Universidade Federal de Pernambuco
Centro de Informática

Graduação em Sistemas de Informação

Um estudo em larga-escala de repositórios Open Source no Github que utilizam containers

Rafael Felipe Pedroza Jordão

Trabalho de Graduação

Recife
15 de Novembro de 2018

Universidade Federal de Pernambuco
Centro de Informática

Rafael Felipe Pedroza Jordão

Um estudo em larga-escala de repositórios Open Source no Github que utilizam containers

Trabalho apresentado ao Programa de Graduação em Sistemas de Informação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção de grau de Bacharel em Sistemas de Informação.

Orientador: *Vinícius Cardoso Garcia*

Recife
15 de Novembro de 2018

Universidade Federal de Pernambuco
Centro de Informática

Rafael Felipe Pedroza Jordão

Um estudo em larga-escala de repositórios Open Source no Github que utilizam containers

Monografia submetida ao corpo docente da Universidade Federal de Pernambuco, defendida e aprovada em ___ de _____ de _____.

Banca Examinadora:

Orientador: _____
Vinicius Cardoso Garcia

Examinador: _____
Kiev Santos da Gama

Recife
15 de Novembro de 2018

Agradecimentos

Agradeço a Deus pela força de vontade, proteção e bênçãos concedidas.

Agradeço aos meus pais, Paulo Fernando Rodrigues Jordão e Etiene Maria Pedroza Jordão, e minha avó, Paulina Rodrigues Jordão, que sempre me apoiaram, acreditaram, batalharam por mim e fizeram o possível e o impossível para que eu conseguisse chegar onde cheguei até o momento.

Agradeço a minha companheira e futura esposa, Taciani Aparecida Vieira da Silva, que me acompanhou e apoiou durante todo esse penoso percurso, esquecendo os próprios problemas para poder ouvir os meus e me reconfortar em momentos de necessidade.

Aos meus professores por me auxiliarem neste caminho edificante que é o saber, e principalmente a Vinicius Cardoso Garcia e Bernadette Farias Lóscio, que me agraciaram com a possibilidade de poder transmitir o que aprendi a outros alunos, além de fornecerem as ferramentas que precisei para me tornar o profissional que sou hoje.

Não menos importante, aos meus companheiros de caminhada e grandes amigos, Augusto Lima, Carlos Melo e Pedro Neto, que compartilharam comigo as batalhas vivenciadas durante nossa graduação, transformando uma trajetória desgastante em uma jornada no mínimo divertida. Viva os *WildCats!*

Agradeço a todos, de coração!

Resumo

Uma das plataformas que mais fomenta e oferece suporte a comunidade de TI é o *GitHub*, por disponibilizar o armazenamento de projetos, facilitar o trabalho em conjunto e a gerência de projetos bem como o compartilhamento dos mesmos. Por disponibilizar esse valor, de forma gratuita, grande parte dos projetos *Open Source* utilizam o *GitHub* como seu repositório. Uma das formas de se analisar as tendências do mercado de TI é observar o que projetos *Open Source* estão utilizando atualmente, inquirindo sobre novas tecnologias e formas de utilizá-las. Dada a importância dos containers, que já é reconhecida, não só por possibilitar uma entrega mais rápida de valor, mas também como por dinamizar, padronizar e possibilitar a automatização de processos antes enrijecidos e demorados, como o processo de implantação, vê-se necessário, por ser uma tecnologia relativamente nova, estudos e análises sobre o tema. Com isto tudo em mente, este estudo tem como intuito construir e disponibilizar uma grande base de dados sobre repositórios *Open Soure* no *Github* que utilizem *containers* (119 mil repositórios, 419 mil *Dockerfiles*, 35 mil *docker-compose.yml*, 178 linguagens) para prover uma análise sobre a utilização das *Best Practices* de construção de *containers*, bem como para auxiliar em estudos subsequentes com o foco nos mesmos.

Palavras-chave: containers, open source, dataset, GitHub

Abstract

One of the platforms that most fosters and supports the IT community is GitHub, by making project storage available, facilitating joint work and project management as well as sharing of projects. By making this value available for free, most Open Source projects use GitHub as their repository. One way to look at trends in the IT market is to look what the Open Source projects are currently using, inquiring about new technologies and ways to use them. Given the importance of containers, which is already recognized, not only for enabling faster delivery of value, but also for dynamizing, standardizing and enabling the automation of processes that are stiff and time-consuming, such as the implementation process, it is necessary, because it is a relatively new technology, studies and analyzes on the subject. With this in mind, this study aims to build and make available a large database of open source repositories in GitHub that use containers (119,000 repositories, 419,000 Dockerfiles, 35,000 docker-compose.yml, 178 languages) to provide an analysis of the use of the Best Practices of container construction, as well as to assist in subsequent studies with the focus on them.

Palavras-chave: containers, open source, dataset, GitHub

Índice de Figuras

Índice de Tabelas

Sumário

1 Introdução

1.1 - Motivação

As metodologias atuais de desenvolvimento de software como conhecemos não seriam possíveis sem o auxílio de uma plataforma poderosa como o *GitHub*[1] que serve como um armazenador de repositórios públicos e gratuito. Entre os benefícios do *Github*, deve-se marcar a facilidade de compartilhamento e cooperação no desenvolvimento de projetos, bem como o grande auxílio à gerência de projetos, indo desde um simples quadro *Kanban*[2] a informações estatísticas do repositório. Por manter a gratuidade de repositórios públicos, o *Github* concentra grande parte dos projetos *Open Source*[3], o que traz uma grande quantidade de informações consistentes ao uso de tecnologia de informação, mostrando um grande potencial para análises e estudos quanto as tendências, servindo como uma espécie de termômetro para o uso de tecnologias atuais e quais estão entrando no *hype*.

Cloud Computing[4], basicamente, representa a tecnologia e modelo computacional que permeia o nosso dia a dia, desde o início do século XXI [1], tecnologia esta que só foi possível graças aos avanços da virtualização[5], já que a mesma desenvolveu a capacidade de escalonamento e administração de recursos como processamento, memória, rede e armazenamento [2]. Inicialmente as *Virtual Machines*[6] (VM's) que compuseram o cerne da *Cloud,* porém, com o passar dos anos, algumas necessidades trouxeram à tona problemas, como a necessidade de imagens completas de *Guest OS*, consumindo armazenamento, processamento e memória que poderiam estar sendo usados para aplicação, além do overhead de inicialização (levando normalmente minutos para inicializar), caso ocorra falhas ou atualizações, em que a *VM* precise ser desligada, acentuando o fato de serem *Stateful*[7] [2].

---

[1] http://github.com
[2] https://www.significados.com.br/kanban/
[3] https://opensource.org/about
[4] É um estilo de computação em que a escalabilidade e elasticidade de TI são entregues como um serviço usando a internet. Segundo gartner (https://www.gartner.com)
[5] É uma tecnologia que permite criar serviços de TI valiosos usando recursos que tradicionalmente estão vinculados a um determinado hardware.
[6] É uma implementação de software de uma arquitetura voltada a hardware que executa instruções pré-definidas similarmente a uma CPU (do inglês Central Processing Unit).
[7] Que mantém ou depende do estado anterior de uma aplicação, processo, etc. contrasta com Stateless. Retirado do dicionário Oxford.

Em meio a isto surgiram os *containers*, mais em específico o *Docker*[8], que representa o passo seguinte na tecnologia de virtualização com foco maior na entrega de software de forma portável, conseguindo suprir as necessidades que as *VM's* não conseguiam suprir, enquanto facilita novas metodologias, como o *DevOps*. [1][2] Os *containers* Docker são criados a partir de imagens, e estas são criadas a partir de um arquivo nomeado *Dockerfile* que possui instruções de configuração de infraestrutura (basicamente *Infrastructure as Code*[9]) [3].

O fato dessa tecnologia ser relativamente nova (lançada há 5 anos, em 13 de março de 2013) resulta em uma baixa quantidade de análises com este tema. A dificuldade de obtenção de grandes quantidades informações diretamente da principal plataforma de armazenamento de repositórios atualmente, o *Github*, que limita as pesquisas em sua API para apenas 1000 resultados, evidencia uma necessidade de disponibilização de data-sets focado na premissa de *containers* para o fomento de novas pesquisas e análises com o tema.

1.2 - Objetivos

1.2.1 - Gerais

O objetivo deste trabalho é extração e disponibilização de meta-dados (como quantidade de branches, quantidade de commits, data de criação, branch padrão, qual o repositório pai, quantidade de forks, data de criação, linguagens, licença, nome, arquivo readme, quantidade de releases, quantidade de stars, data do último update, url e quantidade de watchers) provenientes de repositórios *Open Source* do *Github* que possuam arquivos pertinentes a tecnologia *Docker*. Além disto, este trabalho tem como objetivo uma análise exploratória da utilização das *Best Practices* de construção de *Dockerfiles* descritas pela documentação do Docker [4] e sobre o crescimento do uso de containers na plataforma.

[8] https://www.docker.com/
[9] http://infrastructure-as-code.com/

1.2.2 - Específicos

- Localização e mineração de repositórios *Open Source* no *Github* que utilizem tecnologia *Docker*.
- Localização e mineração do conteúdo dos arquivos da tecnologia *Docker*.
- Análise exploratória sobre a utilização de *Best Practices* de construção de *Dockerfile*, procurando responder às seguintes perguntas:
    - A maioria dos repositórios seguem as *Best Practices* Selecionadas de construção de *Dockerfile*?
    - Qual linguagem tem maior porcentagem de *Dockerfile* enquadrados nas *Best Practices* Selecionadas?
    - Quais as principais razões para os *Dockerfiles* não seguirem as *Best Practices* Selecionadas*?*
- Análise sobre o crescimento de repositórios que utilizam containers na plataforma *GitHub*, procurando responder às seguintes perguntas:
    - Qual a linguagem mais predominante na utilização de *Docker*?
    - Qual ano obteve maior crescimento na criação de repositórios que utilizam a tecnologia *Docker* com as *Best Practices* Selecionadas?

1.2.3 - Estrutura do Trabalho

Este documento é composto por 5 capítulos:

- Capítulo 1 - Introdução: Demonstra a motivação e objetivos do trabalho;
- Capítulo 2 - Contexto: Apresenta todo o contexto teórico e tecnológico para o entendimento dos capítulos subsequentes;
- Capítulo 3 - Mineração: Apresenta toda a etapa de mineração, com dificuldades enfrentadas
- Capítulo 4 - Análise: Foca na resolução das perguntas de pesquisa levantadas nos objetivos
- Capítulo 5- Conclusão: Apresenta e sumariza as conclusões obtidas no decorrer do trabalho e apresenta trabalhos futuros.

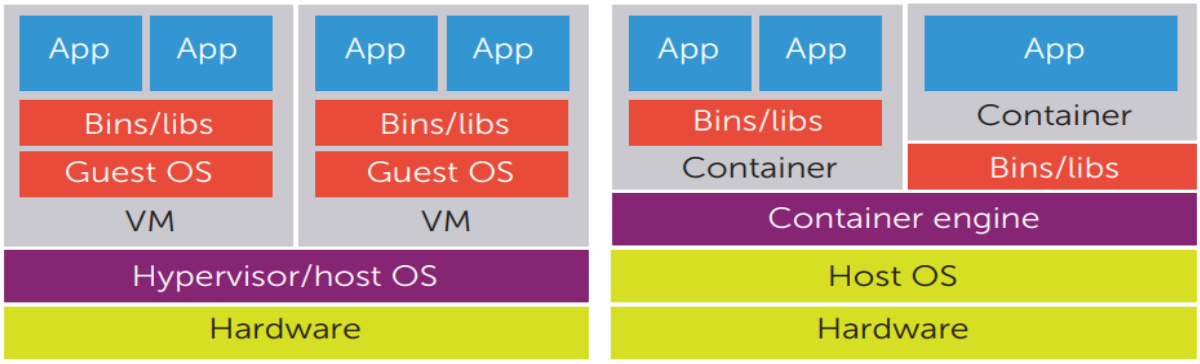
2 *Contexto*

2.1 *Containers*

O termo é derivado do transporte de produtos sobre oceanos, que é feito utilizando grandes containers para isolamento e melhor administração da carga. [5]

Segundo PAHL [1], Containers, basicamente, são técnicas de virtualização baseadas em *namespaces*[10] e *cgroups*[11] e são particularmente adequados para administração de aplicativos no contexto *PaaS*[12]. Os mesmos são representados por imagens leves, em contraste com as *Virtual Machines* que utilizam em imagens completas de Sistemas Operacionais. Além disso, os processos dentro do container são quase totalmente isolados. Comparado com as VM's, a nível de arquitetura, os containers conseguem o isolamento e melhor desempenho quanto a utilização de recursos pois compartilha do mesmo Kernel da máquina Hospedeira, não necessitando assim de um S.O. a cada novo container, diferente das VM's que necessitam de um S.O. completo, com seu próprio kernel, como pode ser verificado na Figura 1.

Figura 1- Arquitetura de virtualização, a esquerda o modelo tradicional de hypervisor a direita o modelo baseado em containers



Fonte: [2] Figura 1 Virtualization architecture. - Página 2

[10] Uma classe de elementos (por exemplo, endereços, locais de arquivos, etc.) nos quais cada elemento possui um nome exclusivo para essa classe, embora possa ser compartilhado com elementos em outras classes. Retirado do dicionário Oxford.

[11] Grupos de controle que permitem alocar recursos — tais como tempo da CPU, memória do sistema, largura de banda de rede ou combinações destes recursos — entre grupos de usuários definidos de tarefas (processos) rodando em um sistema.

[12] É uma ampla coleção de serviços de infra-estrutura de aplicativos (incluindo plataforma de aplicativos, integração, gerenciamento de processos de negócios e serviços de banco de dados). Segundo gartner (https://www.gartner.com)

Capítulo 2

DUA e RAJA [6] trazem uma tabela comparativa entre a tecnologia tradicional de virtualização, *Virtual Machine*, e a virtualização baseada em containers, em especialmente nos parâmetros de Guest OS, Comunicação, Segurança, Performance, Isolamento, Tempo de inicialização e armazenamento, representada aqui pela Tabela 1.

Tabela 1 - Comparação entre virtualização baseada em VM's e virtualização baseada em containers

Parâmetro	Virtual Machines	Containers
Guest OS	Cada VM roda em hardware virtual e kernel carregado na própria memória.	Todos os containers dividem os mesmo OS e kernel. imagem do kernel é carregada na memória física.
Comunicação	Por entre dispositivos Ethernet.	Mecanismos de comunicação entre processos, como *Signals*, *Pipes*, *Sockets*, etc.
Segurança	Depende da implementação do Hypervisor.	Mandatory Access Control (MAC) pode ser reaproveitado
Performance	VM's sofrem de uma pequena sobrecarga, pois as instruções da máquina precisam ser traduzidas do Guest Os para o Host Os.	Containers possuem uma performance quase nativa quando comparados com o Host Os.
Isolamento	Não é possível compartilhar lib's, arquivos e etc entre Guest OS	Subdiretórios podem ser montados de forma transparente e podem ser compartilhados entre

		containers
Tempo de inicialização	VM's levam alguns poucos minutos para iniciar	Containers podem ser iniciados em poucos segundos
Armazenamento	VM's ocupam mais espaço, já que necessitam de próprio OS e Kernel	Containers ocupam menos espaço por compartilhar o OS e o kernel com o Host

Fonte: [5] Tabela 1

Diferentes modelos de *Container*

Segundo PAHL [2] existem algumas tecnologias de container para diferentes tipos de sistemas operacionais, bem como soluções específicas para plataformas PaaS, tais como:

- Linux (Docker, LXC, LXD, Linux VServer, OpenVZ, RKT [6], e outras variantes como BSD, HP-UX e Solaris)
- Windows (Windows Server Containers [6] e Sandboxie)
- *Cloud PaaS* (Warden/Garden e LXC)

2.2 *Docker*

O *Docker*, criado em 18 de março de 2013, por se apresentar de forma simples, gratuita e por possuir uma ampla documentação de fácil acesso e entendimento, se tornou a ferramenta para containers mais popular atualmente. Ele utiliza imagens, que podem ou ser construídas baseadas em um arquivo de configuração (chamado *Dockerfile*) ou podem ser baixadas por um repositório oficial (*Docker Hub*) ou de terceiros [1][7]. Estas imagens são compostas de um conjunto de camadas de apenas leitura, sendo cada camada definida por uma instrução do *Dockerfile*, que criadas no momento de *Build*, juntamente com os meta-dados que são armazenados utilizando *JSON* (JavaScript Object Notation). Cada camada desta possui as modificações feitas na camada prévia, que começam tipicamente de uma leve distribuição Linux, que por sua vez organiza as imagens em árvores, possibilitando o Docker a subir em uma imagem apenas aquelas modificações relacionadas a ela. [8]

2.2.1 – *Dockerfile*

O *Dockerfile* é o arquivo de configuração e construção de imagens *Docker*. Nele é possível definir, por exemplo, quais variáveis de ambiente deverão existir, qual o diretório será a base para a aplicação, quais bibliotecas serão necessárias, quais portas necessitam estar abertas para a aplicação e etc., então, basicamente, o *Dockerfile* se comporta como *Infrastructure as Code*. É no *Dockerfile* que, por meio do uso de instruções (como FROM, COPY, RUN, EXPOSE), são definidas as camadas de *File System* da imagem, então uma má elaboração do *Dockerfile* pode criar uma imagem que ocupará mais espaço sem necessidade. No tópico *Best Practices* será coberta a forma de se escrever um *Dockerfile* com vista de minimizar camadas e, por conseguinte, garantir uma imagem leve.

2.2.2 – *Docker Compose*

Segundo a documentação do Docker, o *Docker Compose* serve para a construção de ambientes multi-containers, facilitando e aumentando a efemeridade das aplicações. O *Compose* usa um arquivo de extensão .yml, onde o nome padrão usado pela comunidade é *docker-compose.yml*, para a configuração dos serviços no ambiente, onde são cadastradas configurações externas ao container, como local no host que o volume será mapeado, portas do host que serão dadas o *bind* com as portas do container, entre outras. Algumas funcionalidades do *Docker Compose* que auxiliam o desenvolvimento e a implantação são as seguintes:

- Múltiplos ambientes isolados em um mesmo hosts: Com o *Compose* é possível nomear projetos para isolá-los, podendo usá-los para por exemplo, em um host compartilhados, garantir cada aplicação rodando no ambiente estejam isoladas umas das outras.
- Preservar os volumes do serviço: Ao iniciar, ele verifica todos os volumes preservados de serviços prévios, e se achar algum, os reutiliza para garantir que os dados criados não sejam perdidos.
- Apenas recria *containers* que tiveram alterações: Ele reusa containers existentes o máximo possível, para que mudanças feitas no ambiente sejam efetuadas muito rapidamente.

- Uso de variáveis: O *Compose* dá suporte ao uso de variáveis dentro de seu arquivo de configuração, podendo customizar a configuração para diferentes ambientes ou usuários.
- Habilidade para mover configurações em diferentes ambientes: O *Compose* também permite que você crie um arquivo de *Override*, podendo modificar/adicionar algumas configurações adicionais a sua configuração base existente no arquivo .yml.

2.2.3 – *Dockerhub*

O *Dockerhub* é um repositório online que permite o armazenamento e compartilhamento de imagens construídas. Assim como o *Github*, o mesmo é oferecido gratuitamente para os desenvolvedores, porém sem a opção de formar repositórios privados, pois este apenas está disponível para contas que pagam mensalidade. Neste repositório é possível encontrar imagens geradas pela própria empresa. Os repositórios de desenvolvedores seguem o padrão "nome do desenvolvedor/repositório" [8]

2.2.4 - *Best Practices* na construção de *Dockerfiles*

Em sua documentação [4], o Docker dedica um capítulo totalmente voltado para as *Best Practices* na construção de *Dockerfiles*, auxiliando desenvolvedores a criar containers os mais efêmeros possível, ou seja, um container pode ser destruído e refeito baseado na imagem construída e ter o mínimo de configuração adicional possível para retornar ao mesmo estado anterior. Resumidamente, estas são as boas práticas, separadas por Instrução:

- `FROM`:
  - Utilizar *Multi-stage Builds* - Usar mais de um `FROM` possibilita em diminuir drasticamente o tamanho final de uma imagem, sem a preocupação de diminuir o número de camadas intermediárias. A ideia é você construir sua aplicação utilizando uma imagem mais completa, e depois utilizar outro `FROM`, copiando os arquivos necessários para uma base mais limpa.

- Utilizar imagens oficiais como base - A documentação recomenda o uso de imagens *Alpine*[13] sempre que possível, por serem leves e ainda assim serem uma distribuição completa do Linux.

- `RUN`:
  - Evitar `RUN apt-get upgrade` e `dist-upgrade` - As camadas inferiores (herdadas pelo `FROM`) possuem pacotes que não podem ser atualizados em uma camada *Read Only* sem privilégios. Se você sabe de um pacote em particular que está desatualizado, utilize o `apt-get install -y` para atualizar automaticamente.
  - Sempre combine o `RUN apt-get update` com `apt-get install` num mesmo `RUN` - ao utilizar o `RUN apt-get update` em uma única instrução, o mesmo fica gravado no cache após o *Build*, e por isso, qualquer `apt-get install` utilizado/modificado após a geração da primeira imagem irá trazer pacotes defasados.
  - Combine `set -o pipefail &&` a comandos que utilizam *pipe* (|) - O interpretador de comandos utilizado pelo *Docker* faz a avaliação de sucesso do comando apenas pelo último comando do *pipe* o que pode acarretar em instruções retornando sucesso inadvertidamente.
  - Não utilize `RUN cd ... &&` mais instruções - Mudar de diretórios pelo `RUN` apenas torna o código mais ilegível, além de mais difíceis de manter, por isso deve-se usar a instrução `WORKDIR` para alterar locais.
  - Não utilize `RUN sudo` - o comando `sudo` possui retorno TTY[14] imprevisível, além de possuir um comportamento de *Signal-Fowarding* que pode causar problemas. Caso necessite realmente o uso do `sudo`, considere usar o *gosu*[15].

- `ADD` e `COPY`:
  - Sempre que possível, use `COPY` ao invés de `ADD` - `COPY` e `ADD` são bem similares, com a pequena diferença de que `ADD` pode acessar arquivos remotamente e possui um auto extrator de extensões tar, porém, por questões de transparência, o `COPY` é indicado para uso em casos mais comuns, como simples cópia de

[13] https://hub.docker.com/_/alpine/
[14] Abreviação de Teletypewriter
[15] https://github.com/tianon/gosu

arquivos enquanto o `ADD` apenas para caso em que existe necessidade de auto extração de pacotes.
    - Nunca utilize `ADD` para pegar pacotes remotos - Por questões de tamanho de imagem, deve-se usar curl ou wget com pacotes remotos, pois você pode, no mesmo comando `RUN`, deletar os pacotes desnecessários após a extração, sem a necessidade de adicionar mais uma camada.
- `ENTRYPOINT`:
    - Utilize o comando principal no `ENTRYPOINT` e os parâmetros padrões no `CMD` - Isso deixa o container com mais possibilidades de uso, além de, como exemplo, ser possível adicionar como parâmetro padrão um --help, auxiliando o usuário, mas tampouco tirando a habilidade de passar parâmetros customizados para o container.
- `WORKDIR`:
    - Utilize sempre caminhos absolutos - Isso melhora a clareza do código e facilita a manutenção do mesmo.

2.3 *Github* e *Open Source*

Basicamente, o *Github* é um serviço de hospedagem de controle de versões que usa Git, que vai além de ser um mero serviço simples. O *Github* possuí várias funcionalidades que auxiliam e facilitam o desenvolvimento de projetos, sejam eles pequenos ou grandes, com poucos ou vários colaboradores. Alguns dessas funcionalidades são:

- Revisão de código: O *Github* auxilia na revisão de código, dando ferramentas que auxiliam esse processo de desenvolvimento, facilitando a colaboração entre usuários.
- Rastreabilidade de *Issues* e Requisição de funcionalidades: A plataforma traz de uma forma simples e prática a capacidade de rastrear e gerenciar pedidos de funcionalidades e problemas ocorridos durante o desenvolvimento do projeto.
- Integração com outras plataformas: O *Github* se comunica e integra outras plataformas utilizadas durante o desenvolvimento, como o *Slack* ou o *TravisCI* para a comunicação, viabilizando uma melhor experiência e praticidade.

- Documentação ao lado do código: o *Github* fornece a possibilidade de hospedagem direta de documentação utilizando o *Github Pages*, com o gerador de Sites Jekyll, além de possuir uma seção dedicada para Wiki em cada repositório.
- Gráficos com status do repositório: Apresenta de forma simples informações como frequência de código, contribuições, trafego, listas de dependência, entre outros.
- Alertas de segurança de vulnerabilidades: A plataforma verifica e alerta sobre possíveis vulnerabilidades no código, principalmente quanto a dependência e atualizações necessárias.
- Leitores de diversos formatos direto pelo navegador: A plataforma possui leitores de algumas extensões, tais como PDF, PSD, renderizações 3D.

O *Github* trabalha com acesso gratuito para repositórios públicos, motivo pelo qual conseguiu atrair a maioria dos projetos *Open Source* para a plataforma, e possui um plano de assinatura para quem deseja hospedar repositórios privados.

O movimento *Open Source*, nascido em 1998, foi caracterizado por alguns como a nova forma de se desenvolver softwares [9] e até como a forma "superior" de se produzir código [10], que demonstrava um desafio ao mercado comercial de software, desafio esse que não operava com as mesmas regras que seus competidores, mas ameaçava de fazê-lo mais rápido, melhor e mais barato.

2.4 *GHTorrent*

É um serviço que captura *Event Streams* e dados do GitHub e disponibiliza gratuitamente para a comunidade na forma de *dumps* do *MongoDB* desde 2012 [11]. Atualmente o mesmo disponibiliza também um *dump* no formato relacional para *MySql*, além do acesso direto ao banco de dados deles, apenas por indicação de índices previamente conhecidos, com a concessão da sua *API Key* do *GitHub*.

2.5 *Kaggle*

O Kaggle é uma plataforma feita para a comunidade de *Data-Science* e *Machine Learning* com o propósito de compartilhar conjuntos de dados, fomentar a formação de novos *Data-Scientists* e engenheiros de *Machine Learning*, e promover competições entre profissionais da área.

Kernels

O *Kaggle* possuí em sua plataforma a funcionalidade de criação de *Kernels*, que possibilitam a execução de códigos bem como acesso direto aos data-set dos desafios cadastrados. Os *Kernels* utilizam *R* ou *Python* como linguagem e possuem como principal objetivo a disponibilização e disseminação de analises e conhecimento dentro da comunidade. Existem basicamente três tipos de *Kernels*:

- *Scripts* puros: São *scripts* fonte escritos em *R* ou *Python* puro, apenas para a execução de código sequencial.
- *Scripts RMarkdown*: É um tipo especial de *Script*, geralmente utilizado pelos usuários de *R* na comunidade, que combina a sintaxe *R* com a *Markdown*.
- *Notebooks*: Baseados em *Jupyter Notebooks*, consistem em uma sequência de células que podem ser *Markdown* ou uma linguagem a escolha (entre *R* e *Python*).

Para ter acesso a criar *Kernels*, é necessário o cadastro na plataforma. Na seção de anexos estará disponível um link para um tutorial de como se criar *Kernels* dentro da plataforma.

2.6 Sumário

Capítulo 2

Este capítulo trouxe as definições e conceitos básicos necessários para entender o objetivo proposto pelo projeto. No capítulo seguinte, será abordada a etapa de mineração e construção do conjunto de dados necessário para as análises.

3 *Mineração*

Para a mineração, optou-se por utilizar a linguagem Python, pois a mesma traz algumas facilidades, tanto no momento da extração quanto para a análise. Todos os scripts utilizados neste estudo, bem como todo o material coletado, estarão disponibilizados em um repositório no GitHub no qual o link se encontra na seção de anexos.

3.1 Localizando os repositórios

O principal objetivo deste trabalho é a obtenção de um grande conjunto de dados, que possa servir para análises posteriores, contendo informações sobre repositórios do *GitHub* que utilizam a tecnologia de containers, mais em específico o *Docker*, além das informações dos arquivos de configuração *Dockerfile* e *docker-compose.yml*. Porém, existem limitações no uso da *API* disponibilizada pelo *GitHub*, tais como a limitação no retorno máximo de busca em 1000 resultados e a limitação de apenas 5000 requests por hora por *API Key*, e por conta disso se viu necessário a obtenção dos repositórios alvo do estudo por outros meios.

Apesar do *GHTorrent* ser mais completo, já que o seu banco de dados do MongoDB possui mais de 10 Terabytes de dados disponibilizados (possuindo mais de 74 Milhões de repositórios capturados, com seus devidos commits, issues, comentários, eventos, etc) não existe uma ligação direta entre os arquivos e o repositório, necessitando assim uma pesquisa custosa em cada commit de cada repositório (que somados, superam a quantidade de 847 Milhões de commits) em busca dos arquivos requeridos e, por conta disto, a escolha foi feita em favor de utilizar o *Kaggle*, por disponibilizar de forma fácil os dados bem como um meio de filtro dos dados, podendo assim, ao invés de adquirir todo o conjunto de dados, adquirir apenas aqueles que são relevantes ao projeto. Com isso, foi criado um *Kernel* no *Kaggle*, que fornece acesso direto ao conjunto de dados no servidor, para obter a base para seguir com a mineração fazendo as seguintes consultas em SQL:

```
SELECT repo_name,path
FROM `bigquery-public-data.github_repos.files`
WHERE path LIKE '%ockerfile';
```

```
SELECT repo_name,path
FROM `bigquery-public-data.github_repos.files`
WHERE path LIKE '%docker-compose.yml
```

As duas queries procuram por caminhos de arquivo que possuem em seu final o nome dos arquivos definidos pela documentação do Docker, que são: Dockerfile e docker-compose.yml. Pelo fato de existirem arquivos nos repositórios nomeados por “dockerfile” com “d”, se viu necessário o uso de uma *wildcard* (%) na posição do “D”, ficando assim %ockerfile, além disso, como queremos obter o arquivo em qualquer lugar da árvore de arquivos dos repositórios, se faz o uso da wildcard também no “docker-compose.yml”. O resultado gerou dois arquivos .csv, cada um deles possuindo o mínimo de informação necessária para a extração do GitHub, apenas nome do repositório seguido de caminho do arquivo, sendo eles, docker-compose.csv que possui 2,54 Megabytes de tamanho totalizando 42957 linhas e o *dockerfile*.csv com 28,6 Megabytes de tamanho totalizando 445490 linhas, cada linha representando um arquivo (*Dockerfile* ou docker-compose.yml). Posteriormente, todos os dois arquivos .csv foram importados em um banco MySql para melhor organização dos dados.

3.2 Estruturando o conjunto de dados

Pelo motivo de os dados serem variados, a decisão foi de utilizar o MongoDB para saída de dados, por ser mais maleável que o modelo relacional e dar a possibilidade de exportação em JSON e outros formatos mais comuns e de fácil utilização. Os dados foram divididos em 3 Coleções:

- repos - Coleção com informações diretas sobre os repos com a seguinte estrutura:
    - _id - Id do documento
    - branches - Quantidade de número de *branches* do repositório
    - commits - Quantidade total de commits
    - created_at - Data de criação do repositório
    - default_branch - *branch master* do repositório
    - forked_from - Id do repositório de pai

    - forks - Quantidade de forks do repositório
    - languages - Linguagens do repositório, em formato de objeto com a linguagem e sua porcentagem
    - license - Licença de uso do repositório
    - name - Nome do repositório
    - readme - Arquivo readme do repositório em formato string
    - releases - Quantidade de releases do repositório
    - stars - Quantidade de stars do repositório
    - updated_at - Data do último update do repositório
    - url - Url do repositório
    - watchers - Quantidade de watchers do repositório
- dockerfile_repos - Coleção com informações do *Dockerfile*, bem como seu conteúdo:
    - _id - Id do *Dockerfile*
    - config - Conteúdo do *Dockerfile*
        - Config<N> - Cada FROM significa um Config, onde N é um número inteiro sequencial iniciado em 1
    - path - Caminho até o *Dockerfile*
    - repo - Id do repositório
    - repoName - Nome do repositório
- dockercompose_repos - Coleção com informações do *docker-compose.yml*, bem como seu conteúdo:
    - _id - Id do *docker-compose.yml*
    - path - Caminho até o *docker-compose.yml*
    - repo - Id do repositório
    - repoName - Nome do repositório
    - yml - Conteúdo do *docker-compose.yml*

3.3 Capturando os dados

Para a mineração foi utilizado um *script multi-thread* em *Python*, que se tornou a linguagem mais usada para mineração e data science, utilizando requests simples para

informações diretas do repositório e utilizando *GitHub.py* para informações que utilizem o *Ajax* e por isso não podem ser obtidos por request.

"Driblando" parte das limitações da *API* do *GitHub*, foram utilizadas quatro chaves de acesso em modo *round robin*, assim subindo o limite de 5000 requests por hora para 20000, além disso, foram usados 24 threads em paralelo para tornar a extração de repositórios mais rápida e para a extração de *Dockerfile* e *docker-compose.yml* 400 threads, por não possuir a limitação da API.

Pelo fato de os dados serem de janeiro de 2018, alguns repositórios haviam deletado os arquivos ou não existiam mais, por este motivo não foi possível a captura de todo conjunto obtido pelo Kaggle, na tabela 2 é possível ver a quantidade obtida com a porcentagem total de sucesso na extração.

Tabela 2 - Total de arquivos extraídos

	Quantidade do Kaggle	Quantidade total extraída	Porcentagem de sucesso
Dockerfile	445490	429329	~ 96,37%
docker-compose.yml	42957	35948	~ 83,68%

Fonte: Próprio Autor

No total, com estes arquivos extraídos, foi possível obter 119144 repositórios, com 178 linguagens.

3.4 Sumário

Este capítulo abordou as definições e motivos para a configuração utilizada durante a fase de mineração, bem como trouxe a estruturação e dicionário dos dados obtidos.

O próximo capítulo irá abordar a análise exploratória feita bem como trará as respostas encontradas paras as perguntas definidas no item 1.2.2 do capítulo 1.

4 Análise

4.1 Análise quantitativa

Antes de partir para a Análise e respostas das perguntas de pesquisa, precisamos verificar como os dados estão distribuídos para que possam ser verificados possíveis outliers ou erros no conjunto de dados. Verificamos os dados quanto a quantidade de *Dockerfiles* por repositório.

Foi verificado que a amplitude total, de 12298, que significa a diferença entre o mínimo valor e o máximo valor registrado. Sendo esse máximo representando dois repositórios, o *resin-io-library/base-images* e o *nghiant2710/base-images*, sendo este último um fork do primeiro, que por serem repositórios legítimos com quantidades verificadas, foram mantidos na análise. Além da amplitude, outra informação que vale ressaltar é a moda, sendo esta igual a 1, que indica que a maioria dos repositórios só possui um *Dockerfile* e verificando os quartis, é possível notar que há um crescimento lento quando ordenado crescentemente, onde o 1º e 2º quartis são iguais a 1 e o 3º quartil é igual a 2, indicando que os repositórios com maior concentração de *Dockerfiles* estavam presentes no último quartil. Devida a grande diferença entre o máximo e o valor apresentado no 3º quartil isso houve a necessidade de uma verificação mais minuciosa quanto aos percentis, sendo verificados os 90º e 95º percentis, apresentando valores de 6 e 13, confirmando que 5% ou menos dos repositórios estão concentrando a maioria dos *Dockerfiles*. A sumarização dessas informações pode ser encontrada na Tabela 2.

Tabela 3 - Sumarização das informações da análise quantitativa sobre Dockerfiles em repositórios

Moda	1	Amplitude Total	12298	2º Quartil	1
Mediana	1	Mínimo	1	3º Quartil	2
Média	4,45581	Máximo	12299	90º percentil	6
Desvio Padrão	50,6149	1º Quartil	1	95º Percentil	13

Fonte: Próprio Autor

4.2 Análise sobre *Best Practices*

Este índice tem como objetivo responder às perguntas presentes no sub-índice 1.2.2 que dizem respeito a *Best Practices*.

4.2.1 - A maioria dos repositórios seguem as *Best Practices* Selecionadas de construção de *Dockerfile*?

Para responder essa pergunta, foi necessário primeiro selecionar quais Best Practices poderiam ser verificadas e as seguintes foram selecionadas por serem facilmente verificadas:

- `RUN apt-get update` sem `install`
- `RUN` com pipe e sem *fail cautions*
- `RUN` com `cd`
- `RUN` com `sudo`
- `RUN` com `apt-get upgrade` ou `dist-upgrade`
- `ADD` com endereço remoto
- `ADD`, no lugar de `COPY`, sem utilizar o arquivo tar
- `WORKDIR` sem *path* absoluto
- *Multi-stage build*, ou seja, arquivos com vários `FROM`, com o último estágio possuindo mais camadas que os estágios antecessores

Com isso, cada linha dos *Dockerfiles* foi verificada em busca de informações que se encaixam nas regras estipuladas acima, por comparação de strings ou por contagem de camadas. No total, dos 419321 *Dockerfiles* extraídos, 255817 não obedeceram às regras listadas a cima e 163504 seguem as regras listadas acima. Desta forma, apenas 38,99% dos *Dockerfiles* extraídos seguem as Best Practices. Os dados estão sumarizados na tabela 3.

Tabela 4 - Sumarização das informações do índice 4.2.1

	Números Absolutos	Porcentagem
Dockerfiles que seguem as *Best Practices* listadas	163504	38,9925618%
Dockerfiles que fogem das *Best Practices*	255817	61,007438216%

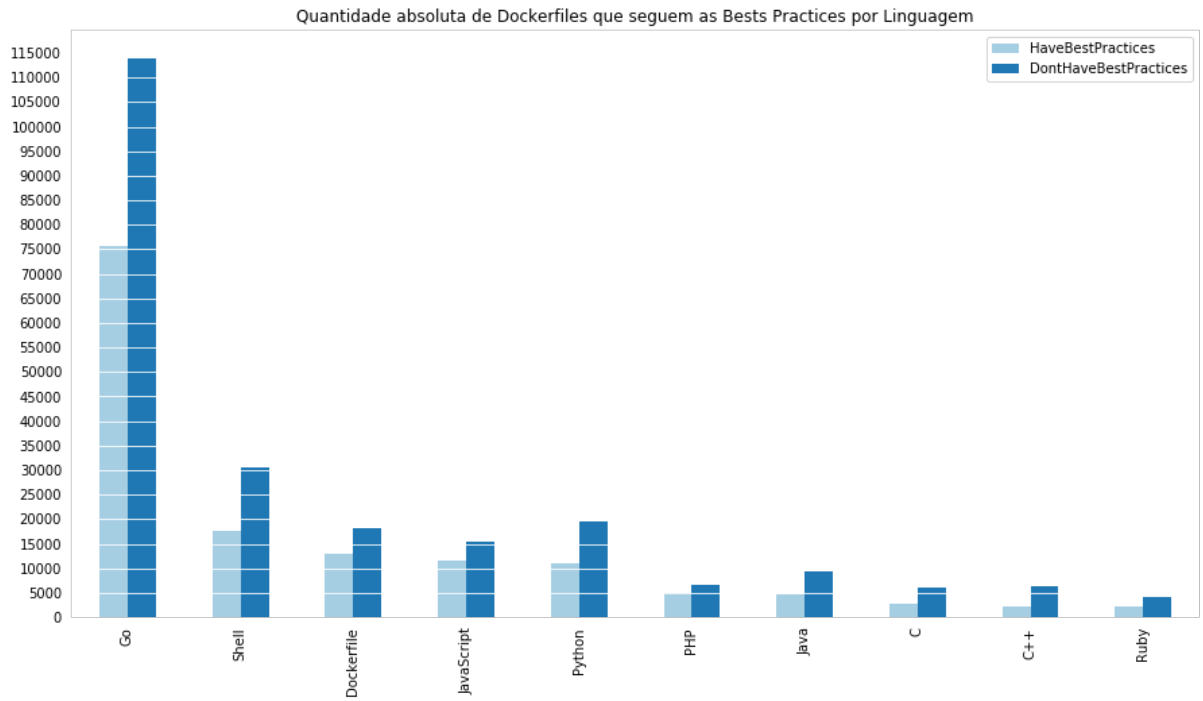
Fonte: Próprio Autor

Como resposta para pergunta, a maioria dos *Dockerfiles* extraídos não seguem as *Best Practices* selecionadas pelo trabalho, ou seja, a grande maioria dos repositórios não possuem imagens tão efêmeras quanto possível de se alcançar, caso possuíssem 100% de uso das mesmas.

4.2.2 - Qual linguagem tem maior porcentagem de *Dockerfile* enquadrados nas *Best Practices* selecionadas?

Para responder esta pergunta, foi verificada a linguagem predominante em cada repositório justamente com as informações obtidas para a última resposta. Porém, devida a quantidade de linguagens obtidas, que foi de 148, se torna inviável a criação de um gráfico que abranja todas. Sendo assim, foi necessário agrupar os dados por linguagem, limitar a informação pelas 10 linguagens que possuíam o maior número de *Dockerfiles* que seguem as *Best Practices* e plotar um gráfico de barras, para melhor visualização. O mesmo pode ser verificado na Figura 2.

Figura 2 - Gráfico da Quantidade Absoluta de Dockerfiles que seguem as Best Practices por Linguagem

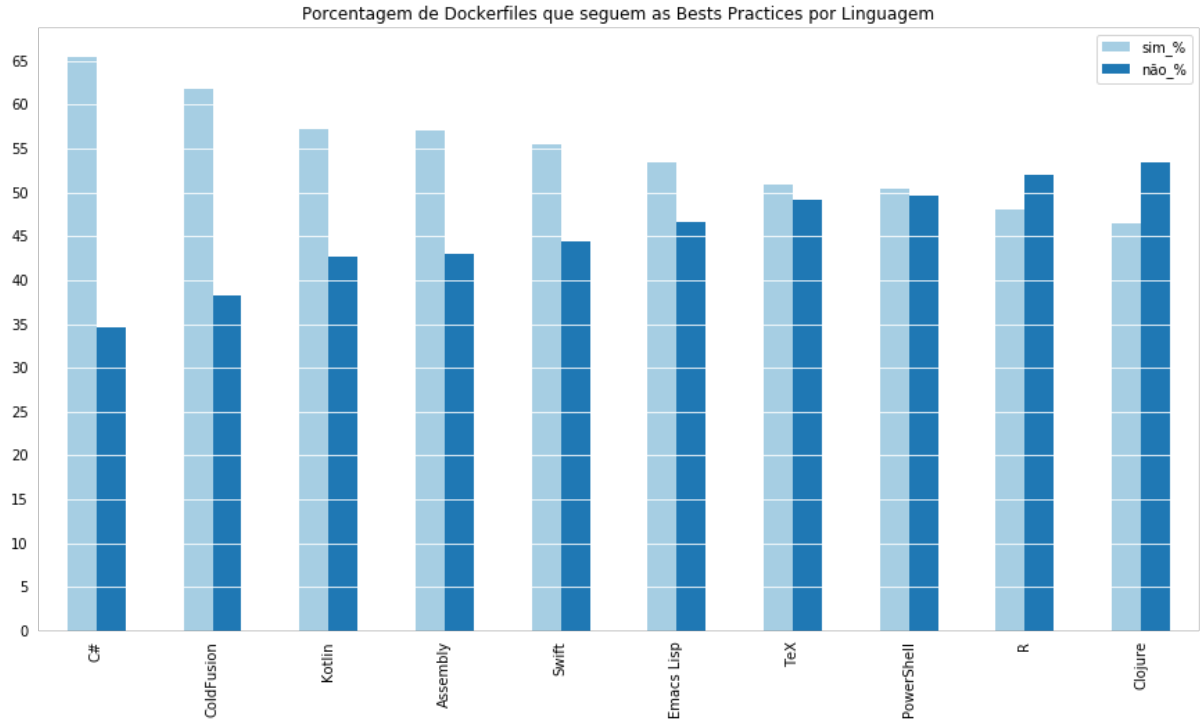


Fonte: Próprio Autor

É interessante observar a quantidade absoluta da linguagem GO (189484 no total e 75543 seguindo as Best Practices selecionadas), que é usado como linguagem principal para o próprio Docker e para o Kubernetes, que é um orquestrador de containers *Open Source*.

Por números absolutos, vemos que as linguagens seguem o padrão geral, em que a maioria dos *Dockerfiles* não seguem as *Best Practices*, porém como a pergunta abrange não os números absolutos, mas a porcentagem maior, faz-se necessário um novo gráfico descrito aqui pela Figura 3.

Figura 3 - Gráfico da porcentagem de Dockerfiles que seguem as Best Practices por Linguagem



Fonte: Próprio Autor

Como podemos verificar, C# tem a maior porcentagem de *Dockerfiles* que se adequam às Best Practices de 65.43%, mesmo com um número absoluto de *Dockerfiles* muito menor, em comparação com a linguagem GO, com apenas 2193 no total.

4.2.3 - Quais as principais razões para os *Dockerfiles* não seguirem as *Best Practices* Selecionadas?

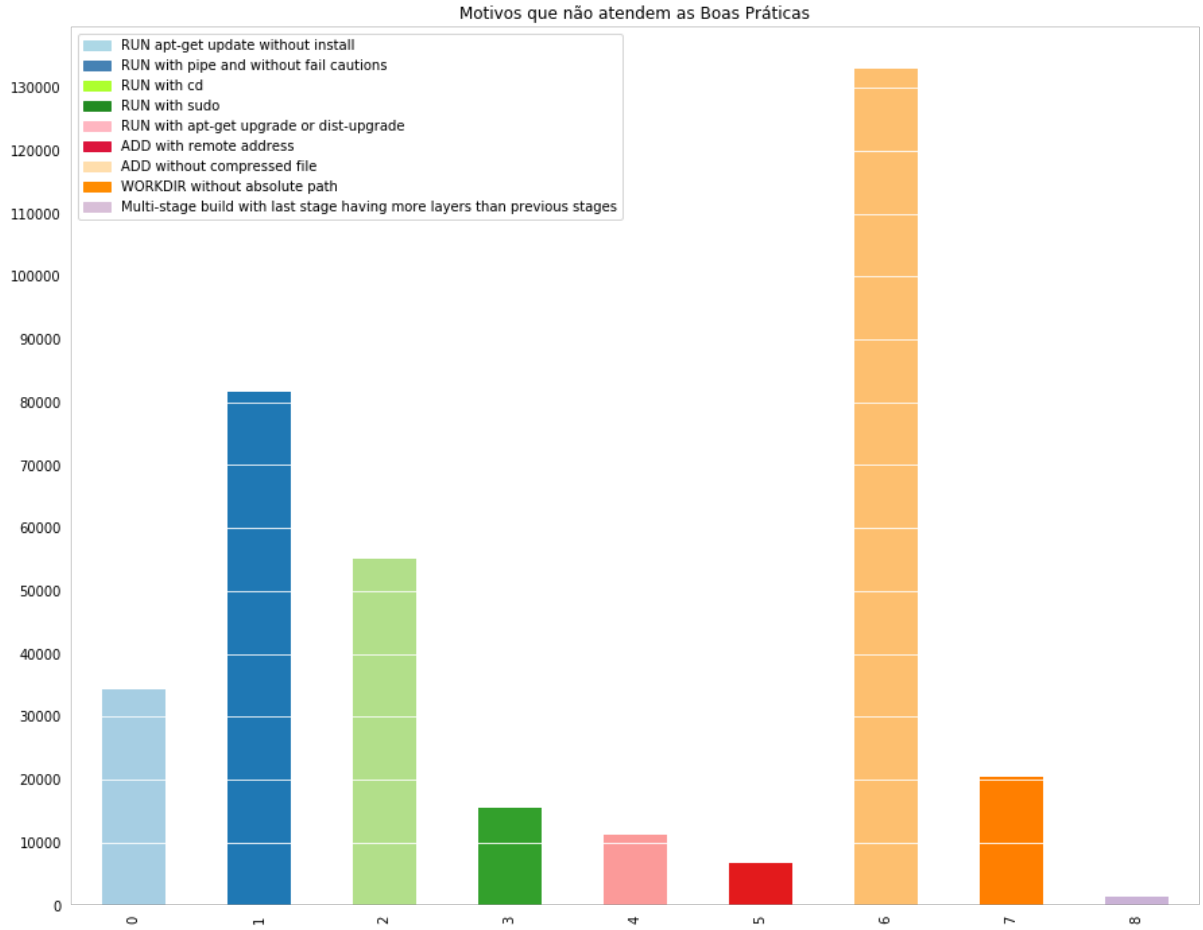
Aproveitando já os resultados da primeira pergunta, pode-se sumarizar os dados em um simples gráfico de barra e assim poder verificar qual regra é a mais desobedecida. Para melhor visualização, as regras foram colocadas de 0 a 8, seguindo a seguinte sequência:

0. RUN apt-get update sem install
1. RUN com pipe e sem *fail cautions*
2. RUN com cd

3. RUN com sudo
4. RUN com apt-get upgrade ou dist-upgrade
5. ADD com endereço remoto
6. ADD, no lugar de COPY, sem utilizar o arquivo tar
7. WORKDIR sem *path* absoluto
8. *Multi-stage build*, ou seja, arquivos com vários FROM, com o último estágio possuindo mais camadas que os estágios antecessores

O gráfico em questão está presente aqui pela Figura 4.

Figura 4 - Razões de o Dockerfile não obedecer as Best Practices



Fonte: Próprio Autor

Como pode-se verificar, a mais comum é a usar o ADD, no lugar do COPY, sem um arquivo .tar, diminuindo a clareza do código bem como dificultando a manutenibilidade do documento.

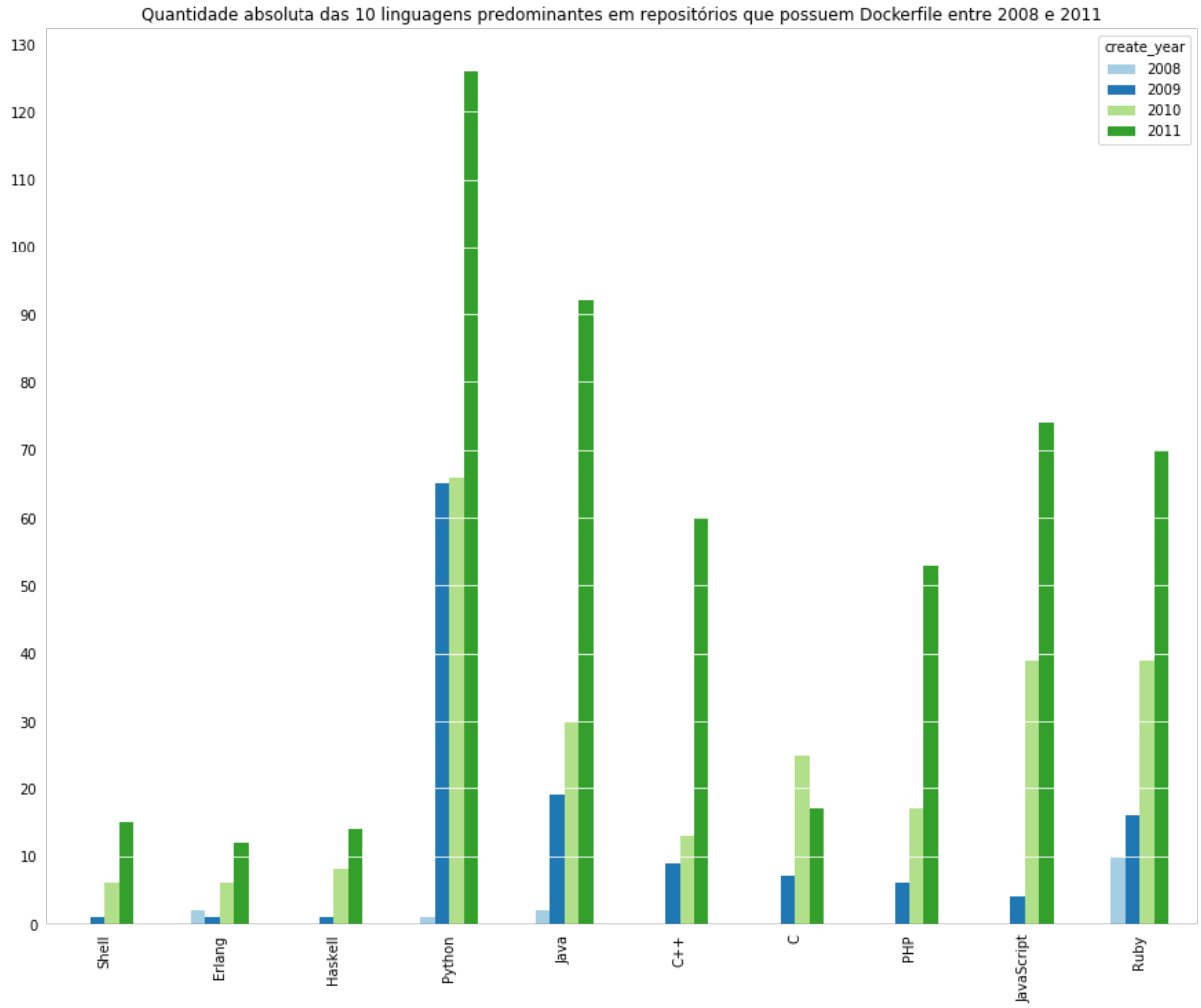
4.3 Análise sobre o crescimento de repositórios que utilizam containers

Para esta análise, uma face diferente dos dados foi verificada. Foi necessário dividir os repositórios por linguagem e por ano, para assim obter as respostas das duas perguntas a seguir.

4.3.1 - Qual a linguagem mais predominante na utilização de Docker?

Para melhor visualizar e entender o crescimento do uso de containers, foram criados três gráficos abrangendo quadriênios para melhor legibilidade dos mesmos, ou seja, de 2008 a 2011, de 2012 a 2015 e por fim de 2016 a janeiro de 2018. As Figuras 5, 6 e 7, representam respectivamente tais gráficos.

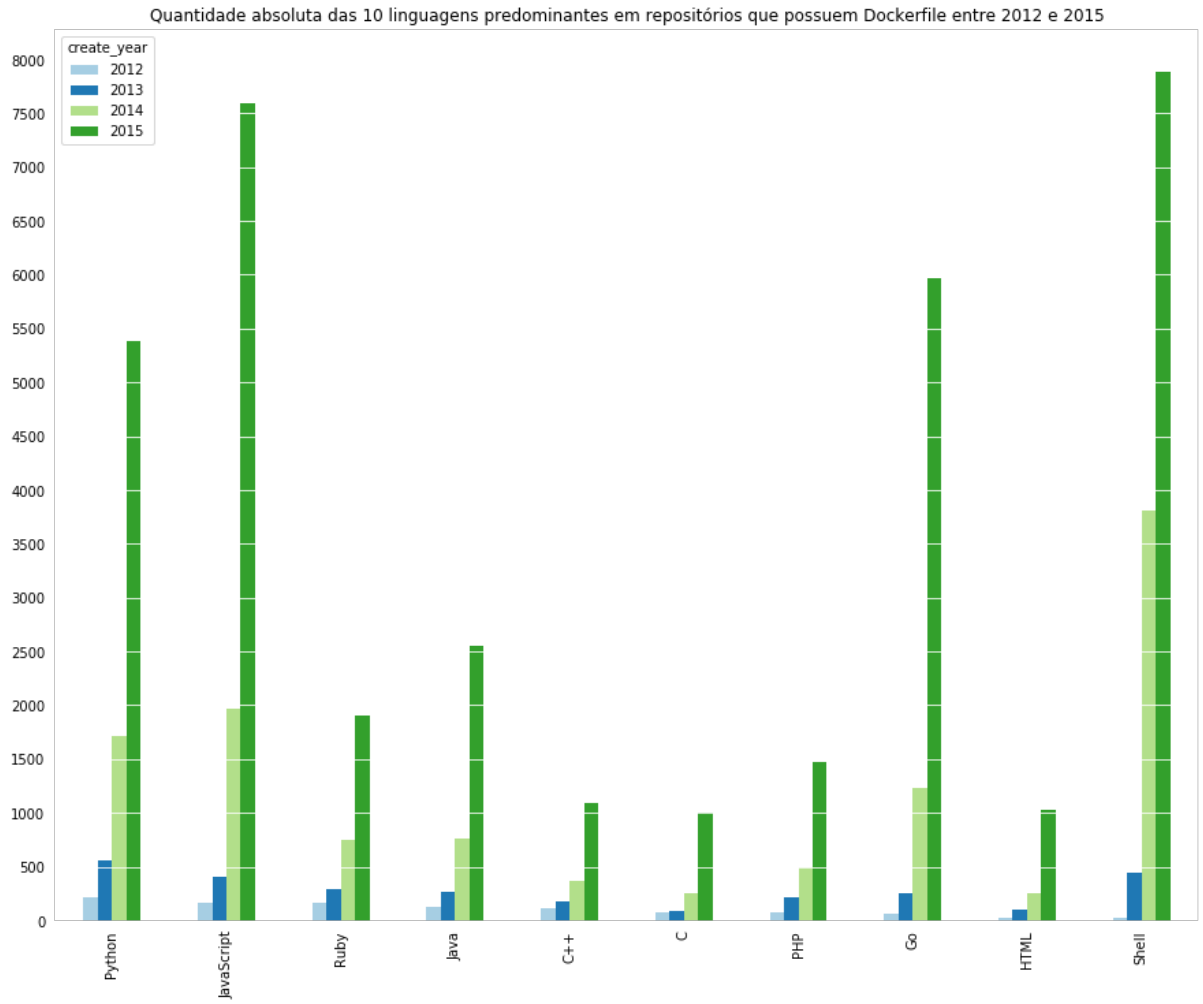
Figura 5 - Quantidade absoluta das 10 linguagens predominantes em repositórios que possuem Dockerfile entre 2008 e 2011



Fonte: Próprio Autor

Na Figura 5 é possível verificar que apenas três linguagens possuem presença em repositórios criados em 2008, que são a Erlang, Python, Java e Ruby, e com exceção da Erlang, todas as três demonstraram um ótimo crescimento nos próximos anos, em especial ao Python que se vê predominante entre repositórios criados entre esses anos.

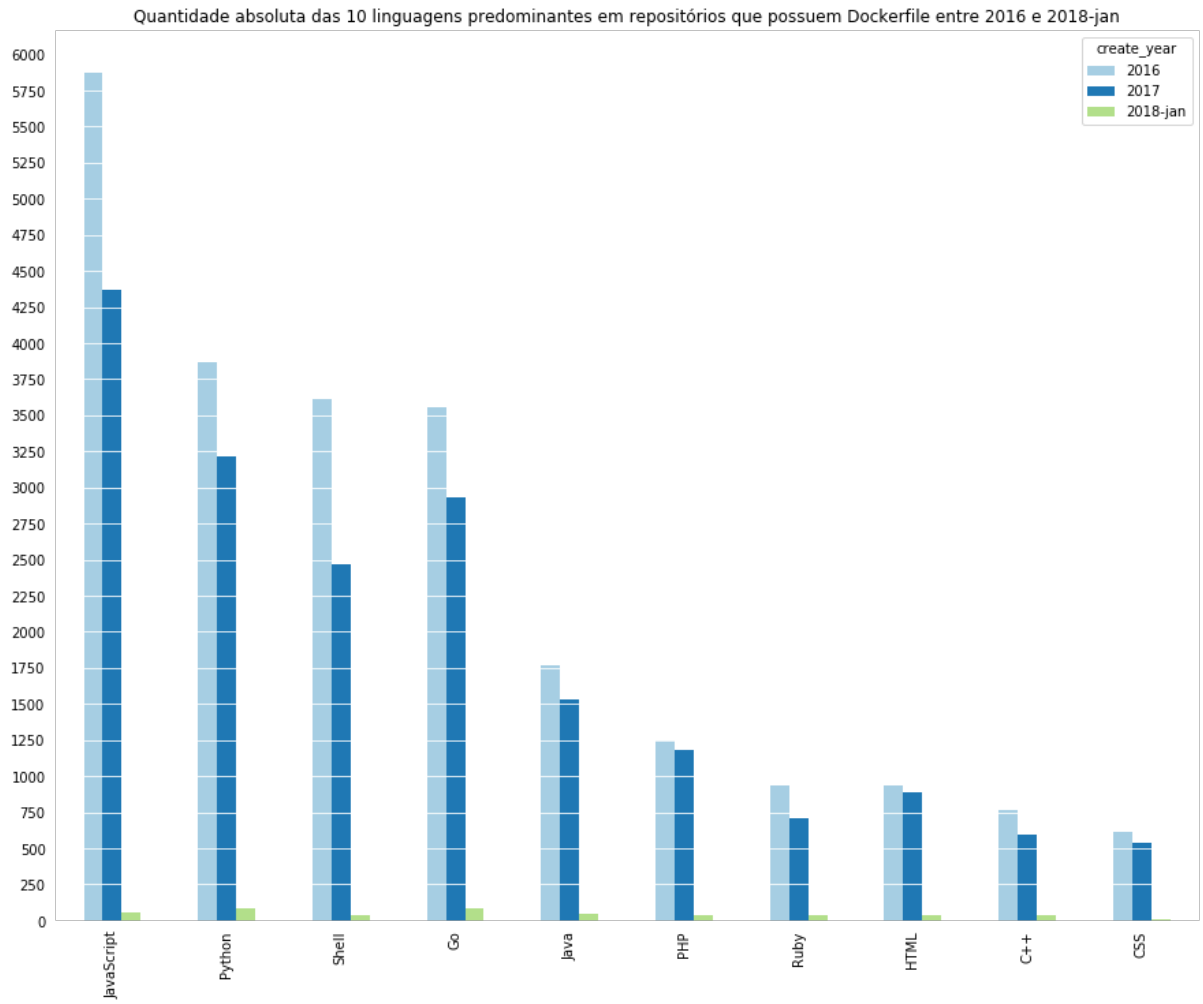
Figura 6 - Quantidade absoluta das 10 linguagens predominantes em repositórios que possuem Dockerfile entre 2012 e 2015



Fonte: Próprio Autor

A esta altura, o Python dá lugar ao Shell e ao JavaScript. Vale salientar que repositórios com Shell predominante são mais comumente compostos apenas do *Dockerfile* e de um script inicializador.

Figura 7 - Quantidade absoluta das 10 linguagens predominantes em repositórios que possuem Dockerfile entre 2016 e 2018-jan



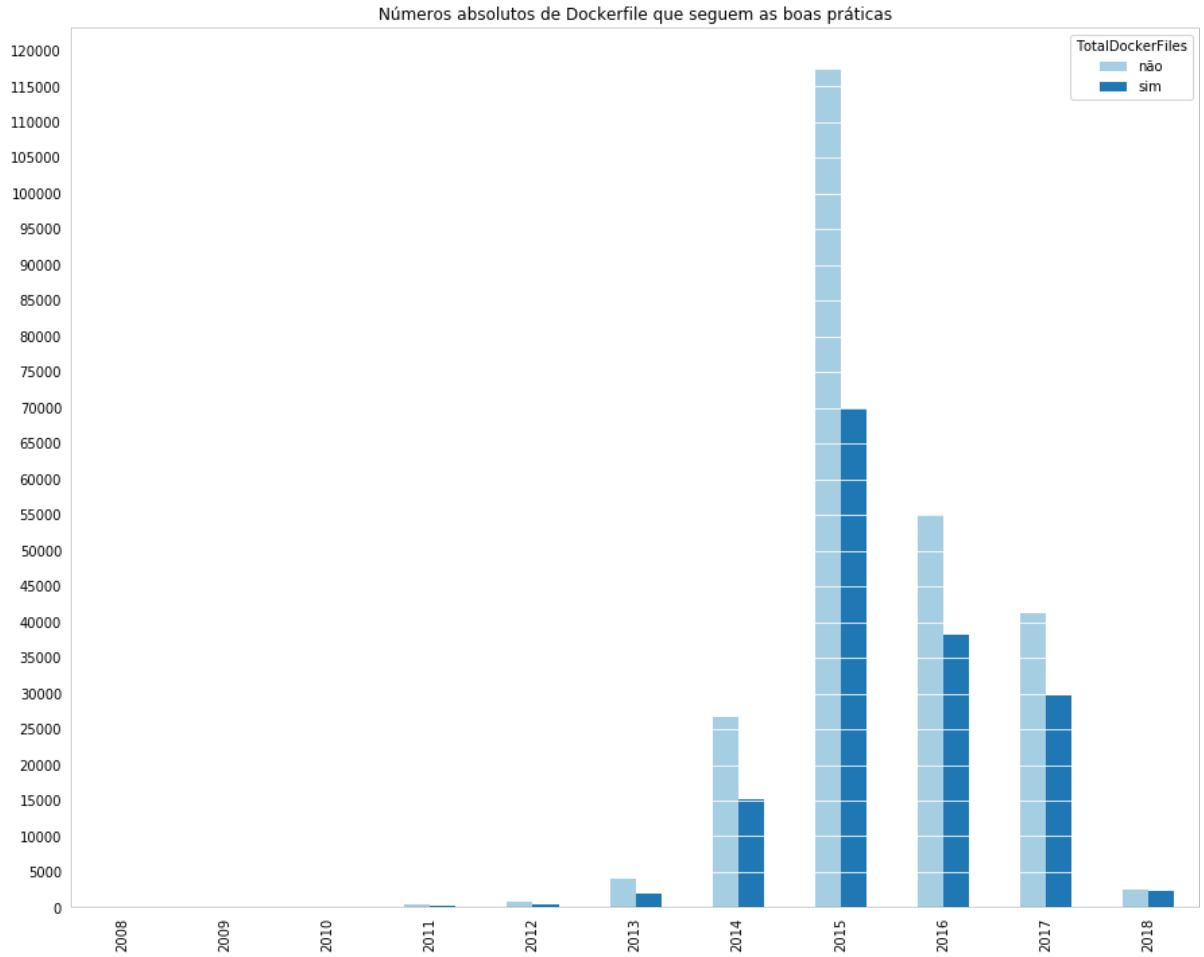
Fonte: Próprio Autor

No último período analisado, pode-se notar uma informação nova, que será interessante averiguar em estudos futuros, que é a baixa do uso de containers entre os anos 2015, 2016 e 2017. Mesmo com a baixa de utilização, JavaScript se mantém na frente como linguagem predominante, estando presente em um total de 20548 repositórios.

4.3.2 - Qual ano obteve maior crescimento na criação de repositórios que utilizam a tecnologia *Docker* com as *Best Practices* Selecionadas?

O conjunto de dados foi focado na distribuição de *Dockerfiles* por ano. A ideia é primeiro tentar conseguir a resposta utilizando gráficos simples, como demonstra a Figura 8.

Figura 8 - Números absolutos de Dockerfile que seguem as Best Practices por ano



Fonte: Próprio Autor

Visualmente é possível verificar que existiu um grande salto entre os repositórios criados em 2014 e 2015, que representa um acréscimo de aproximadamente 55 mil. Além disso, é possível observar melhor a queda brusca sofrida nos anos 2016 e 2017, caindo quase pela metade de *Dockerfiles* que seguem as *Best Practices* dentre 2015 e 2016. Porém, para ter certeza, será necessária uma análise mais minuciosa, e para isto verificamos comparamos a porcentagem de crescimento entre os anos, utilizando a seguinte formula:

$$\frac{V_0 - V_f}{V_0} x^{100}$$

Onde V_0 é o valor inicial e V_f o valor final. Com isso, obteve-se a tabela 5, que destaca

que, apesar de visualmente 2015 ser maior, o mais crescimento está entre 2013 e 2014 (descartando-se o ano intervalo inicial de 2008 e 2009, onde ocorre o aumento inicial de repositórios com as características inferidas).

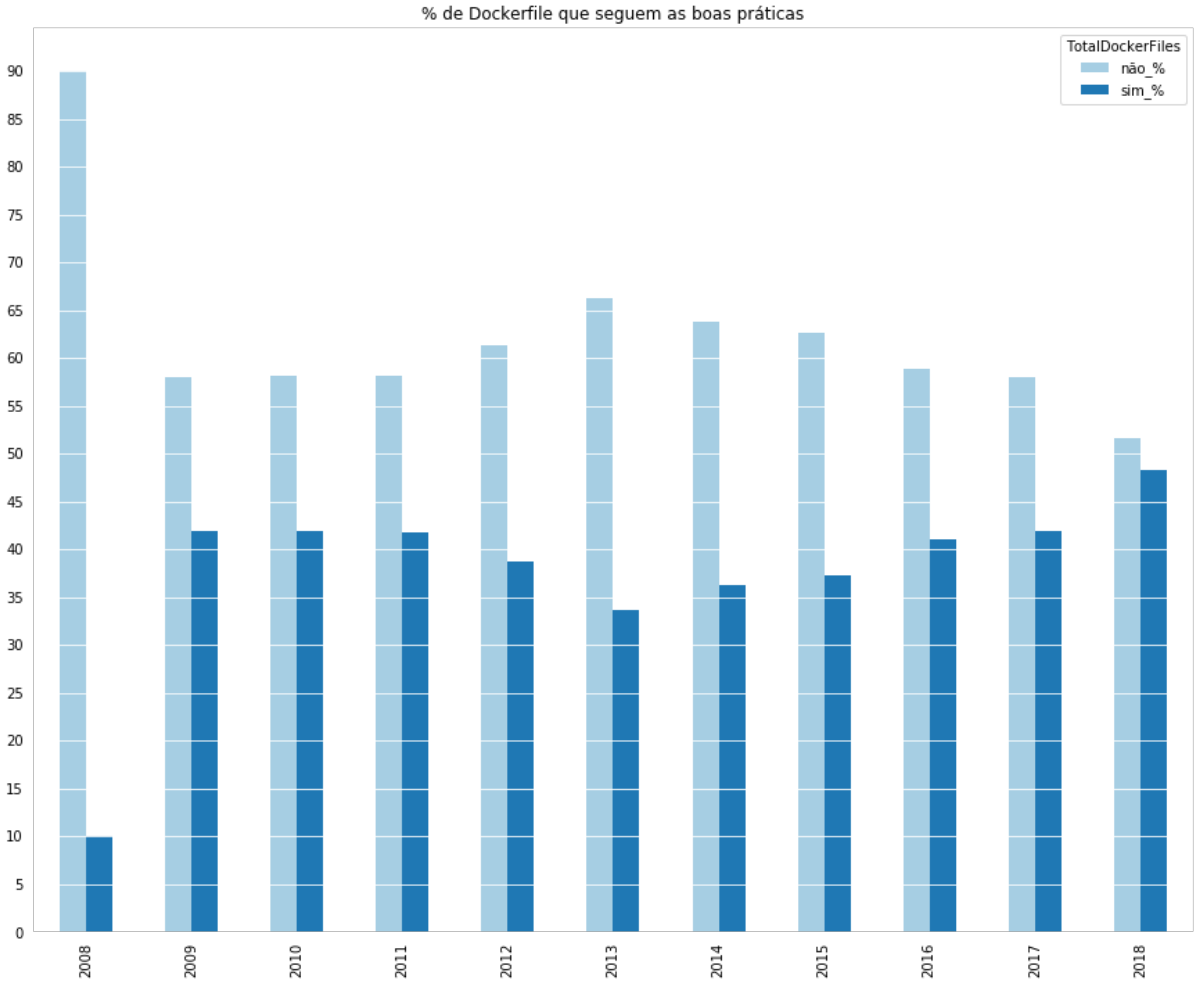
Tabela 5 – Crescimento de Dockerfile em repositório por ano

Ano	Total de Dockerfiles	% aproximada de crescimento	Total de Dockerfiles que seguem as Best Practices selecionadas	% aproximada de crescimento
2008	10	100%	1	100%
2009	93	830%	39	3800%
2010	215	131%	90	130%
2011	287	219%	287	218%
2012	1390	102%	538	87%
2013	6060	335%	2043	279%
2014	41814	590%	15144	641%
2015	187246	347%	69933	361%
2016	93304	-50%	38317	-45%
2017	71082	-23%	29801	-22%
Janeiro de 2018	4810	-	2323	-

Fonte: Próprio Autor

Além disso, é interessante trazer mais um gráfico, verificando a porcentagem de Dockerfiles que seguem as Best Practices. A Figura 9 a seguir representa este gráfico, e com ela podemos ver um pequeno padrão, apesar pequeno, nos leva a entender que existe um padrão entre o crescimento de Dockerfiles com a porcentagem de Dockerfiles que seguem as Best Practices selecionadas, sendo o ano de 2013 com o pior percentual, caso se ignore o ano inicial de 2008 que possuía uma amostragem muito pequena.

Figura 9 - Porcentagem de Dockerfiles que seguem as Best Practices por ano



Fonte: Próprio Autor

5 *Conclusão*

O desenvolvimento deste estudo propiciou uma nova luz sobre como os containers são utilizados pela comunidade *Open Source*. Ficou claro que a maioria não segue as *Best Practices* propostas pela documentação do *Docker*, e que a data de criação também tem certa influência, ou seja, a "idade" de um repositório influencia na qualidade de escrita do *Dockerfile*.

Acredito que a disponibilização dos dados obtidos auxilie a comunidade de Data Science a encontrar novas informações relevantes para o tema abordado.

Vale pontuar que a queda brusca verificada entre os anos de 2015, 2016 e 2017 pode apresentar uma certa anomalia ou confirmaria o hypecycle do Gartner[16], onde uma tecnologia após um imenso hype começa a decair em uso para após isso começar a ter um uso mais contínuo, o que seria interessante de se verificar mais profundamente em trabalhos futuros.

A principal contribuição deste trabalho é o data-set que está disponibilizado via Google Drive e possui link cadastrado no README do repositório de scripts. O poderá servir para outros trabalhos que tenham interesse de inquerir sobre Containers no GitHub, explorando outras informações disponíveis no mesmo.

5.1 Ameaças e limitações ao estudo

Algumas ameaças e limitações foram identificadas quanto a validade do estudo:

- Pelas limitações da plataforma *Github* e limitações de recursos de Hardware, o estudo pode não abranger todos os repositórios existentes.
- A base presente no *Kaggle* possui atualizações apenas até primeiro de janeiro de 2018, não sendo possível localizar os repositórios nos meses posteriores.
- A análise não compreendeu todo o conjunto de dados disponível, verificando e comparando outras faces dos repositórios capturados, novamente por limitações de Hardware, as análises simples presentes no estudo consumiam uma quantidade superior a 12 Gigabytes de memória RAM, incapacitando qualquer correlação mais complexa que pudesse vir a ser feita.

[16] https://www.gartner.com/en

- O documento que descreve as *Best Practices* na construção de *Dockerfiles* possui mais pontos dos que o os verificados no estudo, porém com uma conotação semântica, podendo a quantidade de arquivos que não estão utilizando todas as boas práticas aumentar.
- As analise considera que caso um Dockerfile não obedeça apenas uma regra, já se enquadra como não utilizador das Best Practices, o que pode ser injusto, necessitando assim outra analise quanto a cobertura de regras.

5.2 Lições aprendidas

Durante o estudo, certas constatações foram observadas e levantadas:

- Boa parte dos repositórios encontrados no GitHub possuem linguagem majoritária como *Shell* ou *Dockerfile*, sendo a soma dos dois tipos de repositórios quase a mesma (totalizando 19375 repositórios) da linguagem mais predominante *JavaScript* (com 20548 repositórios), indicando que o Github é apenas usado como repositório para os arquivos de configuração por esses tipos de projeto e que a imagem gerada é uma modificação simples de imagens previas.
- Mais da metade dos arquivos de configuração do *Docker Compose* capturados não possuem chave indicativa de para qual versão especifica aquele arquivo foi construído.
- Foram encontrados alguns repositórios que possuíam arquivos nomeados como *docker-compose* e *Dockerfile* em extensão .exe para Windows.
- Existem poucos *Dockerfile* que utilizam as vantagens da construção em multi-estágios, porém foi observado que existem arquivos que possuem quantidade superior a 25 estágios, ou seja, possuem em sua configuração ao menos 25 entradas ao comando FROM
- Apesar da recomendação sobre utilizar o COPY no lugar do ADD em casos de apenas cópia simples de arquivos, a grande maioria dos repositórios capturados que não seguem as boas práticas, falharam nesse quesito.

5.3 Trabalhos futuros

- **Encontrar um padrão nos docker-compose.yml** - Conseguir diferenciar um *.yml* comum de um *.yml* de configuração para containers;

- **Aprofundar o estudo sobre o declínio do uso de *Dockerfiles*** - Descobrir o que causou um declínio tão brusco entre o espaço de tempo especificado no trabalho;
- **Pluralizar o tipo do container** - Conseguir informações de outros tipos de containers, além de orquestradores (Kubernetes, Rancher, etc).
- **Obter mais informações sobre outros arquivos presentes no container** - Alguns atributos/análises necessitam de mais dados para conclusões, como por exemplo o *.dockerignore*.
- **Aprofundar e considerar cobertura sobre as Best Practices -** O trabalho atual não considera porcentagem de cobertura das Best Practices, sendo assim, um trabalho futuro seria analisar sobre o quanto das Best Practices os Dockefiles cobrem.

Referências Bibliográficas

[1] SANTORO, Corrado et al. Wale: A Dockerfile-Based Approach to Deduplicate Shared Libraries in Docker Containers. In: 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech). IEEE, 2018. p. 785-791.

[2] PAHL, Claus. Containerization and the paas cloud. IEEE Cloud Computing, v. 2, n. 3, p. 24-31, 2015.

[3] CITO, Jürgen et al. An empirical analysis of the Docker container ecosystem on GitHub. In: Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on. IEEE, 2017. p. 323-333.

[4] Best practices for writing Dockerfiles, link: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

[5] DUA, Rajdeep; RAJA, A. Reddy; KAKADIA, Dharmesh. Virtualization vs containerization to support paas. In: Cloud Engineering (IC2E), 2014 IEEE International Conference on. IEEE, 2014. p. 610-614.

[6] AQUA, Aqua. Docker Alternatives: Learn about Docker alternatives, how each alternative differs from Docker, and discover the road ahead for Docker alternatives. Disponível em: https://www.aquasec.com/wiki/display/containers/Docker+Alternatives+-+Rkt%2C+LXD%2C+OpenVZ%2C+Linux+VServer%2C+Windows+Containers. Acesso em: 18 Nov. 2018.

[7] ANDERSON, Charles. Docker [software engineering]. IEEE Software, v. 32, n. 3, p. 102-c3, 2015.

Referências Bibliográficas

[8] COMBE, Theo; MARTIN, Antony; DI PIETRO, Roberto. To Docker or not to Docker: A security perspective. IEEE Cloud Computing, v. 3, n. 5, p. 54-62, 2016.

[9] MOCKUS, Audris; FIELDING, Roy T.; HERBSLEB, James. A case study of open source software development: the Apache server. In: Proceedings of the 22nd international conference on Software engineering. Acm, 2000. p. 263-272.

[10] LERNER, Josh; TIROLE, Jean. The open source movement: Key research questions. European economic review, v. 45, n. 4-6, p. 819-826, 2001.

[11] GOUSIOS, Georgios; SPINELLIS, Diomidis. GHTorrent: GitHub's data from a firehose. In: Mining software repositories (msr), 2012 9th IEEE working conference on. IEEE, 2012. p. 12-21.

Anexos

[1] Tutorial de Kernel do Kaggle para iniciantes: https://www.kaggle.com/m2skills/datasets-and-tutorial-kernels-for-beginners

[2] Repositório contendo scripts e dados extraídos: https://github.com/rafjordao/Github_Docker_Extractor