UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE GRADUAÇÂO EM CIÊNCIA DA COMPUTAÇÃO

LEONARDO DOS ANJOS SILVA

Ferramentas de integração de código: uma comparação entre *merge* estruturado e textual com separadores customizados

Recife

2025

LEONARDO DOS ANJOS SILVA

Ferramentas de integração de código: uma comparação entre *merge* estruturado e textual com separadores customizados

Trabalho de conclusão de curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Ciência da Computação.

Área de Concentração: Engenharia de Software

Orientador (a): Paulo Henrique Monteiro Borba

Recife

2025

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

Silva, Leonardo dos Anjos.
Ferramentas de integração de código: uma comparação entre merge estruturado e textual com separadores customizados / Leonardo dos Anjos Silva. - Recife, 2025.
65 p. : il., tab.

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

1. Engenharia de software. 2. Integração de código. 3. Conflitos de integração. 4. Desenvolvimento de sistemas. 5. Comparação de ferramentas. I. Borba, Paulo Henrique Monteiro. (Orientação). II. Título.

000 CDD (22.ed.)

LEONARDO DOS ANJOS SILVA

Ferramentas de integração de código: uma comparação entre *merge* estruturado e textual com separadores customizados

Trabalho de conclusão de curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Ciência da Computação.

Aprovado em: 11/08/2025

BANCA EXAMINADORA

Prof. Dr. Paulo Henrique Monteiro Borba (Orientador)

Universidade Federal de Pernambuco

Profa. Dra. Paola Rodrigues de Godoy Accioly (Examinadora Interna)

Universidade Federal de Pernambuco

AGRADECIMENTOS

À minha mãe e meu pai, que sempre valorizaram a minha educação e garantiram que não me faltassem recursos para eu correr atrás dos meus objetivos e realizar meus sonhos.

À minha irmã e meu irmão, que me ensinaram e ensinam muita coisa e com quem compartilhei diversos momentos de lazer.

À minha namorada Dayane Lira, por fazer meus dias mais felizes e me entender como ninguém, além de me dar todo o apoio necessário na reta final do curso.

A todos os amigos que fiz durante o curso, pelos momentos de descontração e esforço nos projetos em grupo. Em especial, a Matheus, Elaine, Guilherme e Heitor.

Ao MaratonaCIn, o grupo com o qual eu mais me identifiquei durante a graduação e que me proporcionou muitos aprendizados e momentos inesquecíveis. Em especial, aos meus companheiros de time e grandes amigos Edson e Gabmei, por tornarem o ambiente estressante de um *contest* divertido.

Ao meu orientador Paulo Borba e a Guilherme Cavalcanti pelo auxílio fundamental para a conclusão deste trabalho e durante a minha iniciação científica.

RESUMO

É comum no desenvolvimento de *software* ter desenvolvedores trabalhando em um mesmo projeto em paralelo. Suas mudanças são posteriormente combinadas em uma versão final do código com o auxílio de uma ferramenta de *merge*. Quando os colaboradores modificam um mesmo arquivo, ferramentas de integração podem reportar conflitos. Estes devem ser resolvidos manualmente pelo desenvolvedor que iniciou o processo de *merge*, o que consome tempo e pode levar à introdução de erros à aplicação. Pensando em tornar o processo de integração mais eficiente, diferentes abordagens para ferramentas de *merge* foram propostas, entre elas: *merge* estruturado, em que os códigos são interpretados e é feito um pareamento dos nós das árvores sintáticas das versões integradas; e *merge* textual com separadores customizados, em que o código é particionado por separadores sintáticos da linguagem (como "{", "}" e ";" em Java) e servido como entrada para um *merge* textual tradicional. Outros trabalhos estudaram como essas ferramentas se comparam com a solução padrão amplamente adotada — *merge* textual por linhas —, mas focaram apenas em uma linguagem de programação, como Java ou Python. Também não foi estudado como as abordagens estruturada e textual com separadores customizados se comparam entre si. Este trabalho visa preencher essas lacunas, estudando o comportamento destas soluções em cenários reais de projetos populares desenvolvidos em três linguagens de programação distintas: Go, JavaScript e Rust. Através da análise de métricas de acurácia, busca-se compreender as vantagens e limitações de cada método, contribuindo para discussões sobre a adoção de soluções mais eficazes. Os resultados mostram que o uso de uma abordagem estruturada diminui o número de cenários de integração com conflitos, mas com a desvantagem de deixar mais conflitos passarem despercebidos (falsos negativos), enquanto a ferramenta textual com separadores customizados pode funcionar como uma solução intermediária entre as abordagens puramente textual e estruturada.

Palavras-chaves: integração de código; conflitos de integração; engenharia de software; desenvolvimento de sistemas; comparação de ferramentas.

ABSTRACT

In software development, it is common for two or more developers to work on the same project at the same time. Their changes are later combined in a final version of the code with the help of a merge tool. When the contributors modify the same file, merge tools may report integration conflicts. These must be solved manually by the developer who initiated the merge. This takes time and is prone to introducing errors in the application. Aiming to improve the efficiency of the merge process, different merge approaches were proposed, among them: structured merge, in which code is parsed and nodes of the revisions' syntax trees are matched; and textual merge with custom separators, in which code is split by syntactic separators (such as "{", "}" and ";" in Java) and the result is used as input to a traditional textual merge tool. Other works have studied how such tools compare to the standard well-adopted solution — line-based textual merge —, but have focused on just one programming language, either Java or Python. Also, there haven't been any studies comparing the structured and custom-separators-based merge approaches. This work intends to evaluate how these tools behave in real-world merge scenarios from popular projects developed in three distinct programming languages: Go, JavaScript and Rust. By analysing accuracy metrics, it seeks to comprehend the advantages and limitations of each method, contributing to discussions on the adoption of more effective solutions. The results show that using a structured tool reduces the number of merge scenarios with conflicts, but at the cost of missing some important conflicts (false negatives). Meanwhile, the custom-separators-based textual tool may serve as an intermediate solution between the purely textual and structured approaches.

Keywords: code integration; merge conflicts; software engineering; system development; merge tools comparison.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Desenvolvimento de *software* é uma atividade que, naturalmente, exige muita colaboração (BIRD et al., 2009). Aplicações são normalmente construídas por uma equipe, com desenvolvedores trabalhando em paralelo na mesma base de código. Nesse cenário, é comum que o código-fonte seja mantido em um sistema de controle de versão, como o *git*, com os desenvolvedores fazendo uso de *branches* para fazer mudanças no código de forma independente dos outros colaboradores. Essas alterações são posteriormente combinadas em uma versão final do código, em um processo conhecido como *merge*.

Neste processo, ferramentas como o diff3 criam novas versões dos arquivos modificados, incluindo as mudanças de ambos os desenvolvedores envolvidos. Mas quando parte das alterações acontece na mesma área de código, as ferramentas reportam conflitos de integração. Esses conflitos precisam ser resolvidos manualmente, exigindo um trabalho cognitivo de quem iniciou o processo de *merge* para entender como melhor combinar as modificações.

Muitas vezes, porém, essas ferramentas reportam conflitos que poderiam ser resolvidos automaticamente se as mesmas fossem capazes de entender a sintaxe das linguagens de programação dos arquivos envolvidos no processo (LARSEN et al., 2022). Pensando nisso, surgiram algumas ferramentas de integração de código estruturadas, como JDime (LEßENICH, 2012), Spork (LARSEN et al., 2022), Mergiraf (DELPEUCH, 2025) e Last Merge (DUARTE; BORBA; CAVALCANTI, 2025). Estas são capazes de interpretar o código integrado, tornando possível resolver alguns conflitos sem necessidade de intervenção de desenvolvedores. Além disso, foram desenvolvidas ferramentas semiestruturadas também (APEL et al., 2011; CAVALCANTI; BORBA; ACCIOLY, 2017), que combinam as abordagens estruturada e textual, parando a interpretação do código a partir de um determinado nível hierárquico das árvores sintáticas.

Por outro lado, ferramentas de *merge* estruturadas têm um alto custo de desenvolvimento, uma vez que precisam saber interpretar múltiplas linguagens de programação, e possuem um tempo de execução médio maior em comparação com soluções textuais, como o *git merge* (CAVALCANTI et al., 2024). Por estes motivos, foram propostas ferramentas como o CSDiff (CLEMENTINO; BORBA; CAVALCANTI, 2021) e SepMerge (ARAUJO; BORBA; CAVALCANTI, 2024), de natureza textual, mas que fazem um pré-processamento nos arquivos integrados, particionando o código por separadores sintáticos customizados (como "{", "(", ";" em Java, por exemplo), com o objetivo de simular um *merge* estruturado sem seus pontos negativos.

Este tipo de solução apresenta um novo conjunto de desafios. Ao particionar os códigos, os separadores sintáticos ficam em linhas isoladas, fazendo com que os arquivos finais tenham muitas linhas iguais. Isto prejudica o algoritmo de casamento de linhas do diff3 entre as três versões do arquivo — a original (*base*), a modificada pelo primeiro desenvolvedor (*left*) e a modificada pelo segundo desenvolvedor (*right*) —, que pode erroneamente casar duas linhas iguais que ocorrem em áreas bem distintas do texto, ocasionando mais conflitos (ARAUJO; BORBA; CAVALCANTI, 2024). Além disso, separadores sintáticos podem aparecer no código com outras funções, como parte de um literal do tipo *string*, por exemplo. Nesse caso, não faz sentido particionar este trecho de código em linhas, uma vez que o separador não delimita partes independentes do literal *string*.

1.1 MOTIVAÇÃO

Entender como diferentes tipos de ferramentas de integração de código se comportam em diversos cenários é importante para avaliar a utilidade de tais soluções. Compará-las com a solução mais difundida (*merge* textual por linhas) ajuda a entender se a adoção de uma nova ferramenta poderia contribuir para um processo de integração mais rápido e eficiente, em que os desenvolvedores precisariam resolver conflitos desnecessários com menos frequência e ainda teriam segurança no resultado da ferramenta.

Alguns estudos foram conduzidos envolvendo as abordagens estruturada, semi-estruturada e textual com separadores customizados. Estes reproduzem diversos cenários de integração reais utilizando múltiplas soluções e comparam os resultados coletando métricas como: número total de conflitos reportados e número de cenários com conflitos por ferramenta; e quantidade de falsos positivos (situações em que uma ferramenta reportou conflito e não deveria) e falsos negativos (situações em que uma ferramenta não reportou conflito, mas deveria) adicionais por par ordenado de ferramentas. Além disso, esses estudos realizam uma análise manual sobre alguns cenários para melhor classificar os resultados como erros ou acertos de uma solução (CAVALCANTI et al., 2019; CLEMENTINO; BORBA; CAVALCANTI, 2021; CAVALCANTI et al., 2024; ARAUJO; BORBA; CAVALCANTI, 2024).

Por outro lado, estes experimentos testaram o desempenho das soluções em apenas uma linguagem de programação, como Python ou Java. Avaliar o comportamento das mesmas em outras linguagens é importante para levar em consideração as particularidades de cada uma e entender se as soluções generalizam, ou se podem apresentar resultados significativamente

diferentes por linguagem.

1.2 OBJETIVOS

Apesar de existirem estudos sobre cada uma das abordagens de integração de código mencionadas, nunca foi realizada uma comparação direta entre as soluções estruturada e textual com separadores customizados. Este trabalho tem como objetivo principal comparar estas abordagens entre si e com a solução padrão — *merge* textual por linhas —, visando fornecer métricas sobre a acurácia das ferramentas e compreender o comportamento das mesmas em cenários reais de integração de código.

As soluções serão testadas em três linguagens de programação diferentes e populares — Go, JavaScript e Rust —, com o objetivo de observar as vantagens e limitações das abordagens e identificar possíveis diferenças nos resultados e comportamento das ferramentas por linguagem.

Para este fim, fazem parte dos objetivos do estudo também adicionar suporte a múltiplas linguagens de programação às ferramentas testadas, bem como evoluir o ferramental de suporte para a reprodução dos cenários de integração e comparação dos resultados, permitindo a reprodução do experimento com fidelidade e facilitando sua extensão.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo visa apresentar os conceitos fundamentais para a compreensão do estudo e discutir os trabalhos relacionados, utilizados como ponto de partida e referência para este.

2.1 CONCEITOS FUNDAMENTAIS

Um sistema de controle de versão (VCS) é aquele que permite a manutenção de múltiplas versões de um projeto em um repositório. Os arquivos da aplicação são rastreados pelo VCS e cada alteração em algum deles dá origem a uma nova versão do mesmo. Um desenvolvedor pode alterar múltiplos arquivos e registrar as alterações em novas versões por meio de um *commit*. A lista de *commits* que compõem um projeto monta a história da aplicação (BAUDIš, 2014).

Desenvolvedores podem iniciar suas contribuições a partir de qualquer *commit* do repositório e evoluírem o código de forma independente em paralelo. Isto é feito por um mecanismo conhecido como *branch*. Uma *branch* registra o histórico de *commits* realizados a partir de um *commit* base (de onde o desenvolvedor que a criou iniciou suas modificações). Todo repositório possui uma *branch* principal, normalmente denominada *main* ou *master*.

As contribuições feitas de forma independente por dois desenvolvedores em *branches* distintas podem ser combinadas em um único *commit*, chamado *commit* de *merge*, onde *merge* se refere ao processo de integração de código. Um cenário de integração consiste em uma quádrupla de *commits*: o *commit* de *merge*, resultante da integração das mudanças dos dois desenvolvedores envolvidos no processo; os dois *commits* que deram origem a ele (contendo as mudanças dos colaboradores e aqui chamados de *left* e *right*); e o primeiro *commit* ancestral em comum de *left* e *right*, aqui chamado de *base* (CLEMENTINO; BORBA; CAVALCANTI, 2021).

O processo de *merge* costuma ser semiautomático. Uma ferramenta de *merge* é utilizada para combinar as alterações, mas pode reportar conflitos de integração. Esses conflitos precisam ser resolvidos de forma manual pelo desenvolvedor que iniciou o processo para dar continuidade ao mesmo. Existem diversas ferramentas de *merge* disponíveis, mas a solução padrão amplamente adotada utiliza uma abordagem textual por linhas (LARSEN et al., 2022). Esta abordagem consiste em fazer um casamento ótimo das linhas das três versões de um arquivo (revisões) — a original (*base*), a modificada pelo primeiro desenvolvedor (*left*) e a

base.js

```
const greet = (name) => {
    console.log(`Hello, ${name}`);
}

greet('world');
```

left.js

```
const greet = (name) => {
    console.log(`Hello there, ${name}`);
}

greet('Earth');
```

right.js

```
const GREETING = 'Hi';

const greet = (name) => {
    console.log(`${GREETING}, ${name}`);
}

greet('world');
```

Figura 1 – Exemplo de versões de um arquivo JavaScript com mudanças conflitantes e não conflitantes

diff3.js

```
const GREETING = 'Hi';

const greet = (name) => {
<<<<<<< MINE
    console.log(`Hello there, ${name}`);
=======
    console.log(`${GREETING}, ${name}`);
>>>>>>> YOURS
}

greet('Earth');
```

Figura 2 – Resultado da integração das revisões da Figura 1 pelo diff3

modificada pelo segundo desenvolvedor (*right*) — e construir uma nova versão do arquivo com as contribuições dos dois colaboradores, reportando conflitos se as alterações acontecem na mesma área de código e são diferentes (KHANNA; KUNAL; PIERCE, 2007). Por exemplo, ao integrar as revisões da Figura 1, o diff3, ferramenta popular desta abordagem, reporta um conflito apenas no corpo da função `greet`, uma vez que as outras alterações feitas por *left* e *right* não são conflitantes. Este resultado pode ser visto na Figura 2.

2.1.1 Ferramentas de *merge* Estruturado

Outra abordagem para integração de código consiste em usar uma ferramenta capaz de interpretar os códigos das revisões e transformá-los em árvores sintáticas abstratas. Ferramentas

base.js

```
const greet = (name) => {
  console.log(`Hello, ${name}`);
  console.log('Nice to meet you!');
}
```

left.js

```
const greet = (name) => {
  console.log(`Hi, ${name}`);
  console.log('Nice to meet you!');
}
```

right.js

```
const greet = (name) => {
  console.log(`Hello, ${name}`);
  console.log('Very nice to meet you!');
}
```

Figura 3 – Exemplo de versões de um arquivo JavaScript com mudanças em linhas consecutivas

diff3.js

```
const greet = (name) => {
<<<<<<< MINE
  console.log(`Hi, ${name}`);
  console.log('Nice to meet you!');
=======
  console.log(`Hello, ${name}`);
  console.log('Very nice to meet you!');
>>>>>>> YOURS
}
```

mergiraf.js

```
const greet = (name) => {
  console.log(`Hi, ${name}`);
  console.log('Very nice to meet you!');
}
```

Figura 4 – Resultados da integração das revisões da Figura 3 pelo diff3 e Mergiraf

deste tipo fazem o casamento dos nós das árvores geradas para determinar que mudanças são conflitantes (LEßENICH, 2012; LARSEN et al., 2022; DELPEUCH, 2025). Assim, são capazes de isolar alterações que, apesar de acontecerem na mesma área de código, não são conflitantes. Por exemplo, na Figura 3, *left* e *right* alteraram linhas consecutivas no corpo da mesma função. Mergiraf (DELPEUCH, 2025), uma ferramenta de *merge* estruturado, soube combinar as mudanças corretamente, pois as alterações aconteceram em nós diferentes da árvore sintática do código interpretado. Enquanto isso, o diff3 reportou um conflito, uma vez que não existe uma linha em comum entre *left* e *right* que separa as alterações. Estes resultados podem ser vistos na Figura 4.

Estas soluções são mais difíceis de desenvolver, visto que precisam saber interpretar diversas linguagens de programação (CAVALCANTI et al., 2024). Além disso, a construção das árvores sintáticas abstratas, o casamento dos nós e a transformação da árvore que representa o código integrado de volta em texto aumentam o tempo de execução destas ferramentas (LARSEN et al., 2022), prejudicando sua adoção. Resultados recentes (DUARTE; BORBA; CAVALCANTI, 2025)

mostraram que é possível reduzir esse tempo, mas a abordagem estruturada ainda requer um esforço extra de implementação e envolve um custo computacional grande.

2.1.2 Ferramentas de *merge* Semi-Estruturado

Similar ao *merge* estruturado, ferramentas de *merge* semi-estruturado procuram combinar as vantagens dos métodos textual e estruturado. Para isso, seu algoritmo permite parar a interpretação do código a partir de um determinado nível hierárquico da árvore sintática, como, por exemplo, o corpo de métodos e funções (CAVALCANTI et al., 2019; TAVARES et al., 2019). Desta forma, essas soluções fazem um casamento de nós das árvores sintáticas, para os níveis hierárquicos acima do pré-determinado, e casamento de linhas para os textos (não interpretados) abaixo do mesmo nível.

2.1.3 Ferramentas de *merge* Textual com Separadores Customizados

Há ainda uma abordagem textual que utiliza separadores sintáticos customizados. Esta consiste em pré-processar os códigos das revisões, particionando-os pelos separadores customizados e adicionando marcadores temporários, antes de efetuar um *merge* textual por linhas tradicional (CLEMENTINO; BORBA; CAVALCANTI, 2021; ARAUJO; BORBA; CAVALCANTI, 2024). Sua proposta é a de simular um *merge* estruturado, procurando isolar trechos de código independentes, delimitados por separadores sintáticos conhecidos. Por exemplo, "{" e "}" são utilizados por várias linguagens de programação para delimitar blocos de código, como corpos de funções. Ferramentas desse tipo podem então utilizar "{" e "}" como separadores customizados para isolar os corpos de funções de suas assinaturas, simulando o funcionamento do *merge* estruturado, em que a assinatura e o corpo de uma mesma função se encontram em nós diferentes da árvore sintática.

Na integração das revisões vistas na Figura 3, por exemplo, SepMerge (ARAUJO; BORBA; CAVALCANTI, 2024), uma ferramenta textual com separadores customizados, reporta o mesmo resultado que o Mergiraf, visto à direita da Figura 4, se separadores como "(" , ")" e ";" forem utilizados. Isto acontece porque, ao particionar o código por "(", ")" e ";", as mudanças de *left* e *right* são separadas por um conjunto de linhas em comum entre as revisões.

É importante notar, porém, que esta abordagem pode incorretamente isolar trechos de código não independentes. Por exemplo, o código pode conter um literal do tipo *string* com

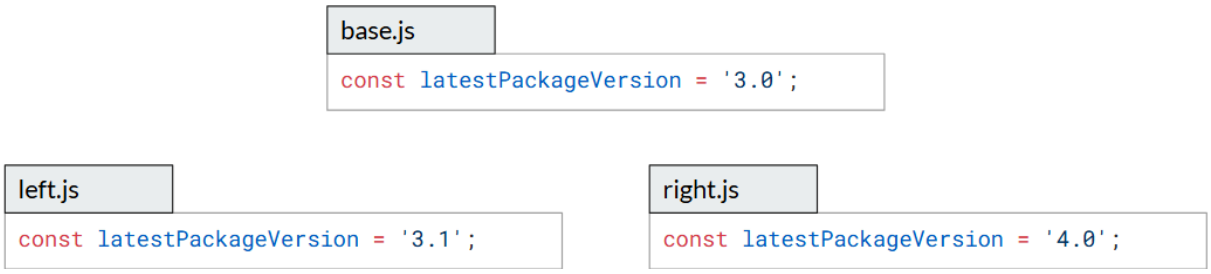

Figura 5 – Exemplo de integração de código em JavaScript com alterações conflitantes

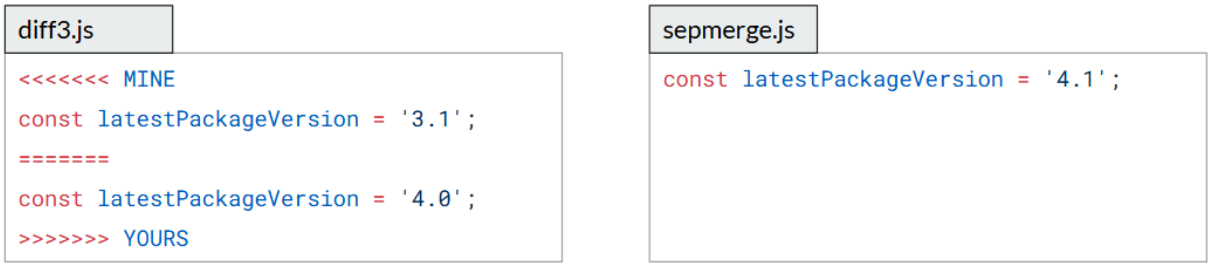

Figura 6 – Resultados da integração das revisões da Figura 5 pelo diff3 e SepMerge

separadores dentro, como no exemplo da Figura 5. Nela, *left* e *right* alteraram a mesma *string*. Caso o SepMerge inclua "." como um separador sintático para JavaScript, a ferramenta incorretamente conclui que as alterações não foram conflitantes, como pode ser visto à direita da Figura 6.

Além disso, o particionamento do código pode gerar muitas linhas novas exatamente iguais. Isto pode afetar o algoritmo de casamento de linhas do *merge* textual por linhas, fazendo com que a ferramenta reporte conflitos difíceis de entender (ARAUJO; BORBA; CAVALCANTI, 2024).

2.2 TRABALHOS RELACIONADOS

Outros estudos avaliaram o desempenho das diferentes abordagens descritas em cenários reais de integração de projetos de código aberto. Clementino, Borba e Cavalcanti (2021) propuseram o CSDiff, a primeira ferramenta de *merge* textual com separadores customizados e avaliaram seu desempenho em cenários de integração de projetos Java em comparação com o diff3, utilizando métricas similares às deste estudo. Este concluiu que a ferramenta poderia aumentar em até 105% o número de conflitos reportados, por reportar conflitos mais granulares, mas diminui o número de cenários com arquivos conflitantes. Também é discutido um problema de alinhamento e pareamento das linhas de código após o pré-processamento

da ferramenta, que pode ser um causador de falsos negativos e positivos. Problemas similares foram identificados neste trabalho e são discutidos na Seção 4.

Araujo, Borba e Cavalcanti (2024) evoluíram sobre os resultados do CSDiff, desenvolvendo o SepMerge, que utiliza o CSDiff apenas nos conflitos reportados pelo diff3. O estudo compara a nova ferramenta com o diff3 e CSDiff em cenários de integração de projetos Python. Os resultados mostraram que foi possível diminuir o número de conflitos reportados em até 40% na comparação com o CSDiff e em até 13% quando comparado com o diff3.

Cavalcanti et al. (2019) compararam as abordagens estruturada e semi-estruturada, simulando o uso de ferramentas dessas abordagens em mais de 40.000 cenários de integração de mais de 500 projetos desenvolvidos em Java. O estudo constatou que as soluções diferiram em 24% dos cenários com conflito, tendo a abordagem semi-estruturada apresentado um número maior de falsos positivos e a estruturada uma quantidade maior de falsos negativos.

Cavalcanti et al. (2024) tentaram combinar as abordagens semi-estruturada e textual com separadores customizados em uma ferramenta denominada Sesame. Nela, o *merge* textual com separadores customizados é utilizado no lugar do *merge* textual por linhas para integrar as mudanças nos corpos de métodos em Java. Os resultados mostraram que essa ferramenta foi eficiente em reduzir tanto o número de conflitos reportados quanto a quantidade de arquivos com esses conflitos, na comparação com ambos o diff3 e a ferramenta semi-estruturada original. Também foi observada uma grande diminuição nos falsos positivos da ferramenta nova, acompanhados de um aumento significativo no número de falsos negativos.

Este estudo visa evoluir sobre estes trabalhos em dois aspectos: comparar diretamente as abordagens estruturada e textual com separadores customizados, com o objetivo de compreender suas similaridades e diferenças de comportamento; e testar as ferramentas em três linguagens de programação diferentes, procurando identificar particularidades de cenários de integração de cada linguagem e entender a influência nos resultados das ferramentas.

3 METODOLOGIA

Este trabalho é de natureza aplicada, uma vez que os resultados aqui apresentados podem contribuir para a decisão de adoção de uma nova ferramenta de integração de código, buscando tornar o processo de *merge* mais eficiente, e facilitar a condução de experimentos semelhantes com novas ferramentas e outras linguagens de programação. A realização do mesmo envolveu os seguintes passos de desenvolvimento:

- Escolher linguagens de programação e projetos populares e relevantes para extrair cenários reais de integração de código;
- Investigar estado atual de ferramentas de integração de código estruturadas e textuais com separadores customizados e estendê-las com suporte a múltiplas linguagens de programação;
- Incrementar ferramental de suporte para reproduzir cenários de integração e comparar resultados;
- E coletar métricas e conduzir análise manual sobre alguns dos resultados das ferramentas testadas.

Nas seções seguintes, são discutidos estes passos em detalhes, apresentando as decisões feitas e explicando o raciocínio por trás delas.

3.1 LINGUAGENS

Como primeiro passo do trabalho, foi necessário escolher três linguagens de programação distintas para simular o uso de diferentes tipos de ferramentas de *merge* em cenários de integração de projetos escritos nestas linguagens. A escolha se pautou em três fatores: popularidade, familiaridade do autor e suporte nas ferramentas testadas.

3.1.1 Popularidade

O objetivo final deste experimento é fornecer métricas aproximadas sobre a corretude dos resultados de diversas ferramentas de *merge* em cenários reais de integração, facilitando a

tomada de decisão sobre uma possível adoção de uma solução diferente para o processo. Escolher linguagens com maior popularidade garante a cobertura de diversos cenários de integração comuns e uma maior relevância nos resultados do estudo.

3.1.2 Familiaridade

Uma das etapas deste trabalho consiste em realizar uma análise manual sobre alguns cenários de integração. Esta atividade exige compreender as mudanças de ambos os desenvolvedores envolvidos no processo, verificar a corretude dos resultados apresentados pelas ferramentas de *merge* e decidir a melhor forma de combinar as alterações. Dessa forma, ter familiaridade com a linguagem é importante para garantir um raciocínio correto e uma categorização adequada dos cenários.

Além disso, estender as ferramentas de integração com suporte a múltiplas linguagens de programação também é um objetivo do estudo. Ter conhecimento sobre a estrutura sintática das linguagens auxilia neste processo, tornando mais rápida a tomada de decisão sobre quais separadores customizados utilizar para cada linguagem, por exemplo.

3.1.3 Suporte nas ferramentas

Adiante, a Seção 3.2 descreve o estado atual das ferramentas de *merge*, as soluções escolhidas para o estudo e os incrementos feitos para adicionar suporte a múltiplas linguagens. A ferramenta de *merge* estruturada escolhida foi o Mergiraf, por suportar múltiplas linguagens de programação sem necessidade de muita configuração adicional.

Visando diminuir o tempo de desenvolvimento do estudo e configuração das ferramentas, foram priorizadas linguagens de programação com suporte nas ferramentas. Em particular, Ruby foi considerada como uma das linguagens do estudo, mas foi observado que Mergiraf tinha um suporte limitado à linguagem, com pouca configuração definida. Desta forma, apesar de permitir integrar arquivos escritos nesta linguagem, os resultados poderiam ser abaixo do esperado para a ferramenta.

3.1.4 Decisão

Baseado nos fatores discutidos acima, foram escolhidas as linguagens Go, JavaScript e Rust. Go é amplamente utilizado em *softwares* em que desempenho e concorrência são essenciais (CHENEY, 2017; CHABBI; RAMANATHAN, 2022). JavaScript é a principal linguagem de programação de desenvolvimento *web*, sendo a linguagem padrão dos navegadores (Stack Overflow, 2024; W3Techs, 2025; Mozilla Developer Network, 2025). Rust, apesar de mais recente, vem crescendo bastante em uso, sendo adotada por projetos em que segurança de memória e desempenho são importantes (TAFT, 2024; Rust Foundation, 2025; TAFT, 2025).

As três linguagens possuem suporte na ferramenta de integração estruturada utilizada no estudo e são conhecidas pelo autor, com experiência em duas delas.

3.2 FERRAMENTAS DE INTEGRAÇÃO

Para avaliar o comportamento dos diferentes tipos de solução para integração de código, foram escolhidas uma ferramenta de *merge* estruturada e uma textual com separadores customizados. Esta decisão se deu por dois motivos: acelerar o tempo de desenvolvimento do estudo, evitando a necessidade de adicionar suporte a múltiplas linguagens de programação em diversas ferramentas diferentes; e diminuir o tempo de execução do experimento de reprodução dos cenários de integração, visto que um número maior de ferramentas executadas poderia aumentar substancialmente o atraso para obter e analisar os resultados.

Além disso, o diff3 também é executado nos cenários de integração do experimento, uma vez que o objetivo deste trabalho é entender como as abordagens se comparam entre si e com a solução padrão amplamente adotada. As escolhas das soluções dos outros tipos, bem como as adições feitas às ferramentas escolhidas, são discutidas nas seções seguintes.

3.2.1 Ferramentas Estruturadas

Como o objetivo principal do estudo é avaliar o desempenho das ferramentas em três linguagens de programação, na escolha da ferramenta estruturada, foram priorizadas aquelas com suporte parcial ou total a múltiplas linguagens. É o caso do Mergiraf (DELPEUCH, 2025) e do Last Merge (DUARTE; BORBA; CAVALCANTI, 2025).

Porém, Last Merge possui uma limitação. Ao transformar a árvore sintática que representa

o código integrado em texto (um processo conhecido como *render*), seu algoritmo não preserva o espaçamento das revisões do código, mas apenas imprime o conteúdo dos nós terminais, separando-os por quebra de linha. Esta abordagem, apesar de funcionar bem com Java, faz com que adicionar suporte a múltiplas linguagens à ferramenta seja mais complicado. Isto acontece porque o algoritmo pode gerar código com sintaxe inválida ou semântica diferente da esperada em algumas linguagens. Por exemplo, ao adicionar suporte a JavaScript à ferramenta e tentar integrar as revisões da Figura 7, ela produz o resultado da esquerda da Figura 8, com uma interpolação de *string* iniciando e terminando em linhas diferentes, resultando em uma *string* com quebras de linha.

base.js

```
const v = 10;
const str = `v is ${v}`;
```

left.js

```
const v = 11;
const str = `v is ${v}`;
```

right.js

```
const v = 10;
const str = `v = ${v}`;
```

Figura 7 – Exemplo de integração de código em JavaScript

lastmerge.js

```
const

v
=
11
;

const

str
=

`
v =

${
v
}
`
;
```

mergiraf.js

```
const v = 11;
const str = `v = ${v}`;
```

Figura 8 – Resultado da integração das revisões da Figura 7 utilizando o Last Merge e Mergiraf

Esta limitação, porém, não é conceitual, mas sim de implementação. Mergiraf possui um algoritmo mais elaborado para transformar a árvore sintática em texto e já tem suporte a

diversas linguagens de programação. Por exemplo, ao integrar as revisões da Figura 7, a ferramenta imprime a saída que pode ser vista à direita da Figura 8.

Como seria necessário evoluir o algoritmo de *render* do Last Merge para adicionar suporte a múltiplas linguagens de programação à ferramenta, pensando em economizar tempo de desenvolvimento, Mergiraf foi escolhida como a representante das ferramentas estruturadas no estudo.

3.2.2 *Merge* Textual com Separadores Customizados

Existem duas ferramentas principais de natureza textual com uso de separadores customizados: CSDiff e SepMerge. Outros estudos constataram um problema de alinhamento e pareamento das linhas de código após pré-processamento do CSDiff (CLEMENTINO; BORBA; CAVALCANTI, 2021), resultando em alguns falsos negativos e positivos da ferramenta. SepMerge foi proposto como uma forma de amenizar o problema (ARAUJO; BORBA; CAVALCANTI, 2024). Para isso, ele faz uso de um mecanismo de *auto-tuning*, em que o diff3 é executado primeiramente e o algoritmo do CSDiff entra em ação apenas nos conflitos reportados pela primeira. Pensando então em validar a eficácia da solução proposta pelo SepMerge em múltiplas linguagens, esta foi a ferramenta de *merge* textual com separadores customizados escolhida.

Foi observado, porém, que o conjunto de separadores utilizados pela ferramenta era constante e independente da linguagem utilizada nos arquivos integrados. Dessa forma, apesar de não apresentar erros na integração de arquivos com extensões diferentes da esperada, os resultados da ferramenta nesses cenários poderiam ser abaixo do esperado, uma vez que as linguagens de programação possuem diferentes separadores sintáticos. Portanto, foi necessário incrementar a ferramenta com suporte a:

- Identificar a linguagem de programação pela extensão dos arquivos integrados;
- Selecionar conjunto de separadores customizados de acordo com a linguagem;
- E permitir adição de novas linguagens de forma facilitada.

Além disso, visando também entender a influência da escolha dos separadores e do tamanho do conjunto escolhido no resultado do *merge*, a ferramenta também foi incrementada com uma opção de linha de comando que permite a utilização de um conjunto maior de separadores sintáticos customizados por linguagem.

Com isso, o experimento pôde executar duas versões do SepMerge para cada cenário de integração — com o conjunto menor e maior de separadores de cada linguagem. A Tabela 1 mostra os conjuntos finais de separadores, separados por espaço, de cada linguagem utilizada no estudo.

Linguagem	Separadores Padrão	Separadores Extras
Go	`{ } [ ] ( ) ; ,`	`&& || += -= *= /= %= :=`
JavaScript	`{ } [ ] ( ) ; ,`	`&& || += -= *= /= %= => **`
Rust	`{ } [ ] ( ) ; ,`	`&& || += -= *= /= %= -> =>`

Tabela 1 – Separadores customizados utilizados pelo SepMerge por linguagem

É possível perceber que o conjunto menor de separadores de cada linguagem utilizada no estudo é o mesmo. Isto se deu apenas pelo motivo de que as três linguagens possuem uma estrutura sintática parecida no que diz respeito a separadores de blocos de código. Por exemplo, o corpo de uma função é delimitado por "{" e "}" e os argumentos por "(" e ")" nas três linguagens.

Além disso, o conjunto maior das três linguagens possui uma interseção. Os operadores *booleanos* "&&" e "||" e os de atribuição composta "+=", "-=", "*=", "/=" e "%=" estão presentes em muitas linguagens de programação, incluindo as utilizadas no estudo.

Os separadores dos conjuntos maiores foram deliberadamente escolhidos para serem, em maior parte, operadores das linguagens, visando observar a influência em cenários de integração em que desenvolvedores modificam operandos diferentes da mesma expressão. Porém, foi preciso atentar-se a dois problemas na escolha:

- **Alguns operadores são *substrings* de outros separadores**: por exemplo, o operador de adição "+" aparece como parte do operador de incremento e atribuição "+=" e o de subtração "-" como parte do sinal de retorno de função "->" em Rust. Desta forma, incluir tais operadores como separadores da ferramenta poderiam fazer com que a mesma particionasse o código de forma errada com mais frequência do que poderia ser desejado, podendo ser um causador de falsos negativos.

- **Operadores podem ter mais de uma função sintática**: por exemplo, o operador de multiplicação "*" também é operador de *dereferencing* em Go. É importante notar que, como operador de multiplicação, ele é binário, mas como de *dereferencing*, tem aridade 1. Isto também pode influenciar em uma partição não ideal do código.

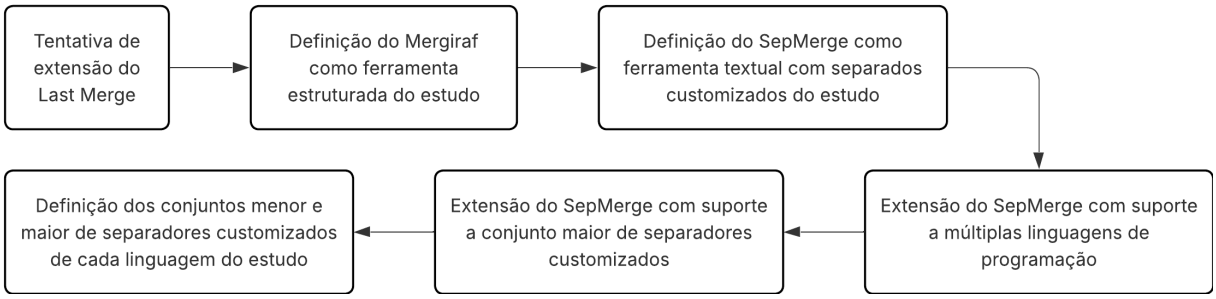

Figura 9 – Passo a passo da escolha e extensão das ferramentas de *merge* do estudo

A constatação desses problemas levou a uma escolha mais conservadora dos operadores utilizados, priorizando aqueles que não apresentam os riscos comentados acima. A Figura 9 ilustra o passo a passo da escolha e extensão das ferramentas de *merge* utilizadas no estudo.

3.3 PROJETOS

Tendo escolhido as linguagens de programação do experimento, foi necessário escolher projetos representantes de cada linguagem para extrair cenários de integração reais. Esta escolha consistiu em quatro passos:

1. Pré-seleção dos projetos de código aberto mais populares de cada linguagem no GitHub;
2. Computação do número de *commits* de *merge* de cada projeto selecionado;
3. Filtragem dos projetos pré-selecionados;
4. Limitação do número de projetos por linguagem.

Ao final do processo, 7 projetos foram escolhidos para cada linguagem. As seções seguintes explicam o passo a passo da escolha.

3.3.1 Pré-seleção

GitHub é uma plataforma para hospedagem de código-fonte e a mais popular da categoria (Stack Overflow, 2022). Nela, há um mecanismo para favoritar projetos. Projetos públicos possuem o número de usuários que o favoritaram exposto. Pensando em selecionar projetos conhecidos e relevantes para a comunidade, foi feita uma pré-seleção dos 20 projetos mais populares de cada linguagem disponíveis no GitHub de forma pública. Para isso, foi utilizada

a CLI (*command line interface*) gh (GitHub, Inc., 2025) provida pelo GitHub. Os comandos — executados no terminal de uma distribuição Linux — para encontrar os projetos foram os seguintes:

```
1    gh search repos --language go --sort stars --order desc --limit 20 --
         json name,url
2    gh search repos --language javascript --sort stars --order desc --limit
          20 --json name,url
3    gh search repos --language rust --sort stars --order desc --limit 20 --
         json name,url
```

Estes comandos buscam os 20 projetos mais populares do GitHub (segundo a métrica de favoritos) nas linguagens especificadas (Go, JavaScript e Rust) e imprimem os resultados no formato JSON, incluindo nome e URL dos projetos.

3.3.2 *Commits* de *merge*

Para reproduzir os cenários de integração dos projetos escolhidos, é necessário extrair os *commits* de *merge* dos repositórios dos projetos. Desta forma, visando dar prioridade a projetos com uma diversidade maior de cenários de integração, foi calculado o número de *commits* de *merge* dos projetos pré-selecionados de cada linguagem e as listas foram reordenadas segundo esta métrica (ficando os projetos com maior quantidade de *commits* desse tipo no topo). Isto foi feito com o auxílio de ferramentas como o `git log` (Software Freedom Conservancy, 2025) e `wc` (KERRISK, 2025), executando-se o seguinte comando em um terminal Linux:

```
1    git --no-pager log --merges --pretty=%H | wc -l
```

A primeira parte do comando (antes do `|`) lista os *hashes* (um por linha) de todos os *commits* de *merge* na *branch* principal do repositório. Esta lista é então fornecida como entrada para a segunda parte do comando (`wc -l`), que conta e imprime o número de linhas fornecidas. Este comando foi executado de forma automática para todos os 20 projetos pré-selecionados de cada linguagem, com o auxílio de um *script* em JavaScript desenvolvido pelo autor, disponível no Apêndice A.

3.3.3 Filtragem

O terceiro passo da seleção consistiu em aplicar um filtro nos projetos pré-selecionados e reordenados. Este eliminou projetos que não eram *softwares* — como coleções de algoritmos ou exercícios de cursos — e aqueles com um número de *commits* de *merge* abaixo de 200.

Como discutido em 3.4, a ferramenta de suporte à reprodução dos cenários de integração utilizada seleciona aleatoriamente 100 *commits* de *merge* de cada projeto após uma determinada filtragem dos mesmos. Garantir um número mínimo de 200 *commits* de integração em cada projeto foi uma medida pensada para evitar ter resultados enviesados no experimento, em que a maior parte dos *commits* de algum projeto é filtrada, bem como permitir uma maior variedade de combinações de cenários de integração para a ferramenta de reprodução escolher.

3.3.4 Limite

Após a filtragem descrita na Subseção 3.3.3, cada linguagem de programação ficou com uma lista de projetos associada com tamanho diferente, a menor contendo 7 projetos. Pensando em manter o número de cenários de integração processados de cada linguagem próximo — permitindo assim uma comparação direta entre os resultados encontrados em cada experimento —, foi decidido limitar o número de projetos de cada linguagem ao tamanho da menor lista. Com isso, foram 7 projetos escolhidos para cada linguagem do estudo e eles podem ser vistos na Tabela 2.

Go	JavaScript	Rust
kubernetes	three.js	rust
moby	open-webui	zed
etcd	react	union
prometheus	svelte	meilisearch
minio	uptime-kuma	rustdesk
lazygit	reveal.js	vaultwarden
traefik	node	bat

Tabela 2 – Projetos representantes de cada linguagem do estudo

Adiante, a Seção 4.1 mostra que foi possível coletar um total de 1.799 cenários de integração e 209.667 arquivos integrados dos projetos selecionados. A execução das ferramentas testadas para um número tão alto de arquivos integrados eleva o tempo de execução do expe-

rimento e a quantidade de armazenamento necessária para salvar os resultados das soluções significativamente. Ao mesmo tempo, esses números se mostraram suficientes para observar tendências nos resultados das ferramentas, sendo possível observar similaridades entre as métricas coletadas para as três linguagens. Dessa forma, não foi necessário adicionar mais projetos ao estudo.

3.4 FERRAMENTAL DE SUPORTE

Para realizar o experimento, foi necessário utilizar uma aplicação para reprodução de cenários de integração de código com diversas ferramentas de *merge*. Foi escolhido o Mining Framework (SPGroup, 2025). Desenvolvido em Groovy, ele é capaz de:

- Pré-processar os projetos fornecidos;
- Filtrar os *commits* de *merge* de cada projeto;
- Extrair dados dos *commits* de integração coletados;
- E realizar um pós-processamento nos dados coletados.

A Figura 10 ilustra o passo a passo da aplicação.

Para os fins deste estudo, o pré-processamento dos projetos e pós-processamento dos resultados não foram necessários. Portanto, esta seção foca em descrever o filtro de *commits* e a coleta dos dados. Além disso, durante a execução do trabalho, foi identificada uma oportunidade de melhoria na aplicação. Assim, esta seção também descreve as melhorias introduzidas e a execução do experimento.

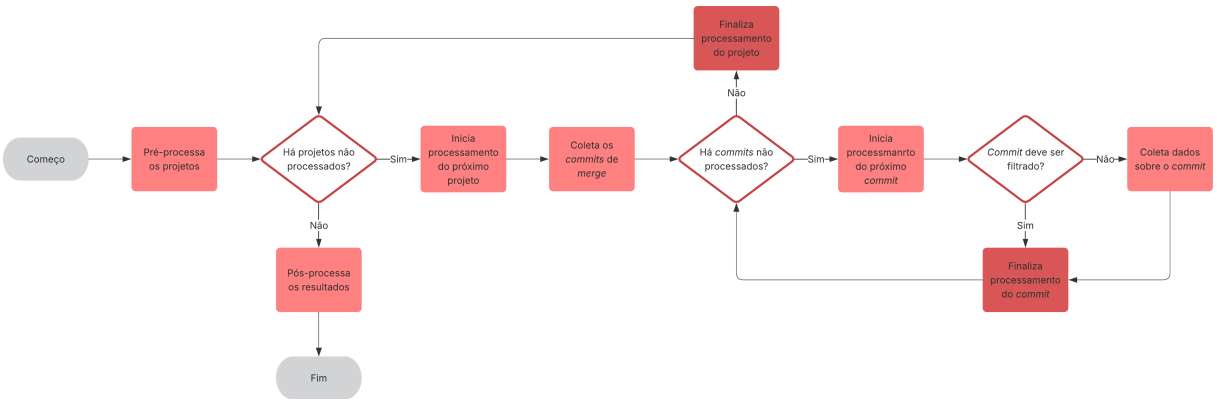

Figura 10 – Passo a passo do Mining Framework

3.4.1 Filtro de *commits*

Como um conflito de integração só pode acontecer em um arquivo quando este é alterado por dois desenvolvedores, foi decidido eliminar do experimento todos os cenários de integração sem arquivos modificados por ambos os desenvolvedores envolvidos no *merge*. Este filtro, já implementado na aplicação, tem o benefício adicional de diminuir o tempo de execução do experimento, uma vez que menos cenários são processados.

3.4.2 Coleta de dados

Esta é a etapa mais importante da aplicação. Nesta, são utilizadas informações como os *commits* que deram origem ao *commit* de *merge* para:

- Restaurar as versões (*base*, *left* e *right*) dos arquivos que foram integrados;
- Executar as diferentes ferramentas de *merge* nas revisões dos arquivos coletados;
- E comparar os resultados das ferramentas.

Como discutido na seção 3.2, neste estudo, são executadas três ferramentas de *merge*: diff3, Mergiraf e SepMerge. Sendo a última executada duas vezes, uma delas com a opção de linha de comando para usar o conjunto maior de separadores customizados.

Durante a execução das ferramentas, os resultados são armazenados em múltiplos arquivos (um para cada execução), junto às revisões (*base*, *left* e *right*) utilizadas no *merge* e ao arquivo de *merge* disponível no repositório.

Com isso, os arquivos são posteriormente processados e comparados. Para cada arquivo, as seguintes informações são extraídas:

- Projeto, *hash* do *commit* de *merge* e nome do arquivo;
- Número de conflitos reportados por cada ferramenta em sua execução;
- Equivalência dos conteúdos dos arquivos resultantes de cada par de ferramentas;
- E equivalência dos conflitos dos arquivos resultantes de cada par de ferramentas.

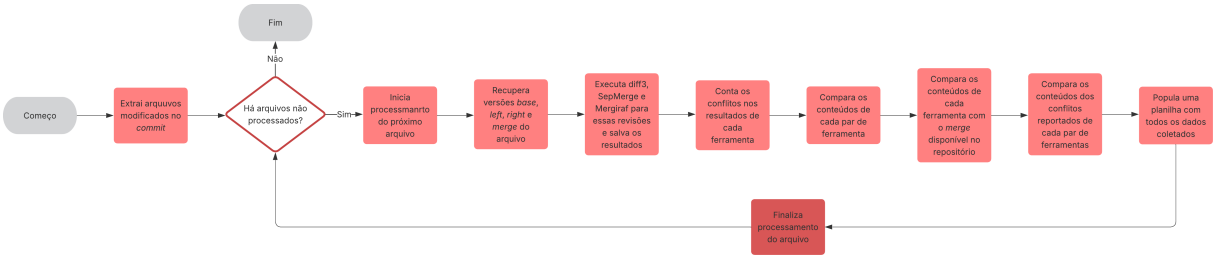

Figura 11 – Passo a passo da coleta de dados do Mining Framework

Além disso, os conteúdos dos arquivos resultantes de cada ferramenta são comparados por equivalência com o conteúdo do *merge* disponível no repositório. Para determinar a equivalência dos conteúdos dos arquivos e dos conflitos reportados, são deletados todos os espaços em branco na comparação, eliminando assim a influência de má indentação por parte de alguma ferramenta. A Figura 11 ilustra o passo a passo desta etapa da execução do Mining Framework.

3.4.3 Melhorias

A aplicação tem suporte a limitar os *commits* de *merge* processados apenas por intervalo de datas. Alguns projetos podem ter um número muito maior de *commits* de integração do que outros em um mesmo intervalo de tempo. Para evitar que isso enviesasse o estudo, foi proposta uma mudança no funcionamento da aplicação, introduzindo duas novas opções de linha de comando, `–max-commits-per-project` e `–random-seed`. A primeira tem o propósito de limitar o número de *commits* de *merge* processados por projeto por um máximo, evitando assim uma grande disparidade no número de cenários de integração coletados por projeto. Porém, pensando em permitir a escolha de um conjunto de *commits* de *merge* que abrange a grande variedade de cenários que pode existir no intervalo de datas fornecido, foi decidido adicionar aleatoriedade na escolha. Desta forma, o novo funcionamento da aplicação consiste em embaralhar os *commits* de *merge* no intervalo de datas fornecido e, se o valor de `–max-commit-per-project` é `x`, selecionar os `x` primeiros da lista que não são filtrados.

É importante, porém, garantir que o experimento pode ser reproduzido com fidelidade. O comportamento aleatório introduzido poderia ferir essa premissa, uma vez que cada execução do estudo poderia processar um conjunto diferente de cenários de integração. Pensando nisso, o valor de `–random-seed` é utilizado como *seed* para o gerador de números pseudoaleatórios utilizado pelo algoritmo para embaralhar a lista de *commits* de *merge*. Como a geração de

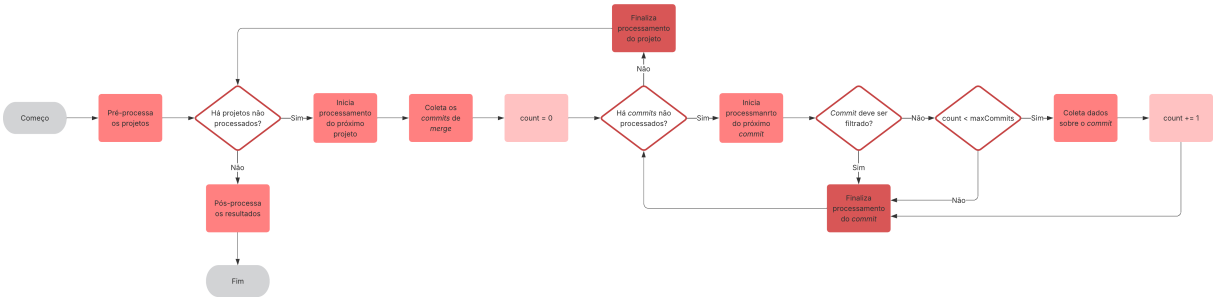

Figura 12 – Passo a passo atualizado do Mining Framework

números pseudoaleatórios pode ser completamente determinada pela *seed*, se o experimento for executado com os exatos mesmos parâmetros (entre eles `–random-seed` e o intervalo de datas), os mesmos cenários de integração serão processados.

A Figura 12 ilustra o passo a passo da ferramenta após essas modificações.

3.4.4 Execução

Com as ferramentas de integração prontas para integrar arquivos das três linguagens escolhidas para o estudo e a aplicação para reprodução dos cenários de integração devidamente incrementada e configurada, foi possível iniciar a execução do experimento.

Este foi executado em um *container* docker, cujo arquivo de configuração pode ser consultado no Apêndice B. Ele foi executado três vezes, uma vez para cada linguagem. Cada execução forneceu como entrada para a aplicação uma lista diferente de projetos, correspondente às escolhas discutidas na Seção 3.3 e que podem ser vistas na Tabela 2.

A Tabela 3 mostra os valores das opções de linha de comando da aplicação utilizados no experimento. O valor de `–random-seed` foi definido de forma arbitrária. Este e `–until-date` — que define um limite superior para o intervalo de datas permitido para os *commits* de *merge* coletados — estão presentes para garantir que os experimentos podem ser reproduzidos com fidelidade. O valor de `–max-commits-per-project` foi definido como 100 para eliminar o viés que um projeto com um grande número de cenários de integração pode trazer e permitir uma liberdade, por parte do Mining Framework, na escolha dos *commits* de *merge*, uma vez que os projetos foram escolhidos de forma a terem, pelo menos, 200 cenários de integração.

Além destas opções, cada experimento precisa definir o valor da opção `–extension`, que representa a extensão esperada dos arquivos integrados. Esta opção é utilizada para selecionar apenas os arquivos da linguagem estudada no experimento. Neste trabalho, os valores utilizados

-max-commits-per-project	-random-seed	-until-date
100	42	2025-06-30

Tabela 3 – Valores das opções de linha de comando utilizadas nos experimentos

para os estudos de Go, JavaScript e Rust foram, respectivamente, `.go`, `.js` e `.rs`.

3.5 MÉTRICAS E ANÁLISE

Tendo os experimentos finalizado suas execuções, foram coletadas métricas sobre os resultados. O Mining Framework produz como saída uma planilha com os dados coletados sobre as ferramentas, discutidos na Seção 3.4.2. Esta planilha foi consumida e filtrada pelo autor para extrair as seguintes métricas por linguagem:

- Número de projetos, cenários de integração e arquivos processados;
- Quantidade total de conflitos, cenários com conflitos e arquivos com conflitos por ferramenta;
- E falsos positivos e negativos adicionais por par ordenado de ferramentas.

O número de projetos, cenários e arquivos processados, além de servir como uma verificação preliminar de que os experimentos foram executados com sucesso, permite cálculos de porcentagem com as demais métricas, possibilitando extrapolar os resultados para outras situações.

As quantidades de conflitos, cenários com conflito e arquivos com conflito por ferramenta fornecem ideias da utilidade das diferentes soluções para integração de código no dia a dia, uma vez que resolver conflitos leva tempo e exige bastante atenção. Mas essas métricas, quando analisadas isoladamente, não são suficientes para justificar a preferência de uma ferramenta sobre outra. Por exemplo, uma aplicação que sempre ignora as mudanças de um dos desenvolvedores ao integrar as revisões de um arquivo nunca reportaria conflitos, mas dificilmente apresentaria resultados corretos. Da mesma forma, uma ferramenta que reporta conflito em todo arquivo modificado por dois desenvolvedores nunca apresentaria uma combinação incorreta das alterações introduzidas, mas exigiria ação do desenvolvedor com frequência. Assim, faz-se necessário classificar os resultados das ferramentas. Um resultado pode ser:

- **Falso positivo**: se a ferramenta reportou conflito e alguma outra ferramenta consegue integrar as revisões sem conflitos.
- **Falso negativo**: se a ferramenta não reportou conflito, mas apresentou um resultado incorreto, que não representa uma combinação válida das mudanças.

- **Correto**: se a ferramenta não reportou conflito e soube combinar corretamente as alterações, ou se reportou conflito e nenhuma outra ferramenta conseguiria integrar as revisões sem reportar conflitos.

Mas classificar um resultado é uma atividade que exige bastante tempo, uma vez que é necessário entender o cenário de integração — buscando compreender a influência das mudanças de *left* e *right* — e propor uma combinação válida das alterações. Fazer isso de forma manual para um número muito grande de arquivos integrados, como é o caso deste trabalho, torna-se inviável.

Por isso, este estudo, assim como outros, utiliza uma categorização aproximada e que pode ser computada de forma automática. A ideia consiste em classificar os resultados como falso positivo **adicional**, falso negativo **adicional** ou indefinido. Para isso, os resultados das ferramentas são comparados par a par e com o conteúdo do resultado do *merge* armazenado no repositório. A Figura 13 ilustra como é feita a classificação.

Um falso positivo adicional de uma ferramenta F1 com relação a uma ferramenta F2 implica em dizer que F1 reportou um conflito em uma situação em que F2 soube integrar as mudanças corretamente. De forma similar, um falso negativo adicional de F1 sobre F2 mostra que F1 integrou as alterações de forma incorreta, enquanto F2 preferiu reportar um conflito e exigir ação do desenvolvedor.

É importante ressaltar, porém, que esta métrica é apenas uma aproximação, uma vez que usa o resultado do *merge* disponível no repositório como fonte de verdade para o resultado correto esperado da integração. Mas este pode ter sido resultado de um falso negativo da ferramenta utilizada no *merge* ou de uma resolução incorreta de conflitos reportados, por parte de um desenvolvedor.

Visando diminuir erros provenientes desta aproximação e identificar padrões nas diferenças de resultados entre as ferramentas, foram também conduzidas algumas análises manuais sobre casos de falsos positivos e negativos adicionais e números muito grandes de conflitos reportados por uma ferramenta.

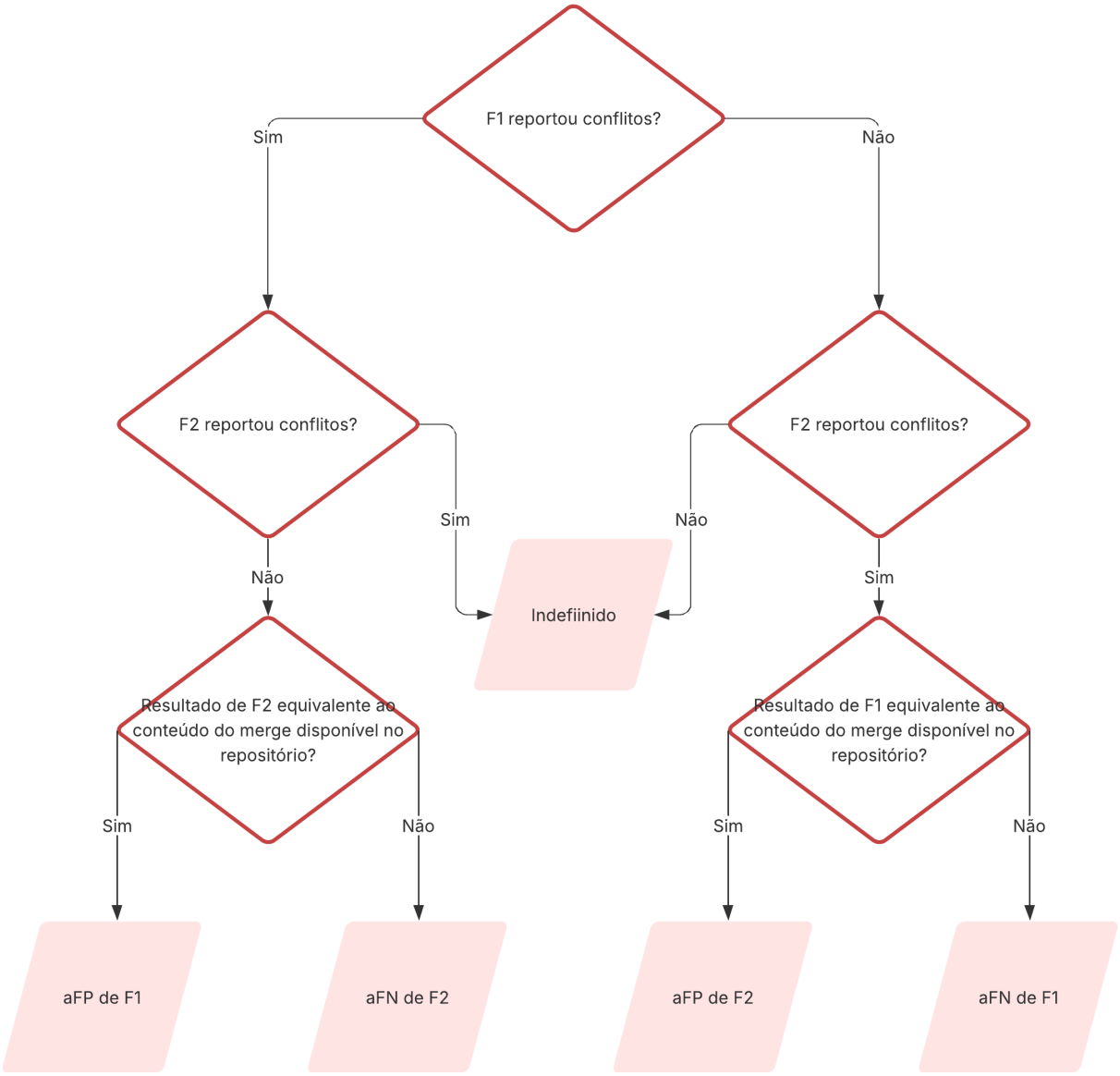

Figura 13 – Fluxograma da categorização dos resultados de ferramentas de integração F1 e F2 como falso positivo (aFP) ou negativo (aFN) adicional de uma sobre a outra

4 RESULTADOS E DISCUSSÃO

Este capítulo se encarrega de apresentar os resultados coletados nos experimentos das três linguagens, utilizando as métricas e a análise manual discutidas na Seção 3.5, e fazer observações sobre os dados analisados, procurando encontrar padrões de comportamento das ferramentas. Além disso, são discutidas ameaças à validade dos dados aqui apresentados.

4.1 COLETA

A primeira informação extraída dos experimentos foram as quantidades de projetos, cenários de integração e arquivos processados nas execuções. A Tabela 4 mostra os resultados.

	Go	JavaScript	Rust
Projetos	7	6	7
Cenários	629	600	570
Arquivos	112397	39481	57789

Tabela 4 – Quantidade de projetos, cenários de integração e arquivos processados em cada experimento

É possível notar que o experimento de JavaScript teve apenas 6 projetos processados, apesar de 7 terem sido fornecidos como entrada para a ferramenta de reprodução dos cenários. Isto aconteceu porque, apesar dos 7 projetos terem um mínimo de 200 *commits* de *merge*, o projeto open-webui não tinha nenhum com arquivos modificados por ambos os desenvolvedores. Dessa forma, todos os *commits* foram filtrados pela ferramenta.

Pelo mesmo motivo, os experimentos de Go e Rust não tiveram um total de 700 cenários processados: nem todos os projetos tinham 100 *commits* de *merge* com arquivos com alterações dos dois desenvolvedores. A Tabela 5 mostra o número de cenários e arquivos processados por projeto.

É possível observar também um número muito alto de arquivos integrados no experimento. Ao extrair a média de arquivos em cada cenário de integração, por exemplo, obtém-se números como 178, 65 e 101 nos experimentos de Go, JavaScript e Rust, respectivamente.

Buscando entender o motivo de números tão grandes, foi constatado que alguns dos projetos utilizados no estudo possuem suas pastas de dependências rastreadas pelo *git*. É o caso dos projetos kubernetes (Cloud Native Computing Foundation, 2025) e moby (Docker Inc., 2025), por exemplo. Assim, mudanças nas dependências dos projetos em *commits* de *merge* fizeram

Linguagem	Projeto	Cenários	Arquivos
Go	kubernetes	100	63126
Go	moby	100	11104
Go	etcd	100	7928
Go	prometheus	100	6029
Go	minio	29	403
Go	lazygit	100	3422
Go	traefik	100	20385
JavaScript	three.js	100	9030
JavaScript	open-webui	0	0
JavaScript	react	100	6759
JavaScript	svelte	100	2726
JavaScript	uptime-kuma	100	3071
JavaScript	reveal.js	100	727
JavaScript	node	100	17168
Rust	rust	100	43582
Rust	zed	100	4269
Rust	union	38	1193
Rust	meilisearch	100	3580
Rust	rustdesk	100	3525
Rust	vaultwarden	100	1068
Rust	bat	32	302

Tabela 5 – Quantidade de cenários de integração e arquivos processados por projeto

com que arquivos que não pertencem diretamente aos projetos estudados, mas que também são arquivos escritos nas linguagens estudadas no experimento, fossem também integrados.

Além disso, projetos como three.js (MrDoob, 2025), react (Meta Platforms, Inc., 2025b) e rust (Rust Project Developers, 2025) podem ter ciclos de vida longos para algumas *branches* de desenvolvimento, com várias funcionalidades e correções sendo publicadas em uma mesma *release*. Isto faz com que um único *commit* de *merge* na *branch* principal do projeto tenha vários arquivos modificados.

4.2 CONFLITOS

Também foram coletadas métricas sobre o número total de conflitos, cenários de integração com conflitos e arquivos com conflitos reportados por cada ferramenta. As Tabelas 6, 7 e 8 mostram os resultados por experimento. Os valores entre parênteses representam a por-

centagem sobre o total de cenários processados. Além disso, a versão do SepMerge executada com o conjunto maior de separadores foi denominada Extensive SepMerge.

Ferramenta	Conflitos	Cenários com conflito	Arquivos com conflito
Diff3	769	117 (~18.6%)	334
SepMerge	722	107 (~17.0%)	282
Extensive SepMerge	722	107 (~17.0%)	281
Mergiraf	502	84 (~13.3%)	187

Tabela 6 – Conflitos, cenários e arquivos com conflito por ferramenta no experimento de Go

Ferramenta	Conflitos	Cenários com conflito	Arquivos com conflito
Diff3	601	201 (~33.5%)	372
SepMerge	35201	182 (~30.3%)	304
Extensive SepMerge	34728	180 (~30.0%)	298
Mergiraf	389	144 (~24.0%)	225

Tabela 7 – Conflitos, cenários e arquivos com conflito por ferramenta no experimento de JavaScript

Ferramenta	Conflitos	Cenários com conflito	Arquivos com conflito
Diff3	320	98 (~17.2%)	212
SepMerge	384	78 (~13.7%)	161
Extensive SepMerge	384	78 (~13.7%)	161
Mergiraf	227	70 (~12.3%)	150

Tabela 8 – Conflitos, cenários e arquivos com conflito por ferramenta no experimento de Rust

Nota-se que, consistentemente, as abordagens estruturada e textual com separadores customizados reportaram conflitos em menos cenários de integração do que a abordagem textual por linhas, tendo o Mergiraf apresentado os melhores resultados segundo essa métrica. SepMerge foi capaz de reduzir o número de cenários de integração com conflitos do diff3 em ~8.5%, ~9.5% e ~20.4% nos experimentos de Go, JavaScript e Rust, respectivamente. A ferramenta estruturada foi ainda mais eficaz, reduzindo os cenários com conflitos do SepMerge em ~21.5%, ~20.9% e ~10.3%.

Apesar das diferenças no experimento de Rust, observa-se que a ferramenta estruturada reportou conflitos em ~28.2%, ~28.4% e ~28.6% menos cenários na comparação com o diff3 nos experimentos de Go, JavaScript e Rust, respectivamente. Uma tendência semelhante pode ser observada no número de arquivos com conflitos. Além disso, as duas versões do SepMerge

obtiveram resultados bem próximos com relação a essas métricas, com a versão Extensive tendo se saído ligeiramente melhor nos experimentos de Go e JavaScript.

Ademais, percebe-se que um número menor de cenários e arquivos com conflitos não necessariamente implica em um número menor de conflitos reportados. Por exemplo, as duas versões do SepMerge foram as ferramentas com o maior número de conflitos reportados nos experimentos de JavaScript e Rust, apesar de apresentarem resultados melhores que o diff3 nas outras métricas.

Em particular, SepMerge e Extensive SepMerge reportaram muito mais conflitos que o diff3 e Mergiraf no experimento de JavaScript, multiplicando o número de conflitos reportados pela ferramenta textual por linhas em mais de 57 vezes e pela ferramenta estruturada em mais de 89 vezes. Para entender o motivo dessa discrepância, foram analisados todos os arquivos em que pelo menos uma das versões do SepMerge reportou 100 ou mais conflitos. Foram 18 arquivos analisados, com um total de 34.416 conflitos reportados pelo SepMerge e 33.943 pelo Extensive SepMerge. Na análise, foi constatado que todos os arquivos eram minificados.

Um arquivo JavaScript minificado é aquele que teve seu conteúdo reduzido ao mínimo necessário para execução (Cloudflare, Inc., 2025). Esses arquivos são normalmente gerados por ferramentas de minificação em um dos passos do *build* da aplicação. No processo de minificação, são removidos espaços em branco desnecessários e comentários do arquivo original, e os nomes das variáveis são encurtados. Assim, arquivos desse tipo costumam ter apenas uma ou algumas linhas de código, mas nenhuma linha em branco. Como exemplo, o trecho de código abaixo foi extraído de um arquivo minificado do projeto three.js.

```
1 var THREE=THREE||{};if(!window.Int32Array)window.Int32Array=Array,window.
    Float32Array=Array;THREE.Color=function(b){this.setHex(b)};
```

Como a abordagem do diff3 é textual por linhas, ao integrar arquivos minificados, a ferramenta costuma reportar poucos conflitos. Por exemplo, em 12 dos 18 casos analisados, o diff3 reportou apenas 1 conflito. E a maior quantidade de conflitos reportados pela ferramenta textual por linhas nesses casos foi 30, contra 10503 do SepMerge e 10609 da versão com mais separadores.

Esses arquivos também não apresentam grandes problemas para o Mergiraf, que teve exatamente as mesmas saídas que o diff3 em todos os casos analisados. A documentação da ferramenta estruturada explica que a mesma pode utilizar o resultado do *merge* textual por linhas como solução alternativa em casos de falha de interpretação do código integrado ou

base.js

```
function compute(a, b) {
  const c = a + b;
  return c;
}
```

left.js

```
function add(x, y) {
  const sum = x + y;
  return sum;
}
```

right.js

```
function sum(v1, v2) {
  const result = v1 + v2;
  return result;
}
```

Figura 14 – Exemplo de integração de código em JavaScript com alterações conflitantes

diff3.js

```
<<<<<<< MINE
function add(x, y) {
  const sum = x + y;
  return sum;
=======
function sum(v1, v2) {
  const result = v1 + v2;
  return result;
>>>>>>> YOURS
}
```

Figura 15 – Resultado do diff3 para a integração da Figura 14

inconsistências detectadas após o algoritmo de *merge* estruturado (DELPEUCH, 2025). Este mecanismo pode explicar os resultados idênticos, mas uma análise mais minuciosa é necessária para entender o motivo.

Por outro lado, o particionamento do código por separadores customizados feito no pré-processamento do SepMerge faz com que as revisões do código minificado passem a ter um número muito maior de linhas, o que acarreta também em um número muito maior de conflitos. Isso é resultado da natureza de simulação do *merge* estruturado da ferramenta textual com separadores customizados. Por exemplo, na Figura 14, *left* e *right* alteraram o nome, argumentos e variável no corpo da mesma função. Os resultados da integração dessas revisões pelo diff3, Mergiraf e SepMerge podem ser vistos nas Figuras 15 e 16, tendo as ferramentas reportado 1, 3 e 5 conflitos, respectivamente.

mergiraf.js

```
<<<<<<< MINE
function add(x, y) {
=======
function sum(v1, v2) {
>>>>>>> YOURS
<<<<<<< MINE
  const sum = x + y;
=======
  const result = v1 + v2;
>>>>>>> YOURS
<<<<<<< MINE
  return sum;
=======
  return result;
>>>>>>> YOURS
}
```

sepmerge.js

```
<<<<<<< MINE
function add
=======
function sum
>>>>>>> YOURS
(
<<<<<<< MINE
x
=======
v1
>>>>>>> YOURS
,
<<<<<<< MINE
 y
=======
 v2
>>>>>>> YOURS
)
 {
<<<<<<< MINE
  const sum = x + y
=======
  const result = v1 + v2
>>>>>>> YOURS
;
<<<<<<< MINE
  return sum
=======
  return result
>>>>>>> YOURS
;
}
```

Figura 16 – Resultados do Mergiraf e SepMerge para a integração da Figura 14

4.3 FALSOS POSITIVOS E NEGATIVOS ADICIONAIS

Por fim, como discutido na Seção 3.5, é importante classificar os resultados das ferramentas. Este estudo compara as saídas de cada par de ferramentas de forma automática, classificando os resultados como falso positivo adicional, falso negativo adicional ou indefinido. Além disso, foi conduzida uma análise manual sobre alguns falsos positivos e negativos adicionais do Mergiraf em comparação com o SepMerge e vice-versa.

4.3.1 Falsos Positivos Adicionais

As Tabelas 9, 10 e 11 mostram a quantidade de falsos positivos adicionais nos experimentos de Go, JavaScript e Rust, respectivamente. O valor na linha da ferramenta F1 e coluna da ferramenta F2 deve ser interpretado como o número de falsos positivos adicionais da ferramenta

F1 em comparação com a ferramenta F2. Ou seja, a quantidade de arquivos em que F1 reportou conflitos, F2 não o fez e o resultado de F2 é equivalente ao conteúdo original integrado no repositório do projeto. Por exemplo, a Tabela 9 mostra que diff3 teve 37 falsos positivos adicionais em comparação com o SepMerge no experimento de Go.

	Diff3	SepMerge	Extensive SepMerge	Mergiraf
Diff3	-	37	38	82
SepMerge	1	-	1	49
Extensive SepMerge	1	0	-	48
Mergiraf	2	4	4	-

Tabela 9 – Falsos positivos adicionais no experimento de Go

	Diff3	SepMerge	Extensive SepMerge	Mergiraf
Diff3	-	24	27	65
SepMerge	2	-	3	45
Extensive SepMerge	2	0	-	44
Mergiraf	1	2	4	-

Tabela 10 – Falsos positivos adicionais no experimento de JavaScript

	Diff3	SepMerge	Extensive SepMerge	Mergiraf
Diff3	-	35	35	37
SepMerge	1	-	0	16
Extensive SepMerge	1	0	-	16
Mergiraf	1	13	13	-

Tabela 11 – Falsos positivos adicionais no experimento de Rust

As métricas mostram que o diff3 apresenta as maiores taxas de falsos positivos adicionais, enquanto o Mergiraf apresenta as menores. Isto evidencia o poder da ferramenta estruturada, que, por interpretar o código integrado, consegue melhor isolar as alterações dos desenvolvedores e integrá-las corretamente. Além disso, novamente, as duas versões do SepMerge aparecem como uma solução intermediária, tendo registrado taxas muito próximas do Mergiraf no que diz respeito a falsos positivos adicionais das ferramentas em comparação com o diff3.

Nota-se, mais uma vez, que SepMerge e Mergiraf apresentaram taxas muito próximas no experimento de Rust. Enquanto a ferramenta estruturada tem ~91.8% e ~95.5% menos falsos positivos adicionais nos experimentos de Go e JavaScript, respectivamente, a redução foi de apenas ~18.7% no experimento de Rust.

left.js

```
let node: HTMLElement | null = null;
if (renderer !== null) {
    node = ((renderer.getNativeFromInternal(id): any): HTMLElement);
}
```

base.js

```
let node: HTMLElement | null = null;
if (renderer !== null) {
    node = ((renderer.findNativeByFiberID(id): any): HTMLElement);
}
```

right.js

```
let nodes: ?Array<HTMLElement> = null;
if (renderer !== null) {
    nodes = ((renderer.findNativeByFiberID(id): any): ?Array<HTMLElement>);
}
```

Figura 17 – Revisões de um dos falsos positivos adicionais do Mergiraf em comparação com o SepMerge

Também foi observado que o SepMerge teve um total de 4 falsos positivos adicionais em comparação com sua versão Extensive, esta última tendo apresentado números ligeiramente menores na comparação com Mergiraf nos experimentos de Go e JavaScript. Ademais, Extensive SepMerge não reportou nenhum falso positivo adicional na comparação com a versão com menos separadores. Este resultado sugere que um número maior de separadores customizados pode aproximar a ferramenta textual da estruturada.

Entre os casos de falsos positivos adicionais do Mergiraf em comparação com o SepMerge, está o da Figura 17. Este exemplo é um trecho de código de um arquivo integrado do projeto react, no experimento de JavaScript. Nele, um desenvolvedor (*left*) renomeou uma função, enquanto o outro (*right*) alterou uma das linhas com chamada à função renomeada. A Figura 18 mostra os resultados do diff3 e Mergiraf para esse caso.

Observa-se que as ferramentas reportaram resultados idênticos. Isto aconteceu porque as revisões do arquivo integrado usam Flow, uma ferramenta de checagem de tipos (Meta Platforms, Inc., 2025a). Ela permite adicionar anotações de tipo ao código JavaScript, se comportando de forma similar ao TypeScript. Mas códigos que utilizam essas anotações do Flow não conseguem ser interpretados pelo Mergiraf, o que faz com que a ferramenta utilize o resultado do diff3 como alternativa.

diff3.js

```
let nodes: ?Array<HTMLElement> = null;
if (renderer !== null) {
<<<<<<< MINE
    node = ((renderer.getNativeFromInternal(id): any): HTMLElement);
=======
    nodes = ((renderer.findNativeByFiberID(id): any): ?Array<HTMLElement>);
>>>>>>> YOURS
}
```

mergiraf.js

```
let nodes: ?Array<HTMLElement> = null;
if (renderer !== null) {
<<<<<<< MINE
    node = ((renderer.getNativeFromInternal(id): any): HTMLElement);
=======
    nodes = ((renderer.findNativeByFiberID(id): any): ?Array<HTMLElement>);
>>>>>>> YOURS
}
```

Figura 18 – Resultados da integração das revisões da Figura 17 pelo diff3 e Mergiraf

sepmerge.js

```
let nodes: ?Array<HTMLElement> = null;
if (renderer !== null) {
    nodes = ((renderer.getNativeFromInternal(id): any): ?Array<HTMLElement>);
}
```

Figura 19 – Resultado da integração das revisões da Figura 17 pelo SepMerge

Por outro lado, o SepMerge pôde utilizar os parênteses para isolar as alterações dos dois desenvolvedores, conseguindo então integrar os resultados sem reportar conflito, como pode ser visto na Figura 19.

Nas Figuras 20 e 21, vemos as revisões de um falso positivo adicional do SepMerge em comparação com o Mergiraf. Neste caso, do projeto meilisearch (Meilisearch, 2025) do experimento de Rust, *left* consertou a indentação de um trecho de código alterado por *right*. A Figura 22 mostra que Mergiraf soube integrar corretamente as mudanças.

Enquanto isso, SepMerge reportou um conflito errado, como pode ser visto na Figura 23. Ao particionar os códigos das revisões pelos separadores customizados de Rust, a ferramenta incorretamente interpretou a mudança na indentação feita por *left* como sendo conflitante com parte (os caracteres que aparecem antes de "(") da primeira linha introduzida por *right*,

base.js

```
HttpResponse::Ok().json(VersionResponse {
     commit_sha: env!("VERGEN_SHA").to_string(),
     build_date: env!("VERGEN_BUILD_TIMESTAMP").to_string(),
     pkg_version: env!("CARGO_PKG_VERSION").to_string(),
 })
```

Figura 20 – Versão *base* de um dos falsos positivos adicionais do SepMerge em comparação com o Mergiraf

left.js

```
HttpResponse::Ok().json(VersionResponse {
     commit_sha: env!("VERGEN_SHA").to_string(),
     build_date: env!("VERGEN_BUILD_TIMESTAMP").to_string(),
     pkg_version: env!("CARGO_PKG_VERSION").to_string(),
 })
```

right.js

```
let commit_sha = match option_env!("COMMIT_SHA") {
     Some("") | None => env!("VERGEN_SHA"),
     Some(commit_sha) => commit_sha
};
let commit_date = match option_env!("COMMIT_DATE") {
     Some("") | None => env!("VERGEN_COMMIT_DATE"),
     Some(commit_date) => commit_date
};

HttpResponse::Ok().json(VersionResponse {
     commit_sha: commit_sha.to_string(),
     build_date: commit_date.to_string(),
     pkg_version: env!("CARGO_PKG_VERSION").to_string(),
 })
```

Figura 21 – Versões *left* e *right* de um dos falsos positivos adicionais do SepMerge em comparação com o Mergiraf

entendendo que o restante das mudanças introduzidas por *right* aconteceram em uma área de código que *left* não modificou. Isto caracteriza um problema de alinhamento das linhas de código (após pré-processamento da ferramenta) das revisões por parte do SepMerge (CLEMENTINO; BORBA; CAVALCANTI, 2021).

4.3.2 Falsos Negativos Adicionais

Quando olhamos para os falsos negativos adicionais, o Mergiraf apresenta os piores resultados e o diff3 os melhores. As Tabelas 12, 13 e 14 mostram os números coletados. Similar às

mergiraf.js

```
let commit_sha = match option_env!("COMMIT_SHA") {
    Some("") | None => env!("VERGEN_SHA"),
    Some(commit_sha) => commit_sha
};
let commit_date = match option_env!("COMMIT_DATE") {
    Some("") | None => env!("VERGEN_COMMIT_DATE"),
    Some(commit_date) => commit_date
};

HttpResponse::Ok().json(VersionResponse {
    commit_sha: commit_sha.to_string(),
    build_date: commit_date.to_string(),
    pkg_version: env!("CARGO_PKG_VERSION").to_string(),
})
```

Figura 22 – Resultado da integração das revisões das Figuras 20 e 21 pelo Mergiraf

sepmerge.js

```
<<<<<<< MINE
 HttpResponse::Ok
=======
 let commit_sha = match option_env!
>>>>>>> YOURS
 (
 "COMMIT_SHA") {
    Some("") | None => env!("VERGEN_SHA"),
    Some(commit_sha) => commit_sha
 };
 let commit_date = match option_env!("COMMIT_DATE") {
    Some("") | None => env!("VERGEN_COMMIT_DATE"),
    Some(commit_date) => commit_date
 };

 HttpResponse::Ok().json(VersionResponse {
    commit_sha: commit_sha.to_string(),
    build_date: commit_date.to_string(),
    pkg_version: env!("CARGO_PKG_VERSION").to_string(),
 })
```

Figura 23 – Resultado da integração das revisões das Figuras 20 e 21 pelo SepMerge

Tabelas 9, 10 e 11, um valor na linha da ferramenta F1 e coluna da ferramenta F2 deve ser interpretado como o número de falsos negativos adicionais da ferramenta F1 em comparação com a F2. Em outras palavras, a quantidade de arquivos em que F1 não reportou conflito, F2 o fez e o resultado de F1 não corresponde ao conteúdo de *actual*.

	Diff3	SepMerge	Extensive SepMerge	Mergiraf
Diff3	-	1	1	1
SepMerge	17	-	0	7
Extensive SepMerge	17	0	-	7
Mergiraf	68	57	57	-

Tabela 12 – Falsos negativos adicionais no experimento de Go

	Diff3	SepMerge	Extensive SepMerge	Mergiraf
Diff3	-	0	0	0
SepMerge	46	-	0	9
Extensive SepMerge	49	3	-	11
Mergiraf	83	45	44	-

Tabela 13 – Falsos negativos adicionais no experimento de JavaScript

	Diff3	SepMerge	Extensive SepMerge	Mergiraf
Diff3	-	1	1	1
SepMerge	18	-	0	10
Extensive SepMerge	18	0	-	10
Mergiraf	27	18	18	-

Tabela 14 – Falsos negativos adicionais no experimento de Rust

Estes resultados são consistentes com os apresentados em outros estudos (CAVALCANTI et al., 2019; CAVALCANTI et al., 2024) e mostram uma grande desvantagem das abordagens estudadas, visto que um resultado falso negativo é altamente prejudicial para o processo de desenvolvimento — *bugs* podem ser introduzidos na aplicação de forma silenciosa se os resultados, apesar de incorretos, forem sintaticamente válidos e os testes do projeto não possuírem cobertura adequada. Além disso, mais uma vez, evidenciam a natureza do SepMerge de simular o *merge* estruturado, uma vez que as duas versões da ferramenta apresentaram resultados intermediários entre os do diff3 e Mergiraf.

Novamente, foi observada uma similaridade no comportamento do SepMerge e do Mergiraf no experimento de Rust. Enquanto a ferramenta estruturada aumentou o número de falsos

base.js

```
CLIENT
    .get(url)
    .header("cookie", cookie_str)
    .send()?
    .error_for_status()
    .map_err(Into::into)
```

left.js

```
if check_icon_domain_is_blacklisted(...)
{
    err!("Favicon rel linked to ...");
}
CLIENT
    .get(url)
    .header("cookie", cookie_str)
    .send()?
    .error_for_status()
    .map_err(Into::into)
```

right.js

```
if cookie_str.is_empty() {
    CLIENT
        .get(url)
        .send()?
        .error_for_status()
        .map_err(Into::into)
} else {
    CLIENT
        .get(url)
        .header("cookie", cookie_str)
        .send()?
        .error_for_status()
        .map_err(Into::into)
}
```

Figura 24 – Revisões de um dos falsos negativos adicionais do Mergiraf em comparação com o SepMerge

negativos adicionais na comparação com o SepMerge em mais de 7 vezes no experimento de Go e em 4 vezes no experimento de JavaScript, o número cresceu apenas 0.8 vezes no experimento de Rust. Estes resultados sugerem que a ferramenta textual com separadores customizados pode ter um comportamento bem parecido com a solução estruturada em Rust. O motivo dessa semelhança deve ser estudado em mais profundidade, podendo estar relacionado ao conjunto de separadores customizados escolhidos para a ferramenta textual ou à configuração utilizada para a linguagem na ferramenta estruturada.

Alguns falsos negativos adicionais foram analisados manualmente. A Figura 24 mostra trechos de código de revisões de um arquivo do projeto vaultwarden (GARCIA, 2025) do experimento de Rust. Partes do código foram substituídas por "..." na figura para facilitar a compreensão. Neste exemplo, *left* adicionou uma verificação de erro antes do conteúdo visto em *base*, enquanto *right* moveu este conteúdo para dentro de um bloco *else* e adicionou um bloco *if* correspondente logo acima.

Na integração dessas revisões, como pode ser vista na Figura 25, diff3 e SepMerge reportaram conflitos, mas diferentes. Por particionar o código por separadores como "(", SepMerge conseguiu extrair uma parte não conflitante das mudanças (a partir de "(url)" em ".get(url)").

Mergiraf, por outro lado, integrou as mudanças de forma incorreta, movendo a verificação

diff3.js

```
<<<<<<< MINE
if check_icon_domain_is_blacklisted(...) {
    err!("Favicon rel linked to ...");
}
CLIENT
    .get(url)
    .header("cookie", cookie_str)
    .send()?
    .error_for_status()
    .map_err(Into::into)
=======
if cookie_str.is_empty() {
    CLIENT
        .get(url)
        .send()?
        .error_for_status()
        .map_err(Into::into)
} else {
    CLIENT
        .get(url)
        .header("cookie", cookie_str)
        .send()?
        .error_for_status()
        .map_err(Into::into)
}
>>>>>>> YOURS
```

sepmerge.js

```
<<<<<<< MINE
if check_icon_domain_is_blacklisted(...) {
    err!("Favicon rel linked to ...");
}
CLIENT
    .get
=======
if cookie_str.is_empty() {
    CLIENT
        .get(url)
        .send()?
        .error_for_status()
        .map_err(Into::into)
} else {
    CLIENT
        .get
>>>>>>> YOURS
(
url)
        .header("cookie", cookie_str)
        .send()?
        .error_for_status()
        .map_err(Into::into)
}
```

Figura 25 – Resultados da integração das revisões da Figura 24 pelo diff3 e SepMerge

mergiraf.js

```
if cookie_str.is_empty() {
    CLIENT
        .get(url)
        .send()?
        .error_for_status()
        .map_err(Into::into)
} else {
    if check_icon_domain_is_blacklisted(...) {
        err!("Favicon rel linked to ...");
    }
    CLIENT
        .get(url)
        .header("cookie", cookie_str)
        .send()?
        .error_for_status()
        .map_err(Into::into)
}
```

actual.js

```
if check_icon_domain_is_blacklisted(...) {
    err!("Favicon rel linked to ...");
}

if cookie_str.is_empty() {
    CLIENT
        .get(url)
        .send()?
        .error_for_status()
        .map_err(Into::into)
} else {
    CLIENT
        .get(url)
        .header("cookie", cookie_str)
        .send()?
        .error_for_status()
        .map_err(Into::into)
}
```

Figura 26 – Resultado do Mergiraf e integração correta das revisões da Figura 24

de erro introduzida por *left* para dentro do bloco *else* adicionado por *right*. O resultado da ferramenta estruturada e a integração correta podem ser vistos na Figura 26.

Entre os falsos negativos adicionais do SepMerge com relação ao Mergiraf, está o caso da Figura 27, do projeto prometheus (Prometheus Authors, 2025) do experimento de Go. Nele, *left* e *right* adicionaram funções distintas no final do mesmo arquivo de teste. Parte dos corpos

base.js

```
func TestReadyAndHealthy(t *testing.T) {
    // ...
}
```

left.js

```
func TestReadyAndHealthy(t *testing.T) {
    // ...
}

func TestRoutePrefix(t *testing.T) {
    // ...
}
```

right.js

```
func TestReadyAndHealthy(t *testing.T) {
    // ...
}

func TestDebugHandler(t *testing.T) {
    // ...
    handler := New(opts)
    // ...
}
```

Figura 27 – Revisões de um dos falsos negativos adicionais do SepMerge em comparação com o Mergiraf

mergiraf.js

```
func TestReadyAndHealthy(t *testing.T) {
    // ...
}

<<<<<<< MINE
func TestRoutePrefix(t *testing.T) {
    // ...
=======
}

func TestDebugHandler(t *testing.T) {
    // ...
    handler := New(opts)
    // ...
>>>>>>> YOURS
}
```

sepmerge.js

```
func TestReadyAndHealthy(t *testing.T) {
    // ...
}

func TestRoutePrefix(t *testing.T) {
    // ...
}

func TestDebugHandler(t *testing.T) {
    // ...
    handler := New(opts)
    // ...
}
```

Figura 28 – Resultados da integração das revisões da Figura 27 pelo Mergiraf e SepMerge

das funções foi substituído por "..." para facilitar a compreensão.

Nesta integração, Mergiraf reportou conflitos, enquanto SepMerge não o fez, como pode ser visto na Figura 28. A ferramenta textual corretamente adicionou as duas funções no final do arquivo de teste. Mas fora do contexto deste arquivo, *left* também alterou a função *New* para receber dois parâmetros. Assim, ao integrar as mudanças, também foi necessário alterar a chamada desta função, que pode ser vista na função de teste introduzida por *right*.

Em conclusão, as métricas sobre conflitos reportados pelas ferramentas mostraram que Mergiraf foi capaz de reduzir em ~28% o número de cenários com conflitos do diff3 nos três experimentos, enquanto o SepMerge diminuiu o mesmo número em cerca de 9% nos expe-

rimentos de Go e JavaScript e ~20% no de Rust, tendo se aproximado mais da ferramenta estruturada neste último. Esta aproximação também foi percebida no número de falsos positivos e negativos adicionais computados para cada par de ferramentas. Ao analisar estas métricas, constatou-se que a ferramenta textual por linhas apresenta as maiores quantidades de falsos positivos adicionais e o Mergiraf as menores. A situação é invertida ao olhar para os falsos negativos adicionais, porém, com a ferramenta estruturada apresentando as maiores taxas. Em todas as métricas computadas por ferramenta, o SepMerge apresentou resultados intermediários entre a ferramenta textual e estruturada, tendo suas duas versões estudadas se comportado de forma bem similar, com poucas diferenças.

4.4 AMEAÇAS À VALIDADE

Este estudo focou apenas em projetos desenvolvidos em três linguagens de programação: Go, JavaScript e Rust. Estas não representam a grande variedade de opções disponíveis atualmente e os projetos desenvolvidos nessas não carregam toda a diversidade de cenários de integração possíveis. Adicionar outras linguagens ao estudo poderia contribuir para a identificação de outras vantagens ou limitações das abordagens testadas, bem como ajudar a identificar padrões de acertos e erros das ferramentas.

Neste trabalho, foram testadas apenas uma ferramenta de cada abordagem mencionada. Isto representa um risco, visto que ferramentas diferentes da mesma abordagem poderiam apresentar resultados distintos. Também não foram testadas outras abordagens, como o *merge* semi-estruturado. Trabalhos futuros poderiam evoluir outras ferramentas de integração, adicionando suporte a múltiplas linguagens de programação e compará-las.

Por fim, como o processo de análise manual consome bastante tempo, este experimento optou por uma classificação dos resultados mais simples e aproximada, tendo realizado uma análise manual sobre um pequeno subconjunto dos arquivos integrados. Uma análise mais abrangente é importante para diminuir erros provenientes da métrica aproximada.

5 CONCLUSÃO

Este trabalho teve o objetivo de comparar diferentes abordagens de ferramentas de integração de código em três linguagens de programação diferentes. Foram analisados o comportamento de uma ferramenta de cada tipo de solução — textual por linhas, estruturada e textual com separadores customizados — em cenários reais de integração de projetos populares. Para isso, as ferramentas de integração foram estendidas com suporte a múltiplas linguagens e um ferramental de suporte à reprodução de cenários de integração foi incrementado.

Dos resultados obtidos, foram coletadas métricas sobre a acurácia das saídas das ferramentas e a eficiência das mesmas em resolver conflitos que a solução padrão amplamente adotada não é capaz de resolver. As métricas, juntamente a uma análise manual de alguns arquivos integrados, foram utilizadas para identificar as vantagens e limitações das abordagens estudadas.

A análise mostrou que a solução estruturada é muito eficiente em diminuir o número de cenários de integração e arquivos com conflitos, tendo apresentado uma redução de ~91.8%, ~95.5% e ~18.7% nos falsos positivos adicionais em comparação com o SepMerge nos experimentos de Go, JavaScript e Rust, respectivamente. Em contrapartida, a abordagem aumentou os falsos negativos adicionais em mais de 7 vezes no experimento de Go, 4 vezes em JavaScript e 0.8 vezes em Rust.

A abordagem textual com separadores customizados se mostrou uma solução intermediária entre a ferramental textual por linhas e a estruturada, tendo sido capaz de reduzir o número de cenários de integração com conflitos do diff3 em ~8.5%, ~9.5% e ~20.4% nos experimentos de Go, JavaScript e Rust, respectivamente. Duas versões da ferramenta foram testadas, uma delas com um conjunto maior de separadores customizados. As duas apresentaram resultados muito próximos, mas as métricas sugerem que uma quantidade maior de separadores pode aproximar a ferramenta do *merge* estruturado.

As tendências observadas nos resultados dos experimentos de Go e JavaScript foram consistentes entre si, mas uma diferença significativa foi observada no volume de conflitos reportados pelo SepMerge e Extensive SepMerge no experimento de JavaScript. Uma análise manual pôde constatar que o motivo da diferença estava relacionado à presença de arquivos minificados nos projetos. Além disso, os resultados do experimento de Rust mostraram uma maior similaridade no comportamento das duas versões da ferramenta textual com separadores customizados e da

solução estruturada. Trabalhos futuros podem focar em procurar os motivos dessa semelhança.

Os resultados sugerem que a adoção de uma nova ferramenta de *merge* pode trazer benefícios para o processo de integração de código. Mas as soluções também trazem desvantagens, como uma alta no número de conflitos reportados quando se utiliza o SepMerge e no número de falsos negativos adicionais quando o Mergiraf é utilizado.

REFERÊNCIAS

APEL, S.; LIEBIG, J.; BRANDL, B.; LENGAUER, C.; KÄSTNER, C. Semistructured merge: Rethinking merge in revision control systems. In: *Proceedings of the 19th Symposium on the Foundations of Software Engineering and the 13th European Software Engineering Conference (ESEC/FSE 2011)*. Szeged, Hungary: ACM, 2011. p. 190–200. ISBN 978-1-4503-0443-6.

ARAUJO, F.; BORBA, P.; CAVALCANTI, G. Refinando a precisao da detecçao de conflitos: Uma analise do csdiff com abordagem focalizada. In: *Anais do XXXVIII Simposio Brasileiro de Engenharia de Software (SBES)*. Curitiba, PR, Brasil: Sociedade Brasileira de Computaçao, 2024. p. 246–256. Disponível em: <https://sol.sbc.org.br/index.php/sbes/article/view/30366>.

BAUDIš, P. *Current Concepts in Version Control Systems*. 2014. CoRR, abs/1405.3496. ArXiv preprint, publicado em 14 de maio de 2014. Disponível em: <https://arxiv.org/abs/1405.3496>.

BIRD, C.; RIGBY, P.; BARR, E.; HAMILTON, D.; GERMAN, D.; DEVANBU, P. The promises and perils of mining git. *Mining Software Repositories, International Workshop on*, v. 0, p. 1–10, 05 2009.

CAVALCANTI, G.; BORBA, P.; ACCIOLY, P. Evaluating and improving semistructured merge. In: *Proceedings of the ACM on Programming Languages*. [S.l.: s.n.], 2017. (OOPSLA, OOPSLA), p. 1–27.

CAVALCANTI, G.; BORBA, P.; ANJOS, L. dos; CLEMENTINO, J. Semistructured merge with language-specific syntactic separators. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2024. Also available on arXiv as arXiv:2407.18888. Disponível em: <https://arxiv.org/abs/2407.18888>.

CAVALCANTI, G.; BORBA, P.; SEIBT, G.; APEL, S. The impact of structure on software merging: Semistructured versus structured merge. In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. San Diego, CA, USA: IEEE/ACM, 2019. Reproduziu mais de 40.000 cenarios de merge de mais de 500 projetos. Disponível em: <https://github.com/spgroup/s3m/raw/master/svj/preprint.pdf>.

CHABBI, M.; RAMANATHAN, M. K. A study of real-world data races in golang. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*. San Diego, CA, USA: ACM, 2022. p. 474–489. Disponível em: <https://arxiv.org/abs/2204.00764>.

CHENEY, D. *Why Go*. 2017. <https://dave.cheney.net/2017/03/20/why-go>. Acesso em: 20 jul. 2025.

CLEMENTINO, J.; BORBA, P.; CAVALCANTI, G. Textual merge based on language-specific syntactic separators. In: *Anais do XXXV Simpósio Brasileiro de Engenharia de Software (SBES)*. Joinville, RS, Brasil: Sociedade Brasileira de Computaçao, 2021. Disponível em: <https://sol.sbc.org.br/index.php/sbes/article/view/18820>.

Cloud Native Computing Foundation. *kubernetes: Production-grade container orchestration*. 2025. <https://github.com/kubernetes/kubernetes>. Acesso em: 20 jul. 2025.

Cloudflare, Inc. *Por que minificar JavaScript?* 2025. <https://www.cloudflare.com/pt-br/learning/performance/why-minify-javascript-code/>. Acesso em: 20 jul. 2025.

DELPEUCH, A. *Mergiraf: A Syntax-Aware Merge Driver for Git*. 2025. <https://mergiraf.org/>. Acesso em 20 de julho de 2025.

Docker Inc. *moby: A collaborative project for the container ecosystem*. 2025. <https://github.com/moby/moby>. Acesso em: 20 jul. 2025.

DUARTE, J. P.; BORBA, P.; CAVALCANTI, G. *LastMerge: A language-agnostic structured tool for code integration*. 2025. Preprint arXiv:2507.19687 [cs.SE]. Enviado 25 de julho de 2025.

GARCIA, D. *vaultwarden*. 2025. <https://github.com/dani-garcia/vaultwarden>. Acesso em: 20 jul. 2025.

GitHub, Inc. *GitHub CLI*. 2025. <https://cli.github.com/>. Acesso em: 20 jul. 2025.

KERRISK, M. *wc(1) - Linux manual page*. 2025. <https://man7.org/linux/man-pages/man1/wc.1.html>. Acesso em: 20 jul. 2025.

KHANNA, S.; KUNAL, K.; PIERCE, B. C. A formal investigation of diff3. In: ARVIND, V.; PRASAD, S. (Ed.). *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2007)*. Springer, Berlin, Heidelberg, 2007. (Lecture Notes in Computer Science, v. 4855), p. 485–496. Disponível em: <https://link.springer.com/chapter/10.1007/978-3-540-77050-3_40>.

LARSEN, S.; FALLERI, J.; BAUDRY, B.; MONPERRUS, M. Spork: Structured merge for java with formatting preservation. *CoRR*, abs/2202.05329, 2022. Accepted for publication in *IEEE Transactions on Software Engineering*. Disponível em: <https://arxiv.org/pdf/2202.05329>.

LEßENICH, O. *Adjustable Syntactic Merge of Java Programs*. Dissertação (Master's Thesis) — Saarland University, February 2012. Disponível em: <https://www.se.cs.uni-saarland.de/projects/jdime/deploy/ma_olaf.pdf>.

Meilisearch. *meilisearch*. 2025. <https://github.com/meilisearch/meilisearch>. Acesso em: 20 jul. 2025.

Meta Platforms, Inc. *Getting Started – Flow*. 2025. <https://flow.org/en/docs/getting-started/>. Acesso em: 20 jul. 2025.

Meta Platforms, Inc. *React – A JavaScript library for building user interfaces*. 2025. <https://github.com/facebook/react>. Acesso em: 20 jul. 2025.

Mozilla Developer Network. *JavaScript*. 2025. <https://developer.mozilla.org/en-US/docs/Web/JavaScript>. Acesso em: 20 jul. 2025.

MrDoob. *three.js – JavaScript 3D library*. 2025. <https://github.com/mrdoob/three.js>. Acesso em: 20 jul. 2025.

Prometheus Authors. *prometheus*. 2025. <https://github.com/prometheus/prometheus>. Acesso em: 20 jul. 2025.

Rust Foundation. *2024 State of Rust Survey Results*. 2025. Rust Blog. Acesso em: 20 Jul. 2025. Disponível em: <https://blog.rust-lang.org/2025/02/13/2024-State-Of-Rust-Survey-results/>.

Rust Project Developers. *Rust programming language*. 2025. <https://github.com/rust-lang/rust>. Acesso em: 20 jul. 2025.

Software Freedom Conservancy. *git-log Documentation*. 2025. <https://git-scm.com/docs/git-log>. Acesso em: 20 jul. 2025.

SPGroup. *miningframework: A framework for software repository mining*. 2025. <https://github.com/spgroup/miningframework>. Acesso em: 20 jul. 2025.

Stack Overflow. *Stack Overflow Developer Survey 2022*. 2022. Survey. GitHub: 87Disponível em: <https://survey.stackoverflow.co/2022>.

Stack Overflow. *Stack Overflow Developer Survey 2024*. 2024. Acesso em: 20 jul. 2025. Disponível em: <https://survey.stackoverflow.co/2024/>.

TAFT, D. K. Rust growing fastest, but javascript reigns supreme. *The New Stack*, 2024. Disponível em: <https://thenewstack.io/rust-growing-fastest-but-javascript-reigns-supreme>.

TAFT, D. K. Survey: Memory-safe rust gains 45% of enterprise development. *The New Stack*, February 2025. Disponível em: <https://thenewstack.io/survey-memory-safe-rust-gains-45-of-enterprise-development>.

TAVARES, A.; BORBA, P.; CAVALCANTI, G.; SOARES, S. Semistructured merge in javascript systems. In: *34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. [S.l.: s.n.], 2019. p. 1014–1025.

W3Techs. *JavaScript usage statistics, July 2025*. 2025. Acesso em: 20 jul. 2025. Disponível em: <https://w3techs.com/technologies/comparison/cp-javascript>.

Apêndices

APÊNDICE A – *SCRIPT* PARA AUXILIAR NA SELEÇÃO DE PROJETOS

```
1  const fs = require('fs')
2  const path = require('path')
3  const { spawn } = require('node:child_process')
4
5  const parseInput = () => {
6    const command = process.argv[2]
7
8    const projectsFilePath = process.argv[3]
9    const projectsFileContent = fs.readFileSync(projectsFilePath, 'utf-8')
10   const projects = JSON.parse(projectsFileContent)
11
12   return { command, projects, projectsFilePath }
13 }
14
15 const getRepoPath = projectName => {
16   const REPOS_DIR_PATH = path.resolve('repos')
17   return path.resolve(REPOS_DIR_PATH, projectName)
18 }
19
20 const cloneProjects = projects => {
21   projects.forEach(({ name, url }) => {
22     const repoUrl = `${url}.git`
23     const outputPath = getRepoPath(name)
24
25     const gitClone = spawn('git', ['clone', repoUrl, outputPath])
26     gitClone.stdout.on('data', data => console.log(data.toString()))
27     gitClone.stderr.on('data', data => console.error(data.toString()))
28   })
29 }
30
31 const getMergeCommits = async (projectName, sinceDate, untilDate) => {
32   return new Promise((resolve, reject) => {
33     const repoPath = getRepoPath(projectName)
34
35     const gitLogOptions = ['--no-pager', 'log', '--merges', '--pretty=%H']
36     if (sinceDate !== undefined) {
37       gitLogOptions.push(`--since="${sinceDate}"`)
```

```
38      }
39
40      if (untilDate !== undefined) {
41        gitLogOptions.push(`--until="${untilDate}"`)
42      }
43
44      const gitLog = spawn('git', gitLogOptions, { cwd: repoPath })
45      const count = spawn('wc', ['-l'])
46
47      gitLog.stdout.pipe(count.stdin)
48
49      let countOutput = ''
50      count.stdout.on('data', data => countOutput += data.toString())
51
52      let countError = ''
53      count.stderr.on('data', data => countError += data.toString())
54
55      count.on('close', statusCode => {
56        if (statusCode === 0) {
57          resolve(+countOutput)
58        } else {
59          reject(new Error(`Could not retrieve number of merge commits from $
                {projectName}: ${countError}`))
60        }
61      })
62    })
63  }
64
65  const decorateProjectsWithMergeCommits = async projects => {
66    const MIN_YEAR = 2010
67    const MAX_YEAR = 2025
68
69    for (let i = 0; i < projects.length; i++) {
70      const project = projects[i]
71
72      const mergeCommits = {}
73      mergeCommits.total = await getMergeCommits(project.name)
74
75      for (let year = MIN_YEAR; year <= MAX_YEAR; year++) {
```

```
76        let sinceDate = `${year}-01-01`
77        let untilDate = `${year}-06-30`
78        let dateRangeKey = `${year}.1`
79
80        mergeCommits[dateRangeKey] = await getMergeCommits(project.name,
              sinceDate, untilDate)
81
82        sinceDate = `${year}-07-01`
83        untilDate = `${year}-12-31`
84        dateRangeKey = `${year}.2`
85
86        mergeCommits[dateRangeKey] = await getMergeCommits(project.name,
              sinceDate, untilDate)
87      }
88
89      projects[i] = { ...project, mergeCommits }
90    }
91  }
92
93  const createProjectsCSV = projects => {
94    let csvContent = 'path,name\n'
95    projects.forEach(({ name, url }) => {
96      csvContent += `${url},${name}\n`
97    })
98
99    return csvContent
100 }
101
102 const updateFileName = (filePath, fileNamePrefix) => {
103   const dirName = path.dirname(filePath)
104   const updatedFileName = `${fileNamePrefix}${path.basename(filePath)}`
105   return path.resolve(dirName, updatedFileName)
106 }
107
108 const run = async () => {
109   const { command, projects, projectsFilePath } = parseInput()
110   if (command === 'clone') {
111     cloneProjects(projects)
112   } else if (command === 'decorate') {
```

```
113     await decorateProjectsWithMergeCommits(projects)
114     projects.sort((a, b) => b.mergeCommits.total - a.mergeCommits.total)
115
116     const outputFileName = updateFileName(projectsFilePath, 'decorated-')
117     const outputFileContent = JSON.stringify(projects)
118
119     fs.writeFileSync(outputFileName, outputFileContent)
120   } else if (command === 'csv') {
121     const csvContent = createProjectsCSV(projects)
122     const outputFileName = path.resolve(path.dirname(projectsFilePath), '
          projects.csv')
123     fs.writeFileSync(outputFileName, csvContent)
124   }
125 }
126
127 run()
```

APÊNDICE B – ARQUIVO DOCKERFILE DO EXPERIMENTO

```
1  FROM ubuntu:22.04
2
3  RUN apt-get update && apt-get install -y curl nano zip unzip git openjdk
       -17-jdk
4
5  RUN apt-get update && \
6      apt-get install ca-certificates-java && \
7      apt-get clean && \
8      update-ca-certificates -f;
9
10 ENV JAVA_HOME /usr/lib/jvm/java-17-openjdk-amd64/
11 RUN export JAVA_HOME
12
13 RUN curl -s "https://get.sdkman.io" | bash
14
15 SHELL ["/bin/bash", "-c"]
16
17 RUN source "/root/.sdkman/bin/sdkman-init.sh"   \
18                 && sdk install gradle 8.13      \
19                 && sdk install groovy 3.0.23
20
21 COPY miningframework /home/miningframework
22
23 WORKDIR /home/miningframework
24
25 RUN chmod +x dependencies/mergiraf
```