Towards Continuous Evaluation of Prompt Engineering Strategies for Automated Unit Test Generation Using LLMs

Brena Marques Ribeiro
ICMC/USP
São Carlos, SP, Brazil
brena.marques@usp.br

Vânia de Oliveira Neves
UFF
Niteroi, RJ, Brazil
vania@ic.uff.br

Simone R. S. Souza
ICMC/USP
São Carlos, SP, Brazil
srocio@icmc.usp.br

Abstract

Automated unit test generation using Large Language Models (LLMs) has attracted increasing attention in Software Engineering for its potential to reduce manual effort and improve productivity during the software development process. However, the rapid evolution of generative models and the diversity of prompt engineering strategies make it challenging to conduct continuous, reproducible, and comparable analyses over time. In this context, this study presents the proposal and initial development of PromptUnitLab, a prototype tool designed for the continuous execution, monitoring, and comparison of prompting strategies applied to automated unit test generation. The main purpose is to provide an extensible environment that keeps pace with the evolution of generative AI models while reducing the manual effort required for experiment configuration, execution, and analysis. The prototype was developed in Python and integrates with language models via APIs, enabling comparison of different prompting techniques using metrics such as code coverage, latency, and computational cost. In this initial phase of the research, preliminary functionalities were implemented for dynamic model configuration, comparative visualization of results, and storage of experimental history. Initial results demonstrate the proposed approach's ability to support continuous analysis and experimental comparisons across different unit test generation configurations. Therefore, it is expected that PromptUnitLab will contribute as a decision support tool for experiments involving Learning Language Models (LLMs), fostering future research in test automation and the continuous evaluation of generative models in Software Engineering.

Keywords

Large Language Models, Prompt Engineering, Software Testing, Automated Test Generation, Unit Testing, Generative Artificial Intelligence, Software Engineering.

1 Introduction

Software testing is a fundamental yet challenging activity in the software development lifecycle, requiring both a deep understanding of system requirements and proficiency in the programming language and development environment. As one of the pillars of Software Engineering, software testing aims to ensure the reliability, functionality, and quality of software systems [17]. However, creating representative and effective test cases remains a time-consuming and error-prone task, particularly given the growing complexity of modern systems [7]. This challenge has encouraged the adoption of automated and intelligent approaches to support test case generation. Unit testing focuses on the smallest components of a system, such as functions, methods, and classes, to identify errors in logic, data structures, and implementation [6]. By enabling the isolated validation of components during development, unit testing provides a suitable scenario for evaluating automated approaches.

Techniques based on Artificial Intelligence (AI) have emerged as a natural evolution, offering greater adaptability, predictive capabilities, and the potential to generate more realistic test data [13][7]. AI encompasses a broad range of approaches, including rule-based systems, traditional machine learning models such as decision trees and neural networks, and, more recently, generative models. Among these, generative adversarial networks (GANs) and autoregressive models stand out. *LLMs* have gained significant attention due to their ability to understand, generate, and transform natural language at scale, making them promising for tasks such as automated test case generation and script creation [3].

Despite these advances, the practical adoption of techniques such as prompt engineering still faces important challenges, especially in selecting the most suitable approach. Moreover, the rapid evolution of generative AI models introduces an additional challenge for experimental research in this field: approaches, metrics, and results can quickly become outdated due to the constant updates of available models. Consequently, there is a growing need for tools capable of performing continuous, reproducible, and extensible evaluations, enabling researchers to monitor the evolution of prompting techniques and language models over time.

To address these challenges, this work proposes developing **PromptUnitLab**, a framework that supports continuous experimentation with automated unit test generation using LLMs. This framework enables comparative analyses across different prompt engineering strategies, language models, and experimental metrics while reducing the manual effort required for experiment configuration and evaluation. The **PromptUnitLab** framework considers Python programs because this programming language is widely adopted in both academia and industry and is well integrated with AI frameworks and software testing ecosystems [15].

Therefore, this work intends to answer the question: *"Which prompting technique presents better results for unit testing generation?"*. The proposed framework aims to support comparative analyses of prompting techniques by properly using metrics such as coverage, cost, efficiency, and latency.

This paper is organized as follows: Section 2 presents the main related work. Section **??** describes the project design of this research. Finally, Section 4 concludes the paper and presents opportunities for future work.

2 Related Work

This section presents the main studies related to automated unit test generation using LLMs. The studies are presented grouped by

main themes related to our research, with the limitations found discussed. Finally, the research gaps that motivate the development of the PromptUnitLab framework are discussed.

2.1 Benchmarking Platforms and LLM Evaluation Environments

The growing adoption of Large Language Models (LLMs) has provoked the development of benchmarking platforms to compare model performance across a range of tasks. Among these initiatives, arena-based evaluation environments, such as Arena.ai [1], enable users to compare outputs from multiple LLMs under similar conditions, thereby supporting model ranking and performance assessment.

These platforms play an important role in helping practitioners and researchers evaluate and select models. However, their primary focus is on comparing different LLMs rather than on systematically evaluating prompt engineering strategies applied to the same model. As a result, they provide limited support for studies investigating how different prompting techniques influence the quality of generated software artifacts, particularly in the context of automated unit test generation.

Furthermore, despite the increasing popularity of benchmarking environments, the scientific literature still shows a limited number of studies dedicated to reusable infrastructures for continuous experimentation and comparison of prompting techniques. This limitation becomes even more relevant in domains where model behavior is highly sensitive to prompt design decisions.

2.2 LLMs for Automated Unit Test Generation

The rapid growth of research on LLM-based software testing is evidenced by the systematic literature review conducted by Augusto et al. [2]. The authors provide a comprehensive roadmap of the field by organizing existing studies into different research categories and identifying current trends. Their findings indicate that approximately 65% of the selected studies were published between 2024 and 2025, highlighting the recent expansion and ongoing consolidation of this research area.

Several frameworks have been proposed to improve the effectiveness of automated unit test generation using LLMs. Among them, ChatUniTest [4] and CasModaTest [12] adopt iterative workflows that combine test generation, validation, and refinement stages. These approaches leverage feedback mechanisms to improve the correctness and quality of generated tests.

Other studies focus on increasing code coverage and exploring complex execution paths. For example, HITS [18] introduces a method slicing strategy to guide the test generation process, while Code-Aware Prompting [14] incorporates structural information extracted from the source code into prompts to improve test generation effectiveness.

The literature also includes approaches targeting specialized testing scenarios. Examples include ExLong [19], which focuses on generating tests for exceptional behaviors, property-based testing approaches [16], and methods for regression test generation from software commits [9].

Although these studies differ in their architectures, workflows, and optimization strategies, their primary objective is to improve the quality and effectiveness of generated tests. Consequently, they do not focus on providing reusable experimental infrastructures capable of systematically comparing different prompting strategies under controlled conditions.

2.3 Prompt Engineering Strategies for Test Generation

Beyond developing new frameworks, a significant portion of the literature investigates how prompt engineering techniques affect the performance of LLM-based test generation systems. According to the systematic review by Augusto et al. [2], approaches based on Few-Shot Prompting, Fine-Tuning, and iterative feedback mechanisms are among the most frequently adopted strategies.

Coverage-guided prompting techniques have also gained increasing attention for their ability to improve execution coverage and encourage exploration of complex code paths. Similarly, iterative refinement approaches attempt to improve generated tests by incorporating feedback from previous executions into subsequent prompt iterations.

Figure 1 summarizes the distribution of prompting and training strategies identified in the literature review conducted by Augusto et al. [2].

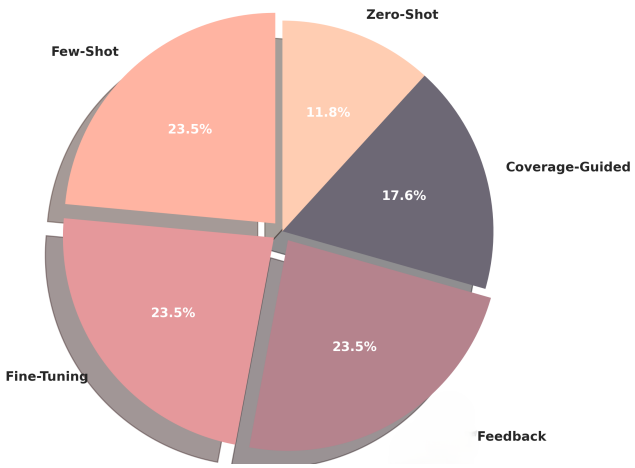

Figure 1: Distribution of prompting and training strategies identified in the literature [2].

The greater number of these techniques demonstrates interest in understanding how prompt design influences the quality of test generation. Nevertheless, most studies evaluate prompting strategies within isolated experimental settings, making it difficult to reproduce results and compare techniques under consistent conditions.

2.4 Research Gap

Despite the substantial progress achieved by existing frameworks and prompting strategies, an important gap remains in the literature. Most studies focus on proposing new methods for test generation or evaluating specific techniques within a fixed experimental setup. Comparisons between prompting strategies are typically conducted

within individual studies and are not supported by reusable infrastructure that facilitates continuous experimentation.

Additionally, the rapid evolution of generative AI models introduces significant reproducibility challenges. Experimental results can quickly become outdated due to model updates, newly released architectures, and changes in model behavior over time. Consequently, reproducing and extending previous experiments often requires substantial manual effort.

These limitations highlight the need for extensible and reusable experimental environments that can systematically compare prompt engineering techniques, track performance over time, and support reproducible experimentation for automated unit test generation.

To address these limitations, this work proposes PromptUnitLab, a framework that supports continuous, comparative, and reproducible evaluations of prompt engineering techniques for automated unit test generation with LLMs.

To our knowledge, this framework is innovative in providing a reusable experimental environment for systematically assessing prompting strategies under controlled conditions. By enabling reproducible experiments and supporting the integration of new models and techniques over time, the framework aims to facilitate more consistent and extensible research on LLM-based software testing.

3 The PromptUnitLab Framework

This paper presents an ongoing project that follows a Design Science Research methodology, whose primary objective is to design and evaluate a software artifact to address the lack of reusable infrastructures for continuously comparing prompt engineering techniques applied to automated unit test generation using LLMs.

The research is conducted in three phases: (i) a literature review starting from the comprehensive systematic review by Augusto et al. [2], from which we selected the subset of studies directly related to *unit test generation* and *prompt engineering*, complemented by additional works on evaluation metrics; (ii) design and prototyping of the **PromptUnitLab** framework based on identified gaps; and (iii) planned experiments to evaluate the framework, detailed in Section 3.5.

3.1 Framework Overview

PromptUnitLab is an extensible experimental environment for continuous, reproducible, and comparative evaluation of prompt engineering strategies for unit test generation using LLMs. Rather than proposing a new prompting technique, it provides a reusable infrastructure to systematically compare strategies, models, and configurations.

Figure 2 presents the conceptual architecture, organized into four components:

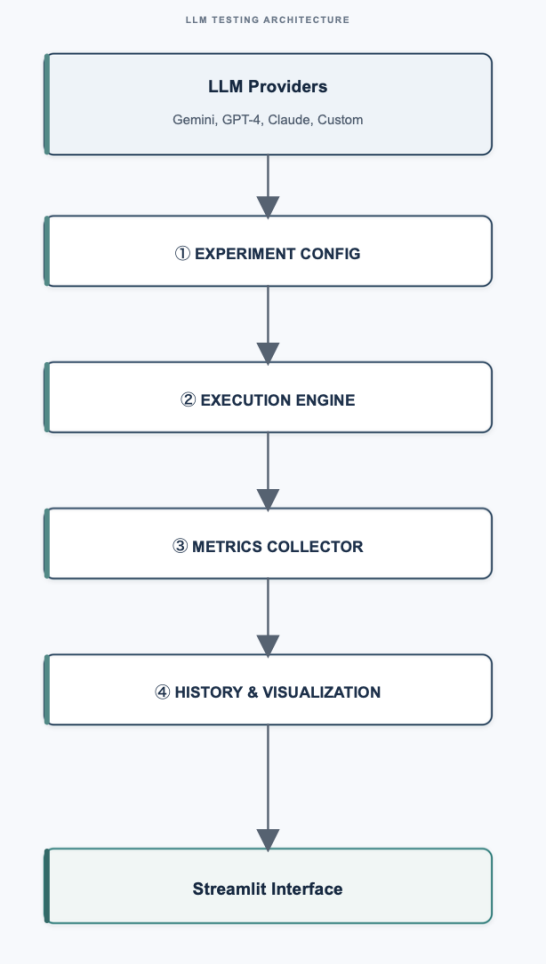

Figure 2: Conceptual architecture of PromptUnitLab.

(1) **Experiment Configuration.** Defines experiment parameters: target functions, prompting techniques, LLM providers, and repetition count. The framework is model-agnostic via a unified API abstraction layer.
(2) **Execution Engine.** Orchestrates the end-to-end pipeline: for each prompt×model combination, it generates tests, executes them in an isolated environment, and triggers metric collection. Batch mode supports sequential or parallel execution.
(3) **Metrics Collector.** Organized in two layers:
   - **Prompt-level** (via **Langfuse** [8]): captures latency, token consumption (input/output), computational cost (tokens × provider pricing), and model metadata (version, parameters, timestamp) for each LLM call.
   - **Code-level** (via **DeepEval** [5]): computes line and branch coverage (via `coverage.py`), execution success rate, and assertion density of generated tests.
(4) **History & Visualization.** Persistent record of all experiments with comparative dashboards and time-series views for tracking performance across model versions.

A key design principle is **extensibility**: abstract interfaces for prompting strategies, LLM providers, and metrics allow new techniques to be integrated without modifying the core execution pipeline.

It is important to emphasize that the **main contribution is the paradigm of continuous evaluation**, not the tool itself. By enabling systematic comparison over time, PromptUnitLab shifts focus from one-shot comparisons,which quickly become outdated,to ongoing monitoring across model versions, API changes, and evolving benchmarks.

3.2 Research Questions

Based on literature gaps, this work investigates how prompting techniques impact test generation and whether PromptUnitLab provides a practical environment for continuous experiments. Questions are divided into experimental evaluation and tool validation, each explained with its planned answering approach.

3.2.1 *Experimental Evaluation RQs.*

- **RQ1:** How do different prompting techniques impact code coverage, latency, and computational cost?
  Answer: Each technique (zero-shot, few-shot, chain-of-thought, etc.) will be applied to the same target functions using the same LLM. Metrics from Langfuse (latency, tokens, cost) and DeepEval (coverage) will be compared via descriptive statistics and paired analysis.
- **RQ2:** How do LLM-generated tests compare to traditional automated test generation?
  Answer: Same target functions tested with Pynguin [**?**]. Comparison across all metrics plus syntactic validity and executability.
- **RQ3:** Can continuous execution identify performance variations from model evolution?
  Answer: Identical configurations will be executed periodically over months. Time-series analysis will detect discontinuities coinciding with model updates, validating the continuous evaluation paradigm.

3.2.2 *Tool Validation RQs.*

- **RQ4:** Can the tool support selecting the best configuration for a given task?
  Answer: Evaluate whether dashboard visualizations and ranking features allow identifying optimal configurations per target and criterion.
- **RQ5:** Does the tool reduce experimentation effort versus manual analysis?
  Answer: Controlled study measuring total time for setup, execution, and analysis of multi-technique comparisons with and without PromptUnitLab.
- **RQ6:** Can new techniques and models be added without structural modifications?
  Answer: Measure lines of code changed and time to integrate a new technique and provider, verifying no changes to core execution engine.

3.3 Evaluation Metrics

Metrics are organized into two categories reflecting the two evaluation dimensions: *how* the prompting technique performs as an LLM interaction and *what* it produces as a testing artifact.

3.3.1 *Prompt-Level (via Langfuse).*

- **Latency:** end-to-end time from prompt submission to test code receipt (trace duration per Langfuse call).
- **Token Consumption:** input (prompt) and output (completion) tokens as reported by the LLM provider.
- **Computational Cost:** tokens × provider-specific per-token pricing; Langfuse supports automatic estimation.
- **Model Metadata:** version, generation parameters (temperature, top-p, max tokens), and timestamp for reproducibility and drift analysis.

Covers RQ1 (technique comparison) and RQ3 (temporal variation).

3.3.2 *Code-Level (via DeepEval).*

- **Line Coverage:** percentage of executable lines exercised by tests (via `coverage.py`).
- **Branch Coverage:** percentage of decision branches exercised.
- **Execution Success Rate:** proportion of tests that compile, execute, and pass.
- **Assertion Density:** assertions per test case as proxy for test thoroughness.

Covers RQ1 (test quality) and RQ2 (comparison with traditional methods).

3.3.3 *Current Status.*

At this stage, **only the architecture design and non-functional UI prototypes have been developed**. The UI mockups (Section 3.4) illustrate intended screens, but integration with Langfuse and DeepEval has not yet been implemented. The experimental agenda (Section 3.5) represents planned validation. The architecture is explicitly designed for these integrations, but concrete implementations remain future work.

3.4 Prototype Implementation

A prototype was developed to demonstrate architectural feasibility. The selected language is **Python** (widespread in academia, industry, and AI testing ecosystems [15]). Initial experiments use **Gemini 3** for its code-generation performance and API availability.

The prototype offers a **graphical user interface** illustrating the intended workflow. It is important to emphasize these are **non-functional UI prototypes**: they present layout, navigation, and mock data, but backend integration with Langfuse, DeepEval, and the LLM execution pipeline has not been implemented.

Figures 3–5 present the experiment configuration, metrics dashboard, and comparison views. Through the configuration interface (Figure 6), users can register API keys. The execution history module (Figure 7) records past runs and results.

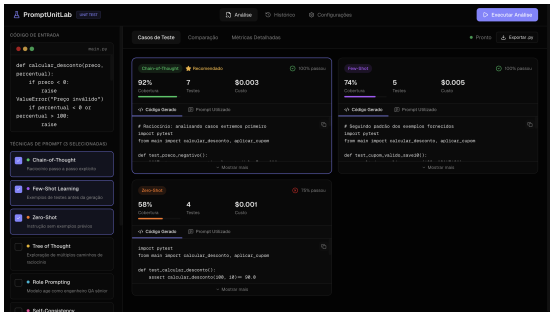

Figure 3: Main interface (UI prototype – non-functional).

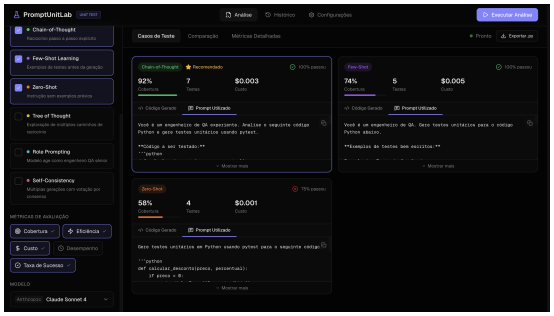

Figure 4: Metrics dashboard (UI prototype – non-functional).

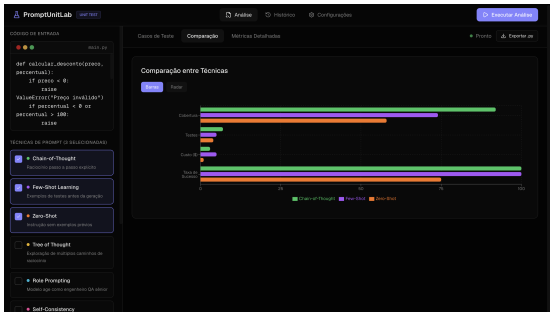

Figure 5: Comparison view (UI prototype – non-functional).

3.5 Experimental Agenda

The experiments follow these steps: (1) selection of open-source Python projects; (2) application of prompting techniques via PromptUnitLab; (3) execution in isolated environment; (4) collection of metrics (Langfuse + DeepEval); (5) comparative analysis; (6) comparison with traditional tools (e.g., Pynguin [**?**]). Comparison considers coverage, cost, execution time, and test validity.

3.6 Methodology Summary

The research validates the **continuous evaluation paradigm**: the framework defines *what* the artifact does, the metrics (Langfuse + DeepEval) define *how* performance is measured, and the experimental agenda outlines *how* validation proceeds. The main contribution is enabling researchers to track prompting technique

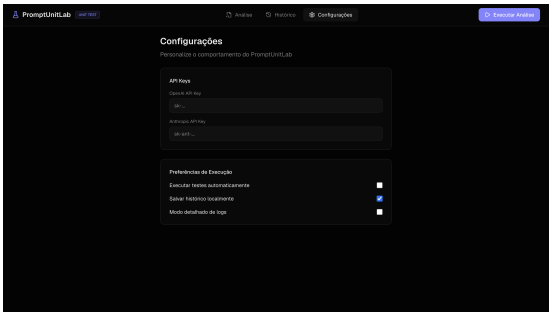

Figure 6: Configuration interface (UI prototype – non-functional).

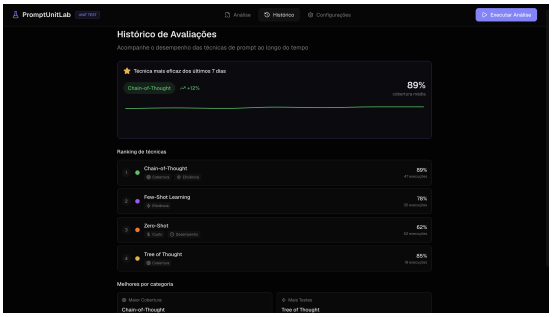

Figure 7: Execution history (UI prototype – non-functional).

performance over time,across model updates, API changes, and evolving benchmarks,rather than relying on one-shot comparisons.

4 Conclusion and Future Work

This work addressed the challenges of automated unit test generation using *Large Language Models* (LLMs), with a particular focus on the rapid evolution of generative AI models and prompting strategies. Rather than identifying a single superior prompting technique, the main objective of this research was to develop an extensible, reusable experimental environment that can continuously monitor, execute, and compare different prompt engineering approaches and language models over time.

The main contribution of this research is the proposal of an extensible infrastructure that supports continuous experimentation in a rapidly evolving research area. The proposed approach contributes to both the software engineering research community and practical experimentation with generative AI for software testing, reducing manual effort and improving reproducibility in comparative evaluations.

Despite the obtained contributions, this study presents some limitations. The experiments were restricted to Python and relied on a limited subset of prompting techniques and language models. Additionally, the evaluation focused primarily on quantitative metrics such as coverage, latency, and computational cost, without delving into qualitative aspects such as the maintainability and readability of the generated tests.

This research is ongoing, and the preliminary findings are promising. Our future work will extend this research in several key areas:

(1) Expanding the experimental evaluation to additional programming languages, broader datasets, and more diverse testing scenarios.
(2) Conducting systematic comparisons among different language models, considering performance, cost, reliability, and adaptability over time.
(3) integrating traditional automated testing tools and human-centered evaluations to better assess the practical applicability of generated tests in real software development environments.

Artifact Availability

At this stage of the research, the proposed artifact, PromptUnitLab, is still under development and in the prototyping phase. A project webpage describing the artifact is available at [10, 11]. The source code, experimental package, and related materials are not yet publicly available. The authors intend to release the complete artifact after the development and evaluation phases are completed.

Acknowledgements

The authors thank CNPq, through process numbers 30679/2025-8, 406941/2025-4, and 420025/2023-5, and FAPERJ- Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro, through process E-26/204.379/2025 for their support.

AI-based tools, such as ChatGPT, were used to support code generation, translation, and textual revision of this paper. However, the literature review, experimental design, analyses, and conclusions presented in this work were conducted entirely by the authors.

References

[1] Arena AI. 2026. Arena AI: The Official AI Ranking & LLM Leaderboard. https://arena.ai/
[2] Cristian Augusto, Antonia Bertolino, Guglielmo De Angelis, Francesca Lonetti, and Jesús Morán. 2025. Large language models for software testing: A research roadmap. *arXiv preprint arXiv:2509.25043* (2025).
[3] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit test generation using generative AI: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code*. 54–61.
[4] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 572–576.
[5] Confident AI. [n. d.]. DeepEval: The Open-Source LLM Evaluation Framework. https://deepeval.com/. Accessed: July 16, 2026.
[6] Márcio Eduardo Delamaro, José Carlos Maldonado, and Mário Jino. 2016. *Introdução ao Teste de Software*. Elsevier, Rio de Janeiro. Acesso em: 20 abr. 2026.
[7] Elson Kurian, Daniela Briola, Pietro Braione, and Giovanni Denaro. 2023. Automatically generating test cases for safety-critical software via symbolic execution. *Journal of Systems and Software* 199 (2023), 111629. doi:10.1016/j.jss.2023.111629
[8] Langfuse. 2026. Langfuse: Open Source AI Engineering Platform. https://langfuse.com/. Accessed: 2026-07-16.
[9] Jing Liu, Seongmin Lee, Eleonora Losiouk, and Marcel Böhme. 2025. Can llm generate regression tests for software commits? *arXiv preprint arXiv:2501.11086* (2025).
[10] Marques Brena et al. 2026. PromptUnitLab. https://v0-mestrado-theta.vercel.app/. Acesso em: 20 jul. 2026.
[11] Marques Brena et al. 2026. PromptUnitLab Demonstration Video. https://drive.google.com/file/d/1w1l6DSe1UKyuq5k00nW-pMJU4vdF49WQ/view. Accessed: 2026-05-23.
[12] Chao Ni, Xiaoya Wang, Liushan Chen, Dehai Zhao, Zhengong Cai, Shaohua Wang, and Xiaohu Yang. 2024. Casmodatest: A cascaded and model-agnostic self-directed framework for unit test generation. *arXiv preprint arXiv:2406.15743* (2024).
[13] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*. 75–84. doi:10.1109/ICSE.2007.37
[14] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 951–971.
[15] TIOBE Software BV. 2026. TIOBE Programming Community Index. https://www.tiobe.com/tiobe-index/ Índice mensal de popularidade de linguagens de programação baseado em resultados de mecanismos de busca. Acesso em: 20 abr. 2026.
[16] Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. 2023. Can large language models write good property-based tests? *arXiv preprint arXiv:2307.04346* (2023).
[17] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936. doi:10.1109/TSE.2024.3368208
[18] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. Hits: High-coverage llm-based unit test generation via method slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268.
[19] Jiyang Zhang, Yu Liu, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2025. exLong: Generating exceptional behavior tests with large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 1462–1474.