**UNIVERSIDADE FEDERAL DO CEARÁ**

**CAMPUS DE QUIXADÁ**

**CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE**

**GUSTAVO IVENS OLIVEIRA SILVA**

# INVESTIGANDO FATORES QUE AFETAM O TEMPO DE CORREÇÃO DE ERROS DE INTEGRAÇÃO CONTÍNUA

**QUIXADÁ**

**2022**

GUSTAVO IVENS OLIVEIRA SILVA

# INVESTIGANDO FATORES QUE AFETAM O TEMPO DE CORREÇÃO DE ERROS DE INTEGRAÇÃO CONTÍNUA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Orientadora: Prof. Dra. Carla Ilane Moreira Bezerra

QUIXADÁ

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

---

S58i Silva, Gustavo Ivens Oliveira.
Investigando fatores que afetam o tempo de correção de erros de integração contínua / Gustavo Ivens Oliveira Silva. – 2022.
55 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Software, Quixadá, 2022.
Orientação: Profa. Dra. Carla Ilane Moreira Bezerra.

1. Integração Contínua. 2. Erros. 3. Software. 4. Regras de Associação. I. Título.

CDD 005.1

---

GUSTAVO IVENS OLIVEIRA SILVA

# INVESTIGANDO FATORES QUE AFETAM O TEMPO DE CORREÇÃO DE ERROS DE INTEGRAÇÃO CONTÍNUA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Aprovada em: ___/___/___

BANCA EXAMINADORA

---

Prof. Dra. Carla Ilane Moreira Bezerra (Orientadora)
Universidade Federal do Ceará (UFC)

---

Prof. Dr. Anderson Gonçalves Uchôa
Universidade Federal do Ceará (UFC)

---

Prof. Me. Camilo Camilo Almendra
Universidade Federal do Ceará (UFC)

Aos meus pais, Antonio Ferreira da Silva e Maria Helena da Silva Oliveira, por todo apoio e incentivo que me deram durante a minha trajetória acadêmica.

## AGRADECIMENTOS

Primeiramente a Deus por ter me concedido saúde para chegar até aqui e força para superar as dificuldades vividas durante o curso.

Aos meus pais, por sempre terem me incentivado a estudar e por todo o esforço que fizeram para que eu pudesse chegar até aqui.

A todos os meus familiares, que muitas vezes ajudaram a custear minhas viagens de volta para Quixadá e por todo o carinho que tem por mim.

À professora Dra. Carla Ilane Moreira Bezerra, por todos os conselhos e direcionamentos fornecidos para a conlusão deste trabalho.

Agradeço também a todos os amigos que fiz durante a minha trajetória na universidade, que me ajudaram em tantas situações e que fizeram desses quatro anos um período mais leve. Obrigado por terem divido tantos momentos comigo e por terem me permitido fazer parte da história de vocês.

Por fim, agradeço a todos que torceram por mim, que me ofereceram um momento de escuta, que me disseram palavras de incentivo e que me fizeram acreditar que este sonho era possível.

Muito Obrigado!

“Quando tudo for escuro e nada iluminar,
quando tudo for incerto
e você só duvidar...
É hora do recomeço.
Recomece a ACREDITAR.”

(Bráulio Bessa)

## RESUMO

Integração Contínua (IC) é uma importante prática da Engenharia de Software Moderna. IC sugere que os desenvolvedores integrem suas mudanças locais com a linha de base do projeto de modo frequente, idealmente diversas vezes aos dia, para tornar a incorporação dessas mudanças uma tarefa mais fácil e rápida. Tal prática é hoje amplamente adotada por diversos projetos de código aberto e fechado em razão principalmente dos benefícios que pode ocasionar ao projeto, tais como o aumento da qualidade do software produzido e uma maior agilidade no desenvolvimento. Diante dessa ampla adoção da prática de IC, diversas pesquisas empíricas tem sido realizadas para estudar diferentes aspectos ligados a este tema. Algumas dessas pesquisas enfocam a análise de *builds* falhos e demostraram que o tempo de correção de erros de IC, que são erros que ocorrem durante a integração das mudanças enviados pelo desenvolvedor, é bastante alto. Isto configura um fator problemático por conta das atividades de desenvolvimento efetivamente produtivas, como a adição de novas *features* e a correção de *bugs*, ficarem pausados enquanto os desenvolvedores trabalham na análise e correção dos erros de integração ocorridos. Desta forma, este trabalho tem como objetivo analisar a influência do perfil do desenvolvedor, das características do projeto e da complexidade do *build* no tempo de correção de erros de IC, para analisar o que faz a correção desse tipo de erro demorar mais ou menos tempo para ser concluída. Para atingir esse objetivo foram extraídos *builds* falhos de 18 projetos de código-fechado de uma grande empresa de desenvolvimento de software e calculado para cada um deles um conjunto de 12 métricas extraídas da literatura sobre análise de falhas de *build*. Para analisar a relação entre os fatores definidos e o tempo de correção, foi aplicado na base de dados formada pelo valor das métricas de cada *build* falho uma técnica de mineração de dados denominada extração de regras de associação. Isso permitiu revelar correlações significativas entre os fatores estudados e a duração do intervalo de correção de erros de IC. Os resultados obtidos sugerem que: (i) desenvolvedores mais experientes levam menos tempo para corrigir erros de IC, (ii) *builds* falhos originados em fases inicias do projeto são resolvidos em pequenos intervalos de tempo, e (iii) *builds* mais complexos podem tornar o intervalo de correção mais longo.



**Palavras-chave:** Integração Contínua. Erros. Software. Regras de Associação

## ABSTRACT

Continuous Integration (CI) is an important practice of Modern Software Engineering. CI suggests that developers integrate their local changes with the project baseline frequently, ideally several times a day, to make incorporating them more accessible and faster. This practice is now widely adopted by several open and closed-source projects mainly due to its benefits to the project, such as increased quality of the software produced and greater agility in development. Given this broad adoption of CI practice, several empirical kinds of research have been conducted to study different aspects of this topic. Some of these researches focus on the analysis of failed builds and have shown that the time to correct CI errors, which occur during the integration of changes submitted by the developer, is relatively high. This configures a problematic factor because the effectively productive development activities, such as adding new features and the correction of bugs, are paused. At the same time, the developers work on the analysis and correction of the integration errors that occurred. In this way, this work aims to analyze the influence of the developer profile, the characteristics of the project and the complexity of the build on the correction time of CI errors to analyze what makes the correction of this type of error take time more or less time to complete. To achieve this goal, flawed builds were extracted from 18 closed-source projects of a large software development company and a set of 12 metrics extracted from the literature on the analysis of build flaws were calculated for each of them. To analyze the relationship between the defined factors and the correction time, a data mining technique called association rule extraction was applied to the database formed by the value of the metrics of each failed build. This allowed revealing significant correlations between the factors studied and the duration of the CI error correction interval. The results obtained suggest that: (i) more experienced developers take less time to correct CI errors, (ii) builds flaws that originated in early phases of the project are resolved in small time intervals, and (iii) more complex builds can make the fixed interval longer.



**Keywords:** Continuous Integration. Errors. Software. Association Rules

## LISTA DE FIGURAS

## LISTA DE QUADROS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| IC | Integração Contínua |
| XP | *Extreme Programing* |
| SCV | Sistema de Controle de Versões |
| MWW | Mann-Whitney-Wilcoxon |
| CSV | *Comma Separated Values* |

## SUMÁRIO

# 1 INTRODUÇÃO

A Integração Contínua IC é um processo de engenharia de software que prevê, dentre outras coisas, a constante incorporação de pequenas mudanças no código fonte à *branch* principal de um repositório compartilhado (BELLER *et al.*, 2017a). IC visa aumentar a qualidade do código e reduzir o tempo decorrido entre a implementação de uma nova versão do software e sua consequente disponibilização para o ambiente de produção, através da automação e execução frequente de um conjunto de atividades, tais como *build*, teste e análise estática de código.

Nos últimos anos, a IC tem sido cada vez mais adotada na indústria e abordada por pesquisas em engenharia de software (HASSAN, 2019; RAUSCH *et al.*, 2017). Neste sentido, estudos anteriores como os de Hilton *et al.* (2016) e Elazhary *et al.* (2021), se dedicaram a analisar de maneira empírica como práticas de IC vêm sendo aplicadas em projetos de software, dando ênfase aos custos, desafios, benefícios e implicações que a adoção dessas práticas traz para as empresas e projetos de software. Outros trabalhos, concentram-se na investigação de más práticas de IC, que consistem de escolhas precipitadas quanto a forma de adotar IC e que podem trazer consequências negativas aos projetos, como por exemplo a dimuição da qualidade do sistema e um maior esforço para concluir as atividades de integração. Posto isso, trabalhos como os de Zampetti *et al.* (2020) e Vassallo *et al.* (2019) enfocam a definição e identificação de más práticas de IC, enquanto trabalhos como o de Silva e Bezerra (2020) concentram-se na avaliação de seus efeitos sobre a qualidade de software dos projetos. Há também estudos que investigam como a IC interage e influencia outras práticas de desenvolvimento de software, como por exemplo, a revisão de código baseada em *pull requests* (RAHMAN; ROY, 2017; ZAMPETTI *et al.*, 2019).

Erros de IC são erros que ocorrem durante a compilação e teste das alterações enviadas ao repositório compartilhado, e que resultam em *builds* falhos. Kerzazi *et al.* (2014) e Rausch *et al.* (2017) apontam que *builds* com falha podem levar a ineficências e prejudicar profundamente o processo de desenvolvimento, a medida que atrasam o projeto enquanto o problema está sendo analisado e corrigido e que podem bloquear o trabalho de outros membros da equipe.

Pesquisas recentes em erros de IC tem investigado principalmennte as causas ou razões fundamentais que levam a ocorrência de *builds* com falha, através da mineração de dados provenientes de reposítórios de software, com o intuito de encontrar indicadores quantificáveis que possam ser utilizados para prever o sucesso ou a falha de um *build* (ISLAM; ZIBRAN,

2017; KERZAZI *et al.*, 2014; RAUSCH *et al.*, 2017). Ainda nesse contexto, trabalhos como os de Hassan (2019) e Vassallo *et al.* (2020) estudam os erros mais comuns em *builds* defeituosos para caracterizar padrões de correção e implementar ferramentas de correção automática.

Nesse sentido, Vassallo *et al.* (2020) ressalta em seu trabalho que desenvolvedores não só precisam encontrar a causa da falha de *build* e consertá-la, mas também agir rápido para evitar atrasos. Dentro deste cenário, Silva e Bezerra (2020) revelam por meio da análise do histórico de *builds* que o tempo despendido para a correção de erros de *build* em projetos comerciais é alto, chegando a meses em alguns casos. Com isso, compreender o que faz uma correção ser mais ou menos demorada pode ser útil para reduzir sua duração.

Dessa forma, este trabalho tem como proposta investigar, por meio de um estudo exploratório com projetos de código fechado, como fatores relacionados (i) ao perfil dos desenvolvedores, (ii) as características do projeto e (iii) a complexidade do *build*, afetam o tempo de correção de erros de IC. Espera-se que uma compreensão abrangente dessa relação possa auxiliar os desenvolvedores a refletir sobre o processo de desenvolvimento e definir ações para melhorar sua eficácia.

O objetivo geral deste trabalho é investigar como o tempo de correção de erros de IC é afetado por características dos desenvolvedores, dos projetos e do próprio *build*. Como objetivos específicos deste trabalho estão: (i) definir um conjunto de métricas para representar aspectos do perfil dos desenvolvedores, dos projetos e da complexidade do *build*; (ii) desenvolver um *script* para calcular essas métricas a partir de dados obtidos de repositórios de projetos de código fechado e (iii) identificar as correlações mais significativas entre as métricas definidas e o tempo de correção de erros de IC.

O restante deste trabalho está organizado da seguinte maneira. No Capítulo 2 são apresentados os principais conceitos para o desenvolvimento deste trabalho. No Capítulo 3 realiza-se a apresentação e discussão dos trabalhos relacionados, ressaltando suas semelhanças e diferenças ao trabalho aqui proposto. No Capítulo 4 a metodologia empregada para o desenvolvimento deste trabalho é explicada. No Capítulo 5 os resultados obtidos são apresentados e discutidos. E por fim, no Capítulo 6 são apresentados a conclusão e trabalhos e futuros.

# 2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentados os conceitos principais necessários para o entendimento e desenvolvimento deste trabalho. A Seção 2.1 aborda o conceito de IC retratando sua origem, principais tarefas e funcionamento, bem como, a definição de erros de IC. A Seção 2.2 explica o conceito de mineração de repositórios de software e a Seção 2.3 trata do processo de mineração de dados explicando seus principais tipos de análise e aprofundando a técnica de mineração de regras de associação que é empregada neste trabalho.

## 2.1 Integração contínua

A Integração Contínua (IC) é uma prática da Engenharia de Software Moderna que prega a integração frequente de pequenas mudanças no código com a ramificação principal de um repositório compartilhado. Ela surgiu como uma das práticas ágeis proposta pela metodologia *Extreme Programing* (XP), sobre a ideia de que grandes integrações de código são uma fonte de dor para os desenvolvedores, pois eles tem que resolver de maneira manual diversos conflitos. Dessa maneira, IC recomenda integrar o código de maneira frequente, pois isso faz com que as integrações sejam menores e que, portanto, a quantidade de conflitos seja reduzida (VALENTE, 2020).

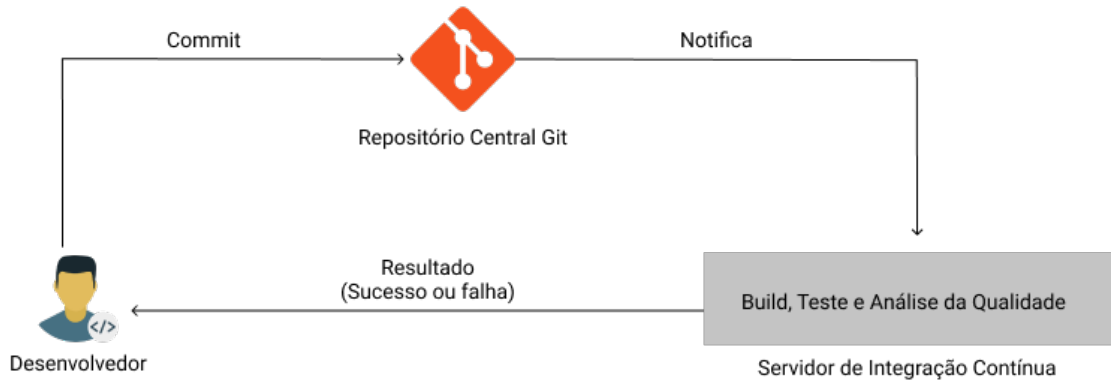
Um *pipeline* de IC diz respeito ao conjunto de etapas que devem ser executadas para disponibilizar uma nova versão do software. Um *pipeline* habitual de IC costuma ser formado por três etapas essenciais:

- ***build:*** onde ocorre a compilação de todos os arquivos de código para gerar uma versão executável do sistema. Dependendo da linguagem adotada para o desenvolvimento do projeto esse passo pode não integrar o *pipeline* de IC.
- **teste**: onde é verificado se as mudanças inseridas comprometem o comportamento do sistema. Essa etapa ocorre a partir da execução de uma série de testes automatizados, tais como testes de unidade, testes de integração e testes de sistema.
- **análise estática**: onde o código é analisado sintaticamente para verificar se o projeto continua atendendo as métricas de qualidade definidas pela equipe.

A Figura 1 ilustra o funcionamento do processo de IC. Um servidor de IC é o mecanismo central para utilização de IC em um projeto de desenvolvimento de software. Ele permite a execução automatizada das atividades comentadas acima e de outras atividades que possam

compor o processo adotado pela organização. Seu papel é evitar que códigos problemáticos sejam mesclados com a linha principal do projeto e notificar os desenvolvedores sobre o resultado de suas mudanças. O servidor de IC trabalha monitorando o Sistema de Controle de Versões (SCV), como por exemplo o Git, para capturar novas alterações no código do projeto. Assim, quando um desenvolvedor realiza um *push* em sua máquina, isto é, envia suas modificações (*commits*) para o SCV, o servidor de IC clona o repositório e inicia o *build* do projeto. Após o *build* ser concluído, os testes automatizados são executados e quando nenhum deles falha, executa-se a análise estática para verificar a conformidade com os padrões de qualidade. Por fim, com o código aprovado o desenvolvedor pode fazer o *merge* de suas mudanças com à linha principal do projeto ou caso tenha recebido um resultado de falha deve analisar o problema e trabalhar em novas alterações para corrigir o *build* (MEYER, 2014).

Figura 1 – Processo de integração contínua.




Fonte: Adaptado de Valente (2020).

Existem atualmente uma série de servidores de IC disponíveis no mercado como Travis CI[1], Jenkins[2], Gitlab CI/CD[3], CircleCI[4] e Team City[5]. A maioria deles funciona de maneira independente e não necessita ser instalado localmente. Ao invés disso, eles costumam ser implementados como serviços que podem ser acoplados aos sistemas de hospedagem de código como o GitHub[6] e o GitLab[7]. Neste trabalho, os projetos alvo do estudo são aqueles que utilizam como ferramenta de hospedagem de código o GitHub e como servidor de IC uma

[1] Disponível em: <https://travis-ci.org/> Acesso em: 16 ago. 2021
[2] Disponível em: <https://www.jenkins.io/> Acesso em: 16 ago. 2021
[3] Disponível em: <https://docs.gitlab.com/ee/ci/> Acesso em: 16 ago. 2021
[4] Disponível em: <https://circleci.com/> Acesso em: 16 ago. 2021
[5] Disponível em: <https://www.jetbrains.com/pt-br/teamcity/> Acesso em: 16 ago. 2021
[6] Disponível em: <https://github.com/> Acesso em: 16 ago. 2021
[7] Disponível em: <https://about.gitlab.com/> Acesso em: 16 ago. 2021

ferramenta interna mantida pela organização parceira.

### *2.1.1 Erros de integração contínua e seu tempo de correção*

A execução de todas as etapas do *pipeline* de IC é tipicamente chamada de *build* ou construção. Quando ocorre falha durante a execução de uma dessas fases costuma-se dizer que o *build* quebrou. Um *build* quebrado indica que há problemas com as modificações que o desenvolvedor enviou ao repositório e consertá-lo é uma tarefa que deve ser feita o quanto antes (VALENTE, 2020). De maneira semelhante, os erros que causam a quebra do *build* são chamados de falhas de *build* e nesse trabalho são o que identificamos como erros de IC.

O tempo de correção de erros de IC compreende o intervalo de tempo entre um *build* quebrado e a próximo *build* executado com sucesso e representa um ponto central do processo de IC (SILVA; BEZERRA, 2020). Estudos recentes investigaram de maneira indireta o tempo de correção desses erros e verificaram que no geral ele é alto. Por exemplo, em seu trabalho Silva e Bezerra (2020) analisaram os intervalos de correção de *builds* defeituosos de dois projetos industriais através dos registros de *log* do Travis CI, e observaram que mais da metade dos *builds* com falha presentes nos dois projetos levaram mais de três horas para serem consertados. Outro resultado interessante deste estudo, é o fato de que em alguns casos a correção do *build* se estendeu ao longo de meses, o que vai de encontro a recomendação de corrigir o *build* o mais rápido possível.

Assim, erros de IC configuram um ponto problemático dentro do processo de desenvolvimento de software em virtude, principalmente, do atraso que podem ocasionar ao andamento do projeto. Isso ocorre porque as atividades de desenvolimento e liberação de *features* ficam presas enquanto o problema está sendo analisado e corrigido pelos desenvolvedores (KERZAZI *et al.*, 2014). De maneira geral, quanto mais tempo você leva pra corrigir um erro, menos tempo você emprega em atividades efetivamente produtivas e mais você retarda o envio de novas *features* e correções de *bugs* para o cliente (MEYER, 2014).

Um *build* realizado com sucesso funciona também como um atestado de que as mudanças enviadas pelo desenvolvedor não afetam o comportamento e nem a qualidade do sistema. Quando os desenvolvedores quebram o *build* eles estão impedindo que outros desenvolvedores da equipe façam suas alterações com confiança. Nesse sentido, quando o *build* está quebrado e outros desenvolvedores continuam fazendo mudanças e enviando-as torna-se muito mais difícil descobrir se suas alterações falharam ou não, tornando a depuração mais complexa a cada novo

*commit* (MEYER, 2014). Com isso, quanto mais tempo o *build* permanece quebrado maior é a chance de outros membros da equipe clonarem suas alterações incorretas e enviarem mais códigos ao repositório, ativando novos *builds* errados e fazendo com que se empregue ainda mais tempo em atividades de correção de erros.

## 2.2 Mineração de repositórios de software

Durante a manutenção e desenvolvimento de projetos de software uma série de dados são gerados a partir das diversas atividades desempenhadas pelos desenvolvedores, tais como o desenvolvimento, a revisão e integração de código. Nesse sentido, repositórios de software são os locais que concentram os artefatos de um projeto de software, como por exemplo, a sua base de código, histórico de versões, *bugs* relatados e discussões de mudança (GÜEMES-PEÑA *et al.*, 2018). Desta forma, a mineração de repositórios de software é uma área da Engenharia de Software Empírica que se concentra na utilização de ferramentas e técnicas de mineração de dados para descobrir informações úteis e acionáveis a partir dos dados presentes em repositórios de software. As informações descobertas a partir das atividades de mineração de repositórios de software permitem um melhor entendimento do processo de desenvolvimento e evolução do software e fornecem apoio a tomada de decisões importantes por parte dos desenvolvedores e gerentes (MSR, 2022).

Existe atualmente uma ampla adoção de plataformas de hospedagem de código baseadas em git, como o GitHub e GitLab, entre projetos industriais e abertos. Essas plataformas mantém APIs REST que possibilitam a exploração dos dados históricos dos milhares de repositórios que abrigam de maneira simples e direta, fazendo com que sejam, portanto, uma importante fonte de dados para pesquisas que enfocam a análise empírica de fenômenos ligados ao desenvolvimento de software. Um exemplo recente de trabalho que utiliza a mineração de repositórios de software para estudar aspectos ligados a construção de sistemas é o trabalho de Coutinho *et al.* (2022). Neste trabalho, os autores coletaram aproximadamente 45 mil *commits* de sete projetos de software para analisar, por meio da extração de regras de associação, quais características ligadas ao processo e ao desenvolvedor influenciam a degradação do design do projeto. Outro trabalho recente que envolve mineração de repositórios de software é o trabalho de Soares *et al.* (2021), que extraiu dados de mais de 97 mil pull requests para estudar, também através da mineração de regras de associação, como aspectos sociotécnicos e características ligadas aos contribuidores externos e ao próprio processo de contribuição afetam o tempo de

vida de *pull requests* em projetos abertos.

Posto isto, este trabalho enfoca a mineração de repositórios de software a partir da coleta de dados de *commits*, desenvolvedores e *builds* de 18 projetos de software fechados para investigar, também por meio da extração de regras de associação, como aspectos relacionados ao perfil dos desenvolvedores, as características do projeto e a complexidade do *build* influenciam o intervalo de tempo necessário para corrigir erros de IC.

## 2.3 Mineração de dados

Mineração de dados ou *data mining* é o processo de descobrir automaticamente informações úteis em grandes repositórios de dados. Técnicas de *data mining* são empregadas para analisar grandes bases de dados com o intuito de encontrar padrões implícitos, inicialmente desconhecidos e que dificilmente seriam encontrados por um observador humano ou através de técnicas e algoritmos tradicionais de exploração de dados (TAN *et al.*, 2014)

A mineração de dados surgiu como resultado do esforço de pesquisadores de diferentes áreas para enfrentar os desafios impostos pelos novos *datasets* às técnicas de análise tradicional. Dentre esses desafios, destacam-se (i) a escalabilidade, que está ligada ao tamanho cada vez maior que as bases de dados vem assumindo; (ii) a alta dimensionalidade, que diz respeito a utilização de centenas e até mesmo milhares de atributos para representar os dados; e a (iii) heterogeneidade e complexidade dos dados, que estão associados, respectivamente, à análise de bases de dados cujos registros são formados por atributos de diferentes tipos e ao surgimento de objetos de dados mais complexos, como dados climáticos e de DNA. Assim, a mineração de dados é formada por ideias provenientes de diferentes áreas, sobretudo, da estatística, inteligência artificial, banco de dados e computação paralela e distribuída (HAN *et al.*, 2011; TAN *et al.*, 2014).

Uma tarefa de mineração de dados se refere ao tipo de análise que se deseja executar ou ao tipo de padrão que se quer buscar nos dados. Desse modo, existem quatro tarefas principais de mineração de dados, são elas (TAN *et al.*, 2014):

– **Análise preditiva:** engloba os modelos que tem por objetivo determinar o valor de uma variável de interesse com base nos demais atributos que compõem um registro. Para essa tarefa, dois modelos principais podem ser usados, a classificação e a regressão. A classificação é usada quando a variável que se deseja conhecer assume um valor categórico e consiste no processo de prever a qual classe pertencem os objetos para os quais o valor

da classe ainda é desconhecido. A regressão, por sua vez, é usada para prever o valor de variáveis contínuas.

- **Análise associativa:** diz respeito a procura por padrões de relacionamento entre os atributos que compõem os dados. Dessa forma, a ideia geral por trás desta tarefa é encontrar elementos que costumam aparecer juntos ou cuja ocorrência resulte na presença de um outro. Exemplos de padrões extraídos com esse tipo de tarefa são as regras de associação e os padrões sequenciais.
- **Análise de *clusters*:** se refere a tarefa de separar os objetos de dados em grupos ou *clusters*, de modo que os objetos dentro de um *cluster* sejam o mais semelhante possível e ao mesmo tempo bastante diferentes de objetos de outros *clusters*. Uma das principais aplicações da análise de *clusters* é a formação de perfis de clientes.
- **Análise de anomalias:** é a tarefa de identificar objetos cujas características fogem do padrão apresentado pelo restante dos dados. Esses objetos discrepantes são chamados de anomalias ou *outliers*. A ideia central de um algoritmo de detecção de anomalias é garantir a detecção desses objetos incomuns e evitar ao máximo marcar um objeto que obedece aos padrões apresentados pela base de dados como *outlier*, isto é, gerar falsos positivos. A principal aplicação da análise de anomalias consiste na identificação de fraudes, como por exemplo as que podem ocorrer com as compras efetuadas por cartão de crédito.

Neste trabalho, foi executada a tarefa de análise associativa através da implementação de um modelo mineração de regras de associação muldimensionais, com o intuito de verificar correlações entre o perfil do desenvolvedor, as características do projeto e a complexidade do *build* com o tempo de correção de erros de IC.

### *2.3.1 Mineração de regras de associação*

A mineração de regras de associação é uma tarefa de mineração de dados descritiva que tem como objetivo encontrar padrões comuns e correlações significativas em um conjunto de dados. Regras de associação representam padrões de relacionamento que ocorrem com frequência em uma base de dados e que indicam a existência de uma forte ligação entre os itens de dados (TAN *et al.*, 2014; SOARES *et al.*, 2021).

A definição formal de uma regra de associação é a seguinte: Dado um conjunto de itens $I = \{i_1, i_2, i_3, ..., i_n\}$ e uma base de dados de transações $T = \{t_1, t_2, t_3, ..., t_n\}$ onde cada cada transação $t_i$ é constituída por um conjunto não vazio de itens pertencentes a $I$, com cada transação

possuindo um identificador único chamado TID, uma regra de associação é uma implicação da forma $X \rightarrow Y$, onde $X$ e $Y$ são conjuntos disjuntos de items pertencentes a $I$ e que ocorrem nas transações de $T$ (HAN *et al.*, 2011).

Dessa maneira uma regra de associação $X \rightarrow Y$ sugere um relacionamento entre o conjunto de itens $X$ e o conjunto de itens $Y$ indicando que a ocorrência do antecedente $X$ pode resultar ou aumentar a chance da ocorrência do consequente $Y$. Dito isto, é importante destacar que a quantidade de regras que podem ser mineradas em uma base de dados qualquer é exponencial e que, além disso, boa parte delas não é relevante, pois podem ocorrer por acaso. Portanto, a ideia central de um algoritmo de extração de regras de associação é encontrar o conjunto de regras mais interessantes dentre aquelas que podem ser extraídas.

Sendo assim, duas métricas essencias são empregadas para mensurar a relevância ou a força de uma regra de associação, são elas: Suporte e Confiança. O Suporte de uma regra $X \rightarrow Y$ se refere a proporção de transações em $T$ que possuem ao mesmo tempo o conjunto de itens $X$ e o conjunto de itens $Y$ e pode ser calculado pela expressão a seguir: $Sup(X \rightarrow Y) = \frac{|T_{X \cup Y}|}{|T|}$, onde $T_{X \cup Y}$ é o conjunto das transações que possuem $X$ e $Y$ e $T$ é o conjunto de todas as transações. A Confiança, por sua vez, analisa a validade de uma regra calculando para as transações que contém $X$ a proporção daquelas que também contém $Y$, significando assim a probabilidade de ocorrer $Y$ uma vez que $X$ já ocorreu. A Confiança é calculada através da seguinte expressão: $Conf(X \rightarrow Y) = \frac{Sup(X \rightarrow Y)}{Sup(X)}$, onde $Sup(X)$ corresponde a quantidade transações que possuem $X$ (TAN *et al.*, 2014).

Deste modo, o problema formal da mineração de regras de associação consiste em encontrar para uma determinada base de dados todas as regras que atendem a um Suporte e Confiança mínimos definidos pelo usuário. Assim sendo, os algoritmos que endereçam este problema costumam dividí-lo em dois subproblemas. O primeiro deles é encontrar conjuntos de items que atendam ao Suporte mínimo especificado, chamados de *frequent itemsets*, e o segundo é gerar todas as regras de associação que atendam ao limite mínimo de Confiança a partir dos *frequent itemsets* extraídos (HAN *et al.*, 2011; TAN *et al.*, 2014).

Além do Suporte e Confiança, outra métrica que pode ser usada para filtrar regras de associação importantes é o *Lift*. Essa métrica indica o quão mais provável torna-se a ocorrência do conjunto $Y$, dado que o conjunto $X$ já ocorreu. Ela serve para avaliar se a relação/influência entre o antecedente $X$ e o consequente $Y$ é nula, positiva ou negativa. Uma relação nula ocorre quando $Lift = 1$ indicando que $X$ não interfere na ocorrência de $Y$. Por outro lado, uma relação

positiva ocorre quando $Lift > 1$, demonstrando que $X$ aumenta as chances de ocorrência de $Y$, enquanto uma relação negativa ocorre quando $Lift < 1$ revelando que a presença de $X$ diminui as chances de ocorrência de $Y$ (SOARES *et al.*, 2021).

Um dos cenários mais típicos onde a mineração de regras de associação pode ser aplicada é a análise da cesta de compras do mercado. Neste cenário, deseja-se saber quais produtos costumam ser comprados juntos para possibilitar o desenvolvimento de estratégias de venda. Posto isso, cada item presente em uma transação do conjunto de dados representa a presença de um produto na cesta de compras e o predicado ou dimensão das regras mineradas é a compra de certos produtos. Regras desse tipo, onde somente uma única dimensão é estudada e que envolvem dados com valores categóricos, são conhecidas como regras de associação unidimensionais e são extraídas a partir de bases de dados transacionais.

O Quadro 1 ilustra uma base de dados transacional para a análise da cesta de mercado. Assumindo um percentual de 20% como Suporte mínimo e de 50% como Confiança mínima uma regra relevante que pode ser extraída da base de dados ilustrada é $\{Leite, Fraldas\} \rightarrow \{Cerveja\}$, que indica que pessoas que compram Leite e Fraldas em uma mesma compra também costumam ou tem mais chance de comprar cerveja. Essa regra apresenta Suporte de 40%, visto que os items Leite, Fraldas e Cerveja aparecem juntos em 2 das 5 transações, e Confiança de 67%, uma vez que ocorrem 3 transações com os produtos Leite e Fraldas e em somente 2 dessas também aparece o produto Cerveja.

Quadro 1 – Exemplo de base de dados transacional para o cenário da análise de cesta de compras.

| **TID** | **Produtos comprados** |
|---|---|
| 1 | {Pão, Leite} |
| 2 | {Pão, Fraldas, Cerveja, Ovos} |
| 3 | {Leite, Fraldas, Cerveja, Refrigerante} |
| 4 | {Pão, Leite, Fraldas, Cerveja} |
| 5 | {Pão, Leite, Fraldas, Refrigerante} |

Fonte: Adaptado de Tan *et al.* (2014).

Neste trabalho, são mineradas regras de associação multidimensionais, que são aquelas que possuem mais de um predicado ou dimensão e que costumam envolver, além de dados categóricos, dados quantitativos. Assim, as dimensões presentes nas regras mineradas neste trabalho estão relacionadas as métricas utilizadas para extrair informações dos desenvolvedores e projetos. A definição formal de uma regra de associação multidimensional é dada a seguir.

Dada uma base de dados $D$, uma regra de associação muldimensional $X \rightarrow Y$ é uma

implicação na forma: $X_1 \wedge X_2 \wedge ... \wedge X_n \rightarrow Y_1 \wedge Y_2 \wedge ... \wedge Y_m$, onde $1 \leq n$, $1 \leq m$ e $X_i(1 \leq i \leq n)$ e $Y_i(1 \leq i \leq m)$ são condições definidas em termos de atributos diferentes de $D$ (SOARES *et al.*, 2021).

Dessa forma, agora a ocorrência do antecedente $X$ e do consequente $Y$ só é considerada quando todas as condições definidas para eles são satisfeitas. Para ilustrar esse conceito considere o Quadro 2, que apresenta uma base de dados relacional para o problema da análise da cesta de mercado, mas que agora contém, além dos itens que são comprados, informações sobre o perfil dos clientes.

Quadro 2 – Exemplo de base de dados relacional para o cenário da análise de cesta de compras.

| **TID** | **Produtos comprados** | Gênero | Idade |
|---|---|---|---|
| 1 | {Pão, Leite} | Feminino | 17 |
| 2 | {Pão, Fraldas, Cerveja, Ovos} | Masculino | 25 |
| 3 | {Leite, Fraldas, Cerveja, Refrigerante} | Masculino | 21 |
| 4 | {Pão, Leite, Fraldas, Cerveja} | Masculino | 28 |
| 5 | {Pão, Leite, Fraldas, Refrigerante} | Masculino | 30 |
| 6 | {Pão, Refrigerante} | Feminino | 19 |

Fonte: Adaptado de Tan *et al.* (2014).

As regras mineradas a partir da base de dados hipotética apresentada no Quadro 3 podem ter até 3 dimensões, pois os dados são representados por meio de 3 atributos (produtos, gênero e idade). Uma das regras que podem ser extraídas mantendo o Suporte mínimo de 20% e a Confiança mínima de 50% é $\{Genero = Masculino \wedge Idade \geq 18\} \rightarrow \{Produto = Cerveja\}$. Para essa regra o Suporte é de 50%, pois metade das transações obdecem as condições presentes no antecedente e consequente, enquanto a Confiança é de 75%, pois 3 das 4 transações que satisfazem as condições do antecedente resultam também na compra do produto cerveja.

Por fim, é importante ressaltar que para conseguir extrair regras de associação muldimensionais em casos que envolvem valores numéricos é necessário primeiro realizar uma discretização nos dados, a fim de criar classes de intervalo nas quais os dados possam ser encaixados.

# 3 TRABALHOS RELACIONADOS

Neste capítulo, serão apresentados alguns trabalhos relacionados que serviram como base teórica e prática para o desenvolvimento deste trabalho. Foram considerados trabalhos relacionados, estudos recentes que usam mineração de repositórios de software no contexto de erros de IC e que aplicaram regras de associação.

## 3.1 *Insights Into Continuous Integration Build Failures*

Em Islam e Zibran (2017), são investigados os fatores que causam falhas de *build* a partir de um grande estudo empírico realizado com vários projetos de código fonte aberto. Os fatores cujos autores concentram seu trabalho são: (i) a complexidade de uma tarefa, visto essencialmente como a quantidade de linhas e de arquivos alterados em cada *push*, (ii) a estratégia de construção e modelo de desenvolvimento,isto é, desenvolvimento baseado no *trunk* e desenvolvimento baseado em *pull request* e (iii) atributos de nível de projeto, a saber, o tamanho do projeto e das equipes de desenvolvimento.

Deste modo, o objetivo principal do trabalho foi compreender, a partir de evidências quantitativas, como esses fatores influenciam o resultado de um *build* de IC, de modo que fosse possível explicar por quais razões um *build* quebra e refletir sobre o que pode ser feito para evitar essa quebra.

Para realizar a pesquisa os autores utilizaram uma coleção de dados elaborada (BELLER *et al.*, 2017b) que consistia de código fonte e informações de *builds* de 1090 projetos de software de código fonte aberto, desenvolvidos nas linguagens Ruby e Java. Tal coleção, foi montada combinando informações obtidas a partir do Travis CI, do Git e do GitHub. Para analisar a relação entre os fatores definidos e o resultado da construção, analisou-se a significância estatística das correlações estabelecidas a partir do teste de Mann-Whitney-Wilcoxon (MWW) e do teste *Chi-squared*. Como resultado, descobriu-se que a complexidade de uma tarefa e as estratégias de *build* e modelo de desenvolvimento afetam fortemente a falha de um *build*.

As principais diferenças entre o trabalho realizado por Islam e Zibran (2017) e o apresentado neste trabalho, é que este trabalho busca identificar fatores que influenciam no tempo de correção de falhas de *build*, ao invés de fatores que podem originar falhas de *build*. Além disso, no trabalho citado as informações que compõem a base de dados utilizada são extraídas de projetos de código fonte aberto. Neste trabalho, as informações que representam os fatores alvo

do estudo foram extraídas de projetos de código fonte fechado.

## 3.2 *Why Do Automated Builds Break? An Empirical Study*

Em Kerzazi *et al.* (2014) investiga-se, através de um estudo empírico com um grande sistema industrial, o impacto que *builds* quebrados trazem para um projeto, as circunstâncias em que ocorrem essas falhas e quais os fatores relevantes para conseguir predizer o resultado de um *build*.

Para realizar o trabalho, em vez de estudar vários projetos em um nível mais elevado, os autores optaram por estudar um único projeto de maneira detalhada, combinando análise qualitativa e quantitativa para gerar resultados válidos e significativos dentro de um ambiente industrial. Dessa forma, o estudo foi conduzido dentro de uma grande organização de desenvolvimento de software, que tinha como principal produto um sistema financeiro baseado na *web*. Desse sistema, foram extraídas informações de 3.214 *builds* produzidos durante um período de 6 meses para compor a base de dados necessária para executar a análise quantitativa.

Para mensurar o impacto das falhas de *build* no projeto, os autores examinaram o seu custo direto, defininido como o custo da equipe que causa ou depende diretamente do código quebrado. Assim, para cada *branch* calculou-se o intervalo de tempo entre cada *build* quebrado e o próximo executado com sucesso e identificou-se também quantas pessoas estavam ligadas a essa *branch*. Deste modo, o custo das falhas em uma *branch* é dado pelo intervalo de tempo que o *build* permaneceu quebrado multplicado pela quantidade de pessoas envolvidas. Esse cálculo estima a quantidade de horas-homem perdida em cada *branch* e reflete um valor monetário. Como resultado, verificou-se que o tempo médio de quebra do *build* em uma *branch* varia de 35,7 a 85,3 minutos e que durante os 6 meses considerados no estudo foram perdidas no mínimo 893 horas-homem.

As circunstâncias em que ocorrem as falhas foram analisadas através de entrevistas semiestruturadas com 28 engenheiros de software da empresa. As entrevistas compõem a parte qualitativa do estudo e foram feitas para identificar possíveis fatores relacionados as falhas de *build* que serviram como entrada para a última etapa do estudo, que era determinar quais desses fatores estavam fortemente ligados a ocorrência dessas falhas.

Para descobrir quais fatores mantinham forte relação com *builds* defeituosos empregou-se alguns modelos estatísticos. Para fatores representados como variáveis independentes categóricas foi utilizado o teste *Chi-squared* e para os aspectos representados como variáveis

independentes contínuas utilizou-se o teste MWW. Além disso, a importância de cada variável na explicação da ocorrência dos erros de *build* foi calculada usando um modelo *Randon Forest*.

Como principais resultados para essa última etapa do estudo, os autores encontraram que a função do desenvolvedor, o número de colaboradores na *branch* e a natureza do trabalho (novo recurso, correção de *bug*, etc) são fatores que estão fortemente ligados a quebra do *build* e que podem ser usados para prever sua ocorrência.

As diferenças entre este trabalho e o de Kerzazi *et al.* (2014), são que este trabalho investiga um número maior de projetos industriais e identifica fatores que explicam o tempo de correção de erros de IC, à medida que o trabalho citado estuda um único projeto industrial para identificar aspectos fortemente ligados a ocorrência de erros de IC.

## 3.3 *What Factors Influence The Lifetime Of Pull Requests?*

Em Soares *et al.* (2021) realizou-se um estudo empírico para identificar as causas raiz do tempo de vida de *pull requests*. O tempo de vida de uma *pull request* é definido como o intervalo de tempo entre a sua abertura e seu fechamento e foi estudado levando em consideração aspectos sociotécnicos e características dos contribuidores externos e do processo de contribuição.

Para realizar o estudo, os autores coletaram, através da ferramenta GHTorrent[1], 97.467 *pull requests* de 30 projetos de código aberto hospedados no GitHub. Esses projetos foram selecionados de forma aleatória e atendendo a condição de terem pelo menos 500 *pull requests* registradas, considerando o período de janeiro de 2011 a outubro de 2015. Em seguida, para encontrar correlações significativas entre os aspectos estudados e o tempo de vida de *pull requests*, que representam as causas raiz do tempo de vida, eles utilizaram uma técnica de mineração de dados descritiva chamada de extração de regras de associação a partir do algoritmo Apriori, disponível na ferramenta de mineração Weka. Por fim, uma pesquisa qualitativa foi executada com o intuito de compreender melhor os padrões de relacionamento identificados na etapa de mineração.

Como resultados, observou-se que (i) há uma relação efetiva entre o tempo de vida de uma *pull request* e sua aceitação; (ii) as características estruturais afetam de maneira isolada ou combinada o tempo de vida; (iii) as características dos contribuidores externos mantém influência sobre a duração da *pull request* e que (iv) os artefatos modificados e sua localização, assim como

[1] Disponível em: <https://ghtorrent.org/> Acesso em: 26 jul. 2021

o número de comentários e o revisor são fortes preditores para o tempo de vida.

A principal diferença entre o trabalho de Soares *et al.* (2021) e este trabalho, é que o trabalho citado investiga o tempo de vida *pull requests* a partir de dados coletados de projetos de código aberto, enquanto este trabalho investiga o intervalo de correção de erros de IC a partir de dados coletados de projetos de código fonte fechado.

## 3.4 Análise comparativa

Assim como ocorre em Kerzazi *et al.* (2014) e em Islam e Zibran (2017), este trabalho também aborda como tema geral falhas de *build* em ambientes de IC. No entanto, diferente do que ocorre nestes dois trabalhos, que se dedicam a identificar fatores que causam falhas de *build*, este trabalho visa identificar fatores que possam justificar a variação no tempo de correção dessas falhas.

Islam e Zibran (2017) realizaram seu trabalho analisando dados de compilações extraídas de vários projetos de código aberto do GitHub e não consideram em sua análise a influência das características dos profissionais. Já Kerzazi *et al.* (2014), identificaram aspectos relacionados a falhas de *build* analisando de maneira profunda um grande sistema industrial, através de uma abordagem que contou com análise qualitativa e quantitativa. Entretanto, os aspectos estudados por ele não englobaram características relacionadas ao projeto.

Desta forma, este trabalho diverge dos acima citados porque estuda como fatores relacionados ao perfil dos desenvolvedores, as características dos projetos e a complexidade do *build* afetam o intervalo de correção de erros de IC em projetos de código fonte fechado. Para isso foi empregada uma metodologia que realiza a análise descritiva de *builds* falhos, a partir de uma técnica de mineração de dados chamada de extração de regras de associação.

Por fim, assim como feito em Soares *et al.* (2021), este trabalho tem como objetivo analisar o tempo necessário para executar uma tarefa de desenvolvimento de software. Entretanto, o trabalho citado utliliza a mineração de regras de associação para encontrar características que influenciam o tempo de vida de uma *pull request*, ao passo que este utliliza mineração de regras de associação para analisar aspectos que afetam o tempo de correção de erros de IC.

O Quadro 3 apresenta as principais características encontradas nos trabalhos relacionados e a comparação com o presente trabalho.

Quadro 3 – Análise comparativa entre os trabalhos relacionados e este trabalho.

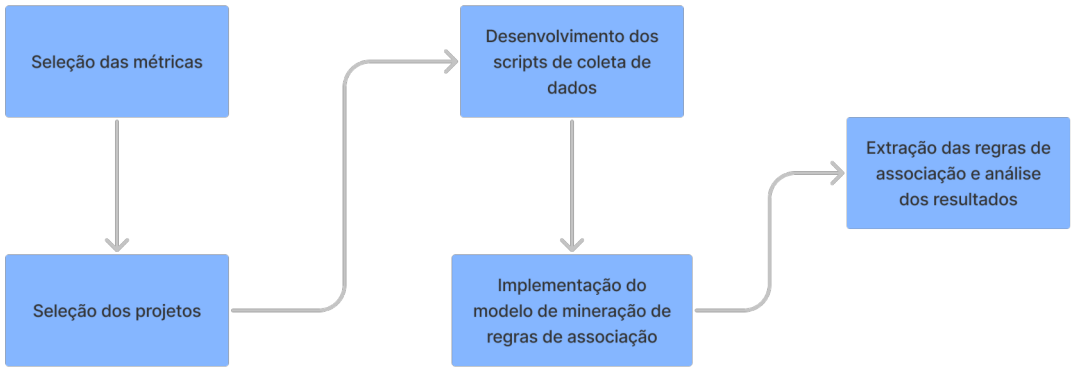
| **Trabalho** | **Utiliza dados de projetos de código fechado** | **Realiza análise descritiva** | **Análisa perfil dos desenvolvedores** | **Analisa aspectos do projeto** |
|---|---|---|---|---|
| Islam e Zibran (2017) | Não | Não | Não | Sim |
| Kerzazi *et al.* (2014) | Sim | Não | Sim | Não |
| Soares *et al.* (2021) | Não | Sim | Sim | Não |
| Este trabalho | Sim | Sim | Sim | Sim |

Fonte: Elaborado pelo autor.

# 4 METODOLOGIA

Este capítulo apresenta as etapas seguidas para alcançar os objetivos deste trabalho. A Figura 2 resume essas etapas e as próximas seções descrevem cada uma delas.

Figura 2 – Procedimentos metodológicos




Fonte: Elaborado pelo autor.

## 4.1 Seleção das métricas

A fim de se obter um conjunto inicial de métricas capazes de representar e avaliar o impacto das características dos desenvolvedores, dos projetos e da complexidade do *build* no tempo de correção de erros de IC, foi realizada uma revisão da literatura existente sobre análise de falhas de *build* de forma *ad-hoc*. Os estudos identificados na revisão são compostos principalmente por trabalhos que empregam modelos estátisticos e técnicas de aprendizado de máquina para tentar predizer o sucesso ou a falha de um *build* com base no histórico de *builds* dos projetos.

O conjunto inicial de métricas extraído foi posteriomente analisado para escolher as métricas com maior potencial de influência no tempo de correção de erros de IC. Essa análise contou com o auxílio de um profissional com experiência em IC e resultou no conjunto de métricas apresentadado no Quadro 4. Todas as métricas definidas tem seu valor extraído por meio da API REST do GitHub e inicialmente são representadas como valores numéricos. Dessa forma, para extrair correlações entre entre essas métricas, na fase de pré-processamento dos dados, foi necessário discretizar seus valores criando intervalos para classificar cada uma delas. Para fazer isso foi utilizada técnica de quantiles através do algoritmo *qCut*[1] disponível na biblioteca de

[1] Disponível em: <https://pbpython.com/pandas-qcut-cut.html> Acesso em: 14 fev. 2022

análise e manipulação de dados Pandas. Tal algoritmo considera a distribuição de frequência dos dados e consegue criar intervalos chamados de *bins* com aproximadamente a mesma quantidade de elementos. Com o intuito de melhorar a interpretação dos padrões extraídos também foi fornecido como entrada para o algotimto o conjunto de rótulos para os intervalos criados. No Quadro 4, junto a descrição das métricas é possível ver também os rótulos e limites dos intervalos usados para discretizar os valores. As métricas das linhas 2 a 7 foram usadas para mensurar propriedades do fator Perfil do Desenvolvedor. Já as métricas das linhas 9 a 11 foram usadas para mensurar propriedades relacionadas ao fator Características do Projeto, enquanto as métricas das linhas 13 a 15 foram usadas para mensurar aspectos do fator Complexidade do *Build*.

Nesse ponto, é importante destacar que cada transação do *dataset* formado neste trabalho representa um *build* defeituoso e é constituída pelo valores das métricas elencadas no Quadro 4. Além disso, é importante lembrar também que o trabalho foi desenvolvido levando em consideração a política interna da empresa de que o desenvolvedor que quebra o *build* é também o desenvolvedor responsável por corrigí-lo.

## 4.2 Seleção dos projetos

Para formar a base de dados necessária para a execução deste trabalho, utilizou-se como fonte de dados um conjunto de 18 projetos fechados de uma empresa de desenvolvimento de software de grande porte, que atua no mercado há mais de 20 anos. A empresa conta atualmente com um quadro de mais de 1000 funcionários espalhados por diferentes países e mantém na sua organização do GitHub mais de 1500 repositórios de software relacionados a projetos públicos e fechados.

A razão para se utilizar projetos fechados se encontra no fato de que a maioria das pesquisas anteriores que exploraram repositórios de software para estudar questões ligadas a IC o fizeram com projetos de código fonte aberto. Dessa forma, há pouco conhecimento empírico sobre esse tema em ambientes industriais. Assim, para selecionar o conjunto de projetos que serviu como fonte de dados, inicialmente procurou-se no perfil da organização no GitHub por projetos que atendessem aos seguintes critérios:

1. O projeto utiliza IC a pelo menos um ano.
2. O projeto utiliza a ferramenta interna de IC da empresa.
3. O projeto possui pelo menos 700 *commits*.

4. O projeto teve ao menos um *commit* em outubro[2] de 2021.

O primeiro critério foi definido com o intuito de tornar os resultados mais consistentes evitando a escolha de projetos com um nível muito inicial de IC e que ativaram poucos *builds* desde o seu início. O segundo critério foi estabelecido para restringir os projetos quanto a família de projetos que a organização parceria permitiu usar no estudo. Já o terceiro critério é necessário para tentar maximizar a quantidade de *builds* extraídos, sobre a ideia de quanto maior a quantidade de commits, maior também a quantidade de *builds* executados. E o quarto e último critério foi usado para filtrar projetos que tiveram atividade recente.

Após essa seleção inicial, uma lista com 30 projetos que atenderam aos critérios estabelecidos foi encaminhada a dois profissionais da organização parceira para que eles pudessem revisar o conteúdo de cada projeto inicialmente selecionado e decidir se o projeto poderia ou não fazer por parte do estudo. Desta forma, por questões de confidencialidade e segurança os profissionais decidiram por remover 12 dos 30 projetos presentes na lista inicial, resultando assim nos 18 projetos que compõem a fonte de dados para o cálculo das métricas descritas na seção anterior. O Quadro 5 apresenta uma visão geral sobre os projetos incluídos no estudo.

## 4.3 Desenvolvimento dos *scripts* para coleta de dados

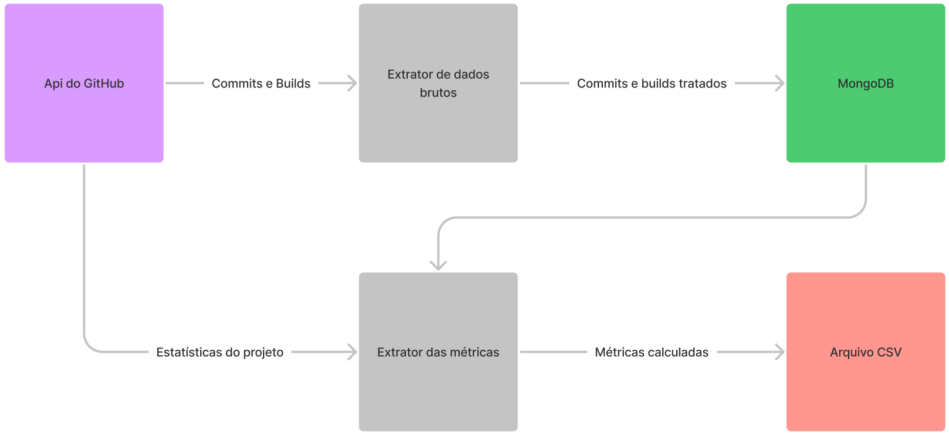
Para automatizar o processo de coleta das métricas definidas na Seção 4.1 foram desenvolvidos dois *scripts* na linguagem de programação Python. O código desses *scripts* está disponível a partir deste link. A Figura 3 ilustra o processo de extração das métricas para cada projeto. O primeiro desses *scripts*, chamado de Extrator de Dados Brutos, tem como principais funcionalidades se comunicar com a API do GitHub para extrair os *commits* e *builds* de um dado repositório, criptografar essas informações e armazená-las em um banco MongoDB para consulta posterior. De maneira mais específica, dado o nome de um repositório como entrada, esse *script* captura primeiro todos os *commits* de um repositório tendo como base para a consulta a *branch master* e em seguida os armazena no MongoDB. Após isso, para cada *commit* capturado uma nova requisição é feita a API de Status do GitHub para verificar se o *commit* disparou ou não um *build* e, caso tenha disparado, as informações sobre o *build* são persistidas no banco.

Um *build* neste trabalho é representado como um *commit* que possui *status* de construção definido, que pode ser *"success"* ou *"failure"*. Assim, o autor do *build* é o autor do *commit* que disparou o *build*, o *status* do build é o *status* do *commit*, a data do *build* é data em

[2] Outubro de 2021 foi o período em que ocorreu a coleta dos dados.

que o *status* foi alterado e os *commits* do *build* são o conjunto de *commits* entre dois *commits* que disparam *build*.

Figura 3 – *Script* de coleta das métricas.



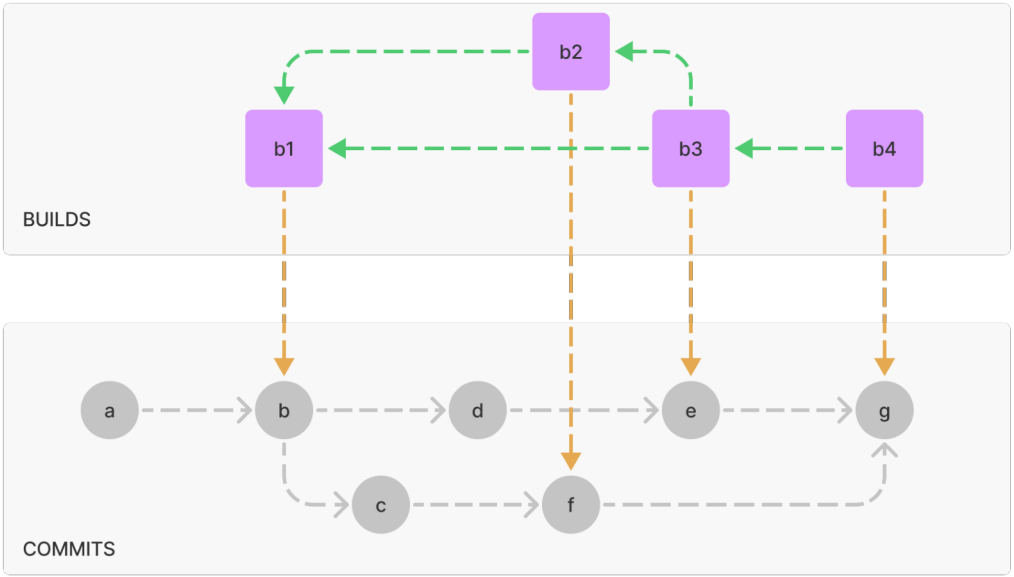

Fonte: Elaborado pelo autor.

Após extrair os *builds* do projeto, o Extrator de dados brutos constrói o histórico de *builds* usando a estratégia definida em Rausch *et al.* (2017) e ilustrada pela Figura 4. Partindo do *build* mais recente e percorrendo os *commits* por meio do campo *patterns* identificou-se quais *commits* compõem cada *build* e qual o antecessor de cada *build* de modo a formar uma linha do tempo linear que permitisse o cálculo do intervalo de correção de *builds* defeituosos.

O segundo *script*, chamado de Extrator de Métricas, consulta os *builds* defeiutosos de um dado projeto a partir do MongoDB e para cada um deles calcula as métricas definidas anteriormente. Como saída desse *script* é gerado um arquivo do tipo *Comma Separated Values* (CSV) com os valores das métricas calculdas para todos os *builds* defeituosos do projeto. Os arquivos gerados para cada projeto foram usados posteriormente como entrada para a criação do *dataframe* no modelo de mineração de regras de associação implementado.

Ao final da etapa de coleta de dados foram extraídos um total de 10128 *builds*, sendo 9715 de sucesso e 413 falhos. Os *builds* falhos compõem o corpo de estudo deste trabalho e sua quantidade reduzida é explicada por dois motivos principais. O primeiro deles é que, diferente do que ocorre em outros trabalhos, como os de Islam e Zibran (2017) e Kerzazi *et al.* (2014), considerou-se como falhos somente os *builds* que tiveram *status* igual a *"failure"* em vez de considerar como falhos todos os *builds* que tem *status* diferente de *"success"*, o que inclui

Figura 4 – Mapeamento entre *commits* e *builds*




Fonte: Adaptado de Rausch *et al.* (2017)

além dos marcados como "failure" aqueles marcados como *"error"* ou *"pending"*, por exemplo. Além disso, historicamente ocorre um desbalanceamento entre a quantidade de *builds* falhos e de sucesso em trabalhos que estudam *builds* defeituosos.

Dito isto, é importante destacar que a quantidade reduzida de *builds* falhos encontrada não invalida os resultados do trabalho, uma vez que a mineração de regras de associação é uma tarefa de natureza descritiva e que objetiva revelar padrões inicialmente desconhecidos em uma base de dados, não sendo necessário, portanto, uma uma vasta base de dados para encontrar resultados úteis.

## 4.4 Implementação do modelo de mineração de regras de associação

A fim de encontrar correlações entre os fatores estudados e a duração do intervalo de correção de erros de IC, foi implementado um modelo de mineração de regras de associação no Google Colaboratory[3], uma ferramenta mantida pelo Google que permite a execução de código Python na nuvem de maneira simples e prática através do navegador e que, por conta disso, é bastante usada para ciência de dados. O modelo criado e o *dataset* originado pela junção dos arquivos de métrica de cada projeto estão disponíveis neste link.

Em um primeiro momento, ocorre no modelo a etapa de pré-processamento dos dados, onde são realizadas a importação dos arquivos CSV gerados, o tratamento de valores

[3] Disponível em: <https://colab.research.google.com/notebooks/welcome.ipynb?hl=pt-BR> Acesso em: 23 jul. 2021

ausentes e a discretização dos valores numéricos em categóricos. A partir de uma análise manual do *dataframe* formado, percebeu-se a existência de alguns valores ausentes para as métricas "dev_avg_commits_size" e "build_correction_interval". Desse modo, para não perder as informações de nenhuma das linhas do *dataframe* foi aplicada uma técnica de tratamento sugerida em Han *et al.* (2011) que substitui os valores ausentes encontrados pelo valor da média da métrica.

Para extrair as regras de associação foram utilizados os algoritmos *Apriori* e *Association Rules*, disponíveis na biblioteca mlxtend[4]. O primeiro deles é responsável por encontrar *frequent itemsets* no *dataframe* discretizado enquanto o segundo é encarregado de extrair as regras de associação a partir dos *frequent itemsets* gerados.

Como último passo para a conclusão do modelo de mineração das regras de associação, foram definidos os valores mínimos para as medidas de Suporte e Confiança das métricas a serem extraídas, com o intuito de garantir que as regras geradas não acontecessem de forma aleatória. Desta maneira, escolheu-se um valor mínimo de 10% para o Suporte, que representa um total de 41 instâncias de *builds* falhos, e de 30% para a Confiança.

## 4.5 Extração das regras de associação e análise dos resultados

A última etapa realizada para a concretização deste trabalho consistiu na execução do modelo desenvolvido na etapa anterior e análise das regras mineradas. Depois de executar o modelo criado, o conjunto de regras gerado como saída foi filtrado para que contivesse somente as regras que tem como consequente a métrica *build_correction_interval*. Para faciltar o trabalho de análise o conjunto regras foi novamente filtrado para que contivesse agora apenas as regras que tem até 3 antecedentes. Concluída a aplicação dos filtros, cada uma das regras resultantes foi analisada a partir de seus valores de *Lift*.

[4] Disponível em: <http://rasbt.github.io/mlxtend/> Acesso em: 13 jan. 2022

Quadro 4 – Métricas selecionadas para a realização do estudo.

| Métrica | Descrição | Referência |
|---|---|---|
| **Perfil do Desenvolvedor** | | |
| dev_commit_freq | Quantidade média de commits enviados pelo dev por semana no repositório do projeto. Onde: *"few commits per week"* = até 4 commits; *"average number of commits per week"* = entre 4 e 7 commits; *"many commits per week"* = mais de 7 commits. | Rausch *et al.* (2017) |
| dev_avg_size_commits | Tamanho médio em linhas de código dos commits enviados pelo dev para o repositório do projeto. Onde: *"short commits"* = até 48 mudanças;*"mid commits"* = entre 48 e 88 mudanças; *"long commits"* = mais de 88 mudanças. | Nova métrica |
| dev_commit_rate | Porcentagem que os commits enviados pelo dev para o repositório do projeto representa em todo o repositório. Onde: *"low rate of commits"* = Até 6%; *"mean rate of commits"* = Entre 6 e 17%; *"high rate of commits"* = Mais de 17%. | AlOmar *et al.* (2020) |
| dev_project_time | Diferença em dias entre o commit atual e o primeiro commit do dev no repositório. Onde: *"short time in project"* = até 54 dias; *"mid time in project"* = entre 54 e 174 dias, *"long time in project"* = mais do que 174 dias. | Rausch *et al.* (2017) |
| dev_fail_history | Taxa de falha de compilações pelo dev atual no passado do projeto. Onde: *"few failures"* = até 5% de falha; "mean failures"= entre 5 e 16% de falha, "many failures"= mais de 16% de falha. | Saidani *et al.* (2020) |
| dev_exp | Número de compilações que o dev executou no passado do projeto. Onde *"few builds"* = até 24 builds; "average number of builds"= entre 24 e 65 builds, *"many builds"* = mais de 65 builds. | Saidani *et al.* (2020) |
| **Características do Projeto** | | |
| project_size | Tamanho do projeto em linhas de código até o build atual. Onde: *"small"* = até 3840 linhas de código; *"middle"* = entre 3840 e 8273 linhas de código; *"great"* = mais de 8273 linhas de código. | Islam e Zibran (2017) |
| project_age | Diferença em dias entre o primeiro commit do projeto e o commit que disparou o build atual. Onde: *"new"* = até 522 dias; *"intermediary"* = entre 522 e 923 dias; *"old"* = mais de 923 dias. | Nova métrica |
| project_team_size | Quantidade de contribuidores no projeto nos últimos 60 dias. Onde: *"small team"* = até 4 pessoas; *"big team"* = entre 4 e 12 pessoas. | Kerzazi *et al.* (2014), Islam e Zibran (2017) |
| **Complexidade do *Build*** | | |
| build_qty_commits | Quantidade de commits no build atual. Onde: *"few commits"* = até 18 *commits*, *"many commits"* = mais de 18 commits. | Rausch *et al.* (2017) |
| build_mod_files | Quantidade de arquivos modificados no build atual. Onde: *"few files"* = até 2 arquivos; *"average number of files"* = Entre 2 e 7 arquivos; *"many files"* = mais de 7 arquivos. | Rausch *et al.* (2017) |
| build_mod_lines | Quantidade de linhas modificadas no build atual. Onde: *"few files"* = até 11 linhas; *"average number of lines"* = Entre 11 e 121 linhas; *"many files"* = mais de 121 linhas. | Rausch *et al.* (2017) |
| build_correction_interval | Tempo em minutos entre um *build* falho e o próximo com sucesso. Onde: *"small interval of correction"* = até 32 minutos; *"mid interval of correction"* = Entre 32 e 220 minutos; *"long interval of correction"* = Entre 220 e 1430 minutos. | Kerzazi *et al.* (2014) |

Fonte: Elaborado pelo autor

Quadro 5 – Visão geral dos projetos incluídos no estudo.

| **Nome do Projeto** | ***Commits*** | **Linhas de código** | **Idade** |
|---|---|---|---|
| Projeto 1 | 3397 | 15630 | 3 anos |
| Projeto 2 | 3337 | 15796 | 5 anos |
| Projeto 3 | 2262 | 11251 | 5 anos |
| Projeto 4 | 1989 | 24921 | 3 anos |
| Projeto 5 | 1572 | 9606 | 2 anos |
| Projeto 6 | 1378 | 11557 | 2 anos |
| Projeto 7 | 1224 | 12991 | 3 anos |
| Projeto 8 | 1105 | 18410 | 4 anos |
| Projeto 9 | 1088 | 18892 | 3 anos |
| Projeto 10 | 1054 | 5484 | 3 anos |
| Projeto 11 | 1045 | 5174 | 2 anos |
| Projeto 12 | 1947 | 6777 | 3 anos |
| Projeto 13 | 890 | 1946 | 4 anos |
| Projeto 14 | 825 | 6923 | 4 anos |
| Projeto 15 | 783 | 11324 | 3 anos |
| Projeto 16 | 777 | 5913 | 4 anos |
| Projeto 17 | 758 | 10137 | 2 anos |
| Projeto 18 | 727 | 15174 | 3 anos |

Fonte: Elaborado pelo autor

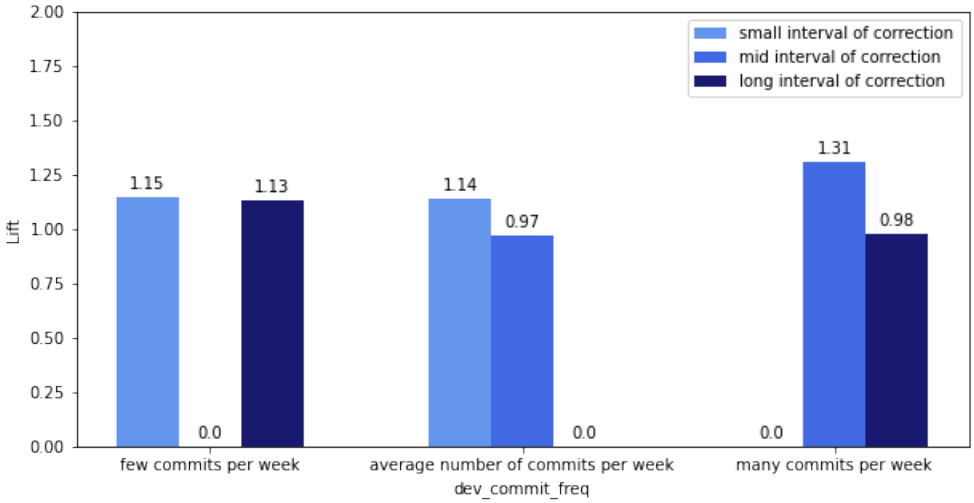
# 5 RESULTADOS

Este capítulo apresenta e discute os resultados obtidos para este trabalho. A Seção 5.1 trata da correlação entre o perfil do desenvolvedor e o tempo de correção de erros de IC. Na Seção 5.2 é mostrado como as caracteristicas do projeto afetam o tempo de correção de erros de IC enquanto a Seção 5.3 aborda a influência da complexidade do *build* no tempo de correção de erros IC. A Seção 5.4 discute os principais achados e suas implicações.

## 5.1 Impacto do perfil do desenvolvedor

Para estudar a relação do perfil do desenvolvedor com a duração do intervalo de correção de erros de IC, foram analisados atributos obtidos a partir de sua atividade no repositório do projeto: (i) a quantidade média de *commits* enviados por semana, (ii) o tamanho médio em linhas de código de seus *commits*, (iii) a taxa de *commits* no repositório, (iv) o tempo em dias do desenvolvedor no projeto, (v) a experiência do desenvolvedor expressa como a quantidade de *builds* que ele executou no projeto e (vi) a sua taxa de falhas de *build*.

Figura 5 – Relação da frequência de *commits* com o tempo de correção de erros de IC




Fonte: Elaborado pelo autor.

As correlações extraídas sugerem que a quantidade média de *commits* enviados por semana afeta de maneira moderada o tempo de correção de erros de IC quando o desenvolvedor envia muitos *commits* por semana. Esse comportamento pode ser visto na Figura 5, que retrata os valores de *Lift* para as regras do tipo $dev\_commit\_freq \rightarrow build\_correction\_interval$. É possível notar na figura que quando o autor do *build* defeituoso é um desenvolvedor que envia muitos

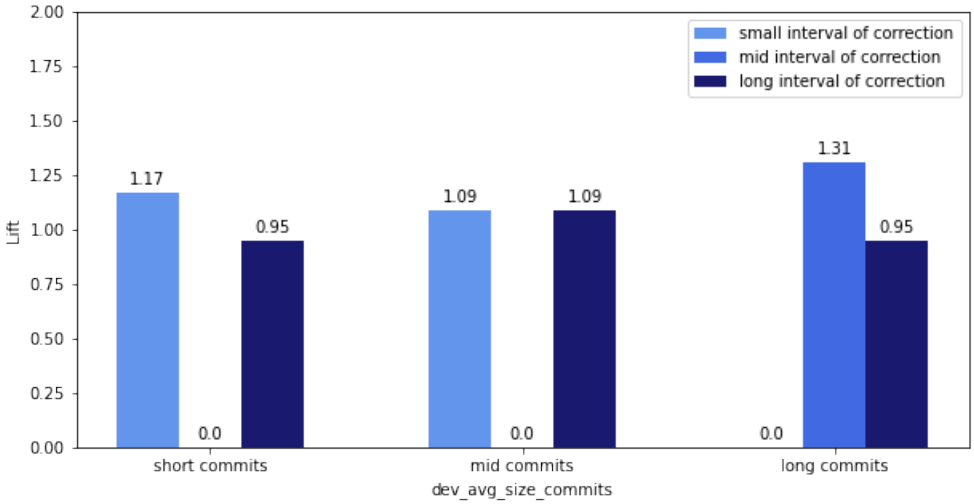
*commits* por semana, as chances de se obter um intervalo de correção médio aumentam em 31% (*Lift* = 1.31). Isso propõe, a princípio, que:

> **Achado 1:** *Desenvolvedores que enviam muitos commits por semana conseguem corrigir erros de IC em intervalo razoável de tempo.*

Assim como a quantidade de *commits* enviados por semana, o tamanho médio dos *commits* enviados pelo desenvolvedor influencia moderadamente a duração do intervalo de correção de *builds* defeituosos. A Figura 6 exibe o *Lift* para as regras onde *dev_avg_commits_size* é o antecedente e *build_correction_interval* é o consequente. É possível verificar que quando *dev_avg_commits_size* = *long commits* as chances de um intervalo de correção médio aumentam em 31% (*Lift* = 1.31), ao passo que quando *dev_avg_commits_size* = *short commits* as chances de um pequeno intervalo de correção aumentam em 17% (*Lift* = 1.17). Isso sugere que:

> **Achado 2:** *Manter os commits menores pode contribuir para um menor tempo de correção de erros de IC.*

Figura 6 – Relação do tamanho médio de *commits* do dev com o tempo de correção de erros de IC



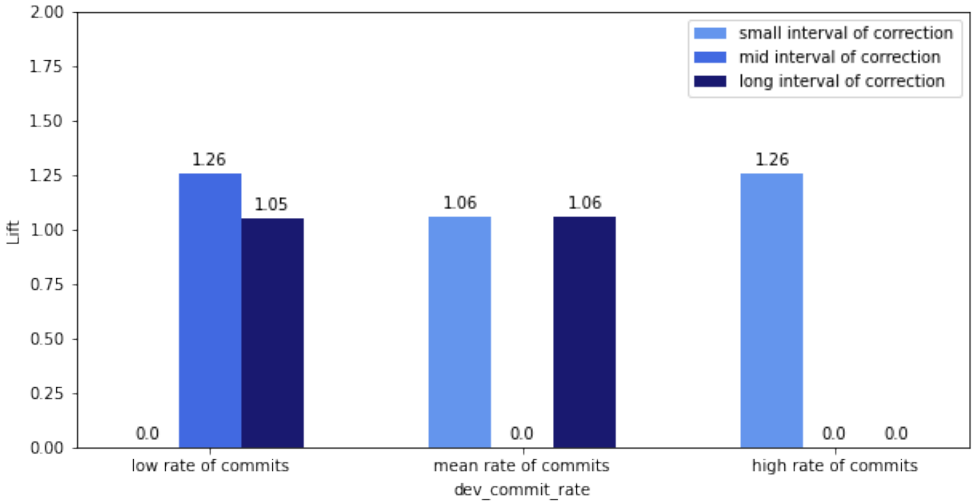

Fonte: Elaborado pelo autor.

Verificou-se também, que existe influência moderada da taxa de *commits* na duração do intervalo de correção de *builds* defeituosos. Na Figura 7 são apresentados os valores de *Lift* para as regras em que *dev_commit_rate* é o antecedente e *build_correction_interval* é o consequente. Nessa figura, consegue-se perceber que quando o autor do *build* quebrado é um

desenvolvedor com uma alta taxa de *commits* as chances do *build* ser corrigido em um menor intervalo de tempo aumentam em 26% (*Lift* = 1.26). Ao mesmo tempo em que quando o autor do *build* quebrado é um desenvolvedor com uma baixa taxa de *commits*, as chances do *build* ser corrigido em um intervalo de tempo médio aumentam novamente em 26%. Esse fato, indica de maneira inicial que:

> **Achado 3:** *Desenvolvedores que contribuem mais para o projeto geram builds falhos que tendem a serem corrigidos mais rapido.*

Figura 7 – Relação da taxa de *commits* do dev com o tempo de correção de erros de IC

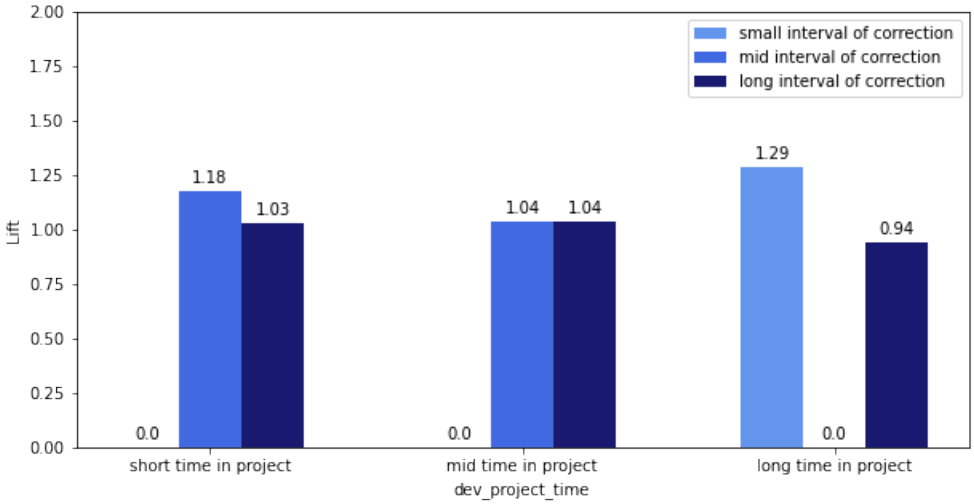



Fonte: Elaborado pelo autor.

Semelhante a taxa de *commits*, o tempo em dias do desenvolvedor no projeto também mantém relação moderada com a extensão do intervalo de correção de erros de IC. A Figura 8 exibe as regras do tipo *dev_project_time* $\rightarrow$ *build_correction_interval*. Observa-se que quando *dev_project_time* = *long time in project* as chances de se obter um pequeno intervalo de correção aumentam em quase 30%, ao passo que quando *dev_project_time* = *short time in project* as chances de se ter um médio intervalo de correção aumentam em 18%. Isso indica inicialmente que:

> **Achado 4:** *Desenvolvedores que estão a mais tempo em contato com o projeto tem mais facilidade para corrigir builds falhos.*

Figura 8 – Relação do tempo de projeto do dev com o tempo de correção de erros de IC

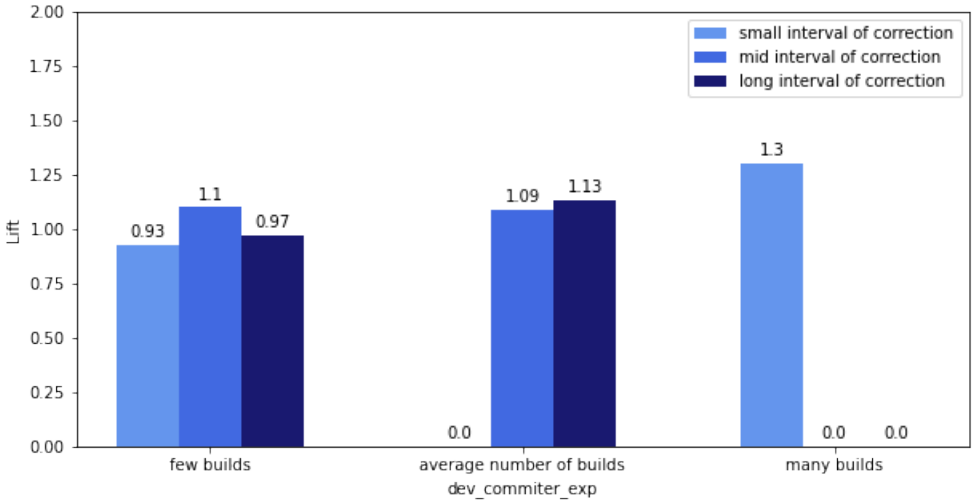



Fonte: Elaborado pelo autor.

As correlações mineradas, mostraram também que existe relação entre a quantidade de *builds* executados pelo dev no passado e o tempo de correção de erros de IC. A Figura 9 expressa as regras que englobam essa relação. A partir dessa figura é possível inferir preliminarmente que:

**Achado 5:** *O fato do autor do build defeituoso ter disparado muitos builds no passado do projeto contribui de maneira moderada para um menor intervalo de correção.*

Figura 9 – Relação da experiência do desenvolvedor com o tempo de correção de erros de IC

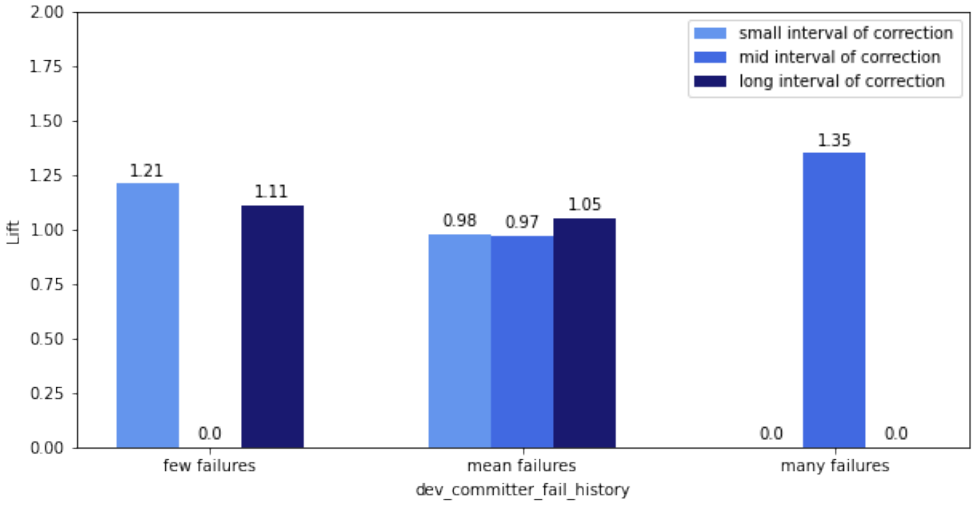



Fonte: Elaborado pelo autor.

Constatou-se ainda, que não existe uma relação clara entre a taxa de falhas de *build*

do desenvolvedor e o tempo de correção de erros de IC. Isso ocorre porque tanto um pequeno percentual quanto um grande percentual de falhas colaboraram para uma redução do intervalo de correção. A Figura 10 mostra as regras que englobam a taxa de falhas e o tempo de correção. É possível ver que quando o desenvolvedor ativa poucas falhas ocorre aumento de 21% nas chances de se obter um pequeno intervalo de correção e quando o desenvolvedor ativa muitas falhas há um aumento de 35% nas chances de se ter um intervalo de correção médio.

Figura 10 – Relação da taxa de falhas do dev com o tempo de correção de erros de IC




Fonte: Elaborado pelo autor.

## 5.2 Impacto das características do projeto

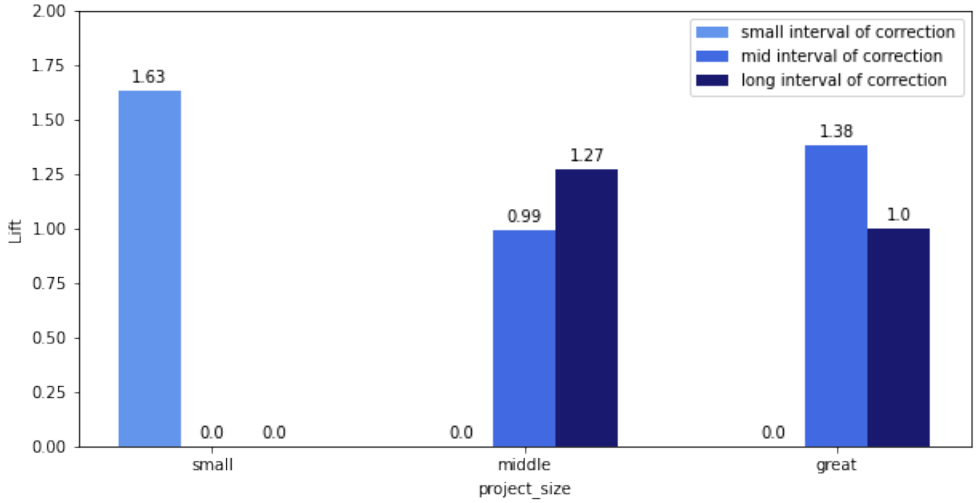
Um projeto de *software* pode ser caracterizado de acordo com uma série de propriedades, como por exemplo a quantidade de linhas de código, o tamanho da equipe e sua idade em dias. Neste trabalho, analisou-se se essas três características mantém alguma relação com o tempo de correção de erros de IC.

Os resultados obtidos indicam que o tamanho do projeto influencia fortemente o tempo de correção de erros de IC. A Figura 11 mostra os valores de *Lift* para as regras em que *project_size* é o antecedente e *build_correction_interval* é o consequente. É possível perceber que:

**Achado 6:** *Quando o projeto tem um tamanho pequeno as chances do intervalo de correção de builds defeituosos ser menor aumentam significativamente.*

Esse comportamento fica claro quando analisa-se a regra *project_size* = *small* → *build_corretion_interval* = *small interval of correction*. Para esse caso, quando o *build* defeituoso ocorre quando projeto está pequeno, as chances de se ter um pequeno intervalo de correção aumentam em 63% (*Lift* = 1.63).

Figura 11 – Relação do tamanho do projeto com o tempo de correção de erros de IC




Fonte: Elaborado pelo autor.

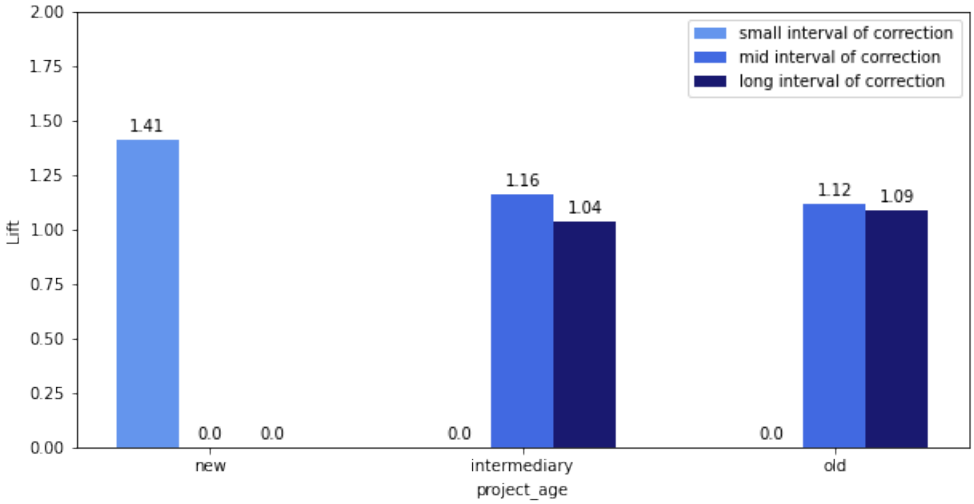
Da mesma forma que o tamanho do projeto, observou-se que a idade do projeto também mantém relação com o tempo de correção de erros de IC. A Figura 12 exibe o valor de *Lift* para as regras do tipo *project_age* → *build_correction_interval*. Percebe-se que:

> **Achado 7:** *Builds defeituosos que ocorrem quando o projeto tem pouca idade, tem mais chance de ter um pequeno intervalo de correção.*

Esse comportamento pode ser visto através da regra *project_age* = *new* → *build_correction_interval* = *small interval of correction*. Neste caso, *builds* que quebram quando o projeto ainda tem pouca idade tem 41% (*Lift* = 1.41) mais chance de terem um menor intervalo de correção.

Além disso, os resultados sugerem que existe correlação entre o tamanho da equipe do projeto e a duração do intervalo de correção de *builds* quebrados. De acordo com a Figura 13, *builds* que quebram quando o projeto tem uma equipe grande tem 23% (*Lift* = 1.23) mais chances de serem resolvidos em menos tempo. Entretanto, não é possível afirmar que o tamanho da equipe por si só contribui para a redução do tempo de correção de *builds* falhos, uma vez que

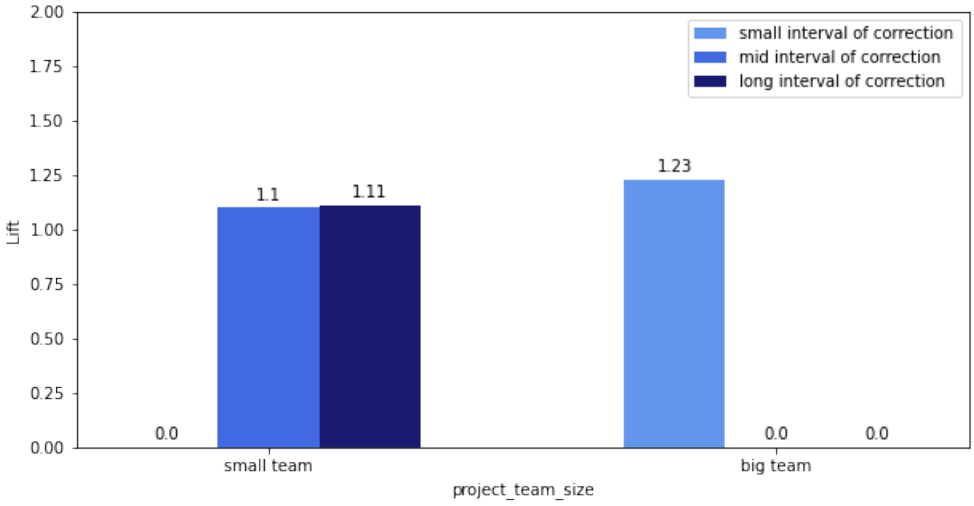
Figura 12 – Relação da idade do projeto com o tempo de correção de erros de IC




Fonte: Elaborado pelo autor.

a confiança da regra *project_team_size = big team → build_correction_interval = small interval off correction* é de 41%, enquanto a confiança da regra contrária, *build_correction_interval = small interval of correction → project_team_size = big team* é de 60%.

Figura 13 – Relação do tamanho do time com o tempo de correção de erros de IC




Fonte: Elaborado pelo autor.

Os resultados sugerem ainda que a conjunção de certas características do projeto contribui para aumentar as chances de *builds* falhos terem intervalos de correção menores e médios. O Quadro 6 apresenta regras que combinam diferentes características do projeto em seu antecendente e o intervalo de correção em seu consequente. A regra 1, destaca que os *builds* falhos que ocorrem quando o projeto ainda está em fases iniciais tem 71% (*Lift* = 1.71) mais

chances de terem um pequeno intervalo de correção. A regra 3, por sua vez, ressalta que um time maior com um projeto pequeno pequeno aumentam em 53% (*Lift* = 1.53) a chance de *builds* defeituosos demorarem menos pra serem corrigidos. Já as regras 2 e 4, sugerem que mantendo o tamanho da equipe do projeto pequena e aumentando o tamanho do projeto aumentam-se as chances de obter um intervalo de correção médio.

Quadro 6 – Regras formadas pela conjunção de atributos de projeto.

| # | Antecedente | Consequente | *Lift* | Confiança | Suporte |
|---|---|---|---|---|---|
| 1 | *project_age = new* ∧ *project_size = small* | *build_correction_interval = small interval of correction* | 1.71 | 0.57 | 0.12 |
| 2 | *project_team_size = small team* ∧ *project_size = great* | *build_correction_interval = mid interval of correction* | 1.61 | 0.53 | 0.12 |
| 3 | *project_team_size = big team* ∧ *project_size = great* | *build_correction_interval = small interval of correction* | 1.53 | 0.51 | 0.1 |
| 4 | *project_team_size = small team* ∧ *project_size = middle* | *build_correction_interval = mid interval of correction* | 1.28 | 0.42 | 0.12 |

Fonte: Elaborado pelo autor

## 5.3 Impacto da complexidade do *build*

Para analisar se a complexidade do *build* afeta a duração do intervalo de correção de *builds* falhos, verificou-se de que maneira a quantidade de *commits*, de arquivos e de linhas modificadas no *build* impactam seu intervalo de correção.

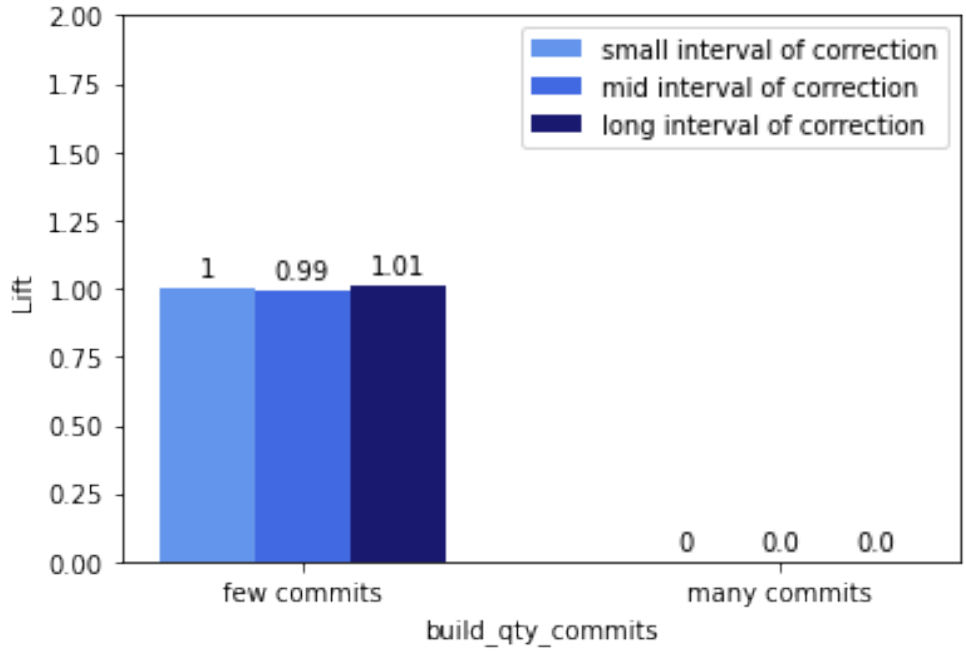
Os resultados extraídos mostraram que:

> **Achado 8:** *Não existe, no contexto adotado neste trabalho, uma relação efetiva entre a quantidade de commits envolvidos em builds falhos e seu tempo de correção.*

A Figura 14 mostra os valores de *Lift* para as regras do tipo *build_qty_commits* → *build_correction_interval*. Como é possível ver na figura todas as regras desse tipo tem valores *Lift* muito próximos a 1 indicando que ocorre independência entre o antecedente e consequente das regras e que, portanto, não existe uma correlação forte entre a quantidade de *commits* e o tempo de correção de erros de IC.

A quantidade de arquivos no *build*, por sua vez, interfere de maneira tímida na duração do intervalo de correção de erros de IC. A Figura 15 exibe os valores de *Lift* para as regras em que a quantidade de arquivos é o antecedente e a duração do intervalo de correção é o consequente. É possível notar que:

Figura 14 – Relação quantidade de *commits* no build com o tempo de correção de erros de IC




Fonte: Elaborado pelo autor.

**Achado 9:** *Uma menor quantidade de arquivos contribui de maneira suave para um menor intervalo de correção.*

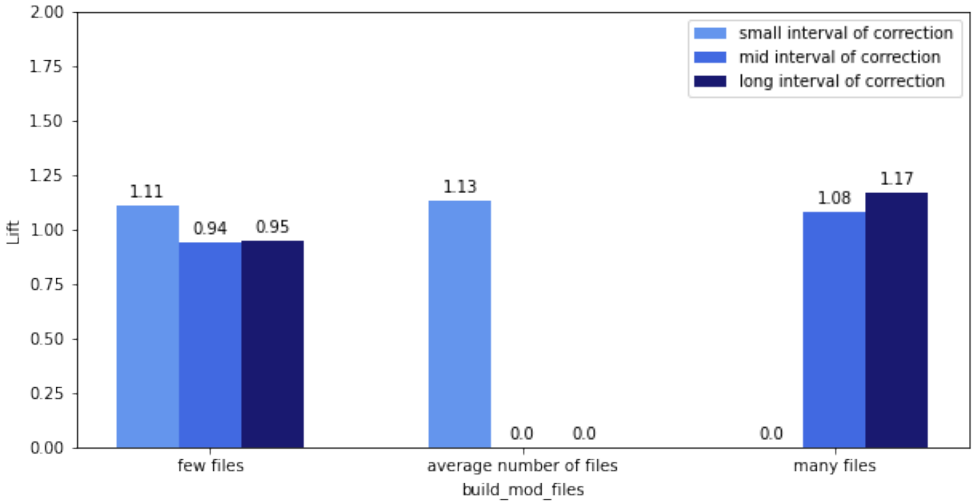
Quando *build_mod_files = few files* as chances de se obter um intervalo de correção pequeno aumentam em 11% (*Lift* = 1.11), ao passo que a chances de se obter um grande intervalo de correção diminuem em 5% (*Lift* = 1.5). Além disso, a ideia de que uma menor quantidade de arquivos alterados contribui de modo leve para um menor tempo de correção é reforçada pela regra *build_mod_files = many files* $\rightarrow$ *build_correction_interval = long interval correction*, que indica que *builds* falhos que contam com uma maior quantidade de arquivos alterados tem 17% (*Lift* = 1.17) mais chances de possuírem um grande intervalo de correção.

De modo semelhante a quantidade de arquivos, os resultados mostram que:

**Achado 10:** *A quantidade de linhas alteradas impacta levemente no tempo de correção de erros de IC, com builds que envolvem menos linhas tendo mais chances de possuírem um pequeno intervalo de correção.*

A Figura 16 mostra o *Lift* para as regras que envolvem a quantidade de linhas alteradas e a duração do intervalo de correção. Nota-se que quando o *build* falho envolve poucas

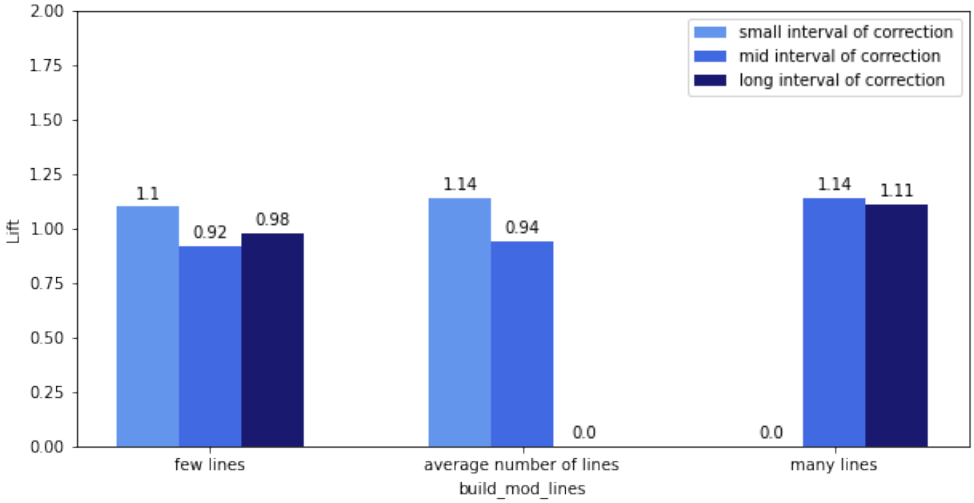
Figura 15 – Relação da quantidade de arquivos modificados no *build* com o tempo de correção de erros de IC




Fonte: Elaborado pelo autor.

linhas as chances de um intervalo de correção pequeno aumentam em 10% (*Lift* = 1.1). Similar a isso, uma quantidade média de linhas alteradas contribui também para um pequeno intervalo de correção ao mesmo tempo que muitas linhas alteradas aumentam as chances de um intervalo de correção médio.

Figura 16 – Relação da quantidade de linhas modificadas no *build* com o tempo de correção de erros de IC




Fonte: Elaborado pelo autor.

Analisou-se também como a conjunção de características do *build* influenciam o tempo de correção de erros de IC. O Quadro 7 apresenta as principais regras encontradas formadas pela junção dos valores das métricas de *build*. De maneira geral, as regras no Quadro 7 sugerem que *builds* com um menor conjunto de mudanças colaboram para um menor intervalo

de correção. Enquanto, *builds* com um maior conjunto de mudanças aumentam as chances de um intervalo de correção longo.

Quadro 7 – Regras formadas pela conjunção de atributos de *build*.

| # | Antecedente | Consequente | *Lift* | Confiança | Suporte |
|---|---|---|---|---|---|
| 1 | *build_mod_lines = many lines* ∧ *build_qty_commits = few commits* ∧ *build_mod_files = many files* | *build_correction_interval = long interval of correction* | 1.22 | 0.41 | 0.1 |
| 2 | *build_qty_commits = few commits* ∧ *build_mod_files = many files* | *build_correction_interval = long interval of correction* | 1.21 | 0.4 | 0.12 |
| 3 | *build_mod_lines = many lines* ∧ *build_qty_commits = few commits* | *build_correction_interval = long interval of correction* | 1.15 | 0.38 | 0.12 |
| 4 | *build_mod_lines = average number of lines* ∧ *build_qty_commits = few commits* | *build_correction_interval = small interval of correction* | 1.14 | 0.38 | 0.12 |
| 5 | *build_qty_commits = few commits* ∧ *build_mod_files = average number of files* | *build_correction_interval = small interval of correction* | 1.13 | 0.38 | 0.11 |
| 6 | *build_mod_lines = few lines* ∧ *build_qty_commits = few commits* ∧ *build_mod_files = few files* | *build_correction_interval = small interval of correction* | 1.11 | 0.37 | 0.11 |
| 7 | *build_mod_lines = many lines* ∧ *build_qty_commits = few commits* | *build_correction_interval = mid interval of correction* | 1.11 | 0.37 | 0.12 |
| 9 | *build_qty_commits = few commits* ∧ *build_mod_files = few files* | *build_correction_interval = small interval of correction* | 1.11 | 0.37 | 0.14 |
| 8 | *build_mod_lines = few lines* ∧ *build_qty_commits = few commits* | *build_correction_interval = small interval of correction* | 1.1 | 0.37 | 0.13 |

Fonte: Elaborado pelo autor

## 5.4 Discussão

Diante do que foi apresentado na Seção 5.1, que traz os resultados obtidos para a relação do perfil do desenvolvedor com a duração do intervalo de correção de erros IC, concluiu-se que:

> **Resultado 1:** *o perfil do desenvolvedor influencia o tempo de correção de erros de IC através, principalmente, dos atributos taxa de commits(dev_commit_rate), tempo de projeto (dev_project_time) e número de builds executados (dev_commiter_exp).*

Os resultados apontam que *builds* falhos disparados por desenvolvedores que possuem muito tempo de projeto, alta taxa de *commits* e muitos *builds* executados tem mais chance de serem resolvidos dentro de um pequeno intervalo de tempo. Essas características remontam a um desenvolvedor com maior experiência e que já vivenciou muita coisa no projeto. Talvez

por isso, esse perfil de desenvolvedor consiga resolver os erros que surgem durante a integração de suas mudanças de maneira mais rápida. Dessa forma, essa relação reforça a importância dos desenvolvedores menos experientes, que estão a pouco tempo no projeto e detém pouco entendimento do mesmo, procurarem os desenvolvedores mais experientes quando se deparam com erros ligados a IC que não conseguem resolver de maneira imediata. Além disso, uma outra prática que pode ser adotada para aproveitar todo o conhecimento adquirido pelos desenvolvedores mais experientes em relação aos erros de IC é deixar registrado em algum local, como por exemplo, numa seção do README do repositório do sistema, como lidar com os problemas que ocorrem com mais frequência no projeto.

A partir do que foi exposto na Seção 5.2, que aborda os resultados para relação do tempo de correção de erros de IC com as características do projeto, foi possível concluir que:

**Resultado 2:** *As características do projeto afetam o tempo de correção de erros de IC de maneira forte, sobretudo, por meio dos atributos de tamanho e idade do projeto.*

As regras encontradas para essa relação indicam que quando o projeto está em suas fases iniciais, com valores baixos de idade e tamanho, as chances de *builds* falhos terem um pequeno intervalo de correção aumentam significativamente. Uma das possíveis razões para esse comportamento reside no fato de que nesse ponto da vida do projeto, a sua equipe de desenvolvimento ainda é composta em grande parte por profisionais que deram início ao construção projeto e que, portanto, detém maior entendimento dele como um todo.Essa relação chama ainda a atenção para a necessidade de se adotar ferramentas que auxiliem o processo de correção de erros de IC conforme o projeto vai se tornando grande, como por exemplo, ferramentas que sumarizem a extensa saída de *log* dos servidores de IC e que permitam um rápido entendimento do problema.

Por fim, a partir das informações exibidas na Seção 5.3, que expõe as regras mineradas para a avaliar a relação da complexidade do *build* com o tempo de correção de erros de IC, foi possível conlcluir que:

**Resultado 3:** *A complexidade do build influencia o tempo de correção de erros de IC através dos atributos número de linhas e número de arquivos modificados, com builds falhos mais complexos tendo maiores chances de possuírem um intervalo de correção mais longo.*

Essa influência ocorre de maneira leve quando se considera os atributos de complexidade isoladamente e de maneira moderada quando se considera a conjunção desses atributos. Essa relação de influência ressalta a importância dos desenvolvedores adotarem a boa prática de integrar o seu código de maneira frequente, pois isso contribui para que o conjunto de mudanças seja menos complexo e consequentemente para que se gaste menos tempo identificando qual mudança ocosionou o erro. Uma ação que pode ser adotada para se alcançar *builds* menos complexos é incorporar uma regra de *lint* aos *scripts* de *pre-commit* do projeto para que um *warning* seja exibido sempre que um *commit* ultrapassar um determinado limite de linhas e de arquivos.

## 5.5 Ameaças à validade

De acordo com a definição Wohlin *et al.* (2012), as principais ameaças à validade deste estudo residem na correção dos *scripts* de coleta de dados(Validade Interna) e na capacidade de generalização dos resultados encontrados (Validade de Extensão). Para mitigar a ocorrência de erros em relação ao cálculo de cada uma das métricas inseridas no trabalho, os dois *scripts* desenvolvidos para calcular seus valores foram revisados por um profissional de maior experiência em desenvovimento de software e que detém bom conhecimento de IC.

Em função do tamanho reduzido do *dataset* formado, a generalização dos resultados obtidos neste estudo se restringe aos projetos da organização parceira que possuem características semelhantes aos que foram selecionados para este trabalho, não sendo possível estender os resultados descobertos para projetos de código aberto ou projetos fechados de outras organizações. Nesse contexto, é importante destacar que foram considerados falhos somente os *builds* que tiveram *status* de construção igual a *failure*, ao invés de considerar como falhos todos os *builds* que tem *status* de construçao diferente de *success*, como ocorre em outras trabalhos. Os achados deste trabalho, no entanto, indicam que correlações semelhantes podem estar presentes em outros projetos e pretende-se realizar novos estudos para investigar isso.

Outros dois pontos ligados a Validade de Construção desse trabalho são o tratamento de valores ausentes aplicado e o próprio conjunto de métricas selecionado. Para não perder nenhuma linha do dataset formado, optou-se por substituir os valores ausentes encontrados para as métricas *dev_avg_commits_size* e *build_correction_interval* pela média de seus valores. Desta forma, isso pode ter contribuido, mesmo que minimamente, para a ocorrência de alguma regra. Já o conjunto de métricas selecionado representa uma ameaça a Validade de Construção em

virtude, principalmente, da quantidade reduzida de métricas por fator. No final deste trabalho notou-se, por exemplo, a necessidade de incluir mais métricas para mensurar a experiência do desenvolvedor.

# 6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho realizou um estudo exploratório com 18 projetos de código fechado de uma empresa de grande porte de desenvolvimento de software com o objetivo de analisar a influência do perfil do desenvolvedor, das características do projeto e da complexidade do *build* na duração do intervalo de correção de erros de IC.

Para tanto, inicialmente foi feita uma coleta a partir da literatura existente sobre análise de falhas de *build*, um conjunto de métricas capazes de mensurar aspectos do perfil do desenvolvedor, das caracteristicas dos projetos e da complexidade dos *builds*.

Em seguida, dois *scripts* Python foram desenvolvidos para auxiliar no cálculo dessas métricas. O primeiro deles, extrai os *commits* e *builds* para cada projeto fornecido pela organização parceira. Enquanto o segundo, calcula o valor das métricas para cada *build* falho encontrado num projeto.

Por fim, para extrair e avaliar correlações significativas entre os três fatores estudados e o tempo de correção de erros de IC, a partir das métricas extraídas, foi empregada uma técnica de mineraração de dados chamada de mineração de regras associação.

## 6.1 Resultados encontrados

Após concluir a execução dos procedimentos metodológicos pensados para este trabalho, as correlações extraídas mostraram que o perfil do desenvolvedor influencia moderamente o tempo de correção de erros de IC, com *builds* ativados por desenvolvedores mais experientes tendo mais chances de serem corrigidos em menores intervalos de tempo. Além disso, descobriu-se também que as características do projeto afetam fortemente a duração do intervalo de correção de erros de IC, com *builds* disparados quando o projeto está em suas fases inciais (com pouca idade e pequeno tamanho) estando correlacionados a pequenos intervalos de correção. Encontrou-se ainda que a complexidade do *build* afeta de maneira leve a duração do intervalo de correção de erros de IC quando se considera os atributos de complexidade de maneira individual e que a complexidade influência o tempo de correção de maneira moderada quando se considera a conjunção de seus atributos. Desta forma, observou-se que *builds* falhos mais complexos (com mais linha e arquivos modificados) tem maiores chances de possúirem um longo intervalo de correção.

## 6.2 Trabalhos Futuros

Como trabalhos futuros pretende-se inicialmente adicionar novas métricas para mensurar propriedades complementares dos três fatores analisados neste trabalho e após isso analisar,de maneira semelhante ao que ocorre em Uchôa *et al.* (2021) e Coutinho *et al.* (2022), como a interação entre esses três fatores contribui para a duração do intervalo de correção de erros de IC.

Pretende-se também, conduzir uma pesquisa qualitativa para identificar as razões que justificam a ocorrência das relações de influência descobertas neste trabalho, assim como, refazer este estudo com projetos de código aberto hospedados no GitHub, de modo a verificar se os resultados obtidos neste trabalho se repetem no contexto de projetos abertos e para que se consiga formar uma base mais ampla de *builds* falhos e que permita assim a generalização dos resultados encontrados.

Além disso, pretende-se ainda realizar uma pesquisa empírica para examinar se a adição de más práticas de IC contribui ou não para um longo intervalo de correção de erros de IC em projetos de código fechado.

# REFERÊNCIAS

ALOMAR, E. A.; PERUMA, A.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A. On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In: **Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops**. New York, NY, USA: Association for Computing Machinery, 2020. (ICSEW'20), p. 342–349. ISBN 9781450379632. Disponível em: https://doi.org/10.1145/3387940.3392193. Acesso em: 15 fev. 2022.

BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. Oops, my tests broke the build: An explorative analysis of travis ci with github. In: **2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)**. [*S. l.*: *s. n.*], 2017. p. 356–367.

BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In: **Proceedings of the 14th International Conference on Mining Software Repositories**. IEEE Press, 2017. (MSR '17), p. 447–450. ISBN 9781538615447. Disponível em: https://doi.org/10.1109/MSR.2017.24. Acesso em: 15 fev. 2022.

COUTINHO, D.; UCHÔA, A.; BARBOSA, C.; SOARES, V.; GARCIA, A.; SCHOTS, M.; PEREIRA, J. A.; ASSUNçãO, W. K. G. On the influential interactive factors on degrees of design decay: A multi-project study. In: **Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, Honolulu, Hawaii, March 15–18**. [*S. l.*: *s. n.*], 2022. p. 1–12.

ELAZHARY, O.; WERNER, C.; LI, Z. S.; LOWLIND, D.; ERNST, N. A.; STOREY, M.-A. Uncovering the benefits and challenges of continuous integration practices. In: **IEEE Transactions on Software Engineering**. [*S. l.*: *s. n.*], 2021. p. 1–1.

GÜEMES-PEÑA, D.; LÓPEZ-NOZAL, C.; MARTICORENA-SÁNCHEZ, R.; MAUDES-RAEDO, J. Emerging topics in mining software repositories. **Progress in artificial intelligence**, Springer, v. 7, n. 3, p. 237–247, 2018.

HAN, J.; PEI, J.; KAMBER, M. **Data Mining**: Concepts and techniques. Elsevier Science, 2011. (The Morgan Kaufmann Series in Data Management Systems). ISBN 9780123814807. Disponível em: https://books.google.com.br/books?id=pQws07tdpjoC. Acesso em: 15 fev. 2022.

HASSAN, F. Tackling build failures in continuous integration. In: **Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering**. IEEE Press, 2019. (ASE '19), p. 1242–1245. ISBN 9781728125084. Disponível em: https://doi.org/10.1109/ASE.2019.00150. Acesso em: 15 fev. 2022.

HILTON, M.; TUNNELL, T.; HUANG, K.; MARINOV, D.; DIG, D. Usage, costs, and benefits of continuous integration in open-source projects. In: **2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [*S. l.*: *s. n.*], 2016. p. 426–437.

ISLAM, M. R.; ZIBRAN, M. F. Insights into continuous integration build failures. In: **2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)**. [*S. l.*: *s. n.*], 2017. p. 467–470.

KERZAZI, N.; KHOMH, F.; ADAMS, B. Why do automated builds break? an empirical study. In: **2014 IEEE International Conference on Software Maintenance and Evolution**. [*S. l.*: *s. n.*], 2014. p. 41–50.

MEYER, M. Continuous integration and its tools. **IEEE Software**, v. 31, n. 3, p. 14–16, 2014.

MSR. Mining Software Repositories Conference, 2022. Disponível em: https://conf.researchr.org/home/msr-2022. Acesso em: 15 fev. 2022.

RAHMAN, M. M.; ROY, C. K. Impact of continuous integration on code reviews. In: **2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)**. [*S. l.*: *s. n.*], 2017. p. 499–502.

RAUSCH, T.; HUMMER, W.; LEITNER, P.; SCHULTE, S. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In: **2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)**. [*S. l.*: *s. n.*], 2017. p. 345–355.

SAIDANI, I.; OUNI, A.; CHOUCHEN, M.; MKAOUER, M. W. Predicting continuous integration build failures using evolutionary search. **Information and Software Technology**, v. 128, p. 106392, 2020. ISSN 0950-5849. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584920301579. Acesso em: 15 fev. 2022.

SILVA, R. B. T.; BEZERRA, C. I. M. Analyzing continuous integration bad practices in closed-source projects: An initial study. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (SBES '20), p. 642–647. ISBN 9781450387538. Disponível em: https://dl.acm.org/doi/10.1145/3422392.3422474. Acesso em: 15 fev. 2022.

SOARES, D. M.; JúNIOR, M. L. de L.; MURTA, L.; PLASTINO, A. What factors influence the lifetime of pull requests? **Software: Practice and Experience**, v. 51, n. 6, p. 1173–1193, 2021. Disponível em: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2946. Acesso em: 15 fev. 2022.

TAN, P.; STEINBACH, M.; KUMAR, V. **Introduction to Data Mining**. Pearson, 2014. ISBN 9789332571402. Disponível em: https://books.google.com.br/books?id=NW86tAEACAAJ. Acesso em: 15 fev. 2022.

UCHÔA, A.; BARBOSA, C.; COUTINHO, D.; OIZUMI, W.; ASSUNÇAO, W. K.; VERGILIO, S. R.; PEREIRA, J. A.; OLIVEIRA, A.; GARCIA, A. Predicting design impactful changes in modern code review: A large-scale empirical study. In: IEEE. **2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)**. [*S. l.*], 2021. p. 471–482.

VALENTE, M. T. **Engenharia de Software Moderna**: Princípios e práticas para desenvolvimento de software com produtividade. [*S. l.*: *s. n.*], 2020.

VASSALLO, C.; PROKSCH, S.; GALL, H. C.; PENTA, M. D. Automated reporting of anti-patterns and decay in continuous integration. In: IEEE. **2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)**. [*S. l.*], 2019. p. 105–115.

VASSALLO, C.; PROKSCH, S.; ZEMP, T.; GALL, H. Every build you break: Developer-oriented assistance for build failure resolution. In: **Empirical Software Engineering**. [*S. l.*]: Springer, 2020. v. 25, n. 3, p. 2218–2257.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. [*S. l.*]: Springer Science & Business Media, 2012.

ZAMPETTI, F.; BAVOTA, G.; CANFORA, G.; PENTA, M. D. A study on the interplay between pull request review and continuous integration builds. In: **2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [*S. l.*: *s. n.*], 2019. p. 38–48.

ZAMPETTI, F.; VASSALLO, C.; PANICHELLA, S.; CANFORA, G.; GALL, H.; PENTA, M. D. An empirical characterization of bad practices in continuous integration. **Empirical Software Engineering**, v. 25, 03 2020.