Universidade Federal de Juiz de Fora
Faculdade de Engenharia & Instituto de Ciências Exatas
Bacharelado em Engenharia Computacional

Um estudo sobre a natureza dos conflitos de merge de software na linguagem Python

Luan Reis Ciribelli

JUIZ DE FORA
AGOSTO, 2025

Um estudo sobre a natureza dos conflitos de merge de software na linguagem Python

Luan Reis Ciribelli

Universidade Federal de Juiz de Fora
Faculdade de Engenharia & Instituto de Ciências Exatas
Mecanica Aplicada Computacional
Bacharelado em Engenharia Computacional

Orientador: Gleiph Ghiotto Lima de Menezes
Coorientador: Heleno de Souza Campos Junior

JUIZ DE FORA
AGOSTO, 2025

Um estudo sobre a natureza dos conflitos de merge de software na linguagem Python

Luan Reis Ciribelli

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO FACULDADE DE ENGENHARIA & INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM ENGENHARIA COMPUTACIONAL.

Aprovada por:

Gleiph Ghiotto Lima de Menezes
Doutor em Computação

Heleno de Souza Campos Junior
Doutor em Computação

André Luiz de Oliveira
Doutor em Ciência da Computação

Leonardo Vieira dos Santos Reis
Doutor em Ciência da Computação

JUIZ DE FORA
18 DE AGOSTO, 2025

Resumo

Palavras-chave: Merge, Python, Conflito

Introdução: A crescente complexidade dos projetos de software exige intensa colaboração entre desenvolvedores, o que frequentemente resulta em conflitos durante o *merge* de código. Estudos anteriores têm investigado esses conflitos, mas geralmente se concentram em linguagens como Java, deixando uma lacuna na compreensão dos conflitos em linguagens com características sintáticas e estruturais distintas, como Python.

Métodos: Este estudo analisou conflitos de *merge* em 50 repositórios Python hospedados no GitHub. Foi desenvolvida uma abordagem automatizada que extrai e classifica os *merges* do histórico do projeto, distinguindo aqueles com e sem conflitos. Em seguida, os *chunks* conflitantes foram identificados a partir dos delimitadores de conflito. A análise incluiu a extração de métricas, como o número e o tamanho dos *chunks* (em linhas de código), a identificação dos construtos da linguagem presentes nos arquivos python e a categorização das decisões dos desenvolvedores.

Resultados: Os dados indicam que a maioria dos *merges* com falha envolve uma quantidade reduzida de *chunks* conflitantes, com mediana de um *chunk* por *merge* e 75% dos *chunks* contendo até 13 linhas de código, embora casos extremos com milhares de linhas também tenham sido observados. A análise dos construtos da linguagem revelou que elementos como chamadas de métodos e declarações de variáveis são os mais prevalentes, enquanto as estratégias de resolução mais comuns consistem na preservação completa de uma das versões (Versão 1 ou Versão 2).

Discussão: Os achados sugerem que, embora a maioria dos resultados seja semelhante entre duas linguagens distintas, é possível observar a influência de características específicas da linguagem na natureza dos conflitos de *merge*. Isso levanta a hipótese de que certos padrões de conflito podem ser, em parte, agnósticos à linguagem, enquanto outros refletem peculiaridades sintáticas ou estruturais do código em Python.

Conclusão: Este estudo amplia a compreensão sobre conflitos de *merge* em projetos Python, oferecendo uma base empírica que complementa pesquisas realizadas em outras linguagens. Os resultados fornecem subsídios para o desenvolvimento de ferramentas automatizadas de resolução de conflitos e para investigações futuras na área de *merge* colaborativo de código.

Abstract

Keywords: Merge, Python, Conflict

Introduction: The increasing complexity of software projects demands intense collaboration among developers, which often results in conflicts during code integration. Previous studies have investigated these conflicts but generally focus on languages like Java, leaving a gap in understanding conflicts in languages with distinct syntactic and structural characteristics, such as Python.

Methods: This study analyzed merge conflicts in 50 Python repositories hosted on GitHub. An automated approach was developed to extract and classify merges from the project history, distinguishing between those with and without conflicts. Subsequently, conflicting chunks were identified from the conflict delimiters. The analysis included the extraction of metrics, such as the number and size of chunks (in lines of code), the identification of language constructs present in python files, and the categorization of developer decisions.

Results: The data indicate that most failed merges involve a small number of conflicting chunks, with a median of 1 chunk per merge and 75% of chunks containing up to 13 lines of code, although extreme cases with thousands of lines occur. The analysis of language constructs revealed that elements such as method calls and variable declarations are the most prevalent, while the most common resolution strategies are the complete preservation of one of the versions (Version 1 and Version 2).

Discussion: The findings suggest that while most results are similar between two different languages, it was possible to observe the influence that unique language characteristics can have on the nature of the Merge.

Conclusion: This study expands the understanding of merge conflicts in Python projects, providing an empirical basis that complements studies conducted in other languages. The results offer a foundation for the development of future automated conflict

resolution tools and for further investigations in the area of collaborative code integration.

Agradecimentos

Agradeço aos meus pais, Elma e Kristhian, e aos meus irmãos, Diego e Davi. Sem o apoio de vocês ao longo de toda a minha vida, eu definitivamente não seria quem sou hoje. Amo muito cada um de vocês, e minha gratidão vai muito além do que consigo colocar em palavras.

Sou muito grato ao meu orientador, Gleiph, que em 2019 me deu a oportunidade de entrar no projeto e, com isso, me apresentou a esse universo acadêmico. Sem sua confiança e orientação, este trabalho — e tantos outros — não teriam acontecido.

Também agradeço ao meu coorientador, Heleno, que desde que chegou ao grupo trouxe conhecimentos que ajudaram a levar nossas pesquisas a outro nível. Estendo meu agradecimento ao João Pedro Carvalho por todo o apoio nos trabalhos que fizemos juntos.

Aos meus amigos, obrigado por tornarem a vida na faculdade muito mais leve e divertida.

Por fim, agradeço à Universidade Federal de Juiz de Fora, que, com excelentes professores e incentivo constante à pesquisa, contribuiu demais para minha formação.

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

A medida que os projetos de software se tornam mais complexos, a colaboração entre desenvolvedores desempenha um papel crucial na progressão contínua desses projetos. No entanto, essa colaboração muitas vezes resulta em conflitos durante o processo de *merge* de diferentes versões do código-fonte. Conforme evidenciado na literatura, aproximadamente 10 a 20% (BRUN et al., 2011; KASI; SARMA, 2013) dos *merges* (MENS, 2002) resultam em conflitos. Esses conflitos surgem quando desenvolvedores editam a mesma região de um artefato de *software* em paralelo. Diversos estudos (MENS, 2002; ZIMMERMANN, 2007; LESSENICH; APEL; LENGAUER, 2015; BRUN et al., 2011; KASI; SARMA, 2013; CAVALCANTI; ACCIOLY; BORBA, 2015; YUZUKI; HATA; MATSUMOTO, 2015; AHMED et al., 2017; SANTOS; MURTA, 2012) investigam as principais causas desses conflitos, como evitá-los e quais características dos repositórios podem influenciar sua ocorrência. No entanto, grande parte dessas pesquisas adota uma abordagem generalista, sem considerar as particularidades das linguagens de programação envolvidas, ou concentra-se exclusivamente em projetos escritos em Java (SHEN et al., 2023; AHMED et al., 2017; GHIOTTO et al., 2020), uma linguagem orientada a objetos. Esse foco limitado desconsidera a diversidade de outras linguagens, cujas gramáticas e paradigmas podem influenciar a dinâmica dos conflitos e as estratégias utilizadas para resolvê-los, evidenciando a necessidade de ampliar o escopo das investigações para uma compreensão mais abrangente do problema.

Neste trabalho — que dá continuidade ao Trabalho de Conclusão de Curso apresentado no âmbito das Ciências Exatas (CIRIBELLI; GHIOTTO; CAMPOS, 2022) — é proposta uma abordagem para aprofundar o entendimento dos conflitos de *merge* em projetos de software escritos em Python. Para isso, são analisados os padrões de resolução de conflitos em 50 repositórios de código-fonte, guiados pelas seguintes questões de pesquisa: (i) “qual é a distribuição do número de *chunks* conflitantes por *merge* em conflito?”(ii) “qual é a distribuição do tamanho desses *chunks*, medidos em linhas de código (LOC)?”(iii)“ qual é a distribuição dos construtos da linguagem envolvidos nos *chunks*

conflitantes?" e (iv) "qual é a distribuição referente às decisões de resolução adotadas pelos desenvolvedores?" Os resultados obtidos fornecem *insights* valiosos para a compreensão da natureza dos conflitos de *merge* e para a identificação de estratégias eficazes de resolução.

Os resultados obtidos neste estudo indicam que certos padrões observados em conflitos de *merge* — como a quantidade reduzida de *chunks* por *merge*, o tamanho dos *chunks* em linhas de código e a predominância de um a dois construtos da linguagem por *chunk*— são também evidenciados em estudos prévios com outras linguagens, como Java. Por exemplo, 75% dos *merges* com conflitos em Python apresentaram até três *chunks*, número comparável aos 76% relatados em Java (GHIOTTO et al., 2020). Da mesma forma, 76% dos *chunks* em Python envolveram até dois construtos da linguagem, valor similar ao encontrado no estudo em Java. Esses paralelos, em linguagens de diferentes paradigmas, sugerem que alguns aspectos dos conflitos de *merge* podem ser independentes da linguagem utilizada, refletindo características mais gerais do processo de desenvolvimento colaborativo. Tais achados reforçam o potencial para o desenvolvimento de estratégias e ferramentas de resolução de conflitos que sejam menos sensíveis à linguagem de programação e sim as características agnósticas de linguagens observadas, aumentando sua aplicabilidade em ambientes heterogêneos.

Este trabalho está organizado da seguinte forma: no Capítulo 2, é apresentado a fundamentação teórica, abordando conceitos relacionados a Sistemas de Controle de Versão (SCV) e análise de código-fonte. No Capítulo 3, é discutido os trabalhos relacionados. No Capítulo 4, é detalhado a abordagem metodológica adotada. No Capítulo 5, é apresentado os resultados obtidos, discutindo padrões observados e suas implicações. Por fim, no Capítulo 6, é concluído o trabalho, destacando suas contribuições, limitações e possíveis direções para pesquisas futuras.

2 Fundamentação Teórica

Neste capítulo são abordados aspectos fundamentais relacionados ao controle de versão de software e análise de código-fonte. Na Seção 2.1 é explorada a importância dos Sistemas de Controle de Versão (SCV) na colaboração entre desenvolvedores, destacando conceitos como *commit*, ramos e resolução de conflitos de *merge*. Na Seção 2.2 é discutida a relevância da análise de código-fonte para coletar estruturas de linguagens de programação, com foco na linguagem Python, além de apresentar o *ANTLR*, uma ferramenta utilizada para fazer este tipo de análise.

2.1 Controle de Versão e Análise de Conflitos

Os SCV são amplamente utilizados para facilitar a colaboração entre desenvolvedores de software, permitindo a criação de novas versões ao concluir uma tarefa e a navegação entre essas versões conforme necessário. Exemplos desses SCVs incluem o *Git* (CHACON; STRAUB, 2014) e o *Subversion* (FITZPATRICK; COLLINS-SUSSMAN; PILATO, 2008). Esses sistemas incorporam conceitos e funcionalidades, como *commits* e ramos (*branches*), para acompanhar o histórico de desenvolvimento de um projeto, registrar mudanças individuais e integrar versões por meio de *merges*.

Durante o desenvolvimento de um software, os desenvolvedores efetuam adições, remoções e modificações nos artefatos, como o código-fonte e a documentação, para introduzir novas funcionalidades ou corrigir *bugs*, resultando na criação de novas versões chamadas de *commits*. Esses *commits* armazenam diversas informações cruciais para a manutenção do histórico do projeto, como a data da sua realização, o autor das alterações, o conteúdo modificado e uma mensagem descritiva.

Cada *commit* pode estar ligado a um ou mais *commits* anteriores, denominados seus ancestrais, que representam versões anteriores do projeto. O ancestral mais próximo de um *commit* é chamado de pai, e, em casos de *merge* entre diferentes ramos do código, um *commit* pode possuir múltiplos pais. Quando há necessidade de identificar um ponto

de referência comum entre duas versões distintas do projeto, utiliza-se o conceito de ancestral comum, que representa a última versão compartilhada antes das modificações divergirem. Esse conceito é essencial para operações como *merge*, em que as mudanças realizadas em diferentes ramos são combinadas em um *commit*.

Cada um dos círculos na Figura 2.1 representa um *commit*, identificado por um nome único, que representa uma versão específica do projeto. Esses *commits* funcionam como "fotografias"do estado do projeto em um ponto no tempo, permitindo que os desenvolvedores acompanhem e gerenciem as alterações ao longo do processo de desenvolvimento. As setas mostram as ligações de cada *commit* a seus respectivos pais, por exemplo, "C2"é pai de "C4".

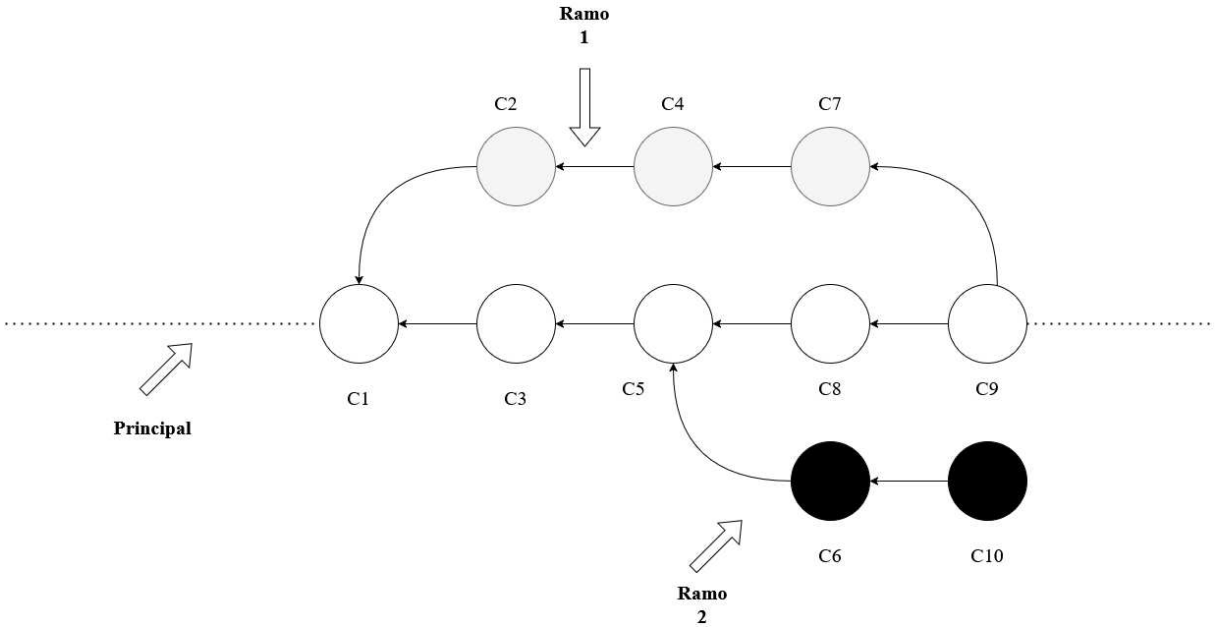


Figura 2.1: Imagem ilustrativa do processo de *merge*. Fonte: o autor

Para gerenciar essas modificações organizadamente, os desenvolvedores trabalham em diferentes ramos (do inglês, *branches*). Os ramos podem implementar novos recursos, corrigir *bugs* ou realizar outras tarefas de desenvolvimento (BERCZUK; APPLETON, 2020). Cada ramo é representado na Figura 2.1 por *commits* de cores diferentes, representando uma linha de desenvolvimento independente, permitindo que os desenvolvedores trabalhem em paralelo sem interferir no trabalho uns dos outros. A Figura 2.1 possui três ramos diferentes, o "*ramo 1*" representado pelos círculos cinza-claro, o ramo *Principal* representado pelos círculos brancos e o "*ramo 2*" representado pelos círculos pretos. As alterações feitas em cada um dos ramos são isoladas das outras, cada ramo compartilha

o histórico até o *commit* no qual ocorre a bifurcação e depois possuem suas próprias histórias. Por exemplo, na Figura 2.1, "C1" é o antecedente de ambos "C2" e "C3", logo, ambos os ramos mantêm o histórico de "C1". Porém, após divergirem, cada um possui suas próprias alterações independentes enquanto mantém em sua base a história de "C1".

Quando é necessário integrar o código de uma ramificação em outra, utiliza-se a funcionalidade de *merge*. Conforme destacado por Mens (2002), o *merge* é essencial para integrar linhas de desenvolvimento paralelas em sistemas de larga escala, permitindo que modificações feitas isoladamente em ramificações sejam combinadas em uma versão unificada. Na Figura 2.1, esse processo pode ser observado quando o Ramo 1 é reintegrado à linha principal por meio do *merge* do *commit* "C7" com o *commit* "C8" no ramo principal, gerando o *commit* "C9".

Durante o *merge*, o SCV identifica as alterações feitas em cada ramo, com base no *commit* ancestral comum entre os dois ramos, e tenta combinar automaticamente as modificações. Essa combinação pode ter sucesso ou não dependendo se há alterações concorrentes numa mesma região de código. Caso a combinação automática não funcione, ocorre um *conflito textual de merge* (MENS, 2002). Como esse será o único tipo de *conflito* tratado nesse trabalho, será tratado somente como conflito de *merge*.

Esses conflitos de *merge* ocorrem em 10% a 20% dos *merges* (BRUN et al., 2011; KASI; SARMA, 2013). As seções de código envolvidas nesses conflitos são conhecidas como *chunks*. O exemplo apresentado na Listagem 2.1 ilustra um conflito de *merge* que contém o *chunk* resultante de um conflito ocorrido no Repositório "scrapinghub/portia".

Na Listagem 2.1 pode-se observar um *chunk* gerado utilizando o *merge* do SCV *Git*. A seção delimitada pelos marcadores "`<<<<<<<` HEAD" e "`=======`" representa a Versão 1 (V1) do código. A seção delimitada por "`=======`" e "`>>>>>>>` *hash*" representa a Versão 2 (V2), sendo que *hash* corresponde a uma sequência alfanumérica que identifica de forma única o commit associado. Esse *hash* é fixo para cada *merge* específico, mas pode variar entre diferentes *merges*, uma vez que reflete o commit de origem de cada trecho em conflito. O texto que cerca os marcadores é chamado de contexto, que corresponde ao código comum entre as duas versões. Na Listagem 2.1, a Versão 1 corresponde ao trecho das linhas 75 a 78 (inclusive), e a Versão 2 na linha 80. As linhas 71 a 73 e 82 a 84

```
71          disallow = set(disallow)
72      else:
73          disallow = []
74  <<<<<<< HEAD
75      id1, id2 = id_to_int(id1), id_to_int(id2)
76      if id1 == id2:
77          id2 = ~id2 - 1
78      seed = id1 ^ id2
79  =======
80      seed = id_to_int(id1) ^ id_to_int(id2)
81  >>>>>>> 26b2d7e784dddaa0a8acef4fc687cc21637768de
82      generator = Random(seed)
83
84      def format_id():
```

Listagem 2.1: Exemplo de um trecho (*chunk*) em conflito em Python. Repositório: scrapinghub/portia. *Merge* com hash: 4c4819663e31bc8b882c2f66654a59be0118b8cb

representam o contexto.

Uma vez detectados os conflitos de *merge*, são necessárias técnicas para resolvê-los. Estas podem variar desde um processo manual — frequentemente demorado — passando por um processo interativo, no qual o o desenvolvedor de software precisa resolver todos os *chunks* individualmente, até uma ferramenta de resolução de conflitos totalmente automatizada (MENS, 2002). Ghiotto et al. (2020) propuseram as seguintes classificações para as resoluções de conflitos de *merge*:

- Versão 1: resolução que aceita o conteúdo da Versão 1, descartando todo o conteúdo da Versão 2.
- Versão 2: resolução que mantém o conteúdo da Versão 2, removendo o conteúdo da Versão 1.
- Nenhum: resolução que rejeita o conteúdo das versões 1 e 2, deixando o código resultante em branco.
- Concatenação: resolução que concatena ambas as versões, mantendo o texto integral de ambas as versões, em qualquer ordem.
- Combinação: resolução que incorpora, em uma ordem de intercalação, algumas linhas de código selecionadas de ambas as versões, sem escrever novas linhas ou

modificar as linhas selecionadas. Ressalta-se que essa classificação é aplicada apenas quando a resolução não se enquadra nos critérios de Versão 1, Versão 2 ou Concatenação, os quais possuem precedência na hierarquia de classificação.

- Novo Código: resolução que adiciona código que não existia na Versão 1 ou na Versão 2.

2.2 Análise de código-fonte

Como observado no *chunk* na Listagem 2.1, os *chunks* podem conter estruturas da linguagem de programação e representam seções de código que incluem diferentes instruções e elementos sintáticos da linguagem. Em uma linguagem de programação, as estruturas da linguagem referem-se aos elementos básicos que compõem a sintaxe da linguagem e definem a sua gramática. Essas estruturas incluem declarações de variáveis, condicionais, laços, funções, entre outros. Essas informações pertinentes ao conteúdo dos *chunks* não são analisadas pelo *merge* textual, mas podem ser exploradas por outras ferramentas, como analisadores de código-fonte capazes de identificar e interpretar tais estruturas.

Na análise de código-fonte, é comum utilizar gramáticas para descrever a estrutura de linguagens de programação. De forma simples, uma gramática é um conjunto de regras que define as instruções para gerar uma linguagem. No contexto da computação, essas regras servem para definir a estrutura de programas, expressões e comandos. Segundo Sipser (2012), uma gramática é composta por regras de substituição, chamadas produções, que dizem como símbolos podem ser trocados por outros para gerar frases de uma linguagem. Essas regras envolvem variáveis (também chamadas de *não-terminais*, que representam categorias sintáticas a serem expandidas) e símbolos terminais (os elementos básicos da linguagem, que não podem mais ser substituídos). Uma dessas variáveis é escolhida como ponto de partida. A partir dessas substituições, é possível gerar as sentenças válidas da linguagem.

Em uma gramática, cada regra de produção define como uma construção sintática pode ser formada, geralmente utilizando outras construções sintáticas ou símbolos terminais (*tokens*). No exemplo, apresentado na Listagem 2.2, é mostrada uma regra escrita

```
funcdef
    : 'def' name parameters ('->' test)? ':' block
    ;
```

Listagem 2.2: Exemplo de gramática do Python retirada do Repositório de gramáticas do ANTLR.

na notação da ferramenta ANTLR. Nessa notação, palavras-chave são representadas entre aspas simples, e partes optativas são indicadas por um ponto de interrogação. Na gramática em questão, são especificadas as regras para a formação de uma declaração de método. Nesse caso, uma função deve conter a palavra-chave *'def'*, seguida de um nome e dos parâmetros. Tanto o *'name'* quanto os *'parameters'* são regras definidas na gramática. Em seguida, há o comando opcional de uma seta com uma regra de teste `('->' test)?`, seguido por dois pontos (':'), que é um *token*, e, finalmente, um bloco de código ('block'), que também é uma regra definida na gramática.

Os construtos da linguagem (do inglês, *Language Constructs*) referem-se aos comandos definidos pela gramática de uma linguagem de programação. Em Python, alguns exemplos de construtos da linguagem incluem "*if*", "*for*", "*while*", "*def*", entre outros. Cada um desses comandos tem um significado específico na linguagem e é essencial para expressar diferentes algoritmos e fluxos de controle.

A gramática da linguagem Python[1] possui características distintas em relação a outras linguagens de programação amplamente utilizadas, como Java, C++ e JavaScript. Uma de suas principais particularidades é o uso da indentação para delimitação de blocos de código, substituindo o uso de chaves, comum em linguagens como C++ e Java (KASWAN; DHATTERWAL; BALAMURUGAN, 2023).

Para realizar a análise de código-fonte em Python, uma ferramenta amplamente utilizada é o *ANTLR (Another Tool for Language Recognition)* (PARR, 2013). ANTLR v4 é um gerador de analisadores que pode ser usado para ler, processar, executar ou traduzir texto estruturado, ou arquivos binários. Ele facilita a análise gramatical de código-fonte Python, possibilitando a extração de informações estruturais.

[1] https://docs.python.org/3/reference/grammar.html

3 Trabalhos Relacionados

A ocorrência de conflitos durante o *merge* de código tem sido amplamente estudada. Diferentes abordagens analisam a relação entre a frequência das modificações, a complexidade estrutural do código e as práticas adotadas no desenvolvimento colaborativo.

Yuzuki, Hata e Matsumoto (2015) realizaram uma investigação em dez projetos escritos em Java e *open-source* e observaram que conflitos de *merge* ocorrem predominantemente devido a edições concorrentes na mesma região de um método, remoções integrais de métodos e operações de renomeação. Esses achados indicam que a granularidade das alterações é um fator crítico, aumentando a probabilidade de conflitos conforme trechos do código são simultaneamente modificados.

No contexto da taxa de integração, Nguyen e Ignat (2017) analisaram quatro repositórios, dois em C e um em Pearl e um em Ruby, e verificaram que repositórios com menor taxa de integração — definida como a proporção de arquivos alterados concorrentemente em relação ao total de arquivos modificados — tendem a apresentar índices mais elevados de conflitos. Esse resultado sugere que tanto o volume quanto a distribuição das mudanças influenciam diretamente a incidência de conflitos.

Além disso, Leßenich et al. (2018) propuseram um conjunto de indicadores quantitativos, incluindo o número de *commits* realizados em períodos específicos e o volume de alterações simultâneas. Embora isoladamente esses fatores não apresentem forte correlação com falhas no *merge*, sua análise combinada permite identificar padrões nos conflitos.

Fatores estruturais do código também impactam a formação de conflitos. Ahmed et al. (2017) analisaram 163 projetos Java e 6.979 *merges* com falhas, evidenciando que a presença de *code smells* torna o código mais suscetível a conflitos. Em particular, métodos com alta complexidade, duplicações internas e operações excessivas apresentaram maior propensão a falhas semânticas.

Ribeiro, Costa e Santos (2022) investigaram os fatores que influenciam a ocorrência de conflitos de *merge* sob a perspectiva de desenvolvedores brasileiros. Por meio de uma

pesquisa com 109 desenvolvedores somado a entrevistas com 15 profissionais, os autores identificaram que a falta de comunicação e o tempo prolongado de isolamento de *branches* são os principais fatores que contribuem para a incidência de conflitos. Além disso, práticas como a melhoria da comunicação entre os membros da equipe, a redução da duração dos *branches* e a realização de *commits* frequentes foram destacadas como estratégias eficazes para mitigar conflitos. O estudo também aponta que a experiência dos desenvolvedores e a evolução das ferramentas de controle de versão desempenham um papel fundamental na prevenção e gestão de conflitos.

Cunha, Accioly e Borba (2022) analisaram a ocorrência de conflitos de *merge* em repositórios locais de desenvolvedores, destacando a importância de considerar ações que não são visíveis em repositórios remotos compartilhados. A pesquisa envolveu a análise de 95 arquivos de *log*, que continham os comandos de *git* locais utilizados, de 61 desenvolvedores. Esses *logs* revelam que cenários de *merge* ocultos, como aqueles realizados por meio dos comandos *git rebase*, *cherry-pick* e *squash*, ocorrem aproximadamente seis vezes mais do que os cenários de *merge* explícitos, como aqueles conduzidos por *git merge*. Além disso, os resultados indicam que *merges* apresentam uma taxa de conflitos de até 37% nos repositórios locais, enquanto nos cenários ocultos a taxa mínima de conflitos observada foi de 10%. Esses conflitos, resolvidos localmente, não são registrados em repositórios remotos, sugerindo que estudos baseados exclusivamente em dados públicos podem subestimar a real frequência dos conflitos. A pesquisa também aponta que a escolha de comandos de *merge* é influenciada tanto pela experiência dos desenvolvedores quanto por diretrizes organizacionais, sendo que desenvolvedores menos experientes tendem a preferir comandos mais simples, como *merge*.

Shen et al. (2023) realizaram um estudo abrangente sobre a natureza dos conflitos de *merge* em projetos *open-source* escritos em Java, analisando 208 repositórios e identificando 117.218 cenários de *merge*, dos quais 15.886 apresentaram conflitos textuais, 79 resultaram em falhas de compilação e 33 em falhas de teste. Os resultados indicaram que a maioria dos conflitos textuais foi resolvida mantendo integralmente uma das versões, enquanto conflitos mais complexos, como os de compilação e teste, exigiram a combinação de edições acompanhadas de ajustes adicionais para corrigir erros estruturais

e semânticos. Além disso, os autores evidenciaram que ferramentas atuais são limitadas na detecção e resolução de conflitos além do nível textual, sendo incapazes de identificar com precisão a causa raiz de falhas em compilação e testes.

Num estudo anterior,Ciribelli et al. (2022), foi implementada uma ferramenta cujo objetivo é auxiliar pesquisadores na análise de conflitos de *merge* em projetos de software. A ferramenta consegue recriar *merges* em repositórios *Git* e extrair informações relevantes sobre conflitos, como decisões dos desenvolvedores e construções linguísticas envolvidas nos conflitos, para projetos em Java, C++ e Python. A ferramenta foi comparada aos resultados de estudos anteriores e teve a reprodutibilidade de 87,75% das decisões dos desenvolvedores e 67,27% das construções linguísticas de estudos anteriores. Além disso, em uma análise de 100 *merges* de projetos Python, a ferramenta obteve 100% de precisão na classificação das decisões dos desenvolvedores, permitindo comparações entre diferentes linguagens de programação.

Ghiotto et al. (2020) conduziram um extenso estudo empírico sobre a natureza dos conflitos de *merge* em projetos *open-source* escritos em Java, analisando um total de 2.731 repositórios. A partir de uma análise manual inicial em cinco projetos, os autores desenvolveram ferramentas para realizar uma análise automatizada de 960.366 casos de *merge*, dos quais 25.328 resultaram em conflitos. O estudo coletou dados detalhados sobre os *chunks* conflitantes — incluindo quantidade, tamanho (em linhas de código), construtos sintáticos envolvidos e estratégias manuais de resolução adotadas pelos desenvolvedores. Os resultados indicaram, por exemplo, que 87% dos *chunks* conflitantes continham apenas linhas já presentes nas versões conflitantes, e que em 94% desses casos, cada versão tinha menos de 50 linhas. Além disso, os autores observaram padrões recorrentes na forma como desenvolvedores e projetos distintos tendem a resolver conflitos, e identificaram dependências entre *chunks* em 14% a 46% dos *merges* com múltiplos conflitos. Com base nesses achados, o estudo propõe direções para aprimorar ferramentas de *merge*, como apresentar ordens sugeridas de resolução ou aproveitar o histórico de resoluções passadas para auxiliar decisões futuras.

Embora os estudos existentes tenham contribuído significativamente para a compreensão dos conflitos de *merge*, ainda há lacunas importantes a serem exploradas, especi-

almente no que diz respeito à influência das características da linguagem de programação na natureza dos conflitos e nas estratégias de resolução adotadas. Diferentemente das abordagens anteriores, que se concentram majoritariamente em Java ou em aspectos mais gerais dos conflitos, este trabalho investiga os padrões de conflitos de *merge* na linguagem Python, considerando suas particularidades sintáticas e estruturais. Nossa hipótese é de que a sintaxe baseada em indentação, o suporte a múltiplos paradigmas de programação e a não obrigatoriedade de declarações explícitas de tipos são fatores que podem impactar tanto a incidência quanto a resolução de conflitos nessa linguagem. Dessa forma, esta pesquisa propõe ampliar o escopo da literatura ao oferecer uma análise detalhada sobre como conflitos de *merge* ocorrem e são resolvidos em projetos Python, contribuindo para um entendimento mais abrangente do problema.

4 Abordagem

Neste capítulo é apresentada a abordagem adotada neste trabalho. Na Seção 4.1 é apresentada uma visão geral da abordagem em alto nível. Na Seção 4.2 é discutido como identificar os *chunks* em *merges* em conflito e é apresentado o exemplo de um *chunk*. Na Seção 4.3 é apresentado como as informações dos *chunks*, como número de linhas, decisão do desenvolvedor e construto da linguagem, são extraídas e, por fim, na Seção 4.4 é introduzida a implementação da ferramenta que reexecuta os *merges* e coleta informações relacionadas aos conflitos e as ferramentas auxiliares utilizadas.

4.1 Visão Geral

Resolver conflitos de *merge* é um dos desafios no desenvolvimento colaborativo de software (BRUN et al., 2011; KASI; SARMA, 2013). A análise de como esses conflitos surgem e são resolvidos oferece *insights* valiosos para entender o impacto das decisões dos desenvolvedores e, potencialmente, evitar conflitos futuros ou fornecer subsídios para a criação de ferramentas que possam auxiliá-los na resolução de conflitos. Como discutido no Capítulo 3, os trabalhos propostos na literatura investigam as principais causas desses conflitos, como evitá-los e quais características dos repositórios podem influenciar sua ocorrência com um foco na linguagem Java (SHEN et al., 2023; AHMED et al., 2017; GHIOTTO et al., 2020), sobrando lacunas a serem exploradas em outras linguagens de programação. Deste modo, este trabalho analisa os conflitos de *merge* em arquivos escritos na linguagem Python e investiga se suas particularidades influenciam a dinâmica dos *merges*.

A abordagem proposta, representada na Figura 4.1, recebe como entrada um repositório. A partir dele, todos os *merges* presentes no histórico do projeto são identificados e recriados. Essa etapa é fundamental porque os Sistemas de Controle de Versão (SCV) detectam conflitos somente no momento da execução do *merge*, não armazenando informações detalhadas sobre as regiões afetadas, apenas o resultado das soluções dos con-

flitos. Sem essa recriação, não seria possível identificar quais trechos do código geraram conflitos, como foram delimitados pelo SCV e de que forma essas regiões foram estruturadas no arquivo resultante. Ao recriar os *merges*, garante-se uma análise detalhada da ocorrência e da natureza dos conflitos. Cada *merge* é analisado e classificado em duas categorias: com ou sem conflito. Os *merges* sem conflito têm somente suas informações básicas extraídas, como os identificadores dos *commits*, autores, datas e mensagens, sendo descartados das etapas subsequentes. Por outro lado, os *merges* em conflito passam por uma análise aprofundada.

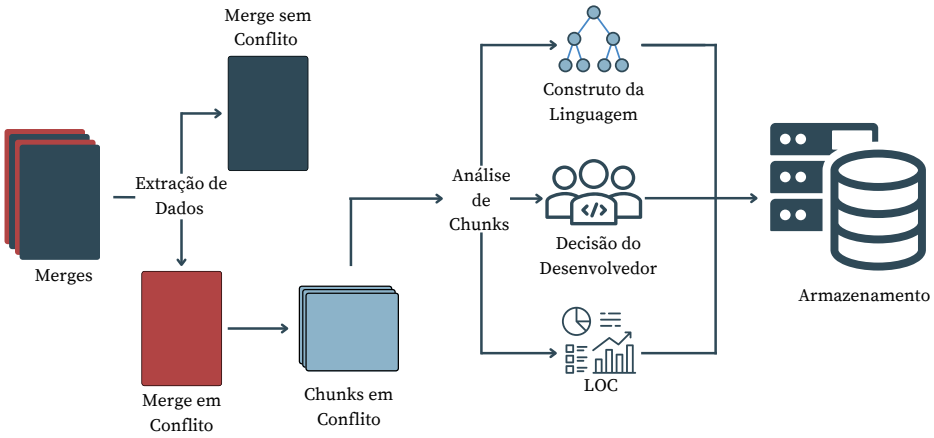


Figura 4.1: Fluxo geral da abordagem adotada, Fonte: Autor

A análise aprofundada compreende, para cada *merge* com conflito, a extração dos mesmos dados coletados nos *merges* sem conflito e, em seguida, os arquivos envolvidos em conflitos são identificados e analisados. Verifica-se a presença de *chunks* conflitantes, pois nem todos os arquivos em um *merge* com conflito contêm regiões explicitamente demarcadas como conflitantes. Somente os arquivos que possuem *chunks* são considerados para as próximas etapas da análise.

Os *chunks* conflitantes são extraídos com suas respectivas linhas de código e o contexto circundante, permitindo compreender melhor a estrutura dos conflitos. Além disso, utilizando um analisador gramatical, identificam-se os construtos da linguagem presentes nos *chunks*, como estruturas condicionais, chamadas de métodos e declarações

de variáveis. Por fim, analisa-se a decisão do desenvolvedor para resolver o conflito, classificando-a em categorias como Versão 1, Versão 2, Concatenação, Combinação ou Novo Código.

4.2 Identificação de *chunks* em *merges* em conflito

Após identificar os *merges* em conflito no repositório, esta etapa concentra-se na detecção dos *chunks* correspondentes. Para esta análise, são considerados apenas *merges* com dois pais, sendo excluídos tanto os *merges* do tipo *octopus*[2] quanto aqueles que, ao falharem, não apresentam *chunks*. A exclusão do *octopus* justifica-se pelo fato de essa estratégia não gerar *chunks*. Uma vez selecionados os *merges* relevantes, cada caso é analisado de forma detalhada, com a extração de metadados como identificadores únicos, autores, datas e mensagens dos respectivos *commits*.

Para exemplificar o processo, utilizou-se o repositório *faceswap*[3]. Os identificadores do pai 1 e do pai 2 são respectivamente `7dd1122` e `a046248`. Esse *merge* gerou conflitos em um total de 7 arquivos, conforme apresentado na Listagem 4.1.

Na Listagem 4.1 é possível observar que foi detectado sete arquivos em conflitos, que foram destacados com a cor vermelha na Listagem 4.1. Como resultado, o desenvolvedor precisa intervir manualmente para resolver cada um dos conflitos presentes em cada arquivo antes de concluir o *merge*. Dentre os sete arquivos listados, cada um contém alterações concorrentes feitas nos dois ramos que estão no processo de *merge*.

Os *merges* classificados como conflitantes passam por uma extração dos arquivos em conflito. Nos arquivos identificados, procede-se à detecção dos *chunks* conflitantes.

A Listagem 4.2 ilustra um exemplo real de *chunk* conflitante extraído do arquivo em conflito "lib/utils.py", um dos arquivos da Listagem 4.1, esse código representa a impressão de 4 opções de *input* para o usuario na V1 e apenas 3 opções na V2. Nesse caso, o conflito ocorre devido a diferenças na forma como a entrada do usuário é tratada e validada em cada versão.

[2] ⟨https://git-scm.com/docs/merge-strategies⟩
[3] https://github.com/deepfakes/faceswap

```
MINGW64 faceswap ((7dd1122...))
$ git merge a046248389b96a5a8dcd7fcf19b9668031f83f51
Auto-merging lib/cli/launcher.py
CONFLICT (content): Merge conflict in "lib/cli/launcher.py"
Auto-merging lib/model/initializers.py
CONFLICT (content): Merge conflict in "lib/model/initializers.py"
Auto-merging lib/model/layers.py
CONFLICT (content): Merge conflict in "lib/model/layers.py"
Auto-merging lib/model/normalization/normalization_common.py
CONFLICT (content): Merge conflict in "lib/model/normalization/
   normalization_common.py"
Auto-merging lib/model/optimizers_tf.py
CONFLICT (content): Merge conflict in "lib/model/optimizers_tf.py
   "
Auto-merging lib/utils.py
CONFLICT (content): Merge conflict in "lib/utils.py"
Auto-merging plugins/train/model/_base.py
CONFLICT (content): Merge conflict in "plugins/train/model/_base.
   py"
Auto-merging setup.py
Automatic merge failed; fix conflicts and then commit the result.
```

Listagem 4.1: Resultado do *merge* 03a8b62.

```
91
92          print("First time configuration. Please select the
   required backend")
93          while True:
94  <<<<<<< HEAD
95              selection = input("1: AMD, 2: CPU, 3: NVIDIA, 4:
   Apple: ")
96              if selection not in ("1", "2", "3", "4"):
97                  print("'{}' is not a valid selection. Please
   try again".format(selection))
98  =======
99              selection = input("1: AMD, 2: CPU, 3: NVIDIA: ")
100             if selection not in ("1", "2", "3"):
101                 print(f"'{selection}' is not a valid selection
   . Please try again")
102 >>>>>>> a046248389b96a5a8dcd7fcf19b9668031f83f51
103                 continue
104             break
105         fs_backend = self._backends[selection].lower()
```

Listagem 4.2: Exemplo de *chunk* conflitante extraído do repositório *faceswap* (*merge* 03a8b6228e2093b5b1436631e81630c46da6b875)

4.3 Extração de dados dos *chunks*

Os *chunks* são estruturas que possuem diversas informações implícitas como: os construtos da linguagem, as decisões dos desenvolvedores ao solucioná-los e a quantidade de linhas de código. Essas informações são coletadas na abordagem para descrever a natureza dos conflitos de *merge* presentes em arquivos escritos na linguagem Python.

Os analisadores de código-fonte podem reconhecer, a partir da regra gramatical, construtos da linguagem como, por exemplo, estruturas condicionais ou definição de método, como visto na Listagem 2.2. Conforme descrito na Seção 2.2, a análise de código-fonte depende de gramáticas que são definidas baseadas na sintaxe de cada linguagem de programação, inviabilizando a aplicação de uma única gramática para todas as linguagens. Por esse motivo, arquivos que não são escritos na linguagem Python são descartados da análise de construtos, permitindo que a análise se concentre exclusivamente nos padrões de conflito dessa linguagem.

Para garantir que os construtos fossem corretamente identificados, foi necessário utilizar um analisador de código-fonte capaz de reconhecer as estruturas sintáticas particulares do Python. A Listagem 4.2 ajuda a visualizar esse processo, apresentando um conflito no qual podem ser observadas estruturas condicionais (`IfStatement`), localizadas nas linhas 96 e 100, e utilização de variáveis (`Variable`), encontradas na atribuição nas linhas 95 e 99 seguido de uma declaração de método nas mesmas linhas com o "*input*".

Durante a análise dos construtos presentes nos conflitos de *merge*, foi adotado o critério denominado *outmost*, criado para guiar a extração das estruturas sintáticas relevantes. Esse critério consiste em identificar apenas os construtos mais externos dentro da hierarquia da árvore sintática gerada pelo analisador, desde que estejam completamente envolvidos no trecho em conflito. Por exemplo, quando um bloco condicional completo está inteiramente presente no conflito, ele é extraído como um único construto, desconsiderando os elementos mais internos como variáveis ou chamadas de função. No entanto, se apenas parte da estrutura condicional estiver no conflito, o construto é extraído juntamente com todos os seus construtos internos. Essa abordagem permite reduzir a granularidade da análise, evitando a coleta excessiva de elementos internos e focando nas estruturas de maior abrangência que apresentam o núcleo do conflito.

Voltando à Listagem 2.1, é possível observar um exemplo claro da aplicação do critério *outmost*. O bloco condicional iniciado na linha 76 com o comando "`if id1 == id2 :`"está completamente contido dentro dos delimitadores de conflito, o que permite a extração da estrutura condicional de forma isolada. Nesse caso, os elementos internos — como a atribuição à variável *id2* — não são extraídos separadamente, pois já estão encapsulados pelo construto mais externo. Já na Listagem 4.2, embora também haja uma estrutura condicional dentro do conflito, é importante notar que o bloco *if*, localizado nas linhas 96 e 100, não está completo: sua instrução de controle de fluxo *continue*, que pertence semanticamente ao corpo do *if*, encontra-se fora dos delimitadores de conflito. Por essa razão, a estrutura condicional como um todo não é considerada um construto externo completo. Nesse caso, o critério *outmost* permite a extração de elementos internos, como o comando *print*, uma chamada de função, que está integralmente dentro do trecho em conflito e representa a menor unidade sintática relevante disponível.

A decisão adotada pelo desenvolvedor para resolver um conflito de *merge* é identificada por meio da análise dos *chunks* conflitantes, incluindo o contexto, e da solução que o desenvolvedor criou. Esse processo envolve a comparação textual entre o estado do código antes e depois da resolução do conflito, permitindo determinar qual ação foi tomada. Essa comparação é feita ao se análisar o resultado do conflito, identificando os arquivos que contém os *chunks*. Nesses *chunks*, são observado as linhas de contexto que cercam o conflito, em seguida essas mesmas linhas são buscadas no arquivo gerado pela solução do conflito, e caso o contexto exista é verificado quais trechos foram preservados, modificados ou removidos.

As linhas de código (LOC) extraídas correspondem ao total de linhas do trecho, desconsiderando apenas os marcadores de conflito. Ressalta-se que todas as linhas são contabilizadas, incluindo linhas em branco. Nesse contexto, podem ser considerados o total de linhas de cada versão (V1 ou V2), bem como o total de linhas do *chunk*. Retomando o exemplo da Listagem 2.1, o tamanho de V1 corresponde a 4 linhas, o de V2 a 1 linha, e o tamanho total do *chunk* a 5 linhas.

Além da análise do conteúdo do *chunk* conflitante, é essencial considerar o contexto do *chunk*, ou seja, as linhas que antecedem e sucedem o trecho em conflito. Essas

linhas ajudam a delimitar o escopo das alterações e permitem compreender as informações como a decisão do desenvolvedor. A categorização das decisões é feita a partir dessa análise, sendo possível classificar cada resolução em uma das categorias estabelecidas. Utilizando o *chunk* apresentado na Listagem 4.2 para auxiliar a visualização das decisões do desenvolvedor, as decisões dos desenvolvedores podem ser:

- **Versão 1**: O conflito é resolvido mantendo integralmente a versão *V1*. Nesse caso, o resultado é composto pelas linhas do contexto iniciais (91–93), as linhas de *V1* (95–97) e as linhas de contexto finais (103–105), eliminando o restante do conflito.

- **Versão 2**: O conflito é resolvido mantendo integralmente a versão *V2*. Nesse caso, o resultado é composto pelas linhas do contexto iniciais (91–93), as linhas de de *V2* (99–101) e as linhas de contexto finais (103–105), eliminando o restante do conflito.

- **Concatenação**: As versões *V1* e *V2* são mantidas integralmente e concatenadas em qualquer ordem. Por exemplo, ambas as versões podem ser mantidas na sequência original, removendo somente os marcadores do conflito (linhas 94, 98 e 102) e mantendo as linhas do contexto iniciais (91–93) e as linhas de *V1* (95–97) as de *V2* (99–101) e as linhas de contexto finais (103–105).

- **Combinação**: A resolução combina partes de *V1* e *V2*. No exemplo, uma possível combinação seria preservar as linhasdo contexto iniciais (91–93) e mesclar trechos específicos, como as linhas 95, 96 e 101 seguidos das linhas de contexto finais (103–105).

- **Novo Código**: O conflito é resolvido por meio da introdução de um novo trecho de código, podendo ou não ser mantida parte do código original. Um exemplo seria substituir as linhas 95 e 96 por um código com uma quinta opção de input e manter as linhas de contexto (91–93 e 103–105) e a linha 97, com as linhas 95 e 96 substituídas por uma linha que não existe em nenhuma das versões.

- **Nenhum**: O conflito é resolvido removendo integralmente o trecho conflitante sem inserir nenhum código alternativo. Isso ocorreria ao simplesmente apagar as linhas 94 a 102. Mantendo somente as linhas de contexto (91–93 e 103–105).

- **Impreciso**: A resolução do conflito envolve modificações no código ao redor do *chunk* conflitante (91–93 e 103–105), impossibilitando identificar com precisão tanto a região exata na qual o conflito ocorreu quanto a estratégia adotada pelo desenvolvedor para resolvê-lo. Quando uma linha é modificada, o Git classifica como removida e adicionada, não permitindo manter o rastreio da região na qual ocorreu o conflito. Essas características impossibilitam a categorização precisa da decisão tomada, uma vez que o conflito deixa de ser rastreável no código final.

- **Postergado**: O conflito não é resolvido no momento do *merge* e o *commit* resultante mantém os delimitadores do conflito, adiando a resolução para um momento posterior. Um exemplo deste tipo de resolução é manter todas as linhas (92 a 105).

4.4 Implementação

A abordagem proposta foi implementada em uma ferramenta que possibilita a coleta dos dados de forma sistemática e automatizada. Essa ferramenta desempenha um papel essencial na identificação dos *merges*, na extração dos arquivos envolvidos, na detecção dos *chunks* conflitantes e das informações extraídas deles, como classificação das decisões dos desenvolvedores, quantidade de linhas de código (LOC) e os construtos da linguagem.

A primeira etapa desse processo utiliza um *SCV* para navegar pelo histórico dos repositórios e recriar os *merges*. O *SCV* permitiu não somente a recuperação eficiente dos dados, mas também a execução dos *merges* exatamente como ocorreram no repositório original, garantindo a fidelidade das informações coletadas. Para essa tarefa, foi utilizado o Git (CHACON; STRAUB, 2014), um SCV amplamente adotado na comunidade de desenvolvimento de software.

Uma vez identificados os *merges* conflitantes, tornou-se necessária uma análise detalhada dos arquivos e *chunk* em conflito. Para isso, foi utilizado um analisador sintático capaz de compreender a estrutura da linguagem Python. Para essa tarefa foi utilizado o *ANTLR* (*ANother Tool for Language Recognition*), uma ferramenta que permite a criação de analisadores capazes de interpretar gramáticas.

O repositório de gramáticas do ANTLR[4] reúne gramáticas para diversas linguagens de programação. Para este trabalho, foi utilizada a gramática correspondente à linguagem Python disponível nesse repositório[5].

A ferramenta implementada segue os passos apresentados no Capítulo 4 e permite a automação desse processo, permitindo a análise em larga escala dos conflitos, garantindo a coleta e análise eficiente de dados. Ela recebe como entrada uma lista de repositórios Git e realiza a identificação dos *merges* conflitantes, a análise dos arquivos envolvidos e a extração das informações relevantes dos *chunks*. Os dados extraídos podem ser armazenados em um banco de dados relacional SQL ou em arquivos no formato `JSON`, compatíveis com a interface gráfica da ferramenta, que permite a visualização e inspeção dos resultados. Essa abordagem garante flexibilidade na execução e na análise dos conflitos, possibilitando estudos em larga escala de maneira sistemática e organizada.

[4]https://github.com/antlr/grammars-v4
[5]https://github.com/antlr/grammars-v4/tree/master/python/python3

5 Estudo Prático

Este capítulo apresenta a metodologia experimental adotada para a análise dos conflitos de *merge* em repositórios de software escritos na linguagem Python. O objetivo do estudo prático é investigar padrões de ocorrência e resolução desses conflitos, fornecendo evidências empíricas que possam contribuir para a melhoria dos processos de *merge*.

O capítulo está estruturado da seguinte forma: Na Seção 5.1, são definidas as questões de pesquisa que guiam a análise, estabelecendo os aspectos específicos dos conflitos de *merge* que se deseja compreender. Em seguida, na Seção 5.2 são detalhados os critérios utilizados para a seleção dos repositórios analisados, visando diversidade e representatividade na amostra escolhida. Posteriormente, na Seção 5.3, descreve-se o processo de coleta de dados, abordando as etapas de extração e organização das informações relevantes para a pesquisa.

Com base nos dados coletados, na Seção 5.4 são apresentados os resultados obtidos para cada uma das questões de pesquisa propostas na Seção 5.1, acompanhados de análises quantitativas que exploram possíveis padrões nos conflitos identificados. Por fim, na Seção 5.5, discute-se a validade dos achados e as limitações do estudo, destacando fatores que podem influenciar a interpretação dos resultados e sugerindo direções para pesquisas futuras.

5.1 Questões da Pesquisa

Esta pesquisa busca aprofundar a compreensão dos conflitos de *merge* em projetos de software escritos em Python. O objetivo é identificar padrões e fatores que podem influenciar a ocorrência e a resolução desses conflitos, fornecendo bases para o desenvolvimento de ferramentas de *merge* mais eficazes. A partir disso, foram elaboradas as quatro questões de pesquisa descritas a seguir.

Questão 1: Qual é a distribuição no número de *chunks* conflitantes por *merges* em conflito?

Rationale: O número de *chunks* conflitantes pode estar relacionado à complexidade de resolver um *merge* em conflito. Intuitivamente, espera-se que quanto maior o número de *chunks*, maior o esforço exigido dos desenvolvedores para compreender e resolver os conflitos. Essa análise busca explorar possíveis padrões sobre a distribuição no número de *chunks* conflitantes por *merge* em conflito.

Métricas: Número absoluto de *chunks* por *merge* em conflito.

Questão 2: Qual é a distribuição de tamanho dos *chunks* conflitantes medidos em linhas de código (LOC)?

Rationale: Especula-se que o tamanho dos *chunks* conflitantes, medido em LOC, possa impactar a dificuldade de resolução. *Chunks* maiores podem exigir maior esforço de análise, enquanto *chunks* menores poderiam ser mais simples de resolver. Esta questão investiga a distribuição desses tamanhos.

Métricas: Contagem de LOC por V1, V2 e *chunk*.

Questão 3: Qual é a distribuição em termos de construtos da linguagem envolvidas nos *chunks* conflitantes?

Rationale: Os construtos da linguagem presentes nos *chunks* conflitantes podem influenciar a natureza e a complexidade do conflito. Espera-se, por exemplo, que conflitos envolvendo estruturas simples, como *imports*, sejam mais fáceis de resolver do que aqueles que envolvem estruturas mais complexas, como métodos ou classes. Esta análise procura identificar quais construtos da linguagem ocorrem com maior frequência.

Métricas: Frequência de construtos da linguagem nos *chunks* (ex.: *imports*, métodos, funções, classes, variáveis globais).

Questão 4: Qual é a distribuição das decisões dos desenvolvedores?

Rationale: As decisões dos desenvolvedores ao resolver conflitos de *merge* podem fornecer indícios importantes sobre os desafios enfrentados durante o processo de resolução. Espera-se que certas decisões como a escolha de uma das versões sejam mais frequentes do que outras, como a combinação ou adição de novo código, devido aos achados da literatura (GHIOTTO et al., 2020). Esta questão visa investigar a distribuição dessas decisões.

Métricas: As frequências de cada uma das decisões dos desenvolvedores como V1, V2 e*new code*

5.2 Seleção de Projetos

Para a coleta de dados deste trabalho, foram selecionados repositórios de código aberto (*open-source*) hospedados na plataforma *GitHub*, com o objetivo de garantir diversidade, representatividade e relevância na análise dos conflitos de *merge*. A seleção dos projetos seguiu critérios específicos, estabelecidos para buscar qualidade e consistência nos dados analisados:

- Apresentar um histórico de desenvolvimento com mais de 8000 *commits*, assegurando uma quantidade substancial de dados e maior probabilidade de ocorrência de *merges* conflitantes;
- Ter o Python definido como linguagem principal, a linguagem com maior porcentagem de código armazenada no repositório, uma vez que este trabalho se concentra exclusivamente na análise de conflitos em projetos escritos nessa linguagem;
- Disponibilizar documentação na seção *wiki* em português ou inglês, permitindo compreender a finalidade e a estrutura do projeto a partir de fontes oficiais, o que ajuda a evitar a inclusão de repositórios cujo conteúdo principal não esteja relacionado ao código-fonte.

Durante o processo de triagem, foram excluídos repositórios que, embora atendessem aos critérios quantitativos, não correspondiam à finalidade prática de um repositório

de código-fonte — como projetos voltados à publicação de livros ou conteúdos puramente textuais. Essa filtragem foi realizada por meio da análise manual das documentações complementares, descrições e estrutura dos diretórios.

Ao final da etapa de seleção, um total de 200 repositórios foi definido como amostra inicial para análise. Destes, 50 repositórios foram processados até a data limite estabelecida para a conclusão deste trabalho, compreendendo a coleta e análise dos dados referentes aos conflitos de *merge*. A seleção ocorreu conforme a ordem em que os repositórios foram retornados pelo sistema de arquivos durante a varredura em lote da pasta no Linux. Os 200 projetos analisados foram selecionados com base na ordem de retorno fornecida pela API pública do GitHub, respeitando os critérios mencionados anteriormente.

Todos os projetos selecionados atendiam aos critérios estabelecidos. Entre eles, encontram-se *frameworks* amplamente utilizados na comunidade Python, como *Django* e *Flask*, além de ferramentas de uso prático, como *Cython*, um compilador de Python para C, e *Medusa*, uma plataforma de *E-commerce*.

5.3 Coleta de dados

Os 50 projetos analisados possuem um total de 600.377,00 *commits*, 166.425,00 *merges* dos quais 12.530,00 possuem conflitos e geraram 69.052,00 *chunks*, permitindo a análise sobre os padrões de *merge* e resolução de conflitos adotados pelos desenvolvedores. Esses dados cobrem 50 projetos distintos, cujas características estatísticas estão resumidas na Tabela 5.1.

	Merges	**Merges com Conflito**	**Porcentagem de Merges com Conflito**	**Arquivos em Conflito**	**Chunks**
Media	3.001,04	252,34	7,33	578,06	1.381,04
Desvio Padrão	4.534,23	531,90	5,61	984,70	2.645,98
Min	15,00	0,00	0,00	0,00	0,00
25%	923,50	40,00	2,98	80,50	140,25
50%	2.021,50	112,50	6,37	295,00	402,50
75%	3.972,50	232,50	10,53	557,50	1.022,00
Max	31.847,00	3.596,00	23,90	5.472,00	13.848,00
Total	166.425,00	12.530,00	7,53	13.028,00	69.052,00

Tabela 5.1: Tabela Resultados dos Projetos

Como observado na Tabela 5.1, dos 166.425,00 *merges*, 12.530,00 apresentaram conflitos, representando uma taxa geral de 7,53%. A frequência de conflitos, no entanto, variou significativamente entre os projetos analisados. Por exemplo, o projeto *mahdibland/V2RayAggregator* apresentou a maior taxa de conflitos, com 23,90%. Em contraste, os projetos *security_content* e *openstack/openstack* registraram nenhum e somente um conflito, respectivamente, mesmo com dezenas de operações de *merge*.

A média da taxa de conflitos entre os projetos foi de 7,33%, com desvio padrão de 5,60 pontos percentuais. Essa alta variabilidade, representada pelo desvio padrão e também evidenciada pela diferença entre os valores mínimo (0,00%) e máximo (23,90%), indica que os projetos têm comportamentos bastante distintos em relação à ocorrência de conflitos. Além disso, mais da metade dos projetos (mediana de 6,37%) têm taxas inferiores à média, sugerindo a presença de projetos com valores extremos que puxam a média para cima — como o *mahdibland/V2RayAggregator*.

Outra forma de entender a distribuição dos conflitos é observar os quartis: 25% dos projetos apresentaram taxa inferior a 2,98%, enquanto 25% tiveram taxas superiores a 10,53%. Isso demonstra que uma parcela significativa dos repositórios enfrenta conflitos com relativa frequência, enquanto a maioria concentra-se em níveis mais baixos.

Além da frequência, a intensidade dos conflitos foi analisada por meio da quantidade de arquivos afetados e de *chunks* em cada conflito. A média por projeto foi de aproximadamente 578 arquivos e 1.381 *chunks* com conflitos, mas novamente com alta dispersão: os valores máximos atingem 5.472 arquivos e 13.848 *chunks*. Projetos como *Giskard-AI/giskard*, *inventree/InvenTree* e *sagemath/sage* se destacam nesse aspecto, sugerindo conflitos mais extensos e potencialmente mais complexos.

Por fim, observa-se que o número de *merges* por projeto também é bastante desigual. Enquanto a mediana é de 2.021 *merges*, o projeto *sagemath/sage* sozinho acumula 31.847 operações, mais de 10 vezes a média (3.001). Apesar disso, sua taxa de conflitos (11,29%) está próxima do terceiro quartil, indicando poucos conflitos no processo de *merge* mesmo em cenários de alta atividade. Já projetos com baixa atividade de *merge*, como *mahdibland/V2RayAggregator* e *oils-for-unix/oils*, apresentaram taxas de conflitos desproporcionalmente elevadas, o que pode sugerir políticas de *merge* menos maduras ou

elevada sobreposição de mudanças em trechos sensíveis do código.

5.4 Resultados

5.4.1 Qual a distribuição no número de *chunks* conflitantes por *merges* em conflito?

O gráfico de barras da Figura 5.1 apresenta a distribuição absoluta do número de *chunks* conflitantes por *merge*. Observa-se que a maior parte dos conflitos ocorre em *merges* com poucos *chunks*: cerca de 9.600,00 casos têm até três *chunks* conflitantes, sendo que o pico de frequência aparece nos conflitos com apenas um *chunk*. A partir desse ponto, a frequência decresce rapidamente, mantendo um comportamento semelhante até as barras correspondentes a mais de 16 *chunks*, que ainda somam 462,00 ocorrências.

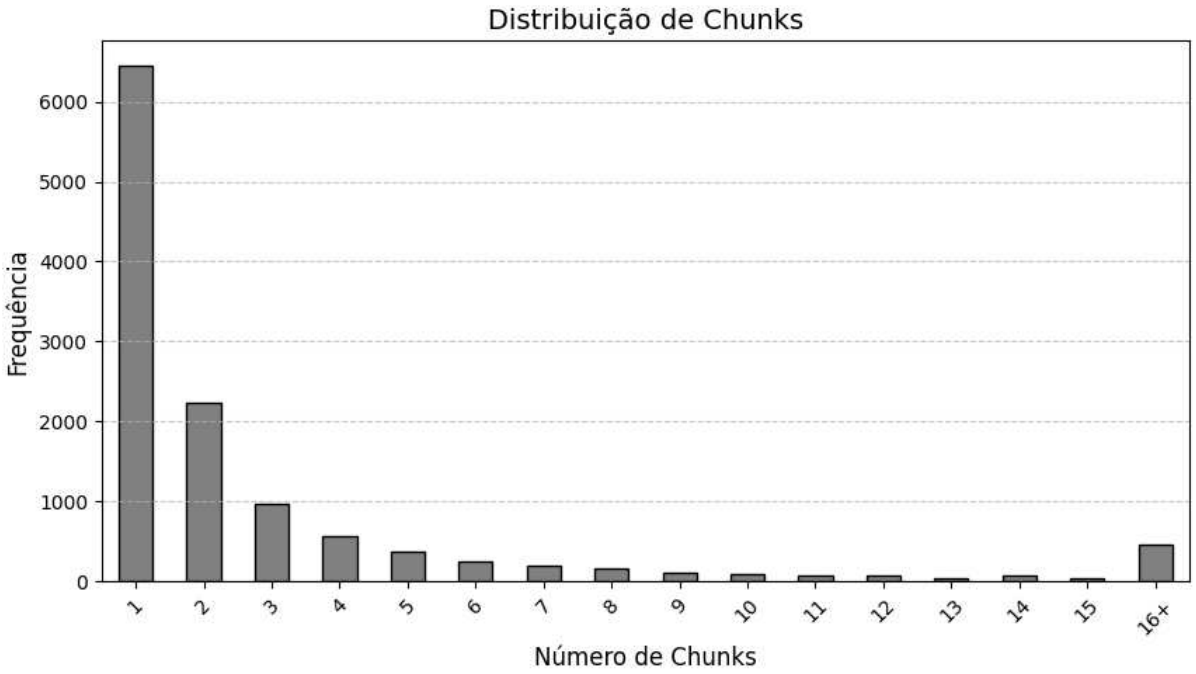


Figura 5.1: Distribuição de *chunks* por *merges* conflitantes.

O *box plot* da Figura 5.2 corrobora essa distribuição ao demonstrar que a mediana do número de *chunks* por *merge* em conflito é de um *chunk*, enquanto o terceiro quartil alcança o valor de três *chunks*. Dessa forma, 75,00% dos *merges* em conflito apresentam até 3 *chunks*, sendo que 51,08% concentram-se em casos com apenas um *chunk*. Embora a maior parte dos conflitos envolva poucos *chunks*, a dispersão dos dados é ampla: valo-

res superiores a seis representam 10,50% dos casos, indicando a presença de ocorrências isoladas com um número substantivamente maior de conflitos — o episódio mais extremo contém 3.738,00 *chunks* no repositório *Picard*[6].

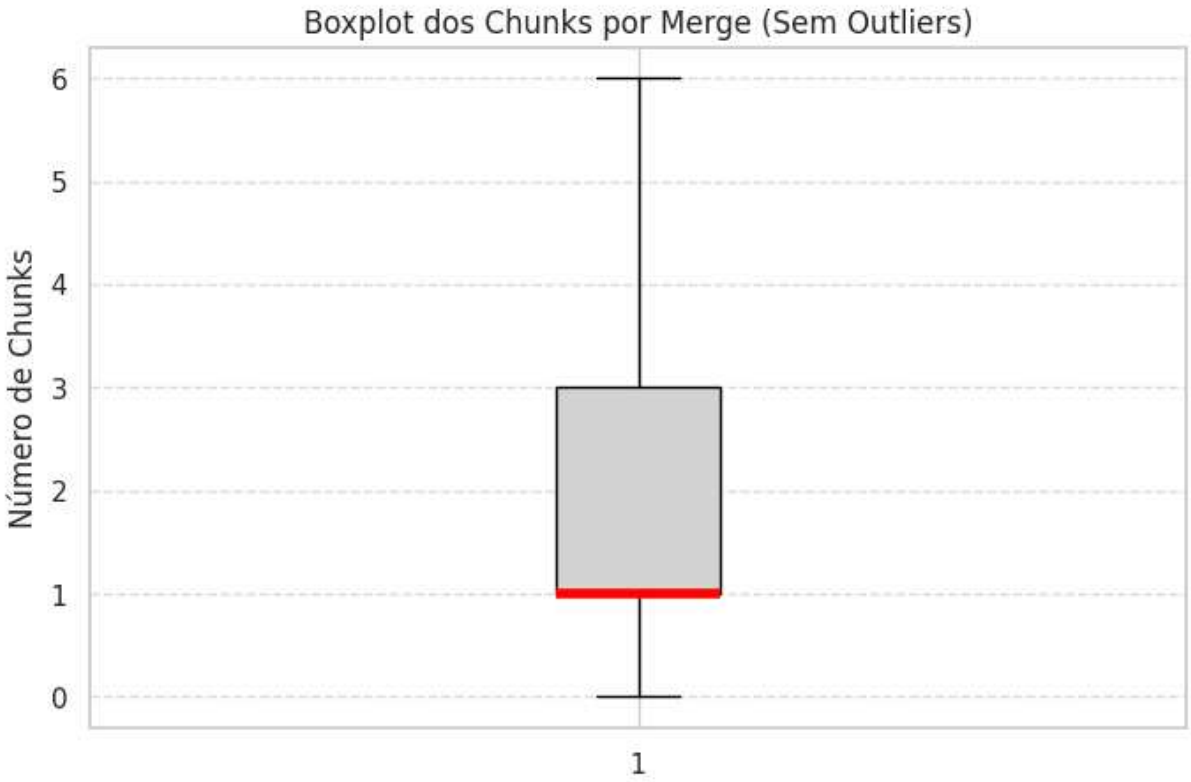


Figura 5.2: Box Plot mostrando a distribuição dos *chunks*.

Os dados descritivos evidenciam essa distribuição tendendo a *merges* com menos *chunks*. O valor médio do número de *chunks* conflitantes por *merge* é de 5,47 *chunks*, com um desvio padrão de 58,55, refletindo a grande variabilidade entre os casos.

Esses resultados indicam que a maioria dos *merges* com conflito envolve poucos blocos de código afetados e que conflitos com mais de 6 *chunks* são relativamente raros, ocorrendo em 10,50% dos casos. No estudo de referência conduzido por Ghiotto et al. (2020), focado em projetos Java, foi observado que 76% dos *merges* em conflito possuem quatro ou menos *chunks* enquanto nesse estudo 75% dos *merges* possuem menos de 3. Essa semelhança entre os dois estudos sugere que a dimensão dos conflitos — em termos de quantidade de *chunks* — tende a seguir um padrão comum, independentemente das duas linguagens de programação analisadas. Dessa forma, há indícios de que o número de *chunks* conflitantes por *merge* pode ser uma característica, nessa análise , agnóstica à

[6]https://github.com/broadinstitute/picard

linguagem utilizada no projeto.

5.4.2 Qual é a distribuição de tamanho dos *chunks* conflitantes medidos em linhas de código (LOC)?

A análise do tamanho dos *chunks* conflitantes em linhas de código foi conduzida sob duas perspectivas complementares: o tamanho total do *chunk*, obtido pela soma das linhas de V1 e V2, e a análise individual das linhas presentes em cada versão. Essa abordagem permite observar tanto a extensão total do código envolvido no conflito quanto as diferenças entre as modificações conflitantes realizadas em cada *branch*.

A Figura 5.3 apresenta a distribuição de número total de linhas, no qual o eixo X é o número de linhas de V1 e V2 somados e o eixo Y é a frequência absoluta de ocorrências, sendo possível observar que os conflitos tendem a envolver pequenas porções de código. Por exemplo, conflitos com duas linhas possuem 20.840,00 ocorrências o que representa 30,18% dos dados. A maioria, 51.801,00 (o que representa 75.02% do total), dos *chunks* encontra-se concentrada entre 1 e 13 linhas de código.

A dispersão dos dados é evidenciada no *boxplot* da Figura 5.4, que exclui os *outliers* para melhor visualização. Os dados demonstram que 75% dos *chunks* conflitantes possuem até 13 linhas e a mediana está em 5 linhas. Por outro lado, o tamanho médio dos *chunks* é 33,70 linhas, mas esse valor é amplamente influenciado por casos extremos. Por exemplo, o maior *chunk* registrado no conjunto é de 37.314,00 linhas, caracterizando-se como um *outlier*.

Além disso, foi realizada uma análise comparativa entre o número de linhas de código das versões V1 e V2 nos *chunks* conflitantes, como mostrado na Figura 5.5, um gráfico de dispersão comparando o número de linhas de V1 e de V2 por *chunk*. A maioria dos pontos está próxima da origem, indicando que a maioria dos conflitos, 75,00% envolve até 6 linhas tanto no tamanho de V1 quanto no de V2. Observa-se ainda uma tendência diagonal, sugerindo uma correspondência entre as dimensões das versões.

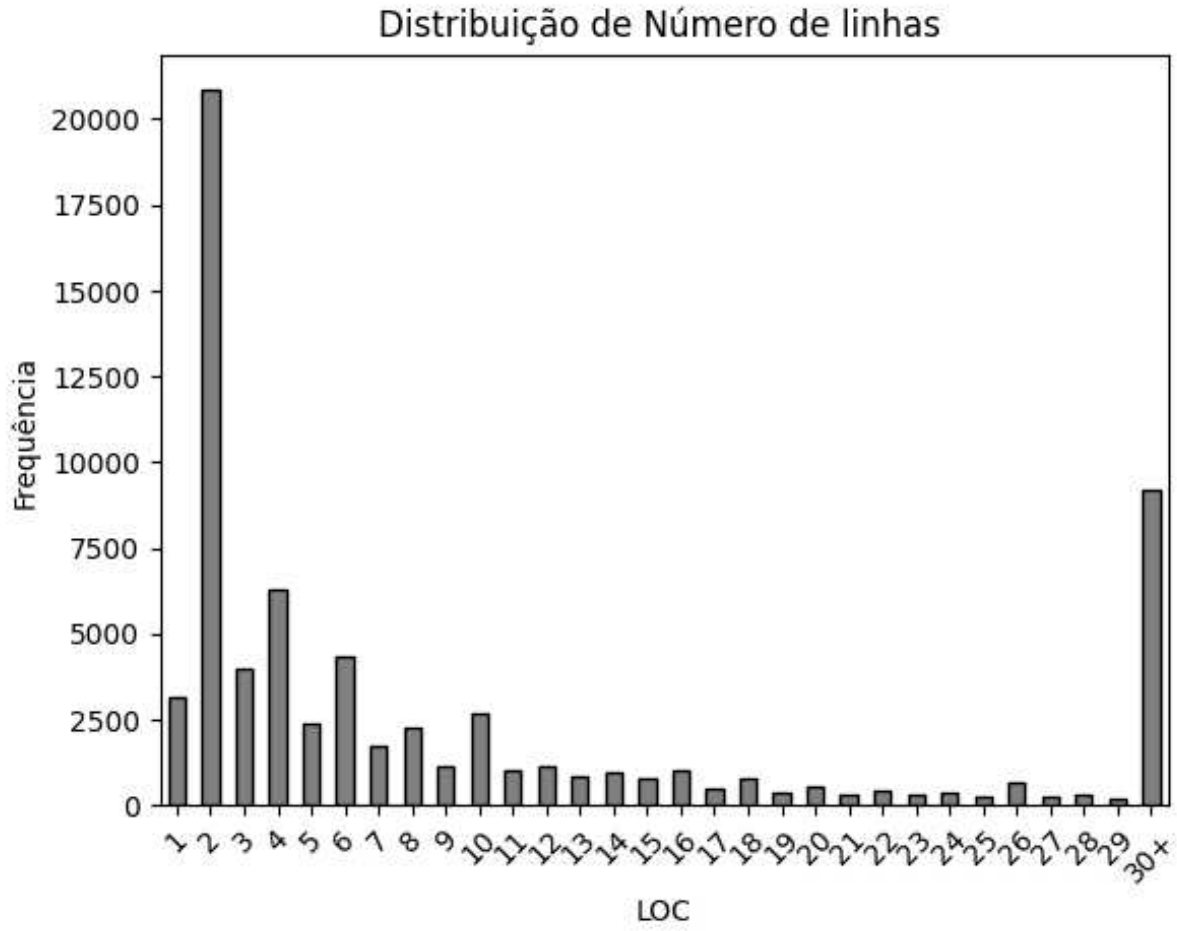
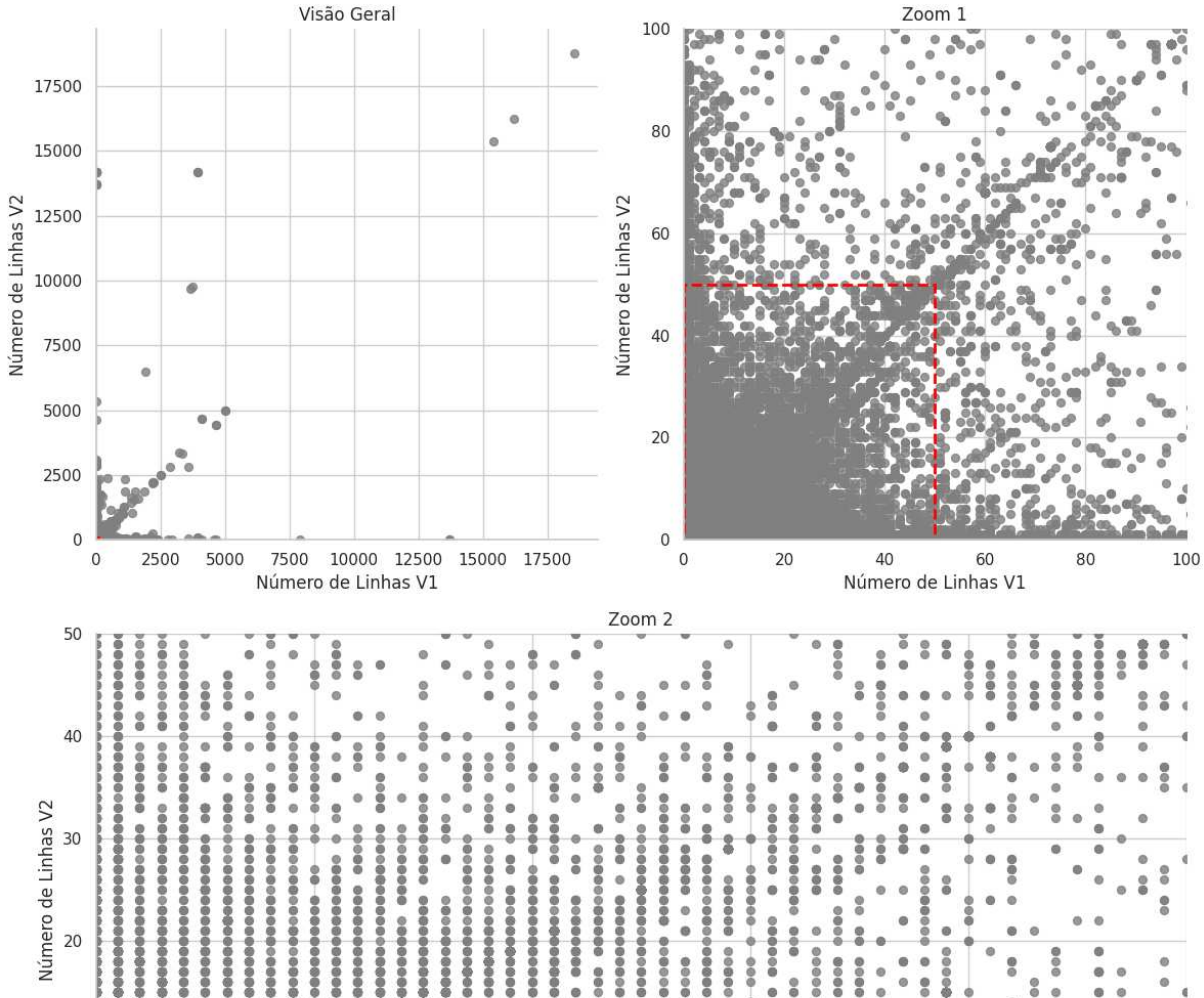


Figura 5.3: Distribuição do tamanho dos *chunks* conflitantes em linhas de código.



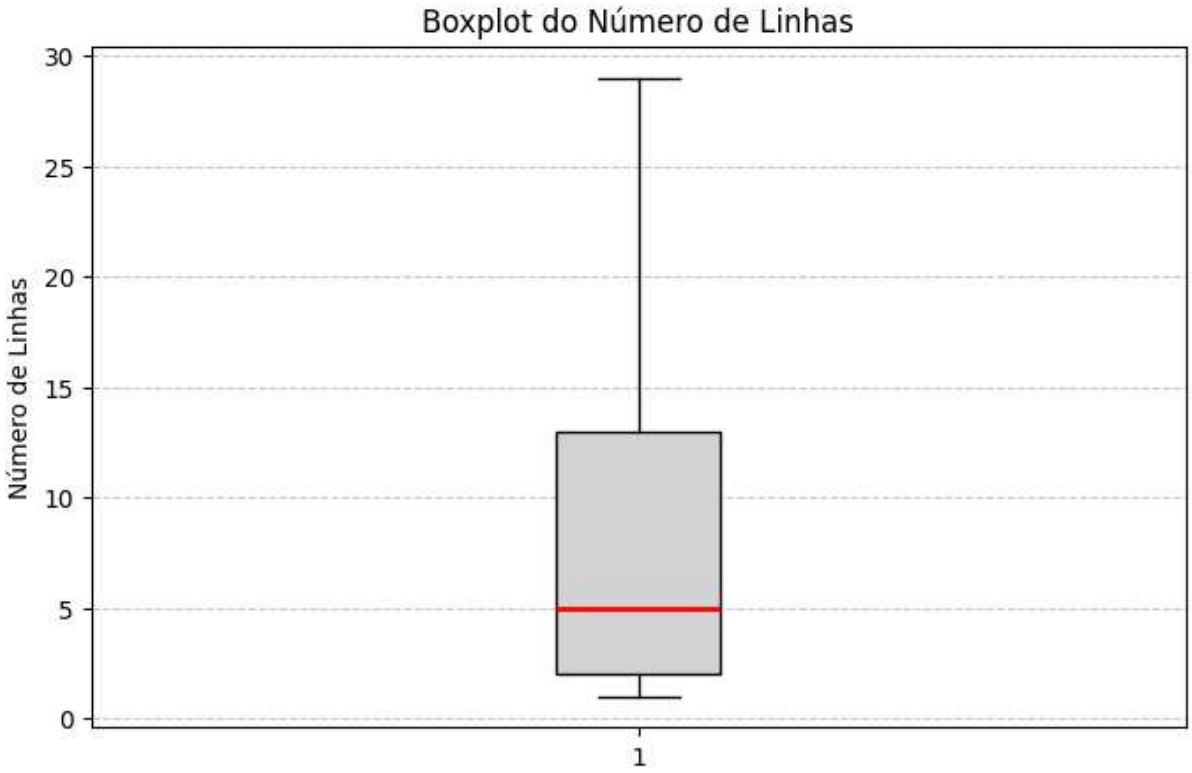


Figura 5.4: *Box plot* do tamanho dos *chunks* conflitantes, sem *outliers*.

Apesar disso, há casos de maior discrepância, como conflitos envolvendo 25.000 linhas somente em V2 (com V1 vazia), ou 35.000 linhas em V1 (com V2 vazia). Essas ocorrências indicam operações como reescrita total de arquivos, inserções massivas ou exclusões extensas de código, podendo estar relacionadas a refatorações ou reestruturações mais profundas.

Em comparação com o estudo conduzido por Ghiotto et al. (2020), que analisou projetos Java, nota-se um padrão semelhante: 96% dos *chunks* em conflito naquele estudo possuíam menos de 50 linhas em cada versão (ou menos de 100 linhas no total). Neste estudo, com projetos em Python, observa-se que 75% dos *chunks* possuem menos de 13 linhas totais (somando V1 e V2), e apenas 12.96% ultrapassam 30 linhas. Além disso 95,78% dos casos menos de 50 linhas em cada versão, ambos os resultados sugerem que a maioria dos conflitos tende a ser localizada e de tamanho reduzido.

Essas observações indicam uma possível recorrência de padrões similares entre linguagens distintas. A concentração de conflitos em trechos pequenos pode ser uma característica geral do processo de *merge*, mais do que um traço específico da linguagem de programação. No entanto, a presença de conflitos mais extensos em ambas as análises

reforçam a necessidade de atenção a casos excepcionais, que podem exigir abordagens diferenciadas para resolução.

5.4.3 Qual é a distribuição em termos de construtos da linguagem envolvidas nos *chunks* conflitantes?

A identificação dos construtos da linguagem presentes nos *chunks* conflitantes permite compreender melhor os elementos estruturais que mais frequentemente se associam a conflitos em *merges*. Essa análise pode revelar tanto padrões sintáticos complexos que dificultam a resolução automática, quanto estruturas comuns que, pela sua alta frequência no código-fonte em geral, tendem a aparecer com maior regularidade nos conflitos.

A Figura 5.6 apresenta a frequência absoluta dos diferentes construtos da linguagem identificados nos conflitos analisados. Importa destacar que cada construto é contabilizado no máximo uma vez por *chunk* — ou seja, mesmo que apareça múltiplas vezes, ele é registrado somente uma vez. Essa abordagem visa medir a presença estrutural do construto no conflito, e não sua repetição local, alinhando-se ao objetivo de capturar a diversidade de estruturas envolvidas, não sua densidade.

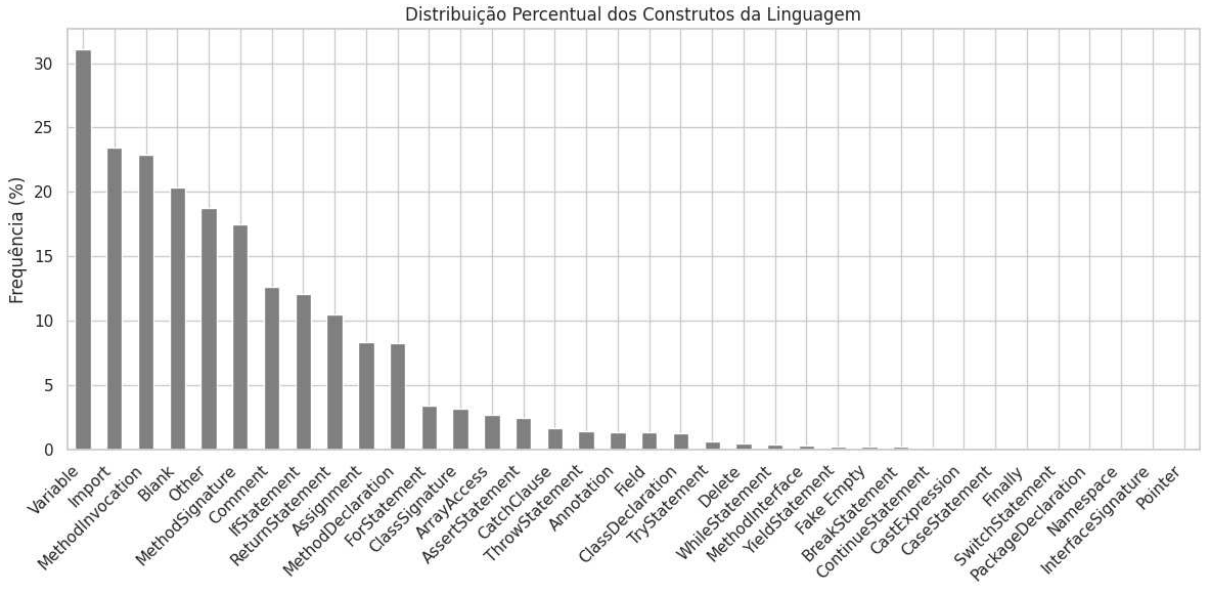


Figura 5.6: Distribuição dos construtos da linguagem nos conflitos.

A análise evidencia que certos construtos, como invocação de métodos (22.8%), declarações de variáveis (31.1%) e importações (23.4%), são mais frequentes nos conflitos. Há duas possíveis interpretações para esse padrão. Primeiramente, pode indicar que

esses construtos, por sua própria natureza sintática ou semântica, são mais propensos a gerar conflitos, especialmente quando sujeitos a edições concorrentes. Alternativamente, sua alta ocorrência pode refletir simplesmente sua prevalência no código em geral — ou seja, estruturas mais usadas tendem naturalmente a aparecer mais em conflitos, sem necessariamente serem mais "conflituosas"por si só.

Essa ambiguidade metodológica também foi abordada no estudo em Java (GHIOTTO et al., 2020), que propôs essa questão de pesquisa. Nele, os autores analisaram a frequência de construtos tanto nos *chunks* conflitantes quanto no código-fonte completo, identificando que certos elementos, como invocação de métodos, aparecem menos proporcionalmente nos conflitos do que em todo o código. Isso sugere que, embora comuns, não necessariamente são mais sujeitos a conflito. Neste estudo, não foi realizada uma análise comparativa com a base completa de código, limitando interpretações definitivas quanto à "conflituosidade"relativa de cada construto.

Ainda segundo Ghiotto et al. (2020), 52% dos *chunks* em conflito envolviam somente um construto da linguagem e 80% envolviam dois ou menos. Os resultados encontrados neste estudo em Python são comparáveis: como mostra a Figura 5.7, 49,2% dos *chunks* envolvem somente um construto e 76% envolvem até dois, revelando uma estrutura de conflitos também pouco diversificada. Essa similaridade percentual entre linguagens distintas sugere que conflitos geralmente se concentram em pequenas regiões do código, com baixa complexidade estrutural — reforçando a viabilidade de técnicas específicas para resolver conflitos envolvendo um número limitado de construções sintáticas.

Vale relembrar que somente os construtos externos (*outmost*, explicado na Seção 4.4) presentes nos *chunks* são registrados. Essa escolha segue a metodologia do estudo original e se justifica pelo fato de que os conflitos geralmente ocorrem em blocos contíguos, nos quais há uma estrutura predominante responsável pela divergência. Ao focar nesses construtos principais, busca-se captar os elementos centrais da lógica em conflito, evitando o viés introduzido por estruturas internas que podem não ser diretamente relevantes para o surgimento do conflito.

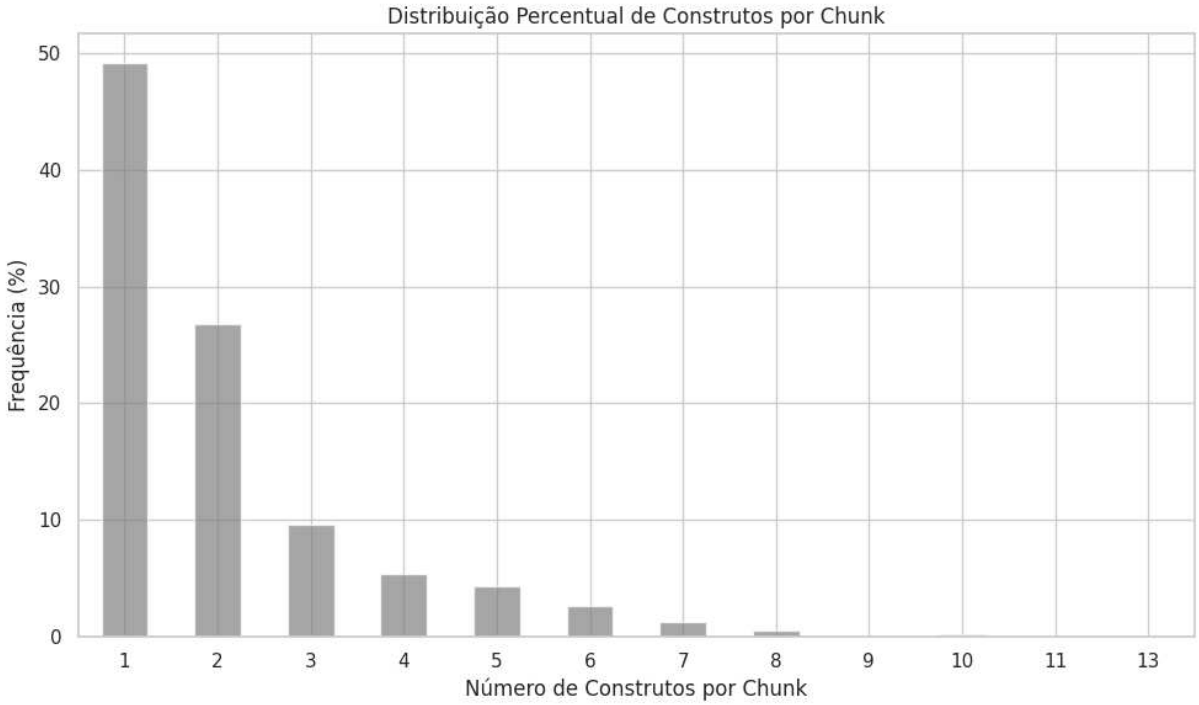


Figura 5.7: Distribuição percentual do número de construtos por *chunk* em conflito.

5.4.4 Qual é a distribuição das decisões dos desenvolvedores?

Para entender como os desenvolvedores resolvem conflitos em *merges*, esta análise examina a distribuição das decisões tomadas sobre os *chunks* conflitantes. A Figura 5.8 apresenta a quantidade total de cada tipo de decisão observada nos dados, com eixo X representando a decisão do desenvolvedor e o Y a porcentagem dos *chunks* com essa decisão.

Observa-se que as decisões majoritárias consistem na escolha integral por uma das versões em conflito, com "Versão 1"sendo adotada em 41,88% dos casos (aproximadamente 28 mil casos), seguida pela "Versão 2"com 28.60% (cerca de 20 mil). Essas duas categorias representam uma significativa parcela dos conflitos analisados (70,48%) indicando que, na maioria das situações, os desenvolvedores optam por manter o código de uma das versões envolvidas no conflito. Decisões como "Novo Código"(8,06%),"Imprecisão"(8,01), "Combinação"(7,57%), e "Concatenação"(4.17%) aparecem em menor escala, mas ainda com presença relevante, refletindo cenários no qual a resolução exigiu maior intervenção manual ou reestruturação do código.

A Figura 5.9 explora a distribuição percentual dessas decisões dos desenvolvedores por projeto, removendo os *outliers* para facilitar a visualização de tendências gerais. A decisão por "Versão 1"é a mais comum, com uma mediana de 36,6% dos conflitos re-

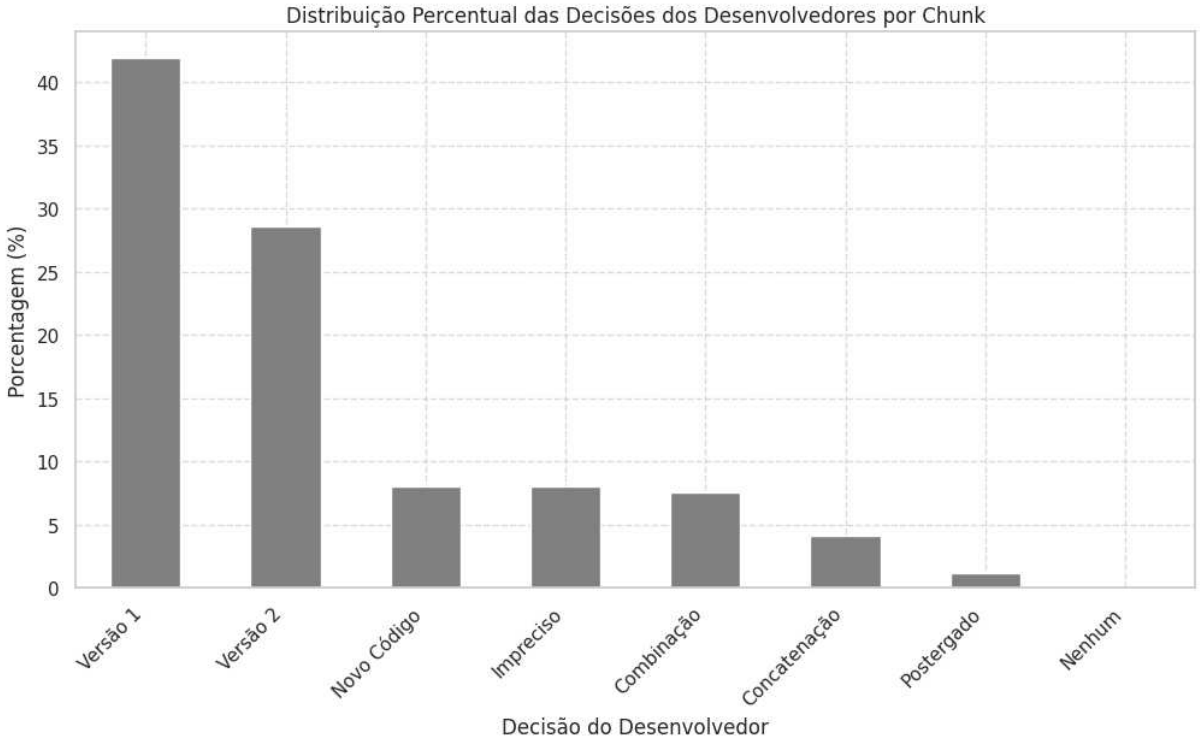


Figura 5.8: Distribuição das decisões dos desenvolvedores por *chunk*.

solvidos dessa forma em cada projeto, seguida por "Versão 2"com 21,6%. Embora casos extremos existam (como projetos com mais de 90% das decisões em "Versão 1"), a maioria dos projetos apresenta uma distribuição mais equilibrada, com uso também de decisões alternativas.

Decisões que indicam maior elaboração por parte dos desenvolvedores — como "Combinação"e "Novo Código"— possuem medianas de aproximadamente 9,5% e 7,1%, respectivamente. Isso mostra que, mesmo não sendo dominantes, são estratégias relevantes no cenário real de resolução de conflitos. Já a "Concatenação"(4,52%), aparece com frequência de aproximadamente metade da combinação.

Outras categorias como "Postergado"(0,44%), e "Nenhum"(0,03%) são pouco frequentes, indicando que, em geral, os desenvolvedores não adiam a resolução ou deixam conflitos não resolvidos nos *commits* analisados. A baixa ocorrência da categoria "Nenhum"também sugere uma predominância de *merges* com contribuições efetivamente aplicadas.

Por fim, os dados por projeto evidenciam variações consideráveis entre equipes e contextos. A dispersão das decisões mostra que não há uma abordagem única para todos os casos, sendo comum que diferentes projetos adotem estratégias variadas dependendo

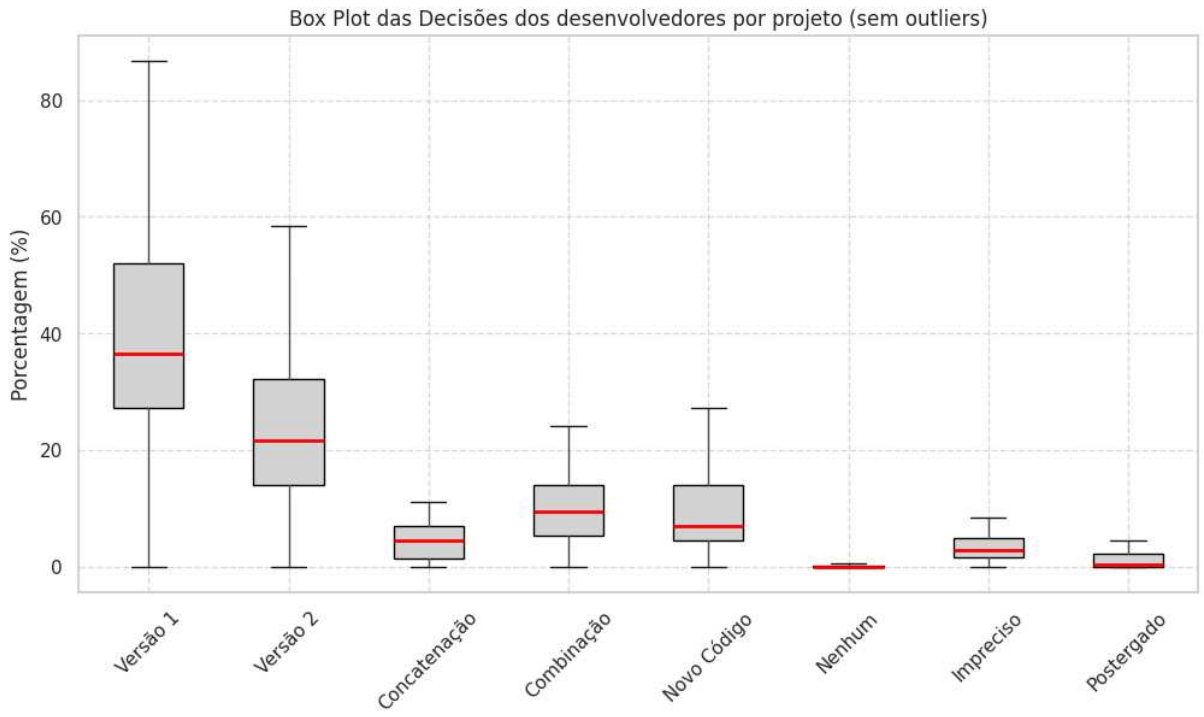


Figura 5.9: *Box plot* das decisões dos desenvolvedores por projeto, sem *outliers*.

da complexidade do código e do estilo de desenvolvimento.

5.5 Discussão

Este estudo teve como objetivo analisar as características estruturais de conflitos em *merges* de projetos Python, com ênfase na comparação com um estudo anterior conduzido com projetos em Java (GHIOTTO et al., 2020). A partir dos resultados obtidos, é possível identificar padrões similares entre as duas linguagens como o tamanho médio de *chunks* por *merge* em conflito, bem como observar algumas particularidades das linguagens como a frequência de *chunks* sem construtos ("*blank*") na linguagem Python.

Em relação à quantidade de *chunks* conflitantes por *merge*, observou-se que a maioria dos casos está concentrada em conflitos de pequena escala: 75,00% dos *merges* com conflito apresentaram até três *chunks*. Apenas 3,66% ultrapassaram 16 *chunks* por *merge*, evidenciando que conflitos com grande quantidade de *chunks* são uma minoria. De forma comparável, o estudo com Java relatou que 76,00% dos *merges* conflitantes continham até quatro *chunks*. Isso sugere que, independentemente da linguagem, a maioria dos conflitos tende a envolver regiões localizadas do código, o que pode ter implicações diretas na

complexidade da resolução manual ou automatizada.

Quanto ao tamanho dos *chunks* conflitantes em linhas de código, verificou-se uma forte concentração em intervalos curtos: 75,00% dos *chunks* apresentaram no máximo 13,00 linhas somando V1 e V2, e a mediana foi de 5,00 linhas. Esses resultados indicam que os conflitos, em geral, envolvem modificações com poucas linhas, embora existam exceções extremas, como *chunks* com mais de 35.000,00 linhas. A separação da análise por versões V1 e V2 revelou assimetrias pontuais entre os lados do conflito e foi importante para identificar padrões estruturais específicos em cada versão, como inserções ou remoções em massa.

A análise de construtos da linguagem envolvida nos conflitos também revelou similaridades importantes com o estudo em Java. A Figura 5.7 mostra a distribuição percentual do número de construtos por *chunk*. Notou-se que 49,2% dos *chunks* envolvem somente um construto, e 76% envolvem até dois, valor idêntico ao observado no estudo com Java. Isso reforça a hipótese de que a maioria dos conflitos se dá em contextos sintáticos simples, ainda que envolvam estruturas semanticamente relevantes. Tal padrão pode indicar que abordagens de resolução automatizada mais especializadas, focadas em pequenos conjuntos de construtos, podem vir a ser eficazes em ambas as linguagens.

Por fim, ao considerar quais construtos são mais frequentemente envolvidos nos conflitos, identificou-se uma predominância de elementos comuns, como chamadas de métodos, declarações de variáveis e estruturas condicionais. Essa distribuição pode refletir tanto a frequência desses elementos no código geral quanto uma propensão maior a causar conflitos em certos contextos de modificação.

Além das características estruturais, a análise também abordou as decisões adotadas pelos desenvolvedores na resolução dos conflitos. Os dados indicam que a maioria dos *chunks* conflitantes é resolvida com a escolha direta de uma das versões em conflito. A decisão “Versão 1”foi adotada em 41,88% dos casos, enquanto “Versão 2”apareceu em 28,60%, totalizando 70,48% das decisões. Essa predominância de escolhas unilaterais foi também observada no estudo com Java, apontando para uma tendência generalizada de resolução pragmática e direta, possivelmente por motivos de simplicidade, familiaridade com o código, ou confiança em determinada linha de desenvolvimento.

Decisões mais elaboradas, como "Novo Código"(8,06%), "Combinação"(7,57%) e "Concatenação"(4,17%), embora menos frequentes, representam uma parcela dos conflitos. Essas decisões indicam situações em que nenhuma das versões conflitantes pôde ser adotada isoladamente, exigindo intervenção manual e reestruturação. A existência dessas categorias reflete a complexidade semântica de certos conflitos, que não pode ser capturada apenas pela análise estrutural.

Em conjunto, os resultados deste estudo sugerem que algumas características estruturais dos conflitos — como a quantidade de *chunks* por *merge*, o tamanho dos *chunks* em linhas e a simplicidade sintática predominante — não parecem ser exclusivas da linguagem de programação analisada. Do mesmo modo, as estratégias predominantes de resolução de conflitos também seguem padrões similares entre as linguagens. Assim, há indícios de que esses comportamentos sejam mais gerais e independentes da linguagem, o que pode abrir espaço para o desenvolvimento de ferramentas e técnicas de *merge* mais universais e agnósticas quanto à linguagem de origem dos projetos.

5.6 Ameaças à validade

Este estudo considerou 50 projetos Python, totalizando 166.425 *merges* totais e 12.530 *merges* com conflitos. Embora esse número represente uma amostra relevante, ainda assim não cobre a totalidade dos projetos de software. Assim, os resultados devem ser interpretados como indicativos de tendências gerais, mas não necessariamente generalizáveis para todos os contextos, especialmente em projetos com características muito específicas ou de domínios altamente especializados.

Uma possível ameaça à validade externa está relacionada ao critério linguístico adotado na seleção dos projetos. Foram incluídos somente repositórios com documentação principal em português ou inglês. Embora essa decisão vise garantir a compreensão adequada do conteúdo e a consistência da análise manual, ela pode ter levado à exclusão de projetos relevantes mantidos em outras linguagens.

Adicionalmente, a classificação dos repositórios na busca do GitHub depende de um algoritmo próprio da plataforma, que pode ser influenciado por fatores dinâmicos como popularidade, atualizações recentes ou engajamento da comunidade. Isso pode

afetar a replicabilidade exata da seleção em momentos distintos. Para mitigar esse risco, a busca foi realizada utilizando filtros claros de linguagem e com critérios de tamanho para garantir a relevância dos projetos analisados.

Outra limitação metodológica está na exclusão de arquivos que não possuem a extensão `.py`. Arquivos como `.ipynb` (Jupyter notebooks), `Pipfile`, `requirements.txt` e outros arquivos de configuração, embora comuns em projetos Python, não foram incluídos na análise. A decisão por restringir a análise ao código Python puro se deu para garantir a uniformidade da extração e classificação dos *chunks* conflitantes, dado que esses outros formatos possuem estruturas específicas que exigiriam técnicas diferentes de análise. A ausência desses arquivos pode impactar o entendimento completo da natureza dos conflitos em alguns cenários, especialmente em projetos de ciência de dados ou infraestrutura. No entanto, essa delimitação foi necessária para preservar a consistência do método adotado, e estudos futuros podem estender o escopo para avaliar esses casos separadamente.

Apesar dessas limitações, este estudo buscou seguir boas práticas metodológicas para aumentar a transparência e a reprodutibilidade dos resultados. Primeiramente, foi realizada uma coleta em larga escala envolvendo diferentes tipos de projetos, de modo a captar padrões diversos de comportamento. Em seguida, adotaram-se critérios objetivos para a seleção dos repositórios (como linguagem principal e tamanho mínimo em *commits*), com o objetivo de reduzir escolhas arbitrárias. Na etapa de extração e análise, optou-se por considerar apenas arquivos com extensão ".py", visando manter a uniformidade técnica na identificação e classificação dos *chunks* conflitantes. Dessa forma, ainda que existam restrições, buscou-se conduzir o trabalho de maneira sistemática para contribuir com evidências sobre a natureza dos conflitos de *merge* em projetos Python.

6 Conclusão

Este trabalho investigou os conflitos de *merge* em projetos escritos na linguagem Python, analisando padrões na ocorrência e resolução desses conflitos. A partir de um conjunto de 50 repositórios extraídos do *GitHub*, foram exploradas métricas como o número de *chunks* conflitantes, o tamanho dos *chunks* em linhas de código, a distribuição dos construtos da linguagem envolvidos nos conflitos e as decisões tomadas pelos desenvolvedores para resolvê-los.

Os resultados obtidos indicam que, apesar das diferenças sintáticas e de paradigma entre linguagens de programação, existem padrões recorrentes nos conflitos de *merge*, reforçando a importância de um entendimento mais amplo desse fenômeno. A análise revelou tanto semelhanças, como o numero de linhas nas versões do conflito, quanto diferenças, como a ordem dos construtos mais frequentes, em relação a estudos prévios focados na linguagem de programação Java, sugerindo que as características específicas de cada linguagem podem influenciar a dinâmica dos *merges*, especialmente no que diz respeito aos construtos mais frequentemente envolvidos nos conflitos e às estratégias de resolução adotadas.

Além disso, a investigação contribui para a ampliação do conhecimento sobre conflitos de *merge*, fornecendo informações que podem ser utilizadas em pesquisas futuras voltadas para a melhoria de ferramentas automatizadas de resolução de conflitos. Ao aprofundar a análise sobre Python, este estudo complementa abordagens anteriores e destaca a semelhanças de múltiplas linguagens ao desenvolver soluções para *merge* de código mas também algumas diferenças que devem ser consideradas.

6.1 Contribuições

A principal contribuição deste trabalho é o aprofundamento do entendimento sobre conflitos de *merge* em Python e suas relações com estudos anteriores focados na linguagem de programação Java. Ao expandir o escopo de análise para uma linguagem com múltiplos

paradigmas, como Python, a pesquisa adiciona novas perspectivas sobre a influência da sintaxe e da estrutura da linguagem na ocorrência e resolução dos conflitos.

Os achados reforçam que a maioria das características observadas possui natureza agnóstica em relação à linguagem de programação, indicando que os padrões identificados tendem a se manter independentemente do contexto da linguagem analisada. A única característica que se destacou de forma mais relevante refere-se à frequência dos construtos da linguagem envolvidos nos conflitos. Esse conhecimento pode servir como base para pesquisas futuras em Engenharia de Software, contribuindo para a formulação de estratégias de resolução de conflitos de *merge* mais eficazes e que sejam, em grande parte, independentes da linguagem utilizada.

6.2 Trabalhos Futuros

Como trabalhos futuros, propõe-se a ampliação do estudo para uma quantidade maior de repositórios, bem como a inclusão de projetos desenvolvidos em diferentes linguagens de programação e paradigmas, com o intuito de permitir uma análise comparativa mais abrangente e robusta dos padrões de conflito. Outra direção promissora consiste em investigar como os achados deste estudo — como a frequência de determinados construtos e as estratégias de resolução mais adotadas — podem ser incorporados ao desenvolvimento de ferramentas automatizadas de apoio à resolução de conflitos. Especificamente, tais ferramentas poderiam ser aprimoradas para oferecer sugestões baseadas em soluções recorrentes para tipos específicos de construtos, respeitando as particularidades sintáticas e semânticas de cada linguagem analisada.

Outra possibilidade de pesquisa é a aplicação desses resultados na criação de modelos preditivos capazes de antecipar a ocorrência de conflitos com base nas características do código e do histórico de *merges* de um projeto. Compreender melhor os padrões de conflitos em diferentes contextos pode contribuir para o desenvolvimento de abordagens que reduzam a necessidade de intervenção manual no processo de *merge* de código, tornando-o mais eficiente e menos propenso a erros.

Por fim, a continuidade dessa linha de pesquisa pode fortalecer o entendimento sobre conflitos de *merge* de forma mais ampla, permitindo que futuras soluções sejam

desenhadas considerando não somente uma linguagem específica, mas um conjunto diversificado de linguagens e paradigmas de programação. Dessa forma, espera-se que este estudo sirva como um passo inicial para investigações mais profundas sobre a natureza dos conflitos de *merge* e suas implicações no desenvolvimento de software colaborativo.

Bibliografia

AHMED, I. et al. An empirical examination of the relationship between code smells and merge conflicts. In: IEEE. *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.], 2017. p. 58–67.

BERCZUK, S.; APPLETON, B. *Software configuration management patterns: effective teamwork, practical integration*. [S.l.]: Addison-Wesley Professional, 2020.

BRUN, Y. et al. Proactive detection of collaboration conflicts. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. [S.l.: s.n.], 2011. p. 168–178.

CAVALCANTI, G.; ACCIOLY, P.; BORBA, P. Assessing semistructured merge in version control systems: A replicated experiment. In: IEEE. *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.], 2015. p. 1–10.

CHACON, S.; STRAUB, B. *Pro git: Everything you need to know about Git*. Second. Apress, 2014. Disponível em: ⟨https://git-scm.com/book/en/v2⟩.

CIRIBELLI, L.; GHIOTTO, G.; CAMPOS, H. Um estudo sobre a natureza do merge de software em python. In: *Trabalho de Conclusão de Curso de Ciências Exatas*. [S.l.: s.n.], 2022.

CIRIBELLI, L. R. et al. Merge nature: a tool to support research about merge conflicts. In: *Workshops on Software Visualization, Evolution and Maintenance*. [s.n.], 2022. Disponível em: ⟨https://api.semanticscholar.org/CorpusID:254000618⟩.

CUNHA, M.; ACCIOLY, P.; BORBA, P. The private life of merge conflicts. In: *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2022. p. 353–362.

FITZPATRICK, B.; COLLINS-SUSSMAN, B.; PILATO, C. M. *Version Control with Subversion, Second Edition*. 1003 Gravenstein Highway North Sebastopol, CA 95472: [s.n.], 2008. 430 p. This book is released under a Creative Commons license. Disponível em: ⟨http://svnbook.red-bean.com/⟩.

GHIOTTO, G. et al. On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github. *IEEE Transactions on Software Engineering*, IEEE, v. 46, n. 8, p. 892–915, 2020.

KASI, B. K.; SARMA, A. Cassandra: proactive conflict minimization through optimized task scheduling. In: NOTKIN, D.; CHENG, B. H. C.; POHL, K. (Ed.). *Proceedings of the 35th International Conference on Software Engineering*. [S.l.]: IEEE / ACM, 2013. p. 732–741. ISBN 978-1-4673-3076-3.

KASWAN, K. S.; DHATTERWAL, J. S.; BALAMURUGAN, B. *Python for Beginners*. [S.l.]: Chapman and Hall/CRC, 2023.

LESSENICH, O.; APEL, S.; LENGAUER, C. Balancing Precision and Performance in Structured Merge. *Automated Software Engineering*, v. 22, n. 3, p. 367–397, set. 2015. Disponível em: ⟨http://intranet.infosun.fim.uni-passau.de/publications/docs/LAL15ase.pdf⟩.

LESSENICH, O. et al. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering*, Springer, v. 25, n. 2, p. 279–313, 2018.

MENS, T. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, IEEE, v. 28, n. 5, p. 449–462, 2002.

NGUYEN, H. L.; IGNAT, C.-L. Parallelism and conflicting changes in git version control systems. In: *IWCES'17-The Fifteenth International Workshop on Collaborative Editing Systems*. [S.l.: s.n.], 2017.

PARR, T. *The Definitive ANTLR 4 Reference*. 2. ed. Raleigh, NC: Pragmatic Bookshelf, 2013. ISBN 978-1-93435-699-9. Disponível em: ⟨https://www.safaribooksonline.com/library/view/the-definitive-antlr/9781941222621/⟩.

RIBEIRO, B. B.; COSTA, C.; SANTOS, R. Pereira dos. Understanding and analyzing factors that affect merge conflicts from the perspective of software developers. *Journal of Software Engineering Research and Development*, v. 10, p. 12:1 – 12:17, Oct. 2022. Disponível em: ⟨https://journals-sol.sbc.org.br/index.php/jserd/article/view/2576⟩.

SANTOS, R. de S.; MURTA, L. G. P. Evaluating the branch merging effort in version control systems. In: IEEE. *2012 26th Brazilian Symposium on Software Engineering*. [S.l.], 2012. p. 151–160.

SHEN, B. et al. A characterization study of merge conflicts in java projects. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, v. 32, n. 2, mar. 2023. ISSN 1049-331X. Disponível em: ⟨https://doi.org/10.1145/3546944⟩.

SIPSER, M. *Introduction to the Theory of Computation*. Cengage Learning, 2012. (Introduction to the Theory of Computation). ISBN 9781133187813. Disponível em: ⟨https://books.google.com.br/books?id=4J1ZMAEACAAJ⟩.

YUZUKI, R.; HATA, H.; MATSUMOTO, K. How we resolve conflict: an empirical study of method-level conflict resolution. In: IEEE. *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*. [S.l.], 2015. p. 21–24.

ZIMMERMANN, T. Mining workspace updates in cvs. *MSR*, p. 11–11, 2007.