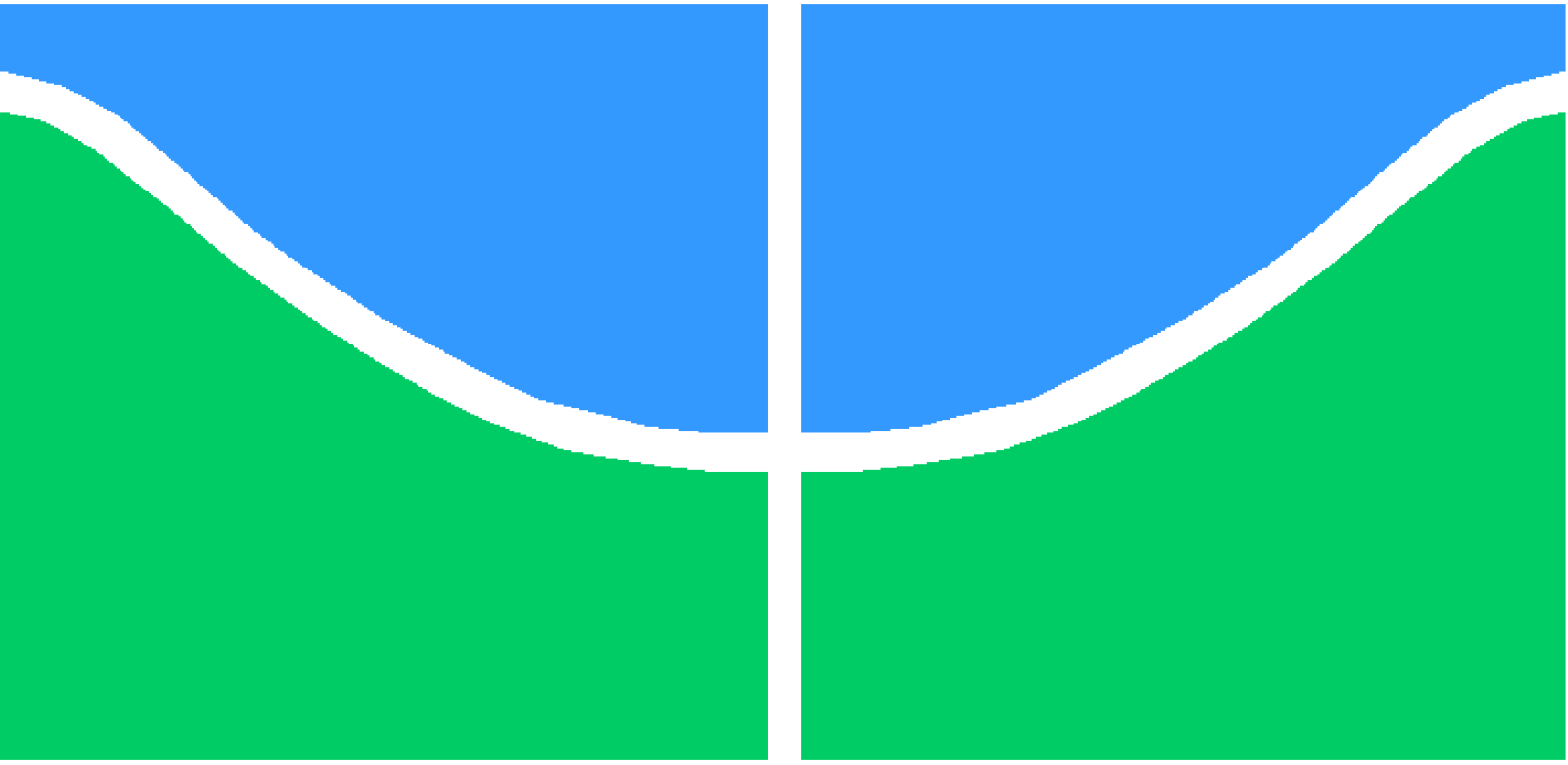

Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Um estudo sobre boas práticas adotadas na documentação de repositórios de *software* livre

Autor: André de Sousa Costa Filho
Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF
2021

André de Sousa Costa Filho

Um estudo sobre boas práticas adotadas na documentação de repositórios de *software* livre

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF

2021

André de Sousa Costa Filho

Um estudo sobre boas práticas adotadas na documentação de repositórios de *software* livre/ André de Sousa Costa Filho. – Brasília, DF, 2021-

67 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Renato Coral Sampaio

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2021.

1. Estudo. 2. Boas práticas. I. Prof. Dr. Renato Coral Sampaio. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Um estudo sobre boas práticas adotadas na documentação de repositórios de *software* livre

CDU 02:141:005.6

André de Sousa Costa Filho

Um estudo sobre boas práticas adotadas na documentação de repositórios de *software* livre

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 21 de maio de 2021:

Prof. Dr. Renato Coral Sampaio
Orientador

Prof. Dr. Bruno César Ribas
Convidado 1

Prof. Dr. Fábio Macêdo Mendes
Convidado 2

Brasília, DF
2021

Este trabalho é dedicado à todos aqueles que me apoiaram e me deram forças durante o decorrer do curso.

Resumo

Este trabalho realiza uma análise das principais regras e boas práticas relacionadas à documentação utilizadas atualmente nos repositórios do Github. Nele é exposto o estado atual do desenvolvimeno colaborativo de *software*, os conceitos principais por trás do mesmo e a importância do uso de boas práticas no desenvolvimento colaborativo. Além destes pontos, são apresentados os principais conceitos do Git, os impactos e importância que a documentação de *software* possui, as boas práticas de repositórios de *software* propostas pela *Core Infrastructure Initiative* e pelo Github, assim como sua importância e feitas análises em alguns grandes repositórios de software livre.

Palavras-chaves: Git. Commit. Boas práticas. Versionamento. Desenvolvimento. Software. Documentação. Workflow.

Abstract

This document analyzes the main rules and best practices currently used with Git. It exposes the current state of collaborative software development as well as the key concepts behind it, and the importance of using good practices in modern collaborative development. In addition to these points, we present the main concepts of Git, the impacts and importance that a commit has as a part of software documentation, the main good practices of writing a commit message, some of the conventions created by the community in order to help the writing of good commit messages and introduces some tools designed to help the developer write commit messages following those good practices guidelines. Software repositories good practices are also analyzed, as well as their importance and impacts in the current scenario of collaborative software development.

Key-words: git. commit. commit conventions. good practices. versioning. software development. software. software documentation. workflow.

Lista de ilustrações

Lista de abreviaturas e siglas

CLI	Abreviação para *Command Line Interface*, se refere à aplicações executadas na linha de comando.
UI	Abreviação para *User Interface*.
FLOSS	Abreviação para *Free Libre Open-Source Software*.
RoR	Abreviação para *Ruby on Rails*.
CII	Abreviação para *Core Infrastructure Iniciative*.
MVC	Apreviação para *Model View Controller*.

Sumário

1 Introdução

No cenário de desenvolvimento de software atual, é praticamente obrigatório o uso de ferramentas de versionamento, seja para código, configurações ou para documentação em geral (LOELIGER; MCCULLOUGH, 2012).

O uso de sistemas e técnicas de versionamento e revisão de versões se tornaram essenciais para o desenvolvimento de software atualmente. Tais sistemas são encontrados em três diferentes paradigmas: sistemas de versionamento local, sistemas de controle de versão distribuídos - *DVCS* - e sistemas de controle de versão centralizados - *CVCS* (ZOLKIFLI AMIR NGAH*, 2018).

Com a contínua expansão da comunidade de software livre, se tornou cada vez mais comum o uso de sistemas DVCS por conta das desvantagens ou impedimentos que os outros paradigmas proporcionavam para o desenvolvimento colaborativo (ZOLKIFLI AMIR NGAH*, 2018) (LOELIGER; MCCULLOUGH, 2012).

Inserido neste cenário, o *Git* é uma das ferramentas mais poderosas e utilizadas pelos desenvolvedores de software espalhados mundo afora (OLSSON; VOSS, 2014). O *Git*, é um software de versionamento de arquivos no formato distribuído criado por Linus Torvalds e se tornou recentemente a principal tecnologia utilizada para versionamento de repositórios de software (OPENHUB, 2019), como demonstra o gráfico de comparação de repositórios tirado do OpenHub (Figura 1). De um espaço amostral de 1.296.669 repositórios, aproximadamente 70% são versionados utilizando o Git, sendo que o Subversion - ou SVN - a segunda maior tecnologia, é utilizado em aproximadamente 25% dos mesmos.

O Git é um software livre que segue a filosofia *FLOSS* e, provavelmente por conta disto, possui uma vasta gama de ferramentas em seu ecossistema a fim de dar suporte ao desenvolvedor. Além de várias CLIs e UIs voltadas para a ajuda no uso da ferramenta, existem os gerênciadores de repositórios do Git (DAWSON; STRAUB, 2016).

Estes gerenciadores são plataformas no qual são submetidas as alterações dos repositórios, sendo as mais utilizadas o *Github*, *GitLab* e *Bitbucket*. Com o uso destas ferramentas, o já presente suporte ao desenvolvimento colaborativo no git se torna mais significante, visto que elas proveêm suporte para diversas formas de organização e distribuição de tarefas para equipes, além de diversas integrações com ferramentas externas.

Tal suporte fez com que as comunidades de software livre utilizassem dessas plataformas para manter seus projetos *open-source* abertos para contribuições externas, de uma forma mais acessível e - em geral - menos burocrática. Considerando o fato de times de desenvolvimento cada vez mais distribuídos estarem se tornando uma realidade progres-

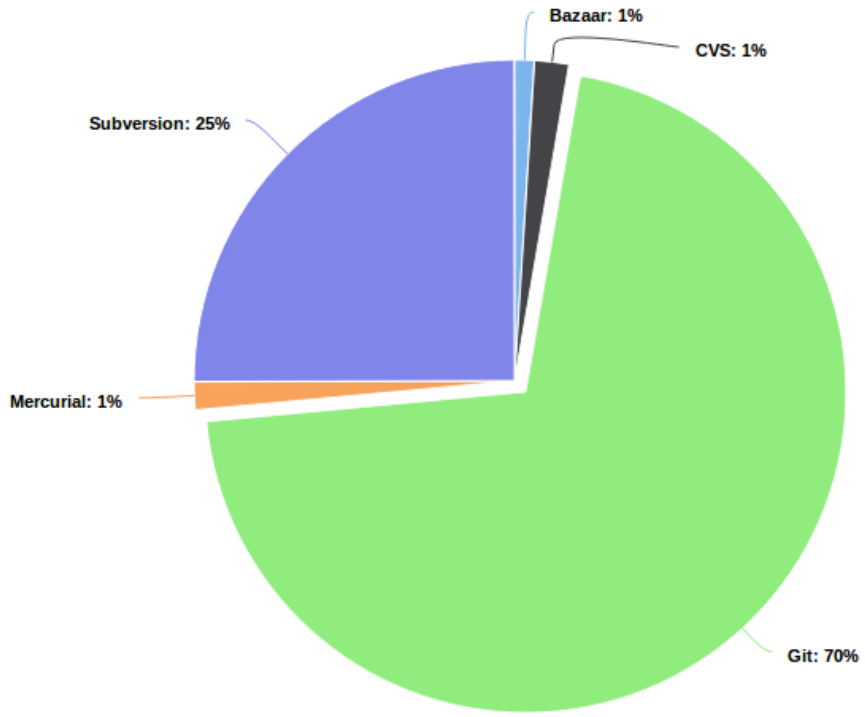

Figura 1 – Comparação de número de repositórios com o Git e outras tecnologias. Fonte: *openhub*

sivamente mais comum (STACKOVERFLOW, 2019), é possível afirmar que a demanda para plataformas colaborativas é sólida e não deixará de ser significante facilmente.

Com a popularização destas plataformas e do workflow envolvido para a contribuição em softwares livres, começaram a surgir diferentes tipos de documentação, regras e guias voltados à auxiliar desenvolvedores a contribuírem em projetos, mantendo ao mesmo tempo a organização e qualidade dos mesmos.

Seja essa documentação composta dos clássicos guias de contribuição e *READMEs*, até os novos templates de *pull request*, *issues*, é essêncial que essa documentação exista para guiar novos contribuidores.

Pensando nisso, o Github lançou o dashboard de ferramentas para comunidade em 14 de junho de 2017 (FUKUI, 2017) proporcionando recomendações de documentação consideradas como boas práticas para repositórios de software livre, esse dashboard pode ser visto na aba *insights* opção *community*, conforme a Figura 2.

Segundo essas recomendações, um bom repositório de software livre deve possuir pelo menos os seguintes artefatos (GITHUB, 2019a):

- Descrição do repositório
- Arquivo README
- Código de conduta

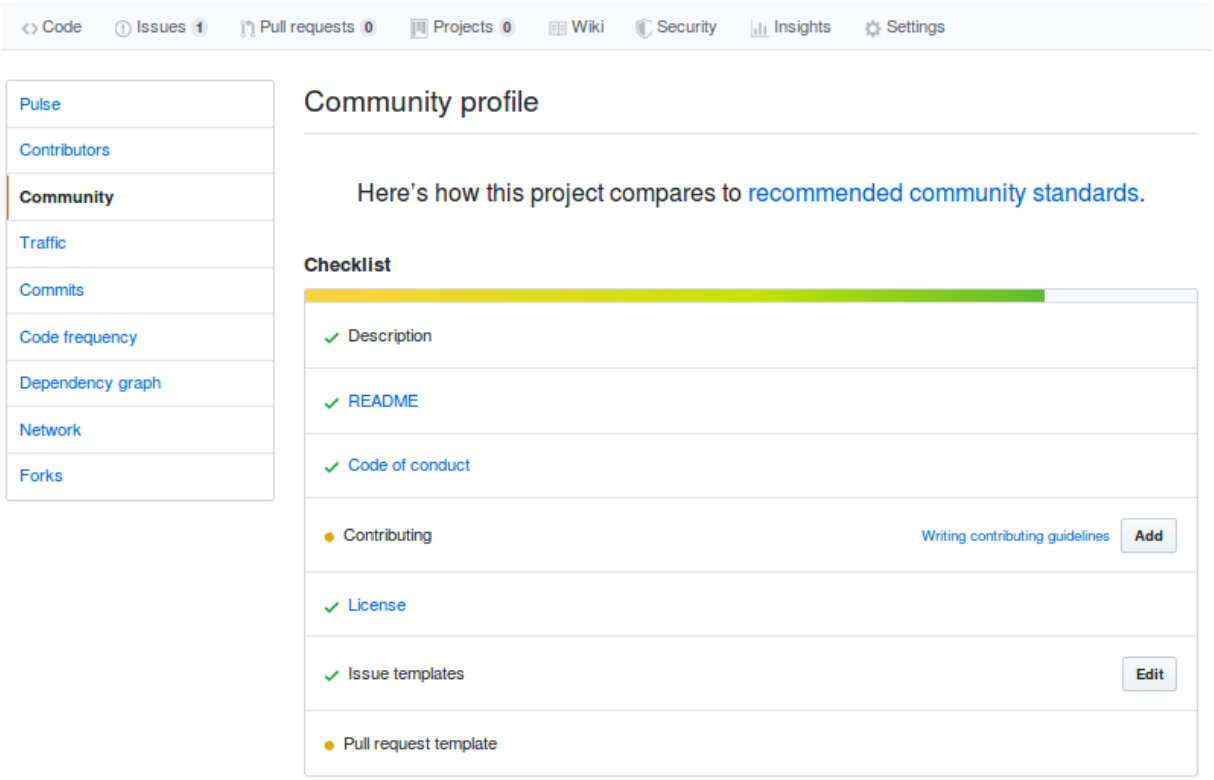

Figura 2 – Exemplo de dashboard de comunidade no Github

- Guia de contribuição
- Licença
- Template de *issue*
- Template de *pull request*

Embora aparentemente tenha tomado para si o papel de guia do desenvolvimento *open-source*, por conta de ser a maior e mais usada plataforma de host de repositórios no mundo, não são todos que seguem estas recomendações do Github. Outras plataformas atualmente não possuem este mesmo recurso de dashboard ou guias como os presentes no Github. Nelas estes são inexistentes ou não se encontram de forma explícita.

Vale ressaltar que no final de 2020 o GitHub também inseriu em seu painel de comunidade o indicador da configuração para habilitar aos administradores do repositório aceitar denúncias de conteúdo no repositório e afins.

Um dos maiores pontos de divergência entre as comunidades de desenvolvimento é o que são consideradas boas práticas e boas políticas de repositório. Existem diversas 'fraturas' acerca do que comunidades de desenvolvimento definem sobre o que é ou não uma boa prática. Por exemplo, o guia de projetos open-source do github define uma checklist de elementos que um bom repositório de software livre deve possuir (GITHUB, 2019b),

já a Linux Foundation define uma outra checklist (ABERNATHY IBRAHIM HADDAD, 2019), possuindo diferentes elementos entre um e outro.

Sejam essas diferenças relacionadas aos commits durante o desenvolvimento de um projeto ou à mínima documentação necessária para um repositório de software livre, a realidade é que existem diversas opiniões - algumas até conflitantes - sobre o assunto.

Ao mesmo tempo que essa ampla gama de opiniões é algo capaz de provocar confusão para o desenvolvedor, ela também provê diferentes experiências e possibilidades de abordagem. Tal flexibilidade é algo notório principalmente considerando a popularidade de metodologias ágeis no cenário de desenvolvimento de software atual, e inclusive a importância que estas mesmas dão para a experimentação e empiricidade em seus processos de desenvolvimento.

1.1 O problema

Com tamanha diversidade de padrões, convenções e formatos, é comum que exista bastante inconsistência dentro dos repositórios e projetos de software. Este fato pode ocorrer por vários fatores, sendo o principal deles a adaptação à um contexto diferente de desenvolvimento.

Tal adaptação é custosa de tempo e a adaptabilidade necessária de um desenvolvedor à um contexto de trabalho é um requisito do mercado cada vez mais cobrado. Essa adaptação se torna mais custosa na mesma proporção da quantidade de regras e políticas adotadas na equipe de desenvolvimento.

Uma forma simples de se perceber isso é olhando a árvore de commits - *commit tree* - de um projeto de software. Mesmo com diversas boas práticas de escrita de commit diferentes documentadas, muitos desenvolvedores desconhecem ou não aplicam. Ou no caso dessas boas práticas serem aplicadas e exigidas, caso venha a se ter um novo membro na equipe, a curva de aprendizado dele será impactada.

Uma solução para esses problemas necessitaria de, ao mesmo tempo, ser adaptável para lidar com diferentes padrões e contextos, e ser corretiva além de capaz de guiar a aplicação de boas práticas de acordo com os mesmos.

1.2 Objetivos

Este trabalho tem como proposta esclarecer e identificar as boas práticas presentes e mais utilizadas no cénario de desenvolvimento de *software*. Para isso, os seguintes objetivos devem ser realizados:

- Identificar e realizar uma análise dos principais padrões e boas práticas utilizadas,

assim como suas características, em um determinado espaço amostral de repositórios do Github e Gitlab.

2 Metodologia

Retomando ao problema apresentado na Seção 1.1 e aos objetivos do trabalho na Seção 1.2, será realizada uma pesquisa científica descritiva com o objetivo de se levantar as características presentes nos repositórios de software livre.

Uma pesquisa descritiva tem por característica a exposição de características de um determinado objeto de pesquisa (OLIVEIRA, 2011). Para este trabalho, o objeto de pesquisa será um pequeno espaço amostral contendo repositórios de software livre com comunidades maduras.

Serão selecionados seis repositórios de software que podem ser considerados de grande porte e com comunidades de desenvolvimento ativas e maduras.

Assim, no final da análise, serão levantadas as características de tais repositórios quanto à sua documentação e quais as práticas adotadas pelos mesmos atualmente. A partir dos resultados obtidos, teremos documentados o formato de documentação adotada por estes repositórios e a relação da mesma com as boas práticas vistas anteriormente.

A síntese destes resultados será uma lista mínima de documentação necessária, tendo em vista o que está sendo realmente aplicado na prática pelas maiores comunidades de desenvolvimento. Vale resaltar que a análise de dará em repositórios hospedados da plataforma Github.

2.1 Pontos de análise

Considerando o que foi informado sobre a pesquisa a ser realizada, os principais pontos de análise dessa pesquisa serão:

- Compatibilidade com os padrões estabelecidos pelo Github, como mostrado no capítulo 1.
- Compatibilidade com os padrões estabelecidos pelo *Core Infrastructure Initiative* da *Linux Foundation*, descrito na Seção 3.3.3.
- Presença de documentação relacionada ao projeto, como o arquivo *README*, licença, versionamento em *tags*, *et cetera*.
- Presença de documentação destinada ao auxílio de contribuidores, como guia de contribuição, política de *branches*, política de *commits*, *templates* para *issues* e *pull requests* e código de conduta.

- Presença de documentação de negócio do projeto, como guias para o usuário edescrição.
- Local de hospedagem da documentação do projeto, se está disponível em *wiki* ou como website.
- Tamanho da comunidade de desenvolvimento.
- Relações entre o tamanho da comunidade com o uso das boas práticas.

Vale resaltar que por conta deste trabalho ter como foco a análise da documentação dos repositórios, iremos analisar apenas os itens que dizem respeito ao mesmo quando verificarmos a compatibilidade com os padrões estabelecidos pelo *Core Infrastructure Initiative*.

Tais critérios são: *conteúdos básicos, controle de versão, notas de lançamento e relatórios de falhas*. Os demais pontos definidos pela *Core Infrastructure Inicative*, são mais relacionados à ferramentas e práticas técnicas, como por exemplo o emprego ou não de ferramentas de análise estática, fugindo do tema deste trabalho.

2.2 Cronograma

Dado o período de realização do trabalho sendo de 4 meses, o cronograma de realização será dividido em três etapas, cada qual com suas atividades de pesquisa análise e documentação, como pode ser visto na Figura 3.

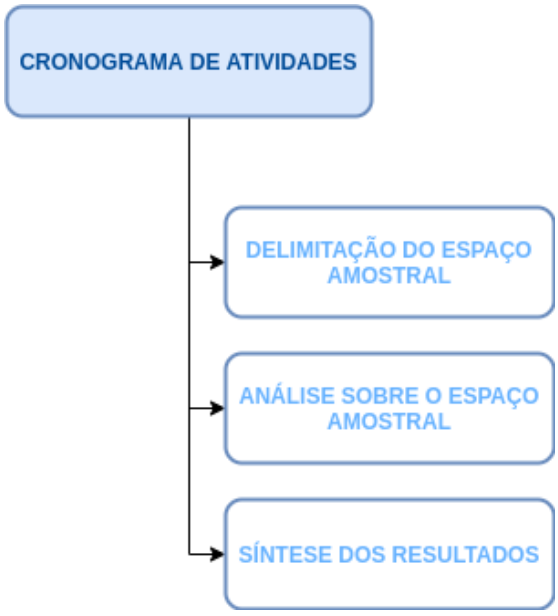

Figura 3 – Cronograma de atividades da segunda etapa do trabalho de conclusão de curso.

3 Fundamentação teórica

Neste capítulo serão abordados os seguintes assuntos: o uso da ferramenta Git junto de seus principais conceitos; será aprofundado o conceito do *commit* dentro da ferramenta Git, assim como as boas práticas relacionadas ao mesmo e sua importância como documentação de software; serão também exploradas as ferramentas de auxílio à escrita de *commits* seguindo convenções sobre o mesmo; além de ser abordadas as principais convenções relacionadas à repositórios de software.

3.1 O uso do Git

O Git é uma ferramenta bastante poderosa e de fácil uso em sua forma básica, porém seu uso pode se tornar bastante complexo ao se usar funcionalidades mais avançadas. O objetivo desta seção é introduzir elementos e conceitos presentes no restante do trabalho, para garantir um melhor entendimento dos conceitos.

Seja individualmente ou de forma colaborativa, o uso da ferramenta Git gira em torno de três funções fundamentais: *commits*, *branches* e interações com outros repositórios. Cada uma dessas permite, respectivamente, o versionamento, o trabalho em paralelo e o controle de versionamento distribuído.

Commit é o nome dado para cada versão que o usuário cria com a ferramenta. Um *commit* é formado por: uma hash de identificação criada pelo Git, a identificação do autor, data de criação, uma mensagem associada, a *hash* do *commit* pai e da *branch* a qual o mesmo pertence, além das modificações realizadas.

Essas versões podem ser acessadas à qualquer momento pelo desenvolvedor, por meio da hash identificadora de cada *commit*. O Git é capaz de listar todas as versões que se possui até aquele momento a partir do commando **git log** como pode ser visto na Figura 3.1.

Nessa exibição, as alterações realizadas - chamadas dentro da ferramenta de *diff* - são ocultadas para o usuário, assim sendo, se torna perceptível a importância da mensagem presente em um *commit*. Mais adiante, boas práticas e conceitos relacionados ao *commit* serão melhor abordados na Seção 3.1.

Um outro conceito fundamental do Git são as *branches*. Uma *branch* funciona como uma lista ligada de *commits*, já que cada um possui a referência do seu anterior. Por padrão, o Git proporciona ao usuário uma *branch* padrão, chamada *master*. Essa branch é criada junto ao primeiro *commit* de um repositório. O uso de *branches* permite o desenvolvimento paralelo de diversas atividades dentro de um repositório de trabalho.

Figura 4 – Exemplo de um commit como visto em um **git log**.

Relacionado as *branches*, está a árvore de *commits*. Ela é uma forma de visualizar as *branches* e seus respectivos *commits*. Uma representação de uma árvore de *commits* - ou *commit tree* - pode ser vista na Figura 5. A utilização de *branches* é um dos pilares principais que suportam a capacidade do Git de permitir trabalho simultâneo e colaborativo, visto que cada colaborador pode realizar alterações em sua *branch* sem interferir com o trabalho de outro.

Branches também são extremamente importantes para uma boa organização do repositório, geralmente possuindo em repositórios de software sua própria política a ser seguida pelos contribuidores.

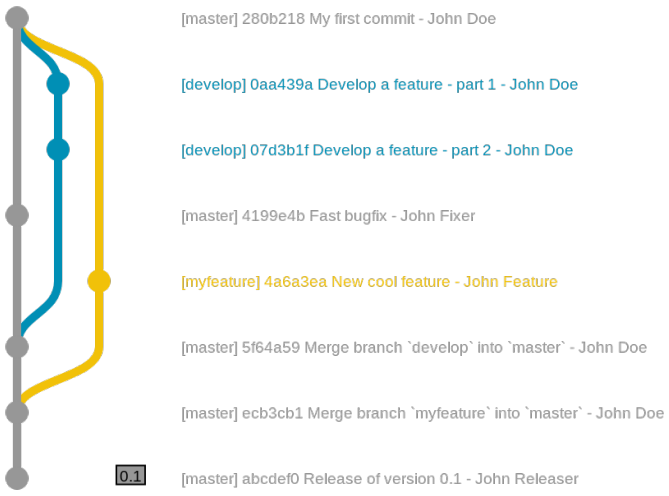

Figura 5 – Exemplo de uma *git tree* com 3 *branches*. Fonte: <http://jsfiddle.net/fracz/q76vj8ow>

Por se tratar de um sistema de versionamento distribuído, o Git possui ferramentas

de gerenciamento de repositórios, permitindo que o usuário gerencie as mudanças realizadas no seu repositório local e as submeta para o repositório em um servidor e vice-versa. O Git realiza tais ações por meio da mesclagem de *branches*. Essa mesclagem pode ocorrer entre *branches* presentes no repositório local e entre *branches* localizadas no repositório local e na nuvem.

3.2 O *commit*

3.2.1 O uso de *commits* como parte de documentação de software

É algo já estabelecido na área de engenharia e desenvolvimento de software a importância da documentação. Seja em forma de documentos, tutoriais, testes e diagramas ou seja na forma de artefatos menos formais, como *issues* e discussões a cerca de código (SAINI, 2014).

Segundo Sommerville, a documentação de um produto de software faz parte do que é denominado software (SOMMERVILLE, 2011), portanto artefatos presentes no processo de desenvolvimento de software com o intuito de informar ou ajudar desenvolvedores ou colaboradores externos fazem parte da documentação do software por consequência.

Durante o processo de desenvolvimento de software, especialmente considerando a cultura de colaboração presente atua(STACKOVERFLOW, 2019), desenvolvedores somos constantemente incentivados a versionar o código que trabalhamos. A importância de tal prática é deveras considerável, visto que um bom gerenciamento de versões permite um controle maior nas alterações submetidas.

Como dito previamente, no cenário atual de desenvolvimento de software a principal ferramenta de versionamento utilizada é o Git. O sistema de versionamento do Git ocorre a partir de versões chamadas de *commit*. Usualmente ele é apresentado para o usuário com os seguintes dados: nome do autor, *hash* identificadora, horário de criação do *commit* e mensagem, sendo o último o único a indicar as mudanças realizadas no *commit*.

Por mais que existam formas de enxergar as alterações realizadas naquela versão, tal ação demanda tempo e sua análise é dependente da atomicidade das alterações presentes no *commit*. Assim, o principal e melhor indicador do conteúdo presente no *commit* se torna a mensagem escrita pelo autor do mesmo.

Porém mesmo conhecendo o valor que possui uma clara mensagem de *commit*, muitos desenvolvedores insistem no erro de negligenciar o valor da mesma. Alguns criam até mensagens que não refletem de forma alguma as alterações. Como dito por Peter Hutterer:

> *Re-establishing the context of a piece of code is wasteful. We can't avoid it*

> *completely, so our efforts should go to reducing it to as small as possible. Commit messages can do exactly that, and as a result, a commit message shows whether a developer is a good collaborator.* (HUTTERER, 2009)

Assim a importância dada à uma mensagem presente no commit é reforçada. Da mesma forma que uma mensagem não significativa pode afetar negativamente uma revisão, uma mensagem clara e bem formulada pode efetivamente reduzir o custo de tempo de uma revisão. Em seu artigo sobre a importância de mensagens de *commit*, Jimmy King descreve uma experiência na qual foram poupadas horas de revisão de código para solucionar uma falha no sistema por se ter no histórico de *commits* uma mensagem deveras significativa. (KING, 2018)

3.2.2 Principais boas práticas de *commit*

Como dito na seção anterior, um *commit* é uma versão de um conjunto de mudanças realizadas, sejam elas em um ou vários artefatos de *software*. Ao criar um novo *coommit*, é requerido do usuário que ele digite uma mensagem que, idealmente, resume as mudanças presentes na versão.

Com o passar do tempo e com a crescente comunidade de usuários do Git e suas plataformas de hospedagem de repositórios, como o Github, os desenvolvedores passaram a definir boas práticas que deveriam ser seguidas para a escrita de bons *commits*.

Atualmente possuimos inúmeros livros tutoriais com foco no uso da ferramenta Git. Alguns livros são dedicados exclusivamente para as boas práticas de uso da ferramenta como, por exemplo, é o caso do livro *Git best practices guide* de Eric Pidoux (PIDOUX, 2014).

Quanto à definições presentes na literatura de o que é tratado como boa prática em relação à *commits*, existem alguns pontos em comum, como: a atomicidade e a estrutura de um *commit*. Outros pontos são vagos ou são tomados como implícitos, como é o caso do tamanho da mensagem e sua objetividade.

Atomicidade de um *commit* se refere ao agrupamento das mudanças por versão. Um *commit* atômico deve possuir um conjunto de mudanças pontuais, dentro de um único contexto. Assim, é garantida uma melhor navegação entre as versões caso venha a ser necessário regredir versões. Porém, é necessário reforçar que essas alterações, mesmo que atômicas devem ser significativas (PIDOUX, 2014; LOELIGER; MCCULLOUGH, 2012; HODSON, 2012). Um exemplo de atomicidade de *commit* pode ser dado da seguinte forma:

> Suponha que durante a correção de uma falha em um *software* qualquer, você tenha de alterar trechos de código em diferentes camadas da arquitetura do

programa. Um desenvolvedor que não segue a boa prática de atomicidade mexeria em todo o código e salvaria as alterações uma única vez.

Porém isso significa que caso algo que ele tenha feito tenha gerado uma nova falha - ou algo do tipo - ele teria que criar um novo *commit* de correção para a "correção"anterior. A situação se agrava caso não seja o mesmo desenvolvedor a corrigir a segunda falha.

Já um desenvolvedor que segue essa boa prática, geraria uma versão para cada camada ou contexto de código para que, caso algo inexperado aconteça, seja possível se identificar facilmente onde ocorreu o erro e então se tomar o melhor curso de ação para a correção do mesmo. Seja revertendo uma versão, seja corrigindo com uma nova versão.

A estrutura de um *commit*, como definida por boas práticas se dá como demostrado na Figura 6. A estrutura é composta por uma curta mensagem e uma linha em branco para facilitar a leitura, seguida de um texto com a descrição detalhada das alterações presentes naquele *commit*.

Mesmo não sendo enfatizada em todas as comunidades de desenvolvimento, essa estrutura é o padrão usado e ensinado na literatura. Um dos mais fiéis seguidores desta boa prática é a comunidade de desenvolvimento do Kernel do Linux. (LOELIGER; MCCULLOUGH, 2012; NARęBSKI, 2016; SILVERMAN, 2013; HODSON, 2012; PIDOUX, 2014; OLSSON; VOSS, 2014)

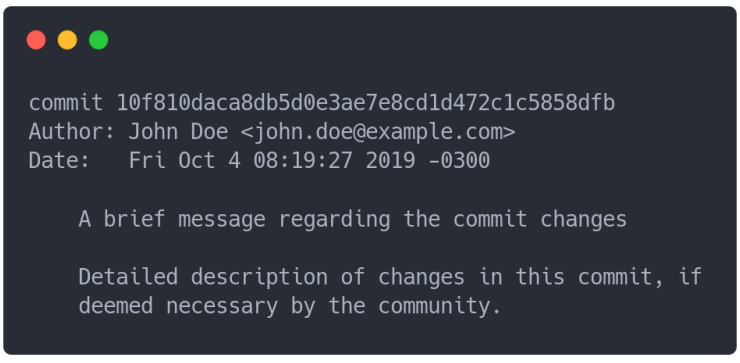

Figura 6 – Exemplo da estrutura de um *commit*.

Complementando a estrutura de um *commit*, temos definido o tamanho ideal de uma mensagem. Definida como 50 caracteres para fins de facilitar a leitura e para manter a objetividade da mesma (HODSON, 2012; CHACON, 2009; SILVERMAN, 2013). Mas

objetividade e a optimização da mensagem a ser passada, possuem dificuldade de escrita relativa para cada desenvolvedor. Alguns autores propõem o uso de marcadores na mensagem para ajudar a criar arquivos de listagem de mudanças - ou *changelogs* (OLSSON; VOSS, 2014).

Tal prática ajudou a comunidade a criar padrões de estruturas de mensagens de *commit* capazes de descrever de forma mais eficaz as alterações presentes no mesmo. Esses padrões são conhecidos como convenções de *commit*.

3.2.3 Convenções de *commit*

Uma convenção é um acordo entre duas ou mais partes, sendo geralmente sobre regras e normas. Assim, uma convenção de *commit* - ou *commit convention* - é o nome dado à um conjunto de regras e normas definidas para a elaboração de uma mensagem de *commit*.

Tais convenções visam estabelecer formas padrões de escrita de mensagens para *commits* de forma a garantir mensagens significativas enquanto garantindo a homogeneidade entre as mesmas dentro de um repositório de *software*. Atualmente, temos como uma das mais difundidas a convenção *conventional commits*, usada em vários projetos importantes como o Electron.js (CONVENTIONALCOMMITS, 2016).

Algumas das vantagens de se usar convenções são as possibilidades de integrar *commits* com funcionalidades de ferramentas externas. É possível, por exemplo, ao se usar a convenção *conventional commits*, se gerar um *changelog* perfeitamente formatado com o uso da ferramenta *conventional-changelog* (CONVENTIONALCOMMITS, 2016).

Porém para funcionarem de acordo com o proposto, e para se obter o máximo proveito de suas vantagens, é necessário que a equipe de desenvolvimento tenha a disciplina de seguir as regras de tal convenção.

A seguir, nos aprofundaremos nas peculiaridades de algumas das convenções mais comuns: a *Angular.js git commit convention*, a *Karma commit message convention*, a *Conventional commits* e a *Symphony CMF convention*. Essa análise cobrirá a documentação, seus principais pontos, exemplos de uso e seus pontos positivos e negativos.

3.2.3.1 A convenção *angular.js*

Documentada em 2016 pelos desenvolvedores da ferramenta Angular em seu guia de contribuição no Github (ANGULAR, 2011). Embora sua criação tenha sido para um projeto específico, essa convenção serviu como base para várias outras, como a *Karma* e .

A convenção tem como objetivo principal facilitar a leitura das mensagens ede forma mais fácil durante a visualização do histórico do projeto, além da geração do *chan-*

gelog.

Segundo a documentação da convenção, uma mensagem de *commit* deve ser formatada seguindo o padrão da Figura 7. A primeira linha é obrigatória, sendo apenas opcional o campo referente ao escopo das alterações. As demais linhas também são opcionais. É importante resaltar que o cabeçalho do *commit* deve ser escrito com fonte em caixa baixa, sem capitalização e ponto final. (ANGULAR, 2011)

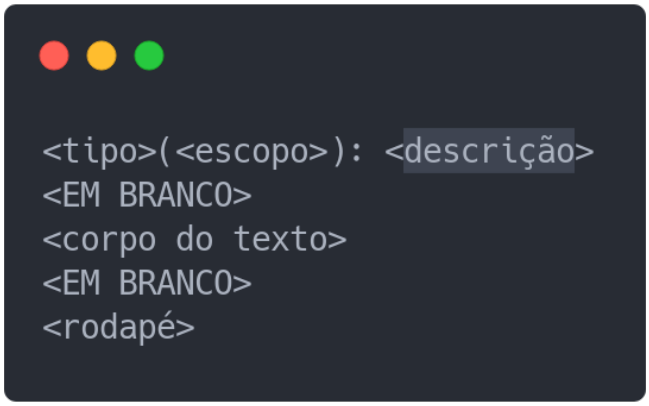

Figura 7 – Exemplo da estrutura de um *commit* seguindo a convenção angular.

A convenção é bem específica nos tipos de *commit* a serem preenchidos, e é dada uma lista de tipos possíveis para o contribuidor. (ANGULAR, 2011) Tal lista pode ser observada abaixo.

- **build**: Usada para mudanças que afetam o sistema de build ou dependências externas.
- **ci**: Para mudanças que involvem arquivos e scripts de integração contínua.
- **docs**: Mudanças que envolvem apenas arquivos de documentação.
- **feat**: Para introdução de novas funcionalidades.
- **fix**: Para mudanças que envolvem a solução de uma falha - ou um *bug*.
- **perf**: Para alterações que envolvem a melhoria da performance da aplicação.
- **refactor**: Para mudanças que não se encaixam como correção de falhas nem como uma nova funcionalidade.

- **style**: Para mudanças relacionadas com o estilo do código.
- **test**: Para mudanças que corrigem ou adicionam novos testes.

O campo referente à escopo é definido como sendo opcional e, originalmente, deve fazer referência ao pacote npm que o mesmo afeta. A descrição deste campo na documentação original descreve os pacotes que podem ser colocados neste campo. Porém para fins de convenção, tal especificidade é prejudicial.

No campo de descrição, o usuário deve escrever uma mensagem sucinta das alterações usando o verbo na forma imperativa, sem capitalização e ponto final. O campo de corpo do texto, também deve ser escrito na forma imperativa e deve conter a motivação das mudanças. O corpo do texto é opcional. (ANGULAR, 2011)

O campo de rodapé, é opcional a menos que existam mudanças que involvem descontinuidadede suporte com versões passadas, sendo obrigatória a descrição da incompatibilidade seguindo o formato *BREAKING CHANGE: <mensagem>*. No rodapé também devem ser referênciadas as *issues* que são resolvidas com aquele *commit*. (ANGULAR, 2011)

Alguns exempos de uso podem ser obervados nas Figuras 8 e 9, sendo a primeira de um *commit* presente no repositório oficial do Angular e a segunda um *commit* presente no repositório da ferramenta commit-helper, que utiliza como base a convenção angular.

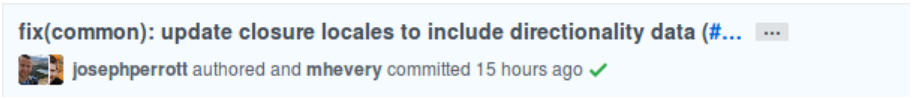

Figura 8 – Exemplo da estrutura de um *commit* seguindo a convenção angular, no repositório da ferramenta Angular.

3.2.3.2 A convenção Karma

A convenção Karma possui bastante em comum com a convenção angular, sendo indicada inclusive em sua documentação que foi baseada nas especificação da convenção presente no projeto Angular. A estrutura dos commits seguindo a convenção é a mesma da Figura 7 devido à sua semelhança com a convenção angular. (KARMA, 2014) Um exemplo de *commit* com a convenção karma pode ser vista na Figura 10.

A primeira linha da mensagem - ou cabeçalho, deve possuir no máximo 70 caracteres e ambos o tipo e o escopo devem ser escritos em caixa baixa. O escopo, assim como na convenção angular, não é necessáriamente obrigatório, porém sua abordagem é diferente por indicar agora o módulo ou componente na qual a mudança afeta, e sendo vazio também caso a mudança seja global.

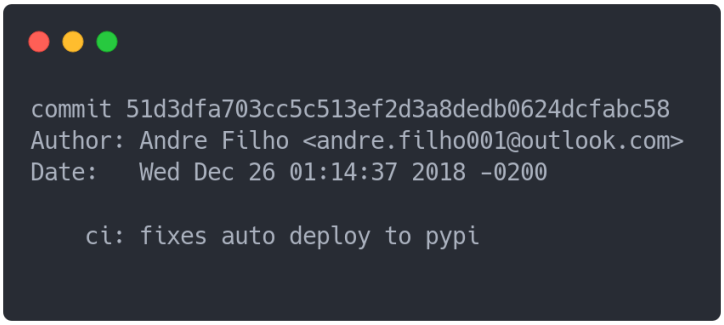

Figura 9 – Exemplo de um *commit* seguindo como base a convenção angular, presente no *commit log* da ferramenta commit-helper.

A classificação de tipo de *commit*, ocorre com algumas marcações similares porém com descrições diferentes e mais simples, como podemos ver na lista abaixo. As regras para o corpo, rodapé e flexão do verbo se mantém iguais ao da convenção angular.

- **feat**: Usado quando é adicionada uma nova funcionalidade para o **usuário**.
- **fix**: Usado quando é corrigido um problema para o **usuário**.
- **docs**: Usados para mudanças na documentação.
- **style**: Usado para formatação de código apenas.
- **refactor**: Usado para refatoração de código que afeta ou pode afetar a lógica em produção.
- **test**: Usado para adição e/ou correção de testes.
- **chore**: Usado para alterações em scripts e demais casos que não afetam produção.

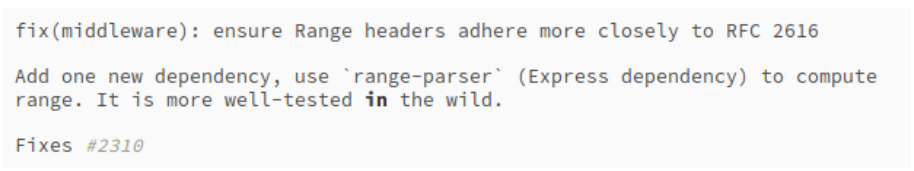

Figura 10 – Exemplo de um *commit* seguindo a convenção karma (KARMA, 2014)

3.2.3.3 A convenção *conventional commits*

Como dito no começo da Seção 3.2.3, atualmente esta convenção tem sido adotada por vários projetos e é o principal foco das ferramentas commitzen e commitlint. Essa convenção também tem é fortemente influenciada pela convenção angular, como está em sua documentação. (CONVENTIONALCOMMITS, 2016)

```
refactor!: drop support for Node 6

BREAKING CHANGE: refactor to use JavaScript features not available in Node 6.
```

Figura 11 – Exemplo de um *commit* seguindo a convenção *conventional commits* (CONVENTIONALCOMMITS, 2016)

Sua estrutura, assim como da convenção Karma, é semelhante à da convenção angular. Sua notação altera e denota apenas os tipos de *fix, feat* e *BREAKING CHANGE*, porém deixa aberta para o usuário o uso dos demais tipos da convenção angular (CONVENTIONALCOMMITS, 2016). Abaixo podemos ver a descrição dos tipos definidos pela convenção. Um exemplo de *commit* seguindo a convenção pode ser visto na Figura 11

- **fix**: Corrige uma falha no código. Essa alteração deve ser correlacionada ao *patch* do versionamento semântico, que será visto com mais detalhes na Seção 3.3.2.
- **feat**: Introduz uma nova funcionalidade para o sistema. Esse tipo possui relação com o *minor* presente no versionamento semântico.
- **BREAKING CHANGE**: Escrito como *BREAKING CHANGE* no rodapé do *commit*, ou na forma do caracter exclamação enquanto junto de outro tipo de *commit*. Esse tipo delimita uma alteração que possui grande impacto na aplicação. É correlacionado com o campo *MAJOR* presente no versionamento semântico.

Apenas pela especificação dos tipos de *commit*, podemos perceber um dos maiores apelos desta convenção e o motivo de sua popularidade: integração com outras práticas de desenvolvimento. Com um processo de integração contínua um pouco mais elaborado é possível, por exemplo, ajustar a versão do versionamento semântico a partir dos próprios *commits*. (CONVENTIONALCOMMITS, 2016)

3.2.3.4 A convenção *symfony cmf*

Diferentemente das demais convenções expostas até o momento, a convenção do *symfony CMF* não é similar ou fortemente influênciada pela convenção presente no projeto Angular.

Esta convenção é menos conhecida que as outras, e possui uma abordagem bastante diferente das anteriores, como podemos ver na Figura 12. Outra peculiaridade dessa convenção é o fato dela ter uma forma reduzida de escrita, mostrada na Figura 13.

No cabeçalho do *commit* existem dois campos o escopo e a descrição das mudanças. Embora tenha deixado em aberto a definição de escopo, pelo exemplo podemos deduzir que se trata da classe modificada (SYMFONY, 2016), esta deve acompanhar a breve descrição, ambas com fonte capitalizadas.

No corpo do *commit*, a convenção espera que se liste as *issues* corrigidas no *commit*, assim como uma descrição detalhada das mudanças. Caso seja necessário, existem dois campos para documentação de depreciação: *BC Breaks* e *Deprecations*. O primeiro não possui a especificação por escrito na documentação da convenção, porém pelo exemplo dado é possível deduzir que este campo é para documentar mudanças em chamadas de funções e outras alterações simalares, enquanto o segundo campo trata de depreciações de dependências do código.

Em sua versão reduzida, é adicionado um campo, com o tipo do *commit*. Esse tipo varia entre *bug, feature* e *minor*. O primeiro tipo é para mudanças relacionadas à qualquer tipo de correção no código, o segundo é usada para indicar a adição de uma nova funcionalidade no sistema e o último é usado para mudanças vistas como menos significativas, como estilo de código.

Figura 12 – Exemplo estrutural de um *commit* seguindo a convenção *symfony cmf* (SYMFONY, 2016)

Figura 13 – Exemplo estrutural reduzida de um *commit* seguindo a convenção *symfony cmf* (SYMFONY, 2016)

3.3 Convenções de repositórios de software

De nada adiantam as boas práticas de *commit* para facilitar o desenvolvimento, se o repositório para onde o código é submetido está em situação caótica ou sem processos e padrões definidos de forma a facilitar a manutenção de sua base de código ou documentação.

É possível um projeto seguir em frente sem muitos problemas se seus desenvolvedo-

res não mantém boas práticas de *commit*, mas essa possibilidade não se mantém a mesma quando se trata de organização de repositórios e seus respectivos fluxos de trabalho.

Um repositório de *software* possui três pilares principais: o fluxo de trabalho, o *software* que está sendo produzido e as pessoas envolvidas no projeto - sejam elas colaboradores ou não, sua interação ocorre através de artefatos de software, ou seja documentação. Nesta seção iremos abordar esses três eixos relacionados aos repositórios de *software*.

3.3.1 O *git flow*

Na Seção 3.1 discutimos o que são *branches* e a relevância das mesmas para o desenvolvimento de *software*. Sabemos que dentre as principais funções das *branches* está a organização do fluxo de trabalho durante o processo de desenvolvimento de *software*.

Tendo em vista o potencial agregado de organização provindo da funcionalidade de *branches* do Git, é natural que desenvolvedores passassem a criar formas de se trabalhar com o Git tirando o máximo proveito do sistema de *branches* do mesmo.

Neste cenário, surge o fluxo de trabalho com o Git que ficou conhecido como *gitflow*. Em um artigo, Vincent Driessen (DRIESSEN, 2010) publicou um modelo de trabalho com *branching* que rapidamente ganhou a atenção da comunidade (PIDOUX, 2014).

O *gitflow* possui várias *branches*, que possuem papéis bem definidos, de forma a sempre se saber onde as alterações estão e a confiabilidade de cada modificação. Uma representação gráfica do *gitflow* pode ser vista na Figura **??**. O *gitflow* divide as branches em duas classes: as principais e as de suporte. O mapeamento de *branches* ocorre da seguinte maneira: (DRIESSEN, 2010)

- A *branch* **master** será responsável por manter versões em produção sempre com a versão estável.
- A *branch* **develop** reflete as mudanças que estão sendo desenvolvidas. Assim que tais mudanças entram em ponto de entrarem em produção, elas são inseridas na *master*.
- As *branches* de **feature** são onde os desenvolvedores e colaboradores irão produzir alterações no código para adicionar novas funcionalidades. Essa *branch* somente se origina da *develop* e somente sofre mesclagem com a *develop*.
- As *branches* de **release** carregam em si mudanças que podem em breve ser colocadas em produção.
- A *branch* **hotfix** age de forma semelhante à *branch release*, porém suas modificações são emergenciais e ela é criada a partir da *branch master* e é mesclada em ambas as *branches master* e *develop*.

Quanto à classificação das *branches*, o *gitflow* trata como principais a *develop* e a *master*, mantendo o restante como suporte (DRIESSEN, 2010).

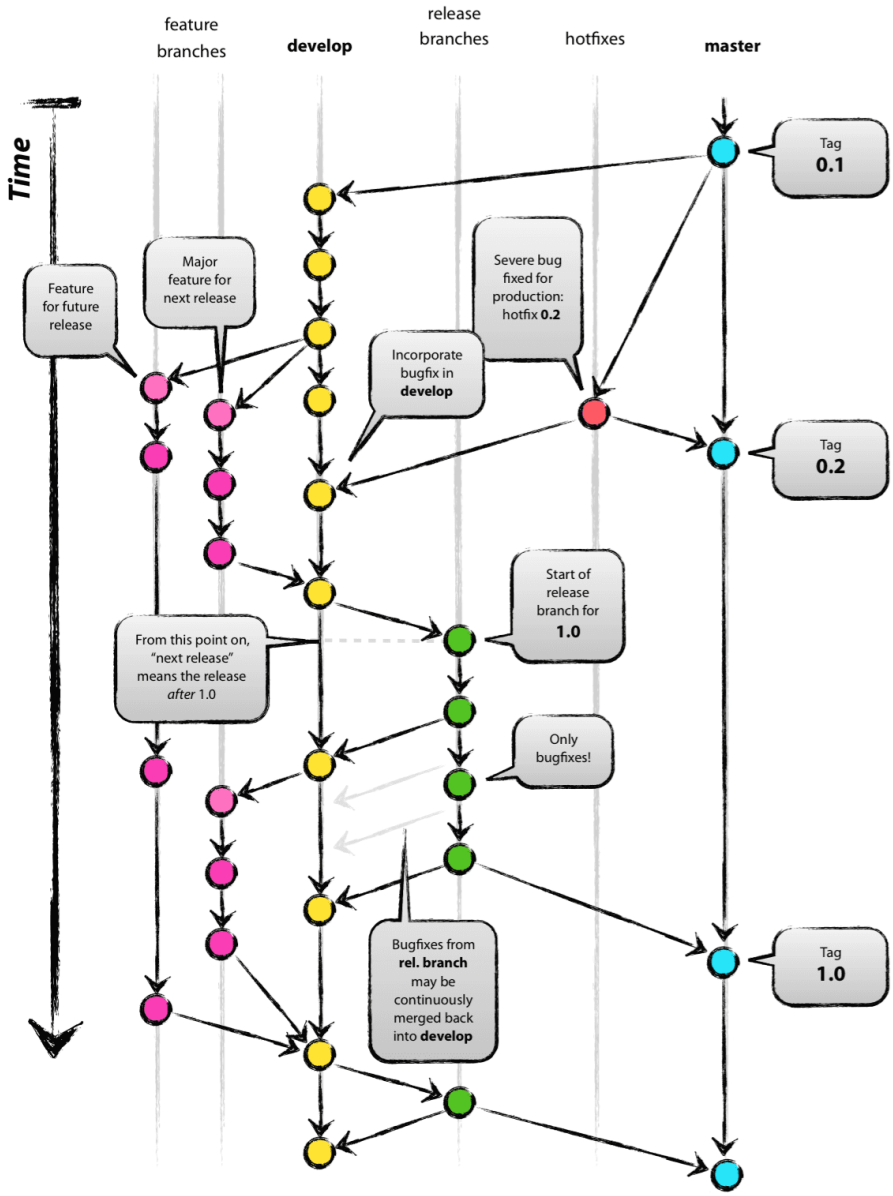

Figura 14 – Representação gráfica do modelo *gitflow*. Fonte: <https://nvie.com/posts/a-successful-git-branching-model/>

As vantagens do *gitflow* são perceptíveis desde a introdução do mesmo na comunidade de desenvolvimento de *software*. Sua divisão de *branches* permite e proporciona suporte para o desenvolvimento paralelo de funcionalidades enquanto se trabalha em modificações urgentes, entre outros pontos. Hoje, o *gitflow* é base para as mais diversas modelagens de fluxos de desenvolvimento utilizando o Git, sendo ensinado junto de vários tutoriais de uso do Git.(PIDOUX, 2014)

3.3.2 O *semantic versioning*

Assim como é feito o versionamento de artefatos de *software*, como código e documentação, um *software* em produção também é versionado. O *semantic versioning*, criado pelo co-fundador do Github e criador do Gravatar Tom Preston-Werner, tem como propósito evitar os problemas criados pela constante evolução das dependências usadas pelo *software* (PRESTON-WERNER, 2011).

Esses problemas são o que ele chama de "*dependency hell*", que é o ponto em que suas dependências acabam por levar seu projeto à estagnação por conta de incompatibilidade das mesmas.(PRESTON-WERNER, 2011)

Assim, Preston-Werner criou o sistema de versionamento semântico. Ele consiste em uma forma de versionamento seguindo o formato $MAJOR.MINOR.HOTFIX$, assim cada ponto da versão deve ser incrementado de acordo com o impacto das alterações na versão anterior (PRESTON-WERNER, 2011).

Seguindo esse formato, *MAJOR* se refere à alterações que trazem incompatibilidade com as versões anteriores. *MINOR* se refere à alterações que trazem novas funcionalidades, provendo ainda suporte à versão anterior de forma retroativa. *HOTFIX* se refere à correções de problemas, enquanto mantém o suporte à versões passsadas (PRESTON-WERNER, 2011).

A partir deste sistema, é possível manter as dependências atualizadas enquanto se previne incompatibilidades. Um programa que segue o versionamento semântico não trará incompatibilidades em sua versão 1.9.0 comparado com a versão 1.2.6.

Assim, é possível de se utilizar dependências menos fixas, enquanto se mantém a segurança de funcionamento das mesmas. Hoje, o *semantic versioning* é o padrão utilizado para o versionamento de *software*.

3.3.3 Documentação

Como dito no capítulo 1, a documentação de *software* presentes no repositórios, varia bastante de equipe para equipe, de projeto para projeto. Possivelmente, o fator causa para essa variação de opiniões seja a influência das metodologias ágeis no processo de documentação das equipes.

Diferentes equipes possuem diferentes opiniões sobre o que é documentação necessária para um repositório. Essa realidade é perceptivel no mundo de desenvolvimento de *software* livre, onde existem diversas equipes, com diversos projetos, e diferentes níveis de documentação associados.

Em uma tentativa de unificar a mínima documentação presente nos repositórios de software presentes em sua plataforma, o Github lançou o painel da comunidade, que

define uma lista de requisitos com alguns documentos que devem estar presentes para seu repositório ser considerado amigável à comunidade(FUKUI, 2017). Tal painel e lista de requisitos foram mostrados anteriormente no capítulo 1 e na Figura 2.

Outra iniciativa que surgiu para melhorar a documentação de *software* foi a da Linux Foundation, no qual também foi estabelecida uma lista de requisitos para verificar a qualidade da documentação e saúde do repositório para aceitar contribuidores externos (ABERNATHY IBRAHIM HADDAD, 2019). Tal lista define sete pontos principais para análise, sendo que cada um destes pontos possuem vários pontos de análise. Os itens são:

- Considerações;
- Planejamento e estratégia de negócio;
- Revisão legal;
- Revisão técnica;
- Governança e processos;
- *Branding* e *marketing*;
- Lançamento e manutenção;

Uma outra iniciativa, que surgiu da *Linux Foundation Core Infrastructure Initiative* - ou CII, é uma medalha para se colocar no seu repositório de *software*. Para conseguir tal medalha, um dos donos do repositório deve submeter o repositório link do repositório do Github para análise no website de boas práticas da CII. Existem vários níveis de medalhas, que passam por progressão à medida que são adotadas as boas práticas no repositório (CII, 2001).

A partir do momento de submissão, o repositório será analizado automaticamente em busca do cumprimento das boas práticas definidas pelo CII. As boas práticas analizadas por eles são separadas em seis pontos de análise, são eles (CII, 2001):

- Identificação, licença e documentação sobre o projeto;
- Controle de alterações, versão e *releases*;
- Relatórios de falhas e vulnerabilidades;
- Qualidade, Integração contínua e testes;
- Segurança de código;
- Análise estática de código;

Como podemos observar os pontos de análise são um pouco semelhantes à lista anterior, porém o processo automático de análise torna o processo bem mais confortável para os interessados em melhorar a qualidade da documentação de seus repositórios.

Porém, para se adequar ao definido como boas práticas, o processo pode ser bastante árduo devido à quantidade de pontos de análise e, mesmo caso esteja de acordo com um destes padrões a chance de não esta de acordo com outro é consideravel.

4 Análise de práticas adotadas por repositórios de software livre

Neste capítulo, serão visitados alguns grandes repositórios de software livre e analisadas algumas de suas práticas aplicadas em relação à documentação para novos contribuidores. Esses repositórios serão analisados a partir da plataforma do Github.

Vale ressaltar que esta análise será feita com base nas boas práticas do Github, presente no painel de *insight* de cada um desses repositórios, além dos pontos presentes na checklist do CII. Ambas as análises serão focadas na documentação dos repositórios.

A partir desses dados poderemos verificar semelhanças e diferenças entre cada um deles e assim procurar por padrões e abordagens diferentes entre cada um deles. Conforme dito anteriormente, vários repositórios de diferentes tamanhos possuem diferentes visões para com sua documentação e comunicação com seus contribuidores, assim poderemos documentar e comparar as características de suas abordagens.

Foram escolhidos para esta análise repositórios com comunidades ativas e que possuem bastante visibilidade e impactos no desenvolvimento de software . Os repositórios analisados foram os seguintes:

- Linux Kernel
- Linguagem Ruby
- Linguagem Python
- Gerênciador de pacotes npm
- Framework Ruby on Rails
- Framework Django

4.1 Linux Kernel

Como citado previamente, a ferramenta Git foi criada por Linus Torvalds para efetuar o gerenciamento de versões do kernel do Linux. Devido a sua importância no desenvolvimento de software livre, foi realizada a escolha deste repositório.

Atualmente este repositório possui 11.518 contribuidores, 321 *pull requests* abertos e 5.847 commits de 1.149 autores diferentes adicionados à branch master nos últimos 30 dias (GITHUB, 2021). Vale resaltar que o repositório não deixa sua lista de *issues* exposta. Com essas métricas astronômicas podemos entender a movimentação que este repositório possui, devido à massiva atividade da comunidade.

4.1.1 Documentação interna e *guidelines* do Github

Ao contrário do comum em relação à softwares livres, em especial aos com a comunidade mais novas, o arquivo README do repositório vai direto ao ponto, informando ao contribuidor que existem diversos guias para usuários e desenvolvedores do kernel, o instrui à como compilar localmente a documentação, expõe o hiperlink da mesma no website do kernel e, finalmente, direciona o contribuidor para a pasta de documentação presente na raiz do repositório.

Na raiz do projeto temos a pasta *Documentation* onde encontramos uma extensa lista de subpastas, cada qual contendo outra extensa lista de subpastas, com os mais diversos assuntos relacionados ao kernel. Seguindo a orientação do README na página inicial do repositório, seguimos para o arquivo de README presente em *Documentation/admin-guide*.

Neste arquivo, temos as informações básicas sobre a última release do kernel. Neste documento encontram-se as seguintes informações:

- Uma breve descrição sobre que é o Linux;
- Em que *hardware* o kernel pode ser executado;
- Uma breve seção sobre a documentação, na qual entraremos em detalhes à seguir;
- Instruções de compilação e instalação do kernel;
- Informações de onde encontrar com detalhes a lista com as versões de pacotes de dependências do kernel;
- Um extenso guia de contato para se caso algo der errado, com o contato para a lista de e-mail do repositório;

No referente à documentação temos o conteúdo a seguir, que deixa bastante clara a finalidade da documentação no repositório do projeto como referência técnica, e não como guia.

> *This README is not meant to be documentation on the system: there are much better sources available* (KERNEL, 2021b).

> *There are various README files in the Documentation/ subdirectory: these typically contain kernel-specific installation notes for some drivers for example. Please read the :ref:‘Documentation/process/changes.rst <changes>‘ file, as it contains information about the problems, which may result by upgrading your kernel* (KERNEL, 2021b).

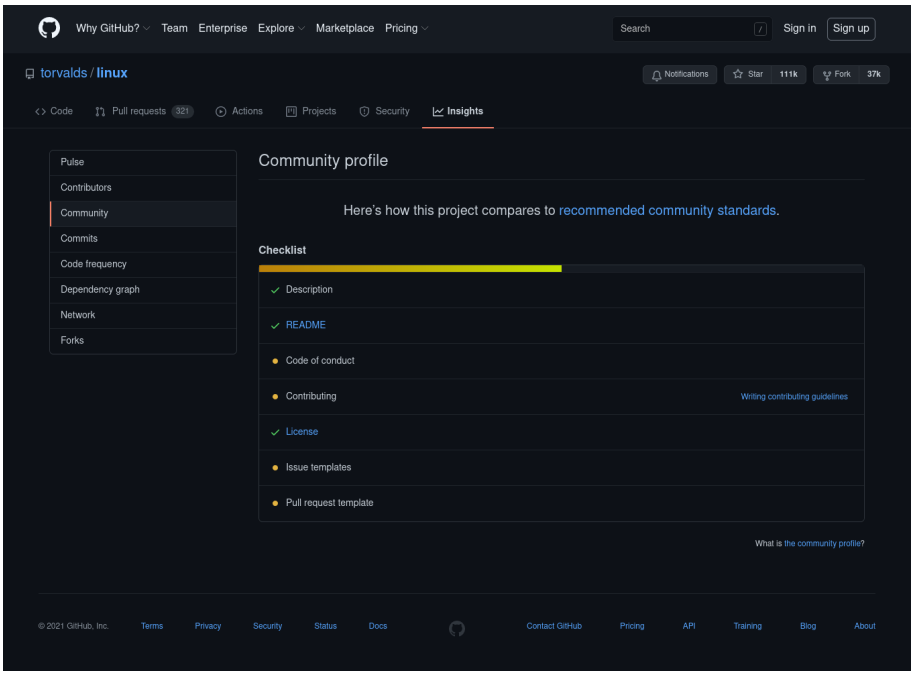

Figura 15 – Painel de comunidade do GitHub para o repositório do kernel do Linux.

No repositório do kernel no Github temos o painel de guidelines da comunidade indicando que apenas o básico se encontra presente no repositório - descrição, arquivo README e licença de software - como podemos observar na Figura 15.

Porém no README na página inicial do repositório temos o hiperlink para a documentação orientada para o usuário e para o desenvolvedor pode ser encontrada no website de documentação do Kernel, em <https://www.kernel.org/doc/html/latest/>.

Presentes nesta documentação temos uma seção dedicada à introdução de novos contribuidores ao desenvolvimento do Kernel, onde encontramos uma extensa lista de manuais e guias, separados nos seguintes temas:

- Trabalhando com a comunidade de desenvolvimento do Kernel;
- Ferramentas de desenvolvimento para o Kernel;
- Como escrever documentação do Kernel;
- Guias de *hacking* do Kernel;
- Tecnologias de *tracing* do Linux;
- Manual do manutenedor do Kernel;
- *Fault-injection*;
- *Livepatching* do Kernel;

Como o foco deste trabalho é a análise de documentações relacionadas à contribuição de software livre, daremos foco à documentação relacionada ao foco do trabalho presente no primeiro e terceiro item dos temas listados acima.

Na seção *Trabalhando com a comunidade de desenvolvimento do kernel* é possível encontrar guias sobre vários pontos do ciclo de contribuição, classificados de essênciais até guias técnicos (KERNEL, 2021c).

Aqui notamos a presença dos arquivos recomendados pelo Github que não são encontrados no repositório: o Código de conduta e o guia de contribuição, sendo este o documento atual e seus documentos anexos. Outros documentos, como guidelines de *pull requests* podem ser encontrados no manual do mantenedor do kernel, onde também temos comentários sobre *commits* (KERNEL, 2021a).

Outros documentos encontrados que merecem destaque e que possuem um grande impacto na transparência da organização do repositório ou ajudam o contribuidor estão listados abaixo.

- Guia para o processo de desenvolvimento do kernel;
- Estilo de código do kernel;
- Guia de submissão de *patches*;
- *Checklist* para a submissão de patches para o kernel do Linux;
- Números mágicos do Linux;
- Lista de manutenedores e como submeter mudanças no kernel;
- Como escrever documentação do kernel;

4.1.2 Análise das *guidelines* do CII

Tendo discutido os pontos relevantes do repositório e a compatibilidade do mesmo para com as guidelines do Github, passaremos agora para analisar com os critérios propostos pelo *CII*.

Ao analisar os critérios, percebemos que o Kernel abrange quase todos os pontos requeridos pela *CII*. As exceções, porém, foram as seguintes:

1. **Sugestões não acatadas**
   - **É sugerido que sejam usadas para lançamentos os formatos de *semantic versioning* ou *calendar versioning***: Embora seja parecido, o kernel não utiliza de nenhumas das duas formas de versionamento. Porém esse é um requisito opcional dentro das guidelines.
2. **Disparidade com as boas práticas**
   - **O projeto deve usar um *issue tracker* para se rastrear *issues* individualmente**: o kernel faz seu gerênciamento de issues através de listas de e-mail e não possui um *issue tracker* em sua pipeline, como é notado dentro da documentação do mesmo:

     > *Problem is: the Linux kernel lacks a central bug tracker where you can simply file your issue and make it reach the developers that need to know about it. That's why you have to find the right place and way to report issues yourself. You can do that with the help of a script (see below), but it mainly targets kernel developers and experts. For everybody else the MAINTAINERS file is the better place* (KERNEL, 2021d).

Outro ponto que vale a pena citar é o fato de que as notas de lançamento entram na exceção dos requisitos de boas práticas, como previsto dentro da mesma para casos em que o usuários tipicamente não conseguem atualizar o software por conta própria.

4.2 Linguagem Ruby

A linguagem Ruby foi lançada em 1995 por Yukihiro Matsumoto com o intuito de ser uma linguagem visando ser agradável para o programador e inpirada nas linguagens Lisp, Smalltalk e Perl em conjunto com uma sintaxe complexa e expressiva e uma biblioteca rica, (FLANAGAN, 2008) a linguagem atingiu uma grande aceitação em 2006 e hoje se encontra na posição de 11º lugar entre as linguagens de programação em termos de crescimento e popularidade. (BV, 2021)

A linguagem é totalmente livre, em termos de uso, extensão, cópia e distribuição (LANG.ORG, 2021) criando por consequência, uma grande comunidade de desenvolvimento colaborativo.

Seu repositório no Github, apesar de ser um *mirror* do repositório principal no Subversion, possui 316 *pull requests* abertos, 336 contribuidores e nos últimos 30 dias 54 autores inseriram 261 *commits* na *branch master*. Números menos impressionantes que os do kernel, porém bastante significantes.

4.2.1 Documentação interna e análise das *guidelines* do Github

Na página inicial do repositório, o colaborador é recebido com um longo e informativo arquivo README.

Neste arquivo temos uma visão geral do projeto, como a descrição do mesmo, suas principais funcionalidades, dicas e guia de instalação, como se inscrever na lista de e-mail, onde tirar dúvidas, e *links* para documentação oficial em <https://ruby-lang.org> e para alguns documentos como o guia de contribuição e licença.

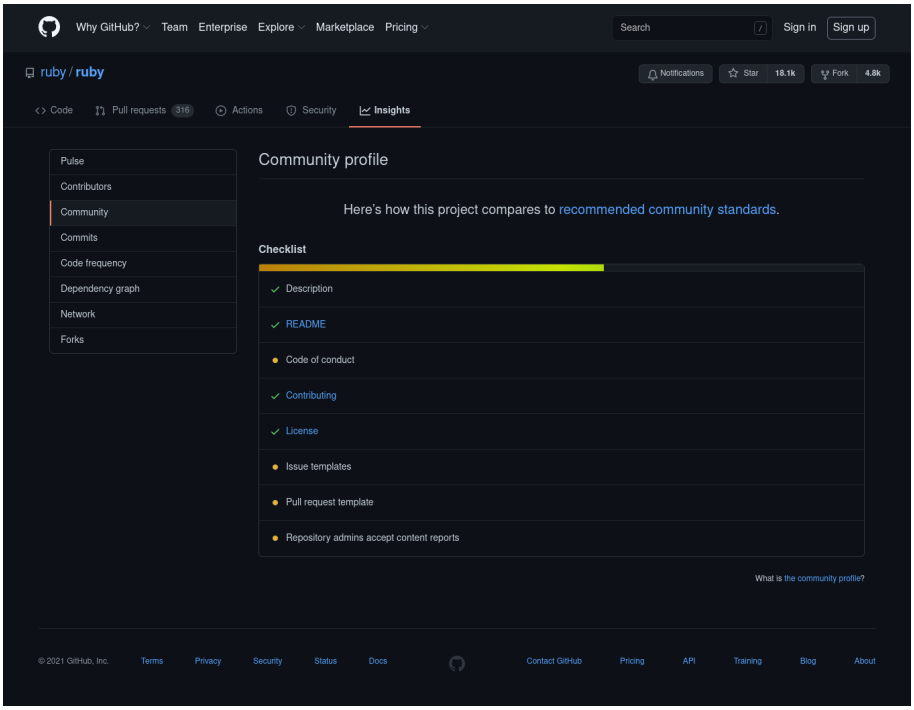

Figura 16 – Painel de comunidade do GitHub para o repositório da linguagem ruby

O repositório atende algumas das *guidelines* do Github como observado na Figura 16, não atendendo os pontos de *templates* para *issues e pull requests* além do código de conduta.

Com as *issues* desabilitadas no repositório, não faz sentido contarmos como falta um template para o mesmo, até por conta de no README termos vários links com

instruções para reportar problemas, sendo essas diferentes dependendo do S.O. em que o mesmo foi encontrado.

Dentro do repositório porém temos uma pasta nomeada *doc* que contém, entre outros, documentos relacionados à mais técnicos, como guias de contribuição e guias de síntaxe de escrita de códigos.

4.2.2 Documentação externa e análise das *guidelines* do *CII*

Externo ao repositório temos o website oficial da linguagem <https://www.ruby-lang.org/>, onde encontramos a documentação da linguagem e onde encontramos guias de contribuição, além de vários outros links para entrar em contato com outros desenvolvedores e com mantenedores, as formas de contato com a comunidade variam de lista de e-mail até servidor do *Discord*.

Também podemos encontrar o tracker de issues do projeto com seus próprios guias de como reportar bugs e vulnerabilidades da linguagem, que também possuem listas de e-mail separadas.

Além desta documentação, o Ruby possui também um website específico para a documentação técnica em <https://ruby-doc.org/>. Complementar a essa documentação, o website oficial da linguagem provê listas de trabalhos acadêmicos e livros sobre a linguagem.

Com base nos critérios propostos pelo *CII*, é possível afirmar perante a análise que o projeto da linguagem Ruby encontra é condizente com as boas práticas analisadas, não sendo encontrados pontos de discordância com a *checklist* de análise.

4.3 Linguagem Python

Criada por Guido van Rossum e lançada em 1991, a linguagem Python é uma das linguagens mais famosas no mercado atualmente, estando somente atrás da linguagem C e subindo de 3º para 2º lugar no período entre maio de 2020 e maio de 2021 (BV, 2021).

A linguagem conta com poderosas estruturas de dados e bibliotecas, além de suportar diversos paradigmas de programação (GOALKICKER.COM, 2021). A versão mais atual da linguagem é a 3.x, que será a versão na qual referiremos neste trabalho.

Além de ser a segunda mais popular linguagem de programação atualmente (BV, 2021), a linguagem Python é amplamente difundida em diversas comunidades de desenvolvimento e na academia, inclusive quando se trata de *machine learning* e processamento de dados.

A linguagem também é difundida como um excelente ponto de entrada para novos

programadores (HALL, 2009) e portanto uma das primeiras documentações de software em que um desenvolvedor virá a ter contato.

Atualmente o repositório da linguagem possui 1.431 *pull requests* em aberto e possuindo 1.540 contribuidores. Nos últimos 30 dias, foram submetidos 402 *commits* para a *branch master*, realizados por 127 autores, sendo um repositório bastante ativo.

4.3.1 Documentação do repositório e análise das *guidelines* do *Github*

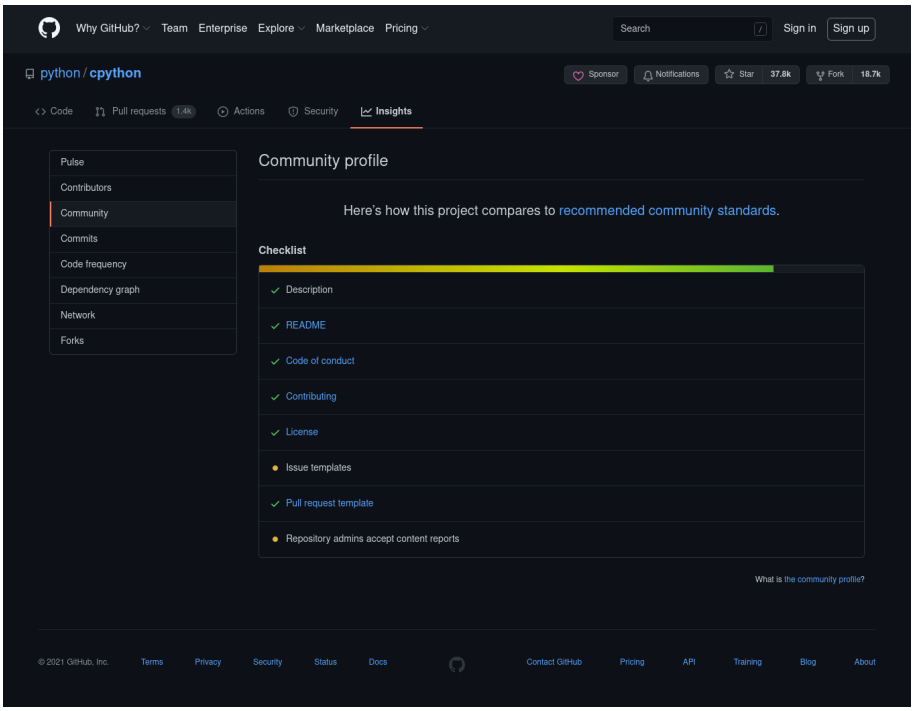

Figura 17 – Painel de comunidade do GitHub para o repositório da linguagem python.

No repositório do Github da linguagem, nomeado como **python/cpython**, temos os arquivos de README, código de conduta, guia de contribuição, licença e template de *pull request*. As seguintes informações podem ser obtidas no README:

- Informações gerais;
- Informações de como contribuir;
- Informações de uso;
- Instruções de compilação/instalação;
- Link para changelogs e outros logs de mudanças do projeto;
- Link para documentação externa e para documentação interna ao repositório;
- Guia para atualização da versão anterior;

- Guia para execução de testes e link para documentação aprofundada do assunto;
- Guia para instalação de múltiplas versões;
- Informações sobre o *Issue Tracker* e lista de e-mail;
- Guia para submissão de propostas de melhoria;
- Link para cronograma de lançamentos;
- Informações de licença e direitos de cópia;

A partir do que podemos perceber na Figura 17, o repositório segue bastante das *guidelines* sugeridas pela plataforma, sendo que somente o modelo de escrita de *issues* estaria faltando. Porém o repositório não utiliza o painel de *issues* da plataforma, não sendo necessária a presença do mesmo.

Outro ponto de observação é o fato de que vários dos documentos estão descritos de forma resumida e/ou são links para a documentação com maior carga de informações fora do repositório, hospedadas no site de documentação.

4.3.2 Documentação externa e análise das *guidelines* do *CII*

A documentação externa ao repositório da linguagem é hospedada no website <https://www.python.org/> onde podemos encontrar informações sobre eventos, lançamentos, suporte específico à plataformas, informações sobre implementações alternativas, guias de desenvolvedor, documentos relacionados à comunidade e mais informações quanto à contribuição ao projeto.

É possível encontrar informações sobre propostas de melhoria, além do *issue tracker* do projeto e as listas de e-mail para contato com a comunidade e mantenedores do mesmo. Ao realizar a análise com o checklist do *Core Infrastructure Initiative*, não foram encontradas discrepâncias com as boas práticas propostas.

Importante notar que na documentação interna do repositório não possuímos informações sobre o ciclo de vida de um *pull request*, que é o centro do workflow do projeto (FOUNDATION, 2021b).

Embora seja totalmente alinhada com o próprio repositório, essa documentação é acessada somente pelo guia de desenvolvedor presente em <https://devguide.python.org/> no capítulo 3.

O projeto possui um extenso e completo guia do desenvolvedor que possui 33 seções mais um apêndice, cobrindo desde onde o desenvolvedor deve buscar ajuda até o projeto do compilador do CPython.

4.4 Npm

Fundado em 2014 e recentemente adquirido pelo Github, o npm é um gerenciador de pacotes para bibliotecas e módulos JavaScript para o Node e é dividido em três pontos: (NPM, 2021a)

- **npm:** é o gerenciador de pacotes.
- **npm Registry:** é a coleção disponível publicamente de pacotes e bibliotecas da comunidade de desenvolvimento em JavaScript. É o maior registro de código do mundo. (NPM, 2021b)
- **npm CLI:** é a aplicação cliente que gerência os pacotes na máquina do desenvolvedor e o ajuda a publicar tais pacotes no registry.

No cenário atual de desenvolvimento de software, o JavaScript é uma linguagem de programação presente em grande parte do cotidiano do desenvolvedor, tomando o 1º lugar no ranking de linguagens de programação mais populares na pesquisa do StackOverflow de 2020 (STACKOVERFLOW, 2020).

A linguagem permaneceu em 7º lugar no índice da Tiobe nos últimos 12 meses estando, em termos de popularidadade na frente de linguagens como Php (9º), Ruby (11º), Swift (18º) e Perl (19º). (BV, 2021)

Estaremos nesta seção avaliando o repositório do **npm CLI**. O repositório foi recentemente transferido de *npm/npm* para *npm/cli*, e possui 698 contribuidores. Nos últimos 30 dias 12 autores publicaram 84 *commits* na *branch latest*.

O repositório possui no momento 727 *lançamentos* publicados, 832 *issues* e 5 *pull requests* em aberto.

4.4.1 Documentação do repositório e análise das *guidelines* do *Github*

A documentação no repositório é direta, o README já aborda os requisitos, uso da ferramenta e como instalar, além de um breve FAQ sobre a *branding* e uma seção de links e recursos. Esta última com âncoras para a documentação, o *bug tracker*, o *roadmap*, feedback, calendário de eventos, suporte, entre outros.

Com exceção do link para o *bug tracker*, que direciona para o painel de *issues* do repositório, todos os outros links levam à documentações externas ao mesmo.

Estas referências variam entre outros repositórios do npm (roadmap, *RFCs* e feedback), página hospedada no GitHub Pages (status do projeto - <https://npm.github.io/statusboard/>), calendário do Google (calendário de eventos) e páginas do website oficial do npm (demais links).

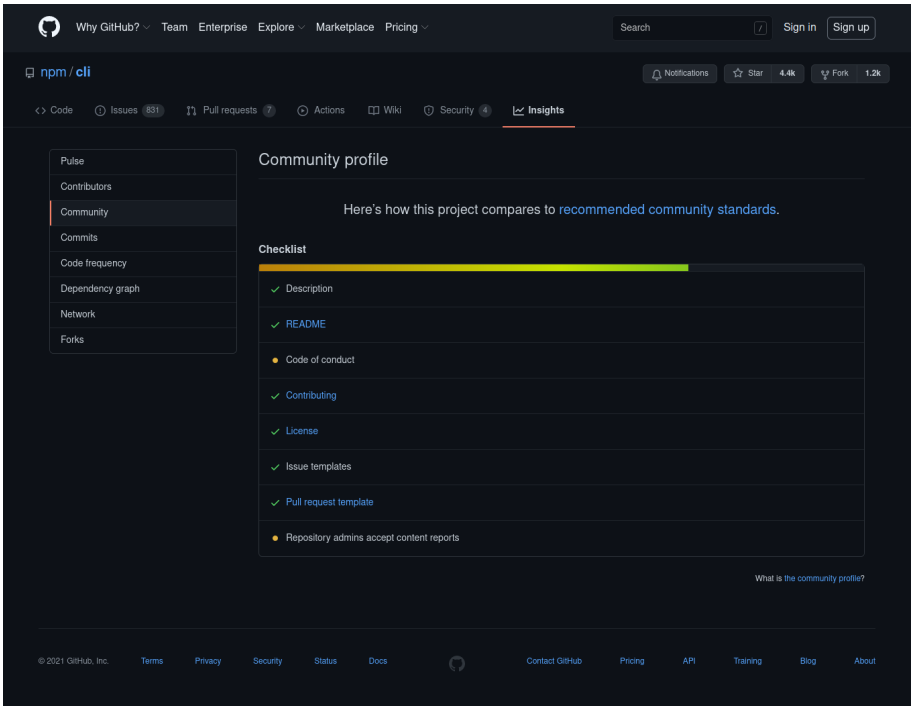

Figura 18 – Painel de comunidade do GitHub para o repositório do npm CLI.

Conforme visualizado no painel de comunidade, figura 18, vemos que a maioria dos pontos de análise é cumprida, com exceção do código de conduta. Este documento porém possui uma referência externa ao mesmo dentro do arquivo CONTRIBUTING.

Outro ponto é o fato do repositório possuir uma pasta *docs* na raiz do projeto, com algumas documentações sobre a ferramenta e uma pasta *changelogs* que armazena a lista de changelogs passados do projeto. Na raiz do mesmo temos o changelog atual no arquivo CHANGELOG.

4.4.2 Documentação externa e análise das *guidelines* do *CII*

Embora o site oficial do npm seja em <https://www.npmjs.com/>, a documentação do mesmo se encontra no subdomínio *https://docs.npmjs.com/*, devidamente exposto e a mostra no website oficial.

A documentação externa é extensa e abrange todos os três produtos que fazem parte do npm. A documentação relacionada ao CLI possui links próprios e pode ser facilmente achada na página inicial da documentação. A mesma abrange seus comandos, a configuração da mesma e guia de uso da ferramenta, sendo navegável por versões.

Documentos relacionados à contribuição e comunidade não são encontrados junto à documentação do produto, estes valem para todo o ecossistema do npm e se encontram separados em outro subdomínio, que infelizmente não é facilmente encontrado no subdomínio de documentação e no website principal não é encontrado facilmente, embora seja

citado que o npm é OSS e aceita contribuições.

Em <https://npm.community/> temos o painel de comunidade do npm, com os mesmos links encontrados no README do repositório, acrescido de um link para o fórum arquivado do npm.

O documento de código de conduta do npm pode ser encontrado na parte de políticas do website oficial, em <https://www.npmjs.com/policies/conduct>. Estando bastante separado dos outros pontos de documentação.

Realizando a análise das guidelines do *CII*, temos algumas incompatibilidades que podem ser vistas abaixo. Vale observar também que o repositório possui uma grande quantidade de *issues* que não foram respondidas nem comentada pelos mantenedores, algumas datam desde 2019.

1. **Disparidade com as boas práticas**

   - **O site do projeto deve explicar o processo de contribuição**: Não possui explicação sobre o processo de contribuição.
   - **O site do projeto deve incluir os requisitos que definem uma contribuição aceitável:** Assim como o item anterior, o site não possui este documento.
   - **O software produzido deve ser lançado como FLOSS:** O npm é somente OSS, e não FLOSS.

Embora sejam um número significativo, não respondidas sendo 44,29% para melhorias e 36,48% para bugs, eles ainda não violam o proposto pelo CII, que determina que pelo menos 50% sejam respondidas.

4.5 Framework Ruby on Rails

O Ruby on Rails, é um *framework* de desenvolvimento web open source que segue a arquitetura em camadas MVC. Atualmente se encontra na versão 6.1.3.2 e é um dos frameworks de desenvolvimento web mais populares. (STACKOVERFLOW, 2020)

O Rails é escrito em Ruby e é um framework bastante poderoso, focado em permitir uma maior produtividade ao desenvolvedor, com uma grande biblioteca integrada e a facilidade de escrita proporcionada pela linguagem.

A comunidade do framework é bastante integrada à da linguagem Ruby e bastante ativa, com diversos eventos e palestras todo ano em diversos países do mundo. É usado por grandes empresas como Basecamp, HEY, GitHub, Shopify, Twitch, SoundCloud, Hulu, Zendesk, AirBNB entre outros (RAILS, 2020).

Outro ponto importante é que esse projeto, assim como as linguagens Ruby e Python, são geralmente participantes em eventos como o *Google Summer of Code*, tendo um grande fluxo de novos desenvolvedores participando como contribuidores.

Seu repositório do GitHub possui 445 lançamentos, conta com 4.244 contribuidores e possui 351 *issues* e 252 *pull requests* abertos. Nos últimos 30 dias, 76 autores inseriram 204 *commits* na *branch master*, sendo um repositório bem ativo e mantido.

4.5.1 Documentação do repositório e análise das *guidelines* do *Github*

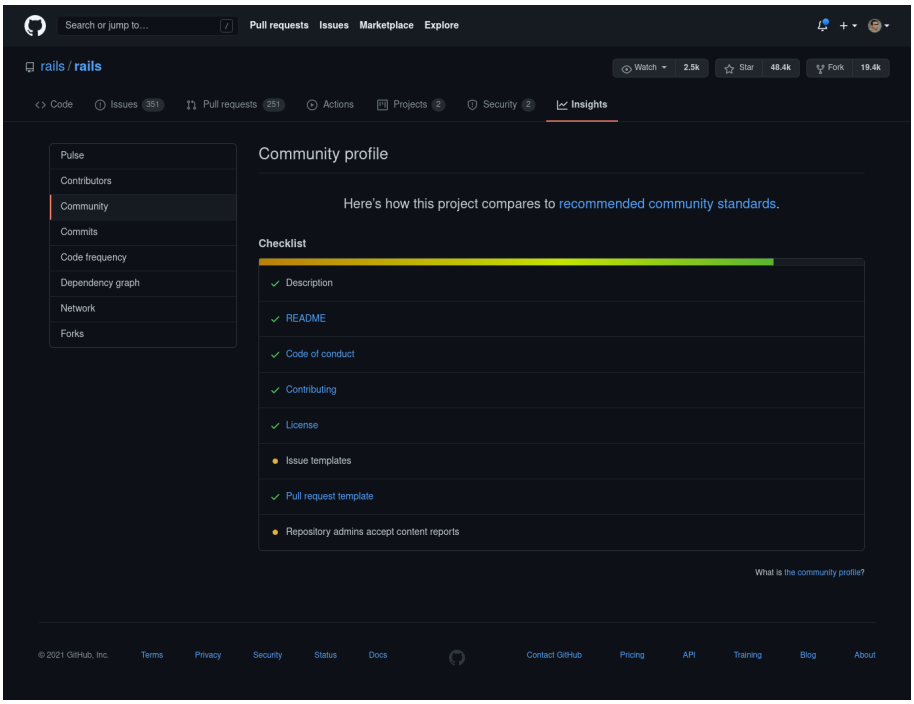

Figura 19 – Painel de comunidade do GitHub para o repositório do Ruby on Rails.

No repositório encontramos no README várias informações relevantes, como: descrição da arquitetura, como instalar e iniciar um projeto com o framework, onde encontrar guias e documentação (externos), a licença utilizada e como contribuir.

A documentação do repositório é bem focada no que diz respeito ao repositório, mantendo apenas o que é referente à contribuições. Como pode ser visto na figura 19, o repositório atende a todas as exigências das boas práticas do Github exceto a da presença de *issue templates*.

Porém, embora conste no painel que o repositório não possua tal *template*, o repositório de fato o possui e o mesmo pode ser encontrado no caminho *.github/issue_template.md*.

Esta dispariedade ocorre devido ao Github ter alterado a forma de criação dos mesmos e o presente no repositório, embora ainda funcione como template, foi criado

anteriormente à essa mudança. Portanto, o repositório cobre todos os pontos de boas práticas do Github.

4.5.2 Documentação externa e análise das *guidelines* do *CII*

O Rails possui como website principal o <https://rubyonrails.org/>, que possui diversas âncoras que redirecionam o usuário para vários assuntos de interesse ao projeto:

- Link para o blog próprio onde são apresentadas a novas atualizações, entre outros assuntos;
- Link para o subdomínio onde são apresentadas a documentação e guias de uso;
- Link para documentação para a API do mesmo;
- Link para o fórum de discussão;
- Link com a apresentação do time e informações sobre a comunidade;
- Link para contribuição, que leva o usuário para a página do Github;

Além destes apresentados acima, a página inicial também possui outros links diretos para documentos de políticas como código de conduta, licença, política de manutenção, segurança e marca. A navegação é bem clara e dividida em vários assuntos diferentes e específicos, ajudando a encontrar as informações buscadas.

O Rails possui uma documentação bastante extensa e completa, disponível em várias línguas diferentes. Dentre estas documentações fica em especial o *Rails Guides* (<https://guides.rubyonrails.org/>), que possui uma ampla gama de exemplos e guias de uso das funcionalidades do framework.

Quanto às boas práticas do CII, o Rails possui algumas inconformidades quanto ao sugerido pelo mesmo. Essas inconformidades inclusive são referentes ao seu website, são essas:

1. **Disparidade com as boas práticas**
   - **O site do projeto deve explicar o processo de contribuição**: Não possui explicação sobre o processo de contribuição no website, sendo este presente apenas no repositório, porém o website possui referência navegável para a documentação presente no repositório.
   - **O site do projeto deve incluir os requisitos que definem uma contribuição aceitável:** Assim como o item anterior, o site não possui este documento, mas um link para a documentação presente no repositório.

4.6 Framework Django

O Django é um framework para o desenvolvimento de aplicações web em Python, que promove um desenvolvimento rápido e performático, contando com as diversas bibliotecas presentes na linguagem.

Sua arquitetura é a *Model View Template*, ou MVT, que é bastante parecida à arquitetura MVC adotada pelo Rails. O Django se encontra entre os frameworks de desenvolvimento web mais populares (STACKOVERFLOW, 2020) e é utilizado por diversas empresas para suas aplicações, entre elas estão a Mozilla, Instagram, National Geographic e o Pinterest (FOUNDATION, 2021a).

O repositório possui 322 lançamentos, 2.044 contribuidores e possui 169 *pull requests* em aberto. As *issues* estão desabilitadas no repositório. Nos últimos 30 dias 30 autores publicaram 94 *commits* para a *branch master*.

4.6.1 Documentação do repositório e análise das *guidelines* do Github

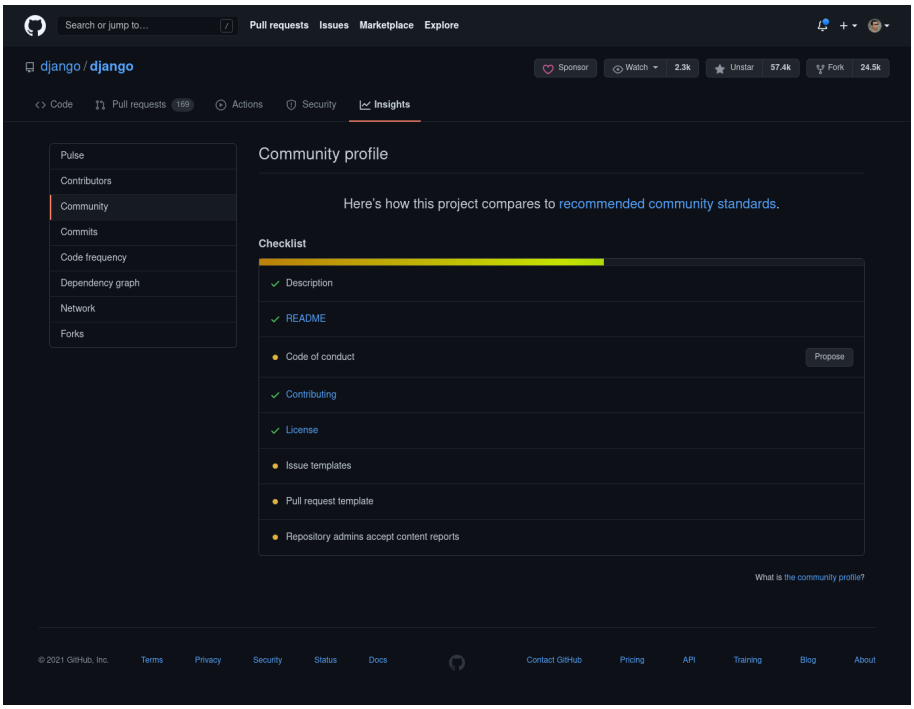

Figura 20 – Painel de comunidade do GitHub para o repositório do *framework* Django.

O README do projeto possui de forma bastante sucinta informações sobre o que é o projeto, onde está localizada a documentação da ferramenta tanto internamente e externamente ao repositório, onde conseguir ajuda, onde encontrar a documentação relacionada à contribuidores, onde contribuir financeiramente para o projeto e como executar os testes.

Dentro do repositório temos a pasta *docs*, onde temos a mesma documentação encontrada no site oficial quanto ao framework, porém em formato de texto. Instruções de como gerar essa documentação em formato html local são encontradas no README presente na pasta.

Quanto às boas práticas propostas pelo Github, o repositório falha em alguns pontos, como a falta de documento de código de conduta e *template* de *pull request*. Por conta de não utilizar a funcionalidade de *issues* no repositório, podemos ignorar a falta de *templates* para as mesmas.

4.6.2 Documentação externa e análise das *guidelines* do *CII*

Em <https://www.djangoproject.com/> temos o website oficial do framework, onde podemos encontrar o centro de documentação do projeto. Conforme dito na seção 4.6.1, alguns dos documentos requisitados pelas diretrizes do Github são encontradas nessa documentação.

O website possui âncoras para visão geral, documentação, notícias, código fonte, o *issue tracker* do projeto, sobre e para doações para o projeto. Seguindo o link da documentação, temos o tutorial e documentação do *framework*.

No link de documentação temos informações quanto ao fórum de discussões, e as listas de e-mail para contribuidores e usuários, além de um link paraum guia para contribuidores localizado em <https://docs.djangoproject.com/en/dev/internals/contributing/> onde são descritos os processos e políticas para a contribuição ao projeto.

Realizando a análise das diretrizes do CII, temos os seguintes pontos observados:

1. **Pontos não verificados:**
    - **O projeto deve tomar conhecimento da maioria dos *bug reports*/pedidos de melhoria submetidos nos últimos 2-12 meses:** o *issue tracker* funciona por tickets, e com filtros não usuais o que requeriria uma análise manual em um grande volume de tickets. Portanto não realizada;
2. **Disparidade com as boas práticas**
    - **É sugerido o uso do *Semantic Versioning* ou *Calendar Versioning* para numeração de releases**: O Django utiliza de um sistema de nomenclatura de releases muito parecido com o *Semantic Versioning*, porém não existe separação entre lançamentos MAJOR e MINOR;

5 Síntese

5.1 Sínteses individuais dos repositórios

No Capítulo 4 foi feita uma análise de como estavam organizadas e quais documentações se encontravam presentes nos repositórios escolhidos. À seguir serão apresentadas as sínteses e observações colhidas das análises

5.1.1 Linux Kernel

Em resumo, é possível perceber que o projeto do kernel do Linux abrange vários documentos que vão bastante além do recomendado pelo Github. Além disso, possui uma cobertura bastante abrangente das boas práticas propostas na checklist do *CII*, entrando em desconformidade em apenas uma.

O kernel é um projeto com uma comunidade bastante madura e extremamente ativa, o que leva a entender que a experiência acumulada deles no desenvolvimento colaborativo dentro de sua comunidade levou à decisão de que esta documentação era necessária, o que leva a questionar se com uma comunidade deste porte, o uso de uma ferramenta de *issue tracking* realmente é necessária.

5.1.2 Linguagem Ruby

Considerando os pontos levantados nas seções anteriores, é possível perceber que o projeto da linguagem Ruby atende as especificações sugeridas pelo Github, e do *CII*.

Além desses pontos é válido resaltar na extensa divulgação de livros relacionados à linguagem e suas boas práticas, além de eventos e conferências de comunidade presentes na página da linguagem, que pode ser acessada em: <https://www.ruby-lang.org>.

5.1.3 Linguagem Python

A documentação do projeto CPython é bastante extensa e clara para com o desenvolvedor que busca contribuir com o projeto, entrando inclusive em concordância com ambas as *guidelines*. Dentre os outros repositórios análisados este foi o primeiro a termos uma maior abrangência com o proposto pelo Github.

A documentação externa em si é bastante completa e extensa sendo várias vezes citada e ancorada na documentação presente no repositório. Assim como visto na linguagem Ruby na seção 4.2, temos bastante divulgação de eventos voltados à comunidade na página oficial da linguagem.

5.1.4 *CLI* do npm

A partir da análise, podemos ver que o repositório npm segue de forma mais completa as diretrizes do GitHub e acaba por entrar em disparidade com as boas práticas do CII.

Alguns documentos possuem dificuldades de acesso diferentes dependendo de onde o desenvolvedor inicia a procura do mesmo à exemplo do código de conduta, que se procurado a partir do repositório se acha um link direto, mas no painel de comunidade do site oficial do npm o mesmo não se encontra facilmente.

5.1.5 *Framework* Ruby on Rails

O Rails possui uma documentação bem separada entre seu repositório, website principal, website de guias e website de documentação da API, cada qual com sua área de foco e possuindo conteúdo bastante completo para a sua finalidade.

Isso acabou por gerar uma acordância perfeita com as guidelines do Github, mas não com as guidelines do CII. Por conta de estar bem separada a documentação, informações com presença exigida pelo CII no website principal são encontradas apenas como referência externa, que foi o caso da documentação explicando o processo de contribuição.

5.1.6 *Framework* Django

Com a análise do repositório do Django temos que o mesmo possui dispariedades mínimas com ambas as *guidelines* análisadas. Sendo a da documentação do repositório (falta do documento de código de conduta e de *template* para *pull requests*) sendo suprida, em parte na documentação externa.

Deixando de lado a disparidade com ambas as *guidelines*, o repositório apresenta uma documentação bastante abrangente, sendo que os pontos levantados na seção anterior em sua maioria são de impacto mínimo.

5.2 Síntese

Considerando o que foi visto na seção anterior a respeito das análises realizadas em cima dos repositórios, podemos visualizar que, pelo menos no espaço amostral escolhido, os seguintes pontos se apresentam como constantes:

1. É comum ter-se ***issue tracker* externalizados no projeto**.

2. É comum a presença de **pastas internas ao projeto para documentação**.

3. Diversas ferramentas do Github **não são comumente** utilizadas, como o *issue tracker* e a *wiki*.

Foi observado que dentre os repositórios análisados poucos utilizam o espectro total das ferramentas disponibizadas pelo Github, tais como o painel de *issues* e painel do *github projects*.

Também foi observado que alguns repositórios utilizam de uma pasta de documentação no repositório. O conteúdo dessa pasta, como vimos nos repositórios das seções 4.1.1, 4.3.1, 4.4.1 e 4.6.1 varia bastante, indo desde especificações bastante técnicas até documentação de uso.

Quanto ao uso de outras funcionalidades presentes no Github, dado o tamanho dos repositórios manter centralizada a informação e documentação internamente à plataforma faz pouco sentido, como seria com o uso de recursos como o *github wiki*, ou no caso do *issue tracker*, as razões de uso para a opção para um externo podem ser bastante variadas, desde maior funcionalidades e integrações para visibilidade de problemas existentes no software.

Outro fato observado foi de que, salvas pequenas inconsistências como o uso sugerido de licenças aprovadas pela *Open Source Iniciative*, os repositórios apresentam uma grande conformidade com ambas as *guidelines* de boas práticas, tendo em alguns casos divergência apenas na localização das documentações.

Justamente por conta destes pontos conflitantes, um projeto de software livre que deseja seguir ambas as *guidelines* acabará por ter duplicadas alguns pontos de sua documentação, ou acabará por ficar em desacordo com uma delas.

6 Conclusão e trabalhos futuros

6.1 Conclusão

Neste trabalho foram apresentados conceitos relacionados ao gerênciamento de versões de código, suas ferramentas, plataformas e conceitos relacionados, além de apresentadas algumas das boas práticas existentes na comunidade.

Com base nas *guidelines* apresentadas, foi feita uma análise em seis grandes repositórios de software livre ativos e populares para se verificar se e como esses repositórios implementam essas boas práticas, tendo como foco a documentação dos mesmos, principalmente relacionada à contribuidores externos.

Ao realizar estas análises, foram levantados os pontos de acordo e discrepância de cada um destes projetos com as boas práticas, além de pontos interessantes de documentações não presentes nas *guidelines*.

Embora possua um espaço amostral pequeno, a conclusão retirada dessa pesquisa é a de que em repositórios de desenvolvimento colaborativo maduros existe um alto nível de acordância com as propostas de boas práticas estabelecidas por ambas as *guidelines*.

Na maior parte das vezes, as discrepâncias encontradas são motivadas por questões de localização. Isso mostra uma inconsistência do modelo de boas práticas apresentado por ambas as *guidelines*. Como por exemplo a localização do código de conduta ou do guia de contribuição, ora citado para ser localizado no repositório, ora no website do projeto.

Outro fato que podemos concluir é que as *guidelines* do Github podem ser consideradas fracas. Isto ocorre por conta do mesmo considerar apenas a presença dos documentos no repositório e não possuir nenhum requisito de conteúdo.

Isso cria um problema pois faz com que os repositórios mantenham arquivos simbólicos na sua documentação dentro do repositório, como vemos em alguns dos analisados. Documentos vazios com *links* para os documentos em si não deveriam entrar como válidos para as boas práticas.

6.2 Trabalhos futuros

Embora tenha sido realizada uma análise em cima de repositórios de software importantes e de porte relevante, este trabalho pode ser bastante enriquecido ao aumentar o espaço amostral do mesmo, abordando não só repositórios de grande porte e com comunidades bem amadurecidas.

Assim abordando e analisando também repositórios agrupados em temas similares ou não, com diversos tamanhos e realizando comparações entre os mesmos, a fim de melhor visualizar os padrões discrepantes e comuns entre cada espaço amostral.

A criação de uma proposta de *guideline* para melhorar a qualidade da análise feita pelo Github para com a sua documentação é de grande interesse para com os desenvolvedores da plataforma. Assim seria evitado o problema de documentos simbólicos citado na Seção 6.1.

Referências

ABERNATHY IBRAHIM HADDAD, G. M. J. M. J. S. C. *Starting an opensource project*. 2019. <https://www.linuxfoundation.org/resources/open-source-guides/starting-open-source-project/#7>, Acessado em 05 de novembro de 2019. Citado 2 vezes nas páginas 20 e 40.

ANGULAR. *Angular Contributing Guide*. 2011. <https://github.com/angular/angular/blob/master/CONTRIBUTING.md>, Acessado em 3 de dezembro de 2019, documento original: <https://docs.google.com/document/d/1QrDFcIiPjSLDn3EL15IJygNPiHORgU1_OOAqWjiDU5Y/edit#>. Citado 3 vezes nas páginas 30, 31 e 32.

BV, T. S. *TIOBE Index for May 2021*. 2021. <https://www.tiobe.com/tiobe-index/>, Acessado em 4 de maio de 2021. Citado 3 vezes nas páginas 47, 49 e 52.

CHACON, B. S. S. *Pro Git*. 2. ed. [S.l.]: Apress, 2009. Citado na página 29.

CII. *CII Best Practices Badge Program*. 2001. <https://bestpractices.coreinfrastructure.org/en>, Acessado em 6 de dezembro de 2019. Citado na página 40.

CONVENTIONALCOMMITS. *Conventional Commits*. 2016. <https://www.conventionalcommits.org/en/v1.0.0/>, Acessado em 30 de novembro de 2019. Citado 3 vezes nas páginas 11, 30 e 34.

DAWSON, C.; STRAUB, B. *Building Tools with Github*. 2. ed. [S.l.]: O'Reilly Media, Inc., 2016. ISBN 978-1-491-93350-3. Citado na página 17.

DRIESSEN, V. *A successful Git branching model*. 2010. <https://nvie.com/posts/a-successful-git-branching-model/>, Acessado em 5 de dezembro de 2019. Citado 2 vezes nas páginas 37 e 38.

FLANAGAN, Y. M. D. *The Ruby Programming Language*. 1. ed. [S.l.]: O'Reilly Media Inc., 2008. ISBN 0-596-51617-7. Citado na página 47.

FOUNDATION, D. S. *Django overview*. 2021. <https://www.djangoproject.com/start/overview/>, Acessado em 15 de maio de 2021. Citado na página 57.

FOUNDATION, P. S. *Lifecycle of a Pull Request*. 2021. <https://devguide.python.org/pullrequest/#introduction>, Acessado em 13 de maio de 2021. Citado na página 51.

FUKUI, K. *Post de lançamento do community tools*. 2017. <https://github.blog/2017-06-14-new-community-tools/>, Acessado em 25 de agosto de 2019. Citado 2 vezes nas páginas 18 e 40.

GITHUB. *Starting a opensource project*. 2019. <https://opensource.guide/starting-a-project/>, Acessado em 24 de agosto de 2019. Citado na página 18.

GITHUB. *Starting an opensource project - pre-launch checklist*. 2019. <https://opensource.guide/starting-a-project/#your-pre-launch-checklist>, Acessado em 05 de novembro de 2019. Citado na página 19.

GITHUB. *Web archive of torvalds/linux pulse at May 4th - 2021*. 2021. <https://archive.is/zUPTh>, Acessado em 4 de maio de 2021. Citado na página 44.

GOALKICKER.COM. *Python notes for professionals*. 2021. <https://books.goalkicker.com/PythonBook/PythonNotesForProfessionals.pdf>, Acessado em 12 de maio de 2021. Citado na página 49.

HALL, J.-P. S. T. *Python 3 for Absolute Beginners*. 1. ed. [S.l.]: Apress, 2009. ISBN 978-1-4302-1633-9. Citado na página 50.

HODSON, R. *Git Succinctly*. 1. ed. [S.l.]: Syncfusion Inc., 2012. Citado 2 vezes nas páginas 28 e 29.

HUTTERER, P. *On Commit Messages*. 2009. <http://who-t.blogspot.com/2009/12/on-commit-messages.html>, Acessado em 26 de novembro de 2019. Citado na página 28.

KARMA. *Karma - Git commit message*. 2014. <http://karma-runner.github.io/4.0/dev/git-commit-msg.html>, Acessado em 3 de dezembro de 2019. Citado 3 vezes nas páginas 11, 32 e 33.

KERNEL. *Creating Pull Requests*. 2021. <https://www.kernel.org/doc/html/latest/maintainer/modifying-patches.html>, Acessado em 4 de maio de 2021. Citado na página 46.

KERNEL. *README*. 2021. <https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/README.rst>, Acessado em 4 de maio de 2021. Citado na página 45.

KERNEL. *Working with the kernel development community*. 2021. <https://www.kernel.org/doc/html/latest/process/index.html>, Acessado em 4 de maio de 2021. Citado na página 46.

KERNEL, L. *Reporting issues*. 2021. <https://www.kernel.org/doc/html/latest/admin-guide/reporting-issues.html>, Acessado em 10 de maio de 2021. Citado na página 47.

KING, J. *On the Importance of Commit Messages*. 2018. <https://americanexpress.io/on-the-importance-of-commit-messages/#a-case-study>, Acessado em 25 de novembro de 2019. Citado na página 28.

LANG.ORG ruby. *About ruby*. 2021. <https://www.ruby-lang.org/en/about/>, Acessado em 4 de maio de 2021. Citado na página 48.

LOELIGER, J.; MCCULLOUGH, M. *Version Control with Git, Second Edition*. 2. ed. [S.l.]: O'Reilly Media, Inc., 2012. ISBN 978-1-449-31638-9. Citado 3 vezes nas páginas 17, 28 e 29.

NARęBSKI, J. *Mastering Git*. 1. ed. [S.l.]: Packt Publishing Ltd., 2016. ISBN 978-1-78355-375-4. Citado na página 29.

NPM, I. *About npm*. 2021. <https://www.npmjs.com/about>, Acessado em 13 de maio de 2021. Citado na página 52.

NPM, I. *About npm*. 2021. <https://www.npmjs.com>, Acessado em 13 de maio de 2021. Citado na página 52.

OLIVEIRA, M. F. de. Metodologia científica: um manual para a realização de pesquisas em administração. *Biblioteca da Universidade Federal de Goiás – Campus Catalão*, p. 22, 2011. <https://adm.catalao.ufg.br/up/567/o/Manual_de_metodologia_cientifica_-_Prof_Maxwell.pdf>. Citado na página 23.

OLSSON, A.; VOSS, R. *Git Version Control Cookbook*. 1. ed. [S.l.]: Packt Publishing Ltd., 2014. ISBN 978-1-78216-845-4. Citado 3 vezes nas páginas 17, 29 e 30.

OPENHUB. *Openhub compare repositories*. 2019. <https://www.openhub.net/repositories/compare>, Acessado em 26 de agosto de 2019. Citado na página 17.

PIDOUX, E. *Git Best Practices Guide*. 1. ed. [S.l.]: Packt Publishing Ltd., 2014. ISBN 978-1-78355-373-0. Citado 4 vezes nas páginas 28, 29, 37 e 38.

PRESTON-WERNER, T. *Semantic versioning*. 2011. <https://semver.org/>, Acessado em 5 de dezembro de 2019. Citado na página 39.

RAILS, R. on. *Ruby on Rails*. 2020. <https://rubyonrails.org/>, Acessado em 14 de maio de 2021. Citado na página 54.

SAINI, D. J. R. Significance of software documentation in software development process. *International Journal of Engineering Innovation Research*, 2014. Citado na página 27.

SILVERMAN, R. E. *Git Pocket Guide*. 2. ed. [S.l.]: O'Reilly Media, Inc., 2013. ISBN 978-1-449-32586-2. Citado na página 29.

SOMMERVILLE, I. Engenharia de software. In: _____. 3. ed. Rua Nelson Francisco, 26, Limão CEP 02712-100 – São Paulo – SP Brasil: Pearson Education do Brasil, 2011. cap. 1, p. 4. Citado na página 27.

STACKOVERFLOW. *StackOverflow Developer Survey*. 2019. <https://insights.stackoverflow.com/survey/2019>, Acessado em 26 de agosto de 2019. Citado 2 vezes nas páginas 18 e 27.

STACKOVERFLOW. *Developer survey 2020*. 2020. <https://insights.stackoverflow.com/survey/2020>, Acessado em 14 de maio de 2021. Citado 3 vezes nas páginas 52, 54 e 57.

SYMFONY. *Symfony Commit Conventions*. 2016. <https://symfony.com/doc/current/cmf/contributing/commits.html>, Acessado em 3 de dezembro de 2019. Citado 3 vezes nas páginas 11, 35 e 36.

ZOLKIFLI AMIR NGAH*, A. D. N. N. Version control system: A review. *Procedia Computer Science*, n. 135, p. 408–415, 2018. <https://www.sciencedirect.com/science/article/pii/S1877050918314819>. Citado na página 17.