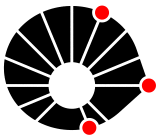

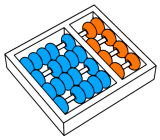

Avaliação do Nível da Aplicação de Práticas de Entrega e Integração Contínua em Repositórios de Código Aberto

G. A. Destro *B. B. N. França*

Relatório Técnico - IC-PFG-19-31
Projeto Final de Graduação
2019 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Avaliação do Nível da Aplicação de Práticas de Entrega e Integração Contínua em Repositórios de Código Aberto

Gabriel Augusto Destro[1], Breno Bernard Nicolau de França[2]

[1] g167744@dac.unicamp.br

[2] Instituto de Computação Universidade Estadual de Campinas (UNICAMP), Caixa Postal 6176

13083-970 Campinas-SP, Brasil

breno@ic.unicamp.br

Resumo. Esse projeto teve como objetivo desenvolver o CDMAP, uma ferramenta usada para caracterizar projetos disponibilizados em repositórios GitHub enquanto à adoção de práticas de integração e entrega contínua, por meio da coleta de dados públicos desses repositórios de software livre. Para isso, uma análise foi realizada para levantar as informações que poderiam ser extraídas do GitHub, e que fossem relevantes para a aplicação dessas práticas, fazendo uso de sua API REST e da clonagem direta do repositório. Concluído o desenvolvimento, foi feita uma análise de dois repositórios com o objetivo de verificar a capacidade da ferramenta de realizar a comparação entre o dados obtidos de cada um deles.

1. Introdução

A entrega e integração contínua são práticas de engenharia de software que visam desenvolver, testar e disponibilizar software de maneira regular, consistente e automatizada. Tais práticas são aplicadas em diversos projetos da indústria de software tendo como objetivo manter o qualidade e estabilidade da aplicação, assim tornando o processo mais eficiente e menos propenso a falhas. Elas são aplicadas a partir de um deployment pipeline, que é usado para automatizar os procedimentos envolvidos no desenvolvimento, compilação, testes e entrega de uma aplicação. Esse pipeline, entretanto, varia de acordo com o projeto, já que diferentes processos e ferramentas são usados de acordo com as tecnologias escolhidas e necessidades específicas de cada organização.

O GitHub é uma plataforma popular que faz uso do sistema de controle Git para hospedar código fonte de projetos públicos e privados, bastando que o usuário que deseja hospedar ou contribuir com um projeto possua uma conta na plataforma. Ele também possui uma API REST que pode ser usada para extrair informações relacionadas ao projeto que não poderiam ser obtidas apenas tendo acesso ao código fonte do mesmo. Portanto, é possível identificar repositórios de software livre que façam uso de práticas de entrega contínua, extraindo dados que sejam relevantes para a caracterização da aplicação de tais práticas.

Analisar esses repositórios com relação a aplicação de práticas de entrega contínua pode ajudar a compreender a maneira como elas são adotadas, o que pode resultar em uma melhor compreensão dos benefícios e limites associados a elas. Tal análise pode então, posteriormente, auxiliar estudos futuros que busquem melhorar a adoção e qualidade das mesmas por meio da mineração de repositórios abertos.

Logo, esse projeto teve como objetivo desenvolver uma ferramenta capaz de caracterizar projetos disponibilizados em repositórios GitHub no que tange à adoção de práticas de integração e entrega contínua, através da coleta de dados públicos desses repositórios de software livre. Para isso, uma análise foi realizada com base nas informações coletadas dos repositórios a respeito do nível de aplicação de práticas de entrega e integração contínua em tais repositórios. Após o desenvolvimento da ferramenta, foi realizada uma análise comparando dois repositórios. Essa análise teve como objetivo mostrar que a ferramenta poderia ser usada compará-los enquanto à aplicação de tais práticas. Além disso, a ferramenta desenvolvida durante esse projeto poderá ser disponibilizada publicamente e, posteriormente, usada em outros projetos que busquem realizar a extração desses dados para outras pesquisas.

2. Fundamentação Teórica

Humble e Farley [1] não apenas definem as práticas de Integração e Entrega Contínua, como também explicam como elas podem ser usadas de maneira eficaz através de padrões e boas práticas. Os autores definem a entrega e integração contínua como práticas de engenharia de software em que desenvolvedores constroem (do inglês, *build*), executam, testam e disponibilizam software de maneira regular, consistente e automatizada. Portanto, a partir dessa definição, é possível depreender que essas práticas englobam uma parte considerável do processo de desenvolvimento de software, entretanto, elas cobrem desde o momento em que a demanda tipificada chega até o time de desenvolvimento até o momento em que ela é entregue.

Assim, eles descrevem o deployment pipeline como o padrão central para o livro. Se tratando, em essência, de uma implementação automatizada dos procedimentos envolvidos no desenvolvimento, compilação, testes e entrega de uma aplicação. Entretanto, os autores descrevem esse pipeline como algo que varia de organização para organização, de equipe para equipe. Dessa forma, embora eles possuam os mesmos objetos e diretrizes, as equipes devem adaptá-lo de forma a suprir suas necessidades. O pipeline funciona de modo que cada mudança feita no código-fonte, ambiente ou dados de uma aplicação provoca a criação de uma nova instância desse pipeline. Um exemplo de pipeline, é descrito na figura 1.

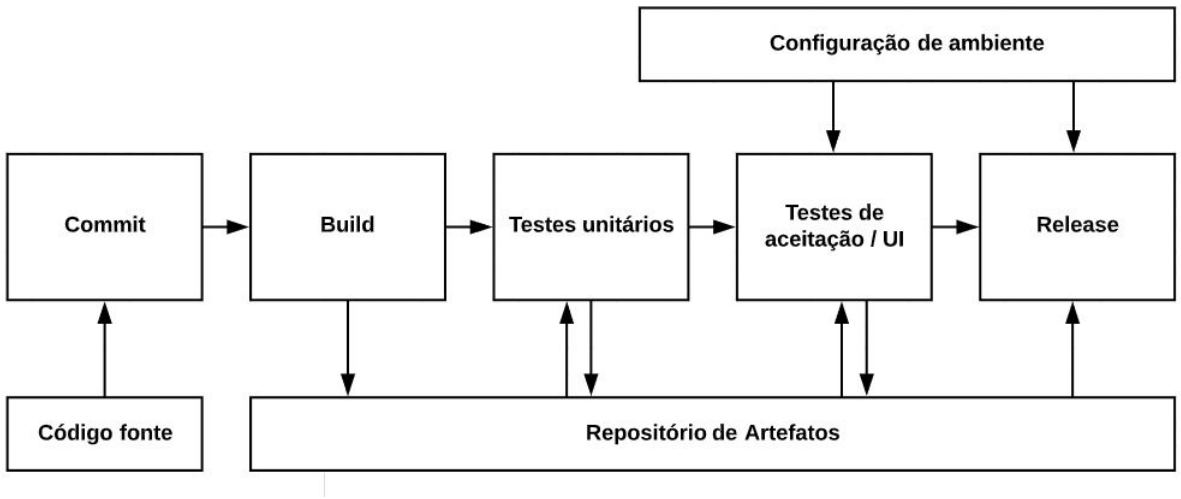

Figura 1. Exemplo de deployment pipeline (adaptado de [1]).

Nos primeiros capítulos, Humble e Farley [1] descrevem, de maneira geral, as principais práticas envolvidos na integração e entrega contínua, testes e na anatomia do pipeline de implantação. Portanto, essa caracterização é importante para compreender o funcionamento dessas práticas e assim, não só determinar quais seriam as informações relevantes a serem obtidas dos repositórios GitHub, como também entender métricas envolvidas que ajudassem a compreender o nível de aplicação dessas práticas.

Munaiah e colegas [2] descrevem o Reaper, uma ferramenta criada com o objetivo de avaliar projetos de software livre hospedados no GitHub em relação a adoção de boas práticas de engenharia de software. A ferramenta Reaper serviu de referência para compreender a maneira usada para a obtenção da informação desses repositórios, como o uso da API do GitHub e a obtenção de informações através do repositório clonado. Algumas das informações obtidas foram as mesmas em ambos os trabalhos, por exemplo, a identificação de ferramentas usadas para o gerenciamento de integração contínua e a cobertura de testes, portanto esse trabalho também foi útil para entender como extrair e organizar alguns desses dados.

A Tabela 1, faz uma breve comparação a respeito das informações extraídas entre o Reaper e a ferramenta CDMAP, desenvolvida nesse projeto. É importante levar em conta que ambas as ferramentas têm objetivos distintos. Enquanto o Reaper se propõe a obter informações gerais

sobre a aplicação de técnicas de engenharia de software, para então classificar o repositório como um projeto planejado ou não. A ferramenta deste trabalho busca indícios do uso de práticas de entrega contínua nesses repositórios, não fazendo nenhum tipo de classificação. Portanto, embora ambas usem formas similares de obtenção dos dados, o Reaper possui uma proposta mais abrangente.

Tabela 1. Comparação entre o Reaper e a ferramenta desenvolvida.

	REAPER	**CDMAP**
Integração Contínua	Apenas identifica se existe ou não.	Identifica e lista as ferramentas encontradas para CI.
Build, gerenciamento de dependências e integração de artefatos.	Não obtém informação se o repositório possui ou não tais ferramentas.	Identifica ferramentas e frameworks usados no repositório.
Documentação	Obtém o número de linhas de comentário por linhas de código.	Não extrai esse dado.
Histórico	Obtém a média de *commits* por mês.	Obtém a média e desvio padrão de *commits*, *releases* e *tags* por mês e dia.
Issues	Obtém o número médio de *issues* criados por mês.	Obtém as métricas para todos os *issues* e *pull requests*, além de separá-los pelos rótulos, dividindo-os em duas categorias: *bugs* e *features*. Para cada rótulo e categoria, é obtida a relação de *issues* fechados e abertos. Para cada cenário, são obtidos a média e o desvio padrão, obtendo também esses dados para a frequência de criação por dia e mês.
Licença	Identifica se o repositório possui uma licença.	Não extrai esse dado.
Testes automatizados	Obtém o número de linhas de código em classes de teste em relação ao número total de linhas de código. Faz isso apenas para testes unitários.	Obtém o número de classes de testes unitários, de UI e de aceitação em relação ao número de classes totais. Separa essa informação por *framework* encontrado.
Classificação do repositório	Classifica o repositório como um projeto “planejado” em termos de engenharia de software ou não.	Não faz qualquer classificação sobre do repositório.

Diferentemente do Reaper, esse projeto se propôs a criar uma ferramenta que obtivesse as informações relevantes apenas às práticas de integração e entrega contínua. Dessa forma, a parte do trabalho, como realizada em [2], referente a classificação desses repositórios não foi levada em consideração.

Finalmente, Mäkinen e colegas [3] conduziram um estudo com diversas organizações que desenvolvem software de diferentes áreas. Como resultado, os autores reuniram informações relevantes a respeito de como cada uma delas aplica as diferentes etapas do processo de desenvolvimento, buscando ao final, relacionar o nível de aplicação de tais práticas com a eficiência do processo de entrega de software. Junto a isso, os autores levantaram as ferramentas mais usadas para cada etapa do pipeline. Esses dados foram usados como base para definir quais ferramentas seriam suportadas inicialmente pela ferramenta desenvolvida nessa proposta e qual etapa do deployment pipeline ela buscava automatizar.

Nem todas as etapas da cadeia de ferramentas discutidas em [3] estão dentro do *deployment pipeline*, logo nem todas elas estão diretamente relacionadas com as práticas de entrega contínua. Portanto, algumas etapas foram desconsideradas. Em algumas etapas do pipeline não foi possível obter as informações referentes a utilização ou não da ferramenta pelo repositório. Isso acontece porque muitas vezes não existem traços da utilização de tais ferramentas no repositório.

A Tabela 2 (adaptada de [3]), organiza algumas das ferramentas utilizadas para cada uma das etapas do desenvolvimento, entretanto, trata-se de uma simplificação, pois omite etapas do pipeline não cobertas por este trabalho, assim como ferramentas mencionadas no trabalho original. A partir dessa tabela, é possível observar que a relação de ferramenta para etapa nem sempre é um para um, o que dificulta o processo de identificação da adoção da etapas do pipeline.

Tabela 2. Ferramentas usadas nas práticas de entrega contínua agrupadas por etapa do *deployment pipeline*.

Etapa	Ferramentas
Controle de Versão	• Git • Subversion
Construção (Build)	• Jenkins • Maven • Visual Studio • GCC
Integração Contínua	• Jenkins • TeamCity • Travis CI • Circle CI
Repositório de artefatos	• Artifactory
Gerenciamento de dependências	• Docker
Testes Unitários	• JUnit • Mockito • Mocha • Google Test
Testes de UI	• Selenium • Mocha • Cucumber
Testes de aceitação	• JUnit • Selenium • Cucumber
Deployment	• Jenkins

3. Metodologia

O desenvolvimento da ferramenta consistiu inicialmente em um estudo para compreender como dados que dessem indícios do uso das práticas envolvidas na integração e entrega contínuas poderiam ser obtidos a partir de repositórios GitHub. A revisão bibliográfica (ver Seção 2) ajudou a atingir o objetivo de compreender o tema, traçar quais dados seriam extraídos dos repositórios e as métricas estipuladas.

Uma vez definido os dados que seriam adquiridos, foi dado início ao desenvolvimento da ferramenta. Ao realizar uma busca, tendo inserido o nome do repositório e a chave de autenticação do GitHub, o servidor dá início a obtenção dos dados, em que são feitas as requisições para a API REST e a obtenção dos dados do repositório clonado. Uma vez obtida toda a informação necessária, é carregada uma página para ser exibida ao usuário, dando a possibilidade para que ele faça download de um arquivo JSON contendo os dados.

A obtenção da informação foi feita em duas etapas, a primeira consistiu na clonagem do repositório para extrair informações que não precisassem ser obtidas a partir da página do GitHub. A partir disso foram obtidos *commits*, *releases*, *tags* e *badges*. Assim como a obtenção de informação relativa às classes de teste e de ferramentas externas ligadas as etapas do pipeline.

A segunda etapa de obtenção consistiu no uso da API REST[1] do GitHub, que é uma API que retorna informações hospedadas na página com base na string de busca usada. Essa ferramenta foi utilizada para obter informações que estava hospedadas unicamente na página do repositório no GitHub, ou seja, não poderiam ser obtidas a partir do repositório clonado. Portanto, ela foi usada para a obtenção de tópicos e *issues*.

Após o desenvolvimento da ferramenta, foi realizado uma comparação entre dois repositórios com o objetivo de mostrar que essa ferramenta pode ser usada para tal finalidade. A adição da

[1] A documentação oficial da API REST do GitHub pode ser encontrada em: https://developer.github.com/v3/

funcionalidade de download do arquivo JSON, permite que o usuário ainda compare mais do que dois repositórios de cada vez, já que ele pode tratar esses dados usando outra ferramenta.

4. CDMap

O CDMAP foi implementado usando um servidor Spring Boot[2] que é utilizado para processar as requisições do usuário. Elas são processadas e os resultados são exibidos em páginas JSP (*Java Server Pages*).

Com relação aos métodos de obtenção das informações, para que fossem feitas a clonagem do repositório e a obtenção dos dados dependentes do Git, como *commits* e *tags*, foi usada a biblioteca JGit[3], que já faz o mapeamento e tratamento dos dados internos do Git. Já os dados disponíveis apenas na página do GitHub foram obtidos usando sua própria API REST. Para realizar as requisições REST e navegar pelo sistema de arquivos a procura das ferramentas, foram usadas as bibliotecas nativas do Java, plataforma usada para o desenvolvimento da ferramenta.

Para converter os objetos Java gerados a partir da coleta de informações, para então gerar o relatório em formato JSON, foi usado a biblioteca Jackson JSON, que faz o mapeamento direto das classes para JSON. A aplicação é hospedada usando um servidor Tomcat[4]. O Maven[5] foi usado como ferramenta de build e de gerenciamento de dependências e a ferramenta foi versionada usando o Git.

Com relação a construção da solução, a figura 2 mostra um pouco a maneira como como cada um dos módulos da ferramenta se comunicam. A requisição REST do cliente é tratada por um controlador, que solicita aos módulos “JGit Repository” e “GitHub API Client”, para que seja feita a obtenção dos dados. Enquanto o primeiro clona o repositório e obtém os dados referentes ao Git, o segundo faz requisições a API do GitHub.

---

[2] Spring Boot: https://spring.io/projects/spring-boot
[3] JGit: https://www.eclipse.org/jgit/
[4] Tomcat: http://tomcat.apache.org/
[5] Maven: https://maven.apache.org/

As ferramentas e frameworks são obtidos pelo módulo “External Tools”, que percorre os arquivos do repositório. Os dados são tratados pelo módulo “Métricas”, que agrupa os dados e realiza os cálculos das porcentagens, médias e desvios padrões.

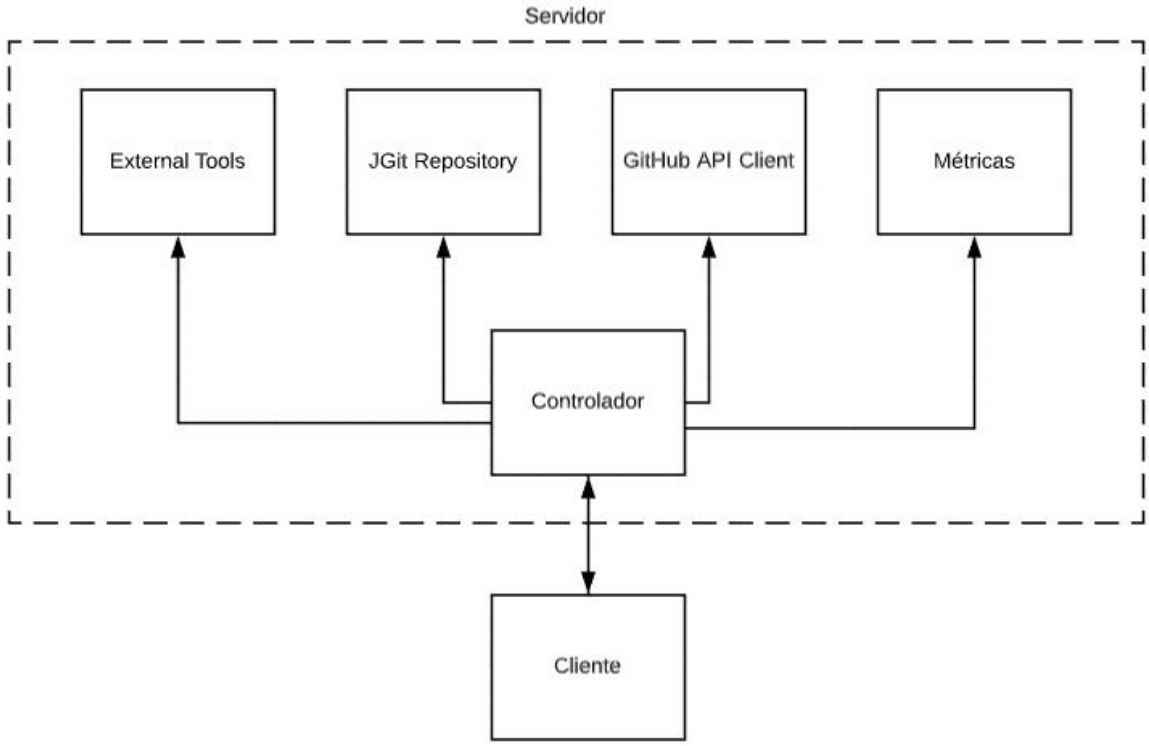

Figura 2. Organização dos principais módulos da ferramenta.

o fluxo de execução do CDMAP, que está presente na Figura 3, apresenta o processo de aquisição dos dados. Primeiramente é realizada a clonagem do repositório e captura das informações contidas nele que são referentes ao Git ou podem ser obtidas a partir do arquivo “Readme”. Depois, através da API REST do GitHub, são capturados os dados que estão disponíveis apenas na página do GitHub. Na próxima etapa, os arquivos e pastas do repositório são vasculhados a fim de detectar ferramentas e *frameworks* externos que possam ser usados para a aplicação das práticas de integração e entrega contínuas. Finalmente, é feita a montagem do relatório com os dados que são exibidos para o usuário convertidos para JSON.

Figura 3. Fluxo de execução da ferramenta.

Existem outras ferramentas que hospedam projetos relevantes de software livre, como o BitBucket, por exemplo, que também possui uma API REST própria. Entretanto, para esse trabalho, devido ao tempo demandado para suportar diferentes plataformas, apenas o GitHub utilizado dada sua ampla utilização e adoção pela comunidade de software livre.

Nesse caso, dois métodos foram usados para obter os dados dos repositórios. Primeiro, a API do GitHub, limitando-se a obter informações sobre *releases*, *issues*, tópicos e outros dados disponíveis apenas na plataforma online e não no repositório clonado. A partir dela são feitas requisições através da Web, e cada requisição retorna uma resposta no formato JSON, que então é tratado seguindo os padrões da documentação oficial da API para obter os dados.

O segundo método de obtenção dos dados é através do repositório clonado pela ferramenta, e tem como objetivo a economia de requisições a API, já que há um limite[6]. Algumas das informações obtidas são referentes aos *commits* e *tags* do repositório, *badges* são obtidas a partir do arquivo README e os *hooks* locais na pasta ".git". A partir dele também são obtidas informações relativas às ferramentas externas usadas no projeto. Tais ferramentas nos dão indícios como: cobertura de testes, partes do deployment pipeline implementadas e *frameworks* adotados. Para a obtenção de ferramentas dependentes da linguagem de programação usada, elas apenas serão detectadas caso o projeto esteja implementado usando a plataforma Java. Até o momento, essa é a única linguagem de programação suportada pela ferramenta.

[6] Limite de uso da API do GitHub:
https://developer.github.com/apps/building-github-apps/understanding-rate-limits-for-github-apps/

Para a obtenção das ferramentas externas, foi dado suporte a um conjunto de ferramentas separadas de acordo com seu uso em cada etapa do pipeline. A figura 4 apresenta o pipeline adotado, assim como as ferramentas suportadas inicialmente para cada uma das etapas.

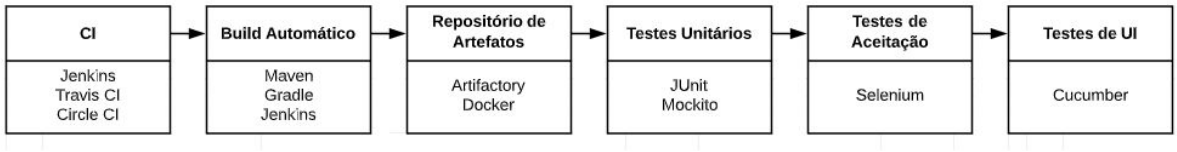

Figura 4. Pipeline adotado com as ferramentas suportadas para cada etapa.

As ferramentas foram obtidas realizando buscas pelos arquivos do repositório. Em alguns casos, como ferramentas de CI para automatização do *build* como o Jenkins[7] e Travis CI[8], foram procurados por arquivos específicos no projeto, como procurar arquivos com o nome Jenkinsfile, no caso do Jenkins, se o arquivo for encontrado, foi considerado que ele usa a ferramenta, dando um indício que o projeto em questão usa uma ferramenta relacionada a essa etapa do pipeline.

Para outros tipos de *frameworks*, como o Cucumber[9], foram procurados por extensões específicas. Mas para a maioria dos frameworks suportados, como *frameworks* de teste, como JUnit[10] e Mockito[11], a maneira de identificar foi verificando cada um dos arquivos para aquela linguagem de programação e procurando pelos imports de cada uma dessas ferramentas. Isso é feito buscando por todos os arquivos com a extensão da linguagem de programação usada, por exemplo ".java", então são analisados os imports de cada arquivo. Ao encontrar a linha "*import org.junit.*;*", a aplicação detecta que essa classe faz uso do *framework* JUnit, por exemplo. Essas ferramentas foram selecionados a partir de [3], em que o autor separa diferentes ferramentas para diferentes tipos de etapas do pipeline.

Também foram obtidas as *badges*, elas são uma espécie de *tags* que dão algum tipo de informação a respeito do projeto, por exemplo, do estado *build* automático, se a última tentativa

[7] Jenkins: https://jenkins.io/
[8] Travis CI: https://travis-ci.com/
[9] Cucumber: https://cucumber.io/
[10] JUnit: https://junit.org/junit5/
[11] Mockito: https://site.mockito.org/

foi bem-sucedida ou não. Elas são obtidas procurando suas referências através de links externos no arquivo README do projeto. Para obter essa informação, foram usadas expressões regulares, por exemplo "*!\\[(.*?)\\]\\((.*?)\\)*", para extrair as variáveis contidas nos link, quando disponíveis, e outras informações que foram obtidas seguindo a padronização encontrada através dos repositórios que as usavam.

Para identificar ferramentas de repositório de artefatos, foi dado suporte ao Artifactory[12] e ao Docker[13], para isso, foram verificados os arquivos de gerenciamento de dependências. Assim foi detectado seu uso caso estivesse listado como uma das dependências do projeto. Isso no entanto, não é garantia de que ele seria reconhecido em todos os casos.

Também foi desenvolvida uma funcionalidade que permite que o usuário faça o download de um arquivo (formato JSON) contendo os dados exibidos na sumarização da análise. Ela permite que o usuário possa tratar os dados de uma maneira mais simples, dando a possibilidade de que ele analise diversos repositórios simultaneamente usando um script próprio, por exemplo.

5. Discussão e Resultados

5.1 Aquisição de dados e desenvolvimento da ferramenta

Uma das dificuldades encontradas na obtenção das informações através da API do GitHub foi o limite de requisições que podem ser feitas por hora: 60 requisições para usuários não autenticados e 5000 para usuários autenticados, sendo que o limite pode ser estendido caso a organização em questão atenda certos requisitos, como possuir mais de 20 usuários, ou mais de 20 repositórios. Nesses casos, ganham-se mais 50 requisições por hora para cada usuário por repositório ou usuário registrado, sendo 12.500 requisições por hora o limite máximo. Para saber quantas requisições ainda podem ser realizadas, é possível fazer uma consulta via API. Devido à quantidade escassa de requisições disponibilizados pela API para usuários não autenticados, a aplicação obrigada que o usuário informe, junto com o endereço do repositório, um token de

[12] JFrog Artifactory: https://jfrog.com/artifactory/
[13] Docker: https://www.docker.com/

acesso, pois para repositórios maiores esse número de 60 requisições se esgota rapidamente. Tal chave pode ser cadastrada no GitHub. Considerando essa limitação, a aplicação foi construída de modo a tentar buscar o mínimo de informações necessárias dessa API.

Com a ferramenta, coletamos os *commits* de todas as *branches*. Dessa maneira, é possível ter uma ideia da frequência com que a base de código da aplicação é incrementada. Embora não seja possível dizer com certeza o momento em que é iniciada a execução do pipeline, é comum que ele seja executado a cada novo commit em *branches* específicas, como a *master*, já que *branches* de *feature* e desenvolvimento normalmente não disparam o pipeline. Portanto, uma maior frequência pode também significar que a aplicação é testada mais frequentemente no caso de testes automáticos serem executados. Um efeito colateral dessa maneira de obter essa informação é que também entram nos cálculos os *commits* que não são incrementos, como *merges* e *releases*, entretanto, é razoável supor que esse tipo de *commit* seja uma porcentagem pequena de todos os *commits*. Como o objetivo é apenas obter um indício do nível de aplicação das práticas, isso não é crítico.

Ao tentar obter os *releases* do repositório, alguns problemas foram encontrados. Primeiramente, a chamada da API do GitHub usada para obtê-los não funciona corretamente para todos os projetos testados. Haviam casos em que, ao fazer a requisição, não eram retornados nenhum dado ou então apenas o release mais recente, mesmo que na interface Web do GitHub um número maior de releases fossem listados. Nesse caso, a alternativa foi, a partir do repositório clonado, obter também as *tags* para todas as *branches*. Embora não seja possível garantir que uma *tag* esteja necessariamente associada a um *release*, essa é uma prática comum. Diferenciar *tags* que são ou não *releases* é uma tarefa complexa, pois depende do modelo de *branches* usado no projeto. *Releases* podem ser feitos na *master*, ou em outra *branch* de *release.* Ainda, é possível que cada *release* tenha um *branch* novo, aliado a isso, não existe um padrão de nomes para *tags* de *releases*. Esse conjunto de fatores dificulta a tentativa de identificar se uma *tag* está ou não associada a um *release*, mas pode ser considerada uma aproximação.

Outro problema encontrado referente aos *releases* é que nem todo release é associado a uma entrega de software, mas pode, por exemplo, ser uma entrega da documentação associado a um

release de software. Portanto, foi considerado que um *release* se refere a algum tipo de entrega, podendo ser uma entrega de software, de documentação ou mesmo um *release candidate*, ou seja, um *release* que em algum momento possa ser removido caso não seja aprovado.

Algumas dessas informações obtidas podem ser traduzidas em métricas que indicam o nível de aplicação dessas técnicas. A partir dos dados dos *commits* e *releases*, é possível obter o número médio de *commits* por dia e por mês, o tempo médio entre eles, assim como o desvio padrão. A frequência de *releases* é uma métrica importante, pois nessa prática, idealmente, o tempo entre entregas devem ser frequentes e constantes. A frequência de *commits* também é relevante, pois os incrementos no código devem ser constantes de modo a serem constantemente testados através da inicialização do pipeline.

O GitHub também possui uma sessão de *issues* que pode auxiliar na documentação do projeto. Essa seção pode conter informações referentes a *bugs* e *features*. Como nem todos os repositórios hospedados usam a ferramenta de *issues*, esse dado não pode ser obtido para todos os projetos. Existem organizações que usam ferramentas externas para organizar as *issues*. Nesse caso, obtemos esses dados apenas para aqueles projetos que fizessem uso da ferramenta do GitHub.

A partir dos *issues* obtidos também foram possíveis obter algumas dessas métricas, como a relação de *issues* fechados para o total de *issues* e o tempo médio para que eles fossem encerrados, o que foi chamado de *Lead Time*, sendo considerado o tempo desde que uma ideia é concebida, como uma *feature* nova ou um *bug* descoberto, até o momento em que ele é finalizado. A partir desses dados é possível filtrar alguns que pudessem trazer alguma informação de destaque, como *bugs* e *features* através das rótulos.

Para isso, buscamos por palavras-chaves, por exemplo, "*bug*" e "*feature*", nas *labels* das *issues* de maneira a agrupar esses grupos relevantes. Nesse caso, é possível capturar a relação de *bugs* encerrados para o total de *bugs* e o tempo médio e desvio padrão até que eles fossem fechados, sendo feito o mesmo para as *features*. Caso essas *labels* não sejam identificadas, a informação

não pode ser obtida. Esses dados são importantes para ter uma ideia da capacidade da entrega de software desde o momento em que ele foi inicialmente documentado.

A respeito do método usado para a obtenção dos dados de ferramentas e *frameworks*, ele foi considerado mais adequado do que procurar pelas classes do *framework* ou verificar se eles existem em alguma ferramenta de controle de dependências como o Maven ou o Gradle[14], pois além de ser um método mais assertivo, ele também nos fornece mais informações. Identificar a quantidade de classes que importam determinado *framework* é relevante pois permite que se tenha uma noção da quantidade de classes que usam esse framework em relação a quantidade de classes total do arquivo. Ainda, isso permite ter uma noção da cobertura de testes do projeto, sendo testes automatizados uma etapa importante da prática de integração contínua.

Uma dificuldade enfrentada nessa etapa é identificar exatamente em que etapa do deployment pipeline cada ferramenta está sendo usada, já que não existe uma descrição do pipeline externalizada nas ferramentas de CI e uma mesma ferramenta pode ser usada em diferentes etapas. Embora não seja possível dizer exatamente qual etapa do pipeline aquela ferramenta cobre, muitas vezes a ferramenta é usada majoritariamente para aquela etapa, ou mesmo quando usada para outras etapas, são etapas relacionadas. Por exemplo, o *framework* de testes JUnit pode ser usado para diferentes tipos de testes, embora seja majoritariamente usado para testes unitários, mas ele não pode ser usada para automação de *build*. É necessário considerar também que o uso da ferramenta não garante que aquela etapa do pipeline é efetivamente coberta, a métrica de cobertura de código, comparando o total de classes de teste com o número de classes totais ajuda a compreender melhor o uso do *framework*.

Para os *badges*, a dificuldade foi justamente definir adequadamente qual seria a padronização usada. Por conta disso, algumas informações podem ter ficado de lado, já que em um mesmo repositório, nem todo badge usava o mesmo padrão. Alguns desses *badges* possuíam urls que retornavam um JSON que continham algumas informações. Entretanto, devido a falta de padronização, essa informação não foi obtida.

[14] Gradle: https://gradle.org/

Até o momento, para a obtenção das informações que são dependentes da linguagem de programação do projeto apenas a linguagem Java é suportada. Isso se deve ao fato de que ao buscar pelos arquivos é necessário conhecer sua extensão. Além disso, diferentes linguagens possuem diferentes sintaxes, o que significa que o regex que realiza a busca deve mudar para cada linguagem. No entanto, não seria complexo dar suporte a novas linguagens de programação, bastaria alterar a ferramenta para que fosse feito uso do padrão Strategy, por exemplo.

5.2 Comparação de repositórios

Realizamos uma análise de dois repositórios com o objetivo de evidenciar a possibilidade de comparar repositórios em relação a adoção das práticas de integração e entrega contínuas com base nas informações coletadas. Os repositórios escolhidos para análise foram o ExoPlayer[15] e o Litho[16].

O número de contribuições e de contribuidores fornece uma noção da atividade e escala do projeto, além da popularidade daquele repositório. Nesse caso, ambos os repositórios têm a mesma escala de grandeza. Na análise, realizada no dia 28/11/2019, foram detectados 144 contribuidores e 6670 contribuições no ExoPlayer contra 199 contribuidores e 8509 contribuições no Litho.

Em relação às badges e os tópicos, eles podem ser úteis para se ter uma visão geral do projeto, embora compará-las com as de outro repositório talvez não seja relevante. Nessa análise, apenas o Litho possuía badges e o ExoPlayer apenas tópicos.

[15] Repositório do ExoPlayer no GitHub: https://github.com/google/ExoPlayer
[16] Repositório do Litho no GitHub: https://github.com/facebook/litho

(a) (b)

Figura 5. A esquerda (a), temos as badges encontradas para o repositório do Litho. A direita (b), os tópicos encontrados para o repositório do ExoPlayer.

Também foi possível comparar quais etapas do pipeline estabelecido os repositórios cobriam com base nas ferramentas suportadas pela ferramenta. A tabela 3 apresenta as ferramentas encontradas em cada etapa do pipeline para ambos os repositórios.

Tabela 3. Ferramentas detectadas para cada etapa do pipeline.

	CI	**Build**	**Repositório de Artefatos**	**Testes Unitários**	**Testes de Aceitação**	**Testes de UI**
ExoPlayer	-	Gradle	-	JUnit, Mockito	-	-
Litho	Circle CI	Gradle	-	JUnit, Mockito	-	-

A Tabela 4 apresenta os dados obtidos referentes a *commits*, *tags* e *releases*. Foram obtidos a média e o desvio padrão para o intervalo de tempo entre cada liberação, a quantidade por dia e por mês. Para o repositório do ExoPlayer, a API do GitHub não retornou nenhum *release*, apesar de que, no momento dessa análise, a página do repositório no GitHub mostra que ele possui 154 releases, que é o mesmo número de tags detectadas.

Tabela 4. Comparação entre commits, tags e releases de ambos os repositórios.

	Intervalo	Por Dia	Por Mês	Intervalo	Por Dia	Por Mês	Intervalo	Por Dia	Por Mês
ExoPlayer									
	Commits (8508)			Tags (154)			Releases (-)		
Média	5h 36m	4	130	11d 4h	0	2	-	-	-

D. Pad.	11m	106	38	1m	-	9	-	-	-
Litho									
	Commits (10130)			Tags (55)			Releases (38)		
Média	2h 18m	10	326	10d 11h	0	3	23d 16h	0	1
D. Pad.	0h 16m	1575	266	1m	-	5,1	1m	-	4,12

Com relação ao uso de testes, em ambos os repositórios, apenas *frameworks* tipicamente usados para testes unitários foram encontrados. Ambos os repositórios fazem uso do JUnit e do Mockito. É possível notar a partir das porcentagens que todas as classes que faziam uso do JUnit também usavam o Mockito no ExoPlayer.

As proporções entre arquivos com a extensão java e arquivos de teste, estão disponíveis na tabela 5. Como não foram encontrados *frameworks* de teste de UI ou de aceitação, essas categorias foram omitidas. Arquivos de teste são arquivos java que continham em seus *imports* a referência para um *framework* de testes.

Tabela 5. Comparação entre os dados de teste para cada um dos repositórios.

	Arquivos Java	**Arquivos de Teste**	**Porcentagem**	**JUnit**	**Mockito**
ExoPlayer	891	199	22,33%	22,33%	2,92%
Litho	1309	288	22%	21,7%	9,17%

Os *issues* foram divididos em duas categorias, *features* e *bugs*, para cada uma delas, os *issues* foram divididos entre abertos e fechados, sendo obtidos a quantidade e a porcentagem em relação a quantidade total, também foi obtido o *Lead Time*. Entretanto, para ambos os repositórios, nenhum *issue* categorizado como *feature* foi encontrado. Além disso, foi identificado que, ao retornar as *issues*, também eram obtidos os *pull requests*. A Tabela 6 apresenta os dados obtidos dos *issues* de ambos os repositórios.

Tabela 6. Comparação entre os dados obtidos para os *issues* de cada repositório.

	Geral	Abertos	Fechados	Lead Time	Pull Requests
ExoPlayer					
Todos	6684	362 (5,42%)	6322 (94,58%)	40d 22h	800
Bugs	977	80 (8,19%)	897 (91,81%)	77d 10h	-
Litho					
Todos	610	60 (9,84%)	550 (90,16%)	64d 15h	319
Bugs	3	1 (33,33%)	2 (66,67%)	152d 14h	-

Entretanto, comparar repositórios dois-a-dois pode não ser um método eficiente quando se deseja comparar os dados de vários repositórios simultaneamente. Para isso, o JSON disponível para download na ferramenta pode ser interpretado por outra ferramenta externa de modo a comparar informações de diversos repositórios simultaneamente.

6. Conclusão

Com o desenvolvimento dessa ferramenta, foi possível obter uma série de informações acerca do nível de aplicação de práticas de integração de entrega contínua de repositórios de software livre hospedados no GitHub.

Embora exista a limitação do número de requisições realizadas a API do GitHub, tal limitação pode ser contornada através da clonagem do repositório na máquina. Os dados também foram obtidos em um intervalo de tempo relativamente curto, visto que bastou alguns minutos para que fossem adquiridas as informações dos repositórios testados, embora esse tempo possa ser menor ou maior dependendo do tamanho do repositório em questão e do volume de dados.

Também foi observado que através dessa ferramenta é possível realizar a comparação entre repositórios. Isso foi feito através da análise dos projetos ExoPlayer e Litho, hospedados no

GitHub. Tendo em vista que a análise de repositórios dois-a-dois não é a ideal para analisar um grande número de repositórios simultaneamente, também foi disponibilizado para download o JSON dos dados obtidos.

Entretanto, é necessário levar em conta a limitação dos métodos usados para a aquisição dos dados analisados. Além do CDMAP dar suporte a um número limitado de ferramentas, a maneira como a ferramenta detecta tais *frameworks* em alguns casos também é dependente da linguagem de programação utilizada pelo projeto. Logo, dar suporte a *frameworks* e linguagens de programação novas geram um trabalho adicional.

7. Trabalhos Futuros

A expansão dessa ferramenta, dando suporte a mais linguagens de programação, ferramentas e *frameworks* é o principal objetivo futuro. Para isso, além do trabalho necessário para prover a cobertura de novas tecnologias, será necessário realizar a fatoração de alguns módulos para tornar essa inclusão mais simples. Também será necessário fazer isso levando em conta e escalabilidade da ferramenta, tendo em vista que um aumento no número de tecnologias suportadas pode aumentar consideravelmente o tempo necessário para coletar as informações do repositório.

Além disso, algumas melhorias podem ser realizadas na forma de obtenção de algumas das métricas, como melhorar o cálculo do tempo médio entre *commits* para levar em conta o paralelismo provocado por dois ou mais contribuidores trabalhando em *branchs* diferentes simultaneamente.

Com relação a performance, paralelizar tarefas que são independentes entre si pode diminuir consideravelmente o tempo de espera. Realizar otimizações na busca dos dados pela API do GitHub pode contribuir para a redução do número de requisições.

Referências

[1] Jez Humble and David Farley. Continuous delivery: reliable software releases through build, test, and deployment automation. 2011, Pearson Education, Inc.

[2] Nuthan Munaiah, Steven Kroh, Craig Cabrey, Meiyappan Nagappan. Curating GitHub for engineered software projects. 18 April 2017, Springer Science+Business Media New York 2017.

[3] Simo Mäkinen, Marko Leppänen, Terhi Kilamo, Anna-Liisa Mattila, Eero Laukkanen, Max Pagels e Tomi Männistö. Improving the delivery cycle: A multiple-case study of the toolchains in Finnish software intensive enterprises,. Information and Software Technology, 80 (2016), 175–194.