UNIVERSIDADE FEDERAL DA BAHIA
INSTITUTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

INVESTIGANDO A ASSOCIAÇÃO ENTRE RAJADAS DE MUDANÇAS E O STATUS DA BUILD

Juvenal Constantino de Macêdo Junior

DISSERTAÇÃO DE MESTRADO

Salvador, Ba
18 de Dezembro de 2023

UNIVERSIDADE FEDERAL DA BAHIA
INSTITUTO DE COMPUTAÇÃO

Juvenal Constantino de Macêdo Junior

INVESTIGANDO A ASSOCIAÇÃO ENTRE RAJADAS DE MUDANÇAS E O STATUS DA BUILD

Trabalho apresentado ao PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO do INSTITUTO DE COMPUTAÇÃO da UNIVERSIDADE FEDERAL DA BAHIA como requisito parcial para obtenção do grau de Mestre em CIÊNCIA DA COMPUTAÇÃO.

Orientador: RODRIGO ROCHA GOMES E SOUZA
Co-orientador: TIAGO OLIVEIRA MOTTA

Salvador, Ba
18 de Dezembro de 2023

Ficha catalográfica.

JUNIOR, JUVENAL C. M

Investigando a associação entre Rajadas de Mudanças e o Status da *Build* / Juvenal Constantino de Macêdo Junior– Salvador, Ba, 18 de Dezembro de 2023.

49p.: il.

Orientador: RODRIGO ROCHA GOMES E SOUZA.
Co-orientador: TIAGO OLIVEIRA MOTTA.
Dissertação de Mestrado– UNIVERSIDADE FEDERAL DA BAHIA, INSTITUTO DE COMPUTAÇÃO, 18 de Dezembro de 2023.

1. Rajadas de Mudanças 2. Integração Contínua 3. Repositório de Software 4. Controle de Versão 5. Estudo Empírico 6.Engenharia de Software. I. SOUZA, RODRIGO R. G.. II. MOTTA, TIAGO OLIVEIRA.
III. UNIVERSIDADE FEDERAL DA BAHIA. INSTITUTO DE COMPUTAÇÃO. IV. Título.

CDU 004.274

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DA BAHIA
INSTITUTO DE COMPUTAÇÃO
PGCOMP - Programa de Pós-Graduação em Ciência da Computação
http://pgcomp.ufba.br

"*Investigando a associação entre Rajadas de Mudanças e o Status da Build*"

Juvenal Constantino de Macêdo Junior

Dissertação apresentada ao Colegiado do Programa de Pós-Graduação em Ciência da Computação na Universidade Federal da Bahia, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

Banca Examinadora

Rodrigo Rocha

Prof. Dr. Rodrigo Rocha Gomes e Souza (Orientador - PGCOMP)

Prof. Dr. José Amancio Macedo Santos (UEFS)

Machado

Prof. Dr. Ivan do Carmo Machado (UFBA)

…

AGRADECIMENTOS

Inicialmente, o meu muito obrigado a Deus. A razão principal por ter conseguido chegar até aqui, sem ele não conseguiria concluir este estudo e essa etapa de minha vida pessoal, acadêmica e profissional.

Agradeço aos meus pais, Juvenal Constantino de Macêdo e Maria do Carmo Rodriguês, que sempre acreditaram e me apoiaram.

Agradeço à minha esposa Diana Calhau Barbosa, que acompanhou todo esse processo me apoiando incondicionalmente.

Agradeço aos meus irmãos Diego Rodriguês, Adriano Calhau, Johacia Costa pelo grande apoio no período que estive em Salvador.

Agradeço à Universidade Federal da Bahia, Universidade Salvador.

Agradeço ao professor Rodrigo Rocha, ser humano incrível e excelente profissional. Um orientador que fez toda diferença! Obrigado por tudo Rodrigo!

Agradeço ao co-orientador, Tiago Oliveira Motta, grande amigo que contribuiu, ao longo desta dissertação, com considerações, intuições, indagações, reflexões que fortaleceram esta pesquisa de mestrado.

Agradeço aos professores das componentes curriculares que cursei durante este mestrado, todos com grandes contribuições ao meu conhecimento científico e acadêmico: Ivan Machado, Cláudio Sant'Anna, Eduardo Almeida e Christina Flach.

Agradeço aos amigos e amigas do Laboratório de Engenharia de Software, em especial a Roselane Silva, Moara Brito e Luciane Meconi que me deram total apoio e suporte em salvador.

Agradeço a todos os pesquisadores e profissionais que dispuseram de seu tempo e intelectualidade para responder aos *survey* aplicados ao longo deste estudo.

Agradeço aos meus amigos atuais de trabalho em especial meus chefes Ivan Couto e Anny Jayara, que, mesmo diante da correria do dia a dia proporcionam-me todo suporte e cuidado.

RESUMO

Rajadas de commits são sequências de modificações realizadas por desenvolvedores, que ocorrem no código dentro de um curto período de tempo. Em projetos que adotam a prática de Integração Contínua (IC), toda vez que uma modificação é finalizada, cria-se uma nova versão do código, gerando uma nova *build*. Nessa nova versão, as mudanças realizadas são verificadas automaticamente, executando testes de unidade e relatando o resultado de falha ou sucesso da *build* para os desenvolvedores. Nesse sentido, o objetivo desta pesquisa é realizar um estudo empírico para verificar a associação entre rajadas de commits e falhas na *build*. O primeiro passo foi a realização de um estudo empírico a partir da mineração de repositórios, por meio do qual se identificou a relação das rajadas de commits e falha na build. Após esse estudo, foi conduzido um *survey*, para se avaliar os resultados obtidos, considerando as opiniões de desenvolvedores que trabalham com IC. Dentre as descobertas realizadas, os resultados do primeiro estudo mostram que em alguns projetos a taxa de sucesso de *builds* após rajadas de mudanças tendem a diminuir. No entanto, não podemos generalizar os resultados para todos os projetos visto que, na maioria dos projetos estudados a diferença não foi estatisticamente significativa. No segundo estudo, a maioria dos participantes da pesquisa concordam que a proximidade da data de entrega de um projeto é um fator responsável pela rajada de commits e falha na build. No entanto, não podemos generalizar os resultados, visto que alguns desenvolvedores apresentaram pequenas discordâncias quanto aos responsáveis pela falha na build. Desse modo, os resultados deste trabalho pretendem contribuir com a comunidade de desenvolvedores que utilizam IC, ajudando-os a reduzir falhas na *build*, podendo orientar trabalhos futuros sobre boas práticas de desenvolvimento.

Palavras-chave: Rajadas de Mudanças, Integração Contínua, Repositório de Software, Controle de Versão, Estudo Empírico, Engenharia de Software.

ABSTRACT

Commit bursts are sequences of changes made by developers that occur in code within a short period of time. In projects that adopt the practice of Continuous Integration (CI), every time a modification is completed, a new version of the code is created, generating a new *build*. In this new version, the changes made are verified automatically, running unit tests and reporting the result of failure or success of *build* to the developers. In this sense, the objective of this research is to carry out an empirical study to verify the association between bursts of commits and failures in *build*. The first step was to carry out an empirical study based on the mining of deposits, through which the relationship between bursts of commits and construction failure was identified. After this study, a *survey* was followed to evaluate the results obtained, considering the opinions of developers who work with CI. Among the discoveries made, the results of the first study show that in some projects the success rate of *builds* after bursts of changes tends to decrease. However, we cannot generalize the results to all projects since, in most of the studied projects, the difference was not statistically significant. In the second study, most survey participants agree that the proximity of a project's delivery data is a factor responsible for bursts of commits and build failure. However, we cannot generalize the results, since some developers had small disagreements about who was responsible for the construction failure. Thus, the results of this work intend to contribute to the community of developers who use CI, helping them to reduce failures in *build*, facilitating future work on good development practices.

Keywords: Change Bursts; Software Repository; Continuous Integration; Version Control; Empirical Study.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

INTRODUÇÃO

De acordo com Nagappan et al. (2010), rajadas de mudanças são as várias modificações no código que ocorrem em um curto período de tempo. Tais modificações, quando feitas durante determinados períodos do desenvolvimento de *software*, são indicadores de futuros defeitos. Ao mesmo tempo, implementar muitas mudanças em um curto espaço de tempo torna o processo de desenvolvimento mais complexo, podendo causar novos defeitos, visto que os autores também relatam que no desenvolvimento de *software* toda mudança induz a um risco.

O processo de desenvolvimento de *software* é visto como uma atividade contínua que consiste em efetuar várias modificações no código ao longo do tempo do desenvolvimento, como eliminar artefatos não mais necessários, corrigir um defeito ou contribuir para futuras manutenções. (NAGAPPAN et al., 2010).

Para facilitar o processo de contribuição e manutenção do *software*, ferramentas auxiliares, como o sistema de controle de versões, ajudam a registrar as mudanças feitas ao longo do tempo, de forma que o desenvolvedor possa recuperar versões específicas do código (REBOUÇAS et al., 2017). Com isso, o sistema de controle de versão permite identificar a evolução do *software* através de *commits* (modificações finalizadas que são realizadas no código) (HUMBLE; FARLEY, 2014). Alguns exemplos de informações que podem ser extraídas a partir do *commit* são o autor, a data e a hora da execução da mudança. O controle de versão também é utilizado como parte crucial para o sucesso da Integração Contínua.

Fowler (2006) define Integração Contínua (IC) como uma prática de desenvolvimento de *software* na qual desenvolvedores integram seu código em uma linha principal compartilhada com frequência, e verificam a qualidade de suas contribuições continuamente através das *builds*. Em outras palavras, a cada mudança realizada no código, o colaborador submete suas alterações para o servidor de IC, e todos os testes são executados automaticamente. No projeto *Travis CI*, por exemplo, após a execução dos testes, é retornado um dos seguintes status da *build*: *passed* (passou), *failed* (falhou), *canceled* (cancelado), ou *error* (erro) (RAHMAN; ROY, 2017).

Na IC, as *builds* são construídas com o objetivo de agilizar o processo de desenvolvimento (ISLAM; ZIBRAN, 2017). Desse modo, é esperado que essas *builds* não falhem após os testes unitários, evitando-se, assim, o atraso do projeto e interrupção da equipe de desenvolvimento, pois de acordo com Humble e Farley (2014), com a falha da *build* os desenvolvedores precisam interromper o desenvolvimento para solucionar o problema. De forma semelhante, Fowler (2006) afirma que a não ocorrência da falha da *build* proporciona uma significante redução dos problemas de integração e agilidade no desenvolvimento de *software*. Tais falhas são ocasionadas por alterações feitas no código, seja devido aos erros encontrados ou à necessidade de adicionar novas funcionalidades, tornando-se um processo contínuo de mudanças no *software* (TRAVIS-CI, 2017).

Visto que existe uma ausência de estudos sobre a associação das rajadas de commits com o status da *build*, percebe-se a necessidade de apresentar os resultados desta pesquisa à comunidade de desenvolvedores e pesquisa em Engenharia de *software*. Sendo assim, realizou-se um estudo quanti-qualitativo para caracterizar a percepção de desenvolvedores frente aos resultados obtidos até aqui. A próxima subseção apresenta o objetivo geral, em seguida os objetivos específicos e por fim um conjunto de trabalhos relacionados a este estudo.

1.1 OBJETIVO GERAL

O objetivo geral deste trabalho é estudar a relação entre as rajadas de mudanças e o status da *build*. Com o intuito de atingir esse objetivo, realizaremos um estudo exploratório que inclui a mineração de um conjunto de dados do projeto *TravisTorrent* para responder à seguinte questão de pesquisa.

- **QP1:** *Builds* que ocorrem após rajadas de mudanças possuem maior propensão a terem falhas?

  Não encontramos estudos que verifiquem esse tipo de análise. Contudo, Nagappan et al. (2010) relatam que implementar muitas mudanças em um curto período de tempo complica o processo de desenvolvimento, levando a defeitos. Diante disso, acreditamos que após as rajadas de mudanças, as *builds* tendem a falhar. A partir dessa questão formulamos as seguintes subquestões de pesquisa:

  QP1.1: Qual a percepção dos desenvolvedores sobre a causa das rajadas de commits?

  QP1.2: Qual a percepção dos desenvolvedores sobre influência da rajada de commits na falha da *build*?

  Para responder às subquestões, realizou-se um survey aplicado a desenvolvedores experientes. O Capítulo 4 detalha cada uma delas.

1.2 OBJETIVOS ESPECÍFICOS

São objetivos de pesquisa específicos deste trabalho:

O1. Identificar rajadas de commits e suas causas.

A identificação das rajadas de commits é feita através de um algoritmo de detecção de rajadas. Para as possíveis causas aplicou-se um questionário com desenvolvedores para melhor compreensão.

O2. Verificar a associação entre rajadas de commits e a falha na *build*.

Será investigada as *builds* que falharam antes e após as rajadas, para associar os resultados e verificar se houve ou não impacto na falha da *build*.

1.3 TRABALHOS RELACIONADOS

Alguns trabalhos, como (BOGARD; TIEDERMAN, 1986), (ZHU; SHASHA, 2003), e (ZHANG; SHASHA, 2006) propuseram metodologias de detecção de rajadas em coletas de emails, textos e postagens no Twitter. No entanto, os métodos de detecção de rajadas propostos não foram aplicados ao contexto de mudanças de software e não focaram em correlacionar rajadas com *builds*.

O estudo de Nagappan et al. (2010) mostrou que as métricas de rajadas de mudanças produzem uma excelente capacidade preditiva de *bugs*. Eles realizaram um experimento utilizando métricas temporais, métricas de pessoas e métricas de mudanças no código (*churn*) e verificaram a quantidade de mudanças no código para prever a densidade de defeitos no nível do arquivo. Como resultado, eles encontraram que a precisão e o *recall* excedem 90% para as rajadas de mudanças no sistema operacional Windows Vista. No entanto, uma limitação desse trabalho é que o experimento foi aplicado em apenas dois sistemas comerciais, e não em sistemas de código aberto. Apesar de os autores implementarem métricas de rajadas de mudanças, eles não utilizaram um *dataset* de integração contínua.

Islam e Zibran (2017) realizaram um estudo empírico utilizando o conjunto de dados dos projetos *TravisTorrent* e *Travis CI*, e descobriram que os resultados da *build* são significantemente afetados pelo número de linhas alteradas no código, número de arquivos e número de contribuintes na *build*.

Já Rebouças et al. (2017) utilizaram o conjunto de dados do projeto *TravisTorrent* para comparar as taxas de sucesso das *builds* feitas por contribuintes ocasionais e não ocasionais e concluíram que não há diferença representativa no sucesso da *build*, sendo semelhantes em 85% dos projetos analisados. Ainda que o estudo dos autores tenha envolvido o conjunto de dados do *TravisTorrent* associado à IC, ele não investiga rajadas.

Beller, Gousios e Zaidman (2017) concluíram que os testes realizados nas *builds* são os principais fatores responsáveis pela falha da *build*. Embora este trabalho tenha focado principalmente nos impactos dos testes, ele também relatou outros fatores que influenciam os resultados da falha da *build*, tais como a linguagem de programação associada ao número de testes executados e o uso de vários ambientes de integração.

O nosso trabalho se diferencia dos três estudos apresentados, Islam e Zibran (2017), Rebouças et al. (2017), Beller, Gousios e Zaidman (2017) pois buscamos investigar falhas na *build* com as rajadas de mudanças.

Kerzazi, Khomh e Adams (2014) investigaram o causador da falha da *build*, analisaram 3.214 builds produzidos em uma grande empresa de *software* no período de 6 meses,

para descobrir os principais fatores que contribuem para a *build* falhar. Eles identificaram que as falhas de compilação se correlacionam com o número de colaboradores simultâneos em filiais, o tipo de itens de trabalho desempenhado em uma agência, e os papéis desempenhados pelas partes interessadas das compilações, por exemplo, desenvolvedores versus integradores. Já os pesquisadores Kwan, Schroter e Damian (2011), realizaram um estudo de caso no projeto IBM RAtional Team Concert, examinaram o efeito das mudanças da congruência sociotécnicas na probabilidade de sucesso da *build*. O estudo foi realizado em um grande sistema industrial, e puderam concluir que que o número de colaboradores em uma agência e o período durante o qual uma mudança é feita são fatores que podem ser manipulados para reduzir a falha da *build*. Ambos se diferenciam do nosso trabalho, uma vez que não relacionaram com as rajadas de commits.

Como se percebe, embora a pesquisa em Engenharia de *Software* tenha colocado em pauta a percepção dos desenvolvedores sobre a falha da *build*, não se tem disponível um estudo sobre a percepção desse público relacionada com a rajada de commits.

1.4 ESTRUTURA DA DISSERTAÇÃO

Este trabalho está organizado da seguinte forma: o Capítulo 1 apresenta as definições iniciais e o contexto deste trabalho, assim como o problema e as questões de pesquisa. No Capítulo 2, está explicitada a fundamentação teórica que aborda os principais conceitos de *build*, IC, *Travis CI*, controle de versão e rajadas de commits. O Capítulo 3 está relacionado à questão de pesquisa **QP1** e descreve o estudo exploratório, apresentando o design do estudo e resultados. Já o Capítulo 4 está relacionado às subquestões de pesquisa **QP1.1 e QP1.2**, e traz as informações obtidas através do questionário que realizamos como os resultados e a conclusão.

Capítulo 2

REVISÃO BIBLIOGRÁFICA

Este Capítulo apresenta os conceitos que antecedem a escrita deste projeto e nos quais ele se baseia, fornecendo descrições gerais, métodos e abordagens relativas ao campo de estudo proposto. Serão descritos os conceitos relativos à *Build*, Integração Contínua, *Travis CI*, Controle de Versão e as Rajadas de Mudanças.

2.1 O QUE É UMA BUILD?

Uma *build*, é a compilação de todos os arquivos do código para gerar uma versão executável do sistema (ZHANG; SHASHA, 2006). Para melhor compreender seu significado, Duvall, Matyas e Glover (2007) definem *build* da seguinte forma:

> Uma *build* é muito mais do que uma compilação (ou suas variações em linguagens dinâmicas). Uma *build* pode consistir na compilação, teste, inspeção e implantação, entre outras coisas. Uma *build* funciona como o processo para colocar o código fonte em conjunto e verificar se o *software* funciona como uma unidade coesa.

Existem ferramentas que auxiliam a construção da *build*, como *Ant*, *Grunt*, *Gulp* ou *Maven*. No entanto, cabe ao desenvolvedor definir o escopo a ser automatizado, podendo ser: i) um teste de classe, ii) criação de uma tabela no banco de dados, iii) compilação ou iv) compressão de arquivos, por exemplo, *CSS* e *JavaScript* (FOWLER, 2006).

A Figura 2.1 apresenta um *script* de *build* representando as seguintes funcionalidades: deixar o projeto pronto para ser compilado, efetuar a compilação do código fonte, fazer a integração com o banco de dados, realizar a execução dos testes, fazer as inspeções no código e, por fim, implantar o *software*.

De acordo com Humble e Farley (2014), as falhas nos testes de *commit* podem ser atribuídas a uma das três seguintes causas: (i) um erro de sintaxe foi introduzido no código e detectado por compilação em linguagens compiladas; (ii) um erro semântico foi introduzido na aplicação, causando a falha de um ou mais testes; (iii) houve algum

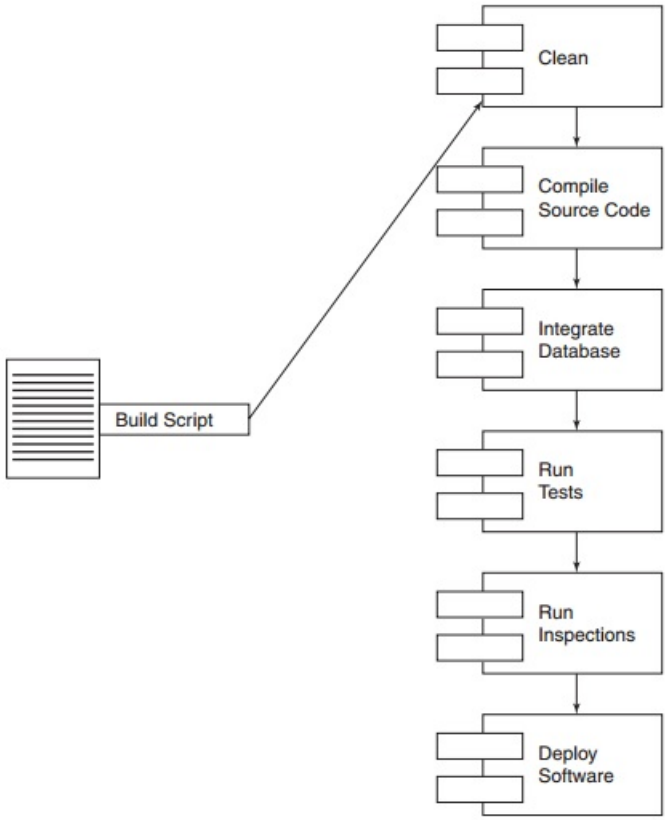


Figura 2.1 Integração contínua: melhorando a qualidade do *software* e reduzindo o risco (DUVALL; MATYAS; GLOVER, 2007).

problema com a configuração da aplicação ou de ambiente (incluindo o sistema operacional). Independentemente do problema, em caso de falhas o estágio de *commit* deve avisar os desenvolvedores no momento em que os testes terminarem e oferecer um resumo das razões das falhas, como uma lista de testes que não tiveram sucesso, erros de compilação ou quaisquer outras condições de erro. Os desenvolvedores devem ter fácil acesso à saída gerada pela execução do estágio da *build*, que por sinal pode ter sido executada em várias máquinas diferentes.

Com isso, é observada a importância de se manter o funcionamento constante do *software*, visto que a IC coopera durante o desenvolvimento, podendo evitar falha da *build*.

2.2 INTEGRAÇÃO CONTÍNUA

A Integração Contínua (IC) é uma prática de desenvolvimento que envolve a integração frequente de alterações de código em um repositório. Os desenvolvedores podem integrar com frequência, enquanto cada integração é verificada por uma compilação automatizada e testada (TUFANO; SAJNANI; HERZIG, 2019). As compilações automatizadas fornecem feedback antecipado sobre o processo da integração e são fundamentais, ajudando

também a equipe de lançamento para automatizar totalmente o processo de liberação de *software*, com o objetivo de acelerar o lançamento com segurança, seja para indústria ou pesquisa (FOWLER, 2006).

A colaboração geograficamente distribuída desencadeou o trabalho paralelo dos desenvolvedores, válido para *software* aberto ou comercial (XIA; LI, 2017). Uma maneira eficiente de integrar o código fonte mais rápido e averiguar o resultado é utilizando a IC, pois a mesma é considerada um componente relevante para o desenvolvimento paralelo de *software* (ISLAM; ZIBRAN, 2017).

A IC objetiva diminuir os riscos de falha no código por meio do monitoramento recorrente das mudanças realizadas, que permite uma integração constante do código (FOWLER, 2006). Os sistemas que utilizam IC tornam o processo de teste automatizado e esse processo passa a ser recorrente durante o desenvolvimento, ou seja, a cada alteração efetuada, a integração é verificada por uma *build* automatizada (incluindo testes) para detectar erros de integração o mais rápido possível (FOWLER, 2006). Para compreender melhor a IC, a Figura 2.2 ilustra seu funcionamento.

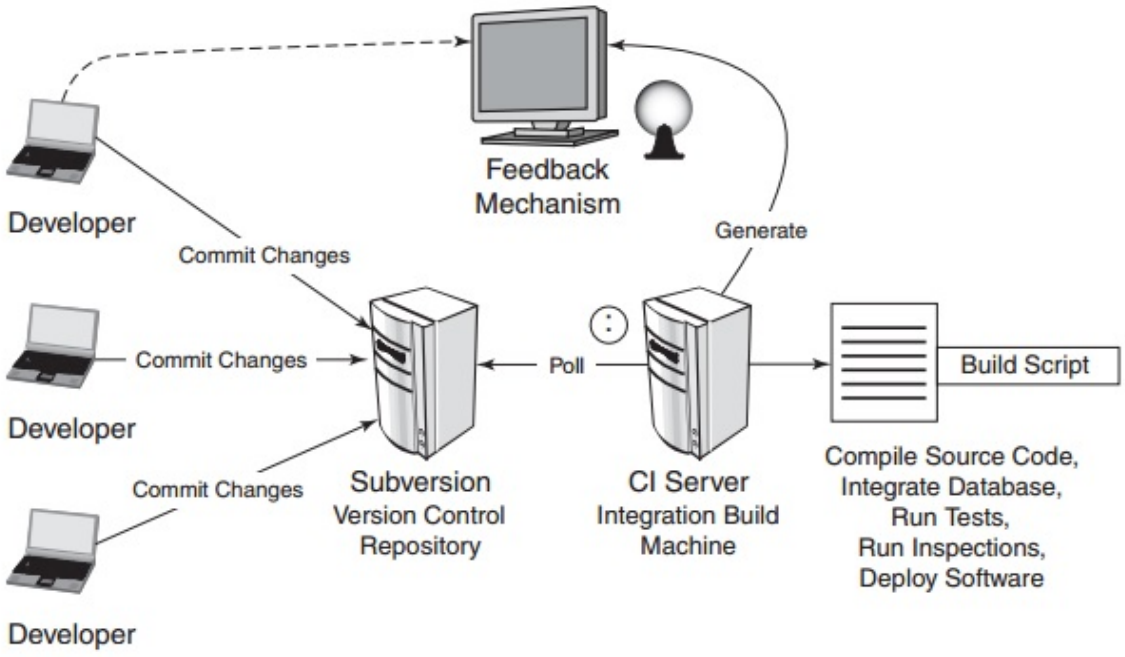


Figura 2.2 Integração contínua: melhorando a qualidade do *software* e reduzindo o risco (DUVALL; MATYAS; GLOVER, 2007)

A Figura 2.2 descreve o cenário em que um servidor de integração contínua é utilizado. Os ambientes dos desenvolvedores representados nesta figura pelos notebooks, são distintos, totalizando três, no qual cada um faz uma cópia do projeto através do repositório de controle de versão, que permite efetuar as mudanças necessárias no código. Após as alterações, o desenvolvedor envia o *commit* para o repositório central. Em seguida, o servidor de IC analisa as modificações e executa uma *build* de integração. Após executar os testes na *build*, se ocorrer alguma falha, o desenvolvedor será notificado, permitindo

que o mesmo efetue as correções necessárias para que a *build* tenha sucesso. A próxima subseção apresenta os benefícios da integração continua.

2.2.1 Benefícios da Integração Contínua

O objetivo desta subseção é destacar as características importantes da ferramenta de Integração Contínua na sua utilização. De acordo com Duvall, Matyas e Glover (2007), as principais vantagens em se utilizar um servidor de IC são:

- Reduzir riscos: a integração possibilita a detecção e redução de erros e códigos quebrados implantados pelos desenvolvedores;
- Reduzir processos manuais repetitivos: permite que a equipe seja mais produtiva, pois a *build* executa um conjunto de testes automáticos através do servidor de IC;
- Gerar *software* implementável a qualquer momento e em qualquer lugar: a IC permite fazer pequenas alterações no código-fonte e integra essas mudanças com o resto da base do código em uma base regula;
- Permitir melhor visibilidade do projeto: possibilita ter melhor gerenciamento do sistema e visualizar informações relacionadas ao desenvolvimento como módulos complexos e frequência de código com defeito;
- Estabelecer uma maior confiança no produto do time de desenvolvimento: as integrações constantes podem deixar algumas equipes sufocadas, pois não têm conhecimento dos impactos de suas mudanças no código. No entanto, após os desenvolvedores serem informados pelo sistema IC sobre a ocorrência de erros, juntamente com outros membros da equipe, eles passam a ter mais confiança em efetuar as mudanças necessárias.

Com isso, nota-se a importância do servidor de IC e, segundo Beller, Gousios e Zaidman (2017), o Travis CI[1] era a ferramenta mais utilizada em 2017.

A próxima subseção apresenta a ferramenta *Travis CI*.

2.3 TRAVIS CI

Travis CI é um serviço de IC de código aberto e distribuído que, através da integração com o GitHub, possibilita aos projetos criar e executar seus procedimentos IC sem terem que manter suas próprias infraestruturas (TRAVIS-CI, 2017). Em 2010 surgiu como um projeto de código aberto e, em 2012, tornou-se uma empresa. A partir de setembro de 2017, passou a suportar mais de 26 linguagens de programação diferentes, englobando *Java*, *C++*, *Scala*, *Python*, *R* e *Visual Basic* (TRAVIS-CI, 2017).

[1]Esse estudo utilizou a ferramenta Travis CI, uma vez que era a mais aplicada na época (2017), momento em que o estudo foi realizado. No entanto, atualmente é o GitHub Actions – ver https://blog.jetbrains.com/teamcity/2023/07/best-ci-tools/.

Além disso, a *Travis CI* possui serviço de teste gratuito, após isso, o serviço passa a ser pago, fornece builds não públicas para repositórios privados no *GitHub*, além da sua edição da comunidade, gratuita para sistemas *open source*. A edição privada da *Travis CI* fornece um ambiente de construção mais rápido.

De forma a contribuir com o uso da IC, os repositórios de *software* são fundamentais para que haja comunicação e registro de informações em massa, permitindo contribuições geograficamente paralelas (várias pessoas em lugares diferentes), o que possibilita o cruzamento de informações. Logo, na próxima seção, são apresentados os fundamentos sobre mineração de repositórios de *software*.

2.4 MINERAÇÃO DE REPOSITÓRIOS DE SOFTWARE

Repositório de *software* é toda ferramenta que apoie o processo de desenvolvimento de *software* e que retenha informações sobre as atividades realizadas pelos agentes do processo. Assim, sistemas de controle de versão de *software* (CVS, SVN e Git etc.), sistemas de gerenciamento de defeitos e/ou issues (Bugzilla, Mantis, Jira, etc.), históricos de comunicação (listas de e-mails, discussões, etc) e bancos de dados, são exemplos de repositórios de *software* (KHAN; AHSAN, 2016). Tais repositórios, na prática, são usados apenas para armazenar e reter informações. A seguir, uma breve descrição de algumas tecnologias usadas como repositórios de software.

- **Sistema de Controle de Versão** Segundo Pressman e Maxim (2016), o Sistema de Controle de Versão (SCV) "combina procedimentos e ferramentas para gerenciar diferentes versões dos objetos de configuração criados durante o processo de *software*". Por isso, o controle de versão deve ser utilizado com o apoio da integração contínua para se obter êxito durante o processo de desenvolvimento (XIA; LI, 2017). O SCV permite realizar o trabalho colaborativo, ou seja, vários desenvolvedores compartilham informações e trabalham em conjunto (DUVALL; MATYAS; GLOVER, 2007), além de possibilitar algumas vantagens no que concerne ao trabalho em equipe (SPINELLIS, 2005) tais como:

  - Armazenar a última versão do *software*.
  - Disponibilizar informações sobre as alterações no *software*, permitindo que os desenvolvedores não refaçam trabalhos já finalizados.

  No intuito de contribuir com a Integração Contínua, existem ferramentas para gerenciar o controle de versão, como *CVS*, *Subversion* e *Git*. Com essas ferramentas, os desenvolvedores podem trabalhar em equipe, necessitando de um servidor que assuma a responsabilidade de versionamento do sistema, para que os envolvidos tenham acesso ao estado em que o sistema se encontra (SPINELLIS, 2005).

- **Histórico de comunicação** oriundo de diversas ferramentas que podem ser utilizadas para permitir a comunicação e o armazenamento do histórico de mensagens

trocadas entre os desenvolvedores. São exemplos destes mecanismos: Listas de e-mails, listas de discussões, IRC (Internet Relay Chat) e bate-papo.

De acordo com Duvall, Matyas e Glover (2007) pesquisadores e desenvolvedores de *software* iniciaram a mineração desses repositórios, e acreditam fortemente que tais repositórios são fontes muito importantes de informação que podem apoiar o processo de manutenção de *software*. O campo de mineração de repositórios de *software* avalia o conjunto de dados disponível em repositórios para descobrir informações interessantes e desejadas sobre diferentes projetos de *software*. Segundo (HASSAN; XIE, 2010), é possível realizar diferentes análises com os dados coletados. Algoritmos de mineração podem ser utilizados na busca de tais informações para explicar diferentes tarefas do processo de desenvolvimento do *software*.

Isso é possível, já que os repositórios de *software* têm à disposição uma grande quantidade de dados produzidos durante as atividades de desenvolvimento: milhares ou milhões de linhas de código são escritas, defeitos são relatados e resolvidos, discussões técnicas acontecem nas listas de e-mail e gerenciadores de requisitos etc. Analisar tais dados pode aumentar o entendimento sobre: (i) o projeto, (ii) os desenvolvedores, (iii) os processos que governam a evolução e manutenção do *software*. Isso permite que decisões sejam tomadas de forma mais fundamentada (HASSAN, 2008). As análises também possibilitam que aprendamos mais sobre a engenharia de *software* em si.

Nesse contexto, a mineração de repositórios de *software* é uma área de pesquisa voltada para a recuperação, interligação e análise dos dados históricos produzidos durante o desenvolvimento de *software* e que estão armazenados em repositórios. É um campo de pesquisa relativamente novo e tem atraído o interesse de diversos pesquisadores. Hoje, uma grande parcela dos estudos empíricos em engenharia de *software* envolve algum tipo de mineração de repositórios (GEROSA et al., 2015).

O estudo de Śliwerski, Zimmermann e Zeller (2005), por exemplo, realizaram mineração de código e de suas mudanças a partir do sistema de controle de versão, afim de indicar os *commits* que posteriormente acarretaram em correções, ou seja, mudanças no código que causaram problemas. Os autores concluíram que quanto maior o tamanho de um *commit* maior a probabilidade dele acarretar em uma correção. Outra conclusão apontada foi que *commits* que também são correções possuem três vezes mais possibilidade de induzir outras correções, quando comparado aos *commits* de melhoria.

Contudo, podemos notar que a mineração de repositórios pode ser utilizada para diversas finalidades, e que o tipo de mineração, assim como os repositórios de *software* variam conforme o problema que se quer investigar. Todavia, é importante salientar, que o sucesso da área de mineração de repositórios deve-se, em grande parte, à abundante disponibilização de dados viabilizada pelo movimento de *software* livre. Projetos de grande visibilidade, como o *Linux, Eclipse IDE* e vários outros da *Apache software Foundation e da Free Sofware Foundation*, assim como, mais recentemente, projetos hospedados no *GitHub e TravisTorrent*, são frequentemente escolhidos para realização de estudos cientifícos (GEROSA et al., 2015). A próxima seção apresenta uma breve apresentação sobre rajadas de mudanças no código.

2.5 RAJADAS DE MUDANÇAS

Sistemas de *software* são suscetíveis a mudanças e podem ser alterados a qualquer momento, seja devido aos erros encontrados ou à funcionalidade adicional que é exigida. Tais mudanças tornam-se um processo contínuo no *software* (NAGAPPAN et al., 2010).

Com isso, surgem as rajadas de mudanças, que, segundo Nagappan et al. (2010), são sequências de mudanças que ocorrem no código dentro de um curto período de tempo. Por exemplo, quando o desenvolvedor contribui com o código e efetua alterações em um curto período de tempo, essas mudanças são classificadas como rajadas de mudanças. Mas como identificar se as mudanças realizadas aconteceram em um curto período tempo?

Para o algoritmo identificar as modificações no código e classificá-las como rajada ou não, também é necessário utilizar informações disponíveis no sistema de controle de versão, visto que a cada alteração do desenvolvedor, atualizam-se informações do servidor, submetendo *commit*, que é o conjunto de alterações realizadas. Cada *commit* gera uma nova revisão no repositório, que contém as modificações feitas, data, hora e autor. Essas informações permitem ao algoritmo classificar o *commit* que faz parte de uma rajada. Por isso, rajadas no código (projetos) são períodos em que a frequência de *commit* é muito maior do que a frequência média de um projeto, em outras palavras um projeto pode ser 1 *commit* por minuto, em outro pode ser 1 *commit* por dia. De acordo com Zagorisios (2015) no algoritmo de Kleinberg (2003) a abordagem utilizada foi baseada na modelagem do fluxo usando um autômato de estado infinito, no qual as rajadas aparecem naturalmente como transições de estado, sendo controlado pelos parâmetros s (frequências de um estado) e gamma (custo de transição de estado).

Vale ressaltar que o sistema de controle de versão funciona como uma cópia de todos os arquivos e diretórios em um determinado momento da evolução do projeto. As revisões antigas são mantidas e podem ser recuperadas e analisadas sempre que desejado (HUMBLE; FARLEY, 2014).

Portanto, um código possui rajadas quando sua frequência de alterações é encontrada em uma taxa incomum de mudanças. Para Zagorisios (2015), a detecção de eventos ou rajadas possui uma variedade de aplicações, a exemplo do monitoramento do tráfego de rede ou do preço de um mercado de ações, observações astronômicas ou mineração de dados da Web (ZAGORISIOS, 2015).

Yuan, Jia e Yang (2007) demonstram um algoritmo de detecção de rajadas isento de parâmetros, com base em dados sofisticados estruturais que detectam rajadas em vários tamanhos de janelas concomitantemente. O estudo de Fung et al. (2005) propõe um quadro probabilístico, apoiado na distribuição binomial, para identificar palavras em rajadas, além de utilizar análise espectral para categorizar palavras em quatro categorias para descobrir eventos importantes e menos relatados. No entanto, o algoritmo de Kleinberg foi o escolhido, pois, além de possuir uma biblioteca implementada na linguagem R, é considerado o algoritmo mais bem sucedido na detecção de rajadas de mudanças (FUNG et al., 2005).

Estudos como (BOGARD; TIEDERMAN, 1986), (ZHANG; SHASHA, 2006) e (ZHU; SHASHA, 2003) apresentaram metodologias para detecção de rajadas. No âmbito da mineração de dados, ocasionalmente, vários algoritmos de detecção de rajadas foram

desenvolvidos.

Dentre os algoritmos existentes, em nosso estudo utilizaremos o algoritmo de Kleinberg para a detecção de rajadas. É um algoritmo modelado como um autômato infinito, aplica-se a estados de rajada e não rajada para períodos discretos de tempo, determinando a ocorrência da frequência (maior ou menor) dos eventos (KLEINBERG, 2003). A base para a definição do que deve ser classificado como uma frequência de rajada é fornecida por uma cadeia de Markov, baseada no funcionamento de alguns sistemas estocásticos, sendo o estado subsequente de uma cadeia de acontecimentos que está vinculado apenas ao seu estado atual (KLEINBERG, 2003). Na próxima seção descrevemos sobre o algoritmo utilizado nessa pesquisa.

2.6 ALGORITMO DE KLEINBERG

A detecção de rajadas é uma maneira de identificar períodos de tempo em que algum evento ocorre com frequência, podendo ser usada para identificar modismos ou "explosões" de eventos/rajadas ao longo de um determinado período (KLEINBERG, 2003).

Para melhor compreensão do algoritmo de *Kleinberg*, utilizamos um modelo pré definido, a partir do estudo de Marinsek (2017).

Aqui está a ideia básica: um conjunto de eventos, consistindo em eventos alvo e não-alvo, é observado em cada momento t. O estudo de Marinsek (2017) utilizou-se o exemplo de título de pôster, assim, os eventos alvo podem consistir em títulos de pôster que incluem a palavra conectividade e eventos não-alvo podem consistir em todos os outros títulos de pôster (ou seja, todos os títulos de pôster que não incluem a palavra conectividade). O número total de alvo em cada momento é indicado por d e o número de eventos de destino é indicado por r. A proporção p de eventos alvo em cada momento é igual a r/d.

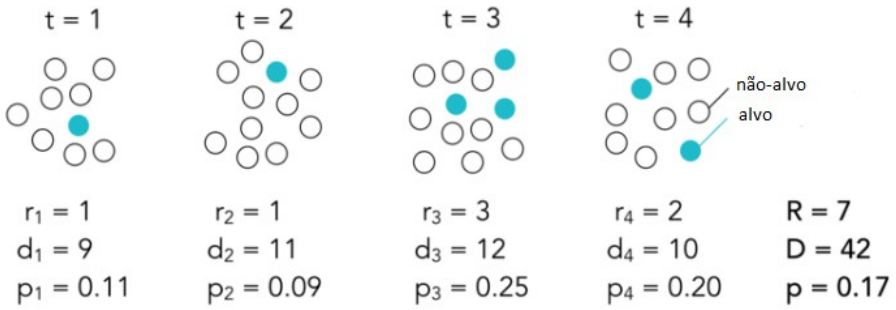


Figura 2.3 Estados (MARINSEK, 2017)

A detecção de rajadas pressupõe que há vários estados (ou modos) que correspondem à diferentes probabilidades de eventos alvo. Alguns estados têm probabilidades de alvo altas, muito baixas e outros têm probabilidades de alvo moderadas. Se assumirmos que existem apenas dois estados possíveis, podemos pensar no estado com menor probabilidade, como o estado da linha de base e no estado com maior probabilidade como o estado "Rajadas". A probabilidade da linha de base é igual à proporção geral de eventos de

destino:

$$Po = R/D$$

em que R é a soma dos eventos de destino em cada momento e D é a soma do total de eventos em cada momento.

A probabilidade do estado é igual à probabilidade da linha de base multiplicada por alguns s constantes. Isso significa que o valor de s pode ser ajustado. Se s for grande, a probabilidade de eventos de destino precisará ser alta para entrar em um estado intermitente.

$$P1 = s * po$$

Quando você está em um determinado estado, espera-se que, em média, os eventos de destino ocorram com a probabilidade associada a esse estado. Às vezes, a proporção de eventos de destino será maior que o esperado e, às vezes, será menor que, devido ao ruído aleatório. O objetivo da detecção de intermitência é prever em que estado o sistema está baseado na sequência de proporções observadas. Em outras palavras, dadas as proporções observadas de eventos alvo em cada lote de eventos, o algoritmo de detecção de intermitência determinará quando o sistema provavelmente estava no estado de linha de base e quando estava provavelmente no estado intermitente.

A determinação de em qual estado o sistema está em um dado momento depende de duas coisas:

1. A qualidade do ajuste entre a proporção observada e a probabilidade esperada de cada estado. Quanto mais próxima a proporção observada estiver da probabilidade esperada de um estado, maior a probabilidade de o sistema estar nesse estado.

2. A dificuldade de fazer a transição do estado anterior para o próximo estado. Existe um custo associado à entrada em um estado mais alto, mas nenhum custo associado à permanência no mesmo estado ou ao retorno a um estado inferior. O custo de transição é igual a zero, ao fazer a transição para um estado inferior ou permanecer no mesmo estado.

O custo total da transição de um estado para outro acontece quando as duas etapas descritas anteriormente ocorrem, permitindo encontrar a sequência de estados ideal, q. A sequência de estados ideal é a sequência de estados que minimiza o custo total ou, em outras palavras, a sequência que melhor explica as proporções observadas.

O que foi encontrado a partir do algoritmo Viterbi e Marinsek (2017). A ideia básica é simples: primeiro, calcula-se o custo de estar em cada estado em $t = 1$ e escolhe-se o estado com o custo mínimo; calcula-se o custo da transição do estado atual em $t = 1$ para cada estado possível em $t = 2$ e, novamente, escolhe-se o estado com o custo mínimo. Repetiu-se essas etapas para todos os pontos do tempo para obter uma sequência de estados que minimize a função de custo (MARINSEK, 2017).

A sequência de estados informa quando o sistema estava em um estado elevado ou "estourado". Podemos repetir essas etapas para diferentes eventos alvo (por exemplo, palavras diferentes nos títulos dos pôsteres) para criar uma linha do tempo de quais eventos eram populares ao longo do tempo.

2.6.1 Simulação com Dados Pré definidos

Marinsek (2017) implementou o algoritmo de detecção de rajadas em R e criou uma série temporal com rajadas artificiais para testar o código. A série de tempo consistia em 1000 pontos de tempo e rajadas foram adicionadas de $t = 200$ a $t = 399$ e $t = 700$ a $t = 799$. Esta é a aparência do curso de tempo bruto:

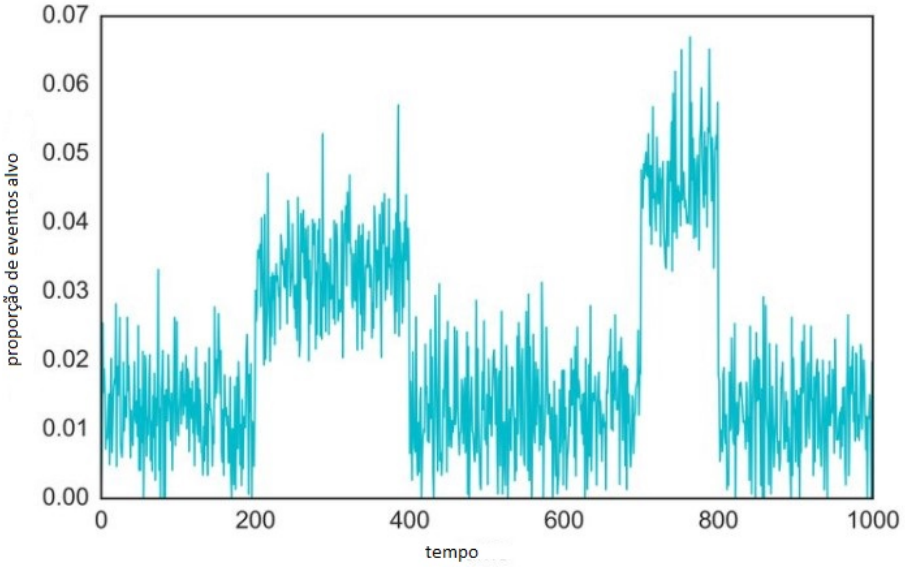


Figura 2.4 Proporção de eventos alvo (MARINSEK, 2017)

Definindo s como 2 e gama como 1, o algoritmo identificou uma rajada de $t = 701$ a $t = 800$ e 32 pequenas rajadas entre $t = 200$ e $t = 395$. O que isso nos diz? O algoritmo de detecção de rajadas pode identificar facilmente períodos nos quais a proporção de eventos alvo é muito maior do que o normal, mas tem mais dificuldade para identificar rajadas mais fracos, especialmente na presença de ruído. Repete-se a análise usando valores diferentes para s e gama para ter uma ideia de como esses valores afetam a detecção de rajadas.

Aqui, as rajadas de cada análise (representados com barras azuis) são plotados na mesma linha do tempo, em que s é a distância entre os estados. Quando s é pequeno, a diferença entre as probabilidades esperadas dos estados também é pequena. Quando aumentamos s enquanto mantemos a constante gama (como mostrado nas primeiras quatro linhas de tempo), obtemos rajadas mais curtos. Essencialmente, estamos dividindo rajadas maiores em rajadas menores, pois estamos aumentando o limite que as proporções observadas precisam atingir para serem consideradas um rajadas.

Gama determina o quão difícil é entrar em um estado superior. Como não há custo associado a permanecer no mesmo estado ou retornar a um estado inferior, a mudança de gama deve afetar apenas o início das explosões e não seus finais. É possivel imaginar que à medida que gama aumenta, obtém-se rajadas menores e mais curtas, pois, novamente, torna-se mais difícil entrar em um estado de rajadas. Não é óbvio na linha do tempo, mas se olhar para o ponto inicial e final da rajadas que sobreviveu a todas as configurações

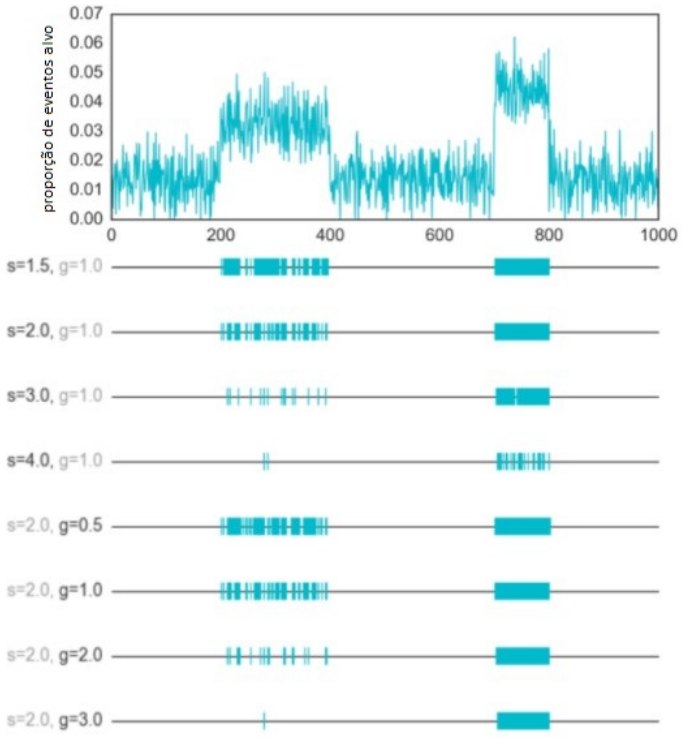


Figura 2.5 proporção de eventos alvo (MARINSEK, 2017)

de gama, percebe-se que a rajadas termina em $t = 281$, independentemente do gama. No entanto, começa em $t = 274$ quando *gama* é $0,5$; em $t = 279$, quando *gama* é 1; e $t = 280$, quando *gama* é 2 ou 3.

Os resultados da detecção de rajadas dependem muito dos parâmetros que são utilizados, pois, de acordo com Marinsek (2017), é um método útil se você estiver interessado em tendências gerais em um grande conjunto de dados ao longo do tempo.

O próximo capítulo apresenta o primeiro estudo desta pesquisa de mestrado, no qual foram explorados repositórios do Travis CI e do GitHub, a partir de mensagens de *commit*, com auxílio do algoritmo de *Kleinberg* para detecção das rajadas de *commits*.

Capítulo 3

UM ESTUDO EMPÍRICO SOBRE RAJADAS DE MUDANÇAS E FALHA NA BUILD

3.1 MÉTODOS DA PESQUISA

Este estudo é de natureza exploratória. Para atingir os objetivos e responder à QP1, foram utilizados os métodos apresentados na Figura 3.1.

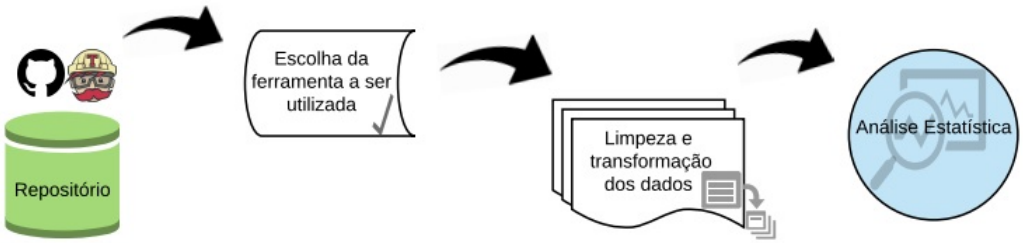


Figura 3.1 Resumo da Metodologia

Efetuou-se a mineração do conjunto de dados *TravisTorrent*[1], livremente disponível, sintetizado a partir do *Travis CI* e do *GitHub*. A metodologia do estudo pode ser resumida da seguinte forma:

- **Repositório** – Para a realização do estudo, foi necessário utilizar um conjunto de dados para extrair as informações das rajadas de mudanças e correlacioná-las com o status da *build*. Os dados utilizados foram os *logs* de *commits* do GitHub associados aos dados do TravisTorrent.
- **Escolha da ferramenta** – O algoritmo de *Kleinberg* (KLEINBERG, 2003) foi utilizado para a detecção de rajadas de mudanças, pois está implementado na biblioteca *bursts* do R[2], que consiste em uma linguagem de programação para cálculos

[1]https://travistorrent.testroots.org/
[2]https://www.r-project.org/

estatísticos e geração de gráficos, o que permite a automatização no processo da pesquisa. Além disso, ele é considerado o melhor algoritmo de detecção de rajadas segundo Tamura e Kitakami (2014).

- **Limpeza e transformação de dados** – Nessa etapa, realizou-se seleção, agrupação dos dados e aplicação do algoritmo *Kleinberg*.

- **Análise de dados** – Após a limpeza e transformação de dados, os resultados encontrados foram analisados estatisticamente a fim de responder às questões de pesquisa propostas.

Dessa forma, realizou-se um estudo empírico para responder à seguinte questão de pesquisa:

QP1: *Builds* que ocorrem após rajadas de mudanças possuem maior propensão a terem falhas?

As *builds* com falhas representam uma ameaça para a eficiência do desenvolvimento, uma vez que Islam e Zibran (2017), relatam que essas falhas impedem o trabalho dos desenvolvedores e atrasam o progresso do projeto.

De acordo com Islam e Zibran (2017), os fatores que causam essas falhas devem ser identificados, pois a identificação dos fatores permite aos desenvolvedores tomar medidas cautelosas para minimizar seus impactos e, portanto, reduzir substancialmente o tempo de desenvolvimento. Assim sendo, investigamos se há uma relação de proporção entre a quantidade de rajadas e o sucesso na *build*, ou seja, verificamos a correlação entre rajadas e o sucesso na *build*.

Com o intuito de responder à questão de pesquisa, efetuamos a mineração de um repositório de software. O estudo é baseado nos seguintes artefatos de software: nome do projeto, identificador da *build*, status da *build*, e o identificador dos *commits*. O nome do projeto é fundamental para associar com os demais artefatos, o identificador da *build* é um registro de compilação fornecido pelo *Tavis IC*, o status da *build* é um *feedback* da execução de um conjunto de testes automatizados, por fim o identificador dos *commits* sendo um registro das alterações e informações das mudanças realizadas no código, uma fonte de dados importante no contexto de mineração de repositórios. Por meio desses artefatos podemos extrair as informações necessárias para analisar a associação entre rajadas e o status da *build*.

Dessa forma, escolhemos projetos *open source* em virtude da disponibilidade de seus artefatos. Com base na literatura utilizamos um algortimo para a detecção automática das rajadas, bem como usamos uma ferramenta para associar o status da *build* com o registro de alterações do software (*commits*). Desse modo, realizamos a extração desses dados e, então, efetuamos a análise de ocorrência entre as rajadas e builds.

Para realização deste estudo, foi necessário efetuar a seleção de um repositório, uma ferramenta de apoio, a execução do estudo, um algoritmo para detectar as rajadas, escolha do nível, e, por fim efetuamos a análise. As subseções seguintes apresentam uma descrição detalhada das etapas mencionadas.

3.1.1 Seleção dos repositórios de software

O repositório *TravisTorrent* foi escolhido para este estudo por disponibilizar acesso conveniente a seus conjuntos de dados arquivados e recursos analíticos gratuitos: Permite acesso diretamente no formato *SQL* no navegador para executar as consultas na infraestrutura e baixar dumps SQL ou o conjunto de dados compactado como um arquivo CSV (1,8 GB descompactado). Ele também fornece documentação e um tutorial de introdução. Todas as ferramentas e dados do *TravisTorrent* estão no domínio público.

O vínculo desse repositório com integração contínua (Travis CI) possibilita a manipulação dos dados necessários para este estudo. Segundo Beller, Gousios e Zaidman (2017), o *Travis IC* tornou-se a plataforma de IC mais utilizada para o desenvolvimento de *software* de código aberto.

No conjunto de dados final foram identificados quatro tipos de resultados de *build*: passou, falhou, cancelou e erros (VASILESCU et al., 2014). Consideramos apenas o resultado de compilação falhou e passou, pois são suficientes para responder ao estudo proposto. Os resultados de compilação restantes foram desconsiderados durante o estudo. A Tabela 3.1 detalha as informações coletadas do *data set*.

Tabela 3.1 Resumo do *Data Set*

Número de projetos	1.179
Número de commits	35.060.655
Numero de buids	365.417
builds que falharam	74.496
builds que passaram	258.108
builds canceladas	722
builds com erro	32.091

3.1.2 Seleção de uma Ferramenta de Apoio

As seguintes plataformas foram utilizadas para execução desse estudo e atualmente estão disponíveis.

R[3] é uma linguagem para computação estatística que permite, por meio de pacotes externos, a agregação de outras funcionalidades como as de mineração de dados.

RStudio[4] é um software livre que é usado como ambiente de desenvolvimento integrado para R.

O R foi escolhido por possuir a biblioteca *bursts*[5], uma implementação do algoritmo de *Kleinberg*. Além disso, o R possibilita a automatização da manipulação dos dados, pois, é uma linguagem de programação para cálculos estatísticos e gráficos, sendo adequada para a execução do nosso estudo.

[3]https://cran.r-project.org/

[4]https://www.rstudio.com/

[5]https://cran.r-project.org/web/packages/bursts/index.html

3.1.3 Execução do Estudo

Para a execução deste estudo, utilizaremos a revisão instantânea do *travistorrent_11_1_2017.csv.gz* de um grande conjunto de dados preparado por Beller, Gousios e Zaidman (2017). Esse conjunto de dados consiste em informações sobre o histórico de mudanças e *builds* de 1.179 projetos de *software* desenvolvidos em *Java e Ruby*. Todas essas informações são coletadas de duas fontes diferentes:

- *Travis CI* - uma ferramenta CI;
- *GitHub* - uma plataforma de colaboração.

Então, essas informações coletadas são combinadas para preparar o conjunto de dados ou seja, a combinação de um ambiente de IC simplificado, popular e fortemente acoplado (TRAVIS CI) ao sistema de controle de versão (GIT) e à plataforma de colaboração (GITHUB) permitiu agregar os dados disponíveis em um único conjunto de dados. À vista disso, definimos alguns critérios para realização seleção de dados. Projetos que tinham mais que quarenta e oito semanas e entre cem e mil *commits*, sendo essa escolha baseada nos testes que fizemos ao, observarmos que a ocorrências de rajadas, em projetos com menos de 48 semanas e abaixo de 100 *commits*, não apresentavam rajadas suficientes em distribuição de níveis para nossa análise, uma vez que definimos trabalhar com o algoritmo de *kleinberg* que detecta as rajadas em níveis.

Tabela 3.2 Projetos selecinados

Projetos	Tempo>48 semanas	commits >100 e <1000
1136	647	647

3.1.4 Seleção do Algoritmo para Detectar Rajadas

Tamura e Kitakami (2014) afirmam que o algoritmo de detecção de rajadas de *Kleinberg* é o algoritmo mais bem sucedido para se detectar períodos de rajadas relacionados a uma palavra-chave, tópicos ou eventos. O algoritmo de detecção de rajadas temporal busca encontrar certos períodos de tempo em que uma palavra-chave ocorre em alta frequência.

Após a execução do algoritmo *Kleinberg* no projeto *Rails*, disponível no TravisTorrent, obteve-se a detecção de quatro níveis de rajadas conforme a Figura 3.2. Os níveis são características fornecidas pelo algoritmo, cuja probabilidade, no primeiro nível, de alguma frequência ser classificada como rajada é muito alta e, dessa forma, toda a cadeia de eventos tende a apresentar tal classificação (KLEINBERG, 2003).

Observa-se que o eixo x apresenta a linha do tempo em que essas rajadas acontecem, cada traço horizontal representa a duração de uma rajada; e no eixo y os níveis que as rajadas são classificadas. Para realizar a classificação das rajadas utilizou-se a associação entre as seguintes variáveis disponíveis no *dataset*, apresentadas na Tabela 3.3.

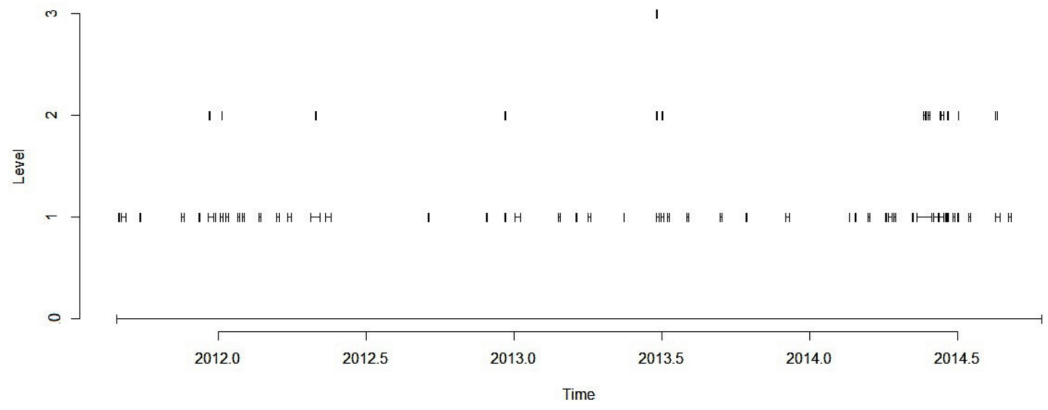


Figura 3.2 Detecção de rajadas de mudanças no projeto Rails

Tabela 3.3 Variáveis utilizadas no estudo

Variáveis	**Descrição**
gh_project_name	Nome do projeto no GitHub
tr_build_id	O ID de compilação analisado, conforme relatado pela Travis CI.
tr_status	O status da compilação (passou, falhou, erro, cancelou) retornado da API *Travis CI*.
gh_build_started_at do projeto que disparou a *build*	Horário em que ocorreu o *commit*

3.1.5 Escolha do nível

Os níveis encontrados pelo algoritmo são classificados de acordo com a densidade dos *commits*, ou seja, à medida que o número do nível aumenta, a densidade tende aumentar (KLEINBERG, 2003).

Vale ressaltar que, no primeiro nível a probabilidade de alguma frequência ser classificada como rajada é muito alta e, dessa forma, toda a cadeia de eventos tende a apresentar tal classificação (KLEINBERG, 2003). À medida que os níveis aumentam, a probabilidade de detecção de rajadas diminui e se encerra quando nenhuma frequência de eventos for classificada como uma rajada, conforme apresentado na Figura 3.2. Para nosso estudo a seleção dos níveis foi definida de acordo com alguns testes realizados. No primeiro nível(0) o algoritmo classifica tudo como rajada e por isso não foi escolhido; ao analisarmos os níveis acima de 3, percebemos que existem poucas rajadas e muitos projetos não teriam tais níveis. Sendo assim, nossa escolha foi baseada em testes e conveniência para

efetuar a análise. A Tabela 3.2 mostra a quantidade de projetos, o período e a quantidade de commits estabelecida. Posteriormente, tabulamos essas informações seguindo os critérios.

3.1.6 Procedimentos das Análises

Para responder à QP1, analisamos a proporção de builds bem sucedidas tanto em commits que fazem parte de rajadas quanto em commits que não o fazem. Para verificar a significância desses resultados, tabulamos os dados de todos os projetos avaliados, conforme a tabela[6], plotamos histogramas a fim de verifícar a distribuição dos dados e utilizamos o teste Qui Quadrado (X^2). O teste de independência Qui-Quadrado é usado para descobrir se existe uma associação entre a variável de linha e a variável coluna em uma tabela de contingência construída a partir de dados de uma amostra. Para isso, duas hipóteses foram levantadas: a nula e alternativa. A primeira significa que as variáveis não estão associadas, em outras palavras, elas são independentes. A segunda significa que as variáveis estão associadas, ou dependentes. Uma vez verificada as diferenças entre as proporções de rajadas e o sucesso das *builds*, calculamos o tamanho do efeito, utilizando o V de Cramer, e a interpretação dos resultados foi baseada em Souza e Silva (2017 apud COHEN, 1988, p. 2) em que $V < 0,3$ é considerado como efeito pequeno, $V < 0,5$ considerado como efeito médio e $V > 0,5$ é considerado o tamanho de efeito grande. A próxima secção descreverá os resultados deste estudo.

3.2 RESULTADOS

3.3 ANÁLISE DOS RESULTADOS

Através da realização deste estudo é possível averiguar e interpretar os dados para atingir a finalidade da pesquisa, para, assim, responder à seguinte questão de pesquisa:

QP1: *Builds* que ocorrem após rajadas de mudanças possuem maior propensão a terem falhas?

Selecionamos uma amostra de 647 projetos, analisamos a distribuição da quantidade de rajadas por projeto e associamos com o sucesso da *build* para os níveis 1, 2 e 3 de rajadas.

Para melhor visualização, os dados serão apresentados por meio de histograma e gráfico de setores.

Em todos os níveis foi feito o mesmo tratamento. O teste qui-quadrado foi usado para verificar as seguintes hipóteses:

- H0: Não existe diferença entre as proporções de sucesso em *builds* após rajada de mudanças;
- H1: Existe diferença entre as proporções de sucesso em *builds* após rajada de mudanças.

[6]https://github.com/JuvenalJr/Mestrado

Assim, dividimos os projetos entre significativos (com valor de $p < 0,05$) e não significativos (com valor de $p >= 0,05$). Em seguida medimos o tamanho do efeito com o V de Cramer dos projetos significativos.

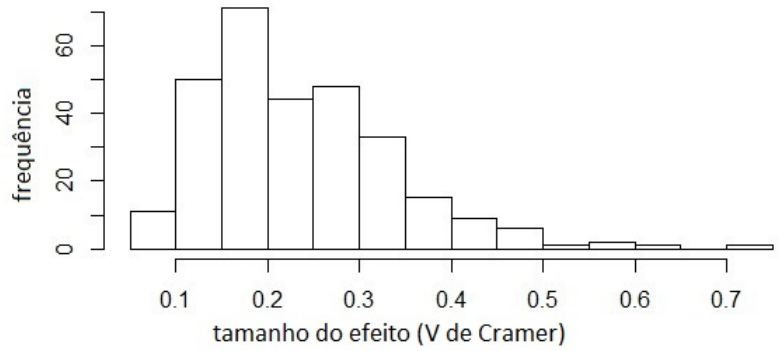


Figura 3.3 Histograma dos projetos significativos nível 1

Nível 1 Observando a Figura 3.3, podemos perceber que o histograma não possui simetria no valor do efeito nos projetos considerados significativos. Portanto, a partir da análise descritiva dos dados, observamos que a influência das rajadas para projetos com efeito grande (com valor de $V > 0,5$) ocorreu na minoria dos projetos.

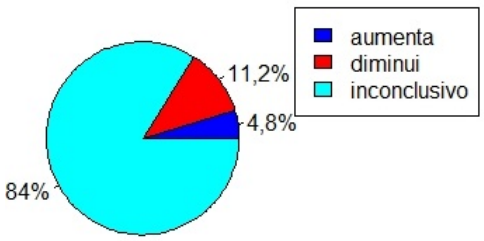


Figura 3.4 Tendência de sucesso da *build* em projetos com rajadas nível 1

A Figura 3.4 mostra a tendência da taxa de sucesso na *build* em relação aos 647 projetos, sendo que, para os considerados estatisticamente significativos, a taxa de sucesso tende a diminuir (setor em vermelho) ou aumentar (setor em azul escuro) quando ocorrer rajadas. Para o nível 1, identificamos que 16% dos projetos da amostra selecionada eram significativos; 11,2% tendem a diminuir a taxa de sucesso da *build* quando ocorre rajadas e apenas 4,8% tende aumentar a taxa de sucesso da *build*. Entretanto, 84% dos projetos no nível 1 não foram significativos.

Dessa forma, não podemos rejeitar H0, pois apesar da proporção de sucesso após rajadas não ser a mesma nos projetos significativos, este número representa apenas 16% do

número total de projetos no nível 1, sendo 84% projetos inconclusivos (não são significativos).

Nível 2 Considerando o mesmo procedimento aplicado ao nível anterior, a Figura 3.5 mostra o histograma dos projetos no nível 2.

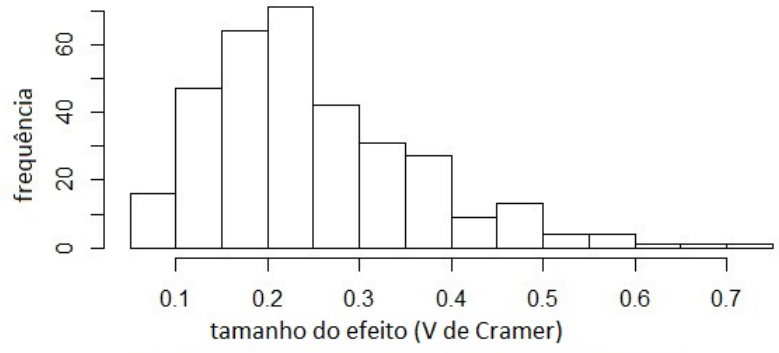


Figura 3.5 Histograma dos projetos significativos com valor no efeito no nível 2

A Figura 3.5, de forma semelhante ao anterior, não apresenta uma simetria na distribuição dos dados. Percebe-se que, para a maioria dos projetos significativos, o tamanho do efeito é pequeno (com valor de $V < 0,3$).

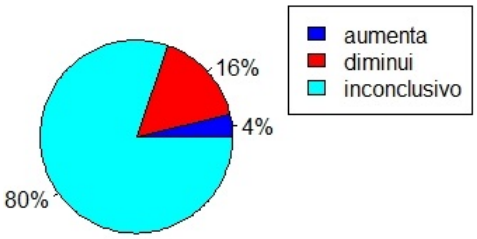


Figura 3.6 Tendência do sucesso da *build* em projetos com rajadas nível 2

A Figura 3.6 mostra a tendência da taxa de sucesso na *build* em relação aos número total de projetos. De forma semelhante ao nível 1, quando ocorre rajadas nos projetos significativos, a taxa de sucesso diminui (setor em vermelho) para 16% dos projetos e a taxa de sucesso aumenta (setor em azul escuro) para 4% dos projetos. Contudo, 80% dos projetos no nível 2 não foram significativos.

Logo, no nível 2 também não podemos rejeitar H0, pois apesar da proporção de sucesso após rajadas não ser a mesma proporção nos projetos significativos, 80% dos projetos não são significativos.

Nível 3 No último nível, obtemos um número maior de projetos que foram considerados significativos (23%), mas a quantidade de projetos não significativos continua sendo a maioria (77%). As Figuras 3.7 e 3.8 descrevem os resultados para o nível 3.

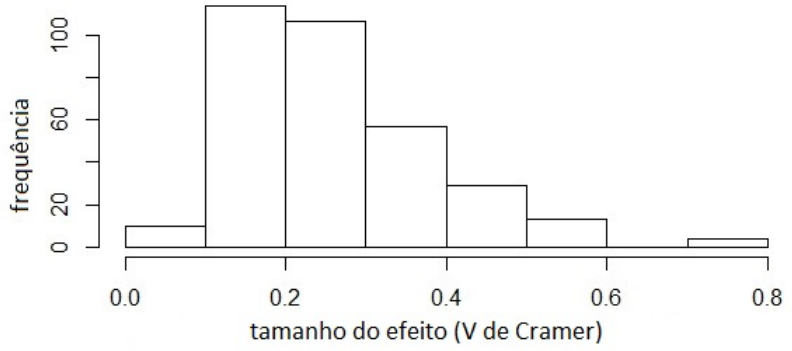


Figura 3.7 Histograma dos projetos significativos com valor no efeito no nível 3

De forma semelhante aos níveis anteriores, o histograma apresentado na Figura 3.7 mostra assimetria dos dados e o gráfico de setores mostra que a tendência da taxa de sucesso na *build* em relação aos projetos significativos tende a diminuir ou aumentar quando ocorrer rajadas. Observa-se na Figura 3.7 para a maioria dos projetos que o tamanho do efeito classificou-se entre pequeno (com $V < 0,3$) e médio (com $V < 0,5$)

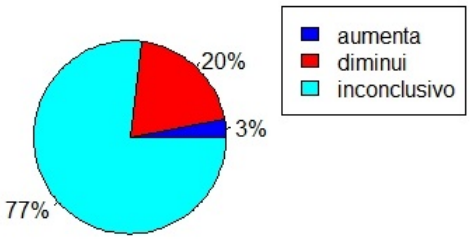


Figura 3.8 Tendência do sucesso da *build* em projetos com rajadas nível 3

Dos 23% dos projetos significativos para o nível 3, 20% tende a diminuir (setor em vermelho) a taxa de sucesso na *build* quando ocorre rajada enquanto apenas 3% tende aumentar (setor em azul escuro). Mesmo assim, não podemos rejeitar H0, já que, apesar da proporção do sucesso de *builds* com rajada não ser a mesma, o número de projetos inconclusivos continuou em destaque no nível 3 (77% dos projetos).

Em suma, podemos concluir que, nos três níveis de rajadas, dentre os projetos em que a diferença é estatisticamente significativa, builds possuem maior tendência a falhar quando fazem parte de uma rajada. Na maior parte dos projetos, no entanto, a associação entre status da *build* e rajada não é estatisticamente significativa. A próxima secção descreve as ameaças a validade deste estudo.

3.4 AMEAÇAS À VALIDADE

Nesta seção são discutidas as limitações das análises realizadas. Nesse sentido, listamos, a seguir, as ameaças à validade levantadas e as medidas tomadas para reduzi-las.

Validade Interna: Este estudo analisou apenas uma variável independente, a ocorrência de rajadas de mudança. É possível que essa variável seja influenciada por outros fatores que não foram analisados, como estágio de um projeto no ciclo de vida do software. Para mitigar essa ameaça, analisamos centenas de projetos de diferentes ciclos de vida de um período amplo.

Validade Externa: Uma ameaça externa trata-se da generalização do estudo. Os projetos estudados são todos em Java e Ruby de código aberto, portanto não podemos generalizar os resultados para estudos em outras linguagens de programação. No entanto, os projetos disponíveis têm equipes de tamanhos diferentes, domínios e tamanhos diferentes. Além disso, este estudo pode ser replicado para outros projetos que utilizem IC.

Validade de Construto: Uma ameaça à validade de construto do estudo é a escolha do algoritmo para detecção das rajadas nos projetos. Para tentar mitigar essa ameaça, foi utilizado o Kleinberg, que é um algoritmo clássico para detecção de rajadas. Além disso, foram utilizados 3 níveis de cada projeto para minimizar o risco da escolha de apenas um nível, que poderia não ser adequado. A próxima subseção apresenta a conclusão do estudo deste capitulo.

Validade de Conclusão: Usamos diversos testes estatísticos para ter um embasamento mais consistente em nossas conclusões sobre as informações extraídas dos dados minerados dos projetos de interesse. Em determinadas situações os testes estatísticos podem falhar em rejeitar ou aceitar a hipótese nula devido a diversos fatores como violação de requisitos, alto valor de significância, tamanho da amostra insuficiente. Para garantir a credibilidade dos testes utilizados neste estudo, selecionamos projetos que continham entre cem e mil commits, o que permitiu o algoritmo classificar as rajadas.

3.4.1 Conclusão

Neste capítulo, estudamos a relação entre modificações de código criadas pelos desenvolvedores em um curto período de tempo e falhas na *build*. Ao melhor do nosso conhecimento, este é o primeiro estudo que analisa empiricamente a associação entre rajadas de mudança e falhas na *build*. Para realização do estudo, foi analisado um conjunto de dados compostos por 1.136 projetos de código aberto e integrados com GitHub, extraídos do repositório *TravisTorrent*. Através da mineração desses dados, foi respondida uma questão de pesquisa sobre a propensão de falhas em *builds* que ocorrem após essas rajadas de mudanças.

Nossos resultados mostram que para os 3 níveis avaliados em projetos significativos, a

taxa de sucesso de *builds* após rajadas de mudanças tendem a diminuir. No entanto, não podemos generalizar os resultados para todos os projetos do TravisTorrent que utilizam integração contínua, visto que a quantidade de projetos significativos foi relativamente pequena comparado com os projetos que não foram significativos (inconclusivos). Diante disso, diferentemente do estudo de Nagappan (NAGAPPAN et al., 2010) em que rajadas de mudanças indicaram mais erros durante o desenvolvimento de software, em nosso estudo, não foi possível mostrar uma associação significativa entre falhas na *build* e ocorrência de rajadas de mudanças.

Logo, este trabalho contribuiu para a comunidade de desenvolvedores com uma análise sobre os dados do TravisTorrent. Além disso, concluímos que, visando a qualidade do código, desenvolvedores podem ser mais cautelosos na frequência de modificações realizadas no código em um curto período de tempo, porém não é o fator principal para gerar falhas na *build*. O próximo capítulo complementará este estudo.

Capítulo 4

AVALIANDO A ASSOCIAÇÃO ENTRE RAJADAS DE COMMITS E STATUS DA BUILD SOB A PESPECTIVA DE DESENVOLVEDORES

Diante da ausência de estudos sobre a associação das rajadas de commits com o status da *build*, o objetivo deste capítulo é complementar o anterior, visto que, no primeiro estudo, exploramos um dataset e, neste, buscou-se entender, junto aos desenvolvedores, a percepção deles em relação às rajadas de commits e à falha da *build*. Para melhor compreensão, a próxima seção apresenta o design do estudo realizado.

4.1 DESIGN DO ESTUDO

Nesta subseção, encontra-se o design deste estudo. Inicialmente, apresenta-se uma visão geral. Posteriormente, há o detalhamento dos tópicos na seguinte ordem: as subquestões de pesquisa; o questionário e o teste piloto aplicado; a descrição dos dados coletados e como estes foram analisados; a apresentação dos resultados relacionando-os com as subquestões de pesquisa que norteiam este capitulo; e, por fim, são descritas as ameaças à validade e as conclusões.

4.1.1 Visão Geral

Neste estudo, um *survey* foi construído para coletar dados de desenvolvedores e de participantes em geral com experiência em Integração Contínua. O processo de construção deste *survey* seguiu as ideias apresentadas por Kitchenham e Pfleeger (2002a, 2002b). Enquanto o primeiro trabalho (KITCHENHAM; PFLEEGER, 2002a) orientou o caráter do estudo aqui descrito, o segundo (KITCHENHAM; PFLEEGER, 2002b) conduziu a construção do questionário utilizado neste *survey*. Esse estudo é de natureza exploratória, pois deseja investigar se builds que ocorrem após rajadas de mudanças possuem maior propensão a terem falhas e as causas das rajadas de commits na perspectiva dos desenvolvedores. Esse método foi escolhido, por ser largamente utilizado em estudos empíricos (PUNTER et al., 2003). Além da validação ser importante por parte de pesquisadores e desenvolvedores, podem surgir potenciais consumidores dos resultados obtidos.

4.1.2 Subquestões de Pesquisa

Com o objetivo de analisar nossa questão de pesquisa: **QP1: Builds que ocorrem após rajadas de mudanças possuem maior propensão a terem falhas?** Formulamos as seguintes subquestões de pesquisa:

QP1.1: Qual a percepção dos desenvolvedores sobre a causa das rajadas de commits?

Nesta subquestão foram apresentadas as causas que corroborariam para a existência de rajadas de commits. Para isso, duas questões do survey destinam-se ao questionamento da causa das rajadas de commits.

QP1.2: Qual a percepção dos desenvolvedores sobre influência da rajada de commits na falha da *build*?

Nesta subquestão, quatro questões foram apresentadas no survey aos desenvolvedores, para compreender a relação da falha da *build* com rajadas de *commits*. A seguir, há o detalhamento da construção do survey.

4.1.3 Questionário

A pesquisa tem 6 seções principais com um total de 14 perguntas, sendo 10 questões (72%) obrigatórias e as demais opcionais.

A seção I do Survey consiste na apresentação da pesquisa. A seção II apresenta o termo de consentimento livre e esclarecido. A seção III descreve as definições para compreensão da pesquisa. A seção IV define se o participante trabalha ou não com integração contínua. A seção V refere-se à identificação do participante, e possui seis questões (4 questões de múltipla escolha, 2 questões subjetivas). Essas perguntas têm como objetivo identificar o perfil do participante, incluindo informações como nome, nível de escolaridade, experiência, área que atua.

A seção VI define as questões sobre rajada de *commits* e *build*. Essa seção tem como objetivo compreender a perspectiva dos desenvolvedores sobre esse assunto. O Apêndice A contém *printscreens* do formulário. A próxima subseção apresenta o processo do teste piloto.

4.1.4 Pesquisa de Teste Piloto

O estudo piloto foi realizado com 3 participantes, os quais trabalham ou trabalharam com Integração Contínua. Para a versão piloto, um formulário foi criado a fim de contribuir com a estrutura da pesquisa e indicar sugestões de melhorias. O formulário continha 6 questões relacionadas com a adequação das questões sobre a pesquisa, como tópico, o número de perguntas a serem respondidas, o tempo necessário para responder o formulário e a relevância da pesquisa.

Para as perguntas 1 e 2, considerou-se um intervalo de 5 pontos (escala de Likert), em que 1 refere-se à ruim e 5 refere-se à a ótimo. As questões foram estruturadas da seguinte forma: (1) Como você avalia o número de perguntas no formulário? (2) Como você avalia a adequação das perguntas do formulário ao tema "Rajadas de commits e builds"? (3) Quanto tempo você gastou para responder todo o questionário? (4) Alguma

sugestão para melhorias (5) Você poderia citar os pontos positivos da Pesquisa? e (6) Gostaria de prover comentários adicionais sobre o questionário?

O feedback sugeriu pequenas modificações na estrutura da pesquisa. As alterações incluíram adicionar opções de resposta a algumas perguntas, remover perguntas, alterar palavras para melhorar a compreensão do participante e destacar o objetivo da pergunta. Com base nos problemas identificados, foi gerada uma versão melhorada do instrumento de pesquisa.

4.1.5 Coleta dos Dados

Para desenvolvimento da pesquisa, adotou-se a ferramenta online Google Forms. Essa ferramenta foi escolhida por facilitar a coleta, pois, além de ser acessível via web, facilita a extração dos dados em formato de dados conhecidos.

O formulário ficou disponível por cerca de dois meses, no segundo semestre de 2021. O link URL para pesquisa foi distribuído por email (contatos individuais, grupo de pesquisa) e redes sociais, como LinkedIn, Facebook e WhatsApp. Potenciais participantes foram informados sobre as políticas de privacidade do estudo de forma clara e detalhada.

4.1.6 Análise

Para análise das subquestões já descritas, aplicou-se a análise descritiva e qualitativa dos dados. Para as perguntas abertas sendo elas curtas ou longas, foram empregadas técnicas da Teoria Fundamentada nos Dados (CHARMAZ, 2006; CORBIN; STRAUSS, 2008) para a codificação inicial e axial.

Utilizou-se a ferramenta Delve[1] um software que analisa dados qualitativos, por se tratar de uma ferramenta online e colaborativa para auxiliar na análise dos dados. De acordo com os resultados apresentados, os dados foram comparados entre os diferentes perfis dos respondentes. No processo da codificação inicial, foram identificadas as expressões que respondem ao questionamento realizado (STRAUSS; CORBIN, 1990), analisando-se palavra por palavra, linha por linha. Já na codificação axial busca-se identificar os códigos iniciais mais significativos. Nesse processo, os códigos são relacionados de forma a observar se eles podem ser agrupados e abstraídos em categorias.

Os códigos encontrados foram agrupados de acordo com as suas propriedades, formando, assim, conceitos que representam categorias. Essas categorias foram analisadas em conjunto com outros pesquisadores, com o objetivo de proporcionar uma maior clareza sobre o fenômeno em questão. Finalmente, as categorias foram relacionadas entre si. Na prática, os passos da codificação inicial e axial se sobrepõem e se unem, devido à interatividade do processo. Os códigos e categorias identificados passaram por sucessivas revisões, sendo que, ao final da presente versão da análise, foram produzidos 45 códigos associados a 8 categorias.

[1] https://delvetool.com/

4.2 RESULTADOS

4.2.1 Caracterização dos Participantes

A pesquisa recebeu 78 respostas, sendo que 18 delas foram descartados, pois 14 não tinham qualquer experiência anterior com ou trabalha com IC, os outros 3 participaram do piloto, e um respondeu à pesquisa mais de uma vez. No final, tivemos 60 respostas, sendo 57 respondentes do Brasil e três dos EUA.

Em relação ao nível de escolaridade, 51% possuíam curso Superior, 13% fizeram Pós-Graduação lato sensu (Especialização), 25% Pós-Graduação stricto sensu (Mestrado), 5% Pós-Graduação stricto sensu (Doutorado), 4% Curso Superior incompleto, apenas 2% Ensino médio, conforme a Figura 4.1.

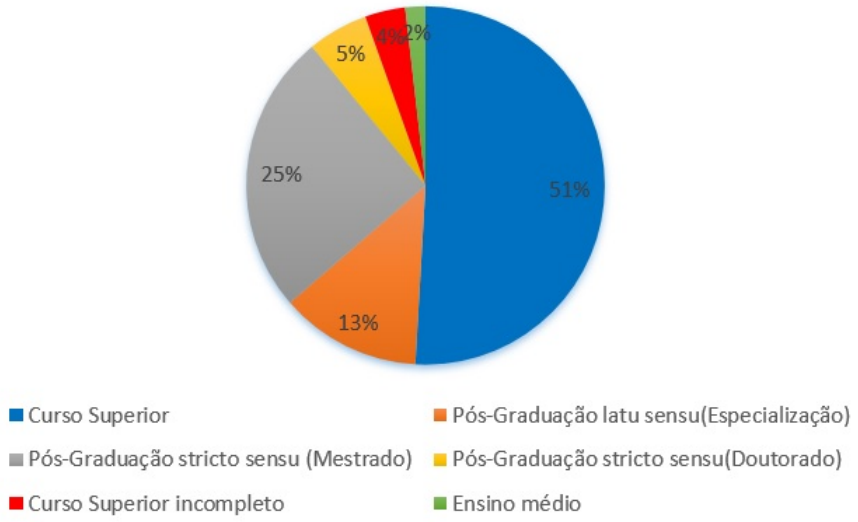


Figura 4.1 Titulação máxima declarada pelos participantes deste estudo

Em relação às áreas em que os participantes atuam, são: desenvolvedor, gerente técnico, devops, arquiteto de software, engenheiro de software, analista de negócios, gerente de projetos, analista de qualidade, engenheiro de teste e líder técnico. Percebe-se que os profissionais não atuam só como desenvolvedores de software, mas em vários papéis.

No tocante à experiência relatada entre os participantes, elaborou-se uma escala intervalar, com 5 pontos: de 0 a 1 ano, de 1 a 2 anos, de 3 a 4 anos, de 4 a 5 anos e por fim mais de 5 anos de experiência. Logo, 44,4% dos participantes possuíam mais de 5 anos de experiência, conforme a Figura 4.2.

Em relação ao serviço de hospedagem de repositório utilizado, a maioria dos participantes 75% (41) selecionou o GitHub, embora outros repositórios foram mencionados, tais como: BitBucket – git e mercurial 33,3% (18), e apenas um participante declarou não trabalhar com repositório. Diante disso, percebe-se que a maioria trabalha ou já trabalhou com IC e utilizaram um serviço de hospedagem de repositórios.

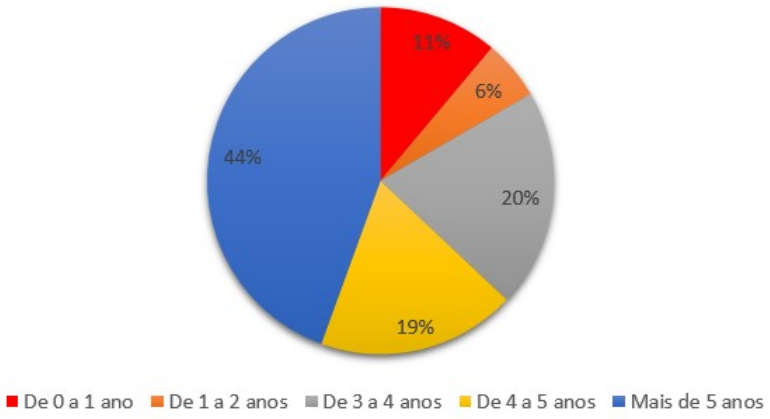


Figura 4.2 Experiência máxima declarada pelos participantes deste estudo

4.2.2 QP1.1: Qual a percepção dos desenvolvedores sobre a causa das rajadas de commits?

Para responder a essa questão, os desenvolvedores relataram suas opiniões com base em suas experiências. Desse modo, obtivemos três grupos diferentes: no primeiro grupo, dezenove participantes (de um total de 47) concordaram que a proximidade de uma data de entrega do projeto é um dos fatores responsáveis pela rajada de commits. O Pesquisador 06 relatou o seguinte:

> "Considerando, como boa prática, que cada commit deve exercer apenas 1 objetivo e que clientes tendem a querer o produto no mínimo tempo possível, o prazo geralmente fica corrido para a entrega, consequentemente tende a acarretar em mais commits nesse período para bater a meta."

O segundo grupo acredita que a correção de bugs é responsável pelas rajadas de commits. Assim, um total de dezesseis participantes relataram que a correção de bugs contribui para a rajadas de commits, sendo relatado pelo desenvolvedor 03 que: "A correção de bugs pode gerar mais commits, especialmente quando a quantidade de bugs é grande."

Já no terceiro grupo os desenvolvedores não pontuaram um único causador, mas classificaram que depende de vários fatores, como descrito pelo desenvolvedor 11: "Rajadas de commits depende de diversos fatores, não é muito fácil definir apenas um causador."As demais causas como: testes incompletos, número de contribuintes, experiência do desenvolvedor, requisitos não são definidos, maturidade do time e processos da empresa foram citados pelo grupo.

O desenvolvedor 17 relatou que "Durante uma rajada de commits, é mais provável que ocorra alguma falha no processo, visto que, por vezes, alguns desses commits acabam não passando por todos os processos de testes, como, por exemplo, um peer review".

Já o desenvolvedor 21 descreveu que:

> "A quantidade de contribuintes e diversos níveis de experiência pode acarretar em liberações de bugs que precisem ser corrigidos, seja por falhas na comunicação, erros naturais do dia a dia etc. E levando em conta que contribuintes com menos experiência naturalmente levariam mais tempo (mais commits) para subir suas implementações".

Enquanto o desenvolvedor 03 relatou que: " Quando os requisitos não são bem definidos, a quantidade de alterações no código ou implementação de pequenas funcionalidades geralmente aumenta, o que gera mais commits num curto intervalo de tempo."

O desenvolvedor 32 descreveu o seguinte: "Depende muito da maturidade do time e dos processos da empresa".

Após os relatos, percebe-se que existem diversas opiniões em relação a um causador das rajadas de commits. Diante disso, ainda que o maior grupo acredite que a proximidade da data de entrega do projeto seja o responsável pelas rajadas, não se pode generalizar.

Para melhor compreensão do que foi descrito, vejamos a Figura 4.3 que representa o resumo e resultado da codificação realizados para esta questão.

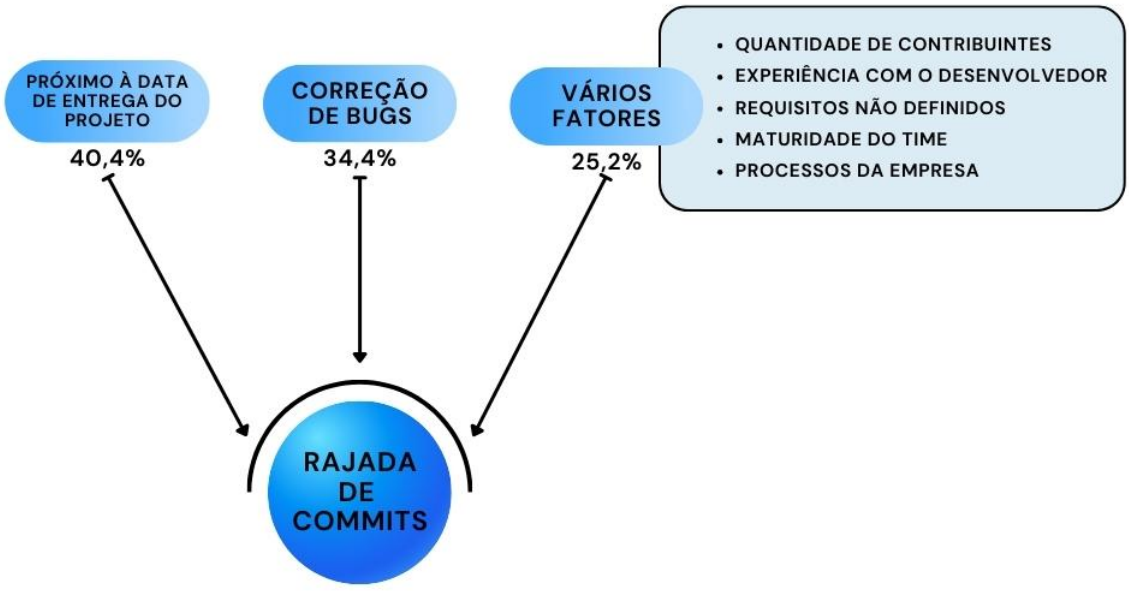


Figura 4.3 Resumo da codificação axial

4.2.3 QP1.2: Qual a percepção dos desenvolvedores sobre influência da rajada de commits na falha da build?

Conforme detalhado na seção de design deste estudo, a sexta seção do questionário relacionado dedica-se a coletar, junto aos participantes, as experiências vivenciadas para a compreensão das rajadas de commits e a falha na *build*. Para responder a essa QP, utilizou-se duas questões do survey: na primeira perguntamos aos desenvolvedores se concordariam com a seguinte afirmação: **"Um commit que faz parte de uma rajada de commits é mais propenso a resultar em falha da *build* do que um commit que não faz parte de uma rajada?"**. Como os participantes foram convidados a justificar a resposta do questionário, ao analisar as respostas, encontrou-se três grupos de participantes.

O primeiro, quatorze participantes (de um total de 32) apontaram que a falha na *build* está associada a *commits* que fazem parte de rajadas de *commits*. O Pesquisador 33 relatou em sua resposta o porquê dessa afirmativa: "Um commit de rajada de commits naturalmente tende a sofrer um processo de code review menos rígido (uma vez que os revisores vão ter mais trabalho e menos tempo de revisar todo o código)".

No segundo grupo, três participantes (de um total de 32) discordaram, apontaram que um *commits* que faz parte da rajada de *commits* não influencia na falha da *build*. O participante 12 relatou que: "um commit errado pode acontecer a qualquer momento e apenas um corrigir". E por fim, no terceiro grupo as respostas foram diversificadas, relatando possíveis causas que contribuiriam para a falha ocorrer, como a resposta do desenvolvedor 18:

> "Não é possível obter uma regra a partir da afirmativa. Outros fatores devem ser levados em conta além da mera contagem de commits: nível de experiência de quem realiza o commit, qualidade percebida dos efeitos do commit ao longo do tempo, trata-se de uma refatoração realizada de forma controlada".

As demais causas registradas foram: testes incompletos, commit incompleto, vários commits, política de commits e revisão do código. Segundo o desenvolvedor 32: "Depende muito da politica de commits estabelecidos e revisão do código"

A segunda pergunta realizada para ajudar a compreensão foi: **"Suponha que uma *build* tenha falhado; após essa falha você acredita que é mais provável ocorrer uma rajada de commits?"** Após as análises identificou-se três grupos, o primeiro grupo (12 do total de 32) acredita que, após a falha da *build*, ocorrerá rajadas de commits. Como descreve o desenvolvedor 18: "Com minha pouca experiência é comum falhar a *build*, significa que tem algo ali pra corrigir. As rajadas de commits ocorrem quando algum ou vários problemas são liberados sem quebrar a *build*, daí é commit atrás de commit."

O segundo grupo (6 de um total de 32) discorda do primeiro, acredita que após a falha da *build* não ocorrerá uma rajada de commits para solucionar, veja a resposta do desenvolvedor 30: "No que tenho observado no meu contexto, uma quebra de *build* geralmente requer uma mudança simples solucionável em uma *build* (correção de arquivos de configuração, correção de teste unitário, etc)". Já o terceiro grupo (14 do total de 32)

descreveram que depende de vários fatores como relatou o desenvolvedor 10: “Em alguns casos, se a falha for técnica (não depende de commit, mas de infra), não haverá commits. Além disso, o próprio sistema do repositório (CI/CD) bem configurado manterá o *build* anterior em funcionamento”.

As demais causas registradas foram: processos da empresa, requisitos mal definidos, teste mal realizados. Segundo o desenvolvedor 8: “Depende muito dos testes e revisão do código serem bem feitos”

Apesar de ser identificado que a proximidade da data de entrega contribui para que ocorram as rajadas de commits, e que tais rajadas têm relação com a falha na *build*, em contrapartida outro grupo não acredita que a falha na *build* tenha associação com as rajadas de commits, em geral, os desenvolvedores citaram outras causas que provocam as rajadas de commits e a falha na *build*. Diante disso, não se pode afirmar que existe apenas um fator responsável pela rajada e falha na *build*, como pode ser observado na Figura 4.4.

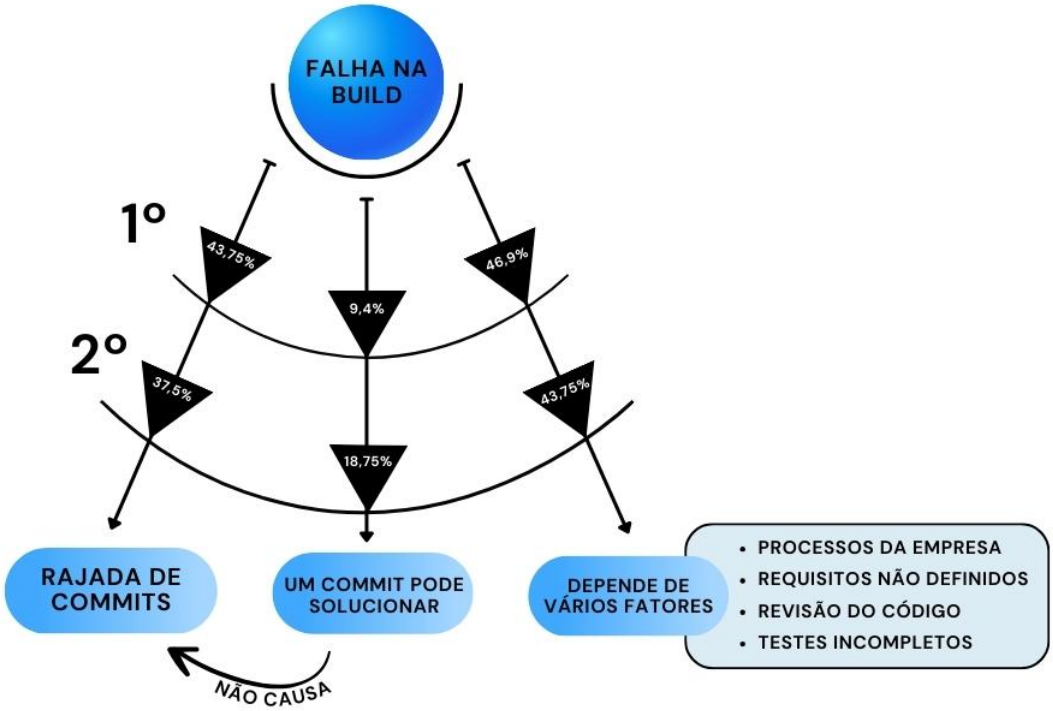


Figura 4.4 Resumo da codificação axial

A próxima seção apresenta as ameaças à validade deste estudo.

4.3 AMEAÇAS À VALIDADE

Apresenta-se as ameaças à validade deste estudo e as medidas que foram tomadas para reduzir o impacto conforme descritas a seguir.

Validade de Construto No questionário aplicou-se conceitos quanto à definição de rajadas de commits e *build* com o objetivo de facilitar a compreensão dos participantes quanto ao que foi perguntado no questionário. No entanto, pode haver dificuldades no entendimento da questão.

Para diminuir o viés causado pela falta de compreensão, restrigiu-se, a participação a quem tivesse experiência profissional com IC, visto que seria suficiente para responder as perguntas e, assim, atingir os objetivos da pesquisa. Para garantir que os participantes compreendessem exatamente o que estava sendo perguntado em cada pergunta do questionário, foi realizada uma aplicação piloto com dois desenvolvedores e um pesquisador em que um dos aspectos observados foi a compreensão das perguntas. Como resultado dessa aplicação piloto, duas perguntas foram reformuladas.

Validade de Conclusão Durante a análise, é preciso ter atenção com a codificação das respostas oferecidas pelos participantes, para garantir que os dados e as análises sejam as mesmas, independentemente dos pesquisadores que a realizarem.

Para tratar essa ameaça, a codificação foi revisada por completo por um pesquisador independente. Esse processo se deu com o autor desta dissertação fazendo a codificação de cada resposta. Paralelamente, um pesquisador independente procedeu a codificação destas respostas. Ao final, os dois pesquisadores se reuniram e fizeram o alinhamento dos códigos, quando, para cada código atribuído de forma divergente, houve nova avaliação e foi realizada uma atividade de consenso.

Validade Externa As ameaças externas estão relacionadas à generalização dos dados obtidos. Para este estudo, foi identificada como ameaça externa o tamanho da amostra obtida, assim como a diversidade dos respondentes. Embora existam participantes de outro país (EUA), a grande maioria dos participantes atua no Brasil. Apesar de atuarem profissionalmente em diferentes organizações/centros e institutos de pesquisa e, de possuírem experiência acumulada diversa entre si, não se pode garantir que os resultados obtidos representem qualquer cenário onde a pesquisa seja replicada.

Validade Interna A primeira ameaça interna que se deve considerar é a seleção de participantes, porque eles foram escolhidos, inicialmente, por meio de amostragem de conveniência. No entanto, um terço dos participantes foi convidado indiretamente, reduzindo essa ameaça. Outra ameaça é que os participantes poderiam ser afetados negativamente pelo tédio e pelo cansaço. Para minimizar essa ameaça, ofereceu-se um formulário online de preenchimento, com duração entre 15 e 20 minutos. Realizou-se, também, um estudo piloto no qual três desenvolvedores responderam a uma versão preliminar do questionário e foram monitorados durante esse processo. A ordenação das seções e a versão final das

perguntas sofreram alterações a partir das lições aprendidas com o estudo piloto.

A seguir, são apresentadas as conclusões deste estudo.

4.4 CONCLUSÕES

Esse capítulo revelou duas importantes contribuições para pesquisadores e desenvolvedores sobre a associação da rajada de commits na falha da *build*. Primeiro, os resultados fornecem informações dos fatores que podem provocar as rajadas de commits e consequentemente a falha da *build*. Um dos fatores mais citados pelos participantes como responsável pela falha da *build* foi **a proximidade da data da entrega de um projeto**. Em segundo lugar, a investigação qualitativa mostrou outras circunstâncias típicas em que uma *build* pode falhar, como o número elevado de colaboradores, correção de bug, testes incompletos, commit incompleto, vários commits, política de commits, revisão do código, maturidade do time, processos da empresa.

Embora não seja possível afirmar que existe apenas um fator responsável pela rajada e a falha na *build*, o objetivo final desse tipo de análise é reduzir a quantidade de builds que falham e, assim, diminuir a perda de tempo. Com base em nossas descobertas, a proximidade da data de entrega de um projeto é um fator acionável que pode ser manipulado para reduzir a falha da *build*.

Capítulo 5

CONCLUSÃO

Neste trabalho, investigamos se mudanças realizadas em uma rajada de mudanças possuem maior propensão a resultar em falhas na build em projetos que adotam a prática de integração contínua. Para isso, analisamos quantitativamente uma amostra de 1.136 projetos de código aberto hospedados no GitHub que usam o Travis CI para integração contínua. Além disso, realizamos um survey com desenvolvedores para entender sua percepção sobre possíveis causas de rajadas de commits e de falhas na build, bem como sobre a relação entre os dois.

Os resultados do primeiro estudo mostram que a taxa de sucesso tende a diminuir após essas rajadas. O estudo foi limitado a uma amostra relativamente pequena de projetos significativos, não permitindo generalizações para todos os projetos que utilizam Integração Contínua.

No survey, os participantes descreveram vários fatores como contribuintes da falha da *build*: a proximidade da data de entrega do projeto, o número de colaboradores, correção de bug, testes incompletos, commit incompleto, vários commits, política de commits, revisão do código, maturidade do time e processos da empresa.

Embora não possa se afirmar que existe apenas um fator responsável pela rajada e a falha na *build*, a proximidade da data de entrega do projeto é um fator acionável que pode ser manipulado para reduzir a falha da *build*.

Diante disso, ainda que no estudo do capítulo 3 não tinha sido possível mostrar uma associação significativa entre falhas na *build* e ocorrência de rajadas de mudanças, as descobertas relatadas em ambos os estudos podem ajudar a comunidade de pesquisadores e desenvolvedores a orientar melhor trabalhos futuros sobre práticas de desenvolvimento.

Ao final desta pesquisa é importante destacar os pontos relacionados que ainda podem ser explorados, e que pretende-se continuar pesquisando:

- Algoritmos que detectam rajadas podem conter valiosos detalhes na detecção de rajadas.
- Caracterização de novos projetos de Software Livre: Este trabalho buscou projetos com características ímpares, e que podem, ou não, se reproduzirem em outros

projetos dessa natureza. Um dos intentos do autor desta dissertação e de seus orientadores é dar continuidade a esse estudo tendo como fonte projetos com outros tipos de características.

- Novas visões do causador das falhas na *build*: Esta pesquisa apontou tópicos que estão relacionados à falhas na *build*. Considerando um item tão abstrato como rajadas de commits, é possível que, na visão de outros pesquisadores e desenvolvedores, novos tópicos sejam acrescentados à lista apresentada por esta dissertação.

REFERÊNCIAS BIBLIOGRÁFICAS

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

BOGARD, D.; TIEDERMAN, W. Burst detection with single-point velocity measurements. *Journal of Fluid Mechanics*, Cambridge University Press, v. 162, p. 389–413, 1986.

CHARMAZ, K. *Constructing grounded theory: A practical guide through qualitative analysis*. [S.l.]: sage, 2006.

COHEN, J. *Statistical power analysis for the behavioral sciences. 2nd*. [S.l.]: Hillsdale, NJ: erlbaum, 1988.

CORBIN, J.; STRAUSS, A. Strategies for qualitative data analysis. *Basics of Qualitative Research. Techniques and procedures for developing grounded theory*, v. 3, n. 10.4135, p. 9781452230153, 2008.

DUVALL, P. M.; MATYAS, S.; GLOVER, A. *Continuous integration: improving software quality and reducing risk*. [S.l.]: Pearson Education, 2007.

FOWLER, M. *Integração contínua*. 2006. Disponível em: ⟨http://www.pensamentos.com/integraocontnua⟩.

FUNG, G. P. C. et al. Parameter free bursty events detection in text streams. In: VLDB ENDOWMENT. *Proceedings of the 31st international conference on Very large data bases*. [S.l.], 2005. p. 181–192.

GEROSA, M. A. et al. Mineração de repositórios de software livre. *Revista da Sociedade Brasileira de Computação*, p. 19–24, 2015.

HASSAN, A. E. The road ahead for mining software repositories. In: IEEE. *Frontiers of Software Maintenance, 2008. FoSM 2008.* [S.l.], 2008. p. 48–57.

HASSAN, A. E.; XIE, T. Software intelligence: the future of mining software engineering data. In: ACM. *Proceedings of the FSE/SDP workshop on Future of software engineering research*. [S.l.], 2010. p. 161–166.

HUMBLE, J.; FARLEY, D. *Entrega Contínua: Como Entregar Software de Forma Rápida e Confiável*. [S.l.]: Editora Bookman, Brasil, 2014.

ISLAM, M. R.; ZIBRAN, M. F. Insights into continuous integration build failures. In: IEEE PRESS. *Proceedings of the 14th International Conference on Mining Software Repositories*. [S.l.], 2017. p. 467–470.

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

KHAN, A.; AHSAN, S. N. Predicting bug inducing source code change patterns. In: IEEE. *Open Source Systems & Technologies (ICOSST), 2016 International Conference on*. [S.l.], 2016. p. 29–35.

KITCHENHAM, B. A.; PFLEEGER, S. L. Principles of survey research part 2: designing a survey. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 27, n. 1, p. 18–20, 2002a.

KITCHENHAM, B. A.; PFLEEGER, S. L. Principles of survey research: part 3: constructing a survey instrument. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 27, n. 2, p. 20–24, 2002b.

KLEINBERG, J. Bursty and hierarchical structure in streams. *Data Mining and Knowledge Discovery*, Springer, v. 7, n. 4, p. 373–397, 2003.

KWAN, I.; SCHROTER, A.; DAMIAN, D. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering*, IEEE, v. 37, n. 3, p. 307–324, 2011.

MARINSEK, N. *Detecting 'bursts' in time series data with Kleinberg's burst detection algorithm*. 2017. Disponível em: ⟨https://nikkimarinsek.com/blog/kleinberg-burst-detection-algorithm⟩.

NAGAPPAN, N. et al. Change bursts as defect predictors. In: IEEE. *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. [S.l.], 2010. p. 309–318.

PRESSMAN, R.; MAXIM, B. *Engenharia de Software-8ª Edição*. [S.l.]: McGraw Hill Brasil, 2016.

PUNTER, T. et al. Conducting on-line surveys in software engineering. In: IEEE. *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.* [S.l.], 2003. p. 80–88.

RAHMAN, M. M.; ROY, C. K. Impact of continuous integration on code reviews. In: IEEE PRESS. *Proceedings of the 14th International Conference on Mining Software Repositories*. [S.l.], 2017. p. 499–502.

REBOUÇAS, M. et al. How does contributors' involvement influence the build status of an open-source software project? In: IEEE. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2017. p. 475–478.

ŚLIWERSKI, J.; ZIMMERMANN, T.; ZELLER, A. When do changes induce fixes? In: ACM. *ACM sigsoft software engineering notes*. [S.l.], 2005. v. 30, n. 4, p. 1–5.

SOUZA, R.; SILVA, B. Sentiment analysis of travis ci builds. In: IEEE PRESS. *Proceedings of the 14th International Conference on Mining Software Repositories*. [S.l.], 2017. p. 459–462.

SPINELLIS, D. Version control systems. *IEEE Software*, IEEE, v. 22, n. 5, p. 108–109, 2005.

STRAUSS, A.; CORBIN, J. *Basics of qualitative research*. [S.l.]: Sage publications, 1990.

TAMURA, K.; KITAKAMI, H. A new parallelization model for detecting temporal bursts in large-scale document streams on a multi-core cpu. In: IEEE. *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on*. [S.l.], 2014. p. 519–524.

TRAVIS-CI. *What is Travis CI*. 2017. Disponível em: ⟨https://github.com/travis-ci/travis-ci/blob/2ea7620f4be51a345632e355260b22511198ea64/README.textile\#goals.⟩

TUFANO, M.; SAJNANI, H.; HERZIG, K. Towards predicting the impact of software changes on building activities. In: IEEE. *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. [S.l.], 2019. p. 49–52.

VASILESCU, B. et al. Continuous integration in a social-coding world: Empirical evidence from github. In: IEEE. *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. [S.l.], 2014. p. 401–405.

XIA, J.; LI, Y. Could we predict the result of a continuous integration build? an empirical study. In: IEEE. *Software Quality, Reliability and Security Companion (QRS-C), 2017 IEEE International Conference on*. [S.l.], 2017. p. 311–315.

YUAN, Z.; JIA, Y.; YANG, S. Online burst detection over high speed short text streams. In: SPRINGER. *International Conference on Computational Science*. [S.l.], 2007. p. 717–725.

ZAGORISIOS, P. Text clustering using bursty information. 2015.

ZHANG, X.; SHASHA, D. Better burst detection. In: IEEE. *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. [S.l.], 2006. p. 146–146.

ZHU, Y.; SHASHA, D. Efficient elastic burst detection in data streams. In: ACM. *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. [S.l.], 2003. p. 336–345.

Apêndice

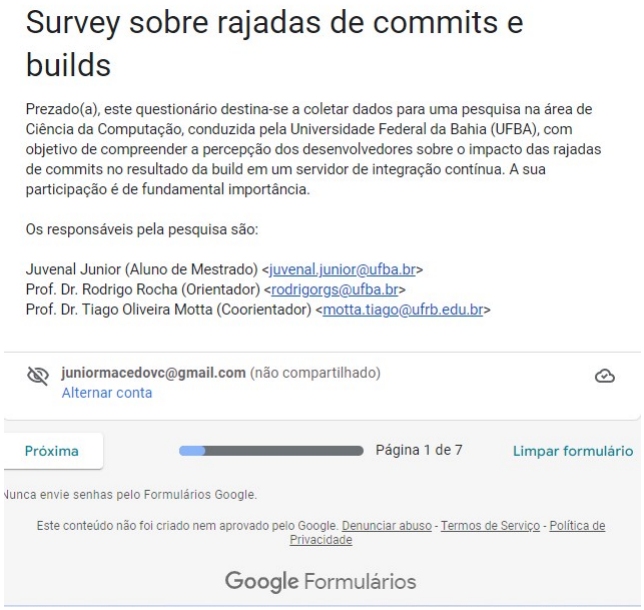
A

FORMULÁRIOS UTILIZADOS NO ESTUDO DE AVALIAÇÃO DOS RESULTADOS OBTIDOS PELOS DESENVOLVEDORES SOBRE AS RAJADAS DE COMMITS E O STATUS DA BUILD

Este Apêndice apresenta o formulário utilizado no estudo de avaliação dos resultados obtidos por esta dissertação e apresentado no Capítulo 4 e direcionado para os Desenvolvedores que tem como linguagem nativa o Português, mas também foi disponibilizada uma versão em língua Inglesa desse formulário.

A.1 FORMULÁRIO APRESENTADO À DESENVOLVEDORES EM LÍNGUA PORTUGUESA

A seguir são apresentados um conjunto de figuras que juntas, representam as telas oferecidas aos desenvolvedores que optaram por preencher o formulário em língua Portuguesa. Para completa compreensão, considere as figuras (A.1 à A.11) na sequência apresentada.



Survey sobre rajadas de commits e builds

Prezado(a), este questionário destina-se a coletar dados para uma pesquisa na área de Ciência da Computação, conduzida pela Universidade Federal da Bahia (UFBA), com objetivo de compreender a percepção dos desenvolvedores sobre o impacto das rajadas de commits no resultado da build em um servidor de integração contínua. A sua participação é de fundamental importância.

Os responsáveis pela pesquisa são:

Juvenal Junior (Aluno de Mestrado) <juvenal.junior@ufba.br>
Prof. Dr. Rodrigo Rocha (Orientador) <rodrigorgs@ufba.br>
Prof. Dr. Tiago Oliveira Motta (Coorientador) <motta.tiago@ufrb.edu.br>

juniormacedovc@gmail.com (não compartilhado)
Alternar conta

Próxima Página 1 de 7 Limpar formulário

Nunca envie senhas pelo Formulários Google.

Este conteúdo não foi criado nem aprovado pelo Google. Denunciar abuso - Termos de Serviço - Política de Privacidade

Google Formulários

Figura A.1 Formulário para desenvolvedores em língua Portuguesa - Seção 1.

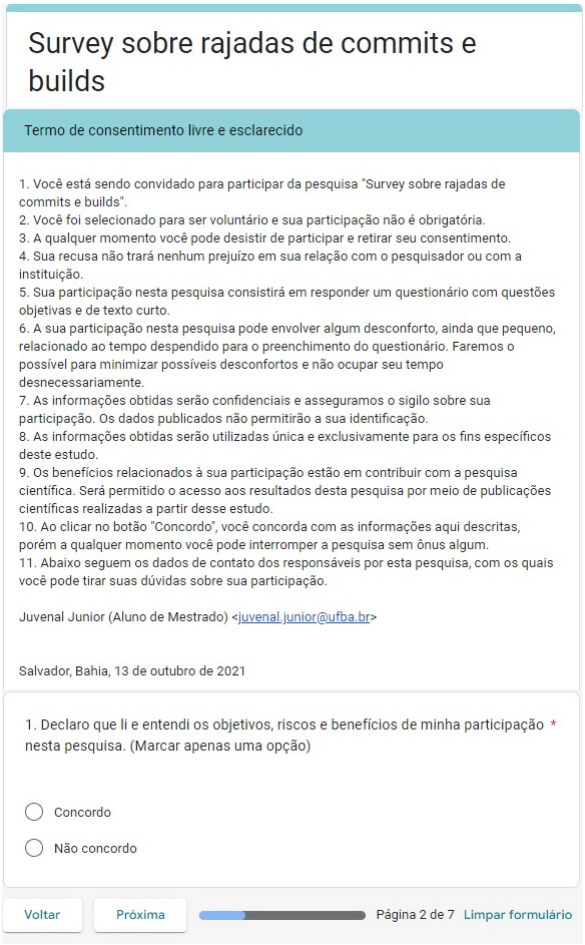


Figura A.2 Formulário para desenvolvedores em língua Portuguesa - Seção 2.

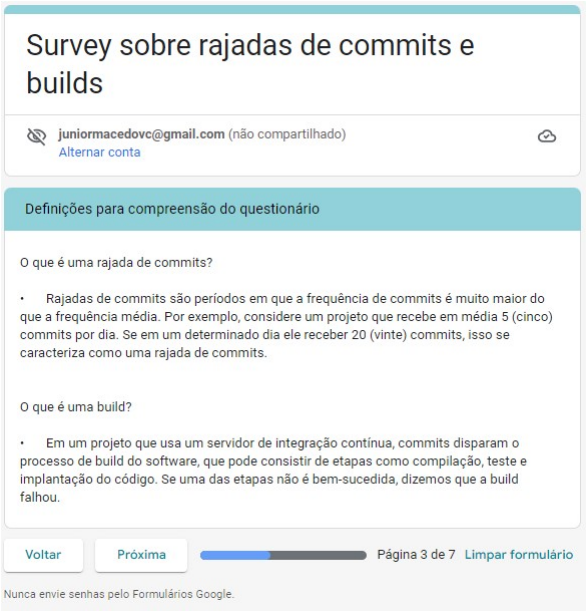


Figura A.3 Formulário para desenvolvedores em língua Portuguesa - Seção 3.

Figura A.4 Formulário para desenvolvedores em língua Portuguesa - Seção 4.

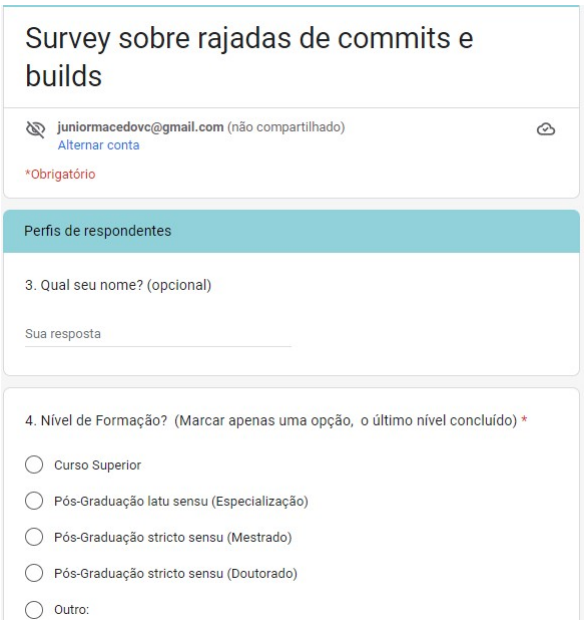


Figura A.5 Formulário para desenvolvedores em língua Portuguesa - Seção 5 (parte 1).

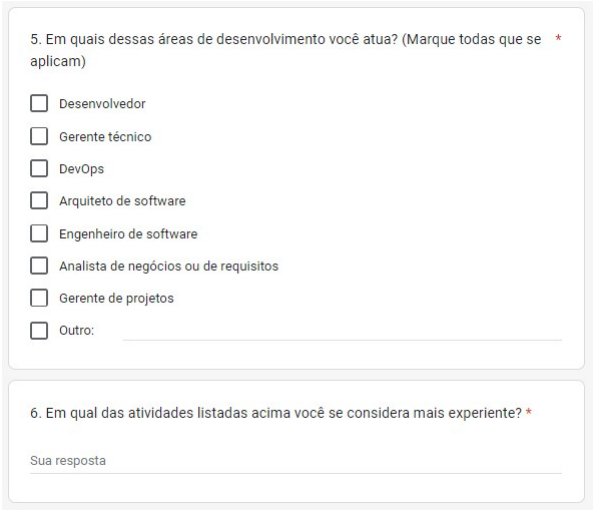


Figura A.6 Formulário para desenvolvedores em língua Portuguesa - Seção 5 (parte 2).

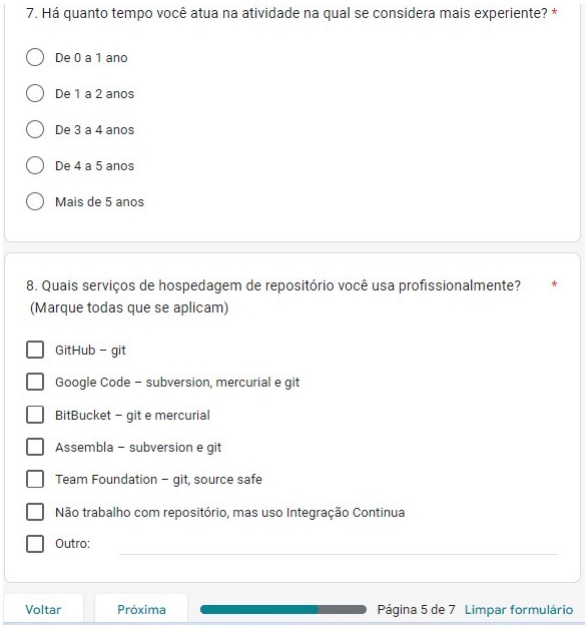


Figura A.7 Formulário para desenvolvedores em língua Portuguesa - Seção 5 (parte 3).

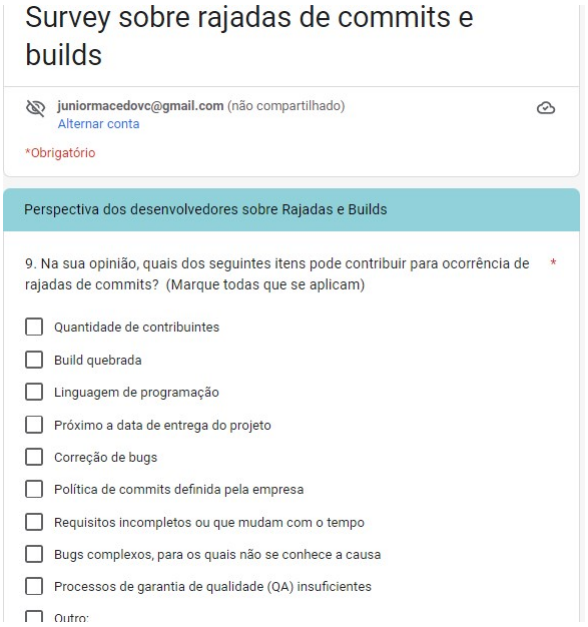


Figura A.8 Formulário para desenvolvedores em língua Portuguesa - Seção 6 (parte 1).

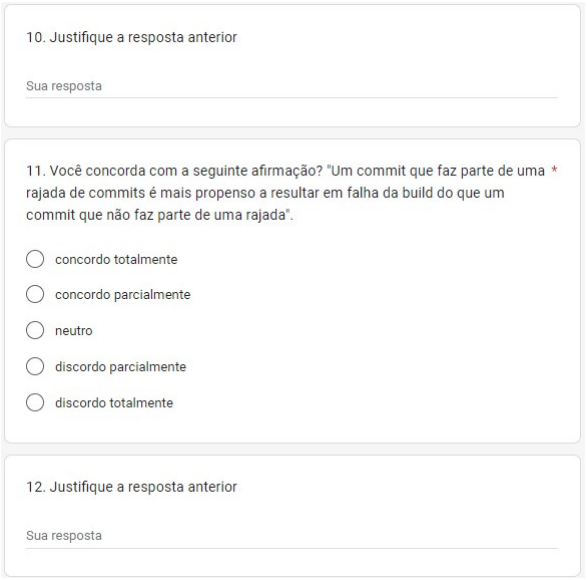


Figura A.9 Formulário para desenvolvedores em língua Portuguesa - Seção 6 (parte 2).

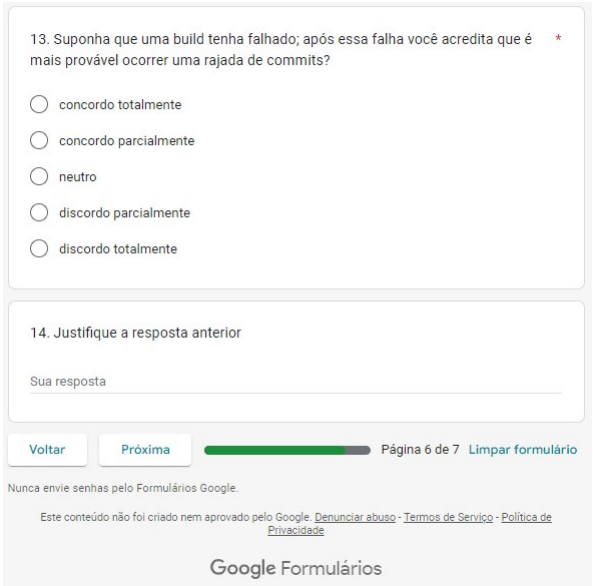


Figura A.10 Formulário para desenvolvedores em língua Portuguesa - Seção 6 (parte 3).

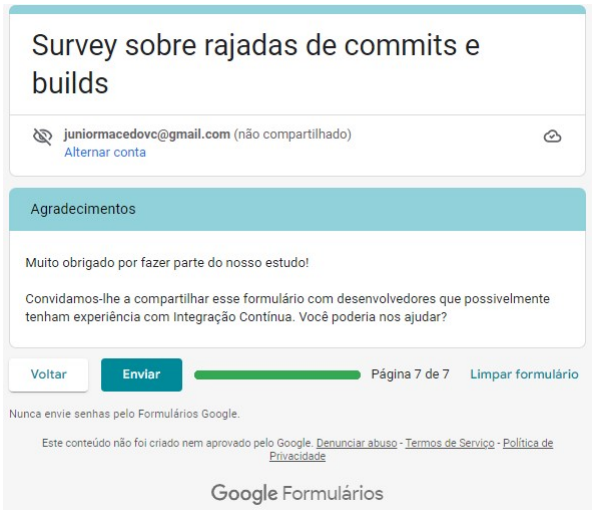


Figura A.11 Formulário para desenvolvedores em língua Portuguesa - Seção 7.