INSTITUTO FEDERAL
Pernambuco

AVALIANDO O *MERGE* SEMI-ESTRUTURADO: um estudo comparativo de corretude e desempenho da ferramenta SESAME

EVALUATING SEMISTRUCTURED MERGE: a comparative evaluation of the SESAME tool

David Lucas Alves de Almeida
dlaa@discente.ifpe.edu.br
Guilherme José de Carvalho Cavalcanti
guilherme.cavalcanti@belojardim.ifpe.edu.br

RESUMO

A integração de modificações concorrentes é um desafio crítico no desenvolvimento colaborativo de software. Enquanto ferramentas de *merge* não estruturadas (textuais) frequentemente geram conflitos espúrios, abordagens estruturadas, apesar de mais precisas, podem introduzir erros semânticos silenciosos e possuem alto custo computacional. Neste contexto, a ferramenta SESAME propõe uma abordagem semi-estruturada inovadora, utilizando separadores sintáticos para inferir a estrutura do código de forma leve. Este trabalho apresenta uma avaliação empírica do SESAME através de uma replicação diferenciada do estudo de Schesch et al. (2024). Adotando um protocolo experimental rigoroso, avaliou-se a ferramenta não apenas pela contagem de conflitos, mas pela corretude semântica verificada através da execução de suítes de testes automatizados em um vasto conjunto de projetos Java. O desempenho do SESAME foi contrastado com ferramentas do estado da arte (Git *Merge*, Spork e IntelliMerge) utilizando um *dataset* robusto com mais de 5.000 cenários de *merge*, com métricas de Corretude, Tempo de Execução e Redução de Esforço (*Effort Reduction*). Os resultados revelam que, embora ferramentas puramente estruturadas maximizem a resolução de conflitos, elas introduzem taxas elevadas de erros (11% no Spork) e impõem latência proibitiva. Em contraste, o SESAME consolidou-se como um *sweet spot*: manteve a segurança próxima ao padrão textual (apenas 3% de erro) com um tempo de execução e uma ordem de grandeza inferior às ferramentas estruturadas, validando-se como uma alternativa pragmática e eficiente para esteiras de Integração Contínua.

Palavras-chave: *merge* Semi-Estruturado. Ferramentas de *merge*. Conflitos de *merge*. Avaliação de Desempenho. Corretude de Software.

Instituto Federal de Pernambuco. Campus Belo Jardim. Curso de Bacharelado em Engenharia de Software. 18 de novembro de 2025.

ABSTRACT

The integration of concurrent modifications is a critical challenge in collaborative software development. While unstructured (textual) merge tools often generate spurious conflicts, structured approaches, although more accurate, can introduce silent semantic errors and have high computational costs. In this context, the SESAME tool proposes an innovative semi-structured approach, using syntactic separators to infer the code structure in a lightweight manner. This work presents an empirical evaluation of SESAME through a differentiated replication of the study by Schesch et al. (2024). Adopting a rigorous experimental protocol, the tool was evaluated not only by counting conflicts, but also by semantic correctness verified through the execution of automated test suites on a vast set of Java projects. SESAME's performance was compared with state-of-the-art tools (Git Merge, Spork, and IntelliMerge) using a robust dataset with over 5,000 merge scenarios, with metrics for Correctness, Execution Time, and Effort Reduction. The results reveal that, although purely structured tools maximize conflict resolution, they introduce high error rates (11% in Spork) and impose prohibitive latency. In contrast, SESAME has established itself as a sweet spot: it maintained security close to the textual standard (only 3% error) with an execution time and order of magnitude lower than structured tools, validating itself as a pragmatic and efficient alternative for Continuous Integration pipelines.

Keywords: Semistructured merge. Merge Tools. Merge Conflicts. Performance Evaluation. Merge Correctness.

---

1 INTRODUÇÃO

O desenvolvimento de software moderno é um processo inerentemente colaborativo, onde equipes trabalham simultaneamente em diferentes partes de um mesmo projeto. Para coordenar essas contribuições, Sistemas de Controle de Versão (VCS), como o Git, são indispensáveis. No entanto, a operação de *merge*, que é o mecanismo central para integrar o trabalho de diferentes desenvolvedores, permanece um desafio crítico e um conhecido problema na engenharia de software.

As ferramentas de *merge* tradicionais, ditas não estruturadas (ou textuais), tratam o código como simples sequências de linhas. Embora sejam rápidas e universais, elas carecem de compreensão sintática, o que frequentemente resulta em "conflitos espúrios" em mudanças que são, na verdade, semanticamente independentes, conforme Cavalcanti, Borba *et al.* (2024). No outro extremo, ferramentas estruturadas analisam a árvore sintática do código (AST) para evitar esses conflitos, mas à custa de maior complexidade e especificidade para cada linguagem. (Schesch *et al.*, 2024) Neste cenário, surge uma terceira via: o *merge* semi-estruturado. Esta abordagem procura um equilíbrio, analisando estruturas de alto nível (como classes e métodos) e tratando o resto como texto. A ferramenta SESAME insere-se nesta categoria, propondo uma técnica inovadora que utiliza separadores sintáticos para refinar a resolução de conflitos sem a complexidade de uma análise sintática completa.

Contudo, a eficácia destas ferramentas tem sido difícil de medir. Historicamente, a avaliação de ferramentas de *merge* tem sido limitada por metodologias que não distinguem adequadamente um *merge* correto de um incorreto. Como demonstrado no estudo recente de Schesch *et al.* (2024), intitulado “Evaluation of Version Control *Merge* Tools”, muitos estudos anteriores focaram-se apenas na contagem de conflitos reportados. Esta métrica é enganadora: uma ferramenta pode ser “silenciosa” (não reportar conflitos), mas gerar um código com *merge* incorreto, introduzindo erros de compilação ou bugs sutis no comportamento do programa. O autor argumenta que um *merge* incorreto é significativamente mais custoso para o ciclo de desenvolvimento do que um conflito explícito, pois exige depuração posterior. Este trabalho propõe-se a preencher uma lacuna na literatura ao avaliar empiricamente a ferramenta SESAME sob uma ótica de rigor metodológico que faltou às suas avaliações preliminares.

Para tal, este estudo configura-se como uma replicação diferenciada (*differentiated replication*) do trabalho de Schesch *et al.* (2024). A replicação de estudos experimentais é um pilar fundamental para o amadurecimento da engenharia de software como ciência, permitindo verificar a generalização de resultados e validar novas tecnologias em benchmarks estabelecidos. Enquanto o estudo original de Schesch *et al.* (2024) avaliou ferramentas como Git *merge*, Spork e IntelliMerge, este trabalho estende a sua infraestrutura experimental para incluir e avaliar o SESAME. A avaliação não se baseará apenas na contagem de conflitos, mas utilizará a execução de suítes de testes automatizados em milhares de cenários de *merge* reais (extraídos de projetos open-source) como um proxy para verificar a corretude do código resultante.

O objetivo principal é determinar a eficácia do SESAME em termos de correção, desempenho e redução de esforço, posicionando-o comparativamente face ao estado da arte. Especificamente, este estudo pretende responder: onde se situa o SESAME no compromisso entre reduzir conflitos textuais e evitar a introdução de erros silenciosos.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta os conceitos fundamentais sobre controle de versão, as diferentes abordagens de *merge* e as metodologias para avaliação da eficácia dessas abordagens. A operação de *merge* é o processo pelo qual as contribuições de diferentes linhas de desenvolvimento são integradas.

2.1 Versionamento de Software e o Desafio do *Merge*

O desenvolvimento de software moderno é uma atividade eminentemente colaborativa, na qual múltiplos desenvolvedores trabalham em paralelo. Para gerenciar essa complexidade, os Sistemas de Controle de Versão (VCS) são ferramentas indispensáveis. A operação de *merge* é o processo pelo qual as contribuições de diferentes linhas de desenvolvimento são integradas. Os estudos de Mens (2002) sobre o tema mostram que o *merge* de software é reconhecido como uma área de pesquisa complexa e desafiadora.

O processo de *merge* mais comum é o *three-way merge* (*merge* de três vias), que utiliza três versões de um arquivo para calcular o resultado final: a versão base (o ancestral comum mais próximo), e as duas versões modificadas, *left* e *right* (ou local

e remota). O objetivo de uma ferramenta de *merge* é integrar as alterações de forma automática. Quando isso não é possível, a ferramenta reporta um conflito, que exige intervenção manual do desenvolvedor.

2.2 O Espectro das Ferramentas de *Merge*

As ferramentas de *merge* podem ser classificadas em um espectro de abordagens, que variam em complexidade, precisão e custo computacional.

2.2.1 *Merge* Não Estruturado (Textual)

A abordagem mais tradicional e amplamente utilizada, presente em ferramentas como o git *merge* padrão, é o *merge* não estruturado. Essa técnica trata os arquivos de código-fonte como simples sequências de linhas de texto, utilizando algoritmos como o diff3 para identificar e mesclar as diferenças (Khanna; Kunal; Pierce, 2007).

- **Vantagens:** São rápidas, consomem poucos recursos e são agnósticas à linguagem de programação, podendo ser aplicadas a qualquer tipo de arquivo de texto.
- **Desvantagens:** Sua principal fraqueza é a falta de compreensão sobre a estrutura sintática do código. Isso leva à geração de conflitos espúrios (falsos positivos), por exemplo, quando dois desenvolvedores adicionam métodos diferentes na mesma região do arquivo. Além disso, podem falhar em detectar conflitos semânticos reais (falsos negativos), resultando em *merges* que parecem limpos, mas que quebram a compilação ou o comportamento do programa.

2.2.2 *Merge* Estruturado

Para superar as limitações do *merge* textual, pesquisadores desenvolveram o *merge* estruturado, que analisa a estrutura sintática do código-fonte. Ferramentas como o Spork baseiam-se em correspondência de nós em ASTs, enquanto o IntelliMerge utiliza grafos de elementos de programa. Ambas buscam precisão máxima, mas sofrem com o *overhead* de desempenho e complexidade de implementação.

- **Vantagens:** Ao compreender a estrutura do programa, podem resolver conflitos que confundem as ferramentas textuais, como a reordenação de declarações de métodos ou campos. Isso resulta em uma maior precisão e na redução de conflitos espúrios.
- **Desvantagens:** São inerentemente específicas para cada linguagem, exigindo um esforço significativo de implementação para cada nova linguagem suportada. Além disso, as análises e *merges* de ASTs completas podem ser computacionalmente caras, especialmente em arquivos grandes. Exemplos notáveis de ferramentas estruturadas incluem JDime, Spork e Mastery.

2.2.3 *Merge* Semi-Estruturado

Buscando um ponto de equilíbrio, a abordagem semi-estruturada foi proposta como um híbrido entre as duas anteriores. Ferramentas semi-estruturadas, como a FST-Merge, analisam apenas as estruturas de alto nível do código (como declarações de classes, métodos e campos), representando-as em uma AST parcial. As estruturas de baixo nível, como corpos de métodos e expressões, são tratadas como texto plano e mescladas com o auxílio de uma ferramenta não estruturada (Apel *et al.*, 2011).

Embora essa abordagem reduza a complexidade e o custo computacional do *merge* estruturado, ela herda parte das imprecisões do *merge* textual ao lidar com as estruturas de baixo nível.

2.2.4 A Abordagem Inovadora do SESAME

Dentro da categoria semi-estruturada, a ferramenta SESAME, proposta por Cavalcanti, Borba *et al.* (2024), introduz uma técnica inovadora. Em vez de se basear em uma AST parcial, a SESAME infere a estrutura do código alavancando separadores sintáticos específicos da linguagem (como , , (,), ; em Java). A ferramenta pré-processa as versões do código, inserindo quebras de linha antes e depois de cada separador. Isso faz com que blocos de código que antes estavam na mesma linha (por exemplo, a condição e o corpo de um if) passem a ocupar linhas distintas e não-consecutivas. Em seguida, um algoritmo de *merge* não estruturado é invocado sobre o código pré-processado, sendo capaz de resolver muitos conflitos que a abordagem semi-estruturada tradicional não conseguiria.

Para ilustrar o funcionamento, considere o mecanismo de inferência de estrutura. Conforme detalhado por Cavalcanti, Borba *et al.* (2024), o processo segue três etapas principais:

- **Pré-processamento:** A ferramenta insere quebras de linha e marcadores especiais (*placeholders*) antes e depois de cada separador sintático. Isso isola os blocos lógicos em linhas distintas.
- ***Merge* Textual:** Uma ferramenta não estruturada (diff3) é executada sobre esse código transformado. Como os blocos agora estão em linhas diferentes, o algoritmo textual consegue distinguir mudanças que antes colidiriam na mesma linha.
- **Pós-processamento**: Os marcadores e quebras de linha artificiais são removidos, restaurando o código original com o *merge*.

2.3 Exemplo Prático de Comportamento das Ferramentas

Para demonstrar a capacidade das ferramentas em resolver conflitos granulares, utiliza-se um cenário que envolve modificações simultâneas na definição de um laço de repetição (*loop*) e a adição de novos métodos.

A Figura 1 apresenta o estado inicial:

- **Base:** Método `iterar` padrão.
- ***Left*:** Altera o início do laço (`i=1`) e adiciona o método `subtrair`.
- ***Right*:** Altera o fim do laço (`i<11`) e adiciona o método `multiplicar`.

Base

```java
public class Teste {
    public void iterar() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Iteração: " + i);
        }
    }
}
```

***Left* (Local)**

```java
public class Teste {
    public void iterar() {
        for (int i = 1; i < 10; i++) {
            System.out.println("Iteração: " + i);
        }
    }
    public int subtrair(int a, int b) {
        return a - b;
    }
}
```

***Right* (Remoto)**

```java
public class Teste {
    public void iterar() {
        for (int i = 0; i < 11; i++) {
            System.out.println("Iteração: " + i);
        }
    }
    public int multiplicar(int a, int b) {
        return a * b;
    }
}
```

Figura 1 – Cenário de *merge*: modificações concorrentes na mesma linha do `for` e adição simultânea de métodos distintos.

A seguir, demonstra-se como cada categoria de ferramenta processa este cenário.

2.3.1 *Merge* Textual (Git *Merge*)

O Git *Merge* (não estruturado) trata o arquivo como uma sequência de linhas. Como existem modificações que ocorreram no mesmo ponto de inserção nos dois lados, a ferramenta não consegue determinar a ordem correta e reporta um conflito, exigindo intervenção manual (Figura 2).

2.3.2 *Merge* Estruturado (Spork)

O Spork (estruturado) transcende a visão textual ao construir a Árvore Sintática Abstrata (AST). Ele identifica que as modificações no laço `for`, embora ocupem a mesma linha física, referem-se a nós filhos distintos dentro da estrutura do `ForStatement`: a inicialização (`ForInit`) alterada pelo ramo *Left* e a expressão de condição (`Expression`) ajustada pelo ramo *Right*.

Além disso, ao analisar o escopo da classe `Teste`, a ferramenta interpreta as inserções ao final do arquivo não como uma colisão de blocos de texto, mas como a adição de dois novos nós `MethodDeclaration` independentes (`subtrair` e `multiplicar`). Isso

```
1  public class Teste {
2
3      public void iterar() {
4  <<<<<<< HEAD
5          for (int i = 1; i < 10; i++) {
6  =======
7          for (int i = 0; i < 11; i++) {
8  >>>>>>> branch-right
9              System.out.println("Iteração: " + i);
10         }
11     }
12
13 <<<<<<< HEAD
14     public int subtrair(int a, int b) {
15         return a - b;
16     }
17 =======
18     public int multiplicar(int a, int b) {
19         return a * b;
20     }
21 >>>>>>> branch-right
22 }
23
```

Figura 2 – Falha dupla do Git *Merge*. Ocorre um conflito intra-linha na definição do laço `for` e um conflito de inserção no final da classe, onde a ferramenta não consegue determinar a ordem de inclusão dos novos métodos `subtrair` e `multiplicar`.

permite que o Spork realize uma fusão granular das propriedades do laço e, simultaneamente, anexe os novos métodos como irmãos na árvore, resultando em um código válido e sem conflitos (Figura 4).

2.3.3 *Merge* Semi-estruturado (SESAME)

O SESAME (semi-estruturado) utiliza separadores sintáticos (‘{’, ‘}’, ‘,’, ‘(’, ‘)’ e ‘;’) para inferir a estrutura. Ao pré-processar o texto inserindo quebras de linha nos separadores, ele isola os blocos de código. Isso permite que o algoritmo textual subjacente distinga as duas inserções como blocos independentes, alcançando o mesmo resultado de sucesso do Spork, mas com menor custo computacional (Figura 3).

```java
public class Teste {

    public void iterar() {
        for (int i = 1; i < 11; i++) {
            System.out.println("Iteração: " + i);
        }
    }

    public int subtrair(int a, int b) {
        return a - b;
    }

    public int multiplicar(int a, int b) {
        return a * b;
    }

}

```

Figura 3 – Resultado obtido tanto pelo Spork quanto pelo SESAME: fusão automática bem-sucedida (Clean *merge*).

2.4 Visualização do Processamento Interno

Para compreender por que o resultado difere, é necessário observar a representação interna que cada ferramenta constrói a partir do código fonte.

2.4.1 Representação em Árvore (Estruturado)

Enquanto o Git opera cegamente sobre linhas de texto, o Spork converte o código em uma Árvore Sintática Abstrata (AST). A Figura 4 ilustra a hierarquia estrutural interpretada pela ferramenta.

O conflito é mitigado porque o Spork não enxerga uma colisão de caracteres na linha do laço, mas sim alterações em nós filhos distintos do `ForStatement`: a inicialização (`ForInit`) vinda do *Left* e a condição de parada (`Expression`) vinda do *Right*. Simultaneamente, a ferramenta compreende que os métodos `subtrair` e `multiplicar` são novos nós independentes que devem ser anexados ao nó pai `ClassDeclaration`, permitindo a reconciliação completa da estrutura.

2.4.2 Representação com Separadores (SESAME)

O SESAME não constrói a árvore completa, mas transforma o arquivo inserindo marcadores artificiais ao redor de separadores sintáticos específicos: **{ }, ( ), ; e ,** (Cavalcanti; Borba *et al.*, 2024).

A Figura 5 demonstra como as linhas de código são “explodidas” verticalmente. Observe como os parênteses, vírgulas e pontos-e-vírgulas são isolados por marcado-

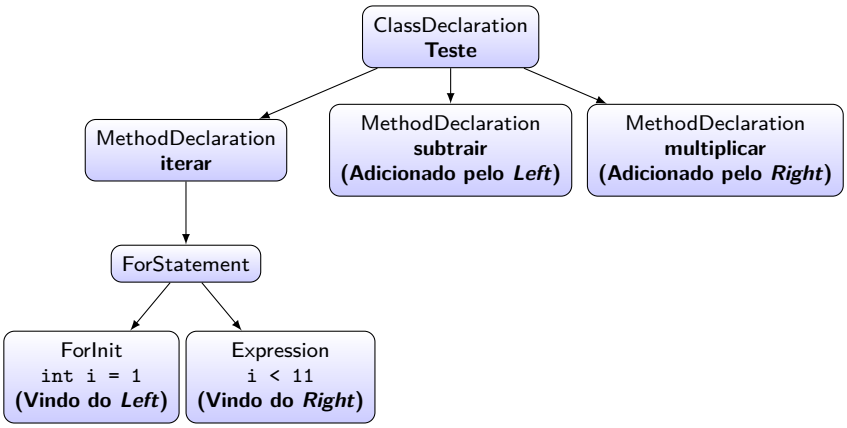

Figura 4 – Árvore Sintática Abstrata (AST) resultante do Spork.

res $$$$$$. Isso permite que o algoritmo de *diff* alinhe a estrutura do código mesmo se houver mudanças em linhas adjacentes.

***Left* (Transformado)**

```
public class Teste
$$$$$${
    public void iterar
    $$$$$$(
    $$$$$$)
$$$$$${
        for
        $$$$$$(
        $$$$$$int i = 1
        $$$$$$;
        $$$$$$i < 10
        $$$$$$;
        $$$$$$i++
        $$$$$$)
        $$$$$${
            System.out.println
            $$$$$$(
                $$$$$$"Iteração: " + i
            $$$$$$)
            $$$$$$;
        $$$$$$}
$$$$$$}
    public int subtrair
    $$$$$$(int a
    $$$$$$,
    $$$$$$int b
    $$$$$$)
    $$$$$${
        return a - b
        $$$$$$;
    $$$$$$}
$$$$$$}

```

***Right* (Transformado)**

```
public class Teste
$$$$$${
    public void iterar
    $$$$$$(
    $$$$$$)
$$$$$${
        for
        $$$$$$(
        $$$$$$int i = 0
        $$$$$$;
        $$$$$$i < 11
        $$$$$$;
        $$$$$$i++
        $$$$$$)
        $$$$$${
            System.out.println
            $$$$$$(
                $$$$$$"Iteração: " + i
            $$$$$$)
            $$$$$$;
        $$$$$$}
$$$$$$}
    public int multiplicar
    $$$$$$(int a
    $$$$$$,
    $$$$$$int b
    $$$$$$)
    $$$$$${
        return a * b
        $$$$$$;
    $$$$$$}
$$$$$$}

```

Figura 5 – Visualização detalhada do SESAME: inserção de quebras e marcadores ($$$) nos separadores sintáticos configurados, isolando cada elemento estrutural.

2.5 O Desafio de Avaliar Ferramentas de *Merge*

Tão importante quanto desenvolver novas ferramentas é a capacidade de avaliá-las de forma justa e realista. Enquanto o desenvolvimento do SESAME foca na eficiência da resolução de conflitos, a sua validação necessita de um critério de aceitação mais rigoroso do que a mera contagem de linhas. Adota-se, portanto, a perspectiva de Schesch *et al.* (2024), que prioriza a integridade semântica do software resultante.

Conforme apontado por Schesch *et al.* (2024), a literatura anterior sofre com metodologias que apresentam falhas significativas, listadas a seguir:

1. **Falta de Distinção da Correção:** A maioria das avaliações existentes mede o sucesso de uma ferramenta apenas pela sua capacidade de produzir um *merge* limpo (sem conflitos), tratando todos os *merges* limpos como igualmente bem-sucedidos. No entanto, um *merge* limpo pode ser incorreto, ou seja, pode introduzir um erro de compilação ou um defeito de comportamento que só será detectado posteriormente.
2. **Dados Pouco Representativos:** Muitas avaliações utilizam cenários de *merge* sintéticos ou coletados apenas da branch principal dos repositórios, o que não reflete a variedade e a complexidade dos *merges* que ocorrem em branches de desenvolvimento, que são frequentemente deletadas após o a conclusão do processo.
3. **"Verdade Fundamental" Questionável:** Alguns estudos comparam o resultado da ferramenta com a versão final do código *commitada* pelo desenvolvedor. Contudo, se o próprio desenvolvedor utilizou uma ferramenta de *merge* imperfeita para gerar essa versão, o estudo pode acabar penalizando uma ferramenta mais correta.

Para superar essas limitações, uma metodologia de avaliação mais rigorosa foi proposta. Ela se baseia na execução de suítes de teste automatizadas como um proxy para a verificação da correção do *merge*. Assim, um *merge* limpo é classificado como correto se todos os testes passarem, e incorreto se algum teste falhar. Adicionalmente, a metodologia introduz uma métrica de "Redução de Esforço", que leva em conta o custo relativo (k) de um *merge* incorreto, partindo da premissa de que consertar um bug silencioso é significativamente mais caro do que resolver um conflito de *merge* explícito.

2.6 Replicação na Engenharia de Software

A replicação é um componente essencial do método científico, permitindo que a comunidade verifique se resultados empíricos são consistentes e generalizáveis para além do contexto de um único estudo. Segundo Carver *et al.* (2014), a replicação pode ser definida como a repetição deliberada de um estudo empírico com o objetivo de determinar se os resultados originais podem ser reproduzidos. Sem a replicação, observações isoladas podem ser fruto do acaso ou de condições específicas irreprodutíveis, impedindo a consolidação do conhecimento na área.

Na Engenharia de Software, as replicações podem ser classificadas em diferentes tipos, dependendo do grau de fidelidade ao estudo original. Shull *et al.* (2008) distinguem entre replicações exatas, onde se busca seguir os procedimentos originais o mais fielmente possível, e replicações conceituais, que buscam responder à mesma questão de pesquisa utilizando métodos diferentes . Mäntylä, Lassenius e Vanhanen (2010) expandem essa visão, discutindo as replicações diferenciadas, onde variações deliberadas são introduzidas nas variáveis do experimento, como a alteração da população estudada ou das ferramentas avaliadas, para testar a robustez das teorias.

Este trabalho classifica-se como uma replicação diferenciada do estudo conduzido por Schesch *et al.* (2024). Adota-se, de forma rigorosa, a mesma infraestrutura experimental, bem como as mesmas métricas de avaliação (correção via suítes de teste) e critérios de sucesso do estudo original. A diferenciação principal reside na introdução de uma nova variável independente: a ferramenta de *merge* SESAME. Desta forma, o estudo não apenas valida a metodologia proposta por Schesch *et al.* (2024), mas também estende o corpo de conhecimento ao avaliar o comportamento de uma nova abordagem semi-estruturada frente aos *baselines* da literatura.

2.7 Síntese e Lacuna na Literatura

A literatura apresenta um campo de pesquisa ativo, com um espectro de ferramentas de *merge* que buscam equilibrar simplicidade, desempenho e precisão. Ao mesmo tempo, emerge um consenso sobre a necessidade de metodologias de avaliação mais robustas, que considerem a correção do código resultante como a principal métrica de sucesso.

Neste ponto, identifica-se uma clara lacuna: a ferramenta inovadora **SESAME**, com sua abordagem única baseada em separadores sintáticos, foi avaliada em seu trabalho original antes da consolidação desta metodologia de avaliação mais sofisticada. Portanto, seu desempenho em termos de correção, sua taxa de geração de *merges* incorretos e sua posição em comparação com o estado da arte (ferramentas como Spork, IVn, etc.) sob esta ótica mais rigorosa, são desconhecidas. Este trabalho preenche essa lacuna ao executar o protocolo de Schesch *et al.* (2024) sobre o SESAME, permitindo uma comparação direta de *Effort Reduction* com as ferramentas estado da arte.

3 METODOLOGIA

3.1 Desenho do Estudo

Este estudo caracteriza-se como uma pesquisa experimental quantitativa. O desenho experimental segue o protocolo de replicação diferenciada, reutilizando a infraestrutura de avaliação, o conjunto de dados e as métricas propostas por Schesch *et al.* (2024). A variável independente introduzida é a ferramenta de *merge* SESAME, cujo desempenho será contrastado com as ferramentas de referência (*baselines*) do estudo original: Git *Merge* (abordagem textual) e Spork (abordagem estruturada).

3.2 Seleção e Preparação do Conjunto de Dados

Para garantir a representatividade dos resultados, utiliza-se o conjunto de dados curado por Schesch *et al.* (2024), derivado de repositórios Java de alta qualidade listados nos *datasets* "*GitHub's Greatest Hits*" e "*Reaper*".

O processo de filtragem dos cenários de *merge* obedeceu aos seguintes critérios rigorosos para isolar a responsabilidade da ferramenta:

1. **Relevância:** Apenas projetos Java que utilizam sistemas de automação de *build* Maven ou Gradle.

2. **Sanidade dos Pais:** Foram selecionados apenas cenários onde ambos os *commits* pais (as duas versões a sofrerem o *merge*) compilam e passam em todos os testes existentes. Isso assegura que qualquer falha detectada após o *merge* foi causada exclusivamente pelo processo de *merge* e não por defeitos pré-existentes.

3. **Diversidade de Fontes:** O *dataset* inclui tanto *merges* do ramo principal (main branch) quanto de ramos secundários (*feature branches*), capturando a complexidade real do fluxo de desenvolvimento, conforme recomendado por Schesch *et al.* (2024).

3.3 Integração da Ferramenta SESAME

A infraestrutura original de Schesch foi projetada para ser extensível, exigindo que novas ferramentas sejam integradas via shell *scripts* padronizados. Para permitir a avaliação do SESAME, foram desenvolvidos dois artefatos de integração distintos, seguindo as diretrizes do *framework* original:

1. **Integração Básica (sesame.sh):** Foi desenvolvido um *script wrapper* que configura o SESAME como um *merge* driver nativo do Git. A implementação completa deste *script* encontra-se no **Apêndice A**. Ele automatiza a invocação do executável sesame.jar utilizando a estratégia de separadores (-tms csdiff), garantindo que os parâmetros de revisão base, local e remota sejam passados corretamente.

2. **Integração Híbrida (sesame_plus.sh):** Para uma comparação justa com os resultados de Schesch *et al.* (2024), que avaliaram ferramentas em conjunto com utilitários de limpeza, foi criado um segundo *script* denominado *sesame_plus.sh* (disponível no **Apêndice B**). Este artefato orquestra a execução sequencial do SESAME seguida pela ferramenta Plume-lib *Merging*, reutilizando a lógica de integração da infraestrutura original para garantir consistência com os *baselines*.

3.4 Procedimento Experimental e Execução de Testes

A execução do experimento seguiu a infraestrutura de orquestração disponibilizada pelo estudo original. O processo não se resume a uma única etapa de processamento, mas constitui um fluxo de trabalho composto por sete estágios sequenciais. Esse *pipeline* foi mantido integralmente nesta replicação para assegurar a comparabilidade dos dados, o isolamento de falhas e a fidelidade à metodologia de referência. Abaixo, descreve-se o fluxo de dados e os módulos de *script* responsáveis por cada fase:

1. **Inicialização e Rastreabilidade**: A etapa inicial congela o estado dos repositórios analisados. O *script* registra os *hashes* exatos dos *commits* (HEADs) de cada projeto no momento da execução. Esse procedimento assegura a rastreabilidade total do experimento, garantindo que qualquer tentativa futura de replicação utilize exatamente as mesmas versões do código-fonte, eliminando divergências causadas por atualizações nos repositórios remotos.

2. **Validação de Sanidade**: Para mitigar ameaças à validade interna, o *pipeline* realiza uma pré-validação dos pais do *merge*. Antes de tentar qualquer fusão, o sistema executa a suíte de testes nos *commits* originais (*Left* e *Right*). Apenas os cenários onde os pais passam nos testes (*Tests_passed*) são mantidos. Repositórios instáveis ou com falhas pré-existentes são descartados nesta etapa, garantindo que falhas observadas posteriormente sejam causadas exclusivamente pelo processo de *merge* e não por defeitos anteriores.

3. **Mineração e Amostragem**: O componente Java `FindMergeCommits` varre o histórico do Git para identificar cenários de *merge* candidatos. Na sequência, o módulo de amostragem aplica critérios de filtragem rigorosos (como a exclusão de *merges* triviais ou sem alterações de código). Com base no parâmetro de configuração n_merges, o sistema seleciona a amostra final — seja o *dataset* completo ou um subconjunto estatisticamente representativo.

4. **Análise Estática Prévia**: Antes da execução das ferramentas, realiza-se uma caracterização dos cenários selecionados. Este módulo coleta métricas sobre a complexidade dos conflitos, número de arquivos envolvidos e linhas de código impactadas, preparando o terreno para correlacionar essas características com o sucesso ou falha das ferramentas posteriormente.

5. **Execução e Teste de Corretude**: Este é o núcleo do processamento experimental. O *script* itera sobre cada cenário de *merge*, isola o ambiente e invoca a ferramenta alvo (SESAME, Spork ou Git, por exemplo). Após a tentativa de fusão, o sistema tenta compilar e executar a bateria de testes do projeto resultante. O estado final é capturado e classificado em categorias discretas: Sucesso, Falha de Compilação, Falha de Teste ou Conflito Não Resolvido.

6. **Medição de Desempenho**: Executada sobre o *dataset* de controle, esta etapa foca na coleta de métricas temporais. O *pipeline* reexecuta os *merges* instrumentando tanto o tempo de CPU quanto o tempo total (*wall-clock*). Para assegurar precisão estatística e mitigar ruídos causados pelo "aquecimento" da JVM (*warm-up*) ou *cache* de disco, cada cenário é executado múltiplas vezes, calculando-se a média aritmética dos tempos válidos.

7. **Consolidação de Resultados**: A etapa final agrega os *logs* brutos e arquivos CSV gerados por todas as fases anteriores. O módulo processa estatisticamente os dados, calculando taxas de acerto (*Recall*) e médias de tempo, e gera automaticamente os artefatos visuais (tabelas e gráficos em formato LaTeX/PGF) apresentados neste trabalho, eliminando erros de transcrição manual.

Essa abordagem em estágios garante que a avaliação dos algoritmos de *merge* ocorra sobre dados limpos e verificados, isolando variáveis de confusão e assegurando a robustez das conclusões apresentadas.

3.4.1 Protocolo de Invocação das Ferramentas

Para cada cenário de *merge* processado pelo estágio de execução (passo 5 acima), o protocolo de invocação das variantes (SESAME e SESAME+) obedece a um fluxo rigoroso de verificação:

1. ***Checkout* e Preparação:** O *framework* realiza o *checkout* forçado da versão local (*branch1*) e configura o estilo de conflito do Git para `diff3`, garantindo que os marcadores de conflito contenham a informação do ancestral comum.
2. **Execução do *Merge*:** O *script wrapper* correspondente (`sesame.sh` ou `sesame_plus.sh`) é invocado. No caso do **SESAME+**, implementou-se uma lógica de contingência: se o SESAME padrão deixar conflitos não resolvidos, a biblioteca *Plume-lib* é acionada imediatamente com a estratégia `merge-plumelib` na tentativa de resolver as seções restantes automaticamente.
3. **Verificação de Sucesso:** O sistema verifica o código de retorno da ferramenta. Se for diferente de 0 ou se o Git ainda reportar arquivos com marcadores de conflito (`diff-filter=U`), o *merge* é classificado como **Não Tratado (*Unhandled*)**.
4. **Validação de Corretude:** Apenas se o *merge* for reportado como limpo (código de retorno 0), o fluxo avança para a fase de compilação e execução da suíte de testes.

3.5 Métricas de Avaliação

A classificação dos resultados baseia-se na integridade semântica do código, e não apenas na textual:

- ***Merge* Correto (*Correct*):** *Merge* limpo e todos os testes passam.
- ***Merge* Incorreto (*Incorrect*):** *Merge* limpo, mas ocorre falha na compilação ou nos testes.
- ***Merge* Não Tratado (*Unhandled*):** A ferramenta reporta conflito.

Métrica de Redução de Esforço (*Effort Reduction*): Para comparar quantitativamente as ferramentas, utiliza-se a métrica de *Effort Reduction* proposta por Schesch *et al.* (2024). Esta métrica penaliza desproporcionalmente os *merges* incorretos, assumindo que corrigir um erro silencioso é mais custoso do que resolver um conflito explícito.

A fórmula utilizada é:

$$EffortReduction = 1 - \frac{Unhandled + (Incorrect \times k)}{NumMerges} \quad (1)$$

Onde:

- *k* é o fator de custo relativo de um *merge* incorreto. Conforme a análise de sensibilidade de Schesch, o valor de *k* varia (e.g., de 1 a 20) para demonstrar como a penalidade afeta o ranking das ferramentas.

3.6 Disponibilidade de Dados e Materiais

A fim de garantir a transparência e a reprodutibilidade deste estudo, todo o código-fonte, *scripts* de orquestração e os dados brutos gerados durante os experimentos foram disponibilizados publicamente.

É importante ressaltar que, embora este documento discuta os resultados focados nas ferramentas **SESAME, Spork, IntelliMerge e Git *Merge*** para fins de síntese e foco analítico, o repositório contém a execução estendida englobando todas as ferramentas avaliadas na infraestrutura original.

Nota sobre Execução Reduzida: Adicionalmente, disponibiliza-se no mesmo repositório um *dataset* de "Execução Rápida" (*Shortcut*), contendo métricas coletadas sobre uma amostragem de 10 repositórios distintos (totalizando 100 *commits* de *merge*). Este subconjunto serve como referência para pesquisadores que desejam validar o ambiente de execução rapidamente sem a necessidade de processar a carga de trabalho massiva completa.

O pacote de replicação completo encontra-se referenciado sob o identificador de Almeida (2025) e pode ser acessado diretamente através do endereço:

`https://github.com/davidalmd/AST-Merging-Evaluation-TCC`

4 RESULTADOS

Nesta sessão, apresentam-se os dados obtidos a partir da execução experimental da ferramenta SESAME, contrastados com as ferramentas de referência (*baselines*) definidas no estudo original de Schesch *et al.* (2024) e o estado da arte em *merge* estruturado.

4.1 Visão Geral dos Dados de Corretude

A Tabela 1 sumariza o desempenho bruto de cada ferramenta sobre o conjunto de dados processados. A partir de um universo inicial de 69.119 candidatos minerados, aplicaram-se filtros de relevância (arquivos Java modificados) e sanidade (pais aprovados nos testes), resultando em um *dataset* final composto por **5.365** cenários de *merge* amostrados para execução. Observa-se que o *dataset* desafiou significativamente todas as abordagens, com nenhuma ferramenta atingindo 60% de taxa de sucesso sem intervenção manual.

4.2 Análise de Redução de Conflitos

A capacidade de resolver conflitos automaticamente varia drasticamente entre os paradigmas. O Git *Merge* (ort) falhou em resolver **51%** dos cenários (2682 casos).

As ferramentas puramente estruturadas, como o Spork, demonstraram a maior agressividade na resolução, reduzindo a taxa de conflitos não tratados para **34%**. Contudo, essa agressividade tem um custo na corretude, conforme discutido a seguir.

Tabela 1 – Desempenho comparativo das ferramentas de *merge*. Dados extraídos da execução experimental. Os valores indicam o número absoluto e a porcentagem em relação ao total de cenários.

Ferramenta	**Corretos**		**Incorretos**		**Não Tratados (Conflitos)**	
	#	%	#	%	#	%
Git *Merge* (ort)	2450	47%	92	2%	2682	51%
Spork	2889	55%	558	11%	1777	34%
IntelliMerge	446	9%	3659	70%	1119	21%
SESAME	**2461**	**47%**	**174**	**3%**	**2589**	**50%**
SESAME+	**2472**	**47%**	**173**	**3%**	**2579**	**49%**

O SESAME, posicionando-se como uma abordagem semi-estruturada, obteve uma taxa de conflitos não tratados de **50%** (2589 casos). Embora a redução numérica direta em relação ao Git seja modesta (aproximadamente 3,5% de redução no número absoluto de conflitos), o SESAME mantém a segurança próxima à do paradigma textual, evitando as distorções estruturais observadas em ferramentas mais complexas.

4.3 Análise de Corretude e Erros Silenciosos

A métrica mais crítica para a adoção industrial, como destacado por Schesch *et al.* (2024), é a taxa de *merges* incorretos — cenários onde a ferramenta reporta sucesso, mas o código resultante quebra (falha de compilação ou teste).

- **O Risco Estruturado:** O Spork, apesar de resolver mais conflitos, introduziu erros em **11%** dos casos (558 falhas). O IntelliMerge mostrou-se inviável para este *dataset*, com uma taxa de erro de **70%**, comportando-se como um *outlier* negativo.
- **A Segurança Semi-Estruturada:** O SESAME apresentou uma taxa de erro de apenas **3%** (174 casos), um valor muito próximo ao do Git (2%).

Isso indica que o SESAME consegue aplicar inteligência estrutural (via separadores) sem assumir os riscos de "alucinação estrutural" que ferramentas baseadas em AST completa (como o Spork) tendem a cometer quando a árvore sintática não pode ser perfeitamente reconciliada.

4.4 Redução de Esforço (*Effort Reduction*)

Para ponderar o compromisso entre resolver conflitos e evitar erros, a métrica de *Effort Reduction* penaliza as ferramentas conforme o custo k de corrigir um erro introduzido aumenta.

A Figura 6 (baseada no arquivo `cost_without_manual.pdf` gerado pelo experimento) ilustra este cenário:

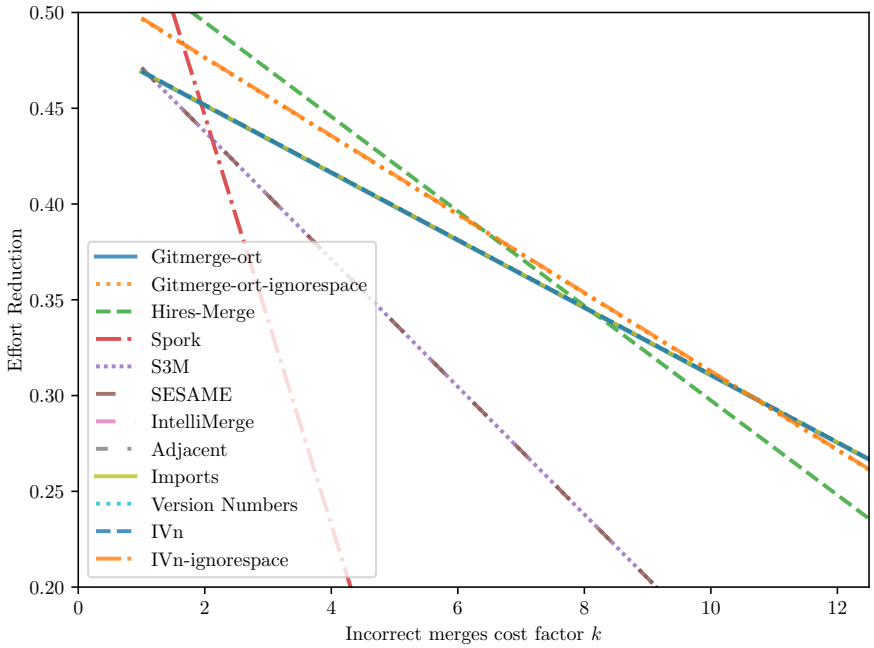

Figura 6 – Curva de Redução de Esforço (*Effort Reduction*). O eixo X representa o fator de custo k para erros incorretos. Fonte: Dados experimentais.

Observa-se que:

- Para k=0 (custo de corrigir erro igual ao de resolver conflito), o Spork lidera devido à sua alta taxa de resolução bruta.
- No entanto, a curva do Spork apresenta um declínio acentuado. À medida que k aumenta (penalizando a introdução de bugs), a vantagem do Spork desaparece rapidamente devido aos seus 11% de erros.
- A curva do SESAME mantém-se muito mais estável e próxima à do Git *Merge*-ort, demonstrando que a ferramenta é “segura”. Ela oferece uma alternativa conservadora que não impõe passivos técnicos ocultos ao desenvolvedor.

Esta análise sugere que o SESAME é uma alternativa viável para equipes que priorizam a redução de conflitos manuais, desde que possuam uma suíte de testes robusta para capturar os eventuais erros de integração introduzidos.

4.5 Análise de Tempo de Execução

Além da corretude, a viabilidade de uma ferramenta em ambientes de Integração Contínua (CI) depende do seu custo temporal. A Tabela 2 apresenta os tempos de execução obtidos no experimento em larga escala.

Tabela 2 – Tempo de execução (em segundos) por cenário de *merge*. Fonte: Dados experimentais.

Ferramenta	Média (s)	Mediana (s)	Máximo (s)
Git *Merge*-ort	0.05	0.04	1.48
SESAME	**0.97**	**0.45**	**91.5**
Spork	3.12	1.73	127.0
IntelliMerge	1.81	0.93	114.0

Observa-se que o custo da estruturação completa cobra seu preço. O **Spork**, com média de 3.12s, mostrou-se aproximadamente **62 vezes mais lento** que o *baseline* textual (Git *Merge*-ort, 0.05s).

Em contrapartida, o **SESAME** (0.97s) demonstrou ser cerca de **3 vezes mais rápido que o Spork**. Embora ainda adicione um *overhead* em relação ao Git (sendo cerca de 19x mais lento), ele se mantém abaixo da barreira psicológica de 1 segundo na média, validando a hipótese de que a inferência leve de estrutura via separadores é significativamente menos custosa do que a construção de ASTs completas para todo o projeto.

Isso demonstra a eficiência da abordagem baseada em separadores, que evita o custo computacional da construção de ASTs completas, conforme teorizado por Cavalcanti, Borba *et al.* (2024).

5 DISCUSSÃO

Nesta sessão, interpretam-se os resultados obtidos à luz das questões de pesquisa formuladas sob a ótica da viabilidade industrial e do equilíbrio entre eficiência e eficácia, discutindo as implicações do desempenho do SESAME frente às ferramentas de referência. Adicionalmente, abordam-se as ameaças à validade inerentes ao desenho experimental de replicação.

5.1 Análise Detalhada de Desempenho

A análise dos dados revela nuances importantes sobre o compromisso (*trade-off*) entre a redução de conflitos e a garantia de corretude semântica. Antes de consolidar a viabilidade das ferramentas, é necessário dissecar o comportamento delas em relação às duas métricas primárias de qualidade.

5.1.1 Eficácia na Redução de Conflitos

Os resultados indicam uma distinção clara entre as capacidades dos paradigmas. O Spork (estruturado) demonstrou a maior capacidade bruta de resolução, deixando apenas 34% dos cenários como “Não Tratados” , uma redução significativa em comparação aos 51% do Git. Isso confirma que a construção da AST completa permite resolver conflitos complexos que ferramentas mais simples não conseguem interpretar.

Por outro lado, o SESAME (semi-estruturado) apresentou uma taxa de conflitos não tratados de 50%, um desempenho estatisticamente próximo ao do Git. Isso sugere que a estratégia de separadores, embora segura, possui um teto de resolução mais baixo que a análise estruturada profunda. O SESAME resolve conflitos de formatação e ordenação simples (como confirmado por Cavalcanti, Borba *et al.* (2024)), mas "devolve" para o usuário a decisão em casos de alta complexidade sintática, onde o Spork tentaria (e muitas vezes conseguiria) intervir.

5.1.2 Impacto na Corretude e Robustez do Software

É na análise de corretude que a vantagem da abordagem conservadora do SESAME se materializa. Um dos pontos centrais da metodologia de Schesch *et al.* (2024) é a penalização de ferramentas que realizam o *merge* sem conflitos, mas geram código "quebrado".

Os dados revelam que a agressividade do Spork cobra um preço alto: a ferramenta introduziu erros silenciosos em 11% dos casos (558 cenários), uma taxa preocupante para ambientes de produção. O IntelliMerge, por sua vez, mostrou-se instável neste *dataset*, com 70% de erros.

Em contraste, o SESAME manteve uma taxa de erro de apenas 3% (174 casos) , marginalmente superior aos 2% do Git. Isso demonstra que, ao evitar a reconstrução complexa da árvore sintática, o SESAME mitiga o risco de "alucinações estruturais" - onde a ferramenta monta um código sintaticamente válido, mas semanticamente incorreto. O SESAME provou ser "seguro por padrão", alinhando-se à filosofia de que é preferível solicitar intervenção manual (conflito) do que introduzir um defeito silencioso no sistema.

5.2 O *Sweet Spot* da Abordagem Semi-Estruturada

Os dados corroboram a tese de que existe um ponto de equilíbrio (*sweet spot*) entre o tratamento textual puro e a análise estruturada pesada. Enquanto o **Spork** maximiza a resolução de conflitos (55% de sucesso) ao custo de alta latência (3.12s) e maior risco de quebra de código (11% de erro), e o **Git** maximiza a velocidade (0.05s) e segurança ao custo de baixa resolução, o **SESAME** oferece um caminho intermediário viável.

Com um tempo de execução de 0.97s e uma taxa de erro controlada (3%), o SESAME prova ser uma extensão segura ao fluxo de trabalho tradicional. A análise do *Heatmap* (Figura 7) reforça essa conclusão: o SESAME apresenta alta concordância com as decisões do Git (cores claras/azuladas nas interseções), enquanto ferramentas como IntelliMerge divergem agressivamente (comportamento de *outlier*), o que explica sua alta taxa de falhas.

5.3 Viabilidade para Integração Contínua

A diferença de ordem de grandeza no tempo de execução (Tabela 2) é o fator determinante para a adoção. Ferramentas que exigem a construção completa da AST para

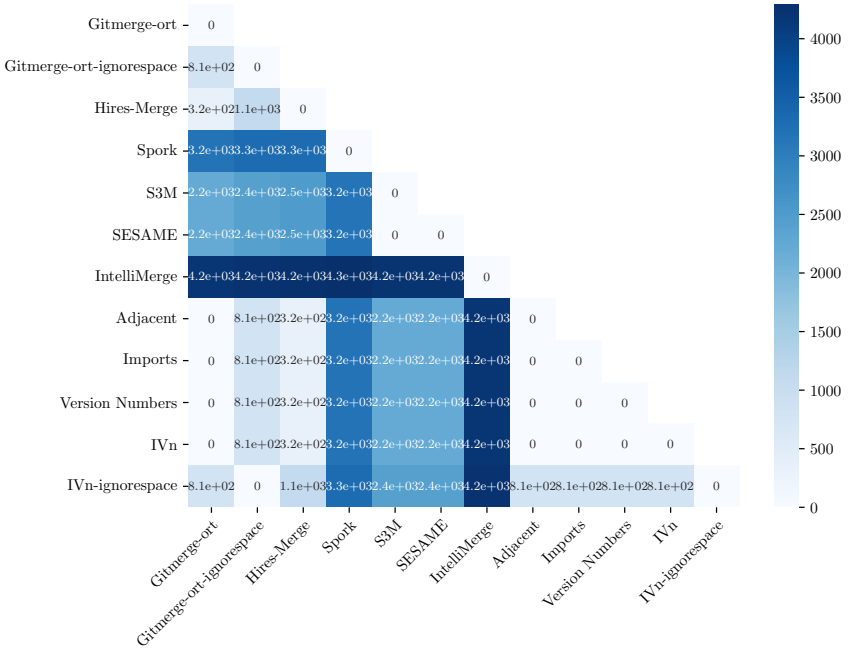

Figura 7 – Mapa de calor de divergência de resultados (*HeatMap*). As células representam o número de cenários onde o resultado final diferiu entre as ferramentas comparadas.

todo o projeto (como IntelliMerge e Spork) sofrem com problemas de escalabilidade. O SESAME, ao focar apenas nos nós conflitantes e usar separadores leves, mantém a performance dentro de limites aceitáveis para *pipelines* de integração contínua ágeis, onde o *feedback* rápido é essencial.

Em suma, embora o Spork seja teoricamente mais poderoso, o custo operacional e o risco de erros silenciosos tornam o SESAME uma alternativa pragmática superior para equipes que buscam reduzir conflitos manuais sem sacrificar a velocidade de desenvolvimento ou a estabilidade do *build*.

5.4 Desafios Práticos da Execução Experimental

A replicação do estudo de Schesch *et al.* (2024) revelou barreiras de entrada significativas para pesquisadores que desejam avaliar novas ferramentas de *merge*. Para além dos resultados numéricos, é fundamental documentar as dificuldades técnicas enfrentadas, pois elas impactam diretamente a viabilidade de adoção dessas ferramentas em fluxos de integração contínua (CI/CD) reais.

Dois obstáculos principais foram identificados durante a execução do experimento:

5.4.1 Custo Computacional e Tempo de Execução

A inclusão de ferramentas estruturadas (*Spork* e *IntelliMerge*) no *benchmark* alterou drasticamente a escala temporal do experimento.

Enquanto a avaliação de ferramentas puramente textuais (como o Git *Merge*) ou semi-estruturadas leves (como o SESAME) permite o processamento de milhares de cenários em questão de minutos ou poucas horas, as ferramentas estruturadas exigem a construção completa da Árvore Sintática Abstrata (AST) para cada revisão do código (Base, *Left* e *Right*).

Esse *overhead* de processamento transformou a execução do experimento em uma tarefa de múltiplas semanas. No contexto de pesquisa, isso impactou severamente a produtividade, impedindo iterações rápidas (ciclos de *feedback*) e testes de configuração. Extrapolando para a indústria, esse custo temporal sugere que o uso síncrono dessas ferramentas em esteiras de Integração Contínua (CI/CD) pode ser proibitivo para grandes repositórios sem uma estratégia robusta de *cache* ou paralelismo.

5.4.2 Opacidade de Progresso e Arquitetura de Baixa Observabilidade

Uma limitação crítica observada na infraestrutura de avaliação disponibilizada é a ausência de geração de resultados parciais. O *framework* opera sob uma lógica monolítica: os dados brutos são processados sequencialmente, mas a consolidação das métricas (tabelas) e a geração visual (gráficos) ocorrem exclusivamente no estágio final do *script*.

Essa arquitetura resulta em uma **opacidade operacional** durante o longo período de processamento. Não há *checkpoints* visuais que permitam validar se a configuração está correta após as primeiras horas, fazendo com que o pesquisar perca a visibilidade sobre a integridade dos dados durante a execução. O risco inerente a este modelo é a **detecção tardia de falhas:** erros de configuração ou exceções no módulo de plotagem podem ocorrer após dias de computação, exigindo intervenções manuais complexas para recuperar os dados armazenados em *cache* e evitar a perda total do tempo de processamento investido.

A falta de *feedback* incremental não apenas dificulta a detecção precoce de erros de configuração, mas também impõe uma barreira psicológica e técnica para a manutenção e evolução do próprio *framework* de avaliação.

5.5 Ameaças à Validade

Como em qualquer estudo empírico, existem limitações que devem ser consideradas na interpretação destes resultados:

- **Validade Interna:** A execução dos testes automatizados pode sofrer com o problema de testes intermitentes (***flaky tests***). Embora tenha-se adotado a estratégia de reexecução (5 vezes) proposta por Schesch *et al.* (2024), não é possível garantir a eliminação total de falsos negativos.
- **Validade Externa:** O estudo limitou-se a projetos Java que utilizam Maven/Gradle. Embora o *dataset* seja representativo de projetos de alta qualidade, os resultados podem não ser generalizáveis para outras linguagens ou paradigmas de programação onde os separadores sintáticos tenham papéis diferentes.

- **Validade de Construto:** Utilizou-se a passagem em suítes de teste como *proxy* para corretude. É possível que *merges* classificados como “Corretos” contenham defeitos não cobertos pelos testes existentes, uma limitação intrínseca a este tipo de avaliação automatizada.

6 TRABALHOS RELACIONADAS

A avaliação de ferramentas de *merge* é uma área ativa de pesquisa. Esta sessão situa o presente trabalho em relação a estudos anteriores de avaliação e a outras ferramentas propostas na literatura.

6.1 Estudos de Avaliação de Ferramentas

Historicamente, a avaliação de ferramentas de *merge* focou-se predominantemente na contagem de conflitos. Cavalcanti, Accioly e Borba (2015), por exemplo, replicaram o estudo de Apel *et al.* (2011) para avaliar o *FSTMerge*, utilizando a redução de conflitos textuais como métrica principal em um *dataset* expandido.

Diferentemente dessas abordagens, Schesch *et al.* (2024) introduziram um paradigma focado na corretude via execução de testes, argumentando que a ausência de conflitos não implica sucesso da integração. O presente trabalho alinha-se a esta vertente mais rigorosa, estendendo a análise para incluir a abordagem inovadora do SESAME.

6.2 Ferramentas de *Merge* Alternativas

Além das ferramentas avaliadas neste estudo (Git, Spork, IntelliMerge, SESAME, etc.), a literatura apresenta diversas outras propostas:

- **JDime:** Uma das primeiras ferramentas de *merge* estruturado para Java, que alterna entre *merge* não estruturado e estruturado (*auto-tuning*). No entanto, estudos anteriores indicaram problemas de desempenho e complexidade.
- **FSTMerge:** Uma ferramenta pioneira em *merge* semi-estruturado proposta por Apel *et al.* (2011). Diferente do SESAME, que usa separadores, o FSTMerge depende de uma gramática anotada para construir árvores parciais.
- **Abordagens baseadas em Aprendizado de Máquina:** Ferramentas recentes como DeepMerge e MergeBERT tentam resolver conflitos aprendendo padrões de resolução a partir de históricos de repositórios, mas ainda enfrentam desafios de disponibilidade e generalização.

O SESAME diferencia-se destas ferramentas ao propor uma inferência de estrutura leve baseada em separadores, buscando um ponto ótimo entre a precisão das ferramentas estruturadas e o desempenho das textuais.

7 CONCLUSÃO

Este trabalho apresentou uma avaliação empírica da ferramenta de *merge* semi-estruturado SESAME, conduzida através de uma replicação diferenciada do estudo de Schesch *et al.* (2024). O objetivo principal foi determinar se a abordagem baseada em separadores sintáticos oferece um compromisso vantajoso entre a precisão das ferramentas estruturadas e a eficiência das textuais.

7.1 Considerações Finais

Os resultados evidenciam que o SESAME ocupa um "*sweet spot*" no espectro de ferramentas de integração. Embora ferramentas puramente estruturadas (como o Spork) apresentem taxas superiores de resolução de conflitos, elas o fazem ao custo de uma latência proibitiva (em média 3 vezes maior que o SESAME) e uma taxa elevada de erros semânticos silenciosos (11%).

Em contraste, o SESAME demonstrou ser uma alternativa segura e eficiente. A ferramenta foi capaz de resolver conflitos de ordenação e formatação inatingíveis pelo Git, mantendo uma taxa de corretude estatisticamente equivalente à abordagem textual padrão. A análise de *Effort Reduction* confirma que, para cenários industriais onde a estabilidade do *build* e a velocidade de *feedback* são prioritárias, a abordagem semi-estruturada oferece o melhor retorno sobre o investimento, evitando a complexidade computacional excessiva e os riscos de integridade associados à construção de ASTs completas.

Conclui-se que a inferência de estrutura via separadores é uma adição viável ao ecossistema de Integração Contínua, provendo inteligência sintática leve sem sacrificar o desempenho do fluxo de desenvolvimento.

7.2 Trabalhos Futuros

Como direções para pesquisas futuras, sugere-se:

- **Generalização da Abordagem:** Avaliar o SESAME em linguagens com características sintáticas distintas (como Python, baseada em indentação, ou C#) para verificar a robustez da estratégia de separadores.
- **Refinamento de Algoritmos:** Investigar os casos residuais de *merges* incorretos (os 3% observados) para aprimorar os algoritmos de realinhamento textual pós-processamento.
- **Evolução da Infraestrutura de Avaliação:** Refatorar o *framework* de experimentação utilizado neste estudo para suportar processamento incremental e relatórios parciais, mitigando a arquitetura de baixa observabilidade identificada durante a replicação.
- **Validação Industrial:** Expandir o *dataset* para incluir projetos proprietários de larga escala, verificando se os padrões de conflito em ambientes corporativos diferem dos encontrados em projetos *open-source*.

AGRADECIMENTOS

Agradeço, primeiramente, a Deus, por ter sido meu sustento e fortaleza diante das dificuldades e desafios desta caminhada. À minha família, expresso minha profunda gratidão pelo apoio incondicional. Obrigado por não me deixarem conviver com a solidão; desde a minha saída de casa até esta conclusão, provaram que a distância não passou de um detalhe físico. Estendo esse agradecimento aos meus amigos, a quem também atribuo o mérito de não terem permitido que a solidão fizesse parte dos meus dias. Sou imensamente feliz pelas pessoas incríveis que foram colocadas em minha vida e grato pelo privilégio da convivência ao longo destes anos. Que este encerramento de ciclo seja apenas o início de amizades para a vida toda. Por fim, agradeço a todos os professores que contribuíram para a minha formação e, de modo especial, ao meu orientador, Prof. Dr. Guilherme Cavalcanti, pelos conhecimentos partilhados e pela valiosa orientação na condução deste trabalho.

APÊNDICES

A *SCRIPT* DE INTEGRAÇÃO BÁSICA (SESAME.SH)

Este *script* configura o SESAME como um driver de *merge* no Git e executa a ferramenta com a estratégia de diferenciação baseada em separadores (`csdiff`).

Programa 1 – *Wrapper* para execução do SESAME (sesame.sh)

```
#!/usr/bin/env sh

# usage: <scriptname> [--verbose] <clone_dir> <branch-1> <branch-2>
# <clone_dir> must contain a clone of a repository.
# Merges branch2 into branch1, in <clone_dir>.
# Return code is 0 for merge success, 1 for merge failure, 2 for script failure.
# For merge failure, also outputs "Conflict" and aborts the merge.

set -o nounset

verbose=
if [ "$1" = "--verbose" ] ; then
  verbose="$1"
  shift
fi

if [ "$#" -ne 3 ]; then
  echo "Usage: $0 [--verbose] CLONE_DIR BRANCH1 BRANCH2" >&2
  exit 2
fi

clone_dir=$1
branch1=$2
branch2=$3

SCRIPT_PATH="$(dirname "$0")"; SCRIPT_PATH="$(eval "cd \"$SCRIPT_PATH\" && pwd")"
ROOT_PATH="$(realpath "${SCRIPT_PATH}/../../../")"
JAR_RELATIVE_PATH="jars/sesame.jar"
```

```
JAR_ABSOLUTE_PATH="${ROOT_PATH}/${JAR_RELATIVE_PATH}"

TOOL_NAME="sesame"

TOOL_COMMAND="java -jar ${JAR_ABSOLUTE_PATH} %A %O %B -o %A -c false -l
    false -tms csdiff"

cd "$clone_dir" || { echo "$0: cannot cd to $clone_dir"; exit 2; }

git config --local merge.${TOOL_NAME}.name "${TOOL_NAME} merge driver"
git config --local merge.${TOOL_NAME}.driver "${TOOL_COMMAND}"

echo "*.java merge=${TOOL_NAME}" >> .gitattributes

git checkout "$branch1" --force
git merge --no-edit "$branch2"
retVal=$?

if [ $retVal -ne 0 ]; then
    echo "${TOOL_NAME}.sh: Conflict"
fi

exit $retVal
```

B *SCRIPT* DE INTEGRAÇÃO HÍBRIDA (SESAME_PLUS.SH)

Este *script* implementa a estratégia composta, executando o SESAME e, subsequentemente, acionando a ferramenta Plume-lib para refinamento do resultado.

Programa 2 – Orquestrador Híbrido (sesame_plus.sh)

```
#!/usr/bin/env sh

# usage: <scriptname> [--verbose] <clone_dir> <branch-1> <branch-2>

MERGE_SCRIPTS_DIR="$(cd "$(dirname "$0")" && pwd -P)"

if [ "$1" = "--verbose" ] ; then
  shift
fi

clone_dir=$1
branch1=$2
branch2=$3

merge_script="sesame.sh"
plumelib_strategy=""

"$MERGE_SCRIPTS_DIR"/merge_script_then_plumelib.sh "$clone_dir" "$branch1"
    "$branch2" "$merge_script" "$plumelib_strategy"
```

REFERÊNCIAS

ALMEIDA, David Lucas Alves de. **AST-Merging-Evaluation-TCC**. [*S. l.*]: GitHub, 2025. https://github.com/davidalmd/AST-Merging-Evaluation-TCC.

APEL, Sven *et al.* Semistructured Merge: Rethinking Merge in Revision Control Systems. *In:* PROCEEDINGS of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11). Szeged, Hungary: ACM, 2011. p. 43–53. DOI: 10.1145/2025113.2025119.

CARVER, Jeffrey C *et al.* Replications of software engineering experiments. **Empirical Software Engineering**, Springer, v. 19, p. 267–276, 2014. DOI: 10.1007/s10664-013-9290-8.

CAVALCANTI, Guilherme; ACCIOLY, Paola; BORBA, Paulo. Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment. *In:* 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). [*S. l.: s. n.*], 2015. p. 1–10. DOI: 10.1109/ESEM.2015.7321191.

CAVALCANTI, Guilherme; BORBA, Paulo *et al.* Semistructured Merge with Language-Specific Syntactic Separators. *In:* PROCEEDINGS of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24). Sacramento, CA, USA: ACM, 2024. ISBN 979-8-4007-1248-7. DOI: 10.1145/3691620.3695483.

KHANNA, Sanjeev; KUNAL, Keshav; PIERCE, Benjamin C. A formal investigation of diff3. *In:* PROCEEDINGS of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'07. [*S. l.*]: Springer-Verlag, 2007. p. 485–499. DOI: 10.1007/978-3-540-77050-3_37.

MÄNTYLÄ, Mika V; LASSENIUS, Casper; VANHANEN, Jari. Rethinking replication in software engineering: Can we see the forest for the trees? *In:* IEEE. PROCEEDINGS of the 1st International Workshop on Replication in Empirical Software Engineering Research (RESER). [*S. l.: s. n.*], 2010. v. 2, p. 1–10.

MENS, Tom. A state-of-the-art survey on software merging. **IEEE Transactions on Software Engineering**, v. 28, n. 5, p. 449–462, 2002. DOI: 10.1109/TSE.2002.1000453.

SCHESCH, Benedikt *et al.* Evaluation of Version Control Merge Tools. *In:* PROCEEDINGS of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24). Sacramento, CA, USA: ACM, 2024. ISBN 979-8-4007-1248-7. DOI: 10.1145/3691620.3695075.

SHULL, Forrest J *et al.* The role of replications in empirical software engineering. **Empirical Software Engineering**, Springer, v. 13, n. 2, p. 211–218, 2008. DOI: 10.1007/s10664-008-9060-1.