Journal of Software Engineering Research and Development, 2026, 6:1, doi: 10.5753/jserd.2026.7565

Assessing the Effectiveness of Large Language Models in Detecting Semantic Conflicts

Nathalia Barbosa [**Centro de Informática, Universidade Federal de Pernambuco** | ***nfab@cin.ufpe.br***]
Paulo Borba [**Centro de Informática, Universidade Federal de Pernambuco** | ***phmb@cin.ufpe.br***]
Léuson Da Silva [**Polytechnique Montreal** | ***leuson-mario-pedro.da-silva@polymtl.ca***]

Abstract

Semantic conflicts occur when a developer introduces changes to a codebase that unintentionally affect the behavior of changes integrated in parallel by other developers. Since merge tools used in practice cannot detect this type of conflict, complementary tools have been proposed, such as SAM (SemAntic Merge), based on the SMAT approach and relies on the generation and execution of unit tests in Java. Despite showing good conflict detection capabilities, SAM presents a high rate of false negatives (existing conflicts not signaled by it). Part of this problem is due to the natural limitations of unit test generation tools, specifically Randoop and EvoSuite. To understand if these limitations can be overcome by large language models (LLMs), this work proposes, and integrates into SMAT, LUCIA (LLM-based Unit-tests for Conflict Identification & Analysis), a new test generation tool based on the Ollama framework to interface with various local LLMs, including the Llama, Gemma, DeepSeek, and Qwen families. We then explore these models' capability to generate tests, using different interaction strategies, prompts with different contents, and different model parameter configurations. We evaluate the results with two distinct samples: a benchmark with simpler systems, used in related work, and a more significant sample based on complex systems used in practice. Finally, we evaluate the effectiveness of LUCIA in detecting conflicts, comparing the selected LLMs with each other and against the original test generation tools used by SAM: Randoop, Randoop Clean, EvoSuite, and Differential EvoSuite. Our evaluation shows that LLMs effectively detect semantic conflicts, with Llama and Qwen outperforming the other models. While no single model could surpass traditional tools, multiple models combined achieved superior performance in conflict detection. Overall, the new extension identified three additional conflicts undetected by SAM, representing a 200% improvement in identifying unique conflicts compared to the single-model approach (Code Llama 70B) used in our previous work. These results reinforce previous findings that LLMs can generate unit tests that are effective in detecting semantic conflicts, while demonstrating the superior effectiveness of multi-model approaches.

Keywords: *Semantic conflicts, Unit test generation, LLMs.*

1 Introduction

During collaborative software development, it is common for multiple developers to work simultaneously on different parts of the code. In these scenarios, conflicts may arise during the integration (merge) of changes (Zhang et al. (2022)), especially when two developers concurrently modify the same line of code or consecutive lines of a file (Maddila et al. (2021)). These are the so-called textual conflicts, which require manual intervention to be resolved. Usually, it is necessary to review the changes from both developers and decide which version should be kept, or if a combination of the two is necessary.

Although effective in detecting textual conflicts, current integration tools do not identify all types of conflicts. A more subtle category, semantic conflicts, occurs when parallel changes unintentionally affect the system's behavior, resulting in test failures or runtime failures, even if the code is merged and compiled successfully. These conflicts, also called behavioral semantic conflicts (Da Silva et al. (2024)), differ from build conflicts (Da Silva et al. (2022); Brun et al. (2011); Sarma et al. (2012)), which include compilation errors, or syntactic and static semantic conflicts. By their nature, behavioral semantic conflicts can remain hidden for several versions, making their detection and resolution a challenge. To detect semantic conflicts, complementary approaches have been proposed, such as SMAT (Silva et al. (2020)), which uses automated unit test generation to identify behavioral changes introduced during the merge. This approach applies heuristics on test results to signal possible conflicts: for example, if a test fails in the common version (we will refer to this version in the rest of the work as "Base"), passes in one of the modified versions ("Left" or "Right"), but fails again after the merge ("Merge"), this indicates a possible semantic conflict. However, despite its effectiveness in detecting semantic conflicts, this test-based approach faces significant limitations, especially a high rate of false negatives, as observed in evaluations of the tool SAM (Da Silva et al. (2024)). These limitations are linked to the capacity of the test generation tools used, such as Randoop (Pacheco and Ernst (2007)) and EvoSuite (Fraser and Arcuri (2011)), which often do not cover all relevant scenarios.

Given recent advances in large language models (LLMs), the hypothesis arises that these limitations can be overcome. Models such as Code Llama 70B (Rozière et al. (2024)) have shown great potential in code generation and automated tests, being competitive with traditional tools in test coverage and utility. Recent studies (Schäfer et al. (2024); Pan et al. (2025); Zhang et al. (2026)) have demonstrated that LLMs can generate tests with high coverage and originality. In particular, ASTER (Pan et al. (2025)) has shown to be competitive with state-of-the-art tools, including EvoSuite, in terms of cov-

erage, while also producing more "natural" tests preferred by developers. Such advances suggest that LLMs can offer new strategies for generating tests that are more effective in detecting subtle faults. Our previous study (Barbosa et al. (2025)) proposes a tool based on the Code Llama 70B model to generate unit tests for Java programs, integrated into the SMAT architecture. The initial results indicate that LLMs can generate tests capable of detecting semantic conflicts, identifying additional conflicts not detected by SAM using traditional test generation tools, but also highlight challenges related to the complexity of the code and the computational cost of using LLMs for this purpose.

To further explore the potential of LLMs in this context, this work extends our previous study by integrating multiple LLMs into SMAT via our proposed tool, LUCIA (LLM-based Unit-tests for Conflict Identification & Analysis), evaluating their effectiveness in generating tests for semantic conflict detection. This way, we explore different models available through Ollama, such as Llama 4, Gemma 3, DeepSeek R1 and Qwen 3, each with distinct architectures and training data. To evaluate the potential of this tool, we conducted an empirical study with a dataset of merge scenarios in Java, some of them containing semantic conflicts. The study evaluates the impact on test generation and semantic conflict detection of different LLM models, prompt strategies (zero-shot and 1-shot), variations in temperature, and combinations of context provided to the model.

Overall, the LLM-based tool detected three additional conflicts compared to the original approach (Da Silva et al. (2024); Silva et al. (2020)) when evaluated on the same dataset. One of these conflicts had already been detected by Code Llama 70B in our prior work, while the remaining two represent new detections. Furthermore, four model configurations (Llama 4 16x17B and Qwen 3 32B, both with 1-shot prompts at temperatures 0.0 and 0.7) detected the same number of conflicts (six) as Differential EvoSuite, the best traditional tool used by SMAT, thereby surpassing the performance of Code Llama 70B. Regarding the different prompt strategies, the results show that the 1-shot configuration was the most effective in generating tests that detect semantic conflicts for almost all evaluated models, while temperature variations had a limited impact on conflict detection. This finding goes in a different direction of our previous results, in which the zero-shot configuration with a temperature of 0.0 detected the most conflicts when using Code Llama 70B.

When evaluating compilation rates of generated tests, Gemma 3 12B stood out with the highest compilation rate (31.68% and 29.06% for 1-shot with temperatures 0.0 and 0.7, respectively). Just like in the previous work, 1-shot configurations generally achieved higher compilation rates than zero-shot ones, indicating that the presence of an example in the prompt can assist in generating more syntactically correct tests. The fastest models are not necessarily the most effective at detecting conflicts. Yet, combining two configurations allowed us to detect more conflicts (seven) than Differential EvoSuite. While this combined approach is faster than the best single-model configurations, it remains slower than traditional tools. Thus, the main contributions of this work are:

- Evidence that LLMs can generate tests capable of detecting semantic conflicts in merge scenarios, enhancing the detection capabilities within the SMAT approach.
- Analysis of the effect of each model, different prompts, configuration variations, and contexts provided to a set of LLMs for test case generation aimed at detecting semantic conflicts.
- Integration of LUCIA, a modular LLM-based tool into the SMAT architecture, enabling the use of multiple language models to generate tests.
- Proposal of a test generation tool with language-agnostic components (LLM-based test generation and prompt engineering) that can be adapted for other programming languages beyond Java, though requiring language-specific execution infrastructure (e.g., test runners, compilation systems).
- Empirical evidence that code complexity directly impacts the compilation rate of LLM-generated tests.

The remainder of this paper is organized as follows: Section 2 presents the motivation of the work and reviews the operation of SMAT. Section 3 details the adopted methodology, including the integration of LUCIA into SMAT and the experimental procedures. In Section 4, we evaluate the models' performance in comparison with other tools and discuss the results. Section 5 discusses the threats to validity of the study, addressing possible limitations and biases. Section 6 addresses related work, and finally, Section 7 presents conclusions and future directions.

2 Motivating example and background

Semantic conflicts often go unnoticed by traditional version control tools, which focus only on textual differences and ignore behavioral interactions between integrated changes. To exemplify what semantic conflicts are, consider the following example, based on the work of Silva et al. (2020): two developers, Nivan and Renato, work simultaneously on a project, each making changes to the same method of a class, as shown in Listing 1.

```
1 cleanText() {
2     normalizeWhiteSpace();
3     removeComments();
4     removeDuplicateWords();
5 }
```

Listing 1: Method changed by developers

Aiming to make the text more readable, both make contributions to the `cleanText()` method of the `Text` class. Nivan implements and adds a call to the `normalizeWhiteSpace()` method, which replaces multiple white spaces with a single space. Let us call the version of the code with this change Left. Renato, in turn, implements and adds a call to the `removeDuplicateWords()` method, which removes consecutive duplicate words. Let us call this other version of the code with such change Right. The integrated version of the code, which combines the changes of both, is called Merge, and this is the

version illustrated in Listing 1. Both test their functions individually and they work as expected; however, when the code is integrated, a semantic conflict arises: the execution order of the methods `normalizeWhiteSpace()` and `removeDuplicateWords()` affects the final result of the processed text. Observe, for example, the behavior of the function `cleanText()` when applied to an input like `HELLO␣␣HELLO␣␣WORLD`. If the Left version is executed, that is, if only `normalizeWhiteSpace()` is applied, the result will be `HELLO␣HELLO␣WORLD`, keeping two occurrences of `HELLO` and one space between the words. On the other hand, if the Right version is executed, that is, if only `removeDuplicateWords()` is applied, the result will be `HELLO␣␣␣␣WORLD`, with only one occurrence of `HELLO` and four spaces between the words. When both functions are applied in sequence (Merge version), the combined effect reveals an unexpected behavior: `HELLO␣␣WORLD`, that is, one of the occurrences of `HELLO` is removed, but the spaces between the words remain, and they only constitute a duplication after the call of the last method. This is a classic example of a semantic conflict, where the order of application of functions affects the final result and the emergent behavior is not trivially predictable.

The interaction between these changes only reveals its side effect when a test is executed with a specific input (containing both duplicate words and spaces), and such input may not be present in the original test suite, since developers tend to create test cases focused on the most common usage scenarios and that validate functionalities in isolation. Thus, this type of problem is difficult to identify with manual inspection, reinforcing the need for automated approaches based on tests to detect such conflicts.

2.1 SMAT

In this subsection, we present SMAT (Silva et al. (2020); Da Silva et al. (2024)), a tool developed to detect semantic conflicts in code integration scenarios. We first discuss its operation, highlighting how automated test generation and execution identify unexpected interactions between parallel changes. Next, we address the limitations observed in current versions of SMAT, particularly regarding test coverage, and introduce our proposal to enhance the tool through the integration of LUCIA. This integration provides access to multiple local LLMs to explore their potential in increasing conflict detection effectiveness.

To address the complexity of manual identification, SMAT automates the detection of these conflicts by generating unit tests for specific merge scenarios. The tool receives as input a set of merge scenarios, where each includes the project metadata and a quadruple of commits representing the Base (common to all), Left (modification by one developer), Right (modification by another developer), and Merge (result of the integration between Left and Right) versions. Additionally, it processes the paths of code files, target classes, and the elements simultaneously modified by Left and Right. Consistent with the technical requirements of the analyzed projects, the entire SMAT environment operates using Java 8 and JUnit 4.12 by default.

The SMAT architecture is composed of the Test Generation, Test Execution, Test Dynamic Analysis, and Output Generation modules. The operation of these modules is summarized in three main steps:

- **Test generation**: The goal of this step is to generate the test suites. For this, the tools EvoSuite, Differential EvoSuite, Randoop, and Randoop Clean are employed to automatically create tests for the Left and Right versions of the code, each containing modifications in the same class and element. It is worth noting that the SMAT architecture is designed to be extensible, facilitating the integration of new test generation tools whenever necessary.
- **Compilation and execution**: This step aims to verify the behavior of the code versions against the generated tests, seeking to identify execution differences. Thus, the created test suites are compiled and executed in all four code versions (Base, Left, Right, and Merge), with the process repeated multiple times to ensure greater reliability of the results. Results considered invalid are discarded to avoid erroneous analyses.
- **Reports and heuristics**: Finally, this step aims to interpret the test results to detect possible semantic conflicts between the versions. To do this, specific heuristics are applied to the obtained results, identifying patterns that characterize conflicts. For example, a conflict is considered when a test fails in Base and Merge, but passes in Left or Right (or vice versa); or even, when only Merge presents failure or success.

The initial experiments of Silva et al. (2020) show that the approach based on unit tests can be effective: the SMAT heuristic correctly identified 4 out of 15 conflicts, without false positives, but still with a significant number of false negatives. In later works (Da Silva et al. (2024)), the analysis was expanded to 85 scenarios and four test generation tools, resulting in the detection of 9 out of 29 conflicts, reinforcing previous findings, but with three false positives and a considerable false negative rate.

Despite the advances, limitations still exist, especially related to the coverage of automated tests, which may not identify certain subtle interactions between the Left and Right versions of the code, leading to the mentioned false negatives. Traditional tools often do not capture these nuances, making the integration of LLMs an opportunity to explore new test generation strategies and discover which prompts and approaches are most effective to enhance semantic conflict detection.

Thus, SMAT can benefit directly from recent advances in LLMs. At first, Code Llama 70B model was selected for integration into SMAT due to its strong performance in code-related tasks. The choice of Code Llama 70B was, thus, based on three main criteria: its specialization in code, being trained for programming tasks and understanding of languages like Java; its size of 70 billion parameters, which offers a balance between complex reasoning capacity and computational feasibility for local execution; and its availability as an open-source model, allowing reproducibility of experiments and total control over the execution environment, essential aspects for academic research. The incorporation of the tool into SMAT allowed evaluating, comparing, and improving

methods for test generation and semantic conflict detection. In this work, we extend this integration by incorporating not only one model, but multiple LLMs, allowing us to evaluate and compare the performance of different models in generating tests for semantic conflict detection.

2.2 Selected Large Language Models

In this subsection, we present the selected LLMs for integration into SMAT to enhance its test generation capabilities. The selected models are as follows:

- **DeepSeek R1 8B**: A distilled reasoning model based on the Qwen3 8B architecture (specifically the DeepSeek-R1-0528 checkpoint). It is designed to provide competitive reasoning capabilities in a lightweight parameter class, leveraging knowledge distilled from the larger 671B model (Guo et al. (2025)).
- **DeepSeek R1 32B**: A distilled reasoning model built upon the Qwen 2.5 32B base. It balances computational efficiency with high-performance inference, outperforming smaller optimized models in mathematical and coding benchmarks (Guo et al. (2025)).
- **DeepSeek R1 70B**: A large-scale distilled reasoning model based on the Llama 3.3 70B Instruct architecture. It offers frontier-level reasoning capabilities comparable to proprietary models, suitable for complex tasks and logic problems (Guo et al. (2025)).
- **Gemma 3 4B**: A lightweight multimodal model optimized for consumer hardware that achieves performance competitive with the previous generation's 27B model across math, coding, and multilingual benchmarks (Team et al. (2025)).
- **Gemma 3 12B**: A mid-sized multimodal model that balances efficiency and capacity, featuring the same vision understanding and 128K context window as the larger models to handle complex reasoning tasks (Team et al. (2025)).
- **Gemma 3 27B**: The series' most capable model, offering performance comparable to Gemini 1.5 Pro and outperforming significantly larger open models in blind evaluations for high-demand reasoning and coding applications (Team et al. (2025)).
- **Llama 4 16x17B**: Also known as Llama 4 Scout, this multimodal model has 17 billion active parameters with 16 experts. It is more powerful than all previous Llama models and is competitive with Gemma 3, Gemini 2.0 Flash-Lite, and Mistral 3.1 across a broad range of widely reported benchmarks (Meta AI (2025)).
- **Qwen3 30B**: A Mixture-of-Experts (MoE) model with 30 billion total parameters but only 3 billion activated per token, designed to optimize inference efficiency while maintaining high performance. It leverages strong-to-weak distillation to achieve reasoning capabilities comparable to dense 32B models with significantly lower computational costs (Yang et al. (2025)).
- **Qwen3 32B**: The flagship dense model of the series, establishing state-of-the-art reasoning performance for its size by outperforming previous models like QwQ-32B and competing with OpenAI-o3-mini. It delivers performance comparable to the previous generation's 72B model across general, math, and coding benchmarks (Yang et al. (2025)).

These models were selected based on three key criteria essential for our study: availability, architectural diversity, and proven capabilities. First, regarding availability and efficiency, all selected models are deployable via the Ollama framework (Ollama (2023)), ensuring the reproducibility of our local execution environment. Second, to ensure architectural diversity, we included varied architectures, ranging from distilled reasoning models (DeepSeek R1) and Mixture-of-Experts (Qwen3 30B) to standard dense models (Qwen3 32B) and multimodal experts (Llama 4, Gemma 3), to investigate how different structural approaches impact test generation for semantic conflict detection. Finally, the selection considered models with established coding and reasoning capabilities, as identifying semantic conflicts likely benefits from reasoning about program states and logic beyond simple syntactic completion. By evaluating these high-performing models, we aim to determine if their advanced reasoning capabilities translate into improved effectiveness in generating tests that can uncover semantic conflicts during code integration.

3 Methodology

To investigate whether LLMs can be useful for unit test generation aimed at detecting semantic conflicts, we conducted a series of experiments using two distinct samples (datasets), each with specific motivations and characteristics. The first dataset contains merge scenarios from significant projects, some of them containing semantic conflicts, allowing the evaluation of the proposed tool's capability to detect this type of problem. The second dataset, on the other hand, does not contain merge scenarios nor semantic conflicts, and is composed of final versions of simpler projects, being used to evaluate the tool's test generation and compilation capability. To enable these experiments, we developed and integrated into SMAT a new automated test generation module, whose operation is based on the Ollama framework. Next, the process of integrating the module into SMAT is presented, as well as the datasets selected for the study and additional information on the procedures adopted during the experiments.

3.1 Integration of LUCIA into SMAT

As mentioned in Section 2, the SMAT architecture is modular, allowing the addition of new tools and functionalities as needed. In this context, we developed a complete automated test generation tool, which uses different LLMs accessible through the Ollama framework. We also designed a pipeline that incorporates several additional steps of processing, integration, and validation. The new tool expands the existing Test Generation, adding a pipeline that performs everything from the extraction and preparation of source code information, through the construction and sending of prompts to the model, to the handling, validation, and integration of the generated tests into the SMAT flow.

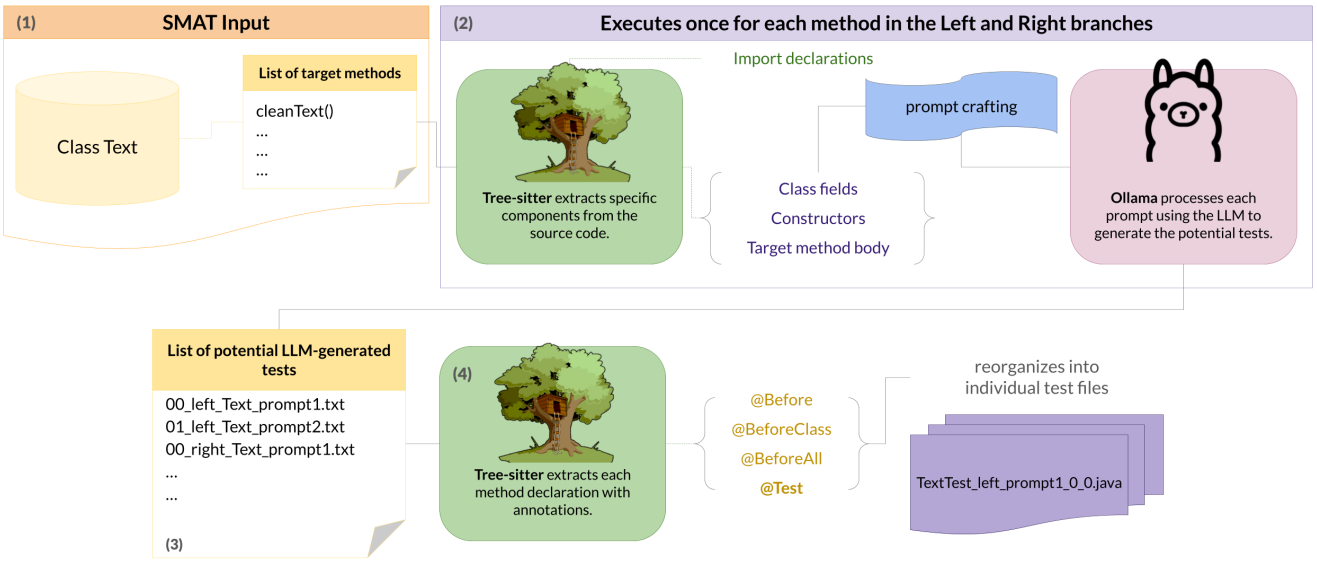

Figure 1. Architecture and output processing of the LUCIA tool integrated into SMAT.

Integration Overview. The integration of LUCIA into SMAT is implemented through a pipeline that uses information from the code under analysis to generate executable unit tests. The flow starts with the extraction of relevant elements from the source code, which are then formatted into structured prompts to be sent to the LLM. The model's response containing the test code goes through a cleaning and validation process before being compiled and executed by the other SMAT components. This process is organized in sequential steps, as detailed below:

1. In addition to the elements cited in Section 2.1, the SMAT input was modified to include, along with each element simultaneously altered by Left and Right in the target class, a textual summary of the changes made by each branch. In the case of the mergedataset, these summaries were already available and were manually created by humans, not by LLMs. Thus, to apply this approach to other datasets, it will be necessary to manually produce the summaries of the changes. This change was made to evaluate whether including this information assists the LLM in generating tests capable of detecting conflicts. As illustrated in Figure 1, step (1) shows the input information that is initially used for the generation of prompts: the target classes (`Class Text`) and their respective target methods, which are the elements altered by Left and Right that will be tested (`cleanText()`).
2. As highlighted in step (2), for each of the target methods informed in the input, we use the parsing library Tree-sitter (Tree-sitter (2024)) to build the AST (*Abstract Syntax Tree*) of the source code, which allows structuring the code to collect information for the generation of prompts. From this AST, we extract class fields, constructors, and the method body, in addition to `import` statements, which are appended to the tests at the end of the iteration to ensure their compilation. Due to the nature of Tree-sitter, with few modifications, this library can be used to generate ASTs for any programming language, which makes LUCIA adaptable to contexts other than Java.

Prompt Construction. The extracted information is organized into different *prompt* formats, composed of messages of types `system`, `user`, and `assistant`, each with a specific function in the interaction with the language model. According to White et al. (2023), *prompts* can be structured using *prompt patterns*, which act as reusable standards for solving recurring problems in the use of LLMs. The `system` message corresponds to the *Persona* pattern, assigning a specific identity or role to the model, with the goal of guiding its behavior throughout the conversation. This message establishes the general context of the interaction and defines guidelines for the style and content of the generated responses. Below is the `system` message used in this study:

SYSTEM

> You are a senior Java developer with expertise in JUnit testing.
> Your task is to provide JUnit tests for the given method in the class under test, considering the changes introduced in the left and right branches.
> You have to answer with the test code only, inside code blocks (```).
> The tests should start with `@Test`.

Note that while the `system` message defines the persona's expertise, it does not explicitly restrict the generation to specific versions of the Java language or the JUnit framework. This intentional choice was made to evaluate the models' baseline behavior; however, as discussed in Section 5, this lack of version constraint may lead models to generate code using modern standards. Since the SMAT environment is based on Java 8 and JUnit 4.12, such discrepancies are a known, albeit infrequent, factor among the reported compilation failures.

The `user` messages provide the main input to the model and can be related to the *Recipe* and *Context Manager* patterns. These messages describe code information, the target

method to be tested, and instructions for the model, delimiting the task scope and the desired requirements. Next, an example of a `user` message that our module uses to generate tests for the `cleanText` method of the `Text` class:

USER

Here is the context of the method under test in the class `Text` on the `left` branch:

Class fields:

```
public String text;
```

Constructors:

```
public Text(String text) {
    this.text = text;
}
```

Target Method Under Test:

```
public void cleanText() {
    Text inst = new Text(text);
    inst.normalizeWhiteSpace();
    inst.removeComments();
    this.text = inst.text;
}
```

Now generate JUnit tests for the method under test, considering the given context.
Remember to create meaningful assertions.
Write all tests inside code blocks (```), and start each test with `@Test`.

In this message, which composes a zero-shot type prompt, the context of the target method is provided, including class fields, constructors, and the method body. Then, the model is instructed to generate JUnit tests for this method, considering the provided information. The `assistant` messages are used here only in the 1-shot configuration. This use aligns with the *Template* pattern, whose goal is to provide an example that serves as a model for generating consistent outputs suitable for the proposed task. The prompts used were built from sets of messages containing different parts of the scenario context, as previously introduced:

(i) Summary of changes introduced by Left and Right: a textual summary of the alterations made in each branch. For example, in the case of the `cleanText()` method, the summary presented to the model was:

> Left wanted to extend the method `cleanText()` to also remove duplicated whitespace in the text by adding the method call `normalizeWhitespace()`.
>
> Right wanted to clean the text by removing consecutive duplicated words.

(ii) Class fields;
(iii) Constructors; and
(iv) Body of the target method.

For each prompt format (zero-shot and 1-shot), as detailed in Table 6, 8 prompt variations were created. Each variation combines different subsets of these context parts to explore which information is most relevant for generating effective tests, introducing greater variety in interactions with the model and, consequently, increasing the diversity of the generated tests. Below is the general structure of the prompts used:

- **Zero-shot:** The prompt is composed of a system message (`system`), followed by user messages (`user`) that provide the context of the target method to be tested.
- **1-shot:** In this format, the prompt first presents a complete example of interaction before the actual request. The structure contains the system message (`system`), followed by a pair of example messages (one from the `user` with a fake method and a corresponding response from the `assistant` with the generated test). Only after this example is the `user` message with the actual target method presented.

All prompts used for each project tested in the experiments are available in the online appendix of this work Barbosa et al. (2026).

Model Invocation. These prompts are sent, via the REST API of Ollama (an open-source platform for local execution of LLMs), to the different models available through this framework. Although the models can return multiple responses for each prompt to create diverse test cases, we configured the tool to generate a single response per target method in each branch. This decision aligns with our use of fixed seeds (detailed in Section 3.3), as consistent initialization typically results in limited variation across subsequent outputs. Consequently, multiple generations would offer fewer benefits in this setup compared to scenarios involving random seeds, where additional outputs could significantly enhance diversity.

Output Processing. Figure 1 also illustrates the output processing of LUCIA, which is similar regardless of the LLM used. As shown in step (3), the LLM responses are considered here as potential test suites. Each of them goes through a cleaning step using regular expressions, with the goal of removing segments that do not correspond to valid Java code. This includes natural language, irrelevant comments, formatting artifacts, and characters that do not belong to the language lexicon, which could compromise the compilation and execution of tests.

After this cleaning, we again use Tree-sitter, highlighted in step (4), to parse the resulting output and extract the relevant methods. Specifically, the Tree-sitter query is designed to search not only for the `@Test` annotation but also for System Under Test (SUT) preparation annotations, such as `@Before`, `@BeforeClass`, and `@BeforeAll`. The final extracted test code includes the setup methods alongside the test method, if they are generated by the model; otherwise, solely the test method is extracted. Identifying the `@Test` annotation is an essential first step to ensure the generated code is recognized by JUnit as an executable test. Furthermore, although the exact frequency and impact were not quantitatively measured

Table 1. Scenarios discarded from the mergedataset due to the absence of files in the *master* branch of the repository.

Project	Class	Method
libgdx	Lwjgl3ApplicationConfiguration	copy
libgdx	Lwjgl3Application	newWindow
elasticsearch	RestNodesAction	getTableWithHeader
elasticsearch	RestIndicesAction	getTableWithHeader
elasticsearch	RestShardsAction	getTableWithHeader
ReactiveX	TestScheduler	triggerActions

in this study, the absence of generated SUT preparation methods or their annotations can significantly affect the test suites, potentially leading to compilation errors or runtime failures (e.g., `NullPointerException`) due to uninitialized SUT components.

Then, each test is stored in a unique file, identified by a name that concatenates the target class, the branch, the prompt, the method index, and a sequential number, such as in `TextTest_left_prompt1_0_0.java`. The goal of this is to maximize the utilization of the generated tests: if one or more tests present compilation problems, the others can still be executed without prejudice to the evaluation process. This approach also facilitates the organization of experiments and the subsequent analysis of results, allowing the isolation of failures and the identification of patterns in the generated errors.

3.2 Datasets

A priori, we used the mergedataset as a basis, as described by Da Silva et al. (2024), which originally contains 85 scenarios. However, our final dataset was reduced to 79 scenarios, as 6 of them were discarded due to the absence of files in the *master* branch of the repository, as detailed in Table 1. This subset of the mergedataset is composed of 29 open-source Java projects, covering diverse software areas, and includes 29 scenarios with semantic conflicts and 50 without conflicts.

To complement the analysis, we also selected a second dataset, composed of two systems considered "toy" due to their lower complexity. These projects, simpler and more didactic, were extracted from the dataset used in the work of Pan et al. (2025). Since it is a work focused only on test generation, these projects do not contain merge scenarios and, therefore, do not present semantic conflicts. We will refer to these projects as the *ASTER dataset*:

- **Eclipse Cargo Tracker**: an open-source cargo tracking application, built with Jakarta EE and structured based on Domain-Driven Design principles. The application simulates common flows of enterprise systems, such as logistics monitoring, routing, and event handling (Eclipse Foundation (2020)).
- **DayTrader 8**: a stock trading system benchmark developed with Java EE 8. The application implements typical features of financial systems, such as user login, account inquiry, and execution of buy and sell orders. Due to its compact nature, it is widely used as a reference for functional and performance testing in Java EE application servers (OpenLiberty (2018)).

These two were selected because they simulate systems with behavior close to that of real applications, albeit with lower structural and code complexity compared to the projects in the mergedataset. The other projects present in the original dataset of Pan et al. (2025) were discarded for two main reasons. Some of them (Commons CLI, Codec, Compress, and JXPath), which are Apache Commons APIs, were considered too simple and possibly used in the model's training, which could bias the results; in turn, PetClinic was not included due to Java version incompatibility with SMAT. This choice allows evaluating the model's capability to generate unit tests both in more controlled and didactic contexts and in scenarios extracted from real systems.

3.3 Experimental Procedures

In this study, we aim to answer the following research questions:

- **RQ1:** Do specific configurations of prompt format, temperature, or model architecture enable the generation of more effective tests for detecting semantic conflicts?
- **RQ2:** To what extent is the proposed tool effective for the automatic detection of semantic conflicts in merge scenarios?
- **RQ3:** Do any of the evaluated LLMs outperform the others in test generation and semantic conflict detection?
- **RQ4:** How does code complexity impact the compilation rate of tests generated by LLMs?

Parameter Configuration. The choice of parameters used in the experiments was fundamental to evaluating the behavior of the models in different test generation scenarios. We varied the temperature and alternated the prompt format to assess their impact on the generated tests. The random seed was fixed for all models to ensure reproducibility, except for Code Llama 70B, for which two distinct seed values were used to evaluate the sensitivity of the results to initialization.

The **temperature** is a parameter that controls the degree of randomness of the model's responses. In this study, we evaluated two settings: 0.0 and 0.7. Lower values (such as 0.0) tend to produce more deterministic and conservative outputs, serving as a baseline for comparison. Conversely, higher values (such as 0.7) stimulate greater diversity and creativity, potentially generating more varied tests (Holtzman et al. (2020)). We specifically selected 0.7 as our higher temperature setting because it is the default setting for many models, allowing us to simulate a standard usage scenario where this parameter is not heavily customized.

The **seed** parameter defines the initial state of the model's random number generator. By fixing the seed, we can achieve similar outputs across different executions, which is essential for reproducibility. This control is critical, as previous studies show that distinct seeds can lead to substantially different outcomes even with identical hyperparameters (Dodge et al. (2020)). By using different values, it is possible to evaluate the stability of results and identify possible variations arising from the non-deterministic nature of LLMs. The seed values (42 and 123) were selected as arbi-

trary constants to introduce variance, rather than being determined through empirical search or optimization.

Finally, the **prompt format** directly influences how the model interprets the task. Two approaches were evaluated: *zero-shot*, where the model receives only the instructions and the context of the target method, and *1-shot*, where a test example is provided before the actual request.

Experiment 1: Semantic Conflict Detection. Using the mergedataset (79 scenarios), we evaluated the effectiveness in detecting semantic conflicts (RQ1, RQ2 and RQ3). We executed 36 distinct configurations, combining both temperature values (0.0 and 0.7) and two prompt formats (zero-shot and 1-shot) across 9 different models. For Code Llama 70B, we used two different seed values (42 and 123) for each temperature and prompt format combination, resulting in a total of 8 configurations for this model. Each configuration applies 8 distinct prompts for each target method in each branch.

Experiment 2: Generation and Compilation Capability. To evaluate the impact of code complexity (RQ4), we compared test generation and compilation across both datasets using the Code Llama 70B model, with temperature fixed at 0.0 and five distinct prompt variations (four zero-shot and one 1-shot). The *ASTER dataset* does not include change summaries as it does not contain merge scenarios. Our choice to limit this experiment to Code Llama 70B alone is justified by both practical and methodological considerations. First, the high computational cost and time required to execute the entire model ensemble across additional datasets were prohibitive. Second, Code Llama 70B serves as a representative baseline for our analysis. Given that newer models generally exhibit higher compilation rates and superior instruction-following, they are expected to mirror the performance trends observed here, albeit with potentially narrower margins.

Analysis of Results. The analyses were performed based on SMAT reports, including execution time metrics, quantity of tests generated, compiled, and executed, in addition to detected conflicts. Additionally, we manually verified the tests that identified semantic conflicts to validate the results and avoid false positives.

All prompts used are available in the online appendix of this work (Barbosa et al. (2026)). The experiments with the newer models were conducted on a machine running Ubuntu, equipped with 256GB of RAM, an Intel Xeon Silver 4316 processor (80 cores), and a NVIDIA L40S 46GB GPU, while the experiments with Code Llama 70B were performed on a machine with 251GB of RAM, an Intel Xeon Silver 4316 processor (40 cores), and two NVIDIA A100 80GB PCIe GPUs.

It is worth noting that these hardware differences can affect only the inference speed, preserving comparability of the semantic conflict detection results.

Table 2. Test generation results with Code Llama 70B for different configurations. Values are averages of two independent runs (seeds 42 and 123). * indicates conflict not previously identified by SMAT tools.

Metric	Temperature 0		Temperature 0.7	
	1-shot	zero-shot	1-shot	zero-shot
Average generated tests	10404	8592	1383	2358
Average compiled tests	1253	586	124	121
Average generated tests per method	131.70	108.76	17.50	29.85
Average compiled tests per method	15.86	7.42	1.57	1.53
Average compilation rate	12.04%	6.82%	8.97%	5.13%
Detected conflicts	1*	3*	3*	1*

4 Results

To answer the research questions discussed in the previous section, we conducted the experiments described previously and present the results below. The section is structured around the research questions, starting with a quantitative analysis of test generation, followed by semantic conflict detection and comparison with other tools, and ending with an analysis of the dataset impact and execution time.

4.1 Quantitative Analysis of Test Generation

Table 2 presents the results of test generation and conflict detection using the LUCIA tool with the model Code Llama 70B, considering the different temperature and prompt format configurations. To construct the table, two independent runs were performed for each format and temperature, each with 8 prompts. Each run was performed with a different seed, and then the results were aggregated to obtain the presented averages, while the quantity of detected conflicts was considered as the union of conflicts identified in both runs. We can observe that temperature 0.0 resulted in significantly higher test generation, with an average of 10404 tests in the 1-shot configuration and 8592 in the zero-shot configuration. In contrast, temperature 0.7 generated a considerably smaller amount of tests, with averages of 1383 and 2358 tests for the 1-shot and zero-shot configurations, respectively. A hypothesis for this behavior is that lower temperatures make the model more deterministic, favoring pattern repetition and, consequently, increasing the volume of generated tests. On the other hand, higher temperatures stimulate greater diversity in responses but tend to reduce the total amount of produced examples, since the model explores less predictable and, often, shorter paths. This effect highlights how the choice of temperature can directly impact both the volume and variety of tests obtained in each configuration.

The compilation rate was also higher with temperature 0.0, peaking at 12.04% in the 1-shot configuration. As the lower temperature favors pattern repetition, it is possible that the generated tests are more consistent and, therefore, more prone to compiling successfully. Although the zero-shot configuration generated a large volume of tests, its compilation rate was lower (6.82%), indicating that the absence of examples in the prompt may have hindered the generation of syntactically correct tests.

With temperature 0.7, the compilation rates were 8.97% (1-shot) and 5.13% (zero-shot). Notably, all four experimental configurations were able to detect semantic conflicts, with each identifying at least the conflict previously not detected

Table 3. Quantitative performance metrics for all evaluated models, sorted by mean execution time. In bold: the ten best compilation rates.

Model	Mean Execution Time per Output	Detected Conflicts	Generated Tests	Compiled Tests	Compilation Rate
Gemma 3 4B (1-shot, temperature 0.0)	4.34	3	2809	475	**16.91**
Gemma 3 4B (1-shot, temperature 0.7)	6.41	4	2617	412	**15.74**
Gemma 3 12B (1-shot, temperature 0.7)	10.81	4	5348	1554	**29.06**
Gemma 3 12B (1-shot, temperature 0.0)	13.36	4	5997	1900	**31.68**
Gemma 3 4B (zero-shot, temperature 0.7)	13.38	4	4256	397	9.33
Gemma 3 27B (1-shot, temperature 0.7)	19.74	5	3571	606	**16.97**
Gemma 3 12B (zero-shot, temperature 0.7)	21.76	3	4617	494	10.70
Gemma 3 27B (1-shot, temperature 0.0)	21.83	4	3974	797	**20.06**
Gemma 3 27B (zero-shot, temperature 0.7)	32.90	2	2232	250	11.20
Gemma 3 4B (zero-shot, temperature 0.0)	33.16	5	4366	482	11.04
Gemma 3 12B (zero-shot, temperature 0.0)	38.48	3	4996	625	12.51
DeepSeek R1 32B (1-shot, temperature 0.7)	40.34	5	4260	355	8.33
DeepSeek R1 32B (1-shot, temperature 0.0)	40.92	4	4623	450	9.73
Gemma 3 27B (zero-shot, temperature 0.0)	44.11	3	2293	242	10.55
DeepSeek R1 32B (zero-shot, temperature 0.7)	45.42	4	5147	259	5.03
Qwen 3 30B (1-shot, temperature 0.0)	46.60	5	2847	463	**16.26**
Qwen 3 30B (1-shot, temperature 0.7)	48.30	5	2851	395	13.85
Llama 4 16x17b (1-shot, temperature 0.0)	51.19	6	5791	1188	**20.51**
DeepSeek R1 32B (zero-shot, temperature 0.0)	51.66	2	5642	387	6.86
Llama 4 16x17b (1-shot, temperature 0.7)	54.21	6	5289	864	**16.34**
Qwen 3 30B (zero-shot, temperature 0.7)	62.01	4	4351	540	12.41
Qwen 3 30B (zero-shot, temperature 0.0)	72.68	3	4463	598	13.40
Llama 4 16x17b (zero-shot, temperature 0.7)	83.81	5	8054	948	11.77
Llama 4 16x17b (zero-shot, temperature 0.0)	86.57	5	8721	1242	14.24
Qwen 3 32B (1-shot, temperature 0.7)	107.37	6	4008	488	12.18
Qwen 3 32B (1-shot, temperature 0.0)	112.13	6	4155	538	12.95
DeepSeek R1 70B (1-shot, temperature 0.0)	141.82	5	6343	1044	**16.46**
DeepSeek R1 70B (1-shot, temperature 0.7)	150.85	5	5755	841	14.61
DeepSeek R1 70B (zero-shot, temperature 0.0)	151.41	3	6913	816	11.80
Qwen 3 32B (zero-shot, temperature 0.7)	158.40	5	5608	430	7.67
Qwen 3 32B (zero-shot, temperature 0.0)	159.41	4	5589	441	7.89
DeepSeek R1 70B (zero-shot, temperature 0.7)	161.82	3	6205	656	10.57
DeepSeek R1 8B (1-shot, temperature 0.7)	219.30	0	64	1	1.56
DeepSeek R1 8B (zero-shot, temperature 0.7)	270.01	0	26	0	0.00
DeepSeek R1 8B (1-shot, temperature 0.0)	281.70	0	6	0	0.00
DeepSeek R1 8B (zero-shot, temperature 0.0)	299.45	0	0	0	0.00

by other SMAT tools. In particular, the zero-shot configuration with temperature 0.0 identified three conflicts in a single run, while each seed of the 1-shot configuration with temperature 0.7 detected two, one of them being common to both seeds. Therefore, the zero-shot configuration with temperature 0.0 stood out for detecting the highest number of semantic conflicts in a single run.

Comprehensive results for the other evaluated models, including test generation and conflict detection across different temperature and prompt configurations, are detailed in Table 3. To construct the table, one run was performed with a fixed seed value (123) for each format and temperature, each with 8 prompts. We can observe that the top ten configurations (highlighted in bold) in compilation rate are 1-shot configurations, with Gemma 3 12B standing out with the highest compilation rates (31.68% and 29.06% for temperatures 0.0 and 0.7, respectively). Such an observation corroborates the trend observed with Code Llama 70B, where the presence of an example in the prompt seems to assist in generating more syntactically correct tests. Also, all zero-shot configurations have lower compilation rates than their respective 1-shot counterparts, even with some models generating a higher volume of tests in the zero-shot format.

This consistency extends to the decoding parameters. Similar to the baseline results, the temperature effect observed with Code Llama 70B is replicated across the other models: lower temperatures (0.0) generally lead to higher test generation volumes and compilation rates compared to higher temperatures (0.7).

These results indicate that the choice of temperature directly impacts both the quantity and quality of the generated tests. Lower temperatures favor higher volume and superior compilation rates, while higher temperatures tend to reduce both indicators. Furthermore, the *prompt* strategy also influences the syntactic quality of tests, with the provision of an example (1-shot) leading to more compilable and, therefore, syntactically more correct codes.

4.2 RQ1: Do specific configurations of prompt format, temperature, or model architecture enable the generation of more effective tests for detecting semantic conflicts?

Tables 4 and 5 offer a detailed view of the conflict detection capability, highlighting the distinct detection coverage pro-

Table 4. Semantic conflicts detected by Code Llama 70B by configuration. Values in parentheses: (temperature, seed). ✓indicates detected conflict. In bold: conflict not previously identified by SMAT tools.

Conflict (Project::Class::Element)	zero-shot (0, 42)	zero-shot (0, 123)	zero-shot (0.7, 42)	zero-shot (0.7, 123)	1-shot (0, 42)	1-shot (0, 123)	1-shot (0.7, 42)	1-shot (0.7, 123)
`antlr4::Python2Target::python2Keywords`	✓	✓	—	—	—	—	—	—
`antlr4::Python3Target::python3Keywords`	✓	✓	—	—	—	—	—	—
`spring-boot::AtomikosProperties::asProperties()`	—	—	—	—	—	—	✓	—
`spring-boot::AtomikosPropertiesTest::testProperties()`	—	—	—	—	—	—	—	✓
`cloud-slang::SlangImpl::getAllEventTypes()`	✓	✓	✓	✓	✓	✓	✓	✓

Table 5. Semantic conflicts detected across all evaluated models/configurations. DS-R1 = DeepSeek R1, GM3 = Gemma 3, LL4 = Llama 4, QW3 = Qwen 3. ✓indicates detected conflict. In bold: conflict not previously identified by SMAT tools.

Conflict (Project::Class::Element)	DS-R1-32B zero-shot T=0.0	DS-R1-32B zero-shot T=0.7	DS-R1-32B 1-shot T=0.0	DS-R1-32B 1-shot T=0.7	DS-R1-70B zero-shot T=0.0	DS-R1-70B zero-shot T=0.7	DS-R1-70B 1-shot T=0.0	DS-R1-70B 1-shot T=0.7	GM3-12B zero-shot T=0.0	GM3-12B zero-shot T=0.7	GM3-12B 1-shot T=0.0	GM3-12B 1-shot T=0.7	GM3-27B zero-shot T=0.0	GM3-27B zero-shot T=0.7	GM3-27B 1-shot T=0.0	GM3-27B 1-shot T=0.7	GM3-4B zero-shot T=0.0	GM3-4B zero-shot T=0.7	GM3-4B 1-shot T=0.0	GM3-4B 1-shot T=0.7	LL4-16X17B zero-shot T=0.0	LL4-16X17B zero-shot T=0.7	LL4-16X17B 1-shot T=0.0	LL4-16X17B 1-shot T=0.7	QW3-30B zero-shot T=0.0	QW3-30B zero-shot T=0.7	QW3-30B 1-shot T=0.0	QW3-30B 1-shot T=0.7	QW3-32B zero-shot T=0.0	QW3-32B zero-shot T=0.7	QW3-32B 1-shot T=0.0	QW3-32B 1-shot T=0.7
`antlr4::Python2Target::python2Keywords`	✓	✓	✓	✓	—	—	✓	✓	—	✓	✓	✓	✓	✓	✓	✓	—	—	—	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`antlr4::Python3Target::python3Keywords`	—	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	—	—	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`cloud-slang::SlangImpl::getAllEventTypes()`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	—	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`elasticsearch::IndexScopedSettings::BUILT_IN_INDEX_SETTINGS`	—	—	—	—	✓	✓	✓	✓	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	✓	✓	✓	✓	✓	✓	✓
`HikariCP::HikariPool::addConnection()`	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	✓	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
`spring-boot::AtomikosProperties::asProperties()`	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	✓	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
`spring-boot::AtomikosPropertiesTests::testDefaultProperties()`	—	✓	✓	—	—	—	—	—	—	✓	—	—	—	—	—	—	—	—	✓	✓	✓	✓	✓	✓	—	—	—	✓	—	—	✓	✓
`spring-boot::AtomikosPropertiesTests::testProperties()`	—	—	—	✓	—	—	—	—	—	—	—	—	—	—	✓	✓	✓	✓	—	—	—	—	✓	✓	—	—	—	—	—	—	✓	✓	✓
`spring-boot::ExplodedArchive::getUrl()`	—	—	—	✓	—	—	—	✓	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
`Storm::KafkaSpoutConfig::toString()`	—	—	—	—	—	—	✓	—	✓	—	—	✓	—	—	—	—	✓	✓	✓	✓	✓	✓	✓	✓	—	—	—	✓	—	—	—	—	—

vided by the experimental configurations. No configuration in isolation was able to detect all conflicts.

The `getAllEventTypes()` conflict (illustrated in Listing 2, with full versions detailed in Listings 5, 6, 7, and 8 of Appendix B) was the only one identified in all eight experimental runs with Code Llama 70B, and in almost all runs for the other models, while remaining undetected by all other tools integrated into SMAT. In this merge scenario, both the Left and Right branches add distinct elements to the same list, resulting in different collection sizes: 23 elements in the Left branch and 21 in the Right. Meanwhile, the Base and Merge versions contain 22 elements each, though their specific contents differ.

A possible explanation for why tools like EvoSuite missed this conflict lies in the experimental constraints adopted in previous works, such as the choice of seeds, the execution timeout (300 seconds per scenario), and the fitness function employed to guide test generation. These factors may restrict search space exploration or hinder the generation of specific assertions for this type of divergence. Merely increasing EvoSuite's budget may not improve results under the same fitness function. However, refining the fitness function in conjunction with a larger budget could potentially enhance detection capabilities.

The proposed tool, on the other hand, managed to identify this inconsistency by generating tests that explicitly verify the size of the list returned by the method.

Listing 3 presents a representative example of the generated tests that managed to detect this conflict. The test assumes the expectation of the Left branch (23 elements) and, for this reason, fails when executed in the context of the Right branch, thus revealing the semantic conflict.

```
1 @Test
2 public void test00() {
3 SlangImpl slang = new SlangImpl();
4 Set<String> eventTypes = slang.
      getAllEventTypes();
5 assertEquals(23, eventTypes.size());
6 }
```

Listing 3: Example of test generated by the Code Llama 70B model to detect the `getAllEventTypes()` conflict.

For Code Llama, the two conflicts related to the `antlr4` project were observed only in the zero-shot runs with temperature 0.0. The `asProperties()` conflict occurred exclusively in the 1-shot configuration with temperature 0.7 and seed 42, while `testProperties()` was detected only with temperature 0.7 and seed 123.

We observed a clear specialization. The runs in **zero-shot with temperature 0.0** were the only ones capable of identifying the two conflicts in the `antlr4` project. On the other hand, the conflicts in the `spring-boot` project were de-

```
private Set<String> getAllEventTypes() {
    Set<String> eventTypes = new HashSet<>();
    // ...
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_ERROR);
    + eventTypes.add(ScoreLangConstants.EVENT_TASK_START); // LEFT
    eventTypes.add(ScoreLangConstants.EVENT_INPUT_START);
    // ...
    eventTypes.add(ScoreLangConstants.EVENT_BRANCH_END);
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_EXPRESSION_START); // RIGHT
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_EXPRESSION_END); // RIGHT
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_OUTPUT_START); // RIGHT
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_OUTPUT_END); // RIGHT
    + eventTypes.add(ScoreLangConstants.EVENT_SPLIT_BRANCHES); // RIGHT
    + eventTypes.add(ScoreLangConstants.EVENT_JOIN_BRANCHES_START); // RIGHT
    + eventTypes.add(ScoreLangConstants.EVENT_JOIN_BRANCHES_END); // RIGHT
    eventTypes.add(ScoreLangConstants.EVENT_EXECUTION_FINISHED);
    return eventTypes;
}
```

Listing 2. Merged version of the `getAllEventTypes()` method from the `cloud-slang::SlangImpl` class (abbreviated). Added lines are highlighted in green, while deleted lines are marked in red.

tected exclusively by the runs in **1-shot with temperature 0.7**. This result suggests that the variation in temperature and prompt strategy allows exploring different types of faults, indicating that an approach combining multiple configurations is more effective to maximize conflict detection.

However, it is important to highlight that this strategy is computationally more expensive. For example, test generation times for each configuration varied from about 6 hours (1-shot, temperature 0.7) to almost 28 hours (1-shot, temperature 0) for the entire mergedataset. Thus, using more than one configuration to detect conflicts may be unfeasible in scenarios where time is a critical factor. More on execution times in Section 4.6.

As shown in Table 5, the diversity in detection coverage across configurations is also evident in the results of other evaluated models. In total, 10 semantic conflicts were detected across all models and configurations, with no single configuration identifying all conflicts. Considering configurations in isolation, the best performance was limited to 6 detected conflicts, achieved by Llama 4 and Qwen 3 32B.

The `getAllEventTypes()` conflict[1], previously discussed, was also detected here by all models and almost all configurations. Other two conflicts not previously identified by SMAT tools were detected by multiple models: `getUrl()` in `spring-boot`[2] and `BUILT_IN_INDEX_SETTINGS` in `elasticsearch`[3]. The `getUrl()` conflict was identified by DeepSeek R1 32B and 70B in the 1-shot configuration with temperature 0.7, while the `BUILT_IN_INDEX_SETTINGS` conflict was detected by DeepSeek 70B, Qwen 3 32B and Qwen 3 30B in almost all configurations.

The detection of the `getUrl()` conflict (detailed in Listings 9, 10, 11, and 12 of Appendix B) in `spring-boot` warrants specific attention as it represents a distinct category of merge issues known as *post-integration changes*. In this merge scenario, both the Left and Right branches modified the parameters of the URL object returned by the `getUrl()` method in the `ExplodedArchive` class. Since both changes occurred on the same lines, a textual conflict arose. This conflict was resolved by the integrator, who adopted neither of the two changes. Instead, Merge version used the `URI.toURL()` method, diverging from both implementations by not using the constructor of the `URL` class directly. This case involves a textual conflict resolved by an integrator, where the resolution introduced a semantic conflict. Da Silva et al. (2022) defines *post-integration changes* as modifications intended to resolve textual merge conflicts that inadvertently introduce build issues. In our case, there is not a build issue per se, but rather a semantic conflict that emerged from the resolution of a textual conflict. Although not originally labeled as a semantic conflict in the dataset, the detection is positive as it adheres to the formal definition of interference: the generated tests pass in Base, Left, and Right versions, but fail in the Merge version.

The generated tests that detected this conflict fail on the assertion involving the URL's authority field. While the Base, Left, and Right versions instantiate a `URL` object using a constructor that infers an empty string for the authority field, the Merge resolution employs the `URI.toURL()` conversion method. This method parses the URI string representation, and due to the specific syntax of the string (lacking the double slash), results in the authority field being set to `null` rather than empty string, causing the divergence detected by the models. Listing 4 presents one test case that successfully detected this divergence, by asserting that the URL's authority is an empty string. It is noteworthy that this subtlety regarding the URL authority field was not captured by the test suites generated by previous SMAT tools, underscoring the improved fault detection capability of the evaluated LLMs in complex integration scenarios.

```
@Test
public void test00() throws
```

[1] https://github.com/spgroup/mergedataset/tree/master/cloud-slang/20bac30d9bd76569aa6a4fa1e8261c1a9b5e6f76

[2] https://github.com/spgroup/mergedataset/tree/master/spring-boot/af20dc6cc45c032573413c401f9f73aa75371744

[3] https://github.com/spgroup/mergedataset/tree/master/elasticsearch/d896886973660785aac45275ddb110c1a6babc57

Table 6. Comparison of semantic conflicts detected by prompt context: standalone Code Llama 70B versus the aggregate of other evaluated models. * indicates a conflict not previously identified by SMAT tools.

Prompt	Context	Code Llama (Standalone)	Other Models (Aggregate)
`prompt1`	Target method body	1*	7*
`prompt2`	Left and Right changes summary, target method body	2*	7*
`prompt3`	Class fields, target method body	1*	6*
`prompt4`	Constructors, target method body	1*	5*
`prompt5`	Left and Right changes summary, class fields, target method body	2	8*
`prompt6`	Left and Right changes summary, constructors, target method body	1*	5*
`prompt7`	Class fields, constructors, target method body	1	6*
`prompt8`	Left and Right changes summary, class fields, constructors, target method body	1*	6*

```
    MalformedURLException {
    // Create a temporary directory for
        testing
    File tempDir = new File("target/test-
        temp");
    tempDir.mkdir();

    ExplodedArchive archive = new
        ExplodedArchive(tempDir);
    URL url = archive.getUrl();

    // Verify the expected components of
        the URL
    assertEquals("file", url.getProtocol()
        );
    assertEquals("", url.getAuthority());
    assertEquals(-1, url.getPort());
    assertTrue(url.getPath().contains(
        tempDir.getAbsolutePath().replace(
        "\\", "/")));

    // Clean up the temporary directory
    tempDir.deleteOnExit();
}
```

Listing 4: Test generated by DeepSeek R1 32B detecting the semantic conflict in `spring-boot`.

Table 6 explores how different prompt contexts influence detection, comparing the results of the Code Llama 70B baseline with the combined performance of the other models evaluated within the LUCIA tool. Across all configurations, every prompt succeeded in identifying at least one conflict. Notably, no single prompt configuration was capable of detecting all identified conflicts on its own, which highlights the importance of using complementary strategies.

While Code Llama 70B reached its highest detection rate in prompts `prompt2` and `prompt5`, the ensemble of other models achieved a higher total coverage, reaching up to eight detected conflicts in `prompt5`. In both prompts, the context provided to the model includes a summary of changes performed in the Left and Right branches (*Left and Right changes summary*), that is, a concise description of the alterations made by each developer. However, this fact does not necessarily imply a causal relationship between the presence of this context in the prompt and the quantity of detected conflicts. It is important to consider that the ”Other Models” column aggregates results from multiple architectures, while Code Llama 70B reflects the performance of a standalone model. This architectural diversity likely contributes to the superior coverage and allows for a broader evaluation of prompt robustness across different models, an analysis that is not feasible when examining Code Llama 70B in isolation. These pieces of information lead us to answer the first research question:

RQ1

The results demonstrate that no single model or configuration is capable of detecting all semantic conflicts. Different architectures showed complementary strengths: while larger models generally achieved higher recall, reasoning-oriented models identified unique, complex conflicts missed by others. Variations in parameters also played a key role, with non-deterministic settings revealing specific edge cases. Therefore, an ensemble approach combining different models and strategies proves more effective than relying on any single configuration, although it comes with increased computational costs.

4.3 RQ2: To what extent is the proposed tool effective for the automatic detection of semantic conflicts in merge scenarios?

Table 8 positions all models evaluated in this work in relation to established test generation tools integrated into SMAT. Also, the best configuration for Code Llama 70B (zero-shot, temperature 0.0) and the union of all Code Llama 70B experimental runs are highlighted for comparison.

The union of all experimental runs of Code Llama 70B detected a total of **5 conflicts**, a performance comparable to EvoSuite and superior to Randoop, although significantly more expensive. In the new study, four model configurations outperformed these results by detecting **6 conflicts** each: Llama4 16x17B and Qwen3 32B, both in the 1-shot configuration with temperatures 0.0 and 0.7. These configurations detected the same number as Differential EvoSuite, which was the best performing traditional tool.

Most models reached their best performance (italicized)

in the 1-shot configuration, except for Gemma 3 12B, which favored zero-shot at 0.0 temperature. On the other hand, temperature had a minor impact on results, with the best outcomes distributed across both settings for different models.

By combining the detections from traditional tools and LUCIA, the total number of identified conflicts in the dataset increased from 9 to 12 out of 29 scenarios. This integration represents a 15% reduction in the total number of false negatives compared to the previous state of the art in SMAT. Individually, LUCIA achieved a recall of 34.5% (10/29), surpassing the 31% (9/29) achieved by the other tools combined. A key highlight is the detection of **three novel conflicts** that were missed by all other tools. Even Differential EvoSuite, which detected the highest number of conflicts, only shared four detections with our approach, failing to identify these three unique cases.

Despite the overall improvement, LUCIA failed to detect two conflicts successfully identified by Differential EvoSuite, both within the Fitnesse project: the `setUp` method in `SlimTableFactoryTest` and `SlimTableFactory` constructor in the `SlimTableFactory` class. Our analysis revealed that the models failed in these scenarios because they could not generate a single compilable test method. The most frequent compilation errors included missing symbols for internal variables, such as `map` and `slimTableFactory`, and attempts to use dependencies not provided in the prompt context, such as the `Mockito` package. Additionally, models struggled with Java access modifiers, often attempting to access members like `Disgracer` that were not public. These failures suggest that while LLMs are effective at exploring complex logic, they remain sensitive to structural and dependency constraints where search-based tools like Differential EvoSuite are more robust.

This divergence in detected conflicts suggests that the different test generation strategies explore distinct aspects of the code, indicating that a hybrid approach combining the tools could maximize semantic conflict detection.

Furthermore, the best individual Code Llama 70B configuration was able to find 3 conflicts, surpassing established tools like Randoop and Randoop Clean, which detected only 2 conflicts each. The approximate times for the execution of these configurations were about 1 day for the zero-shot configuration with temperature 0.0 (approximately 23 hours for each of the two seeds) and more than 5 days for the union of all experimental configurations (about 132 hours). As noted in Table 3, other models also demonstrated effectiveness in conflict detection with a much lower execution time compared to Code Llama 70B, although still higher than traditional tools. A negative point, therefore, is that the proposed tool is substantially slower than the others.

Importantly, in contrast to traditional tools, no false positives were observed during the evaluation of LUCIA. Conflict detection does not depend only on tests that fail or pass, but rather on specific SMAT heuristics that analyze divergent behavior patterns between Base, Left, Right, and Merge versions, mitigating the risk of false positives due to trivial tests.

RQ2

The proposed tool proved effective for the automatic detection of semantic conflicts in merge scenarios, achieving a recall of 34.5% and surpassing traditional tools like Randoop and EvoSuite in the number of detected conflicts, while achieving detection performance competitive with Differential EvoSuite. Furthermore, the combination of different LLM configurations identified three novel conflicts missed by other SMAT tools, reducing the total number of false negatives by 15% and raising the overall detection rate to 12 out of 29 conflicts, albeit at the cost of significantly higher execution times.

4.4 RQ3: Do any of the evaluated LLMs outperform the others in test generation and semantic conflict detection?

Analyzing Tables 3, 5, and 8, we observe that no single model dominates across all performance metrics. There is a significant trade-off between execution speed and detection capability: smaller models like Gemma 3 4B are extremely fast (approx. 4 seconds per output) but miss some conflicts, while larger models like DeepSeek R1 70B provide deeper reasoning (detecting unique edge cases) but require substantially more time (over 2 minutes per output) and computational resources.

However, model size alone does not guarantee effectiveness. A notable exception is the DeepSeek R1 8B model, which generated very few tests and detected no conflicts across all configurations. This poor performance is attributed to a high frequency of timeouts during generation. The model's average execution time approached the 300-second timeout limit, indicating that most generation attempts exceeded this threshold. This suggests that the 8B parameter model, despite being designed for reasoning tasks, may have been too small to effectively handle the reasoning process required for this task, potentially getting lost in the iterative reasoning steps and failing to produce viable test outputs within the allotted time.

When compared to traditional tools, a gap remains. Although tools like Differential EvoSuite may detect fewer semantic nuances, they are generally lightweight and do not require the massive GPU infrastructure essential for running the largest models. However, LLMs offer a distinct advantage in semantic breadth. The distribution of detections across mergedataset reveals that conflicts vary significantly in difficulty. As shown in Table 5, certain conflicts such as `python2Keywords` and `getAllEventTypes()` are highly frequent, being identified by nearly all successful configurations. In contrast, conflicts like `addConnection()` and `asProperties()` are more rare, identified by only a single specific configuration each. This distribution provides a strong rationale for the use of ensembles: while smaller models effectively cover frequent conflicts, specialized configurations are required to capture the edge cases that other models might miss. To leverage this without incurring prohibitive costs, the results suggest that combining specific models and

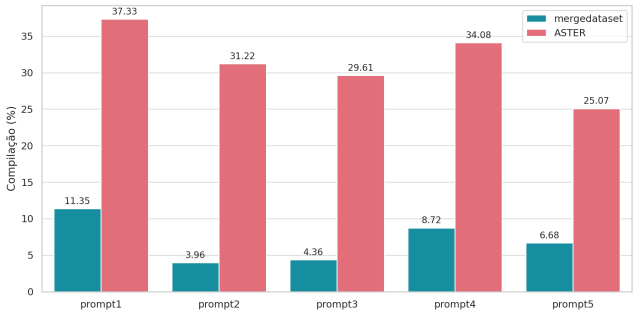

Figure 2. Comparison of the compilation rate between the mergedataset and ASTER datasets for different prompts.

configurations is superior to relying on a single "best" model. We identified two optimal combinations:

- **Surpassing Traditional Tools:** A combination of Gemma 3 4B (1-shot, T=0.0) and Gemma 3 27B (1-shot, T=0.7) detects a total of 7 distinct semantic conflicts, surpassing the 6 identified by Differential EvoSuite. Detailed performance metrics and individual execution times are available in Table 3.
- **Full Coverage:** To capture all 10 semantic conflicts identified in this study, a heterogeneous ensemble of four configurations is required: Gemma 3 27B (1-shot, T=0.7), DeepSeek R1 70B (1-shot, T=0.7), Gemma 3 4B (0-shot, T=0.0), and Gemma 3 4B (1-shot, T=0.0). As shown in Table 5, this ensemble ensures complete coverage by aggregating the unique detections of each model.

These combinations illustrate that no single model excels universally; rather, strategic ensembles tailored to specific detection goals yield the best results, while being mindful of the trade-offs between detection capability and computational cost. Thus, we answer the third research question:

RQ3

No single model stands out as the definitive solution for test generation and conflict detection. While traditional tools remain more resource-efficient, LLMs provide superior semantic coverage when deployed in ensembles. Combining fast, smaller models with specialized larger models allows for detecting a wider range of conflicts, surpassing traditional baselines in quantity and optimizing the computational cost compared to using only large models.

4.5 RQ4: How does code complexity impact the compilation rate of tests generated by LLMs?

To evaluate the impact of dataset complexity on results, we compared the compilation rate of tests generated by Code Llama 70B using the mergedataset and the ASTER dataset, as shown in the graph of Figure 2. The selected prompts 1 to 5 contain the following context parts:

- `prompt1`: (1-shot) class fields, constructors, and target method body.
- `prompt2`: (zero-shot) target method body.
- `prompt3`: (zero-shot) class fields and target method body.
- `prompt4`: (zero-shot) constructors and target method body.
- `prompt5`: (zero-shot) class fields, constructors, and target method body.

Table 7. Estimated total generation times for the mergedataset and average time per scenario. Values compare traditional tools, Code Llama 70B baselines, and the proposed ensembles.

Approach	Total Time	Avg. Time/Scenario (s)
Traditional Tools		
Differential EvoSuite	1h 57min	5.6 s
EvoSuite	6h 7min	17.4 s
Code Llama 70B Baselines		
Code Llama 70B (1-shot, T=0.0)	27h 45min	79.0 s
Code Llama 70B (zero-shot, T=0.0)	22h 58min	65.4 s
Code Llama 70B (1-shot, T=0.7)	6h 5min	17.3 s
Code Llama 70B (zero-shot, T=0.7)	9h 28min	27.0 s
Code Llama 70B (union of all configurations)	132h 32min	377.5 s
New Models & Ensembles		
Gemma 3 4B (1-shot, T=0.0)	**1h 31min**	**4.3 s**
Ensemble: Surpassing Traditional (7 conflicts)	**8h 27min**	**24.1 s**
Ensemble: Full Coverage (10 conflicts)	**73h 4min**	**208.1 s**

The selection of these specific prompts was motivated by the need to adapt the evaluation to the context of the ASTER dataset, which does not possess semantic conflict scenarios. We kept four zero-shot variations that include different combinations of context elements (class attributes, constructors, and target method body), but excluded the summary of changes from Left and Right branches, since this information is not relevant in the context. Additionally, we included a 1-shot prompt with all context information to evaluate the impact of providing an example in test generation, even in a dataset without merge scenarios.

The results show a drastic and consistent increase in the compilation rate when using ASTER: the relative gain varied from +228.9% to +688.9%. It is important to note that this study focused on the compilation capability of the generated tests, without evaluating quality metrics such as code coverage or quality consistency across multiple executions of the prompts, aspects that could complement the analysis of the tool's effectiveness. These data help to clarify the fourth research question:

RQ4

We observed that the compilation rate was much higher in the ASTER dataset compared to the mergedataset. This suggests that the complexity of the code in the dataset directly influences the model's capability to generate compilable tests.

4.6 Execution Time

Table 7 compares the execution times of traditional test generation tools (EvoSuite and Differential EvoSuite), standalone LLM-based approaches (Code Llama 70B baselines and newer models), and the proposed ensemble configurations. Previously, the tool with Code Llama 70B presented significant challenges regarding efficiency. Processing the entire mergedataset with its fastest configuration took ap-

proximately 6 hours, while the union of all configurations required over 132 hours.

However, the introduction of newer, more efficient models significantly alters this landscape. The fastest individual model evaluated, `Gemma 3 4B`, completes test generation in just **1 hour and 31 minutes** for the entire mergedataset, averaging **4.3 seconds per scenario**. This makes it not only comparable but faster than Differential EvoSuite (1h 57min), although with a lower detection rate (3 vs. 6 conflicts). Despite this lower detection rate, the highly efficient execution cycles of lightweight models like those in the Gemma family demonstrate significant potential. Their rapid processing speeds make them strong candidates for future work. Specifically, researchers could leverage these faster models to explore more exhaustive hyperparameter configurations, such as varied temperature settings.

To balance efficiency and high recall, the ensembles (detailed in Section 4.4) offer competitive alternatives:

- **Surpassing Traditional Tools:** The combination of `Gemma 3 4B` and `Gemma 3 27B` takes approximately **8h 27min**, averaging **24.1 seconds per scenario**. While this is slower than Differential EvoSuite, it achieves a higher detection rate. Crucially, this ensemble detects 7 semantic conflicts compared to EvoSuite's 5 and Differential's 6, with an efficiency rate of roughly 0.82 conflicts/hour (the same as EvoSuite, while worse than Differential's 3.06 conflicts/hour).
- **Full Coverage:** The heterogeneous ensemble required to detect all 10 conflicts totals approximately **73h 4min**, averaging **208.1 seconds per scenario**. Although computationally expensive, this is nearly **two times faster** than the union of Code Llama 70B configurations (132h) while detecting **double the number of conflicts** (10 vs. 5). The bulk of this time is attributed to the deep reasoning model (`DeepSeek R1 70B`), which is essential for capturing edge cases that no other model or tool could identify.

While traditional tools remain highly efficient for standard conflicts, modern lightweight LLMs have closed the speed gap, albeit requiring significantly more powerful hardware. Furthermore, optimized ensembles provide a scalable trade-off as they can slightly exceed the runtime of traditional tools to achieve superior detection rates or invest significant compute time to uncover complex, otherwise undetectable semantic conflicts, though at the expense of much higher energy and computational consumption compared to traditional CPU-based tools.

5 Threats to Validity

This study presents limitations that may impact the generalization and interpretation of the results.

Internal Validity. A primary concern in LLM-based studies is data contamination. Since the evaluated models (Gemma 3, DeepSeek R1, Llama 4, Qwen 3) are trained on massive datasets of public code, it is possible that projects from the mergedataset were seen during training, potentially inflating performance.

While this extension mitigates the bias of evaluating a single architecture by including models of varying sizes (4B to 70B) and families, the selection is not exhaustive and excludes proprietary models (e.g., GPT-4), which limits the assessment of the full state-of-the-art.

Furthermore, the impact of hardware heterogeneity on the results must be addressed. As the experiments were conducted across different environments (e.g., varying GPU capabilities between the Code Llama 70B baseline and newer models), the reported execution times reflect both intrinsic model complexity and resource disparities. However, it is crucial to note that while these hardware differences impact efficiency metrics (inference time), they do not introduce noise regarding the effectiveness of semantic conflict detection. The semantic content of the generated tests is determined by the model weights and parameters, ensuring that the conflict detection capabilities remain comparable regardless of the underlying infrastructure. Consequently, execution time metrics should be interpreted as indicative orders of magnitude rather than strict benchmarks.

Construct Validity. The evaluation metrics focused primarily on compilation rates and the quantitative detection of semantic conflicts. We did not extensively evaluate other quality aspects of the generated tests, such as code coverage, readability, or potential flakiness (intermittent failures not caused by the merge).

A further threat involves the technical alignment between the generated code and our experimental environment constraints. Both SMAT and the analyzed projects are based on Java 8 and JUnit 4.12. However, since the prompts (detailed in Section 3.1) do not explicitly restrict the library versions, LLMs may generate test code using features from newer frameworks. This mismatch can lead to compilation failures that do not necessarily reflect the model's inability to understand the code logic, but rather a version discrepancy. While this could potentially deflate the reported compilation rates, our observations indicate that this behavior was infrequent and did not meaningfully impact our overall findings.

The prompt engineering space is vast. While our study was limited to zero-shot and 1-shot configurations with two temperature settings (0.0 and 0.7), alternative strategies such as Chain-of-Thought or agentic approaches could potentially yield different results. Similarly, although exploring intermediate temperature values like 0.5 or the maximum limit of 1.0 might offer a more granular view of model behavior, doing so was computationally infeasible given the total execution time across multiple model families. We opted for 0.7 because it represents the default configuration for many models, reflecting common usage and providing a balance between diversity and syntactic coherence.

External Validity. The results are based on the mergedataset, which, although composed of real-world open-source projects, is restricted to the Java language. Consequently, the findings may not extrapolate to other programming languages with different syntactic structures or type systems.

Another significant limitation is the sample size; with only 79 merge scenarios, the statistical power of the conclusions is constrained. The non-deterministic nature of LLMs also poses a threat to reproducibility. While we fixed seed val-

ues to minimize variance, we chose 42 and 123 as arbitrary constants. These specific values may not represent the most optimal configuration for the models. Although evaluating a broader range of seeds would further attest to the stability of the results, doing so was limited by the computational costs of the experimental runs. Furthermore, inherent stochasticity in floating-point operations on GPUs means that exact replication of text generation may not always be guaranteed across different hardware environments.

6 Related Work

This section presents a detailed analysis of related work, organized into tools for semantic conflict detection and resolution in merges, approaches for automated test generation using LLMs and agentic techniques. For each category, we discuss the methodologies employed, the results obtained, and the main differences regarding our proposal.

Semantic Conflict Detection and Resolution The problem of semantic conflicts in merges has been addressed from different perspectives in the literature. SafeMerge (Sousa et al. (2018)), used as a baseline in the work by Silva et al. (2020), applies compositional verification to detect semantic conflicts in merges. Its approach is grounded on the construction of a parameterized program, in which changes introduced by different code versions (Left and Right) are encapsulated as edits in a common structure derived from the Base version. This parameterized program is then submitted to a relational verification that seeks to prove, by formal means, that the behavior of the code resulting from the merger is compatible with the expected behaviors of both modified versions. This technique allows capturing semantic conflicts not observable from simple textual analysis, promoting a deeper level of verification. Despite its formal rigor, SafeMerge presents practical limitations. The computational complexity of relational verification, especially the construction of product programs and the need to generate invariants for behavioral equivalence validation, may compromise the tool's scalability in real-world scenarios and larger codebases. Furthermore, when using the same heuristic criteria adopted by SMAT for conflict characterization, it is observed that SafeMerge's approach generates more false positives. Specifically, SafeMerge presents an imprecision rate of 3.8%, while our LLM-based approach reached a rate of 0% false positives, demonstrating greater precision in semantic conflict detection. For comparison, SMAT itself presents a rate of 3%, which reinforces the advantage of our proposal in this aspect. It is worth noting that this zero false-positive rate remained consistent across all evaluated LLMs in our current study, reinforcing the robustness of the approach across different model architectures and parameter settings. This precision is particularly significant given that our approach detected 3 novel semantic conflicts not identified by any other SMAT tool, demonstrating both high precision and improved recall.

The work by Zhang et al. (2022) proposes the tool *GMERGE*, which suggests automatic solutions for merge conflicts, including both textual and semantic conflicts. Like our approach, *GMERGE* uses pre-trained LLMs and explores k-shot strategies in prompt creation. However, there are important differences: *GMERGE* focuses on conflicts in a single project (Microsoft Edge) and its evaluation is centered on the observation of compiler messages. The perception of conflict for the authors is defined when there are no textual merge conflicts, but the merge still results in a broken build, failing tests, or unintentional runtime behavior. Conflict detection, for them, depends on the occurrence of compilation errors, as the tool uses Clang messages to identify relevant commits that cause conflicts. This can be a problem if the compiler provides poorly targeted messages. In contrast, our perception of semantic conflict occurs when parallel changes result in test failures or runtime errors, even if the code is merged and compiled successfully, which differs from build conflicts, the focus of the work by Zhang et al. (2022). Furthermore, while *GMERGE* mainly explores the feasibility of repairing conflicts, our focus is on detection. The complementarity between *GMERGE* and our approach is particularly interesting: while *GMERGE* acts as a repair tool focused on conflicts that have already manifested through compilation errors, our proposal works as a 'preventive detector" that identifies semantic conflicts. This difference in scope suggests a hybrid approach as future work, where our methodology would be applied first to detect latent conflicts, followed by a modified version of *GMERGE* to resolve them.

Another relevant approach is presented by Maciel et al. (2024). It is another extension of the SMAT tool. Their approach aims to overcome the limitation of existing tools that are restricted to simpler scenarios, where conflicting contributions occur within the same method. To this end, they adapt and evaluate SMAT to detect conflicts considering the interference caused by changes made in different methods and classes. The tool explores test creation using test generation tools already incorporated into SMAT, in addition to a customized version of EvoSuite called Focused EvoSuite. Focused EvoSuite was proposed to optimize test generation, focusing exclusively on altered methods, unlike the standard version of EvoSuite which considers criteria applicable to the entire class. It is important to note that the dataset used by Maciel et al. (2024) is different from ours (mergedataset). They used a sample of 613 synthetic merge scenarios created from the Defects4J benchmark (Just et al. (2014)), representing a sample six times larger than previous studies. All 613 scenarios in their sample presented at least one semantic conflict identified by the execution of the project's own tests. The results revealed the detection of 230 distinct semantic conflicts, which represents 37.5% of the total sample. In comparison, our work used 79 scenarios, with 29 scenarios containing semantic conflicts. Across all evaluated LLMs and configurations, our approach detected a total of 10 distinct semantic conflicts, including 3 novel conflicts not identified by any previous SMAT tool. Notably, four individual configurations (Llama 4 16x17B and Qwen 3 32B, both with 1-shot prompts at temperatures 0.0 and 0.7) matched the performance of Differential EvoSuite (6 conflicts each), while a strategic ensemble of two lightweight models (Gemma 3 4B and Gemma 3 27B) detected 7 conflicts, surpassing all traditional tools. Both approaches seek to improve SMAT's capability to detect semantic conflicts, but through distinct strategies: they improve the granularity of test generation with ex-

isting tools, while we explore the potential of LLMs to generate more effective tests. The results of Maciel et al. (2024) and ours reinforce the potential of tools in conflict identification, and both studies emphasize the importance of adopting multiple automated test strategies and tools to detect this type of conflict.

Regarding an approach with dynamic analysis, we highlight the work by Moraes et al. (2024), which proposes a methodology for detecting semantic conflicts in JavaScript. They employ dynamic analysis to specifically detect "overriding assignments" (OAs) in JavaScript code. The approach does not need asserts in the code under analysis, requiring only that the final version of the merged code be executed, unlike our test-based approach. To validate their proposal, the authors transpiled more than 60 test cases from a related work and performed an empirical study with 159 merge scenarios from 50 open-source JavaScript projects. In that study, they correctly detected one semantic conflict scenario of overriding assignment, with zero false positives, just like our approach. Their approach also differs by being specific to JavaScript, while our work, a priori, is aimed at Java, but could be adapted to other languages with some modifications in the test generation and execution pipeline. The work demonstrates the potential of dynamic analysis for semantic conflict detection and can be seen as complementary to our approach, which relies on tests and LLMs.

Finally, we present a work focusing on static analysis for semantic conflict detection. In their work, De Jesus et al. (2024) propose a technique that executes static analysis algorithms on the merged version of the code, which is annotated with metadata to indicate instructions modified by each developer. This technique was implemented for the Java language using the Soot framework and includes four main analyses: Interprocedural Data Flow (DF), Interprocedural Confluence (CF), Interprocedural Override Assignment (OA), and Program Dependence Graph (PDG). The technique was evaluated with a dataset of 99 experimental units extracted from merge scenarios of 39 projects, with 33 units with interference and 66 without. The results showed a significant capability for interference detection, with an F1 Score of 0.50 and an Accuracy of 0.60. Compared to previous works, the approach by De Jesus et al. (2024) presented a better F1 Score and recall, but with significantly lower precision, generating 26 false positives. Despite this, the computational performance was notably better, with a median execution time of 17.8 seconds for the four analyses. In contrast to our approach, the work by De Jesus et al. (2024) uses exclusively static analysis, while our methodology combines LLM-based test generation with dynamic analysis through the execution of these tests. This fundamental difference results in distinct trade-offs: while the static analysis by De Jesus et al. (2024) offers significantly lower execution times (17.8 seconds), our approach achieves superior precision, recording no false positives compared to the 26 false positives reported by them. Regarding execution times, our current study demonstrates substantial improvements: while Code Llama 70B configurations ranged from 6 to 28 hours per configuration, newer lightweight models such as Gemma 3 4B complete test generation in approximately 1.5 hours for the entire dataset, making them competitive with traditional tools like Differential EvoSuite (1h 57min) in terms of execution speed.

Test Generation with LLMs In the context of test generation with LLMs, we highlight four recent approaches: *TESTPILOT* Schäfer et al. (2024), *ASTER* Pan et al. (2025), *CITYWALK* Zhang et al. (2026), and *HITS* Wang et al. (2024). *TESTPILOT* is a test generation tool aimed at JavaScript, which contrasts with our approach, which focuses on Java; its pipeline includes extracting documentation comments to enrich the context provided to the model, while our pipeline relies only on structural code information (attributes, constructors, and method body), without resorting to textual documentation. Paradoxically, this limitation can become an advantage in the context of semantic conflict detection, as it forces the LLM to rely exclusively on the structure and semantics of the code, reducing the possibility of being 'misled" by inconsistent documentation. Furthermore, *TESTPILOT* prioritizes generating tests that do not fail, seeking to maximize coverage, while our focus is on detecting semantic conflicts in merge scenarios. We observed in our study that using different contexts in prompts led to the detection of distinct conflicts, suggesting that different combinations of prompt and model parameters allow exploring different types of faults and maximizing detection. While *TESTPILOT* demonstrates that including documentation and usage examples improves effectiveness in test generation, our results indicate that, even without textual documentation, structural code information is sufficient to detect semantic conflicts in Java. Moreover, our study reveals that the 1-shot configuration (which provides one example in the prompt) was consistently more effective across nearly all evaluated models, with the top ten compilation rates all belonging to 1-shot configurations. This pattern held true for conflict detection as well: for almost all models, the best performance was achieved with 1-shot prompts.

ASTER, in turn, is a tool for test generation in Java and Python, which explores different contexts and mocking techniques to increase the coverage of generated tests. Unlike our approach, which executes test generation in a single step for each scenario, *ASTER* adopts an iterative process, refining tests until a desired coverage is reached. Additionally, *ASTER* emphasizes the utility and understandability of generated tests, aspects also considered in our analysis, but with an emphasis on conflict detection capability. In terms of context for prompt generation, *ASTER* uses a pre-processing phase that performs extensive static analysis of the program. This includes extracting detailed information such as the test scope (target methods), relevant constructors, auxiliary methods (getters/setters), call chains for private methods, and data to facilitate mocking (identifying attributes, types, and methods to be mocked). This context-rich approach contrasts with ours, which relies on a simpler analysis, focusing only on attributes, target class constructors, and the target method body. Moreover, the nature of the dataset we used does not require identifying private methods or attributes that are not directly accessible, which simplifies the test generation process. As for post-processing, *ASTER* employs a phase dedicated to repairing generated tests, correcting compilation and runtime errors. This involves sanitizing the LLM output, correcting assertion failures, and, notably, a coverage augmentation pro-

cess, where additional prompts are created to instruct the LLM to generate test cases that exercise lines of code initially uncovered. In our work, the focus of post-processing is on validating tests to identify semantic conflicts; although the generated tests are executed to determine their behavior, there is no explicit test repair phase or iterative coverage increase with the LLM in the same way as in *ASTER*. Our tests are evaluated primarily by their ability to break in conflict scenarios, and not by their total coverage.

CITYWALK proposes a pipeline for test generation in C++ that incorporates Retrieval-Augmented Generation (RAG) techniques. Its main focus is to generate executable and high-coverage unit tests capable of finding bugs, overcoming specific C++ challenges such as pointers and virtual functions. In terms of contexts used, *CITYWALK* employs extensive static analysis to extract configuration dependencies (use of third-party libraries and build environment requirements) and cross-file data dependencies (data flow analysis in ASTs to capture interactions between files). Furthermore, it retrieves code snippets and documentation as intent contexts, using a hybrid retrieval strategy (semantic for documentation and exact match for code). In contrast, our approach, focused on Java, uses information local to the class. Although our approach does not use RAG, we recognize the potential of this technique for future works, especially because contextual knowledge external to the class source code can be crucial for detecting semantic conflicts that may emerge from complex interactions between different system parts that are not obvious from the local context. One point of convergence between *CITYWALK* and our approach is the exploration of different LLMs for test generation. They evaluate models such as GPT-4o, CodeGeeX4, and DeepSeek-V2.5. Similarly, our current study explores model diversity, but with a focus on open-source models accessible through Ollama: Code Llama 70B, DeepSeek R1 (8B, 32B and 70B), Gemma 3 (4B, 12B, and 27B), Qwen 3 (30B and 32B), and Llama 4 (16x17B). This choice prioritizes reproducibility and accessibility, enabling researchers to replicate experiments without proprietary API dependencies. Our results demonstrate that this diverse set of open-source models can not only detect semantic conflicts missed by traditional tools but also, when strategically combined, surpass the performance of individual configurations. *CITYWALK*'s results show that the tool generates fewer compilation errors and achieves higher coverage with fewer tests when compared to solutions that use models directly, reinforcing the potential of LLMs in test generation, when well associated with other processes. In particular, *CITYWALK* outperformed GPT-4o by 42.05% in compilation success rate and by 26.41% in line coverage, with a significantly smaller quantity of generated tests. In comparison, our approach shows varying compilation rates depending on the model and dataset used. For the mergedataset, compilation rates hit a maximum of 31.68% (with Gemma 3 12B, 1-shot, temperature 0.0), while Code Llama 70B average is 8.24%. For the simpler *ASTER dataset*, Code Llama 70B achieved substantially higher rates (31.46% on average), demonstrating that dataset complexity significantly impacts compilation success. While *CITYWALK* reached an average compilation rate of 70.18% on a C++ dataset, this difference can be attributed not only to the nature of the datasets used, but also to the approaches' distinct focus: *CITYWALK* employs extensive RAG-based context augmentation specifically optimized for compilation success, while our approach prioritizes semantic conflict detection capability.

Finally, Wang et al. (2024) propose an approach for high-coverage test generation in complex Java methods. HITS solves the problem of LLMs performing poorly on complex methods by decomposing them into code "slices" and generating tests for each one through Chain-of-Thought (CoT) prompting, differing from our simpler prompting strategy. "Slicing" simplifies the LLM analysis scope, increasing the coverage of generated tests. HITS outperforms other LLM-based approaches and EvoSuite in terms of line and branch coverage. This approach is relevant to our proposal, as both investigate test generation with LLMs in complex methods and use EvoSuite as a reference, although with different focuses. Other distinctions lie in the model choice and parameters: HITS uses gpt-turbo-3.5-0125 and evaluates top-p and temperature, while our approach explores a diverse set of models (Code Llama 70B, DeepSeek R1, Gemma 3, Qwen 3, and Llama 4 in various sizes) and focuses on temperature, seed and prompt format variations (zero-shot vs 1-shot). Additionally, HITS includes a step for retrieving non-executable tests, absent in our methodology.

Agentic Approaches for Test Generation Yuan et al. (2024) proposed *ChatTester*, a technique designed to mitigate the generation of uncompilable or incorrect tests by models like ChatGPT. Evaluating on the Defects4J dataset, they introduced an iterative feedback loop where compiler and execution errors are fed back to the model to refine the tests. More recently, Silva et al. (2025) extended this line of inquiry with *ChatTesterMut*, a comparative benchmarking study evaluated on the SF110 dataset. Their work contrasts proprietary models (e.g., Gemini 1.5 Flash, GPT-4o) against small open-weights models (e.g., Gemma 2, DeepSeekCoder), focusing on test quality metrics such as mutation scores, code smells, and cyclomatic complexity. While *ChatTesterMut* provides valuable insights into the individual capabilities of small models for general unit testing, our work differs fundamentally in scope, objective, and strategy. First, we shift the focus from generating tests for isolated stable classes to the dynamic context of software merges, processing tuples of four versions (Base, Left, Right, Merge). Second, our evaluation metric is not mutation coverage or code quality, but the specific ability to expose semantic conflicts — defined as behavioral divergences that can be revealed through some heuristics in test execution results across the merge versions. Finally, while *ChatTesterMut* benchmarks models individually to identify the best performers, our findings demonstrate that combined model configurations yield superior results in conflict detection. Instead, we propose a heterogeneous ensemble approach, leveraging the complementary strengths of models ranging from 4B to 70B parameters to uncover subtle integration issues that individual models miss. Specifically, our results show that no single model detects all semantic conflicts: the best individual configurations (Llama 4 16x17B and Qwen 3 32B) detected 6 conflicts each, matching Differential EvoSuite's performance. However, strategic ensembles can detect 7 con-

flicts (Gemma 3 4B + Gemma 3 27B) or even all 10 identified conflicts when combining four complementary configurations, demonstrating the value of model diversity over individual model optimization.

Another notable agentic approach is presented by Jain and Goues (2025). They introduced *TestForge*, an agentic framework for feedback-driven test generation. Unlike standard zero-shot approaches, TestForge treats test generation as an iterative process, using execution feedback and coverage reports to refine tests for Python projects. While *TestForge* focuses on an agentic loop to maximize correctness (pass rate) and coverage through iterative repairs using a single model backend, our work adopts a different strategy: instead of refining a single test suite iteratively, we exploit the diverse reasoning capabilities of different architectures to uncover semantic conflicts in Java. Furthermore, while *TestForge* aims for tests that pass and cover code, our objective is to generate tests that expose behavioral divergences between merge versions (i.e., tests that pass in parents but fail in the merge), a specific challenge that requires capturing subtle semantic interferences rather than just achieving high coverage. Hence, while both works leverage LLMs for test generation, they address different problems with distinct methodologies. Our multi-model evaluation reveals that different architectures exhibit complementary detection capabilities: reasoning-oriented models like DeepSeek R1 70B detect unique complex conflicts (e.g., post-integration changes in URL handling), while faster models like Gemma 3 efficiently capture more common patterns, suggesting that the iterative refinement strategy of *TestForge* could potentially be enhanced by incorporating model diversity rather than relying solely on execution feedback from a single model backend.

7 Conclusion

This work demonstrated that a tool based on large language models represent a promising alternative for the detection of semantic conflicts in code. We explored multiple open-source models accessible through the Ollama framework, including Code Llama 70B, DeepSeek R1 (32B and 70B), Gemma 3 (4B, 12B, and 27B), Qwen 3 (30B and 32B), and Llama 4 (16x17B), across different prompt strategies (zero-shot and 1-shot) and temperature settings (0.0 and 0.7). The results obtained surpass traditional automated tools, achieving a recall of 34.5% by detecting 10 distinct conflicts out of the 29 present in the mergedataset. This includes three novel conflicts not identified by any previous SMAT tool, without introducing false positives across all configurations.

The experiments revealed four key findings regarding the use of LLMs for semantic conflict detection. First, no single model or configuration was able to identify all observed conflicts: heterogeneous ensembles combining different models and configurations proved significantly more effective, with different architectures detecting complementary types of conflicts. The best individual configurations (Llama 4 16x17B and Qwen 3 32B with 1-shot prompts) matched Differential EvoSuite's performance (6 conflicts), while strategic ensembles surpassed it, detecting 7 conflicts (Gemma 3 4B + 27B) or all 10 conflicts when combining four complementary configurations, although with increased computational costs. Second, the *1-shot prompt* format consistently outperformed *zero-shot* across nearly all models, with all top ten compilation rates belonging to 1-shot configurations. This contradicts our previous findings with Code Llama 70B alone, suggesting that optimal prompt strategies may be model dependent. Third, model size and execution time do not directly correlate with detection capability: smaller, faster models like Gemma 3 4B (completing in 1.5 hours) detected unique conflicts missed by larger models, while reasoning oriented models like DeepSeek R1 70B identified complex edge cases requiring deeper analysis. Finally, dataset complexity significantly impacts compilation rates: the simpler *ASTER dataset* yielded substantially higher compilation rates (31.46% average for Code Llama 70B) compared to the *mergedataset* (3.67%–31.68% range across models), confirming that code complexity directly influences test generation success.

The results present direct implications for the development of LLM-based tools for semantic conflict detection. The diversity in detection coverage across models and configurations demonstrates that ensemble strategies, rather than single model optimization, should be the focus of future research. Our results show that integrating LUCIA into the SMAT ecosystem, using the four configurations of the Full Coverage ensemble, increases the total detection rate from 9 to 12 out of 29 conflicts. This represents a 15% reduction in the total number of false negatives compared to the previous state of the art. When analyzing ensemble strategies, we noted that pairing lightweight models (Gemma 3 4B, 1.5h execution) with mid-sized models (Gemma 3 27B) is sufficient to surpass traditional tools. Meanwhile, employing the four-model ensemble achieves full coverage (10 conflicts) in 73 hours — significantly more efficient than the 132+ hours required by Code Llama 70B alone for fewer conflicts (5). Beyond efficiency, the identification of three unique conflicts previously undetected represents a 200% improvement compared to the single unique conflict identified by Code Llama 70B in our previous work (Barbosa et al. (2025)).

The predominance of 1-shot configurations in achieving better results highlights the importance of providing examples in prompts, though optimal strategies vary by model architecture. The computational cost analysis reveals that modern lightweight LLMs have closed the efficiency gap with traditional tools: Gemma 3 4B executes faster than Differential EvoSuite (1.5h vs 2h), demonstrating that LLM-based detection is now practically viable for real world scenarios. Future works should investigate intelligent ensemble selection algorithms that automatically identify optimal model combinations for specific project characteristics, balancing detection coverage with computational resources.

As future directions, we intend to investigate methods to improve the compilation rate of generated tests, particularly for complex codebases. Given the distinct detection patterns observed between different architectures, we plan to explore hybrid approaches that leverage reasoning oriented models like DeepSeek R1 for complex scenarios while using faster models like Gemma 3 4B for common patterns. The high efficiency of these lightweight models also presents a unique opportunity for exhaustive hyperparameter exploration. Future

research could investigate how broader temperature settings or alternative sampling strategies impact detection stability, an evaluation that remains computationally expensive for larger architectures. Furthermore, we plan to investigate the integration of RAG techniques to provide richer context, explore agentic approaches with iterative refinement, and evaluate the potential of combining LLM-based detection with traditional static analysis tools to create comprehensive semantic conflict detection systems. Finally, adapting the tool to other programming languages beyond Java represents a promising direction.

Artifact Availability

The source code of the SMAT version used in this work, including access to local LLMs (LUCIA), is available at Barbosa (2025). The experimental artifacts can be accessed at Barbosa et al. (2026).

Acknowledgements

For partially supporting this work, we would like to thank INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grants 408817/ 2024-0 and 401032/ 2025-6, and the members of the Software Productivity Group (SPG).

References

Barbosa, N. (2025). SMAT with LUCIA tool integration. `https://github.com/spgroup/SMAT/pull/18`.

Barbosa, N., Borba, P., and Da Silva, L. (2026). Supplementary data for: Assessing the effectiveness of large language models in detecting semantic conflicts. `https://doi.org/10.5281/zenodo.20938400`.

Barbosa, N., Borba, P., and Silva, L. (2025). Detecção de conflitos semânticos com testes gerados por llm. In *Anais do X Simpósio Brasileiro de Testes de Software Sistemático e Automatizado*, pages 18–27, Porto Alegre, RS, Brasil. SBC.

Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. (2011). Crystal: precise and unobtrusive conflict warnings. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 444–447, New York, NY, USA. Association for Computing Machinery.

Da Silva, L., Borba, P., Maciel, T., Mahmood, W., Berger, T., Moisakis, J., Gomes, A., and Leite, V. (2024). Detecting semantic conflicts with unit tests. *Journal of Systems and Software*, 214:112070.

Da Silva, L., Borba, P., and Pires, A. (2022). Build conflicts in the wild. *Journal of Software: Evolution and Process*, 34(4):e2441.

De Jesus, G. S., Borba, P., Bonifácio, R., and De Oliveira, M. B. (2024). Lightweight semantic conflict detection with static analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 343–345, New York, NY, USA. Association for Computing Machinery.

Dodge, J., Ilharco, G., Schwartz, R., Farhadi, A., Hajishirzi, H., and Smith, N. (2020). Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping.

Eclipse Foundation (2020). Eclipse Cargo Tracker: Applied Domain-Driven Design Blueprints for Jakarta EE. `https://github.com/eclipse-ee4j/cargotracker`.

Fraser, G. and Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419.

Guo, D., Yang, D., Zhang, H., et al. (2025). Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. (2020). The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Jain, K. and Goues, C. L. (2025). Testforge: Feedback-driven, agentic test suite generation.

Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA. Association for Computing Machinery.

Maciel, T., Borba, P., Silva, L., and Burity, T. (2024). Explorando a detecção de conflitos semânticos nas integrações de código em múltiplos métodos. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software*, pages 181–191, Porto Alegre, RS, Brasil. SBC.

Maddila, C., Nagappan, N., Bird, C., Gousios, G., and van Deursen, A. (2021). Cone: A concurrent edit detection tool for large-scale software development. *ACM Trans. Softw. Eng. Methodol.*, 31(2).

Meta AI (2025). The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. `https://ai.meta.com/blog/llama-4-multimodal-intelligence/`.

Moraes, A., Borba, P., and Silva, L. (2024). Semantic conflict detection via dynamic analysis. In *Anais do XXVIII Simpósio Brasileiro de Linguagens de Programação*, pages 53–61, Porto Alegre, RS, Brasil. SBC.

Ollama (2023). Ollama. `https://ollama.com/`.

OpenLiberty (2018). DayTrader8 Sample. `https://github.com/OpenLiberty/sample.daytrader8`.

Pacheco, C. and Ernst, M. D. (2007). Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816.

Pan, R., Kim, M., Krishna, R., Pavuluri, R., and Sinha, S. (2025). ASTER: Natural and Multi-Language Unit Test Generation with LLMs . In *2025 IEEE/ACM 47th Interna-*

tional Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pages 413–424, Los Alamitos, CA, USA. IEEE Computer Society.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. (2024). Code llama: Open foundation models for code.

Sarma, A., Redmiles, D. F., and van der Hoek, A. (2012). Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, 38(4):889–908.

Schäfer, M., Nadi, S., Eghbali, A., and Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

Silva, E., Coelho, R., and Silva, L. (2025). Llms as test generators: A comparative benchmarking study. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software*, pages 25–36, Porto Alegre, RS, Brasil. SBC.

Silva, L. D., Borba, P., Mahmood, W., Berger, T., and Moisakis, J. (2020). Detecting semantic conflicts via automated behavior change detection. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 174–184.

Sousa, M., Dillig, I., and Lahiri, S. K. (2018). Verified three-way program merge. *Proc. ACM Program. Lang.*, 2(OOPSLA).

Team, G., Kamath, A., Ferret, J., et al. (2025). Gemma 3 technical report.

Tree-sitter (2024). Tree-sitter: A parser generator tool and incremental parsing library. `https://tree-sitter.github.io/tree-sitter/`.

Wang, Z., Liu, K., Li, G., and Jin, Z. (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*, ASE '24, page 1258–1268, New York, NY, USA. Association for Computing Machinery.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., and Schmidt, D. C. (2023). A prompt pattern catalog to enhance prompt engineering with chatgpt. In *Proceedings of the 30th Conference on Pattern Languages of Programs*, PLoP '23, USA. The Hillside Group.

Yang, A., Li, A., Yang, B., et al. (2025). Qwen3 technical report.

Yuan, Z., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., and Lou, Y. (2024). Evaluating and improving chatgpt for unit test generation. *Proc. ACM Softw. Eng.*, 1(FSE).

Zhang, J., Kaufman, M., Mytkowicz, T., Piskac, R., and Lahiri, S. (2022). Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper). In *ISSTA 2022: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM.

Zhang, Y., Lu, Q., Liu, K., Dou, W., Zhu, J., Qian, L., Zhang, C., Lin, Z., and Wei, J. (2026). Citywalk: Enhancing llm-based c++ unit test generation via project-dependency awareness and language-specific knowledge. *ACM Trans. Softw. Eng. Methodol.*, 35(5).

A Tables

Table 8. Conflicts detected by tool. [1] represents false positives. Data extracted from Da Silva et al. (2024). In bold: traditional test generation tools and Code Llama 70B. In italics: better configurations of LLM-based test generation tools.

Test Generation Tool	Conflicts Detected
Differential EvoSuite	6
Llama4 16x17B (1-shot, temperature 0.0)	6
Llama4 16x17B (1-shot, temperature 0.7)	6
Qwen3 32B (1-shot, temperature 0.0)	6
Qwen3 32B (1-shot, temperature 0.7)	6
EvoSuite	5 [1]
Code Llama 70B (union of all experimental configurations)	5
DeepSeek R1 32B (1-shot, temperature 0.7)	5
DeepSeek R1 70B (1-shot, temperature 0.0)	5
DeepSeek R1 70B (1-shot, temperature 0.7)	5
Gemma3 27B (1-shot, temperature 0.7)	5
Gemma3 4B (zero-shot, temperature 0.0)	5
Llama4 16x17B (zero-shot, temperature 0.0)	5
Llama4 16x17B (zero-shot, temperature 0.7)	5
Qwen3 30B (1-shot, temperature 0.0)	5
Qwen3 30B (1-shot, temperature 0.7)	5
Qwen3 32B (zero-shot, temperature 0.7)	5
DeepSeek R1 32B (1-shot, temperature 0.0)	4
DeepSeek R1 32B (zero-shot, temperature 0.7)	4
Gemma3 12B (1-shot, temperature 0.0)	4
Gemma3 12B (1-shot, temperature 0.7)	4
Gemma3 27B (1-shot, temperature 0.0)	4
Gemma3 4B (1-shot, temperature 0.7)	4
Gemma3 4B (zero-shot, temperature 0.7)	4
Qwen3 30B (zero-shot, temperature 0.7)	4
Qwen3 32B (zero-shot, temperature 0.0)	4
Code Llama 70B (zero-shot, temperature 0.0)	3
DeepSeek R1 70B (zero-shot, temperature 0.0)	3
DeepSeek R1 70B (zero-shot, temperature 0.7)	3
Gemma3 12B (zero-shot, temperature 0.0)	3
Gemma3 12B (zero-shot, temperature 0.7)	3
Gemma3 27B (zero-shot, temperature 0.0)	3
Gemma3 4B (1-shot, temperature 0.0)	3
Qwen3 30B (zero-shot, temperature 0.0)	3
Randoop	2
Randoop Clean	2 [1]
DeepSeek R1 32B (zero-shot, temperature 0.0)	2
Gemma3 27B (zero-shot, temperature 0.7)	2
DeepSeek R1 8B (1-shot, temperature 0.7)	0
DeepSeek R1 8B (zero-shot, temperature 0.7)	0
DeepSeek R1 8B (1-shot, temperature 0.0)	0
DeepSeek R1 8B (zero-shot, temperature 0.0)	0

B Listings

```
private Set<String> getAllEventTypes() {
    Set<String> eventTypes = new HashSet<>();
    eventTypes.add(EventConstants.SCORE_FINISHED_EVENT);
    eventTypes.add(EventConstants.SCORE_BRANCH_FAILURE_EVENT);
    eventTypes.add(EventConstants.SCORE_FINISHED_BRANCH_EVENT);
    eventTypes.add(EventConstants.SCORE_NO_WORKER_FAILURE_EVENT);
    eventTypes.add(EventConstants.SCORE_PAUSED_EVENT);
    eventTypes.add(EventConstants.SCORE_ERROR_EVENT);
    eventTypes.add(EventConstants.SCORE_FAILURE_EVENT);
    eventTypes.add(ScoreLangConstants.SLANG_EXECUTION_EXCEPTION);
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_START);
```

```
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_END);
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_ERROR);
    eventTypes.add(ScoreLangConstants.EVENT_INPUT_START);
    eventTypes.add(ScoreLangConstants.EVENT_INPUT_END);
    eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_START);
    eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_END);
    eventTypes.add(ScoreLangConstants.EVENT_BRANCH_START);
    eventTypes.add(ScoreLangConstants.EVENT_BRANCH_END);
    eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_EXPRESSION_START);
    eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_EXPRESSION_END);
    eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_OUTPUT_START);
    eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_OUTPUT_END);
    eventTypes.add(ScoreLangConstants.EVENT_EXECUTION_FINISHED);
    return eventTypes;
}
```

Listing 5: Base version of the `getAllEventTypes()` method from the `cloud-slang::SlangImpl` class.

```
private Set<String> getAllEventTypes() {
    Set<String> eventTypes = new HashSet<>();
    eventTypes.add(EventConstants.SCORE_FINISHED_EVENT);
    eventTypes.add(EventConstants.SCORE_BRANCH_FAILURE_EVENT);
    eventTypes.add(EventConstants.SCORE_FINISHED_BRANCH_EVENT);
    eventTypes.add(EventConstants.SCORE_NO_WORKER_FAILURE_EVENT);
    eventTypes.add(EventConstants.SCORE_PAUSED_EVENT);
    eventTypes.add(EventConstants.SCORE_ERROR_EVENT);
    eventTypes.add(EventConstants.SCORE_FAILURE_EVENT);
    eventTypes.add(ScoreLangConstants.SLANG_EXECUTION_EXCEPTION);
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_START);
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_END);
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_ERROR);
    eventTypes.add(ScoreLangConstants.EVENT_INPUT_START);
    eventTypes.add(ScoreLangConstants.EVENT_INPUT_END);
    eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_START);
    eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_END);
    eventTypes.add(ScoreLangConstants.EVENT_BRANCH_START);
    eventTypes.add(ScoreLangConstants.EVENT_BRANCH_END);
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_EXPRESSION_START);
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_EXPRESSION_END);
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_OUTPUT_START);
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_OUTPUT_END);
    + eventTypes.add(ScoreLangConstants.EVENT_SPLIT_BRANCHES);
    + eventTypes.add(ScoreLangConstants.EVENT_JOIN_BRANCHES_START);
    + eventTypes.add(ScoreLangConstants.EVENT_JOIN_BRANCHES_END);
    eventTypes.add(ScoreLangConstants.EVENT_EXECUTION_FINISHED);
    return eventTypes;
}
```

Listing 6: Right version of the `getAllEventTypes()` method from the `cloud-slang::SlangImpl` class. Added lines are highlighted in green, while deleted lines are marked in red.

```
private Set<String> getAllEventTypes() {
    Set<String> eventTypes = new HashSet<>();
    eventTypes.add(EventConstants.SCORE_FINISHED_EVENT);
    eventTypes.add(EventConstants.SCORE_BRANCH_FAILURE_EVENT);
    eventTypes.add(EventConstants.SCORE_FINISHED_BRANCH_EVENT);
    eventTypes.add(EventConstants.SCORE_NO_WORKER_FAILURE_EVENT);
    eventTypes.add(EventConstants.SCORE_PAUSED_EVENT);
    eventTypes.add(EventConstants.SCORE_ERROR_EVENT);
    eventTypes.add(EventConstants.SCORE_FAILURE_EVENT);
    eventTypes.add(ScoreLangConstants.SLANG_EXECUTION_EXCEPTION);
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_START);
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_END);
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_ERROR);
    + eventTypes.add(ScoreLangConstants.EVENT_TASK_START);
    eventTypes.add(ScoreLangConstants.EVENT_INPUT_START);
```

```
    eventTypes.add(ScoreLangConstants.EVENT_INPUT_END);
    eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_START);
    eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_END);
    eventTypes.add(ScoreLangConstants.EVENT_BRANCH_START);
    eventTypes.add(ScoreLangConstants.EVENT_BRANCH_END);
    eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_EXPRESSION_START);
    eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_EXPRESSION_END);
    eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_OUTPUT_START);
    eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_OUTPUT_END);
    eventTypes.add(ScoreLangConstants.EVENT_EXECUTION_FINISHED);
    return eventTypes;
}
```

Listing 7: Left version of the `getAllEventTypes()` method from the `cloud-slang::SlangImpl` class. Added lines are highlighted in green, while deleted lines are marked in red.

```
private Set<String> getAllEventTypes() {
    Set<String> eventTypes = new HashSet<>();
    eventTypes.add(EventConstants.SCORE_FINISHED_EVENT);
    eventTypes.add(EventConstants.SCORE_BRANCH_FAILURE_EVENT);
    eventTypes.add(EventConstants.SCORE_FINISHED_BRANCH_EVENT);
    eventTypes.add(EventConstants.SCORE_NO_WORKER_FAILURE_EVENT);
    eventTypes.add(EventConstants.SCORE_PAUSED_EVENT);
    eventTypes.add(EventConstants.SCORE_ERROR_EVENT);
    eventTypes.add(EventConstants.SCORE_FAILURE_EVENT);
    eventTypes.add(ScoreLangConstants.SLANG_EXECUTION_EXCEPTION);
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_START);
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_END);
    eventTypes.add(ScoreLangConstants.EVENT_ACTION_ERROR);
    + eventTypes.add(ScoreLangConstants.EVENT_TASK_START);
    eventTypes.add(ScoreLangConstants.EVENT_INPUT_START);
    eventTypes.add(ScoreLangConstants.EVENT_INPUT_END);
    eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_START);
    eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_END);
    eventTypes.add(ScoreLangConstants.EVENT_BRANCH_START);
    eventTypes.add(ScoreLangConstants.EVENT_BRANCH_END);
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_EXPRESSION_START);
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_EXPRESSION_END);
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_OUTPUT_START);
    - eventTypes.add(ScoreLangConstants.EVENT_ASYNC_LOOP_OUTPUT_END);
    + eventTypes.add(ScoreLangConstants.EVENT_SPLIT_BRANCHES);
    + eventTypes.add(ScoreLangConstants.EVENT_JOIN_BRANCHES_START);
    + eventTypes.add(ScoreLangConstants.EVENT_JOIN_BRANCHES_END);
    eventTypes.add(ScoreLangConstants.EVENT_EXECUTION_FINISHED);
    return eventTypes;
}
```

Listing 8: Merged version of the `getAllEventTypes()` method from the `cloud-slang::SlangImpl` class. Added lines are highlighted in green, while deleted lines are marked in red.

```
public URL getUrl() throws MalformedURLException {
    FilteredURLStreamHandler handler = this.filtered ? new FilteredURLStreamHandler()
            : null;
    return new URL("file", "", -1, this.root.toURI().getPath(), handler);
}
```

Listing 9: Base version of the `getUrl()` method from the `spring-boot::ExplodedArchive` class.

```
public URL getUrl() throws MalformedURLException {
        FilteredURLStreamHandler handler = this.filtered ? new FilteredURLStreamHandler()
                        : null;
        return new URL("file", "", -1, this.root.toURI().toURL().getPath(), handler);
}
```

Listing 10: Right version of the `getUrl()` method from the `spring-boot::ExplodedArchive` class.

```
public URL getUrl() throws MalformedURLException {
    return new URL("file", "", -1, this.root.toURI().getPath());
}
```

Listing 11: Left version of the `getUrl()` method from the `spring-boot::ExplodedArchive` class.

```
public URL getUrl() throws MalformedURLException {
    return this.root.toURI().toURL();
}
```

Listing 12: Merged version of the `getUrl()` method from the `spring-boot::ExplodedArchive` class.