Dissertation presented to the Instituto Tecnológico de Aeronáutica, in partial fulfillment of the requirements for the degree of Master of Science in the Graduate Program of Engenharia da Computação, Field of (Area).

Fernando Gusmão Zanchitta

LARGE LANGUAGE MODELS FOR AUTOMATED PROGRAM REPAIR: AN EMPIRICAL STUDY WITH TRACEBACKS AND PROMPT ENGINEERING

Dissertation approved in its final version by signatories below:

Prof. Dr. Filipe Alves Neto Verri
Advisor

Prof. Dr. ... INSERIR
Pro-Rector of Graduate Courses

Campo Montenegro
São José dos Campos, SP - Brazil
2025

Cataloging-in Publication Data
Documentation and Information Division

Gusmão Zanchitta, Fernando
Large Language Models for Automated Program Repair: An Empirical Study with Tracebacks and Prompt Engineering / Fernando Gusmão Zanchitta.
São José dos Campos, 2025.
30p.

Dissertation of Master of Science – Course of Engenharia da Computação. Area of (Area) – Instituto Tecnológico de Aeronáutica, 2025. Advisor: Prof. Dr. Filipe Alves Neto Verri.

1. Automatic Program Repair. 2. Large Language Model. 3. Prompt Engineering. I. Instituto Tecnológico de Aeronáutica. II. Large Language Models for Automated Program Repair: An Empirical Study with Tracebacks and Prompt Engineering.

BIBLIOGRAPHIC REFERENCE

GUSMÃO ZANCHITTA, Fernando. **Large Language Models for Automated Program Repair: An Empirical Study with Tracebacks and Prompt Engineering**. 2025. 30p. Dissertation of Master of Science – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSION OF RIGHTS

AUTHOR'S NAME: Fernando Gusmão Zanchitta
PUBLICATION TITLE: Large Language Models for Automated Program Repair: An Empirical Study with Tracebacks and Prompt Engineering.
PUBLICATION KIND/YEAR: Dissertation / 2025

It is granted to Instituto Tecnológico de Aeronáutica permission to reproduce copies of this dissertation and to only loan or to sell copies for academic and scientific purposes. The author reserves other publication rights and no part of this dissertation can be reproduced without the authorization of the author.

Fernando Gusmão Zanchitta
Av. Jorge Zarur 231, Bl 6, Ap.301
12.243-081 – São José dos Campos–SP

LARGE LANGUAGE MODELS FOR AUTOMATED PROGRAM REPAIR: AN EMPIRICAL STUDY WITH TRACEBACKS AND PROMPT ENGINEERING

Fernando Gusmão Zanchitta

Thesis Committee Composition:

Prof. Dr.	Alan Turing	Presidente	-	ITA
Prof. Dr.	Filipe Alves Neto Verri	Advisor	-	ITA
Prof. Dr.	Linus Torwald		-	UXXX
Prof. Dr.	Richard Stallman		-	UYYY
Prof. Dr.	Donald Duck		-	DYSNEY
Prof. Dr.	Mickey Mouse		-	DISNEY

ITA

Aos amigos da Graduação e Pós-Graduação do ITA por motivarem tanto a criação deste template pelo Fábio Fagundes Silveira quanto por motivarem a mim e outras pessoas a atualizarem e aprimorarem este excelente trabalho.

Acknowledgments

Primeiramente, gostaria de agradecer ao Dr. Donald E. Knuth, por ter desenvolvido o TeX.

Ao Dr. Leslie Lamport, por ter criado o LaTeX, facilitando muito a utilização do TeX, e assim, eu não ter que usar o Word.

Ao Prof. Dr. Meu Orientador, pela orientação e confiança depositada na realização deste trabalho.

Ao Dr. Nelson D'Ávilla, por emprestar seu nome a essa importante via de trânsito na cidade de São José dos Campos.

Ah, já estava esquecendo... agradeço também, mais uma vez ao TeX, por ele não possuir vírus de macro :-)

"If I have seen farther than others,
it is because I stood on the shoulders of giants."
— Sir Isaac Newton

Resumo

Placeholder abstract

Abstract

Well, the book is on the table. This work presents a control methodologie for the position of the passive joints of an underactuated manipulator in a suboptimal way. The term underactuated refers to the fact that not all the joints or degrees of freedom of the system are equipped with actuators, which occurs in practice due to failures or as design result. The passive joints of manipulators like this are indirectly controlled by the motion of the active joints using the dynamic coupling characteristics. The utilization of actuation redundancy of the active joints allows the minimization of some criteria, like energy consumption, for example. Although the kinematic structure of an underactuated manipulator is identical to that of a similar fully actuated one, in general their dynamic characteristics are different due to the presence of passive joints. Thus, we present the dynamic modelling of an underactuated manipulator and the concept of coulpling index. This index is used in the sequence of the optimal control of the manipulator.

List of Figures

List of Tables

Contents

1 Introduction and Background

This chapter covers the motivation for this thesis, along with research goals, research questions, and a list of contributions.

1.1 Motivation

Modern software systems continuously evolve with inevitable bugs due to feature deprecation, new features added, and refactoring. These bugs are widely known as a destructive problem, generating costs up to trillions of dollars every year.

Bugs and errors in code are a recurring problem in the software industry, leading to collateral effects of multiple sizes. As a consequence, there is a substantial amount of time spent only to solve these problems in an efficient way with litle or no human intervention.

In APR research of the last decade, most of traditional methods include search-based, constraint-based, and template-based approaches.These studies led to the development of methods such of SimFix, VarFix, SearchRepair, and Tfix. More recent studies use machine learning and deep learning techniques were employed to improve program repair tasks. With the growth of Large Language Models (LLMs), the Software Engineering area was significantly transformed, specially due to Code LLMs. These models are either pretrained or finetuned on programming languages, directly affecting areas such as code summarization, code generation, bug detection and also APR.

1.2 Problem Statement

1.3 Objective and Research Questions

This project aims to study different techniques of prompt engineering to increase the performance of Patch correction tasks.

RQ1: Comparison between specialized code models vs general-purpose models?

RQ2: Does style based prompt improve the overall effectiveness of responses?

RQ3: Does system prompt improve performances of APR tasks?

1.4 Contribution

2 Related Works

This chapter provides a concise overview of key works in automated program repair and the application of large language models to code correction tasks. Also, present multiple related projects that state current trend research into this field.

3 Methodology

This chapter details the methodology adopted for evaluating Large Language Models (LLMs) in the context of Automated Program Repair (APR) using Python code. It describes the experimental setup, including the use of the Pytracebugs dataset, the prompt engineering strategies, and the suite of evaluation metrics designed to assess both syntactic and semantic correctness of model-generated repairs. The chapter also explains the implementation details, such as model execution, configuration of generation parameters, and the error handling and monitoring mechanisms employed to ensure reliable and reproducible results. The goal is to provide a clear and concise account of the procedures and tools used, enabling transparent and repeatable experimentation. This chapter details the methodology adopted for evaluating Large Language Models (LLMs) in the context of Automated Program Repair (APR) using Python code. It describes the experimental setup, including the use of the Pytracebugs dataset, the prompt engineering strategies, and the suite of evaluation metrics designed to assess both syntactic and semantic correctness of model-generated repairs. The chapter also explains the implementation details, such as model execution and configurations.

3.1 Dataset and datapreparation

3.1.1 Pytracebugs

The experiment uses the Pytracebugs dataset (Akimova *et al.*, 2021), which contains Python source codes from GitHub repositories at the granularity of function snippets and their corresponding Traceback errors; see the example below.

Most frequent traceback errors can be found in Figure 3.2, with the most frequent ones being Atribute Error and TypeError with 16.84% and 16% of the values respectivelly. Also, there is a significant amount of erros that appear less than 1% of the time, and were aggregated into the 'Others' Category.

Some published works that use this dataset aim to study problems such as Vulnerability Detection (Zhao *et al.*, 2024) and fault localization (Kulkarni *et al.*, 2024).

FIGURE 3.1 – An example of a single data point from the Pytracebugs dataset.

```python
def noise(self, value):
  self.noise_covar.initialize(value)

```

Listing 3.1 – a) Buggy Code Snippet (`before_merge`)

```
import gpytorch
gl = gpytorch.likelihoods.GaussianLikelihood()
gl.initialize(noise=1)
Traceback (most recent call last):
File ""<stdin>"", line 1, in <module>
File ""../gpytorch/gpytorch/module.py"", line 89, in initialize
setattr(self, name, val)
File ""../lib/python3.6/site-packages/torch/nn/modules/module.py"",
   line 579, in __setattr__
object.__setattr__(self, name, value)
File ""../gpytorch/gpytorch/likelihoods/gaussian_likelihood.py"",
   line 63, in noise
self.noise_covar.initialize(value)
TypeError: initialize() takes 1 positional argument but 2 were given

```

Listing 3.2 – b) Traceback Error (`full_traceback`)

```python
def noise(self, value):
  self.noise_covar.initialize(noise=value)

```

Listing 3.3 – c) Fixed Code Snippet (`after_merge`)

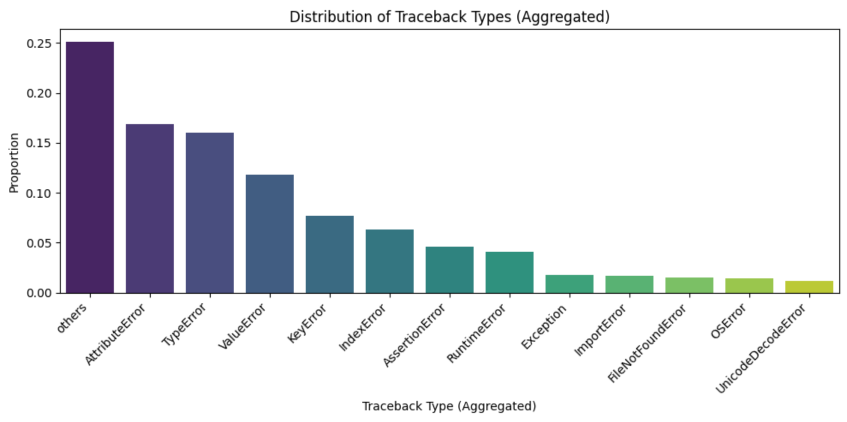

FIGURE 3.2 – Distribution of Error Types in the dataset - Less than 1% relevant are consolidated into ’others’ category.

3.1.2 LLM Models

The model selection for this experiment was guided by three criteria: (1) representation of different architectural approaches, (2) availability through OpenRouter API, and (3) cost-effectiveness for large-scale experimentation. Four models were selected to provide a balanced comparison across these dimensions.

Two open-source models specialized in code generation were included: Qwen 2.5 Coder 32B (instruct) (Hui *et al.*, 2024) and Codestral 2501 (MistralAi, 2025). These models represent the current state-of-the-art in open-source code generation, with Qwen 2.5 Coder 32B achieving competitive performance on coding benchmarks while maintaining significantly lower computational requirements compared to larger models.

Two closed-source models were selected for comparison: Claude 3.5 Sonnet and GPT-4o. These models represent the current frontier of general-purpose language models and provide a baseline for evaluating whether specialized code models offer advantages over general-purpose architectures in automated program repair tasks.

The selection of exactly four models balances statistical power requirements with computational feasibility. While larger model sets could provide more comprehensive comparisons, the chosen sample size allows for robust statistical analysis while maintaining manageable experimental costs.

Although these models are all similarly competitive in structured benchmarks, they differ considerably in terms of usage cost. For reference, the token cost for each selected model is shown in Table 3.1.

TABLE 3.1 – Token usage price per model.

Model	**Input Tokens (1M)**	**Output Tokens (1M)**
Claude Sonnet 3.5	$3.0	$15.0
Gpt 4o	$2.5	$10
Codestral 2501	$0.3	$0.9
Qwen 2.5 Coder 32B Instruct	$0.05	$0.20

3.2 Experimental Design

3.2.1 Prompt variations

To test the model's sensitivity to the prompt's instructions, all prompts were formatted using a consistent template, shown in Listing 3.4. This structure ensures that each model receives the context (buggy code and traceback error) in a clear, delimited format, and is explicitly asked to return only the corrected code.

The core of the experiment lies in the variation of the `instruction_prompt` placeholder and the addition of system prompt. We designed three distinct instructions, which we will refer to as P1 (Baseline) and P2 (Style-aware) and P3 (System Prompt), to evaluate the impact of a more detailed directive. The specific text for each instruction is detailed

```
{{ instruction_prompt }}

### BUGGY CODE:
{{ buggy_code }}

### ERROR:
{{ traceback_error }}

### RETURN ONLY THE CORRECTED CODE BELOW:

IMPORTANT: Return ONLY the corrected/requested code. Do not include
    any explanations, comments about the changes, or other text. Just
    return the pure code.
```

Listing 3.4 – The general template used for all prompts.

TABLE 3.2 – Instruction variations for the `instruction_prompt` variable.

Prompt ID	**Instruction Text**	**System Prompt**
P1 (Baseline)	You are a helpful assistant that corrects the code based on the traceback error.	False.
P2 (Style-aware)	You are a helpful assistant that corrects the code based on the traceback error. You must respect the original code structure and the original code style.	False.
P3 (System Prompt)	You are a helpful assistant that corrects the code based on the traceback error.	True.

in Table 3.2.

3.2.1.1 Justification of Prompt Variations

The selection of these three specific prompt variations was driven by theoretical considerations in prompt engineering and their relevance to automated program repair tasks. Each variation was designed to test a distinct hypothesis about how LLMs process and respond to different types of instructions.

P1 (Baseline): This prompt serves as the experimental control, providing minimal instruction to establish a baseline performance level. The simplicity of this prompt allows us to measure the inherent capability of each model without additional constraints, serving as a reference point for evaluating the effectiveness of more specific instructions. This approach follows the principle of establishing a null hypothesis in experimental design, where we can assess whether additional prompt complexity actually improves performance.

P2 (Style-aware): The style-aware prompt introduces explicit constraints about preserving code structure and style, addressing a critical concern in automated program repair. Previous research has shown that LLMs can generate syntactically correct but stylistically inconsistent code (Wang *et al.*, 2025), which may reduce code maintainability and increase cognitive load for developers. By explicitly instructing the model to respect original code structure and style, we test whether such constraints lead to more maintainable and contextually appropriate patches. This variation is particularly relevant for production environments where code consistency is paramount.

P3 (System Prompt): This variation tests the hypothesis that system-level instructions, which have higher precedence in the API message hierarchy, can provide more consistent and reliable behavior than instruction-level prompts. System prompts are designed to establish global behavioral patterns and are processed differently by the model's architecture. By moving the instruction to the system level, we investigate whether this architectural difference results in more deterministic and consistent code generation, which is crucial for automated repair systems that require reliability and reproducibility.

The systematic variation between these three conditions allows us to isolate the effects of instruction specificity (P1 vs. P2) and instruction placement (P2 vs. P3), providing insights into both the content and delivery mechanisms that influence LLM performance in code repair tasks.

3.2.1.2 Prompt Roles in API requests

- **System prompt:** It is an initialization message that sets global behavior for the model across a conversation or a request. It encodes policies, personas, output formatting, and safety constraints. In most APIs, it has a higher precedence than user instructions. The typical contents usually are guardrails, style registers, and output contracts (e.g., "always return strict python code").
- **Instruction prompt:** describes a specific task to perform. it is scoped to the current step/turn and it is subordinate to the system message when they conflict. Typical contents usually are task descriptions, inputs, and desired outputs for the tasks.

At inference, the API concatenates messages (System -> Developer -> user->assistant history) into a single token sequence. "Priority" is learned behavior, not a hard rule.

3.2.2 Experiment tracking

The experimental design employed a systematic sampling approach to ensure statistical validity while maintaining computational feasibility. A random sample of 300 buggy code instances was selected from the Pytracebugs dataset for each model-prompt combination, resulting in a total of $4 \times 300 \times 3 = 3600$ experimental trials.

To estimate the statistical power for our sample size, we approximated the paired comparison using Cohen's d and a normal distribution. For $n = 300$ paired instances per condition, a family-wise significance level $\alpha_{\text{family}} = 0.01$ with $K = 4$ comparisons yields an individual threshold

$$\alpha_{\text{indiv}} = \frac{\alpha_{\text{family}}}{K} = 0.0025,$$

corresponding to

$$z_{1-\alpha_{\text{indiv}}/2} \approx 3.09.$$

Assuming a small-to-medium effect size of $d \approx 0.3$, the non-central parameter is

$$\text{ncp} = d \cdot \sqrt{n} \approx 5.196.$$

The resulting approximate power is

$$\text{Power} \approx 1 - \Phi(z_{1-\alpha_{\text{indiv}}/2} - \text{ncp}) \approx 0.982,$$

indicating that the chosen sample size is sufficient to detect meaningful effects while keeping computational cost reasonable.

To ensure statistical rigor in our evaluation, we conducted a power analysis to verify that our sample size of 300 paired instances per condition was adequate. We used a family-wise significance threshold of $\alpha = 0.01$ for confirmatory comparisons, applying corrections for multiple testing across models. With this setup, 300 samples provide high power (approximately 0.98) to detect effects of small-to-medium size (Cohen's $d \approx 0.3$), which matches the expected effect sizes in this task. Detecting very small effects ($d \approx 0.2$) would require larger samples, but our chosen size balances computational cost and statistical validity, ensuring that meaningful differences can be detected with strong confidence.

For each response, the following evaluation metrics were computed: AST similarity, AST-normalized score, CodeBLEU, N-gram overlap, weighted N-gram match, syntax correctness, and dataflow preservation. Each metric was evaluated against both the original buggy code and the ground truth correction to provide comprehensive performance assessment.

3.3 Evaluation Method

The evaluation framework employs multiple complementary metrics to assess the quality of generated code repairs across different dimensions: structural similarity, semantic preservation, and syntactic correctness. This multi-faceted approach ensures comprehensive assessment of LLM performance in automated program repair tasks.

3.3.1 Text Similarity Metrics

Both AST-based metrics described below are text similarity scores computed on the string representations of Abstract Syntax Trees, rather than direct tree structure comparisons.

3.3.1.1 AST Text Score

The Abstract Syntax Tree (AST) (Fischer; Lusiardi; Wolff von Gudenberg, 2007) score measures the structural similarity between the generated repair code and the ground truth correction by comparing the text representations of their parsed trees. This metric is particularly relevant for our type of problem since it captures the fundamental program structure independently of formatting, variable names, or comment variations.

The AST score is computed by:

1. Parsing both the generated code and ground truth into their respective AST representations;
2. Converting the ASTs to their string representations using `ast.dump()`;
3. Computing the text similarity between these string representations using Sequence Matcher;

A score of 1.0 indicates perfect structural similarity, while 0.0 represents completely different program structures. This metric is sensitive to control flow changes, function structure modifications, and overall program architecture, making it essential for evaluating whether the LLM has preserved the intended program logic.

3.3.1.2 AST Normalized Text Score

The AST normalized score addresses a fundamental limitation of the raw AST score by normalizing identifiers and constants to focus purely on structural similarity. This metric is particularly valuable in automated program repair as it distinguishes between structural changes and mere naming variations, which are often irrelevant to the actual repair quality.

The normalization process involves:

1. Parsing both the generated code and ground truth into AST representations
2. Applying a normalization transformation that:

- Replaces variable names with normalized identifiers (`_var_1`, `_var_2`, etc.)
- Replaces function names with normalized identifiers (`_func_1`, `_func_2`, etc.)
- Replaces class names with normalized identifiers (`_class_1`, `_class_2`, etc.)
- Replaces literal constants with a placeholder (`_const_`)

3. Converting the normalized ASTs to their string representations using `ast.dump()`
4. Computing text similarity between the normalized AST string representations using Sequence Matcher

This approach is crucial for automated program repair evaluation because it focuses on the essential structural logic rather than superficial naming differences. For example, a repair that changes variable names from `user_input` to `input_data` would receive a perfect normalized AST score, while maintaining the same structural integrity. This metric is particularly valuable for identifying whether LLMs are generating repairs that preserve the intended program logic and control flow, regardless of identifier choices.

To illustrate the difference between AST and AST normalized representations, consider the following example of a Python function:

Original AST

```
Module(
  body=[
  FunctionDef(
    name='check_status',
    args=arguments(
      posonlyargs=[],
      args=[
        arg(arg='monster'),
        arg(arg='status_name')],
      kwonlyargs=[],
      kw_defaults=[],
      defaults=[]),
    body=[
      Return(
        value=Call(
          func=Name(id='any', ctx=Load()),
          args=[
            GeneratorExp(
              elt=Name(id='t', ctx=Load()),
              generators=[
                comprehension(
                  target=Name(id='t', ctx=Store()),
                  iter=Attribute(
                    value=Name(id='monster', ctx=Load()),
                    attr='status',
                    ctx=Load()),
                  ifs=[
                    Compare(
                      left=Attribute(
                        value=Name(id='t', ctx=Load()),
                        attr='name',
                        ctx=Load()),
                      ops=[
                        Eq()],
                      comparators=[
                        Name(id='status_name', ctx=Load())
    ])],
                  is_async=0)])],
          keywords=[]))],
    decorator_list=[])],
  type_ignores=[])
```

AST Normalized

```
Module(
  body=[
  FunctionDef(
    name='_func_1',
    args=arguments(
      posonlyargs=[],
      args=[
        arg(arg='monster'),
        arg(arg='status_name')],
      kwonlyargs=[],
      kw_defaults=[],
      defaults=[]),
    body=[
      Return(
        value=Call(
          func=Name(id='_var_1', ctx=Load()),
          args=[
            GeneratorExp(
              elt=Name(id='_var_2', ctx=Load()),
              generators=[
                comprehension(
                  target=Name(id='_var_2', ctx=Store()),
                  iter=Attribute(
                    value=Name(id='_var_4', ctx=Load()),
                    attr='status',
                    ctx=Load()),
                  ifs=[
                    Compare(
                      left=Attribute(
                        value=Name(id='_var_2', ctx=Load()),
                        attr='name',
                        ctx=Load()),
                      ops=[
                        Eq()],
                      comparators=[
                        Name(id='_var_6', ctx=Load())])],
                  is_async=0)])],
          keywords=[]))],
    decorator_list=[])],
  type_ignores=[])
```

FIGURE 3.3 – Side-by-side comparison of AST and AST normalized representations for the same Python function. The normalized version replaces specific identifiers with generic placeholders (`_func_1`, `_var_1`, etc.) while preserving the structural logic.

As shown in Figure 3.3, the AST normalized representation replaces specific identifiers like `check_status`, `any`, `monster`, `status_name`, and `t` with generic placeholders (`_func_1`, `_var_1`, `_var_4`, `_var_6`, and `_var_2` respectively). This normalization allows the metric to focus purely on structural similarity, ignoring naming variations that are irrelevant to the actual repair quality.

The AST normalized score provides a more robust assessment of repair quality by isolating structural changes from naming conventions, making it essential for evaluating whether the core program architecture and logic flow have been preserved in the generated repair.

3.3.2 Structural Similarity Metrics

3.3.2.1 Syntax Match

3.3.3 Dataflow Preservation

3.3.3.1 Dataflow

3.3.4 Semantic Similarity Metrics

3.3.4.1 Workflow N-gram

3.3.4.2 CodeBLEU

3.4 Implementation Details

3.4.1 Model Execution

All API calls to the proprietary models (GPT-4o and Claude 3.5 Sonnet) and the open-source models (Qwen 2.5 and Codestral) were managed through the **OpenRouter** API aggregation service. This approach was chosen to ensure a consistent and reproducible experimental setup across all models from a single interface.

The specific model identifiers used on the platform are listed below. To ensure that the results were as deterministic as possible and to facilitate a fair comparison, the generation parameters were kept constant for all API calls: a **temperature of 0** and a **top_p of 1.0** were used. All experiments were conducted in **July 2025**.

3.4.2 Retry Logic and Monitoring

Our project implements a robust error handling mechanism, to ensure reliable operation. We captured and logged all API failures, and employed an exception handling strategy. So when individual API requests failed, then the system gracefully handles these exceptions by storing error messages in a result array, alowing the experiment to continue processing subsequent samples while maintaining a complete audit trail of all failures.

Also, the system utilizes structured logging to track metric calculation anomalies, specifically for CodeBLEU. We captured detailed information about dataflow extraction problems that could affect evaluation accuracy.

4 Results

4.1 Experimental Results Summary

Table 4.1 presents the comprehensive results of our experiment, showing the performance of all four models across the three prompt conditions for each evaluation metric. The results are formatted as mean ± standard deviation to provide both central tendency and variability measures.

TABLE 4.1 – Comprehensive metrics comparison across all models and prompt conditions (mean ± std).

Model	**Prompt**	**AST Score**	**Text Score**	**AST Norm.**	**CodeBLEU**	**N-gram**	**W. N-gram**	**Syntax**	**Dataflow**
Claude 3.5 Sonnet	Baseline	0.60 ± 0.34	0.72 ± 0.23	0.61 ± 0.34	0.73 ± 0.17	0.64 ± 0.27	0.71 ± 0.24	0.76 ± 0.19	0.71 ± 0.26
	Style-aware	0.61 ± 0.34	0.72 ± 0.23	0.62 ± 0.33	0.73 ± 0.17	0.64 ± 0.26	0.72 ± 0.24	0.77 ± 0.19	0.71 ± 0.26
	System	0.55 ± 0.37	0.72 ± 0.22	0.57 ± 0.37	0.73 ± 0.17	0.65 ± 0.26	0.71 ± 0.24	0.76 ± 0.19	0.72 ± 0.26
Codestral 2501	Baseline	0.76 ± 0.28	0.79 ± 0.22	0.77 ± 0.27	0.78 ± 0.19	0.73 ± 0.26	0.77 ± 0.24	0.81 ± 0.19	0.74 ± 0.27
	Style-aware	0.76 ± 0.28	0.80 ± 0.22	0.77 ± 0.27	0.78 ± 0.18	0.73 ± 0.26	0.77 ± 0.24	0.81 ± 0.19	0.75 ± 0.26
	System	0.76 ± 0.29	0.80 ± 0.22	0.77 ± 0.27	0.77 ± 0.18	0.73 ± 0.26	0.77 ± 0.24	0.81 ± 0.19	0.74 ± 0.27
GPT-4o	Baseline	0.70 ± 0.30	0.75 ± 0.26	0.73 ± 0.29	0.72 ± 0.22	0.65 ± 0.29	0.72 ± 0.28	0.77 ± 0.22	0.69 ± 0.29
	Style-aware	0.73 ± 0.29	0.77 ± 0.24	0.74 ± 0.28	0.75 ± 0.20	0.68 ± 0.28	0.74 ± 0.26	0.79 ± 0.21	0.71 ± 0.28
	System	0.74 ± 0.29	0.78 ± 0.23	0.76 ± 0.26	0.75 ± 0.19	0.69 ± 0.27	0.75 ± 0.25	0.80 ± 0.19	0.71 ± 0.27
Qwen 2.5 Coder 32B	Baseline	0.74 ± 0.29	0.79 ± 0.22	0.75 ± 0.27	0.76 ± 0.19	0.70 ± 0.28	0.75 ± 0.25	0.80 ± 0.20	0.73 ± 0.27
	Style-aware	0.70 ± 0.32	0.77 ± 0.21	0.72 ± 0.31	0.77 ± 0.18	0.71 ± 0.26	0.75 ± 0.24	0.80 ± 0.19	0.74 ± 0.26
	System	0.73 ± 0.29	0.78 ± 0.23	0.75 ± 0.28	0.76 ± 0.19	0.70 ± 0.27	0.75 ± 0.24	0.80 ± 0.19	0.74 ± 0.27

The results in Table 4.1 reveal several key insights about model performance and prompt sensitivity:

- **Model Performance Ranking**: Codestral 2501 consistently achieves the highest scores across most metrics, followed by Qwen 2.5 Coder 32B, GPT-4o, and Claude 3.5 Sonnet

- **Prompt Sensitivity**: The style-aware prompt (P2) shows modest improvements in some metrics, particularly for Qwen 2.5 Coder 32B in text score (0.80 ± 0.22 vs baseline 0.79 ± 0.22)

- **Metric Consistency**: AST scores show the highest variability (std $\sim$ 0.28-0.37), while syntax match scores demonstrate the most consistency (std $\sim$ 0.19-0.22)

- **Performance Stability**: Codestral 2501 shows the most consistent performance across prompt variations, suggesting robustness to prompt engineering changes

These results provide a comprehensive foundation for understanding the effectiveness of different prompt engineering strategies and their impact on LLM-based automated program repair performance.

4.1.0.1 Statistical Significance Analysis

To evaluate the observed performance differences, we conducted a statistical test using the Wilcoxon signed-rank test with multiple comparison corrections. Our analysis focused on identifying statistically significant differences ($p < 0.01$) across prompt conditions while controlling for multiple testing effects through Bonferroni correction. The statistical analysis reveals several critical insights that complement the descriptive statistics presented above. We decided to choose the Wilcoxon test, since each metric value has a range of $[0, 1]$, so they are not normally distributed. Also, there is a typical asymmetry in these similarity tests, with the presence of outliers. So the Wilcoxon test can be more robust than a paired t-test.

The statistical analysis reveals significant performance differences across models. Codestral 2501 consistently outperforms all other models in both AST normalized and CodeBLEU scores across all prompt conditions, with p-values as low as 4.19×10^{-23} and effect sizes ranging from 0.185 to 0.738. Claude 3.5 Sonnet shows the weakest performance, with significantly lower scores compared to all other models. GPT-4o demonstrates intermediate performance, while Qwen 2.5 Coder 32B shows competitive results, particularly in CodeBLEU metrics.

4.1.0.2 Statistical Results Summary

Table 4.2 presents the comprehensive statistical analysis results, showing all significant model comparisons ($p < 0.01$) across different prompt conditions and metrics. The table includes p-values, effect sizes, and win/loss percentages to provide a complete picture of model performance differences.

Table 4.2 shows that Codestral 2501 achieves the highest win rates (36-55%) across all model comparisons, with effect sizes ranging from 0.673 to 0.738. Claude 3.5 Sonnet shows the lowest win rates (15-27%), while GPT-4o and Qwen 2.5 Coder 32B demonstrate intermediate and competitive performance respectively.

Insira Conclusão

TABLE 4.2 – Statistical significance results for model comparisons across prompt conditions (p < 0.01).

Model A	Model B	Metric	Prompt	P-value	Effect Size	Wins %	Ties %	Losses %
Claude 3.5 Sonnet	Codestral 2501	AST Norm.	Baseline	4.19×10^{-23}	0.185	16.0	13.3	70.7
			System	1.73×10^{-23}	0.195	16.7	14.3	69.0
			Style-aware	6.92×10^{-22}	0.185	15.7	15.3	69.0
	Codestral 2501	CodeBLEU	Baseline	7.54×10^{-20}	0.226	20.3	10.0	69.7
			System	3.10×10^{-19}	0.220	19.7	10.7	69.7
			Style-aware	7.67×10^{-20}	0.223	19.3	13.3	67.3
	GPT-4o	AST Norm.	Baseline	1.09×10^{-10}	0.328	29.7	9.7	60.7
			System	8.59×10^{-18}	0.274	24.3	11.3	64.3
			Style-aware	1.27×10^{-13}	0.271	23.3	14.0	62.7
	GPT-4o	CodeBLEU	Baseline	2.33×10^{-3}	0.384	36.0	6.3	57.7
			System	5.44×10^{-7}	0.358	32.7	8.7	58.7
			Style-aware	1.44×10^{-7}	0.341	30.0	12.0	58.0
	Qwen 2.5 Coder 32B	AST Norm.	Baseline	1.32×10^{-14}	0.261	22.3	14.3	63.3
			System	3.41×10^{-14}	0.282	23.7	16.0	60.3
			Style-aware	8.23×10^{-10}	0.310	25.7	17.3	57.0
	Qwen 2.5 Coder 32B	CodeBLEU	Baseline	8.19×10^{-12}	0.301	26.7	11.3	62.0
			System	2.11×10^{-13}	0.282	24.3	13.7	62.0
			Style-aware	2.06×10^{-10}	0.318	27.7	13.0	59.3
Codestral 2501	GPT-4o	AST Norm.	Baseline	3.65×10^{-9}	0.710	51.3	27.7	21.0
			System	9.89×10^{-5}	0.658	43.7	33.7	22.7
			Style-aware	6.95×10^{-6}	0.667	44.7	33.0	22.3
	GPT-4o	CodeBLEU	Baseline	9.61×10^{-14}	0.738	55.3	25.0	19.7
			System	4.89×10^{-10}	0.699	48.7	30.3	21.0
			Style-aware	8.18×10^{-8}	0.679	47.3	30.3	22.3
	Qwen 2.5 Coder 32B	AST Norm.	Baseline	1.56×10^{-5}	0.673	36.3	46.0	17.7
			System	1.40×10^{-5}	0.675	37.3	44.7	18.0
			Style-aware	4.58×10^{-6}	0.673	38.3	43.0	18.7
	Qwen 2.5 Coder 32B	CodeBLEU	Baseline	1.81×10^{-6}	0.678	39.3	42.0	18.7
			System	1.26×10^{-6}	0.685	42.0	38.7	19.3
			Style-aware	3.53×10^{-8}	0.702	41.7	40.7	17.7
GPT-4o	Qwen 2.5 Coder 32B	CodeBLEU	Baseline	1.74×10^{-4}	0.404	30.0	25.7	44.3
			System	-	-	-	-	-
			Style-aware	-	-	-	-	-

5 Conclusão

Conclua

References

AKIMOVA, E. N.; BERSENEV, A. Y.; DEIKOV, A. A.; KOBYLKIN, K. S.; KONYGIN, A. V.; MEZENTSEV, I. P.; MISILOV, V. E. PyTraceBugs: A Large Python Code Dataset for Supervised Machine Learning in Software Defect Prediction. *In:* 2021 28th Asia-Pacific Software Engineering Conference (APSEC). [*S.l.: s.n.*], 2021. p. 141–151. DOI: `10.1109/APSEC53868.2021.00022`. Cit. on p. 16.

FISCHER, G.; LUSIARDI, J.; WOLFF VON GUDENBERG, J. Abstract Syntax Trees - and their Role in Model Driven Software Development. *In:* INTERNATIONAL Conference on Software Engineering Advances (ICSEA 2007). [*S.l.: s.n.*], 2007. p. 38–38. DOI: `10.1109/ICSEA.2007.12`. Cit. on p. 22.

HUI, B.; YANG, J.; CUI, Z.; YANG, J.; LIU, D.; ZHANG, L.; LIU, T.; ZHANG, J.; YU, B.; LU, K.; DANG, K.; FAN, Y.; ZHANG, Y.; YANG, A.; MEN, R.; HUANG, F.; ZHENG, B.; MIAO, Y.; QUAN, S.; FENG, Y.; REN, X.; REN, X.; ZHOU, J.; LIN, J. **Qwen2.5-Coder Technical Report**. [*S.l.: s.n.*], 2024. arXiv: `2409.12186 [cs.CL]`. Available from: `https://arxiv.org/abs/2409.12186`. Cit. on p. 18.

KULKARNI, A. A.; NIRANJAN, D. G.; SAJU, N.; SHENOY, P. R.; ARYA, A. Graph-Based Fault Localization in Python Projects with Class-Imbalanced Learning. *In:* SPRINGER. INTERNATIONAL Conference on Engineering Applications of Neural Networks. [*S.l.: s.n.*], 2024. p. 354–368. Cit. on p. 16.

MISTRALAI. **Codestral 2501**. 2025. Available from: `https://mistral.ai/news/codestral-2501`. Visited on: 25 Jan. 2025. Cit. on p. 18.

WANG, Y.; JIANG, T.; LIU, M.; CHEN, J.; MAO, M.; LIU, X.; MA, Y.; ZHENG, Z. **Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models**. [*S.l.: s.n.*], 2025. arXiv: `2407.00456 [cs.SE]`. Available from: `https://arxiv.org/abs/2407.00456`. Cit. on p. 20.

ZHAO, Y.; GONG, L.; HUANG, Z.; WANG, Y.; WEI, M.; WU, F. Coding-ptms: How to find optimal code pre-trained models for code embedding in vulnerability detection? *In:* PROCEEDINGS of the 39th IEEE/ACM International Conference on Automated Software Engineering. [*S.l.: s.n.*], 2024. p. 1732–1744. Cit. on p. 16.

FOLHA DE REGISTRO DO DOCUMENTO

$^{1.}$ CLASSIFICAÇÃO/TIPO DM	$^{2.}$ DATA 25 de março de 2015	$^{3.}$ DOCUMENTO Nº DCTA/ITA/DM-018/2015	$^{4.}$ Nº DE PÁGINAS 30

$^{5.}$ TÍTULO E SUBTÍTULO:
Large Language Models for Automated Program Repair: An Empirical Study with Tracebacks and Prompt Engineering

$^{6.}$ AUTOR(ES):
Fernando Gusmão Zanchitta

$^{7.}$ INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):
Instituto Tecnológico de Aeronáutica – ITA

$^{8.}$ PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:
Cupim; Cimento; Estruturas

$^{9.}$ PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:
Cupim; Dilema; Construção

$^{10.}$ APRESENTAÇÃO: ( ) **Nacional** (**X**) **Internacional**
ITA, São José dos Campos. Curso de Mestrado. Programa de Pós-Graduação em Engenharia Aeronáutica e Mecânica. Área de Sistemas Aeroespaciais e Mecatrônica. Orientador: Prof. Dr. Adalberto Santos Dupont. Coorientadora: Prof^{a}. Dr^{a}. Doralice Serra. Defesa em 05/03/2015. Publicada em 25/03/2015.

$^{11.}$ RESUMO:
Placeholder abstract

$^{12.}$ GRAU DE SIGILO:
(**X**) **OSTENSIVO** ( ) **RESERVADO** ( ) **SECRETO**