UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

Nathalia Fernanda de Araújo Barbosa

Detecção de Conflitos Semânticos com Testes de Unidade Gerados por LLM

Recife
2025

Nathalia Fernanda de Araújo Barbosa

Detecção de Conflitos Semânticos com Testes de Unidade Gerados por LLM

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Engenharia da Computação.

Orientador (a): Paulo Henrique Monteiro Borba

Coorientador (a): Léuson Mário Pedro da Silva

Recife

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Barbosa, Nathalia Fernanda de Araújo.
Detecção de Conflitos Semânticos com Testes de Unidade Gerados por LLM / Nathalia Fernanda de Araújo Barbosa. - Recife, 2025.
31 p. : il., tab.

Orientador(a): Paulo Henrique Monteiro Borba
Cooorientador(a): Léuson Mário Pedro da Silva
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2025.
Inclui referências, apêndices.

1. Engenharia de Software. 2. Testes de unidade. 3. Inteligência Artificial. 4. LLM. I. Borba, Paulo Henrique Monteiro. (Orientação). II. Silva, Léuson Mário Pedro da. (Coorientação). IV. Título.

000 CDD (22.ed.)

Nathalia Fernanda de Araújo Barbosa

Detecção de Conflitos Semânticos com Testes de Unidade Gerados por LLM

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Engenharia da Computação.

Aprovado em: 11/08/2025

BANCA EXAMINADORA

_____

Prof. Dr. Paulo Henrique Monteiro Borba (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Breno Alexandro Ferreira de Miranda (Examinador Interno)

Universidade Federal de Pernambuco

Detecção de Conflitos Semânticos com Testes de Unidade Gerados por LLM

Nathalia Barbosa[1], **Paulo Borba**[1], **Léuson Da Silva**[2]

[1] Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Recife – PE – Brasil

[2]Polytechnique Montreal
Montreal – Canadá

nfab@cin.ufpe.br, phmb@cin.ufpe.br, leuson-mario-pedro.da-silva@polymtl.ca

Abstract. *Semantic conflicts arise when a developer introduces changes to a codebase that unintentionally affect the behavior of changes integrated in parallel by other developers. Traditional merge tools are unable to detect such conflicts, so complementary tools like SMAT have been proposed. SMAT relies on generating and executing unit tests: if a test fails on the base version, passes on a developer's modified version, but fails again after merging with another developer's changes, a semantic conflict is indicated. While SMAT is effective at detecting conflicts, it suffers from a high rate of false negatives, partly due to the limitations of unit test generation tools such as Randoop and Evosuite. To investigate whether large language models (LLMs) can overcome these limitations, we propose and integrate a new test generation tool based on Code Llama 70B into SMAT. We explore the model's ability to generate tests using different interaction strategies, prompt contents, and parameter configurations. Our evaluation uses two samples: a benchmark with simpler systems from related work, and a more significant sample based on complex, real-world systems. We assess the effectiveness of the new SMAT extension in detecting conflicts. Results indicate that, although LLM-based test generation remains challenging and computationally expensive in complex scenarios, there is promising potential for improving semantic conflict detection.*

Resumo. *Conflitos semânticos ocorrem quando um desenvolvedor introduz mudanças em uma base de código que afetam, de maneira não intencional, o comportamento de mudanças integradas em paralelo por outros desenvolvedores. Como as ferramentas de* merge *usadas na prática não conseguem detectar esse tipo de conflito, foram propostas ferramentas complementares, como SMAT, que é baseada na geração e execução de testes de unidade — se um teste falha na versão base do código, passa na versão modificada por um desenvolvedor, mas falha novamente na versão resultante do* merge *dessa com a de outro desenvolvedor, sinaliza-se um conflito semântico. Apesar de apresentar boa capacidade de detecção de conflitos, SMAT apresenta alta taxa de falsos negativos (conflitos existentes mas não sinalizados pela mesma). Parte desse problema, deve-se às limitações naturais de ferramentas de geração de testes de unidade, no caso, Randoop e Evosuite. Para entender se essas limitações podem ser superadas por modelos de linguagem de grande porte (LLMs), este*

trabalho propõe, e integra ao SMAT, uma nova ferramenta de geração de testes baseada no Code Llama 70B. Exploramos então a capacidade desse modelo de gerar testes, com diferentes estratégias de interação, prompts *com diferentes conteúdos, e diferentes configurações de parâmetros do modelo. Avaliamos os resultados com duas amostras distintas, um* benchmark *com sistemas mais simples, usados em trabalhos relacionados, e uma amostra mais significativa baseada em sistemas complexos e utilizados na prática. Por fim, avaliamos a eficácia da nova extensão do SMAT na detecção de conflitos. Os resultados indicam que, embora a geração de testes por LLM em cenários complexos ainda represente um desafio e seja computacionalmente custosa, há potencial promissor na identificação de conflitos semânticos.*

Conflitos semânticos de código, Geração de testes de unidade, LLMs.

1. Introdução

Durante o desenvolvimento colaborativo de software, é comum que múltiplos desenvolvedores trabalhem simultaneamente em diferentes partes do código. Nesses cenários, conflitos podem surgir durante a integração (*merge*) das alterações [Zhang et al. 2022], especialmente quando dois desenvolvedores modificam a mesma linha de código ou linhas consecutivas de um arquivo de forma concorrente [Maddila et al. 2021]. Esses são os chamados conflitos textuais, que exigem intervenção manual para serem resolvidos. Normalmente, é necessário revisar as alterações de ambos os desenvolvedores e decidir qual versão deve ser mantida, ou se uma combinação das duas é necessária.

Embora eficazes em detectar conflitos textuais, as ferramentas de integração atuais não identificam todos os tipos de conflitos. Uma categoria mais sutil, os conflitos semânticos, ocorre quando alterações paralelas afetam o comportamento do sistema de forma não intencional, resultando em falhas de testes ou falhas em tempo de execução, mesmo que o código seja mesclado e compilado com sucesso. Esses conflitos, também chamados de *conflitos semânticos comportamentais* [Da Silva et al. 2024], diferem de conflitos de *build* [Da Silva et al. 2024, Brun et al. 2011, Sarma et al. 2012], que incluem erros de compilação, ou conflitos sintáticos e semânticos estáticos. Por sua natureza, conflitos semânticos comportamentais podem permanecer ocultos por várias versões, tornando sua detecção e resolução um desafio.

Para detectar conflitos semânticos, ferramentas complementares foram propostas, como o SMAT [Silva et al. 2020], que utiliza geração automatizada de testes de unidade para identificar mudanças de comportamento introduzidas durante o *merge*. O SMAT aplica heurísticas sobre os resultados dos testes para sinalizar possíveis conflitos: por exemplo, se um teste falha na versão comum (vamos nos referir à essa versão no restante do trabalho como “Base”), passa em uma das versões modificadas (“Left” ou “Right”), mas volta a falhar após o *merge* (“Merge”), isso indica um possível conflito semântico. No entanto, apesar de sua eficácia na detecção de conflitos semânticos, o SMAT enfrenta limitações significativas, especialmente uma alta taxa de falsos negativos, ou seja, conflitos existentes que não são detectados [Da Silva et al. 2024]. Essas limitações estão ligadas à capacidade das ferramentas de geração de testes utilizadas pelo SMAT, como Randoop [Pacheco and Ernst 2007] e EvoSuite [Fraser and Arcuri 2011], que frequentemente não cobrem todos os cenários relevantes.

Diante dos avanços recentes em modelos de linguagem de grande porte (LLMs), surge a hipótese de que essas limitações podem ser superadas. Modelos como o Code Llama 70B [Meta AI 2023] têm mostrado grande potencial na geração de código e testes automatizados, sendo competitivo com ferramentas tradicionais em cobertura e utilidade dos testes. Estudos recentes [Schäfer et al. 2024, Pan et al. 2025, Zhang et al. 2025] demonstraram que LLMs podem gerar testes com alta cobertura e originalidade. Em particular, o ASTER [Pan et al. 2025] mostrou-se competitivo com ferramentas de ponta (incluindo EvoSuite) em termos de cobertura, além de produzir testes mais "naturais" e preferidos por desenvolvedores. Tais avanços sugerem que LLMs podem oferecer novas estratégias para a geração de testes mais eficazes na detecção de falhas sutis. Dessa forma, este trabalho propõe e integra ao SMAT uma nova ferramenta de geração de testes que tem como centro o LLM Code Llama 70B, explorando diferentes estratégias de *prompt engineering* e configurações do modelo para detectar conflitos semânticos.

Para avaliar o potencial dessa ferramenta, realizamos um estudo empírico com um dataset de cenários de *merge*, alguns deles contendo conflitos semânticos. O estudo avalia o impacto na geração de testes e na detecção de conflitos semânticos de diferentes estratégias de *prompt* (*zero-shot* e *1-shot*), variações de temperatura, *seed* e combinações de contexto fornecido ao modelo. Os resultados mostram que a configuração *zero-shot* com temperatura 0.0 apresentou o melhor desempenho no *dataset* avaliado, identificando o maior número de conflitos em uma única execução, incluindo um conflito inédito não detectado em trabalhos prévios [Da Silva et al. 2024, Silva et al. 2020] com o mesmo *dataset*. Por outro lado, a configuração *1-shot* com temperatura 0.0 teve a melhor taxa de compilação de testes, indicando que a presença de um exemplo no *prompt* pode auxiliar na geração de testes mais sintaticamente corretos. Além disso, para avaliar o impacto da complexidade dos *datasets* na geração e compilação dos testes, um segundo *dataset* com sistemas mais simples foi selecionado do trabalho de [Pan et al. 2025]. Constatou-se que a taxa de compilação dos testes gerados varia significativamente entre os *datasets*, mostrando como a complexidade do código afeta a eficácia da geração de testes por LLMs.

Assim, as principais contribuições deste trabalho são:

- Análise do efeito de diferentes *prompts*, variações de configuração e contextos fornecidos a um modelo de linguagem para geração de casos de teste visando detectar conflitos semânticos.
- Evidência de que LLMs podem gerar testes capazes de detectar conflitos semânticos em cenários de *merge*, contribuindo para ampliar o potencial do SMAT em detectar conflitos.
- Integração estrutural de uma ferramenta baseada em um modelo de linguagem à arquitetura do SMAT, facilitando a adição de outros modelos.
- Integração de uma ferramenta que pode ser utilizada, com pequenas adaptações, para gerar testes em qualquer linguagem de programação, ampliando o escopo de uso do SMAT.

O restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta a motivação do trabalho e revisa o funcionamento do SMAT. A Seção 3 detalha a metodologia adotada, incluindo a integração da ferramenta baseada no Code Llama 70B ao SMAT e os procedimentos experimentais. Na Seção 4, avaliamos o desempenho do modelo em comparação com outras ferramentas e discutimos os resultados. A Seção 5 discute as ameaças à vali-

dade do estudo, abordando possíveis limitações e vieses. A Seção 6 aborda os trabalhos relacionados e, por fim, a Seção 7 apresenta as conclusões e direções futuras.

2. Motivação

Conflitos semânticos geralmente passam despercebidos por ferramentas tradicionais de controle de versão, que focam apenas em diferenças textuais e ignoram interações comportamentais entre as mudanças integradas. Para exemplificar o que são conflitos semânticos, considere o seguinte exemplo, baseado na motivação do trabalho de [Silva et al. 2020]: dois desenvolvedores, Nivan e Renato, trabalham simultaneamente em um projeto, cada um fazendo alterações em um mesmo método de uma classe, conforme mostrado na Listagem 1.

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

Listagem 1. Método alterado pelos desenvolvedores

Com o objetivo de tornar o texto mais legível, ambos fazem contribuições ao método `cleanText()` da classe `Text`. Nivan implementa e adiciona uma chamada ao método `normalizeWhiteSpace()`, que substitui múltiplos espaços em branco por um único espaço. Vamos chamar a versão do código com essa alteração de Left. Renato, por sua vez, implementa e adiciona uma chamada ao método `removeDuplicateWords()`, que remove palavras duplicadas consecutivas. Vamos chamar essa outra versão do código com tal alteração de Right. A versão integrada do código, que combina as alterações de ambos, é chamada de Merge, e esta é a versão ilustrada na Listagem 1. Ambos testam suas funções individualmente e elas funcionam como esperado, no entanto, quando o código é integrado, um conflito semântico surge: a ordem de execução dos métodos `normalizeWhiteSpace()` e `removeDuplicateWords()` afeta o resultado final do texto processado. Observe, por exemplo, o comportamento da função `cleanText()` quando aplicada a uma entrada como `HELLO␣␣HELLO␣␣WORLD`. Se a versão Left for executada, ou seja, se apenas `normalizeWhiteSpace()` for aplicada, o resultado será `HELLO␣HELLO␣WORLD`, mantendo duas ocorrências de `HELLO` e um espaço entre as palavras. Por outro lado, se a versão Right for executada, isto é, se apenas `removeDuplicateWords()` for aplicada, o resultado será `HELLO␣␣␣␣WORLD`, com apenas uma ocorrência de `HELLO` e quatro espaços entre as palavras. Quando ambas as funções são aplicadas em sequência (versão Merge), o efeito combinado revela um comportamento inesperado: `HELLO␣␣WORLD`, ou seja, uma das ocorrências de `HELLO` é removida, mas os espaços entre as palavras permanecem, e eles só passam a constituir uma duplicação após a chamada do último método. Esse é um exemplo clássico de conflito semântico, em que a ordem de aplicação das funções afeta o resultado final e o comportamento emergente não é trivialmente previsível.

A interação entre essas mudanças só revela seu efeito colateral quando um teste é executado com uma entrada específica (contendo tanto palavras quanto espaços duplicados), e tal entrada pode não estar presente na suíte de testes original, uma vez que os

desenvolvedores tendem a criar casos de teste focados nos cenários mais comuns de uso e que validam as funcionalidades de forma isolada. Assim, esse tipo de problema é difícil de identificar com inspeção manual, reforçando a necessidade de abordagens automatizadas baseadas em testes para detectar tais conflitos.

2.1. SMAT

Nesta subseção, apresentamos o SMAT [Silva et al. 2020, Da Silva et al. 2024], uma ferramenta desenvolvida para detectar conflitos semânticos em cenários de integração de código. Inicialmente, discutimos seu funcionamento, destacando como a abordagem baseada em geração e execução automatizada de testes permite identificar interações inesperadas entre mudanças paralelas. Em seguida, abordamos as limitações observadas nas versões atuais do SMAT, especialmente relacionadas à cobertura dos testes gerados por ferramentas tradicionais. Por fim, introduzimos a proposta de aprimorar o SMAT por meio da integração de uma ferramenta baseada em modelos de linguagem, como o Code Llama 70B [Meta AI 2023], explorando o potencial dessas tecnologias para aumentar a eficácia na detecção de conflitos semânticos.

Como a identificação manual de conflitos semânticos é uma tarefa complexa, o SMAT foi desenvolvido para identificá-los utilizando a geração automatizada de testes de unidade em cenários de *merge*. A ferramenta recebe como entrada um conjunto de cenários de *merge*, sendo que cada cenário inclui informações como o nome do projeto, uma quádrupla de *commits* — representando as versões Base (comum a todos), Left (modificação de um desenvolvedor), Right (modificação de outro desenvolvedor) e Merge (resultado da integração entre Left e Right) — além dos caminhos dos arquivos de código, das classes-alvo e dos elementos que foram modificados simultaneamente por Left e Right.

A arquitetura do SMAT é composta pelos módulos *Test Generation*, *Test Execution*, *Test Dynamic Analysis* e *Output Generation*. O funcionamento desses módulos se resume em três etapas principais:

- **Geração de testes**: O objetivo desta etapa é gerar as suítes de testes. Para isso, as ferramentas EvoSuite, Differential EvoSuite, Randoop e Randoop Clean são empregadas para criar automaticamente testes para as versões Left e Right do código, cada uma contendo modificações em uma mesma classe e elemento. Vale destacar que a arquitetura do SMAT é projetada para ser extensível, facilitando a integração de novas ferramentas de geração de testes sempre que necessário.
- **Compilação e execução**: Esta etapa visa verificar o comportamento das versões do código frente aos testes gerados, buscando identificar diferenças de execução. Assim, as suítes de testes criadas são compiladas e executadas em todas as quatro versões do código (Base, Left, Right e Merge), sendo o processo repetido múltiplas vezes para garantir maior confiabilidade dos resultados. Resultados considerados inválidos são descartados para evitar análises equivocadas.
- **Relatórios e heurísticas**: Por fim, esta etapa tem como propósito interpretar os resultados dos testes para detectar possíveis conflitos semânticos entre as versões. Para isso, heurísticas específicas são aplicadas sobre os resultados obtidos, identificando padrões que caracterizam conflitos. Por exemplo, considera-se um conflito quando um teste falha em Base e Merge, mas passa em Left ou Right (ou vice-versa); ou ainda, quando apenas Merge apresenta falha ou sucesso.

Os experimentos iniciais de [Silva et al. 2020] mostram que a abordagem baseada em testes de unidade pode ser eficaz: a heurística do SMAT identificou corretamente 4 conflitos em 15 casos, sem falsos positivos, mas ainda com um número significativo de falsos negativos. Em trabalhos posteriores [Da Silva et al. 2024], a análise foi ampliada para 85 cenários e quatro ferramentas de geração de testes, resultando na detecção de 9 conflitos entre 29, reforçando os achados anteriores, mas com três falsos positivos e uma taxa de falsos negativos considerável.

Apesar dos avanços, ainda existem limitações, sobretudo relacionadas à cobertura dos testes automatizados, que podem não identificar determinadas interações sutis entre as versões Left e Right do código, levando aos falsos negativos mencionados. Ferramentas tradicionais frequentemente não capturam essas nuances, tornando a integração de LLMs uma oportunidade de explorar novas estratégias de geração de testes e descobrir quais *prompts* e abordagens são mais eficazes para potencializar a detecção de conflitos semânticos.

Dessa forma, o SMAT pode se beneficiar diretamente dos avanços recentes em LLMs. No momento em que esta pesquisa foi conduzida, em 2024, os modelos da família Code Llama haviam sido lançados recentemente, e já se destacavam em tarefas de geração e compreensão de código em relação a outros modelos *open-source* disponíveis, especialmente no suporte a linguagens populares como Java [Meta AI 2023], foco da arquitetura do SMAT. A escolha do Code Llama 70B para a integração da ferramenta no SMAT foi, assim, fundamentada em três critérios principais: sua especialização em código, sendo treinado para tarefas de programação e compreensão de linguagens como Java; seu tamanho de 70 bilhões de parâmetros, que oferece um equilíbrio entre capacidade de raciocínio complexo e viabilidade computacional para execução local; e sua disponibilidade como modelo *open-source*, permitindo reprodutibilidade dos experimentos e controle total sobre o ambiente de execução, aspectos essenciais para pesquisa acadêmica. A incorporação da ferramenta ao SMAT permitiu avaliar, comparar e aprimorar métodos de geração de testes e detecção de conflitos semânticos. Outros modelos não foram utilizados para fins de comparação devido a limitações de tempo e recursos computacionais, reforçando o foco na avaliação do Code Llama 70B neste contexto.

3. Metodologia

Para investigar se LLMs podem ser úteis para geração de testes de unidade com o objetivo de detectar conflitos semânticos, conduzimos uma série de experimentos utilizando duas amostras (*datasets*) distintas, cada uma com motivações e características específicas. O primeiro *dataset* contém cenários de *merge* de projetos significativos, alguns deles contendo conflitos semânticos, permitindo avaliar a capacidade da ferramenta proposta em detectar esse tipo de problema. Já o segundo *dataset* não contém cenários de *merge* nem conflitos semânticos, e é composto por versões finais de projetos mais simples, sendo utilizado para avaliar a capacidade de geração e compilação de testes pela ferramenta. Para viabilizar esses experimentos, desenvolvemos e integramos ao SMAT um novo módulo de geração de testes automatizados, cujo funcionamento é baseado no modelo Code Llama 70B. A seguir, o processo de integração do módulo ao SMAT é apresentado, bem como os *datasets* selecionados para o estudo e informações adicionais sobre os procedimentos adotados durante os experimentos.

3.1. Integração da Ferramenta Baseada no Code Llama 70B

Como mencionado na Seção 2, a arquitetura do SMAT é modular, permitindo a adição de novas ferramentas e funcionalidades conforme necessário. Neste contexto, desenvolvemos uma ferramenta completa de geração de testes automatizados, que utiliza o modelo Code Llama 70B como parte central, mas incorpora diversas etapas adicionais de processamento, integração e validação. A nova ferramenta expande o *Test Generation* já existente, agregando um *pipeline* que realiza desde a extração e preparação das informações do código-fonte, passando pela construção e envio de *prompts* ao modelo, até o tratamento, validação e integração dos testes gerados ao fluxo do SMAT.

Figura 1. Arquitetura da ferramenta baseada no Code Llama 70B integrada ao SMAT.
Diagrama mostrando a integração entre a ferramenta proposta e o SMAT, incluindo etapas como extração de informações, construção de *prompts* e invocação do modelo.

Visão Geral da Integração A integração da nova ferramenta ao SMAT é implementada através de um *pipeline* que utiliza informações do código em análise para gerar testes de unidade executáveis. O fluxo inicia com extração de elementos relevantes do código-fonte, que são então formatados em *prompts* estruturados para serem enviados ao LLM. A resposta do modelo contendo o código dos testes passa por um processo de limpeza e validação antes de ser compilado e executado pelos demais componentes do SMAT. Este processo é organizado em etapas sequenciais, conforme detalhado a seguir:

1. Além dos elementos citados na Seção 2.1, a entrada do SMAT foi modificada para incluir, junto a cada elemento alterado simultaneamente por Left e Right na classe-alvo, um resumo textual das mudanças realizadas por cada branch. No caso do *mergedataset*, esses resumos já estavam disponíveis e foram elaborados manualmente por humanos, não por LLMs. Assim, para aplicar essa abordagem em outros datasets, será necessário produzir manualmente os resumos das mudanças. Essa alteração foi feita para avaliar se a inclusão dessa informação auxilia o LLM na geração de testes capazes de detectar conflitos. Conforme ilustrado na Figura 1, o passo (1) mostra as informações de entrada que são inicialmente usadas para a geração dos *prompts*: as classes-alvo (`Class Text`) e os seus respectivos métodos-alvo, que são os elementos alterados por Left e Right os quais serão testados (`cleanText()`).
2. Como destacado no passo (2), para cada um dos métodos-alvo informados na entrada, utilizamos a biblioteca de parsing *Tree-sitter* [Tree-sitter 2024] para construir a AST (*Abstract Syntax Tree*) do código-fonte, o que permite estruturar o código de forma a coletar informações para a geração dos *prompts*. A partir dessa

AST, extraímos atributos da classe, construtores e o corpo do método, além das declarações de `import`, que são anexadas aos testes no fim da iteração para garantir sua compilação. Devido à natureza do *Tree-sitter*, com poucas modificações, essa biblioteca pode ser utilizada para gerar ASTs de qualquer linguagem de programação, o que torna a ferramenta proposta adaptável a outros contextos além do Java.

Construção dos prompts As informações extraídas são organizadas em diferentes formatos de *prompts*, compostos por mensagens dos tipos `system`, `user` e `assistant`, cada uma com uma função específica na interação com o modelo de linguagem. Segundo [White et al. 2023], os *prompts* podem ser estruturados por meio de *prompt patterns*, que funcionam como padrões reutilizáveis para resolver problemas recorrentes no uso de LLMs. A mensagem do tipo `system` corresponde ao padrão *Persona*, atribuindo ao modelo uma identidade ou papel específico, com o objetivo de orientar seu comportamento ao longo da conversa. Essa mensagem estabelece o contexto geral da interação e define diretrizes para o estilo e o conteúdo das respostas geradas. Abaixo está a mensagem do tipo `system` utilizada neste estudo:

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`.

As mensagens do tipo `user` fornecem a entrada principal ao modelo, podendo ser relacionadas aos padrões *Recipe* e *Context Manager*. Nessas mensagens estão descritas informações do código, o método-alvo a ser testado e instruções para o modelo, delimitando o escopo da tarefa e os requisitos desejados. A seguir, um exemplo de mensagem do tipo `user` que nosso módulo usa para gerar testes para o método `cleanText` da classe `Text`:

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`.

Nessa mensagem, que compõe um *prompt* do tipo *zero-shot*, o contexto do método-alvo é fornecido, incluindo os atributos da classe, construtores e o corpo do método. Então, o modelo é instruído a gerar testes JUnit para esse método, considerando as informações fornecidas.

Já as mensagens do tipo `assistant` são utilizadas aqui apenas na configuração *1-shot*. Esse uso está alinhado ao padrão *Template*, cujo objetivo é fornecer um exemplo que sirva de modelo para a geração de saídas consistentes e adequadas à tarefa proposta.

Os *prompts* utilizados foram construídos a partir de conjuntos de mensagens contendo diferentes partes do contexto dos cenários, como previamente introduzido:

(i) Resumo das mudanças introduzidas por Left e Right: um resumo textual das alterações realizadas em cada *branch*. Por exemplo, no caso do método `cleanText()`, o resumo apresentado ao modelo foi:
> Left adicionou uma chamada ao método `normalizeWhiteSpace()`, que substitui múltiplos espaços em branco por um único espaço.
> Right adicionou uma chamada ao método `removeDuplicateWords()`, que remove palavras duplicadas consecutivas.

(ii) Atributos da classe;
(iii) Construtores; e
(iv) Corpo do método-alvo.

Para cada formato de *prompt* (*zero-shot* e *1-shot*), conforme detalhado na Tabela 4, foram criadas 8 variações de *prompt*. Cada variação combina diferentes subconjuntos dessas partes do contexto, de forma a explorar quais informações são mais relevantes para a geração de testes eficazes, introduzindo maior variedade nas interações com o modelo e, consequentemente, aumentando a diversidade dos testes gerados. Abaixo está a estrutura geral dos *prompts* utilizados:

- **Zero-shot:** O *prompt* é composto por uma mensagem de sistema (`system`), seguida pelas mensagens do usuário (`user`) que fornecem o contexto do método-alvo a ser testado.
- **1-shot:** Neste formato, o *prompt* primeiro apresenta um exemplo completo de interação antes da solicitação real. A estrutura contém a mensagem de sistema (`system`), seguida por um par de mensagens de exemplo (uma do `user` com um método fictício e uma resposta correspondente do `assistant` com o teste gerado). Somente após esse exemplo, é apresentada a mensagem do `user` com o método-alvo real.

Todos os *prompts* usados para cada projeto testado nos experimentos estão disponíveis no apêndice online deste trabalho [Barbosa 2025a].

Invocação do modelo Esses *prompts* são enviados, por meio da API REST do *Ollama* [Ollama 2023] — plataforma *open-source* para execução local de LLMs —, ao modelo Code Llama 70B, que retorna múltiplas respostas (em quantidade configurável) para cada método testado. Para os experimentos realizados, apenas uma resposta foi solicitada para cada *prompt*, por método-alvo em cada *branch* (Left e Right).

Figura 2. Processamento da saída da ferramenta baseada no Code Llama 70B. Diagrama ilustrando as etapas de processamento da saída gerada pela ferramenta proposta.

Processamento da saída A Figura 2 ilustra o processamento da saída da ferramenta baseada no Code Llama 70B, que ocorre após a invocação do modelo. Conforme mostrado no passo (3), as respostas do LLM são consideradas aqui como potenciais suítes de testes. Cada uma delas passa por uma etapa de limpeza utilizando expressões regulares, com o objetivo de remover trechos que não correspondem a código Java válido. Isso inclui linguagem natural, comentários irrelevantes, artefatos de formatação e caracteres que não pertencem ao léxico da linguagem, os quais poderiam comprometer a compilação e execução dos testes.

Após essa limpeza, utilizamos novamente o *Tree-sitter*, destacado no passo (4), para analisar a saída resultante e extrair os métodos de teste por meio da identificação de métodos anotados com a anotação `@Test`. Isso é um primeiro passo para garantir que os testes gerados sejam válidos, pois essa anotação é essencial para que o JUnit reconheça os métodos como testes a serem executados.

Então, cada teste é armazenado em um arquivo único, identificado por um nome que concatena a classe-alvo, a *branch*, o *prompt*, o índice do método e um número sequencial, como em `TextTest_left_prompt1_0_0.java`. O objetivo disso é maximizar o aproveitamento dos testes gerados: caso um ou mais testes apresentem problemas de compilação, os demais ainda poderão ser executados sem prejuízo ao processo de avaliação. Essa abordagem também facilita a organização dos experimentos e a análise posterior dos resultados, permitindo isolar falhas e identificar padrões nos erros gerados.

3.2. Datasets Utilizados

A priori, utilizamos como base o *mergedataset*, conforme descrito por [Da Silva et al. 2024], que originalmente contém 85 cenários. Contudo, nosso conjunto de dados final foi reduzido para 79 cenários, pois 6 deles foram descartados por ausência de arquivos no repositório, conforme detalhado na Tabela 1. Esse subconjunto do *mergedataset* é composto por 29 projetos Java de código aberto, abrangendo diversas áreas de software, e inclui 29 cenários com conflitos semânticos e 50 sem conflitos.

Projeto	Classe	Método
libgdx	Lwjgl3ApplicationConfiguration	copy
libgdx	Lwjgl3Application	newWindow
elasticsearch	RestNodesAction	getTableWithHeader
elasticsearch	RestIndicesAction	getTableWithHeader
elasticsearch	RestShardsAction	getTableWithHeader
ReactiveX	TestScheduler	triggerActions

Tabela 1. Cenários descartados do *mergedataset* por ausência de arquivos no repositório.

Para complementar a análise, selecionamos ainda um segundo *dataset*, composto por dois sistemas considerados "toy" devido à sua menor complexidade. Esses projetos, mais simples e didáticos, foram extraídos do *dataset* utilizado no trabalho de [Pan et al. 2025]. Por se tratar de um trabalho com foco apenas na geração de testes, esses projetos não contêm cenários de *merge* e, portanto, não apresentam conflitos semânticos. Vamos nos referir a esses projetos como *dataset ASTER*:

- **Eclipse Cargo Tracker**: uma aplicação *open-source* de rastreamento de cargas, construída com Jakarta EE e estruturada com base nos princípios de Domain-Driven Design. A aplicação simula fluxos comuns de sistemas corporativos, como monitoramento logístico, roteamento e controle de eventos [Eclipse Foundation 2020].
- **DayTrader 8**: um benchmark de sistema de negociação de ações desenvolvido com Java EE 8. A aplicação implementa funcionalidades típicas de sistemas financeiros, como login de usuários, consulta de contas e execução de ordens de compra e venda. Por sua natureza compacta, é amplamente utilizada como referência para testes funcionais e de desempenho em servidores de aplicação Java EE [OpenLiberty 2018].

Esses dois foram selecionados por simularem sistemas com comportamento próximo ao de aplicações reais, ainda que com menor complexidade estrutural e de código em comparação aos projetos do *mergedataset*. Os demais projetos presentes no *dataset* original de [Pan et al. 2025] foram descartados por dois motivos principais. Alguns deles (Commons CLI, Codec, Compress e JXPath), que são APIs da Apache Commons, foram considerados muito simples e possivelmente utilizados no treinamento do modelo, o que poderia enviesar os resultados; por sua vez, o PetClinic não foi incluído devido à incompatibilidade de versão do Java com o SMAT. Essa escolha permite avaliar a capacidade do modelo em gerar testes de unidade tanto em contextos mais controlados e didáticos quanto em cenários extraídos de sistemas reais.

3.3. Configuração dos Parâmetros Experimentais

A escolha dos parâmetros utilizados nos experimentos foi fundamental para avaliar o comportamento do modelo Code Llama 70B em diferentes cenários de geração de testes. Os parâmetros explorados foram a temperatura e a *seed*, mas também alternamos o formato do *prompt*.

A **temperatura** é um parâmetro que controla o grau de aleatoriedade das respostas do modelo. Valores mais baixos (como 0.0) tendem a produzir saídas mais determinísticas e conservadoras, enquanto valores mais altos (como 0.7) estimulam maior diversidade e criatividade nas respostas, podendo gerar testes mais variados.

O parâmetro ***seed*** define o estado inicial do gerador de números aleatórios do modelo, garantindo a reprodutibilidade dos experimentos. Ao utilizar diferentes *seeds*, é possível avaliar a estabilidade dos resultados e identificar possíveis variações decorrentes da natureza não determinística dos LLMs.

Por fim, o **formato do *prompt*** influencia diretamente a forma como o modelo interpreta a tarefa. Foram avaliadas duas abordagens: *zero-shot*, em que o modelo recebe apenas as instruções e o contexto do método-alvo, e *1-shot*, em que é fornecido um exemplo de teste antes da solicitação real.

3.4. Procedimentos Experimentais

Neste estudo, buscamos responder às seguintes questões de pesquisa:

- **RQ1:** Entre as configurações de *prompt* e parâmetros avaliadas, alguma delas permite ao modelo gerar testes mais eficazes para identificar conflitos semânticos?
- **RQ2:** O modelo Code Llama 70B é útil para a detecção automática de conflitos semânticos em cenários de *merge*?
- **RQ3:** Como a complexidade do código em análise impacta os resultados do LLM?

Para abordar essas questões, este estudo:

(i) investiga como diferentes configurações de *prompt* e parâmetros influenciam a geração de testes capazes de identificar conflitos semânticos (RQ1);
(ii) avalia a eficácia do modelo Code Llama 70B na detecção automática desses conflitos em cenários reais de merge (RQ2); e
(iii) analisa de que forma a complexidade do código impacta os resultados obtidos (RQ3).

As análises foram realizadas a partir dos relatórios gerados pelo SMAT, que incluem métricas como tempo de execução, quantidade de testes gerados, compilados e executados, além dos conflitos semânticos detectados. Adicionalmente, analisamos manualmente os testes gerados para verificar se os que apontaram conflitos semânticos realmente identificam conflitos existentes e se não se tratam de falsos positivos.

Conduzimos, então, um primeiro experimento focado na detecção de conflitos semânticos utilizando o *mergedataset*. Nessa etapa, executamos o *pipeline* experimental oito vezes, em diferentes configurações que variam o formato dos *prompts* e os parâmetros fundamentais do modelo, como citados na Seção 3.3: a **temperatura** e a ***seed***. Uma imagem ilustrando esse experimento está disponível na Figura 3. Adotamos as temperaturas 0.0 e 0.7, e as *seeds* 42 e 123. Também testamos duas abordagens de *prompt*: *zero-shot* e *1-shot*.

Essa combinação de parâmetros foi escolhida para explorar diferentes aspectos do comportamento do modelo. A temperatura 0.0 foi configurada para obter respostas mais determinísticas e focadas, reduzindo a criatividade do modelo em favor da consistência.

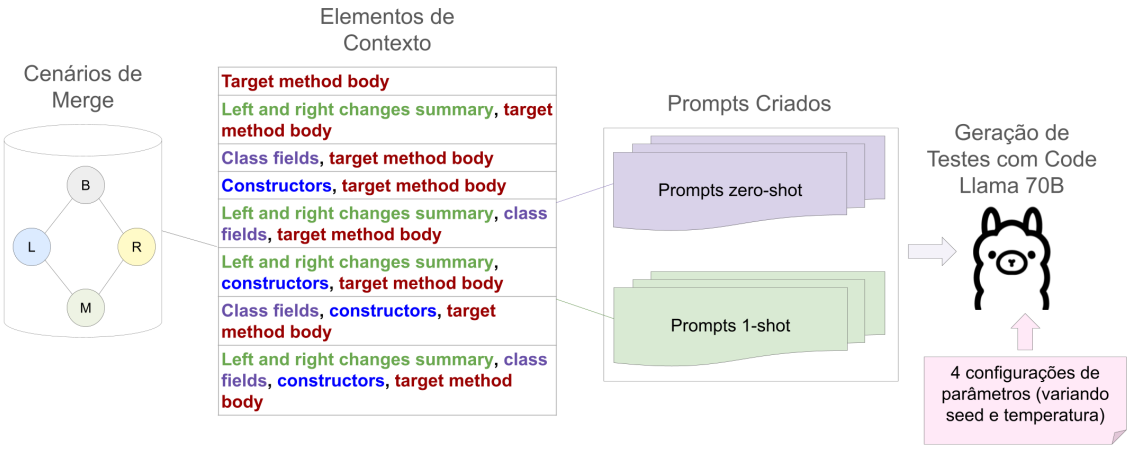

Figura 3. Fluxo experimental para avaliação da ferramenta baseada no Code Llama 70B na geração de testes.

Por outro lado, a temperatura 0.7 (valor padrão) foi utilizada para estimular a geração de testes mais diversos e potencialmente capazes de explorar diferentes tipos de conflitos semânticos. Os formatos de *prompt zero-shot* e *1-shot* foram testados para avaliar se o fornecimento de um exemplo auxilia o modelo na compreensão da tarefa e na geração de testes mais adequados em comparação com instruções sem exemplos. As *seeds* usadas foram escolhidas arbitrariamente, com o objetivo de garantir a reprodutibilidade dos experimentos e verificar a consistência dos resultados em diferentes execuções. Dessa forma, cada execução de uma configuração implica em 8 *prompts* diferentes chamados para cada método-alvo em cada *branch*, totalizando 1264 chamadas do modelo para os 79 cenários do *dataset*.

Em uma segunda fase, conduzimos outro experimento para avaliar a capacidade do Code Llama 70B de gerar e compilar testes automatizados em diferentes contextos, considerando a complexidade dos sistemas. Aqui, medimos o número de testes gerados e compilados com sucesso nos dois *datasets*, deixando de lado a detecção de conflitos semânticos, uma vez que o *ASTER dataset* não contém cenários de *merge*. Como configuração do experimento, fixamos a temperatura do modelo em 0.0 e o aplicamos cinco *prompts* distintos para cada método-alvo. Esses *prompts* foram divididos em duas abordagens, também usadas no experimento anterior: quatro variações *zero-shot*, as quais não contêm o resumo de mudanças por não ser relevante para o *ASTER dataset*, e um prompt *1-shot*, mantido para avaliar o impacto de um exemplo na geração de testes, mesmo em cenários sem conflitos semânticos. Os *prompts* utilizados tanto no experimento de detecção de conflitos semânticos quanto no experimento de geração e compilação de testes estão disponíveis no apêndice online deste trabalho [Barbosa 2025a].

Todos os experimentos foram conduzidos em uma máquina *x86_64* rodando Ubuntu, equipada com 251 GB de RAM, processador Intel Xeon Silver 4316 (40 núcleos) e duas GPUs NVIDIA A100 80GB PCIe.

4. Resultados

Para responder às perguntas de pesquisa discutidas na seção anterior, realizamos os experimentos descritos anteriormente e apresentamos os resultados a seguir. A seção está

estruturada em torno das perguntas de pesquisa, começando com uma análise quantitativa da geração de testes, seguida pela detecção de conflitos semânticos e comparação com outras ferramentas, e finalizando com uma análise do impacto do *dataset* e do tempo de execução.

4.1. Análise Quantitativa da Geração de Testes

Métrica	**Temperatura 0**		**Temperatura 0.7**	
	1S	**ZS**	**1S**	**ZS**
Média de testes gerados	10404	8592	1383	2358
Média de testes que compilaram	1253	586	124	121
Média de testes gerados por método	131.70	108.76	17.50	29.85
Média de testes que compilaram por método	15.86	7.42	1.57	1.53
Taxa de compilação média	12.04%	6.82%	8.97%	5.13%
Conflitos detectados	1*	3*	3*	1*

Tabela 2. Resultados da geração de testes com a ferramenta baseada no Code Llama 70B, considerando diferentes configurações de temperatura e modo de execução (1S = *1-shot*, ZS = *zero-shot*). Os valores são médias de duas execuções independentes (seeds 42 e 123). A métrica *conflitos semânticos detectados* inclui, em cada caso marcado com *, exatamente um conflito que não havia sido previamente identificado por ferramentas do SMAT.

A Tabela 2 apresenta os resultados da geração de testes e detecção de conflitos utilizando a ferramenta baseada no Code Llama 70B, considerando as diferentes configurações de temperatura e de formato de *prompt*. Para a construção da tabela, foram realizadas duas execuções independentes para cada configuração, cada uma com 8 *prompts*. Cada execução foi realizada com uma *seed* diferente, e então os resultados foram agregados para obter as médias apresentadas, enquanto a quantidade de conflitos detectados foi considerada como a união de conflitos identificados em ambas as execuções. Podemos observar que a temperatura 0.0 resultou em uma geração significativamente maior de testes, com uma média de 10404 testes na configuração *1-shot* (1S) e 8592 na configuração *zero-shot* (ZS). Em contraste, a temperatura 0.7 gerou uma quantidade consideravelmente menor de testes, com médias de 1383 e 2358 testes para as configurações 1S e ZS, respectivamente. Uma hipótese para esse comportamento é que temperaturas mais baixas tornam o modelo mais determinístico, favorecendo a repetição de padrões e, consequentemente, aumentando o volume de testes gerados. Por outro lado, temperaturas mais altas estimulam maior diversidade nas respostas, mas tendem a reduzir a quantidade total de exemplos produzidos, já que o modelo explora caminhos menos previsíveis e, muitas vezes, mais curtos. Esse efeito evidencia como a escolha da temperatura pode impactar diretamente tanto o volume quanto a variedade dos testes obtidos em cada configuração.

A taxa de compilação também foi superior com a temperatura 0.0, atingindo o pico de 12.04% na configuração 1S. Como a temperatura mais baixa favorece a repetição de padrões, é possível que os testes gerados sejam mais consistentes e, portanto, mais propensos a compilar com sucesso. Embora a configuração ZS tenha gerado um grande

volume de testes, sua taxa de compilação foi menor (6.82%), indicando que a ausência de exemplos no *prompt* pode ter dificultado a geração de testes sintaticamente corretos.

Com temperatura 0.7, as taxas de compilação foram de 8.97% (1S) e 5.13% (ZS). Notavelmente, todas as quatro configurações experimentais foram capazes de detectar conflitos semânticos, com cada uma identificando, pelo menos, o conflito previamente não detectado por outras ferramentas do SMAT. Em especial, a configuração *zero-shot* com temperatura 0.0 identificou três conflitos numa única execução, enquanto cada *seed* da configuração *1-shot* com temperatura 0.7 detectou dois, sendo um deles comum às duas *seeds*. Portanto, a configuração *zero-shot* com temperatura 0.0 se destacou por detectar o maior número de conflitos semânticos em uma única execução.

Esses resultados indicam que a escolha da temperatura impacta diretamente tanto a quantidade quanto a qualidade dos testes gerados. Temperaturas mais baixas favorecem maior volume e taxas de compilação superiores, enquanto temperaturas mais altas tendem a reduzir ambos os indicadores. Além disso, a estratégia de *prompt* também influencia a qualidade sintática dos testes, com o fornecimento de um exemplo (*1-shot*) levando a códigos mais compiláveis e, portanto, sintaticamente mais corretos.

4.2. RQ1: Alguma configuração permite gerar testes mais eficazes para identificar conflitos semânticos?

Conflito (Projeto::Classe::Elemento)	ZS (0, 42)	ZS (0, 123)	ZS (0.7, 42)	ZS (0.7, 123)	1S (0, 42)	1S (0, 123)	1S (0.7, 42)	1S (0.7, 123)
`antlr4::Python2Target::python2Keywords`	✓	✓	—	—	—	—	—	—
`antlr4::Python3Target::python3Keywords`	✓	✓	—	—	—	—	—	—
`cloud-slang::SlangImpl::getAllEventTypes()`	✓	✓	✓	✓	✓	✓	✓	✓
`spring-boot::AtomikosProperties::asProperties()`	—	—	—	—	—	—	✓	—
`spring-boot::AtomikosPropertiesTest::testProperties()`	—	—	—	—	—	—	—	✓

Tabela 3. Conflitos semânticos detectados, considerando diferentes configurações de temperatura, *seed* e formato de *prompt* (1S = *1-shot*, ZS = *zero-shot*). O par entre parênteses representa a temperatura e a *seed* utilizados — por exemplo, a configuração ZS (0, 42) refere-se a uma execução dos oito *prompts zero-shot* com temperatura 0.0 e *seed* 42. O símbolo ✓indica que o conflito foi detectado na configuração correspondente. Em negrito está destacado um conflito semântico que não havia sido previamente identificado por ferramentas do SMAT.

A Tabela 3 oferece uma visão detalhada da capacidade de detecção de conflitos, evidenciando a **complementaridade entre as configurações experimentais**. Nenhuma configuração isoladamente foi capaz de detectar todos os conflitos.

O conflito `getAllEventTypes()` foi o único identificado em todas as oito execuções experimentais, além de ser o único não previamente detectado por nenhuma das outras ferramentas integradas ao SMAT. Nesse cenário de *merge*, tanto a *branch* Left quanto a *branch* Right adicionaram elementos distintos à mesma lista, resultando em tamanhos finais diferentes para a coleção: a *branch* Left produz uma lista com 23 elementos, enquanto a *branch* Right gera apenas 21.

Uma possível explicação para o fato de ferramentas como o EvoSuite não terem detectado esse conflito envolve limitações impostas pelos parâmetros experimentais ado-

tados em trabalhos anteriores, como as *seeds* utilizadas, o tempo limite de execução (*timeout* de 300 segundos por cenário) e a função de *fitness* empregada para guiar a geração dos testes. Esses fatores podem restringir a exploração do espaço de busca ou dificultar a geração de *asserts* específicos para esse tipo de divergência.

O modelo Code Llama 70B, por outro lado, conseguiu identificar essa inconsistência ao gerar testes que verificam explicitamente o tamanho da lista retornada pelo método. A seguir, apresentamos as implementações do método `getAllEventTypes()` nas quatro versões: Base, Left, Right e Merge. As diferenças entre essas versões evidenciam o conflito semântico analisado:

```
1  private Set<String> getAllEventTypes() {
2      Set<String> eventTypes = new HashSet<>();
3      eventTypes.add(EventConstants.SCORE_FINISHED_EVENT);
4      eventTypes.add(EventConstants.SCORE_BRANCH_FAILURE_EVENT);
5      eventTypes.add(EventConstants.SCORE_FINISHED_BRANCH_EVENT);
6      eventTypes.add(EventConstants.SCORE_NO_WORKER_FAILURE_EVENT)
           ;
7      eventTypes.add(EventConstants.SCORE_PAUSED_EVENT);
8      eventTypes.add(EventConstants.SCORE_ERROR_EVENT);
9      eventTypes.add(EventConstants.SCORE_FAILURE_EVENT);
10     eventTypes.add(ScoreLangConstants.SLANG_EXECUTION_EXCEPTION)
           ;
11     eventTypes.add(ScoreLangConstants.EVENT_ACTION_START);
12     eventTypes.add(ScoreLangConstants.EVENT_ACTION_END);
13     eventTypes.add(ScoreLangConstants.EVENT_ACTION_ERROR);
14     eventTypes.add(ScoreLangConstants.EVENT_INPUT_START);
15     eventTypes.add(ScoreLangConstants.EVENT_INPUT_END);
16     eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_START);
17     eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_END);
18     eventTypes.add(ScoreLangConstants.EVENT_BRANCH_START);
19     eventTypes.add(ScoreLangConstants.EVENT_BRANCH_END);
20     eventTypes.add(ScoreLangConstants.
           EVENT_ASYNC_LOOP_EXPRESSION_START);
21     eventTypes.add(ScoreLangConstants.
           EVENT_ASYNC_LOOP_EXPRESSION_END);
22     eventTypes.add(ScoreLangConstants.
           EVENT_ASYNC_LOOP_OUTPUT_START);
23     eventTypes.add(ScoreLangConstants.
           EVENT_ASYNC_LOOP_OUTPUT_END);
24     eventTypes.add(ScoreLangConstants.EVENT_EXECUTION_FINISHED);
25     return eventTypes;
26 }
```

Listagem 2. Implementação do método getAllEventTypes() na versão Base.

```
1  private Set<String> getAllEventTypes() {
2      Set<String> eventTypes = new HashSet<>();
3      eventTypes.add(EventConstants.SCORE_FINISHED_EVENT);
4      eventTypes.add(EventConstants.SCORE_BRANCH_FAILURE_EVENT);
5      eventTypes.add(EventConstants.SCORE_FINISHED_BRANCH_EVENT);
```

```
6       eventTypes.add(EventConstants.SCORE_NO_WORKER_FAILURE_EVENT)
            ;
7       eventTypes.add(EventConstants.SCORE_PAUSED_EVENT);
8       eventTypes.add(EventConstants.SCORE_ERROR_EVENT);
9       eventTypes.add(EventConstants.SCORE_FAILURE_EVENT);
10      eventTypes.add(ScoreLangConstants.SLANG_EXECUTION_EXCEPTION)
            ;
11      eventTypes.add(ScoreLangConstants.EVENT_ACTION_START);
12      eventTypes.add(ScoreLangConstants.EVENT_ACTION_END);
13      eventTypes.add(ScoreLangConstants.EVENT_ACTION_ERROR);
14      eventTypes.add(ScoreLangConstants.EVENT_TASK_START);
15      eventTypes.add(ScoreLangConstants.EVENT_INPUT_START);
16      eventTypes.add(ScoreLangConstants.EVENT_INPUT_END);
17      eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_START);
18      eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_END);
19      eventTypes.add(ScoreLangConstants.EVENT_BRANCH_START);
20      eventTypes.add(ScoreLangConstants.EVENT_BRANCH_END);
21      eventTypes.add(ScoreLangConstants.
            EVENT_ASYNC_LOOP_EXPRESSION_START);
22      eventTypes.add(ScoreLangConstants.
            EVENT_ASYNC_LOOP_EXPRESSION_END);
23      eventTypes.add(ScoreLangConstants.
            EVENT_ASYNC_LOOP_OUTPUT_START);
24      eventTypes.add(ScoreLangConstants.
            EVENT_ASYNC_LOOP_OUTPUT_END);
25      eventTypes.add(ScoreLangConstants.EVENT_EXECUTION_FINISHED);
26      return eventTypes;
27  }
```

Listagem 3. Implementação do método getAllEventTypes() na versão Left.

```
1   private Set<String> getAllEventTypes() {
2       Set<String> eventTypes = new HashSet<>();
3       eventTypes.add(EventConstants.SCORE_FINISHED_EVENT);
4       eventTypes.add(EventConstants.SCORE_BRANCH_FAILURE_EVENT);
5       eventTypes.add(EventConstants.SCORE_FINISHED_BRANCH_EVENT);
6       eventTypes.add(EventConstants.SCORE_NO_WORKER_FAILURE_EVENT)
            ;
7       eventTypes.add(EventConstants.SCORE_PAUSED_EVENT);
8       eventTypes.add(EventConstants.SCORE_ERROR_EVENT);
9       eventTypes.add(EventConstants.SCORE_FAILURE_EVENT);
10      eventTypes.add(ScoreLangConstants.SLANG_EXECUTION_EXCEPTION)
            ;
11      eventTypes.add(ScoreLangConstants.EVENT_ACTION_START);
12      eventTypes.add(ScoreLangConstants.EVENT_ACTION_END);
13      eventTypes.add(ScoreLangConstants.EVENT_ACTION_ERROR);
14      eventTypes.add(ScoreLangConstants.EVENT_INPUT_START);
15      eventTypes.add(ScoreLangConstants.EVENT_INPUT_END);
16      eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_START);
17      eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_END);
```

```
18      eventTypes.add(ScoreLangConstants.EVENT_BRANCH_START);
19      eventTypes.add(ScoreLangConstants.EVENT_BRANCH_END);
20      eventTypes.add(ScoreLangConstants.EVENT_SPLIT_BRANCHES);
21      eventTypes.add(ScoreLangConstants.EVENT_JOIN_BRANCHES_START)
            ;
22      eventTypes.add(ScoreLangConstants.EVENT_JOIN_BRANCHES_END);
23      eventTypes.add(ScoreLangConstants.EVENT_EXECUTION_FINISHED);
24      return eventTypes;
25  }
```

Listagem 4. Implementação do método getAllEventTypes() na versão Right.

```
1   private Set<String> getAllEventTypes() {
2       Set<String> eventTypes = new HashSet<>();
3       eventTypes.add(EventConstants.SCORE_FINISHED_EVENT);
4       eventTypes.add(EventConstants.SCORE_BRANCH_FAILURE_EVENT);
5       eventTypes.add(EventConstants.SCORE_FINISHED_BRANCH_EVENT);
6       eventTypes.add(EventConstants.SCORE_NO_WORKER_FAILURE_EVENT)
            ;
7       eventTypes.add(EventConstants.SCORE_PAUSED_EVENT);
8       eventTypes.add(EventConstants.SCORE_ERROR_EVENT);
9       eventTypes.add(EventConstants.SCORE_FAILURE_EVENT);
10      eventTypes.add(ScoreLangConstants.SLANG_EXECUTION_EXCEPTION)
            ;
11      eventTypes.add(ScoreLangConstants.EVENT_ACTION_START);
12      eventTypes.add(ScoreLangConstants.EVENT_ACTION_END);
13      eventTypes.add(ScoreLangConstants.EVENT_ACTION_ERROR);
14      eventTypes.add(ScoreLangConstants.EVENT_TASK_START);
15      eventTypes.add(ScoreLangConstants.EVENT_INPUT_START);
16      eventTypes.add(ScoreLangConstants.EVENT_INPUT_END);
17      eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_START);
18      eventTypes.add(ScoreLangConstants.EVENT_OUTPUT_END);
19      eventTypes.add(ScoreLangConstants.EVENT_BRANCH_START);
20      eventTypes.add(ScoreLangConstants.EVENT_BRANCH_END);
21      eventTypes.add(ScoreLangConstants.EVENT_SPLIT_BRANCHES);
22      eventTypes.add(ScoreLangConstants.EVENT_JOIN_BRANCHES_START)
            ;
23      eventTypes.add(ScoreLangConstants.EVENT_JOIN_BRANCHES_END);
24      eventTypes.add(ScoreLangConstants.EVENT_EXECUTION_FINISHED);
25      return eventTypes;
26  }
```

Listagem 5. Implementação do método getAllEventTypes() na versão Merge.

A Listagem 6 apresenta um exemplo representativo dos testes gerados que conseguiram detectar este conflito. O teste assume a expectativa da *branch* Left (23 elementos) e, por essa razão, falha quando executado no contexto da *branch* Right, revelando assim o conflito semântico.

```
1   @Test
2   public void test00() {
3       SlangImpl slang = new SlangImpl();
```

```
4       Set<String> eventTypes = slang.getAllEventTypes();
5       assertEquals(23, eventTypes.size());
6   }
```

Listagem 6. Exemplo de teste gerado pelo modelo Code Llama 70B para detectar o conflito `getAllEventTypes()`.

Os dois conflitos relacionados ao projeto `antlr4` foram observados apenas nas execuções *zero-shot* com temperatura 0.0. Já o conflito `asProperties()` ocorreu exclusivamente na configuração *1-shot* com temperatura 0.7 e *seed* 42, enquanto `testProperties()` foi detectado apenas com temperatura 0.7 e *seed* 123.

Observamos uma clara especialização:

- As execuções em ***zero-shot* com temperatura 0.0** foram as únicas capazes de identificar os dois conflitos no projeto `antlr4`.
- Por outro lado, os conflitos no projeto `spring-boot` foram detectados exclusivamente pelas execuções em ***1-shot* com temperatura 0.7**.

Este resultado sugere que a variação na temperatura e na estratégia de *prompt* permite explorar diferentes tipos de falhas, indicando que uma abordagem combinando múltiplas configurações é mais eficaz para maximizar a detecção de conflitos. No entanto, é importante destacar que essa estratégia é computacionalmente mais custosa. Por exemplo, os tempos de geração de testes para cada configuração variaram de cerca de 6 horas (*1-shot*, temperatura 0.7) até quase 28 horas (*1-shot*, temperatura 0) para o *merge-dataset* inteiro. Sendo assim, usar mais de uma configuração para detectar conflitos pode ser inviável em cenários onde o tempo é um fator crítico. Uma descrição detalhada dos tempos de execução pode ser consultada na Seção 4.5.

Prompt	Contexto	Conflitos Detectados
`prompt1`	Target method body	1*
`prompt2`	Left and Right changes summary, target method body	2*
`prompt3`	Class fields, target method body	1*
`prompt4`	Constructors, target method body	1*
`prompt5`	Left and Right changes summary, class fields, target method body	2
`prompt6`	Left and Right changes summary, constructors, target method body	1*
`prompt7`	Class fields, constructors, target method body	1
`prompt8`	Left and Right changes summary, class fields, constructors, target method body	1*

Tabela 4. Número de conflitos semânticos detectados a partir de diferentes contextos fornecidos nos *prompts*. Valores marcados com * indicam que um dos conflitos detectados por aquele *prompt* não havia sido previamente identificado por ferramentas do SMAT.

A Tabela 4 explora como diferentes contextos de *prompt* influenciam a detecção. Todos os *prompts* foram capazes de identificar ao menos um conflito. Destacam-se os *prompts* `prompt2` e `prompt5`, que detectaram dois conflitos semânticos cada. Em ambos os casos, o contexto fornecido ao modelo inclui um resumo das mudanças realizadas nas branches Left e Right (*Left and Right changes summary*), ou seja, uma descrição concisa das alterações feitas por cada desenvolvedor. No entanto, esse fato não implica

necessariamente uma relação causal entre a presença desse contexto no *prompt* e a quantidade de conflitos detectados.

Essas informações nos levam a responder a primeira pergunta de pesquisa:

RQ1

Os resultados indicam que diferentes combinações de *prompt* e parâmetros do modelo levaram à detecção de conflitos distintos. Entre as configurações avaliadas, a versão *zero-shot* com temperatura 0.0 foi a mais eficaz em detectar conflitos semânticos, identificando até três conflitos em uma única execução.

Ferramenta de Geração de Testes	**Conflitos Detectados**
Differential EvoSuite	6
EvoSuite	5 [1]
Code Llama 70B (união de todas as configurações experimentais)	5 (1*)
Code Llama 70B (apenas os *prompts* em configuração *zero-shot* com temperatura 0.0)	3 (1*)
Randoop	2
Randoop Clean	2 [1]

Tabela 5. Número de conflitos detectados por diferentes ferramentas de testes unitários. Asteriscos (*) indicam conflitos detectados exclusivamente pela ferramenta associada, sem detecção por outras. Números entre colchetes (como [1]) representam falsos positivos reportados. Os dados das outras ferramentas foram extraídos do trabalho de [Da Silva et al. 2024].

4.3. RQ2: O modelo Code Llama 70B é útil para a detecção automática de conflitos semânticos em cenários de merge?

A Tabela 5 posiciona o Code Llama 70B em relação a outras ferramentas de geração de testes. A união de todas as execuções experimentais do Code Llama 70B detectou um total de **5 conflitos**, um desempenho comparável ao da EvoSuite e superior ao da Randoop. O destaque principal é a detecção de um **conflito inédito (*)**, que não foi identificado por nenhuma outra ferramenta, nem mesmo pela Differential EvoSuite, que detectou o maior número de conflitos (6). Além disso, a melhor configuração individual (*zero-shot*, temperatura 0.0) foi capaz de encontrar 3 conflitos, superando ferramentas estabelecidas como a Randoop e a Randoop Clean, que detectaram apenas 2 conflitos cada. Os tempos aproximados para a execução dessas configurações foram de cerca de 1 dia para a configuração *zero-shot* com temperatura 0.0 (aproximadamente 23 horas para cada uma das duas *seeds*) e mais de 5 dias para a união de todas as configurações experimentais (cerca de 132 horas). Um ponto negativo, portanto, é que a ferramenta proposta é substancialmente mais lenta do que as demais. É importante notar que, ao contrário de ferramentas como EvoSuite e Randoop Clean, não foram registrados falsos positivos nos resultados da integração da ferramenta baseada no Code Llama 70B. Com isso, podemos responder à segunda pergunta de pesquisa:

RQ2

A ferramenta baseada no modelo Code Llama 70B demonstrou ser eficaz para a detecção automática de conflitos semânticos em cenários de *merge*, superando ferramentas tradicionais como Randoop e EvoSuite em termos de conflitos detectados. Além disso, foi capaz de identificar um conflito inédito não detectado por outras ferramentas do SMAT, embora sua execução tenha sido significativamente mais lenta.

4.4. RQ3: Como a complexidade do código em análise impacta os resultados do LLM?

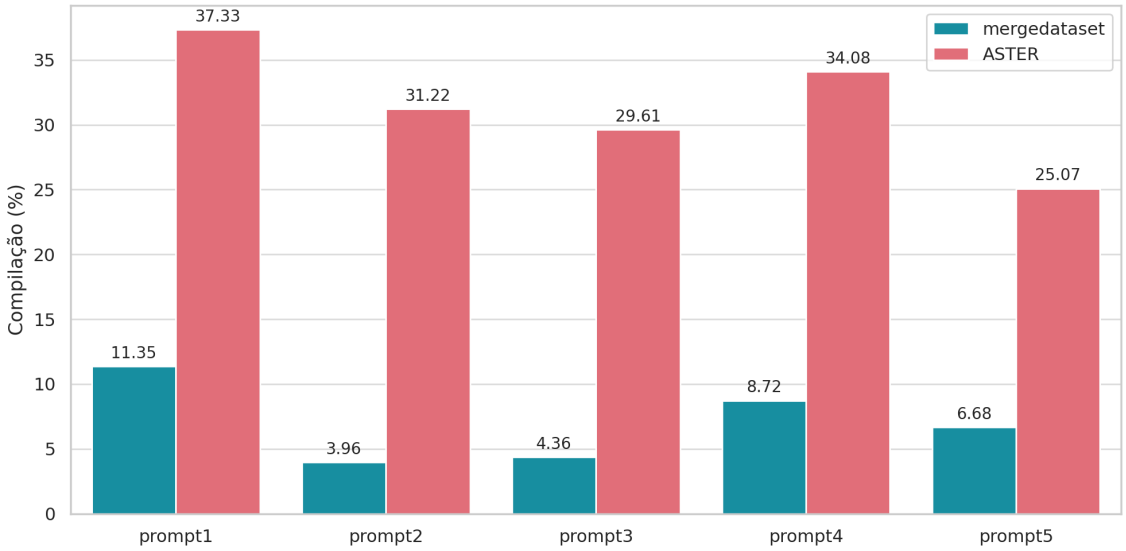

Figura 4. Comparação da taxa de compilação entre os *datasets mergedataset* e *ASTER* para diferentes *prompts*.
Gráfico de barras mostrando a taxa de compilação dos testes gerados pelo modelo Code Llama 70B em dois datasets distintos. Cada barra representa um *prompt* específico, com os valores de taxa de compilação para o mergedataset e o ASTER.

Para avaliar o impacto da complexidade do *dataset* nos resultados, comparamos a taxa de compilação dos testes gerados usando o *mergedataset* e o *ASTER dataset*, conforme mostrado no gráfico da Figura 4. Os *prompts* 1 a 5 selecionados contém as seguintes partes do contexto:

- `prompt1`: (1-shot) atributos da classe, construtores e corpo do método-alvo.
- `prompt2`: (zero-shot) corpo do método-alvo.
- `prompt3`: (zero-shot) atributos da classe e corpo do método-alvo.
- `prompt4`: (zero-shot) construtores e corpo do método-alvo.
- `prompt5`: (zero-shot) atributos da classe, construtores e corpo do método-alvo.

A seleção desses *prompts* específicos foi motivada pela necessidade de adaptar a avaliação ao contexto do *ASTER dataset*, que não possui cenários de conflitos semânticos. Mantivemos quatro variações *zero-shot* que incluem diferentes combinações de elementos de contexto (atributos da classe, construtores e corpo do método-alvo), mas excluímos

o resumo de mudanças das *branches* Left e Right, uma vez que essa informação não é relevante no contexto. Adicionalmente, incluímos um *prompt 1-shot* com todas as informações de contexto para avaliar o impacto de fornecer um exemplo na geração de testes, mesmo em um conjunto de dados sem cenários de *merge*.

Os resultados mostram um aumento drástico e consistente na taxa de compilação ao usar o *ASTER*: o ganho relativo variou de +228.9% a +688.9%. Esses dados ajudam a esclarecer a terceira pergunta de pesquisa:

RQ3

Observamos que a taxa de compilação foi bem maior no *ASTER dataset* em comparação com o *mergedataset*. Isso sugere que a complexidade do código do *dataset* influencia diretamente a capacidade do modelo de gerar testes compiláveis.

4.5. Tempo de Execução

Abordagem	**Tempo de Execução**
EvoSuite	até 13h 9min
Code Llama 70B (*zero-shot*, temperatura 0.0, *seed* 123)	23h 8min
Code Llama 70B (*zero-shot*, temperatura 0.0, *seed* 42)	22h 47min
Code Llama 70B (*zero-shot*, temperatura 0.7, *seed* 123)	10h 2min
Code Llama 70B (*zero-shot*, temperatura 0.7, *seed* 42)	8h 54min
Code Llama 70B (*1-shot*, temperatura 0.7, *seed* 123)	5h 58min
Code Llama 70B (*1-shot*, temperatura 0.7, *seed* 42)	6h 12min
Code Llama 70B (*1-shot*, temperatura 0.0, *seed* 123)	27h 40min
Code Llama 70B (*1-shot*, temperatura 0.0, *seed* 42)	27h 50min
Code Llama 70B (união de todas as configurações experimentais)	132h 30min

Tabela 6. Comparação do tempo de execução entre EvoSuite e a ferramenta baseada no Code Llama 70B. Para o EvoSuite, considerou-se o tempo máximo de 300 segundos por método (valor definido como *timeout* máximo; na prática, o tempo real costuma ser inferior a esse limite). Para o Code Llama 70B, são apresentados os tempos reais de execução para cada configuração, cada uma com 8 *prompts*.

A Tabela 6 apresenta os tempos reais de execução para cada configuração do Code Llama 70B, discriminando os resultados por *seed*. Para o EvoSuite, o tempo apresentado é um teto teórico, pois o valor de 13h 9min considera o limite máximo de 300 segundos por método — um *timeout* adotado de forma padronizada tanto para o EvoSuite quanto para a ferramenta baseada no Code Llama 70B durante a avaliação. Na prática, o tempo real do EvoSuite costuma ser inferior a esse limite.

Os tempos reais de execução do Code Llama 70B variaram significativamente entre as configurações:

- A configuração mais rápida foi *1-shot* com temperatura 0.7, completando em média 6 horas e 5 minutos.
- A configuração mais lenta foi *1-shot* com temperatura 0.0, demandando em média 27 horas e 45 minutos.
- As configurações *zero-shot* apresentaram tempos intermediários: 9 horas e 28 minutos com temperatura 0.7 e 22 horas e 58 minutos com temperatura 0.0.

A união de todas as oito execuções (variando formato do *prompt*, temperatura e *seed*) resulta em um tempo total de 132 horas e 30 minutos, valor substancialmente maior do que o necessário para o EvoSuite. Embora o Code Llama 70B tenha detectado o mesmo número de conflitos que o EvoSuite (5 conflitos, conforme mostrado na Tabela 5), sua demanda computacional é aproximadamente 10 vezes maior.

Esses dados evidenciam que, embora a ferramenta baseada no modelo Code Llama 70B apresente um desempenho promissor na detecção de conflitos semânticos, sua demanda computacional é consideravelmente maior quando comparada ao EvoSuite, especialmente ao se utilizar múltiplas configurações experimentais.

5. Ameaças à Validade

Este estudo apresenta limitações relevantes que podem impactar a generalização dos resultados. O foco exclusivo no modelo Code Llama 70B pode introduzir viés específico, além do risco de contaminação dos dados de treinamento, já que alguns projetos do *mergedataset* podem ter sido utilizados durante o treinamento do modelo. Essa questão, no entanto, é válida para qualquer modelo de linguagem, especialmente os de código, que frequentemente são treinados em grandes conjuntos de dados públicos que incluem projetos de código aberto. Apesar da exploração de diferentes configurações de temperatura, formatos de *prompt* e *seeds*, o conjunto de configurações testadas representa apenas uma pequena parcela das possibilidades, não abrangendo outras técnicas de *prompt engineering*, nem outros parâmetros que poderiam influenciar os resultados. A realização dos experimentos em uma única configuração de *hardware* pode comprometer a reprodutibilidade dos resultados em outros ambientes computacionais. Além disso, os *datasets* utilizados, embora compostos por projetos Java reais, podem não refletir toda a diversidade de projetos existentes, já que se restringem a projetos de código aberto. O foco em Java também limita a extrapolação dos resultados para outras linguagens de programação, que apresentam características sintáticas e semânticas distintas.

Outro ponto importante é o tamanho amostral reduzido (apenas 79 cenários de *merge*), o que limita a significância estatística das conclusões e indica a necessidade de amostras maiores para análises mais robustas. As métricas de avaliação adotadas concentram-se principalmente na taxa de compilação e detecção de conflitos, sem considerar aspectos como cobertura de código ou qualidade dos testes gerados. A avaliação qualitativa dos testes também foi restrita, não abordando critérios como legibilidade e utilidade prática para desenvolvedores. Por fim, a natureza não determinística dos LLMs pode introduzir variabilidade nos resultados que não foi completamente caracterizada, mesmo com o uso de *seeds* fixas.

6. Trabalhos Relacionados

Esta seção apresenta uma análise detalhada dos trabalhos relacionados, organizados em ferramentas para detecção e resolução de conflitos semânticos em *merges*, e abordagens

para geração automatizada de testes utilizando LLMs. Para cada categoria, discutimos as metodologias empregadas, os resultados obtidos e as principais diferenças em relação à nossa proposta.

Detecção e Resolução de Conflitos Semânticos O problema de conflitos semânticos em *merge* tem sido abordado por diferentes perspectivas na literatura. O SafeMerge [Sousa et al. 2018], utilizado como *baseline* no trabalho de [Silva et al. 2020], aplica verificação composicional para detectar conflitos semânticos em *merges*. Sua abordagem é fundamentada na construção de um programa parametrizado, no qual as alterações introduzidas pelas diferentes versões do código (Left e Right) são encapsuladas como *edits* em uma estrutura comum derivada da versão Base. Esse programa parametrizado é então submetido a uma verificação relacional que busca comprovar, por meios formais, que o comportamento do código resultante da fusão é compatível com os comportamentos esperados de ambas as versões modificadas. Essa técnica permite capturar conflitos semânticos não observáveis a partir de simples análise textual, promovendo um nível mais profundo de verificação. Apesar de seu rigor formal, o SafeMerge apresenta limitações práticas. A complexidade computacional da verificação relacional, em especial a construção de *product programs* e a necessidade de geração de invariantes para validação da equivalência comportamental, pode comprometer a escalabilidade da ferramenta em cenários reais e com bases de código maiores. Além disso, ao utilizar os mesmos critérios heurísticos adotados pelo SMAT para caracterização de conflitos, observa-se que a abordagem do SafeMerge gera mais falsos positivos. Especificamente, o SafeMerge apresenta uma taxa de 3.8% de imprecisão, enquanto nossa abordagem baseada em LLMs alcançou uma taxa de 0% de falsos positivos, demonstrando maior precisão na detecção de conflitos semânticos. Para efeito de comparação, o próprio SMAT apresenta uma taxa de 3%, o que reforça a vantagem da nossa proposta nesse aspecto.

O trabalho de [Zhang et al. 2022] propõe a ferramenta *GMERGE*, que sugere soluções automáticas para conflitos de *merge*, incluindo tanto conflitos textuais quanto semânticos. Assim como nossa abordagem, *GMERGE* utiliza LLMs pré-treinados e explora estratégias de *k-shot* na criação de *prompts*. No entanto, há diferenças importantes: *GMERGE* foca em conflitos de um único projeto (Microsoft Edge) e sua avaliação está centrada na observação de mensagens do compilador. A percepção de conflito para os autores é definida quando não há conflitos de *merge* textuais, mas a mesclagem ainda resulta em uma *build* quebrada, testes falhando ou um comportamento de tempo de execução não intencional. A detecção de conflitos, para eles, depende da ocorrência de erros de compilação, pois a ferramenta utiliza mensagens do Clang para identificar *commits* relevantes que causam conflitos. Isso pode ser um problema se o compilador fornecer mensagens pouco direcionadas. Em contraste, nossa percepção de conflito semântico ocorre quando alterações paralelas resultam em falhas de testes ou erros em tempo de execução, mesmo que o código seja mesclado e compilado com sucesso, o que difere de conflitos de *build*, em foco no trabalho de [Zhang et al. 2022]. Além disso, enquanto *GMERGE* explora principalmente a viabilidade de reparar conflitos, nosso foco está na detecção. A complementaridade entre *GMERGE* e nossa abordagem é particularmente interessante: enquanto *GMERGE* atua como uma ferramenta de reparo focada em conflitos que já se manifestaram através de erros de compilação, nossa proposta funciona como um "detector preventivo" que identifica conflitos semânticos. Esta diferença de escopo pode sugerir

como trabalho futuro uma abordagem híbrida, onde nossa metodologia seria aplicada primeiro para detectar conflitos latentes, seguida por uma versão modificada do *GMERGE* para resolvê-los.

Outra abordagem relevante é apresentada por [Maciel et al. 2024]. Se trata de uma outra extensão da ferramenta SMAT. A abordagem deles visa superar a limitação de ferramentas existentes que se restringem a cenários mais simples, onde as contribuições conflitantes ocorrem dentro do mesmo método. Para isso, eles adaptam e avaliam o SMAT para detectar conflitos considerando a interferência causada por mudanças feitas em diferentes métodos e classes. A ferramenta explora a criação de testes utilizando ferramentas de geração de testes já incorporadas no SMAT — EvoSuite, Randoop e Randoop Clean —, além de uma versão customizada do EvoSuite chamada Focused EvoSuite. O Focused EvoSuite foi proposto para otimizar a geração de testes, focando exclusivamente nos métodos alterados, diferentemente da versão padrão do EvoSuite que considera critérios aplicáveis a toda a classe. É importante notar que o dataset utilizado por [Maciel et al. 2024] é diferente do nosso (*mergedataset*). Eles utilizaram uma amostra de 613 cenários sintéticos de *merge* criados a partir do benchmark Defects4J [Just et al. 2014], representando uma amostra seis vezes maior do que estudos anteriores. Todos os 613 cenários em sua amostra apresentavam pelo menos um conflito semântico identificado pela execução dos testes do próprio projeto. Os resultados revelaram a detecção de 230 conflitos semânticos distintos, o que representa 37.5% da amostra total. Em comparação, nosso trabalho utilizou 79 cenários, com 29 cenários contendo conflitos semânticos, e a união de todas as execuções do Code Llama 70B detectou um total de 5 conflitos, incluindo um conflito inédito. Ambas as abordagens buscam aprimorar a capacidade do SMAT de detectar conflitos semânticos, mas através de estratégias distintas: eles aprimoram a granularidade da geração de testes com ferramentas existentes, enquanto nós exploramos o potencial dos LLMs para gerar testes mais eficazes. Os resultados de [Maciel et al. 2024] e os nossos reforçam o potencial das ferramentas na identificação de conflitos, e ambos os estudos enfatizam a importância de adotar múltiplas estratégias e ferramentas de teste automatizadas para detectar esse tipo de conflito.

Se tratando de uma abordagem com análise dinâmica, enfatizamos o trabalho de [Moraes et al. 2024], que propõe uma metodologia para detecção de conflitos semânticos em JavaScript. Eles empregam análise dinâmica para detectar especificamente "overriding assignments" (OAs) em código JavaScript. A abordagem não precisa de *asserts* no código sob análise, exigindo apenas que a versão final do código mesclado seja executada, diferentemente da nossa abordagem baseada em testes. Para validar sua proposta, os autores transpilaram mais de 60 casos de teste de um trabalho relacionado e realizaram um estudo empírico com 159 cenários de *merge* de 50 projetos de código aberto em JavaScript. Nesse estudo, eles detectaram corretamente um cenário de conflito semântico de overriding assignment, com zero falsos positivos, assim como nossa abordagem. A abordagem deles se diferencia também por ser específica para JavaScript, enquanto o nosso trabalho, a priori, é voltado para Java, mas poderia ser adaptado para outras linguagens com algumas modificações no *pipeline* de geração e execução de testes. O trabalho demonstra o potencial da análise dinâmica para detecção de conflitos semânticos e pode ser visto como complementar à nossa abordagem, que se baseia em testes e LLMs.

Por fim, trazemos um trabalho que foca em análise estática para a detecção de

conflitos semânticos. Em seu trabalho, [De Jesus et al. 2024] propõem uma técnica que executa algoritmos de análise estática na versão mesclada do código, a qual é anotada com metadados para indicar as instruções modificadas por cada desenvolvedor. Essa técnica foi implementada para a linguagem Java utilizando o framework Soot e inclui quatro análises principais: *Interprocedural Data Flow* (DF), *Interprocedural Confluence* (CF), *Interprocedural Override Assignment* (OA) e *Program Dependence Graph* (PDG). A técnica foi avaliada com um conjunto de dados de 99 unidades experimentais extraídas de cenários de merge de 39 projetos, sendo 33 unidades com interferência e 66 sem. Os resultados mostraram uma capacidade significativa de detecção de interferência, com um F1 Score de 0.50 e uma Acurácia de 0.60. Comparada a trabalhos anteriores, a abordagem de [De Jesus et al. 2024] apresentou um F1 Score e recall melhores, mas com uma precisão significativamente inferior, gerando 26 falsos positivos. Apesar disso, o desempenho computacional foi notavelmente melhor, com uma mediana de tempo de execução de 17.8 segundos para as quatro análises. Em contraste com nossa abordagem, o trabalho de [De Jesus et al. 2024] utiliza exclusivamente análise estática, enquanto nossa metodologia combina geração de testes baseada em LLM com análise dinâmica através da execução desses testes. Esta diferença fundamental resulta em *trade-offs* distintos: enquanto a análise estática de [De Jesus et al. 2024] oferece tempos de execução significativamente menores (17.8 segundos versus nossos tempos que variam de 6 a 28 horas por configuração), nossa abordagem alcança precisão superior, não registrando falsos positivos em comparação aos 26 falsos positivos reportados por eles.

Geração de Testes com LLMs No contexto da geração de testes com LLMs, destacamos três abordagens recentes: *TESTPILOT* [Schäfer et al. 2024], *ASTER* [Pan et al. 2025] e *CITYWALK* [Zhang et al. 2025].

O *TESTPILOT* é uma ferramenta de geração de testes voltada para JavaScript, o que contrasta com nossa abordagem, que se concentra em Java; seu *pipeline* inclui a extração de comentários de documentação para enriquecer o contexto fornecido ao modelo, enquanto nosso *pipeline* se baseia apenas em informações estruturais do código (atributos, construtores e corpo do método), sem recorrer a documentação textual. Paradoxalmente, esta limitação pode se tornar uma vantagem no contexto de detecção de conflitos semânticos, pois força o LLM a se basear exclusivamente na estrutura e semântica do código, reduzindo a possibilidade de ser "enganado" por documentação inconsistente. Além disso, o *TESTPILOT* prioriza a geração de testes que não falham, buscando maximizar a cobertura, enquanto nosso foco está na detecção de conflitos semânticos em cenários de *merge*. Observamos em nosso estudo que a utilização de diferentes contextos nos *prompts* levou à detecção de conflitos distintos, sugerindo que diferentes combinações de prompt e parâmetros do modelo permitem explorar diferentes tipos de falhas e maximizar a detecção. Enquanto o *TESTPILOT* demonstra que a inclusão de documentação e exemplos de uso melhora a eficácia na geração de testes, nossos resultados indicam que, mesmo sem documentação textual, a informação estrutural do código é suficiente para detectar conflitos semânticos em Java.

O *ASTER*, por sua vez, é uma ferramenta para geração de testes em Java e Python, que explora diferentes contextos e técnicas de *mocking* para aumentar a cobertura dos testes gerados. Diferentemente da nossa abordagem, que executa a geração de testes em uma

única etapa para cada cenário, o *ASTER* adota um processo iterativo, refinando os testes até atingir uma cobertura desejada. Além disso, o *ASTER* enfatiza a utilidade e compreensibilidade dos testes gerados, aspectos também considerados em nossa análise, mas com ênfase na capacidade de detecção de conflitos. Em termos de contexto para a geração de *prompts*, o *ASTER* utiliza uma fase de pré-processamento que realiza análise estática extensiva do programa. Isso inclui a extração de informações detalhadas como o escopo de teste (métodos-alvo), construtores relevantes, métodos auxiliares (*getters*/*setters*), cadeias de chamadas para métodos privados e dados para facilitar o *mocking* (identificação de atributos, tipos e métodos a serem *mockados*). Esta abordagem rica em contexto contrasta com a nossa, que se baseia em uma análise mais simples, focando apenas em atributos, construtores da classe-alvo e corpo do método-alvo. Além disso, a natureza do *dataset* que utilizamos não requer a identificação de métodos privados ou atributos que não sejam diretamente acessíveis, o que simplifica o processo de geração de testes. Quanto ao pós-processamento, o *ASTER* emprega uma fase dedicada ao reparo de testes gerados, corrigindo erros de compilação e *runtime*. Isso envolve a sanitização da saída do LLM, correção de falhas de asserção e, notavelmente, um processo de aumento de cobertura, onde *prompts* adicionais são criados para instruir o LLM a gerar casos de teste que exercitem linhas de código inicialmente não cobertas. Em nosso trabalho, o foco do pós-processamento está na validação dos testes para identificar conflitos semânticos; embora os testes gerados sejam executados para determinar seu comportamento, não há uma fase explícita de reparo de testes ou aumento de cobertura iterativo com o LLM da mesma forma que no *ASTER*. Nossos testes são avaliados principalmente pela sua capacidade de quebrar em cenários de conflito, e não por sua cobertura total.

Já o *CITYWALK* propõe um *pipeline* para geração de testes em C++ que incorpora técnicas de *Retrieval-Augmented Generation* (RAG). Seu foco principal é gerar testes unitários executáveis e de alta cobertura capazes de encontrar bugs, superando os desafios específicos do C++, como ponteiros e funções virtuais. Em termos de contextos utilizados, o *CITYWALK* emprega uma análise estática extensiva para extrair dependências de configuração (uso de bibliotecas de terceiros e requisitos do ambiente de compilação) e dependências de dados *cross-file* (análise de fluxo de dados em ASTs para capturar interações entre arquivos). Além disso, ele recupera trechos de código e documentação como contextos de intenção, usando uma estratégia híbrida de recuperação (semântica para documentação e correspondência exata para código). Em contraste, nossa abordagem, focada em Java, utiliza informações locais à classe. Embora nossa abordagem não utilize RAG, reconhecemos o potencial dessa técnica para trabalhos futuros, especialmente porque o conhecimento contextual externo ao código-fonte da classe pode ser crucial para detectar conflitos semânticos que podem emergir de interações complexas entre diferentes partes do sistema que não são óbvias a partir do contexto local. Um outro ponto de diferenciação do trabalho é o uso de diversos modelos em sua análise, incluindo GPT-4o, CodeGeeX4 e DeepSeek-V2.5, enquanto nossa proposta se concentra exclusivamente no Code Llama 70B. É válido ressaltar que o módulo incorporado ao SMAT nesse trabalho pode ser facilmente adaptado para usar outros modelos, caso necessário. Os resultados do *CITYWALK* mostram que a ferramenta gera menos erros de compilação e alcança maior cobertura com menos testes quando comparada a soluções que utilizam os modelos diretamente, reforçando o potencial dos LLMs na geração de testes, quando bem associados a outros processos. Em particular, o *CITYWALK* superou o GPT-4o em

42.05% na taxa de sucesso de compilação e em 26.41% na cobertura de linha, com uma quantidade significativamente menor de testes gerados. Em comparação, nossa proposta possui uma taxa de compilação muito mais baixa (8.24% em média no *mergedataset* e 31.46% no *ASTER dataset*), enquanto o *CITYWALK* alcançou uma taxa de compilação média de 70.18% em um conjunto de dados em C++. Isso pode ser atribuído à natureza dos *datasets* utilizados, mas também à diferença entre as abordagens.

7. Conclusão

Este trabalho demonstrou que uma ferramenta baseada em modelos de linguagem, como o Code Llama 70B, representa uma alternativa promissora para a detecção de conflitos semânticos em código. Os resultados obtidos superam ou se igualam a ferramentas automatizadas tradicionais, detectando cinco conflitos distintos dentre os 29 presentes no *dataset*, incluindo um conflito inédito, sem introduzir falsos positivos.

Os experimentos revelaram três descobertas sobre o uso de LLMs para detecção de conflitos semânticos. Primeiro, nenhum *prompt* isolado foi capaz de identificar todos os conflitos observados: a combinação de diferentes execuções mostrou-se mais eficaz, evidenciando que diferentes configurações detectam diferentes tipos de conflitos. Entre as configurações testadas, os *prompts zero-shot* com temperatura 0.0 apresentaram melhor desempenho no *mergedataset*, identificando até três conflitos em uma única execução. Segundo, a temperatura do modelo influencia significativamente os resultados. Nas execuções com temperatura 0.7, os *prompts 1-shot* superaram os *zero-shot*, reforçando a importância de explorar diferentes estratégias de configuração para otimizar a detecção. E, em terceiro, não existe correlação evidente entre o número de testes compilados e a quantidade de conflitos identificados. Mas, ainda assim, a taxa de compilação mais baixa no *mergedataset* pode estar relacionada à complexidade dos cenários, conforme sugerido pelos experimentos no *ASTER dataset*.

Os resultados apresentam implicações diretas para o desenvolvimento de ferramentas baseadas em LLMs para detecção de conflitos semânticos. A necessidade de combinar múltiplas configurações sugere que trabalhos futuros devem focar em estratégias de combinação inteligente de diferentes configurações de *prompts* e parâmetros. A sensibilidade dos modelos às configurações de temperatura e tipo de *prompt* evidencia a importância de desenvolver métodos adaptativos que ajustem automaticamente esses parâmetros conforme o contexto do código analisado. O custo computacional significativo observado (10 vezes maior que o EvoSuite) aponta para a necessidade de pesquisas em otimização de eficiência, mantendo a eficácia na detecção. Trabalhos futuros devem investigar métodos para reduzir esse custo, como o uso de modelos menores e mais especializados.

Como direções futuras, pretendemos investigar métodos para melhorar a taxa de compilação dos testes gerados, desenvolver estratégias de ajuste de parâmetros e explorar a combinação adaptativa de *prompts*. Além disso, planejamos investigar o uso de técnicas de RAG, agentes e outros modelos de linguagem para criar ferramentas mais eficientes e eficazes na detecção de conflitos semânticos.

Disponibilidade de Artefatos

O código-fonte da versão do SMAT utilizada neste trabalho, incluindo a integração com o Code Llama, está disponível em [Barbosa 2025b]. Os artefatos experimentais, como arquivos de entrada, resultados dos experimentos (testes gerados, relatórios, gráficos e tabelas) e scripts Python para execução e análise, podem ser acessados em [Barbosa 2025a].

Agradecimentos

Agradecemos aos integrantes do Software Productivity Group, ao INES (Instituto Nacional de Engenharia de Software) pelo apoio disponibilizando os servidores necessários para a execução dos experimentos, e ao CNPq (projeto 465614/2014-0) pelo financiamento deste trabalho.

Referências

[Barbosa 2025a] Barbosa, N. (2025a). Apêndice online. https://github.com/nathaliafab/smat-codellama-artifacts.

[Barbosa 2025b] Barbosa, N. (2025b). Smat com integração code llama. https://github.com/nathaliafab/SMAT/tree/codellama-integration.

[Brun et al. 2011] 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 et al. 2024] 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.

[De Jesus et al. 2024] 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.

[Eclipse Foundation 2020] Eclipse Foundation (2020). Eclipse Cargo Tracker: Applied Domain-Driven Design Blueprints for Jakarta EE. Acessado em: 13 jun. 2025.

[Fraser and Arcuri 2011] 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.

[Just et al. 2014] 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 et al. 2024] 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 et al. 2021] 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 2023] Meta AI (2023). Introducing code llama, a state-of-the-art large language model for coding. Accessed: 2025-05-10.

[Moraes et al. 2024] 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 (2023). Ollama - run large language models locally. Acesso em: 12 jun. 2025.

[OpenLiberty 2018] OpenLiberty (2018). DayTrader8 Sample. Acessado em: 13 jun. 2025.

[Pacheco and Ernst 2007] 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 et al. 2025] Pan, R., Kim, M., Krishna, R., Pavuluri, R., and Sinha, S. (2025). Aster: Natural and multi-language unit test generation with llms.

[Sarma et al. 2012] 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 et al. 2024] 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 et al. 2020] 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 et al. 2018] Sousa, M., Dillig, I., and Lahiri, S. K. (2018). Verified three-way program merge. *Proc. ACM Program. Lang.*, 2(OOPSLA).

[Tree-sitter 2024] Tree-sitter (2024). Tree-sitter - a parser generator tool and incremental parsing library. Acesso em: 12 jun. 2025.

[White et al. 2023] 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.

[Zhang et al. 2022] 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 et al. 2025] Zhang, Y., Lu, Q., Liu, K., Dou, W., Zhu, J., Qian, L., Zhang, C., Lin, Z., and Wei, J. (2025). Citywalk: Enhancing llm-based c++ unit test generation via project-dependency awareness and language-specific knowledge.