Implementando um sistema de conteinerização com Kubernetes usando GitOps

Pedro G. R. Rodrigues[1], Vinicius C. Garcia[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Recife – PE – Brasil

{pgrr,vcg}@cin.ufpe.br

Abstract. *This paper aims to implement a Kubernetes infrastructure using GitOps on a personal local hosted server in order to solve some issues which a raw linux install is not able to solve, from the server first setup and maintenance to the continuous delivery of new apps or updates.*

Resumo. *Esse trabalho visa implementar uma infraestrutura em Kubernetes com GitOps em um servidor pessoal hospedado localmente para solucionar alguns problemas com os quais uma instalação crua do linux não é capaz de resolver, desde a instalação e manutenção do servidor de forma automatizada, até a entrega contínua de aplicações novas ou atualizações em produção.*

1. Introdução

Todo ano são lançados componentes eletrônicos novos e com o tempo os componentes atuais se tornam obsoletos e com isso é gerado muito lixo eletrônico [24], o que foi a motivação inicial para entender como seria possível aproveitar máquinas antigas de forma prática em um contexto mais atual.

Kubernetes é um forte candidato para reaproveitamento de hardware antigo pelos seus baixos requisitos de hardware para rodar containers linux dado que ele é o sistema de gerenciamento de containers mais popular da atualidade [1] e também pelo fato do mesmo ser open source e existir vários serviços que fornecem ele de forma paga (Amazon Web Services e Azure Microsoft são exemplos), mas também de forma gratuita ao usar o mesmo em servidores proprietários ou até computadores pessoais.

Apesar do Kubernetes sozinho já ser uma ferramenta bastante completa, ainda é preciso definir processos para organizar como os artefatos das aplicações são construídos e empacotados até o como eles vão chegar no ambiente de produção, e esses processos podem ser manuais ou automatizados.

O GitOps [3] entra justamente na parte de automatizar processos de modo que ele busca ter descrições declarativas dos estados dos recursos desejados no ambiente de produção para manter uma consistência entre o estado em produção que pode ser volátil, como por exemplo um ambiente no qual múltiplos usuários realizam alterações manualmente nos recursos, e o estado desejado que é o estado declarado no git, um exemplo disso seria um arquivo de configuração declarando que uma aplicação é para funcionar com 3 instâncias de replicação.

A necessidade do GitOps vêm justamente de casos de inconsistência de estados, como por exemplo uma aplicação era para ter 3 instâncias em execução em paralelo,

porém algo mudou esse número, um servidor ficou indisponível ou essa contagem foi alterada manualmente, e não é possível ser atento a todas essas mudanças de forma manual, é justamente nesse ponto em que entra alguma ferramenta que fica de vigília para mudanças entre o estado de produção e o estado desejado, que é o estado descrito no git.

O objetivo geral deste trabalho é investigar formas de melhorar o uso/gestão de uma infraestrutura local de Kubernetes o que deriva nos seguintes pontos mais específicos.

1. Permitir a colaboração remota com mais segurança
2. Melhorar a experiência de deploy de uma aplicação para o desenvolvedor
3. Melhorar a experiência de manutenções programadas para o administrador

2. Fundamentação Teórica

Antes de explicar a metodologia e implementação do trabalho, é preciso compreender alguns conceitos fundamentais que guiaram esse trabalho e também as ferramentas utilizadas para implementar o mesmo.

2.1 Containers e orquestração

Para entender o que é o Kubernetes é necessário entender o que são containers e qual a importância de orquestrar os mesmos.

Resumindo bastante é possível comparar containers com máquinas virtuais, porém mais leves por possuir um isolamento de recursos mais relaxado ao comparado com máquinas virtuais [1], porém igualmente com máquinas virtuais, os contêineres possuem um sistema de arquivos próprio, uma quota de uso de CPU, memória e processamento e pelo fato dele ser desacoplado da infraestrutura é possível executar um mesmo container em ambientes distintos e obter o mesmo resultado.

A orquestração é o processo de gerenciamento desses containers em um ambiente de produção para evitar problemas como containers fora do ar ou não responsivos provocarem tempo de indisponibilidade nos serviços [1] e o Kubernetes é justamente a ferramenta que executa e gerencia containers de forma distribuída.

2.2 DevOps

O termo DevOps é definido como a combinação de desenvolvimento com operações [4] e isso resulta em uma série de princípios e práticas de desenvolvimento e comunicação que permitem a colaboração ágil e eficiente entre os desenvolvedores e os responsáveis pela infraestrutura que os mesmos usam.

Outro termo importante para o assunto é SRE, que significa "Site Reliability Engineering", que pode ser visto como um conjunto de conceitos e regras mais objetivas para implementar DevOps [5] e o termo também é usado como descrição para times de infraestruturas que se utilizam desses conceitos e regras [6].

2.2.1 Princípios do DevOps

O primeiro princípio é de iterações, onde é necessário desenvolver softwares em ciclos assim como definido na metodologia ágil [7], e o princípio a seguir é o de incrementos, que é justamente uma consequência do princípio anterior e dita que após cada iteração sobre o desenvolvimento, deverá ser alcançado alguma melhoria ao comparar o software com a versão anterior.

Em seguida tem o princípio de continuidade que dita que toda etapa do desenvolvimento, seja testes, deploys ou planejamentos, deve ser realizada de forma contínua e outro princípio que ajuda nesse ponto é o de automação, onde é fundamental procurar automatizar tarefas repetitivas.

Outro princípio importante é de *self-service* onde é visado diminuir o tempo gasto em tarefas para outras equipes através de processos que possam permitir os desenvolvedores solicitarem recursos de forma autônoma, como por exemplo, um desenvolvedor deveria ser capaz de solicitar um banco de dados novo a um time de plataforma com o mínimo de processos entre os times.

Os últimos dois princípios são o de colaboração, que apesar de ser um princípio genérico, o mesmo é necessário para sempre buscar formas de se incrementar as interações dentro e entre times, e por último o princípio de holismo no qual o time deve estar estruturado de modo que é possível desenvolver, testar e deployar de forma independente a fim de evitar gargalos quando comparado com um time onde cada pessoa teria uma função específica, como por exemplo, uma pessoa só desenvolve enquanto outra só realiza testes.

2.3 Infraestrutura como Código

Infraestrutura como código, abreviada como IaC, está fortemente relacionada com outra prática fundamental de DevOps, que é a automação de tarefas de infraestrutura, como por exemplo, um desenvolvedor precisa de um banco de dados novo em produção e o time de infraestrutura tem esse processo automatizado de algum modo, e uma dessas maneiras de automação é justamente ter o banco de dados descrito como código e com ajuda de alguma ferramenta essa descrição do banco é traduzida para o ambiente de produção.

O conceito IaC surge após o DevOps como uma necessidade de atender a problemas como o gerenciamento de grandes infraestruturas na nuvem e também uma oportunidade de promover colaboração e agilidade nas demandas de infraestrutura [8].

Vale ressaltar que é possível abordar IaC de duas formas, uma forma declarativa onde definimos o estado esperado do sistema e a ferramenta utilizada para automação se responsabiliza de garantir que esse estado é alcançado, e outra forma que é imperativa onde definimos quais passos precisamos executar para chegar ao estado necessário do sistema [9]. Este trabalho usa uma ferramenta imperativa, o Ansible, para configurar o servidor e outra ferramenta declarativa, o ArgoCD, para definir as aplicações que são executadas no servidor.

2.4 GitOps

Resumindo, GitOps é uma estrutura operacional que se utiliza das práticas de DevOps para automação de infraestrutura de modo que a operação de gerenciamento de infraestrutura seja centrada na experiência do desenvolvedor [10].

GitOps é bastante similar a DevOps, tendo como único ponto de diferença o foco, enquanto DevOps é muito sobre cultura organizacional, o GitOps é focado nos desenvolvedores, assim usando o Git como fonte da verdade para o estado desejado em produção, seja para as aplicações, como para a infraestrutura [3].

Ao implementar GitOps podemos usar 2 abordagens distintas, a de empurrar ("push" em inglês) na qual ao adicionar mudanças ao código, o processo de entrega contínua é responsável por subir essas mudanças no ambiente de produção e a outra abordagem que é puxar ("pull" em inglês) as mudanças na qual há algum sistema que detecta mudanças de forma ativa entre o código e o estado da aplicação a garante que ambos sempre estão em sincronia [3].

A ferramenta de GitOps que será utilizada neste trabalho é o ArgoCD, que é uma ferramenta declarativa de GitOps para o Kubernetes que usa a abordagem de observar as mudanças entre o estado do sistema e o git a fim de manter ambos sincronizados o tempo todo [11].

3. Metodologia

Será utilizado a metodologia de objetivos, questões e métricas [2] (GQM em inglês) que consiste em definir objetivos a serem alcançados, perguntas derivativas desses objetivos e métricas que respondam essas mesmas perguntas e para definir as perguntas é necessário condensar os objetivos gerais deste trabalho em objetivos mais específicos.

Para mensurar o quanto cada objetivo foi alcançado é preciso definir um cenário antes da implementação para ter um ponto base de comparação com o cenário após a implementação e para este trabalho o cenário pré implementação é o servidor linux sem nada a mais instalado e o cenário pós implementação com a configuração do servidor automatizada com GitOps mais o Kubernetes com o ArgoCD.

Para implementar esse trabalho será utilizado um computador Mac mini modelo 2010, com 8 gigas de ram com um Intel Core 2 Duo 2,4GHz usando o sistema operacional Debian sem interface gráfica para evitar disputa de recursos entre o cluster e a interface gráfica e para avaliar as métricas será utilizado a mesma máquina sem e com GitOps na mesma rede residencial para ter uma comparação equivalente.

3.1 Permitir a colaboração remota com mais segurança

Esse objetivo deriva dos possíveis problemas de segurança que podem existir no cenário sem GitOps onde para um usuário remoto colaborar com o servidor é necessário vários acessos que podem ser potenciais de vulnerabilidades no sistema e com isso temos duas perguntas que vão mensurar o quanto o objetivo foi alcançado, a primeira sendo, quais os acessos são precisos ter para colaborar com o servidor e a segunda, quais os problemas de segurança que as abordagens sem e com GitOps teria.

3.2 Melhorar a experiência de deploy de uma aplicação para o desenvolvedor

Esse objetivo propõe investigar o processo de deployment de aplicações em produção a fim de entender possíveis pontos de melhora na jornada de um desenvolvedor subir uma aplicação em produção.

Para entender a jornada do desenvolvedor é necessário definir uma sequência de passos genéricos no qual a maioria das aplicações seguem o mesmo formato, e para este trabalho serão considerados os passos logo a seguir.

1. Acessar a infraestrutura onde essa aplicação será executada.
2. Baixar a instalação da aplicação que representa a versão desejada da mesma na infraestrutura.
3. Instalar as dependências do artefato na infraestrutura.
4. Instalar o artefato em si na infraestrutura.

Independente se é o cenário com ou sem GitOps, esses passos não deixam de existir, eles só acabam sendo automatizados pelas ferramentas usadas para implementar o GitOps, e com isso é possível mensurar uma melhora na experiência de deploy através do número de passos manuais necessários para o deploy em ambos os cenários.

3.3 Melhorar a experiência de manutenções programadas para o administrador

A ideia desse objetivo é investigar alternativas para facilitar a realização de manutenções programadas ao servidor que diminuam a frequência de intervenções manuais no servidor, de modo que o administrador da infraestrutura tenha menos trabalho, como por exemplo, evitar a necessidade de acesso via ssh ou com um teclado.

Para entender como melhorar a experiência de manutenções programados é necessário definir o que é uma manutenção programada, que é basicamente a atualização das bibliotecas e serviços necessárias para executar as aplicações de produção, no cenário com GitOps um exemplo disso seria atualizar a instalação do Kubernetes e no cenário sem GitOps poderia ser somente atualizar as bibliotecas ou o kernel do linux.

Uma vez que o conceito de manutenção programada foi definido, é possível usar a frequência com que é necessário realizar tais manutenções e o quanto das mesmas são automatizáveis no cenário do GitOps, para então avaliar o quanto é possível melhorar a experiência de realizar manutenções programadas.

Tabela 1. Resumo dos objetivos com suas perguntas e métricas equivalentes

Objetivo	Perguntas	Métricas
Permitir a colaboração remota de forma segura	Quais acessos é preciso ter?	Número total de acessos/credenciais necessárias sensíveis
	Quais os problemas de	Lista de vulnerabilidades que

	segurança?	podem ser exploradas
Melhorar a experiência de deploy de uma aplicação para o desenvolvedor	Qual o número de passos necessários para subir uma aplicação?	Número de passos manuais para um usuário subir uma aplicação
Melhorar a experiência de manutenções programadas para o administrador	Com qual frequência devo realizar manutenções programadas?	Frequência mínima de manutenções

4. Implementação

Para implementar esse projeto foi necessário instalar o linux na máquina destino, configurar o linux com tudo necessário para configurar o mesmo de forma remota e instalação e configuração do Kubernetes.

4.1 Instalação do Linux

Para instalar o linux na máquina alvo foi escolhida a distribuição Debian na sua versão 11 sem interface gráfica que além de ser uma distribuição leve, também oferece uma boa similaridade de pacotes com a distribuição Ubuntu que é bastante popular e usa o sistema Debian como base.

Vale ressaltar que existem outras distribuições que também possam ser boas opções para servidores como o Fedora e o CentOS, entre outros, porém não é o objetivo deste trabalho comparar a melhor distribuição para um servidor.

Um ponto importante a se notar é que para instalar o Debian no Mac mini modelo 2010, foi necessário usar a distribuição do Debian com suporte a drivers terceiros para algumas funcionalidades, como foi o caso do driver de rede.

O computador alvo deste trabalho possui 8GB de memória ram e 2 discos rígidos, um disco do tipo Solid State Drive (SSD) de 120GB no qual foi instalado a inicialização do sistema operacional (partição EFI), os arquivos base do sistema (partição /) e a memória swap, e outro disco do tipo Hard Drive Disk (HDD) de 320GB para armazenamento de dados de aplicações terceiras (partição /var) que nesse cenário é onde o Kubernetes vai armazenar os dados das aplicações.

4.2 Configuração do Linux

Após a instalação do linux vêm o primeiro passo para definir a infraestrutura via código, no qual foi usado a ferramenta Ansible [12] e bash para automatizar e descrever o que precisamos configurar e instalar no servidor para usar o Kubernetes no mesmo.

Foi implementado um script que instala todas as dependências necessárias para instalar o Ansible e que também configura uma tarefa assíncrona para atualizar o repositório que contém o nosso código Ansible e rodar o mesmo logo em seguida, assim mantendo o servidor atualizado sem intervenção manual.

O Ansible usa o conceito de receitas para se referir a instruções a serem executadas em alguma máquina, na receita do servidor primeiramente é instalado as bibliotecas necessárias para os passos seguintes, que são o *network-manager* para gerenciamento da rede do servidor e o *curl* para interação com arquivos da internet e suporte a outros scripts posteriores; em seguida é configurado um IP fixo e DNS para a máquina para facilitar o acesso da mesma em uma rede local.

4.3 Configuração do Kubernetes

Existem várias opções para instalar o Kubernetes de forma manual, sem usar algum provedor de nuvem, e para esse trabalho foi escolhida a K3s [13] que é uma instalação mais leve do que a padrão justamente visando ambientes com pouco poder computacional como é o caso de dispositivos de internet das coisas (IoT) ou mini computadores como o Raspberry PI.

Para instalar o K3s foi usado o script de instalação padrão que o mesmo fornece e após isso foi instalada a versão mais recente do ArgoCD no cluster e em seguida é criado uma aplicação que já declara todas as outras aplicações que queremos no cluster [14], assim o cluster já começa a instalar todas as aplicações definidas no git sem necessidade de intervenção manual.

Todo o processo para instalar o Kubernetes, o ArgoCD, e configurar o mesmo utilizou os últimos passos da receita do Ansible para tal.

5. Experimentos e Análise

Para mensurar o quanto foi alcançado dos objetivos propostos na metodologia é necessário comparar os cenários de sem GitOps e sem Kubernetes, onde o cenário sem GitOps pode ser resumido como uma instalação do linux limpa onde o desenvolvedor precisaria de acesso direto ao servidor para subir uma aplicação ao mesmo, e o cenário com GitOps no mesmo servidor.

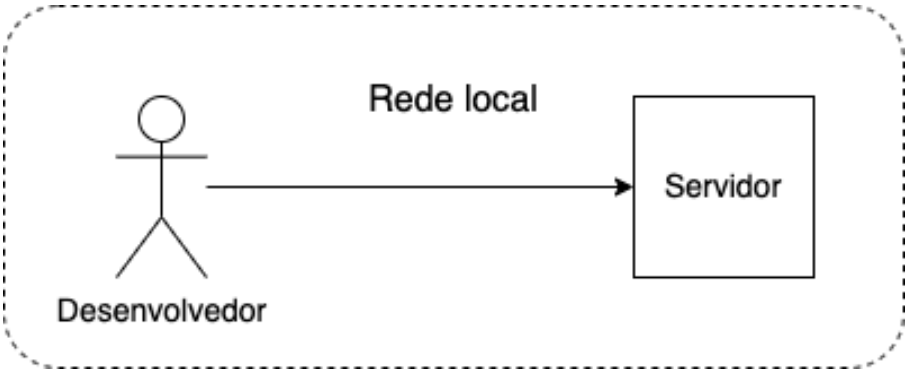

Figura 1. Arquitetura sem GitOps

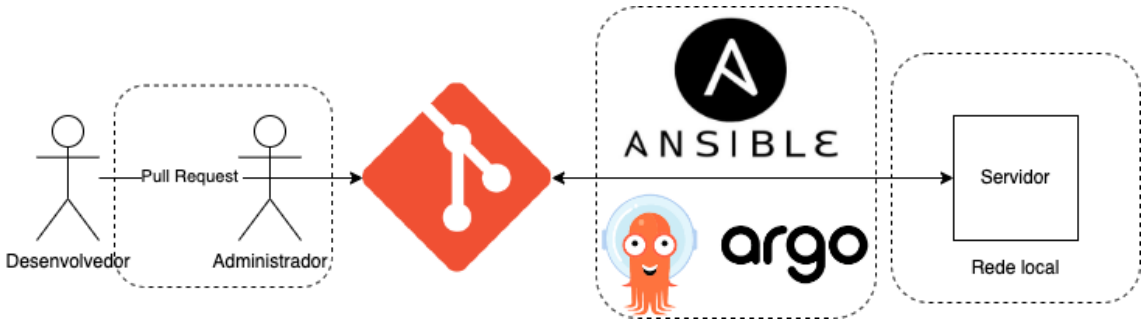

Figura 2. Arquitetura com GitOps

5.1 Permitir a colaboração remota com mais segurança

Para definir se foi possível permitir colaboração remota com mais segurança no projeto é preciso definir os potenciais problemas de segurança em ambos os cenários sem e com GitOps.

No cenário sem GitOps é necessário só é possível colaborar com o servidor caso o colaborador tenha acesso a rede interna onde o servidor se encontra e as credenciais para acesso do mesmo e isso abre brechas para alguns problemas clássicos de segurança a serem listados posteriormente.

que impede a mesma de ocorrer, que nesse caso é falta de conectividade da rede externa para a rede local, o servidor acessa a internet, mas a rede local não é exposta na internet, e mesmo se a rede local for exposta para a internet, isso gera uma necessidade de compartilhamento das senhas e chaves do servidor com terceiros para os mesmos serem capazes de entrar na máquina e subir aplicações.

Tabela 2. Resumo dos pares de perguntas e métricas do objetivo neste tópico

Pergunta	Métrica
Quais acessos é preciso ter?	Número total de acessos/credenciais sensíveis
Quais os problemas de segurança?	Lista de vulnerabilidades que podem ser exploradas

A tabela 2 apresenta os pares de perguntas e métricas necessárias para avaliar a completude do objetivo, que são quais acessos é preciso ter que será mensurado com o número total de acessos e credenciais sensíveis necessárias para acessar a máquina e quais os possíveis problemas de segurança que a implementação pode ter e a lista de vulnerabilidades para tornar essa pergunta mensurável.

Para um usuário colaborar de forma remota no servidor seria necessário os seguintes acessos:

1. Acesso à rede local onde o servidor está instalado.

2. Conhecimento das credenciais de acesso ao servidor como a combinação de usuário e senha com as permissões necessárias para instalar aplicações no mesmo.

A partir do momento que um usuário externo tem acesso à rede local de uma arquitetura, os outros sistemas que ali residem estão vulneráveis a vários ataques, como por exemplo:

1. Ataques de interceptação [15].
2. Ataques de força bruta para descoberta de senhas [15].
3. Instalação de softwares maliciosos como Spywares [16].
4. Ataques de negação de serviço [16].

Enquanto ao comparar com o cenário com GitOps, para contribuir com o servidor é somente necessário abrir um PR ("Pull Request" em inglês) para contribuir com o projeto e o administrador do sistema só teria de aceitar essa contribuição para essa mudança ser aplicada automaticamente em produção, assim não sendo necessário nenhum acesso sensível.

Ao utilizar GitOps, também é eliminada a necessidade de acesso ao usuário linux do servidor, embora esse ponto de vulnerabilidade possa ser contornado criando um usuário com acessos específicos só para desenvolvedores, o risco de se instalar um software malicioso ainda existe, visto que o container executando pode conter algum software malicioso, o mesmo já está em um contexto à parte por estar em um container, assim reduzindo o escopo de possíveis ataques para somente uma tentativa de travar o servidor ao consumir recursos demasiadamente [17], porém ainda seria possível tentar atacar outras aplicações dentro do cluster, assim totalizando 1 vulnerabilidade ainda possível, porém de risco aceitável visto que toda aplicação é passiva de aprovação prévia.

Vale ressaltar que essa métrica está contabilizando ataques clássicos e conhecidos, porém podem haver falhas não descobertas ainda no uso do Kubernetes e containers que não tem como serem contabilizadas neste trabalho, portanto foi considerado como segurança a capacidade de evitar ataques conhecidos.

Tabela 3. Comparação das métricas no ambiente sem e com GitOps

Métrica	Sem GitOps	Com GitOps
Número total de acessos/credenciais sensíveis	2	0
Lista de vulnerabilidades que podem ser exploradas	4	1

A tabela 3 compara as métricas sem e com GitOps no qual foi possível reduzir de 2 para 0 o número de acessos sensíveis necessários e de 3 para 1 o número de vulnerabilidades do servidor, sendo que o risco da vulnerabilidade é aceitável dado o processo de implantação de aplicações novas, sendo assim é possível considerar o objetivo alcançado.

5.2 Melhorar a experiência de deploy de uma aplicação para o desenvolvedor

Para definir o que seria uma possível melhora na experiência de deploy é necessário entender o que está ruim no processo sem GitOps, que nesse caso é o número de passos a serem executados para realizar um deploy, assim como definido na metodologia.

Tabela 4. Resumo dos pares de perguntas e métricas do objetivo neste tópico

Pergunta	Métrica
O que está ruim no processo de deploy?	Número de passos para um usuário subir uma aplicação no cluster

A tabela 4 apresenta a métrica necessária para mensurar o quão ruim está o processo de deploy usando como parâmetro o número de passos necessários para subir uma aplicação no cluster.

Dado que o contexto sem GitOps não possui um sistema de conteinerização, seria necessário instalar a aplicação de forma manual, assim tendo os seguintes passos abaixo.

1. Acessar o servidor
2. Copiar ou baixar o artefato da aplicação
3. Instalar as dependências da aplicação no servidor
4. Instalar a aplicação

Vale lembrar que os passos acima não contam com possíveis conflitos com outras aplicações, visto que o ambiente de execução é compartilhado.

Ao comparar com o cenário com GitOps, os únicos passos necessários para subir uma aplicação no servidor é abrir um PR com a descrição da aplicação em código e esperar até o administrador aceitar, como o segundo passo não é de responsabilidade do desenvolvedor e o objetivo é do ponto de vista do desenvolvedor, vamos considerar como passo somente o ato de abrir um PR.

Tabela 5. Comparação das métricas no ambiente sem e com GitOps

Métrica	Sem GitOps	Com GitOps
Número de passos para um usuário subir uma aplicação	4	1

A tabela 5 mostra uma redução de 4 passos para 1 passo somente ao migrar para uma abordagem GitOps no servidor, e mesmo que ainda seja necessário 1 passo, o mesmo é um passo fundamental que possibilita a revisão de outros desenvolvedores e dos administradores da infraestrutura o que é uma das bases para o conceito de GitOps.

5.3 Melhorar a experiência de manutenções programadas

Para conseguir mensurar esse objetivo é necessário antes definir como será obtida a frequência mínima de manutenção necessária dado as atualizações dos sistemas operacionais e as ferramentas utilizadas é preciso entender qual a necessidade de uma manutenção programada, que nesse caso é somente atualizar o sistema operacional e as ferramentas instaladas no mesmo.

Como a periodicidade com a qual devem ser realizadas manutenções programadas é igual a periodicidade de atualizações no sistema operacional e nas ferramentas, é possível calcular a frequência de atualizações através das frequências com as quais versões novas são lançadas.

Seguindo o sistema de versionamento semântico [18] existem 3 tipos de atualizações que um software pode ter, uma atualização do tipo *patch* só com correções de problemas, uma do tipo *minor* com adição de funcionalidades sem quebrar a compatibilidade com a versão anterior e um terceiro tipo, *major*, onde há mudanças nas interfaces do software que causam incompatibilidade com versões anteriores.

Para calcular a frequência de updates neste trabalho será considerado as atualizações do tipo *patch* visto que as atualizações do tipo *major* não se encaixam como manutenção programada visto que seria necessário rever se o processo de instalação das ferramentas passou por mudanças ou não, o que foge da categoria de manutenção para a categoria de uma atualização ao servidor, e as atualizações do tipo *minor* não mudam nenhuma funcionalidade já existente, então só é necessário atualizar esse tipo caso haja demanda.

Para obter a periodicidade de atualizações será calculada uma média do número de dias entre todas as datas de lançamentos das versões *patch* dos softwares utilizados no projeto.

Tabela 6. Resumo dos pares de perguntas e métricas do objetivo neste tópico

Pergunta	Métrica
Com qual frequência deve ser realizada manutenções programadas?	Frequência mínima de manutenções em dias

A tabela 6 relaciona a pergunta desse objetivo com a métrica necessária para avaliar a mesma que para esse caso é necessário listar o que é necessário para cada uma das abordagens, sem e com GitOps.

Sem GitOps é somente necessário se ater somente às atualizações do Debian, visto que os desenvolvedores são responsáveis pelo manutenção das dependências das aplicações instaladas, e para calcular a frequência de atualizações do Debian é possível consultar o registro de mudanças do mesmo [19] e calcular uma média de 78 dias entre cada atualização para o Debian 11.

Com GitOps continua sendo necessário a atualização do Debian, logo a frequência de atualizações do Debian é incluída nesse cenário, porém há a inclusão da frequência de atualizações do Ansible [20], do K3s [21] e do ArgoCD [22].

Tabela 7. Versões das ferramentas usadas para implementar o GitOps

Ferramenta	Versão *major* e *minor*
Ansible	v2.10
K3s	v1.24
ArgoCD	v2.4

Tabela 8. Frequência de lançamento de versões novas em dias

Ferramenta	Frequência mínima de atualizações em dias
Ansible	29
K3s	16
ArgoCD	7

A tabela 7 mostra a frequência de atualizações de cada ferramenta, com uma média de 29 dias para o Ansible, 16 dias para o K3s e 7 dias para o ArgoCD.

Como a métrica desse objetivo é a frequência mínima de atualização necessária para o servidor, no cenário com GitOps essa métrica corresponde a menor frequência encontrada, que nesse caso a frequência do ArgoCD que é de 7 dias.

Tabela 9. Comparação das métricas no ambiente sem e com GitOps

Métrica	Sem GitOps	Com GitOps
Frequência mínima de manutenções em dias	78	7

A tabela 9 compara a frequência mínima de atualizações em dias com o cenário sem e com GitOps, e embora no cenário com GitOps é necessário realizar manutenções programadas com uma frequência menor, graças às automações do Ansible não é necessário uma intervenção manual para atualizar o servidor uma vez que periodicamente o mesmo já busca por atualizações e aplica as mesmas, assim sendo possível considerar uma melhora também na situações com GitOps.

5. Conclusão

Através das métricas coletadas foi possível observar que ao implementar GitOps foi possível alcançar todos os objetivos propostos e ter uma melhora geral tanto na experiência do desenvolvedor quanto na experiência do administrador do servidor.

O ponto que mais se sobressai na implementação é o quanto o ArgoCD está maduro e entregou automações com o mínimo de esforço necessário, principalmente ao comparar com o Ansible, que não é nativamente feito para GitOps mas foi adaptado para esse cenário, onde existiu a necessidade de escrever alguns códigos a parte para garantir que o servidor esteja periodicamente atualizando o mesmo.

Vale ressaltar que uma possível comparação entre um servidor automatizado com Kubernetes porém sem GitOps contra um servidor com GitOps não traria diferença significativa nas métricas, mas sim no trabalho necessário para implementação e manutenção do mesmo, uma vez que seria necessário código extra em alguns pontos como, automatizar as rotinas de sincronização com o repositório git, aplicar as definições de aplicações no Kubernetes e esse cenário seria ainda pior em um contexto sem containers, em que a automação seria responsável por instalar as dependências das aplicações e ainda gerenciar os contextos a fim de evitar possíveis conflitos entre aplicações.

Todos os códigos usados para gerenciar o estado do servidor estão descritos em um repositório específico para este trabalho para possibilitar uma reprodução do mesmo [23].

5.1 Trabalhos correlatos

A ideia desse trabalho surgiu da vontade de reutilizar um hardware antigo que estava sem uso com linux, porém o foco foi em como trazer uma experiência boa de desenvolvimento e deploy para esse cenário e foi justamente nesse ponto que o GitOps entra como uma possibilidade de facilitar esse processo.

O trabalho [24] explica os problemas gerados a partir do excesso de lixo eletrônico, dado os métodos ineficientes de descarte e desconstrução utilizados para lidar com esse material, e a partir disso é proposto uma implementação e avaliação de um cluster de máquinas antigas utilizando o sistema OpenMPI [25] para construir a interação entre as máquinas e comparar a performance e custo benefício com uma máquina nova.

Há também outro trabalho [26] que utiliza o excesso de lixo eletrônico como introdução, mas é totalmente focado em comparar a performance do uso de diferentes sistemas para implementar a clusterização.

Diferente deste trabalho, o trabalho [27] se propõe a explorar e estudar os conceitos de GitOps de forma extensa sem comparar os ambientes com e sem GitOps e ele também explica mais sobre o Kubernetes e usa todos os conceitos explorados para subir uma aplicação open source, Conduit [28], para exemplificar o uso do ArgoCD para subir uma aplicação em um ambiente de desenvolvimento e de produção.

5.2 Trabalhos futuros

Com base nos trabalhos correlatos, uma possível evolução deste trabalho seria implementar os mesmos princípios para a construção de um cluster de máquinas com o Kubernetes utilizando o GitOps e comparar o mesmo com a performance e usabilidade de um cluster equivalente em algum provedor da nuvem como o serviço EKS da Amazon AWS [29].

Continuando com a ideia de um cluster de Kubernetes, também seria possível comparar o custo de usar tal cluster para tarefas esporádicas como executores de integração contínua, tarefas de agregação e transformação de dados e também processos de aprendizagem de máquina.

Um ponto que vale ressaltar foi que ao observar o fluxo de trabalho com o Kubernetes, foi possível notar alguns pontos não resolvidos no que diz respeito ao fluxo de trabalho do desenvolvedor.

Como um desenvolvedor adicionaria segredos ao Kubernetes de forma remota sem a interação do administrador no cenário onde ele não tem acesso direto ao cluster e somente acesso a interagir via git. É possível pensar em adicionar os segredos ao git com encriptação ou até o uso de alguma ferramenta externa que ajude nesse processo, porém a usabilidade e riscos de cada alternativa precisam ser avaliadas.

Outro ponto importante também é estruturar o processo de expor o servidor local para a internet de forma segura, uma alternativa possível seria somente permitir o acesso da rede externa a rede domiciliar por meio de configurações ao roteador local, porém é necessário avaliar os riscos de expor o IP da rede domiciliar para a internet e também avaliar possíveis alternativas e os riscos de cada alternativa para então decidir a estratégia a ser utilizada.

Referências

[1] KUBERNETES. What is Kubernetes. Disponível em: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>. Acesso em 18 de jul. 2022.

[2] VAN SOLINGEN, R. et al. Goal Question Metric (GQM) Approach. Encyclopedia of Software Engineering, 15 jan. 2002.

[3] BEETZ, F.; HARRER, S. GitOps: The Evolution of DevOps? IEEE Software, 2021.

[4] JABBARI, R. et al. What is DevOps? Proceedings of the Scientific Workshop Proceedings of XP2016 on - XP '16 Workshops, 2016.

[5] Google - Site Reliability Engineering. Disponível em: <https://sre.google/workbook/how-sre-relates/>. Acesso em 12 de set. 2022.

[6] Google - Site Reliability Engineering. Disponível em: <https://sre.google/sre-book/introduction/>. Acesso em: 12 set. 2022.

[7] Sharma, Sheetal & Sarkar, Darothi & Gupta, Divya. (2012). Agile Processes and Methodologies: A Conceptual Study. International Journal on Computer Science and Engineering. 4.

[8] MORRIS, K. INFRASTRUCTURE AS CODE : dynamic systems for the cloud age. S.L.: O'reilly Media, 2021.

[9] What is Infrastructure as Code (IaC)? Disponível em: <https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac>. Acesso em: 12 set. 2022.

[10] What is GitOps? | GitLab. Disponível em: <https://about.gitlab.com/topics/gitops/>.

[11] Argo CD - Declarative GitOps CD for Kubernetes. Disponível em: <https://argo-cd.readthedocs.io/en/stable/>. Acesso em: 12 de set. 2022.

[12] ANSIBLE, RED HAT. How Ansible Works | Ansible.com. Disponível em: <https://www.ansible.com/overview/how-ansible-works>. Acesso em: 12 de set. 2022.

[13] K3s: Lightweight Kubernetes. Disponível em: <https://k3s.io/>. Acesso em: 12 de set. 2022.

[14] Cluster Bootstrapping - Argo CD - Declarative GitOps CD for Kubernetes. Disponível em: <https://argo-cd.readthedocs.io/en/stable/operator-manual/cluster-bootstrapping/>. Acesso em: 12 de set. 2022.

[15] YLLI, E.; FEJZAJ, J. Man in the Middle: Attack and Protection. [s.l: s.n.]. Disponível em: <http://ceur-ws.org/Vol-2872/short08.pdf>. Acesso em: 14 de set. 2022.

[16] Stafford, Tom & Urbaczewski, Andrew. (2004). Spyware: The Ghost in the Machine.. Communications of The Ais - CAIS. 14. 570.

[17] Ormiston, Kathryn & Eloff, Mm. (2006). Denial-of-Service & Distributed Denial-of-Service on The Internet.. 1-14.

[18] PRESTON-WERNER, T. Semantic Versioning 2.0.0. Disponível em: <https://semver.org/>.

[19] Debian ChangeLog. Disponível em: <http://ftp.us.debian.org/debian/dists/bullseye/ChangeLog>. Acesso em: 1 out. 2022.

[20] Tags · ansible/ansible. Disponível em: <https://github.com/ansible/ansible/tags>. Acesso em: 4 out. 2022.

[21] Tags · k3s-io/k3s. Disponível em: <https://github.com/k3s-io/k3s/tags>. Acesso em: 4 out. 2022.

[22] Tags · argoproj/argo-cd. Disponível em: <https://github.com/argoproj/argo-cd/tags>. Acesso em: 4 out. 2022.

[23] RODRIGUES, P. G. R. tg-2022-1. Disponível em: <https://github.com/PedroRossi/tg-2022-1>. Acesso em: 18 out. 2022.

[24] LYNAR, T. M. et al. Clustering Obsolete Computers to Reduce E-Waste. International Journal of Information Systems and Social Change, v. 1, n. 1, p. 1–10, 1 jan. 2010.

[25] Open MPI: Open Source High Performance Computing. Disponível em: <https://www.open-mpi.org/>.

[26] Russo, Ruggero & Lamanna, D. Davide & Baldoni, Roberto. (2005). Distributed software platforms for rehabilitating obsolete hardware.

[27] MATTI, K. GitOps tool Argo CD in service management : Case: Conduit. URN:NBN:fi:amk-2021092718101: [s.n.].

[28] Implementations. Disponível em: <https://github.com/gothinkster/realworld>. Acesso em: 4 out. 2022.

[29] Serviço gerenciado do Kubernetes – Amazon EKS – Amazon Web Services. Disponível em: <https://aws.amazon.com/pt/eks/>. Acesso em: 4 out. 2022.