INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia Eletrónica e Telecomunicações e Computadores

Pipeline CI/CD para Automação e Orquestração de Redes

JOÃO MIGUEL CARACÓIS BORGES

Licenciado em Engenharia Informática, Redes e Telecomunicações

Dissertação para obtenção do Grau de Mestre
Mestrado em Engenharia de Eletrónica e Telecomunicações

Orientador: Professor Doutor Nuno Miguel Machado Cruz

Júri:
Presidente: Professor Doutor Rui António Policarpo Duarte
Vogais: Professor Doutor Nuno Miguel Machado Cruz
Professor Especialista Pedro António Marques Ribeiro

Novembro, 2023

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia Eletrónica e Telecomunicações e Computadores

Pipeline CI/CD para Automação e Orquestração de Redes

JOÃO MIGUEL CARACÓIS BORGES

Licenciado em Engenharia Informática, Redes e Telecomunicações

Dissertação para obtenção do Grau de Mestre
Mestrado em Engenharia de Eletrónica e Telecomunicações

Orientador: Professor Doutor Nuno Miguel Machado Cruz

Júri:
Presidente: Professor Doutor Rui António Policarpo Duarte
Vogais: Professor Doutor Nuno Miguel Machado Cruz
Professor Especialista Pedro António Marques Ribeiro

Novembro, 2023

Agradecimentos

No decorrer desta dissertação tive o apoio de diversas pessoas e instituições, pelo que me encontro grato.

Agradeço ao meu Orientador, Doutor Nuno Miguel Machado Cruz pelo apoio no desenvolvimento deste projeto, bem como toda a disponibilidade e os comentários construtivos ao longo da dissertação.

Agradeço à FCCN pelo apoio e flexibilidade ao longo do desenvolvimento da dissertação, bem como a oportunidade de trabalhar na área permitindo-me seguir os meus estudos e chegar a este ponto.

Agradeço à minha família e amigos por me apoiarem de forma inabalável em todo o meu percurso académico e a todo o incentivo, paciência e ajuda constante.

RESUMO

Novas formas de desenvolvimento de *software*, ambicionam introduzir novas funcionalidades e/ou correções rapidamente em produtos finais, apoiando-se em ferramentas que permitem a construção de *pipeline*s de desenvolvimento incremental (CI/CD). Para tal acontecer, é necessário a construção de formas de validação e teste do código desenvolvido para assim gerir o ciclo de vida destas.

Nas redes de computadores existe a ambição de tratar de toda a infraestrutura de rede como se código fosse (*Network as Code*), usando como base a configuração dos diferentes equipamentos. Esta forma é em tudo semelhante à abordagem usada para desenvolvimento de *software* e aplicar as mesmas estratégias, permite alavancar a introdução de novas funcionalidades ou correções à infraestrutura de rede.

Este trabalho de projeto pretende explorar as abordagens de *Network as Code* em que existe um repositório com a configuração de toda a rede e em que são introduzidos mecanismos para construir testes e simular a rede para execução destes. Após a validação de novas alterações à configuração, o *pipeline* CI/CD deverá então aplicar a configuração aos equipamentos em produção.

Palavras-chave: *Pipeline* CI/CD, Automação de Redes, Virtualização, DevOps, Automação de Testes de Redes

Abstract

New software development approaches aim to swiftly introduce new features and/or corrections into final products, relying on tools that enable the construction of incremental development pipelines (CI/CD). For this to happen, it is necessary to build validation and testing methods for the developed code to manage the lifecycle of these products.

In computer networks, there is an ambition to treat the entire network infrastructure as if it were code (Network as Code), based on the configuration of different devices. This approach closely resembles the one used in software development, and applying the same strategies allows for the introduction of new features or corrections to the network infrastructure.

This project work intends to explore Network as Code approaches where there is a repository with the configuration of the entire network, and mechanisms are introduced to build tests and simulate the network for their execution. After validating new configuration changes, the CI/CD pipeline should then apply the configuration to the production equipment.

Keywords: Pipeline CI/CD, Network Automation, Virtualization, DevOps, Network Testing Automation

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE LISTAGENS

LISTA DE ABREVIATURAS

AOT Ahead-of-Time compilation. 17
API Application Programming Interface. 1, 9, 16, 19, 21, 23
AWS Amazon Web Services. 19

BGP Border Gateway Protocol. 22, 31, 37, 39, 40, 41, 42, 46

CGroup Control Groups. 7
CI/CD Integração Contínua e Entrega Contínua. 1, 2, 3, 9, 18, 19, 20, 21, 25, 33
CLI Command Line Interface. 13, 16, 22

DAL Device Abstraction Layer. 22
DHCP Dynamic Host Configuration Protocol. 31, 37, 41, 42, 47, 50
DNS Domain Name System. 11, 12, 31, 33, 37, 40, 43, 47
DSL Domain-Specific Language. 15

EVE-NG Emulated Virtual Environment – Next Generation. 13, 14, 60

GCP Google Cloud Platform. 19
GNS3 Graphical Network Simulator-3. 13

IP Internet Protocol. 10, 12, 30, 31, 33, 38, 40, 42, 43, 44, 47
IPv4 Internet Protocol version 4. 11, 28, 32, 34, 35, 45
IPv6 Internet Protocol version 6. 11

JSON Javascript Object Notation. 22

NETCONF Network Configuration Protocol. 22
NFV Network Functions Virtualization. 7
NTP Network Time Protocol. 31, 42, 43, 47

OSPF Open Shortest Path First. 31, 32, 36, 37, 38, 39, 40, 41, 43, 45, 46, 55, 57

PID Process ID. 7

PYATS Python Automation Framework for Testing and Software Development. 21, 22, 23

SDN Sofware Defined Network. 10

SNMP Simple Network Management Protocol. 13, 16

SO Sistema Operativo. 6, 7, 9, 10, 18, 21

SSH Secure Shell. 12, 13, 15, 16, 18, 30, 33, 40

TCP Transmission Control Protocol. 10

VM Virtual Machine. 5, 6, 8, 9, 12, 13, 14, 25, 28, 29, 33, 34, 46, 59, 60

1

INTRODUÇÃO

Atualmente, as empresas confrontam-se com o imperativo de manterem uma infraestrutura de rede robusta e ágil, essencial para sustentar as operações, o serviço aos clientes e os processos de inovação. Todavia, observa-se que a gestão convencional de redes é frequentemente lenta, ineficiente e suscetível a erros humanos.

Numa tentativa de resolver estes desafios, fala-se cada vez mais do conceito *Network as Code*, sendo esta uma abordagem que visa guardar configurações de equipamentos num mecanismo de controlo de versões, ter este código como fonte única de verdade e entregar configurações recorrendo a *Application Programming Interface* (API). Ao utilizar esta abordagem seria possível melhorar a automação e consistência na configuração de equipamentos de rede, agilidade no processo de desenvolvimento, em questões de segurança e ao escalar a infraestrutura de rede consoante a necessidade.

Para realmente se conseguir os objetivos acima descritos, é necessário desenvolver a vertente de automação de redes. Este é um conceito que pretende automatizar processos relacionados com a configuração, gestão e entrega de equipamentos físicos e virtuais num contexto de testes ou produção, deixando de existir a necessidade de realizar tarefas repetitivas e vulneráveis ao erro humano. Dado o ritmo elevado que se altera a infraestrutura de rede atualmente, é importante criar um processo que seja o mais eficiente e seguro possível.

Assim, introduz-se o conceito de Integração Contínua e Entrega Contínua (CI/CD), uma prática de desenvolvimento de *software* que visa agilizar a integração de código recorrendo a testes automatizados. Integração contínua refere-se ao constante desenvolvimento de código e à sua testagem, tendo como base um repositório onde se encontra todas as configurações, *templates* e testes. Entrega contínua refere-se ao lançamento automático destas alterações para um ambiente de produção a partir do repositório caso todos os testes tenham sido bem-sucedidos.

1.1. Objetivos

Esta dissertação foca-se no desenvolvimento de uma *pipeline* CI/CD que integra um repositório com ficheiros que descrevam a topologia de rede, as configurações dos equipamentos de rede e os testes aplicáveis à topologia definida. Com todos estes ficheiros e tendo em conta o conceito de CI/CD acima descrito, a topologia definida é simulada através de *software* para que possam ser efetuados os testes definidos para garantir que a rede se encontra funcional e cumpre os requisitos necessários, tendo em conta as configurações presentes. Caso esses requisitos sejam cumpridos, as alterações efetuadas aos ficheiros de configuração são aplicadas nos equipamentos em produção.

O principal objetivo é validar se esta é uma abordagem viável aos problemas acima descritos e desta forma tornar cada vez mais eficiente o desenvolvimento de novas funcionalidades e correção de problemas em infraestruturas de rede.

1.2. Requisitos da Solução

Atendendo ao contexto do problema em análise e aos objetivos propostos, consideram-se como requisitos da solução proposta:

- Repositório de código para armazenar todos os ficheiros necessários para a simulação, configuração e testagem dos equipamentos;
- Uma ferramenta CI/CD para gerir a *pipeline* definida sempre que haja uma alteração no repositório;
- Uma ferramenta para simular uma topologia de rede com diferentes vendedores;
- Uma ferramenta para configurar os equipamentos simulados;
- Uma ferramenta para testar toda a topologia com os requisitos pretendidos e devolver essa informação ao utilizador;
- A arquitetura deve privilegiar a utilização de ferramentas de código aberto;
- Garantia de Fiabilidade, Escalabilidade e Eficiência da arquitetura.

1.3. Organização do Documento

De modo a demonstrar todo o caminho desde a pesquisa até implementação da arquitetura a ser proposta, decidiu-se que a organização do relatório deveria ser feita em 4 capítulos.

No Capítulo 2 é realizado um estudo acerca dos vários componentes a serem utilizados e de possíveis alternativas, começando pelo ambiente de desenvolvimento, seguido de ferramentas de automação e de entrega de topologia de redes, a componente CI/CD e a ferramenta dedicada à realização de testes de configuração à rede.

No que toca ao Capítulo 3, é realizada uma proposta de metodologia e arquitetura, sendo identificados os componentes que compõem a arquitetura, o papel de cada um e como interagem.

No Capítulo 4 é feita a definição dos testes desenvolvidos, tendo em conta o que se pretende avaliar, a metodologia utilizada, alguns desafios encontrados e decisões tomadas. São também detalhados alguns detalhes relativos à implementação da arquitetura e demonstrados alguns exemplos do seu funcionamento.

Por fim, o Capítulo 5 é efetuada a reflexão sobre o trabalho desenvolvido e sobre o trabalho futuro por desenvolver, incluindo os desafios associados ao desenvolvimento da dissertação.

2

ESTADO DE ARTE

Neste capítulo pretende-se fornecer uma visão geral acerca do estado em que se encontra o desenvolvimento relacionado com automação de redes e a sua consequente testagem e análise. A pesquisa efetuada não se limitou a referir apenas uma solução para cada componente do trabalho, pelo que se apresentou a arquitetura das ferramentas, o seu modo de funcionamento, as suas vantagens e desvantagens.

Este capítulo tem como objetivo estabelecer as bases para melhor entender a abordagem utilizada e as suas alternativas, desde o ambiente onde se vai executar o trabalho até às plataformas de automatização e testagem de topologias de rede.

2.1. VIRTUALIZAÇÃO

Virtualização é uma técnica baseada em *software* que permite particionar um sistema computacional em diferentes ambientes isolados semelhantes a um sistema físico, separando aplicações e o sistema operativo dos componentes físicos, conseguindo-se utilizar os recursos do mesmo de uma forma mais eficiente.

Tendo em conta este conceito, começou-se por ter em conta o tipo de ambiente a ser utilizado para o desenvolvimento desta dissertação, pelo que se considerou usar máquinas virtuais. Este é um termo que foi descrito na década de 1960, mas somente na década de 1970 é que começou a ser oferecido como produto pela IBM. Em 1974 foi publicado um dos primeiros artigos relativamente a este tema onde se descrevia a implementação de uma máquina virtual para a arquitetura IBM System/370, que tinha como propósito a partilha de um único computador por vários utilizadores [1].

2.1.1. MÁQUINAS VIRTUAIS

Uma Máquina Virtual (VM) é um ambiente de computação isolado que simula uma máquina física, ou seja, simula componentes de *hardware* como processador, memória, armazenamento e interfaces de rede. Esta é criada a partir de uma camada de abstração em relação ao *hardware* o que permite que um único computador físico execute várias VMs. Cada VM corre o seu Sistema Operativo (SO) e funciona de forma independente de outras VMs mesmo que estejamno mesmo *host*.

De modo a gerir o *hardware* e separar os recursos físicos utilizados pela VM, recorre-se a um hipervisor para reservar os mesmos temporariamente, sendo os recursos particionados do sistema físico para a VM conforme for necessário [2]. Quando a VM se encontra a ser executada e um utilizador/programa realiza uma ação que necessita de recursos adicionais, o hipervisor agenda um pedido ao sistema físico de modo a ser possível aceder à *pool* partilhada de recursos [3].

Em termos de nomenclatura, diz-se:

- *Host* quando se refere ao sistema onde o hipervisor está instalado;
- *Guest* quando se refere às VMs que utilizam recursos do *host*.

Tal como foi referido anteriormente, VMs permitem que diferentes SO corram no mesmo *Host*, sendo que cada um é executado como se estivesse uma máquina física normal, estando limitada pelos recursos atribuídos.

Assim, pode-se dizer que ao utilizar virtualização e por consequência VMs se consegue utilizaros recursos de um sistema de uma forma eficiente.

Pode-se dividir os hipervisores em dois tipos:

- Tipo 1 – também referido como *bare metal*, este encontra-se em contacto direto com o *hardware* do *host*, permitindo que os recursos da VM sejam programados diretamente no *hardware* pelo hipervisor;
- Tipo 2 – também referido como *hosted*, este é um *software* instalado sobre o SO que permite a virtualização. Neste caso os recursos da VM não são programados diretamente no *hardware*, mas sim em cima de um SO do *host* que por sua vez comunica com o *hardware*.

Dentro da virtualização, tem-se ainda diferentes tipos de virtualização como a de servidores, SO e redes, onde cada um tem os seus cenários de utilização [4]:

- Servidores – Permite utilizar ao máximo as capacidades de processamento do *hardware* e maximizar o espaço nos *datacenters*;
- SO – Permite que existam vários sistemas isolados, tendo em conta que esta situação acontece ao nível do *kernel*. Este tipo de virtualização encontra-se associada a contentores (mais detalhes no Capítulo 2.1.2);

- Redes – Também conhecido por *Network Functions Virtualization* (NFV), este é um método de virtualizar serviços de rede, como *routers*, *switches*, balanceadores de carga, *firewall*, entre outros. Ao utilizar este tipo de virtualização, consegue-se executar uma rede num servidor comum, tendo como vantagem a sua escalabilidade e agilidade.

2.1.2. CONTENTORES

Um contentor ou *container*, permite criar um ambiente isolado e que não consome tantos recursos, capaz de executar aplicações e qualquer outro tipo de serviços [2]. Estes tiram partido do mesmo SO do *host*, tal como referido no Capítulo 2.1.1, sendo apenas um processo em execução num *kernel* partilhado por diversos contentores.

Um contentor é caracterizado pela sua imagem, ou seja, todas as aplicações e dependências necessárias para o fim que se pretende dar ao mesmo, desde o seu SO às suas bibliotecas. Um dos principais requisitos para a utilização desta tecnologia é o *host* suportar o *runtime* do contentor, não sendo preciso qualquer modificação ou configuração deste.

Esta tecnologia tira partido de funcionalidades dos SO como *namespaces* e *control groups* (*cgroups*) para conseguir executar dois ou mais contentores na mesma máquina física. *Namespaces* é uma funcionalidade do *kernel* Linux que particiona os seus recursos de modo que um conjunto de processos tenha acesso a um conjunto de recursos, enquanto outro conjunto de processos tem acesso a outro conjunto de recursos, isolando os sistemas. Esta funcionalidade pode ter diferentes tipos como [2]:

- *Process* ID (PID) namespace – designa um conjunto de PIDs para processos que são independentes de outros *namespaces*;
- *User namespace* – tem o seu próprio conjunto de *user* ID e *group* ID, permitindo um processo ter privilégios de *root* dentro de um *namespace*;
- *Network namespace* – tem recursos de rede independentes como tabela de roteamento privada, conjunto de endereços IP, *firewall*, entre outros;
- *Mount namespace* – permite fazer *mount* e *unmount* de *filesystems* sem afetar o *host*.

O *cgroup* também é uma funcionalidade do *kernel* Linux que permite limitar, controlar e priorizar a utilização de recursos de um processo ou conjunto de processos, dado que

por norma existem vários processos a serem executados num contentor.

Assim, quando um contentor é criado, também vai ser criado um *namespace* para este o isolar do *host* e de outros contentores. Com isto, recorre-se à imagem do contentor e é feito o *mount* do seu *filesystem* como *read-only* dentro do *namespace*. Quando o contentor é inicializado, é criado um processo dentro do *namespace* do contentor e a aplicação é executada nesse mesmo processo, sendo que os processos do contentor se encontram isolados dos do *host* e a aplicação pode somente aceder a recursos atribuídos ao mesmo.

2.1.3. Comparação entre VMs e Contentores

Após ter sido efetuada a pesquisa em relação a VMs e contentores, vai-se realizar a comparação entre estas tecnologias. Ambas têm como objetivo executar aplicações e serviços em ambientes virtualizados, no entanto, cada uma tem as suas vantagens e desvantagens, bem como situações a serem utilizadas.

Algumas vantagens ao utilizar VMs:

- Isolamento entre instâncias, visto que cada VM se encontra no seu *hardware* virtualizado e SO;
- Como é realizada virtualização do *hardware* consegue-se ter um bom desempenho;
- Melhor segurança em relação a contentores, devido ao seu isolamento;
- Suportam aplicações *legacy*.

No que toca a desvantagens ao utilizar VMs:

- Consome mais recursos do que contentores, uma vez que é necessário virtualizar *hardware* e SO;
- O período de inicialização é superior em relação a contentores;
- Têm escalabilidade mais limitada em comparação a contentores.

Em relação a contentores tem-se como vantagem:

- Eficiência na utilização de recursos, dado que partilham o *kernel* do *host*;
- Maior rapidez na sua inizialização, visto que apenas necessitam de ter em conta

as aplicações necessárias e as suas dependências;

- Escalabilidade elevada devido a consumirem poucos recursos e serem inicializadas rapidamente;
- Facilidade na sua manutenção, uma vez que é comum utilizar uma arquitetura baseada em micro serviços, o que torna lidar com erros uma tarefa mais simples.

No entanto, contentores também apresentam desvantagens como:

- Fraco isolamento entre diferentes contentores e o próprio SO;
- Pior segurança em relação a VMs, também devido ao fraco isolamento;
- Podem existir problemas de compatibilidade entre SO e aplicações;
- Existência de complexidade ao orquestrar e gerir vários contentores, sendo necessário recorrer a ferramentas especializadas.

Depois da descrição das várias vantagens e desvantagens em relação a VMs e contentores, consegue-se escolher de forma mais consciente qual a melhor opção para atingir o fim pretendido.

Tal como foi referido anteriormente, ambas as tecnologias são bastantes versáteis, pelo que se pretende utilizar as duas neste projeto, isto é, ter-se uma VM para se conseguir um ambiente estável e utilizar contentores para serviços mais pequenos e temporários. Esta é uma metodologia utilizada por empresas, visto que permite com que os desenvolvedores utilizem ambientes semelhantes quando constroem aplicações, sendo também utilizada para facilitar automatização de *pipelines* CI/CD [2][3][5].

2.2. DOCKER

Docker é uma ferramenta que simplifica o processo de criar, entregar e executar aplicações num ambiente utilizando contentores, por isto a utilização desta ferramenta oferece as mesmas vantagens descritas no Capítulo 2.1.3. Este utiliza uma arquitetura cliente-servidor onde o cliente comunica com o Docker *daemon*, visto que realiza tarefas como a construção e gestão de contentores estando à escuta de pedidos à Docker API. O *daemon* e o cliente Docker podem ser executados no mesmo *host* ou como alternativa pode-se conectar o cliente a um *daemon* remoto. Estes comunicam através de uma REST API utilizando UNIX *sockets* ou uma interface de rede. [6]

Dado que Docker utiliza contentores, tem como base imagens Docker que são caracterizadas como sendo independentes, leves e contêm todas as dependências necessárias para uma aplicação ser executada, quer seja código fonte, *runtime*, variáveis

de ambiente, ficheiros de configuração e bibliotecas. Assim, contentores Docker são instâncias de imagens que após serem inicializados podem ser geridos através de diversos comandos.

Para construir imagens Docker, recorre-se ao *Dockerfile*, sendo este um ficheiro que identifica a imagem que se pretende ter num contentor como o seu SO e os comandos que se pretende utilizar ao construir a imagem. As imagens podem ser criadas manualmente pelo utilizador ou, dado que existe uma grande comunidade e pesquisa em torno desta ferramenta, pode-se fazer *pull* de imagens de repositórios como Docker Hub. Por predefinição a opção de procurar imagens neste repositório está ativada, no entanto, é possível ter-se um repositório local de imagens utilizando o Docker Registries que possibilita armazenar imagens Docker. [7]

Uma vertente importante de contentores Docker é a sua capacidade de se conectarem entre si ou com outros sistemas que não sejam Docker, tendo uma componente agnóstica poderosa. Docker utiliza um contentor *network model*, ou seja, uma *Sofware Defined Network* (SDN) que conecta todos os contentores num único *host* podendo criar várias redes. Para isso, recorre a *Docker Network Drivers* para lidar com a comunicação entre contentores, sendo que por predefinição é criada uma rede em modo *bridge*. Na maior parte dos casos este tipo de rede funciona, mas existem outras possibilidades [7]:

- *Bridge* – Utilizado por predefinição, sendo muito comum em contentores independentes que correm aplicações. A desvantagem deste modo é que apesar de funcionar bem para testes locais, não é a melhor opção para ambientes de produção, uma vez que cada vez que um contentor é inicializado é-lhe atribuído um endereço IP diferente. Também permite que contentores que não têm qualquer tipo de relação comuniquem entre si, podendo ser um risco de segurança;

- *Host* – Utiliza endereço IP e porto *Transmission Control Protocol* (TCP) para expor os serviços que se encontram a ser executados dentro do contentor, deixando de existir isolamento entre contentores e *host*. A principal desvantagem é o facto de não ser possível existirem vários contentores no mesmo *host* utilizando o mesmo porto;

- *Overlay* – Utilizado para conectar múltiplos Docker daemon e permitir a utilização de serviços de Docker Swarm ou Kubernetes, ajudando nas interações

entre contentores independentes e os serviços acima descritos;

- *Ipvlan* – Permite que os utilizadores tenham controlo total sobre o endereçamento IPv4 e IPv6;

- *Macvlan* – Utilizado principalmente para aplicações *legacy* ou aplicações que monitorizam tráfego de rede, uma vez que normalmente é esperada uma ligação física à rede, tendo-se de atribuir um endereço MAC à interface de rede de cada contentor Docker que por sua vez o *daemon* utiliza para mapear o tráfego;

- *None* – Quando não se pretende que o contentor tenha acesso a redes externas ou que consiga comunicar com outros contentores;

- *Plugins* externos – também suporta *plugins* externos quando se pretende integrar Docker com *software* especializado, como OpenvSwitch e Cisco ACI.

Docker apresenta ainda uma funcionalidade conveniente em relação aos DNS *nameservers*, que permite com que haja resolução de nomes sem existir qualquer menção no comando Docker. Quando se executa um novo contentor no Docker *host*, o ficheiro "/etc/resolv.conf" é automaticamente copiado para o contentor, estando estes sempre sincronizados. No entanto, as alterações realizadas neste ficheiro são apenas efetuadas quando o contentor não se encontra em execução.

2.3. CONTAINERLAB

Com o aumento que tem existido na adoção de tecnologias de virtualização em arquiteturas de rede, faz sentido existir uma ferramenta que consiga fazer a entrega de topologias de rede usando esta tecnologia. Posto isto, o que começou como uma ferramenta desenvolvida pela Nokia para uso interno, tornou-se numa ferramenta de código aberto que continua a crescer com o apoio da comunidade. [8]

Containerlab é uma ferramenta criada com o propósito de simplificar a criação e gestão de um laboratório de rede composto por vários contentores. Permite aos utilizadores criarem e operarem uma rede virtual complexa, podendo assim testar e desenvolver redes de uma forma mais eficiente.

Como base, o Containerlab utiliza Docker para fazer a entrega e gerir equipamentos de rede. Os utilizadores podem criar topologias de rede ao definirem um ficheiro YAML

que especifica o tipo de equipamento, configuração e conexões com outros equipamentos. A definição da topologia encontra-se dividida entre *nodes* e *links*, onde em cada um se especifica o seu tipo(*kind*) e imagem e quais as ligações que se pretende ter a outros *nodes*, respetivamente. Um dos componentes mais importantes são as imagens, uma vez que são a base de toda a topologia, sendo que algumas estão disponíveis publicamente (Nokia SR Linux), outras necessitam de serem transferidas por parte do utilizador e adicionadas manualmente às imagens Docker disponíveis e outras necessitam de serem compradas.

Ao realizar a entrega de uma topologia, um detalhe importante a ter em conta para realizar automatizações, o nome dos *nodes* seguem a estrutura "clab-nome_da_topologia- nome_do_*node*". Para interagir com os contentores, uma vez que se utiliza Docker, pode-se utilizar o comando "exec" ou utilizar SSH conseguindo através da especificação do IP do contentor ou o nome que se encontra no DNS.

Tal como foi referido em capítulos anteriores, contentores são uma tecnologia que não consome muitos recursos, no entanto algumas imagens de equipamentos não são otimizadas para contentores e atrasam a entrega da topologia, como no caso das imagens referentes a alguns equipamentos Cisco que podem demorar cerca de 25 minutos a serem totalmente inicializadas, dependendo dos recursos da máquina. Com isto é importante refletir a abordagem que se pretende ter ao escolher a topologia e como utilizar esta ferramenta. [9]

Nem todos os fabricantes de equipamentos de rede têm imagens de equipamentos que permitam ser utilizadas em contentores, pelo que as mais relevantes que se encontram disponíveis são:

- Arista cEOS;
- Cisco XRd;
- Juniper cRPD;
- Nokia SR Linux.

No entanto, também são suportadas algumas imagens criadas para serem utilizadas em VMs como:

- Arista vEOS;
- Cisco IOS XRv9k/Nexus 9000v/CSR 1000V;

- Dell FTOS10v;
- Nokia virtual SR OS.

2.4. Emulated Virtual Environment – Next Generation

Emulated Virtual Environment – Next Generation (EVE-NG) também conhecido por EVE-NG é emulador de redes de código aberto que permite criar e testar topologias de rede. As topologias de rede podem ser compostas por uma mistura entre equipamentos físicos e virtuais, oferecendo uma grande integração quando se pensa no contexto de uma empresa.

Este emulador precisa de um hipervisor para lhe fornecer uma infraestrutura que lhe permita executar VMs que vão posteriormente emular equipamentos. Este emulador funciona à base de imagens quer sejam preexistentes ou customizadas, sendo que existe uma grande variedade de imagens de diferentes fabricantes disponíveis, como Cisco, Juniper, Huawei, Dell, Arista, Mikrotik, entre outras.

Para criar uma topologia de rede é utilizada normalmente uma interface *web* com menus interativos e simples, no entanto, a configuração da mesma é feita em YAML podendo ser automatizada.

EVE-NG permite que sejam feitos *snapshots*, clonagem e *templates* de equipamentos, facilitando a reutilização de topologias e configurações. Para aceder aos equipamentos, pode-se recorrer a uma *Graphical User Interface* (GUI) ou pela *Command Line Interface* (CLI) (SSH, SNMP, Telnet), sendo que a CLI permite a realização de *scripts* e automatizações. Também é compatível com ferramentas externas como Ansible, Terraform e *Graphical Network Simulator-3* (GNS3) para criar e gerir as topologias de rede.

Dada a arquitetura deste emulador, consegue-se simular topologias de rede muito complexas e escalar facilmente. Por contrapartida, quando se pretende ter este uso é necessário ter recursos para isso, como processadores poderosos e com muitos núcleos, dispositivos de memória rápida e muita RAM. Ao consumir tantos recursos é imperativo implementar técnicas para melhorar a eficiência do emulador, pelo que algumas das técnicas usadas são:

- Alocação de memória dinamicamente com base nos requisitos e uso do

equipamento virtualizado, no entanto, pode-se estipular um valor fixo em casos pertinentes;

- Comprimir blocos de memória que não são utilizados com muito frequência e armazená-los desta forma;
- Mover blocos de memória entre memória física (RAM) e virtual quando existem poucos recursos disponíveis, também conhecido por *memory swapping*;
- Também conhecida como deduplicação, esta técnica analisa todos os processos que se encontram na memória e elimina informação duplicada. Esta técnica é especialmente útil, visto que se se tiver em conta um laboratório com 10 equipamentos iguais por exemplo, é o equivalente a executar a mesma VM 10 vezes.

Mesmo utilizando estas técnicas de otimização, continua a ser um emulador que consome muitos recursos, por exemplo ao utilizar apenas um *router* Arista vEOS são necessários 2 vCPUs e 6133M de vRAM. Uma possível solução para este problema seria a utilizar contentores, no entanto, esta é uma funcionalidade que apenas se encontra disponível na versão Pro. [10]

2.5. Comparação entre Containerlab e EVE-NG

Tanto o Containerlab como o EVE-NG são ferramentas quer permitem simular uma topologia de rede, mas utilizam diferentes abordagens para atingir este fim.

O Containerlab apresenta uma abordagem orientada à utilização de contentores enquanto o EVE-NG utiliza virtualização. Assim, o desempenho e portabilidade ao utilizar uma topologia oriunda do Containerlab é superior, enquanto o EVE-NG disponibiliza uma estrutura mais robusta e um conjunto de funcionalidades mais rica.

Em termos de suporte de imagens, o EVE-NG tem a vantagem de existirem muitas mais disponíveis, tendo uma biblioteca que oferece uma grande variedade de fabricantes e modelos de equipamentos. Enquanto o Containerlab suporta cerca de 25 imagens de *routers* de vários vendedores, o EVE-NG suporta mais de 50 tendo muitos mais modelos por vendedor e vendedores em si.

A utilização destas duas ferramentas não poderia ser mais diferente, dado que o Containerlab utiliza uma linguagem declarativa denominada YAML e o EVE-NG utiliza uma interface *web*.

Finalizando, após ter sido feita esta pesquisa e comparação, concluiu-se que a utilização

do Containerlab é vantajosa dada a sua facilidade de configuração através de um ficheiro numa linguagem facilmente lida por humanos.

2.6. ANSIBLE

Ansible é uma ferramenta de automação de código aberto utilizada principalmente para automatizar tarefas repetitivas como gerir configurações de equipamentos físicos e virtuais. Quando se refere à gestão de configurações, depreende-se que se procura definir um estado que se pretende ter num equipamento e utilizando esta ferramenta forçá-lo ao mesmo. Este estado pode-se referir a vários componentes como se tem as dependências e pacotes certos instalados, os serviços que se deseja estarem a ser executados, entre outros.

Esta ferramenta utiliza uma linguagem declarativa denominada YAML para definir o estado referido anteriormente, pelo que utiliza uma abordagem de *Domain-Specific Language* (DSL). Ansible utiliza uma arquitetura descentralizada e *agentless* utilizando SSH para se conectar aos dispositivos. Ser *agentless* significa não ser necessário instalar o *software* em questão nas máquinas alvo, facilitando a gestão e a entrega destas.

Existem diversos componentes que compõe Ansible, sendo estes:

- *Control Nodes* – qualquer máquina que tenha Ansible instalado pode agir como um e executar comandos;
- *Managed Nodes* – também referidos como *hosts*, são as máquinas que se pretende interagir;
- Inventário – ficheiro que contém a lista de *managed nodes* e informações referentes aos mesmos, como IP, credenciais, entre outros;
- *Playbooks* – ficheiros escritos em YAML que descrevem as tarefas (*plays*) que devem ser executadas, com a vantagem de serem idempotentes. Neste caso, diz-se que é conferida a capacidade de idempotência, visto que grande parte dos módulos Ansible verifica se o estado final que se pretende atingir já foi alcançado. Tomando como exemplo um equipamento que já se encontra configurado, caso se execute um *playbook* com intuito de alterar a sua configuração sem fazer qualquer alteração à mesma, o *playbook* não é executado;
- Módulos – pode ser comparado a uma função que recebe parâmetros e executa uma ação predeterminada.

Em termos da gestão de configurações, existem dois modelos:

- *Pull* – os *node*s são atualizados dinamicamente conforme as configurações presentes no servidor;
- *Push* – o servidor central envia as configurações para os *nodes.*

Por norma é utilizado o modelo de *push*, no entanto, também é suportado o modelo *pull*, sendo que nesta situação abdica-se da funcionalidade *agentless*, porque é necessário instalar Ansible em todos os *nodes*.

Uma grande vantagem ao utilizar esta ferramenta é que foi construída em Python, permitindo não só executar código nesta linguagem como também criar *plugins* e módulos customizados e ser compatível com o *template engine* Jinja2.

Inicialmente, Ansible foi criado com o intuito de automatizar servidores Linux, pelo que o seu modo de operação foi desenhado para tal. Atualmente pode ser utilizado para automatizar equipamentos de rede, obrigando a existir uma diferenciação entre eles. Ao interagir com um servidor Linux, o *control node* conecta-se via *Secure Shell* (SSH) a cada equipamento, copia o código Python que realiza a automação para os mesmos e é posteriormente executado. Quando se trata de equipamentos de rede, não é feita de imediato conexão SSH para cada equipamento nem é copiado código. Na realidade o código é executado localmente no *control node*, sendo que este pode utilizar SSH, Telnet, *Simple Network Management Protocol* (SNMP) ou APIs para comunicar com os equipamentos, e são enviados comandos CLI, tendo em conta que alguns fabricantes (Cisco IOS-XR e Arista EOS) permitem que sejam copiados ficheiros Python para uma diretoria temporária funcionando como servidores Linux normais [11][12][13].

2.7. JINJA2

Jinja2 é um *template engine* de código aberto desenvolvido para Python que permite aos utilizadores definirem *templates* com marcadores para dados dinâmicos. Uma vez que a aplicação seja executada, estes marcadores são substituídos por dados reais e gera o resultado desejado.

Este é tipicamente utilizado em desenvolvimento *web*, no entanto, oferece funcionalidades relevantes ao trabalho a ser efetuado. Pretende-se utilizar ficheiros YAML para definir os dados e Jinja2 para a estruturar e gerar ficheiros de configuração.

Uma das principais vantagens em utilizar este *template engine* é a existência do conceito de herança, isto é, um *template* pode herdar parâmetros de outro *template*, não sendo necessário repetir código. Existe um *template* base que contem um esquema

comum a vários *templates* e a partir deste, estende-se para outros. Também oferece uma grande versatilidade em termos de operações, podendo utilizar-se ciclos, condicionais, filtros, entre outros que permitem utilizar somente os troços relevantes do *template*. [14]

Assim, fazendo um estudo da ferramenta tem-se como principais vantagens:

- Rapidez e eficiência, dado que compila o código para Python *bytecode* quando é carregado pela primeira vez, resultando num tempo de execução pequeno. Também suporta a funcionalidade *Ahead-of-Time compilation* (AOT);
- Flexível dado a quantidade de funções que suporta como macros, filtros e ciclos;
- Compatível com Python;
- Modo *SandBox* que permite processar *templates* num ambiente seguro, quando estes provêm de fontes não confiáveis;
- Permite execução síncrona e assíncrona, sendo a segunda especialmente importante visto que em alguns casos pode-se carregar muitos dados ou utilizar funções que demoram muito tempo a serem executadas, causando atrasos na execução dos *templates*, que por sua vez têm de aguardar o término de tarefas em segundo plano.

2.8. GITLAB

DevOps é uma abordagem que visa aumentar a capacidade de resposta a mudanças nos vários sistemas existentes num ambiente, entregando serviços mais rápidos e de alta qualidade. Para isso acontecer é necessário idealizar um projeto, desenvolvê-lo e colocá-lo em produção, implicando a realização de vários testes e mudanças em termos de código.

O Gitlab surgiu como uma plataforma *web* que permite gerir diversos repositórios que contêm código fonte, documentação, entre outros, inclui a possibilidade de criar *tickets* que indicam algum mau funcionamento do código e oferece uma componente de CI e CD que permite automatizar o processo de desenvolvimento, testagem e entrega das alterações efetuadas.

Na sua base, Gitlab é um sistema de controlo de versões que facilita a gestão de mudanças realizadas no código fonte ao longo do tempo. Isto é feito graças ao Git que

possibilita criar *branches* para desenvolvimento paralelo de aplicações, alterar troços de código e juntar essas mudanças ao código presente no *branch* principal.

Uma funcionalidade conveniente é a possibilidade de se poder utilizar esta plataforma *self-hosted* ou centralizada. Quando se refere a *self-hosted*, significa que se instala e executa o Gitlab em *hardware* ou estrutura de *cloud* pessoal, permitindo ter controlo total sobre a instância utilizada, customizar de forma a cumprir as necessidades pessoais e integrar com outros sistemas relevantes no contexto que se pretende, no entanto, é necessário existir conhecimento suficiente para manter esta solução. Ao utilizar a opção centralizada, consegue-se ter uma abordagem mais simples tanto de configuração como de uso, não tendo custos relacionados ao *hardware* que mantém o serviço, abdicando do controlo sobre o serviço, customização e segurança na proteção de dados.

A componente CI/CD é cada vez mais essencial para agilizar o trabalho dos desenvolvedores. Esta permite testar quaisquer alterações feitas no repositório, fazer a entrega das alterações em diversos ambientes (produção ou testes) baseado em regras definidas, facilita a visualização da *pipeline* para verificar possíveis erros ou *logs* relevantes e customizá-la com *scripts*, regras e condições para cumprir todos os requisitos sem ser necessário utilizar serviçosou ferramentas externas. [15][16][17]

Um dos aspetos mais importantes da componente CI/CD é o Gitlab Runner. Este é um *software* compatível com diversos SO e arquiteturas, que executa as tarefas indicadas na *pipeline*. Mais uma vez, existe a opção de utilizar *runners* partilhados ou *runners* instalados em *hardware* pessoal, também conhecidos como *self-hosted runners*. Ao utilizar um *runner* partilhado está-se sujeito à disponibilidade destes, ou seja, se existirem muitos utilizadores consecutivos pode existir tempo de espera, sendo que existem um limite de tempo de utilização da *pipeline* e *runners* na versão gratuita do Gitlab. Escolhendo utilizar *self-hosted runners* é necessário registar o mesmo para existir comunicação entre o Gitlab e a máquina onde o *runner* está instalado. Após ser feito o registo tem de se optar por um tipo de executor que por sua vez representa o ambiente onde as tarefas vão ser executadas, podendo escolher entre [17]:

- SSH - conecta-se a um equipamento remoto e executa comandos via SSH
- *Shell* - executa as tarefas onde o *runner* está instalado
- VirtualBox - utiliza a virtualização do VirtualBox para executar as tarefas num ambiente limpo, tendo como requisitos a existência de um servidor SSH e que

tenha uma *shell* compatível com *Bash* ou *PowerShell*

- Docker - utiliza contentores Docker para executar as tarefas
- Kubernetes - utiliza *clusters* de Kubernetes para executar as tarefas, fazendo chamadas à sua API

2.9. JENKINS

Jenkins é servidor de automação de código aberto muito utilizado para o desenvolvimento de *software* e automatização da testagem e entrega de aplicações, sendo considerado uma ferramenta de CI/CD.

Esta ferramenta apresenta uma metodologia simples e poderosa para automatizar *pipelines*, podendo ser integrada com uma grande variedade de tecnologias e criar condições que satisfaçam uma ampla gama de necessidades por parte do utilizador.

Fornece uma plataforma centralizada que permite a automatização de *pipelines*, desde mudanças de código até execução automática de testes unitários e consoante o resultado fazer a entrega das mudanças, recorrendo a tarefas denominadas por *jobs*.

Jenkins é utilizado principalmente através de uma interface *web* que agiliza a configuração e gestão de *jobs*. Cada *job* consiste num conjunto de tarefas a serem executadas como, por exemplo, verificar o código a partir de um sistema de controlo de versões, compilá-lo, realizar testes unitários e fazer a entrega das alterações para um ambiente de produção.

Para aumentar a flexibilidade desta ferramenta, utiliza-se uma funcionalidade baseada em *plugins* que permite estender as funcionalidades já existentes. Dado ter uma comunidade grande e ser uma plataforma reconhecida, dispõe de uma biblioteca de *plugins* diversificada deste integrar serviços básicos como o Git a fazer a entrega para serviços de *cloud* como a *Google Cloud Platform* (GCP) e *Amazon Web Services* (AWS).

Em termos de arquitetura, Jenkins pode utilizar o modelo de arquitetura distribuída, ou seja, um sistema com múltiplos componentes localizadas em diferentes máquinas que coordenam as suas ações para simular a existência de apenas uma. Pode ainda suportar uma arquitetura cliente-servidor, onde existem dois componentes principais na sua arquitetura, sendo estes o *master* e o *slave*. Este tipo de arquitetura apresenta grandes vantagens quando se pensa em cenários complexos, onde não faz sentido colocar a carga toda no servidor Jenkins, pelo que é preferível configurar vários *slaves* para

distribuir a carga por eles. Também é vantajoso caso se pretenda fazer testes em diferentes ambientes, configurar vários *slaves* com diferentes ambientes.

O *master* é o componente central da infraestrutura e, para além dos casos descritos anteriormente, está responsável por agendar e executar tarefas nele mesmo ou em *slaves*, juntar e mostrar os resultados vindos dos *slaves*, monitorizá-los e extrair código de repositórios. No que toca ao *slave*, este é a máquina onde são realmente executadas as tarefas designadas pelo *master* e posteriormente se reporta os resultados ao mesmo. [18] [19]

Assim, pode-se descrever os passos que ocorrem ao executar uma *pipeline* utilizando Jenkins pela seguinte ordem:

- Primeiramente são efetuadas mudanças no código fonte presente num repositório;
- O servidor Jenkins deteta que ocorreram alterações e é efetuado o *pull* do código fonte;
- O *master*, consoante a complexidade das tarefas e *slaves* disponíveis, atribui as tarefas a estes;
- Se existir algum erro o utilizador é notificado, mas caso contrário é feita a entrega no ambiente de testes;
- Após os testes serem concluídos, o *master* recolhe informações sobre estes e notifica o utilizador.

2.10. Comparação entre Gitlab e Jenkins

Cada vez mais é necessário adotar práticas que permitam agilizar o processo entre ser feita uma alteração no código fonte e esta ser aprovada para ser colocada em produção. Assim, foram criadas diversas ferramentas que permitem alcançar este objetivo, tais como Gitlab e Jenkins.

Gitlab é uma ferramenta que para além de permitir ser utilizado como repositório de código, também tem uma componente CI/CD completa. Esta é baseada em *runners* que executam as tarefas descritas num ficheiro YAML.

Jenkins é um servidor de automação que oferece uma componente CI/CD que também é baseada em tarefas, sendo que estas são uma sequência de passos que definem um

processo de testagem.

Uma das grandes vantagens ao utilizar Gitlab em relação a Jenkins, é que não necessita de ferramentas externas para funcionar, isto é, tanto o repositório de código como a *pipeline* CI/CD encontram-se na mesma plataforma ao invés de ser preciso integrar o repositório através de *plugins*. Tanto o Jenkins como o Gitlab têm uma componente de visualização da *pipeline* completa e um sistema de agendamento de tarefas robusto.

Em termos de plataformas suportadas, Gitlab apenas suporta SO baseados em UNIX como Linux CentOS, enquanto Jenkins suporta Linux, Windows e macOS.

Jenkins apresenta ainda algumas vantagens em relação ao Gitlab no que toca à sua licença, sendo esta gratuita em comparação à existência de uma edição gratuita, profissional e *ultimate*. Este tem ainda uma biblioteca de *plugins* muito extensa, contando com mais de 1700 que são atualizados regularmente.

Assim, com tudo o que foi descrito, acredita-se que a ferramenta mais interessante no contexto desta dissertação é o Gitlab dado a sua componente CI/CD ser facilmente configurável graças à possibilidade da utilização de *runners*.

2.11. PyATS/Genie

Python Automation Framework for Testing and Software Development (pyATS) é uma ferramenta de código aberto desenvolvida pela Cisco com o intuito de automatizar e testar topologias de rede.

Esta ferramenta foi construída utilizando Python, disponibilizando um conjunto de APIs e ferramentas de automação como gestão das conexões aos equipamentos, recolha de dados e a sua análise, tendo uma abordagem orientada a dados que permite ter um conjunto de testes que podem ser utilizados em várias configurações de rede.

Uma grande vantagem do pyATS é a possibilidade de integrar com diversos equipamentos de rede de diferentes fabricantes como:

- Arista;
- Cisco ASA;
- Cisco IOS;
- Cisco IOXE;

- Cisco IOSXR;
- Cisco NXOS;
- Juniper.

Estes são apenas alguns exemplos dos equipamentos diretamente compatíveis, no entanto, caso um equipamento não esteja contemplado nesta lista, pyATS também suporta conexões via CLI, NETCONF e RESTCONF.

O componente central desta ferramenta é denominado *testbed* e representa a infraestrutura de rede que se pretende testar. Este é um ficheiro YAML que contém informações sobre os equipamentos de rede, protocolo a utilizar na conexão ao equipamento e credenciais.

Sendo esta uma ferramenta altamente versátil e agnóstica, existe uma *Device Abstraction Layer* (DAL) que viabiliza uma interface comum de interação entre diferentes equipamentos. Esta é uma biblioteca Python denominada Genie e contem todas as ferramentas para a automatização da testagem de redes. Uma das grandes funcionalidades que Genie oferece é o *parse* de comandos para uma estrutura semelhante a JSON, facilitando a análise e coleta de dados. Por exemplo, caso se queira toda a informação em relação a *Border Gateway Protocol* (BGP) de um equipamento, utiliza-se um modelo que envia um conjunto de comandos que devolvem informação sobre BGP e estrutura-os de uma forma previamente documentada.

A partir do momento em que os cenários de teste são desenvolvidos, dada a arquitetura desta ferramenta, é possível executá-los em vários equipamentos de forma paralela, agilizando o processo.

Após o término dos testes, é essencial recolher os dados para serem analisados, pelo que pyATS oferece um sistema que gera relatórios customizados, controla e analisa resultados. É ainda possível realizar integração com *software* externo como Elasticsearch. [20]

Com tudo acima descrito, pode-se relatar os seguintes pontos favoráveis na utilização desta ferramenta:

- Permite a customização e adaptabilidade a vários cenários de teste quer em termos deconfigurações, quer requisitos de testagem;

- Permite diferentes métodos para estabelecer conexões com equipamentos de diversos fabricantes e até modelos mais antigos que não disponham de tecnologias mais recentes como APIs;

- Uma abordagem orientada a dados que facilita a reutilização de testes ou alteração mínima destes para configurações de rede específicas;

- Permite a execução de testes de forma paralela;

- É uma ferramenta de código aberto, pelo que tem grandes contribuições da comunidade em conjunto com as da equipa Cisco.

No entanto, esta pyATS também apresenta alguns pontos negativos tais como:

- Requer um conhecimento elevado na linguagem Python e conceitos de automação deredes, tendo uma curva de aprendizagem grande para iniciantes;

- Apesar de suportar vários equipamentos e de fabricantes diferentes, esta quantidade é baixa em relação a algumas ferramentas anteriormente mencionadas;

- Sendo uma ferramenta em desenvolvimento e recente, podem não existir automações disponíveis, sendo necessário a utilização de *plugins* ou utilização de ferramentas externas.

3

ARQUITETURA E IMPLEMENTAÇÃO PROPOSTAS

3.1. INTRODUÇÃO

Tendo-se realizado o estudo das potenciais tecnologias a serem utilizadas no Capítulo 2, este capítulo tem como objetivo apresentar a solução da arquitetura proposta e as principais considerações na implementação da mesma, de modo a atingir os resultados mencionados no Capítulo 1.

Considerando que o principal objetivo da arquitetura é conseguir agilizar o processo de entrega e testagem de uma rede, de forma a garantir flexibilidade, fiabilidade e eficiência, não se devem tomar decisões que limitem a futura utilidade e adoção desta arquitetura. Por outro lado, existem certas limitações derivadas das tecnologias utilizadas, sendo que algumas ainda são recentes e não existe uma grande comunidade e variedade de informações disponíveis.

Assim, optou-se por definir a arquitetura em cinco pontos principais, sendo estes:

- Gitlab;
- Gitlab Runner;
- Containerlab;
- Ansible;
- Python.

3.2. ORGANIZAÇÃO

O presente capítulo encontra-se organizado em secções tais que:

- Secção 3.3 – Visão geral da arquitetura e lógica da solução proposta;
- Secção 3.4 – Considerações acerca da componente Containerlab;
- Secção 3.5 – Considerações acerca da componente Ansible;

- Secção 3.6 – Considerações acerca da componente CI/CD;

3.3. Visão Geral e Lógica da Arquitetura da solução proposta

Tendo em conta a Figura 1 que representa a visão geral e lógica da arquitetura proposta para implementação, consegue-se identificar os seguintes componentes:

- **Gitlab:** responsável por armazenar todas as configurações dos equipamentos de rede, *playbooks* Ansible, ficheiros de configuração da topologia e testes a serem realizados. O utilizador interage diretamente com esta componente que automaticamente despoleta a *pipeline* CI/CD quando é feito um novo *commit* para o repositório;

- **Gitlab Runner:** responsável por coordenar a comunicação entre o Gitlab e a VM para se orquestrar a entrega da topologia e efetuar testes à mesma. A *pipeline* CI/CD é definida por *jobs* que especificam o que deve ser executado, podendo estes ter diversas condições à sua execução e sucesso;

- **Containerlab:** responsável pela entrega da topologia de rede definida no repositório. De entre todas as componentes utilizadas, esta é a mais limitadora por razões que vão ser discutidas da Secção 3.4;

- **Ansible:** responsável pela automatização de tarefas desde a configuração dos equipamentos à realização de testes mais simples de forma condicional. Este serve ainda como inventário dos equipamentos, onde armazena algumas opções essenciais para que seja possível conectar aos mesmos;

- **Ansible/Python:** tal como foi referido no ponto anterior, Ansible foi também utilizado para realizar alguns testes mais simples, no entanto foi necessário recorrer a *scripts* Python para testes mais complexos.

Pretende-se que cada vez que um utilizador faça uma alteração no repositório a ficheiros relevantes seja despoletada a *pipeline*. Esta começa por escolher um *runner* conforme as *tags* de cada *job* e disponibilidade deste. De seguida, é feita a clonagem do repositório para a VM, ficando com acesso a todos os ficheiros de topologia de rede, configuração e testes dos equipamentos. Estando estes disponíveis é inicializada a topologia recorrendo ao Containerlab seguindo o ficheiro de topologia presente no repositório e é feita a configuração automática destes por parte do Ansible. Finalizada a configuração, são executados testes utilizando Ansible e código customizado Python

que verifica o bom funcionamento da topologia. Caso a *pipeline* tenha sucesso é efetuado um *backup* das configurações presentes nos equipamentos em produção e as alterações de configuração efetuadas são enviadas para os equipamentos em produção.

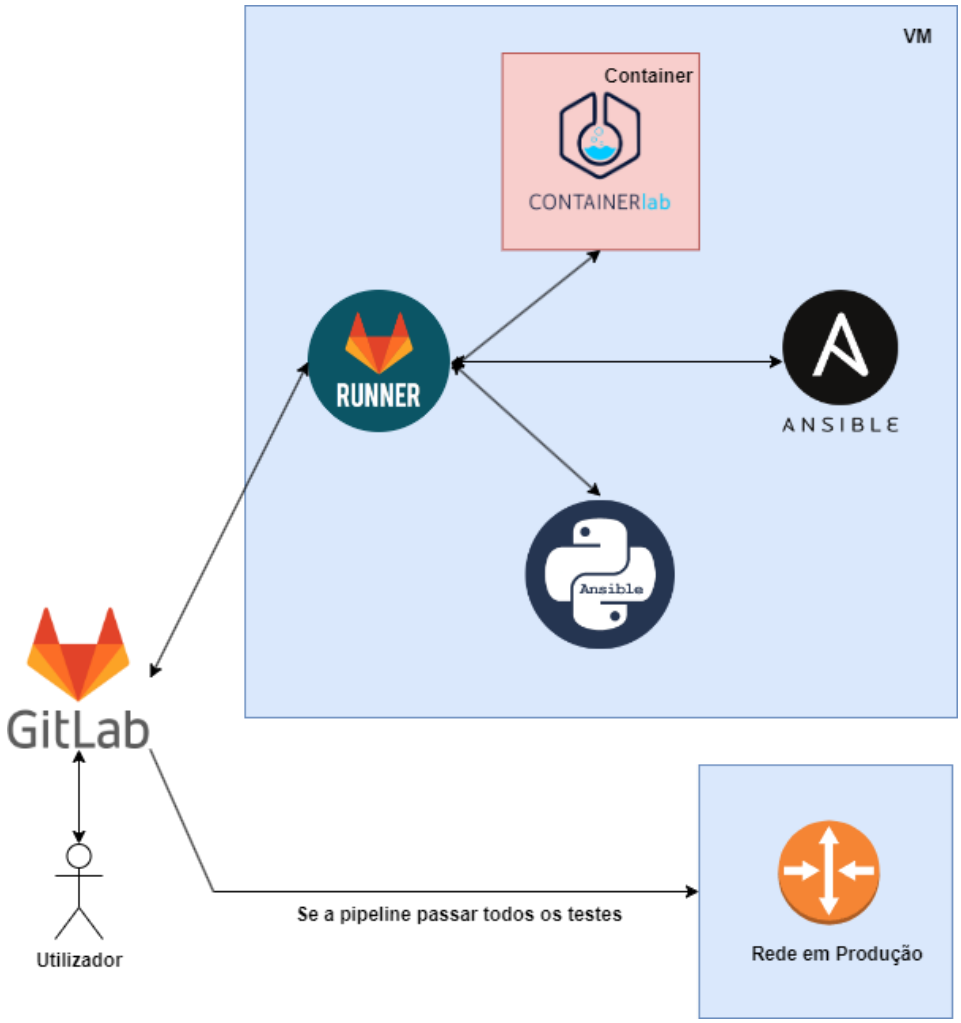


Figura 1 - Visão geral da Arquitetura proposta

De forma a melhor entender a estrutura desta Arquitetura e como realizar a sua instalação e configuração, pode-se consultar o Anexo A.

3.4. Considerações acerca da componente Containerlab

O desenvolvimento da arquitetura proposta encontra-se especialmente dependente do Containerlab, dado que é este que entrega a topologia pretendida.

Para se visualizar com maior detalhe um dos ficheiros utilizados nos testes efetuados, pode-se consultar o Anexo B.

Considere-se a Figura 2 que apresenta um exemplo do ficheiro que descreve a topologia e a respetiva representação gráfica da mesma.

```
name: LAB_TEST

topology:
  nodes:
    ceos1:
      kind: ceos
      image: ceos:4.28.0F
    ceos2:
      kind: ceos
      image: ceos:4.28.0F
    ceos3:
      kind: ceos
      image: ceos:4.28.0F

  links:
    - endpoints: ["ceos1:eth1", "ceos2:eth1"]
    - endpoints: ["ceos2:eth2", "ceos3:eth1"]
```

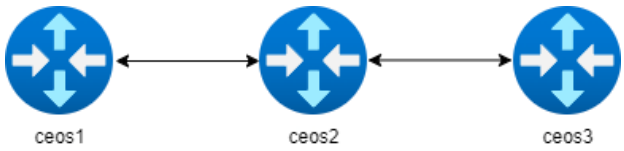


Figura 2 - Exemplo de topologia utilizando Containerlab

Este ficheiro é composto por vários campos que necessitam ser definidos, sendo os mais importantes:

- ***Kind***: indica o tipo de *node* que se pretende, visto que a sequência de execução de comandos difere entre *kinds* desde chamadas de sistema, diretorias montadas e comandos a executar;
- ***Image***: indica a imagem a ser utilizada no *node*, no caso da Figura 2 utilizou-se "ceos:4.28.0F" dado que essa é a versão especifica da imagem, no entanto poder-se-ia utilizar "ceos:*latest*" num cenário empresarial onde é necessário atualizar regularmente para a versão mais recente;
- ***Links:*** define as ligações que se pretende ter na topologia de uma forma simples onde se faz a relação entre o nome do equipamento e a porta de ambos os equipamentos, sendo tratado como *endpoints*. Apenas é necessário ter em atenção que dependendo da *kind*, a primeira porta de serviço disponível pode variar e a forma como é representada.

Apesar destes serem os principais campos a serem definidos, de modo a existir uma maior consistência, utilizou-se ainda o campo "mgmt-ipv4" para definir um endereço IPv4 de gestão específico para cada equipamento. Caso não se especifiquem os endereços são atribuídos sempre na gama "172.20.20.0/24", no entanto nem sempre são dados endereços de forma crescente consoante a ordem definida no ficheiro. A ação de definir endereços de gestão nos equipamentos é feita automaticamente pelo Containerlab, permitindo ferramentas como o Ansible comunicarem com estes sem existir qualquer tipo de configuração por parte do utilizador.

Por motivos que serão discutidos mais à frente vão ser mencionados alguns campos importantes que não foram utilizados como:

- ***Startup-config:*** identifica um ficheiro (local ou remoto) que contem a configuração que se pretende utilizar em cada equipamento. Este campo é particularmente útil caso se pretenda utilizar uma configuração antiga que se tenha feito *backup*, evitando ter de recorrer ao Ansible;

- ***Startup-delay:*** define quantos segundos se pretende que um *node* atrase a sua execução. Esta opção pode ser particularmente útil se existirem dependências entre contentores, para garantir a inicialização completa de serviços e aplicações que necessitem de mais tempo para começar a aceitar pedidos e também permite evitar o pico simultâneo de carga no arranque inicial da topologia;

- ***Binds:*** permite vincular uma diretoria da VM com a do contentor funcionado como uma pasta partilhada.

No desenvolvimento deste trabalho, apesar do que foi referido anteriormente acerca desta ferramenta suportar diversos vendedores, apenas foram utilizados *routers* Arista. Isto deve-se a dois fatores, disponibilidade das imagens e recursos necessários. Os principais vendedores que disponibilizam as imagens dos seus equipamentos otimizados para contentores gratuitamente são a Arista (cEOS) e a Nokia (Nokia SR Linux), enquanto os outros têm um custo. É ainda possível utilizar imagens de equipamentos que não sejam diretamente otimizados para contentores, sendo que a sua utilização reduz a eficiência da *pipeline*. Assim, decidiu-se realizar os testes utilizando *routers* Arista dada a sua rapidez na entrega e pouco consumo de recursos, tendo em conta as métricas descritas de seguida. Para suportar uma arquitetura capaz de lidar com equipamentos de diversos vendedores seria necessário adaptar os

ficheiros de configuração, *templates* Jinja2 e todos os testes desenvolvidos desde os comandos utilizados até à forma como a informação é extraída destes.

Para se melhor entender a diferença de tempo e recursos necessários à execução de *routers* de diferentes vendedores, sentiu-se a necessidade de realizar a comparação destes parâmetros entre *routers* Arista e Cisco, sendo que a Cisco não disponibiliza imagens otimizadas para contentores. Enquanto 5 *routers* Arista demoram cerca de 1 minuto e 30 segundos a serem inicializados, 1 *router* Cisco XRV9k demora cerca de 25 minutos. Em termos de recursos, os mesmos 5 *routers* Arista conseguem ser executados na VM utilizada no desenvolvimento da dissertação (os recursos encontram-se especificados na Secção 4.8), enquanto os recursos recomendados para 1 *router* Cisco XRV9k são 2 *processor cores* e 14GB de memória RAM, segundo a documentação oficial do Containerlab [9].

3.5. Considerações acerca da componente Ansible

Na arquitetura apresentada anteriormente na Secção 3.3 o uso do Ansible corresponde à configuração de equipamentos, inventário dos mesmos e automação de tarefas imprescindíveis na fase de testagem.

Tal como foi referido na secção anterior, a configuração dos equipamentos poderia ter sido feita sem recorrer ao Ansible, podendo simplesmente fornecer um ficheiro com todos os comandos de configuração a serem executados, no entanto optou-se por utilizar uma abordagem diferente para conferir ao utilizador uma experiência mais simples. Optou-se então pela utilização de ficheiros YAML compostos por *arrays*, *strings* e inteiros que representam toda a informação necessária a ser configurada. Recorrendo a *templates* Jinja2, esta informação é transformada numa semântica que pode ser aplicada aos equipamentos. Considere-se a Listagem 1 que apresenta um pequeno exemplo de como é construído o ficheiro de configuração e o respetivo *template* Jinja2.

```yaml
interfaces:
  - name: Ethernet1
    ipv4_addr: 172.16.12.1/24
  - name: Loopback0
    ipv4_addr: 172.16.0.1/32
```

```
{% for interface in interfaces %}
    {% if 'Loopback' in interface.name %}
        interface {{ interface.name }}
        ip address {{ interface.ipv4_addr }}
    {% else %}
        interface {{ interface.name }}
        ip address {{ interface.ipv4_addr }}
        no switchport
    {% endif %}
{% endfor %}
```

Listagem 1 - Exemplo de ficheiro de configuração e *template* Jinja2, respetivamente

A Listagem 2 apresenta o resultado do ficheiro de configuração e *template* Jinja2 da

Listagem 1. No caso deste exemplo pretende-se configurar as interfaces de um equipamento fazendo a distinção pelo seu nome, dada a necessidade de utilizar o comando "*no switchport*" nas interfaces de serviço.

```
interface Ethernet1
ip address 172.16.12.1/24
no switchport
interface Loopback0
ip address 172.16.0.1/32
```

Listagem 2 - Resultado do ficheiro de configuração e *template* anteriores

Tendo em conta o exemplo demonstrado na Listagem *1* e Listagem *2*, após a informação que se encontra no ficheiro de configuração YAML ter sido transformado em comandos pelo *template* Jinja2, o Ansible vai realizar uma conexão SSH para cada um dos *routers* e aplicar as configurações. Tal como foi referido na Secção 3.4, esta conexão apenas é possível porque é atribuído um IP de gestão automaticamente pelo Containerlab.

Estes foram pequenos exemplos destes ficheiros, no entanto pode-se consultar um dos ficheiros de configuração utilizados e o *template* Jinja2 na sua integra nos Anexos C e D, respetivamente.

Uma das limitações desta abordagem é que apenas são aplicadas configurações que estejam devidamente explícitas no ficheiro de configuração e no *template* Jinja2, pelo que é importante referir a estrutura desenvolvida para configurar os diversos serviços e protocolos que se pretende testar. De notar que caso exista alguma alteração sintáxica pelos fabricantes, estas alterações necessitam de ser realizadas no *template*, quer seja alterando os comandos presentes ou introduzir condições que verifiquem a versão do equipamento, tendo como resultado um *template* mais abrangente.

A configuração das credenciais de acesso tem como campos:

- Utilizador;
- Palavra-passe.

Em termos de endereçamento tem-se:

- Nome da interface;
- Endereço IP e máscara.

Para a configuração do OSPF é necessário:

- *Process* ID;
- *Router* ID;
- Interfaces não passivas;
- IP da rede de cada área OSPF e identificador da mesma.

Em relação à configuração do BGP:

- *Autonomous System* (AS);
- Vizinhos BGP, onde se especifica o seu endereço IP e AS;
- Redes que se pretende anunciar, sendo necessário indicar endereço IP e máscara.

A configuração do servidor de *Dynamic Host Configuration Protocol* (DHCP):

- *Subnet* que se pretende entregar IPs;
- *Default Gateway* da *subnet*;
- Servidores DNS que se pretende utilizar;
- O intervalo de endereços que se pretende dar IPs;
- A duração do *lease.*

Para a configuração do *Network Time Protocol* (NTP):

- IP ou nome DNS do servidor que se pretende utilizar.

Com os ficheiros de configuração e *templates* concluídos, seguiu-se para o desenvolvimento da automação de tarefas e testes.

Em termos de automação de tarefas, a principal a ser mencionada é a que permite a realizar a configuração dos equipamentos, representada Listagem 3. Para uma maior coerência e facilidade na construção das automações, definiu-se que os nomes dos ficheiros que contêm a informação para configurar os equipamentos deve ser "clab-nome_da_topologia-nome_do_*node*", ou seja, no caso da Figura 2 seria "clab-

LAB_TEST-ceos1" por exemplo. Para os *templates* Jinja2 também se optou por este método, mas desta vez utilizando o tipo de dispositivo, que no caso de um Arista será "arista.eos.eos".

```
pre_tasks:
  - include_vars: "../../config_vars/{{inventory_hostname}}.yml"

tasks:
#--------------------ARISTA-------------------------#
  - name: Configure Arista Routers using template
    ansible.netcommon.cli_config:
      config: "{{ lookup('template', '../../templates/{{ansible_network_os}}.j2') }}"
    when: ansible_network_os == 'arista.eos.eos'
```

Listagem 3 - *Playbook* para configurar equipamentos

Tendo-se demonstrado um *playbook* mais simples como a configuração dos *routers* da topologia de testes, também se colocou no Anexo E o *playbook* responsável por coordenar todos os testes relativos ao OSPF para se ter como referência na compreensão da metodologia utilizada.

Importante realçar que em todas as *plays* desenvolvidas, utiliza-se a declaração condicional "*when*" de modo a existir maior controlo sobre a execução das mesmas. Dando um exemplo, no que toca a verificar o IPv4 e estado das interfaces este não é muito relevante dado que todas as interfaces necessitam de o ter, todavia nem todas as topologias necessitam de ter *Open Shortest Path First* (OSPF) configurado. Assim, não faz sentido estar a avaliar esse conjunto de testes, pelo que recorrendo a declarações condicionais se vai verificar se existe alguma configuração relativa ao OSPF antes de realizar os seus testes, aumentando a eficiência da pipeline.

Outro aspeto de importância a abordar é o inventário utilizado pelo Ansible. Este é um ficheiro que define quais os *hosts*, grupos de *hosts* e variáveis que controlam como o Ansible interage com os estes. Como apenas se desenvolveu testes para equipamentos Arista, não houve a necessidade de criar um grupo específico para estes, somente um grupo denominado *routers*, no entanto poderia ser útil caso existissem testes para mais vendedores. Definiram-se ainda variáveis para o grupo *routers* necessários para o acesso aos equipamentos como:

- **ansible_become_method:** qual o método de *privilegie escalation* a usar, neste caso é *enable*;
- **ansible_become:** decide se *privilegie escalation* é utilizada, no caso do Arista é equivalente ao comando "*enable*";

- **ansible_user:** representa o utilizador ao entrar no equipamento;
- **ansible_password:** representa a palavra-passe ao entrar no equipamento;
- **ansible_network_os:** informa qual o vendedor do equipamento;
- **ansible_connection:** indica como o Ansible se deve ligar ao equipamento, sendo que ansible.netcommon.network_cli indica que se trata de um equipamento de rede;
- **ANSIBLE_HOST_KEY_CHECKING:** desativar a necessidade de verificar a autenticidade dos *hosts* ao compara a chave SSH com a lista de *know hosts*.

Tal como a definição da topologia de rede mencionada na Secção 3.4, o inventário do Ansible é necessário ser configurado pelo utilizador. Decidiu-se não se automatizar este passo para permitir ao utilizador criar os grupos de *hosts* que pretende e organizá-los da melhor forma tendo em conta a sua necessidade. No caso desta dissertação existem apenas dois grupos, arista e *production*, onde o primeiro tem o nome DNS de todos os equipamentos a serem entregues pelo Containerlab e o segundo contém o nome DNS dos equipamentos em produção, permitindo assim reutilizar a mesma *play* desenvolvida para configurar a topologia de rede alterando apenas o grupo onde vai ser aplicada.

Apesar de alguns testes terem sido desenvolvidos na componente de Ansible, optou-se por discutir a metodologia dos mesmos no Capítulo 4.

3.6. Considerações acerca da componente CI/CD

De forma a utilizar a componente CI/CD do Gitlab é necessário escolher um Gitlab Runner. Tal como foi referido no Capítulo 2.8, pode-se escolher entre *runners* públicos ou *self-hosted*, pelo que após se ter tido em conta as necessidades desta arquitetura optou-se por utilizar um *runner self-hosted.*

Ao contrário de um *runner* público, ao utilizar-se a opção *self-hosted* precisa-se de o instalar e configurar na VM que se pretende utilizar, sendo que apesar deste processo ser simples foi necessário ter em conta que tipo de *runner* se pretendia, particularmente entre a opção *Shell* e Docker. Enquanto *Shell* permite executar comandos diretamente na VM onde o *runner* se encontra instalado, Docker cria um contentor onde os comandos são executados. Como esta implementação utiliza vários *softwares*, seria necessário criar uma imagem Docker customizada com todas as dependências necessárias. Este não é um grande desafio, no entanto é preciso ter em conta que a

entrega dos equipamentos é feita pelo Containerlab que por si só utiliza Docker, ou seja, ao utilizar a opção Docker para o *runner* iria-se ter um contentor Docker que por sua vez iria ter vários contentores correspondentes aos equipamentos da topologia, o que iria aumentar a complexidade da *pipeline* de uma forma desnecessária. Se se pensar num caso onde existem vários utilizadores em simultâneo a tentar utilizar o mesmo repositório, poder-se-ia pensar que isolar as topologias em diferentes ambientes seria benéfico, porém o único cuidado a ter é não utilizar nomes de topologias iguais no Containerlab e os mesmos endereços IPv4.

Com o *runner* instalado e devidamente configurado, resta definir a *pipeline* pretendida através do ficheiro ".gitlab-ci.yml", sendo este o nome padrão a ser utilizado no Gitlab. Para garantir que a *pipeline* é executada na devida ordem, dado que existem passos dependentes de outros, definiram-se diferentes fases que vão ser executadas pela seguinte ordem:

- ***Build:*** realiza a entrega da topologia de acordo com o ficheiro que a define. Esta fase apenas é constituída por um *job* que realiza o comando para ser feita a entrega da topologia;

- ***Configuration:*** configura todos os equipamentos da topologia que tenham um ficheiro de configuração associado. Semelhante à fase anterior, esta apenas é constituído por um *job*, sendo que neste caso vai ser executado um *playbook* Ansible que configura os *routers*;

- ***Test:*** realizam-se todos os testes pretendidos à topologia. Esta fase é constituída por 6 *jobs*, sendo estes cada um dos testes detalhados no Capítulo 4. Cada um destes *jobs* executa um *playbook* Ansible que realiza os testes definidos, quer estes sejam realizados diretamente no Ansible ou executem um *script* Python;

- ***Backup:*** se todos os testes anteriores forem bem-sucedidos, antes que as configurações novas sejam alteradas nos equipamentos em produção é feito um backup das configurações de todos estes, caso aconteça algo inesperado não previsto nos testes realizados. Para isto foi desenvolvido um *playbook* que extrai as informações dos *routers* e coloca-as num ficheiro numa diretoria escolhida;

- ***Cleanup:*** destrói a topologia e todos os contentores associados à mesma, de modo a não ocuparem recursos de uma forma desnecessária. Esta fase é apenas composta por um *job* que realiza o comando *destroy* indicando que a topologia pode ser

destruída;

- *Production:* Caso todos os testes sejam executados com sucesso, as alterações efetuadas são realizadas na topologia em produção. Aqui é executado um *playbook* semelhante ao que configurado os equipamentos, no entanto é aplicado a um grupo diferente referido no inventário Ansible. Neste caso, o inventário é composto por um grupo denominado Arista que contém todos os *routers* entregue na topologia de testes e outro denominado *Production* que contém todos os *routers* presentes na topologia em produção.

Todas estas fases, bem como as instruções de cada *job* podem ser consultadas no Anexo F.

A comunicação entre o Gitlab e Gitlab *Runner* é inicializada pelo *Runner*, onde este verifica periodicamente a instância do Gitlab se existem *jobs* a serem executados. Quando um novo *job* é criado, neste caso quando a *pipeline* é despoletada, o Gitlab seleciona um *Runner* disponível de acordo com a disponibilidade do mesmo e *tag* do *job*. Com o Runner selecionado, são enviadas as instruções que se pretende realizar em cada *job* pelo Gitlab. Por sua vez o *Runner* executa os *jobs* indicados pelo Gitlab na máquina onde está instalado, sendo que durante a execução do mesmo, o *Runner* envia atualizações regularmente ao Gitlab, podendo-se monitorizar o estado dos *jobs* em tempo real. Por fim, após a execução de todos os *jobs* o *Runner* retorna o estado final destes e atualiza a interface do Gitlab.

Considere-se a Listagem 4 que representa o *job* que executa o teste que permite verificar o IPv4 e estado das interfaces dos equipamentos. Tal como foi referido, este *job* enquadra-se na fase *test* e tem a *tag server*. Esta *tag* permite identificar qual o *runner* a ser utilizado, tendo em conta que no presente cenário de teste apenas existe um, no entanto num contexto empresarial onde exista uma grande utilização do repositório faz sentido a existência de vários para acomodar todos os utilizadores. Em termos do campo *script*, este é composto pelo comando que executa o *playbook* Ansible especificando o inventário. É possível ainda observar que no *script* é passada a variável "ANSIBLE_HOST_KEY_CHECKING", porque apesar desta se encontrar descrita no inventário, nem sempre é tida em conta no momento de conexão com os *routers*, pelo que por motivos de consistência é necessário colocá-la em todos os *jobs* que façam a execução de *playbooks* Ansible. É ainda importante notar a utilização do campo "*retry*", que permite em caso de falha de um *job* repeti-lo sem dar a *pipeline* como mal

sucedida, sendo particularmente importante no *job* de configuração de *routers* e alguns testes como OSPF. Isto deve-se ao facto do *job* de configuração ser imediatamente a seguir ao *job* de entrega da topologia e por vezes apesar do *router* já se encontrar acessível, alguns serviços ainda não estarem complemente funcionais, daí também ser importante em testes mais complexos como os de OSPF. Poder-se-ia também ter utilizado o campo "*sleep*" nesta situação, no entanto optou-se pela opção acima descrita para não limitar a *pipeline*.

```
test_ip_address:
    stage: test
    tags:
        - server
    script:
        - ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook roles/check_ip_address/main.yml -i "inventory"
    retry: 2
```

Listagem 4 - *Job* que despoletado para testar o endereço IPV4 e estado das interfaces

4

DEFINIÇÃO E AVALIAÇÃO DOS TESTES REALIZADOS

De modo a conseguir-se uma *pipeline* o mais realista possível em relação a um cenário empresarial, tentou-se encontrar uma metodologia de verificação oficial que dite uma topologia de rede funcional, o que se provou difícil uma vez que cada rede tem os seus objetivos, mecanismos e métricas que regem o quão rigorosa e complexa necessita ser.

Assim, dada a inexistência de tal metodologia, decidiu-se escolher os testes a serem realizados com base nos protocolos de encaminhamento e serviços mais comuns em *routers*. Como esta dissertação tem um foco maior na *pipeline* em si do que na componente de redes não se considera necessário ter um cenário muito complexo, mas sim um cenário diversificado e que demonstre que uma abordagem destas é possível atualmente mesmo tendo as suas limitações.

Em termos de testes para protocolos de encaminhamento, optou-se por desenvolver para OSPF e BGP, dada a sua popularidade e relevância na comunidade de redes. No que toca a serviços, escolheu-se realizar testes para o DHCP e DNS, mesmo tendo em conta que em grandes ambientes empresariais o DHCP é normalmente fornecido através de um servidor dedicado. São ainda verificados todos os elementos configurados por uma questão de sanidade e por fim confirma-se que todos os equipamentos são alcançáveis.

Contudo o que foi descrito acima é importante realçar que se tentou idealizar um equilíbrio entre informação fornecida pelo utilizador e testes que sejam possíveis realizar a partir dos mesmos. Se se obrigasse o utilizador a inserir mais dados no ficheiro de configuração estaríamos presente um cenário muito básico, onde simplesmente se confirmava se o que se encontra nos ficheiros foi configurado corretamente. Ao invés disso, tentou encontrar um equilíbrio para que o utilizador forneça apenas os dados expressamente necessários à configuração pretendida e os testes façam essas verificações sem informação extra.

Nas próximas secções vai ser descrita em maior detalhe a metodologia utilizada na realização dos testes, bem como exemplos de sucesso e falha dos testes.

4.1. Testes de Verificação ao OSPF

No caso do OSPF, pretende-se verificar em que área se encontra cada interface, os vizinhos OSPF, *router-id* e interfaces passivas.

Começou-se por utilizar o comando "*show ip ospf interface* nome_da_interface | *exclude isPassive*" de modo a obter a informação configurada em cada interface relativa ao OSPF, tendo como exemplo da resposta na Figura 3.

```
ceos1#show ip ospf interface Ethernet1 | exclude isPassive
Ethernet1 is up
  Interface Address 172.16.12.1/24, instance 1, VRF default, Area 0.0.0.0
  Network Type Broadcast, Cost: 10
  Transmit Delay is 1 sec, State Backup DR, Priority 1
  Designated Router is 172.16.0.2
  Backup Designated Router is 172.16.0.1
  Timer intervals configured, Hello 10, Dead 40, Retransmit 5
  Neighbor Count is 1
  No authentication
  Traffic engineering is disabled
```

Figura *3* - Exemplo de resposta ao comando *show ip ospf interface* nome_da_interface

Neste caso, escolheu-se realizar o comando para a interface Ethernet1, com o IP 172.16.12.1 que por sua vez pertence a uma rede com uma máscara /24 e encontra-se na área 0.0.0.0, tendo em conta que ao utilizar o filtro "*exclude isPassive*" apenas apresenta *output* caso essa interface seja *no passive.*

Com estas informações adquiridas a partir do *router* e das informações presentes no ficheiro de configuração respetivo, consegue-se validar estes campos.

Para se conseguir validar o *router-id* é necessário recorrer ao comando "*show ip ospf*", uma vez que esta informação não se encontra no comando anterior. A Figura 4 apresenta um excerto do *output* deste comando.

```
ceos1#show ip ospf
OSPF instance 1 with ID 172.16.0.1 VRF default
 Supports opaque LSA
```

Figura 4 - Exemplo de um excerto do *output* do comando *show ip ospf*

Dentro deste mesmo *script* utiliza-se o comando "*show ip ospf neighbors*" para obter os endereços IP dos vizinhos, como se pode visualizar na Figura 5. Tendo estes endereços, é necessário compará-los aos IPs que representam as áreas OSPF, sendo que caso o IP adquirido através do comando se encontre dentro da rede descrita no ficheiro de configuração, dá-se o teste como bem-sucedido.

```
ceos2>show ip ospf neighbor
Neighbor ID     Instance VRF      Pri State                  Dead Time   Address         Interface
172.16.0.1      2        default  1   FULL/BDR               00:00:34    172.16.12.1     Ethernet1
172.16.0.3      2        default  1   FULL/DR                00:00:35    172.16.23.1     Ethernet2
```

Figura 5 - Exemplo de *output* do comando *show ip ospf neighbor*

Com os testes concluídos, os resultados destes podem ser visualizados de duas formas, podendo-se optar por ter mais ou menos detalhe. No caso de se pretender ter mais detalhe é possível ver os *logs* do *job* pela interface *web* do Gitlab, onde dependendo se este foi bem-sucedido ou não, apresenta uma mensagem diferente tal como apresentado na Figura 6.

```
"msg": "Interface Ethernet1 is in area 0.0.0.0 of clab-ceos2-ceos1"
"msg": "Interface Ethernet1 of clab-ceos2-ceos3 is in the wrong area. Current area is 0.0.0.0"
```

Figura 6 – Exemplo de diferentes mensagens caso o *job* que verifica as áreas OSPF tenha sido bem-sucedido ou não, respetivamente

No caso de se pretender uma visualização mais simplificada, também através da interface *web* do Gitlab, consegue-se rapidamente verificar se o job teve sucesso ou não, demonstrado na Figura 7.

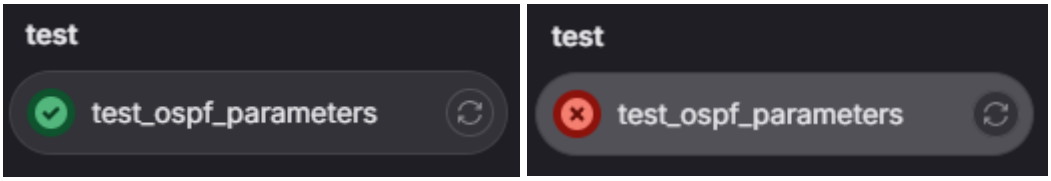


Figura 7 – Exemplo dos diferentes tipos de ícones caso o *job* tenha sucesso ou não, respetivamente

4.2. Testes de Verificação ao BGP

À semelhança dos testes ao OSPF, decidiu-se desenvolver todos os testes relacionados ao BGP num único *script* Python, dada a complexidade na extração de informação dos comandos e a sua posterior validação.

O *playbook* Ansible é semelhante ao utilizado nos testes ao OSPF, sendo que este tem a mesma função, como representado na Listagem 5.

```
- name: execute bgp test script
  ansible.builtin.script:
    executable: python3
    cmd: test_bgp_session.py {{inventory_hostname}} {{ansible_user}} {{ansible_password}} {{ansible_network_os}}
  register: output
  when: bgp is defined
```

Listagem 5 - *Playbook* Ansible para testar BGP

No campo "cmd" presente na Listagem 5, para além do nome do *script* a ser lançado,

são também utilizados alguns parâmetros necessários para realizar a conexão ao *router*, como o nome de DNS, o utilizador, palavra-passe e o tipo de *router* a ser testado.

Com toda a informação passada por parâmetro e recorrendo à biblioteca Netmiko, é realizada a conexão ao *router* via SSH e enviado o comando que se pretende. Esta é uma biblioteca que permite estabelecer conexões a uma variedade de equipamentos, que por predefinição é utilizado SSH para realizar as mesmas, mas também suporta Telnet e Serial. Com a ligação estabelecida é então possível enviar comandos e configurar equipamentos de rede através desta biblioteca. Foi ainda tirado partido de uma outra biblioteca denominada TextFSM que é diretamente compatível com o Netmiko e que permite filtrar a informação retornada por um comando a partir de *templates*. Estes templates encontram-se associados a comandos utilizados nos *routers* e a filtragem da informação é feita através de expressões regulares (regex), sendo que não existem *templates* para todos os comandos, daí não se ter conseguido recorrer a esta biblioteca em todos os testes.

No caso do BGP, pretende-se verificar o IP e os vizinhos BGP, o AS onde estes se encontram e o estado das ligações BGP.

Ao contrário dos testes ao OSPF, este apenas necessita de apenas um comando para obter a informação relevante, sendo este "*show ip bpg summary*". Na Figura 8 encontra-se um exemplo do *output* deste comando.

```
ceos2#show ip bgp summary
BGP summary information for VRF default
Router identifier 172.16.0.2, local AS number 65002
Neighbor Status Codes: m - Under maintenance
  Neighbor         V  AS           MsgRcvd   MsgSent  InQ OutQ  Up/Down State   PfxRcd PfxAcc
  172.16.12.1      4  65001            34        35    0    0 00:24:18 Estab   2      2
  172.16.23.1      4  65003            34        34    0    0 00:24:06 Estab   2      2
```

Figura 8 - Exemplo de *output* do comando *show ip bgp summary*

Utilizando o *output* representado na Figura 8 e as informações que se encontram no ficheiro de configuração do *router*, pode-se comparar as mesmas e averiguar se o BGP se encontra bem configurado. Apesar dos testes se encontrarem todos no mesmo *script*, para que exista uma maior granularidade caso seja preciso identificar erros, separou-se em diferentes funções no *script* onde cada uma retorna um *boolean*, como se pode observar um exemplo de *output* na Figura 9.

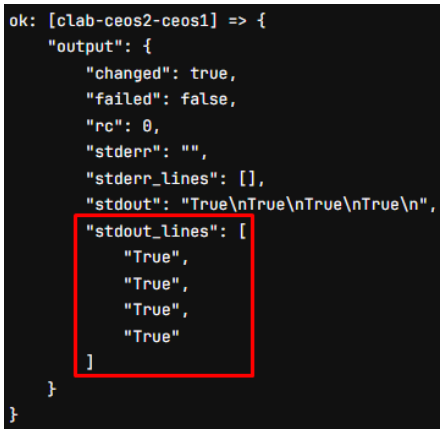


Figura 9 - Exemplo do *output* do *script* para testar BGP

Apenas se considera que o BGP esteja bem configurado e funcional se todos os testes retornarem o estado "*True*". Do lado do Ansible, este vai registar este *output* e consoante o estado retornado, vai considerar o teste como bem-sucedido ou não do lado do Gitlab, também retornando mensagens como exemplificado nos testes ao OSPF.

4.3. Testes de Verificação ao DHCP

Em termos dos testes ao DHCP, encontraram-se dificuldades na realização destes, sendo a mais significante não ser possível testar totalmente este serviço devido a uma limitação no Containerlab, no entanto decidiu-se testar a configuração do servidor DHCP mesmo que de forma incompleta.

O comando utilizado para extrair toda a informação necessária para testar o servidor DHCP encontra-se no comando "*show dhcp server*", tendo como exemplo do *output* deste na Figura 10.

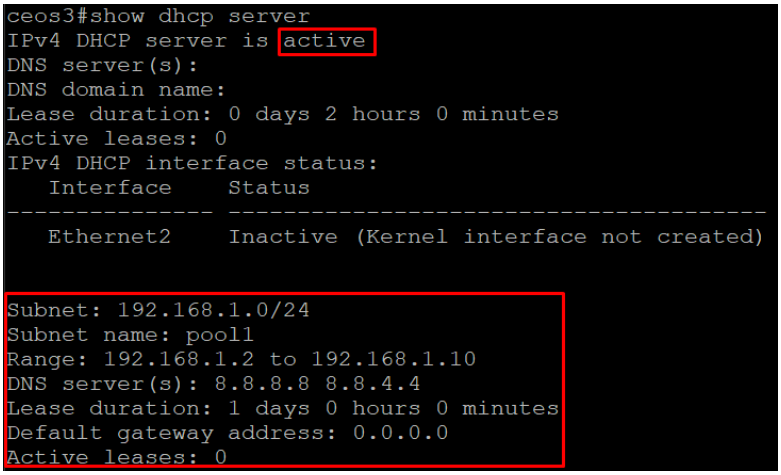


Figura 10 - Exemplo de *output* do comando *show dhcp server*

Tendo em conta a configuração exemplo apresentada na Listagem 6, tem-se todos os dados necessários para validar a mesma. Utilizou-se a mesma metodologia dos testes relativos ao BGP, onde se utilizou um *script* para comparar as informações provenientes tanto do *router* como do ficheiro de configuração e retornar os resultados dos testes ao Ansible.

```yaml
dhcp:
  subnets:
    - network: 192.168.1.0/24
      gateway: 192.168.1.1
      dns-server:
        - 8.8.8.8
        - 8.8.4.4
      start: 192.168.1.2
      end: 192.168.1.10
      lease: 1 days 0 hours 0 minutes
```

Listagem 6 - Excerto de uma configuração exemplo de um servidor DHCP

De notar que, tal como foi discutido anteriormente, apesar do servidor DHCP se encontrar ativo, como a interface no Kernel não foi criada corretamente não é possível atribuir *leases*. O que se pretendia testar seria, utilizando o comando "*show dhcp server leases*", verificar o estado do *lease*, o IP entregue e quais os *routers* que utilizam esta funcionalidade.

4.4. Testes de Verificação ao NTP

Para testar-se a configuração de um servidor de NTP escolheu-se utilizar o comando "*show ntp associations*" que apresenta informações acerca de todos os servidores configurados. No caso da Figura 11, os endereços utilizados para fins de exemplo são

ficcionais, daí a maior parte dos campos se encontrar com o valor 0. No entanto, para configurar um simples servidor de NTP apenas é preciso um comando pelo que se assume que o servidor se encontra funcional caso se consiga realizar um *ping* para o servidor em questão. Assim, através do comando anteriormente referido, recolhe-se os endereços IP ou nomes DNS e realiza-se um *ping* para os mesmos. Neste *ping* são enviados 5 pacotes e considera-se o teste como bem-sucedido se forem recebidos pelo menos 4 pacotes.

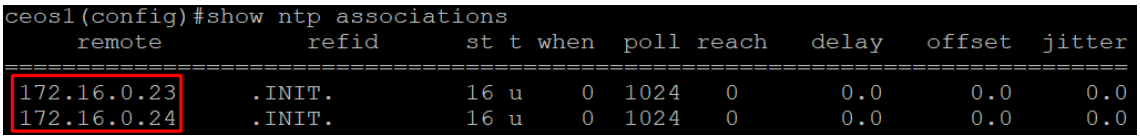


Figura 11 - Exemplo de *output* do comando *show ntp associations*

4.5. Testes de Verificação do Endereçamento e Estado das Interfaces

Os testes realizados para verificar se o endereçamento e o estado das interfaces são mais simples, pelo que se optou por fazer tudo em Ansible. Ao se indicar que um teste é menos complexo em comparação a outro, significa que a informação retornada pelo comando apresenta uma estrutura mais previsível, não necessitando de verificações extra para garantir a correta aquisição da informação.

Estes testes utilizam uma metodologia semelhante aos testes para verificar as áreas OSPF, ou seja, é enviado um comando a partir do Ansible e a partir do *output* do mesmo é feita a verificação do mesmo tendo em conta os ficheiros de configuração. Neste caso utilizou-se o comando "*show ip interface* nome_da_interface", obtendo informações detalhadas de cada interface. A *play* que envia este comando para o *router* recolhe informações do ficheiro de configuração e ao realizar o comando guarda o *output*, como se pode verificar na Listagem 7. Um exemplo do *output* deste comando encontra-se na Figura 12.

```
- name: Collect Interface and IP Status
  eos_command:
    commands:
      - show ip int {{item.name}}
  register: output
  loop: "{{interfaces}}"
  when: ansible_network_os == 'arista.eos.eos'
```

Listagem 7 - *Playbook* para recolher sobre o endereço IP e estado das interfaces de um Arista

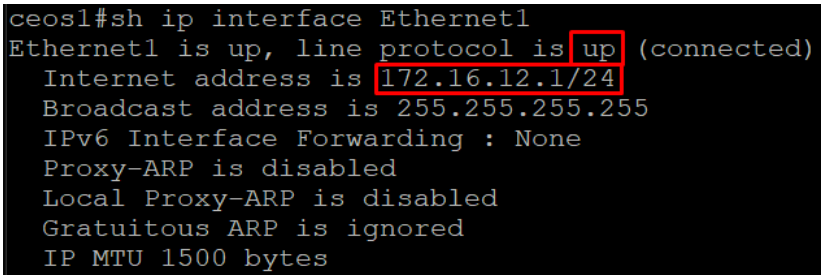


Figura 12 - Exemplo de *output* do comando *show ip interface Ethernet1*

Mais uma vez, com as informações recolhidas é feita a comparação com os ficheiros de configuração e verifica-se o bom ou mau funcionamento da configuração.

4.6. Testes de Verificação ao *Ping* entre *Routers*

Por último, o teste de *ping* tem como objetivo averiguar se todos os *routers* conseguem comunicar uns com os outros quer se encontrem conectados diretamente ou não. Para atingir este fim utilizou-se o comando "*show lldp neighbors detail*", tendo em conta que se optou por recorrer ao protocolo LLDP dado que permite descobrir vizinhos e informações dos mesmos, em particular o seu IP de *loopback*. Optou-se por apenas realizar *pings* para os IPs de *loopback*, uma vez que se pretende verificar que é possível comunicar com todos os *routers* independentemente da ligação entre eles e esta interface encontra-se sempre disponível caso o *router* apresente um funcionamento normal.

O *output* do comando anteriormente referido apresenta muita informação, sendo a maior parte dela inútil para o teste em causa, como pode observar na Figura 13. Apenas se pretende recolher o IP de *loopback* do *router* vizinho. Com o IP é então realizado um *ping* para o mesmo, onde são enviados 5 pacotes, pelo que apenas se considera que este teste teve sucesso se forem recebidos pelo menos 4 deles.

```
ceos2#sh lldp neighbors detail
Interface Ethernet1 detected 1 LLDP neighbors:

  Neighbor 001c.739d.c0f6/"Ethernet1", age 19 seconds
  Discovered 0:27:02 ago; Last changed 0:26:49 ago
  - Chassis ID type: MAC address (4)
    Chassis ID     : 001c.739d.c0f6
  - Port ID type: Interface name(5)
    Port ID      : "Ethernet1"
  - Time To Live: 120 seconds
  - System Name: "ceos1"
  - System Description: "Arista Networks EOS version
  - System Capabilities : Bridge, Router
    Enabled Capabilities: Bridge, Router
  - Management Address Subtype: IPv4
    Management Address         : 172.16.0.1
    Interface Number Subtype   : ifIndex (2)
    Interface Number           : 5000000
    OID String                 :
  - IEEE802.1 Port VLAN ID: 0
  - IEEE802.1/IEEE802.3 Link Aggregation
    Link Aggregation Status: Capable, Disabled (0x01)
```

Figura 13 - Exemplo de *output* do comando *show lldp neighbors detail*

4.7. Teste de Verificação da *Pipeline* Completa

Com todos os testes devidamente desenvolvidos e expostos, falta apenas testar a *pipeline* inteira.

Como foi descrito na secção anterior, a *pipeline* encontra-se dividida entre cinco fases, tal que apenas se avança para a próxima se não houver nenhum erro na anterior e não começa a próxima fase sem terminar os testes da fase anterior, como se pode observar na Figura 14. Neste caso existiu um erro na fase *test* no teste *test_ospf_meighbors* e apesar dos restantes testes da fase terem tido sucesso, a *pipeline* não avançou para a fase de *backup* e c*leanup*.

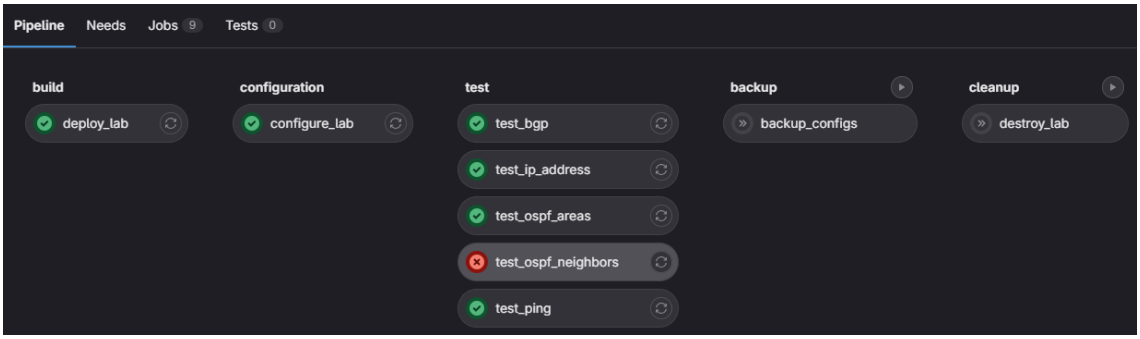


Figura 14 - Exemplo de um pipeline inteira sem sucesso

Na Figura 15 pode-se observar de forma destacada o tempo de execução de toda a *pipeline*. Neste caso, existiu um tempo de execução total de 1 minuto e 29 segundos, desde o momento em que foi realizado o *commit* com as alterações pretendidas até à

conclusão dos testes e destruição da topologia. Este cenário conta com a entrega de 3 *routers* Arista, onde foram configurados os seus endereços IPv4 e informações respetivas ao OSPF. O ponto crítico e de avaliação desta dissertação encontra-se aqui demonstrado, uma topologia entregue e testada com os requisitos definidos num curto espaço temporal.

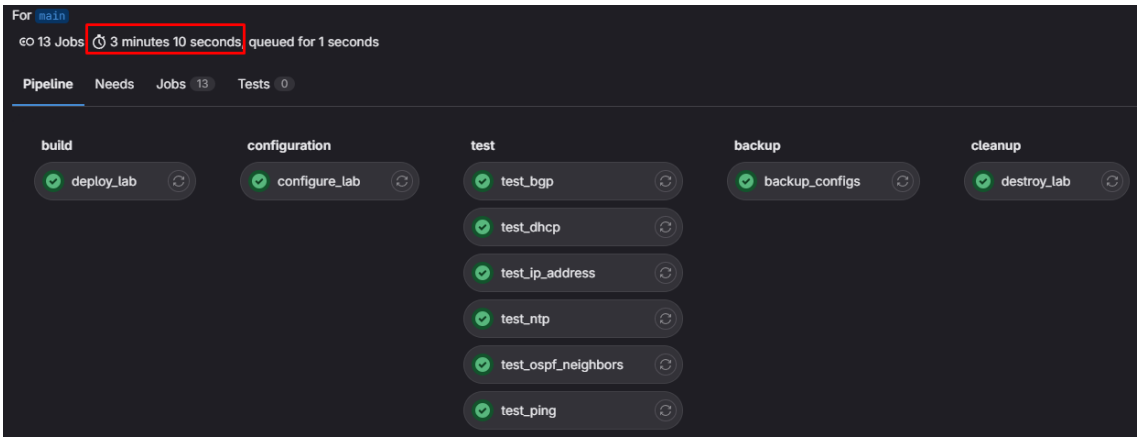


Figura 15 - Exemplo de uma *pipeline* completa

4.8. Resultados dos Testes Efetuados

Após a configuração de toda a *pipeline* e definição de todos os testes realizados, foi possível analisar a solução proposta, de modo a determinar se é uma solução viável. A métrica utilizada para avaliar a proposta é o tempo de execução de toda a *pipeline*, desde o momento em que é feito um *commit* com alterações à topologia de rede ou aos ficheiros de configuração.

De modo a manter os testes coerentes todos eles foram efetuados no mesmo ambiente, sendo este uma VM com Ubuntu 22.04.2 LTS como SO, 12GB de memória RAM, 100GB de armazenamento e 3 *processor cores*.

Como a métrica em estudo é o tempo de execução da *pipeline*, pretende-se avaliar como este varia consoante o tipo de topologia de rede e a quantidade de equipamentos em cada topologia. Tendo em conta que não se avaliou a fase em que as configurações são feitas aos equipamentos em produção, uma vez que seria necessário virtualizá-los o que por sua vez iria ocupar mais recursos, podendo atrasar a *pipeline* com um cenário irrealista.

Entre as diversas topologias de rede existentes, escolheram-se 4 para serem testadas sendo elas *Daisy Chain*, Anel, Estrela e Malha. Todas em estas topologias foram ainda

testadas com 5, 7 e 9 *routers*, tendo sido efetuados 5 simulações para todas as quantidades de *routers* em cada topologia. Nas secções seguintes vão ser detalhados os testes efetuados, bem como apresentados os resultados obtidos.

4.8.1. Testes com Topologia *Daisy Chain*

Esta topologia é das mais simples que existe, onde se tem um *router* diretamente ligado a outro, considere-se a Figura 16.

Figura 16 - Topologia *Daisy Chain* com 5 *routers*

Algo a ter em conta na realização de todos os testes, prende-se no facto de que para testar o DHCP é necessário existir uma ligação entre dois equipamentos, neste caso entre dois routers, sendo essa a razão de existirem duas ligações entre cada um.

Tomando como exemplo a topologia da Figura 16, cada *router* está configurado com duas interfaces que representam uma ligação direta entre eles e duas ligações extra para testar o DHCP, com exceção dos *router*s situados nos extremos que apenas têm uma interface de ligação direta e uma para testar o DHCP. Em termos, de protocolos de encaminhamento, configurou-se tanto OSPF como BGP com o intuito de todos os *routers* estarem alcançáveis uns aos outros, sendo que cada *router* se encontra numa área OSPF e AS diferentes dos outros e contêm *processID* e *router-id* diferentes. No que toca a serviços, foi configurado DHCP com servidores de DNS, intervalos de IP e *leases* diferentes e foi também configurado NTP com diferentes IPs.

Após a realização dos testes com 5 *routers*, realizaram-se os mesmo para topologias semelhantes com 7 e 9 *routers*, tendo-se obtido os tempos de execução apresentados em forma de gráfico na Figura 17.

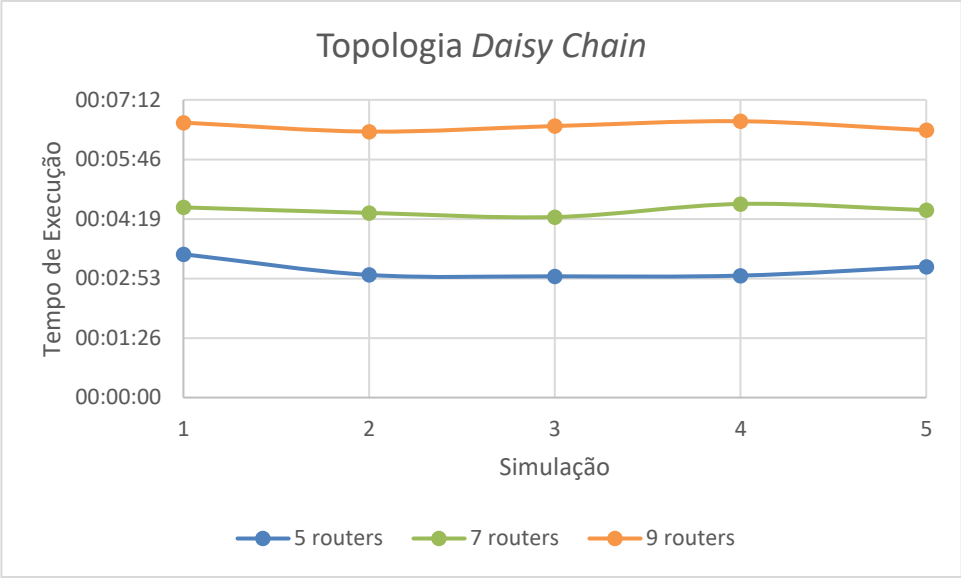


Figura 17 - Comparação do tempo de execução da topologia *Daisy Chain* com quantidades diferentes de *routers*

Analisando a Figura 17 consegue-se perceber que com o aumento de routers na topologia, o tempo de execução médio também será maior. Apesar deste ser o resultado expectável, é importante ter em conta a forma como este tempo aumenta, visto que tendo em conta o tempo médio em cada simulação tem-se:

- Entre a topologia com 5 e 7 *routers* existe um aumento de 1 minuto e 26 segundos;
- Entre a topologia com 7 e 9 *routers* existe um aumento de 2 minutos e 2 segundos.

Conforme os aumentos observados, tendo em conta que o número de *routers* aumentou em duas unidades em cada iteração, o aumento de tempo de execução não é linear. Uma vez que estes tempos de execução representam a duração da *pipeline* inteira incluindo a configuração de todos os *routers* presentes na topologia entregue pelo Containerlab de raiz, consideram-se estes valores aceitáveis. Esta é uma problemática que pode ser verifica em melhor detalhe na Secção 4.8.6.

4.8.2. Testes com Topologia em Anel

A topologia em Anel é semelhante à topologia anterior com a diferença que os extremos também se encontram ligados, considere-se a Figura 18.

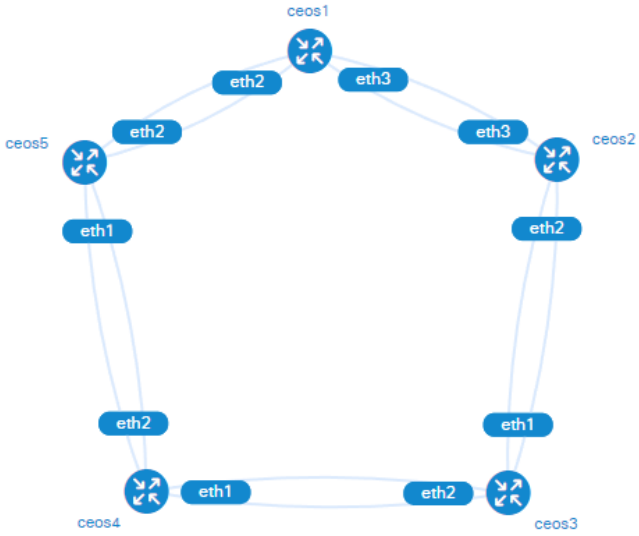


Figura 18 - Topologia em Anel com 5 routers

Tal como nos testes realizados na topologia anterior, também se configurou os mesmos protocolos de encaminhamento e serviços, sendo que neste caso todos os *routers* têm a mesma quantidade de interfaces. Na Figura 19 encontram-se os tempos de execução retirados a partir dos testes efetuados.

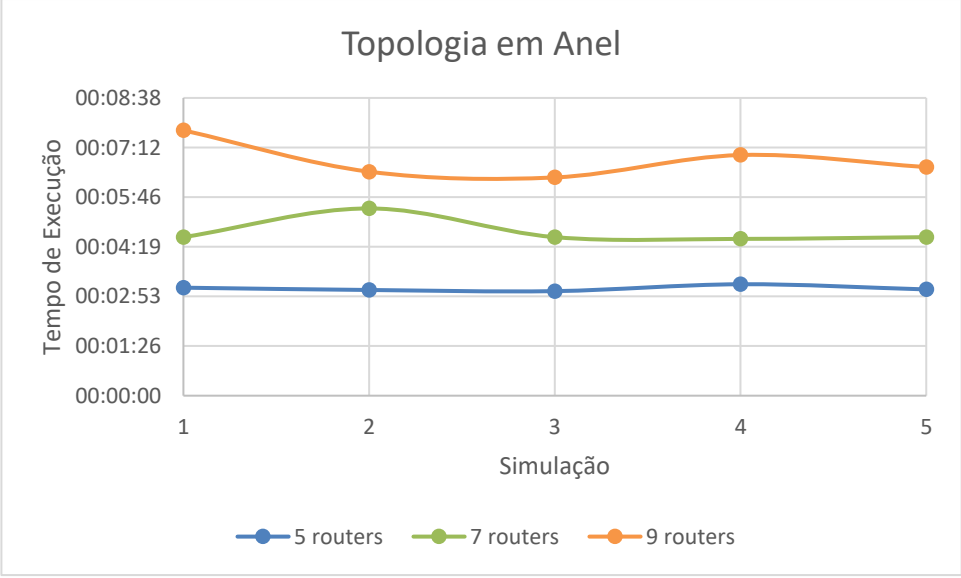


Figura 19 - Comparação do tempo de execução da topologia em Anel com quantidades diferentes de *routers*

Mais uma vez o aumento do tempo de execução médio verificou-se com o aumento de *routers*, tendo diferença uma temporal:

- Entre a topologia com 5 e 7 *routers* existe um aumento de 1 minuto e 38 segundos;

- Entre a topologia com 7 e 9 *routers* existe um aumento de 2 minutos e 5 segundos.

4.8.3. Testes com Topologia em Estrela

A topologia em Estrela começa a diferenciar-se das anteriores, visto que é composta por um *router* central que conecta todos restantes *routers*, considere-se a Figura 20.

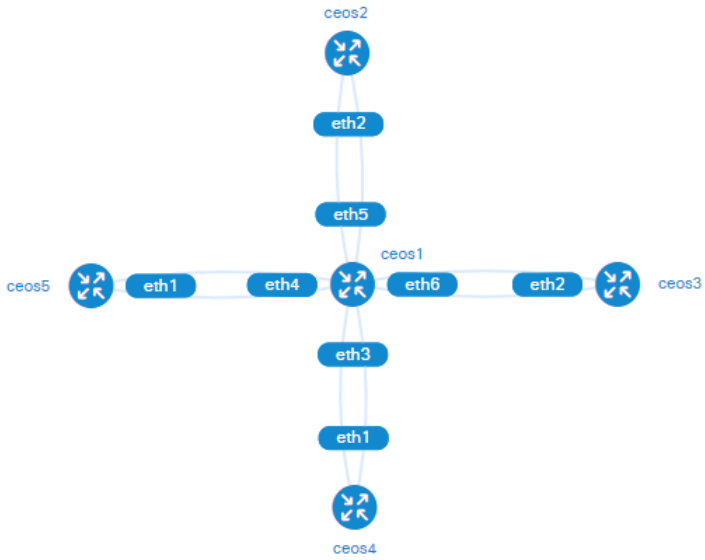


Figura 20 - Topologia em Estrela com 5 *routers*

Tendo em conta a topologia representada na Figura 20, o *router* "ceos1" tem configurado 8 interfaces, sendo quatro delas representam uma ligação direta entre os routers e as restantes quatro para testar o DHCP. Em termos de protocolos de encaminhamento, foram configurados de modo que todos consigam comunicar entre eles utilizando o *router* central.

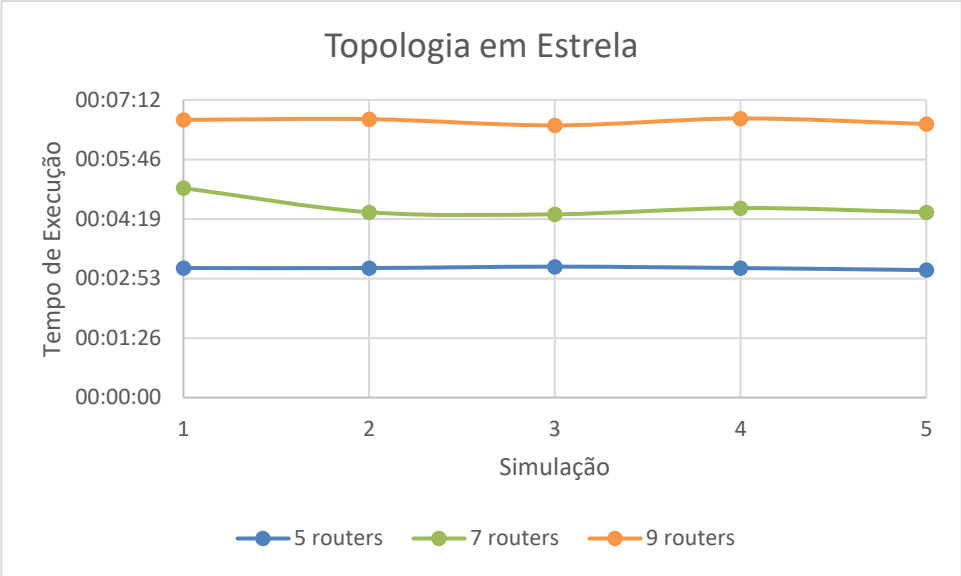


Figura 21 - Comparação do tempo de execução da topologia em Estrela com quantidades diferentes de *routers*

Tendo em conta os tempos de execução presentes na Figura 21, voltou-se a verificar o aumento do tempo de execução médio com o aumento de *routers*, tendo diferença uma temporal:

- Entre a topologia com 5 e 7 *routers* existe um aumento de 1 minuto e 29 segundos;
- Entre a topologia com 7 e 9 *routers* existe um aumento de 2 minutos e 4 segundos.

4.8.4. Testes com Topologia em Malha

A topologia em Malha é das mais complexas em termos de ligações, dado que todas os *routers* se encontram conectados entre si, considere-se a Figura 22.

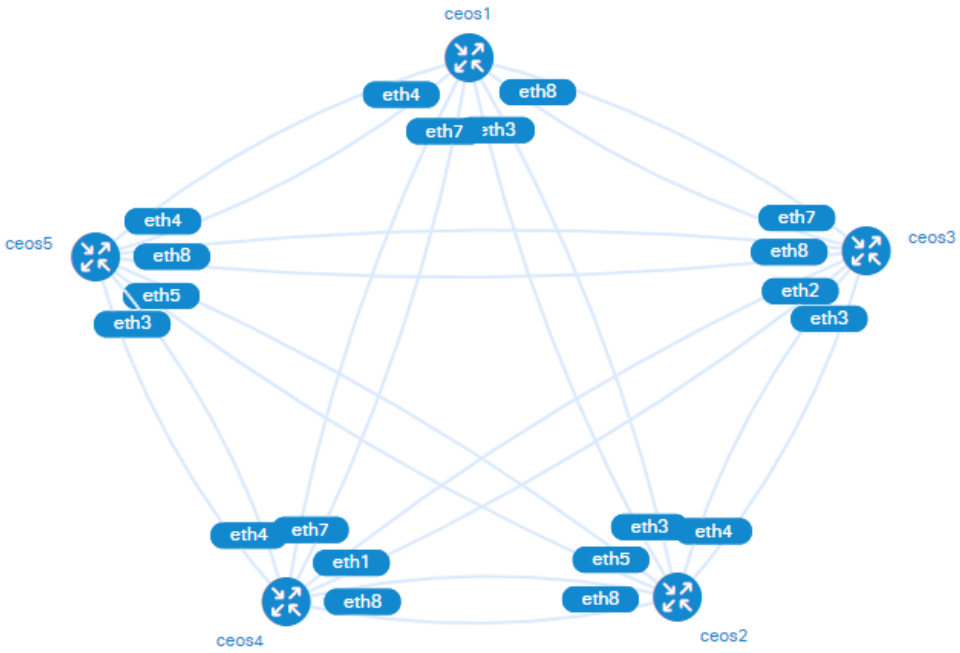


Figura 22 - Topologia em Malha com 5 *routers*

Para testar esta topologia foi necessário duplicar a quantidade de interfaces utilizadas, aumentando assim a complexidade e por consequência é expectável aumentar o tempo de execução.

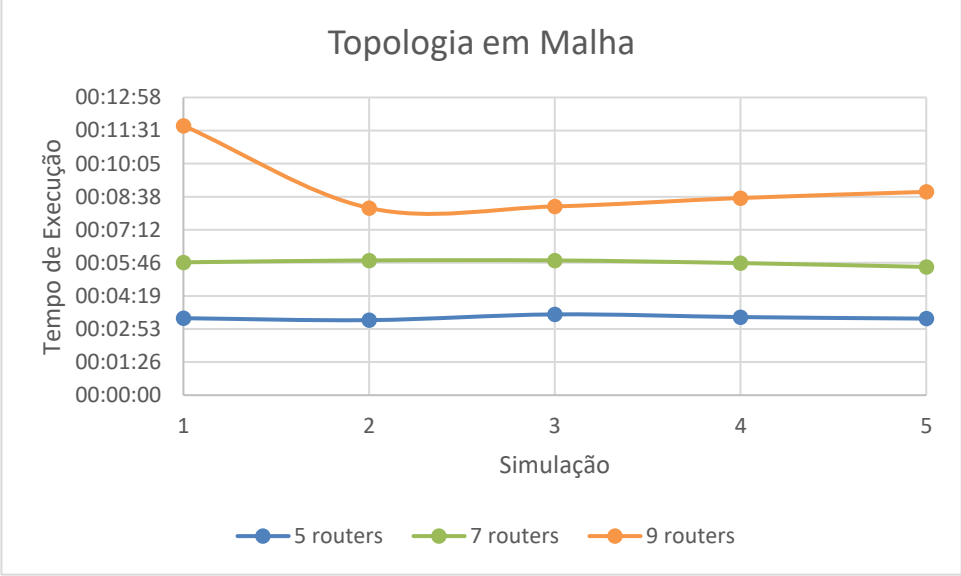


Figura 23 - Comparação do tempo de execução da topologia em Malha com quantidades diferentes de *routers*

A partir da Figura 23, conseguiu-se verificar o aumento do tempo de execução médio com o aumento de *routers*, tendo uma diferença temporal:

- Entre a topologia com 5 e 7 *routers* existe um aumento de 2 minutos e 24 segundos;
- Entre a topologia com 7 e 9 *routers* existe um aumento de 3 minutos e 20 segundos.

4.8.5. Comparação entre as diferentes topologias

A análise anterior permitiu entender como os tempos de execução variam dentro da mesma topologia, no entanto é interessante verificar como este tempo varia entre topologias com a mesma quantidade de *routers.* Para isto criaram-se gráficos que contêm a linha de tendência dos tempos de execução para cada simulação com diferentes quantidades de *routers*.

Considere-se a Figura 24, Figura 25 e Figura 26, que contêm o tempo médio de execução para as simulações dos quatro tipos de topologia de rede com 5, 7 e 9 *routers*, respetivamente.

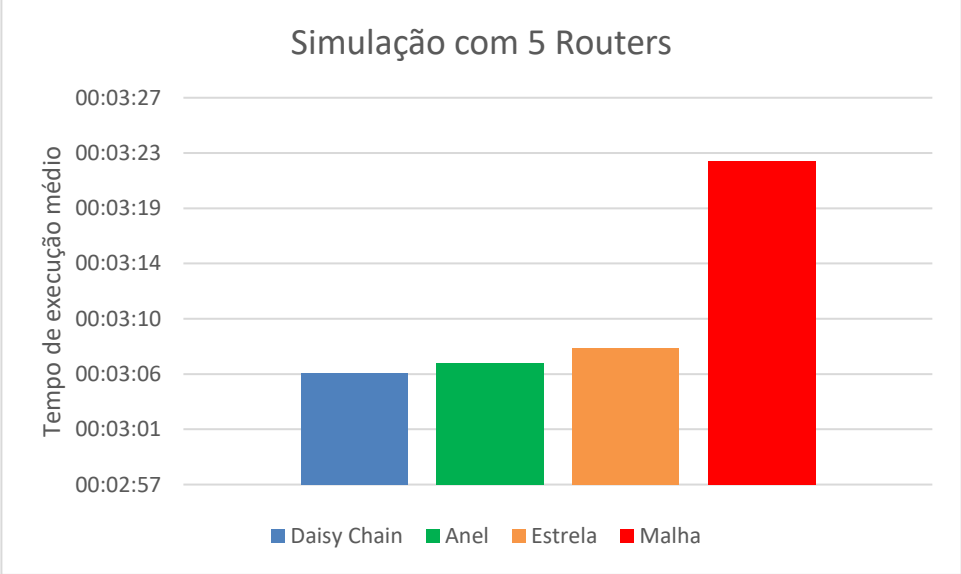


Figura 24 - Comparação do tempo de execução das topologias com 5 *routers*

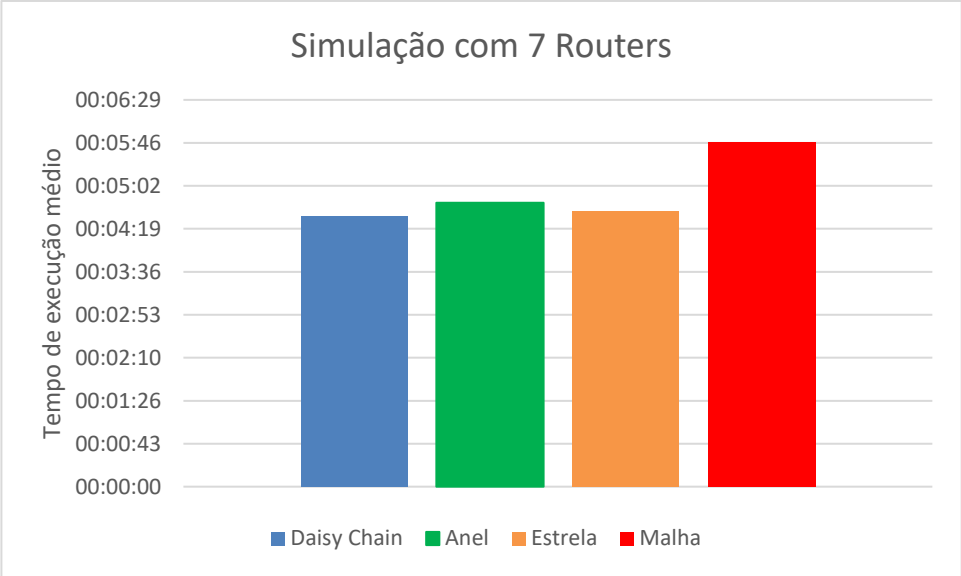


Figura 25 - Comparação do tempo de execução das topologias com 7 *routers*

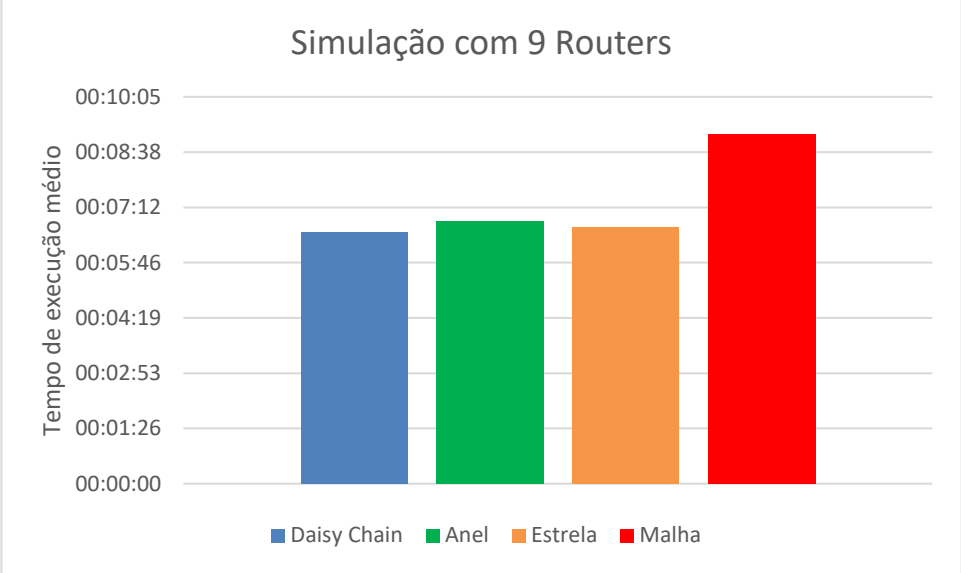


Figura 26 - Comparação do tempo de execução das topologias com 9 *routers*

A tendência mais evidente em todos estes gráficos é o facto de em todos eles a topologia em malha se encontrar com tempos de execução superiores em relação aos restantes. Isto deve-se à definição do ficheiro da topologia de rede, uma vez que para a topologia em malha são necessários mais *endpoints*. Isto é evidente depois de se analisar os tempos de execução da fase *Build* da *pipeline* através da interface *web* do Gitlab, sendo esta fase composta por apenas um *job* que é precisamente aquele que entrega a topologia. Os tempos de execução das restantes fases também aumentam, no entanto muito pouco em comparação com a topologia em malha.

Pode-se também observar que as topologias com a exceção da em malha têm todas tempos de execução semelhantes, o que se deve mais uma vez à quantidade de *endpoints* definidos, sendo que são praticamente a mesma quantidade.

4.8.6. Comparação entre jobs de diferentes topologias

Após se ter estudado como o tempo de execução da *pipeline* varia dentro da mesma topologia com diferentes quantidades de *routers* e entre topologias diferentes, decidiu-se avaliar qual a possível causa para os aumentos vistos. Para isso, foi necessário recolher os tempos de execução de alguns *jobs* em específico, pelo que se escolheu o *job* faz a entrega da topologia de rede, o que configura os *routers* e o que realiza os testes ao OSPF. Estes foram os *jobs* escolhidos, visto que são aqueles que apresentam uma maior variação e têm um impacto superior no tempo de execução da *pipeline*.

Tal como mencionado anteriormente, por vezes alguns jobs não são bem-sucedidos por serem executados imediatamente a seguir à entrega da topologia ou por serem mais complexos, pelo que nestes tempos médios de execução se teve em conta o tempo que demorou até o *job* ser bem-sucedido.

Considere-se a Figura 27, Figura 28 e Figura 29 que apresentam os tempos médios de execução do *jobs* responsáveis pela entrega da topologia, configuração dos *routers* e testar todo o OSPF.

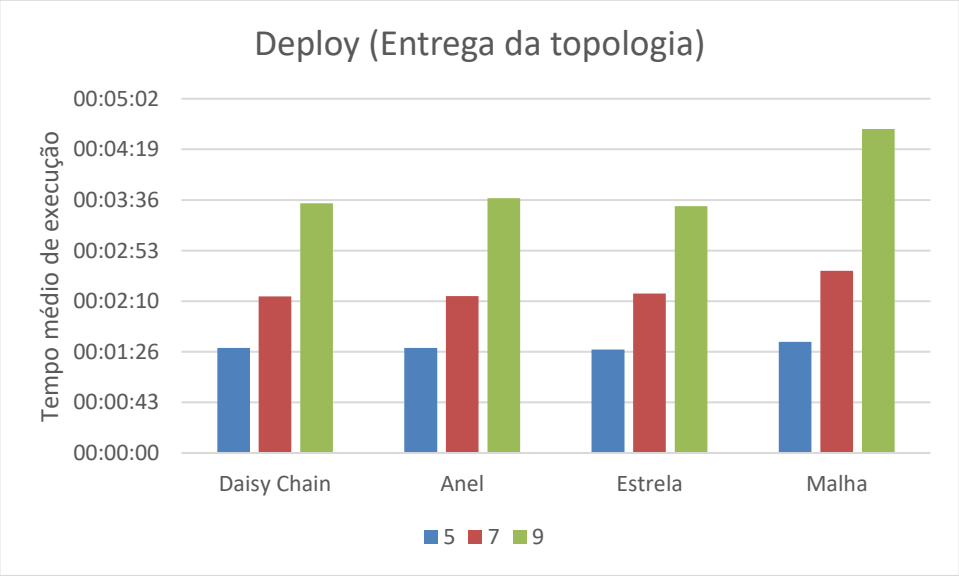


Figura 27 - Comparação do tempo de execução médio no *job deploy* entre diferentes topologias e quantidades de *routers*

Tendo em conta a Figura 24, Figura 25 e Figura 26, é possível observar que a maior parte do tempo de execução da *pipeline* se encontra na entrega da topologia, nestes casos pode-se considerar praticamente metade do tempo. Em relação ao tempo de execução entre topologias e quantidade de *routers*, praticamente todas se mantêm com tempos semelhantes, com exceção da topologia em Malha.

A topologia em questão apresenta um crescimento muito superior às restantes quando é necessário entregar 9 *routers*, mais uma vez devido ao grande aumento de *endpoints* necessários.

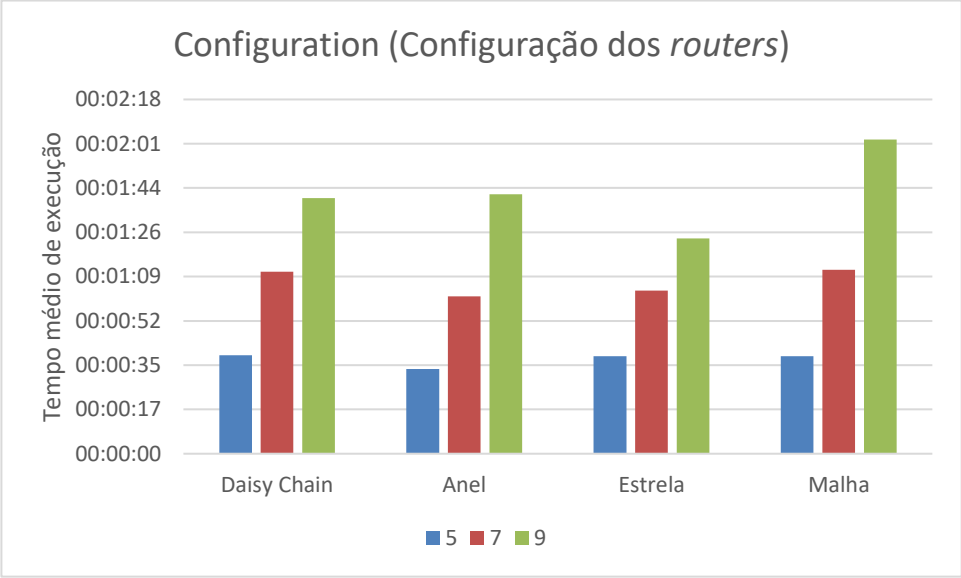


Figura 28 - Comparação do tempo de execução médio no *job configuration* entre diferentes topologias e quantidades de *routers*

No que toca ao job de configuração dos routers, mais uma vez observa-se uma variação semelhante ao caso anterior, com exceção da topologia em Estrela quando se tem 9 routers. Esta exceção deve-se ao *job* não ter falhado nenhuma vez na sua execução permitindo, diminuindo assim o tempo médio de execução. No entanto, se apenas se tiver em conta o tempo que demora o *job* em si e não o tempo que demora a ter sucesso, os tempos seriam semelhantes.

De notar que este *job* representa o tempo de execução médio para a configuração da topologia de testes ser feita desde raiz. Por exemplo, se se analisar as colunas da topologia *Daisy Chain*, consegue-se verificar que cada *router* demora cerca de 7.6, 10.1 e 11.1 segundos a ser configurado, respetivamente com a quantidade de *routers* presentes na topologia.

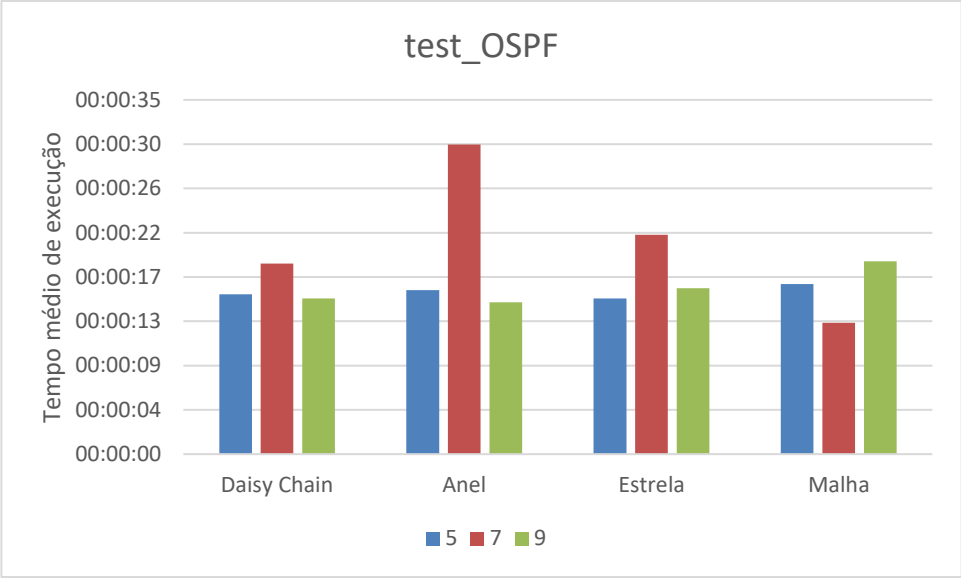


Figura 29 - Comparação do tempo de execução médio no *job test_OSPF* entre diferentes topologias e quantidades de *routers*

Ao analisar os tempos de execução médios do *job* responsável por testar todo o OSPF, consegue-se observar uma diferença nítida no crescimento em relação aos *jobs* anteriormente analisados. Enquanto em praticamente todas as topologias o tempo entre as que contêm 5 e 9 *routers* se encontra próximo, quando se tem 7 *routers* existe uma maior variação. Na topologia *Daisy Chain*, Anel e Estrela o tempo de execução é superior a todos os outros incluindo a topologia em Malha, tal ainda não tinha sido observado. Todavia, ao analisar a topologia em Malha com 7 *routers*, observa-se que este tempo é inferior aos restantes, incluindo com 9 *routers*.

Pode-se concluir que apesar da quantidade de *routers* ser a mesma, em topologias com quantidades semelhantes de *endpoints*, como *Daisy Chain*, Anel e Estrela, a variação do tempo de execução do *job* é idêntica. Com os testes realizados pode-se ainda identificar o ponto onde a topologia que se pretende entregar deixa de ser simples demais para ter tempos de execução inferiores mesmo tendo *jobs* sem sucesso e complexas ao ponto de existir tempo de todos os serviços e funcionalidades do *router* estarem funcionais quando se tenta executar testes mais complexos, não existindo tantos *jobs* sem sucesso.

5

CONCLUSÃO E TRABALHO FUTURO

No decorrer do desenvolvimento da dissertação apresentada foi possível estudar diversas metodologias e tecnologias candidatas a serem utilizadas numa arquitetura que pretende fazer a entrega de uma topologia de rede, configurá-la e realizar testes de modo a verificar o seu bom funcionamento.

A arquitetura e implementação propostas no Capítulo 3 representam um sistema com cinco componentes diferentes: Gitlab, Gitlab Runner, Containerlab, Ansible e Python.

Tem-se o Gitlab como o componente que serve como repositório para todos os ficheiros de configuração de topologia e equipamentos, testes a serem efetuados e ficheiro de especificação da *pipeline* pretendida. Para despoletar a *pipeline* o utilizador apenas necessita de realizar um *commit* para o repositório.

O Gitlab Runner permite a comunicação entre o Gitlab e a VM onde o *runner* se encontra instalado, coordenando assim as ações definidas no ficheiro da *pipeline*. O *runner* oferece garantias de disponibilidade consoante a quantidade de utilizadores, no entanto é sempre possível aumentar o número e especificando qual se pretende utilizar, deixando de existir este problema. Ao utilizá-lo de uma forma *self hosted* é ainda possível obter-se uma maior granularidade em relação ao que se pretende, desde o ambiente onde a *pipeline* é executada até às dependências necessárias aos diversos projetos.

No que toca ao Containerlab, este realiza a entrega da topologia pretendida através da definição de um ficheiro YAML. Esta componente tem como principal característica a sua simplicidade em entregar uma topologia de rede recorrendo a imagens Docker, oferecendo várias opções para quem não pretende implementar algo semelhante a esta dissertação e pretende apenas um laboratório para efeitos de estudo. O facto de utilizar Docker confere uma grande vantagem em termos de eficiência e gestão dos contentores utilizados, tendo em conta que esta é uma tecnologia cada vez mais utilizada.

Em relação ao Ansible, este tem o papel de automatizar tarefas como a configuração de equipamentos, realizar o *backup* dos *routers* em produção e gestão do inventário

dos mesmos. Esta gestão tem um papel importante na execução da *pipeline*, visto que é onde se encontra a informação acerca dos equipamentos entregues pelo Containerlab e que permite estabelecer conexões a cada um deles.

Dada a flexibilidade e facilidade de desenvolver código em Python, especialmente quando existem bibliotecas que auxiliam tanto na conexão aos *routers* como na filtragem da informação retornada pelos mesmos, decidiu-se recorrer a este para realizar testes com retornos menos previsíveis ou quando se pretende utilizar mais do que um comando por teste. Tal como foi referido, apesar do Ansible se encontrar num bom caminho para se tornar uma ótima ferramenta para redes, neste momento não é eficiente realizar testes muito complexos, pelo que se recorreu a Python para estes testes. Foram assim criados alguns *scripts* que se encarregam desses testes, existindo mais flexibilidade no código desenvolvido.

Considera-se que a arquitetura proposta é inovadora e tem espaço para crescer muito no futuro, entanto tem as suas limitações. Por um lado, a utilização de Docker confere uma escalabilidade e eficiência muito superior a *softwares* mais tradicionais como Eve-ng, no entanto não se compara em relação ao número de equipamentos disponibilizados para estes. Apesar de ser uma desvantagem atualmente, com o aumento da popularidade e contributo da comunidade ao Containerlab, irá ser cada vez uma solução mais viável. Esta desvantagem não se prende apenas no Containerlab, mas também nos vários vendedores que não disponibilizam imagens dos equipamentos devidamente otimizadas como fazem para VM.

Em relação ao trabalho futuro do projeto pretende-se continuar o trabalho já desenvolvido, com a motivação de expandir e reforçar a arquitetura proposta. Mais particularmente, pretende-se definir mais testes a serem efetuados e mais complexos relativos aos protocolos e serviços já suportados e também a novos. Pretende-se ainda expandir a quantidade de vendedores suportados, uma vez que de momento apenas foram desenvolvidos testes para equipamentos Arista. Uma outra implementação interessante que se pretende realizar é a adição de uma componente de monitorização como o Grafana, para que seja possível realizar testes a uma topologia de rede mais extensos e regulados com métricas mais especificas às necessidades do utilizador. Por fim, de modo a tornar esta arquitetura mais automatizada, pretende-se mover algumas informações para uma base de dados dedicada ao invés de ser necessário o utilizador especificar nos ficheiros de configuração dos equipamentos, permitindo testes mais pormenorizados e sem a necessidade do utilizador inserir tanta informação.

REFERÊNCIAS

[1] R. P. Goldberg, "Survey of virtual machine research" em Computer, vol 7, no. 6, pp 4-19, Junho 1974, doi: 10.1109/MC.1974.6323581.

[2] G. I. Radchenko, A. B. A. Alaasam, and A. N. Tchernykh, "Comparative Analysis of Virtualization Methods in Big Data Processing", superfri, vol. 6, no. 1, pp. 48–79, Apr. 2019. Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu and W. Zhou, "A Comparative Study of Containers and Virtual Machines in Big Data Environment," 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2018, pp. 178-185, doi: 10.1109/CLOUD.2018.00030.

[3] "What is virtualization?," Red Hat, https://www.redhat.com/en/topics/virtualization/what-is-virtualization (acedido em Jan. 25, 2023).

[4] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale," Proceedings of the 17th International Middleware Conference, 2016. doi:10.1145/2988336.2988337

[5] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," Linux J., vol. 2014, no. 239, 2014.

[6] C. Boettiger, "An introduction to docker for reproducible research," ACM SIGOPS Operating Systems Review, vol. 49, no. 1, 2015. doi:10.1145/2723872.2723882

[7] "Networking overview," Docker Documentation, https://docs.docker.com/network/ (acedido em Fev. 25, 2023).

[8] R. Dodin, "Powering the new NetOps ERA with Containerlab," Nokia, https://www.nokia.com/blog/powering-the-new-netops-era-with-containerlab/ (acedido em Fev. 25, 2023).

[9] R. Dodin, "Containerlab," containerlab, https://containerlab.dev/ (acedido em Fev. 25, 2023).

[10] U. Dzerkals, "EVE-NG Professional Cookbook", https://www.eve-ng.net/wp-content/uploads/2023/03/EVE-CE-BOOK-5.3-2023.pdf (acedido em Fev.25, 2023).

[11] "Ansible documentation," Ansible Documentation, https://docs.ansible.com/ (acedido em Mar. 10, 2023).

[12] K. OKASHA, Network Automation Cookbook: Proven and Actionable Recipes to Automate and Manage Network Devices Using Ansible. PACKT Publishing Limited, 2020.

[13] M. Oswalt, Network Programmability and Automation: Skills for the next-Generation Network Engineer. Beijing: O'Reilly, 2023.

[14] "Jinja," Jinja2, https://jinja.palletsprojects.com/en/3.1.x/ (acedido em Fev. 25, 2023).

[15] P. Choudhury, K. Crowston, L. Dahlander, M. S. Minervini, and S. Raghuram, "Gitlab: Work Where You Want, when you want," Journal of Organization Design, vol. 9, no. 1, Nov. 2020. doi:10.1186/s41469-020-00087-8

[16] J. M. Hethey, Gitlab Repository Management: Delve into Managing Your Projects with GitLab, While Tailoring It to Fit Your Environment. Birmingham: Packt Publ., 2013.

[17] "Gitlab Runner," GitLab, https://docs.gitlab.com/runner/ (acedido em Mar. 12, 2023).

[18] Moutsatsos, Ioannis & Hossain, Imtiaz & Agarinis, Claudia & Harbinski, Fred & Abraham, Yann & Dobler, Luc & Zhang, Xian & Wilson, Christopher & Jenkins, Jeremy & Holway, Nicholas & Tallarico, John & Parker, Christian. Jenkins-CI, an Open- Source Continuous Integration System, as a Scientific Data and Image-Processing Platform. Journal of Biomolecular Screening, 2016. doi: 10.1177/1087057116679993.

[19] Jenkins user documentation, https://www.jenkins.io/doc/ (acedido em Mar. 15, 2023).

[20] "Pyats & Genie," Cisco DevNet, https://developer.cisco.com/docs/pyats/ (acedido em Mar. 15, 2023).

ANEXOS

Anexo A – Guia de Preparação do Ambiente

Toda a *pipeline* foi desenvolvida utilizando uma VM com o SO Ubuntu 22.04.2 LTS, 12GB de memória RAM, 100GB de armazenamento e 3 *processor cores*.

O repositório encontra-se em https://github.com/JoaoCaracois/NetworkAsCode.

Com a VM instalada e configurada, começa-se por instalar o Docker e Containerlab respetivamente:

- sudo apt install docker.io
- echo "deb [trusted=yes] https://apt.fury.io/netdevops/ /" | \

  sudo tee -a /etc/apt/sources.list.d/netdevops.list

  sudo apt update && sudo apt install containerlab

Como apenas se desenvolveu testes para *routers* Arista, é somente necessário obter a imagem para o mesmo. A imagem utilizada no desenvolvimento desta dissertação foi "cEOS64-lab-4.28.0F.tar.xz". Tendo-se obtido a imagem é necessário importá-la para poder ser utilizada pelo Containerlab que por sua vez utiliza o Docker, podendo-se efetuar o comando:

- sudo docker import cEOS64-lab-4.28.0F.tar.xz ceos:4.28.0F

Com estas duas ferramentas instaladas, segue-se a instalação do Ansible sendo recomendado a mesma ser feita através dos comandos abaixo descritos e não através do *packet manager* pip. Assim, tem-se os seguintes comandos:

- sudo apt install software-properties-common
- sudo add-apt-repository --yes --update ppa:ansible/ansible
- sudo apt install Ansible

Em relação ao Python, este vem instalado por predefinição na VM, no entanto é

necessário instalar a biblioteca Netmiko e ipaddress.

Assim, para finalizar a instalação de todo o ambiente apenas é necessário criar conta no Gitlab, criar um projeto no mesmo para guardar todos os ficheiros, realizar o comando "git clone https://github.com/JoaoCaracois/NetworkAsCode.git" para ter acesso a todos os ficheiros desenvolvidos e relevantes à execução da *pipeline* e instalar o Gitlab *Runner*.

A instalação do *Runner* foi efetuada através de um *binary file* utilizando o comando:

- sudo curl -L --output /usr/local/bin/gitlab-runner "https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64"

De seguida é necessário fornecer permissões para este ser executado:

- sudo chmod +x /usr/local/bin/gitlab-runner

Com as permissões atribuídas, criou-se um utilizador para o Gitlab CI:

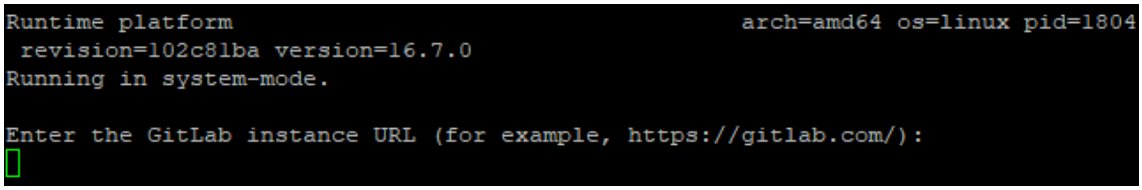
- sudo useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash

Para concluir a instalação *Runner* basta utilizar dois comandos para instalá-lo e executá-lo como um serviço:

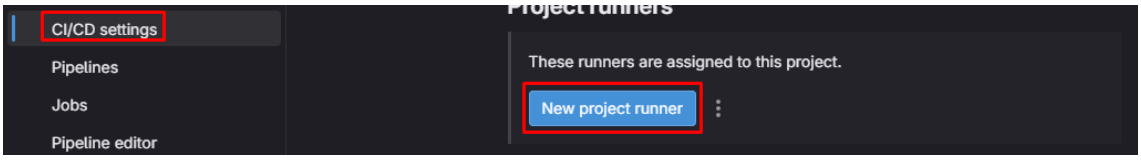
- sudo gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner
- sudo gitlab-runner start

Neste ponto, o Runner encontra-se instalado falta apenas registá-lo no Gitlab. De modo a iniciar este processo executou-se o comando "sudo Gitlab-runner register" e escolheram-se as seguintes opções:

- Basta introduzir o exemplo fornecido pelo próprio comando "https://gitlab.com/" na etapa abaixo demonstrada;



- De seguida é requisitado um *token* de registo. Este é obtido navegando no Gitlab até ao menu CI/CD *Settings* e selecionar a secção *Runners*. Neste menu existe a opção de criar um *project Runner*, sendo que esta opção pode variar caso este seja o primeiro *Runner* do projeto;



- Ao selecionar esta opção pode-se configurar diversos parâmetros do *Runner,* neste caso apenas se indicou o SO da VM onde este vai ser instalado, *tags* para diferenciar este *Runner* e uma descrição (sendo este campo opcional);

- Vai ser então fornecido o *token* que necessita ser colocado na VM, sendo este o próximo passo do lado da VM;

- O próximo passo prende-se na escolha do tipo de *executor* a ser utilizado. Optou-se por utilizar um *executor Shell*, dado que se pretende executar comandos diretamente na VM e este encontra-se na mesma VM onde os testes vão ser executados.

Os componentes do sistema encontram-se devidamente instalados e configurados, caso se pretenda testar a *pipeline* é necessário alterar alguns ficheiros. Se se pretender executar o exemplo presente no repositório tem de se alterar:

- Na diretoria *roles/backup_routers* alterar o destino para os ficheiros de *backup;*

- No ficheiro que define os *jobs* da *pipeline* denominado ".gitlab-ci.yml" alterar o campo *tag* dependendo do escolhido quando se registou o *Runner;*

- Alterar o inventário em formato INI especificando o nome dos *routers* que se encontram em produção. É necessário ter em conta que os nomes dentro do grupo *production* devem ser iguais ao do grupo Arista, retirando o prefixo acrescentado pelo Containerlab. Se um *router* em produção é denominado "ceos1", o *router* entregue pelo Containerlab será "clab-nome_do_lab-ceos1", sendo que "nome_do_lab" é o nome definido no ficheiro de topologia de rede.

Alterando estes parâmetros basta realizar um *commit* para o repositório para despoletar a *pipeline*. Ao se navegar para o menu *Pipelines* no Gitlab é possível acompanhar-se a execução da mesma e verificar os *logs* dos *jobs* executados.

Anexo B – Ficheiro que Define a Topologia Daisy Chain com 5 Routers

```
name: chain

topology:
  nodes:
    ceos1:
      kind: ceos
      image: ceos:4.28.0F
    ceos2:
      kind: ceos
      image: ceos:4.28.0F
    ceos3:
      kind: ceos
      image: ceos:4.28.0F
    ceos4:
      kind: ceos
      image: ceos:4.28.0F
    ceos5:
      kind: ceos
      image: ceos:4.28.0F

  links:
    - endpoints: ["ceos1:eth1", "ceos2:eth1"]
    - endpoints: ["ceos2:eth2", "ceos3:eth1"]
    - endpoints: ["ceos3:eth2", "ceos4:eth1"]
    - endpoints: ["ceos4:eth2", "ceos5:eth1"]
    - endpoints: ["ceos1:eth2", "ceos2:eth3"]
    - endpoints: ["ceos2:eth4", "ceos3:eth3"]
    - endpoints: ["ceos3:eth4", "ceos4:eth3"]
    - endpoints: ["ceos4:eth4", "ceos5:eth2"]
```

Anexo C – Ficheiro de Configuração para um Router Arista na Topologia Daisy Chain com 5 Routers

```
credentials:
  - username: admin1
    password: admin1
interfaces:
  - name: Ethernet1
    ipv4_addr: 172.16.12.1/24
  - name: Ethernet2
    ipv4_addr: 192.168.12.1/24
  - name: Loopback0
    ipv4_addr: 172.16.0.1/32
ospf:
  processID: 1
  router_id: 172.16.0.1
  no_passive_interfaces:
    - Ethernet1
  networks:
    - ip: 172.16.12.0/24
      area: 12.12.12.12
    - ip: 172.16.0.1/32
      area: 0.0.0.0
bgp:
  AS: 65001
  neighbors:
    - ipv4_addr: 172.16.12.2
      AS: 65002
  advertises:
    - network: 172.16.0.1
      mask: 255.255.255.255
    - network: 172.16.12.0
      mask: 255.255.255.0
dhcp:
  subnets:
    - interface: Ethernet2
      network: 192.168.12.0/24
      gateway: 192.168.12.1
      dns_server:
        - 8.8.8.8
        - 8.8.4.4
      start: 192.168.12.2
      end: 192.168.12.10
      lease: 1 days 0 hours 0 minutes
ntp_servers:
  - 172.16.0.1
  - 172.16.1.1
```

Anexo D – Template Jinja2 para Configurar um Router Arista

```
{% for credential in credentials %}
username {{ credential.username }} secret {{ credential.password }}
{% endfor %}
{% for interface in interfaces %}
{% if 'Loopback' in interface.name %}
interface {{ interface.name }}
ip address {{ interface.ipv4_addr }}
{% else %}
interface {{ interface.name }}
ip address {{ interface.ipv4_addr }}
no switchport
{% endif %}
{% endfor %}
{% if ospf is defined %}
ip routing
router ospf {{ ospf.processID }}
router-id {{ ospf.router_id }}
{% for passInt in ospf.no_passive_interfaces %}
no passive-interface {{ passInt }}
{% endfor %}
{% for net in ospf.networks %}
network {{ net.ip }} area {{ net.area }}
{% endfor %}
{% endif %}
{% if bgp is defined %}
ip routing
router bgp {{bgp.AS}}
{% for bgp_neigh in bgp.neighbors %}
neighbor {{bgp_neigh.ipv4_addr}} remote-as {{bgp_neigh.AS}}
{% endfor %}
{% for advertise in bgp.advertises %}
network {{advertise.network}} mask {{advertise.mask}}
{% endfor %}
{% endif %}
{% if dhcp is defined %}
{% for dhcp_subnet in dhcp.subnets %}
interface {{ dhcp_subnet.interface }}
dhcp server ipv4
dhcp server
subnet {{ dhcp_subnet.network }}
default-gateway {{ dhcp_subnet.gateway }}
range {{ dhcp_subnet.start }} {{ dhcp_subnet.end }}
lease time {{ dhcp_subnet.lease }}
dns server {% for server in dhcp_subnet.dns_server %}{{ server }} {%
endfor %}
```

```
{% endfor %}
{% endif %}

{% if ntp_servers is defined %}
{% for ntp_server in ntp_servers %}
ntp server {{ ntp_server }}
{% endfor %}
{% endif %}
```

Anexo E – Playbook Ansible para Realizar Testes ao OSPF

```
---
- name: Test OSPF Configuration
  hosts: arista
  connection: local
  gather_facts: false
  pre_tasks:
    - include_vars: "../../config_vars/{{inventory_hostname}}.yml"

  tasks:

  #----------------------ARISTA----------------------#

    - name: execute install script
      ansible.builtin.script:
        executable: python3
        cmd: test_ospf_neighbors.py {{inventory_hostname}}
{{ansible_user}} {{ansible_password}} {{ansible_network_os}}
      register: output
      when: ospf is defined

    - name: Display result
      debug:
        var: output

    - name: Check OSPF
      assert:
        that:
          - '"True" in output.stdout_lines'
        fail_msg: "OSPF failed to be tested. Please check
{{inventory_hostname}} configuration"
        success_msg: '{{inventory_hostname}} configuration seems
correct.'
      when: ansible_network_os == 'arista.eos.eos' and ntp is defined
```

Anexo F – Ficheiro que Especifica os Jobs da Pipeline

```
stages:
    - build
    - configuration
    - test
    - backup
    - cleanup
    - production

deploy_lab:
    stage: build
    tags:
        - server
    script:
        - cd ..
        - export ANSIBLE_HOST_KEY_CHECKING=False
        - sudo containerlab deploy -t NetworkAsCode/lab_chain.yml --
reconfigure
        # - sudo containerlab graph -t NetworkAsCode/lab_chain.yml
        # - ^C
        - cd ..

configure_lab:
    stage: configuration
    tags:
        - server
    script:
        - ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook
roles/config/main.yml -i "inventory"
    retry: 2

test_ip_address:
    stage: test
    tags:
        - server
    script:
        - ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook
roles/check_ip_address/main.yml -i "inventory"
    retry: 2

test_ospf_neighbors:
    stage: test
    tags:
        - server
    script:
        - ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook
roles/check_ospf_neighbors/main.yml -i "inventory" -vvv
```

```
    retry: 2

test_bgp:
    stage: test
    tags:
        - server
    script:
        - ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook
roles/check_bgp/main.yml -i "inventory"
    retry: 2

test_ping:
    stage: test
    tags:
        - server
    script:
        - ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook
roles/check_ping/main.yml -i "inventory"
    retry: 2

test_dhcp:
    stage: test
    tags:
        - server
    script:
        - ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook
roles/check_dhcp/main.yml -i "inventory"
    retry: 2

test_ntp:
    stage: test
    tags:
        - server
    script:
        - ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook
roles/check_ntp/main.yml -i "inventory"
    retry: 2

backup_configs:
    stage: backup
    tags:
        - server
    script:
        - ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook
roles/backup_routers/main.yml -i "inventory"
    retry: 2

destroy_lab:
    stage: cleanup
```

```
    tags:
        - server
    script:
        - sudo containerlab destroy -t lab_chain.yml
    retry: 2

config_production:
    stage: production
    tags:
        - server
    script:
        - ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook
roles/config_production/main.yml -i "inventory"
    retry: 2
```