AUTOMAÇÃO DE DEPLOYS COM KUBERNETES E ARGO CD: UM ESTUDO SOBRE A IMPLEMENTAÇÃO DE PRÁTICAS GITOPS

Autor: Marcio Junior Sbicigo (FUNDAÇÃO HERMÍNO OMETTO - FHO)
marciosbicigo@alunos.fho.edu.br
Orientador: Prof. Diego Fiori de Carvalho (FUNDAÇÃO HERMÍNO OMETTO - FHO)
dfiori@fho.edu.br

Resumo

Nos últimos anos, a automação dos processos de Integração Contínua (CI) e Entrega Contínua (CD) tornou-se essencial para o desenvolvimento ágil de software, especialmente em arquiteturas orquestradas por contêineres. Este trabalho visa implementar um fluxo automatizado de CI/CD utilizando o Argo CD, uma ferramenta de deploy contínuo baseada em *GitOps*, em conjunto com *Kubernetes*, Docker e GitHub Actions. O projeto foi desenvolvido em uma infraestrutura local que simula um cenário real de deploy de uma aplicação web, abordando desde as etapas de testes e criação das imagens Docker até a automação do deploy em um ambiente *Kubernetes*. A metodologia inclui a integração das ferramentas para garantir automação completa, com foco em reduzir o tempo de implantação, aumentar a confiabilidade, garantir consistência e segurança além da facilidade para recuperar o estado anterior em caso de problemas (*rollback*). Os resultados mostram que o *Argo CD* e *GitOps* facilitam a gestão de deploys, assegurando maior controle e rastreabilidade nas modificações de código e ambientes. O estudo conclui que essas práticas são altamente benéficas para equipes de desenvolvimento e operações, permitindo automação eficaz e segura em ambientes corporativos ou acadêmicos.

Palavras-Chaves: Automação, *Kubernetes*, *GitOps*.

1. Introdução

Nos últimos anos, a demanda por ferramentas de orquestração de contêineres tem crescido significativamente, com o *Kubernetes* se estabelecendo como uma das principais plataformas para automação de implantação, escalonamento e gerenciamento de aplicações em contêineres. A evolução das tecnologias de conteinerização, junto à crescente complexidade das arquiteturas modernas, exige soluções rápidas, escaláveis e eficientes para gestão e automação dos processos de deploy, ou seja, o processo de disponibilização de uma aplicação para uso em ambientes produtivos (Burns; Beda; Hightower, 2019).

Com isso, a implementação de estratégias eficazes de CI/CD tornou-se fundamental para garantir a entrega rápida e segura de software, especialmente em ambientes *Kubernetes*. No entanto, apesar das ferramentas disponíveis, ainda existem desafios significativos na automação completa do processo de *deploy*, como a gestão de configurações, a sincronização entre repositórios *Git* e clusteres *Kubernetes*, e a capacidade de realizar *deploys* consistentes e reversíveis (Arundel; Domingus, 2019).

Este estudo tem como objetivo implementar um ambiente de CI/CD utilizando GitHub Actions para Integração Contínua (CI) e *Argo CD* para Entrega Contínua (CD), em conjunto com Docker e *Kubernetes*, simulando um cenário de deploy em ambiente local. O trabalho explora a automação desde as etapas de testes e construção de imagens Docker até a implantação final através de práticas *GitOps*, proporcionando maior eficiência, segurança, consistência e rastreabilidade nos processos de desenvolvimento e implantação de software.

Ao longo deste trabalho, abordaremos os conceitos teóricos das tecnologias envolvidas e detalharemos a implementação prática do ambiente automatizado de CI/CD. Em seguida, discutiremos os resultados obtidos, analisando-os em relação às expectativas estabelecidas e sintetizando as principais conclusões e sugestões para pesquisas futuras.

2. Fundamentação teórica

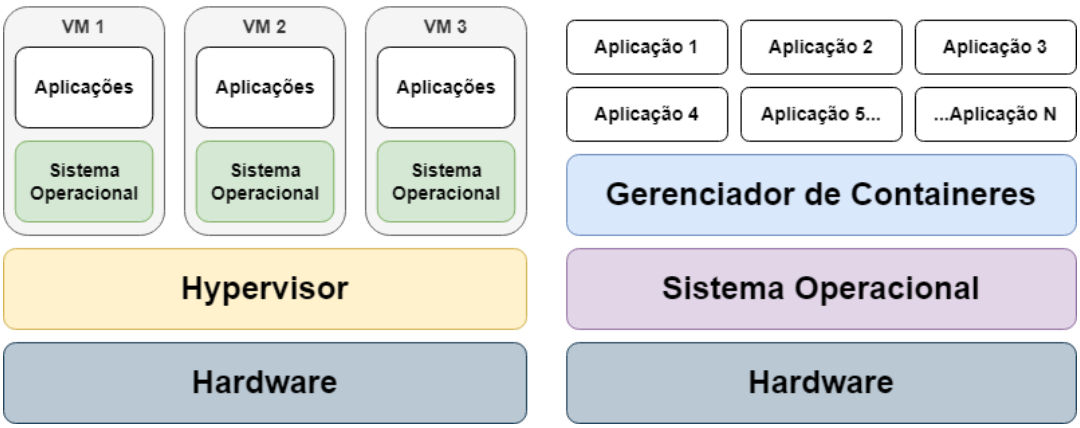
Nesta seção, discutiremos os conceitos e tecnologias-chave para implementar um ambiente de deploy automatizado, como Docker, Kubernetes, CI/CD, GitHub Actions e Argo CD, que integram a automação desde a construção até a implantação das aplicações. Exploraremos os fundamentos dessas tecnologias e sua relevância atual.

2.1 Virtualização baseada em contêineres

A virtualização baseada em contêineres é uma abordagem que ganhou destaque na área de infraestrutura de TI devido à sua capacidade de fornecer ambientes isolados e leves para a execução de aplicações. Ao contrário da virtualização tradicional, onde um *Hypervisor* é responsável por executar múltiplas máquinas virtuais (VMs) de modo que cada uma precise ter seu próprio sistema operacional, a virtualização de contêineres, por sua vez, compartilha o mesmo kernel do sistema operacional primário, o que resulta em menor sobrecarga e maior eficiência. Além disso, essa técnica facilita o desenvolvimento, a escalabilidade e a

portabilidade das aplicações, sendo amplamente adotada por empresas ao redor do mundo (Matthias; Kane, 2015). As diferenças entre as arquiteturas podem ser vistas na figura 1.

Figura 1 – Arquitetura tradicional x Arquitetura baseada em contêineres



Fonte: Próprio Autor.

2.2 Docker

O Docker é uma plataforma que simplifica a criação, empacotamento e execução de aplicações em contêineres, oferecendo um ambiente isolado e consistente. Ele facilita o desenvolvimento e a entrega, permitindo maior portabilidade em diferentes ambientes, como servidores locais e nuvem (Docker, 2024).

2.2.1 Arquitetura Docker

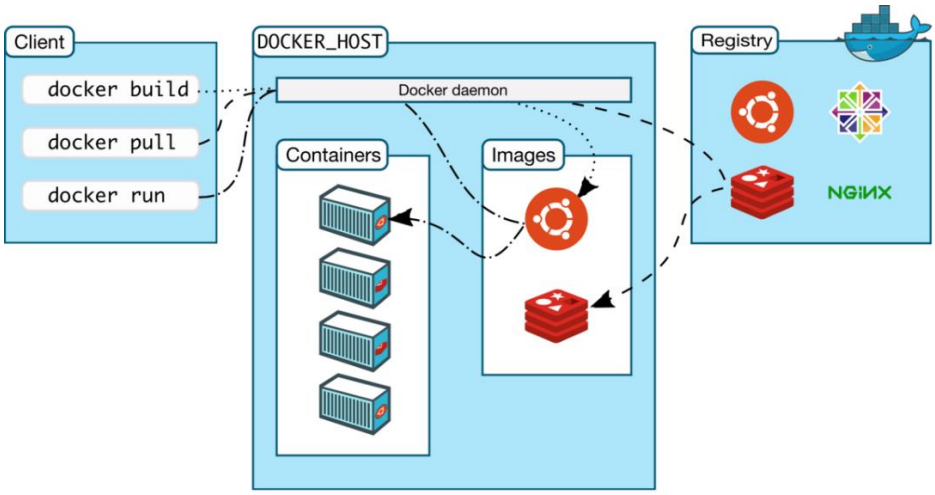
A arquitetura do Docker segue um modelo cliente-servidor simples, onde o *Docker Client* (CLI) e o *Docker Daemon* estão integrados em um único binário. O CLI permite aos usuários interagir com o *Docker Daemon* via comandos, possibilitando a criação, execução e exclusão de contêineres, gerenciamento de imagens, volumes e redes, além de outras operações administrativas.

O *Docker Daemon* é o responsável por executar as ações solicitadas pelo *Docker Client*, ou seja, ele gerencia a criação, execução, exclusão e monitoramento dos contêineres. Além disso, ele utiliza tecnologias do kernel do sistema operacional, como *namespaces* e cgroups, para garantir o isolamento dos contêineres e controlar a alocação de recursos, como CPU e memória. Isso permite que os contêineres funcionem de forma independente e eficiente no mesmo sistema.

Há também o *Docker Registry*, um repositório para armazenar e distribuir imagens Docker, facilitando o versionamento e o compartilhamento em equipes de desenvolvimento ou ambientes de produção (Matthias; Kane, 2015).

Neste trabalho, para armazenar as imagens das aplicações de demonstração, foi utilizado o *Docker Hub*, o *Registry* oficial do Docker.

Figura 2 – Arquitetura Docker



Fonte: Cherry Servers (2022)

Conforme a figura 2, a interação entre os componentes da arquitetura começa com um comando no CLI, como a execução de um contêiner. O CLI envia a solicitação ao Daemon, que busca a imagem no *Docker Registry* se ela não estiver disponível localmente. Após obter a imagem, o Daemon cria e executa o contêiner, gerenciando recursos e garantindo isolamento.

2.2.2 Objetos Docker

Os objetos Docker (*Docker Objects*) são componentes que complementam a arquitetura e o funcionamento do Docker. Eles são utilizados para construir, implantar e gerenciar as aplicações dentro dos contêineres. Neste trabalho, os principais objetos utilizados foram Imagens, *Dockerfile* e Contêineres.

As imagens são pacotes binários contendo tudo o que é necessário para executar uma aplicação, incluindo sistema operacional de base, bibliotecas e dependências. Elas são compostas por camadas imutáveis de somente leitura e servem de base para a criação dos contêineres (Burns; Beda; Hightower, 2019). As imagens podem ser geradas a partir de um *Dockerfile*, que é um arquivo de texto contendo instruções para a construção da imagem. Cada linha do *Dockerfile*

cria uma nova camada dentro da imagem e descreve um dos passos que o Docker seguirá para construir a imagem completa, como por exemplo a instalação de bibliotecas, configuração de variáveis de ambiente, execução de scripts, exposição de portas e etc. (Matthias; Kane, 2015).

Por fim, os contêineres são instâncias em execução de uma imagem. Eles empacotam a aplicação com todas as suas dependências, garantindo que ela funcione de forma consistente e isolada em qualquer ambiente, seja de desenvolvimento ou de produção (Burns; Tracey, 2018).

2.3 Kubernetes

O *Kubernetes* é uma plataforma *open-source* de orquestração de contêineres que se destaca pela automação da implantação, escalonamento e gerenciamento de aplicações conteinerizadas. Através de suas funcionalidades, o *Kubernetes* abstrai a complexidade de trabalhar diretamente com vários contêineres, além de permitir gerenciar a infraestrutura de forma escalável e automatizada, garantindo economia de recursos, alta disponibilidade e resiliência das aplicações (Burns; Beda; Hightower, 2019). Neste trabalho, o *Kubernetes* serve como base para a implementação do Argo CD, ferramenta de Entrega Contínua que automatizará a implantação das aplicações dentro do cluster.

Existem diferentes maneiras de configurar um cluster *Kubernetes*, tanto na nuvem quanto localmente. Para ambientes com pouco poder de processamento, ferramentas leves como Minikube ou Kind são mais recomendadas. Neste trabalho, foi escolhido o Kind, que permite rodar clusteres *Kubernetes* dentro de contêineres Docker. O Kind facilita a criação de clusteres em máquinas locais, sendo leve, fácil de configurar e compatível com fluxos de trabalho baseados em *Kubernetes*, ideal para ambientes de desenvolvimento e testes (Kind, 2024).

2.3.1 Configuração Declarativa

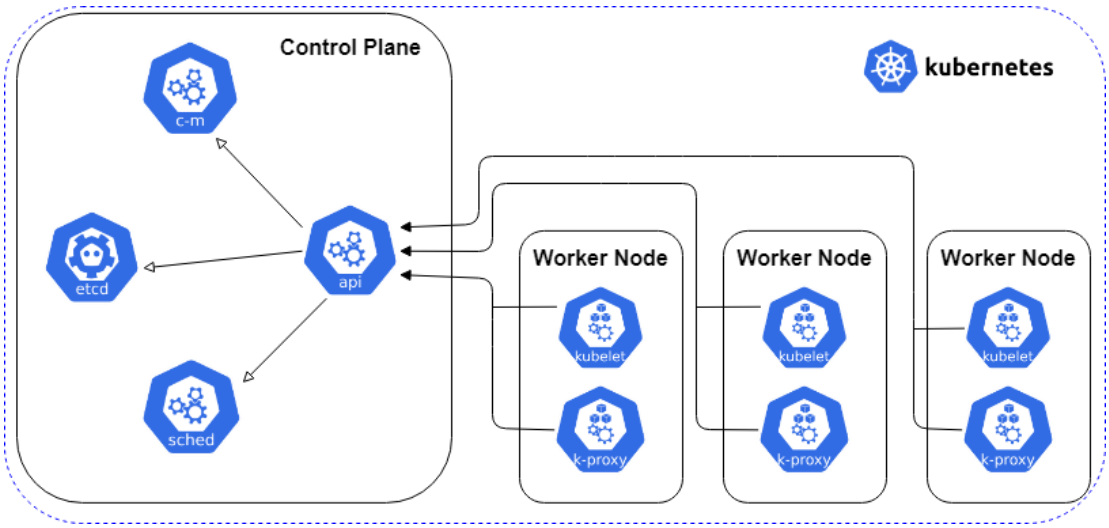
A configuração declarativa é um conceito central no *Kubernetes*, permitindo que os usuários especifiquem um estado desejado, como por exemplo: “Quero cinco réplicas da minha API” ou então “Quando minha API atingir 90% de uso de CPU, suba uma nova instância”, e assim por diante. O *Kubernetes*, então, garante que essas condições declaradas se mantenha, mas para isso, as declarações devem ser feitas em formatos como YAML ou JSON, já que ele não entende linguagem natural. Essa abordagem difere da configuração imperativa, em que os usuários realizam ações diretas, como criar cada réplica manualmente. Embora a configuração

imperativa seja mais simples, a configuração declarativa permite que o sistema tome decisões autônomas e se autocorrija. Assim, o *Kubernetes* pode resolver problemas sem intervenção do usuário, evitando interrupções indesejadas (Burns; Tracey, 2018).

2.3.2 Arquitetura Kubernetes

Um cluster *Kubernetes*, exemplificado na figura 3, é baseado em um modelo de múltiplos nós, onde o Control Plane atua como o nó principal (*Master Node*), responsável pela coordenação e gerenciamento do cluster, e os *Worker Nodes* são os nós de trabalho, encarregados de executar as aplicações e cargas de trabalho. O número de nós pode variar conforme a necessidade do ambiente, permitindo a escalabilidade horizontal conforme o crescimento das demandas (Yuen *et al.*, 2021).

Figura 3 – Arquitetura *Kubernetes*



Fonte: Próprio Autor

No *Control Plane* encontramos componentes como o *kube-apiserver*, que oferece uma API REST para gerenciar o estado do cluster; o *kube-controller-manager*, que ajusta o estado atual ao desejado; e o *kube-scheduler*, que distribui as cargas de trabalho entre os nós disponíveis, assegurando uma utilização eficiente dos recursos. Além disso, o *etcd* funciona como o banco de dados que armazena as informações de configuração do cluster (Yuen *et al.*, 2021).

Já os *Worker Nodes* possuem como componentes principais o *kubelet*, que gerencia os contêineres em execução dentro do nó, e o *kube-proxy*, que facilita a comunicação de rede entre os serviços no cluster, garantindo uma comunicação fluida entre as aplicações (Yuen *et al.*, 2021).

2.3.3 Kubectl

O *kubectl* é uma ferramenta utilizada para interagir com a API do *Kubernetes* (*kube-apiserver*) através de comandos no terminal. Com o *kubectl* é possível gerenciar a maioria dos objetos do *Kubernetes*, tanto de maneira imperativa, quanto de maneira declarativa, aplicando arquivos de configuração, também conhecidos como manifestos YAML (Arundel; Domingus, 2019).

2.3.4 Objetos do Kubernetes

Os objetos do *Kubernetes* são abstrações que representam entidades no cluster, facilitando o gerenciamento das aplicações conteinerizadas. Neste trabalho, foram utilizados cinco objetos principais: *Namespaces*, *Pods*, *Services*, *Persistent Volume Claims* e *Deployments*.

Namespaces dividem o cluster em partes isoladas, permitindo organizar diferentes projetos em ambientes separados uns dos outros, garantindo que recursos de um *namespace* não sejam acessíveis por outro (Arundel; Domingus, 2019).

Pods são a menor unidade implantável, consistindo em um ou mais contêineres que se comunicam entre si. Embora seja possível agrupar múltiplos contêineres em um *Pod*, recomenda-se que cada *Pod* contenha apenas um contêiner para evitar impactos negativos no escalonamento das aplicações, devido a diferentes demandas de recursos (Burns; Beda; Hightower, 2019).

Um *Service* é um recurso que expõe uma aplicação rodando em um conjunto de *Pods*, garantindo acesso consistente, mesmo após reinicializações. O *Service* do tipo NodePort, utilizado neste trabalho, expõe aplicações externamente através do IP do nó e de portas específicas (Kubernetes, 2024). Apesar de simples, o NodePort tem limitações em escalabilidade e balanceamento de carga, sendo o *LoadBalancer* uma alternativa mais robusta, pois distribui o tráfego automaticamente entre os nós (Burns; Beda; Hightower, 2019).

Persistent Volumes representam o armazenamento físico, e garantem a persistência de dados após a exclusão de *Pods*. Já o *Persistent Volume Claim* é a solicitação de acesso a esse armazenamento feita pelos *Pods*, facilitando a portabilidade e replicação (Burns; Tracey, 2018).

Deployments gerenciam atualizações de software de forma eficiente, eliminando o tempo de inatividade e minimizando erros. Eles permitem transições controladas entre versões, utilizando verificações de integridade e intervalos configuráveis entre atualizações dos *Pods*. O

Deployment controller automatiza o processo, tornando possível implementar novas versões de maneira ágil e flexível, facilitando a entrega de software (Burns; Beda; Hightower, 2019).

2.4 Controle de versão e armazenamento de código

Para viabilizar a implementação deste trabalho, foi necessário o uso de ferramentas que armazenassem o código-fonte das aplicações, bem como os arquivos de configuração do estado desejado do cluster *Kubernetes*, permitindo o rastreamento de alterações e facilitando o *rollback* quando necessário. Para isso, foi utilizado o Git em conjunto com o GitHub.

O Git é um sistema de controle de versão amplamente utilizado, que permite o rastreamento inteligente de modificações em projetos. Ele funciona criando um repositório acessível por toda a equipe de desenvolvimento, e organiza as alterações em *branches*. As *branches* são linhas separadas de desenvolvimento, possibilitando a criação de funcionalidades ou correções sem afetar a versão principal do projeto. As alterações são registradas em *commits*, facilitando a recuperação em caso de problemas. Quando as modificações em uma *branch* estão prontas, elas são integradas à *branch* principal via *pull request*, que inclui revisões e aprovações pela equipe.

O GitHub, por sua vez, é uma plataforma web de hospedagem de repositórios Git, permitindo a colaboração remota entre equipes. Embora o desenvolvimento seja feito localmente na maioria das vezes, o GitHub facilita o envio do código e oferece ferramentas como criação de repositórios e edição de arquivos diretamente pela interface web (GitHub, 2024).

2.5 Integração Contínua e Entrega Contínua (CI/CD)

A Integração Contínua (CI) refere-se ao processo de mesclar frequentemente as alterações no código em um repositório Git centralizado, de forma que o código seja verificado por testes automatizados, visando identificar e corrigir problemas rapidamente. Já a Entrega Contínua (CD) é o processo que automatiza a entrega de novas versões de software, minimizando a intervenção manual e aumentando a eficiência do processo de lançamento (Yuen *et al.*, 2021).

No contexto deste trabalho, o GitHub Actions foi utilizado como plataforma de CI, enquanto o *Argo CD* foi utilizado para orquestrar a Entrega Contínua. Isso permitiu a automatização de todo o ciclo de vida das aplicações, desde a construção das imagens Docker até a implantação automática das aplicações no cluster *Kubernetes*.

2.6 GitHub Actions

O GitHub Actions é uma plataforma de CI/CD fornecida pelo *GitHub*, cuja principal proposta é automatizar fluxos de trabalho, também chamados de *workflows*. Esses *workflows* são compostos por etapas, conhecidas como *jobs*, que dividem o processo em tarefas executáveis. Além disso, um *workflow* pode ser disparado automaticamente a partir de *triggers*, que são determinados eventos que ocorrem em um repositório, podendo ser *commits*, *pull requests*, agendamentos ou de forma manual (GitHub, 2024).

2.7 GitOps

GitOps é uma abordagem que utiliza o repositório Git como a única fonte de verdade para a gestão e entrega automatizada da infraestrutura de um sistema. Nessa metodologia, o estado desejado da infraestrutura é versionado, imutável e descrito de forma declarativa. Nesse contexto, a Infraestrutura como Código (IaC) se torna fundamental, permitindo que a infraestrutura seja gerida por meio de arquivos de configuração, como YAML, especialmente no *Kubernetes*. Essa prática permite que as mesmas ferramentas usadas no desenvolvimento de software sejam usadas no gerenciamento da infraestrutura, promovendo maior consistência no processo. Além disso, facilita a replicação de configurações entre diferentes ambientes, reduzindo erros decorrentes de processos manuais e garantindo confiabilidade e previsibilidade nas implantações (Vinto; Bueno, 2023).

2.8 Argo CD

O *Argo CD* é uma ferramenta *open source* do *Argo Project*, projetada para implementar os princípios de *GitOps* e automatizar a Entrega Contínua em clusteres *Kubernetes*. Ele monitora continuamente um repositório Git, garantindo que o estado desejado, definido nos arquivos de configuração, corresponda ao estado real do cluster. Quando há discrepâncias, o *Argo CD* aplica modificações automaticamente e sinaliza problemas na sincronização. Os status de sincronização ajudam a identificar essas discrepâncias: os estados possíveis são "*Synced*" (Sincronizado) e "*OutOfSync*" (Fora de Sincronia). O estado "*OutOfSync*" indica que pelo menos um dos recursos está diferente do esperado (Yuen *et al.*, 2021).

2.8.1 Arquitetura do Argo CD

A arquitetura do *Argo CD* é composta por diversos componentes, destacando-se três principais: *argocd-repo-server*, *argocd-application-controller* e *argocd-server*.

O *argocd-repo-server* é responsável por gerar manifestos *Kubernetes* a partir do repositório Git. Para otimizar desempenho, ele armazena o conteúdo do repositório em cache local e utiliza comandos Git, para baixar apenas as alterações recentes. Em seguida, o *argocd-application-controller* compara o estado atual do cluster *Kubernetes* com os manifestos gerados, corrigindo divergências para garantir que o estado real corresponda ao esperado. Este controlador monitora e atualiza o estado do cluster em tempo real, permitindo uma sincronização eficiente. Por fim, o *argocd-server* apresenta os resultados da sincronização aos usuários, sendo acessível tanto por uma API REST quanto por uma interface web (Yuen *et al.*, 2021).

2.8.2 Application e ApplicationSet

Um *Application* é uma divisão lógica do *Argo CD* que reúne todos os recursos *Kubernetes* necessários para executar uma aplicação específica, incluindo informações sobre o repositório Git monitorado, subdiretórios dos recursos, e detalhes do cluster *Kubernetes* e *Namespace* de destino (Yuen *et al.*, 2021). Já o *ApplicationSet* expande essa funcionalidade ao permitir a criação e o gerenciamento de múltiplas instâncias de *Applications* a partir de um único template, simplificando implantações e garantindo consistência nas configurações, além de facilitar a escalabilidade (Vinto; Bueno, 2023).

3. Materiais e métodos

A implementação deste trabalho seguiu uma metodologia experimental e visou criar um ambiente de desenvolvimento e implantação automatizado com tecnologias de contêineres e orquestração. Focamos no processo de CI/CD, utilizando GitHub Actions para Integração Contínua (CI) e *Argo CD* para Entrega Contínua (CD), aplicando práticas de *GitOps* no gerenciamento das implantações.

Para começar, configuramos um servidor Ubuntu Server 22.04 em rede local, onde instalamos Docker, Kind e *kubectl* para criar um cluster *Kubernetes* e implantar o *Argo CD* dentro do cluster. O processo de CI foi implementado no repositório que armazena o código da aplicação em desenvolvimento e inclui o workflow do GitHub Actions para testes, criação de imagens Docker e envio para o *Docker Hub*.

Para o processo de CD foi criado outro repositório, que armazena os manifestos YAML que representam o estado desejado da aplicação no cluster. Ele reflete um repositório de deploy, que em um cenário real seria utilizado pelo time de infraestrutura, onde os manifestos podem ser ajustados e monitorados pelo *Argo CD* para garantir a implantação consistente no *Kubernetes*. Configuramos também um *ApplicationSet* no *Argo CD* para monitorar continuamente este repositório de CD e aplicar mudanças automaticamente.

Espera-se que este projeto resulte em um ambiente robusto e escalável, permitindo integração e entrega contínuas eficientes, além de facilitar a colaboração entre equipes de desenvolvimento e operações. Com a adoção dessa metodologia, projeta-se uma redução significativa nos tempos de ciclo de desenvolvimento e um aumento na segurança e confiabilidade das implantações.

4. Implementação

Nesta seção, serão detalhados os passos seguidos para implementar o ambiente de automação de *deploys* bem como uma contextualização de como cada ferramenta se aplica ao projeto como um todo.

Para execução dos comandos no terminal Linux, é necessário ter privilégios administrativos.

4.1 Instalação do Docker

Como visto na seção 2.1, a virtualização através de contêineres impacta positivamente de várias formas no ciclo de desenvolvimento e implantação de uma aplicação. E no contexto deste trabalho, o Docker facilitou não só a criação das Imagens das aplicações, mas também do *cluster Kubernetes* usado para implantação dessas imagens.

Os passos da instalação listados a seguir foram baseados na documentação do Docker[1]:

- Configuração das dependências necessárias, como chaves GPG e o repositório oficial do Docker para a versão 22.04 do *Ubuntu*;
- Execução do comando de instalação: `sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin`;

[1] Documentação do processo de instalação do Docker. Disponível em: https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository. Acesso em 4. nov. 2024.

- Adicionar o usuário ao grupo Docker: `sudo usermod -aG docker $USER`.

4.2 Instalação do Kind

O Kind foi a ferramenta utilizada para a criação do cluster *Kubernetes*. Essa abordagem possibilitou a simulação de um ambiente de produção através de contêineres Docker, sem a necessidade de uma infraestrutura complexa.

Os passos da instalação listados a seguir foram baseados na documentação do Kind[2]:

- Para download do binário do Kind: `[ $(uname -m) = x86_64 ] && curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.24.0/kind-linux-amd64`;
- Para dar permissão de execução: `chmod +x kind`;
- Movendo o binário para o diretório bin: `sudo mv kind /usr/local/bin/`.

4.3 Instalação do kubectl

Conforme abordado na seção 2.3.3, o *kubectl* nos permite se comunicar diretamente com a API do *Kubernetes*, tornando possível o gerenciamento dos recursos dentro do cluster.

Os passos da instalação listados a seguir foram baseados na documentação do *Kubernetes*[3]:

- Download do binário: `curl -LO https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl`;
- Para instalação o kubectl: `sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl`;

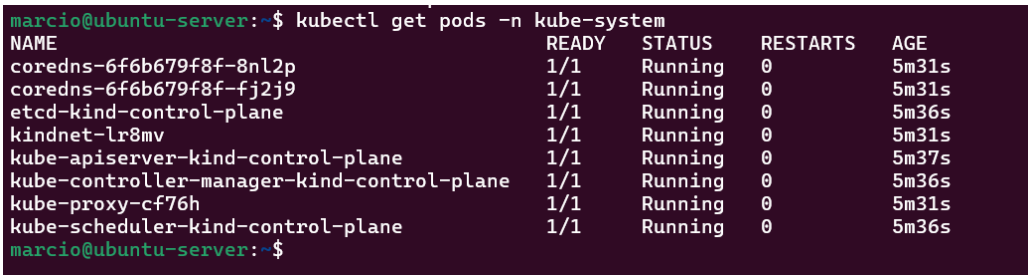
4.4 Configuração do cluster Kubernetes

Para criar um cluster *Kubernetes* usando o *Kind*, bastou executar o comando `kind create cluster`. O *Kind* baixa os componentes principais do *Kubernetes* como contêineres Docker,

[2] Documentação do processo de instalação do Kind. Disponível em: https://kind.sigs.k8s.io/docs/user/quick-start/#installation. Acesso em 31. out. 2024.
[3] Documentação do processo de instalação do kubectl. Disponível em: https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/. Acesso em 4. nov. 2024.

iniciando um nó com esses componentes em execução. Para verificar o status dos *Pods*, como mostrado na figura 4, use o comando `kubectl get pods -n kube-system`. Neste trabalho, foi utilizado um único nó devido aos recursos limitados do servidor.

Figura 4 – Componentes do cluster *Kubernetes* operando



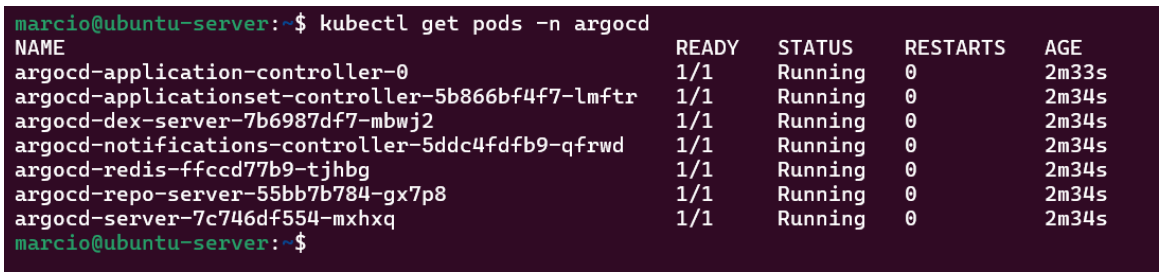
Fonte: Próprio Autor

4.5 Instalação do Argo CD

Para o funcionamento do Argo CD, primeiramente é necessário criar um *Namespace* exclusivo para ele dentro do cluster *Kubernetes*. Isso pode ser feito com o comando `kubectl create namespace argocd`.

Em seguida, a instalação pode ser feita com o comando `kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml`. Após a instalação, o status dos *Pods* pôde ser verificado com o comando `kubectl get pods -n argocd`, conforme mostra a figura 5.

Figura 5 – Componentes do *Argo CD* operando



Fonte: Próprio Autor

Para acessar a interface web do *Argo CD* e visualizar as aplicações implantadas, é preciso primeiramente fazer um direcionamento (*port forward)* através do comando `nohup kubectl port-forward svc/argocd-server -n argocd 9090:80 &`. Dessa forma o *Argo CD* ficará disponível no endereço http://localhost:9090 (ou http://<ip-do-servidor>:9090).

Para autenticar, utilize o usuário *admin* e a senha retornada pelo comando `kubectl get secret argocd-initial-admin-secret -n argocd -o jsonpath="{.data.password}" | base64 -d` (Vinto; Bueno, 2023).

4.6 Automação do processo de CI

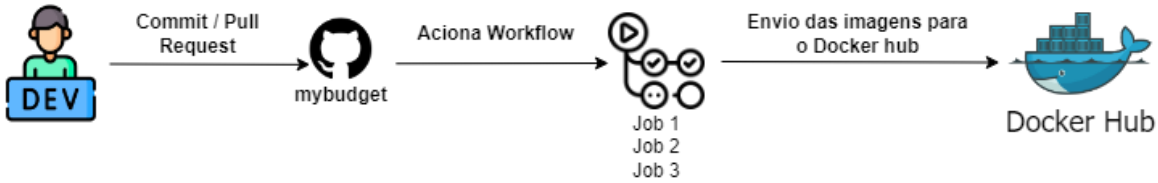
A automação do processo de CI, exemplificada na figura 6, foi configurada no repositório "*mybudget*"[4], que armazena o código fonte de duas aplicações: *frontend* e *backend-api*. Além disso, no diretório ".github\workflows" existe um arquivo chamado "*ci-workflow.yaml*"[5]. Esse *workflow* é acionado sempre que houver commits ou pull requests direcionados à *branch* principal do repositório (*main*). Para demonstrar o processo de CI automatizado, ele é dividido em três *jobs* principais que são interdependentes:

- *Job* 1: *test-backend-api*;
- *Job* 2 (Depende de *test-backend-api*): *test-frontend*;
- *Job* 3 (Depende do sucesso dos anteriores): *build-push-docker-image.*

Os *jobs test-backend-api* e *test-frontend* simulam a execução de testes para as aplicações, validando parâmetros essenciais de funcionamento e comunicação com os bancos de dados. As etapas de teste podem variar de projeto para projeto, dependendo das necessidades específicas. Em todas as situações, a inclusão de testes automatizados é altamente recomendada no ciclo de desenvolvimento de software, pois, à medida que a complexidade do projeto aumenta, os testes manuais tendem a tornar-se inviáveis (Okken, 2017).

Na etapa *build-push-docker-image*, o *workflow* lê o *Dockerfile* localizado no diretório raíz de cada aplicação, monta as imagens Docker e as publica no *Docker Hub*[6].

Figura 6 – Fluxo do processo automatizado de Integração Contínua via GitHub Actions



Fonte: Próprio Autor

[4] Repositório Git destinado ao processo de CI. Disponível em: https://github.com/mjsbicigo/mybudget
[5] Arquivo de Workflow usado para automatizar o processo de CI. Disponível em: https://raw.githubusercontent.com/mjsbicigo/mybudget/main/.github/workflows/ci-workflow.yaml.
[6] Repositórios *Docker Hub* para as aplicações. Disponível em: https://hub.docker.com/u/marciosbicigo

4.7 Implementação do GitOps com Argo CD

O *Argo CD* foi escolhido para realizar a entrega contínua das aplicações no cluster *Kubernetes* devido à sua capacidade de automatizar o gerenciamento de implantações e sincronizar o estado desejado da infraestrutura com o estado real. Com ele, as equipes podem aplicar práticas *GitOps*, controlando mudanças por meio de repositórios Git, o que simplifica as operações e minimiza erros humanos. Essa abordagem permite que os desenvolvedores se concentrem na entrega de valor, enquanto outra equipe gerencia e escala a infraestrutura de forma eficiente (Yuen *et al.*, 2021).

Para facilitar essa automação, utilizamos o recurso *ApplicationSet* do *Argo CD*, que permite criar um template que gera múltiplas aplicações a partir de fontes dinâmicas, como subdiretórios em um repositório Git, por exemplo. Essa configuração facilita a atualização de novas versões ou a inclusão de aplicações adicionais ao projeto, eliminando a necessidade de criar, editar ou remover manualmente recursos diretamente no cluster (Vinto; Bueno, 2023). Um exemplo disso seria a atualização da versão de uma aplicação: Neste caso, bastaria editar a *tag* da imagem no arquivo *deployment.yaml* correspondente. Dessa forma, o *Argo CD*, por meio do *ApplicationSet*, detectaria automaticamente mudanças como essa dentro do repositório, e automaticamente aplicaria as alterações no cluster *Kubernetes*. Isso promove mais segurança, consistência e uma entrega contínua eficiente.

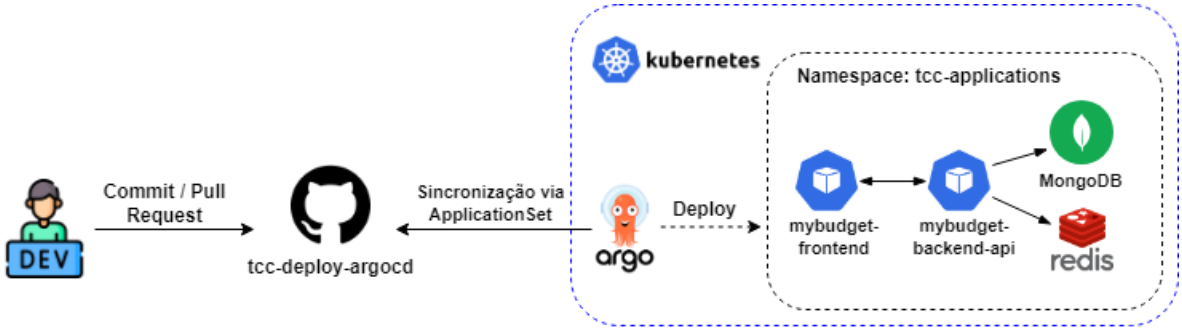
Neste trabalho, elaboramos um manifesto de *ApplicationSet* denominado *tcc-applicationset.yaml*[7], que automatiza a gestão das aplicações de um repositório Git chamado *tcc-deploy-argocd*[8]. Este manifesto foi projetado para monitorar o diretório “apps” do repositório, que contém quatro aplicações distintas: *mybudget-frontend*, *mybudget-backend-api*, *mongodb* e *redis*. As aplicações serão implantadas no *namespace* “*tcc-applications*”.

O comando utilizado para aplicar o manifesto ao cluster foi o seguinte: `kubectl apply -f https://raw.githubusercontent.com/mjsbicigo/tcc-deploy-argocd/main/argocd/tcc-applicationset.yaml`.

O fluxo completo da implantação realizada pelo *Argo CD* foi exemplificado na figura 7.

[7] Arquivo de ApplicationSet utilizado no projeto. Disponível em: https://raw.githubusercontent.com/mjsbicigo/tcc-deploy-argocd/main/argocd/tcc-applicationset.yaml.
[8] Repositório tcc-deploy-argocd, monitorado pelo Argo CD. Disponível em: https://github.com/mjsbicigo/tcc-deploy-argocd

Figura 7 – Fluxo do processo automatizado de Entrega Contínua via Argo CD



Fonte: Próprio Autor

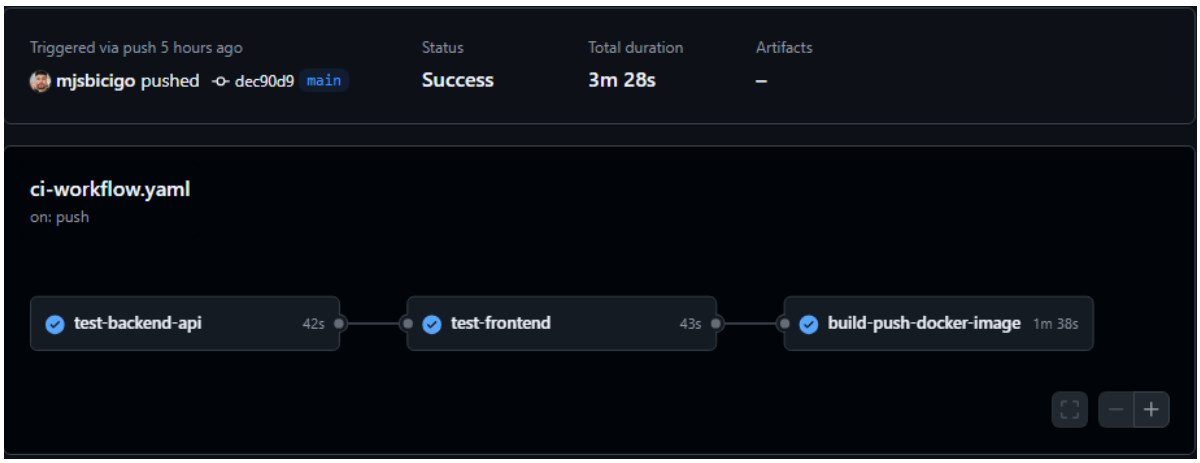
4.8 Validação do ambiente

Para validação do processo de implementação, detalharemos o comportamento do ambiente ao realizar uma atualização de versão nas aplicações *frontend* e *backend-api*.

Ao realizar um commit ou pull request da nova versão na *branch* principal do repositório que armazena o código fonte das aplicações (repositório *mybudget*), o processo de CI foi iniciado e o *workflow* foi automaticamente executado através do GitHub Actions. Conforme mostrado na figura 8, levou cerca de três minutos e meio para realização dos testes, criação das imagens de contêiner e envio para o *Docker Hub*.

A sintaxe de identificação das imagens Docker seguem o seguinte padrão: `nome-do-repositório/nome-da-aplicação:tag`. Essa informação pode ser obtida através do repositório da aplicação no *Docker Hub*.

Figura 8 – Execução do Workflow de CI



Fonte: Próprio Autor

Com posse do endereço das imagens, bastou alterá-los em seus respectivos arquivos "*deployment.yaml*" no repositório monitorado pelo Argo CD. E após um curto período de detecção, ele fez a sincronização automática no cluster *Kubernetes*, aplicando as novas versões.

Figura 9 – Recursos implantados pelo *Argo CD* no cluster *Kubernetes*.

```
marcio@ubuntu-server:~$ date
Tue Oct 29 01:12:20 PM UTC 2024
marcio@ubuntu-server:~$ kubectl apply -f https://raw.githubusercontent.com/mjsbicigo/tcc-deploy-argocd/main/argocd/tcc-a
pplicationset.yaml
applicationset.argoproj.io/tcc-applicationset created
marcio@ubuntu-server:~$ kubectl get all -n tcc-applications
NAME                                READY   STATUS    RESTARTS      AGE
pod/mongodb-7cf5479c56-hz2v8        1/1     Running   0             33s
pod/mybudget-7f45fd5bb6-xnn2p       1/1     Running   0             33s
pod/mybudget-api-5b747f95ff-dhtw4   1/1     Running   1 (27s ago)   33s
pod/redis-68568cb68c-v6pfb          1/1     Running   0             33s

NAME                        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)           AGE
service/svc-mongodb         NodePort    10.96.125.211   <none>        27017:30017/TCP   33s
service/svc-mybudget        NodePort    10.96.171.214   <none>        8080:30080/TCP    34s
service/svc-mybudget-api    NodePort    10.96.130.240   <none>        8081:30081/TCP    34s
service/svc-redis           ClusterIP   10.96.161.130   <none>        6379/TCP          33s

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mongodb        1/1     1            1           33s
deployment.apps/mybudget       1/1     1            1           33s
deployment.apps/mybudget-api   1/1     1            1           33s
deployment.apps/redis          1/1     1            1           33s

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/mongodb-7cf5479c56        1         1         1       33s
replicaset.apps/mybudget-7f45fd5bb6       1         1         1       33s
replicaset.apps/mybudget-api-5b747f95ff   1         1         1       33s
replicaset.apps/redis-68568cb68c          1         1         1       33s
marcio@ubuntu-server:~$ date
Tue Oct 29 01:13:08 PM UTC 2024
marcio@ubuntu-server:~$
```

Fonte: Próprio Autor

O resultado final pode ser visualizado tanto pela interface web do *Argo CD*, quanto pelo comando `kubectl get all -n tcc-applications`, que mostra todos os recursos implantados pelo *Argo CD* no *namespace* "*tcc-applications"*, conforme apresentado na figura 9. Como o Argo CD não possui uma ferramenta nativa para visualização de tempo de deploy, utilizamos o comando "`date`" antes e depois da sincronização, notando uma diferença de pouco menos de 1 minuto para subir as novas versões no cluster.

5. Resultados e discussão

Nesta seção, analisaremos os resultados comparando o processo de deploy manual tradicional com o ambiente automatizado utilizando GitHub Actions e Argo CD.

No cenário manual, o deploy envolve uma série de passos executados em diferentes servidores, o que aumenta o risco de falhas humanas e problemas de segurança. Em contrapartida, o ambiente automatizado reduz esses riscos, oferecendo maior eficiência e confiabilidade. Avaliaremos aspectos como número de etapas, tempo de deploy, segurança, confiabilidade e facilidade de *rollback* (Yuen *et al.*, 2021).

5.1 Número de passos realizados

No processo tradicional, o deploy de uma aplicação envolve operações manuais e repetitivas, como testes locais em máquinas sem automatização, acesso via SSH, cópia e substituição manual de arquivos no servidor, e reinicialização dos serviços. Esse fluxo extenso aumenta a possibilidade de erros humanos e torna o processo lento e difícil de reproduzir. Dependendo da complexidade do projeto, cada deploy pode envolver mais de 10 etapas e exigir coordenação entre desenvolvedores e operadores, tornando o *rollback* em caso de falhas complicado e demorado.

Por outro lado, no ambiente automatizado proposto, os passos manuais são praticamente eliminados. Após um commit no repositório Git, o processo de Integração Contínua foi acionado automaticamente pelo GitHub Actions, que executou testes padronizados e construiu as imagens Docker. Um commit no repositório monitorado pelo *Argo CD* acionou o deploy no cluster *Kubernetes* de forma automatizada. Esse fluxo reduz os passos manuais a apenas 2 (ou 3, caso precise de alguma aprovação manual) garantindo uma execução rápida, confiável e rastreável.

5.2 Tempo de deploy

O tempo de deploy no cenário manual é bastante variável, dependendo de fatores humanos. O desenvolvedor ou operador precisa realizar testes locais e acessar o servidor via SSH, com cada etapa sendo sequencial e manual. Qualquer erro na troca de binários ou na reinicialização de serviços pode resultar em um tempo de inatividade considerável. Assim, um deploy completo pode levar de 20 a 30 minutos ou mais, dependendo da complexidade da aplicação.

Em contraste, no ambiente automatizado, o tempo total de deploy foi drasticamente reduzido. Com o uso do *Argo CD* em conjunto com o GitHub Actions, o processo de CI/CD foram acionados automaticamente após cada commit e ao todo, o deploy levou cerca de 5 a no máximo 10 minutos, dependendo da aplicação e da capacidade de processamento do Servidor. Essa significativa redução no tempo de ociosidade e a eliminação da necessidade de reinicializações manuais foram asseguradas pelo recurso de *Deployment* do *Kubernetes*, que possibilitou a continuidade dos serviços sem interrupções significativas. Dessa forma, a automação não apenas melhora a eficiência operacional, mas também permite que as equipes se concentrem em atividades de maior valor agregado.

5.3 Segurança e confiabilidade

No cenário manual, a segurança e a confiabilidade são pontos críticos. O acesso via SSH a servidores de produção apresenta riscos, pois erros no uso ou armazenamento de credenciais podem comprometer o ambiente. Além disso, a cópia e reinicialização manual de binários aumenta a probabilidade de erro humano, resultando em falhas ou indisponibilidade. Essa execução manual dificulta garantir que todos os deploys sigam o mesmo fluxo, levando a inconsistências entre ambientes.

Em contraste, o processo automatizado com *GitOps* melhorou significativamente tanto a segurança quanto a confiabilidade. Todas as mudanças passaram a ser controladas e versionadas no repositório Git, e o *Argo CD* possibilitou que as alterações fossem aplicadas somente após a aprovação dos administradores, eliminando a necessidade de acesso direto ao servidor. O processo de CI/CD assegurou que cada deploy seguisse exatamente o mesmo fluxo, independentemente de quem fez o commit ou da aplicação, garantindo que o estado do cluster estivesse sempre sincronizado com o desejado. Em um cenário de maior escala, isso proporciona consistência total entre todos os ambientes.

5.4 Facilidade de rollback

No ambiente manual, o *rollback* é um processo demorado e suscetível a erros. Se um deploy falhar, o operador precisa reverter manualmente os arquivos e reiniciar os serviços, o que pode levar a períodos prolongados de indisponibilidade. Com o Argo CD e GitOps, o *rollback* é simples e rápido: pela interface web do Argo CD, é possível reverter para uma versão anterior com apenas alguns cliques, restaurando o estado desejado do cluster em minutos, minimizando o impacto de falhas e garantindo uma rápida recuperação.

6. Conclusão

Este projeto implementou um ambiente de automação de deploys, combinando práticas de Integração Contínua com GitHub Actions e de Entrega Contínua com GitOps e Argo CD em um cluster *Kubernetes* local. A automação trouxe ganhos significativos em eficiência, segurança e confiabilidade ao processo de deploy. Em comparação ao cenário manual, houve uma drástica redução nas etapas operacionais (permitindo a subida de uma aplicação em apenas 2 ou 3 passos) e no tempo total, que passou de 20-30 minutos para apenas 5-10 minutos. Além

disso, o acesso centralizado via GitOps eliminou os riscos de segurança envolvidos no acesso manual aos servidores.

O processo agora é padronizado, rastreável e permite *rollbacks* automáticos em caso de falhas, promovendo um ciclo de desenvolvimento mais seguro e controlado. Os principais desafios incluíram a configuração inicial das ferramentas e as limitações de escalabilidade do ambiente local, mas o sucesso da automação destaca o potencial dessas práticas para otimizar a entrega de software em ambientes reais.

Trabalhos futuros podem explorar a expansão para ambientes em nuvem, a inclusão de testes mais complexos no CI e a integração de monitoramento com ferramentas como Prometheus e Grafana para análise de desempenho. Em resumo, a automação do deploy com GitOps e Argo CD mostrou-se uma solução eficaz para reduzir o tempo de implantação e aumentar a segurança e confiabilidade, tornando o processo de desenvolvimento e entrega mais ágil e robusto.

REFERÊNCIAS

ARUNDEL, John; DOMINGUS, Justin. **Cloud Native DevOps with Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud.** O'Reilly Media. 2019.

BURNS, Brendan; BEDA, Joe; HIGHTOWER, Kelsey. **Kubernetes: Up & Running.** O'Reilly Media. 2019.

BURNS, Brendan; TRACEY, Craig. **Managing Kubernetes: Operating Kubernetes Clusters in the Real World.** O'Reilly Media. 2018.

DOCKER. **Docker Documentation**. 2024. Disponível em: https://docs.docker.com/. Acesso em: 4. nov. 2024.

GITHUB. **GitHub Documentation**. 2024. Disponível em: https://docs.github.com/pt/. Acesso em: 4. nov. 2024.

KUBERNETES. **Kubernetes Documentation**. 2024. Disponível em: https://kubernetes.io/docs/home/. Acesso em: 4. nov. 2024.

MATTHIAS, Karl; KANE, Sean P. **Docker: Up & Running.** O'Reilly Media. 2015.

OKKEN, Brian. **Python Testing with pytest: Simple, Rapid, Effective, and Scalable**. Pragmatic Bookshelf. 2017.

VINTO, Natale; BUENO, Alex S. **GitOps Cookbook: Kubernetes Automation in Practice.** O'Reilly Media. 2023.

YUEN, B. *et al.* **GitOps and Kubernetes: Continuous Deployment with Argo CD, Jenkins X, and Flux.** Manning Publications. 2021.