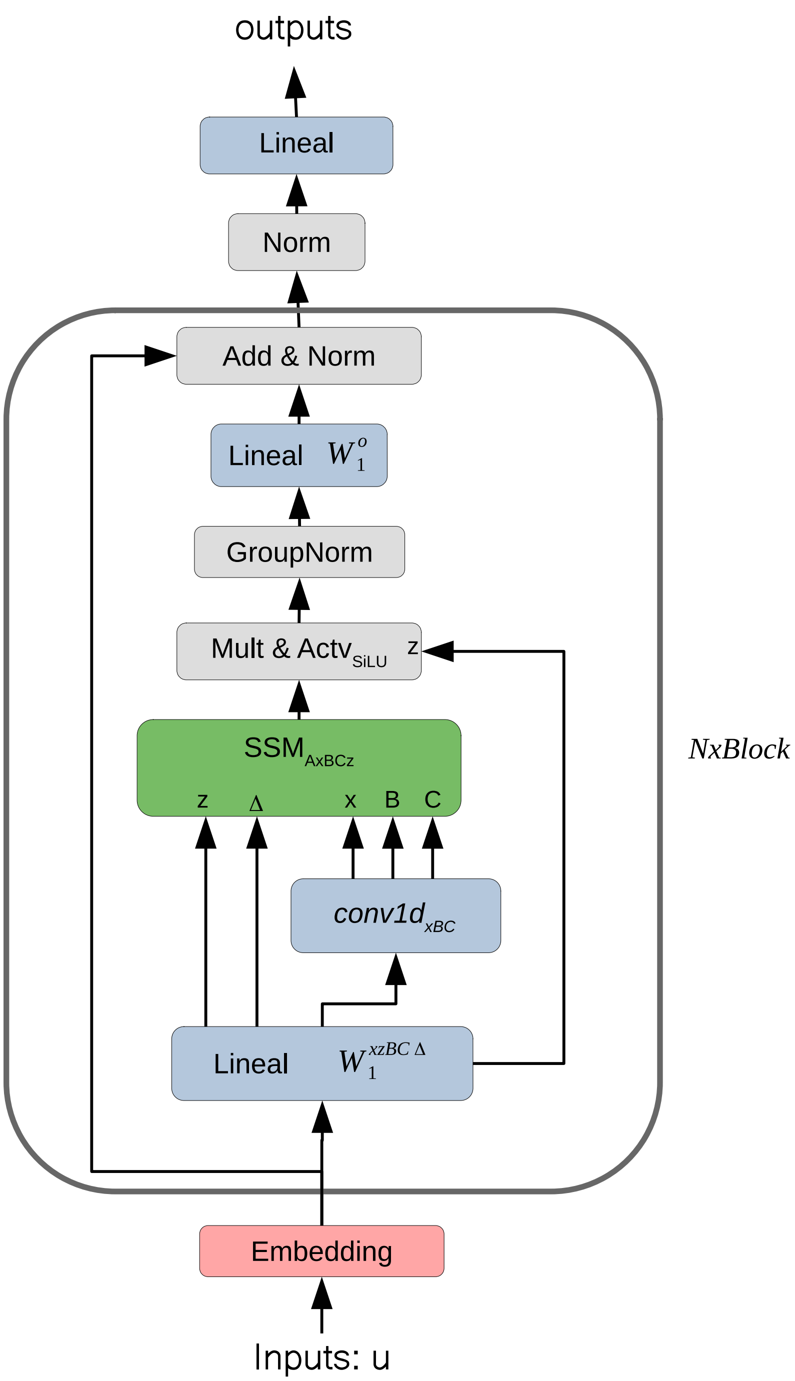



Figure 3.3: Mamba-2 SSM structure.

(with $nheads = \frac{e \cdot d}{headdim}$, where $headdim$ is the size of each attention head), $S_{xBC}$ has shape $\mathbb{R}^{L \times k}$, and $\Delta$ has shape $\mathbb{R}^{L \times nheads}$:

$$z, S_{xBC}, \Delta = \text{split}(x_p) \tag{3.2}$$

Next, a 1D convolution is applied to $S_{xBC}$, producing $x_c$, where $x_c$ has shape $\mathbb{R}^{L \times k}$:

$$x_c = \text{conv1d}(S_{xBC}) \tag{3.3}$$

Then, a split mechanism is applied to $x_c$ to obtain $x, B, C$, followed by a rearrange operation on each. Here, $x$ has shape $\mathbb{R}^{L \times nheads \times m}$, while $B$ and $C$ have shape $\mathbb{R}^{L \times ngroups \times n}$, where $ngroups$ is the number of groups used in the grouped projection of components:

$$x, B, C = \text{rearrange}(\text{split}(x_c)) \tag{3.4}$$

At the core of the block lies a State Space Model (SSM) parameterized by $A, B, C, z$, and $\Delta$, which transforms $x$ into $y$, where $y$ has shape $\mathbb{R}^{L \times nheads \cdot m}$:

$$y = SSM_{A,B,C,z,\Delta}(x) \tag{3.5}$$

A gating mechanism, using an activation function $\phi$ (SiLU), modulates $y$ with $z$, producing $y_g$ (also of shape $\mathbb{R}^{L \times nheads \cdot m}$):

$$y_g = y \cdot \phi(z) \quad (\text{with } \phi \text{ being SiLU}) \tag{3.6}$$

This is then normalized using group normalization (groupnorm), resulting in $y_{gn}$:

$$y_{gn} = \text{groupnorm}(y_g) \in \mathbb{R}^{L \times nheads \cdot m} \tag{3.7}$$

Finally, the output of the block is obtained by projecting $y_{gn}$ through the weight matrix $W^{(o)}$, where $W^{(o)}$ has shape $\mathbb{R}^{(nheads \cdot m) \times d}$, resulting in the final output of shape $\mathbb{R}^{L \times d}$:

$$\text{out} = y_{gn} W^{o} \tag{3.8}$$

This structured design ensures both computational efficiency and scalability, making the Mamba-2 block an excellent choice for high-performance applications.

## 3.4 Integrating LoRA into Mamba2 Architectures

According to the Mamba-2 architecture (see Fig. 3.3) described in Section. 3.3.2, Low-Rank Adaptation (LoRA) is seamlessly integrated into key components of the model, specifically the **input projection matrices** ($W^{(xzBC\Delta)}$ and the **output projection** matrix ($W^{(o)}$) (see Fig. 3.4). By replacing the standard weight matrices in these layers, LoRA enables the model to adapt efficiently to specialized tasks without the need for extensive retraining of the entire model. This approach significantly reduces computational requirements while maintaining flexibility for task-specific fine-tuning. Consequently, the Mamba-2 architecture achieves enhanced performance on targeted tasks, offering a balance between adaptability and computational efficiency.

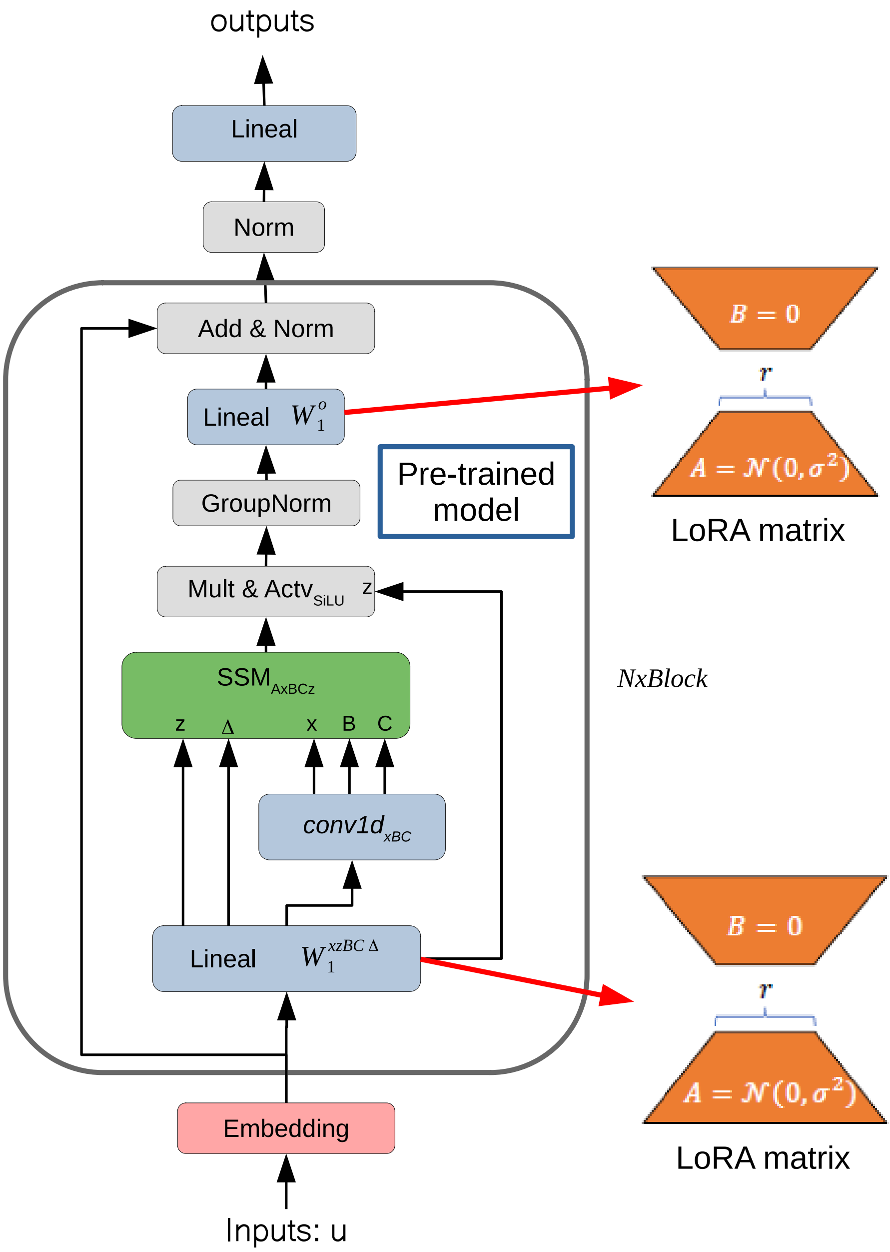



Figure 3.4: Mamba-2 SSM structure with LoRA.

### Input projection

As defined in Section 3.3.2, the input projection is given by the following expression.

$$x_p = uW^{(xzBC\Delta)} \in \mathbb{R}^{L \times d}, \tag{3.9}$$

For simplicity, we replace $W^{(xzBC\Delta)}$ with $W$.

With LoRA, each weight matrix $W$ is decomposed into a sum of the frozen pre-trained weights ($W_{\text{pretrained}}$) and a low-rank adaptation term ($\Delta W$):

$$W = W_{\text{pretrained}} + \Delta W, \quad \Delta W = BA \tag{3.10}$$

where:

- $A \in \mathbb{R}^{r \times d}$,
- $B \in \mathbb{R}^{L \times r}$,
- $r$ is the rank of the low-rank decomposition, with $r \ll \min(d, (h \cdot d_{\text{head}}))$.

Thus, for LoRA-enabled projections:

$$x_p = W_{\text{pretrained}} u + BAu \tag{3.11}$$

The integration of LoRA into the output projection matrix $W^o$ follows the same decomposition methodology as applied to the input projection matrices $W^{(xzBC\Delta)}$.

The decomposition ensures that the output adaptation is concentrated solely on the low-rank components, thereby substantially reducing the number of trainable parameters needed for fine-tuning. This methodology not only minimizes computational overhead but also optimizes hardware utilization and accelerates the training process, making it an efficient solution for fine-tuning large-scale models.

# Chapter 4

# Experimental Procedure and Results

This chapter delineates the experimental methodology and results of fine-tuning the Codestral Mamba 7B Instruct model using the LoRA technique. The selection of this model is motivated by its foundation in SSMs, which offer superior long-range dependency handling, efficient parallelization, and low memory consumption compared to traditional Transformer-based architectures [Gu et al., 2021]. These characteristics make it particularly well-suited for software test case generation, where preserving contextual coherence across multi-step instructions, adapting to complex input variations, and efficiently handling lengthy test scenarios are critical.

To achieve this, the Codestral Mamba 7B model was obtained following the official procedures outlined on Hugging Face[1]. The fine-tuning process was conducted using the foundational codebase provided in the official repository[2], which was systematically adapted to integrate LoRA, a novel addition that had not previously been implemented within this framework. This modification enabled parameter-efficient training, reducing computational overhead while maintaining high model performance. Additionally, the inference phase leveraged the dedicated repository[3], ensuring a streamlined and reproducible deployment process.

By integrating LoRA into the fine-tuning pipeline, this study presents a novel approach to enhancing model adaptability and efficiency in software test automation. The combination of the Codestral Mamba 7B model with LoRA facilitates improved generalization in automated test generation, particularly in frameworks such as Pytest. These advancements contribute to the ongoing evolution of software verification methodologies, highlighting the potential of LLMs to revolutionize automated testing paradigms.

Beyond these advantages, Codestral Mamba excels in computational efficiency, as its

[1]https://huggingface.co/mistralai/Mamba-Codestral-7B-v0.1
[2]https://github.com/mistralai/mistral-finetune
[3]https://github.com/mistralai/mistral-inference

linear scaling properties enable faster inference while maintaining high expressivity. Unlike Transformers, which rely on quadratic self-attention mechanisms, SSMs allow for constant-time retrieval of relevant context, making them more effective for test automation tasks requiring precise logical dependencies [Gu et al., 2020]. Additionally, the model's inherent ability to generalize across structured and unstructured prompts ensures flexibility in generating both formalized test scripts and conceptual descriptions when needed.

The primary objective of this study is to assess the models efficacy in generating high-quality, domain-adaptive, and execution-ready test scripts for diverse software testing scenarios. The integration of LoRA fine-tuning enhances its capacity to align with domain-specific requirements while maintaining computational efficiency, ensuring a balance between performance optimization and adaptability in test case generation.

The fine-tuning process leverages two distinct data sources: the publicly available CodeXGLUE datasets [Lu et al., 2021], which provide a broad spectrum of code-text pairs, and a proprietary database developed by the author, named TestCase2Code[4][5] The inclusion of CodeXGLUE ensures a diverse and generalizable training foundation, while the proprietary TestCase2Code dataset introduces domain-specific knowledge and practical constraints, thereby enhancing the models ability to generate realistic, industry-relevant test scripts. By integrating these datasets, the fine-tuning process balances generalization and specialization, equipping the model with both broad adaptability and task-specific precision.

The Codestral Mamba model is a specialized language model designed to process and generate code efficiently. Its pre-training on diverse programming languages, algorithms, and software development practices positions it as a powerful tool for addressing complex coding tasks. However, to tailor its capabilities to the specific requirements of automated test script generation, LoRA was applied. This technique enables the model to adapt its parameters effectively while maintaining computational efficiency. Additionally, prompt engineering was employed to align the models output with the desired format, ensuring that its responses adhered to the structure and context of the input data.

A key aspect of this study is the integration of both public and private data sources, enabling a comprehensive evaluation of the model's performance. By leveraging the generalizability of the CodeXGLUE dataset and the domain-specific insights provided by the private dataset, the experiments were designed to address a broad spectrum of scenarios. Furthermore, the implementation of LoRA facilitated the development of the Codestral Mamba_QA AI Chatbot, tailored to the specific needs of the company. Codestral Mamba_QA AI enhances response accuracy by leveraging

[4] The TestCase2Code dataset was compiled from a project conducted at DECSIS, ensuring the inclusion of real-world, domain-specific scenarios. Due to confidentiality agreements, this dataset cannot be disclosed.

[5] Additional information about the developed project can be found at `https://web.inductiva.mx/`.

project-specific data, automates routine inquiries to optimize workload management, and adapts to the company's terminology and operational requirements. Additionally, it ensures data security by maintaining internal confidentiality without reliance on third-party solutions, supports continuous learning through user interactions, and provides uninterrupted availability for real-time assistance.

The results presented in this chapter demonstrate the synergistic impact of LoRA fine-tuning and prompt engineering in enhancing the model's ability to tackle natural language code-answering tasks. These findings highlight the role of the Codestral Mamba_QA AI in facilitating an adaptive and efficient workflow, ultimately advancing the state of automated test script generation.

## 4.1 Datasets

### 4.1.1 CodeXGLUE Dataset

In recent years, there has been a significant surge in the application of statistical models, including neural networks, to code intelligent tasks. Inspired by the success of large pre-trained models such as BERT and GPT in NLP, models like IntelliCode and CodeBERT have demonstrated further improvements in code understanding and generation. Despite this progress, the domain of code intelligence has lacked a comprehensive benchmark suite covering diverse tasks.

To address this gap, researchers from Microsoft Research Asia, Developer Division, and Bing introduced CodeXGLUE, a benchmark dataset and open challenge designed specifically for code intelligence. CodeXGLUE, which stands for General Language Understanding Evaluation Benchmark for Code, provides a collection of 14 datasets for 10 diversified code intelligence tasks, along with a platform for model evaluation and comparison. The dataset spans various scenarios, categorized into the following tasks [6]:

- **Code-Code**: Clone detection, defect detection, cloze test, code completion, code repair, and code-to-code translation.
- **Text-Code**: Natural language code search, text-to-code generation.
- **Code-Text**: Code summarization.
- **Text-Text**: Documentation translation.

CodeXGLUE provides a comprehensive suite of tasks, datasets, languages, sizes, baseline systems, providers, and short definitions for each task. For our experiments, we utilize the text-code task, specifically the CONCODE dataset [Iyer et al., 2018] for text-to-code generation. The CONCODE dataset is a large collection of natural

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

language, code, and context tuples derived from online repositories, featuring code from various domains. This dataset introduces new data and methods for learning to generate source code from natural language within the context of a real-world code base.

### CONCODE Dataset Overview

The CONCODE dataset is meticulously organized into training, development, and testing subsets, comprising 100K, 2K, and 2K samples respectively. The dataset is stored in JSON lines format files, where each line encapsulates a JSON object. This object includes a natural language description, contextual information, and the corresponding source code. For a concise summary of the dataset's key characteristics, please refer to Table 4.1.

| Category | Task | Dataset Name | Train/Dev/Test Size |
|---|---|---|---|
| Text-Code | Text-to-Code Generation | CONCODE | 100K/2K/2K |

Table 4.1: CONCODE dataset details.

### Data Format

Each entry in the CONCODE dataset comprises two primary fields: the natural language description (`nl`) and the corresponding source code (`code`). The `nl` field integrates a textual description of the code's functionality with its class environment, demarcated by special tokens such as `concode_elem_sep` and `concode_func_sep`. This structured format ensures that the dataset captures both contextual and functional relationships, thereby facilitating robust learning for text-to-code generation tasks.

The following table illustrates an example of the dataset's JSON structure:

The incorporation of the CONCODE dataset in this study enables a rigorous evaluation of the model's capability to generate accurate and context-aware code snippets based on natural language descriptions. Additionally, utilizing CONCODE facilitates benchmarking against other models, as it is a widely recognized and frequently referenced dataset in the literature. This comparative approach not only validates the performance of our model but also contextualizes its efficacy within the broader landscape of code generation research. For a detailed representation of the dataset's structure, refer to Table 4.2.

## 4.1.2 TestCase2Code Dataset

The TestCase2Code dataset was created in response to a gap identified during the research phase of this dissertation. While various datasets were explored, none contained the specific type of Pytest code necessary for the objectives of this study.

CONCODE Sample Data Representation

```
{
  "nl": "natural_language_description concode_field_sep field1
         concode_elem_sep field2 concode_elem_sep ...
         concode_field_sep method1 concode_elem_sep method2
         concode_elem_sep ...",
  "code": "return_type function_name(parameter_type parameter_name)
          {
             // Function body
             // Can include statements, calls to other functions,
             // operations, etc.
             return return_value;
          }"
}
```

Table 4.2: Dataset's JSON structure.

Pytest, as described in [Okken, 2022], is a Python testing framework originally developed for the PyPy project. Pytest supports a range of test types, including unit tests, integration tests, end-to-end tests, and functional tests, with a rich feature set that includes parameterized tests, fixtures, and assertion rewriting. Despite its widespread adoption in industry and academia, there were no readily available datasets that sufficiently combined manual test cases with their automated counterparts written in Pytest format, particularly in the context of real-world scenarios. The absence of such a database posed a significant challenge.

Given the need for a specialized dataset, the TestCase2Code dataset was conceived as a solution. The dataset's primary objective is to provide a reliable and practical resource for training machine learning models to generate automated functional test cases from manually written ones. This aligns with the research goal of leveraging the Cosdestral Mamba 7B model in combination with the LoRA fine-tuning technique to improve the generation of Pytest code from natural language descriptions of manual test cases. By focusing on manual-to-automated test case transformation, TestCase2Code serves as a key asset in the studys exploration of automating software testing processes, particularly in Python-based testing environments.

The importance of creating the TestCase2Code dataset lies in its tailored approach to real-world, domain-specific testing scenarios. Unlike generic datasets, this dataset was directly derived from a project conducted at the author's workplace. This ensured that the dataset was not only grounded in practical use cases but also rich in contextually relevant examples. The integration of TestCase2Code into the experimental procedures substantially enriches the analysis by providing practical, domain-specific test cases. It bridges the gap between theory and practice by supplying realistic test case data that mirrors the challenges faced by software developers and testers in real-world scenarios.

### Dataset Composition and File Structure

The dataset is structured around 870 distinct elements, each comprising three specific files: one containing automated test code (denoted as **a.txt**), another containing source code information (denoted as **c.txt**), and the last containing a manually written test case (denoted as **m.txt**). These three components together form a comprehensive *triplet* that collectively defines the test case for a given software functionality. In total, there are 2,610 elements within the dataset.

The overall structure of the dataset is outlined in Table 4.3. This structure ensures that each element is thoroughly documented, facilitating a comprehensive evaluation of the software functionality through automated and manual test cases.

**Overall TestCase2Code Dataset Structure**

```
dataset/
├── element_1/
│   ├── 1a.txt ........... Automated functional test case in Pytest code.
│   ├── 1c.txt ....... Source code JavaScript for the tested functionality.
│   └── 1m.txt ................... Manual test case in natural language.
├── element_2/
│   ├── 2a.txt ........... Automated functional test case in Pytest code.
│   ├── 2c.txt ....... Source code JavaScript for the tested functionality.
│   └── 2m.txt ................... Manual test case in natural language.
├── element_3/
│   ├── 3a.txt ........... Automated functional test case in Pytest code.
│   ├── 3c.txt ....... Source code JavaScript for the tested functionality.
│   └── 3m.txt ................... Manual test case in natural language.
├── element_4/
└── ...
```

**Additional elements follow the same structure.**

Table 4.3: TestCase2Code dataset structure.

Each triplet of files contains the following components:

- `a.txt`: This file contains the automated test code written in Pytest. It includes functional tests that are intended to be executed automatically by the Pytest framework to validate the behavior of the software. For an example of the automated test code, refer to Table 4.4.

- `c.txt`: This file contains source code information, specifically JavaScript code written using JSX (JavaScript XML). JSX allows developers to write user interface components in a declarative manner that resembles HTML. The inclusion of this file provides context for the portion of the code being tested. For a detailed example of the source code in JavaScript, refer to Table 4.5.

- `m.txt`: This file contains the manual test case, written in natural language, which specifies the title, steps, and expected results for testing the functionality of the software. These manual descriptions are critical for training machine learning models to generate automated test cases based on textual descriptions. For an example of a manual test case in natural language, refer to Table 4.6.

**Automated Functional Test Case in Pytest Code: Example of Element 5a.txt**

```
url = os.getenv("URL")
passw = os.getenv("PASSW")
@pytest.mark.order(5)
def test_logincredentials(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)
    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")
```

Table 4.4: Automated Functional Test Case.

### Dataset Overview

The dataset consists of two fundamental corpora: manually written test cases in natural language and their corresponding Pytest code, which represents their automated execution. In this section, an overview of these corpora will be provided, highlighting their structure and key characteristics. This dual structure enables the model to learn both the linguistic patterns of instructional test cases and their executable representations, fostering a robust understanding of test case translation.

The manual test case corpus, comprising 86,803 words across 870 files, exhibits a high degree of consistency and standardization critical for ensuring rigorous quality control and uniformity in testing procedures. With a vocabulary size of 939 words, it reflects a specialized and technical lexicon tailored to domain-specific requirements. Structured across 9,486 sentences, the dataset presents a balanced composition that supports both precision and comprehensiveness. The average sentence length of 13.54 words, ranging from a minimum of 4.48 words to a maximum of 186 words, JavaScript Code for the Tested Functionality: Sample of Element 5c.txt

**Note:** This is a sample of the JavaScript code. The complete code includes additional functionality and error handling.

```javascript
import { useDispatch } from "react-redux";
import { setLoading } from "@store/loading/Loading.slice";
import { fetchDefaultParameters, throwResponseErrors } from "@utils/helpers/
    response.helper";
import styles from "./LoginPage.module.scss";
const REQUIRED_FIELDS = ["email", "password"];
function LoginPage() {
  const navigate = useNavigate();
  const dispatch = useDispatch();
  const [form, setForm] = useState({ email: "", password: "" });
  useEffect(() => {
    dispatch(setLoading(true));
    fetch(‘/user/session‘, { ...fetchDefaultParameters(), method: "GET" })

  // More code follows...
}

export default LoginPage;
```

Table 4.5: Source code in JavaScript.

indicates a blend of concise and detailed instructions. This variability enhances the models ability to interpret and generate diverse sentence structures effectively. Furthermore, the lexical diversity score of 92.44 highlights the dataset's richness, ensuring a comprehensive representation of language that strengthens the models ability to generalize across varied test scenarios.

The Pytest code corpus, consisting of 20,600 words, directly corresponds to the manual test cases, providing a structured yet expressive syntax that makes it well-suited for training models in automated test generation. While inherently repetitive due to standardized coding practices, the corpus retains functional diversity, ensuring that the model internalizes essential programming constructs while remaining adaptable to different testing requirements.

With an average lexical diversity of 0.6611, the corpus strikes a balance between repetitive core syntax and varied test logic, reinforcing the models ability to generate robust and reusable test scripts. The dataset also demonstrates consistent sequence lengths, with a mean of 51.90 words and a standard deviation of 11.21, further ensuring that the model captures the structured flow of Pytest cases, from setup to assertions.

Additionally, the low function count per file (1.15 on average) reflects a modular approach to test design, allowing the model to learn well-defined, self-contained test functions. This structured yet flexible nature makes the dataset an invaluable resource, equipping the model with the ability to produce syntactically correct and

Test Case in Natural Language: Sample of Element 5m.txt

**Test Case: Validate Successful Login with Credentials**

| Step | Action | Expected Result |
|---|---|---|
| 1 | Locate the email input field on the login page. | The email input field is identified and present on the page. |
| 2 | Locate the password input field on the login page. | The password input field is identified and present on the page. |
| 3 | Locate the "Enter" button on the login page. | The "Enter" button is identified and present on the page. |
| 4 | Input a valid email address into the email field. | The email address is entered successfully without errors. |
| 5 | Input a valid password into the password field. | The password is entered successfully without errors. |
| 6 | Click the "Enter" button to submit the login form. | The system processes the credentials and initiates the login action. |
| 7 | Verify the navigation to the company selection page. | The URL of the page updates to match the value stored in the environment variable appended with select-company, confirming successful login. |

Table 4.6: Manual Test Case.

functionally meaningful Pytest scripts that align with their corresponding manual test cases in real-world testing scenarios.

### Dataset Partitioning

To facilitate the training and evaluation of machine learning models, the dataset is divided into a training set and a test set. The training set consists of 770 elements, providing a robust foundation for model training. The test set comprises 100 elements, allowing for rigorous evaluation of the model's performance on unseen data. This division ensures that the models are trained on a diverse range of examples while being tested on a separate subset to validate their generalization capabilities.

## 4.2 Preparing the Dataset

To achieve effective fine-tuning of the Codestral Mamba 7B model, the training data must comply with the stringent formatting requirements stipulated by the `mistral-finetune` framework. However, certain modifications are necessary to cater to the specific requirements of the Codestral Mamba model. All data must be stored in the JSONL (JSON Lines) format, where each line constitutes a separate data sample in valid JSON format.

In this study, our focus is exclusively on the *Instruct* data format, which is specifically tailored for instruction-following tasks. The data is organized under the key `"messages"`, which contains a list of dictionaries. Each dictionary includes two primary fields: `"content"` and `"role"`. The `"role"` field designates the participant in the conversation, with possible values of `"user"`, `"assistant"`, or `"system"`. The model is trained (incurring loss) using only those entries where the `"role"` is `"assistant"`, as these represent the responses that the model is expected to generate.

Listing 4.1 illustrates a sample entry of the *Instruct* data format. In this format, `"user"` entries represent the queries or inputs, while `"assistant"` entries correspond to the desired model responses. This sequential arrangement ensures that the model learns to generate appropriate responses based on the preceding user input, thereby enhancing its capability to follow instructions and engage in meaningful dialogue.

Listing 4.1: Example of Instruct data format

```
{
  "messages": [
    {
      "role": "user",
      "content": "User interaction n. 1 contained in document n.1"
    },
    {
      "role": "assistant",
      "content": "Bot interaction n.1 contained in document n.1"
    }
  ]
}
```

In this structure, the `"user"` entries represent the queries or inputs, while the `"assistant"` entries correspond to the desired model responses. This sequential arrangement ensures that the model learns to generate appropriate responses based on the preceding user input, facilitating its ability to follow instructions and engage in meaningful dialogue.

## 4.3 Prompt Training

In this section, we introduce the prompt training approach used to fine-tune the Codestral Mamba 7B model, following the specific structure designed for instruction-following tasks. The training data is organized using the *Instruct* format (Listing 4.1), where each interaction is represented as a sequence of messages, with each message assigned a specific role: "user" for the input queries and "assistant" for the expected model responses. The core objective of this approach is to guide the model in generating contextually appropriate and coherent responses based on the input provided by the user.

The prompt training procedure is carefully structured to ensure the model learns how to follow instructions effectively, aligning its output with the format and nature of real-world task-specific queries. This methodology is designed to improve the models performance on tasks such as question answering, where understanding user queries and generating accurate responses is critical.

### 4.3.1 CodeXGLUE: Structuring Prompts

Figure 4.1 illustrates the sequence of steps involved in the training process, demonstrating how interactions between the "user" and the "assistant" are structured, processed, and utilized to fine-tune the model. As depicted in the figure, an example prompt consists of two key components: the role "user," which identifies the origin of the interaction, and the *content*, which specifies the task to be performed. In this instance, the content directs the generation of a code snippet based on a natural language description and a corresponding code context.

The example used in this figure is derived from the CodeXGLUE repository, specifically from the text-to-code database described at CodeXGLUE repository, specifically from the Text-Code/text-to-code section[7]. This database is characterized by its structured pairing of natural language instructions with code snippets, providing a robust foundation for training models in code generation tasks.

In the visualization, the interaction is represented by two color-coded elements: the red section, which contains the "question" in the form of a natural language instruction, and the green section, which contains the "answer" in the form of the generated code. This structured format ensures clarity in the training process by explicitly defining the input-output relationship, enabling the model to effectively learn how to generate accurate and context-aware code from textual descriptions.

[7] https://github.com/microsoft/CodeXGLUE/tree/main/Text-Code/text-to-code

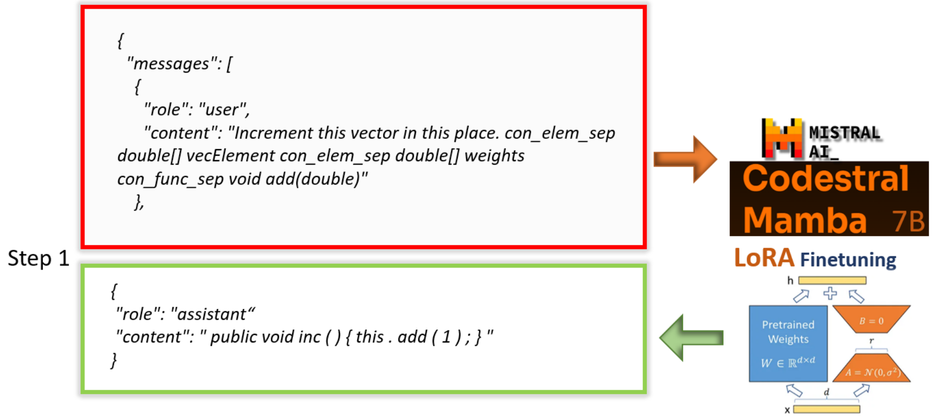



Figure 4.1: Prompt-Based Code Generation to CodeXGLUE/CONCODE.

### 4.3.2 TestCase2Code: Structuring Prompts

Figure 4.2 illustrates the sequence of steps involved in the training process, demonstrating how interactions between the "user" and the "assistant" are structured, processed, and utilized to fine-tune the model. As depicted in the figure, an example prompt consists of two key components: the role "user",which identifies the origin of the interaction, and the *content*, which specifies the task to be performed. In this instance, the content directs the generation of a comprehensive test case using the Pytest library, based on both a provided manual test case and a corresponding .jsx file. This structured input format ensures clarity and consistency, enabling the production of precise test cases aligned with Pytest best practices.

Furthermore, the inclusion of both manual test case instructions and source code enhances the model's capacity to generate high-quality, context-aware outputs, emphasizing the integration of detailed input data in the fine-tuning process. The interaction is represented by the red and green squares, where the red square contains the "question" and the green square represents the "answer." Specifically, the green box contains the response from the assistant.

This visualization highlights the dynamic exchange between the "user" and the "assistant", emphasizing the integration of well-defined prompts and responses. The inclusion of both manual test case instructions and source code in this process enhances the model's ability to generate high-quality, context-aware outputs. It underscores the importance of detailed input and clear formatting in training robust, reliable systems for complex tasks like automated test generation.

## 4.4 Training Configuration

This section details the hyper-parameters and configurations used to fine-tune the Codestral Mamba 7B model on the CodeXGLUE and TestCase2Code dataset, en-

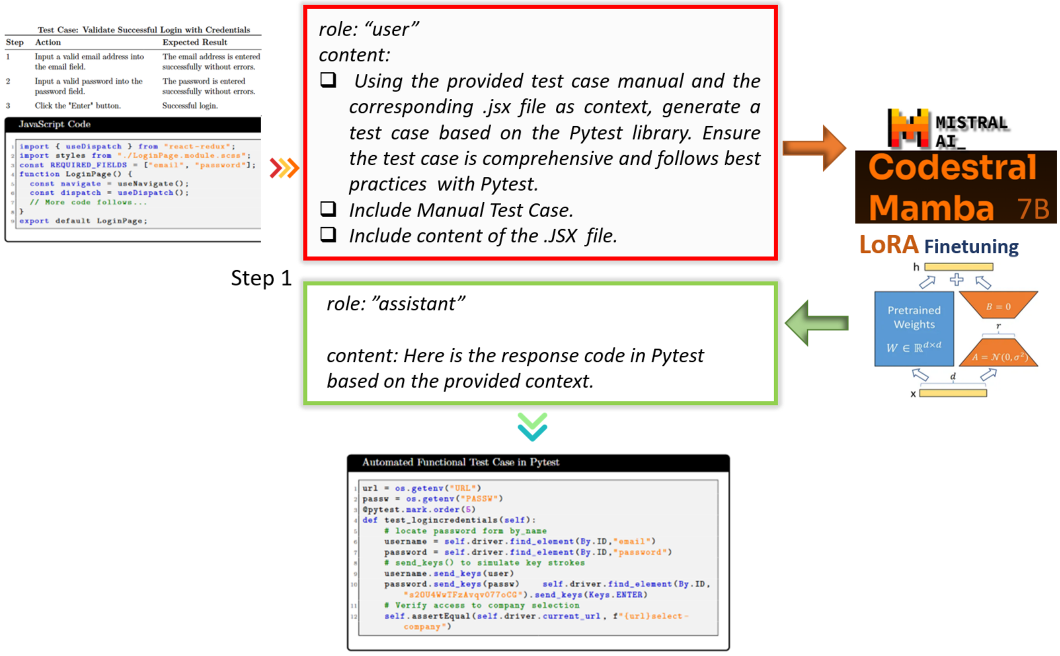



Figure 4.2: Prompt-Based Test Case Generation to TestCase2Code.

suring efficient training and feasibility.

Throughout multiple training iterations, the core hyper-parameters of the base model including the learning rate, batch size, optimizer configurations and loss function parameters remained fixed to preserve the integrity of the original architecture. In contrast, specific experimental adjustments were made to the rank of the LoRA matrices (testing values of 64 and 128) and the maximum sequence length (ranging between 1024 and 4098 tokens). These modifications were systematically evaluated, and the results indicated that such variations did not significantly influence the model's performance. Importantly, all experiments across both datasets were executed using an identical set of hyper-parameters. This uniform approach not only simplifies comparative analysis but also underscores the robustness and generalizability of the fine-tuning strategy across diverse data distributions and tasks, highlighting a key advantage in maintaining a stable training framework.

### Training Process Hyper-parameters

- **Datasets:**
  - `train.json` file inside the database CONCODE in CodeXGLUE.
  - `tc_train.json` file inside TestCase2Code.
- **LoRA Rank:** 128.
- **Sequence Length:** 1024 tokens per batch.

- **Batch Size:** 1, the number of tokens per batch calculated as `seq_len` × `batch_size`.
- **Learning Rate:** $6 \times 10^{-5}$.
- **Optimizer:** AdamW, an adaptive moment estimation optimizer with weight decay.
- **Loss Function:** Cross entropy with masking applied.
- **Number of epochs:** 200.

### Model Parameters Codestral Mamba 7B

For detailed instructions on how to download the Codestral Mamba 7B model, please refer to the official Hugging Face page[8].

1. **Dimensionality (dim)**: 4096 - Size of embedding vectors and hidden states.
2. **Number of Layers (n_layers)**: 64 - Total layers in the neural network.
3. **Vocabulary Size (vocab_size)**: 32768 - Number of unique tokens the model can recognize and generate.
4. **Number of Groups (n_groups)**: 8 - Groups used in operations to reduce computational complexity.
5. **RMS Normalization (rms_norm)**: True - Uses RMS normalization for stabilizing the training process.
6. **Residual in FP32 (residual_in_fp32)**: True - Residual connections computed in 32-bit floating-point precision for numerical stability.
7. **Fused Add Norm (fused_add_norm)**: True - Fused operations for addition and normalization to improve efficiency.
8. **Pad Vocabulary Size Multiple (pad_vocab_size_multiple)**: 1 - Pads vocabulary size for optimized memory alignment.
9. **Tie Embeddings (tie_embeddings)**: False - Input and output embeddings are not shared.
10. **Model Type (model_type)**: "mamba" - Specifies the model type as "mamba".

[8] https://huggingface.co/mistralai/Mamba-Codestral-7B-v0.1

### Hardware Configuration

The training process was conducted on the Vision supercomputer at the University of Évora, leveraging its advanced hardware configuration to optimize performance. The system consists of two compute nodes, each equipped with dual AMD Rome 7742 processors, providing a total of 128 CPU cores, 1TB of system memory, and 8 NVIDIA A100 Tensor Core GPUs, each with 40GB of memory. This powerful infrastructure, interconnected by 8 x 200Gb/s HDR InfiniBand links, allows for high-speed data transfer and efficient parallel processing. Specifically, the training utilized 1 GPU with 40GB of memory, 32 CPU cores, and 122GB of RAM to handle the computational demands of fine-tuning the Codestral Mamba 7B model with LoRA. During training, 624.12 MiB of GPU memory was used for the LoRA matrices, while the model featured 286 million trainable parameters and 7.3 billion non-trainable parameters. This setup ensured optimal performance in processing complex tasks and managing large-scale model adjustments. For more information about the Vision cluster, visit the Vision cluster webpage[9].

Table 4.7 provides an overview of the architectural components of the Codestral Mamba 7B model, detailing the key operations within its layers.

| Layer Name | Operation |
|---|---|
| model | MambaLMHeadModel |
| (backbone) | MixerModel |
| (embedding) | Embedding(32768, 4096) |
| (layers) | ModuleList(64x Block) |
| (mixer) | Mamba2 |
| (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) |

Table 4.7: Layer configurations of the Codestral Mamba 7B model.

Table 4.8 provides a detailed overview of the dimensional specifications of the model's parameters, illustrating the structure and size of both its learnable and non-learnable components. It is important to note that the table represents only the first block, identified as "model.backbone.layers.0", while the complete model consists of a total of 64 such blocks.

## 4.5 Evaluation

Evaluation is a fundamental aspect of assessing the effectiveness and reliability of language models trained for text-to-code generation. This section provides an overview

[9]https://vision.uevora.pt/

| **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] |

Table 4.8: Parameter dimensions of the Codestral Mamba 7B model.

of the metrics and performance indicators used to measure the quality of the outputs. By employing standard metrics and domain-specific measures, the evaluation process ensures an objective and comprehensive analysis of the model's capabilities.

### 4.5.1 Metrics and Performance Indicators

To comprehensively evaluate the model's performance, a variety of metrics and performance indicators are utilized. These metrics provide quantitative measures that help in understanding the model's strengths and areas for improvement. The evaluation process employs both general-purpose and domain-specific metrics to assess different aspects of the generated code's quality. The following subsections delve into the specific metrics used in this evaluation process.

The use of BLEU and CodeBLEU scores ensures a thorough assessment of the model's performance. These metrics provide insights into the syntactic and semantic correctness of the generated code, helping to identify areas for improvement and guiding future enhancements to the model. Additionally, we explicitly evaluate Syntax Match (SM) and Data Flow Match (DM), which are integral components of CodeBLEU, to obtain a more fine-grained understanding of the syntactic and semantic correctness of the generated code.

Throughout our experiments, we evaluate accuracies with respect to Exact Match (EM), SM, DM, and CodeBLEU (CB). SM measures the proportion of matching subtrees between the generated code and the reference codes Abstract Syntax Trees (ASTs) relative to the total number of subtrees in the reference codes AST. DM assesses the percentage of correctly matched, anonymized data flow edges (def-use edges) in the generated code compared to those in the reference code. Since both SM and DM are components of CodeBLEU, evaluating them separately allows for a clearer understanding of the syntactic and semantic correctness of the generated code.

### BLEU Score

The BLEU (Bilingual Evaluation Understudy) score is a widely used metric for evaluating the quality of text generated by machine translation models. In the context of code generation from natural language descriptions, the BLEU score measures the similarity between the generated code and a reference code snippet. It achieves this by comparing n-grams (contiguous sequences of n tokens) in the generated text to those in the reference text, quantifying the level of overlap [Papineni et al., 2002].

The BLEU score ranges from 0 to 1, with higher scores indicating greater similarity to the reference text. This metric is particularly useful for assessing the syntactic correctness of the generated code, as it focuses on the precision of n-gram matches. However, it may not fully capture the semantic correctness or the functional accuracy of the code, making it necessary to complement it with additional evaluation metrics.

The BLEU score is calculated using the following formula:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{4.1}$$

where BP is the brevity penalty, $w_n$ are the weights assigned to different n-gram orders, and $p_n$ are the precision scores for those n-grams. The parameter $N$ represents the maximum length of n-grams considered in the evaluation. Typically, $N$ is set to 4, meaning that BLEU accounts for unigram, bigram, trigram, and 4-gram precision. This ensures a balanced assessment of both individual word accuracy and short phrase coherence in the generated output.

The brevity penalty (BP) is introduced to prevent very short candidate translations from receiving disproportionately high scores. It is defined as:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{4.2}$$

where $c$ is the length of the candidate translation and $r$ is the length of the reference translation. The penalty discourages excessively short outputs by reducing the score when the candidate text is significantly shorter than the reference.

By incorporating multiple n-gram orders and the brevity penalty, BLEU provides a robust measure of syntactic similarity, though it should be complemented with semantic-aware metrics for a more comprehensive evaluation of code generation quality.

The precision scores $p_n$ are calculated as the ratio of the number of matching n-grams in the candidate translation to the total number of n-grams in the candidate translation. The weights $w_n$ are typically set to give equal importance to all n-grams, but they can be adjusted to emphasize certain n-gram lengths if desired.

The BLEU score is a crucial metric for evaluating the syntactic accuracy of code generated from natural language text. Its application ensures that the generated code closely matches the reference code in terms of structure and syntax. While it is a valuable tool for evaluating the quality of generated code, it should be complemented with other metrics that assess semantic and functional correctness to provide a comprehensive evaluation.

#### CodeBLEU Score

The CodeBLEU score is a specialized metric designed specifically for evaluating the quality of generated code. It extends the traditional BLEU score by incorporating additional factors that are crucial for code quality, such as syntactic structure, data flow, and semantic correctness. This holistic approach ensures a more thorough assessment of the generated code's overall quality and correctness [Ren et al., 2020].

CodeBLEU combines four key components:

1. **n-gram match (n-gram)**: Similar to the BLEU score, this component measures the precision of n-gram matches between the generated and reference code, capturing surface-level similarity.
2. **Weighted n-gram match (w-ngram)**: This component assigns varying weights to different n-grams based on their importance in code structure, emphasizing crucial code elements and improving the relevance of the similarity measurement.
3. **Syntax match (SM)**: This component evaluates the syntactic structure of the generated code by comparing its Abstract Syntax Tree (AST) with that of the reference code. The AST captures the hierarchical structure of the code, ensuring that the generated code adheres to the correct syntactic rules.
4. **Data Flow match (DM)**: This component assesses the data flow within the generated code by verifying the correct usage of variables and function calls. It ensures that the logical flow of data through the code is consistent with the reference code, addressing aspects of semantic correctness.

The CodeBLEU score is calculated as a weighted sum of these components:

$$\begin{aligned}\text{CodeBLEU} &= \alpha \cdot \text{n-gram match} + \beta \cdot \text{weighted n-gram match} \\ &+ \gamma \cdot \text{syntax match} + \delta \cdot \text{data flow match}\end{aligned} \tag{4.3}$$

where $\alpha, \beta, \gamma$, and $\delta$ are weights assigned to each component. These weights can be adjusted to emphasize different aspects of code quality based on the specific requirements of the evaluation.

By incorporating these additional factors, the CodeBLEU score provides a more comprehensive evaluation of the generated code, capturing both syntactic and semantic aspects. This makes it a valuable metric for assessing the overall quality and correctness of the code generated by the model. In the context of code generation from natural language text, the CodeBLEU score ensures that the generated code not only matches the reference code syntactically but also adheres to the correct data flow and semantic structure.

### Pass@k Metric

The pass@k metric evaluates functional correctness in code generation tasks by determining whether at least one of the top $k$ generated code samples successfully passes a set of unit tests. Unlike match-based metrics, which compare generated code to a reference solution and may overlook semantically equivalent implementations, pass@k directly measures a model's ability to produce functionally correct code.

To compute this metric, the model generates $n$ candidate solutions for a given programming task, where $n$ represents the total number of generated samples. The metric then examines whether at least one of the top $k$ samplesi.e., the first $k$ solutions among the $n$ generatedis correct, meaning it passes all unit tests. The probability of this event is estimated using the following formula [Chen et al., 2021]:

$$\text{pass@k} = \mathbb{E}_{\text{Problems}} \left[ 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right], \tag{4.4}$$

where $c$ denotes the number of correct solutions among the $n$ generated. This formulation accounts for the likelihood of selecting $k$ samples from $n$ without including any correct ones, ensuring a statistically robust estimate of functional correctness.

Since direct computation of this formula can be numerically unstable, a stable numerical implementation iterates over the product term-by-term to prevent computational overflows. The pass@k metric serves as a crucial indicator of a models ability to generate practical and executable code, aligning evaluation more closely with real-world software development requirements.

### 4.5.2 Model Performance Across Datasets

Evaluating the performance of the model across diverse datasets is essential to understand its generalization capabilities and robustness in different contexts. Each dataset presents unique challenges and characteristics, such as variations in natural language descriptions, code structures, and complexity levels. This subsection provides a detailed analysis of the model's performance on two distinct datasets: the TestCase2Code dataset and the CONCODE/CodeXGLUE dataset. By examining the results across these datasets, we aim to highlight the model's strengths, identify areas for improvement, and assess its adaptability to varied code generation tasks. The following subsections delve into the specific evaluation results and insights derived from these datasets.

To further contextualize the model's performance, Figure 4.3 presents a comprehensive comparative analysis of multiple models across a diverse range of code-related tasks and benchmark datasets. This figure, sourced from Mistral AI, serves as a state-of-the-art reference for evaluating model performance across various coding domains. Specifically, it highlights the effectiveness of 7B parameter models and larger architectures on tasks such as HumanEval, MBPP, Spider, and CruxE, as well as their proficiency in multiple programming languages, including C++, Java, JavaScript, and Bash. By providing a detailed comparison, the figure offers a broader perspective on the capabilities of each model, situating their performance relative to other leading models in the field and enabling a more nuanced assessment of their strengths and limitations.

One notable entry in this comparison is Codestral Mamba, which leverages a novel Mamba architecture rather than the conventional Transformer-based approach. Unlike Transformers, Mamba models enable linear-time inference and have the theoretical ability to process sequences of infinite length. This efficiency makes them particularly well-suited for code productivity tasks, where rapid response times and the ability to handle extensive input contexts are crucial. Consequently, Codestral Mamba has been trained with advanced code reasoning capabilities, allowing it to perform competitively with state-of-the-art Transformer-based models.[10]

#### Model Performance Across the CONCODE/CodeXGLUE Dataset

The CONCODE/CodeXGLUE dataset serves as a critical benchmark for evaluating text-to-code generation models. Table 4.9 presents a comparative analysis of several prominent models, assessed using key metrics such as EM, BLEU Score, and CodeBLEU Score. These metrics collectively provide a comprehensive evaluation of the models' syntactic and semantic accuracy in generating code from natural language descriptions.

This study primarily investigates the effectiveness of integrating Low-Rank Adap-

[10] Figure and performance data sourced from Mistral AI: https://mistral.ai/news/codestral-mamba.

| | HumanEval | MBPP | Spider | CruxE | HumanEval C++ | HumanEval Java | HumanEval JavaScript | HumanEval Bash |
|---|---|---|---|---|---|---|---|---|
| CodeGemma-1.1 7B | 61.0% | 67.7% | 46.3% | 50.4% | 49.1% | 41.8% | 52.2% | 9.4% |
| CodeLlama 7B | 31.1% | 48.2% | 29.3% | 50.1% | 31.7% | 29.7% | 31.7% | 11.4% |
| DeepSeek v1.5 7B | 65.9% | 70.8% | **61.2%** | 55.5% | 59.0% | **62.7%** | 60.9% | **33.5%** |
| Codestral Mamba (7B) | **75.0%** | **68.5%** | 58.8% | **57.8%** | **59.8%** | 57.0% | **61.5%** | 31.1% |
| Codestral (22B) | **81.1%** | **78.2%** | **63.5%** | 51.3% | **65.2%** | **63.3%** | | **42.4%** |
| CodeLlama 34B | 43.3% | 75.1% | 50.8% | 55.2% | 51.6% | 57.0% | 59.0% | 29.7% |

Code

Figure 4.3: Benchmark performance comparison of various models.

tation (LoRA), a Parameter-Efficient Fine-Tuning technique, within the Codestral Mamba model. This implementation marks the first application of LoRA to the Codestral Mamba framework, necessitating a thorough evaluation of its performance relative to established models. Specifically, a subset of parameters within the LoRA matrices is allocated to learning from the CONCODE/CodeXGLUE dataset, enabling the model to adapt to the datasets nuances while preserving the broader knowledge embedded in the pre-trained Codestral Mamba model.

A key distinction of our experiment is that, unlike models exclusively trained on the CONCODE dataset, Codestral Mamba (LoRA) remains a general model that retains its prior knowledge and inherent properties. The LoRA fine-tuning process facilitates adaptation to the dataset without overwriting previously acquired capabilities, thereby leveraging both general and dataset-specific knowledge to enhance code generation performance.

Moreover, the fine-tuning process for Codestral Mamba (LoRA) demonstrated notable computational efficiency during training, completing 200 epochs in just 1.5 hours . This rapid training time highlights the optimized implementation and computational effectiveness of the model, allowing for efficient fine-tuning while maintaining high performance.

Table 4.9 presents the results of our comparative analysis. Performance metrics for all models, except Codestral Mamba, are sourced from prior works [Lu et al., 2021; Chakraborty et al., 2022], whereas the performance of Codestral Mamba (LoRA) is derived from our experimental evaluation.

As previously noted, the primary objective of this experiment is to demonstrate that incorporating LoRA matrices into the Mamba2 model enables it to effectively learn from new data. To validate this objective, it was essential to employ a dataset where the baseline Codestral Mamba model exhibits suboptimal performancethat is, a dataset in which the nature of the contextual instructions or prompts does not align well with the training data, resulting in inadequate code generation.

In this context, the CodeXGlue database was selected due to its unique structure in pairing natural language text with corresponding code. The preparation of these

| 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 | 0.0 | 0.05 | 18.99 |
| Codestral Mamba (LoRA) | 286 M | 22.00 | 40.00 | 41.00 |

Table 4.9: Performance comparison of text-to-code generation models on the CONCODE/CodeXGLUE dataset.

pairs challenges language models that have been conditioned to produce more elaborate responses, often leading to discrepancies when generating code. This misalignment is quantifiably assessed using metrics such as BLEU, EM, and CodeBLEU. Table 4.9 illustrates the performance of Codestral Mamba 7B on this dataset, thereby corroborating its limited performance under these conditions.

In contrast, our findings indicate that the integration of LoRA matrices enables the model to rapidly assimilate new contextual information, leading to significant improvements in performance within just a few training epochs. This enhancement is largely attributable to the strong general knowledge embedded within the base model, which, when augmented with LoRA, demonstrates a marked capacity for adapting to and accurately reproducing desired code outputs.

The results in Table 4.9 highlight a significant performance improvement when incorporating LoRA into the Codestral Mamba model. Notably, the Codestral Mamba (LoRA) model achieves a BLEU score of 40.00 and an EM score of 22.00, demonstrating a substantial increase compared to the base Codestral Mamba model, which recorded near-zero values for these metrics. This improvement underscores the efficacy of LoRA fine-tuning in enhancing the model's ability to generate syntactically and semantically accurate code.

#### Model Performance Across TestCase2Code Dataset

The TestCase2Code dataset is an invaluable resource for advancing research in automated test case generation. Developed using real project values at the Decsis company, this dataset addresses the critical need for Pytest-based functional test cases in real-world scenarios. During the study, no existing database containing both manual and automated functional test cases was found, highlighting the uniqueness and significance of TestCase2Code. By integrating manual test cases with their automated counterparts, it enables the exploration of techniques for generating functional, contextually relevant test cases that align with the complexities of real-world software development. This foundation not only facilitates further research in automated software testing but also bridges the gap between manual and automated testing methodologies.

Table 4.10 presents a comparative analysis of the performance metrics for the Codestral Mamba model, contrasting its baseline with the configuration enhanced through LoRA fine-tuning. The metrics offer a thorough evaluation of the model's baseline capability to generate code that is both syntactically and semantically accurate.

The baseline model, evaluated on the TestCase2Code dataset, demonstrates a Pass@1 metric of 100%, indicating its proficiency in generating syntactically correct code. However, other metrics, such as n-gram (4.82), w-ngram (11.8), SM (39.5), DM (51.4), and CodeBLEU (26.9), reveal its limitations in practical application.

In contrast, the LoRA fine-tuned configuration of the Codestral Mamba model exhibits significant improvements across all evaluated metrics. Specifically, the n-gram score increases to 56.2, w-ngram to 67.3, SM to 91.0, DM to 84.3, and CodeBLEU to 74.7, while maintaining a Pass@1 metric of 100%. These enhancements underscore the effectiveness of LoRA fine-tuning in generating code that is not only syntactically correct but also semantically accurate, thereby addressing the shortcomings of the baseline model.

Furthermore, the training process was remarkably efficient, achieving 200 epochs in just 20 minutes with the Mamba + LoRA configuration. Such efficiency allows for rapid experimentation and fine-tuning, significantly reducing the time required to achieve optimal model performance.

| **Model** | **n-gram %** | **w-ngram %** | **SM %** | **DM %** | **CodeBLEU %** | **Pass@1 %** |
|---|---|---|---|---|---|---|
| Codestral Mamba | 4.82 | 11.8 | 39.5 | 51.4 | 26.9 | 100 |
| Codestral Mamba (LoRA) | 56.2 | 67.3 | 91.0 | 84.3 | 74.7 | 100 |

Table 4.10: Comparison of performance metrics for the Codestral Mamba model in its baseline and LoRA fine-tuned configurations, evaluated on the TestCase2Code dataset.

This accelerated training process not only facilitates faster iterations and fine-tuning but also enables experiment with multiple configurations in a significantly reduced time frame. Additionally, the efficiency of the training pipeline allows for seamless adaptation to new datasets, enabling the model to be retrained on diverse data sources with minimal computational overhead. This flexibility supports the development of a Low-Rank Adaptation (LoRA) library, where fine-tuned versions of the model can be stored and reused for various tasks. As a result, the approach enhances model performance, improves adaptability, and streamlines workflows, making it highly suitable for real-world applications that demand quick deployment, continuous optimization, and task-specific customization.

## 4.6 Results Interpretation

A thorough evaluation of the model's performance across different datasets is essential to understanding its effectiveness in code generation tasks. This section provides a detailed analysis of the results obtained from the evaluation.

The analysis follows a structured approach by evaluating the performance of the model on two benchmark datasets: CONCODE/CodeXGLUE and TestCase2Code.

Furthermore, to contextualize the observed performance trends, we compare the model's results against other state of the art models across multiple tasks. The benchmark comparison presented in Figure 4.3 provides a broader perspective on the model's strengths and limitations relative to competing approaches. This comparative analysis highlights the relative positioning of the Codestral Mamba model within the broader landscape of code generation research.

The following subsections present a detailed discussion of the results obtained for each dataset. The first subsection analyzes the performance on the CONCODE/-CodeXGLUE dataset. The subsequent subsection examines the TestCase2Code dataset results, emphasizing the model's ability to generate functionally accurate and contextually relevant code from structured test case descriptions.

### 4.6.1 Analysis of CONCODE/CodeXGLUE Dataset Results

The results presented in Table 4.9 highlight the substantial impact of integrating Low-Rank Adaptation (LoRA) into the Codestral Mamba model. Without fine-tuning, the baseline Codestral Mamba model exhibits extremely low performance, with an exact match (EM) score of 0.0 and a BLEU score of just 0.05. These values indicate that the base model struggles to generate syntactically and semantically relevant code when applied to the CONCODE/CodeXGLUE dataset. Furthermore, its CodeBLEU score of 18.99 further underscores its limitations in producing functionally meaningful code.

In contrast, when fine-tuned using LoRA, the Codestral Mamba model demonstrates a dramatic improvement across all evaluation metrics. Specifically, the enhanced model achieves an EM score of 22.00, a BLEU score of 40.00, and a CodeBLEU score of 41.00. These results indicate that LoRA enables the model to generate significantly more accurate and coherent code representations. The improvement in BLEU and CodeBLEU scores confirms that the fine-tuned model is better at preserving syntactic and semantic integrity, producing code outputs that align more closely with reference solutions.

From a comparative perspective, the performance of Codestral Mamba (LoRA) is highly competitive with state-of-the-art models. Its EM score of 22.00 closely aligns with CodeT5-base (22.30) and NatGen (22.25), while its BLEU score of 40.00 surpasses most other models in the benchmark, including CodeGPT-adapted (32.79) and GPT-2 (25.37). Although its CodeBLEU score (41.00) falls slightly behind CodeT5-base (43.20) and NatGen (43.73), it remains within a close margin, demonstrating strong generalization capabilities.

These findings underscore the effectiveness of LoRA as a parameter-efficient fine-tuning approach, enabling substantial performance gains without the need for full model retraining. By leveraging LoRA, the model retains its broader pre-trained knowledge while adapting efficiently to the CONCODE/CodeXGLUE dataset. This balance between computational efficiency and performance makes Codestral Mamba (LoRA) particularly well-suited for real-world applications such as automated test case generation, where both accuracy and efficiency are critical.

### 4.6.2 Analysis of TestCase2Code Dataset Results

The evaluation of the Codestral Mamba model on the TestCase2Code dataset reveals significant insights into its code generation capabilities, particularly when enhanced with Low-Rank Adaptation fine-tuning. This section delves into the detailed analysis and interpretation of the results, highlighting the model's strengths and the impact of the fine-tuning process on its performance.

Table 4.10 presents the performance metrics of the Codestral Mamba model model with and without LoRA fine-tuning. The model demonstrates a strong ability to generate syntactically correct code, achieving a perfect 100% Pass@1 score. This indicates that every generated code snippet is free of syntax errors, establishing a solid baseline for code generation.

#### Impact of Fine-Tuning with LoRA

Applying fine-tuning with LoRA, as shown in Table 4.10, leads to notable improvements across all evaluation metrics. The n-gram and w-ngram scores rise significantly, indicating better lexical and structural alignment with the reference code. Additionally, both syntax match and dataflow match scores experience substantial gains, reflecting enhanced syntactic and semantic coherence in the generated code.

A particularly striking result is the dramatic increase in the CodeBLEU score, which rises from 26.9 (baseline) to 74.7 following LoRA fine-tuning. Unlike traditional metrics such as exact match or BLEU, which primarily assess surface-level similarities, CodeBLEU provides a more comprehensive evaluation by integrating syntactic and semantic correctness. This highlights the effectiveness of LoRA in refining the models ability to generate high-quality test cases that align closely with real-world software requirements.

The comparative analysis of the Codestral Mamba models performance before and after fine-tuning underscores LoRA's role in significantly enhancing code generation capabilities. As shown in Table 4.10, the baseline model, trained without LoRA, achieves moderate performance across key metrics. However, incorporating LoRA leads to considerable improvements, demonstrating its potential for scalable and efficient adaptation in automated software testing and development workflows.

#### From Syntactic Validation to Functional Adaptation

The 100% Pass@1 score observed in both the baseline Mamba model and its fine-tuned counterpart confirms their ability to generate syntactically valid code. However, Pass@1 exclusively evaluates structural correctness, without providing insights into functional accuracy or domain relevance. While the baseline model ensures correctness at a syntactic level, this does not inherently guarantee its suitability for specific project requirements. In contrast, the notable improvement in CodeBLEU scores following fine-tuning suggests that, beyond maintaining syntactic integrity, the model achieves greater alignment with project-specific objectives. This enhancement stems from its ability to learn from domain-specific data, refining its responses to better fit the intended application context.

The results underscore the significant impact of LoRA-based fine-tuning on the performance of the Codestral Mamba model. By reinforcing both syntactic correctness and functional accuracy, this approach enables a more tailored adaptation to real-world software development needs. The observed improvements highlight the necessity of employing complementary evaluation metrics such as CodeBLEU, which, unlike Pass@1, offer a more comprehensive assessment of both structural and semantic quality. These findings emphasize the potential of this fine-tuning strategy as a key technique for enhancing the practical applicability of automated code generation models.

## 4.7 Practical Implications

The findings from this study present significant practical applications and implications for the field of software development and automated testing. The integration of advanced language models, such as the Codestral Mamba model enhanced with Low-Rank Adaptation, offers tangible advancements in generating high-quality, context-aware test scripts. This section explores the real-world applications, benefits, and challenges of implementing these findings, highlighting how they can enhance software quality, efficiency, and development workflows.

### 4.7.1 Enhanced Automated Testing

One of the most immediate practical applications of this research is the enhancement of automated testing processes. Traditional manual testing methods are time-consuming and prone to human error, often leading to incomplete test coverage and delayed software releases. By employing the Codestral Mamba model with LoRA fine-tuning, developers can automate the generation of test cases, ensuring comprehensive coverage of software functionalities. This automation not only reduces the manual effort required for test case creation but also accelerates the testing cycle, allowing for more frequent and thorough testing.

A key contribution of this study is the creation of the TestCase2Code dataset, a pioneering resource addressing a critical gap in the field of software testing. To date, no publicly available database of both manual and automated functional test cases has been identified. This dataset was structured using real project data from Decsis, ensuring its relevance and applicability to industry practices. By offering a well-defined repository of test cases, TestCase2Code serves as a foundation for future advancements in automated testing, facilitating improved model training, evaluation, and benchmarking within real-world software development environments.

The model's ability to generate syntactically and semantically accurate code, as evidenced by high CodeBLEU scores, ensures that the test cases are both valid and relevant to the software's intended behavior. This leads to more reliable test results and earlier detection of defects, ultimately improving the overall quality of the software.

Furthermore, with this approach, we can envision a repository where the LoRA matrices are encapsulated by project. This would allow for a modular and scalable solution, enabling different projects to maintain their own fine-tuned matrices tailored to their specific needs. Such a repository would facilitate better organization and management of fine-tuned models, making it easier to adapt and reuse these models across various projects within an organization. This encapsulation would also streamline the process of updating and maintaining the models as software requirements evolve, ensuring that the generated test cases remain relevant and effective over time.

### 4.7.2 Enhancing Software Quality Engineering Efficiency

The adoption of automated test case generation can significantly enhance the efficiency of software quality engineering teams. By reducing the effort required for quality specialists to design automated test cases from scratch, the approach allows them to focus on refining test strategies, analyzing results, and ensuring comprehensive coverage.

Additionally, fine-tuning capabilities allow the model to adapt to specific project requirements, generating test cases that align with the unique context of the software under evaluation. While automation reduces the time spent designing scripts, quality engineers still play a crucial role in validating, refining, and maintaining these test cases to ensure their effectiveness. This balance between automation and human oversight leads to more efficient testing workflows, ultimately improving software reliability and accelerating the development lifecycle.

Furthermore, this research leveraged LoRA fine-tuning to develop a customized chatbot, Codestral Mamba_QA AI Chatbot, specifically designed to meet the needs of the company. This chatbot significantly enhances response accuracy by integrating project-specific data, automates routine inquiries to optimize workload distribution, and seamlessly adapts to the companys terminology and operational requirements. A crucial advantage of this approach is its strong emphasis on data security, as the chatbot operates within an internal infrastructure, eliminating reliance on third-party solutions and safeguarding sensitive information. Additionally, Codestral Mamba_QA AI Chatbot is designed for continuous learning, evolving through user interactions to improve performance over time. Its ability to provide real-time assistance ensures uninterrupted support for employees, thereby streamlining operations and enhancing efficiency.

### 4.7.3 Cost and Time Efficiency

The implementation of automated test case generation using the Codestral Mamba model can lead to substantial cost and time savings. Automated testing reduces the need for extensive manual testing resources, lowering the overall cost of the testing process. Additionally, the model's efficiency in generating high-quality test cases minimizes the time required for test case creation and maintenance, allowing for faster development cycles.

The use of LoRA fine-tuning further enhances cost efficiency by enabling the model to adapt to new tasks with minimal computational overhead. This adaptability ensures that the model remains effective in generating relevant test cases as the software evolves, without the need for extensive retraining or resource allocation.

### 4.7.4 Scalability and Adaptability

The Codestral Mamba model's ability to generate test cases from natural language descriptions makes it highly scalable and adaptable to various software development environments. Whether used in small development teams or large-scale enterprises, the model can be integrated into existing workflows to enhance testing processes. Its adaptability to different programming languages and frameworks further expands its applicability, making it a versatile tool for developers working in diverse technological ecosystems.

### 4.7.5 Continuous Integration and Deployment

In the context of continuous integration and deployment (CI/CD) pipelines, the automated generation of test cases can significantly enhance the reliability and speed of software releases. By integrating the Codestral Mamba model into CI/CD workflows, developers can ensure that comprehensive test suites are automatically generated and executed with each code change. This continuous testing approach helps identify and address issues early in the development cycle, reducing the risk of defects in production and enabling more frequent and reliable software updates.

### 4.7.6 Benefits and Challenges

**Benefits:**

- Consistency and Reliability: Automated test case generation ensures consistent and reliable testing, reducing the variability and errors associated with manual testing.
- Early Defect Detection: By generating and executing tests early in the development cycle, defects can be identified and addressed promptly, reducing the cost and effort of fixing issues later.
- Comprehensive Coverage: The model's ability to generate a wide range of test cases ensures comprehensive coverage of software functionalities, reducing the risk of undetected bugs.
- Enhanced Workflow Automation: The development of the Codestral Mamba_-QA AI Chatbot demonstrates how AI-driven solutions can streamline operations, ensuring seamless adaptation to company-specific requirements.

**Challenges:**

- Initial Setup and Integration: Implementing automated testing requires an initial investment in setting up and integrating the model into existing workflows. This may involve technical challenges and a learning curve for development teams.
- Maintenance and Updates: As software evolves, the model must be continually fine-tuned and updated to generate relevant test cases. This ongoing maintenance can be resource-intensive.
- Ensuring Data Security: While the chatbot solution provides enhanced security by operating within an internal infrastructure, organizations must maintain rigorous data management practices to prevent potential vulnerabilities.

## 4.8 Limitations of the Study

While this study presents significant advancements in the field of automated test case generation using advanced language models, it is essential to acknowledge several limitations that may influence the interpretation and application of the findings. Identifying these limitations provides a clearer understanding of the research's scope and offers opportunities for future improvements.

### 4.8.1 Data Dependency and Generalization

The Codestral Mamba model was fine-tuned using the TestCase2Code dataset, which, while comprehensive, may not fully encapsulate the diversity and complexity of the software project. The dataset was constructed by selecting specific .jsx files associated with each test case, providing relevant project information but not necessarily capturing all critical details.

To address this limitation, future research could explore providing the model with access to the entire project repository on platforms like GitLab. By giving the model complete information about the project, including all relevant files and context, we can facilitate better learning and training. This approach would enable the model to understand the broader context of the software, potentially improving its ability to generate more accurate and contextually relevant test cases.

Future studies should incorporate more comprehensive datasets that include complete project repositories. This would help in evaluating the model's robustness and generalization capabilities across diverse software development environments. By providing the model with a more holistic view of the project.

It is important to note that the Codestral Mamba model is a general code model tested across several databases, not tailored to a specific datasets like CONCODE/-CodeXGLUE or TestCase2Code. The use of the CONCODE/CodeXGLUE dataset in this study serves to demonstrate the effectiveness of applying Parameter-Efficient Fine-tuning techniques, specifically LoRA matrices, in adapting the model to specific tasks. This highlights the model's versatility and potential for application in a wide range of software testing scenarios.

### 4.8.2 Model Fine-Tuning and Computational Resources

The fine-tuning process using Low-Rank Adaptation requires careful configuration and domain-specific data to achieve optimal performance. While LoRA reduces computational overhead compared to full fine-tuning, it still demands significant resources for training and inference, particularly for large models like Codestral Mamba. Smaller development teams or organizations with limited computational resources may find it challenging to implement and maintain such models.

Future research could investigate adaptive fine-tuning strategies that dynamically adjust model complexity based on available computational resources.

### 4.8.3 Evaluation Metrics and Real-World Applicability

The evaluation metrics used in this study, such as CodeBLEU and Pass@1, provide a comprehensive assessment of the model's performance in generating syntactically and semantically accurate code. However, these metrics may not fully capture the nuances of real-world software testing, where factors like code maintainability, readability, and integration with existing systems are crucial. The focus on automated metrics may overlook the practical challenges faced in integrating generated test cases into workflows.

Future studies should consider incorporating additional evaluation metrics that assess the practical usability and maintainability of the generated test cases. Conducting user studies with software developers and quality assurance to gather feedback on the generated test cases real-world applicability could provide valuable insights into the model's practical effectiveness.

### 4.8.4 Adaptation to Evolving Software Requirements

Software development is a dynamic process, with requirements and codebases evolving rapidly. The model's ability to adapt to these changes and generate relevant test cases continuously is a critical aspect of its practical utility. While Low-Rank Adaptation fine-tuning enables the model to adapt to new tasks with minimal overhead, the need for ongoing maintenance and updates to keep the model aligned with evolving software requirements poses a challenge.

Developing automated tools and frameworks that facilitate the continuous fine-tuning and updating of the model could address this limitation. Research into self-adapting models that can learn from evolving codebases and requirements without extensive manual intervention could further enhance the model's practical applicability.

### 4.8.5 Integration with Existing Development Workflows

Integrating the Codestral Mamba model into existing software development workflows, particularly in continuous integration and deployment (CI/CD) pipelines, presents technical and organizational challenges. The model's integration requires compatibility with various development tools, frameworks, and practices, which may vary across different organizations. Ensuring seamless integration and interoperability is crucial for realizing the model's full potential.

Future research should focus on developing standardized integration frameworks and best practices for incorporating advanced language models into CI/CD pipelines. Collaboration with industry partners to pilot these integrations in real-world development environments could provide valuable insights and drive broader adoption.

In conclusion, while this study presents significant advancements in automated test case generation, acknowledging and addressing these limitations is crucial for advancing the field further. By exploring potential solutions and recommendations, future research can build upon these findings to enhance the practical applicability and effectiveness of advanced language models in software development and testing.

## 4.9 Experimental Evaluation and AI-Driven Test Automation

To evaluate the trained model, the **Codestral Mamba_QA AI Chatbot**, a chatbot web service (Virtual Assistant), was developed for testing. The assessment used manual test cases from the *TestCase2Code* database, excluding those from training, along with newly created cases and general instructions.

This section presents the experimental results, focusing on the models performance across various configurations. The analysis examines its ability to generate structured test cases, respond to diverse prompts, and enhance test automation through LoRA scaling.

A key outcome of this thesis is the development of the **Codestral Mamba_QA AI Chatbot**, a fine-tuned intelligent assistant tailored to the companys specific needs. By leveraging project-specific data, it improves response accuracy, seamlessly integrates domain-specific terminology, and optimizes workload distribution. Its deployment within an internal infrastructure enhances data security by eliminating reliance on external solutions and protecting sensitive information.

Another notable advantage of this approach is the flexibility of hyperparameter adjustments. Each parameter can be modified dynamically without the need to reload the model or allocate additional resources, allowing for efficient fine-tuning and adaptation to evolving project demands. Additionally, this framework enhances scalability, enabling seamless integration into continuous testing pipelines while maintaining high efficiency and cost-effectiveness.

To systematically evaluate the chatbots performance across different test case scenarios, a set of key parameters and hyperparameters was defined and adjusted dynamically throughout the experiments:

- **System Prompt**: A predefined instruction that directs the models behavior when generating responses. If configured, it ensures that outputs align with best practices in test case generation, Pytest conventions, and project requirements. If no prompt is set, the chatbot relies on default behaviors, potentially generating broader or less structured responses.

- **Temperature**: A numerical value that controls the randomness of the models output. Lower values (e.g., 0.0) result in deterministic and structured responses, whereas higher values increase diversity, allowing for more varied test case formulations.

- **Factor Scale LoRA**: A scaling factor that determines the extent to which the LoRA fine-tuning influences the models responses. Higher values increase the integration of domain-specific adaptations, whereas lower values retain more general language model characteristics.

- **System Input**: The user-provided input that serves as the basis for generating a response. This input may or may not correspond to a specific test case; it could be a request for test case generation or any other question.
- **System Response**: The output produced by the chatbot, which varies depending on the input and configurations. If the request pertains to test case generation , the response typically includes structured Pytest code. However, depending on the query and the configuration the chatbot provides the answer. When applicable, the chatbot supplements its response with an explanation of the generated test case, detailing the logic, assertions, and methodology behind it. This feature enhances interpretability and ensures alignment with expected testing standards.

These parameters and hyperparameters collectively define the experimental conditions under which the **Codestral Mamba_QA AI Chatbot** was evaluated. By analyzing their impact, this study provides valuable insights into optimizing AI-driven test case generation and adapting automation frameworks to evolving software validation requirements.

### 4.9.1 Evaluation of the Successful Login Query

This section presents an experimental scenario in which the Codestral Mamba model, configured with a LoRA scale factor set to 0, processes a user query related to a successful login. The systems behavior under these conditions is illustrated in Figure 4.4.

Unlike structured test cases stored in the TestCase2Code database, this instance does not conform to a predefined format. Instead, the system processes the input as a general query, interpreting it as a request for conceptual information rather than a formal test case specification. Consequently, instead of generating executable Pytest code, the model provides a detailed explanation of the login process, covering authentication mechanisms, authorization protocols, user access control, and security considerations.

This outcome underscores the models response mechanism when presented with an unstructured prompt and default hyperparameter settings.This behavior is directly influenced by the following experimental conditions:

- **System Prompt**: As no specific prompt was set, the model defaulted to an explanatory response rather than producing a structured test case.
- **Temperature**: Set to 0.0, ensuring a deterministic response that consistently prioritizes factual explanations over variation or alternative test case formats.
- **Factor Scale LoRA**: Configured to 0, meaning no fine-tuning adjustments influenced the models output, leading it to rely solely on its base training rather than aligning with domain-specific test case structures.

- **System Input**: The entered phrase, "successful login," was interpreted as an open-ended request rather than a directive to generate test code, affecting the models response format.
- **System Response**: Given the absence of a structured input format and prompt constraints, the output consisted of an explanatory breakdown of the login process rather than an executable test case.

The experiment demonstrates the adaptability of the Codestral Mamba model in processing diverse input formats. It also highlights the necessity of defining precise system prompts and fine-tuning hyperparameters to achieve the desired outcome whether generating structured test cases or providing explanatory content. These findings contribute to a deeper understanding of AI-driven test automation and reinforce the importance of controlled parameter adjustments in optimizing system performance. This outcome highlights the Codestral Mamba models default behavior when presented with an unstructured input, demonstrating its tendency to provide conceptual explanations in the absence of a structured test case format.

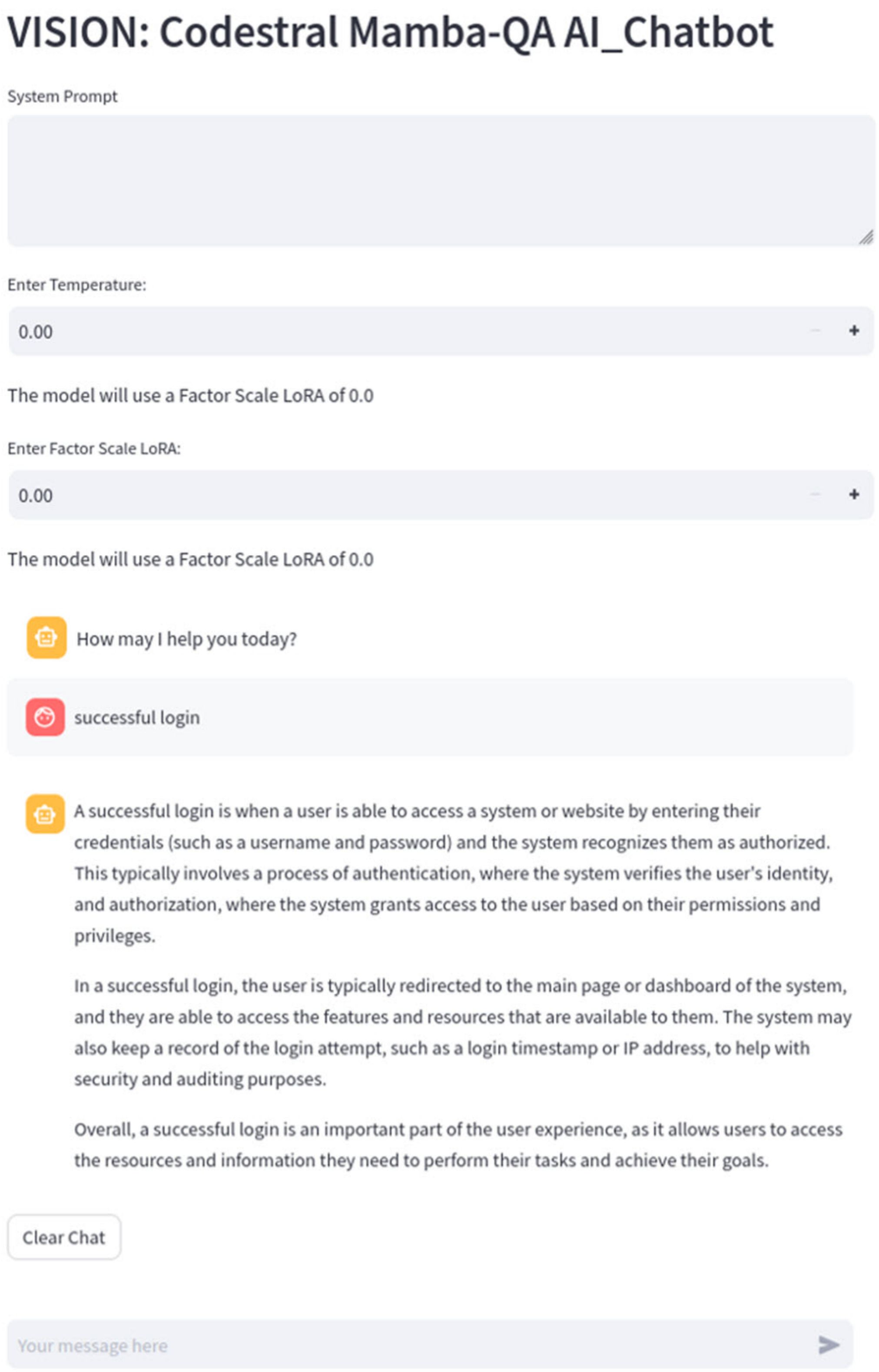



Figure 4.4: Successful login test case execution using Codestral Mamba without LoRA.

### 4.9.2 LoRA-Automated and Manual Test Case Generation

This section presents an experimental scenario in which the Codestral Mamba model, configured with a LoRA scaling factor of 2, processes a user query related to a successful login test case. The systems behavior under these conditions is illustrated in Figure 4.5.

Unlike cases where the model defaulted to an explanatory response, the increased LoRA scaling factor influences the models output significantly. Instead of solely providing a conceptual breakdown, the model autonomously generates both a structured manual test case and an executable automated test script in `pytest`. The structured test case includes explicit steps for human testers, while the generated script incorporates valid identifiers, key validation steps, and assertions ensuring redirection to the appropriate page after login.

This behavior is directly influenced by the following experimental conditions:

- **System Prompt**: As no specific prompt was set, the model relied on its fine-tuned adaptation through LoRA to infer structured test cases.
- **Temperature**: Maintained at 0.0, ensuring deterministic responses that consistently prioritize structured outputs over variation.
- **Factor Scale LoRA**: Set to 2, enabling the model to leverage fine-tuned patterns from structured test case data, resulting in an output that bridges both manual and automated testing methodologies.
- **System Input**: The phrase "successful login test case" was interpreted with sufficient context, prompting the model to generate structured test cases rather than a high-level explanation.
- **System Response**: Due to the combined effects of LoRA scaling and structured learning, the model produced a dual response: a well-defined manual test case and an executable Pytest script.

This experiment highlights the impact of LoRA scaling on structured response generation. The model demonstrates the ability to infer test case structure, seamlessly transitioning from an unstructured user query to a fully executable script. These findings reinforce the value of controlled hyperparameter adjustments in AI-driven test automation, ensuring adaptability to different testing needs while maintaining efficiency.

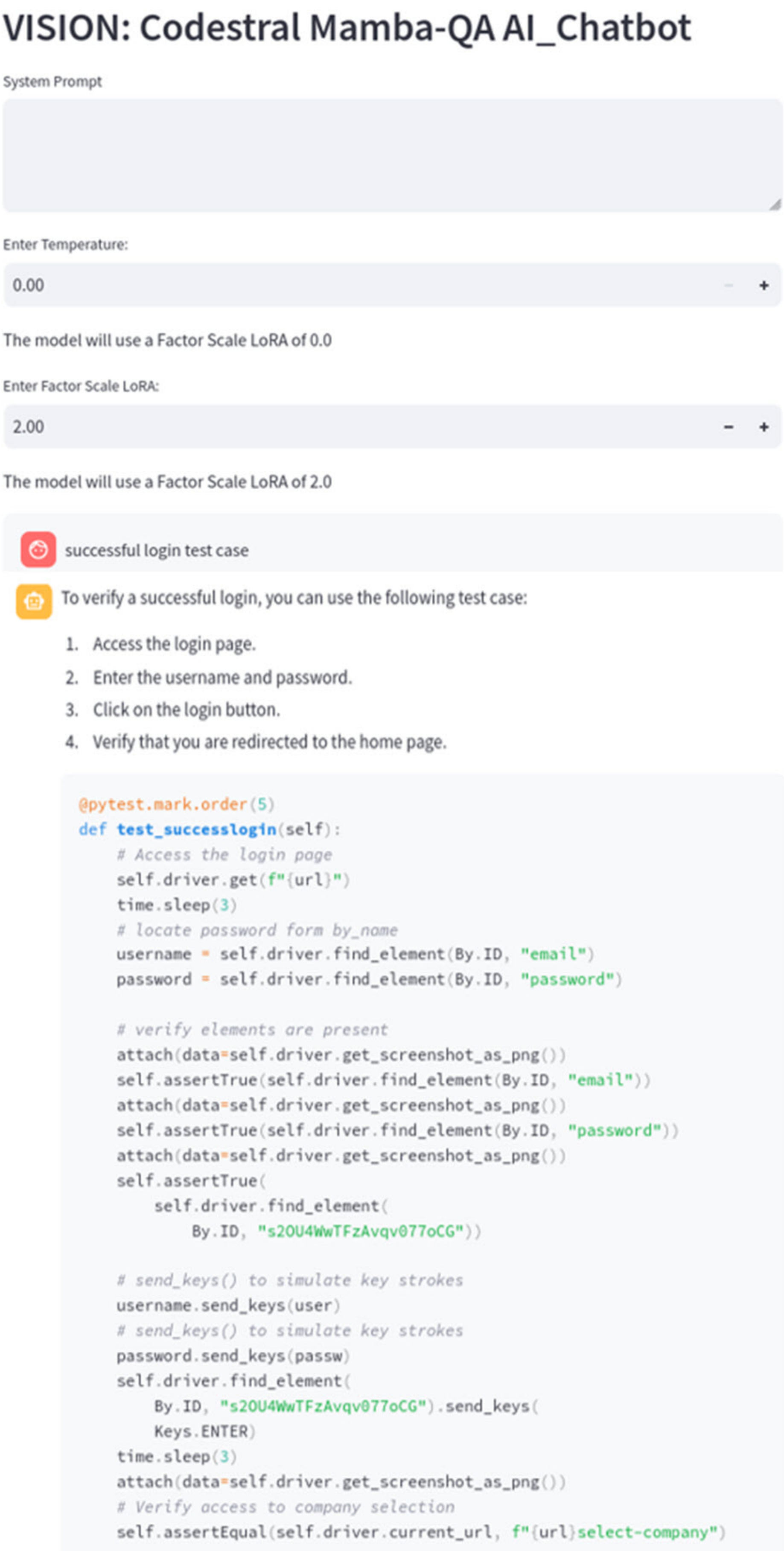



Figure 4.5: Automated test case generation by Codestral Mamba with LoRA scaling factor set to 2.

### 4.9.3 Effect of System Prompt on Test Case Generation

This evaluation examines the impact of introducing a system prompt while maintaining a **LoRA scaling factor of 0** and a **temperature setting of 0**. The systems response under these conditions is illustrated in Figure 4.6.

The following hyperparameters and parameters define the experimental conditions:

- **Temperature**: Fixed at 0, ensuring deterministic outputs without introducing variation in response generation.
- **Factor Scale LoRA**: Set to 0, preventing the model from leveraging fine-tuned enhancements and limiting its knowledge to its base capabilities.
- **System Prompt**: Explicitly instructs the model to generate a Pytest-based test case, providing structured guidance rather than relying on inference.
- **System Response**: The model successfully constructs an automated test case following best practices in *pytest*, despite the absence of contextual data such as the referenced `.jsx` file.

The generated response demonstrates that, even without LoRA-enhanced learning, the model retains its ability to generate structured test cases when provided with an explicit system prompt. The test script systematically:

1. Defines a function `test_successful_login`.
2. Utilizes a *client* fixture to simulate user interaction.
3. Sends a POST request to the login endpoint.
4. Asserts expected responses, such as a status code of 200 and a success message.
5. Verifies redirection to the correct page.

These findings highlight the inherent capability of the Codestral Mamba model to produce well-structured and functional test scripts when given clear instructions. While it does not incorporate additional learned attributes such as specific `ID` values or extended validation steps, it effectively adheres to best practices for Pytest-based automation. This reinforces the value of explicit prompting in guiding AI-driven test generation, ensuring structured and reliable outputs even in the absence of fine-tuned adaptations.

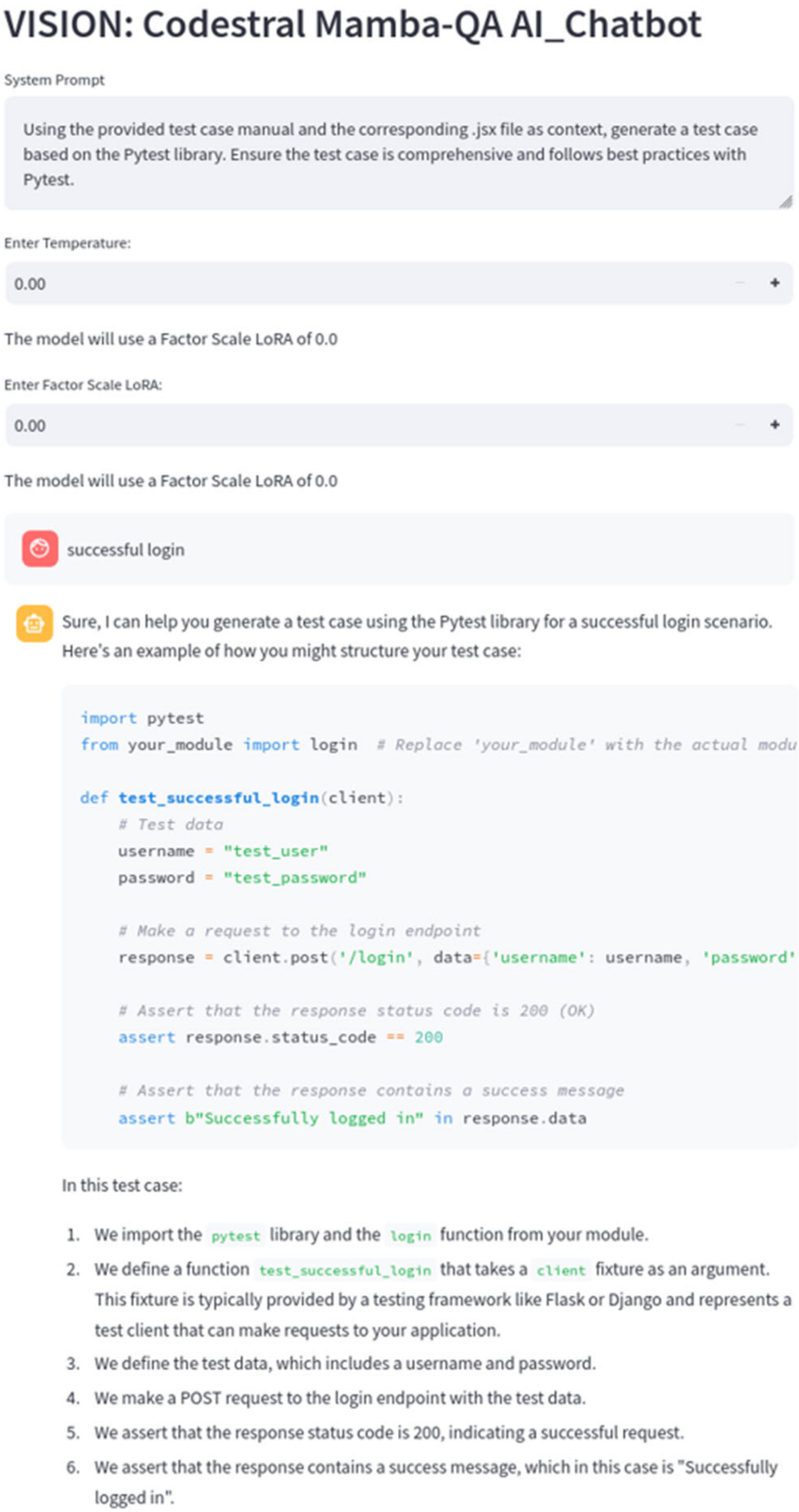



Figure 4.6: Test case generation with system prompt, LoRA scaling factor of 0, and temperature set to 0.

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

This evaluation examines the impact of adjusting temperature and LoRA scaling on the test case generation process using the Codestral Mamba model. The experiment was conducted with a **LoRA scaling factor of 3.0** and a **temperature setting of 0.50**, without an explicit system prompt or structured input file. The system's response under these conditions is illustrated in Figures 4.7 and 4.8.

The following hyperparameters and parameters define the experimental conditions:

- **Temperature**: Set to 0.50, allowing for slight variations in response generation while maintaining coherence in test case structure.
- **Factor Scale LoRA**: Adjusted to 3.0, enabling the model to leverage fine-tuned domain knowledge and produce structured, testable outputs.
- **System Prompt**: Left empty, requiring the model to infer intent based solely on input keywords.
- **System Response**: The model autonomously generates a Pytest-based automated test case, incorporating business logic and element identifiers without explicit contextual instructions.

The generated response demonstrates that **higher LoRA scaling** enhances the models ability to infer structured patterns, as evidenced by the correct assignment of element identifiers (`ID` attributes) and logical assertions verifying login success. The increase in temperature introduces controlled variability, allowing the model to explore different, yet functionally equivalent, implementations of test logic.

The test script systematically:

1. Locates input fields for `email` and `password`.
2. Asserts the presence of required elements.
3. Interacts with UI components, such as buttons and hyperlinks.
4. Captures screenshots for validation.
5. Confirms redirection to the expected home page.

These findings reinforce the adaptability of Codestral Mamba in automated test generation. The model successfully constructs meaningful test cases despite the absence of explicit instructions, relying on LoRA-enhanced knowledge and temperature-based diversification. This highlights its capability to streamline quality assurance workflows while allowing for flexible test adaptations based on project-specific requirements.

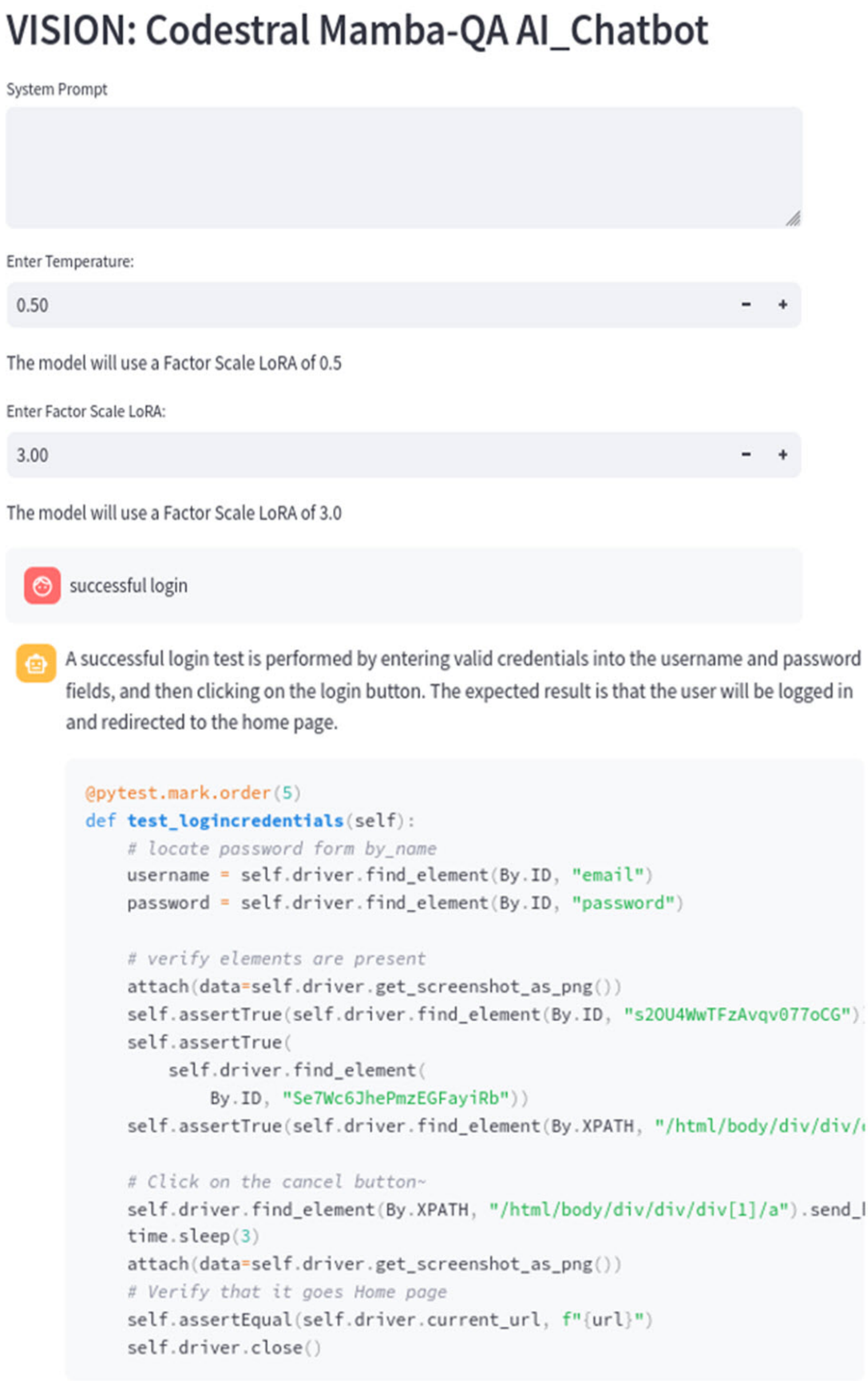



Figure 4.7: First generated test case with temperature = 0.50 and LoRA scaling factor = 3.

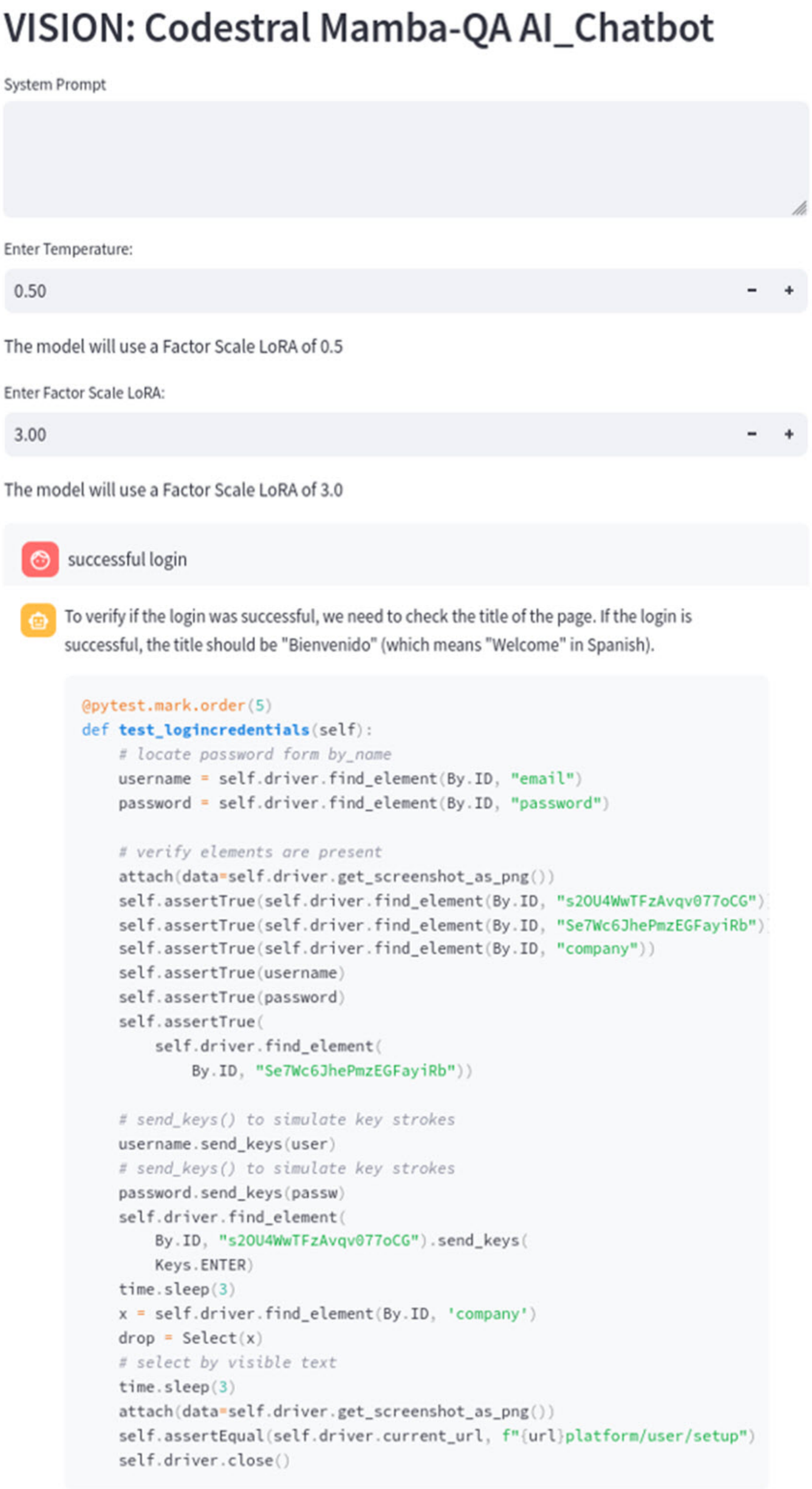



Figure 4.8: Second generated test case with temperature = 0.50 and LoRA scaling factor = 3.

### 4.9.5 Evaluation of Test Case 785

This evaluation examines the performance of the Codestral Mamba model in generating automated test cases under specific experimental conditions. Test case 785, selected from the validation set of the TestCase2Code database, was not previously seen by the model, ensuring an unbiased assessment of its capabilities. The test case pertains to verifying the presence of key fiscal tax-related texts in an annual observation table, as depicted in Figure 4.11.

The experiment was conducted with the following hyperparameter configurations:

- **Temperature**: Set to 0.00, ensuring deterministic outputs with minimal variability.
- **Factor Scale LoRA**: Adjusted to 2.0, enhancing the models ability to generalize from fine-tuned domain knowledge.
- **System Prompt**: Explicitly provided in one configuration and omitted in another to assess its influence on response generation.

Figure 4.9 illustrates the response generated when the system prompt was included, while Figure 4.10 shows the output when the prompt was absent. Notably, the model produced identical responses in both cases, demonstrating a high degree of robustness and an intrinsic understanding of test case generation, independent of prompt guidance.

The correctness of the generated test case is affirmed by its structured alignment with the expected Pytest format. The script effectively:

1. Verifies the presence of predefined texts related to fiscal reporting.
2. Uses parameterized assertions to ensure comprehensive coverage.
3. Captures screenshots for validation, reinforcing test reliability.

These results highlight the effectiveness of Codestral Mamba, augmented with LoRA scaling, in producing consistent, high-quality test cases without requiring extensive guidance. The ability to generate structured test cases autonomously, even under constrained configurations, underscores the model's potential to streamline automated testing workflows and reduce manual effort in quality assurance processes.

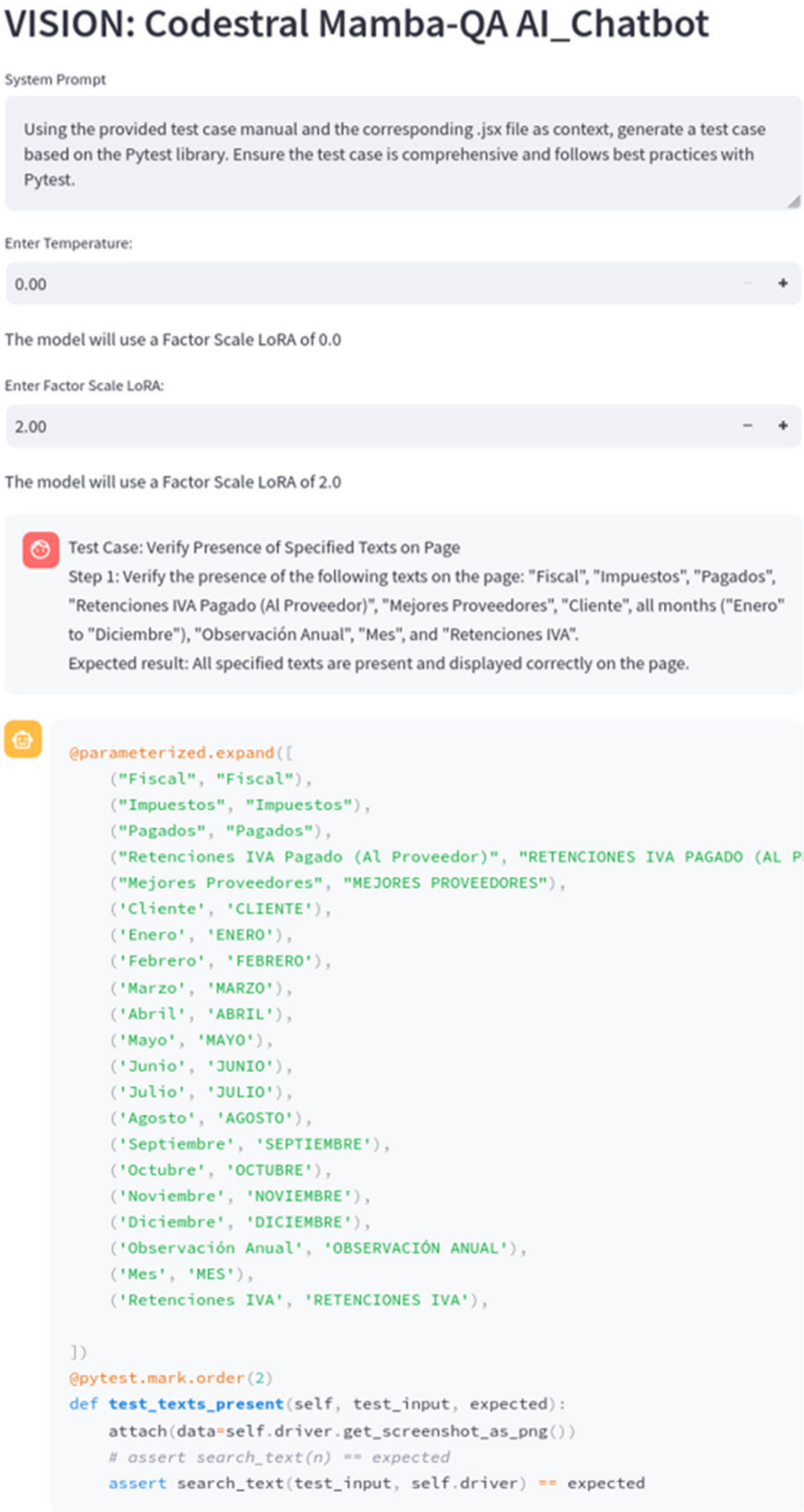



Figure 4.9: Test case 785 with system prompt, temperature = 0, and LoRA scale factor = 2.

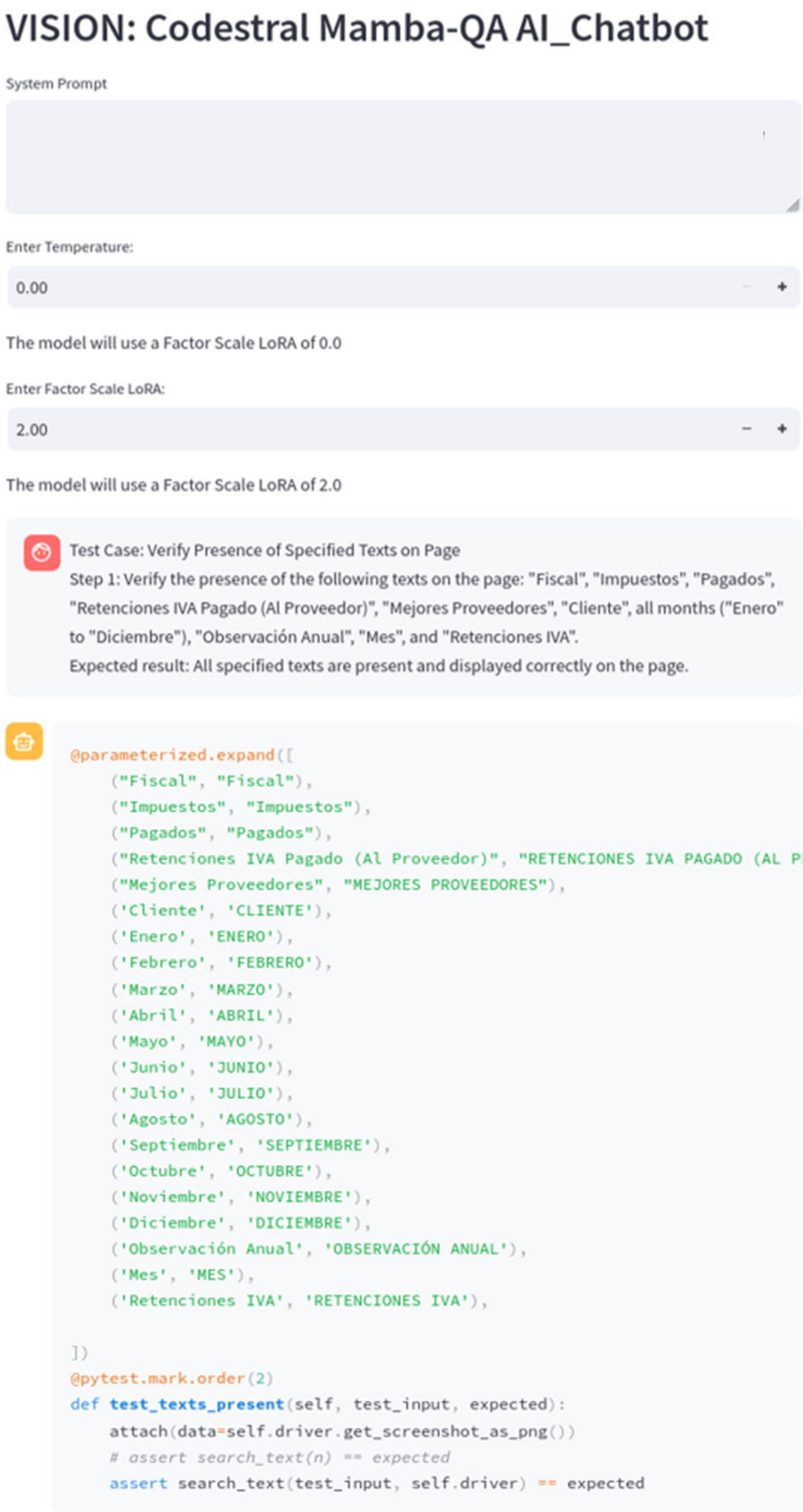



Figure 4.10: Test case 785 without system prompt, temperature = 0, and LoRA scale factor = 2.

**OBSERVACIÓN ANUAL**

| MES | IVA TRASLADADO |
|---|---|
| Enero | $ 5,449,927.11 |
| Febrero | |
| Marzo | |
| Abril | |
| Mayo | |
| Junio | |
| Julio | |
| Agosto | |
| Septiembre | |
| Octubre | |
| Noviembre | |
| Diciembre | |
| **TOTAL** | **$ 5,449,927.11** |

Figure 4.11: Context of test case 785, reviewing annual observation table texts related to fiscal taxes.

### 4.9.6 Evaluation of Test Case 794

This evaluation examines test case 794, which is part of the validation set extracted from the TestCase2Code database. As one of the 100 test cases allocated for validation, this test case had not been encountered by the model during fine-tuning. The objective of this evaluation is to assess the model's ability to correctly generate an automated test case under specific experimental conditions, as illustrated in the accompanying figures.

The test was conducted using the following parameters:

- **Temperature:** Set to 0.00, ensuring a deterministic response by minimizing output variability.
- **LoRA Scale Factor:** Configured to 2.0, leveraging fine-tuned domain-specific knowledge to enhance structured test generation.
- **System Prompt:** Explicitly provided, instructing the model to generate a Pytest-based automated test case while ensuring compliance with best practices.
- **System Response:** The model autonomously generated a structured test case that verifies numerical values in a table and confirms the correct rendering of a bar chart.

As shown in Figure 4.12, the structured input format remained consistent with the database representation, ensuring that the model received contextually rich information. The generated test case accurately verifies:

1. The presence and correctness of numeric values in a fiscal tax-related table.
2. The correct display of a bar chart representing the tax data.
3. The functional link between tabular and graphical data, ensuring consistency in representation.

These validations are crucial in quality assurance workflows, where the correctness of both tabular and graphical representations impacts data integrity and user experience. Figure 4.13 illustrates the relevant fiscal section containing the bar graph under analysis.

The accuracy of the generated test case, even for previously unseen validation data, underscores the robustness and adaptability of the Codestral Mamba model. The experiment demonstrates that:

- The LoRA enhanced model effectively recognizes structured patterns and dependencies in test case design.

- The temperature setting of 0.00 guarantees deterministic behavior, making the model suitable for predictable, repeatable test generation.
- The system prompt had no significant impact on output accuracy, reaffirming the models intrinsic ability to infer test logic from structured inputs.

These findings reinforce the practical applicability of the Codestral Mamba model in automated software testing, particularly in scenarios requiring structured, domain-specific test case generation.

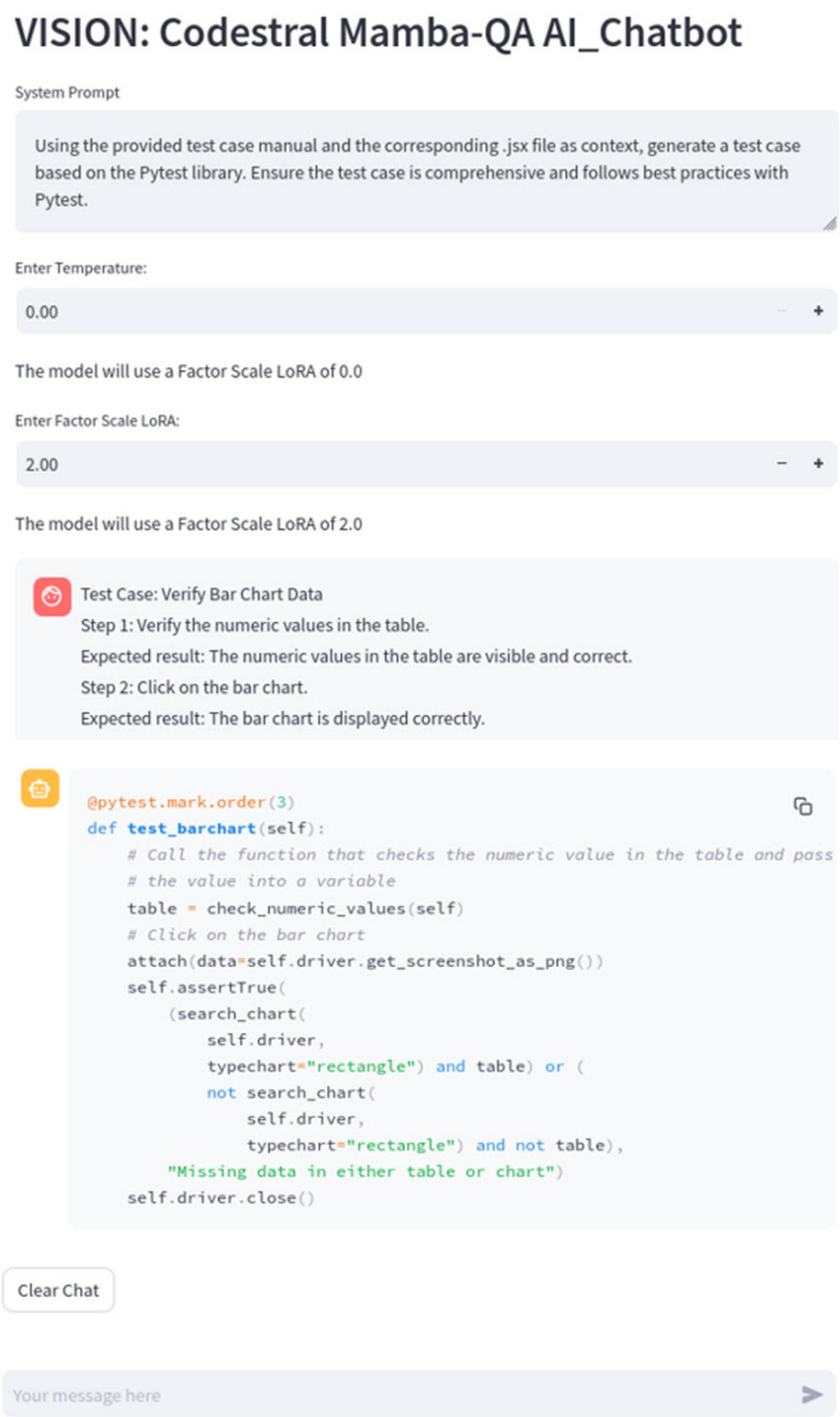



Figure 4.12: Test case 794 with system prompt, temperature = 0, and LoRA scale factor = 2.

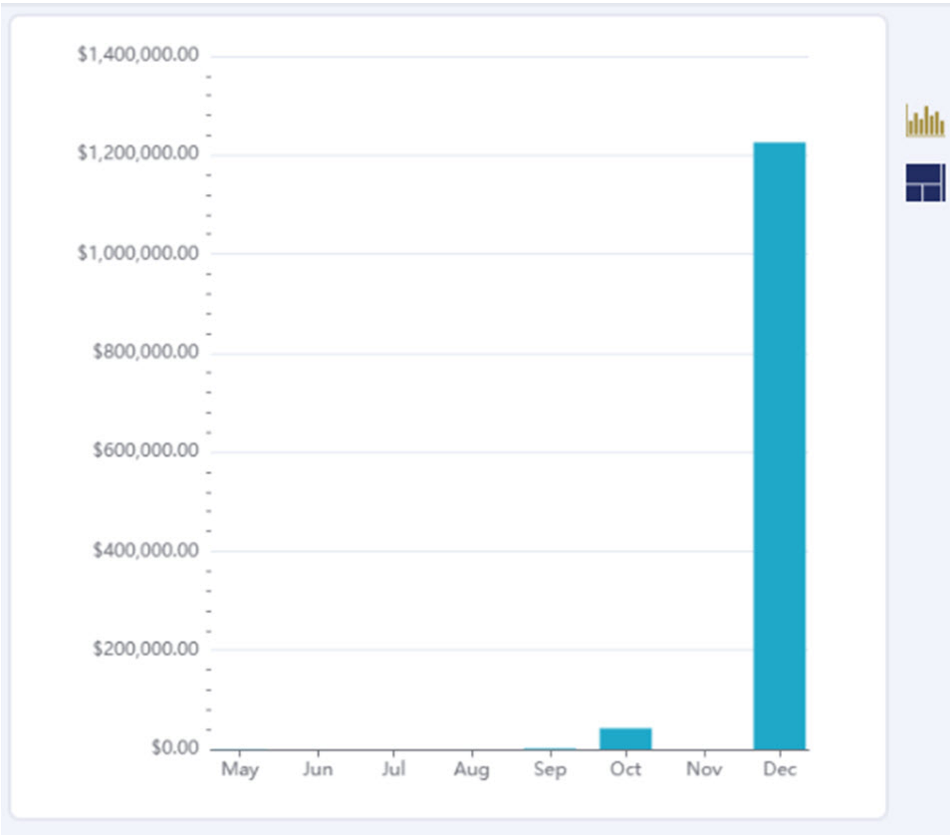



Figure 4.13: Context of test case 794, identifying the bar graph of fiscal section taxes paid.

### 4.9.7 Enhancing Test Coverage with LoRA

This evaluation investigates the impact of increasing the LoRA scale factor on the performance and output of the Codestral Mamba model, specifically focusing on test case generation for improving test coverage. The analysis is based on the document provided in Figure 4.9.7, which was generated with a temperature of 0 and a LoRA scale factor of 3.

By increasing the LoRA scale factor to 3, the model is expected to produce responses that are more focused on the specific information and context of the project. This adjustment aims to enhance the model's ability to generate detailed and relevant test cases, thereby improving overall test coverage.

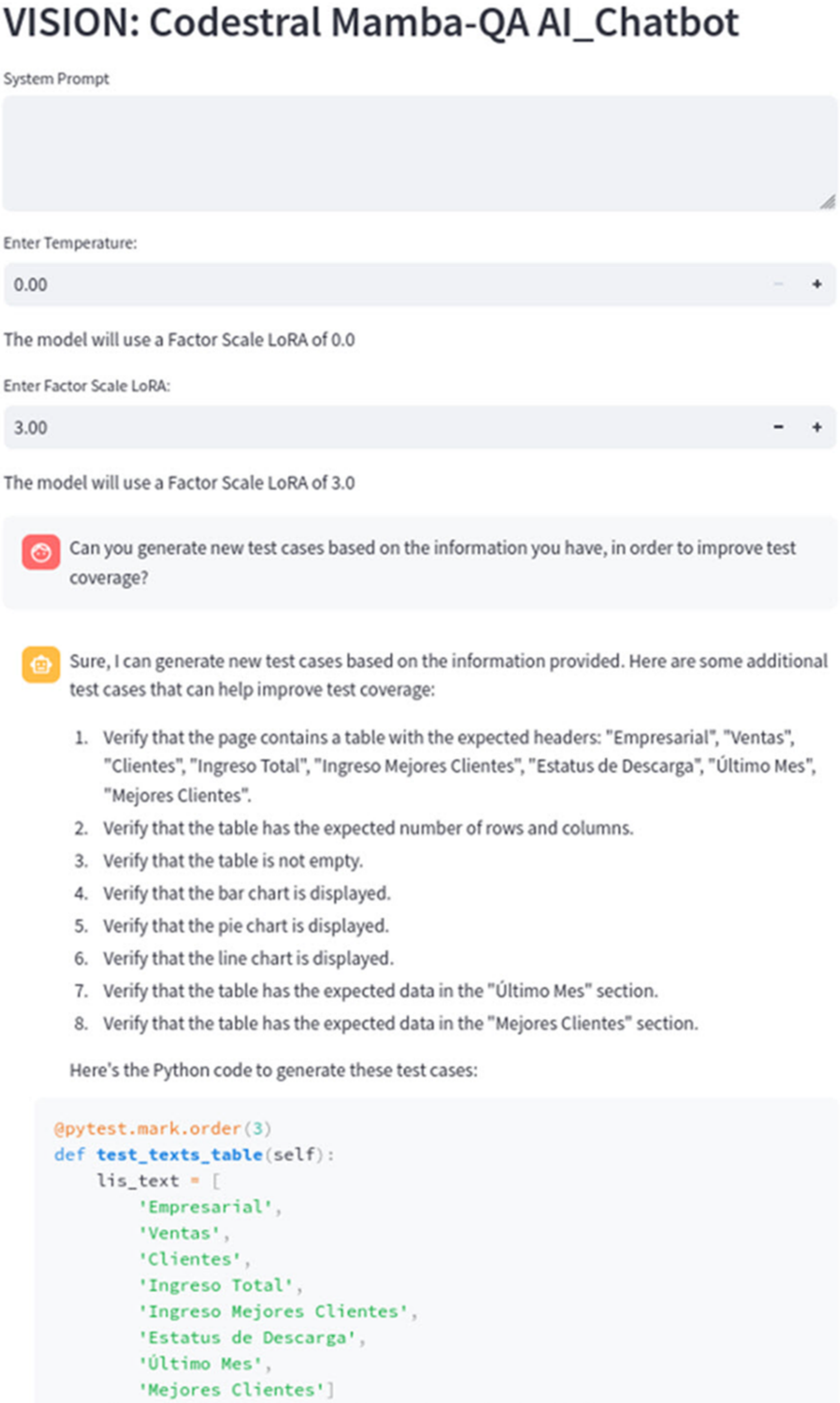
The following parameters and hyperparameters define the experimental setup:

- **Temperature**: Set to 0.00 to ensure deterministic outputs, reducing randomness and enhancing consistency in test case generation.
- **Factor Scale LoRA**: Adjusted to 3.0, allowing the model to leverage fine-tuned domain knowledge and generate structured, testable outputs.
- **System Input**: The model was instructed to generate new test cases to enhance test coverage for the project.
- **System Response**: The model autonomously produced a Pytest-based automated test script, incorporating validation logic and key elements relevant to the testing.

The generated test cases verify multiple critical elements of the project, including the presence of expected table headers (e.g., `"Empresarial"`, `"Ventas"`, `"Clientes"`), the correct number of rows and columns, the existence of graphical elements such as bar charts, pie charts, and line charts, and data validation in sections like `"Último Mes"` and `"Mejores Clientes"`.

Notably, the test cases `test_texts_table_ultimo_mes` and `test_texts_table_mejores_clientes` exhibit syntactic correctness and illustrate the models intelligence in generating relevant tests for business-specific scenarios. The consistency in generating structured test cases indicates that the model effectively learns patterns and adapts to project-specific requirements with increased LoRA scaling.

The inclusion of these test cases significantly strengthens the test suite, ensuring comprehensive validation across different system components. The ability to generate detailed, domain-specific test cases highlights the effectiveness of Codestral Mamba with LoRA scaling in automating quality assurance processes. By systematically analyzing how LoRA influences test generation, this study demonstrates the potential of AI-driven automation in enhancing software reliability and robustness.


VISION: Codestral Mamba-QA AI_Chatbot
System Prompt
Enter Temperature:
0.00
The model will use a Factor Scale LoRA of 0.0
Enter Factor Scale LoRA:
3.00
The model will use a Factor Scale LoRA of 3.0
Can you generate new test cases based on the information you have, in order to improve test coverage?
Sure, I can generate new test cases based on the information provided. Here are some additional test cases that can help improve test coverage:
1. Verify that the page contains a table with the expected headers: "Empresarial", "Ventas", "Clientes", "Ingreso Total", "Ingreso Mejores Clientes", "Estatus de Descarga", "Último Mes", "Mejores Clientes".
2. Verify that the table has the expected number of rows and columns.
3. Verify that the table is not empty.
4. Verify that the bar chart is displayed.
5. Verify that the pie chart is displayed.
6. Verify that the line chart is displayed.
7. Verify that the table has the expected data in the "Último Mes" section.
8. Verify that the table has the expected data in the "Mejores Clientes" section.
Here's the Python code to generate these test cases:
@pytest.mark.order(3)
def test_texts_table(self):
lis_text = [
'Empresarial',
'Ventas',
'Clientes',
'Ingreso Total',
'Ingreso Mejores Clientes',
'Estatus de Descarga',
'Último Mes',
'Mejores Clientes']

```
    attach(data=self.driver.get_screenshot_as_png())
    [self.assertTrue(x.text in lis_text, msg=x.text) for x in self.driver.f
        By.CLASS_NAME, 'iQZQ0rgSXnWCOo0Ij3WJ seZ5AAr8R21kluP7DBie ')]

@pytest.mark.order(4)
def test_number_of_rows_and_columns(self):
    # Verify that the table has 3 rows
    attach(data=self.driver.get_screenshot_as_png())
    self.assertEqual(len(self.driver.find_elements(
        By.XPATH, "/html/body/div/div/div[1]/div/div[2]/div[4]/div[1]/div/d

@pytest.mark.order(5)
def test_table_is_not_empty(self):
    attach(data=self.driver.get_screenshot_as_png())
    # Verify that the table is not empty
    self.assertNotEqual(
        len(
            self.driver.find_elements(
                By.XPATH,
                "/html/body/div/div/div[1]/div/div[2]/div[4]/div[1]/div/div
        0)
@pytest.mark.order(6)
def test_barchart(self):
    # Click on the bar chart button and verify chart
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[4]/div[2]/button[4]/img")
    time.sleep(7)
    attach(data=self.driver.get_screenshot_as_png())
    # Verify bar chart
    self.assertTrue(
        search_chart(
            self.driver,
            typechart="canva"),
        "Missing data chart")

@pytest.mark.order(7)
def test_piechart(self):
    # Click on the pie chart button and verify chart
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[4]/div[2]/button[2]/img")
    time.sleep(7)
    attach(data=self.driver.get_screenshot_as_png())
    # Verify pie chart
    self.assertTrue(
        search_chart(
            self.driver,
            typechart="canva"),
```

```
        "Missing data chart")

@pytest.mark.order(8)
def test_linechart(self):
    # Click on the line chart button and verify chart
    self.driver.find_element(
        By.XPATH,
        "/html/body/div/div/div[1]/div/div[2]/div[4]/div[2]/button[3]/img")
    time.sleep(7)
    attach(data=self.driver.get_screenshot_as_png())
    # Verify line chart
    self.assertTrue(
        search_chart(
            self.driver,
            typechart="canva"),
        "Missing data chart")

@pytest.mark.order(9)
def test_texts_table_ultimo_mes(self):
    lis_text = [
        'Ingreso Mejores Clientes',
        'Cliente',
        'Monto',
        'Facturas',
        'Cliente',
        'Monto',
        'Facturas']
    attach(data=self.driver.get_screenshot_as_png())
    [self.assertTrue(x.text in lis_text, msg=x.text) for x in self.driver.f
        By.CLASS_NAME, 'iQZQ0rgSXnWCOo0Ij3WJ seZ5AAr8R21k1uP7DBie ')]

@pytest.mark.order(10)
def test_texts_table_mejores_clientes(self):
    lis_text = ['Ejecutivo de Cuenta', 'Monto', 'Facturas']
    attach(data=self.driver.get_screenshot_as_png())
    [self.assertTrue(x.text in lis_text, msg=x.text) for x in self.driver.f
        By.CLASS_NAME, 'iQZQ0rgSXnWCOo0Ij3WJ seZ5AAr8R21k1uP7DBie ')]
```

### 4.9.8 Evaluation of Cross-Project Test Generation Using LoRA

The evaluation investigates the Codestral Mamba model's ability to generate automated test cases for projects outside the TestCase2Code database. Specifically, we examine its performance in generating test cases for the ALICE4u [11] project, which was not part of the original training data for the LoRA matrices.

The configuration used in this experiment is detailed in Figure 4.14. The model was set with the following parameters:

- **Temperature:** 0.00 This setting ensures deterministic output, eliminating randomness in test case generation. By maintaining a temperature of zero, the model consistently generates structured and reproducible test cases.
- **LoRA Scale Factor:** 1.0 This factor determines the extent to which the LoRA matrices influence the model's output. A scale factor of 1.0 balances fine-tuned adaptation with the base models pre-trained knowledge, allowing the model to leverage prior learning while incorporating project-specific details.
- **System Prompt:** The model was provided with a natural language prompt instructing it to generate a Pytest-based test case using the provided test case manual and '.jsx' file.

Given this configuration, the model successfully generated a test case for the ALICE4u project, focusing on verifying the login functionality for a Technical Director with valid credentials.

- **System Input:** The model received a .jsx file and a textual system prompt instructing it to generate a comprehensive Pytest-based test case.
- **System Response:** The model produced a structured test case that adhered to best practices in Pytest, including:
  1. Navigating to the login page and filling in the required fields.
  2. Using Selenium-based methods for element selection and interaction.
  3. Simulating key presses to enter the username and password.
  4. Validating navigation to the expected post-login page.

The successful test case generation demonstrates that the model can adapt to unseen projects while maintaining the structural integrity of automated test scripts. Notably:

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

- **Cross-Project Adaptability:** The model correctly inferred the test structure despite ALICE4u not being in the original training dataset.
- **Automated Test Coverage Expansion:** By dynamically generating test cases, the model helps improve test coverage for new projects, reducing manual effort in test design.
- **Best Practice Compliance:** The generated test follows established Pytest conventions, ensuring its applicability in real-world testing workflows.

The ability to generate reliable and structured test cases for projects outside the original training set highlights the value of LoRA-enhanced AI models in software quality assurance. This experiment underscores key benefits:

- **Scalability:** The model can be applied across multiple projects without requiring extensive re-training, making it a scalable solution for automated test case generation.
- **Improved Efficiency:** By automating test case creation, development teams can reduce time spent on manual test writing while maintaining high test coverage.
- **Robust Software Validation:** The generated test cases ensure that critical system functionalities, such as authentication, are thoroughly validated.

This experiment confirms that the Codestral Mamba model, when fine-tuned with LoRA at a scale factor of 1.0, is capable of cross-project test case generation. By leveraging structured prompts and context-aware adaptation, the model enhances software testing efficiency and facilitates broader test coverage in diverse development environments.

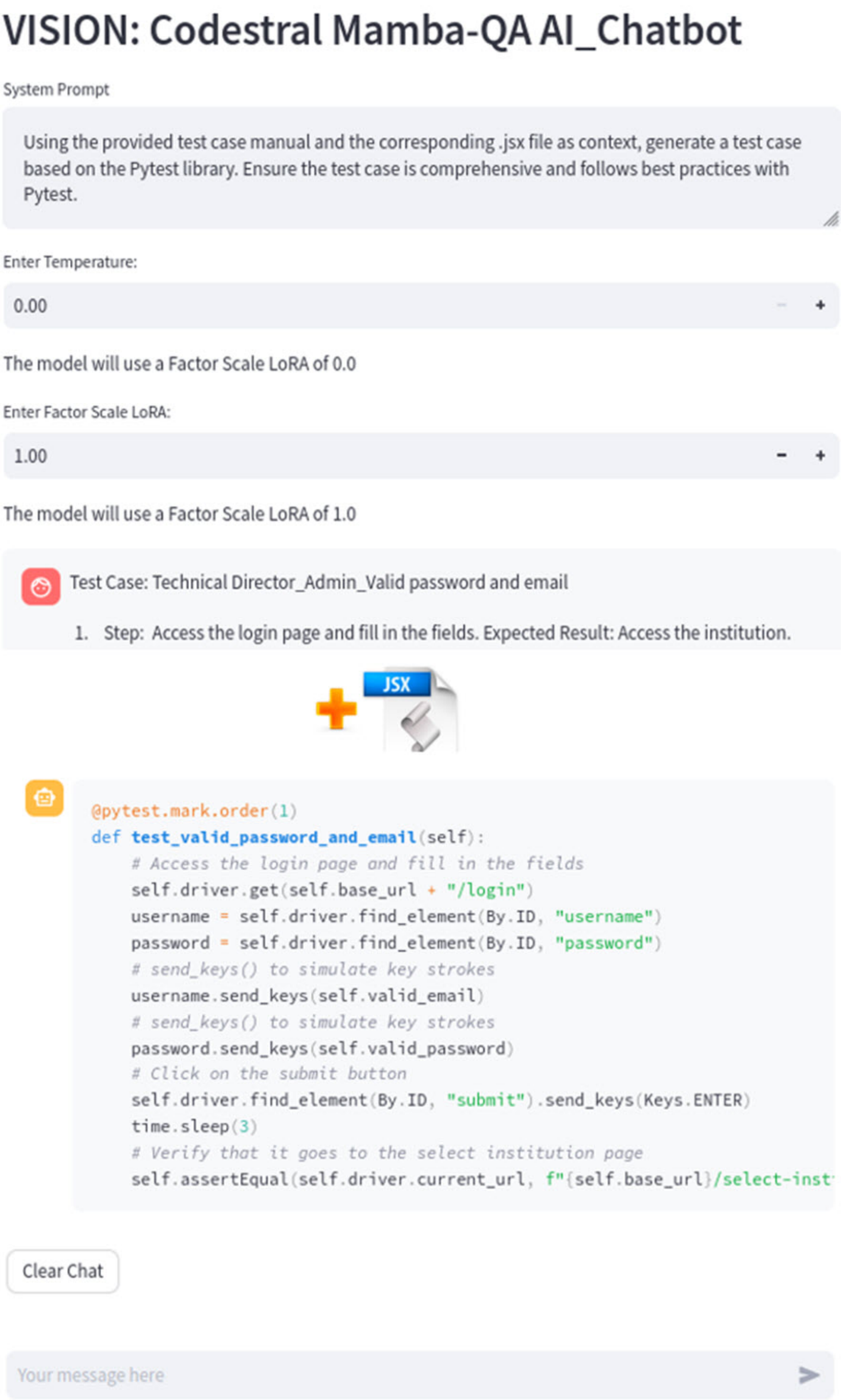



Figure 4.14: Generation of a test case for the ALICE4u project with temperature = 0 and LoRA scale factor = 1.

### 4.9.9 Bug Prevention via Automated Testing

This evaluation examines the capability of the Codestral Mamba model to generate automated test cases aimed at preventing software defects. Specifically, the experiment assesses how effectively the model can infer the requirements of a test case without an explicit system prompt and generate structured Pytest scripts to validate field constraints. The configuration used for this experiment is illustrated in Figure 4.15, where the temperature is set to 0.00 and the LoRA scale factor is set to 2.00. These settings ensure deterministic and contextually relevant outputs, minimizing variability while enhancing the specificity of the generated responses.

To analyze the models adaptability and precision in automated test case generation, the following parameters and hyperparameters were configured:

- **Temperature**: Set to 0.00, enforcing a deterministic response to ensure the generated test cases remain stable and reproducible.
- **LoRA Scale Factor**: Configured to 2.00, enhancing the models ability to incorporate domain-specific knowledge and generate structured, test-ready outputs.
- **System Prompt**: Left empty, requiring the model to infer intent based solely on input keywords, thereby testing its ability to interpret testing requirements without explicit guidance.
- **System Input**: The model was provided with a description of a bug related to input validation in a phone number field. The description included the expected validation behavior and a brief sequence of user actions required to reproduce the issue.
- **System Response**: The model autonomously generated a structured test case using Pytest, incorporating user interface interactions, input validation, and assertion mechanisms.

The bug addressed in this experiment originates from a real-world issue identified by the quality assurance team during routine testing. The defect, which allowed special characters to be erroneously accepted in the phone number field, posed a risk to data integrity and system reliability. By leveraging the chatbot, we were able to translate this documented issue into an automated test case efficiently, demonstrating the practical application of AI-driven test generation in real software development environments.

The objective of the generated test case is to validate the correct behavior of the `"Teléfono"` field by ensuring that special characters are not accepted. Using the `Pytest` framework, the test case follows a structured validation flow, performing key actions such as navigating to the "Administrar/Empresa" section, locating and clearing the phone number input field, inputting special characters (`!@$%&*()`), clicking the "Guardar" (Save) button, and verifying that a validation message is displayed to confirm that incorrect input is correctly rejected.

This experiment highlights the transformative impact of leveraging AI-powered models like Codestral Mamba in automated test case generation. Despite the absence of an explicit system prompt, the model successfully inferred the requirements, structured a logical sequence of validation steps, and produced a Pytest script tailored to enforce input constraints.

The significance of this finding is twofold:

- **Efficiency in Bug Prevention**: The model-generated test case automates validation, reducing reliance on manual testing and improving the reliability of quality assurance workflows.
- **Scalability and Adaptability**: By adjusting LoRA scaling, the model effectively aligns test case generation with domain-specific testing needs, ensuring adaptable and reusable test scripts across various scenarios.

This experiment underscores the tangible benefits of integrating AI-driven automation into software testing workflows. Not only does the Codestral Mamba_QA AI chatbot enabled approach streamline test case generation, but it also bridges the gap between real-world quality assurance findings and actionable, reproducible test scripts. By leveraging Codestral Mambas capacity to interpret structured input and generate robust validation procedures, development teams can enhance test coverage, mitigate defects earlier in the software lifecycle, and reinforce overall system resilience.

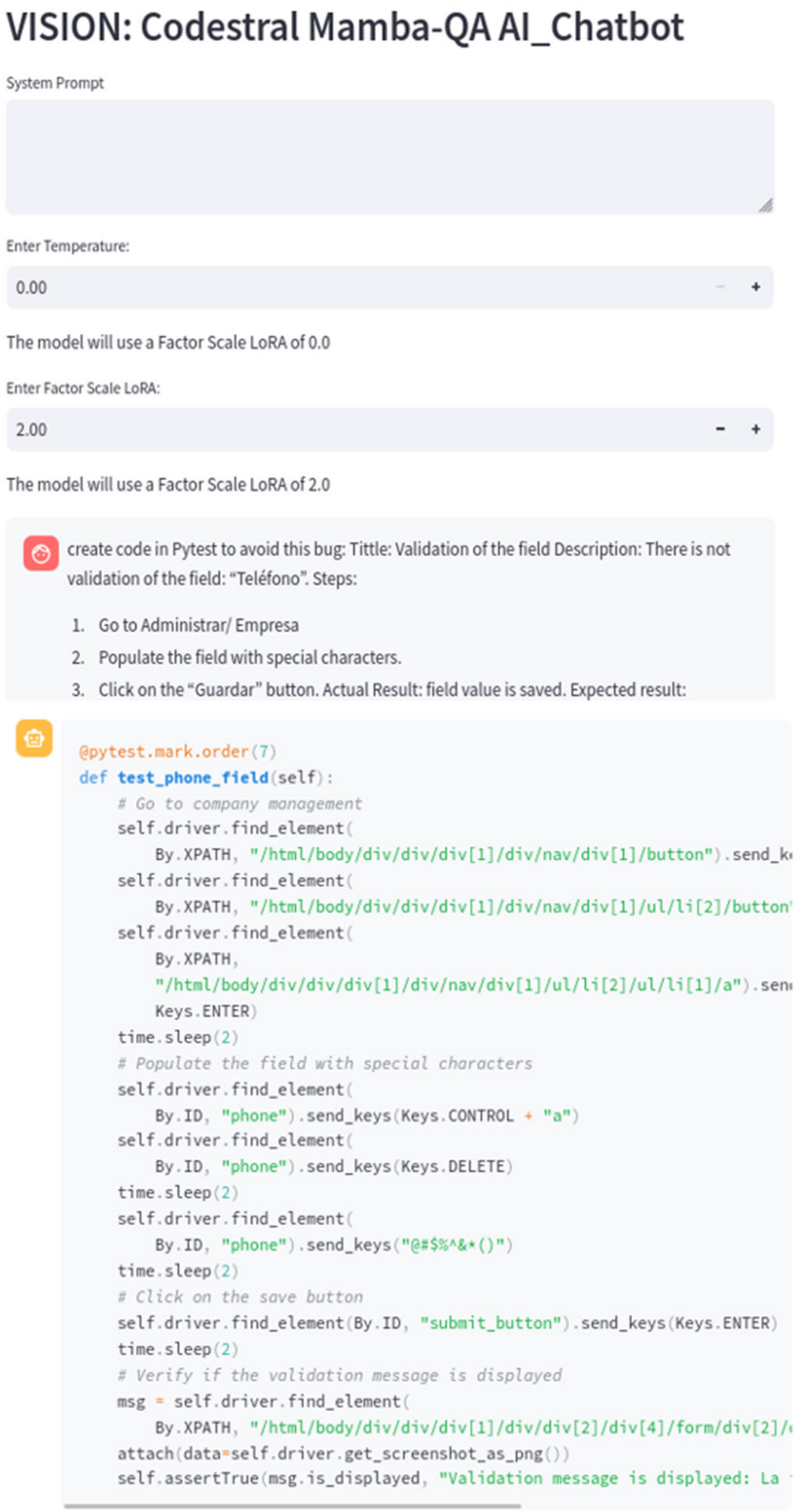



Figure 4.15: Generation of an automated test case to prevent a bug with temperature = 0 and LoRA scale factor = 2.

# Chapter 5

# Discussion and Conclusions

This research provides a comprehensive examination of the application of advanced language models in automated software testing, with a specific focus on generating high-quality, context-aware test scripts. By leveraging the Codestral Mamba model, augmented with Low-Rank Adaptation (LoRA), the study demonstrates significant advancements in automating test case generation, effectively addressing key challenges in software development and quality assurance. Additionally, it introduces novel datasets, methodologies, and tools that contribute to the broader field of software verification and validation, offering new avenues for enhancing the reliability and efficiency of software testing processes.

## 5.1 Summary of Key Findings

The findings of this research highlight the significant advancements achieved through the integration of large language models into software testing processes. The Codestral Mamba model, fine-tuned using LoRA, demonstrates a notable capability in generating test cases that are both syntactically and semantically precise. This proficiency is reflected in the evaluation metrics, with substantial improvements in CodeBLEU and Pass@1 scores, indicating the model's effectiveness in producing high-quality test scripts from natural language descriptions.

A pivotal contribution of this study is the introduction of the TestCase2Code dataset, an innovative benchmark designed to bridge the existing gap in software testing research. Currently, no publicly available dataset provides both manual and automated functional test cases, underscoring the uniqueness of TestCase2Code. Structured using real-world project data from extitDecsis, this dataset ensures direct applicability to industry use cases. By offering a structured repository of test cases, it lays the groundwork for advancing model training, evaluation, and benchmarking in software development environments, facilitating further improvements in automated test generation.

The model's performance was evaluated using two distinct datasetsCONCODE/-CodeXGLUE and TestCase2Codeeach serving a specific purpose in assessing adaptability and robustness. The CONCODE/CodeXGLUE dataset was chosen to determine whether the LoRA fine-tuning approach effectively learned dataset-specific patterns while preserving the models broader generalization capabilities. Experimental results indicate that the Codestral Mamba (LoRA) model achieved competitive results compared to state-of-the-art baselines, demonstrating a significant increase in BLEU and CodeBLEU scores compared to the untuned model, as evidenced in Table 4.9. Unlike prior approaches that focus exclusively on the CONCODE dataset, our fine-tuned model retains general-purpose knowledge while adapting efficiently to task-specific constraints, reinforcing the advantage of parameter-efficient fine-tuning methodologies. Additionally, training efficiency was a key outcome, with the model completing 200 training epochs in just 1.5 hours, underscoring the computational advantages of LoRA.

The evaluation on the TestCase2Code dataset further validated the models ability to generate functionally relevant and executable test cases. Unlike conventional text-to-code benchmarks, this dataset required the model to generate functionally coherent test scripts aligned with real-world software testing requirements. The results, summarized in Table 4.10, demonstrate significant enhancements across key evaluation metrics. The baseline Codestral Mamba model, while achieving a 100% Pass@1 score, exhibited limited performance in metrics such as n-gram overlap (4.82), weighted n-gram (11.8), and semantic matching (39.5). However, after fine-tuning with LoRA, the model showed dramatic improvements, with n-gram increasing to 56.2, weighted n-gram to 67.3, semantic matching to 91.0, and CodeBLEU rising from 26.9 to 74.7. These results highlight the effectiveness of LoRA in enhancing both syntactic accuracy and functional correctness, ensuring the generation of high-fidelity test cases that align with real-world validation criteria.

The efficiency of the fine-tuning process also proved instrumental in enabling rapid experimentation and adaptation. The LoRA fine-tuned model completed 200 training epochs in only 20 minutes, demonstrating a highly optimized training pipeline that facilitates iterative refinements with minimal computational overhead. This efficiency paves the way for a broader application of LoRA-based fine-tuning, supporting the development of reusable fine-tuned models for various software engineering tasks. By maintaining both computational efficiency and model adaptability, the approach strengthens the feasibility of deploying fine-tuned large language models in dynamic software development environments, where rapid testing, optimization, and customization are crucial.

Overall, these findings reinforce the viability of leveraging large language models in automated test case generation while underscoring the effectiveness of LoRA fine-tuning in enhancing model performance. The study provides empirical evidence supporting the adoption of parameter-efficient fine-tuning techniques, contributing to the advancement of AI-driven solutions in software quality assurance and code generation research.

## 5.2 Contributions to the Field of Software Testing

This research makes several notable contributions to the field of software testing:

1. **Advancement in Automated Test Case Generation:** The integration of the Codestral Mamba model with LoRA fine-tuning represents a significant step forward in automated test case generation. By reducing the reliance on manual test case creation, this approach enhances efficiency and consistency in the testing process.

2. **Improved Test Coverage and Quality:** The model's ability to generate a wide range of test cases ensures comprehensive coverage of software functionalities. This leads to earlier detection of defects and improved overall software quality, addressing a critical challenge in software development.

3. **Enhancing Software Quality Engineering Efficiency:** By automating the generation of test cases, software quality engineering teams can focus on refining test strategies, analyzing results, and ensuring comprehensive coverage. This shift in focus enhances productivity and allows for more rapid and reliable software releases.

4. **Creation of the TestCase2Code Dataset:** This dataset introduces an essential benchmark for automated test case generation, addressing a gap in publicly available resources. Structured from real-world project data, it enhances model evaluation and benchmarking while directly contributing to industry practices.

5. **Development of a Customized AI Chatbot:** Utilizing LoRA fine-tuning, this study develops the Codestral Mamba_QA AI Chatbot, an intelligent assistant tailored to the companys needs. This chatbot improves response accuracy by leveraging project-specific data, streamlines workload distribution, and integrates seamlessly with domain-specific terminology and workflows. A key advantage is its robust security framework, operating entirely within an internal infrastructure, ensuring data privacy and eliminating dependence on external solutions.

6. **Impact of LoRA Fine-Tuning and Prompt Engineering:** The study highlights the synergy between LoRA fine-tuning and prompt engineering in refining model performance for natural language code-answering tasks. This combined approach significantly enhances automation in test script generation, demonstrating AI-driven solutions' potential to create more efficient and adaptive workflows.

7. **A Structured Repository for LoRA Matrices:** A structured approach to managing fine-tuned models is proposed, envisioning a repository where LoRA adaptation matrices are systematically stored on a per-project basis. This repository functions as a centralized management system, ensuring that

each project benefits from a customized, optimized model tailored to its specific requirements. By adopting this modular approach, organizations can efficiently manage, update, and reuse these fine-tuned models across different projects, ensuring long-term effectiveness and applicability of generated test cases.

8. **Practical Applicability in Real-World Scenarios:** The model's adaptability to different programming languages and frameworks, along with its seamless integration into continuous integration and deployment (CI/CD) pipelines, underscores its practical utility. Notably, the model maintains its general intelligence, allowing it to handle diverse tasks without sacrificing performance. Additionally, its hyperparameters can be adjusted dynamically without requiring additional resources or reloading, enhancing flexibility in various development environments. This research lays a strong foundation for further exploration and the implementation of advanced language models in real-world software development workflows.

By introducing a novel dataset and a domain-specific AI assistant, this study significantly advances the automation of software testing. These contributions set the stage for more intelligent, efficient, and secure testing methodologies, ultimately transforming how software quality assurance is conducted in real-world environments. Furthermore, the structured repository of LoRA matrices ensures better model management, scalability, and maintainability, reinforcing the long-term viability of AI-driven test automation solutions.

## 5.3 Future Work

The findings of this study open several avenues for future research and potential advancements in the field of automated software testing. By leveraging the capabilities of advanced language models, such as the Codestral Mamba model enhanced with Low-Rank Adaptation, there are numerous opportunities to further enhance the efficiency, accuracy, and practical applicability of automated test case generation. This section outlines suggestions for future research and recommendations for expanding the current work.

### 5.3.1 Expanding Dataset Diversity

The effectiveness of automated test case generation using language models is heavily influenced by the diversity and completeness of the training datasets. While this study utilized the CONCODE/CodeXGLUE and TestCase2Code datasets, these datasets, though valuable, may not fully capture the complexity and variability of real-world software projects. The TestCase2Code dataset, in particular, was created by selecting .jsx files related to specific test cases, which contain project-specific information but lack a holistic representation of the entire codebase. This selective approach may limit the model's ability to generalize across different domains and software architectures.

Future research should explore integrating datasets that provide a more comprehensive view of software projects, including full repositories hosted on platforms like GitLab. By incorporating entire project repositories encompassing source code, documentation, dependencies, and configuration files models could be trained with richer contextual understanding, enhancing their ability to generate more accurate and relevant test cases.

Additionally, exploring methods for dynamically incorporating evolving repositories into the training pipeline could enhance the models ability to adapt to changes in software development practices.

### 5.3.2 Enhancing Evaluation Metrics

While the current study utilized metrics such as CodeBLEU and Pass@1 to evaluate the model's performance, future research should consider incorporating additional evaluation metrics that assess the practical usability and maintainability of the generated test cases. Conducting user studies with software developers to gather feedback on the generated test cases' real world applicability could provide valuable insights into the model's practical effectiveness.

Future research could develop comprehensive evaluation frameworks that include metrics for code readability, maintainability, and integration with existing systems. These frameworks should be designed in collaboration with industry experts to ensure their relevance and applicability to real world software development scenarios.

### 5.3.3 Continuous Learning and Adaptation

Software development is a dynamic process, with requirements and codebases evolving rapidly. Future research should focus on developing models that can adapt to these changes continuously, generating relevant test cases as the software evolves. Exploring self-adapting models that can learn from evolving codebases and requirements without extensive manual intervention could enhance the model's practical applicability.

Developing automated tools and frameworks to support the continuous fine-tuning and updating of the model is essential for maintaining its relevance in dynamic software environments. By enabling the model to adapt to evolving codebases and changing requirements, these advancements would enhance its ability to generate accurate and contextually appropriate test cases over time. Further exploration of self-adapting models capable of learning from real-time project updates without extensive manual intervention could significantly improve the models long-term effectiveness and applicability.

### 5.3.4 Integration with Development Workflows

Integrating advanced language models into existing software development workflows, particularly in continuous integration and deployment (CI/CD) pipelines, presents technical and organizational challenges. Future research should focus on developing standardized integration frameworks and best practices for incorporating these models into CI/CD pipelines. Conduct case studies to evaluate the impact of these integrations on software development processes and outcomes.

The future work outlined above builds upon the findings of this study and provides a strategic direction for advancing automated software testing. Expanding dataset diversity, optimizing fine-tuning techniques, enhancing evaluation metrics, and exploring continuous learning and adaptation can further improve the practical applicability and effectiveness of advanced language models in software development and testing. These advancements will contribute to more robust, efficient, and adaptable automated testing solutions, addressing real-world challenges in software quality assurance.

# Bibliography

AbuSalim, S. W., Ibrahim, R., and Wahab, J. A. (2021). Comparative analysis of software testing techniques for mobile applications. *Journal of Physics: Conference Series*, 1793(1):012036.

Alagarsamy, S., Tantithamthavorn, C., Arora, C., and Aleti, A. (2024). Enhancing large language models for text-to-testcase generation. *arXiv preprint*.

Android Developers (2012). Ui/application exerciser monkey.

Axelrod, A. (2018). *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects*. Apress.

Banik, S. and Dandyala, S. S. M. (2019). Automated vs. manual testing: Balancing efficiency and effectiveness in quality assurance. *International Journal of Machine Learning Research in Cybersecurity and Artificial Intelligence*, 10(1):100–119.

Beizer, B. (1990a). *Software Testing Techniques*. Van Nostrand Reinhold, New York.

Beizer, B. (1990b). *Software Testing Techniques, Second Edition*. Van Nostrand Reinhold, New York.

Chakraborty, S., Ahmed, T., Ding, Y., Devanbu, P. T., and Ray, B. (2022). Natgen: generative pre-training by naturalizing source code. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 18–30.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Pachocki, J., Saunders, W., Hesse, C., and Schulman, J. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Cohn, M. (2009). *Succeeding with Agile*. Addison-Wesley Professional.

Contan, A., Dehelean, C., and Miclea, L. (2018). Test automation pyramid from theory to practice. In *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pages 1–5. IEEE.

Crispin, L. and Gregory, J. (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley, Boston, MA.

Dantas, V. (2023). Large language model powered test case generation for software applications. *Technical Disclosure Commons*. Accessed: 2024-09-26.

Dao, T. and Gu, A. (2024). Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*.

Delgado-Pérez, P., Ramírez, A., Valle-Gómez, K., Medina-Bulo, I., and Romero, J. (2023). Interevo-tr: Interactive evolutionary test generation with readability assessment. *IEEE Transactions on Software Engineering*, 49(4):2580–2596.

Demir, B. and Aksoy, A. (2024a). Implementing strategic automation in software development testing to drive quality and efficiency. *Sage Science Review of Applied Machine Learning*, 7(1):94–119.

Demir, B. and Aksoy, A. (2024b). Implementing strategic automation in software development testing to drive quality and efficiency. *Sage Science Review of Applied Machine Learning*, 7(1):94–119.

Deng, Y., Xia, C. S., Peng, H., Yang, C., and Zhang, L. (2023). Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT*, pages 4171–4186.

Doe, J. (2018). Augmenting the software testing workflow with machine learning.

Dong, Z., Böhme, M., Cojocaru, L., and Roychoudhury, A. (2020). Time-travel testing of android apps. In *ICSE*.

El-Morabea, K., El-Garem, H., El-Morabea, K., and El-Garem, H. (2021). Testing pyramid. *Modularizing Legacy Projects Using TDD: Test-Driven Development with XCTest for iOS*, pages 65–83.

Eldrandaly, K., ElLatif, M. A., and Zaki, N. (2019). Comparative study of software test automation frameworks. *International Journal of Engineering Trends and Technology*, 67(11):94–105.

Fang, W., Wang, K., and Wang, W. (2024). Automated test case generation for webassembly using large language models. In *International Conference on Learning Representations*.

Go, K., Kang, S., Baik, J., and Kim, M. (2016). Pairwise testing for systems with data derived from real-valued variable inputs. *Software: Practice and Experience*, 46(3):381–403.

Gu, A. (2024). Mamba: Introducing working memory in ai models. *Time*.

Gu, A. and Dao, T. (2023). Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*.

Gu, A., Dao, T., Ermon, S., Rudra, A., and Ré, C. (2020). Hippo: Recurrent memory with optimal polynomial projections. *Advances in Neural Information Processing Systems*, 33:1474–1487.

Gu, A., Goel, K., and Re, C. (2021). Efficiently modeling long sequences with structured state spaces. *Advances in Neural Information Processing Systems*, 34:9998–10010.

Harman, M. and McMinn, P. (2010). A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., and Chen, W. (2021). Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Huang, W., Pan, J., Tang, J., Ding, Y., Xing, Y., Wang, Y., Wang, Z., and Hu, J. (2024). Ml-mamba: Efficient multi-modal large language model utilizing mamba-2. *arXiv preprint arXiv:2407.19832*.

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. (2018). Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*.

Jamil, M. A., Arif, M. S., Abubakar, N. S. A., and Ahmad, A. (2016). Software testing techniques: A literature review. In *2016 International Conference on Information and Communication Technology for Muslim World (ICT4M)*, pages 177–182. IEEE.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. (2023). Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Jorgensen, P. C. (2013). *Software Testing: A Craftsman's Approach, Fourth Edition*. CRC Press, Boca Raton, FL.

Jurafsky, D. and Martin, J. H. (2025). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd edition.

Karmarkar, H., Agrawal, S., Chauhan, A., and Shete, P. (2024). Navigating confidentiality in test automation: A case study in llm driven test data generation. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 337–348. IEEE.

Kaur, K., Khatri, S. K., and Datta, R. (2014). Analysis of various testing techniques. *International Journal of System Assurance Engineering and Management*, 5(2):276–290.

Khaliq, Z., Farooq, S. U., and Khan, D. A. (2022). Artificial intelligence in software testing: Impact, problems, challenges and prospect. *arXiv preprint arXiv:2201.05371*.

Khant, D., Changela, F., and Sodha, S. (2016). A review on: Manual vs. automated testing. *International Journal of Advance Research in Engineering, Science & Technology*, 3(13).

Li, Y., Yang, Z., Guo, Y., and Chen, X. (2017). Droidbot: A lightweight ui-guided test input generator for android. In *ICSE*.

Lieber, O., Lenz, B., Bata, H., Cohen, G., Osin, J., Dalmedigos, I., Safahi, E., Meirom, S., Belinkov, Y., Shalev-Shwartz, S., et al. (2024). Jamba: A hybrid transformer-mamba language model. *arXiv preprint arXiv:2403.19887*.

Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D., and Wang, Q. (2023). Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. *CoRR*, arXiv:2310.15780.

Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D., and Wang, Q. (2024). 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*, pages 1–13.

Lops, A., Narducci, F., Ragone, A., Trizio, M., and Bartolini, C. (2024). A system for automated unit test generation using large language models and assessment of generated test suites. *arXiv preprint arXiv:2408.07846*.

Lu, S., Guo, D., Cao, Z., Duan, N., Li, M., Liu, S., Sun, M., Wang, D., Tang, J., et al. (2021). Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*. Available at https://github.com/microsoft/CodeXGLUE.

Maspupah, A. (2023). Literature review: Advantages and disadvantages of black box and white box testing methods. *Jurnal Techno Nusa Mandiri*, 21(2):157–164.

Mathur, A., Pradhan, S., Soni, P., Patel, D., and Regunathan, R. (2023). Automated test case generation using t5 and gpt-3. In *2023 9th International Conference on Advanced Computing and Communication Systems (ICACCS)*, volume 1, pages 1986–1992. IEEE.

Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44.

Myers, G. J. (2004). *The Art of Software Testing (2nd Edition)*. Wiley.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The Art of Software Testing*. John Wiley & Sons.

Nadeau, D., Kroutikov, M., McNeil, K., and Baribeau, S. (2024). Benchmarking llama2, mistral, gemma and gpt for factuality, toxicity, bias and propensity for hallucinations. *arXiv preprint arXiv:2404.09785*.

Nidhra, S. and Dondeti, J. (2012). Black box and white box testing techniquesa literature review. *International Journal of Embedded Systems and Applications*, 2(2):29–50.

Ognawala, S., Petrovska, A., and Beckers, K. (2017). An exploratory survey of hybrid testing techniques involving symbolic execution and fuzzing. *arXiv preprint arXiv:1712.06843*.

Okken, B. (2022). *Python Testing with pytest*. Pragmatic Bookshelf.

OpenAI (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. (2007). Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE)*, pages 75–84.

Pan, M., Huang, A., Wang, G., Zhang, T., and Li, X. (2020). Reinforcement learning-based curiosity-driven testing of android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–164.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 311–318. Association for Computational Linguistics.

Parihar, M. and Bharti, A. (2019). A survey of software testing techniques and analysis. *International Journal of Research*, 6(3):153–158.

Pressman, R. S. (2005). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 6th edition.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Ramadan, A., Yasin, H., and Pektas, B. (2024). The role of artificial intelligence and machine learning in software testing. *arXiv preprint arXiv:2409.02693*.

Ren, S., Guo, D., Deng, S., Wang, S., Lyu, M. R., and King, I. (2020). Codebleu: a method for automatic evaluation of code synthesis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 75–86. ACM.

Research, M. A. (2024). Megalodon: Extending context windows in language models. *VentureBeat*.

Researchers, M. (2024). Liquid neural networks: Redesigning the neural network. *Wired*.

Schäfer, M., Nadi, S., Eghbali, A., and Tip, F. (2023). An empirical evaluation of using large language models for automated unit test generation. *arXiv preprint arXiv:2302.06527*.

Sewnet, A., Kifle, M., and Tilahun, S. L. (2023). The applicability of automated testing frameworks for mobile application testing: A systematic literature review. *Computers*, 12(5):97.

Shtokal, A. and Smołka, J. (2021). Comparative analysis of frameworks used in automated testing on example of testng and webdriverio. *Journal of Computer Sciences Institute*, 19:100–106.

Steenhoek, B., Tufano, M., Sundaresan, N., and Svyatkovskiy, A. (2024). Reinforcement learning from automatic feedback for high-quality unit test generation. *arXiv preprint arXiv:2412.14308*.

Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., and Su, Z. (2017). Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256.

Su, T., Wang, J., and Su, Z. (2021). Benchmarking automated gui testing for android against real-world bugs. In *ESEC/FSE*, pages 119–130.

Sullca, E. (2023). Impact of software testing automation on the development cycle. *Revista de Investigación Científica Huamachuco*, 1:23–27.

Tang, Y., Liu, Z., Zhou, Z., and Luo, X. (2023). Chatgpt vs sbst: A comparative assessment of unit test suite generation. *CoRR*, arXiv:2307.00588.

Thakkar, H. and Manimaran, A. (2023). Comprehensive examination of instruction-based language models: A comparative analysis of mistral-7b and llama-2-7b. In *2023 International Conference on Emerging Research in Computational Science (ICERCS)*, pages 1–6. IEEE.

Umar, M. A. and Chen, Z. (2019). A study of automated software testing: Automation tools and frameworks. *International Journal of Computer Science Engineering*, 8(6):215–223.

Wallace, D. R. and Fujii, R. U. (1989). Software verification and validation: An overview. *IEEE Software*, 6(3):10–17.

Wang, W., Liu, P., Zhang, Z., Zhang, M., and Lin, Z. (2017). Automatic test pattern generator for fuzzing based on finite state machine. *Security and Communication Networks*, 2017.

Watson, C., Tufano, M., Moran, K., Bavota, G., and Poshyvanyk, D. (2020). On learning meaningful assert statements for unit test cases. In *42nd International Conference on Software Engineering (ICSE 20)*, Seoul, Republic of Korea.

Xia, C. S., Paltenghi, M., Tian, J. L., Pradel, M., and Zhang, L. (2024). Fuzz4all: Universal fuzzing with large language models. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, Lisbon, Portugal.

Xiao, X., Li, S., Xie, T., and Tillmann, N. (2013). Characteristic studies of loop problems for structural test generation via symbolic execution. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 246–256.

Xu, C., Zheng, K., Wang, Z., Gan, Z., Wang, Z., Wang, Y., Wang, L., et al. (2023). Parameter-efficient fine-tuning design spaces. *arXiv preprint arXiv:2303.18239*.

Xue, Z., Li, L., Tian, S., Chen, X., Li, P., Chen, L., Jiang, T., and Zhang, M. (2024). 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*, pages 1643–1655.

Yu, S., Fang, C., Ling, Y., Wu, C., and Chen, Z. (2023). Llm for test script generation and migration: Challenges, capabilities, and opportunities. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pages 206–217. IEEE.

Yuan, Z., Lou, Y., Liu, M., Ding, S., Wang, K., Chen, Y., and Peng, X. (2023). No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint*, arXiv:2305.04207.

Zhang, C., Bai, M., Zheng, Y., Li, Y., Ma, W., Xie, X., Li, Y., Sun, L., and Liu, Y. (2023). Understanding large language model based fuzz driver generation. *arXiv e-prints*, pages arXiv–2307.

Zhang, J., Liu, X., and Chen, J. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4).

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y., and Wen, J.-R. (2023). A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Zuo, J., Velikanov, M., Rhaiem, D. E., Chahed, I., Belkada, Y., Kunsch, G., and Hacid, H. (2024). Falcon mamba: The first competitive attention-free 7b language model. *arXiv preprint arXiv:2410.05355*.