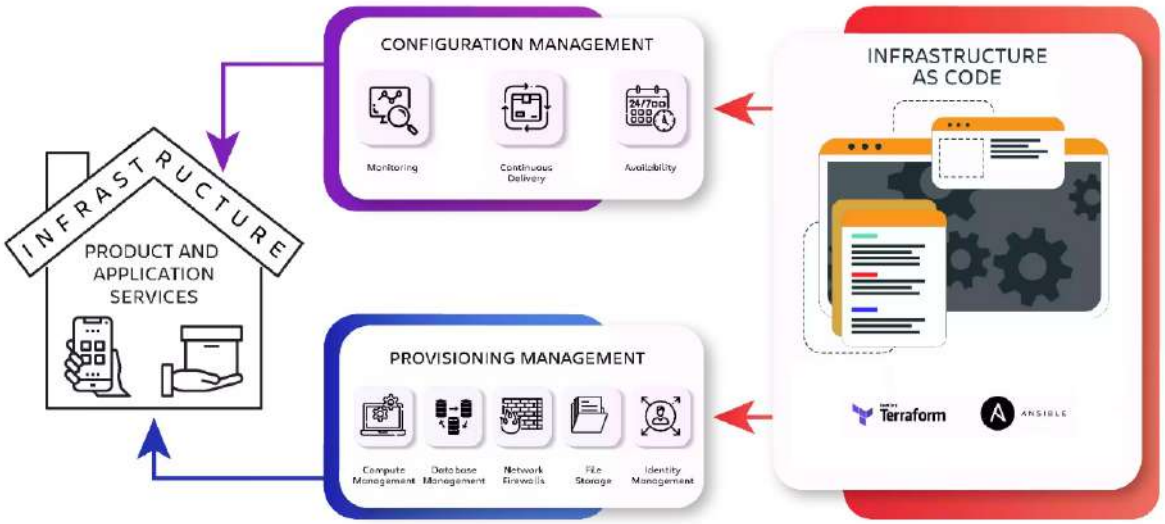
Proposta de Desenvolvimento de Solução para Automação de Entregas Contínuas de Softwares da Secretaria de Segurança Pública do Estado de Mato Grosso



Secretaria de Segurança Pública do Estado de Mato Grosso

Fundação de Amparo a Pesquisa do Estado de Mato Grosso

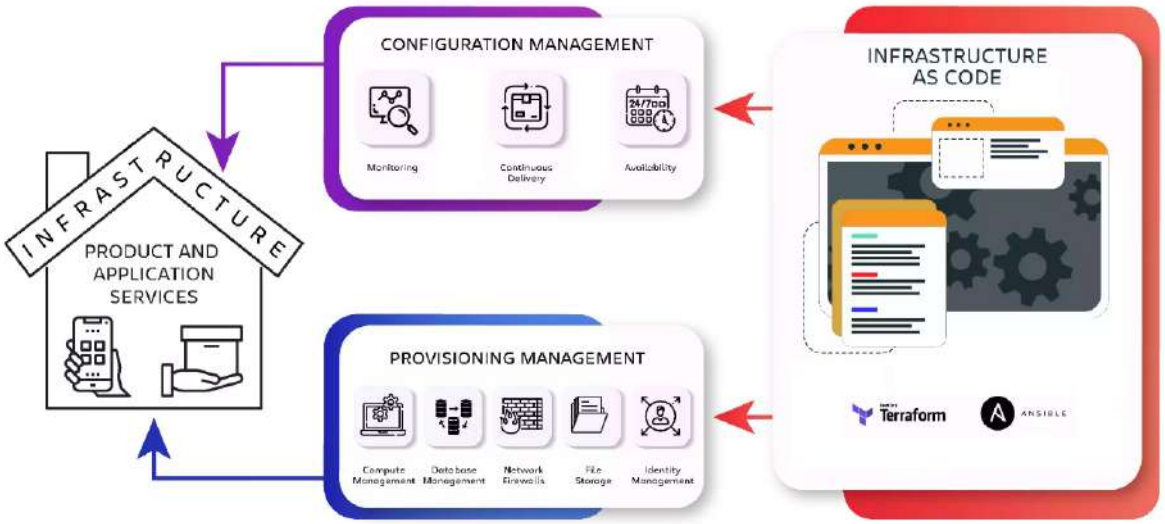
Instituto Federal de Mato Grosso - Campus Cuiabá Cel. Octayde Jorge da Silva

Edital 014/2021

Execução entre 01/03/2022 - 31/11/2022

Instituto Federal de Mato Grosso

Proposta de Desenvolvimento de Solução para Automação de Entregas Contínuas de Softwares da Secretaria de Segurança Pública do Estado de Mato Grosso



Pesquisadores:
Adriano Neres Araújo Souza
Evandro César Freiberger
João Paulo Delgado Preti
Leonardo Luiz Braun
Tiago de Almeida Lacerda

Membros Integrantes da SESP:
Alison Sacal Ferreira de Lima
Bruno de Oliveira Figueiredo
Robson Alexandre Rodrigues
André Santana Machado

Gestão Operacional da SESP:
Walmir Akihiro Oribe
Diana Maria de Lima
Daniel Rios Lima Amaral

Instituto Federal de Mato Grosso

Dados Internacionais de Catalogação na Publicação (CIP) Câmara Brasileira do Livro, MT, Brasil

BRAUN, Leonardo Luiz; FREIBERGER, Evandro César; LACERDA, Tiago A.; PRETI, João Paulo D.; SOUZA, Adriano N. A.;

Proposta de Desenvolvimento de Solução para Automação de Entregas Contínuas de Softwares da Secretaria de Segurança Pública do Estado de Mato Grosso.

Cuiabá-MT: Instituto Federal de Mato GrossoLtda., 2021.

1. Programas de computador. 2. Arquitetura de Software. 3. Sistemas Distribuídos. 4. Microsserviços.

Lista de ilustrações

Lista de tabelas

Sumário

CAPÍTULO 1

Infraestrutura como Código

Neste capítulo será apresentado os conceitos relacionados à Infraestrutura como Código, níveis de maturidade, ...

1.1 O que é?

A IaC (infraestrutura como código) é uma prática DevOps em que o gerenciamento da infraestrutura (redes, máquinas virtuais, balanceadores de carga e topologia de conexão) é realizado por meio de um modelo descritivo, usando um sistema de controle de versão para o código-fonte. IaC permite tratar infraestrutura como software, seguindo o princípio de que o mesmo código-fonte gera o mesmo binário, permitindo gerar o mesmo ambiente toda vez que é aplicado.

Essa nova abordagem de representação da infraestrutura leva ao tratamento do código de forma similar ao que é feito pela equipe desenvolvimento (Figura 3), participando de um processo para codificação, build, teste e implantação.

Figura 1 – Processo de desenvolvimento da infraestrutura como código

Um dos princípios da Infraestrutura como Código (IaC) é a idempotência. Idempotência é a propriedade de que um comando de implantação sempre define o ambiente de destino na mesma configuração, independentemente do estado inicial do ambiente. A idempotência é alcançada configurando automaticamente um alvo existente ou descartando o alvo existente e recriando um ambiente novo.

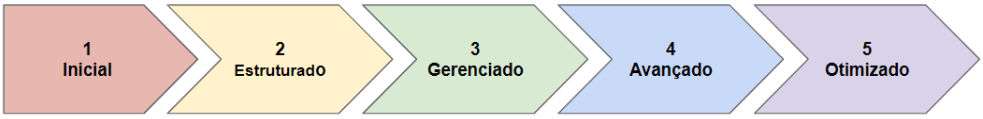
1.2 Níveis de Maturidade

A transformação da cultura da equipe e dessa forma de repensar a infraestrutura ocorre por meio de uma jornada em que vai se elevando a maturidade sobre as práticas necessárias para a implantação de IaC em uma organização.

Não há um consenso referente aos níveis de maturidade, existem classificações utilizando 4,5,6 níveis, com terminologias distintas e por isso organizamos uma classificação própria levando em consideração alguns desses estudos para nortear o planejamento dessa jornada.

Como podemos observar na Figura 2 temos 5 níveis de maturidade:

Figura 2 – Níveis de Maturidade utilizados no projeto



Com relação as práticas necessárias, organizamos em 4 áreas:

1. **Desenvolvimento em IaC**: envolve práticas referentes a especificação e codificação da infraestrutura;
2. **Integração contínua**: práticas de integração e teste da infraestrutura;
3. **Provisionamento e configuração**: uso de IaC para provisionamento e configuração da infraestrutura;
4. **Gerenciamento**: gerenciamento da infraestrutura em funcionamento.

1.2.1 Nível 1 - Inicial

Neste nível os processos não são repetíveis, tem-se pouco controle e as ações são reativas.

Desenvolvimento em IaC	Integração contínua	Provisionamento e configuração	Gerenciamento
- Nada é armazenado em um Sistema de Controle de Versionamento (SCV) - Os scripts são armazenados na infraestrutura, estações de trabalho locais ou ainda em notas	- Não existe servidor de Integração Contínua (IC) - Testes de IaC não são escritos (functional, performance, compliance) - Não é possível testar a infraestrutura antes do provisionamento	- A infraestrutura é manualmente construída em linha de comando ou em interface gráfica (GUI em inglês) - Infraestrutura não pode ser facilmente reconstruída - Provisionar nova infraestrutura é sofrível e inconsistente	- A infraestrutura existente é frágil e não confiável - Correção e atualizações são feitas diretamente na infraestrutura em funcionamento - Troubleshooting é executado diretamente na infraestrutura em funcionamento

1.2.2 Nível 2 - Estruturado

Os processos estão definidos, mas as ações continuam sendo reativas.

Desenvolvimento em IaC	Integração contínua	Provisionamento e configuração	Gerenciamento
- A infraestrutura é parcialmente automatizada utilizando scripts - Nem todo código é checado no SCV - A automação não depende de uma ferramenta de mercado	- Existem alguns testes de IaC - Os testes de IaC são rodados apenas localmente	- Provisionamento é realizado por meio de scripts mas executado para fins específicos (Ad hoc)	- Correções e atualizações são feitas por meio de processo de provisionamento

Desenvolvimento em IaC	Integração contínua	Provisionamento e configuração	Gerenciamento
- Toda infraestrutura está definida como código - Toda IaC está versionada no SCV - Ferramenta de mercado é utilizada para escrever códigos declarativos	- Existe servidor de IaC para fazer pull, build, test and publish dos artefatos de IaC - Testes automatizados são executados para cada check-in - Testes automatizados são executados em ambiente de produção	- O provisionamento da infraestrutura é resultado de uma pipeline de entrega automatizada - O provisionamento é idempotente	- Infraestrutura imutável (Sem conexão SSH) - Infraestrutura é confiável e a performance é previsível

1.2.3 Nível 3 - Gerenciado

Os processos estão definidos e as ações são pró-ativas.

1.2.4 Nível 4 - Avançado

Os processos são mensurados e controlados.

Desenvolvimento em IaC	Integração contínua	Provisionamento e configuração	Gerenciamento
- Todas as alterações são monitoradas em uma ferramenta de Gerenciamento do Ciclo de Vida da Aplicação (ALM m inglês) - Todos os defeitos e bugs são monitorados na ALM	- Builds não são deixadas quebradas - Mudanças são sempre enviadas para produção por um caminho consistente	- Habilidade de realizar o rollback manual rápido e seguro	- Infraestrutura possui alta disponibilidade e é tolerante a falhas

1.2.5 Nível 5 - Otimizado

Focado na melhoria dos processos.

Desenvolvimento em IaC	Integração contínua	Provisionamento e configuração	Gerenciamento
- Melhoria contínua e otimização da IaC baseada na evolução dos padrões de mercado	- Melhoria contínua dos testes em vários níveis	- Provisionamento ocorre sem queda da infraestrutura (Zero-downtime) - Habilidade de realizar roll back automaticamente - Provisionamento self-service	- Infraestrutura é auto-curável, auto-configurável e auto-otimizável

É importante se identificar e fazer uma reflexão sobre o nível de maturidade em que a organização se encontra para então estabelecer um plano e um projeto de melhoria para que o progresso ocorra

por meio de pequenos incrementos. Querer saltar 2-3 níveis é um risco muito grande, são muitas práticas novas a serem absorvidas pela organização, estabelecer um processo de melhoria contínua por meio de pequenos passos permite uma absorção dessas práticas de forma mais efetiva pelas equipes, potencializando o sucesso do planejamento e de sua execução. Maturidade é um termo adequado, não basta apenas o conhecimento, é preciso também do tempo para consolidar esse conhecimento.

CAPÍTULO 2

GitOps e o Versionamento da IaC

Este capítulo descreve a estrutura de pastas, o versionamento de código e o processo de condução de artefatos de um projeto considerando os padrões de versionamento adotado.

2.1 Introdução

Em muitas referências é considerada uma boa prática separar os repositórios git para o código da aplicação e para a descrição da aplicação ou sistema por diversas razões:

- Os ciclos de vida são diferentes em termos de atualização, versionamento e release: mudanças em um não necessariamente implicam mudanças no outro;
- Aspectos de segurança como role-based access control (RABC) são aplicados de forma diferente;
- CI/CD pipelines para o código da aplicação é diferente da pipeline dos manifestos;
- Modelos de organização por branches também diferem: para o repositório da aplicação é comum o uso de GitFlow, GitHub Flow, GitLab Flow, . . . No repositório da IaC as necessidades são diferentes e relacionadas com a configuração de cada ambiente;
- Quando da existência de diversas aplicações que precisam de uma configuração genérica, fica o impasse de qual aplicação (repositório) receberá a configuração;
- Misturar os logs do Git relacionados a modificações na aplicação das modificações de deployment pode ser confuso e de difícil rastreabilidade.

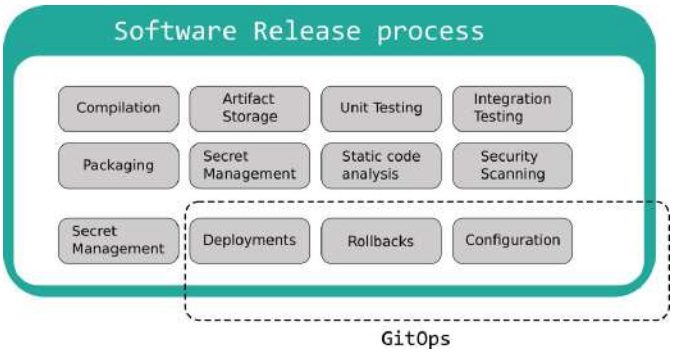
Fica claro que o uso do Git pela equipe Ops difere da equipe Dev, afinal o perfil do profissional e as necessidades são diferentes e complementares. Como os repositórios são distintos: **Code Repo**

(Dev) e **Manifesto Repo** (Ops), a estrutura e a organização dos repositórios pode ser distinta e alinhada as necessidades de cada equipe.

Tendo em vista que teremos repositórios separados, resta a decisão de como organizar o repositório de manifesto para diferentes ambientes e/ou plataformas. Vislumbram-3 possibilidades:

- 1 repositório Git para cada ambiente;
- 1 branch para cada ambiente;
- 1 diretório ou arquivo para cada ambiente.

Figura 3 – Processo de desenvolvimento da infraestrutura como código



A equipe de desenvolvimento pode escolher a estratégia de branches que achar mais adequada e utilizar Git-flow, mas para o **Manifesto Repo não recomendamos o uso de branches por ambiente**, pelos seguintes motivos:

- GitOps cobre uma parte muito específica do ciclo de vida do software;
- A promoção de um ambiente para outro muitas vezes não será um simples merge;
- É comum que nem todas as modificações realizadas em teste devem ser promovidas para a produção;
- Ferramentas como Helm, Kustomize, duas ferramentas muito populares para Kubernetes, organizam os diferentes ambientes segmentando em pastas.

E mesmo assim, diversos aspectos do deploy, como promoção entre ambientes, tratamento de dados sigilosos, teste, não são tratados no paradigma GitOps, ou seja, cabe a equipe definir suas práticas para diversos aspectos da entrega do software.

Como pontos positivos para a organização dos ambientes em pastas, podemos citar:

- Uma única branch para administrar: o que elimina os conflitos de merge;
- Um cluster ou ambiente pode ser criado ou removido simplesmente adicionando ou removendo um diretório;
- Facilidade de mudar de um ambiente para outro, o flow não precisa ser linear;

- Muitas ferramentas para GitOps implementam o conceito de ambientes por pastas.

Como pontos de atenção temos:

- Divergência entre ambientes pode ser mais difícil de detectar (necessidade de ferramenta complementar ao git diff);
- RBAC precisa de uma solução ou processo alternativo para aplicar diferentes permissões entre diretórios (isso pode ser alcançado utilizando plugins como Codeowner);
- Parâmetros comuns a todos os ambientes representam um risco de modificação acidental em ambiente indesejado.

2.2 Pastas por Ambiente

Como organização inicial da estrutura de diretórios para a arquitetura da solução, temos diferentes representações a depender de cada arquitetura:

2.2.1 Arquitetura 1 - Kubernetes, Kustomize e ArgoCD

```
├── base
│   ├── configMap.yaml
│   ├── deployment.yaml
│   ├── kustomization.yaml
│   └── service.yaml
└── overlays
    ├── production
    │   ├── config-map.yaml
    │   ├── deployment.yaml
    │   └── kustomization.yaml
    └── staging
        ├── config-map.yaml
        └── kustomization.yaml
```

2.2.2 Arquitetura 2 - Terraform

A organização dos recursos no terraform está estabelecida na forma de módulos para melhor componentização dos recursos de infraestrutura, permitindo a criação, modificação e destruição desses módulos de forma independente. Ao observar os recursos do *libvirt*, perceba que há um arquivo terraform para cada conjunto de recursos (*domain, network, pool, volume, …*).

Há também na estrutura um espaço reservado para a codificação de testes da infraestrutura por meio da linguagem Go.

2.2.3 Arquitetura 3 - Ansible

As roles do ansible foram organizadas em dois grupos (*instalar_… e configurar_…*) representando de forma independente as etapas de instalação e configuração de uma aplicação. Para cada role temos a seguinte estrutura de diretórios:

```
├── module
│   ├── libvirt                 # KVM no RedHat Enterprise Linux v8.5
│   │   ├── cloud_init.cfg      # instalação de pacotes e criação de usuários
│   │   ├── domain.tf           # criação das VMs
│   │   ├── main.tf             # configuração do provider libvirt
│   │   ├── network_config.cfg  # configuração padrão para a interface de rede
│   │   ├── network.tf          # criação da rede para cada VM
│   │   ├── output.tf           # expõe informação para os arquivos de configuração
│   │   ├── pool.tf             # construção do storage
│   │   ├── template.tf
│   │   ├── variables.tf        # variáveis utilizadas nos arquivos de configuração
│   │   └── volume.tf           # criação dos volumes de armazenamento
├── tests
│   ├── defaults
│   │   ├── defaults.tf         # configuração do teste, localização dos recursos
│   ├── go.mod                  # dependência necessária
│   ├── go.sum                  # dependência necessária
│   ├── infra_test.go           # script de teste
├── README.md
├── main.tf
├── parameters.tf
├── terraform.tfvars
├── variables.tf
```

```
├── environments
│   ├── prod
│   │   ├── hosts       # Inventário dos arquivos de produção
│   ├── release
│   │   ├── hosts       # Inventário dos arquivos de release
│   ├── shared
│   │   ├── hosts       # Inventário dos arquivos comuns aos dois ambientes
├── roles
│   ├── instalar_...    # role de instalação de um determinado app
│   │   ├── defaults
│   │   ├── files
│   │   ├── handlers
│   │   ├── meta
│   │   ├── tasks
│   │   ├── tests
│   │   ├── vars
│   ├── configurar_...  # role de configuração de um determinado app
│   │   ├── defaults
│   │   ├── files
│   │   ├── handlers
│   │   ├── meta
│   │   ├── tasks
│   │   ├── tests
│   │   ├── vars
├── gitlab.yml          # playbook de instalação do gitlab que é comum aos dois ambientes
├── test.yml            # teste para verificação se o app foi instalado e está ativo
├── tools.yml           # playbook de instalação e configuração dos apps
├── validate.yml        # teste para validar a configuração do app
```

defaults: diretório utilizado para definir os valores *default* das variáveis. As variáveis em *default* tem a menor prioridade, sendo facilmente sobrescritas. Se a definição da variável não ocorrer em nenhum outro local, a variável em *defaults/main.yml* será utilizada.

files: diretório utilizado para armazenar arquivos que são necessários para o provisionamento de máquinas, sem alteração. É comum utilizar a task *copy* para referenciar esses arquivos. O interessante é que o Ansible não exige um caminho para recursos que estão armazenados no diretório *files*.

handlers: diretório utilizado para armazenar *handlers* do Ansible, estes representam tarefas (*tasks*) que podem ser marcadas para serem executadas ao término de um *play* do Ansible.

meta: diretório utilizado para armazenar informações de autoria, útil para publicar a *role* em

galaxy.ansible.com. Os metadados de uma *role* do Ansible consiste de autor, plataformas suportadas e dependências.

tasks: este é o diretório onde escrevemos a maior parte das nossas *roles* que incluem todas as *tasks* que a role irá executar. Escrevemos cada conjunto de *tasks* em arquivos separados e o incluimos no arquivo *main.yml* do diretório.

templates: diretório similar a *files*, utilizado para armazenar arquivos que representam modelos de provisionamento, mas neste, os arquivos suportam alteração. A linguagem Jinja2 é utilizada para codificar essas alterações.

tests: diretório utilizado para armazenar testes automatizados sobre as nossas *roles*.

vars: neste diretório criamos os arquivos contendo as variáveis necessárias para as nossas roles. As variávies aqui servem apenas para uso interno das *roles*. É uma boa prática utilizar namespace nos nomes das variáveis para prevenir conflitos de nomenclatura.

2.3 Padrão de Commit

Este documento contém boas práticas para realização de commit nos projetos da SESP para a equipe Ops. As recomendações aqui contidas são uma adaptação da especificação do Conventional Commits e manter este nível de organização e coordenação no versionamento do código ajuda a:

- Melhorar a legibilidade do histórico de versionamento do Git;
- Aumentar a velocidade na hora de procurar por mudanças específicas no código;
- Entender, facilmente, quais mudanças estão sendo publicadas dentro de um pipeline;
- Possibilitar a geração de um CHANGELOG ou de release notes de maneira totalmente automatizada;
- Incentivar os membros da equipe a realizarem commits de maneira pensada e específica, sem realizar commits arbitrários e de difícil interpretação;
- Possibilita a utilização de ferramentas de automação de commits.

Este documento define um conjunto de regras para criar um histórico de commit explícito, o que facilita a criação de ferramentas automatizadas. Esta convenção segue o documento, que descreve os recursos, correções e modificações que quebram a compatibilidade nas mensagens de commit.

Pendencia link não está funcionando

2.3.1 Estrutura

A mensagem do commit deve ser estruturada da seguinte forma:

O commit contém os seguintes elementos estruturais, para comunicar a intenção ao utilizador da sua biblioteca.

```
<tipo>[escopo opcional]: <descrição>

[corpo opcional]

[rodapé opcional]

```

2.3.1.1 Tipos

Os tipos são a descrição inicial de o que o commit está realizando, sendo obrigatórios e devendo seguir um padrão bem definido.

fix: um commit do tipo fix soluciona um problema na sua base de código (isso se correlaciona com PATCH do versionamento semântico).

feat: um commit do tipo feat inclui um novo recurso na sua base de código (isso se correlaciona com MINOR do versionamento semântico).

BREAKING CHANGE: um commit que contém o texto BREAKING CHANGE:, no começo do texto do corpo opcional ou do rodapé opcional, inclui uma modificação que quebra a compatibilidade com versões anteriores (isso se correlaciona com MAJOR do versionamento semântico). Uma BREAKING CHANGE pode fazer parte de commits de qualquer tipo.

Outros: tipos adicionais são permitidos além de fix: e feat:, recomenda-se draft:, chore:, docs:, style:, refactor:, perf:, test:, entre outros.

Também recomendamos *improvement* para commits que melhoram uma codificação atual sem adicionar um novo recurso ou consertar um bug. Observe que esses tipos não são obrigatórios pela especificação do Conventional Commits e não têm efeito implícito no versionamento semântico (a menos que incluam uma BREAKING CHANGE). Um escopo pode ser adicionado ao tipo do commit, para fornecer informações contextuais adicionais e está contido entre parênteses, por exemplo *feat(dns): ...*.

2.3.1.2 Escopo

O escopo do *commit* é uma parte opcional, curta e de fácil compreensão. É nela que iremos dizer qual parte do código foi modificada, como indicar que fizemos alterações em uma determinada característica ou recurso.

2.3.1.3 Descrição

A descrição, juntamente com o tipo, é uma das partes mais importantes do padrão: é aqui que deve ser descrito, de maneira clara, sucinta e simplificada, o que foi realizado no *commit*. É recomendado que essa parte tenha, no máximo, 70 caracteres, para que não se estenda muito.

2.3.1.4 Corpo

O corpo do *commit* é também opcional. Nele, pode-se realizar uma descrição mais detalhada do commit, indicar razões para a realização dele e consequências que ele pode vir a causar, além de alguma outra observação que seja pertinente.

2.3.1.5 Rodapé

O rodapé, assim como o corpo, é opcional e informativo. Ele pode ser usado como uma finalização do *commit*, informando o encerramento de uma issue ou o pertencimento/associação a uma *task* também.

2.3.2 Granularidade dos Commits

A granularidade do conteúdo dos *commits* é um fator que deve seguir um padrão. A granularidade também interfere na rastreabilidade do histórico de mudanças e na colaboração entre os membros da equipe.

A recomendação mais aceita para a granularidade é que cada *commit* seja atômico, considerando o escopo de uma única funcionalidade. Assim, cada commit só pode conter incremento ou correções de uma funcionalidade. Essa diretriz desconsidera o número de arquivos envolvidos, ou seja, não importa quantos arquivos foram afetados, desde que todos representem uma única funcionalidade ou correção.

Benefícios:

- Fácil de reverter sem afetar outras mudanças
- Fácil de fazer outras alterações na hora
- Fácil de mesclar recursos com outros ramos
- Fácil de rastrear com as *Task* em um sistema de *Issue Tracking*

2.3.3 Estratégias de Alteração de Commits

Apesar do objetivo ser de fazer *commits* consistentes, é inevitável situações onde haja a necessidade de alterações. Contudo, é importante estabelecer um padrão de alterações, garantindo também uma boa rastreabilidade das alterações no histórico.

O comando *git commit –amend* é uma forma conveniente de modificar o *commit* mais recente. Ele permite combinar alterações na área de *staging* com o *commit* anterior em vez de criar um novo *commit*.

2.3.3.1 Alterações da mensagem de commits

Algo indesejável, porém não tão incomum, são os erros cometidos na mensagem do *commit*, quer seja um texto que ficou incompleto ou na tipificação do *commit*.

git commit –amend -m "nova mensagem do commit"

Executar este comando quando não existe nada na área de staging permite que você edite a mensagem do último *commit*, sem criar uma nova instância de *commit*.

2.3.3.2 Alterações do conteúdo de commits

Existem situações em que um determinado conjunto de arquivos foram alterados e confirmados em um *commit*. Contudo, logo após o *commit*, verifica-se que uma alteração necessária em um ou mais arquivos não foi realizada e não deseja-se criar um novo *commit*. Para corrigir o erro, basta fazer o *staging* dos arquivos alterados e confirmar com o sinalizador *–amend*.

git commit –amend –no-edit

O sinalizador *–no-edit* vai permitir fazer a correção no *commit* sem alterar a mensagem de confirmação. O *commit* resultante vai substituir o *commit* incompleto e vai parecer que as alterações foram confirmadas em um único *commit*.

Deve-se levar em consideração que outras pessoas podem estar atuando em uma mesma *branch*, portanto, evite ao máximo corrigir commits em *branches* compartilhadas e, quando o fizer, faça o mais rápido possível.

2.3.3.3 Considerações sobre alterações de commits

Apesar de existir a possibilidade de alteração em *commits* mais antigos ou alterações envolvendo vários *commits* ao mesmo tempo, por meio do comando *git rebase*, o padrão sugerido neste documento não encoraja o uso de tal comando, visto que, poderá provocar perdas do histórico e consequentemente dificuldade de rastreabilidade ou retorno a versões antigas.

2.4 Ambiente de Desenvolvimento

O ambiente de desenvolvimento é composto pelas ferramentas e versões abaixo:

- Git 2.35.1: controle do versionamento do código IaC;
- Terraform 1.1.7: código de provisionamento e configuração da infraestrutura;
- Ansible 2.12.2 com python 3.9: código para instalação e configuração de serviços e aplicações;
- GNUPG 2.3.3: para geração de chaves privadas e públicas;
- Visual Studio Code 1.62.3: editor de código.

Para uma construção rápida do ambiente de desenvolvimento disponibilizamos na raiz do repositório o arquivo *shell.nix* com o script para inicialização de um *shell* com as ferramentas necessárias.

Para a execução do script faz-se necessário a instalação do Nix Package Manager.

Com o *nix* instalado basta executar o comando *nix-shell* na raiz do repositório, que instalará as ferramentas e inicializará o Visual Studio Code acessível por URL que será apresentada.

```
{ pkgs ? import <nixpkgs> {}
}:
pkgs.mkShell {
  name="ops-environment";
  buildInputs = [
    pkgs.git
    pkgs.micro
    pkgs.bat
    pkgs.openvscode-server
    pkgs.terraform
    pkgs.gnupg
    pkgs.ansible
  ];
  shellHook = ''
    echo "Ambiente pronto para GitOps ..."
    echo "Iniciando VS Code"
    openvscode-server --folder=.
  '';
}
```

Fontes:

Standard Module Structure — Terraform by HashiCorp
Best Practices — Ansible Documentation
Ansible driven GitOps for OpenShift
GitHub - hseligson1/kustomize-gitops-example: Applied GitOps with Kustomize
Stop Using Branches for Deploying to Different GitOps Environments
Promoting changes and releases with GitOps

CAPÍTULO 3

Protótipo Funcional

O protótipo funcional é composto por um conjunto de softwares que serão utilizados como suporte para demonstrar o provisionamento e a configuração de contêineres e máquinas virtuais com IaC. Seus requisitos são comuns em diversas aplicações e foram planejados de forma a demonstrar os recursos de infraestrutura, tanto em relação ao acesso às ferramentas, quanto em relação à própria esteira de deploy.

Este protótipo faz uso de ferramentas dentro e fora do cluster do Kubernetes e será explicado em detalhes a seguir.

3.1 Registro Geral

É uma aplicação no estilo de microserviço que tem por objetivo representar um ponto de acesso para consulta aos dados básicos de qualquer pessoa física do Estado. Trata-se de uma aplicação selecionada do projeto de Microserviços do Edital 004/2020 - FAPEMAT que utiliza diferentes serviços de base que servem para o propósito de demonstração do provisionamento e configuração de containers no cluster e máquinas virtuais, bem como outros serviços de base necessários para diversas aplicações do ecossistema.

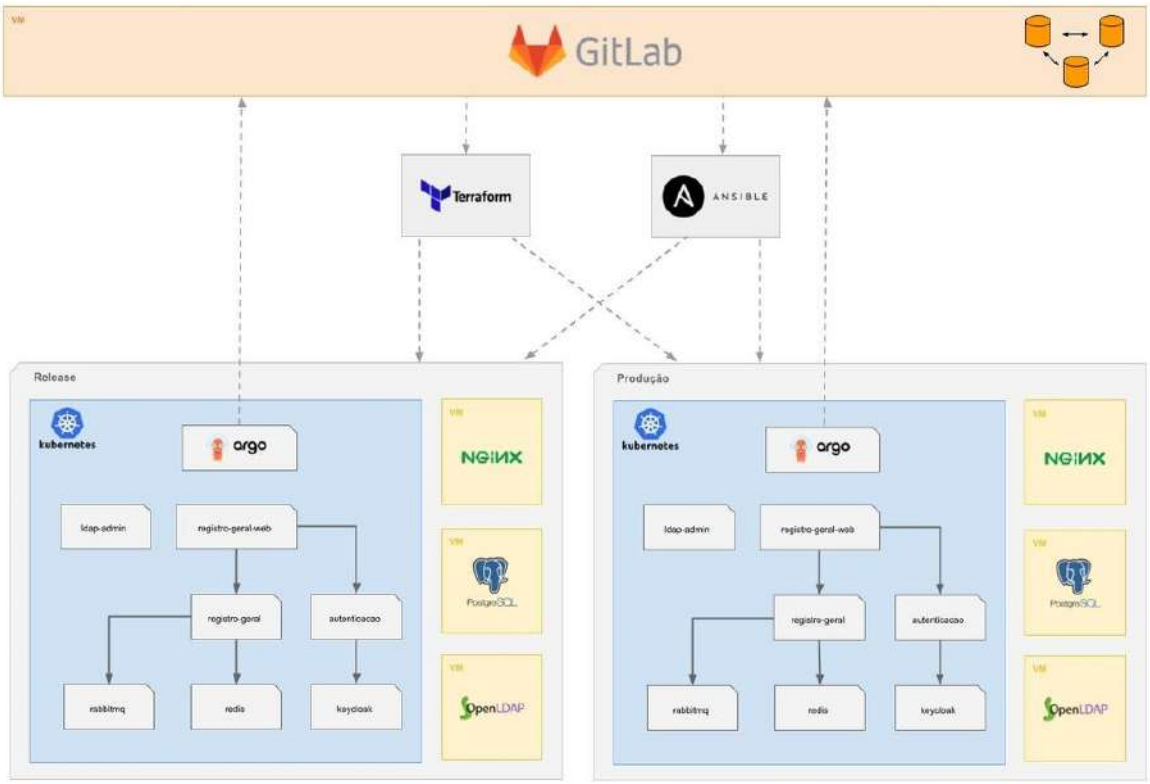
3.2 Infraestrutura

Os serviços e ferramentas do protótipo funcional estão distribuídos entre o cluster do Kubernetes e quatro máquinas virtuais. A Figura 4 ilustra essa distribuição.

Pode-se observar o Kubernetes e as aplicações conteinerizadas, bem como 3 máquinas virtuais, sendo que essa organização é espelhada para dois ambientes:

1. **Release**: responsável por conter os artefatos em execução que estão sendo avaliados para serem promovidos para o ambiente de produção;
2. **Produção**: contêm os artefatos que já foram testados, aprovados, que estão acessíveis para o usuário final e que representam os serviços em operação da organização.

Figura 4 – Serviços e ferramentas do protótipo funcional



Temos mais uma máquina virtual para o Gitlab, este contêm o repositório comum aos dois ambientes, bem como a pipelina CI/CD.

Nas próximas seções apresentamos as aplicações que compoem o protótipo funcional, começando pelo Kubernetes.

3.2.1 Ferramentas e serviços publicados no Kubernetes

Argo CD: Ferramenta de monitoramento contínuo para Kubernetes que compara o estado atual das aplicações com o estado desejado versionado no GitLab.

LDAP Admin: Cliente de acesso ao OpenLDAP.

Registro Web: Camada Frontend do Registro Geral. É um SPA que acessa diretamente o serviço registro-geral e autenticacao.

Registro Geral: Camada Backend do Registro Geral. É uma aplicação desenvolvida em Spring Boot, que compreende as regras negociais referentes ao Registro Geral e acessa diretamente as ferramentas RabbitMQ, Redis e Keycloak.

RabbitMQ: Ferramenta que representa o broker de mensageria. Utilizado para envio e consumo de mensagens assíncronas.

Redis: Ferramenta de caching, utilizada para manutenção de dados não persistentes de forma distribuída performática.

Keycloak: Servidor de identidade e acessos. Centraliza e gerencia as autenticações e autorizações.

3.2.2 Ferramentas publicadas em máquinas virtuais

GitLab: Gerenciador de repositórios com suporte à diversas funcionalidades, como gerenciamento de tarefas e CI/CD. É o ponto central para manutenção dos arquivos relacionados à IaC e código fonte das aplicações.

Nginx: Serviço de load balancing, web server e proxy reverso. É utilizado nesta arquitetura como centralizador de acessos para as ferramentas externas ao cluster do Kubernetes.

PostgreSQL: Banco de dados relacional. Armazena os dados do Registro Geral e do Keycloak.

OpenLDAP: Implementação livre do protocolo LDAP. Centraliza e controla o acesso à diretórios.

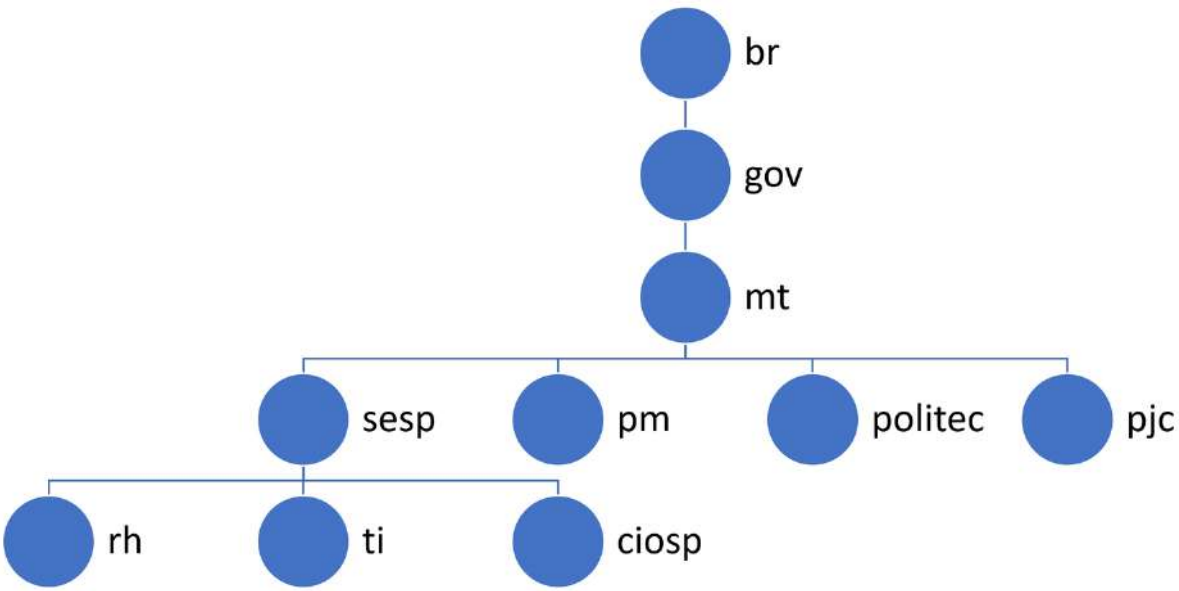
3.3 Autenticação e autorização

A autenticação e autorização compreendem uma grande parte do protótipo funcional, por isso é importante entender os serviços e ferramentas que dão suporte à esta camada.

3.3.1 LDAP

A organização dos usuários no LDAP está representada na árvore ilustrada na figura 5.

Figura 5 – Árvore de usuários no LDAP



Essa organização da árvore com usuários ancorados em diferentes níveis foi intencional para ilustrar no keycloak como realizar a configuração para busca desses usuários nesses diferentes níveis.

A organização da árvore segue o padrão DNS sendo o login por Common Name (CN) ou também chamado de Fully Qualified Domain Name (FQDN). O Domain Component (DC) ficou estabelecido conforme:

dc=mt,dc=gov,dc=br

Os demais ramos da árvore ficaram representados como Organizational Units (OU) aonde os usuário são inseridos.

3.3.2 Keycloak

O Keycloak é um servidor de identidade e acessos, sendo uma ferramenta de código aberto mantida pela Red Hat.

Através dele é possível manter o logon único (SSO), facilitando a integração com diversas fontes de autenticação e autorização, como LDAP, logins sociais ou estratégias personalizadas de autenticação.

Pelo Keycloak também é possível visualizar e gerenciar as sessões, alterar o tempo de vida dos tokens e estabelecer regras de acessos por grupos de usuários. Estes dados podem ser consumidos pelos serviços por meio de uma API, e nesta arquitetura essa tarefa é executada pelo serviço de autenticação, detalhado a seguir.

3.3.3 Serviço de autenticação

Temos na estrutura do protótipo funcional outra aplicação/serviço intitulada *autenticacao* com o papel de *Wrapper* para os serviços do Keycloak. Essa abordagem traz os seguintes benefícios:

- Reduz a exposição do Keycloak para clientes externos, permitindo limitar a exposição de alguns dados sensíveis de configuração do Keycloak, como o CLIENT ID;
- Permite centralizar configurações e modificações relacionadas a autenticação no microserviço, reduzindo o repasse dessas modificações para os clientes;
- Permite resolver problemas de emissor do token, quando o keycloak é acessado por uma URL diferente da sua rede interna.

O projeto encontra-se em Conectar para visualizar para referência.

3.4 Mensageria

O objetivo da **mensageria** é permitir que cada aplicação trate, troque mensagens e colabore com outras aplicações de forma síncrona ou assíncrona. Essa troca de mensagens é uma alternativa ao tradicional modelo de comunicação web utilizando protocolo HTTP, sendo **RabbitMQ** a ferramenta selecionada para compor o conjunto de tecnologias a serem implantadas e necessária para a correta comunicação entre as aplicações Registro Geral e Autenticação.

3.5 Caching

Em um ambiente composto por diversas aplicações ou sistemas organizados segundo um estilo arquitetural distribuído, é comum o uso de algum nível de cache para melhorar o tempo de resposta, ajudar na disponibilidade ou escalabilidade. Nesse contexto **Redis** é outra ferramenta selecionada para compor o protótipo funcional.

As justificativas para a escolha dessas ferramentas que compoem o protótipo funcional foram documentadas no projeto do Edital FAPEMAT 004/2020 do projeto de microsserviços.

CAPÍTULO 4

Preparação do Ambiente

Este documento tem como objetivos descrever o ambiente utilizado para criação e hospedagem dos serviços, artefatos e protótipo funcional do projeto, e as configurações realizadas para preparação do ambiente.

4.1 Hardware do Servidor

O ambiente dispõe de servidor físico utilizado como host para provisionamento de máquinas virtuais necessárias para execução das ferramentas e serviços utilizados no projeto. O servidor tem a seguinte configuração:

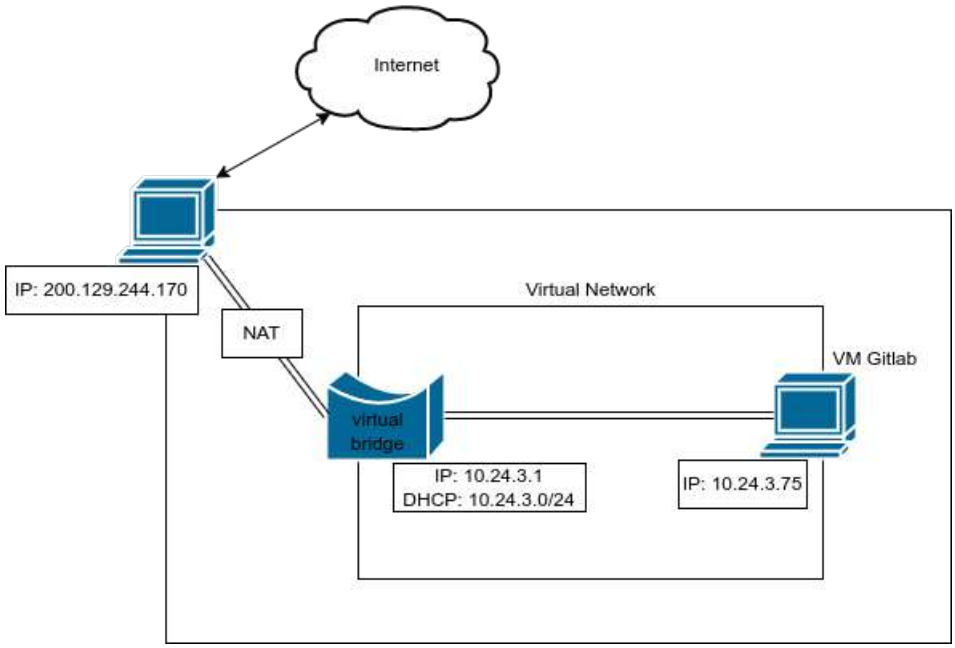
Modelo	Dell Inc. PowerEdge R540
CPU	20x Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
Memória	32 GB
Armazenamento	SSD 600 GB
IP	200.129.244.170

4.2 Arquitetura do ambiente

O ambiente possui o Red Hat Enterprise Linux 8.5 instalado e com recursos de virtualização habilitados. Por questões de disponibilidade de IP, utilizamos redes virtuais (Virtual Networks) com NAT para prover acesso à internet para as máquinas virtuais e também para expor os serviços configurados nessas máquinas. A Figura 6 apresenta o exemplo da máquina virtual do Gitlab configurada na rede virtual (virbr4) 10.24.3.0/24:

4.170) para que as máquinas virtuais navegassem na internet e também realizar NAT para acesso aos serviços instalados nessas máquinas virtuais, como é o caso da porta 80 do NGINX que

Figura 6 – Topologia de rede: exemplo Gitlab



foi exposta na internet para fazer proxy reverso para os demais serviços. Nesse modelo foi possível utilizar um único endereço IP (200.129.24

4.3 Instalação do Red Hat Enterprise Linux 8.5

A instalação do Red Hat Enterprise Linux utilizada foi a padrão e pode ser conferida em Execução de uma instalação padrão RHEL Red Hat Enterprise Linux 8 — Red Hat Customer Portal

Download do PDF

4.3.1 Ativação da Console Web

O Red Hat Enterprise Linux possibilita a ativação de console Web para gerenciamento do Linux via navegador, além de uma interface para gerenciamento dos ambiente de virtualização. A ativação da console é simples e pode ser realizada conforme descrito em Chapter 1. Getting started using the RHEL web console Red Hat Enterprise Linux 8 — Red Hat Customer Portal.

A console do servidor da Fábrica de Software está disponível em https://200.129.244.170:9090/

4.3.2 Instalação e Configuração do KVM e libvirt

Para utilizar os recursos de virtualização e da API libvirt que o Terraform irá utilizar, são necessárias instalações e configurações no Red Hat Enterprise Linux conforme descrito em Chapter 2. Getting started with virtualization Red Hat Enterprise Linux 8 — Red Hat Customer Portal.

O usuário que foi criado no Red Hat Enterprise Linux e será utilizado no Terraform para se conectar e rodar os comando deverá ser adicionado ao grupo do libvirt *sudo usermod -a -G libvirt $(whoami)*.

4.3.3 Configuração de Acesso SSH

O provider libvirt do Terraform utiliza SSH para se concetar ao servidor e a API do libvirt e essa configuração é declarada nos arquivos do Terraform:

```
provider "libvirt" {
  uri = "qemu+ssh://esa@200.129.244.170/system"
}
```

Essa configuração não suporta o uso de usuário e senha para conexão toda vez que o Terraform for executado. Para que essa configuração funcione é necessário a utilização de chaves pública e privada para conexão direta entre o host que irá rodar o Terraform e o servidor alvo.

As configurações necessárias para o acesso direto via SSH estão disponíveis em 12.3.2. Geração de pares de chaves SSH Red Hat Enterprise Linux 8 — Red Hat Customer Portal ou 28.4. Configuração do SSH para gerenciamento remoto no console web Red Hat Enterprise Linux 8 — Red Hat Customer Portal.

4.3.4 Firewall

No ambiente é necessário a configuração do firewall para permitir a exposição dos serviços.

4.3.4.1 Mudança da regra de nftalbes para iptables

Para esse projeto utilizamos o iptables para backend do Firewalld do Red Hat Enterprise Linux. Para fazer isso edite o arquivo */etc/firewalld/firewalld.conf* e altere o *FirewallBackend=nftables para FirewallBackend=iptables*.

Se preferir manter o nftables para o backend leia Capítulo 45. Começando com nftables Red Hat Enterprise Linux 8 — Red Hat Customer Portal

4.3.4.2 Regras do iptable

As regras abaixo são exemplos de regras aplicadas às redes que foram criadas durante a execução do projeto e devem ser adaptadas conforme redes virtuais e máquinas virtuais existentes.

Permitir o encaminhamento de pacotes destinados às máquinas virtuais

```
iptables -I FORWARD -o virbr4 -d 10.24.3.75 -j ACCEPT
iptables -I FORWARD -o virbr5 -d 172.16.0.169 -j ACCEPT
iptables -I FORWARD -o virbr3 -d 172.17.2.71 -j ACCEPT
iptables -I FORWARD -o virbr2 -d 172.16.1.226 -j ACCEPT
iptables -I FORWARD -o virbr4 -d 10.24.3.127 -j ACCEPT
iptables -I FORWARD -o virbr4 -d 10.24.3.75 -j ACCEPT
```

4.3.4.3 Exposição dos serviços para internet

```
#Acesso externo ao OpenLDAP
iptables -t nat -A PREROUTING -d 200.129.244.170 -p tcp --dport 389 -j DNAT --to
    10.24.3.192:389

#Acesso externo ao Postgres
```

```
iptables -t nat -A PREROUTING -d 200.129.244.170 -p tcp --dport 54322 -j DNAT --to
    172.17.2.71:5432

#Acesso externo ao NGINX
iptables -t nat -A PREROUTING -d 200.129.244.170 -p tcp --dport 80 -j DNAT --to
    172.16.0.169:80

#Acesso SSH externo ao Ansible
iptables -t nat -I PREROUTING -p tcp --dport 2224 -j DNAT --to 10.24.3.127:22
```

Relação das redes virtuais com iptables e as regras: Quando uma rede virtual for excluída e recriada ela pode ter um nome diferente do nome anterior, por exemplo, virbr1 passou a se chamar virbr4. Nesse caso as regras de firewall precisam ser ajustadas para refletir a nova disposição da rede.

Quando ocorrer problemas com a rede certifique se a máquina virtual:

1. **consegue acessar a rede externa:** ping 8.8.8.8
2. **consegue resolver DNS:** ping terraform.io
3. **consegue navegar na internet:** curl google.com
4. **se as regras de firewall refletem a rede virtual e o IP correto**

4.4 Terraform

No host onde será executado o Terraform é necessário clonar o projeto. Na pasta terraform basta executar os seguintes comandos:

```
## Iniciando projeto e baixar dependencias
terraform init

# Criando os workspaces do projeto
terraform workspace new release
terraform workspace new production

## Alterando para workspace release
terraform workspace select release

# Visualizando o resultado da infraestrutura planejada
terraform plan

# Aplicando o arquivo de configuracao para provisionamento
terraform apply
```

Referências

https://itnext.io/how-to-use-terraform-to-create-a-small-scale-cloud-infrastructure-abf54fabc9dd

CAPÍTULO 5

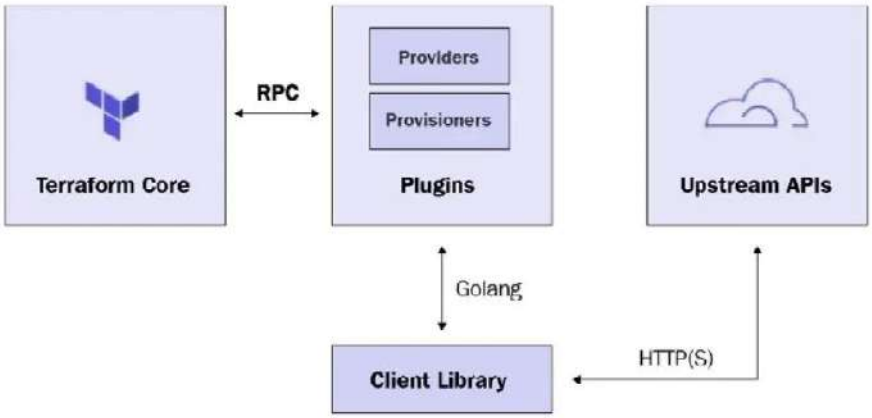
Terraform

Neste capítulo serão apresentadas as configurações do Terraform para a implementação do protótipo funcional.

5.1 Introdução

Terraform é uma ferramenta que permite entregar infraestrutura por meio de código declarativo. Sua arquitetura é baseada em plugins. Conforme pode ser observado na Figura 12, há dois componentes principais: Terraform Core e Terraform Plugins. Terraform Core utiliza chamadas RPC para se comunicar com os plugins e também oferece mecanismos para a descoberta e a utilização desses plugins.

Figura 7 – Arquitetura do Terraform



Os plugins do Terraform são escritos na linguagem Go. Cada plugin expõe uma implementação para um serviço específico, um **provider**, como AWS, ou um **provisioner**, como Bash. Tanto providers quanto provisioners são plugins executados como processos separados e definidos no arquivo de configuração do Terraform.

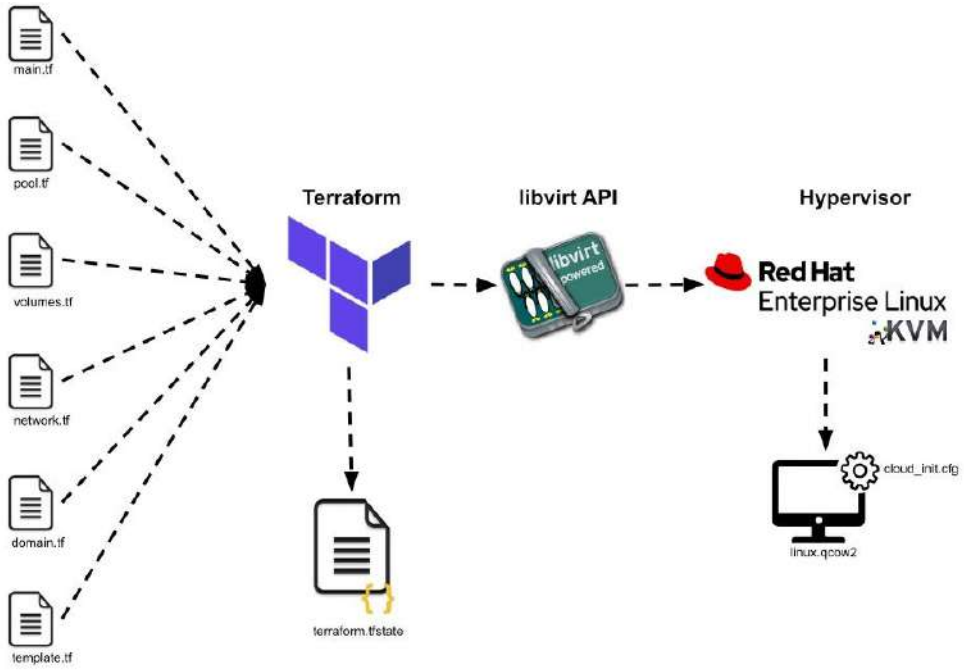
Cada **provider** tem por responsabilidade:

- Inicializar as bibliotecas necessárias para realizar as chamadas de API;
- Autenticar com o provedor da infraestrutura;
- A definição de recursos (resources) que devem estar mapeados com serviços específicos do provedor de infra.

Já cada provisioner tem por responsabilidade executar comandos ou scripts em um determinado recurso.

Na Figura 8 podemos visualizar o provider utilizado no protótipo funcional.

Figura 8 – Terraform libvirt provider



O conjunto de recursos que podem ser utilizados em conjunto para o provisionamento e configuração da infraestrutura são organizados em **módulos**. Um módulo (*module*) representa um conjunto de recursos (resource) que são utilizados em conjunto. Cada configuração do Terraform está relacionado a pelo menos um módulo, que consiste de recursos definidos nos arquivos *.tf*.

Módulos podem ser referenciados e invocados múltiplas vezes, com uma ou diferentes configurações, o que permite empacotar e reutilizar as configurações dos recursos.

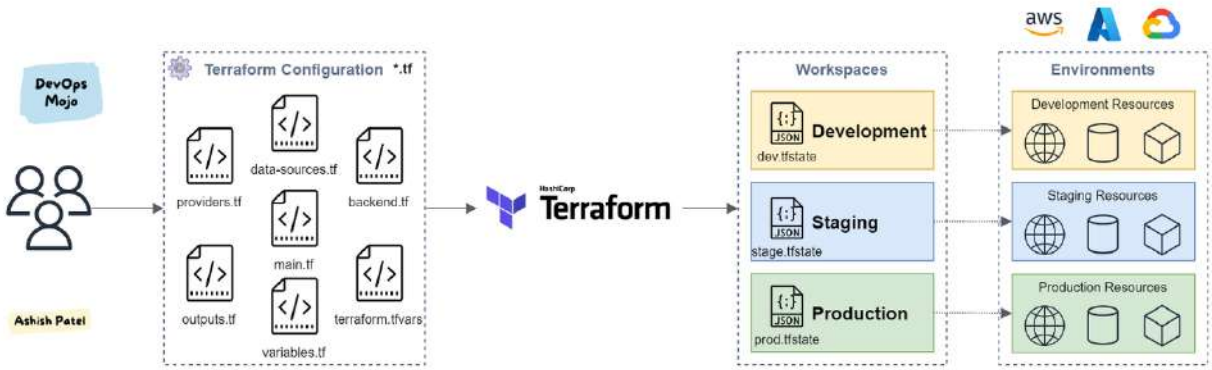
O Terraform pode carregar módulos de repositórios públicos ou privados, o Terraform Registry hospeda uma ampla coleção de módulos públicos para configuração de diversos tipos de infraestrutura. Esses módulos são baixados automaticamente se especificados de forma apropriada (*source* e *version*).

Outro recurso interessante implementado pelo Terraform é o conceito de **Workspace**, ilustrado na Figura 9. Cada configuração do Terraform tem um backend associado que define como as operações são executadas e onde dados como o Terraform state estão armazenados.

Inicialmente o backend tem somente um workspace, nomeado default, e portanto apenas um arquivo de estado associado à configuração. No entanto, Terraform suporta múltiplos workspaces, permitindo manter arquivos de estados diferentes para uma mesma configuração de infraestrutura,

o que nos permite, por exemplo, utilizar uma mesma configuração de infraestrutura para diferentes ambientes (homologação, produção, etc) facilitando a gestão dos recursos.

Figura 9 – Terraform Workspace



5.2 Protótipo Funcional

De acordo com o protótipo funcional o terraform será utilizado para provisionar máquinas virtuais e recursos do Kubernetes. Na pasta **terraform** temos os seguintes arquivos:

```
## Raiz do projeto
main.tf
variables.tf
terraform.tfvars
parameters.tf
backend.tf

## Arquivos do modulo Libvirt (terraform/module/libvirt)
pool.tf
volume.tf
template.tf
network.tf
output.tf

## Arquivos de testes (terraform/tests)
infra_test.go
defaults/defaults.tf
```

Segue explanação:

main.tf: definição dos módulos (plataformas) que o terraform irá provisionar/configurar a infraestrutura;

variables.tf: declaração das variáveis utilizadas na codificação IaC do terraform;

terraform.tfvars: atribuição de valores das variáveis;

parameters.tfvars: valores utilizados pelo módulo do Libvirt para os Workspaces do projeto;

backend.tf: configuração do repositório de estado remoto;

pool.tf: definição do Storage Pool para volumes de armazenamento;

volume.tfvars: definição da imagem padrão e dos volumes de armazenamento das máquinas virtuais;

template.tf: definição dos templates a serem utilizados para configuração de inicialização dos sistemas operacionais;

network.tf: descreve os recursos de rede que devem ser provisionados;

cloud_init.cfg: definição das configurações e definições iniciais do sistema operacional, como hostname, DNS, usuário e chave pública para acesso via SSH;

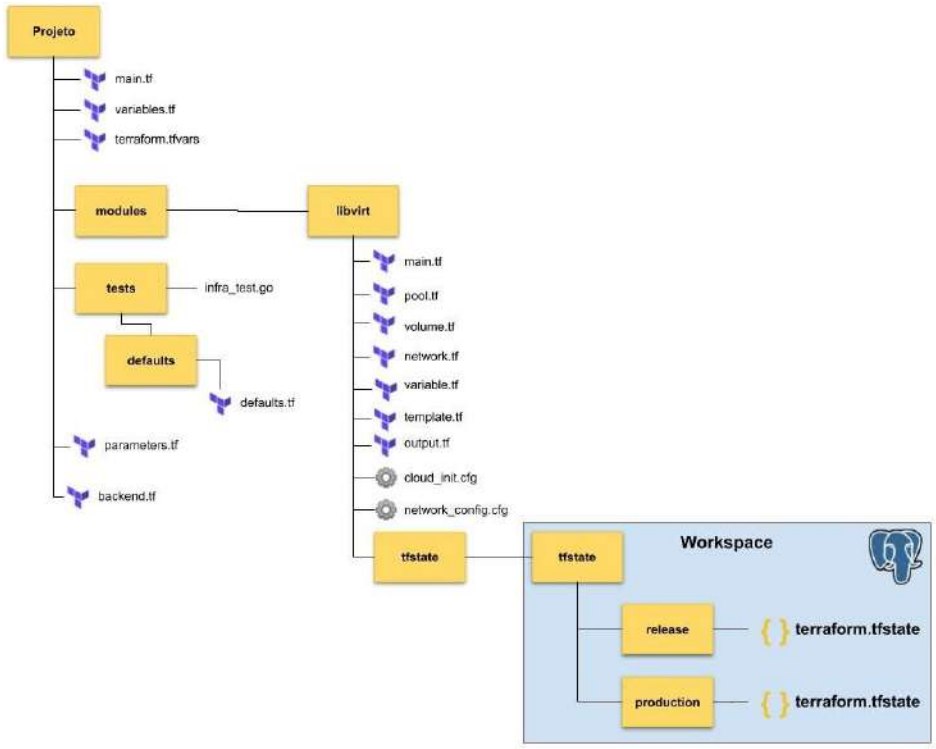
network_config.cfg: definição das configurações de IP dos sistemas operacionais;

infra_test.go: declaração dos testes utilizando a ferramenta Terratest (Go lang);

defaults.tf: declaração dos testes utilizando a funcionalidade de testes do próprio Terraform.

Esses arquivos foram organizados em uma estrutura de pastas e arquivos que possibilitam a separação lógica dos arquivos por recursos e também o controle de provisionamento por ambiente, facilitando assim o entendimento do provisionamento de cada parte da infraestrutura. Na figura abaixo é possível visualizar a disposição das pastas e arquivos.

Figura 10 – Estrutura de pastas e arquivos do projeto



5.2.1 Módulo Libvirt

Tendo em vista que no servidor da fábrica de software está instalado o Red Hat Enterprise Linux (v8.5), sendo que esta passará a fornecer soluções para SESP-MT, as máquinas virtuais são KVM.

Na raiz da pasta terraform temos o arquivo *main.tf* declarando os módulos que serão utilizados conforme abaixo:

```
module "env-libvirt" {
  source  = "./module/libvirt"
  prefix  = var.prefix
}
```

O primeiro módulo está relacionado ao *libvirt*, uma API para gerenciamento de virtualizações e que será utilizada para as VMs KVM. Os módulos estão organizados em subpastas na pasta *module*, temos portanto uma pasta *module/libvirt*, com os arquivos de provisionamento para essa plataforma.

Na pasta *module/libvirt* portanto temos os seguintes arquivos referente ao módulo Libvirt:

variables.tf: declaração e atribuição de valores padrão para as variáveis utilizadas no código;

domain.tf: descreve as VMs que serão provisionadas;

main.tf: configuração do provider a ser utilizado pelo módulo;

pool.tf: definição do Storage Pool para volumes de armazenamento;

volume.tfvars: definição da imagem padrão e dos volumes de armazenamento das máquinas virtuais;

template.tf: definição dos templates a serem utilizados para configuração inicial dos sistemas operacionais;

network.tf: descreve os recursos de rede que devem ser provisionados;

cloud_init.cfg: definição das configurações e definições iniciais do sistema operacional, como hostname, DNS, usuário e chave pública para acesso via SSH;

network_config.cfg: definição das configurações de IP dos sistemas operacionais;

Os arquivos do módulo Libvirt definem o provisionamento de toda a infraestrutura (Storage, volume, network, máquinas virtuais, configuração do sistema operacional e as váriaveis do Terraform)

5.2.2 Configuração do provider

Na pasta *libvirt* temos outro arquivo *main.tf* contendo a configuração do provider:

```
terraform {
  required_version = ">=0.12.0"
  required_providers {
    libvirt = {
      source = "dmacvicar/libvirt"
    }
  }
}

```

```
provider "libvirt" {
  uri = "qemu+ssh://esa@200.129.244.170/system"
}
```

A partir deste ponto temos a definição dos recursos que serão provisionados, sendo que a estrutura é organizada conforme:

```
resource "<TIPO_DO_RECURSO>" "<ID_DO_RECURSO>" {
  <ATRIBUTO> = "<VALOR>"
}
```

Recomendamos a leitura de Terraform Registry para descobrir quais são os tipos de recursos e os atributos que podem ser utilizados pelo provider que está sendo utilizado e também dos exemplos disponíveis em dmacvicar/terraform-provider-libvirt.

5.2.3 Configuração do Storage pool

Em seguida temos a configuração do *pool* do libvirt que representa um storage que pode ser utilizado pelas máquinas virtuais e a configuração da faixa de IPs que poderão ser utilizados:

```
resource "libvirt_pool" "environment" {
  name = "${terraform.workspace}-pool"
  type = "dir"
  path = "/home/esa/environment/${terraform.workspace}"
}
```

5.2.4 Configuração de volumes

Podemos a partir desse momento declarar os *volumes* de armazenamento que serão utilizados, que representam os volumes individuais para cada VM, optamos por utilizar imagens CentOS 7 por haver maior conformidade com o Red Hat Linux e atender os requisitos dos serviços a serem provisionados.

```
resource "libvirt_volume" "centos-qcow2" {
  name   = "${terraform.workspace}-centos-qcow2"
  pool   = "${terraform.workspace}-pool"
  source = "https://cloud.centos.org/centos/7/images/CentOS-7-x86_64-GenericCloud
      -2111.qcow2c"
  depends_on = [
    libvirt_pool.environment
  ]
}

resource "libvirt_volume" "centos" {
  count  = ("${terraform.workspace}" == "production" ? length(var.vm_names) : length(
      var.vm_names_release))
  name   = ("${terraform.workspace}" == "production" ? "${terraform.workspace}-centos
      -${var.vm_names[count.index]}.qcow2" : "${terraform.workspace}-centos-${var.
      vm_names_release[count.index]}.qcow2")
  base_volume_id = "${libvirt_volume.centos-qcow2.id}"
  size   = var.vm_disk_size[count.index]
  pool   = "${terraform.workspace}-pool"
  depends_on = [
    libvirt_volume.centos-qcow2
  ]
}
```

5.2.5 Configuração de Networks

Para que as máquinas virtuais consigam se comunicar com a rede de computadores é necessário a declaração das redes virtuais (*Virtual Networks*) que serão utilizadas na topologia do projeto. A declaração foi realizada no arquivo *network.tf* com o seguinte conteúdo:

```
resource "libvirt_network" "general-network" {
  name = "${terraform.workspace}-network-geral"
  mode = "nat"
  autostart = true

  addresses = [ var.general_network ]
  dhcp {
    enabled = true
  }
}

resource "libvirt_network" "k82-network" {
  name = "${terraform.workspace}-network-k8s"
  mode = "nat"
  autostart = true

  addresses = [ var.k8s_network ]
  dhcp {
    enabled = true
  }
}

resource "libvirt_network" "db-network" {
  name = "${terraform.workspace}-network-db"
  mode = "nat"

  addresses = [ var.db_network ]
  dhcp {
    enabled = true
  }
}

resource "libvirt_network" "app-network" {
  name = "${terraform.workspace}-network-app"
  mode = "nat"
  autostart = true

  addresses = [ var.app_network ]
  dhcp {
    enabled = true
  }
}
```

Utilizou-se redes distintas para os ambientes de Release e Production

5.2.6 Configuração de Templates

O provider libvirt do Terraform possibilita trabalhar com cloud-init para inicialização das instâncias do sistema operacional. Essa abordagem possibilita a customização de imagens Linux a partir de arquivos de templates com configurações específicas. Nesse projeto foram realizadas as seguintes configurações no momento de provisionamento:

- Configuração do Hostname
- Instalação do agente do QEMU
- Instalação do Git
- Criação do usuário ESA com configurações inciais, inclusive de acesso via par de chaves SSH

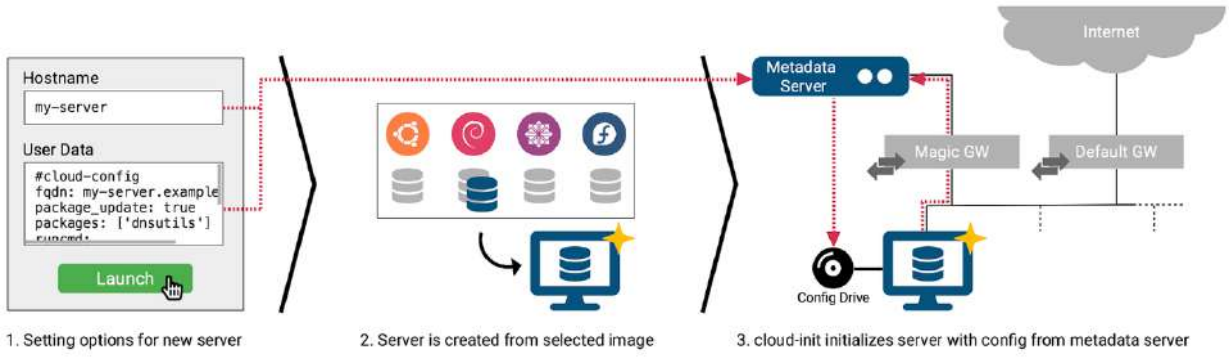
Recomendamos a leitura da documentação do cloud-init cloud-init 22.4.2 documentation

No Terraform é necessário criar os templates a serem utilizados pelos domains além da criação de imagem *iso* que será montada junto à máquina virtual contendo as configurações de inicialização.

Esse método trabalha com imagens para plataformas da nuvem como QCOW2 para estanciar imagens Linux customizadas. Essas imagens já foram criadas previamente e possuem e possuem o pacote do cloud-init instalado.

Para obter maiores informações e fazer download de imagens Linux prontas no formato QCOW2.

Figura 11 – Processo de customização de imagens Linux



Para que seja possível aplicar as configurações de inicialização é necessário criar os arquivos a seguir.

5.2.6.1 Template

template.tf: definição dos templates a serem utilizados para configuração de inicialização dos sistemas operacionais.

```
data "template_file" "user_data" {
  template = file("${path.module}/cloud_init.cfg")

  vars = {
    HOSTNAME = var.vm_names[count.index]
  }

  count = length(var.vm_names)

}

data "template_file" "network_config" {
  template = file("${path.module}/network_config.cfg")

  vars = {
```

```
    IP = var.vm_ips[count.index]
    GW  = var.vm_gw[count.index]
  }

  count = ("${terraform.workspace}" == "production" ? length(var.vm_names) : length(
      var.vm_names_release))

}

resource "libvirt_cloudinit_disk" "commoninit" {
  name           = "commoninit-${terraform.workspace}-${count.index}.iso"
  user_data      = data.template_file.user_data[count.index].rendered
  network_config = data.template_file.network_config[count.index].rendered
  pool           = "${terraform.workspace}-pool"
  count = ("${terraform.workspace}" == "production" ? length(var.vm_names) : length(
      var.vm_names_release))
  depends_on = [
    libvirt_volume.centos
  ]
}
```

5.2.6.2 Inicialização do Sistema Operacional

cloud_init.cfg: definição das configurações e definições iniciais do sistema operacional, como hostname, DNS, usuário e chave pública para acesso via SSH:

```
#cloud-config
# vim: syntax=yaml
#
# **********************
#    ---- for more examples look at: ------
# ---> https://cloudinit.readthedocs.io/en/latest/topics/examples.html
# ******************************
#
# This is the configuration syntax that the write_files module
# will know how to understand. encoding can be given b64 or gzip or (gz+b64).
# The content will be decoded accordingly and then written to the path that is
# provided.
#
# Note: Content strings here are truncated for example purposes.
runcmd:
  # Set hostname
  - hostnamectl set-hostname ${HOSTNAME}
  #- sudo echo "teste" >> /home/esa/.ssh/id_rsa

# install packages
packages:
 - qemu-guest-agent
 - git

groups:
  - admingroup: [root,sys]
  - cloud-users

users:
- name: esa
  gecos: ESA
  primary_group: esa
  groups: users
  sudo:  ALL=(ALL) NOPASSWD:ALL
```

```
  expiredate: '2032-09-01'
  lock_passwd: false
  passwd: $6$rounds=4096$SQ.aQ2T8vCFRvsRc$7BEbdi.
      XBlIiqIbWt5ZCpXPl65tANX65gCwxCfGdYTDCTHuJeYkGB9Wv0.GrNiPtmHA6yMHVYWd8q1QeoT4ti/ #
      teste
  ssh_authorized_keys:
    - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC+EeWE/Py67OJHI55lGrNuJj+
        VBxSaReRnitqut2JsRa6vqgDtpR5UCaFfog1WuupMJfBZDMGmhPxAOfnRv5lz5p61+2
        yAiOELLbLeyyxr51YQRP+nWneNoBlYQjya+U+T5rsF5EQfkIARYY+
        hgtlJ4BDgTG3ALE0F7K5k10qnvpEdnuELh3OUfcyti7242crsAmI2HVINix14bk2k4ohbXsYBrSS+
        XBzOM74SDME7FCkthsp+TWmnmI6mxof21dlfxUMTb9WQBH4fJsVNQ03dbNcMP5W/
        D0TuO03uogijvnDFyzmmx1Xo6EQhXYAFquyCRzNPtUtKFAnVP9H0vQRv esa@localhost

ssh_pwauth: True
password: password1
chpasswd:
  expire: false
  list:
    - root:terraform-libvirt-linux
```

5.2.6.3 Configuração de rede

network_config.cfg: definição das configurações de IP dos sistemas operacionais;

```
version: 2
ethernets:
    eth0:
      dhcp4: false
      # default libvirt network
      addresses: [ ${IP} ]
      gateway4: ${GW}
      nameservers:
        addresses: [ 8.8.8.8 ]
      search: [ example.com ]
```

5.2.7 Variáveis do projeto

O Terraform possibilita a criação de variáveis para armazenamento de informações que podem ser passadas para os recursos. Isso possibilita simplificar o código e também, torná-lo mais legivel e contribuindo com a manutenção do código, evitando por exemplo hard-coding.

Para o projeto foram adotadas as seguintes variáveis afim de provisionar a infraestrutura para os ambientes de Release e Production:

```
variable "prefix" {}
variable "vm_names" {
  description = "Nomes das VMs"
  type = list(string)
  default = ["gitlab","nginx","postgres","k8s","ansible","openldap","elasticstack"]
}

variable "vm_names_release" {
  description = "Nomes das VMs"
  type = list(string)
  default = ["nginx","postgres","k8s","openldap","elasticstack"]
}

variable "vm_memory" {
```

```
  description = "Memoria RAM das VMs"
  type = list(string)
}

variable "vm_cpu" {
  description = "VCPUs das VMs"
  type = list(number)
}

variable "vm_disk_size" {
  description = "Espaco em disco das VMs"
  type = list(number)
}

variable "vm_ips" {
  description = "IPs das VMs"
  type = list(string)
}

variable "vm_gw" {
  description = "IPs das VMs"
  type = list(string)
}

variable "vm_networks" {
  description = "IPs das VMs"
  type = list(string)
}

variable "general_network" {
  description = "Rede geral"
  default = ""
}
variable "k8s_network" {
  description = "Rede Kubernetes"
  default = ""
}
variable "db_network" {
  description = "Rede de banco de dados"
  default = ""
}
variable "app_network" {
  description = "Rede de aplicacoes"
  default = ""
}
```

Recomendamos a leitura de Input Variables - Configuration Language — Terraform — HashiCorp Developer para maiores informações sobre a declaração e uso de variáveis no Terraform.

Para que fosse possível a utilização do mesmo código para publicação de diferentes ambientes utilizando a estratégia de Workspaces, os dados de criação dos diferentes ambientes foram declarados no arquivo *parameters.tf*:

```
variable "prefix" {}
variable "vm_names" {
  description = "Nomes das VMs"
  type = list(string)
  default = ["gitlab","nginx","postgres","k8s","ansible","openldap","elasticstack"]
}

variable "vm_names_release" {
```

```
  description = "Nomes das VMs"
  type = list(string)
  default = ["nginx","postgres","k8s","openldap","elasticstack"]
}

variable "vm_memory" {
  description = "Memoria RAM das VMs"
  type = list(string)
}

variable "vm_cpu" {
  description = "VCPUs das VMs"
  type = list(number)
}

variable "vm_disk_size" {
  description = "Espaco em disco das VMs"
  type = list(number)
}

variable "vm_ips" {
  description = "IPs das VMs"
  type = list(string)
}

variable "vm_gw" {
  description = "IPs das VMs"
  type = list(string)
}

variable "vm_networks" {
  description = "IPs das VMs"
  type = list(string)
}

variable "general_network" {
  description = "Rede geral"
  default = ""
}
variable "k8s_network" {
  description = "Rede Kubernetes"
  default = ""
}
variable "db_network" {
  description = "Rede de banco de dados"
  default = ""
}
variable "app_network" {
  description = "Rede de aplicacoes"
  default = ""
}
```

A infraestrutura é provisionada com base no workspace selecionado. O trecho abaixo é resposável por verificar qual workspace está selecionado e atribuir as informações respectivas:

```
# Cria a variavel environmentvars com workspace selecionado. Caso nenhum esteja ou nao
    esteja declarado acima sera
utilizado "release"

environmentvars = contains(keys(local.env), terraform.workspace) ? terraform.workspace
    : "release""
```

```
5
6 #Atribui o array de valores para a variavel workspace que sera utilizada entao em main
      .tf provendo informacoes as
7 variaveis do modulo libvirt
8
9 workspace = local.env[local.environmentvars]
```

Recomendamos a leitura de https://developer.hashicorp.com/terraform/language/values/locals para maiores informações sobre a declaração de valores e uso de expressões no Terraform.

5.2.8 Configuração de Domains

Temos então o arquivo domain referente ao provisionamento de cada VM do protótipo funcional. No exemplo apenas um domínio é declarado pois utilizamos o recurso do Terraform *[count.index]* para iterar a variável *var.vm_names* e executar a criação das demais VMS conforme o tamanho (length) da variável *var.vm_names*. Isso possibilita que o *[count.index]* seja utilizado com as demais variáveis *var.vm_memory*, *var.vm_cpu*, *var.vm_networks* para criação de todas as máquinas virtuais, pois todas possuem o mesmo número de elementos.

Caso seja necessária remover uma máquina virtual será necessário apagar o elemento da respectiva máquina em todas as variáveis e rodar o Terraform apply.

```
1 resource "libvirt_domain" "new-domain" {
2   name   = ("${terraform.workspace}" == "production" ? "${terraform.workspace}-centos
        -${var.vm_names[count.index]}" : "${terraform.workspace}-centos-${var.
        vm_names_release[count.index]}")
3
4   memory = var.vm_memory[count.index]
5   vcpu   = var.vm_cpu[count.index]
6   qemu_agent = true
7   autostart = true
8   depends_on = [
9     libvirt_volume.centos
10  ]
11
12  network_interface {
13    network_name = "${terraform.workspace}-network-${var.vm_networks[count.index]}"
14    hostname   = ("${terraform.workspace}" == "production" ? "${terraform.workspace}-
          centos-${var.vm_names[count.index]}" : "${terraform.workspace}-centos-${var.
          vm_names_release[count.index]}")
15    wait_for_lease  = true
16  }
17
18  cloudinit =  libvirt_cloudinit_disk.commoninit[count.index].id
19
20  # IMPORTANT: this is a known bug on cloud images, since they expect a console
21  # we need to pass it
22  # https://bugs.launchpad.net/cloud-images/+bug/1573095
23  console {
24    type        = "pty"
25    target_port = "0"
26    target_type = "serial"
27  }
28
29  console {
30    type        = "pty"
```

```
    target_type = "virtio"
    target_port = "1"
  }

  disk {
    volume_id = element(libvirt_volume.centos.*.id, count.index)
  }

  graphics {
    type        = "spice"
    listen_type = "address"
    autoport    = true
  }

  count = ("${terraform.workspace}" == "production" ? length(var.vm_names) : length(
      var.vm_names_release))
}
```

5.2.9 Workspace e TFSTATE

A partir na incialização do projeto e execução do apply no Terraform um arquivo de estado (terraform.tfstate) é criado. Todas as alterações no código que serão aplicadas no alvo são salvas nesse arquivo.

Caso esse arquivo seja perdido, não será possível retomar o status da infraestrutura, ou seja, o Terraform não conseguirá identificar qual o estado atual das configurações no servidor alvo.

Recomandamos a leitura de https://www.terraform.io/language/state para maiores informações.

O Terraform possibilita trabalhar com o recurso de *Workspace*, onde entre outras funcionalidades, possibilita controlar os estados da infraestrutura provisionada para o mesmo código em ambiente diferentes. Para o projeto utilizamos o mesmo código para provisionar a infraestrutura em ambos os ambientes (*Release e Production*), mas iremos controlar o estado em arquivos TFSTATE separados.

Para isso é necessário a criação dos *workspaces release* e *production* da seguinte maneira:

```
## Workspace release ##
terraform workspace new release

- Arquivo de estado fica armazenado em
terraform/modules/libvirt/terraform.tfstate.d/release/terraform.tfstate


## Workspace production ##
terraform workspace new production

- Arquivo de estado fica armazenado em
terraform/modules/libvirt/terraform.tfstate.d/production/terraform.tfstate
```

Dessa forma toda vez que quisermos aplicar uma alteração na infraestrutura e ambiente alvo, precisamos alterar de workspace conforme necessidade. O código criado para o projeto está adaptado para criação conforme workspace selecionado.

terraform workspace select ¡release/production¿

Recomendamos a leitura de https://www.terraform.io/language/state/workspaces para maiores informações.

5.2.9.1 Armazenamento remoto do TFSTATE

Como esse projeto teve a necessidade de implantação de uma pipeline de provisionamento, foi necessário a configuração de um repositório remoto para o arquivo de estado. Para isso, foi criado o arquivo *backend.tf* com as configurações do repositório para armazenamento no banco de dados Postgres:

```
terraform {
  backend "pg" {
    conn_str = "postgres://terraform:terraform123@200.129.244.170:54322/
        terraform_database?sslmode=disable"
  }
}
```

Existem diversos backends disponíveis, recomendamos a leitura da página Backend Type: remote — Terraform — HashiCorp Developer

Recomendamos a leitura da página Backend Configuration - Configuration Language — Terraform — HashiCorp Developer para maiores informações sobre a configuração remota do arquivo de estados.

Referências

Terraform Registry

GitHub - dmacvicar/terraform-provider-libvirt: Terraform provider to provision infrastructure with Linux's KVM using libvirt

https://itnext.io/how-to-use-terraform-to-create-a-small-scale-cloud-infrastructure-abf54fabc9dd

Quick Start KVM libvirt VMs with Terraform and Ansible – Part 1 — DesGehtFei.Net

Quick Start KVM libvirt VMs with Terraform and Ansible – Part 2 — DesGehtFei.Net

Code structure

How to Create Terraform Multiple Environments

Managing Multiple Environments with Terraform

Terraform - Libvirt provisioning automatique

https://subscription.packtpub.com/book/cloud_and_networking/9781800207554/2/ch02lvl1sec15/provisioning-infrastructure-in-multiple-environments

How to Split and Organize Terraform Code Into Modules

11 things to know before beginning work with Terraform I — Endava Blog

11 things to know before beginning work with Terraform II — Endava Blog

Dependency Lock File (.terraform.lock.hcl) - Configuration Language — Terraform — HashiCorp Developer

libvirt_volume with base_volume_id creates file with wrong permissions · Issue 658 · dmacvicar/terraform-provider-libvirt

cloud-init 22.4.2 documentation

How To Use Cloud-Config For Your Initial Server Setup — DigitalOcean

KVM: Testing cloud-init locally using KVM for a CentOS cloud image

How To Use Cloud-Config For Your Initial Server Setup — DigitalOcean

GitHub - gruntwork-io/terratest: Terratest is a Go library that makes it easier to write automated tests for your infrastructure code.

Initialize Servers with cloud-init - cloudscale.ch News

CAPÍTULO 6

Ansible

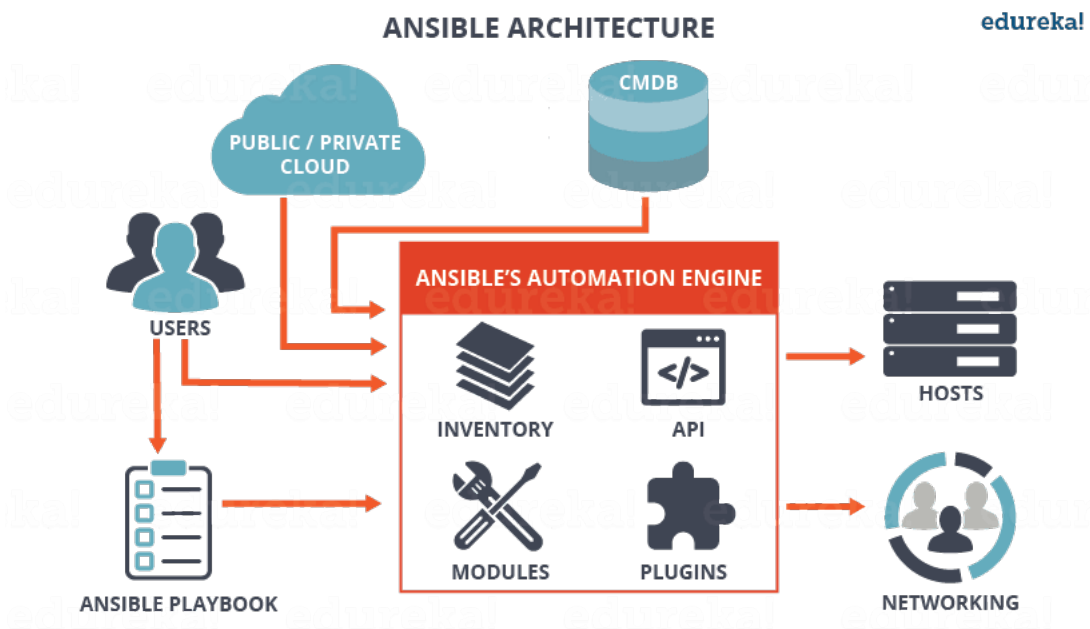
Neste capítulo serão apresentadas as características da ferramenta Ansible, como alternativa ao provisionamento de infraestrutura como código.

6.1 Introdução

Ansible também é uma ferramenta que permite o provisionamento e configuração em IaC, seu ponto mais forte é a capacidade e flexibilidade em configuração.

Quanto à arquitetura dessa ferramenta, podemos observar os componentes na Figura 1. Descrevemos cada um dos elementos conforme segue:

Figura 12 – Arquitetura do Ansible



Inventory: guarda uma lista dos nós ou hosts que representam endereços IPs, bancos de dados, servidores, etc., que precisam ser gerenciados;

API's: permite a comunicação com e entre os módulos;

Modules: são programas instalados nos nós para a execução de alguma tarefa e removidos após o término da mesma;

Plugins: permitem a expansão das funcionalidades do motor principal do Ansible;

Playbooks: consiste do código escrito em YAML pela equipe Ops para descrever as tarefas e o processo de provisionamento e configuração desejado;

Hosts: representam os nós do sistema, que são automatizados e gerenciados pelo Ansible;

Networking: permite a automação das configurações de rede

CMDB (Configuration Management Database): trata-se de um tipo de repositório que consiste da rede completa de computadores ou componentes da infraestrutura de TI

6.2 Introdução

Para possibilitar a execução do Ansible, é necessário que a instalação seja feita em um nó, conhecido como **nó controlador**. Esta instalação poderá diferir, dependendo do sistema operacional escolhido.

Por ser o ponto inicial, sua instalação e configuração demandam intervenção manual, e os passos para instalar e configurar o nó controlador serão tratados a seguir.

É importante lembrar que o sistema operacional escolhido para o nó controlador do protótipo funcional foi o CentOS 7. No entanto, os passos para instalação em outros sistemas operacionais podem ser encontrados na documentação oficial.

6.2.1 Instalação do Ansible

Para instalar o Ansible no CentOS 7 é necessário:

1. Adicionar o repositório:

```
sudo yum -y install epel-release && sudo yum repolist
```

2. Instalar a ferramenta:

```
sudo yum -y install ansible
```

3. Conferir a instalação (Opcional):

```
ansible –version
```

6.2.2 Instalação de módulos adicionais

É comum que durante a construção dos playbooks haja a necessidade da utilização de módulos que não estejam inclusos na instalação padrão - ou **builtin** - do Ansible. Para a execução do protótipo funcional é necessário efetuar a instalação dos seguintes módulos:

- ansible.posix: coleção de recursos para plataformas POSIX

- community.general: conjunto de módulos adicionais para diversas finalidades
- community.postgresql: conjunto de módulos que possiblitam o gerenciamento de bases de dados PostgreSQL

Para efetuar a intalação destes módulos é necessário executar os seguintes comandos no nó controlador:

```
ansible-galaxy collection install ansible.posix
ansible-galaxy collection install community.general
ansible-galaxy collection install community.postgresql
```

6.3 Estrutura

Existem diversas abordagens para a construção e organização do código utilizando Ansible, mas tudo depende do tamanho e complexidade de cada projeto. Para uma análise aprofundada, recomenda-se a referência oficial de boas práticas com Ansible.

Para entendermos a hierarquia de componentes do Ansible é possível organizá-la, do componente mais genérico para a mais espífico, da seguinte forma: ec

Inventory: lista de hosts

Playbook: define quais roles serão aplicadas para cada host

Role: contém uma lista de passos (tasks) a serem executados

Task: uma unidade de execução, que pode fazer uso de várias outras estruturas auxiliares como handlers, variables, templates, etc

A seguir, analisaremos do nível mais genérico para o mais específico, os principais componentes da estrutura de um projeto Ansible, e como eles se aplicam ao protótipo funcional.

6.3.1 Inventory

Os arquivos de inventário são responsáveis por manter a lista de hosts e seus aliases, que serão referenciados pelos playbooks. É comum nomear estes arquivos apenas como *hosts*.

No cenário do protótipo funcional os hosts são diferentes para cada ambiente. Isso significa que os ambientes de **produção** e **release** são equivalentes em ferramentas e aplicações, mas as máquinas virtuais - e consequentemente os IPs delas - serão diferentes ente estes ambientes. Por este motivo, adotou-se uma separação por diretórios, da seguinte forma:

```
entrega-continua
  |- ansible
  |    |- environments
  |    |    |- prod
  |    |    |    |- hosts
  |    |    |- release
  |    |    |    |- hosts
  ...
```

Desta forma, cada ambiente possui seu próprio arquivo *hosts*, contendo os IPs e alias correspondentes. Como exemplo, podemos analisar o arquivo *hosts* referente ao ambiente de produção:

```
[kubernetes]
172.16.1.226

[nginx]
172.16.0.169

[openldap]
10.24.3.192

[postgresql]
172.17.2.71

[all:vars]
ansible_user=esa
```

6.3.2 Playbooks

Os *playbooks* são responsáveis por organizar e definir o que será executado para cada host. É comum haver a necessidade de se criar diferentes playbooks, principalmente para possibilitar a execução separada de um conjunto de regras.

No protótipo funcional os arquivos referentes aos *playbooks* estão logo abaixo do diretório *ansible*, conforme o exemplo abaixo:

```
entrega-continua
  |- ansible
  |     |- gitlab.yml
  |     |- tools.yml
  ...
```

Como exemplo, podemos analisar uma parte do código do playbooks *tools.yml*, referente à configuração do host *nginx*:

```
- hosts: nginx
  become: yes
  roles:
    - instalar_nginx
    - configurar_nginx
...
```

Neste exemplo, podemos perceber:

- A identificação do host, conforme identificado no arquivo de inventário, neste caso *nginx*.
- A instrução *become*, para garantir que a execução seja realizada como *sudo*.
- A lista de roles que deverão ser executadas, na ordem esperada.

6.3.3 Roles

Roles representam um framework para um conjunto completo e independente de variáveis, tarefas (*tasks*), arquivos, templates e módulos.

Em Ansible, a *role* é o primeiro mecanismo para quebrar um *playbook* em múltiplos arquivos. Dessa forma podemos simplificar o código do *playbook*, facilitando o entendimento do script, respectiva manutenção e reuso.

Cada role deve ser limitada a uma funcionalidade, com todos os passos necessários para obter o resultado desejado, que pode ser suficiente para atender a própria *role* ou necessária para outras roles que dela dependem.

É importante entender que roles não são *playbooks*. *Roles* representam pequenas funcionalidades que podem ser utilizadas de forma independente, mas sempre dentro de *playbooks*.

As *roles* do protótipo funcional estão organizadas dentro do diretório *roles*:

```
entrega-continua
  |- ansible
  |     |- roles
  |     |     |- configurar_cluster
  |     |     |- configurar_docker
  |     |     |- configurar_gitlab
  |     |     |- configurar_ingress_default
  |     |     |- configurar_nginx
  |     |     |- configurar_openldap
  |     |     |- configurar_postgresql
  |     |     |- instalar_argocd
  |     |     |- instalar_docker
  |     |     |- instalar_gitlab
  |     |     |- instalar_kind
  |     |     |- instalar_kubectl
  |     |     |- instalar_nginx
  |     |     |- instalar_openldap
  |     |     |- instalar_postgresql
  ...
```

Perceba que as roles foram organizadas em duas categorias: *instalar_...* e *configurar_...*, dessa forma separamos as tasks relacionadas a instalação de cada aplicativo/serviço de sua configuração, facilitando a leitura e a manutenção do código.

Tomemos como exemplo a role *instalar_nginx* de instalação do NGINX (*.../tasks/main.yml*):

```
---
# task para atualizar os pacotes do yum
- name: atualizando pacotes
  yum: name=* state=latest

# task para adicionar um repositorio ao yum
- name: adicionando repositorio
  yum: name=epel-release state=latest

# task para instalar o nginx utilizando yum
- name: instalando
  yum:
    name: nginx
    state: present
    update_cache: yes

# task para iniciar o nginx
- name: iniciando servico
  service:
    name: nginx
    state: started
```

A seguir podemos iniciar o processo de configuração do NGINX, role *configurar_nginx*:

Um handler foi criado (*.../handlers/main.yml*) para reiniciar o NGINX após alterações e que será utilizada por nossas tasks:

```
---
- name: reiniciar nginx
  systemd:
    name: nginx
    state: restarted
```

Um template foi criado (.../templates/nginx.conf.j2) para substituir o arquivo de configuração do Nginx, adicionando as configurações necessárias para o server:

```
...
   server {

    listen 80;

    server_name gitlab.fabricadesoftwareifmt.com.br;


    location / {

        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Port $server_port;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_pass http://10.24.3.75:80;
        proxy_http_version 1.1;
#       proxy_set_header Connection $connection_upgrade;
        proxy_set_header Upgrade $http_upgrade;
        proxy_read_timeout 900s;

    }

}
...
```

Criamos as variáveis (*../vars/prod.yml* e *../vars/release.yml*) que serão utilizadas para preencher o arquivo de configuração:

```
---
argo_cd_url: "..."
ldap_url: "..."
registro_geral_url: "..."
```

Finalmente temos o script de de configuração do NGINX (*.../tasks/main.yml*):

```
---
# task para copiar o arquivo de configuracao do NGINX
- name: copiando arquivo de configuracoes
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/default.d/sespmt.conf
  notify: reiniciar nginx

# task para configurar o Security-Enhanced Linux para permitir que
# scripts e modulos HTTPD possam conectar a rede
- name: configurando SELinux
  ansible.posix.seboolean:
    name: httpd_can_network_connect
```

```
14      state: yes
15      persistent: yes
```

Perceba que ao copiar o arquivo de configuração do NGINX, o *handler* reiniciar *nginx* é notificado para entrar em execução.

6.3.4 Relação de Roles e Tasks do Protótipo Funcional

Aplicação	Roles	Tasks
Docker	instalar_docker	instalando dependencias instalando iniciando servico
	configurar_docker	configurando grupos
Gitlab	instalar_gitlab	verificando configuracao verificando instalacao instalando dependencias baixando script de instalacao instalando repositorio definindo nome do pacote instalando
	configurar_gitlab	reconfigurando (primeiro acesso) criando diretorio de configuracao SSL criando certificacao autoassinado copiando arquivo de configuracao

6.4 Referências

Ansible Best Practices

Ansible Galaxy

How to Manage Multistage Environments with Ansible

Aplicação	Roles	Tasks
NGINX	instalar_nginx	atualizando pacotes adicionando repositorio instalando iniciando servico
	configurar_nginx	copiando arquivo de configuracoes configurando SELinux
OpenLDAP	instalar_openldap	atualizando pacotes instalando dependencias iniciando servico
	configurar_openldap	criando banco de dados aplicando configuracoes de acesso copiando configuracoes iniciais alterando permissoes do arquivo de configuracoes reiniciando servico

Aplicação	Roles	Tasks
PostgreSQL	instalar_postgresql	adicionando repositorio instalando dependencias instalando adapter verificando inicializacao configurando cluster
	configurar_postgresql	configurando permissoes para conexoes remotas configurando interfaces iniciando servico criando usuario do banco de dados criando banco de dados copiando script inicial executando script inicial

Aplicação	Roles	Tasks
Kind	configurar_cluster	copiando arquivo de criacao do cluster checando a criacao do cluster criando o cluster instalando o ingress-nginx
	configurar_ingress_default	copiando o arquivo aplicando
	instalar_argocd	checando namespace criando o namespace criando artefatos
	instalar_kind	instalando aplicando permissao adicionando ao diretorio
	instalar_kubectl	instalando aplicando permissao adicionando ao path

CAPÍTULO 7

Configuração da Pipeline

Este capítulo tem como objetivo descrever a configuração da PIPELINE do Gitlab CI/CD para provisionamento e configuração do projeto.

7.1 Configuração do Terraform no Gitlab

A documentação de configuração do Terraform na PIPELINE do Gitlab está disponível em Infrastructure as Code with Terraform and GitLab — GitLab.

Para facilitar a implementação, foram utilizados como referência os Templates do *gitlab-ci.yaml* disponível em Infrastructure as Code with Terraform and GitLab — GitLab e no repositório lib/gitlab/ci/templates master GitLab.org / GitLab

O Gitlab possibilita o armazenamento do TFSTATE do Terraform utilizando o backend baseado em HTTP. A configuração está disponível em GitLab-managed Terraform state — GitLab. Para esse projeto não utilizamos esse recurso pois ele não suporte o uso de **Workspaces**.

Para trabalhar com Workspaces é necessário o backend do TFSTATE possuir suporte. Verifique em https://www.terraform.io/language/state/workspaces os backends que possuem suporte.

Para configurar a PIPELINE utilizamos o backend do Postgres disponível em Backend Type: pg — Terraform — HashiCorp Developer. Basta seguir a documentação para configurá-lo. O TFSTATE é armazenado na tabela *states* conforme o Workspace que está sendo utilizado conforme exemplo da Figura 13.

Figura 13 – Armazenamento do TFSTATE no Postgres

	id [PK] bigint	name text	data text
1	1	release	{ "version": 4, "terraform_version": "1.1.9", "serial": 1916, "lineage": "d7995f22-4d54-5fa7-2e17-6b53cc6b4a7b", "outputs": { "pool-name": { "v...
2	2	teste	{ "version": 4, "terraform_version": "1.2.3", "serial": 1, "lineage": "8e82498c-c007-81d5-ee53-5414ed2cff19", "outputs": { "pool-name": { "valu...
3	3	production	{ "version": 4, "terraform_version": "1.2.3", "serial": 988, "lineage": "ea6d896c-6887-35c7-379c-55ec7557c228", "outputs": { "pool-name": { "v...

Trabalhar com um backend remoto possibilita a manutenção e operação do Terraform por múltiplos usuários.

Essa configuração é necessária para a PIPELINE e é abordagem similiar a descrita na arquitetura do item “2. Protótipo funcional “ exposta no documento Terraform, porém ela é necessária para a PIPELINE funcionar gravando e consultando o TFSTATE remotamente.

7.2 Imagens do Docker

Para a execução da PIPELINE são requeridas imagens customizadas para rodar os diversos ”jobs”configurados. O Gitlab possui uma imagem padrão do Terraform.

A imagem Terraform do Gitlab utiliza o comando *gitlab-terraforme* não apenas *terraform* para executar as ações.

7.2.1 Imagem do Terraform

Essa é a imagem padrão utilizada na PIPELINE e é baseada na imagem Terraform do Gitlab.

```
FROM registry.gitlab.com/gitlab-org/terraform-images/stable:latest

RUN apk add go
```

7.2.2 Imagem do Golang

A imagem Terraform do Gitlab possui configurações diferentes das esperadas pelo Terratest utilizando Golang, por utilizou-se para o Stage de Test Terraform a imagem oficial do Terraform com a instalação do Golang.

```
FROM hashicorp/terraform:1.2.3

RUN apk add go
```

7.2.3 Imagem do Ansible

Para rodar os Jobs de configuração utilizando o Ansible a seguinte imagem foi construída.

```
FROM ubuntu:22.10

ARG DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install vim ssh python3-pip ansible ansible-lint -y
RUN yes | pip3 install -Iv 'resolvelib<0.6.0'
RUN ansible-galaxy collection install ansible.posix \
  & ansible-galaxy collection install community.general \
  & ansible-galaxy collection install community.postgresql
RUN echo 'root:teste' | chpasswd
RUN sed -i 's/#PermitRootLogin prohibit-password/PermitRootLogin yes/' /etc/ssh/
    sshd_config \
  & sed -i 's/#Port 22/Port 22/' /etc/ssh/sshd_config \
  & sed -i 's/#PubkeyAuthentication/PubkeyAuthentication/' /etc/ssh/sshd_config \
  & sed -i 's/#PasswordAuthentication/PasswordAuthentication/' /etc/ssh/sshd_config
```

7.3 Configuração do range de IPs do Docker

Devido ao uso de diversas redes locais, podem ocorrer conflitos de IP e Redes entre Kubernetes/-Docker e a Rede Local (LAN) devido a alocação padrão do Docker. Caso ocorra problemas de rede é necessário verificar se a rede que o Kubernetes/Docker provisionou no ambiente de conteinerização não é igual ao IP ou rede que se está tentando alcançar na rede LAN ou Virtual Lan. A seguinte configuração é um exemplo de como contornar esse problema:

```
sudo vi /etc/docker/daemon.json

{
  "log-driver": "journald",
  "log-opts": {
  "tag": "{{.Name}}"
  },
  "bip": "172.26.0.1/16"
}
```

Caso existam redes na LAN que possuam o range acima ou maior que ele, sugerimos a alteração para não ocorrer conflitos.

CAPÍTULO 8

Pipeline no GitLab

Este capítulo tem como objetivo descrever a implementação do fluxo de provisionamento de infraestrutura como código, em uma PIPELINE no Gitlab CI/CD.

8.1 Introdução

Nesse novo modelo em que a infraestrutura é representada como código, em que descrevemos o provisionamento e a configuração de VMs, storage, networking, etc. na forma de texto, podemos facilmente definir estados, replicar, destruir e versionar a infraestrutura. Isso facilita o processo de automação e consequentemente obtemos agilidade. Com todas essas características, o desenvolvimento da infraestrutura se torna muito similar ao desenvolvimento de software e levanta questões importantes sobre como podemos realizar testes em IaC.

Um questionamento também importante é se precisamos de fato testar a infraestrutura sendo que as aplicações que porventura são executadas na infra já realizam testes. Devemos considerar que existem fatores a serem considerados na infra como:

- Número de componentes: máquinas virtuais, serviços, loadbalancers, etc;
- Diversidade de ambientes: desenvolvimento, homologação, produção, etc;
- Número de atualizações ou releases que afetam a infraestrutura: (atualizações por hora, dias, meses, . . .)

Ferramentas modernas como Terraform já possuem recursos que nos permitem realizar diversas checagens e esses recursos facilitam o processo de verificação e validação da infraestrutura.

Podemos elencar três tipos básicos de teste em IaC

- Checagem local e estática;
- Aplicar e destruir;
- Teste de imitação.

8.1.1 Checagem Local e Estática

Não é necessariamente um teste e sim uma checagem, contudo são importantes para o processo tendo em vista que permite realizar uma verificação e reduzir a chance de códigos que causem erro na execução da criação ou modificação da infraestrutura. Por exemplo, analisar configurações, corretude do código e dependências antes da execução.

A própria ferramenta Terraform fornece recursos que permitem checar e validar o código da infraestrutura:

- Lint — *terraform fmt -check* and *terraform validate* para a corretudo de código;
- *Terraform fmt –check* permite identificar os arquivos que não estão bem formatados, enquanto *terraform fmt* força a correção de formatação do código nos aquivos *.tf* ;
- Preview — *terraform plan* para checagem das alterações que irão ocorrer na infraestrutura;

Apesar do processo de checagem ser um grande avanço comparado ao modelo tradicional de implantação de infraestrutura, restringe-se ao que é estático, ou seja, elementos dinâmicos, de tempo de execução, não conseguem ser verificados.

Existem ferramentas complementares como TFLint que permite personalizar layouts para a formatação do código e verificar a corretude de valores para os providers de cloud mais conhecidos como AWS e GCP.

8.1.2 Aplicar e Destruir

Neste tipo de teste, avançamos mais um passo e podemos identificar erros de natureza dinâmica. A ideia é construir a infraestrutura por um período curto de tempo e destruí-la em seguida, isso nos permite verificar atributos dinâmicos e dependências.

- Build — *TF_LOG=debug terraform (apply — destroy)* para os detalhes da execução.

Esse tipo de teste pode ocorrer utilizando a própria ferramenta de GitOps como Terraform ou ferramentas que permitem expandir as funcionalidades como Terratest, que é uma coleção de bibliotecas Go que simplificam a interação com os providers.

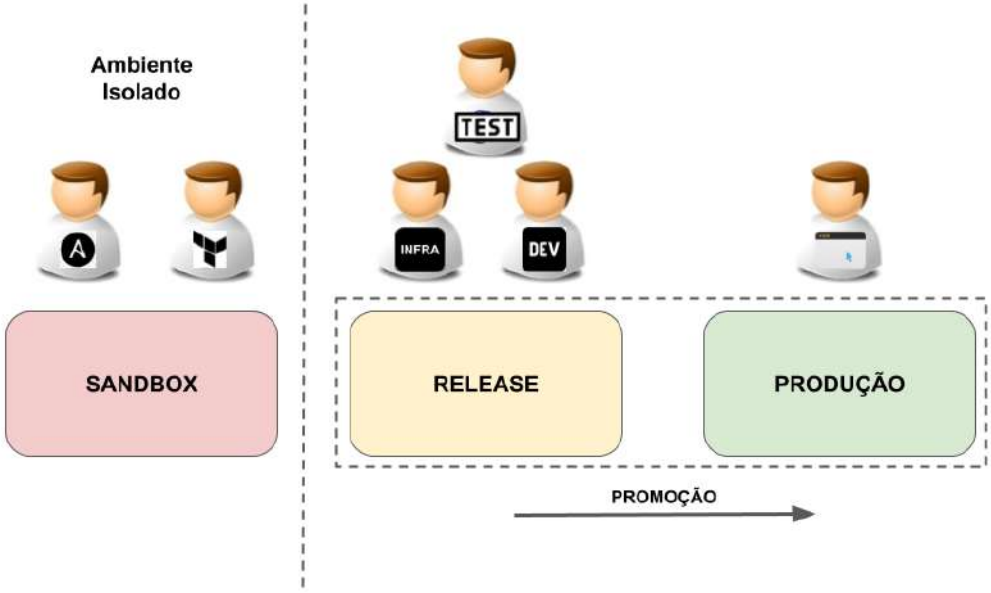
Esse tipo de teste, chamado **sandbox**, deve ocorrer em ambiente isolado para evitar efeitos indesejados na estrutura em produção, conforme podemos observar na figura 14.

O recomendado é que a equipe de operações tenha um ambiente isolado para poder construir, destruir e modificar a infraestrutura livremente, sem receios de causar dano (**sandbox**). Alterações nesse ambiente são realizados apenas para testes pela equipe de operações, podendo verificar novas versões, recursos, configurações, topologias e etc., sem burocracia.

Para estruturas de pequeno porte, a criação e destruição total da infraestrutura em cada teste é viável (minutos), mas para estruturas maiores que demandam um tempo longo para criação e destruição, o recomendado é que o ambiente sandbox seja utilizado de forma progressiva, ou seja, o ambiente é criado mas não destruído, sofrendo alterações progressivas para novos testes.

Já o ambiente de **release** é utilizado para testes de forma mais ampla, não apenas de infraestrutura, sendo utilizado não somente pela equipe de operações, mas também pela equipe de

Figura 14 – Ambientes e seus usuários



desenvolvimento e pelos testadores de software. Esse ambiente pode vir a sofrer modificações que causem instabilidade indesejada para seus usuários, ou até mesmo períodos de inatividade, mas que não afetam os recursos e usuários em produção.

Para efeitos práticos, tendo em vista que não há risco no protótipo funcional por não ser um ambiente em produção, a equipe realiza modificações progressivas diretamente no ambiente de release, sendo este utilizado inclusive para os testes sandbox.

Quando uma versão da infraestrutura é considerada testada e aceita para colocar em **produção** pode-se realizar o processo de **promoção**, que consiste em replicar as novas configurações do ambiente de release em produção.

A promoção pode ocorrer de forma totalmente automatizada, mas recomendamos a existência de uma etapa manual, contemplando uma atividade de autorização. Por exemplo, podemos promover a infraestrutura de release para produção sempre que uma **tag de versionamento** for criada no repositório.

8.1.3 Teste de Imitação

Já neste teste a ideia é imitar as operações requeridas pelas aplicações no nível da infraestrutura. O objetivo é checar se a infraestrutura está configurada corretamente.

Diversas ferramentas podem auxiliar nesse tipo de teste, o próprio Ansible permite codificar testes.

Como plano de testes definimos:

- Terraform
  - Se a VM subiu
  - Portas ativas

  - SSH
  - Resolução DNS
  - IPs ativos
  - Hostname encontrado

- Ansible
  - Verificando a versão do sistema operacional
  - Se o banco de dados foi instalado
  - Se a porta do banco está ativa
  - Se está respondendo adequadamente
  - Se os dados foram carregados

8.1.4 Pontos Fortes e Fracos

Podemos resumir os pontos fortes e fracos de cada técnica conforme segue:

Técnica	Pontos Fortes	Pontos Fracos
Análise Estática	1. Rápida; 2. Estável; 3. Sem necessidade de deploy e provisionamento de recursos; 4. Fácil de aplicar.	1. Captura poucos erros; 2. Não é confiável para publicar em produção.
Aplicar e Destruir	1. Checagem dinâmica; 2. Número bem maior de captura de possíveis erros em comparação com a análise estática; 3. Maior confiabilidade.	1. Pode demorar muito tempo a depender do tamanho da infraestrutura; 2. Consumo de recursos.
Teste de Imitação	1. Amplia a quantidade de erros que podem ser capturados em relação a aplicar e destruir; 2. Confiável;	1. Pode demorar a depender da quantidade de testes; 2. Consumo de recursos; 3. Codificação não é trivial.

8.2 Ferramentas de Teste

Muitas ferramentas tem surgido e estão ganhando espaço no desenvolvimento ou auxílio de testes para a infraestrutura como código, no entanto, dada a diversidade das ferramentas de GitOps e das plataformas de infraestrutura, uma única ferramenta não é adequada para um ambiente multiplataforma.

Podemos listar ferramentas como checkov, snyk, pulumi, terrascan, tfsec, terratest, terrafirma, tflint, conftest, são diversas cujo resultado varia, desempenhando melhor papel a depender da plataforma a ser testada. Por exemplo, a ferramenta snyk em uma comparação [1], apresentou excelente resultado na AWS, mas um resultado bem inferior na GCP.

Abaixo apresentamos algumas ferramentas populares que auxiliam na checagem estática e/ou dinâmica:

Ferramenta	Deploy / Undeploy	Validate	Funciona com
Terratest	Sim	Sim	Terraform, Kubernetes, Packer, Docker, Servers, Cloud APIs.
kitchen-terraform	Sim	Sim	Terraform
Inspec	Não	Sim	Servers, Cloud APIs
Serverspec	Não	Sim	Servers
Goss	Não	Sim	Servers
TFLint	Não	Sim	Terraform
ansible-test	Sim	Sim	Ansible
ansible-lint	Não	Sim	Ansible

Tendo em vista que a arquitetura de solução do protótipo funcional é baseada em Terraform e Ansible, os testes serão realizados priorizando os recursos dessas ferramentas, ressaltando a importância de uma análise qualitativa quando da aplicação desses novos conceitos na infraestrutura da SESP.

8.2.1 Terratest e Terraform test

Apesar do Terraform estar incorporando recursos de teste na sua distribuição padrão, entendemos que no momento desta implementação o Terratest se mostra uma opção mais madura, com mais recursos e melhor documentada.

Os testes devem ser criados e disponibilizados na pasta *tests*, por meio de arquivos terminados em *_test.go*, sendo executado pelo comando i.e. *go test infra_test.go*.

Ao abrir o arquivo *infra_test.go* percebemos que as primeiras linhas são referentes a importação das bibliotecas necessárias para a execução do teste, em nosso exemplo precisamos da biblioteca para comunicação com o Terraform e da biblioteca de asserções (*assert*) que nos permite verificar afirmações que são feitas acerca da infraestrutura.

```
1 package test
2
3 import (
4   "testing"
5
6   "github.com/gruntwork-io/terratest/modules/terraform"
7   "github.com/stretchr/testify/assert"
8 )
```

```
9
10 func TestTerraformProvisioningSespInfrastructure(t *testing.T) {
11   // retryable errors in terraform testing.
12   terraformOptions := terraform.WithDefaultRetryableErrors(t, &terraform.Options{
13     TerraformDir: "../",
14   })
15
16   defer terraform.Destroy(t, terraformOptions)
17
18   terraform.InitAndApply(t, terraformOptions)
19
20   output := terraform.Output(t, terraformOptions, "text")
21   assert.Equal(t, "Hello, World!", output)
22
23   output_vm_names := terraform.Output(t, terraformOptions, "vm-names")
24   assert.Equal(t, "[release-centos-gitlab release-centos-nginx release-centos-postgres
       release-centos-k8s release-centos-ansible release-centos-openldap]",
       output_vm_names)
25
26   output_pool := terraform.Output(t, terraformOptions, "pool-name")
27   assert.Equal(t, "release-pool", output_pool)
28
29   output_volume := terraform.Output(t, terraformOptions, "volume-name")
30   assert.Equal(t, "[release-centos-gitlab.qcow2 release-centos-nginx.qcow2 release-
       centos-postgres.qcow2 release-centos-k8s.qcow2 release-centos-ansible.qcow2
       release-centos-openldap.qcow2]", output_volume)
31
32 }
```

Já a função *TestTerraformProvisioningSespInfrastructure* representa a implementação do teste de provisionamento da infraestrutura do protótipo funcional. Nessa função temos o provisionamento da infraestrutura e a verificação se os nomes das VMs, IPs, storage e volumes foram criados. Se as afirmações não forem verdadeiras o teste falha.

Para efeito de demonstração do uso de teste do Terraform temos no protótipo funcional o arquivo *terraform/tests/defaults/defaults.tf*:

```
1 terraform {
2   required_providers {
3
4     test = {
5       source = "terraform.io/builtin/test"
6     }
7
8     http = {
9       source = "hashicorp/http"
10    }
11  }
12 }
13
14 module "env-module" {
15     prefix = "test"
16     source = "../.."
17
18 }
19
20 resource "test_assertions" "domain" {
21
22     component    =    "domain"
23
24     equal "domain_name" {
```

```
25          description = "Domain name"
26          got         = module.env-module.vm-names
27          want        =   [
28                            "test-test_defaults-centos-gitlab",
29                            "test-test_defaults-centos-nginx",
30                            "test-test_defaults-centos-postgres",
31                            "test-test_defaults-centos-k8s",
32                            "test-test_defaults-centos-ansible",
33                            "test-test_defaults-centos-openldap",
34                            ]
35      }
36
37  }
```

A parte inicial, linhas **1-12** e **14-18** são elementos de configuração referente ao terraform (importações necessárias) e definição do módulo de teste respectivamente (que aponta para o diretório que contêm os arquivos de provisionamento do terraform **../..**). As linhas **20-35** é que definem o recurso de teste (*test_assertions*), onde é possível verificar o teste de domínio em que fazemos referência a variável que recebe o nome das máquinas virtuais instanciadas (*got*) e comparamos com os domínios desejados (*want*), estes devem ser iguais para que o teste passe.

Reiteramos a recomendação de uso do Terratest por conta da maturidade, sendo o exemplo anterior apenas para demonstrar recursos que estão sendo incorporados de forma nativa ao Terraform.

8.2.2 Ansible

Já para o ansible temos como estratégia a criação de *playbooks* que realizam uma sequência de tasks de verificação. Essa estratégia tem por objetivo adotar uma postura progressiva de aprendizado, tendo em vista que a adoção do modelo IaC é complexo e incorpora diversas ferramentas, necessidades e habilidades novas para a equipe de operações. Com o crescimento da maturidade da equipe com IaC e a adoção crescente de testes automatizados, novas ferramentas podem ser incorporadas para melhorar o processo de teste.

Nos exemplos abaixo temos dois tipos de teste:

1. um de **verificação**: que tem o objetivo de avaliar se o que foi planejado realmente foi realizado, em nosso cenário, se o SO instalado é o desejado e na versão desejada;

2. e outro de **validação**: que tem o objetivo de avaliar se o que foi entregue atende as expectativas do cliente (equipe Dev), em nosso cenário, se o SGBD está instalado, com as configurações adequadas de permissão, se o database foi criado com respectivo usuário de acesso.

A seguir apresentamos o arquivo *test.yml* que representa um teste de verificação, observando:

- se o SO instalado é o CentOS;

- se a versão do SO é a 7.

```
1  - name: tests
2    hosts: nginx
3    become: yes
```

```
  gather_facts: no
  tasks:
    - name: registrando a versao do sistema operacional
      shell: cat /etc/os-release
      register: os_check

    - name: confirmando instalacao do centos
      assert:
        that: "'NAME=\"CentOS Linux\"' in os_check.stdout"

    - name: confirmando versao 7 do centos
      assert:
        that: "'VERSION_ID=\"7\"' in os_check.stdout"
```

Perceba que antes da task de verificação há sempre uma task para registrar o output em algum arquivo. As tasks de verificação fazem as asserções nos conteúdos desses arquivos para saber se eles contêm o resultado esperado, caso contrário o teste falha.

Já o arquivo *validate.yml* representa um teste de **validação**, observando:

- se o pacote do banco de dados está instalado;
- se o arquivo das permissões de acesso foi criado;
- se o arquivo de configuração do banco foi criado;
- se o serviço do banco de dados está em execução;
- se o banco de dados **sespmtdb** foi criado;
- se o usuário **sespmt** foi criado para acesso ao banco de dados.

```
- name: tests
  hosts: postgresql
  become: yes
  gather_facts: no
  tasks:
    - name: checando a instalacao do pacote postgresql14
      shell: "yum list installed | grep postgres"
      args:
        warn: false
      register: package_check
      changed_when: package_check.rc != 0

    - name: confirmando a instalacao do pacote
      assert:
        that: "'postgresql14' in package_check.stdout"

    - name: checando permissoes
      shell: "ls /var/lib/pgsql/14/data/pg_hba.conf"
      args:
        warn: false
      register: permissions_check
      changed_when: permissions_check.rc != 0

    - name: confirmando permissoes
      assert:
        that: "'pg_hba.conf' in permissions_check.stdout"

```

```
28      - name: checando configuracoes
29        shell: "ls /var/lib/pgsql/14/data/postgresql.conf"
30        args:
31          warn: false
32        register: configurations_check
33        changed_when: configurations_check.rc != 0
34
35      - name: confirmando configuracoes
36        assert:
37          that: "'postgresql.conf' in configurations_check.stdout"
38
39      - name: checando status do servico
40        shell: "service postgresql-14 status"
41        args:
42          warn: false
43        register: status_check
44        changed_when: status_check.rc != 0
45
46      - name: confirmando o status do servico
47        assert:
48          that: "'active (running)' in status_check.stdout"
49
50      - name: checando criacao do banco de dados
51        shell: "psql -U postgres -c \"\\l\""
52        args:
53          warn: false
54        register: db_schema_check
55        changed_when: db_schema_check.rc != 0
56
57      - name: confirmando criacao do banco de dados
58        assert:
59          that: "'sespmtdb' in db_schema_check.stdout"
60
61      - name: checando criacao do usuario de banco de dados
62        shell: "psql -U postgres -c \"\\du\""
63        args:
64          warn: false
65        register: db_user_check
66        changed_when: db_user_check.rc != 0
67
68      - name: confirmando criacao do usuario de banco de dados
69        assert:
70          that: "'sespmt' in db_user_check.stdout"
```

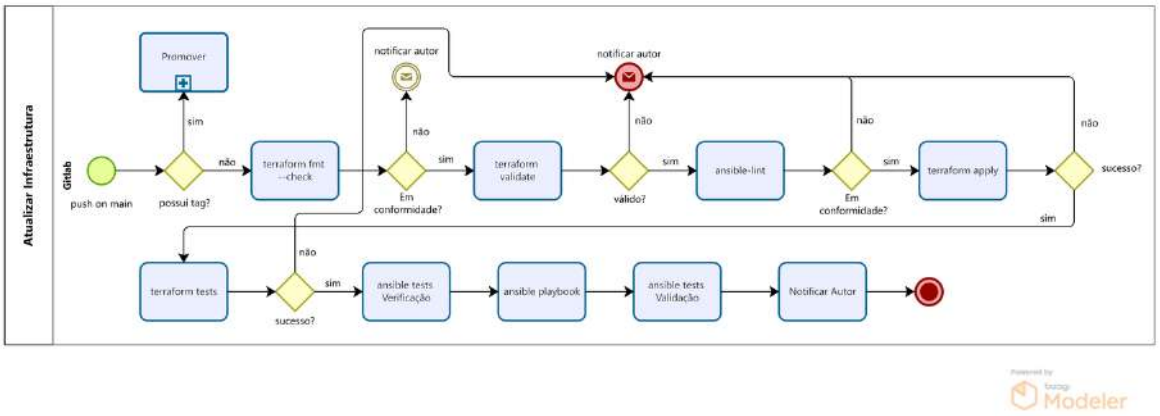
Da mesma forma temos sempre uma task de registro da saída precedendo a task de asserção que verifica o resultado.

8.3 Processo de Teste

O processo de teste é iniciado de forma automatizada toda vez que uma nova versão da infraestrutura é efetivada no repositório central. A figura 15 ilustra essa pipeline no Gitlab.

A primeira etapa do processo consiste em verificar se é uma alteração em release ou a promoção das configurações para o ambiente de produção, isso é verificado por meio de uma tag de versionamento, toda vez que o repositório é versionado semanticamente entende-se que as configurações foram verificadas e validades e podem ser promovidas para o ambiente de produção.

Figura 15 – Pipeline do Gitlab CI



As demais etapas envolvem checagens estáticas e dinâmicas para os códigos do Terraform e do Ansible, bem como a execução de testes automatizados.

A pipeline é codificada no Gitlab CI/CD por meio do arquivo *.gitlab-ci.yml*. Esse arquivo YAML é onde definimos a estrutura e ordem dos *jobs* que o runner deverá executar, bem como as decisões que o runner deve tomar quando determinadas condições ocorrerem.

Nesse arquivo, podemos verificar já no início que as tarefas (*jobs*) a serem executadas estão organizadas em estágios a serem executados na sequência declarada, sendo estes executados baseados na imagem *docker* que possui terraform instalado:

```
stages:
  - validate
  - test
  - build
  - deploy
  - cleanup

image:
  name: registry.gitlab.com/gitlab-org/terraform-images/stable:latest
...
```

Temos também no código a configuração da localização dos nossos scripts terraform, ansible e arquivos de estado, bem como uma pré-configuração por meio de script para habilitar o acesso ao SSH.

```
...
variables:
  TF_ROOT: ${CI_PROJECT_DIR}/terraform  # The relative path to the root directory of
      the Terraform project
  AB_ROOT: ${CI_PROJECT_DIR}/ansible  # The relative path to the root directory of the
       Ansbile project
  TF_STATE_NAME: ${TF_STATE_NAME:-default}  # The name of the state file used by the
      GitLab Managed Terraform state backend

before_script:
  - export TF_WORKSPACE="release"
  - 'command -v ssh-agent >/dev/null || ( apk update && apk add openssh )'
  - eval $(ssh-agent -s)
  - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
  - mkdir -p ~/.ssh
```

```
13   - chmod 700 ~/.ssh
14   - touch ~/.ssh/known_hosts
15   - chmod 700 ~/.ssh/known_hosts
16   - echo "$SSH_KNOWN_HOSTS" >> ~/.ssh/known_hosts
17 ...
```

No estágio validate temos dois jobs, sendo que o primeiro (*fmt*) aplica formatação ao código Terraform e o segundo (*validate tf*) verifica se a formatação foi realizada. Esses dois jobs não interrompem a *pipeline* em caso de falha, tendo em vista que servem apenas para notificar a equipe de códigos que estejam sendo enviados sem formatação.

```
1 ...
2 fmt:
3   stage: validate
4   script:
5     - cd "${TF_ROOT}"
6     - gitlab-terraform fmt
7   allow_failure: true
8
9 validate tf:
10  stage: validate
11  script:
12    - cd "${TF_ROOT}"
13    - gitlab-terraform fmt -check
14  allow_failure: true
15 ...
```

No etágio *build* temos um job (**build tf**) responsável por executar o *plan* do Terraform sobre as modificações que serão realizadas e armazená-lo. Neste caso, uma falha interrompe a pipeline e impede que as modificações sejam aplicadas.

```
1 ...
2 build tf:
3   stage: build
4   script:
5     - cd "${TF_ROOT}"
6     - gitlab-terraform plan
7     - gitlab-terraform plan-json
8   resource_group: ${TF_STATE_NAME}
9   artifacts:
10    paths:
11      - ${TF_ROOT}/plan.cache
12    reports:
13      terraform: ${TF_ROOT}/plan.json
14 ...
```

No estágio *deploy* temos um job (**deploy tf**) responsável por criar um usuário, realizar o download da imagem CentOS que será utilizada pelos scripts do terraform para criação das VMs, podendo então realizar o *terraform apply* para modificar a infraestrutura.

```
1 ...
2 deploy tf:
3   stage: deploy
4   script:
5     - apk update
6     - apk add --upgrade xorriso
7     - adduser -D leonardo
8     - mkdir /home/leonardo/Downloads
9     - cd /home/leonardo/Downloads
```

```
    - wget https://cloud.centos.org/centos/7/images/CentOS-7-x86_64-GenericCloud-2111.
        qcow2c
    - cd "${TF_ROOT}"
    - gitlab-terraform apply
  resource_group: ${TF_STATE_NAME}
  dependencies:
    - build tf
...
```

O estágio *cleanup* só pode ser executado de forma manual, ou seja, não faz parte das etapas automatizadas da pipeline ao efetuar um *commit*. Nesse estágio temos o job (**destroy tf**) tem por objetivo destruir a infraestrutura.

```
...
destroy tf:
  stage: cleanup
  script:
    - cd "${TF_ROOT}"
    - gitlab-terraform workspace select release
    - gitlab-terraform destroy
  resource_group: ${TF_STATE_NAME}
  when: manual
  dependencies:
    - deploy tf
...
```

8.4 Implementação

Alguns processos foram essenciais para o funcionamento do ambiente e os detalhes específicos desta implementação serão tratados a seguir a seguir.

8.4.1 Cluster do Kubernetes

A configuração do cluster do Kubernetes foi realizada utilizando a ferramenta kind. Seu funcionamento utiliza containers Docker e nos permite ter um controle maior de alguns recursos, quando comparado à outras ferramentas similares, como o minikube.

8.4.1.1 Instalação e configuração do kind

A instalação do kind foi incorporada ao playbook *tools.yml* com a role *instalar_kind* e sua configuração é responsabilidade da role *configurar_cluster*.

A estratégia utilizada para configuração foi a criação e execução de um script de definição da estrutura do cluster, que pode ser visualizado a seguir:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
```

```
        node-labels: "ingress-ready=true"
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
- role: control-plane
- role: worker
- role: worker
```

Este script cria uma estrutura com dois *controllers* e dois *workers*, além de atribuir as portas **80** e **443** às portas equivalentes do host, possibilitando assim um futuro acesso à ele por meio do acesso à estas portas.

8.4.1.2 Configuração do Ingress

Uma das estratégias utilizadas para acesso externo ao cluster do Kubernetes é a criação e configuração do Ingress.

No protótipo funcional, por estarmos utilizando o kind, a configuração do Ingress segue os passos sugeridos na própria documentação da ferramenta. A implementação escolhida do Ingress neste cenário foi a do Ingress Nginx.

Esta configuração foi incorporada ao *playbook* tools com a role **configurar_ingress_default**, e tem por objetivo definir todos os contextos necessários para acesso externo das aplicações contidas no cluster.

Um exemplo do arquivo de configuração do Ingress pode ser visualizado abaixo:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: default-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/rewrite-target: "/"
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
  - host: "localhost"
    http:
      paths:
      - pathType: Prefix
        path: "/teste-versao"
        backend:
          service:
            name: teste-versao-service
            port:
              number: 8001
```

8.4.2 Argo CD

O Argo CD é utilizado neste ambiente e atua como o controller para o Kubernetes. Sua instalação consiste basicamente na criação de um *namespace* específico dentro do cluster e na aplicação do respectivo *script*, já incorporado no *playbook* de instalação das ferramentas:

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

Com isso, os artefatos do Argo CD serão criados dentro do cluster e estarão prontos para serem personalizados conforme a necessidade do projeto.

8.4.2.1 Configuração das chaves SSH

No protótipo funcional, devido ao fato do código estar disponível em um diretório privado, é necessário realizar a configuração da chave de acesso. Este processo consiste em três passos:

1. Geração do par de chaves SSH
2. Configuração da chave SSH no repositório
3. Configuração do repositório no Argo CD

8.4.2.2 Configuração do certificado

Entre os artefatos configurados durante a instalação do Argo CD está o serviço **argocd-server**. Para possibilitar o funcionamento correto deste serviço é necessário que seja feita a configuração de um certificado válido.

No protótipo funcional esta configuração foi realizada utilizando o Certbot. Para isso, o subdomínio argocd foi criado, e a configuração foi realizada no Nginx de cada ambiente, conforme exemplo a seguir:

```
server {

    server_name argocd.fabricadesoftwareifmt.com.br;

    client_max_body_size 50m;
    location / {

        proxy_set_header Host argocd;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Port $server_port;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_pass https://172.16.1.226:8080;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_read_timeout 900s;
    }

    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/argocd.fabricadesoftwareifmt.com.br/fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/argocd.fabricadesoftwareifmt.com.br/privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}

## outras configuracoes

```

```
    server {
        if ($host = argocd.fabricadesoftwareifmt.com.br) {
            return 301 https://$host$request_uri;
        } # managed by Certbot

        listen 80;

        server_name argocd.fabricadesoftwareifmt.com.br;
        return 404; # managed by Certbot
    }
```

8.4.3 Estrutura de diretórios do Kubernetes

Com o Argo CD é possível configurar as aplicações de forma declarativa por meio da criação de artefatos do tipo Application.

No protótipo funcional, um exemplo da definição deste artefato pode ser visualizado na role **configurar_teste_versao**. Esta role é responsável por aplicar a *Application* definida a seguir:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: teste-versao
  namespace: argocd
spec:
  destination:
    namespace: default
    server: https://kubernetes.default.svc
  project: default
  source:
    path: kubernetes/applications/teste-versao
    repoURL: git@github.com:jppreti/entrega-continua.git
    targetRevision: HEAD
  syncPolicy:
    syncOptions:
    - CreateNamespace=true
    automated:
      selfHeal: true
      prune: true
```

Nesta definição, é necessário informar no parâmetro **path** o caminho onde as definições do **Deployment** e do Service referentes à esta aplicação estão contidos, dentro do diretório informado no parâmetro **repoURL**.

Por este motivo, definimos uma estratégia para oragnização dos diretórios e arquivos de *deploy* de cada aplicação, que consiste na seguinte estrutura:

```
entrega-continua
  |- kubernetes
    |- applications
      |- autenticacao
        |- deployment.yml
      |- registro-geral
      |- registro-geral-web
      |- teste-versao
    |- tools
      |- keycloak
      |- phpldapadmin
      |- rabbitmq
```

```
13        |- redis
```

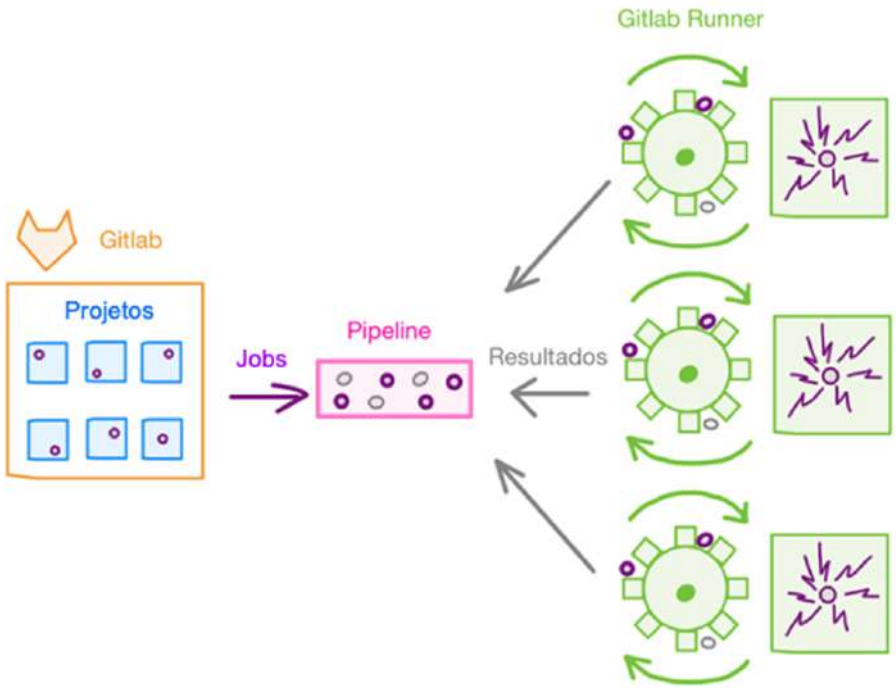
8.5 PIPELINE Gitlab

O objetivo de se configurar uma PIPELINE é a automatização e padronização do processo de provisionamento e configuração do ambiente. As configurações mencionadas nesse documento foram configuradas para serem executadas no Gitlab CI/CD sempre que houver uma alteração na infraestrutura, possibilitando assim a execução planejada de uma nova versão da infraestrutura.

Para configurar a PIPELINE do GitLab é necessário configur o arquivo **.gitlab-ci.yml** e colocar na raiz do repositório. Esse arquivo cria um pipeline, que é executa alterações na infraestrutura a partir do código do repositório. Os pipelines consistem em um ou mais estágios (stages) executados em ordem e cada um pode conter uma ou mais tarefas (jobs) executadas em paralelo. Esses jobs (ou scripts) são executados pelo agente GitLab Runner. Em um ambiente "Self-hosted"e necessário a configuração de um Runner para executar as pipelines.

Os runners executam o código definido em **.gitlab-ci.yml**. Um runner é um agente leve e altamente escalonável que pega um job de CI por meio da API do GitLab CI / CD, executa o job e envia o resultado de volta a instância do GitLab.

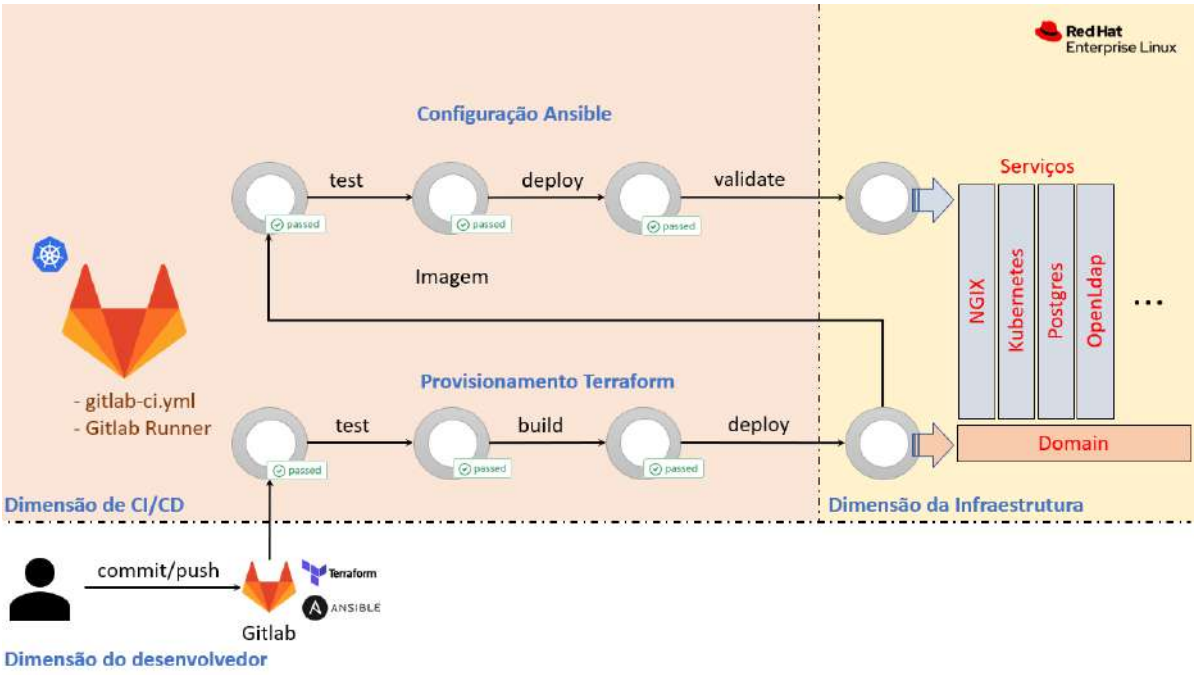
Figura 16 – Gitlab CI/CD



8.5.1 Stages e Jobs da pipeline do projeto

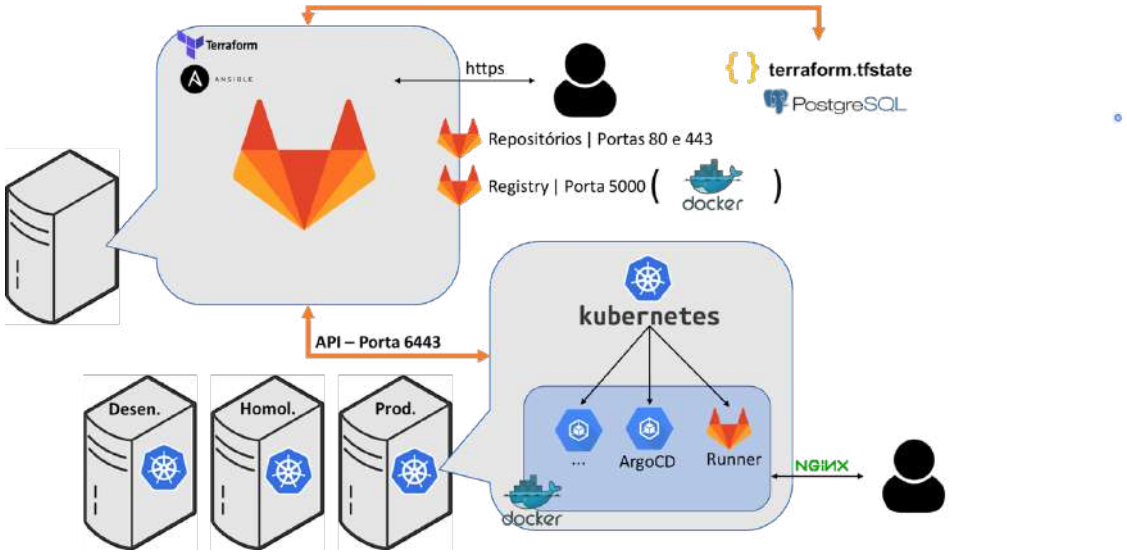
A PIPELINE foi configurada para provisionar a infraestrutura e configurar os serviços e aplicações conforme modelo apresentado na Figura 17.

Figura 17 – PIPELINE de Infraestrutra como Código



Esse modelo provisiona incialmente toda a infraestrutura na nuvem necessária para que os serviços e sistemas sejam configurados posteriormente. A Figura 18 apresenta a infraestrutura provisionada pela PIPELINE que foi configurada:

Figura 18 – Infraestrutura provisionada e configurada pela PIPELINE



8.5.1.1 Provisionamento da infraestrutura

O provisionamento é realizado pelo Terraform e possui os seguintes stages e jobs:

- **Test terraform**
  - **validate tf** - utilizado para checar a formatação dos arquivos e garantir que a pipeline não quebre por erros de identação e falta de arquivos.

  - **test terratest** - executa os testes de unidade utilizando a ferramenta Terratest
  - **test terraform** - executa os testes de unidade utilizando o próprio Terraform
- **Build terraform**
  - **build tf** - realiza o planejamento da infraestrutura a ser provisionada
- **Deploy**
  - **deploy tf** - provisiona a infraestrutura na nuvem
- **Cleanup terraform**
  - **destroy tf** - job manual utilizado para destruir a infraestrutura na nuvem

8.5.1.2 Configuração da infraestrutura e serviços

A **configuração** é realizada pelo Ansible e possui os seguintes *stages*:

- **Test**
  - **test ansible** - realiza os testes preliminares com os hosts previamente provisionados
- **Deploy**
  - **deploy ansible** - realiza as tarefas declaradas no Ansbible
- **Validate**
  - **validate ansible** - realiza a validação das configurações do executadas pelo Ansible

8.5.2 Configuração da PIPELINE do projeto

O arquivo abaixo apresenta as configurações da PIPELINE para o provisionameto e configuração da infraestrutura:

```
stages:
  - test terraform
  - build terraform
  - deploy terraform
  - test ansible
  - deploy ansible
  - validate
  - cleanup terraform

image:
  name: leonardolbraun/gitlab-terraform-golang
  entrypoint:
      - "/usr/bin/env"
      - "PATH=/go/bin:/usr/local/go/bin/:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr
        /bin:/sbin:/bin"

variables:
  TF_ROOT: ${CI_PROJECT_DIR}/terraform  # The relative path to the root directory of
      the Terraform project
```

```
18   AB_ROOT: ${CI_PROJECT_DIR}/ansible  # The relative path to the root directory of the
         Ansbile project
19   TF_CACHE_KEY: ${TF_ROOT}-cache
20   TF_STATE_NAME: ${TF_STATE_NAME:-default}  # The name of the state file used by the
         GitLab Managed Terraform state backend
21
22 before_script:
23   - export TF_WORKSPACE="release"
24   - 'command -v ssh-agent >/dev/null || ( apk update && apk add openssh )'
25   - eval $(ssh-agent -s)
26   - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
27   - mkdir -p ~/.ssh
28   - chmod 700 ~/.ssh
29   - touch ~/.ssh/known_hosts
30   - chmod 700 ~/.ssh/known_hosts
31   - echo "$SSH_KNOWN_HOSTS" >> ~/.ssh/known_hosts
32
33 test terratest:
34   image:
35       name: leonardolbraun/terraform-golang
36       entrypoint: [""]
37   stage: test terraform
38   script:
39     - cd "${TF_ROOT}"/
40     - terraform fmt -check
41     - apk update
42     - apk add --upgrade xorriso
43     - adduser -D leonardo
44     - mkdir /home/leonardo/Downloads
45     - cd /home/leonardo/Downloads
46     - wget https://cloud.centos.org/centos/7/images/CentOS-7-x86_64-GenericCloud-2111.
         qcow2c
47     - cd "${TF_ROOT}"/tests
48     - echo $PATH
49     - go test
50
51 test terraform:
52   image:
53       name: leonardolbraun/terraform-golang
54       entrypoint: [""]
55   stage: test terraform
56   script:
57     - cd "${TF_ROOT}"/
58     - terraform test
59
60 build tf:
61   stage: build terraform
62   script:
63     - cd "${TF_ROOT}"
64     - gitlab-terraform plan
65     - gitlab-terraform plan-json
66   resource_group: ${TF_STATE_NAME}
67   artifacts:
68     paths:
69       - ${TF_ROOT}/plan.cache
70     reports:
71       terraform: ${TF_ROOT}/plan.json
72
73 deploy tf:
74   stage: deploy terraform
75   script:
```

```
76      - apk update
77      - apk add --upgrade xorriso
78      - adduser -D leonardo
79      - mkdir /home/leonardo/Downloads
80      - cd /home/leonardo/Downloads
81      - wget https://cloud.centos.org/centos/7/images/CentOS-7-x86_64-GenericCloud-2111.
            qcow2c
82      - cd "${TF_ROOT}"
83      - gitlab-terraform apply
84    resource_group: ${TF_STATE_NAME}
85    dependencies:
86      - build tf
87
88  destroy tf:
89    stage: cleanup terraform
90    script:
91      - cd "${TF_ROOT}"
92      - gitlab-terraform workspace select release
93      - gitlab-terraform destroy
94    resource_group: ${TF_STATE_NAME}
95    when: manual
96    dependencies:
97      - deploy tf
98
99  test ansible:
100   image: leonardolbraun/ansible
101   stage: test ansible
102   script:
103     - echo "$SSH_PRIVATE_KEY_ANSIBLE" | tr -d '\r' | ssh-add -
104     - echo "$SSH_KNOWN_HOSTS_ANSIBLE" >> ~/.ssh/known_hosts
105     - export ANSIBLE_HOST_KEY_CHECKING=False
106     - cd "${AB_ROOT}"
107     #- ansible-lint -p tools.yml -c .ansible-lint
108     - ansible-playbook -i environments/release/hosts test.yml
109   resource_group: ${TF_STATE_NAME}
110   dependencies:
111     - deploy tf
112
113 deploy ansible:
114   image: leonardolbraun/ansible
115   stage: deploy ansible
116   script:
117     - echo "$SSH_PRIVATE_KEY_ANSIBLE" | tr -d '\r' | ssh-add -
118     - export ANSIBLE_HOST_KEY_CHECKING=False
119     - cd "${AB_ROOT}"
120     - ansible-playbook -i environments/release/hosts tools.yml -e ansible_environment=
            release
121   dependencies:
122     - deploy tf
123
124 validate ansible:
125   image: leonardolbraun/ansible
126   stage: validate
127   script:
128     - echo "$SSH_PRIVATE_KEY_ANSIBLE" | tr -d '\r' | ssh-add -
129     - echo "$SSH_KNOWN_HOSTS_ANSIBLE" >> ~/.ssh/known_hosts
130     - export ANSIBLE_HOST_KEY_CHECKING=False
131     - cd "${AB_ROOT}"
132     - ansible-playbook -i environments/release/hosts validate.yml
133   resource_group: ${TF_STATE_NAME}
```

Referências

[1] Complete guide for picking the right tool for Terraform Security Code Analysis

[2] 6 Kubernetes Testing Tools to Use in Your DevSecOps Pipelines

[3] Introduction to ansible-test

[4] https://www.meshcloud.io/2020/03/13/testing-infrastructure-as-code/

[5] Terraform Infrastructure-as-Code testing: best-practice unit tests & BDD end-to-end scenario...

[6] kind

[7] Ingress

[8] kind – Ingress

[9] Argo CD - Declarative GitOps CD for Kubernetes

[10] Use SSH keys to communicate with GitLab — GitLab

[11] Private Repositories - Argo CD - Declarative GitOps CD for Kubernetes

[12] Certbot Instructions

[13] Declarative Setup - Argo CD - Declarative GitOps CD for Kubernetes

[14] Post — Develatio's Blog

CAPÍTULO 9

Monitoramento

Este capítulo apresenta os principais conceitos sobre monitoramento de sistemas computacionais, bem como possíveis métricas, rastreabilidade, ferramentas de monitoramento e as configurações realizadas no protótipo funcional para a implementação do monitoramento na infraestrutura como código.

9.1 Introdução

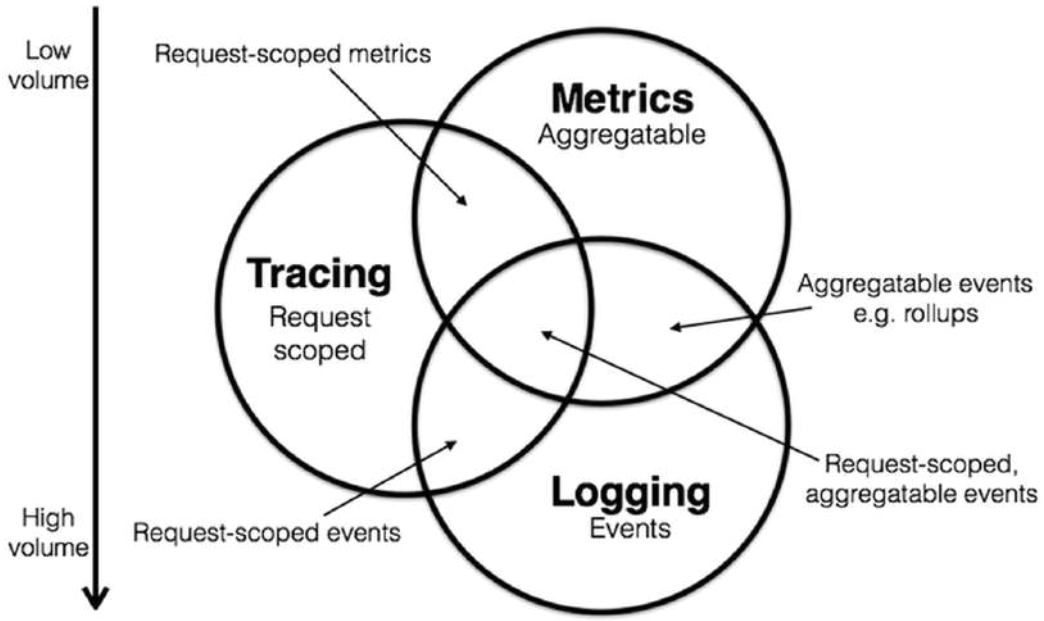
Definimos monitoramento como um processo de captura do comportamento de um sistema por meio de verificações de integridade e métricas ao longo do tempo. Isso ajuda a detectar anomalias: quando o sistema está indisponível, apresenta uma carga incomum, esgotamento de certos recursos ou de outra forma não se comporta dentro de seus parâmetros normais (esperados). O monitoramento envolve a coleta e o armazenamento de métricas de longo prazo, o que é importante, mais do que a detecção de anomalias, mas também a análise da causa desde sua raiz, detecção de tendências e planejamento de capacidade.

Estamos acostumados a monitorar um sistema por vez, quando precisamos olhar um sistema diferente mudamos para outro monitor ou ferramenta. No entanto, com o crescimento do número de recursos a serem observados, faz-se necessário a integração desses dados de monitoramento para facilitar e agilizar o acompanhamento.

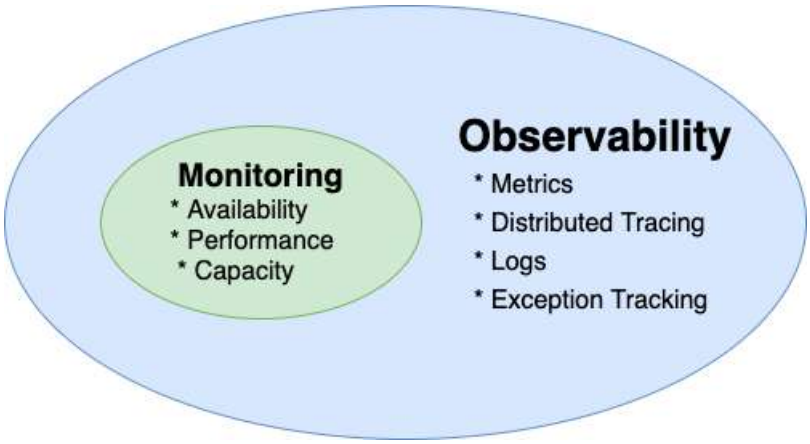
Isso não quer dizer que não é importante poder monitorar um aspecto ou recurso, mas que atuar apenas neste nível pode tornar inviável o processo de monitoramento conforme a quantidade de recursos cresce ao longo do tempo.

Compreender o estado dos sistemas é essencial para manter a estabilidade e a confiabilidade dos serviços. Informações sobre a saúde e a performance do sistema ajuda não apenas na realização de medidas quando da ocorrência de uma falha, mas também ajuda a equipe a realizar modificações na infraestrutura com mais seguraça e confiança. E para contribuir com ações mais pró-ativas da equipe, faz-se necessário coletar métricas ao longo do tempo para poder realizar projeções.

9.2 Observability



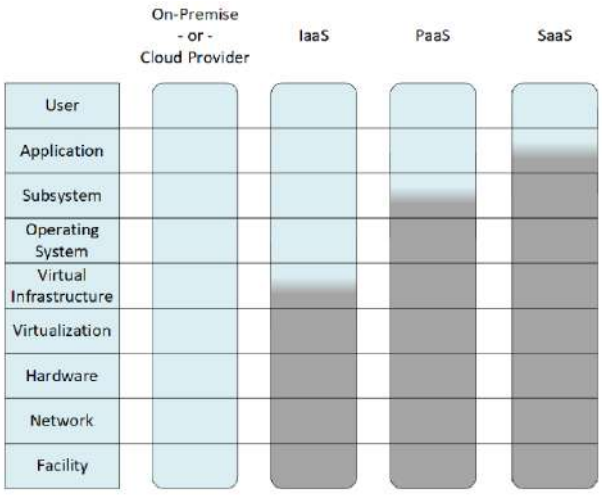
9.2.1 Observability vs. Monitoramento



9.2.2 Modelo de Monitoramento em Camadas

Como podemos observar na figura abaixo, o monitoramento é uma atividade complexa que vai desde o acesso às instalações físicas (camada mais baixa) até o usuário (camada mais alta).

Para estarmos alinhados com o foco do projeto o objetivo é monitorar para garantir a saúde das máquinas virtuais, cluster, contêineres e os serviços provisionados e configurados com Terraform e Ansible.



Daubner, Lukás. Effective computer infrastructure monitoring. Dissertação de mestrado. Universidade Masaryk, 2018.

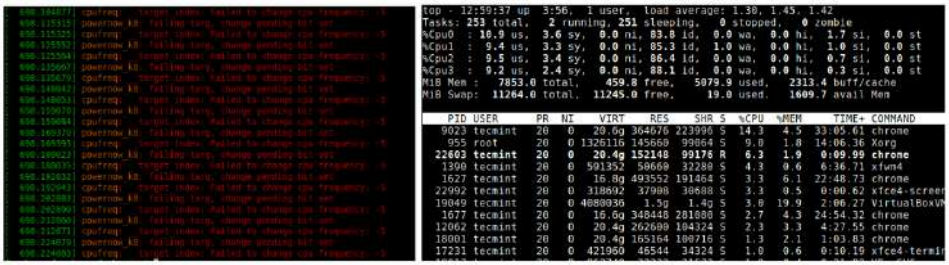
9.2.2.1 Qual a importânia dos logs?

- Network
- Hosts
- Virtualização
- Conteinerização
- Serviços
- Aplicações

9.2.2.2 Formatos de Logs

docker ps / docker stats

- Descentralizado
- Sem histórico
- Monitoramento reativo
- Não é possível escala



```
top - 12:59:37 up  3:56,  1 user,  load average: 1.30, 1.45, 1.42
Tasks: 253 total,   2 running, 251 sleeping,   0 stopped,   0 zombie
%Cpu0  : 10.9 us,  3.6 sy,  0.0 ni, 83.8 id,  0.0 wa,  0.0 hi,  1.7 si,  0.0 st
%Cpu1  :  9.4 us,  3.3 sy,  0.0 ni, 85.3 id,  1.0 wa,  0.0 hi,  1.0 si,  0.0 st
%Cpu2  :  9.5 us,  3.4 sy,  0.0 ni, 86.4 id,  0.0 wa,  0.0 hi,  0.7 si,  0.0 st
%Cpu3  :  9.2 us,  2.4 sy,  0.0 ni, 88.1 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
MiB Mem :   7853.0 total,    459.8 free,   5079.9 used,   2313.4 buff/cache
MiB Swap:  11264.0 total,  11245.0 free,     19.0 used.   1609.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   9023 tecmint   20   0   20.6g 364676 223996 S  14.3   4.5  33:05.61 chrome
    955 root      20   0 1326116 145660  99064 S   9.0   1.8  14:06.36 Xorg
  22803 tecmint   20   0   20.4g 152148  99176 R   6.3   1.9   0:00.99 chrome
   1300 tecmint   20   0  591352  58680  32280 S   4.3   0.6   6:36.71 xfwm4
   1627 tecmint   20   0   16.8g 493552 191464 S   3.3   6.1  22:48.73 chrome
  22992 tecmint   20   0  318692  37908  30688 S   3.3   0.5   0:00.62 xfce4-screen
  19049 tecmint   20   0 4880036   1.5g   1.4g S   3.0  19.9   2:06.27 VirtualBoxVM
   1677 tecmint   20   0   16.8g 348448 281880 S   2.7   4.3  24:54.32 chrome
  12062 tecmint   20   0   20.4g 262600 184324 S   2.3   3.3   4:27.55 chrome
  18001 tecmint   20   0   20.4g 165164 100716 S   1.3   2.1   1:03.83 chrome
  17231 tecmint   20   0  421960  46544  34324 S   1.0   0.6   0:10.19 xfce4-termin
```

```
CONTAINER ID   NAME                    CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O         PIDS
6dabef87438b   hops_app_1              0.01%     125.8MiB / 31.3GiB    0.39%     1.1GB / 151MB     30.2MB / 0B       13
e10951cfef27   sec0419b_mysql_1        0.11%     234.1MiB / 31.3GiB    0.73%     7.42MB / 7.23MB   52MB / 39.5MB     36
5df760550484   sec0419b_app_1          0.01%     55.72MiB / 31.3GiB    0.17%     8.14MB / 6.47MB   50.8MB / 2.11MB   7
4c18d0250bce   hops_mysql_1            0.32%     912.6MiB / 31.3GiB    2.85%     292MB / 2.21GB    86.1MB / 3.42GB   38
5cac82e13be5   portainer_portainer_1   0.03%     26.99MiB / 31.3GiB    0.08%     1.56MB / 6.07MB   36MB / 23.5MB     16
e201222e0859   brt1119b_mysql_1        0.40%     446.2MiB / 31.3GiB    1.39%     5.12MB / 8.33MB   119MB / 72.4MB    45
f3fcb9e08e13   brt1119b_app_1          0.01%     96.42MiB / 31.3GiB    0.30%     9.75MB / 9.11MB   70.5MB / 1.93MB   12
b8bb07d57869   heron-web_app_1         0.01%     8.527MiB / 31.3GiB    0.03%     961kB / 0B        4.7MB / 0B        82
24208d8c94ec   atp0820_app_1           0.01%     24.04MiB / 31.3GiB    0.07%     961kB / 0B        34.1MB / 0B       6
```


CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
6dabef87438b	hops_app_1	0.01%	125.8MiB / 31.3GiB	0.39%	1.1GB / 151MB	30.2MB / 0B	13
e10951cfef27	sec0419b_mysql_1	0.11%	234.1MiB / 31.3GiB	0.73%	7.42MB / 7.23MB	52MB / 39.5MB	36
5df760550484	sec0419b_app_1	0.01%	55.72MiB / 31.3GiB	0.17%	8.14MB / 6.47MB	50.8MB / 2.11MB	7
4c18d0250bce	hops_mysql_1	0.32%	912.6MiB / 31.3GiB	2.85%	292MB / 2.21GB	86.1MB / 3.42GB	38
5cac82e13be5	portainer_portainer_1	0.03%	26.99MiB / 31.3GiB	0.08%	1.56MB / 6.07MB	36MB / 23.5MB	16
e201222e0859	brt1119b_mysql_1	0.40%	446.2MiB / 31.3GiB	1.39%	5.12MB / 8.33MB	119MB / 72.4MB	45
f3fcb9e08e13	brt1119b_app_1	0.01%	96.42MiB / 31.3GiB	0.30%	9.75MB / 9.11MB	70.5MB / 1.93MB	12
b8bb07d57869	heron-web_app_1	0.01%	8.527MiB / 31.3GiB	0.03%	961kB / 0B	4.7MB / 0B	82
24208d8c94ec	atp0820_app_1	0.01%	24.04MiB / 31.3GiB	0.07%	961kB / 0B	34.1MB / 0B	6

9.2.2.3 Desafios dos Logs

- Formato do Log não é consistente
- Necessidade de apoio especializado
- Número de recursos
- Analisar os logs em diferentes formatos

"Qual foi o throughput durante a janela de backup ontem a noite? Prejudicou o processamento do ambiente?"

9.3 Métricas

Métrica é o componente primário a ser coletado e processado pelas ferramentas de monitoramento, o que permite construir uma visão coesa daquilo que está sendo monitorado. Uma etapa importante envolve justamente saber quais componentes monitorar e quais características devem ser observadas.

Neste projeto o foco está no monitoramento realizado pela equipe de operações (Ops team), ou seja, o monitoramento de recursos de um bare metal, máquinas virtuais ou contêineres. Apesar desses elementos serem diferentes, a maioria das métricas que se coletam de um contêiner, por exemplo, podem ser coletadas de uma VM ou um bare metal.

Métrica é uma **representação numérica** medida em um **intervalo de tempo**.

Vislumbramos os seguintes recursos a serem monitorados no ambiente do protótipo funcional:

- **Bare Metal**;
- **Virtual Machine**;
- **Cluster**;
- **Container**.

Quanto as métricas a serem coletadas, elas estão relacionadas a:

- **Memória**;
- **CPU**;
- **Disco**;
- **Tráfego de rede**.

E podem estar relacionadas aos tipos:

- **Saturação**: quando um recurso tem carga maior do que pode suportar;
- **Performance**: valor estabelecido em um intervalo de tempo;

- **Utilização**: percentual ou valor nominal em um instante no tempo;
- **Disponibilidade**: percentual ou valor nominal de recursos livres para uso;
- **Taxa de Transferência**: número médio de volume de dados por unidade de tempo.

9.3.1 CPU

Temos duas métricas de CPU a serem consideradas: CPU física (pCPU) e virtual (vCPU), a primeira referente ao número de processadores disponível em um bare metal e a segunda referente ao número de processadores lógicos que foram associados a uma máquina virtual.

Enquanto o número de vCPUs alocadas a todas as VMs for igual ou menor ao número de pCPUs disponível temos uma situação muito confortável, tendo em vista que os recursos estão subutilizadas já que é improvável que todas as VMs demandem 100% das vCPUs ao mesmo tempo. No entanto, é importante compreender que conforme essa razão aumente, mais as vCPUs terão que aguardar para acessar uma pCPU. É difícil estabelecer um valor para essa razão porque depende da natureza das aplicações sendo executadas em cada VM, mas é importante a sua observação já que contribui significativamente na degradação da performance e vai ajudar a determinar se o ambiente virtualizado deve escalar (para cima ou para baixo) e nas configurações de limites das VMs.

Na tabela a seguir apresentamos algumas métricas:

Métrica	Descrição	Tipo
Readiness	Percentual médio do tempo que uma VM fica em estado de pronto (ready), esperando para acessar uma pCPU.	Saturação
Wait	Total do tempo (ms) que uma a VM está esperando (i.e., VM tem acesso a CPU mas está aguarando a espera de outros recursos ou a finalização de outras operações)	Performance
Usage	Percentual da capacidade das pCPUs sendo utilizadas pelas VMs no servidor.	Utilização
Total Capacity	Capacidade total das pCPUs (MHz) de um servidor disponível para as VMs.	Disponibilidade

9.3.2 Memória

Assim como na CPU na memória também temos duas métricas a serem consideradas: memória física (pMem), disponível para o ambiente de virtualização e memória virtual (vMem), disponível para a VM.

9.3.3 Disco

9.3.4 Tráfego de Rede

Indicador de Desempenho

Métrica	Descrição	Tipo
Swap IN	Quantidade de memória principal (i.e., KiB) que o servidor devolve para a VM a partir da memória secundária.	Saturação
Swap OUT	Quantidade de memória principal (i.e., KiB) que o servidor retira da VM para a memória secundária.	Saturação
Physical Usage	Percentual da capacidade da pMem sendo utilizadas pelas VMs no servidor.	Utilização
Virtual Usage	Percentual da capacidade da vMem sendo utilizada pela VM.	Utilização
Total Capacity	Capacidade total de pMem (i.e., MiB) de um servidor disponível para as VMs.	Disponibilidade

Métrica	Descrição	Tipo
Physical Provisioned	Espaço físico disponível para todas as VMs	Disponibilidade
Virtual Provisioned	Espaço disponível para a VM	Disponibilidade
Physical Usage	Espaço físico utilizado no servidor	Utilização
Virtual Usage	Espaço utilizado pela VM	Utilização
Total Latency	Tempo médio (ms) que o servidor leva para processar um comando da VM	Performance
Read Latency	Tempo médio (ms) que o servidor leva realizar uma operação de leitura	Performance
Write Latency	Tempo médio (ms) que o servidor leva realizar uma operação de escrita	Performance
Usage	Volume médio de operações de I/O (i.e., KB/s)	Taxa de Transferência

Métrica	Descrição	Tipo
Physical Provisioned	Espaço físico disponível para todas as VMs	Disponibilidade
Virtual Provisioned	Espaço disponível para a VM	Disponibilidade
Physical Usage	Espaço físico utilizado no servidor	Utilização
Virtual Usage	Espaço utilizado pela VM	Utilização
Total Latency	Tempo médio (ms) que o servidor leva para processar um comando da VM	Performance
Read Latency	Tempo médio (ms) que o servidor leva realizar uma operação de leitura	Performance
Write Latency	Tempo médio (ms) que o servidor leva realizar uma operação de escrita	Performance
Usage	Volume médio de operações de I/O (i.e., KB/s)	Taxa de Transferência

Existe diferença entre métrica e indicador de desempenho. A métrica é um tipo de indicador que possui uma medida mais bruta, ou seja, possui um formato simples com indicação de valores e quantidades.

Já o indicador de desempenho realiza um cálculo mais minucioso e complexo, apresentando números e suas variáveis, com um resultado mais alinhado com os objetivos do negócio, além do seu histórico evolutivo.

9.4 Log

Log está relacionado ao registro de eventos relevantes em um sistema computacional. Esse registro pode ser utilizado para restabelecer o estado original de um sistema ou para que um administrador conheça e possa analisar o comportamento histórico da aplicação. Um arquivo de log também pode ser utilizado para auditoria e diagnóstico de problemas em sistemas computacionais. Ao organizar os dados de um log precisamos observar as seguintes características:

- A informação é suficiente?
- A informação está suficientemente detalhada?
- Histórico de eventos;
- Níveis de log.

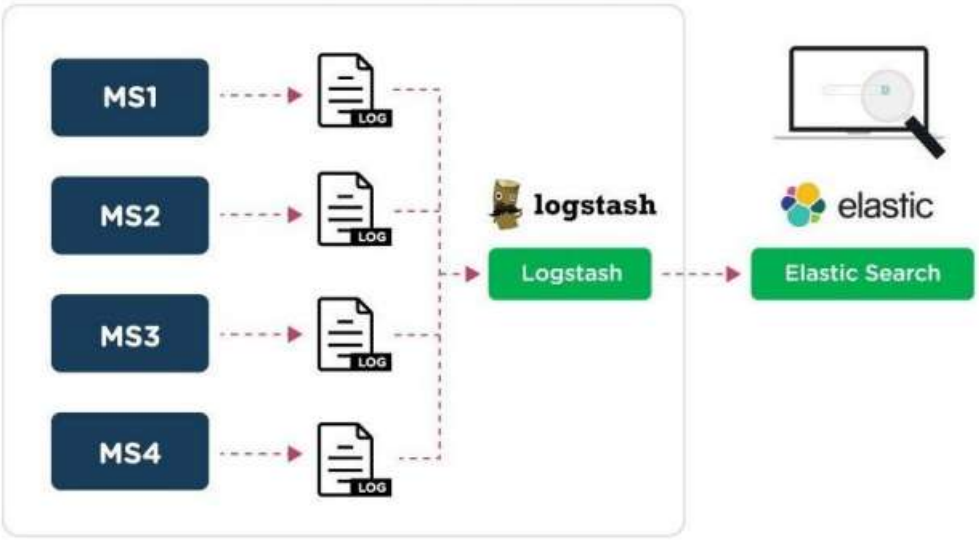
Log é um **registro** idempotente de um **evento** discreto ocorrido em um determinado **instante no tempo**.

Embora você obtenha logs gerais de suas plataformas, sistemas operacionais e servidores de aplicação, não obterá logs da sua aplicação, a menos que os adicione explicitamente em seu código fonte, e nao obterá logs para necessidades adicionais, como auditoria, sem adicioná-los explicitamente. **Você precisa gerar logs para todas as suas diferentes necessidades em toda a profundidade de sua arquitetura**. Existem muitos tipos de log: logs de sistema, logs de aplicativo, logs de auditoria, etc.

É importante ressaltar que para o monitoramento das aplicações que implementam os processos de negócio da organização, faz-se necessário que essas aplicações incluam em seu projeto os aspectos de log e que estes estejam implementados.

No projeto de microsserviços foi apresentada uma proposta arquitetural para o projeto e implementação de log a nível de aplicação para possibilitar o monitoramento e o rastreio e identificação de problemas. Nessa arquitetura ficou definido o uso do Elasticsearch e Logstash.

Figura 19 – Logstash e Elastic Search para centralização e indexação do log de aplicações.



O Logstash é um dos principais produtos do Elastic Stack, é usado para agregar e processar dados e enviá-los ao Elasticsearch. O Logstash é um pipeline de processamento de dados do lado do servidor que permite fazer a ingestão de dados de várias fontes simultaneamente e enriquecer e transformar esses dados antes de serem enviados ao Elastic Search. Já o Elastic Search é responsável pela indexação de dados. No Elasticsearch os usuários podem executar consultas complexas com base em seus dados e usar agregações para recuperar resumos complexos dos dados.

9.5 Ferramentas de Monitoramento

Para o monitoramento da infraestrutura que foi provisionada de forma a acompanhar a estabilidade dos recursos/serviços, para as métricas apresentadas na seção 9.2 recomendamos uma estratégia e um ferramental diferenciado.

9.5.1 Elastic Beats

Os Beats são agentes que residem nos servidores, containers e podem até ser implantados como funções. Estes agentes centralizam os dados no Elasticsearch.

Comparado ao Logstash, Beats são mais leves (consomem poucos recursos de sistema) e concebidos para enviar tipos específicos de dados de operação:

- **Filebeat**: arquivos de log;
- **Auditbeat**: dados de auditoria;
- **Metricbeat**: métricas;
- **Heartbeat**: monitoramento de disponibilidade;
- **Packetbeat**: dados de rede;
- **Functionbeat**: implantado como função (FaaS) em um cloud provider;
- **Winlogbeat**: logs de evento do windows.

Para o protótipo funcional, com vistas a monitorar o que foi provisionado da infraestrutura, selecionamos o **Metricbeat** e o **Heartbeat** para demonstração do monitoramento.

9.5.1.1 Metricbeat

Implantável em hosts Linux, Windows, Mac e containers para obter métricas relacionadas a CPU, memória, sistema de arquivos, E/S de disco, E/S da rede, bem como é possível obter dados sobre os processos em execução.

9.5.1.2 Heartbeat

Heartbeat pings via ICMP, TCP e HTTP, possui suporte a TLS, autenticação e proxies. É possível monitorar hosts que estejam por trás de um balanceador de carga.

9.6 Configurações

Como estamos seguindo os princípios de IaC, devemos provisionar e configurar também todos os elementos que permitam o monitoramento da infraestrutura e seus serviços, pontanto, temos uma nova role do ansible **configurar_observability** com as tasks abaixo (**main.yml**):

```
1  - name: copiando docker-compose
2    copy:
3      src: docker-compose.yml
4      dest: /tmp
5
6  - name: copiando metricbeat.yml
7    copy:
8      src: metricbeat.yml
9      dest: /tmp
10
```

```
- name: registrando docker network
  shell: |
    docker network list
  become: yes
  register: docker_network_output
  changed_when: docker_network_output.rc != 0

- name: criando docker network
  shell: |
    docker network create observability
  when: "'observability' in docker_network_output.stdout"

- name: iniciando servicos
  shell: |
    docker-compose up -d -f /tmp/docker-compose
```

Podemos observar que 2 arquivos são copiados: **docker-compose.yaml** e **metricbeat.yml**.

O arquivo de configuração docker tem por objetivo levantar e configurar os containers relacionados ao ao Metricbeat para coleta das métricas, ao Elasticsearch para indexação dessas métricas e ao kibana para visualização.

Já o arquivo de configuração do Metricbeat começa por ativar e configurar os módulos desejados, estes representam o que será monitorado e quais dados devem ser coletados:

```
metricbeat.modules:
- module: docker
  metricsets: ["container", "cpu", "diskio", "event", "healthcheck", "image", "info",
      "memory", "network"]
  hosts: ["unix:///var/run/docker.sock"]
  period: 10s

- module: elasticsearch
  metricsets: ["node", "node_stats", "cluster_stats", "index"]
  period: 10s
  hosts: ["https://elasticsearch:9200"]
  username: "elastic"
  password: "fabrica2022"
  ssl.verification_mode: none

- module: linux
  period: 10s
  metricsets:
    - "pageinfo"
    - "memory"
    - ksm
    - conntrack
    - "iostat"
      #- "pressure"
      #- rapl
  enabled: true
  #hostfs: /hostfs
  #rapl.use_msr_safe: false
  ...
```

Como podemos verificar temos 3 módulos:

- **docker**
  - Para coleta de dados relativos a contêineres a cada 10 segundos, sendo estes:

    * container: código do container (id), nome do mesmo, nome da imagem do contêiner, quando foi criado, ip, status (i.e. tempo de atividade), comando que foi executado no contêiner;
    * cpu: percentual de tempo de uso da CPU de forma total e por núcleo e ticks total e por núcleo;
    * diskio: quantidade de bytes lidos e escritos (separados ou acumulados), tempo de operação, tempo de espera e quantidade de operações em fila;
    * event: eventos disparados pelo contêiner, como um pull image por exemplo;
    * event: eventos disparados pelo contêiner, como um pull image por exemplo;
    * healthcheck: fornece status de saúde do contêiner (como status e número consecutivo de falhas), o status do contêiner pode ser *running* mas o estado de saúde ser *unhealthy*, por exemplo, já que o gerenciador de contêiner pode não interromper contêineres *unhealthy*. Estes dados estarão apenas disponíveis para os contêineres em que a instrução HEALTHCHECK foi utilizada na construção da imagem docker. Para entender melhor o conceito de healthcheck com docker recomendamos liveBook · Manning.
    * image: identificação da imagem, id da imagem de seu ascendente, quando essa imagem foi criada e tamanho da imagem.
    * info: dados gerais dos contêineres como: número total de contêineres, total de contêineres em pausa, execução e parados e número total de imagens.
    * memory: métricas relacionadas a memória como: total de memória disponível, memória em uso, pico de uso, número de falhas ocorridas por precisar ultrapassar o limite de memória disponível para o contêiner. Recomendamos a leitura de Understand Key Docker Metrics: Monitoring and Logging for Docker Datacenter - Sematext para compreender algumas métricas importantes.
    * network: métricas da rede como: quantidade de entrada e saída de dados, quantidade total de erros de pacotes de entrada e saída e resumos relacionados aos totais de IP, TCP, UDP, ICMP.
    * Para maiores informações acerca do módulo docker recomendamos Docker module — Metricbeat Reference [8.5] — Elastic.

- **elasticsearch**

  – Para coleta dados relativos ao Elasticsearch a cada 10 segundos, sendo estes:

    * node: dados relacionados ao nó do cluster como: bytes totais, bytes totais disponíveis, heap da JVM (máximo e disponível);
    * node_stats: estatísticas para cada node do cluster elasticsearch;
    * cluster_stats: versão do cluster, status;
    * index:
    * Para maiores informações acerca do módulo elasticsearch recomendamos Elasticsearch module — Metricbeat Reference [8.5] — Elastic.

- **linux**
  - Para coleta dados relativos ao sistema operacional a cada 10 segundos, sendo estes dados relativos a paginação, memória, cpu, I/O.
    * Para maiores informações acerda do módulo linux recomendamos Linux module — Metricbeat Reference [8.5] — Elastic.

Para maiores detalhes acerca de cada métrica recomendamos a busca em Docker fields — Metricbeat Reference [8.5] — Elastic.

Continuando no arquivo de configuração do Metricbeat temos também a configuração da comunicação com o Elasticsearch para o envio das métricas coletadas:

```
...
output.elasticsearch:
  hosts: ["https://elasticsearch:9200"]
  protocol: "https"
  username: "elastic"
  password: "fabrica2022"
  ssl.certificate_authorities: ["/usr/share/meatricbeat/config/certs/ca/ca.crt"]
  ssl.certificate: "/usr/share/meatricbeat/config/certs/elasticsearch/elasticsearch.
      crt"
  ssl.key: "/usr/share/meatricbeat/config/certs/elasticsearch/elasticsearch.key"
  ssl.verification_mode: none
  ...
```

E por último nesse mesmo arquivo a configuração da ferramenta de visualização dessas métricas por meio de dashboards com Kibana:

```
setup.kibana:
  host: "http://kibana:5601"
  # protocol: "https"
  username: "elastic"
  password: "fabrica2022"
  # ssl.enabled: true
  # ssl.certificate_authorities: ["/usr/share/meatricbeat/config/certs/ca/ca.crt"]
  # ssl.certificate: "/usr/share/meatricbeat/config/certs/elasticsearch/elasticsearch.
      crt"
  # ssl.key: "/usr/share/meatricbeat/config/certs/elasticsearch/elasticsearch.key"

setup.dashboards.enabled: true
```

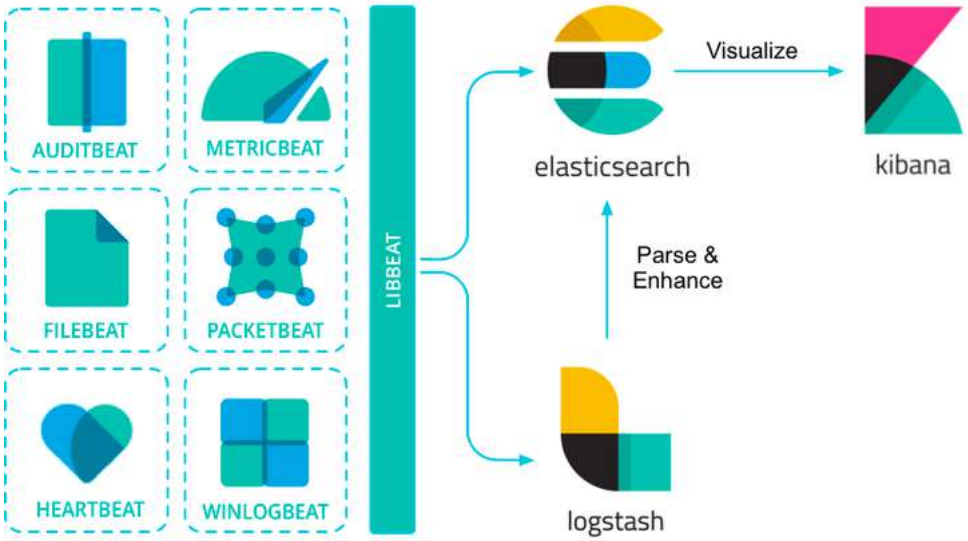
A configuração **setup.dashboards.enable: true** informa ao Kibana que os dashboards referente aos beats configurados devem ser importados e gerados automaticamente. Isso possibilita a geração automática dos dashboards a partir dos dados coletados e armazenados no Elasticsearch.

9.7 Monitoramento

9.7.1 Dashboards

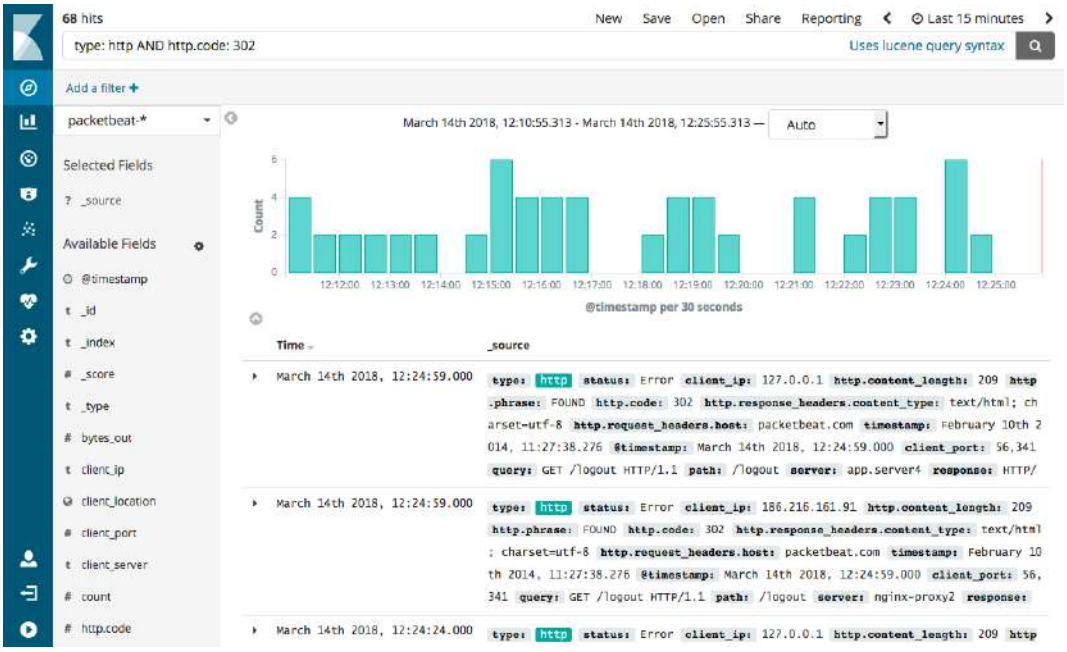
A criação dos dashboards no Kibana se dá pela integração com as informações armazenadas no Elasticsearch. As informações são provinientes dos Beats ou ainda de outras formas de captura e tratamento de dados, como é o caso do Logstash, ilustrado na Figura 20.

Figura 20 – Captura e tratamento dos dados para o Kibana



As informações tratadas e armazenadas ficam disponíveis no Elasticsearch conforme exemplo da Figura 21, podendo ser utilizadas para geração dos dashboards.

Figura 21 – Informações armazeadas no Elastcsearch



O Kibana disponibiliza uma série de dashboards para os mais diversos tipos de monitoramento, como é o caso do Linux, Kubernetes, Docker, NGINX, Postgres, etc.

Na Figura 22 é apresendo o dashboard de monitoramento do Linux a partir das configurações apresentadas no Capitítulo 5. Configurações. Esse dashboard foi automaticamente importado pelo Kibana e apresentad informações que estão sendo armazenadas no inventário de monitoramento do Kibana, que por sua vez utiliza as informações do Elasticsearch.

Figura 22 – Dashboard de monitoramento

Diversos dashboards prontos são disponibilizados pelo Kiabana. Esse dashboards podem ser customizados conforme necessidade. Sugermios a leitura de Dashboard do Kibana para mais informações sobre a configuração de dashboards no Kibana.

9.7.2 Hearbeat

O Kibana possui o dashboard de monitoramento da disponibilidade de serviços conforme citado no Capitulo 4. Esse bashboard utiliza as configurações que são aplicadas no serviço do Heartbeat e as informações encaminhadas ao Elasticsearch.

No código abaixo faz a configuração do Heartbeat para o monitoramento HTTP e também ICMP de alguns serviços:

```
heartbeat.monitors:
- type: http
  schedule: '@every 5s'
  ssl:
    enabled: true
    verification_mode: none
  urls:
    - https://10.24.3.200:9200
    - https://10.24.3.200:5601
    - https://172.16.0.169:8000
  check.response.status: [200, 302, 401, 404, 500]

- type: icmp
  schedule: '@every 5s'
  hosts:
    - 10.24.3.200
    - 172.16.0.169


- type: icmp
  schedule: '@every 5s'
  hosts:
    - 10.24.3.200
    - 172.16.0.169

output.elasticsearch:
  hosts: ["https://10.24.3.200:9200"]
```

```
  protocol: "https"
  username: "elastic"
  password: "fabrica2022"
  ssl.certificate_authorities: ["/usr/share/meatricbeat/config/certs/ca/ca.crt"]
  ssl.certificate: "/usr/share/meatricbeat/config/certs/elasticsearch/elasticsearch.
      crt"
  ssl.key: "/usr/share/meatricbeat/config/certs/elasticsearch/elasticsearch.key"


setup.kibana:
  host: "https://10.24.3.200:5601"
  protocol: "https"
  username: "elastic"
  password: "fabrica2022"
  ssl.enabled: true
  ssl.certificate_authorities: ["/usr/share/meatricbeat/config/certs/ca/ca.crt"]
  ssl.certificate: "/usr/share/meatricbeat/config/certs/elasticsearch/elasticsearch.
      crt"
  ssl.key: "/usr/share/meatricbeat/config/certs/elasticsearch/elasticsearch.key"
```

O Heartbeat possui diversas configurações de monitoramento de serviços utilizando diversos protocolos, como ICMP, HTTP e TCP. Sugerimos a leitura de Uptime Monitoring with Heartbeat and the Elastic Stack para maiores informações.

Basicamente esse trecho do código define as seguintes funcionalidades para cada bloco de código iniciado por:

1. Utiliza do protocolo http para checar páginas publicadas em portas específicas a cada 5 segundos. Para que o Heartbeat entenda que o serviço está disponível, os seguintes códigos http devem ser retornados [200, 302, 401, 404, 500].

2. Utiliza o protocolo ICMP para verificar a resposta do Internet Protocol (IP) de um determinado host ou equipamento.

3. Configuração de envio das informações para o Elasticsearch

4. Configuração do comunicação com o Kibana

9.7.3 Alertas

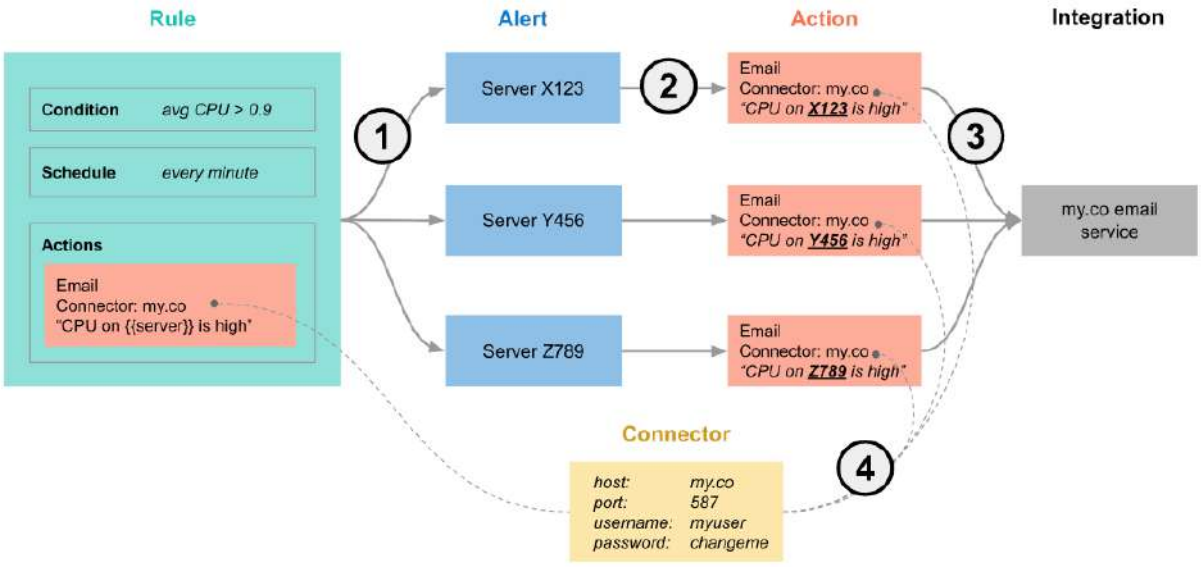
O Kibana disponibiliza a funcionalidade de Alertas com base em critérios pré-definidos, como uso de CPU, memória, trafego de rede, etc.

Os alertas são configurados a partir de regras (Rules) que possuem condições, agendamento de execução e ações a serem realizadas caso aconteçam. Um grande diferencial do Kibana é que as regras podem ser aplicadas de maneira que todo o inventário seja contemplado para uma única Regra, ou ainda para um único host, possibilitando assim que a partir de uma única regra todo o inventária seja monitorado. A Figura 23 apresentado o funcionamento de criação de regras.

Portanto, na Figura 23 é criado uma Regra que executa a cada 1 minuto (**agendamento**) que verifica se a média de uso da CPU está acima de 90% (**condição**). Caso positivo será emitido um alerta e encaminhado e-mail para o administrador (**Ação**).

Os detalhes do funcionamento dos Alertas está disponível em Alerting — Kibana Guide [8.5] — Elastic.

Figura 23 – Funcionamento dos alertas



Sugerimos a leitura de Alerta do Kibana: alertas e ações para dados do Elasticsearch para mais informações sobre a configuração de alertas.

Referências

Métricas para componentes do sistema Kubernetes
kvmtop
https://portal.influxdata.com/downloads/
https://www.ibm.com/docs/en/linuxonibm/pdf/Monitoring-with-RHEL-8-and-KVM.pdf
Install prometheus-libvirt-exporter on Linux — Snap Store

- Kibana queries and filters — Packetbeat Reference [8.5] — Elastic
- Observability (Re)defined - crowdstrike.com What is Observability? A Beginner's Guide — Splunk
- Elastic stack Presentation
- https://files.speakerdeck.com/presentations/cefd467462c6430db0f39c33f269fa2d/wednesday-andrew-kroh-monitoring-docker-with-metricbeat-spotlight-theater.pdf
- Virtualization: Definition, Key Metrics to Monitor - VirtualMetric - Infrastructure Monitoring Blog
- How To Gather Infrastructure Metrics with Metricbeat on CentOS 7 — DigitalOcean

Texto sobre a composição do livro...