BRENO DA SILVA DE OLIVEIRA

Proposta de implantação de processos automatizados para desenvolvimento e entrega de softwares através de *pipelines* de CI/CD

LAVRAS – MG
2021

BRENO DA SILVA DE OLIVEIRA

Proposta de implantação de processos automatizados para desenvolvimento e entrega de softwares através de *pipelines* de CI/CD

Relatório de Estágio Supervisionado apresentado à Universidade Federal de Lavras como parte das exigências do curso de Sistemas de Informação, para obtenção do título de Bacharel.

Prof. DSc. Ahmed Ali Abdalla Esmin
Orientador

LAVRAS - MG
2021

AGRADECIMENTOS

Aos meus pais, Clauber Ferreira e Regilane Lourdes, pelo apoio e incentivo em todos os momentos da minha vida, que foram a sustentação para a realização deste trabalho.

Aos meus irmãos, André de Oliveira e Alex de Oliveira, pela fraternidade e inspirações que nortearam minhas decisões até aqui.

Ao meus tios Jurema Lucia e Ademilson Sebastião (in memorian), pelo apoio e presença fraternal em minha vida, que serviram de alicerce para minhas realizações.

À minha avó, Maria Olice Pereira (in memorian), pelo companheirismo e lições de vida, imprescindíveis para minha formação como pessoa.

Aos amigos e amigas, em especial Guilherme Oliveira, que guiaram minhas escolhas e me apoiaram nos momentos difíceis percorridos neste período.

Aos professores, colegas de curso e de trabalho, especialmente Ahmed Esmin, Eduardo Petrini e Renata Teles, pelos ensinamentos, trocas de experiências e amparo, essenciais para meu crescimento pessoal e profissional.

Por fim agradeço a todos os profissionais da educação, servidores públicos e cientistas que lutam pela educação pública de qualidade para o país.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

SUMÁRIO

1 Introdução

O mercado de desenvolvimento de *softwares* é tradicionalmente volátil, com mudanças frequentes que exigem constantes atualizações dos profissionais e das empresas. Diversas metodologias são utilizadas para auxiliar os profissionais a gerenciar o seu trabalho, buscando equilibrar o tempo de produção e a qualidade dos produtos desenvolvidos de modo com que as demandas do mercado sejam supridas. Entre elas se encontram as metodologias de desenvolvimento ágil.

Ainda que implementem as metodologias ágeis em sua estrutura, as organizações se deparam com diversos problemas relacionados indiretamente ao modelo, que causam falhas graves nos processos adotados. Alguns desses problemas estão relacionados a processos manuais de desenvolvimento, testes e entrega dos produtos de *software* para os clientes finais. Esses processos podem ser lentos, complexos e demandam tempo e conhecimento dos profissionais, podendo afetar a execução dos prazos e impactar diretamente nos negócios das empresas.

Nesse contexto se encontra a empresa Moka Mind Inteligência Artificial onde exerci meu estágio e onde também esse trabalho foi executado. Durante meu tempo de contribuição na organização foi possível notar que, embora houvesse um forte esforço por parte dos membros em executar os processos do *Scrum*, comumente a complexidade e a dificuldade exigida para cada etapa do ciclo de desenvolvimento dos *softwares* impactava negativamente no cumprimento dos prazos planejados para as atividades.

Para tanto esse relatório busca propor uma implantação de processos automatizados nas etapas que envolvem a criação e entrega dos *softwares* aos clientes, através do desenvolvimento de *Pipelines de CI/CD*, de modo a auxiliar a empresa a padronizar os processos envolvidos e melhorar sua capacidade no cumprimento dos prazos determinados, tendo *softwares* mais estáveis e com maior confiabilidade.

Além deste capítulo introdutório, este relatório de estágio está organizado como segue. O Capítulo 2 apresenta a empresa na qual o trabalho foi proposto. O Capítulo 3 apresenta os conceitos e tecnologias essenciais para o entendimento do projeto. No Capítulo 4 são demonstradas as atividades desenvolvidas para o cumprimento da proposta. Por fim, o Capítulo 5 apresenta as considerações finais acerca das atividades desempenhadas durante o período de estágio.

2 Sobre a organização

Este capítulo será dedicado a detalhar as informações a respeito da empresa na qual o presente trabalho foi concebido e desenvolvido. Serão apresentadas suas áreas de atuação e os processos organizacionais adotados para o desenvolvimento dos seus produtos de *softwares*.

2.1 Moka Mind Inteligência Artificial

A empresa Moka Mind é especializada no desenvolvemento de robôs (*bots*) para gestão de infraestrutura predial (elétrica, ar condicionado, iluminação, etc) a partir da coleta de dados via dispositivos *IoT* (internet das coisas), computação em nuvem e inteligência artificial.

Ela foi criada a partir da necessidade de suprir demandas específicas vindas através da empresa parceira LCS Link Engenharia, que já contava com diversos clientes de grande porte, como bancos, usinas, empresas de telefonia, etc.

Com sua sede situada em São Paulo e tendo em Lavras uma matriz especializada em desenvolvimento de softwares, onde a empresa conta com uma equipe de 6 membros, atuo como *Full Stack Developer*, atualmente desempenhando o papel de *Scrum Master* como parte do *Framework* de desenvolvimento ágil Scrum adotada pela organização. Os demais membros também compõe o *Scrum Team*, o *Product Owner* e o *Chapter Lead.*

Durante o tempo de experiência na organização tive a oportunidade de trabalhar em diversos projetos distintos, onde foi possível adquirir uma visão macro dos processos de desenvolvimento e entrega dos produtos da empresa, tema onde o presente trabalho se enquadra.

2.2 Processo Organizacional

O fluxo de desenvolvimento dos sistemas na empresa se faz utilizando uma combinação de práticas recomendadas pela indústria de *software*. O modelo adotado se pauta no *Feature-driven Development* (FDD), ou desenvolvimento orientado a recursos; um processo que busca entregar partes funcionais dos *softwares* de maneira incremental e iterativa, em ciclos constantes, como institui o *framework* Scrum.

As *Features* desenvolvidas são gerenciadas pelo sistema de controle de versões *Git* e o ambiente *GitLab*, ambos abordados nos capítulos seguintes. A equipe de desenvolvimento

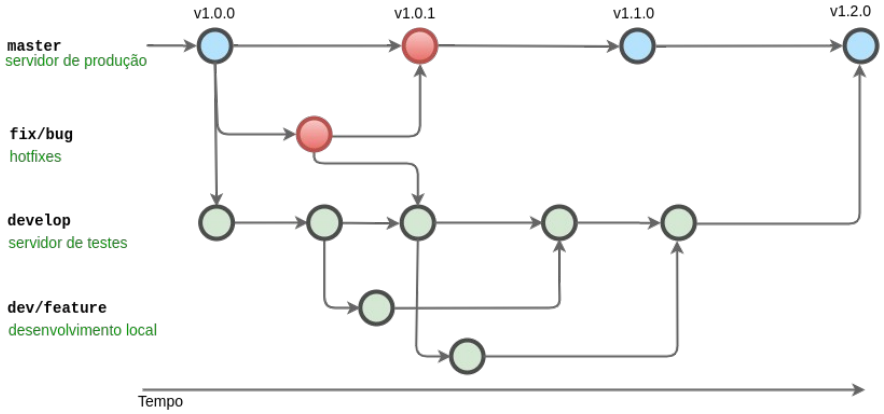
integrante do *Scrum Team* segue o modelo de trabalho conhecido como *Git Workflow*, exibido na *Figura 1*, em que um *branch* (ramificação do código) chamada *dev/feature*, sendo feature o nome específico da atividade a ser desenvolvida, é criada através de uma cópia local do *branch develop*, que contém a versão do código corrente no ambiente de testes.

Figura 1 – *Git Workflow*

Fonte: https://docs.gitlab.com/ee/topics/gitlab_flow.html

A seguir, na *Figura 2*, é detalhado o atual modelo arquitetural dos *branches* utilizados pela empresa.

Figura 2 – *Workflow* dos *Branches*



Fonte: Autor

Um pedido de *Merge* entre os *branches dev/feature* e *develop* é feito pelo desenvolvedor sempre que uma nova *feature* estiver apta a ser revisada, validada e testada no ambiente de testes. Para tanto, as seguintes etapas manuais são necessárias por parte do *Tester*, membro também integrante do *Scrum Team*:

- Revisão do Código (*Code Review*).
- *Download* manual do branch em seu ambiente local.
- Execução manual de *Testes de integração* e *Testes unitários*.
- *Testes funcionais*.

Esta estapa é conhecida como Teste de Software.Concluidos todos os passos, o pedido de *Merge* é aprovado no ambiente do *GitLab* e, então, novos processos manuais são necessários, desta vez por um Administrador com permissões de acesso ao ambiente de testes:

- Efetuar a conexão com o servidor de testes, via protocolo *SSH.*
- Efetuar *download* das novas features contidas no *branch develop* e instalar suas dependências.
- Efetuar o *build* do projeto atualizado e disponibilizar a nova *feature* para testes públicos.

Esta etapa é conhecida como *Deploy ou* Implantação. Os mesmos passos são feitos para *Teste* e *Deploy* no servidor de Produção, onde se encontra a versão estável do produto para uso dos clientes; após a aprovação e validação pela equipe responsável pelos testes de usabilidade das novas *features* no servidor de testes. Assim um conjunto de novas *features* são enviadas para o *Branch master*, onde uma nova *Release* é lançada para os clientes.

O presente documento implementa técnicas que visam eliminar as todas as possíveis etapas manuais, criando processos de testes e *deployments* automatizados, de maneira a buscar o cumprimento dos Atributos de Qualidade presentes na norma ISO/IEC 9126:

- Funcionalidade
- Confiabilidade
- Usabilidade
- Eficiência
- Manutenibilidade
- Portabilidade

3 Fundamentação Teórica

Este capítulo tem como objetivo abordar todos os conceitos necessários para a compreensão dos trabalhos realizados.

3.1 Métodos Ágeis

As metodologias ágeis, em desenvolvimento de *software*, compõe um conjunto de técnicas que visam acelerar e padrozinar a criação e entrega dos produtos, de modo consistente e fracionados, possibilitando revisões e *feedbacks* constantes com o cliente final. O framework de métodologia ágil adotado pela organização, onde o presente trabalho foi executado, é o Scrum.

3.1.1 Scrum

Para Audy (2015), o *framework* Scrum possui um fluxo iterativo-incremental, o que significa que o desenvolvimendo dos produtos são feitos em camadas. Permanentemente é iniciada uma nova camada, construindo uma parte do todo; entregamos, validamos e então reiniciamos na escala de dias ou semanas. São realizadas reuniões diárias, dentro de ciclos semanais, chamados *Sprints*, em uma estratégia mensal para a construção de um produto.

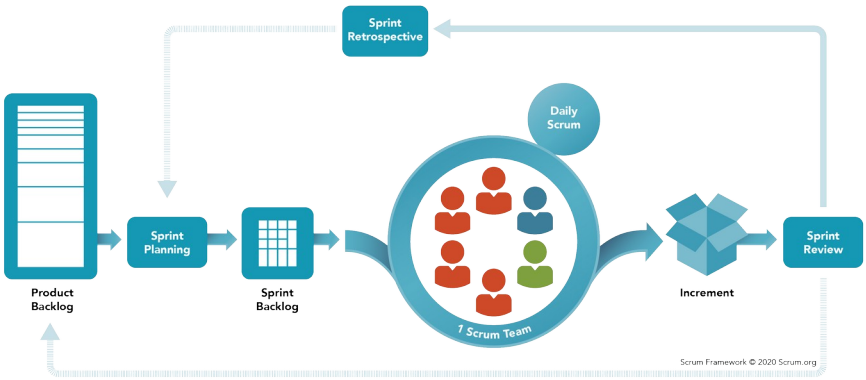
De acordo com Sabbagh (2013), Scrum possibilita que seja entregue, desde o começo do projeto e com certa frequência, partes do produto funcionando. Cada uma dessas entregas proporciona retorno ao investimento realizado pelos clientes do projeto, além de possibilitar o seu *feedback* rápido sobre o produto para que se realizem as mudanças ou adições necessárias durante a sua execução, sem afetar os prazos estabelecidos.

Scrum, para Sabbagh (2013), visa à redução dos riscos de negócios do projeto pela colaboração com os clientes e demais partes interessadas durante todo o seu decorrer. Os riscos também são reduzidos com a produção em ciclos curtos e entregas frequentes de partes prontas do produto, partindo-se das mais importantes em direção às menos importantes.

Segundo Schwaber e Sutherland (2020, p. 6), “A unidade fundamental do Scrum é um pequeno time de pessoas, um *Scrum Team*. O *Scrum Team* consiste em um *Scrum Master*, um *Product* O*wner* e *Developers*.”

Na *Figura 3*, é mostrado os artefatos que compõe os eventos do Scrum dentro de uma *Sprint*.

Figura 3 – Componentes do Scrum



Fonte: https://www.scrum.org/resources/scrum-framework-poster

A partir dos processos do *framework* Scrum adotados pela organização, o presente trabalho se organiza de forma a aplicar técnicas de DevOps na cultura da empresa, proporcionando maior confiabilidade e velocidade nos ciclos da métodologia ágil.

3.2 DevOps

O DevOps é uma nova metodologia de entrega de aplicações de *software*, por meio da colaboração entre as equipes de desenvolvimentos e de operações, se opondo à abordagem tradicional onde o desenvolvimento e as operações estão distanciadas e separadas, cada uma compondo seus nichos organizacionais. Mesmo que a palavra ágil não origine diretamente associada a esta metodologia, ressalta-se que seus criadores Patrick DuBois e Andrew Clay a denominam como "*Agile Infrastrutucture*" (DEBOIS, 2008).

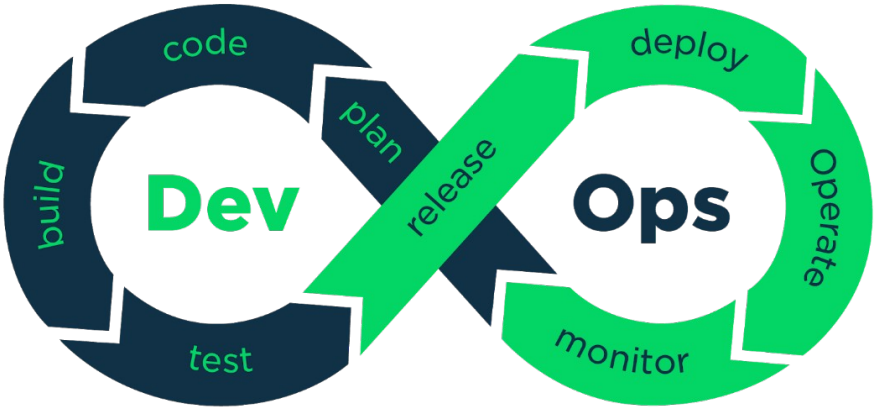
DevOps origina de Dev + Ops, ou seja, *Development* + *Operations*, o que remete para a união de equipes, modificando os nichos existentes em um grupo de equipes que trabalham em função da organização e não apenas em torno de sua atividade inserida nela, relacionando todas as diretrizes pré-xistentes no desenvolvimento de *softwares*.

Wills (2010) sugeriu a caracterização do DevOps embasada em quatro pilares: cultura (*culture*), automatização (*automation*), medição (*measurement*) e partilha (*sharing*). A sigla *CAMS* tem origem dessas quatro expressões. Anos mais tarde, Jez Humble adicionou o quinto pilar denominado *Lean* (L), passando a sigla a ser CALMS.

A execução dos conceitos que compõe o DevOps está intimamente relacionada com o desenvolvimento de *pipelines* de CI/CD, tornando possível os testes e as entregas dos

softwares de maneira automatizada e segura. Abaixo, na *Figura 4*, é mostrado um exemplo de modelo do ciclo DevOps.

Figura 4 – O Ciclo DevOps



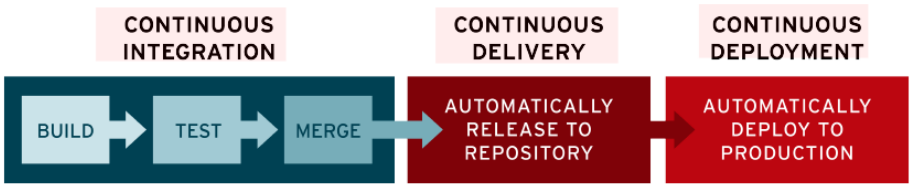
Fonte: https://technology.pl-inetum.group/en/what-exactly-is-devops

3.3 CI/CD

O CI/CD aplica monitoramento e automação contínuos a todo o ciclo de vida das aplicações, incluindo as etapas de teste e integração, além da entrega e implantação. Juntas, essas práticas relacionadas são muitas vezes chamadas de "*pipeline* de CI/CD" e são compatíveis com o trabalho conjunto das equipes de operações e desenvolvimento com métodos ágeis (Redhat, 2021).

O significado dos acrônimos CI e CD são Integração Continua (*Continuous Integration*), Entrega Contínua (Continuous Delivery) e Implantação Contínua (Continuous Deployment) conforme é mostrado na *Figura 5*.

Figura 5 – Diagrama CI/CD



Fonte: https://www.redhat.com/pt-br/topics/devops/what-is-ci-cd

3.3.1 Intregração Contínua (CI)

O conceito de Integração Contínua (CI) originou-se na metodologia ágil XP em uma de suas doze práticas (FOWLER, 2006). A metodologia estabelece que para que o projeto que está em desenvolvimento se adapte bem a cada uma das alterações no código, é fundamental que seja realizada a integração do código em ciclos curtos e no menor período possível, para adquirir melhores resultados em previsibilidade e minimizar as incertezas no projeto (MEDEIROS, 2013).

A integração contínua (CI) é uma prática de desenvolvimento que requer que os desenvolvedores integrem o código em um repositório compartilhado várias vezes ao dia. Cada *Check-in* é então verificado por uma compilação automatizada, permitindo que as equipes detectem problemas antecipadamente (LOELIGER; MCCULLOUGH, 2013).

Existem pequenas variações das etapas, dependendo das ferramentas que escolhidas e dos processos com os quais foram concordados dentro da equipe. Os princípios básicos da CI são:

1. Fazer *check-in* do código com frequência.
2. Automatizar a parte de construção e teste.
3. Sempre testar o código localmente antes de fazer o *check-in*.
4. Nunca mesclar nenhum *branch* com falha no *branch* principal.
5. Retornar seu *status* de mal-sucedido se houve uma falha na compilação ou teste.
6. Fazer disso a prioridade quando a falha acontecer.

3.3.2 Implantação Contínua (CD)

De acordo com Azeri (2017) a termologia CD é tratada de duas maneiras. A primeira trata a termologia como *Continuous Delivery*, ou seja, Entrega Contínua. Esta metodologia objetiva minimizar os conflitos resultantes do processo de implantação do *software* ou da liberação de versão para a implantação. Esse panorama é embasado na integração contínua bem estruturada que engloba um alto nível de automação em todas as fases de desenvolvimento e construção de maneira que a liberação de uma *release* adequada e segura que possa ser desempenhada a qualquer momento. A segunda trata CD como *Continuous Deployment*, ou seja, implantação contínua. Esta metodologia assegura o *deployment* da

aplicação de maneira automática sempre que houver uma funcionalidade nova e fundamental ao projeto.

3.3.3 Pipeline

Um *pipeline* de CI/CD pode ser facilmente entendido como o caminho do processo pelo qual podemos entregar uma única unidade de *software* pronto para produção. Sua equipe escolherá quais serviços usarão para construir isso; não há uma implementação canônica única de um *pipeline* de CI/CD (ROYCE, 2018).

Reduzir custos e complexidade é um benefício crucial do emprego de um *pipeline* de CI/CD. Ao automatizar o processo e delegá-lo a um *pipeline* de CI/CD, o indivíduo não apenas libera recursos preciosos do desenvolvedor para tarefas reais de desenvolvimento de produto, mas também reduz as chances de erro. Além disso, irá ocorrer uma melhora da capacidade de resposta da equipe (COSTA, 2017).

Os elementos de um *pipeline*, no geral, podem ser resumidos em três estágios: *Build*, *Test* e *Deploy*. A realização do processo de CI/CD se deu no presente projeto, através da criação de um *pipeline* contendo instruções específicas para cada uma das etapas do processo de *build*, *tests* e *deploy* dos produtos de *software*, que serão abordadas em detalhes nos tópicos a seguir.

3.4 Protocolo SSH

Acrônimo de Secure Shell, *SSH* é um protocolo que fornece segurança para comunicações de rede. Segundo Barrett e Silverman (2001), todos os dados transferidos entre computadores conectados por uma rede são suscetíveis à manipulação por terceiros. Para evitar isso, o protocolo *SSH* criptografa automaticamente os dados na origem e os descriptografa quando chegam ao destino. Portanto, as duas partes participantes podem se comunicar sem medo de sua comunicação ser interceptada e sem se preocupar com os detalhes intrincados da criptografia.

Fácil de cumprir, pois implementado em formato de *software*, o protocolo *SSH* garante a autenticidade das informações transmitidas, além da confidencialidade. Um uso comum desse protocolo é acessar máquinas remotas, pois ele pode ser implementado por meio de uma senha salva na máquina remota.

Seu uso se deu no projeto como parte da etapa de *deployment* nos servidores de testes e produção, nos *jobs* de *Continuos Delivery* do *pipeline* desenvolvida.

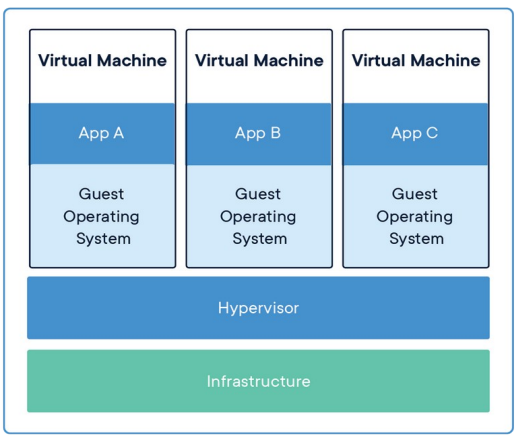
3.5 Containers

Quando uma aplicação é iniciada, ela consome principalmente *CPU*, *RAM*, espaço em disco e acesso à rede. Se várias aplicações coexistirem na mesma máquina, elas irão compartilhar os mesmos recursos, portanto, uma situação arriscada porque se um aplicativo causar uso excessivo de recursos, os outros serão afetados. A tecnologia de *contêiner* elimina precisamente essa situação, fornecendo gerenciamento de recursos ao executar aplicativos em ambientes isolados.

Os *contêineres*, no contexto do desenvolvimento de software, permitem que uma aplicação, junto com suas dependências, seja empacotada em uma imagem de *contêiner*. Os aplicativos contidos em contêineres podem ser testados e implantados como uma instância de imagem de um *contêiner* em um sistema operacional, que por sua vez deve suportar essa tecnologia.

De acordo com Saito et al. (2017), os *contêineres* são semelhantes em tecnologia às máquinas virtuais, pois ambos fornecem uma camada de isolamento entre os aplicativos. No entanto, ao contrário das máquinas virtuais, os *contêineres* compartilham o *kernel* do sistema operacional, exigindo muito menos espaço em disco. Em contraste, os *contêineres* abrangem o aplicativo com todas as suas dependências, ao contrário da abordagem da máquina virtual em que as dependências são instaladas diretamente no sistema operacional. Na *Figura 6* é apresentada a arquitetura de uma máquina virtual.

Figura 6 –Arquitetura Virtual Machine



Fonte: https://www.docker.com/resources/what-container

Todos os componentes utilizados no projeto foram adaptados para serem empacotados em *contêineres*, especificamente os *contêineres* Docker.

3.5.1 Docker

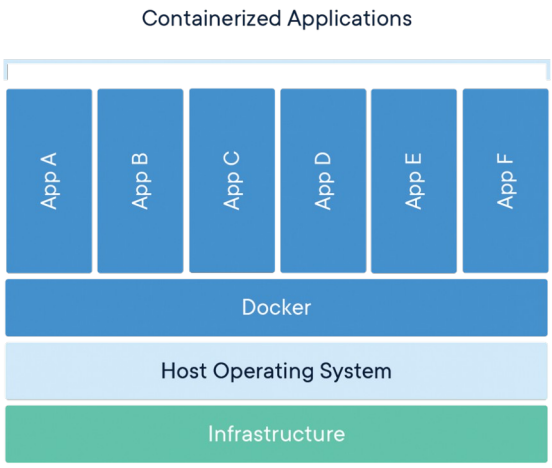
A tecnologia de *contêineres* Docker foi inaugurada no ano de 2013 pela Docker, Inc. O Docker possui tecnologia única concentrando seus requisitos desenvolvedores e operadores de sistemas para dividir as dependências de aplicativos da infra-estrutura (DOCKER, 2020).

Com o Docker podemos criar e gerenciar *contêineres* a partir de imagens chamadas Dockerfile, que contém as informações necessárias para a execução dos mesmos.

Embora o Docker e a virtualização possuam benefícios semelhantes, estes não devem ser confundidos, pois funcionam de forma bem distantes. Enquanto a virtualização isola o ambiente do sistema operacional consumindo demasiado recurso do *host*, o Docker somente utiliza o que é fundamental para a aplicação que será realizada de maneira isolada compartilhando o mesmo host com outros *contêiners* (DOCKER, 2020).

Na *Figura 7* é apresentada a arquitetura de um *contêiner* Docker.

Figura 7 –Arquitetura Docker



Fonte: https://www.docker.com/resources/what-container

O projeto de *software* utilizado como referência para o desenvolvimento do presente trabalho foi adaptado de modo com que fosse possível executá-lo em *contêineres* Docker, o

que possibilitou uma maior escalabilidade e adaptabilidade em sua execução em servidores distintos. O Dockerfile desenvolvido será detalhado nos próximos capítulos.

A orquestração dos *contêineres* criados será feito pelo Docker Compose, ferramenta utilizada para configurar e gerenciar multiplos *contêineres* Docker.

3.6 Versionamento de Código

Segundo Somasundaram (2013), um sistema de gerenciamento de versão de código é um aplicativo capaz de registrar as alterações feitas em um ou mais arquivos ao longo do tempo, de forma que seja possível reverter para uma versão específica daquele arquivo a qualquer momento.

A grande vantagem possibilitada por este tipo de sistema é a organização do projeto, pois é possível manter um histórico de desenvolvimento, permitindo o desenvolvimento de funcionalidades em paralelo a partir de um mesmo código. Além disso, permite criar uma nova versão do projeto sem alterar a versão principal.

O gerenciamento dos códigos do projeto foi feito utilizando a ferramenta Git, atualmente a mais popular entre os desenvolvedores.

3.6.1 GitLab

De acordo com Wayner (2018), os repositórios principais do universo Git são GitHub, GitLab e BitBucket, que possuem seus benefícios e desvantagens que direcionam as escolhas dos desenvolvedores conforme suas necessidades, uma vez que o foco principal desses instrumentos é conectar os desenvolvedores para que estes possam trabalhar em um mesmo código com extrema facilidade.

O GitLab, além disso, disponibiliza ferramentas de integração contínua e entrega contínua (CI/CD), sendo possível criar *pipelines* de para implantação e monitoriamento dos projetos de *software* sem dair da ferramenta.

Como os projetos da organização já se encontram no ambiente do Gitlab e pelo fato de ele conter todas as tecnologias necessárias para a implememtação das etapas do CI/CD, optei por utilizar o GitLab CI/CD para o gerenciamento dos *pipelines* desenvolvidas.

3.7 GitLab CI/CD

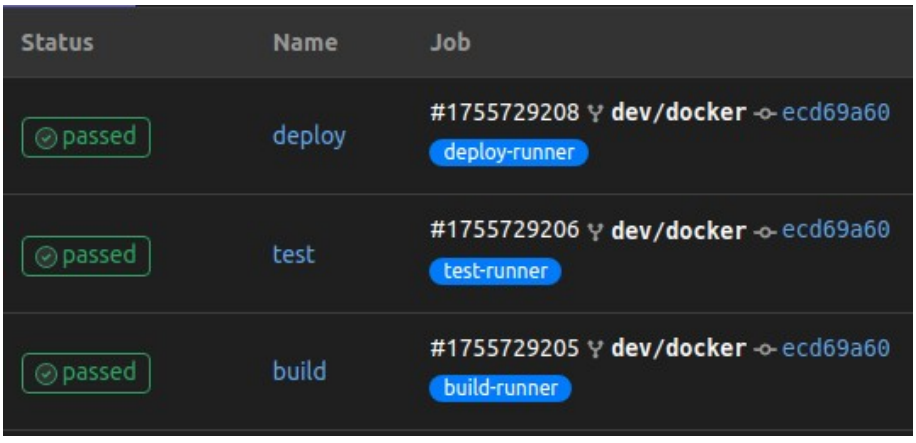
O GitLab CI/CD é um ambiente com um conjunto de ferramentas de apoio para desenvolvimento completo de *pipelines* de CI/CD. Com elas é possível implementar na prática todos os conceitos da Integração Contínua, Entrega Contínua e Implantação Contínua descritos anteriormente, de maneira automática.

As principais ferramentas que compõe o ambiente do GitLab CI/CD são: *Pipelines*, CI/CD *Variables*, *Job artifacts*, *Runners*, *Test cases*, etc. Para seu uso, o ambiente disponibiliza uma vasta gama de configurações, dentre elas *Schedule Pipelines*, *Pipeline Triggers*, *Pipelines* para *Merge Requests*, entre outras. A seguir serão detalhados os componentes do GitLab CI/CD utilizados no presente projeto.

3.7.1 Jobs

Os *Jobs* são os processos que juntos compõem os estágios de um *pipeline*. Cada estágio pode conter vários *jobs* a serem executados. Em nosso cenário será um *job* para cada etapa do *pipeline*, conforme exibido na *Figura 8*.

Figura 8 – GitLab Jobs



Fonte: Autor

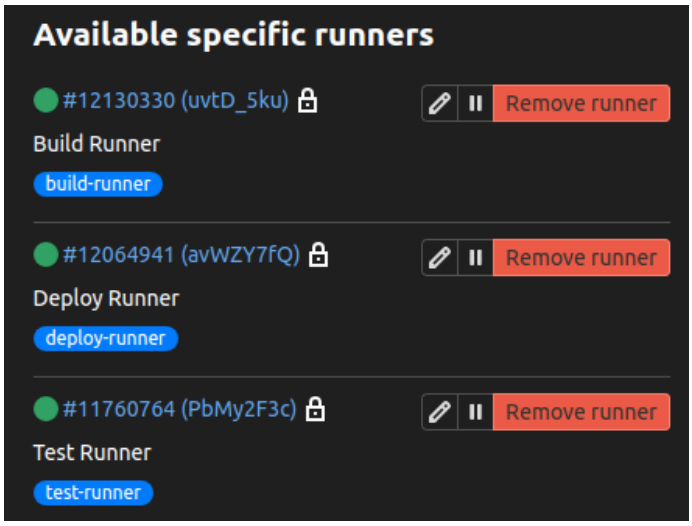
3.7.2 Runners

Os *jobs* do *pipeline* necessitam ser executados em um local definido, que são conhecidos como *Runners*. Um *Runner* é uma instância onde o CI/CD vai ser executado. Ele

pode estar localizado em servidores locais, na nuvem ou no próprio GitLab, através dos chamados *Shared Runners*.

Para melhor gerenciamento dos recursos, os *Runners* serão executados em um servidor local e, assim como os *jobs*, foi utilizada a cardinalidade 1 para 1 entre *Runners* e *Jobs*, conforme é mostrado na *Figura 9*.

Figura 9 – GitLab *Runners*



Fonte: Autor

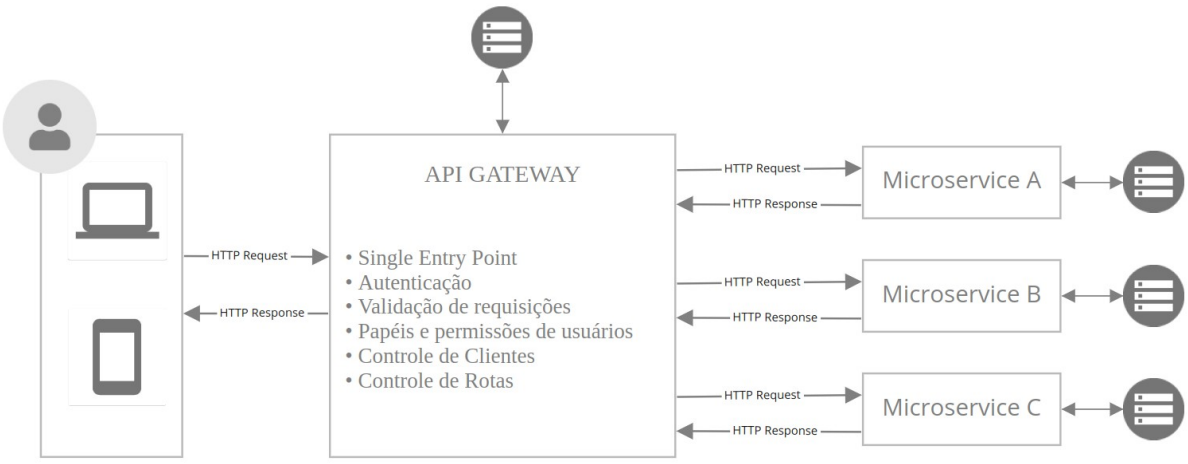
4 Atividades Desenvolvidas

Neste Capítulo serão apresentadas as atividades desenvolvidas, utilizando um projeto real da organização, feito em *Node.js*.

4.1 O projeto

O projeto consiste em um *servidor web* cujo propósito é disponibilizar *endpoints* aptos a receber *requisições HTTP*, efetuar determinados cálculos e devolver os resultados. Tais requisições se originam de uma sitema *web* utilizado pelos clientes da organização, que por sua vez são capturadas e redirecionadas para a aplicação por um *API Gateway* local, conforme a *Figura 10*.

Figura 10 – Diagrama da Arquitetura das Aplicações



Fonte: Autor

O arquivo *app.js*, mostrado no Código 1 é responsável por executar toda a aplicação e disponibilizar o acesso a ela através de uma porta pré-defenida.

Código 1 – Aplicação – *app.js*

```
1. require('dotenv').config();
2. const app = require('./src/server');
3. let port = process.env.PORT || 8080;
4. app.listen(port, () => console.log(`[INFO] Server
   Created on ${port}: ${process.env.NODE_ENV}`));
```

O lançamento do projeto se dará através da execução de um arquivo *Dockerfile* localizado na raiz do projeto. O *Dockerfile* é responsável por instanciar um *contêiner* que executará o servidor *web*. No Código *abaixo* o arquivo é detalhado.

Código 2 – *Dockerfile*

```
1. FROM node:14
2. WORKDIR /usr/src/app
3. COPY package*.json ./
4. RUN npm install
5. COPY . .
6. CMD [ "node", "app.js" ]
```

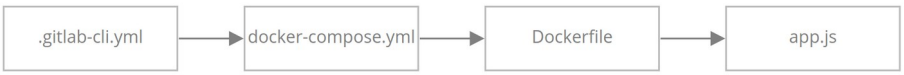
A execução do *Dockerfile* é feita pelo arquivo *docker-compose.yml*, também localizado na raiz do projeto, como se segue no código *abaixo*:

Código 3 – *Docker-compose.yml*

```
1. version: '3'
2. services:
3.   backend:
4.     image: 'node:14'
5.     build: .
6.     ports:
7.       - "8888:8108"
```

Por fim, na *Figura 11*, temos o seguinte fluxo de execução dos arquivos para a execução do projeto:

Figura 11 – Sequência de execução de arquivos



Fonte: Autor

4.2 O CI/CD

Como é descrito na subseção *2.2*, após a conclusão de uma nova *feature*, o desenvolvedor integrante do *Scrum Team*, poderá efetuar um *Merge Request* no Gitlab, da *feature branch* para o *branch develop*, a fim de cumprir com as atividades previstas na *Sprint* e gerar um novo *build* do projeto, no servidor de testes. Na *Figura 12* é possível observar um novo Merge Request sendo criado.

Figura 12 – GitLab *Merge Request*

Fonte: Autor

Sendo efetivado o pedido de *Merge Request*, o GitLab reconhece a existência do arquivo *.gitlab-ci.yml* contido na raiz do projeto e dispara automaticamente o *pipeline* de CI/CD, executando as instruções contidas no arquivo.

O *pipeline* é divido em três fases: *build*, *test* e *deploy*, conforme mostrado no código *abaixo*. Na linha *1* do código, a imagem *docker:latest* é criada como imagem padrão, utilizada por ambos os estágios. Na linha *4* é criado o serviço *Docker-in-Docker (dind)*, indicando que os *Runners* registrados usam executores que utilizarão *contêiners* de imagens *Docker* para executar os *jobs* do CI/CD. Em cada estágio é feita a referência ao seu *Runner*, conforme mencionado na subseção *3.7.2*, através da palavra reservada *tags*.

Código 4 – *.gitlab-ci.yml – Estágio Global*

```
1. image: docker:latest
2.
3. services:
4.   - docker:dind
5.
6. cache:
7.   paths:
8.         - node_modules/
9
10. stages:
11.  - build
12.  - test
13.   - deploy
```

4.2.1 *Build*

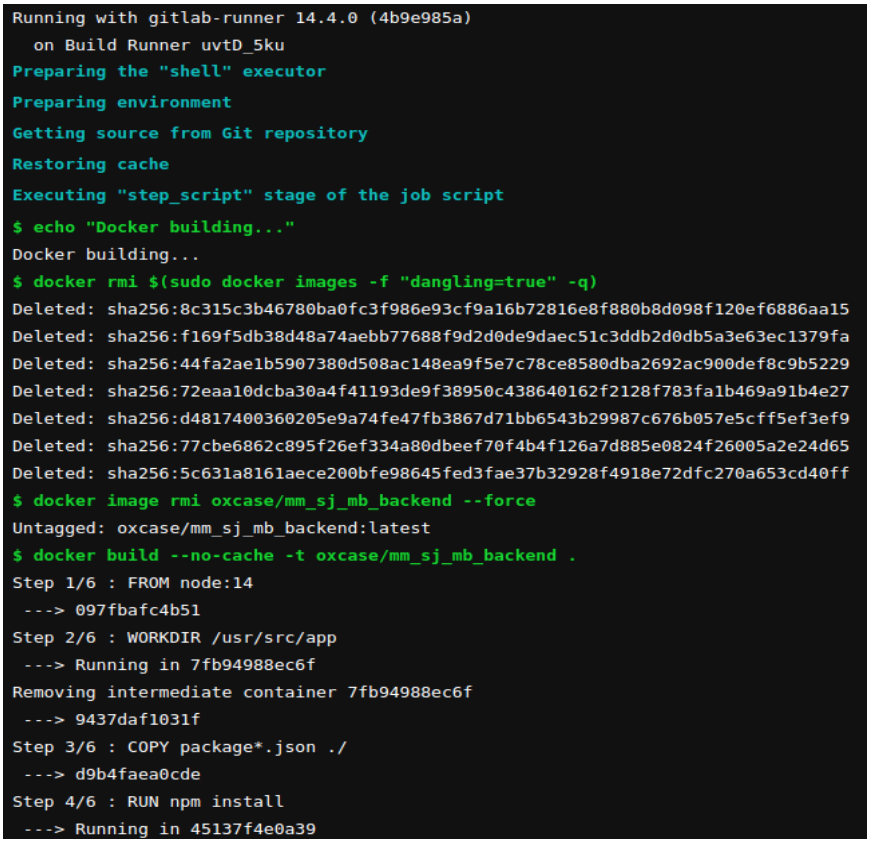
O estágio de *build* é iniciado na linha *14*, confome o código *abaixo*. Na linha *20* se iniciam as instruções para a compilação, com a palavra reservada *script*. Inicialmente as possíveis imagens e *contêiners* pré-existentes são apagados para finalmente, na linha *24*, ser executado o *build* de um novo *contêiner* a partir da imagem indicada na referência à variável de ambiente do GitLab *$IMAGE_NAME*.

Código 5 – *.gitlab-ci.yml – Estágio de Build*

```
14.  build:
15.        stage: build
16.
17.  tags:
18.        - test-runner
19.
20.  script:
21.        - echo "Docker building..."
22.        - docker rmi $(sudo docker images -f
           "dangling=true" -q)
23.        - docker image rmi $IMAGE_NAME --force
24.        - docker build --no-cache -t $IMAGE_NAME .
25.        - echo "Compile completed."
```

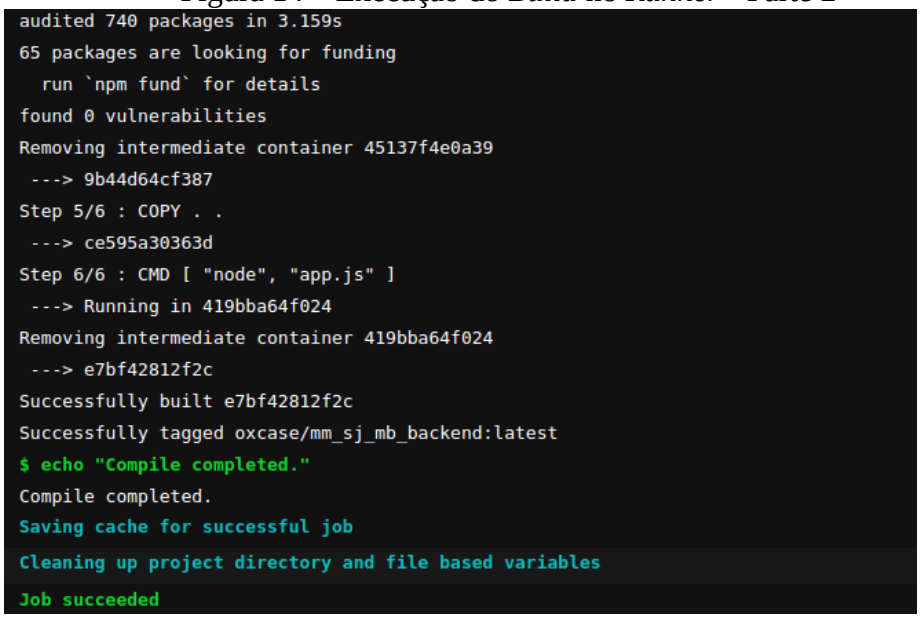
A seguir, nas figuras *13* e *14*, são exibidos os *Logs* da execução do estágio de *Build* em seu respectivo Runner.

Figura 13 – Execução do *Build* no *Runner* - Parte 1



Fonte: Autor

Figura 14 – Execução do *Build* no *Runner* - Parte 2



Fonte: Autor

4.2.2 Teste

O estágio de teste é iniciado na linha *26* do arquivo *.gitlab-ci.yml*, exibida no código *abaixo*.

Código 6 – *.gitlab-ci.yml – Estágio de Teste*

```
26.  test:
27.        stage: test
28.
29.        tags:
30.             - test-runner
31.
32.        script:
33.             - docker rm -f $CONTAINER_NAME
34.             - docker run -d --add-host 'localhost
                  localhost.localdomain localtest:127.0.0.1'
                  --name $CONTAINER_NAME $IMAGE_NAME
35.             - docker exec $CONTAINER_NAME npm run test
```

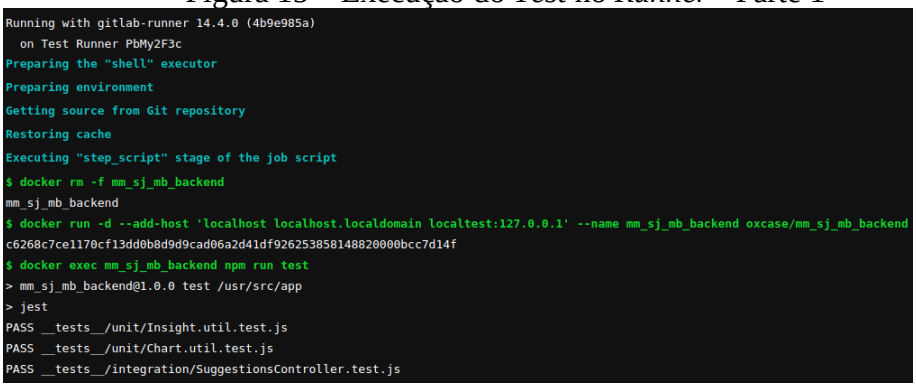
O *script* contido no código, assim como no estágio de *build*, certifica que possíveis *contêiners* pré-existentes sejam removidos para que então sejam executado os comandos

docker run para criar um novo *contêiner* a partir da imagem contida no projeto; e docker *exec* no *contêiner* criado, com o comando *npm run test*, que fará com que o *node execute*, dentro do arquivo *package.json* contido na raiz do projeto, o *script* de teste. Os testes são efetuados usando o *framework Jest*, que executa os testes de integração e unitários, conforme o código *abaixo*.

Código 7 – Trecho do arquivo *package.json*

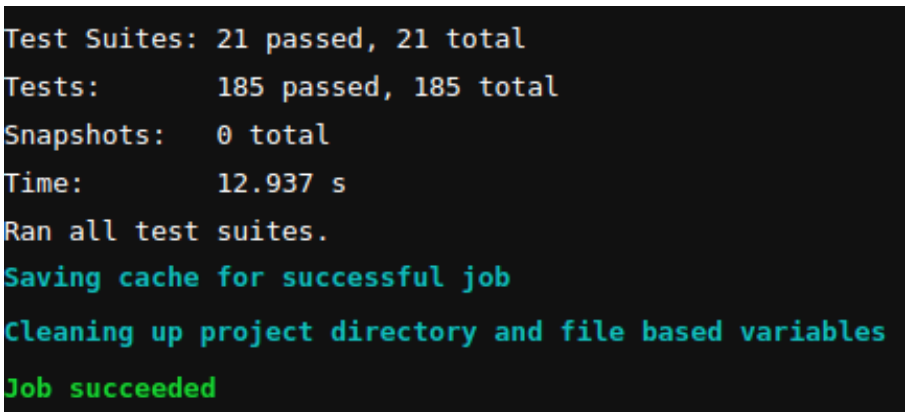
```
1. "scripts": {
2.     "start": "nodemon app.js",
3.     "test": "jest"
4.   }
```

Figura 15 – Execução do *Test* no *Runner* - Parte 1



Fonte: Autor

Figura 16 – Execução do *Test* no *Runner* - Parte 2



Fonte: Autor

4.2.3 Deploy

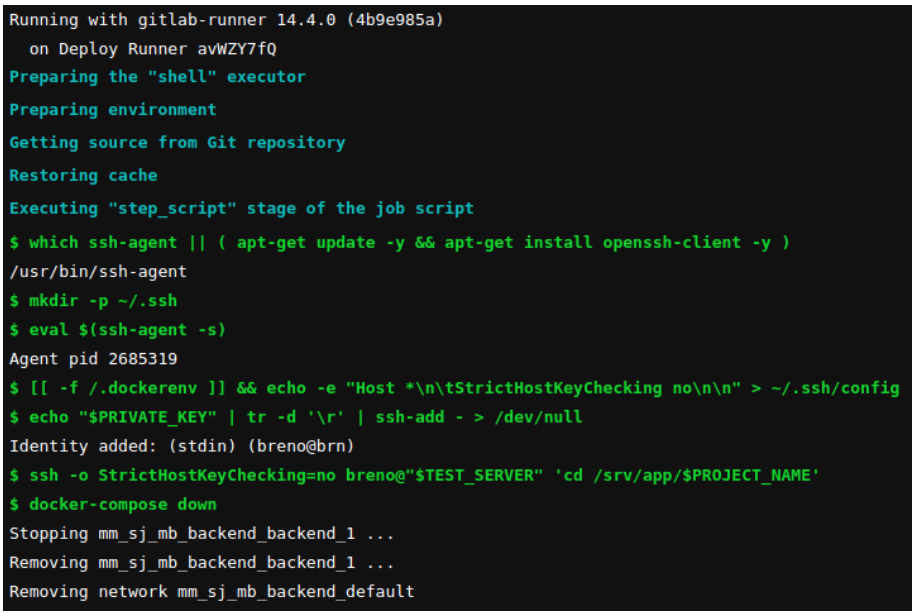
Sendo concluído com êxito todos os testes, conforme é mostrado no código *abaixo*, a última etapa do *pipeline* é disparada: o *deploy*. Na linha *42*, as instruções contidas a partir da palavra *before*_script garantem que todos os comandos necessários para a instalação e configuração do protoloco *SSH* sejam efetivados, para que posteriormente o *script* execute o *script* de *deploy* no servidor. Na linha *48*, é efetuado a conexão com o servidor, utilizando as variáveis de ambiente do GitLab para as referências corretas, e feita a nevagação até a pasta raiz do projeto. Nas linhas *51* e *52* são feitos os processos para criação de um novo *contêiner* com o código atualizado do projeto, utilizando o Docker Compose.

Código 8 – *.gitlab-ci.yml – Estágio de Deploy*

```
36.  deploy:
37.            stage: deploy
38.
39.       tags:
40.       - deploy-runner
41.
42.       before_script:
43.       - 'which ssh-agent || ( apt-get update -y &&
          apt-get install openssh-client -y )'
44.       - mkdir -p ~/.ssh
45.       - eval $(ssh-agent -s)
46.       - '[[ -f /.dockerenv ]] && echo -e "Host *\n\
          tStrictHostKeyChecking no\n\n" > ~/.ssh/config'
47.
48.       script:
49.       - echo "$PRIVATE_KEY" | tr -d '\r' | ssh-add →
          /dev/null
50.       - ssh -o StrictHostKeyChecking=no
          $DEPLOY_USER@"$TEST_SERVER"
          'cd/srv/app/$PROJECT_NAME'
51.       - docker-compose down
52.       - docker-compose up --build -d
```

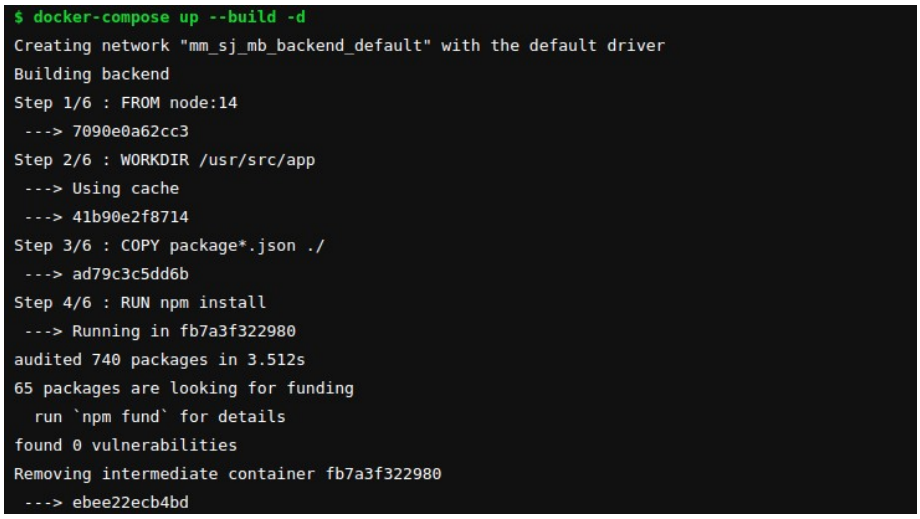
A seguir, nas figuras *17*, *18* e *19* são exibidos os *logs* da execução do estágio de *Deploy* em seu respectivo Runner.

Figura 17 – Execução do *Deploy* no Runner - Parte 1



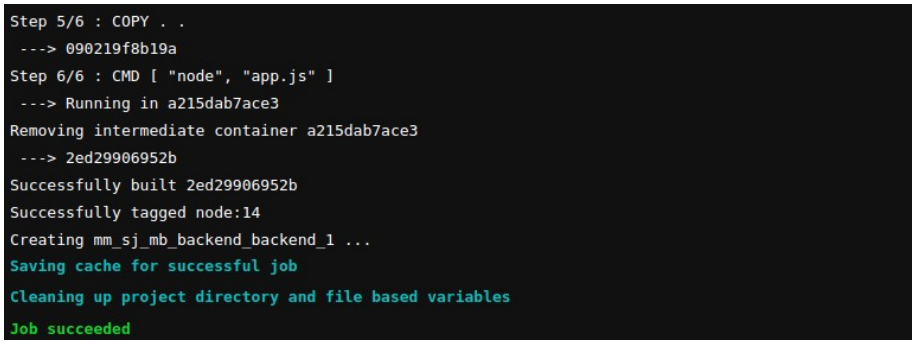
Fonte: Autor

Figura 18 – Execução do *Deploy* no *Runner* - Parte 2



Fonte: Autor

Figura 19 – Execução do *Deploy* no *Runner* - Parte 3

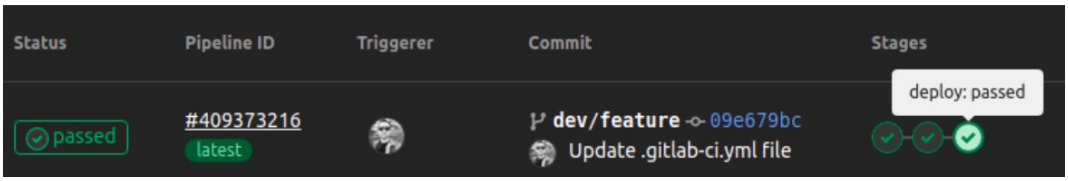


Fonte: Autor

Com isso toda uma nova instância do projeto é criada e as novas *features* desenvolvidas estarão disponíveis para consumo no servidor corrente.

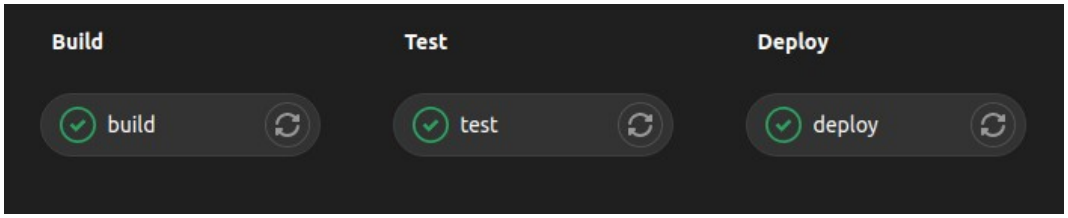
Por fim, utilizando a plataforma do GitLab, é possível certificar o estado de execução de cada estágio do pipeline executado, conforme é detalhado nas figuras *20* e *21*.

Figura 20 – Execução do *Pipeline*



Fonte: Autor

Figura 21 – Execução dos estágios do *Pipeline*



Fonte: Autor

5 Considerações Finais

Inicialmente foi proposto a adoção de técnicas para automatização do processo de *deployment* de *softwares* nos projetos existentes na organização, visto que atualmente são todos feitos de maneira manual, lenta e suscetíveis a erros humanos. Posteriormente, a partir de estudos realizados, concluiu-se que seriam necessárias adaptações em todo o escopo das operações adotadas no desenvolvimento dos produtos de *software* da empresa, considerando práticas culturais que envolvem as metodologias ágeis, técnicas de desenvolvimento orientadas a testes unitários e de integração, adaptações dos projetos para execução acoplados a *contêineres* e, por fim , o desenvolvimento de *pipelines* de CI/CD.

Para tanto, foram apresentados internamente na empresa treinamentos relacionados às tecnologias futuramente adotadas, a fim de auxiliar nos estudos e habituar os membros do *Scrum Team* com os novos processos. Por fim foram executadas as configurações e adaptações necessárias e adotadas as práticas de CI/CD em um projeto real através dos *pipelines* desenvolvidos, de modo a serem gerenéricos visando a sua adaptação nos demais projetos existentes na organização.

Considerando os objetivos propostos, concluo que foram atendidos, de modo que a organização estará apta a adotar todas as práticas necessárias para a implementação dos *pipelines* nos demais projetos, o que será de grande valor para a execução técnica do desenvolvimento de *softwares* assim como para o aperfeiçoamento da metodologia de desenvolvimento ágil, refletindo positivamente junto aos clientes e *stakeholders* da empresa.

Ressalta-se que melhorias poderão ser feitas para aperfeiçoar o gerenciamento dos *pipelines* para se tornarem mais escaláveis na adoção de uma maior quantidade de projetos. Tecnologias como *Kubernetes* e *Harbor Registry* podem ser adotadas e alocados servidores exclusivos para execução dos *Jobs* dos Runners e *Pods* do Kubernetes.

Referências

ASSOCIAÇÃO BRASILEIRA DAS EMPRESAS DE SOFTWARE - ABES. Mercado Brasileiro de Software: panorama e tendências. 2019.

SECRETARIA DE TECNOLOGIA DA INFORMAÇÃO E COMUNICAÇÃO. Roteiro de Métricas de Software do SISP: versão 2.3. Brasília: Ministério do Planejamento, Desenvolvimento e Gestão. Secretaria de Tecnologia da Informação e Comunicação, 2018.

UNIVERSIDADE FEDERAL DE LAVRAS. Biblioteca Universitária. Manual de normalização e estrutura de trabalhos acadêmicos: TCCs, monografias, dissertações e teses. 3. ed. rev., atual. e ampl. Lavras, 2020. Disponível em: http://repositorio.ufla.br/jspui/handle/1/11017. Acesso em: 19/11/2021.

SOMASUNDARAM, R. Git: Controle de versão para todos. Birmingham, Reino Unido: Packt Publishing, 2013.

BARRETT, DJ; SILVERMAN, R. SSH, The Secure Shell: The Definitive Guide. Sebastopol, Califórnia, Estados Unidos: O'Reilly, 2001.

SAITO, H.; CHLOE LEE, H.C.; WU, C.Y. DevOps com Kubernetes. Birmingham, Reino Unido: Packt Publishing, 2017.

Debois, P. (2008). Agile infrastructure and operations: how infra-gile are you? In Agile 2008 Conference (pp. 202–207). IEEE.

Wills, J. (2010). What Devops Means to Me. Retrieved June 24, 2019.

WAYNER, P. GitHub vs Bitbucket vs GitLab: Uma batalha épica pelo mindshare desenvolvedor. CIO. [S.I.]. 2018.

MEDEIROS, M. Extreme Programming - Conceitos e Práticas. Devmedia. [S.I.]. 2013.

AZERI, I. What Is CI/CD. 2017.

COSTA, E. Entrega Contínua de Software - Revista. net Magazine 100. 2017.

LOELIGER, J.; MCCULLOUGH, M. Version Control with Git: Powerful tools and techniques for collaborative software development. [S.l.]: "O'Reilly Media, Inc.", 2013.

SABBAGH, E. Scrum Gestão ágil para projetos de sucesso. 2017.

AUDY, J. Scrum 360: Um guia completo e prático de agilidade. 2015.

SCHWABER, K.; SUTHERLAND, J. O Guia do Scrum. O Guia Definitivo para o Scrum: As Regras do Jogo. 2020.

Glossário

API Gateway: Estrutura de gerenciamento de tráfego que fica entre os serviços do cliente e os servidores de backend.

Branch: Ramificação do Git onde os códigos ficam isolados das demais ramificações. Geralmente utilizado para isolar códigos em desenvolvimento de códigos estáveis.

Chapter Lead: Papel do framework Scrum responsável por gerir e garantir a boa execução dos processos e responsabilidades do Scrum Master e Product Owner.

Check-in: Processo de envio de novas features para repositórios git.

Code Review: Prática de revisão de um código recém desenvolvido por um desenvolvedor que não participou do processo de desenolvimento.

Endpoints: Interface entre aplicações e softwares disponível através da exposição de portas de comunicação.

Features: Funcionalidades de softwares desenvolvidas por um ou mais desenvolvedores.

Framework: Abstração de conjunto de códigos com fins relacionados agrupados em um pacote único, possibilitando ser utilizado por outros códigos.

Full Stack Developer: Cargo em que um desenvolvedor de softwares tem conhecimentos e responsabilidades que envolvem todos os processos e ferramentas para desenvolvimento das aplicações.

Git Workflow: Recomendação de boas práticas a serem adotadas por usuários da tecnologia Git.

Git: Sistema de controle e gerenciamento de versões, usualmente códigos de softwares.

GitLab: Plataforma de hospedagem e gerenciamento de códigos utilizando a tecnologia Git.

Harbor Registry: Sistema responsável por armazenar e disponibilizar Imagens Docker em servidores na nuvem.

Jest: Framework para desenvolvimento de testes automatizados em Node.js.

Kubernetes: Ferramenta responsável por orquestrar e gerenciar contêineres.

Merge: Mesclagem entre códigos de ramificações (*branches)* diferentes.

Node.js: Ambiente de execução da linguagem Javascript no lado do servidor (*server side).*

Pipelines de CI/CD: Série de etapas seguidas para a execução de processos de integração, testes e implantação de softwares de maneira automatizada.

Product Owner: Membro do Scrum responsável por definir e priorizar estórias de usuário para o Scrum Team.

Release: Entrega final de uma versão de um produto de software.

Requisições HTTP: Mensagens enviadas utilizando o procolto HTTP para troca ou obtenção de dados na Web.

Scrum Master: Líder da equipe do Scrum, responsável por proteger e orientar o Scrum Team no cumprimento das práticas do framework.

Scrum Team: Equipe de desenvolvimento composta pelo Scrum Master, Product Owner e pelo time de desenvolvedores.

Scrum: Framework de boas práticas para gerenciamento ágil de projetos.

Servidor de Backend: Local onde as aplicações internas de softwares são executadas.

Servidor web: Local onde são armazenados os códigos e arquivos que compõe um sistema

web, e receber requisições HTTP.

Sprints: Ciclo de tempo de trabalho no Scrum, onde é acrescentado algum valor ou incremento a um produto em desenvolvimento.

Testes de integração: Fase de teste de softwares onde módulos são testados em conjunto.

Testes funcionais: Fase de teste de softwares onde o comportamento e execução da aplicação são testadas a nível de usuário.

Testes unitários: Fase de testes de softwares onde módulos são testados individualmente.

Logs: Processo de registro de eventos relacionados a execução de códigos.

Pods: Menor instância de objeto implantável no Kubernetes.