UNIVERSIDADE FEDERAL DE SÃO CARLOS
Centro de Ciências Exatas e de Tecnologia
Departamento de Computação

Automação de Implantação de ambiente de computação em nuvem

GABRIEL DE JESUS DANTAS

SÂO CARLOS-SP
2024

GABRIEL DE JESUS DANTAS

AUTOMAÇÃO DE IMPLANTAÇÃO DE AMBIENTE DE COMPUTAÇÃO EM NUVEM

Trabalho de conclusão de curso apresentado ao Departamento de Computação como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação em pela Universidade Federal de São Carlos.

Orientador: Prof. Dr. Hélio Crestana Guardia.

São Carlos-SP - 2024

Dedico este trabalho aos colegas e professores que fizeram parte da pesquisa, em especial o professor H.G por abraçar o desafio, à minha família por sempre ser um pilar em minha vida. Agradecimento especial às pessoas que em meu momento mais difícil, me motivaram a continuar aqui, G.S.C., M.C.A.F., N.K.S.V.

Agradecimento especial ao LuizaLabs, que apoiou a realização deste trabalho de pesquisa através do projeto de extensão “Pesquisa e desenvolvimento em tecnologias para data centers utilizando virtualização”, em parceria com DC/UFSCar.

“And so after waiting aimlessly one day like any other I decided to succeed.
I decided not to wait for opportunities but to go out and find them.
I decided to see every problem as challenges with infinite possibilities.
I decided to see the desert as an opportunity to find an oasis.
I decided to see every night as a mystery to solve.
I decided to see each new day as an opportunity to be happy.
That day I discovered that my only hindrances were my own weaknesses, and that they were just pointing at the best way to do better.
That day I stopped fearing to lose and I began to fear not trying at all.
I discovered that it’s not about me and perhaps never was, and it ceased to matter who won or lost.
Today I rejoice in finding I’m better than yesterday.
I learned the hard part – it’s not climbing to the top of the hill but to keep on climbing.
I learned that my greatest triumph is to have the right to call someone “friend.”
I discovered that love is more than just a state of illusions; it is a way of life.
This day I stop being a reflection of my past failures and light up the candle of the present.
I learned that this light is of no use if it’s not to illuminate the path of others.
I decided to change so many things.
That day I learned that our dreams exist only to be realized.
Since that day I no longer sleep to rest. I sleep solely to dream.”

(Walt Disney)

RESUMO

A computação em nuvem é uma vertente recorrentemente utilizada na atualidade, porém possui grande complexidade em processo de implantação, estabilização e manutenção de ambientes (*clusters*) disponíveis para usuários. Para ambientes baseados em OpenStack, tal processo decorre de um fluxo constante de interação humana, variando em média de 2 a 4 horas de processo em casos médios, sem interferências e *debugs*, podendo levar dias nos piores casos, contabilizando processamento feito pela máquina nesse meio. Visando facilitar o fluxo de implantação, considera-se a automação desse processo, para reduzir a interação humana ao máximo, encaminhando como tarefa automática para a máquina a instalação, configuração e comunicação de *softwares* utilizados no ambiente de computação em nuvem. Utilizando Interfaces de Programação de Aplicação (*APIs*) e bibliotecas de *software* de gerenciamento de servidores *bare metal* e Máquinas Virtuais, e de mecanismo de orquestração, o objetivo é transformar o processo de conexão e configuração completa do ambiente em um *script* automatizado. Como resultado, este trabalho gerou estratégias para construção de uma *API* para implantação de um ambiente de computação em nuvem, estratégias para adicionar novos computadores em um ambiente de nuvem estável em funcionamento, além de um *script* de validação que, ao ser executado, configura um infraestrutura OpenStack completa em uma máquina controladora que inicialmente possui somente um sistema operacional e algum tipo de acesso à Internet.

Palavras-chave: Computação em Nuvem; Automação; OpenStack; MAAS; Juju; Ansible; IaC; PoC

ABSTRACT

Cloud computing is a frequently used technology today, but one which is very complex in the process of implementing, stabilizing and maintaining environments (clusters) available to users. For OpenStack-based environments, this process results from a constant flow of human interaction, ranging on average from 2 to 4 hours of processing in average cases, without interference and debugging, and can take days in the worst cases, taking into account the processing done by the machine in this environment. In order to facilitate the implementation flow of a cloud infrastructure, the automation of this process is considered, aiming to reduce human interaction as much as possible, sending the installation, configuration and communication of software used in the cloud computing environment as an automatic task to the machine. Using Application Programming Interfaces (APIs) and software libraries for managing bare metal servers and Virtual Machines, and an orchestration engine, the goal is to transform the process of connecting and completely configuring the cloud environment into an automated script. As a result, this work generated strategies for building an API for deploying a cloud computing environment, strategies for adding new computers to a stable, functioning cloud environment, and a validation script that, when executed, configures a complete OpenStack infrastructure on a controller machine that initially has only an operating system and some type of Internet access.

LISTAGENS

TABELAS

SUMÁRIO

1. INTRODUÇÃO

Previamente a tratar-se do que será investigado neste documento, se faz necessário considerar uma pergunta: o que se entende por computação em nuvem?

Atualmente tornou-se comum o uso do termo computação em nuvem, tal qual a utilização do conceito no cotidiano, enquanto utilizando um dispositivo móvel, tirando fotos, utilizando-se de dispositivos eletrônicos com acesso à Internet, entre outros.

Computação em nuvem é uma plataforma responsável por provisionar serviços e recursos para aplicações, de forma que o usuário final tenha somente uma pendência ao tentar executar algum tipo de aplicação, sendo essa o acesso à Internet. Tal plataforma consiste em um ambiente físico, onde ativamente dados e processos serão computados, que podemos chamar de *data center*, como definido pela amazon (AWS [s.d.]): “uma locação física que armazena máquinas de computação e seus equipamentos de hardware relacionados”. Ademais, em cima dos ambientes físicos são construídos ambientes virtuais, nos quais serão executadas aplicações em nuvem.

Quanto às funções e ao funcionamento de uma plataforma de computação em nuvem, o ambiente trabalha com o conceito de micro serviços, ou seja, uma arquitetura de aplicações independentes que se comunicam via *Application Programming Interfaces (APIs)* leves como proposto pela Red Hat (REDHAT 2023). Cada serviço é responsável por uma funcionalidade distinta, como, por exemplo, serviços para armazenamento em nível de bloco, autenticação de acesso, provisionamento de volumes, criação de máquinas virtuais, entre outros.

Como definido pela IBM (IBM 2024), pode-se considerar que o ambiente de computação em nuvem é o responsável por dinamicamente provisionar, configurar, alterar configurações e desprovisionar servidores conforme a necessidade do administrador. Tais serviços necessitam ter um fácil gerenciamento e, graças à arquitetura de micro serviços, é possível realizar qualquer manobra em um dado serviço sem que o servidor inteiro venha a falhar.

Quanto à capacidade de uma nuvem, ainda a IBM identifica quatro pontos fundamentais:

1. Hospedar uma variedade de *workloads.*
2. Permitir que *workloads* sejam implantados e possuam escalabilidade a partir do provisionamento rápido de máquinas virtuais ou físicas.
3. Suporte para redundância, auto recuperação e modelos de programação de alta escalabilidade que permitem *workloads* de se recuperarem de falhas.

4. Recurso de monitoramento usado em tempo real, permitindo rebalanceamento de alocações quando necessário

Dos pontos listados acima, 2 a 4 decorrem do gerenciamento ativo do ambiente, trabalhando com o provisionamento e desprovisionamento de serviços e aplicações, tal qual reconfigurações, para tratar de melhor forma falhas ou mudanças na arquitetura da plataforma que está sendo utilizada.

O processo para se iniciar um ambiente de computação em nuvem consiste, de forma sucinta, em preparar a conexão de um conjunto de máquinas físicas, de maneira que se comuniquem e tenham acesso à Internet. Isso pode ser feito com o auxílio de um *network switch*, componente de *hardware* que será responsável por interligar máquinas em uma rede comum local e que também possibilita que a partir de uma máquina única com acesso à Internet, outras pertencentes à rede local possam ter acesso à rede mundial de computadores. Após o preparo físico, deve-se realizar a conexão virtual, via aplicações, para então inicializar a instalação dos serviços para enfim possuir o ambiente de computação em nuvem funcional.

Todo o caminho que se traça a partir de um recebimento de máquinas até a estabilização de um ambiente de computação em nuvem é extremamente sequencial e custoso em relação ao tempo tomado pelo processo, podendo, em alguns casos, levar ao menos um dia.

Pensando na continuidade e sequencialidade do processos, neste trabalho foram desenvolvidos uma proposta e um estudo da automação da configuração de um ambiente baseado em OpenStack (OPENSTACK [s.d.]), visando transformar as tarefas que devem ser realizadas para configuração em uma aplicação que simplifica a implantação. Por tratar-se de um programa, a configuração física do ambiente, tais como conexões e cabeamento, é um pré-requisito para o funcionamento.

Além da apresentação da proposta, este documento visa descrever como tal automação contribui em pesquisas e desenvolvimento, visto que a mesma foi testada em ambiente de pesquisa do projeto de extensão “Pesquisa e desenvolvimento em tecnologias para data centers utilizando virtualização” feito pela parceria entre DC/UFSCar e LuizaLabs.

1.1 CENÁRIO

Como ambiente para desenvolvimento, foi realizada uma instalação da plataforma OpenStack utilizando três máquinas físicas. Essas máquinas são: Dell PowerEdge R910, com 64 núcleos e 128 GB RAM, utilizada como controlador, e duas com 80 núcleos e 1024 GB, sendo utilizadas como nós de computação. O nó controlador é o responsável por gerenciar componentes, além de servir como intermediário nas comunicações dos nós de

processamento, conectados apenas a um segmento de rede interno e uma única saída via *switch* para acesso à Internet via redirecionamento de requisições externas.

Qualquer acesso a máquinas virtuais, máquinas de *compute*, serviços, e ao próprio OpenStack *Command-line Interface* (CLI), ou à interface web de gerenciamento (*dashboard)*, pode ser feito realizando conexão ao nó controlador via SSH. Após a estabilização do ambiente, é possível limitar o acesso à rede local, por meio de uma rede virtual privada, utilizando WireGuard (WIREGUARD [s.d.]).

Para implantação da plataforma OpenStack, considerando a configuração física das máquinas já estabilizada, foram utilizadas duas ferramentas na execução trivial, sendo essas Canonical MAAS (*Metal as a Service*) (MAAS [s.d.]) e Canonical Juju (JUJU [s.d.]). O fato de oferecerem uma interface de acesso via comandos *shell* (CLI - *Command-line Interface*) e terem uma *API* própria motivou a criação de um mecanismo de automação da instalação de uma nuvem OpenStack.

A ferramenta MAAS, segundo sua própria documentação, pode ser definida como uma plataforma de nuvem para gerenciamento de servidores *bare metal* e máquinas virtuais, que foi desenhada para atender as necessidades dos operadores e administradores de um *data center.* Nesta proposta, essa ferramenta é responsável pelo gerenciamento das máquinas virtuais utilizadas para o serviço, pelo provisionamento de um servidor DHCP (*Dynamic Host Configuration Protocol)* (DHCP 2023) para identificação das máquinas da rede local e serviço de boot PXE (INTEL 1999), pela criação de *network bridges* necessárias para utilização de *Open Virtual Network*, e de *LXD* (LXD [s.d.]), pela criação de máquinas virtuais, pela configuração e instalação de sistemas operacionais nas máquinas e pelo gerenciamento de energia das máquinas físicas.

Juju é uma ferramenta *open source* que atua como um motor de orquestração para operadores de *software*, permitindo *deployment,* integração e gerenciamento do ciclo de vida de aplicações em qualquer escala ou infraestrutura. Para o funcionamento de Juju, são utilizados operadores denominados *“charms”,* um para cada serviço necessário à instalação do OpenStack. Em suma, essa ferramenta é utilizada para instalar diferentes serviços em diferentes máquinas e integrá-los para comunicarem-se entre si, concomitantemente monitorando e podendo exibir os estados das máquinas.

O termo orquestração tem sentido para seu funcionamento, dado que Juju gerencia e monitora esses serviços. No modelo desenvolvido neste trabalho, tal ferramenta é responsável no processo pelo gerenciamento de diferentes nuvens, gerenciando credenciais, simplificando o acesso às máquinas de serviços, realizando a implantação (*deployment*) dos serviços desejados para o OpenStack e o monitoramento do ambiente.

1.2 TRABALHOS RELACIONADOS

Para compreensão e seleção das ferramentas de automação e seus benefícios, foi benéfico o uso do trabalho intitulado "Estudo de ferramentas de automação em nuvem" (MATTIOLI, 2023), integrante do mesmo projeto de extensão em que se situa esta pesquisa. Nesse estudo, compreende-se a complexidade de uma infraestrutura para computação em nuvem e justifica-se a utilização das tecnologias escolhidas para implantação do OpenStack, utilizando MAAS e Juju, dada a facilidade para suas integrações.

Para entendimento do processo de implantação a ser realizado e seu planejamento, tal qual sua divisão em passos, foi estudado também o trabalho "Projeto e Implementação de um Ambiente de Capacitação em Computação em Nuvem com Virtualização" (OLIVEIRA 2024), que serve também como motivação para esse estudo, em razão de elucidar como o processo se dá de maneira sequencial, e como pode ser automatizado, com exemplificação do *bundle* criado para utilização da ferramenta Juju.

Ainda relacionado à proposta deste trabalho, foi analisado o software CloudLab (CLOUDLAB [s.d.]), uma infraestrutura com propósitos educacionais, utilizada para pesquisa e estudo de computação em nuvem. Sua documentação (CLOUDLAB [s.d.]) o classifica como uma infraestrutura científica e flexível para pesquisas benéficas ao futuro da computação em nuvem, possuindo pesquisas ativas em seu ambiente para sistemas distribuídos, bancos de dados, segurança cibernética, networking, entre outros campos. Deste modo, serve como ambiente de pesquisa para diversos artigos já publicados. A aplicação CloudLab é uma aplicação construída e baseada em Emulab (EMULAB [s.d.]), que, por si só ,já é uma plataforma de testes de internet (*network testbed*), também com propósitos educacionais, que disponibiliza uma variedade de ambientes em nós prontos para testes. Assim, a razão para que CloudLab se baseie nessa plataforma é sua disponibilidade de provisionamento de recursos em níveis físicos, permitindo também o gerenciamento de topologias de redes.

O software CloudLab utiliza o conceito de perfis para poder realizar a implantação de um ambiente em nuvem. Um perfil é uma estruturação do ambiente de nuvem que vai ser gerado na execução da aplicação, contendo as informações de todos os softwares que vão constituir a nuvem em questão, a descrição de hardware necessário junto à sua topologia de internet. Nesse sentido, CloudLab provê versatilidade, disponibilizando inclusive um editor de topologia via GUI (*graphic user interface*), permitindo gerenciamento completo de conexões da arquitetura, tal qual tipo de hardware, tipo de nó e imagem a ser implantada, possibilitando também a execução de *scripts* pós-instalação nas máquinas. Todos os recursos descritos acima são expressos pela aplicação no formato de RSpecs (RSPECS

[s.d.]) do projeto GENI (*Global Environment for Network Innovations*) (GENI [s.d.]), utilizando uma linguagem para descrever recursos, requisições dos recursos e reservas dos mesmos.

Uma vez criada a conta de um usuário, instituído um projeto e definido um perfil para o ambiente de nuvem, no caso de aplicações OpenStack, é necessário realizar a escolha de parâmetros como número de nós de computação, versão do OpenStack, tipo do hardware, tecnologia de rede na camada 2, entre outros, que permitem gerir como será instanciado esse ambiente de computação em nuvem. Deve-se também escolher um dos *clusters* disponíveis para implantar o ambiente entre aqueles pertencentes ao Emulab.

Finalizado o processo de configuração do CloudLab, basta clicar em um botão para iniciar-se o provisionamento, que será abstraído para o usuário enquanto as conexões, criações de máquinas, instalações de sistemas operacionais e configurações de serviços são feitas. Ao final, é gerado para o usuário um comando *ssh* com identificação do nome de usuário e do *host* para acesso à máquina criada no ambiente de nuvem.

1.3 OBJETIVO

Este trabalho tem como foco a construção e execução de uma forma de automação, para que todo o processo de configuração e implantação de um ambiente local de computação em nuvem seja automático, via código, CLI, ou *scripts*. Com isso, busca-se simplificar processos de estabilização de diferentes configurações de infraestrutura OpenStack, possivelmente de forma expansível também para outras plataformas de computação em nuvem.

A automação prevista neste trabalho visa a diminuir a necessidade de interação humana neste processo. Para isso, procurou-se concentrar as atividades necessárias em um único código, que operasse sobre um conjunto de máquinas fisicamente conectadas em rede.

Pensando nesse cenário, considera-se também explorar as possibilidades para que a automação possa ser usada não somente para inicialização de um ambiente, mas também para acoplar um novo conjunto de máquinas (como um *rack*) ao cluster principal de um ambiente ativo, ou em produção.

Diferentemente do que ocorre na plataforma CloudLab, a automação proposta neste trabalho visa permitir o gerenciamento simplificado da parte de configuração de infraestrutura da nuvem, não se limitando somente à seleção de serviços. Enquanto com CloudLab o software configurado é executado sobre um cluster em uma localidade com disponibilidade, a proposta da automação deste estudo é realizar a implantação a partir de dados de endereços IP das máquinas que serão utilizadas no servidor *bare metal*, permitindo versatilidade quanto a aplicações.

Por tratar-se de uma automação e de como oferecer suas funcionalidades na forma de chamadas de uma *API*, considera-se que sua utilização ocorrerá diretamente sobre os equipamentos da infraestrutura física disponível.

A busca por simplicidade de configuração e operação é uma característica comum deste projeto e do projeto CloudLab.

1.4 PROPOSTA

Como mencionado, o processo de implantação de uma infraestrutura de computação em nuvem, apesar de longo, é composto por atividades realizadas de forma sequencial, o que contribui para sua automatização na forma de conjuntos de instruções. De forma sucinta, pode-se descrever tal processo nos seguinte passos:

1. Instalação do MAAS
2. Instalação do PostgreSQL
3. Configuração de usuário e banco de dados do MAAS
4. Configuração e ativação de um servidor DHCP (o que pode ser feito com o MAAS)
5. Comissionamento de nós de computação
6. Criação de *network bridges* para nós de computação
7. Criação de *network bridge* para nó controlador
8. Instalação do LXD e inicialização do *Virtual Machine Host*
9. Criação de máquina virtual para instalação do Juju
10. Instalação do Juju
11. Configuração de modelo de nuvem desejado
12. Implantação dos serviços da plataforma OpenStack, de forma monitorada
13. Apresentação dos parâmetros de acesso para o usuário administrador, incluindo endereço de rede e senha para o acesso ao OpenStack

Em sua versão atual, os mecanismos desenvolvidos realizam todo esse processo, incluindo instalações. Para tanto, usa-se comandos da linguagem Python (PYTHON [s.d.]) e da biblioteca *subprocess* (PYTHON [s.d.]), sendo necessário que cada passo consiga garantir estar completo antes de iniciar o próximo.

Algumas questões devem ser consideradas, contudo, primeiro quanto à garantia de conclusão de cada passo. Por exemplo, no passo 4, após a ativação do servidor DHCP, os nós de computação que estão na rede local não são reconhecidos imediatamente pelo MAAS, sendo que isso pode levar algum tempo. Deste modo, o próximo passo só pode ser

executado quando a condição de que a quantidade prevista de máquinas foi encontrada. A mesma necessidade de auto-verificação e monitoramento automático irá ocorrer para os passos 8, 9 e 12. Segundo, para não ser preciso usar o recurso *subprocess*, é necessário que todas as etapas de instalação sejam realizadas de outra maneira, como, por exemplo, usando a ferramenta Ansible. Terceiro, as execuções de comandos nas ferramentas MAAS e Juju programadas na versão inicial dos mecanismos de automatização desenvolvidos foram realizadas com utilização de chamadas subprocess com invocações de comandos das CLIs de ambas as ferramentas. Porém, visando melhor portabilidade e construção, é desejável realizar tais manobras por meio das bibliotecas LIBMAAS (PYTHON-LIBMAAS [s.d.]) e LIBJUJU (PYTHON-LIBJUJU [s.d.]) na linguagem Python, que serão tratadas posteriormente neste documento.

Em suma, a proposta apresentada neste trabalho consiste em transformar os passos da implantação de uma infraestrutura de computação em nuvem com OpenStack em código executável, o que ainda não ocorre de maneira completa em sua versão atual. As etapas e propostas para a conclusão das funcionalidades previstas são descritas neste trabalho.

Também é alvo deste estudo a criação de estratégias para automatizar a agregação de novas máquinas a um ambiente estável de computação em nuvem, seguindo o padrão de arquivos de configurações.

Assim, as principais etapas deste trabalho são:

- agregar funcionalidades que realizem todos os passos da configuração e implantação de uma plataforma de computação em nuvem;
- discutir mecanismos e estratégias para realizar todas as etapas configuradas de forma automatizada, invocando diretamente as funções das APIs das ferramentas selecionadas;
- consolidar estratégias para a configuração de uma nova plataforma de nuvem e também a adição de recursos novos a instalações já existentes, sem ter que interromper seus funcionamentos.

O restante deste trabalho está organizado da seguinte forma: no capítulo 2, são tratados aspectos conceituais e relacionados às diferentes ferramentas e plataformas de software utilizadas neste trabalho; o capítulo 3 detalha o desenvolvimento do trabalho; no capítulo 4, são apresentadas considerações sobre a ampliação e o aprimoramento dos mecanismos desenvolvidos; o capítulo 5 apresenta e discute resultados obtidos, sendo seguido pelas conclusões no capítulo 6.

2 FERRAMENTAS DE APOIO

As atividades de automatização da construção de uma plataforma de computação em nuvem considerada neste trabalho partem de uma conceitualização do processo, expressa na forma de passos e uma versão gerada a partir das etapas definidas, ambos apresentados em seções seguintes.

Quanto à geração de uma versão de partida, é realizado um processo simples de transformação da conceitualização em formas de passos, para código sequencial de execução, realizando tratamento automático de espera por estados do processo.

Um aspecto não ideal desta versão de partida é a utilização de chamadas ao mecanismo *subprocess* da linguagem python. Este mecanismo permite a realização de invocações de comandos em uma sessão de *shell* e, na versão inicial do mecanismo para implantação de nuvens OpenStack, era utilizado para realizar chamadas aos diferentes programas utilitários e ferramentas utilizados nas etapas do processo estabelecido.

Apesar de facilitador, esse mecanismo de invocações de linhas de comando de *shell* não torna o processo de implantação estável ou portátil, uma vez que toda instância sofre necessidade de alteração manual do comando a ser executado no novo processo. Além disso, pode ser preciso alterar parâmetros no fluxo principal dos comandos para se manter atualizado com versões atualizadas das ferramentas utilizadas.

Para solucionar as principais questões de estabilidade, portabilidade e manutenção da automação, é desejável substituir essas invocações de programas pela utilização de bibliotecas (*APIs*) e aplicações externas que possuam suporte às ferramentas Juju e MAAS utilizadas no processo de implantação sendo refinado.

Conforme definido pela plataforma de nuvem AWS (*Amazon Web Services*) (AWS [s.d.]), *APIs* (*Application Programming Interface*) são caminhos para comunicação entre componentes de diferentes aplicações, normalmente seguindo definições e protocolos regrados. A base das *APIs* são os denominados *endpoints*, que são pontos de acesso para a realização de uma requisição de um cliente a um servidor, gerando uma resposta apropriada.

2.1 LIBMAAS

python-libmaas trata-se de uma *API client* que permite acesso a *endpoints* de um servidor MAAS, com suporte à manutenção dos seguintes objetos e funcionalidades:

- *account*
- *boot-sources and boot-resources*

- *machines, devices, region controllers, rack controllers*
- *events*
- *configuration*
- *tags*
- *version*
- *zones*

Tal biblioteca permite interagir com servidores MAAS, operando nas versões 2.0 ou superior, de maneira síncrona ou assíncrona. Importante ressaltar, contudo, que essa aplicação ainda é considerada uma versão alfa, de domínio público, mas que ainda pode sofrer mudanças. Esta biblioteca oferece uma interface para a adição de novos objetos ao código fonte para que desenvolvedores possam manipular e criar operações e objetos próprios. Um aspecto negativo sobre sua utilização, contudo, é a precariedade de sua documentação, por falta de exemplos e descrições concretas da utilização de algumas chamadas de funções. Assim, seu uso requer um processo de pesquisa e leitura de código para compreensão do que se encontra disponível atualmente na biblioteca e o que deve-se ser implementado pelo desenvolvedor.

Felizmente para esta pesquisa, as rotas de *boot-sources*, *boot-resources*, *machines* e *configuration* satisfazem as necessidades para migrar o fluxo da configuração do ambiente no MAAS apresentado no desenvolvimento do código de automação deste trabalho.

Como exemplo, o trecho a seguir (Listagem 1) ilustra um código para executar login no MAAS para gerenciamento do ambiente. No primeiro bloco de código, vê-se esta funcionalidade sendo implementada com invocações de comandos *shell* usando o mecanismo *subprocesses* para acesso à CLI da aplicação. É gerado uma chave para a *API* do MAAS, e posteriormente é executado login no servidor, usando esta chave.

No segundo exemplo de código (Listagem 2), este mesmo acesso é feito usando chamadas da *API* da biblioteca libmaas.

Listagem 1: Código utilizando biblioteca subprocess para realizar *login* na aplicação MAAS.

```
username = config['maas']['username']
password = config['maas']['password']


# Get MAAS API access key and store in api_key
api_key = f'api-key-{username}-file'
run_cmd(f'maas apikey --username={username} > {api_key}')


# Getting api key value to login
api_key = bash_to_string(f'cat {api_key}')


# Do login using so it is possible to configure maas
run_cmd(f'maas login {username} {MAAS_SERVER} {api_key}')
```

Fonte: Código feito neste trabalho, visto na seção 3.1.

Listagem 2: Código para realizar *login* na aplicação MAAS, utilizando *libmaas*.

```
username = config['maas']['username']
password = config['maas']['password']


# Do login using so it is possible to configure maas

maas_client = await login(MAAS_SERVER, username, password)
```

Fonte: Documentação libmaas

O exemplo apresentado nas Listagens 1 e 2 explicita a diferença entre ambas estratégias para automação do processo. Primeiramente, é possível visualmente notar a simplicidade do código quando utilizando a biblioteca libmaas, que possui menos passos, evitando a criação de uma chave de *API* e armazenamento em ambiente local, e encapsulando em uma única função o processo de *login*, que retorna um *client* para requisições de outras funções na aplicação. Uma grande vantagem dessa proposta é a assincronicidade concebida à solução, já implementada na própria biblioteca, garantindo assim que nesse ponto do código tenha-se um objeto ou resposta, antes de continuar ao próximo passo.

2.2 LIBJUJU

Também para o uso das funcionalidades do orquestrador de software, Juju, utilizado na automação, há uma biblioteca e *API client*, chamada python-libjuju.

Diferentemente da solução vista em 2.1, essa *API client* possui uma documentação vasta sobre as respectivas classes e funções, permitindo encontrar com facilidade operações que realizem as operações sobre esta plataforma. Particularmente úteis no caso deste projeto, são as operações de gerenciamento de infraestrutura, após implantação do ambiente de computação em nuvem.

Para fins de elucidar a utilização da biblioteca, o trecho de código a seguir ilustra uma conexão a um modelo, que é a abstração onde encontra-se o ambiente de nuvem, com suas aplicações máquinas entre outros componentes.

Listagem 3: Criar modelo com libjuju.

```
# Create a Model instance. We need to connect our Model to a Juju api
# server before we can use it.
model = Model()

# Connect to the currently active Juju model
await model.connect()
```

Fonte: Documentação libjuju.

2.3 ANSIBLE

Ansible (ANSIBLE [s.d.]) é uma poderosa ferramenta de automação *open-source* e colaborativa, que provê soluções de automação para provisionamento, configuração, implantação, orquestração e gerenciamento de recursos, entre outros processos de sistemas computacionais. Além disso, suas funcionalidades podem ser ampliadas.

Esta ferramenta é vastamente documentada, incluindo inúmeros casos de uso relevantes. Para sua operação, cria-se o que é denominado *playbook*, basicamente definido na forma de um arquivo no formato YAML, que serve de guia do que será implantado pela ferramenta. Cada *playbook* contém especificações de tarefas que podem ser listadas para construir um fluxo de automação.

Um aspecto muito relevante na operação desta ferramenta na automatização de processos é a possibilidade de armazenamento de respostas das chamadas invocadas em variáveis dentro dos *playbooks*. Dessa maneira, é possível utilizar o resultado de uma chamada da própria CLI do MAAS, por exemplo, para decidir, ou utilizar um gatilho para o próximo passo de um fluxo de operações. Isto também pode ser feito utilizando *loops*, outro

recurso da ferramenta, o qual permite execução de uma tarefa, ou múltiplas, uma quantidade desejada de vezes.

3 DESENVOLVIMENTO

Este capítulo descreve o desenvolvimento da estratégia e o uso de ferramentas para a automatização do processo de configuração e implantação de uma plataforma de computação em nuvem usando OpenStack, frutos deste trabalho de TCC.

Um conjunto de *scripts python*, contendo invocações de comandos via chamadas em sessão de *shell*, foi desenvolvido para realizar as automatizações desejadas e é descrito neste capítulo.

Considerações sobre a substituição da forma de acesso aos serviços externos, trocando a emissão de chamadas em linha de comando por invocações de serviços de APIs das ferramentas utilizadas são apresentadas no capítulo 4.

Visando o desenvolvimento atrelado à simplicidade de compreensão do processo, a estratégia de codificação adotada foi a modularização.

Inicialmente, em colaboração ao projeto de extensão “Pesquisa e desenvolvimento em tecnologias para data centers utilizando virtualização”, desenvolvido em parceria entre DC/UFSCar e LuizaLabs, foi realizado um processo de implantação da infraestrutura de computação em nuvem desejada, usando OpenStack, e anotado cada passo deste processo. Esta atividade deu origem a um documento que serve de guia de implantação desta plataforma. Este guia descreve explicitamente cada uma das atividades que o usuário deve realizar para construir um ambiente de computação em nuvem usando a infraestrutura do projeto, composta por computadores interligados em rede via switch. Este guia é composto por 35 passos, listados a seguir:

1. Acessar o computador controlador onde deseja-se implantar a nuvem.
2. Instalar MAAS.
3. Instalar PostgreSQL.
4. Criar usuário para MAAS.
5. Criar banco de dados para MAAS.
6. Inicializar banco de dados.
7. Criar um usuário administrador para MAAS.
8. Verificar se MAAS server está ativo.
9. Acessar a interface do sistema MAAS via web-browser.
10. Confirmar a configuração do controller.
11. Realizar *login* com usuário administrador criado.
12. Configurar *netplan*.
13. Ativar o servidor DHCP.
14. Adicionar a identificação dos nós / máquinas no MAAS.

15. Realizar comissionamento do nó controlador.
16. Criar *network bridge* para a aplicação NOVA.
17. Criar *network bridge* para o LXD VIRTUAL MACHINE HOST (VM HOST).
18. Instalar lxd e criar VM HOST.
19. Criar uma máquina virtual para instalação do Juju.
20. Instalar o Juju.
21. Resolver configurações de pacotes de Internet para instalação do Juju.
22. Realizar *bootstrap* do Juju.
23. Criar o modelo do OpenStack.
24. Realizar *deploy* do *bundle* do openstack.
25. Destravar (*unseal*) servidor *vault*.
26. Esperar unidades chegarem em estado *active*.
27. Acessar a web UI do OpenStack.

Minúcias em relação às aplicações de cada passo, como são feitas e razões para serem feitas serão exploradas na seção de modularização.

3.1 MODULARIZAÇÃO

O código modularizado é explicado nesta seção e pode ser encontrado na seção Apêndice B deste trabalho.

Em relação às bibliotecas (*APIs*) utilizadas, além das previamente citadas nesse trabalho, ***hashlib*** (PYTHON [s.d.]) foi utilizada para realizar sequências de verificações das mensagens (*message digest*) e para garantir segurança das requisições recebidas; ***lzma*** (PYTHON [s.d.]) foi utilizada para descompactar arquivos; ***tarfile*** (PYTHON [s.d.]) foi usada para leitura e escrita de arquivos do tipo tar; ***urllib3*** (PYPY [s.d.]) forneceu um cliente HTTP para realizar requisições; ***tempfile*** (PYTHON [s.d.]) serviu para criar arquivos e diretórios temporários; ***pyYaml*** (PYYAML [s.d.]) serviu para leitura e manipulação de arquivos yaml; ***json*** (PYTHON [s.d.]) foi responsável por converter mensagens em formato *json* para dicionários em python; ***psycopg2*** (PYPY [s.d.]) foi utilizada para realizar ações no banco de dados *postgresql* (POSTGRESQL [s.d.]); a biblioteca ***sys*** (PYTHON [s.d.]) serviu para tratar argumentos de entradas na execução do código; e a biblioteca ***os*** (PYTHON [s.d.])forneceu a função *getuid*, utilizada para verificar se o código foi executado com permissão *root*, a função *listdir*, para listar os diretórios em um dado path, e a função *path*, para manipular e acessar nomes de caminhos, visando criar ou ler arquivos.

Algumas variáveis globais foram utilizadas para facilitar argumentos das funções: ***ip*** refere-se ao endereço ip da máquina que está sendo utilizada para execução do código; ***config*** é um dicionário que carrega toda informação do arquivo de configuração yaml (explicado na seção seguinte) e, por fim, ***art*** foi utilizado em períodos de testes, para resposta visual de erros.

Para facilitar a criação do código e dependência de saída de execuções de comandos, foi criada a função ***bash_to_string***, que simplesmente executa um comando *bash* e retorna sua saída como *string*.

Duas funções foram criadas para facilitar a execução do fluxo do código, ***run_cmd***, que recebe como argumento um comando *bash* e realiza a execução do mesmo em um terminal, e a função ***apt_install***, importante para instalar qualquer pacote no formato *apt*.

Em razão de o código ter sido construído por meio de execuções práticas em um ambiente, visando comprovar a eficácia da execução e também visando concluir a instalação do ambiente desejado, foram construídas as funções ***clean_postgres*** e ***delete_maas***, sendo que a primeira limpa qualquer dado criado no banco de dados *postgresql* utilizado pela aplicação MAAS, e a segunda realiza exclusão de todas as chaves e dependências utilizadas pela aplicação, deixando o ambiente totalmente limpo como antes da execução do código. Ambas as funções são importantes para realizar uma nova tentativa de execução da implantação, ou para simplesmente desinstalar um ambiente atualmente instalado.

A função ***main*** guia o fluxo do que será executado, de acordo com os passos vistos na seção 3 deste trabalho. Para isso, basicamente foi criado um módulo para criação e configuração do MAAS e outro para o Juju, sendo necessários dois argumentos para execução do código: primeiramente, a função que será executada (a função *main* executa o fluxo completo), seguindo-se o nome do arquivo de configuração *yaml* que deve ser carregado.

A função **maas**, como previamente citada, refere-se à criação e configuração completa da aplicação **MAAS** e foi separada em módulos para simular etapas do processo de implantação de um ambiente em nuvem, visto na seção 3 deste trabalho. Para todas as etapas são medidos o tempo decorrido, utilizando a biblioteca *time*, e posteriormente é impresso o resultado. Deste modo, é possível acompanhar qual passo foi concluído ou falhou.

A função ***installation*** foi criada para fazer a instalação das aplicações necessárias para a aplicação MAAS, nesse caso a própria aplicação via *snap* (SNAPCRAFT [s.d.]) e, em seguida a instalação do banco de dados, definindo uma senha para o usuário *postgres*, para ser utilizado em uma conexão. Finalmente, essa função chama outra, responsável por instalar dependências que serão utilizadas no processo, a ferramenta *ipmitool* (IPMITOOL

[s.d.]), que utiliza a tecnologia *Intelligent Platform Management Interface (IPMI)* para gerenciar sistemas computacionais remotamente, e que será responsável por ligar remotamente as máquinas de *compute*.

Para inicialização e utilização da aplicação MAAS, é necessário criar uma base de dados na aplicação de banco de dados, onde serão armazenadas as informações relativas às máquinas que serão conectadas e configuradas na aplicação. Para isso, é criada a função ***database_configuration***.

Uma das requisições para se realizar o gerenciamento das máquinas físicas e virtuais da aplicação MAAS é a utilização de uma conta de usuário ***administrador***, que precisa ser criada. Para isso, foi criada a função ***maas_create_admin***.

Após a criação e inicialização do *controller* do MAAS, é necessário esperar que seu servidor ***http*** esteja em funcionamento e, para isso, é foi criada a função ***await_http_server***, que simplesmente verifica no *status* da aplicação o campo relativo ao servidor, esperando que possua a resposta "*RUNNING*".

Na primeira utilização do MAAS, algumas ações necessárias incluem gerar um par de chaves para utilização da *API* e salvá-las, já que elas serão necessárias para fazer *login* na CLI. Depois, é preciso definir o endereço de um servidor DNS *(Domain Name System)* e, em seguida, uma imagem para *download* deve ser selecionada e importada. Essa imagem será utilizada para configurar novas máquinas.

O próximo passo é adicionar uma chave ssh para o MAAS, da máquina onde ele está sendo hospedado, permitindo assim que esse servidor que está sendo executado possa se conectar às máquinas que forem adicionadas a ele.

Em seguida, é necessário definir uma faixa de endereços IPs para criação de um servidor DHCP, que será responsável por gerenciar a atribuição de endereços para as máquinas que serão adicionadas na aplicação. Todos esses passos são feitos na função ***first_time_config***, utilizando os serviços das funções ***ip_range***, para calcular e gerar numericamente a faixa de IPs, e *maas_ssh_key*, para criar uma chave ssh e adicionar no MAAS. Após a criação do servidor DHCP, é necessário definir o endereço IP de um *gateway*, permitindo assim que as máquinas dentro desse servidor, possam realizar acesso à Internet, necessário para que façam *downloads*.

A partir desse ponto, antes de prosseguir, é necessário garantir que a imagem do sistema operacional, selecionada e importada, tenha concluído seu *download*. Para isso, a função ***await_image_import*** é responsável por verificar os recursos atuais de *boot*, que estão sendo importados. Se não houver pendências, isso significa que a imagem está disponível.

A versão do MAAS utilizada neste trabalho foi a 3.4/*edge*, na qual encontrou-se uma peculiaridade ao arquivo responsável pelo boot PXE de máquinas adicionadas ao ambiente.

Esse arquivo é o *lpxelinux.0*, utilizado pela aplicação para realizar a comunicação e adição de novas máquinas via PXE.

Ocorre, contudo, que na máquina controladora, onde está instalada a aplicação MAAS, a versão do arquivo citado acima tem alguma falha e, visando resolver esse problema, optou-se por substituí-lo por uma versão atualizada, encontrada em *The Linux Kernel Archives* (KERNEL [s.d.]). Assim, o objetivo da função ***fetch_pxelinux*** desenvolvida é simplesmente realizar o *download* do arquivo da imagem dos arquivos de kernel linux, desempacotar os arquivos e mover o *lpxelinux.0* para dentro do diretório da aplicação MAAS. Essa função foi resultado de colaboração com o aluno Miguel Antonio de Oliveira, durante a pesquisa no projeto de extensão.

O próximo passo das configurações é adicionar as máquinas ao MAAS e, para isso, basta ligar as máquinas, para que elas possam fazer o carregamento do arquivo de *boot* via PXE. A função ***ipmi_reset*** é responsável por esse processo e, para isso, ela primeiramente aguarda as máquinas serem reconhecidas dentro da rede de *subnets* do servidor DHCP, para então utilizar-se dos IPs configurados para as máquinas no servidor como parâmetro de conexão para a ferramenta *ipmitool*, que irá ligar as máquinas, para que sejam adicionadas agora como objetos do tipo máquina ao MAAS.

Uma vez que as máquinas forem ligadas, e adicionadas à listagem de máquinas do MAAS, é necessário fazer o comissionamento das mesmas, que basicamente consiste em instalar o sistema operacional no nó de forma efêmera. Para isso, é utilizada a função ***commission_node*** que, primeiramente, espera até que as máquinas estejam conectadas e listadas e, posteriormente, define a imagem de sistema operacional padrão para comissionamento do MAAS. Feito isso, define um nome comum para as máquinas e as comissiona, permitindo acesso via protocolo ssh. Para isso, duas funções auxiliares foram desenvolvidas: ***get_nodes_id***, que lista as máquinas na aplicação e retorna seus IDs, importante para gerenciamento via CLI, e ***await_node_status***, que irá esperar até que o nó selecionado chegue no estado selecionado. Essa função recebe dois argumentos: ***st***, que é um número referente ao estado que deve ser esperado, e ***ig***, que é um número referente ao estado em que se um nó se encontra. Ao fim dessa função, será novamente necessário que a execução espere os estados dos nós, até que estejam prontos (*ready*), assim finalizados o comissionamento.

A partir do momento em que as máquinas estão prontas para uso, é necessário criar *network bridges*, para que seja possível haver comunicação entre os nós que serão usados para processamento (*compute*). Visando isso, a função ***create_compute_bridge***, gerencia as máquinas para que as *bridges* sejam criadas e associadas à mesma *subnet*.

Para que sejam criadas as máquinas virtuais, faz-se necessário criar uma *network bridge* na própria máquina controladora. Para isso, é necessário editar o arquivo de

configuração *netplan* da máquina (considerando um sistema Linux Ubuntu) e reiniciar a aplicação MAAS, verificando previamente se isso já não está feito, evitando sobrescrever o arquivo de configuração. Isto é feito com a função ***create_lxd_bridge***. Devido à reinicialização, uma nova chamada a *await_http_server* é necessária, para garantir que o servidor da aplicação MAAS está funcional para continuar a configuração.

Para criar as máquinas virtuais foi utilizada a solução LXD, que deve ser instalada e inicializada previamente. A criação e instalação desta solução, responsável pelo gerenciamento das máquinas virtuais, é feita na função ***lxd_init***.

Para finalmente permitir a criação de máquinas virtuais, que serão necessárias para implantação de aplicações dos serviços do *OpenStack*, é necessário criar um *VM Host*. Para isso, basta realizar uma chamada à CLI do MAAS, utilizando o endereço ip do *gateway* do servidor DHCP configurado e nomeando o projeto, o que é feito na função ***create_vm_host***.

A última pendência da aplicação MAAS é a criação de uma máquina virtual, o que é feito pela função ***create_vm***, que está guiada para criação da máquina onde será instalado o Juju.

A partir desse ponto no código, serão executados os passos relativos à aplicação Juju no processo de implantação do ambiente. Esses passos estão agrupados em um módulo *juju*, responsável por criações e configurações do processo que passem pelo orquestrador de *software*.

Para criação da aplicação, é necessária a existência de um diretório para guardar as informações, ou configurações realizadas, de maneira que o usuário que executou esse processo tenha acesso. Uma vez garantida a existência desse diretório, basta instalar o Juju via *snap*.

Uma vez instalada a aplicação Juju, é necessário realizar algumas configurações para garantir o funcionamento da mesma. Inicialmente, deve-se adicionar uma *cloud*, para que seja possível construir uma nuvem e modelos; em seguida, é necessário adicionar a autenticação da *API* do MAAS ao Juju, utilizando a chave de *API*, para que o orquestrador possa se utilizar da outra aplicação para gerenciamento das máquinas. Finalmente, deve-se criar as credenciais de uso do Juju, como codificado na função ***juju_config***.

Para permitir que o Juju consiga instalar as dependências de cada serviço em sua respectiva máquina, nesse ambiente é necessário resolver configurações de pacotes de Internet. Para isso, será realizado o processo de liberar o encaminhamento de pacotes da máquina de *ipv4* e *ipv6*, aplicadas as alterações do arquivo *sysctl.conf* e definida uma regra do tipo *nat* para que os IPs internos utilizem o endereço do *gateway* para terem acesso à Internet. Esse processo é tratado na função *resolve_internet_issues*.

O próximo passo é realizar o *bootstrap* do Juju, que é sua implantação associando-se ao controlador do MAAS, realizando a instalação das dependências e liberando o funcionamento completo do orquestrador.

Finalmente, será criado o *modelo Juju*, onde serão definidos os serviços do OpenStack que serão implantados; isso é feito na função ***create_model***.

O próximo passo consiste em realizar o *deployment* do *bundle* dos serviços do OpenStack, que está disponível na seção 3.4 deste trabalho.

Para finalizar a implantação, é importante realizar o *unseal* da aplicação *vault* (VAULT [s.d.]), que é o gerenciador de acessos que será utilizado pelo OpenStack. Os certificados de máquinas, utilizados para comunicação entre serviços, são abstraídos em estratégias de implantação utilizando Juju, sendo que a aplicação *vault* controla toda a comunicação e validação, por conta própria. Quanto à responsabilidade do desenvolvedor do ambiente, cabe realizar somente o *unseal* e, para isso, foi desenvolvida a função ***unseal_vault***, que irá acessar a máquina que contém essa aplicação, gerar um *token* e chave para desbloqueá-la e armazenar esses em arquivos na máquina *controller*, dado que uma vez geradas as credenciais, essas não poderão ser recuperadas de outra forma.

3.2 ARQUIVO DE CONFIGURAÇÃO YAML

A automação dos procedimentos discutidos até aqui requer o ajuste de diferentes parâmetros e informações. Neste projeto, isso é feito via um arquivo de configuração no formato ***yaml***. O arquivo utilizado para configuração segue uma lógica intuitiva, sendo que cada aplicação possui sua identação, incluindo *MAAS, postgres, lxd* e *juju*. Adicionalmente, *ssh* requer a definição de uma chave ssh a ser criada. Também é preciso definir a máquina virtual (*vm_host*) com as configurações para definir o nome e o projeto a ser criado. Para os nós de processamento (*compute*), é preciso definir a configuração de *network bridges* e um campo (*vm*), com as especificações técnicas da máquina virtual onde será instalado o Juju. Por fim, o campo *net_fix* trata das configurações para permitir encaminhamento de pacotes e acesso à Internet pelas máquinas virtuais. O arquivo completo encontra-se na seção Apêndice C deste trabalho.

3.3 PLAYBOOK ANSIBLE

Como uma segunda etapa deste projeto, um conjunto de *playbooks Ansible* foi construído para confirmar como processos de instalação e configuração via terminal podem ser feitos sem dependência da biblioteca *subprocess*, utilizada nos *scripts* desenvolvidos na versão inicial desenvolvida neste trabalho. Essas configurações foram modularizadas em um arquivo principal denominado *playbook.yaml*, que realiza chamada a outros arquivos, cada

um referente a uma função.

Para isso, foram desenvolvidos *playbooks Ansible* que realizam as seguintes funções:

- Instalação das Aplicações MAAS e *Postgresql;*
- Configuração do banco de dados;
- Inicialização da máquina controller do MAAS;
- Criação de usuário para aplicação MAAS;
- Deleção das aplicações, dos usuários e dos bancos de dados criados.

Listagem 4: *Playbook* principal da estratégia de automação de instalação via *ansible*.

```
- hosts: maas
 # activate privilege escalation
 become: true
 vars_files:
 - variables.yaml
 tasks:
 - name: Install
   ansible.builtin.import_tasks: install.yaml
   when: not teardown

 - name: Config Postgres
   ansible.builtin.import_tasks: postgres_config.yaml
   when: not teardown
  - name: Init MAAS
   ansible.builtin.import_tasks: init.yaml
   when: not teardown

 - name: Create login
   ansible.builtin.import_tasks: admin.yaml
   when: not teardown

 - name: Uninstall
   ansible.builtin.import_tasks: uninstall.yaml
   when: teardown
```

O arquivo *variables.yaml* carrega variáveis pertinentes à instalação das aplicações, como senhas, usuários, versões de instalação e URLs.

Listagem 5: Arquivo de variáveis para *playbook.yaml*.

```
maas_snap_channel: 'edge'
maas_version: '3.4'
maas_installation_type: 'snap'
maas_server_url: 'http://127.0.1.1:5240/MAAS'
maas_username: 'jesus'
maas_password: 'some_password'
maas_email: 'email@gmail.com'
postgres_version: 'postgresql-14'
postgres_user: 'maas'
postgres_password: 'some_password'
postgres_database: 'maas'
ssh_key_filename: 'maas_ssh_key'
# Define teardown tag true to uninstall maas and dependencies
teardown: false
```

A primeira parte do *playbook* consiste na instalação das aplicações MAAS e postgresql.

Listagem 6: *Playbook* instalação de aplicações.

```
- name: Install MAAS
 community.general.snap:
   name: maas
   channel: '{{ maas_version }}/{{ maas_snap_channel }}'
   # present = snap install
   state: 'present'
 # Status of this play
 register: maas_new_installation
 # Condition to run task
 when: not teardown and maas_installation_type | lower == 'snap'
- name: Install PostgreSQL
 ansible.builtin.apt:
   name:
     - '{{postgres_version}}'
   # apt update
   update_cache: true
   cache_valid_time: 3600
   state: 'present'
 register: postgres_new_installation
 when: not teardown
```

Para as operações previstas, também foi criado o arquivo *postgres_config.yaml*, responsável por criar um usuário no banco de dados para o MAAS e criar uma base de dados.

Listagem 7: *Playbook* configuração *postgresql*.

```
- name: Create MAAS Postgres user
  community.postgresql.postgresql_user:
    name: '{{postgres_user}}'
    password: '{{postgres_password}}'
    login_user: 'postgres'
    state: present
  become: true
  become_user: postgres

- name: Create MAAS Postgres Database
  community.postgresql.postgresql_db:
    name: '{{postgres_database}}'
    state: present
    owner: '{{postgres_user}}'
  become: true
  become_user: postgres
```

Finalmente, é inicializado o serviço *MAAS*, com as credenciais do usuário do banco de dados.

Listagem 8: *Playbook* inicialização *MAAS controller*.

```
- name: Initialise MAAS Controller
  ansible.builtin.command: >
    maas init region+rack --database-uri

postgres://{{postgres_user}}:{{postgres_password}}@localhost/{{postgre
s_database}}
    --maas-url {{maas_server_url}}
  # Limit workers for this tasks:
  throttle: 1
```

Com o MAAS inicializado, é necessária a criação de um usuário na aplicação MAAS, com permissão do administrador para gerenciar as máquinas. Para isso, gera-se uma API *key* e é adicionada uma chave ssh criada da máquina para a aplicação.

Listagem 9: *Playbook* criação usuário administrador MAAS e adição de chave ssh.

```
- name: Generate SSH key
  community.crypto.openssh_keypair:
    path: '~/.ssh/{{ssh_key_filename}}'
    type: ed25519
    state: present
  register: ssh_key

- name: Create MAAS admin
  ansible.builtin.shell: >
    maas createadmin
    --username={{maas_username}}
    --password={{maas_password}}
    --email={{maas_email}}
  when: ssh_key is defined
  register: maas_new_admin
  become: true

- name: Await Server Up
  ansible.builtin.shell: |
    maas status | grep http
  register: status
  become: true
  retries: 3
  delay: 10
  until: status.stdout is search('RUNNING')

- name: Generate MAAS API key
  ansible.builtin.shell: |
    maas apikey --username={{maas_username}}
  when: maas_new_admin is defined
  become: true
  register: maas_api_key

- name: Add SSH key to maas
  ansible.builtin.shell: |
    maas login {{maas_username}} {{maas_server_url}}
{{maas_api_key.stdout}}
```

```
    maas {{maas_username}} sshkeys create key="{{ssh_key.public_key}}"
  when: maas_api_key is defined
  become: true
```

Uma *flag teardown* é utilizada no *playbook*, nas variáveis ou na execução, para definir o que será feito. Foi, então, criado um último *playbook*, para desinstalar totalmente todos os dados e aplicações instalados até então. Ele só será executado, e nesse caso somente ele, se a *flag* tiver valor booleano *True*.

Listagem 10: *Playbook* remoção de aplicações.

```
- name: Uninstall MAAS
  community.general.snap:
    name: maas
    state: absent

- name: Delete MAAS Postgres Database
  community.postgresql.postgresql_db:
    name: '{{postgres_database}}'
    state: absent
  become: true
  become_user: postgres

- name: Delete MAAS Postgres user
  community.postgresql.postgresql_user:
    name: '{{postgres_user}}'
    state: absent
  become: true
  become_user: postgres

- name: Uninstall Postgres
  ansible.builtin.apt:
    name: '{{postgres_version}}'
    state: absent

- name: Remove ssh key
  ansible.builtin.file:
    path: '{{item}}'
    state: absent
  with_items:
    - /root/.ssh/{{ssh_key_filename}}
    - /root/.ssh/{{ssh_key_filename}}.pub
  become: true
```

Para a instalação e configuração realizada utilizando as automatizações providas pela ferramenta *Ansible*, algumas funções previamente desenvolvidas como *scripts python*, e listadas anteriormente neste capítulo, tornaram-se obsoletas:

- *installation()*
- *database_configuration()*
- *maas_create_admin()*
- *first_time_config()*

Além disso, os recursos e mecanismos de *playbooks* Ansible permitem eliminar o uso de invocações a comandos usando a biblioteca *subprocess*. Assim, na instalação do Juju, por exemplo, poderão tornarem-se desnecessárias as seguintes funções:

- *juju_install()*
- *juju_config()*
- *resolve_internet_issues()*

3.4 BUNDLE OPENSTACK

Para a implantação dos serviços OpenStack, é utilizado um arquivo de configuração específico (*bundle*), que carrega as informações de configuração de cada serviço, juntamente às integrações e especificações técnicas das máquinas a serem utilizadas. Uma versão inicial deste arquivo foi desenvolvida no trabalho de conclusão de curso "Projeto e Implementação de um Ambiente de Capacitação em Computação em Nuvem com Virtualização" (OLIVEIRA 2024). A partir dessa versão, algumas adaptações foram realizadas para configurar a máquina responsável pela aplicação *cinder*, e integrações para *ovn*. O arquivo completo se encontra no apêndice A deste trabalho.

4. ESTUDOS E DESENVOLVIMENTO

Tendo desenvolvido uma primeira versão dos mecanismos para automatizar a configuração e implantação de uma infraestrutura de computação em nuvem usando OpenStack, os desafios deste trabalho voltaram-se para o refinamento dos *scripts* criados e para a produção de uma versão que utilize invocações diretas às chamadas das *APIs* relevantes. Assim, surgiram duas questões principais.

Primeiro, como criar um *script*, ou a automatização do conjunto de passos desejados, de forma que seja confiável e que permita portabilidade?

Nesse sentido, as estratégias, os mecanismos e os *scripts* desenvolvidos neste trabalho talvez ainda sejam provas de conceito (*Proof of Concept - POC*), validando de forma visual e executável que a ideia de um produto que realize a implantação de um ambiente em nuvem do zero é possível. Ainda que conste como uma solução para o problema, o código inicial de referência, utilizando a biblioteca *subprocess*, possuía problemas, principalmente relacionados à portabilidade. Um exemplo comum é o fato de que a alteração de qualquer alteração na versão da CLI de uma das aplicações utilizadas no fluxo deve gerar alterações manuais dentro do código em cada utilização do comando.

A proposta de automação deste trabalho engloba o processo de implantação de um ambiente de nuvem, a partir de uma máquina que chamamos de *controller*, onde ficam as informações principais e *hosts* do servidor, que inicialmente não possui nenhum ambiente configurado, porém possui acesso à Internet.

Ambientes em produção normalmente possuem uma dependência de expansão de recursos físicos, portanto a automação não define muito bem todos os casos do mundo de computação em nuvem.

Assim, em segundo lugar, percebeu-se uma janela oportuna para acrescentar à proposta de automação deste trabalho, uma alternativa para inserir em um ambiente em produção, uma nova máquina.

4.1 COMO UTILIZAR *APIs* PARA TORNAR OS MECANISMOS MAIS ROBUSTOS

Uma vez visualizado e compreendido o código proposto para automação de implantação visto na seção 3 deste trabalho, foi possível aprimorar a solução desenvolvida, tornando-a mais confiável para a implantação com o uso de *APIs* e bibliotecas que forneçam *clients*, conceito previamente explicado neste trabalho.

Um exemplo de ferramenta poderosa para esta tarefa é o *framework* web FastAPI, utilizado normalmente para construções de *API* utilizando a linguagem Python. Este *framework* possibilita redirecionar de maneira simples as modularizações do código para

endpoints de uma *API*, que pode ser utilizada para que usuários da aplicação requisitem as ações para implantação de um ambiente em nuvem.

Outro aspecto importante no desenvolvimento dos mecanismos de automatização da instalação de uma plataforma de nuvem é a facilidade da manutenção da aplicação, o que foi buscado com o uso do gerenciador denominado Poetry (POETRY [s.d.]). Trata-se de uma ferramenta de gerenciamento de empacotamento e dependência da linguagem Python, que permite de maneira centralizada adicionar bibliotecas para um projeto, apontando suas versões suportadas, de maneira fixa ou em faixa, e permitindo importação de outros repositórios como dependência do projeto. Para instalação do Poetry é necessária a criação de um ambiente virtual dedicado ao projeto, para que seja possível instalar diferentes bibliotecas separadamente do sistema principal, garantindo assim que somente a ferramenta gerencie as dependências, de forma que atualizações no sistema não afetem dependências do projeto.

Uma vez preparado o ambiente de desenvolvimento com ambas ferramentas, a ideia intuitiva para transformação do código em uma *API* é converter as principais funcionalidades que se comunicam as ferramentas gerenciadoras de máquina e softwares, MAAS e Juju, para realizar os passos para implantação, como explicados nas seções anteriores. Assim, para a primeira dependência de configuração do servidor *bare metal*, seria necessário executar apenas um subconjunto das funções (*scripts*) desenvolvidas, que são as seguintes:

- first_time_config()
- set_gateway_ip()
- await_image_import()
- fetch_pxelinux()
- ipmi_reset()
- commission_node()
- await_nodes_status(4)
- create_compute_bridge()
- create_lxd_bridge()
- await_http_server()
- lxd_init()
- create_vm_host()
- create_vm()
- juju_install()
- juju_config()

- resolve_internet_issues()
- juju_bootstrap()
- create_model()
- deploy_bundle()
- unseal_vault()

A partir da separação das rotas em *endpoints* para que sejam chamadas de forma individual, surgiram duas questões: primeiro, como garantir que um passo foi devidamente executado. Essa questão pode ser facilmente englobada na própria estratégia das bibliotecas *libmaas* e *libjuju*, que são bibliotecas assíncronas. Portanto, para a execução de uma chamada para gerenciar certa configuração no ambiente, é preciso acopladar a premissa *await* da linguagem python, que por si garante a espera de uma resposta do serviço. Segundo, é importante saber como garantir que um passo foi executado para executar-se o próximo. Esta segunda questão carrega uma complexidade um pouco maior do que o código de automação proposto neste trabalho, primeiramente devido à migração de escopos de uma execução totalmente síncrona para uma ambientação assíncrona. Nesse quesito, não basta migrar o código atual para uma função em um *endpoint*, pois isso permite que sejam chamados ao mesmo tempo. Esse problema pode ser resolvido utilizando-se o conceito de *worker*, usando uma aplicação para executar em segundo plano uma série de ações.

A ideia para manter o fluxo sequencial fidedigno, evitando que um passo seja executado antes de outro, foi a criação de um objeto para indicar o estado atual da automação e o trabalho realizado pelo *worker*. Assim, os *endpoints* responsáveis por cada função da automação tornam-se referência ao estado do objeto, atualizando o mesmo uma vez que concluída a execução da rota. Qualquer possibilidade de erro pode ser abordada com o tratamento de exceções da linguagem Python, retornando ao cliente a causa do erro e registrando no objeto proposto uma marcação de erro ao invés de atualização no estado, fazendo com que este execute a rota novamente, até que este passo seja concluído.

Uma vez adotada a estratégia para solução de migração para *API* vista acima, se faz necessária a criação de uma rota específica para criação do objeto que será processado pelo *worker*, a fim de realizar a implantação, e por consequência uma rota para a atualização do estado desse objeto. Para a criação desse objeto, pode-se carregar como informação a mesma do arquivo de configuração yaml, para que seja possível em cada passo utilizar essas informações, que são necessárias para a implantação. Toda informação de objeto precisa ser armazenada, o que diretamente faz surgir a necessidade de um banco de dados para que continuamente seja possível alterar e armazenar o estado de um objeto.

Para carregar toda informação necessária para configuração de um objeto, faz sentido pensar sobre estratégias para validação do campo, tendo como possibilidade a alternativa de validação via código na própria rota de criação, verificando a existência dos campos recebidos na chamada dessa rota, baseado no arquivo de configuração yaml esperado, visto na seção 3.2 deste trabalho. Outra alternativa de armazenamento é a utilização da biblioteca *sqlalchemy* (SQLALCHEMY [s.d.]), que permite fazer uma verificação dos campos a partir de seus tipos. Para isso, seria necessário transformar o objeto para que cada campo do arquivo de configuração torne-se um campo variável do objeto e, consequentemente, uma coluna na tabela do banco de dados. A rota de atualização trivialmente torna-se uma co-dependência de armazenamento de dados, o que foi tratado acima.

4.2 COMO ADICIONAR UMA NOVA MÁQUINA EM UMA NUVEM EXISTENTE

Uma vez configurado e estabilizado um ambiente de computação em nuvem, é comum que seja necessário expandir sua capacidade, por exemplo, adicionando novos computadores.

Visando adicionar uma nova máquina em um ambiente de computação em nuvem em funcionamento, serão necessários quatro passos, dos quais três já existem dentro do conjunto de *scripts* explicado na seção 3.1 deste trabalho, bastando apenas algumas modificações para prover essa funcionalidade.

A fim de adicionar uma nova máquina numa configuração de nuvem OpenStack já existente, devem ser realizados os seguintes passos:

1. Adição de máquina ao MAAS: o primeiro passo seria realizar um *PXE boot* da máquina. Para isso, pode-se usar a função *ipmi_reset,* que já foi implementada, e realiza a listagem de todos os *subnet IPs* do servidor MAAS. Dada a quantidade de nós que serão utilizados e utilizando o *ipmitool*, inicia-se as máquinas remotas definidas por seus endereços IP, evitando que máquinas já ligadas sejam ligadas novamente.
2. Comissionamento da nova máquina: atualmente, o comissionamento é feito pela função *commission_node*, que recupera os *nodes_ids* das máquinas, espera até que estejam no *status new*, onde podem ser comissionadas, e realiza a ação. Porém, para adicionar uma nova máquina, uma mudança teria de ser feita, para evitar comissionamento de máquinas já comissionadas, o que pode ser feito verificando e excluindo da lista os nós com *status commissioned*.
3. Criar Ponte (*Network Bridge)*: Após comissionamento, faz-se necessário adicionar uma *network bridge* para que a máquina faça parte de uma rede de internet virtual dentro da nuvem. Esse passo é realizado pela função *create_compute_bridge*, que

tem de ser modificada, pois realiza a criação para todos os nós. Máquinas já ativas devem ser removidas da lista, mantendo somente a nova máquina que será adicionada.

4. Adicionar máquina ao Juju: por fim, seria necessário adicionar a máquina ao Juju e realizar o *deploy* do serviço desejado a máquina. Essa ação teria de ser implementada em uma nova função, que utilize o comando *juju add-machine* da CLI do Juju, que permite adicionar uma máquina existente no MAAS pelo seu IP.

Após a adição de máquina a um ambiente em produção, em razão da necessidade de estabilizar a nuvem, é comum que seja preciso realizar um balanceamento de instâncias e cargas de trabalho. Para isso, podem ser realizadas as denominadas *live migrations*.

Um caso de demanda para migrações, denominada como *live migration*, é a necessidade de movimentação de um serviço instanciado de máquina virtual da nuvem de um *host* para um novo. Tal feito pode ser induzido tendo uma causa de adição de nova máquina, visando expansão, como citado acima, mas não se prendendo unicamente a essa causa. Essa movimentação também é útil quando há necessidade de manutenção de uma máquina de processamento (*compute node*).

Segundo a própria documentação do OpenStack, alguns passos de configuração devem ser seguidos, para realizar uma *live migration*. Esse tipo de migração ainda recebe essa classificação devido ao fato de uma instância continuar em execução enquanto está sendo migrada, não sendo preciso interromper a aplicação. Ademais, migrações podem possuir uma subclassificação baseada na maneira que lidam com unidades de armazenamento da instância sendo migrada. Há três subtipos de *live migration*: *shared storage-based live migration*, onde a instância fonte compartilha com a destino um mesmo armazenamento de discos efêmeros; *block live migration*, no caso em que os discos efêmeros da instância fonte não são compartilhados com a destino. Portanto, para ser realizada a migração, é preciso uma cópia dos discos da fonte para o destino, tomando logicamente mais tempo e aumentando a carga da rede para transferência. O terceiro tipo é denominado *volume-backed live migration,* onde as instâncias utilizam volumes ao invés de discos. Estes podem ser acoplados e desacoplados de uma instância e funcionam similarmente ao que ocorre com um dispositivo USB, basicamente concedendo acesso a leitura e escrita para instância enquanto acoplado e carregando a informação, portanto de certa forma movendo, quando desacoplado e acoplado a nova instância.

Visando realizar qualquer um dos subtipos de *live migratio*n, a própria documentação do OpenStack diz que é necessária a configuração de máquinas de processamento (*compute nodes*), e recomenda-se utilização do driver de virtualização libvirt e do KVM hypervisor.

Para realizar as migrações do tipo *block* e *volume-based live migration*, faz-se necessária a configuração de campos em cada nó de compute relevante à migração, explicitados a seguir, e reinicialização do serviço *nova-compute.* Como essa ação constitui-se principalmente de acesso a máquinas desejadas e alterações de configurações de arquivos, via terminal, além de reinicialização de serviço, assim como os passos vistos anteriormente, uma grande aliada a automação seria a ferramenta Ansible.

Listagem 11: Campos necessários para migrações do tipo *block* e *volume-based.*

```
[libvirt]
live_migration_with_native_tls = true
live_migration_scheme = tls
```

Fonte: Documentação OpenStack, *secure live migration with QEMU-native TLS.*

Por fim, visando *live migrations* do tipo *shared storage*, a documentação do OpenStack indica que existe uma variedade de opções para compartilhamento de armazenamento, sendo que uma forma é utilizar um servidor *Network File System*(NFS) v4, que visa permitir acessar e gerenciar pastas ou arquivos remotos como se fossem locais. Para concluir essa ação, uma estratégia é fazer com o UID e o GID do usuário *nova* estejam idênticos ao do servidor NFS, criar um diretório com espaço suficiente para as instâncias da nuvem, mudar o nível de acesso e execução do diretório de instâncias para que o QEMU possa acessá-lo, exportar o diretório para os *compute hosts*. Uma vez que o servidor NFS esteja configurado, é necessário montar o sistema de arquivos remotos em todos os *compute hosts*, o que pode ser feito adicionando o novo servidor no arquivo /etc/fstab, e testando montar o diretório de instâncias e checando as permissões de acesso para o usuário do *nova*. Novamente, percebe-se em todo o fluxo, série de execuções de comandos e alterações em arquivos via terminal, permitindo mais uma vez o auxílio da ferramenta Ansible para automatizar todo o processo.

5 RESULTADOS

O código que realiza a automação do processo de implantação de ambiente em nuvem, em sua versão já completamente implementada, que ainda realiza invocações em linha de comando, foi totalmente testado em um ambiente de testes denominado Stratus, provisionado pelo projeto de extensão “Pesquisa e desenvolvimento em tecnologias para data centers utilizando virtualização”, resultado da parceria entre DC - UFSCar e LuizaLabs.

Nesse ambiente, foram considerados dois nós para *compute* e uma máquina como *controller*.

A implantação foi realizada pelo código desenvolvido, partindo-se de uma máquina que possuía um sistema operacional Linux instalado e conexão à Internet. Todo o conjunto de serviços de uma nuvem OpenStack foi implantado sobre esse sistema e os outros 2 nós disponíveis.

Os tempos de execução foram apurados por medições que são encontradas no próprio código python visto na seção 3 deste trabalho, utilizando a biblioteca *time* da linguagem python. Para isso, era feita uma medição antes da execução de uma função e outra logo em sequência após o final dessa, retornando a diferença entre as duas. A Tabela 1 mostra os tempos obtidos em cada uma dessas etapas.

Tabela 1: Medição de tempos para implantação da nuvem.

Step	**Time**
Installation	00h 00min 52s
Database config	00h 00min 01s
MaaS Initialization	00h 01min 50s
Await Http Server	00h 00min 14s
MaaS initial config	00h 00min 42s
Await image import	00h 04min 07s
Fetch PXE file	00h 00min 04s
Await Nodes IPs	00h 00min 04s
Reset Nodes Power	00h 00min 48s
Await nodes being visible	00h 03min 34s

Await Node status New	00h 00min 32s
Node Commissioning	00h 09min 52s
Create node's bridges	00h 00min 35s
Create lxd's bridge	00h 00min 00s
Await MaaS server restart	00h 00min 04s
Initialize lxd	00h 00min 00s
Create Lxd Host (Manual)	00h 02min 10s
Create Juju VM	00h 05min 00s
Juju Installation	00h 02min 30s
Network Issues	00h 02min 00s
Juju Bootstrap	00h 20min 10s
Juju Bundle Deploy	01h 45min 00s
Nova Compute deploy	00h 10min 00s
Integrations	00h 00min 15s
Configure Vault	00h 21min 10s
Total:	03h 11min 35s

Durante a execução, encontrou-se um problema, que foi nomeado pelo desenvolvedor da automação por "*our workers were really tired"*. Este erro foi investigado e descobriu-se que é uma consequência de uma instabilidade da própria CLI do MAAS, uma vez que configurado o ambiente, com as máquinas e *bridges* para a implantação de serviços, sobra a tarefa de criar um *VM HOST*, que será responsável na ferramenta por providenciar máquinas virtuais para os serviços que constituem o OpenStack. Este problema pode ser proveniente de instabilidades de rede e também por instabilidades de versionamento de CLI. Dessa maneira, reforça a questão de instabilidade na dependência de execuções utilizando-se somente a CLI via *subprocess*.

Para contornar o problema em momentos de instabilidade, foi criado um contorno que depende de ação humana, via Web UI para criação do VM HOST. Com o desenvolvimento do novo modelo de ativação de comandos via invocações diretas às chamadas das *APIs*, tal problema deixará de existir.

6 CONCLUSÕES

A proposta de criar um conjunto de procedimentos automatizados para a configuração e implantação de uma plataforma de computação em nuvem OpenStack foi realizada com sucesso neste projeto. É claro, contudo, que trata-se de uma tarefa sujeita a refinamentos e aprimoramentos contínuos, como observado pelas inconveniências de usar-se invocações via comandos de sessão *shell* para a execução de programas e configurações.

Uma vez solucionadas as questões levantadas em meio ao desenvolvimento do código, baseando-se nos estudos e testes práticos, é possível concluir a diferença de confiabilidade da aplicação para automação de implantação de ambiente em nuvem. Percebe-se pelo caso encontrado em meio à execução completa do fluxo que, apesar de servir como prova de conceito, depender somente das chamadas via *subprocess* na primeira versão do código desenvolvido, não é suficiente para que o processo de configuração e implantação de uma infraestrutura de computação em nuvem seja confiável.

De maneira mais ampla, o que se pode concluir está diretamente relacionado ao tratamento de exceções. Quando tratando de APIs, instabilidades podem ocorrer, em razão de dependência de requisição e respostas a um servidor. Tratar isso dentro de um código que se apoia em uma API é facilitador, por contar com apoio de bibliotecas para tratamento de exceções, e possibilitar redirecionamento direto em caso de falha de qualquer ponto. A migração para utilizar as bibliotecas que comunicam-se por APIs, tanto a aplicação MAAS, quanto para a Juju, estudado na seção 4.1 deste trabalho, torna-se algo necessário quando as palavras em questão são confiabilidade e portabilidade da aplicação.

Quando o foco é utilizar a aplicação de automação para beneficio em ambientes já em produção, o estudo realizado em torno de *live migrations*, visto na seção 4.2 deste trabalho, possui resultados satisfatórios e promissores, uma vez que o processo para migração de instâncias torna-se possível para o mundo de automação, permitindo assim portar de maneira promissora a requisição para dentro da API, utilizando a assistência de automações *Ansible*. Essa possibilidade é promissora e benéfica a casos de ambientes que desejam adicionar novas máquinas a seu *cluster*, permitindo dessa maneira realizar a adição ao orquestrador de *software* Juju, e posteriormente realizar balanceamento de instâncias, para que a máquina seja incluída.

Em função dos resultados obtidos e das considerações sobre estratégias e resultados, considera-se como próximos passos estabelecer um ambiente virtual agregando as dependências para construção da API, aliando-se à ferramenta *Poetry;* em seguida, parece importante tratar a migração das funções utilizadas no código atual para eliminação total das chamadas da biblioteca *subprocess*, por meio de utilização de chamada de funções das bibliotecas *libjuju* e *libmaas*. Por fim, considera-se o estabelecimento de regras de

negócio por meio de construção de *endpoints* em uma API, que pode ser formulada utilizando-se da ferramenta *FastAPI*.

Ademais, faz-se necessário no futuro, realizar o teste prático dessas etapas, para tornar a aplicação estável, possivelmente em um ambiente de testes que pode passar por fases de ser reiniciado, para testes,.

REFERÊNCIAS

- GOOGLE. *What is cloud computing*. [s.d.].Disponível em:<https://cloud.google.com/learn/what-is-cloud-computing?hl=pt-br>. Acesso em 7 jun 2024.
- SHIVAJI P. M. et al. *Cloud Computing*. 2010. Disponível em: <https://doi.org/10.48550/arXiv.1003.4074>. Acesso em 7 jun 2024.
- AWS. *What is a data center*. [s.d.]. Disponível em: <https://aws.amazon.com/what-is/data-center/#:~:text=A%20data%20center%20is%20a,stores%20any%20company's%20digital%20data>. Acesso em 7 jun 2024.
- REDHAT. *What are microservices*. 2023. Disponível em: <https://www.redhat.com/pt-br/topics/microservices/what-are-microservices>. Acesso em 7 jun 2024.
- MAAS. [s.d.]. Disponível em: <https://maas.io/docs>. Acesso em 7 jun 2024.
- JUJU. [s.d.]. Disponível em: <https://juju.is/docs/juju>. Acesso em 7 jun 2024.
- CLOUDLAB. [s.d.]. Disponível em: <https://www.cloudlab.us/>. Acesso em 7 jun 2024.
- CLOUDLAB. [s.d.]. Disponível em: <https://docs.cloudlab.us/?_gl=1*1k9o577*_ga*MTY5MDM2ODU1NC4xNzE3NzY5NDI3*_ga_6W2Y02FJX6*MTcyMjQ5MDgwMi4xMS4xLjE3MjI0OTA4MTIuMC4wLjA>Acesso em 14 jun 2024.
- EMULAB. [s.d.]. Disponível em: <https://www.emulab.net/portal/frontpage.php>. Acesso em 1 ago 2024.
- RSPECS. [s.d.].Disponível em: <https://groups.geni.net/geni/wiki/GENIExperimenter/RSpecs>. Acesso em 1 ago 2024.
- OPENSTACK. [s.d.] Disponível em <https://www.openstack.org/>. Acesso em 7 jun 2024.
- OPENSTACK. [s.d]. Disponível em: <https://wiki.openstack.org/wiki/Neutron/ML2>. Acesso em 1 ago 2024.
- PYTHON-LIBMAAS. [s.d.]. Disponível em: <https://maas.io/docs/how-to-use-the-python-api-client>. Acesso em 9 ago 2024.
- MAAS. [s.d]. Disponível em: <https://maas.io/docs/api>. Acesso em 9 ago 2024.
- POSTMAN. [s.d.]. Disponível em: <https://www.postman.com/api-platform/api-client/>. Acesso em 9 ago 2024.
- AWS. *What is application programming interface*. [s.d.]. Disponível em: <https://aws.amazon.com/pt/what-is/api/#:~:text=API%20significa%20Application%20Pr

ogramming%20Interface,de%20servi%C3%A7o%20entre%20duas%20aplica%C3%A7%C3%B5es>. Acesso em 9 ago 2024.

- PYTHON-LIBJUJU. [s.d.]. Disponível em: <https://pythonlibjuju.readthedocs.io/en/latest/>. Acesso em 9 ago 2024.
- OPENSTACK. [s.d.]. *Configure Live Migrations*. Disponível em: <https://docs.openstack.org/nova/latest/admin/configuring-migrations.html#section-configuring-compute-migrations>. Acesso em 11 ago 2024.
- OPENSTACK. [s.d.]. *Secure live migration with QEMU-native TLS*. Disponível em: <https://docs.openstack.org/nova/latest/admin/secure-live-migration-with-qemu-native-tls.html>. Acesso em 19 ago 2024.
- FASTAPI. [s.d.]. Disponível em: <https://fastapi.tiangolo.com/>. Acesso em 21 ago 2024.
- WIREGUARD. [s.d.]. Disponível em: <https://www.wireguard.com/>. Acesso em 29 ago 2024.
- PYTHON. [s.d.]. Disponível em: <https://www.python.org/>. Acesso em 9 ago 2024.
- MATTIOLI, M. T. *Estudo de ferramentas de automação para computação em nuvem*. Trabalho de Conclusão de Curso (Bacharelado em Ciência de Computação) - Universidade Federal de São Carlos, 2023.Disponível em: <https://repositorio.ufscar.br/handle/ufscar/18507>. Acesso em 7 jun 2024.
- OLIVEIRA, M. A. *Projeto e implementação de um ambiente de capacitação em computação em nuvem com virtualização*. Trabalho de Conclusão de Curso (Bacharelado em Ciência de Computação) - Universidade Federal de São Carlos, 2023.Disponível em: <https://repositorio.ufscar.br/handle/ufscar/19289>. Acesso em 7 jun 2024.
- INTEL. *Preebot Execution Environment (PXE) Specification Version 2.1*. 1999. Disponível em: <https://web.archive.org/web/20131102003141/http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf>. Acesso em 11 ago 2024.
- PYTHON. *Subprocess management*. [s.d.]. Disponível em: <https://docs.python.org/3/library/subprocess.html>. Aceso em 25 ago 2024.
- PYTHON. *Compression using the lzma algorithm*. [s.d.]. Disponível em: <https://docs.python.org/3/library/lzma.html>. Aceso em 25 ago 2024.
- PYTHON. *Secure hashes and message digests*. [s.d.]. Disponível em: <https://docs.python.org/3/library/hashlib.html>. Aceso em 25 ago 2024.
- PYYAML. [s.d.]. Disponível em: <https://pyyaml.org/>. Aceso em 25 ago 2024.
- PYTHON. *Read and write tar archive files*. [s.d.]. Disponível em: <https://docs.python.org/3/library/tarfile.html>. Aceso em 25 ago 2024.
- PYTHON. *Generate temporary files and directories*. [s.d.]. Disponível em: <https://docs.python.org/3/library/tempfile.html>. Aceso em 25 ago 2024.

- PYTHON. System specific parameters and functions. [s.d.]. Disponível em: <https://docs.python.org/3/library/sys.html>. Aceso em 25 ago 2024.
- PYTHON. Miscellaneous operating system interface. [s.d.]. Disponível em: <https://docs.python.org/3/library/os.html>. Aceso em 25 ago 2024.
- LXD. [s.d.]. Disponível em: <https://canonical.com/lxd>. Acesso em 25 ago 2024.
- GENI. [s.d.]. Disponível em: <https://www.geni.net/about-geni/what-is-geni/>. Acesso em 1 ago 2024.
- ANSIBLE. [s.d.]. Disponível em: <https://www.ansible.com/>. Acesso em 11 ago 2024.
- PYPY. Urllib3 [s.d.]. Disponível em: <https://pypi.org/project/urllib3/>. Acesso em 25 ago 2024.
- PYTHON. JSON enconder and decoder. [s.d.]. Disponível em: <https://docs.python.org/3/library/json.html>. Acesso em 25 ago 2024.
- PYTHON. Subprocess management. [s.d.]. Disponível em: <https://docs.python.org/3/library/subprocess.html>. Acesso em 9 ago 2024.
- PYPY. Psycopg2. [s.d.]. Disponível em: <https://pypi.org/project/psycopg2/>. Acesso em 25 ago 2024.
- POSTGRESQL. [s.d.]. Disponível em: <https://www.postgresql.org/>. Acesso em 25 ago 2024.
- SNAPCRAFT. [s.d.]. Disponível em: <https://snapcraft.io/>. Acesso em 25 ago 2024.
- IPMITOOL. [s.d.]. Disponível em: <https://github.com/ipmitool/ipmitool>. Acesso em 25 ago 2024.
- KERNEL. The linux kernel archives. [s.d.]. Disponível em: <https://kernel.org/>. Acesso em 25 ago 2024.
- VAULT. [s.d.]. Disponível em: <https://developer.hashicorp.com/vault>. Acesso em 26 ago 2024.
- POETRY. [s.d.]. Disponível em: <https://python-poetry.org/>. Acesso em 22 ago 2024.
- SQLALCHEMY. [s.d.]. Disponível em: <https://www.sqlalchemy.org/>. Acesso em 25 ago 2024.
- IBM. 2024. Disponível em: <https://www.ibm.com/topics/cloud-computing>. Acesso em 7 jun 2024.
- DHCP. *Protocolo DHCP*. 2023. Disponível em: <https://learn.microsoft.com/pt-br/windows-server/networking/technologies/dhcp/dhcp-top>. Acesso em 25 ago 2024.

APÊNDICES

A. ARQUIVO *BUNDLE.YAML* JUJU

```
series: jammy

machines:
 '0': # mysql 1, ovn-central 1
   constraints: mem=6G cores=4 root-disk=20G
 '1': # mysql 2, ovn-central 2
   constraints: mem=6G cores=4 root-disk=20G
 '2': # mysql 3, ovn-central 3
   constraints: mem=6G cores=4 root-disk=20G
 '3': # glance
   constraints: mem=6G cores=4 root-disk=20G
 '4': # keystone
   constraints: mem=6G cores=4 root-disk=20G
 '5': # neutron-api
   constraints: mem=6G cores=4 root-disk=20G
 '6': # placement
   constraints: mem=6G cores=4 root-disk=20G
 '7': # nova-cloud-controller
   constraints: mem=6G cores=4 root-disk=20G
 '8': # dashboard
   constraints: mem=6G cores=4 root-disk=20G
 '9': # vault
   constraints: mem=6G cores=4 root-disk=20G
 '10': # cinder
   constraints: mem=4G cores=4 root-disk=532G

relations:
- - nova-cloud-controller:identity-service
 - keystone:identity-service
- - glance:identity-service
 - keystone:identity-service
- - neutron-api:identity-service
 - keystone:identity-service
- - neutron-api:amqp
 - rabbitmq-server:amqp
- - glance:amqp
 - rabbitmq-server:amqp
- - nova-cloud-controller:image-service
 - glance:image-service
```

```
- - nova-cloud-controller:amqp
 - rabbitmq-server:amqp
- - openstack-dashboard:identity-service
 - keystone:identity-service
- - nova-cloud-controller:neutron-api
 - neutron-api:neutron-api
- - placement:identity-service
 - keystone:identity-service
- - placement:placement
 - nova-cloud-controller:placement
- - keystone:shared-db
 - keystone-mysql-router:shared-db
- - glance:shared-db
 - glance-mysql-router:shared-db
- - nova-cloud-controller:shared-db
 - nova-mysql-router:shared-db
- - neutron-api:shared-db
 - neutron-mysql-router:shared-db
- - openstack-dashboard:shared-db
 - dashboard-mysql-router:shared-db
- - placement:shared-db
 - placement-mysql-router:shared-db
- - vault:shared-db
 - vault-mysql-router:shared-db
- - keystone-mysql-router:db-router
 - mysql-innodb-cluster:db-router
- - nova-mysql-router:db-router
 - mysql-innodb-cluster:db-router
- - glance-mysql-router:db-router
 - mysql-innodb-cluster:db-router
- - neutron-mysql-router:db-router
 - mysql-innodb-cluster:db-router
- - dashboard-mysql-router:db-router
 - mysql-innodb-cluster:db-router
- - placement-mysql-router:db-router
 - mysql-innodb-cluster:db-router
- - vault-mysql-router:db-router
 - mysql-innodb-cluster:db-router
- - neutron-api-plugin-ovn:neutron-plugin
 - neutron-api:neutron-plugin-api-subordinate
- - ovn-central:certificates
 - vault:certificates
- - ovn-central:ovsdb-cms
 - neutron-api-plugin-ovn:ovsdb-cms
- - neutron-api:certificates
 - vault:certificates
- - vault:certificates
 - neutron-api-plugin-ovn:certificates
- - vault:certificates
```

```
 - glance:certificates
- - vault:certificates
 - keystone:certificates
- - vault:certificates
 - nova-cloud-controller:certificates
- - vault:certificates
 - openstack-dashboard:certificates
- - vault:certificates
 - placement:certificates
- - vault:certificates
 - mysql-innodb-cluster:certificates
- - cinder-mysql-router:db-router
 - mysql-innodb-cluster:db-router
- - cinder-mysql-router:shared-db
 - cinder:shared-db
- - cinder:cinder-volume-service
 - nova-cloud-controller:cinder-volume-service
- - cinder:identity-service
 - keystone:identity-service
- - cinder:amqp
 - rabbitmq-server:amqp
- - cinder:image-service
 - glance:image-service
- - cinder:certificates
 - vault:certificates
- - cinder-lvm:storage-backend
 - cinder:storage-backend
- - ovn-chassis:nova-compute
 - nova-compute:neutron-plugin
- - rabbitmq-server:amqp
 - nova-compute:amqp
- - nova-cloud-controller:cloud-compute
 - nova-compute:cloud-compute
- - glance:image-service
 - nova-compute:image-service
- - ovn-chassis:ovsdb
 - ovn-central:ovsdb
- - ovn-chassis:certificates
 - vault:certificates
applications:
 glance-mysql-router:
   charm: ch:mysql-router
   channel: 8.0/stable
 glance:
   charm: ch:glance
   channel: 2023.1/stable
   num_units: 1
   to:
   - '3'
```

```
keystone-mysql-router:
  charm: ch:mysql-router
  channel: 8.0/stable
keystone:
  charm: ch:keystone
  channel: 2023.1/stable
  num_units: 1
  to:
  - '4'
neutron-mysql-router:
  charm: ch:mysql-router
  channel: 8.0/stable
neutron-api-plugin-ovn:
  charm: ch:neutron-api-plugin-ovn
  channel: 2023.1/stable
neutron-api:
  charm: ch:neutron-api
  channel: 2023.1/stable
  num_units: 1
  options:
    neutron-security-groups: true
    flat-network-providers: physnet1
  to:
  - '5'
placement-mysql-router:
  charm: ch:mysql-router
  channel: 8.0/stable
placement:
  charm: ch:placement
  channel: 2023.1/stable
  num_units: 1
  to:
  - '6'
nova-mysql-router:
  charm: ch:mysql-router
  channel: 8.0/stable
nova-cloud-controller:
  charm: ch:nova-cloud-controller
  channel: 2023.1/stable
  num_units: 1
  options:
    network-manager: Neutron
  to:
  - '7'
dashboard-mysql-router:
  charm: ch:mysql-router
  channel: 8.0/stable
openstack-dashboard:
  charm: ch:openstack-dashboard
```

```
  channel: 2023.1/stable
  num_units: 1
  to:
  - '8'
rabbitmq-server:
  charm: ch:rabbitmq-server
  channel: 3.9/stable
  num_units: 1
  to:
  - '2'
mysql-innodb-cluster:
  charm: ch:mysql-innodb-cluster
  channel: 8.0/stable
  num_units: 3
  to:
  - '0'
  - '1'
  - '2'
ovn-central:
  charm: ch:ovn-central
  channel: 23.03/stable
  num_units: 3
  to:
  - '0'
  - '1'
  - '2'
vault-mysql-router:
  charm: ch:mysql-router
  channel: 8.0/stable
vault:
  charm: ch:vault
  channel: 1.8/stable
  num_units: 1
  to:
  - '9'
cinder:
  block-device: None
  glance-api-version: 2
  to:
  - '10'
cinder-lvm:
  block-device: sdb
  config-flags: "target_helper=lioadm"
nova-compute:
  config-flags: default_ephemeral_format=ext4
  enable-live-migration: true
  enable-resize: true
  migration-auth-type: ssh
  virt-type: qemu
```

```
ovn-chassis:
  bridge-interface-mappings: br-ex:enp1s0
  ovn-bridge-mappings: physnet1:br-ex
```

B. CÓDIGO AUTOMAÇÃO DE IMPLANTAÇÃO DE AMBIENTE EM NUVEM UTILIZANDO SUBPROCESS

```
import subprocess
import hashlib
import lzma
import tarfile
import urllib3
import tempfile
import time
import yaml
import psycopg2
import json

from sys import argv
from os import getuid
from os import listdir, path


if getuid() != 0:
  raise Exception(f"Você precisa ser root. Use: sudo {'
'.join(argv)}")


def bash_to_string(cmd):
 """
 Get an bash output as python string
 """

 # runs cmd until a desired response is returned
 cond = True
 while(cond):
   try:
     output = subprocess.check_output(["/bin/sh", "-c", cmd])
```

```
      output = output.decode('utf-8').strip()
      cond = False
    except:
        pass
  return output


ip = bash_to_string('hostname -i')
config = {}
art = ''


def run_cmd(cmd: str, /):
  """
  Runs any shell command.
  """


  subprocess.run(["/bin/sh", "-c", cmd]).check_returncode()


def apt_install(cmd: str):
  """
  Install any apt package
  """


  # Guarantee apr install by getting apt update before
  run_cmd('apt update -y')
  run_cmd(f'apt install {cmd} -y')


def clean_postgres():
  """
  Clean postgres created data
  """


  db = config['postgres']['database']
  user = config['postgres']['username']
  del_db = f'DROP DATABASE {db}'
  del_user = f'DROP USER {user}'
```

```
    # Deleting database and user
    run_cmd(f'sudo -iu postgres psql --command="{del_db}"')
    run_cmd(f'sudo -iu postgres psql --command="{del_user}"')


def delete_maas():
    """
    Delete maas, erasing all data
    """

    username = config['maas']['username']

    # Stop maas service, so we can delete the database
    run_cmd('snap stop maas')

    # Cleaning postgres data
    clean_postgres()

    # Completely deleting maas
    run_cmd('snap remove --purge maas')

    # Completely deleting postgre
    run_cmd('apt remove postgresql-14 -y')

    # Delete created ssh key if generated
    try:
        key = config['ssh']['maas_key']
        run_cmd(f'rm /root/.ssh/{key}')
        run_cmd(f'rm /root/.ssh/{key}.pub')
    except:
        print('ssh keys already erased')

    # Delete created api key if generated
```

```
 try:
    run_cmd(f'rm api-key-{username}-file')
 except:
    print('api key already erased')


 # Delete ipmi if installed
 try:
    run_cmd('apt remove ipmitool -y')
 except:
    print('ipmi is not installed')

 # Delete jq if installed
 try:
    run_cmd('snap remove --purge jq')
 except:
    print('jq is not installed')


def main():
 """
 Main
 """
 # Complete all maas steps
 print(f'Creating and configuring maas')
 start = time.time()
 maas()
 print(f'time elapsed to conclude maas steps {(time.time() -
start):.3f}')

 print(f'Creating and configuring juju')
 start = time.time()
 juju()
 print(f'time elapsed to conclude juju steps {(time.time() -
start):.3f}')


def install_dependencies():
  apt_install('ipmitool')
  run_cmd('snap install jq')

def installation():
```

```
    """
    Install maas and postgre
    """


    # Instaling maas and postgresql
    run_cmd(f"snap install maas
--channel={config['maas']['channel']}/edge")
    postgre = config['postgres']['version']
    apt_install(postgre)


    # Define default user password so it can be used in a connection
    cmd = 'ALTER USER postgres PASSWORD \'postgres\''
    run_cmd(f'sudo -iu postgres psql --command="{cmd}"')


    install_dependencies()


def database_configuration():
    """
    Creates and configures maas database
    """


    password = config['postgres']['password']
    user = config['postgres']['username']
    create_user = f"CREATE USER {user} WITH ENCRYPTED PASSWORD
'{password}'"


    conn = psycopg2.connect('user=postgres password=postgres
host=localhost')


    # Apply changes
    conn.autocommit = True

    # Cursor to perform operations
    cur = conn.cursor()
```

```
 # Create maas user
 cur.execute(create_user)


 db = config['postgres']['database']
 cur.execute(f'CREATE DATABASE {db} OWNER {user}')


 # Close
 cur.close()
 conn.close


def maas_create_admin():
 """
 Create admin on maas with given information
 """


 user = config['maas']['username']
 password = config['maas']['password']
 email = config['maas']['email']
 run_cmd(f'maas createadmin --username "{user}" --password
"{password}" --email "{email}"')


def await_http_server():
 """
 Check http server till it is running
 """


 # While status != Running http server is off
 cond = True
 while(cond):
    # Delete extra spaces between words
    try:
       status = ' '.join(bash_to_string('maas status | grep
http').split())
    except:
       continue
    # Creates an array for each column of http status line
```

```
    # furthermore position 1 represents service current status
    status = status.split(' ')
    cond = status[1] != 'RUNNING'


def ip_range():
 """
 Given a starting ip and a size defines a ip range
 """
 start = config['maas']['dhcp']['ip_range']['start']
 size = (256 * config['maas']['dhcp']['ip_range']['size']) - 1
 # Transform ip to list
 first = list(map(str, start.split('.')))
 # Copy first ip to last
 last = first.copy()
 for i in range(3, 0, -1):
    # In each ip position it will sum the
    # size's division by 256 remainder
    sum = int(last[i]) + size%256
    last[i] = str(sum%256)
    # New size for next position
    # equals how many times we exceeded 256 (greater or equal)
    # in this position (size//256) +
    # edge case when position + remainder
    # exceed 256 as well (sum >= 256)
    size = (sum >= 256) + size//256


 return ['.'.join(first), '.'.join(last)]
 def maas_ssh_key():
 """
 Create and add a ssh key to MAAS
 """
 username = config['maas']['username']
 key = config['ssh']['maas_key']
 run_cmd(f'ssh-keygen -t ed25519 -f ~/.ssh/{key} -N ""')
 run_cmd(f'maas {username} sshkeys create key="$(cat
~/.ssh/{key}.pub)"')


def first_time_config():
 """
```

```
    Configure first steps to recent maas instalation
    """
    username = config['maas']['username']
    password = config['maas']['password']


    # Get MAAS API access key and store in api_key
    api_key = f'api-key-{username}-file'
    run_cmd(f'maas apikey --username={username} > {api_key}')


    # Getting api key value to login
    api_key = bash_to_string(f'cat {api_key}')


    # Do login using the key, so it is possible to configure maas
    run_cmd(f'maas login {username} {config['maas']['server']}
{api_key}')


    # Define dns to tigre machine
    dns= config['maas']['dns']
    run_cmd(f'maas {username} maas set-config name=upstream_dns
value="{dns}"')


    # Select Image for download
    version = config['maas']['image_release']
    run_cmd(f'maas {username} boot-source-selections create 1 os="ubuntu"
release="{version}" arches="amd64" subarches="*" labels="*"')


    # Import selected image
    run_cmd(f'maas {username} boot-resources import')


    # Add new sshkey to maas
    maas_ssh_key()


    # Get fabric id for dhcp on eno3 cidr
    cidr = config['maas']['dhcp']['cidr']
```

```
 fabric_id = bash_to_string(f'maas {username} subnet read {cidr} |
grep fabric_id')
 fabric_id = fabric_id.split(':')[1].split(',')[0].strip()


 # Get rack controller name
 symbol = '\'"\''
 rack_controller = bash_to_string(f'maas {username} rack-controllers
read | grep hostname | cut -d {symbol} -f 4')


 # Creating ip range for dhcp
 ip_ranges = ip_range()
 run_cmd(f'maas {username} ipranges create type=dynamic
start_ip={ip_ranges[0]} end_ip={ip_ranges[1]}')


 # Enabling DHCP server
 run_cmd(f'maas {username} vlan update {fabric_id} untagged
dhcp_on=True primary_rack={rack_controller}')


def subnet_id(subnet):
 """
 Gets an subnet id
 """

 username = config['maas']['username']

 id = int(bash_to_string(f'maas {username} subnet read {subnet} | jq
-r \'.id\'').strip())

 return id



def set_gateway_ip():
 """
 Defines gateway ip on a given subnet
 """
```

```
 subnet = config['maas']['dhcp']['subnet']
 gateway = config['maas']['dhcp']['gateway']
 username = config['maas']['username']
 id = subnet_id(subnet)
 run_cmd(f'maas {username} subnet update {id} gateway_ip={gateway}')


def await_image_import():
 """
 Awaiting image importing conclusion
 """
 username = config['maas']['username']


 # Await while image is importing
 cond = True
 while(cond):
    # Check if the image still importing
    status = bash_to_string(f'maas {username} boot-resources
is-importing')
     # If found true on is-importing status, keep waiting
    cond = status.find('true') != -1


def fetch_pxelinux():
 """
 Fetches the lpxelinux.0 file into the proper path.
 """
 http = urllib3.PoolManager()
 url =
"https://mirrors.edge.kernel.org/pub/linux/utils/boot/syslinux/6.xx/sy
slinux-6.03.tar.xz"
 res = http.request("GET", url)


 # Message digest, to secure message received from not guaranteed
secure font
 if hashlib.sha256(res.data).hexdigest() !=
"26d3986d2bea109d5dc0e4f8c4822a459276cf021125e8c9f23c3cca5d8c850e":
    raise ValueError("Validation failed for syslinux-6.03.tar.xz")

 snapshot = None
```

```
 base_path = "/var/snap/maas/common/maas/boot-resources"
 content = 0
 for dir in listdir(base_path):
   if dir.startswith("snapshot"):
     snap_path = path.join(base_path, dir)
     # It may add a temporary snapshot dir at the beginning
     # but it will have only a few folders, so the code is looking
     # for the bigger folder to fetch in
     if len(listdir(snap_path)) > content:
       snapshot = dir
       content = len(listdir(snap_path))


 if not snapshot:
   raise FileNotFoundError("Couldn't find snapshot boot resource
folder")


 lpxe_path = path.join(base_path, snapshot,
"bootloader/pxe/i386/lpxelinux.0")


 with tempfile.NamedTemporaryFile(delete=False) as tmp:
   tmp.write(lzma.decompress(res.data))
   tmp.close()


   tf = tarfile.TarFile(tmp.name)
   reader = tf.extractfile("syslinux-6.03/bios/core/lpxelinux.0")


   with open(lpxe_path, "wb") as out:
     out.write(reader.read())


def ipmi_reset():
 """
 Resets ipmi power, so maas can add machine properly
 """
 username = config['maas']['username']
 subnet = config['maas']['dhcp']['subnet']
```

```
id = subnet_id(subnet)


# Array of ips
subnet_ips = []
# Number of compute nodes
nodes_amount = 2


print('Awaiting nodes ips..')
start = time.time()
# Await number each node be in an ip.
# The minus 1 is due the controller ip that will appear
while len(subnet_ips) - 1 < nodes_amount:
    # Get ips from subnet, to find ipmi server ip
    subnet_ips = bash_to_string(f'maas {username} subnet ip-addresses
{id} | grep ip').split(',')
    subnet_ips = [x.replace('"', '') for x in '
'.join(subnet_ips).split() if x.find('ip') == -1]
print(f'Done after {(time.time() - start):.3f}')


# Cut received string (controller ip) to get only new machine ips
subnet_ips.remove(f'{config['maas']['dhcp']['gateway']}')


for i in subnet_ips:
    user = 'root'
    password = 'root'
    # Reset node power
    print(f'resetting {i}')
    # Connection
    conn = f'ipmitool -I lanplus -H {i} -U {user} -P {password}'
     # Guarantee that there is an ipmi server on the ip
    try:
      run_cmd(f'{conn} chassis power status')
    except:
      continue


    # The node will be either up, so we need to reset it
```

```
    try:
      run_cmd(f'{conn} chassis power reset')
    # Or off so we just need to turn it on
    except:
      run_cmd(f'{conn} chassis power on')


def get_nodes_id(func = lambda x: True):
  """
  Get id for nodes on MAAS
  """
  username= config['maas']['username']
   # Enlist all nodes id
  nodes_id = bash_to_string(f"maas {username} machines read | jq -r
'(.[] | [.hostname, .system_id]) | @tsv' | column -t")
  # Convert bash list to python list
  nodes_id = nodes_id.split('\n')
  nodes_id = [' '.join(x.split()).split()[1] for x in nodes_id if
func(x)]
  return nodes_id


def await_nodes_status(st, ig=-10):
 """
 Awaiting commissioning conlusion
 """


 username= config['maas']['username']


 # Get nodes system_id
 nodes_id = get_nodes_id()
 done = {x : False for x in nodes_id}
 cond = True
 change = True
 # Await every node get status 6 (deployed) 4 (ready) 2 (Failed
Commissioning) 0 (new)
 while(cond):
   if change:
     print(f'Still waiting {nodes_id}')
     change = False
```

```
    for i in nodes_id:
      status = int(bash_to_string(f'maas {username} machine read {i} |
jq .status'))
      done[i] = status == st or status == ig
    for i in done.keys():
      if done[i] == True:
        change = True
        nodes_id.remove(i)
        done[i] = False
    cond = len(nodes_id) > 0


def commission_node():
 """
     Start node comissioning
 """


 username = config['maas']['username']

 print('Awaiting nodes to be visible...')
 start = time.time()
 # Await all nodes being visible
 cond = True
 nodes_amount = config['maas']['nodes_amount']
 while(cond):
     try:
       nodes_id = get_nodes_id()
       cond = len(nodes_id) != nodes_amount
       print(nodes_id)
     except:
       cond = True
 print(f'Done after {(time.time() - start):.3f}')


 print('Taking commission action...')
 # Set default ubuntu release that will be used for comissioning as
jammy
 run_cmd(f'maas {username} maas set-config
name=commissioning_distro_series value="jammy"')
```

```
 print('Awaiting nodes to get status new...')
 start = time.time()
 # Await nodes to be ready for commissioning
 # status 0 = new
 await_nodes_status(0)
 print(f'Done after {(time.time() - start):.3f}')


 for i in range(len(nodes_id)):
    run_cmd(f'maas {username} machine update {nodes_id[i]}
hostname=compute{i}')
    run_cmd(f'maas {username} machine commission {nodes_id[i]}
enable_ssh=1')


def create_compute_bridge():
 """
 Create bridges for compute nodes
 """


 username = config['maas']['username']
 cidr = config['maas']['dhcp']['cidr']




 # Get nodes system_id
 nodes_id = get_nodes_id()


 for i in nodes_id:
    # Get network interface id for current node
    network_interface = bash_to_string(f"maas {username} machine read
{i} | jq -r '.boot_interface.id'")
     # Create bridge and save its id
    bridge_name = config['compute']['bridge']['name']
    bridge_type = config['compute']['bridge']['type']
    bridge = bash_to_string(f'maas {username} interfaces create-bridge
{i} name={bridge_name} parent={network_interface}
bridge_type={bridge_type}| jq .id')
     # Get subnet id where the bridge will be connected
```

```
    subnet_id = bash_to_string(f"maas {username} subnets read | jq -r
'.[] | select(.cidr == \"{cidr}\" and .managed == true).id'")


    # Connecting bridge to subnet
    # Ip assign mode
    mode = config['compute']['bridge']['link_mode']
    run_cmd(f'maas {username} interface link-subnet {i} {bridge}
subnet={subnet_id} mode={mode}')


def create_lxd_bridge():
 """
 Edit netplan to create lxd bridge
 """

 bridge_name = config['lxd']['bridge']


 cloud_file = None
 basepath = '/etc/netplan'
 for dir in listdir(basepath):
    if dir.find('cloud-init'):
       cloud_file = dir


 if not cloud_file:
    raise FileNotFoundError("Couldn't find cloud-init file")

 netplan_path = path.join(basepath, cloud_file)


 # Opening netplan file
 with open(netplan_path,'r') as file:
    netplan_file = yaml.safe_load(file)


 try:
    if bridge_name in netplan_file['network']['bridges'].keys():
       return
 except:
    pass
```

```
 # Removing and saving address from interface
 addr = ['addresses',
netplan_file['network']['ethernets']['eno3'].pop('addresses')]

 # Creating bridge
 netplan_file['network']['bridges'] = {bridge_name : {addr[0] :
addr[1], 'interfaces' : ['eno3']}}


 # Saving cloud-init file
 with open(netplan_path, 'w') as file:
    yaml.dump(netplan_file, file)


 # Apply netplan changes
 run_cmd('netplan apply')


 # Restart maas so it can recognize netplan changes
 run_cmd('snap restart maas')


def lxd_init():
 """
 Initialize lxd
 """


 # lxd init configuration
 password = config['lxd']['password']


 file_name = config['lxd']['init_file_name']
 with open(file_name, 'w+'):
    yaml.dump(config['lxd']['init_file'])


 # Initializing lxd
 run_cmd(f'lxd init --preseed < {file_name}')


def create_vm_host():
```

```
    """
    Create a VM host, to create virtual machines
    """


    username = config['maas']['username']
    lxd_password = config['lxd']['password']
    # Ip is dhcp gateway, because is our bridge gateway
    # due subnet link on bridge
    ip = config['maas']['dhcp']['gateway']


    # Define vm-host type
    type = config['vm_host']['type']
    project_name = config['vm_host']['project']
    name = config['vm-host']['name']
    run_cmd(f'maas {username} vm-hosts create type=\"{type}\"
power_address=\"{ip}\" power_pass=\"{lxd_password}\"
project=\"{project_name}\" name={name}')


def create_vm():
    """
    Create a vm
    """
    username = config['maas']['username']
    tag = config['vm']['juju']['tag']


    # Get VM host id so we can add new vm there
    vm_host_id = bash_to_string(f'maas {username} vm-hosts read | jq -r
\'.[] | .id\'')


    cpus = config['vm']['juju']['cpus']
    ram = config['vm']['juju']['ram'] * 1024
    storage = config['vm']['juju']['storage']
    hostname = config['vm']['juju']['hostname']
    # Compose new vm and stores it's id
    machine_id = bash_to_string(f'maas {username} vm-host compose
{vm_host_id} hostname={hostname} cores={cpus} memory={ram}
disks=1:{storage} | jq .system_id')
```

```
 machine_id = machine_id.replace('"', '')

 # Create tag if it does not exist
 try:
    run_cmd(f'maas {username} tags create name={tag}')
 except:
    pass


 # Adding tag to new vm
 run_cmd(f'maas {username} tag update-nodes {tag} add={machine_id}')


def maas():
 """
 MAAS steps (installation and configuration)
 """


 start = time.time()
 print('Installing maas and postgre...')
 installation()
 print(f'time elapsed to maas installation {(time.time() -
start):.3f}')

 start = time.time()
 print('\nConfiguring Database...')
 database_configuration()
 print(f'time elapsed to db config {(time.time() - start):.3f}')


 start = time.time()
 print('\nInitializing maas...')
 # Initializing maas database and connecting controller
 run_cmd(f'maas init region+rack --database-uri
"postgres://maas:{config['postgres']['password']}@localhost/maas"
--maas-url "{config['maas']['server']}"')


 maas_create_admin()
 print(f'time elapsed to maas init {(time.time() - start):.3f}')
```

```
# Awaiting http server being up
print('Awaiting http server..')
start = time.time()
await_http_server()
print(f'Done after {(time.time() - start):.3f}')


print('\nConfiguring maas for first time...')
start = time.time()
# Does the maas first time configuration
first_time_config()
print(f'time elapsed to configure maas {(time.time() - start):.3f}')


set_gateway_ip()


# Awaiting image import
print('Awaiting image import..')
start = time.time()
await_image_import()
print(f'Done after {(time.time() - start):.3f}')


print(f'Fetching pxe file...')
start = time.time()
# Update pxelinux file on maas
fetch_pxelinux()
print(f'time elapsed to fetch file {(time.time() - start):.3f}')


print('Reseting nodes power (ipmi)...')
start = time.time()
ipmi_reset()
print(f'time elapsed to reset power {(time.time() - start):.3f}')


print('Starting commissioning process...')
start = time.time()
commission_node()
```

```
 print("Awaiting commissioning...")
 # Ready status = 4
 await_nodes_status(4)
 print(f'time elapsed to commission {(time.time() - start):.3f}')


 print('Creating node\'s bridges...')
 start = time.time()
 create_compute_bridge()
 print(f"time elapsed to create node's bridges {(time.time() -
start):.3f}")


 print('Creating lxd\'s bridge...')
 start = time.time()
 create_lxd_bridge()
 print(f"time elapsed to create lxd's bridge {(time.time() -
start):.3f}")


 print('Server restarted, awaiting http server...')
 start = time.time()
 await_http_server()
 print(f'Done after {(time.time() - start):.3f}')

 print('Initializing lxd service...')
 start = time.time()
 lxd_init()
 print(f"time elapsed to initialize lxd {(time.time() - start):.3f}")


 print('Creating lxd vm host...')
 start = time.time()
 create_vm_host()
 print(f"time elapsed to create lxd vm host {(time.time() -
start):.3f}")

 print(f'Creating juju vm')
 start = time.time()
 create_vm()
 # In order to await vm creation
```

```
 await_nodes_status(6, 4)
 print(f"time elapsed to create juju vm {(time.time() - start):.3f}")


def await_services():
 """
 Await till all units are on active state
 """


 json_status = bash_to_string("juju status --format json")
 json_status = json.loads(json_status)
 applications = bash_to_string("juju status --format json | jq
'.applications | keys'")
 applications = json.loads(applications)
 for app in applications:
   if 'units' not in json_status['applications'][app]:
     continue
   for unit in json_status['applications'][app]['units'].values():
     if unit['workload-status']['current'] != 'active':
       return False


 return True


def juju_install():
 """
 Juju installation
 """
 # Create juju directory in case it does not exists
 check_dir = bash_to_string('find ./.local/share/juju')


 if(check_dir.find('No such file or directory') != -1):
   run_cmd('mkdir -p ./.local/share/juju')


 # Install juju via snap
 channel = config['juju']['channel']
 run_cmd(f'snap install juju --channel {channel}')
```

```
def juju_config():
 """
 Juju configuration, adding cloud and credentials
 """


 # Create juju-cloud file
 cloud_file = 'juju-cloud.yaml'
 with open(cloud_file, 'w+'):
    yaml.dump({'clouds' : config['juju']['clouds']})

 # Create clouds on juju
 clouds = list(config['juju']['clouds'].keys())
 for i in clouds:
    run_cmd(f'juju add-cloud --client -f {cloud_file} {i}')


 # Get maas api key to authorize juju credential
 username = config['maas']['username']
 api_key = bash_to_string(f'cat api-key-{username}-file')


 # Create credentials on juju
 credentials = list(config['juju']['credentials'].keys())
 for i in credentials:
    config['juju']['credentials']['i']['maas-oauth'] = api_key


 credential_file = 'juju-credentials.yaml'
 with open(credential_file, 'w+'):
    yaml.dump({'credentials' : config['juju']['credentials']})


 for i in credentials:
    run_cmd(f'juju add-credential --client -f {credential_file} {i}')


def resolve_internet_issues():
 with open('/etc/sysctl.conf', 'r') as file:
    sysctl = file.readlines()
```

```
 # Allow ip forwarding
 ipv4 = 'net.ipv4.ip_forward=1'
 ipv6 = 'net.ipv6.conf.all.forwarding=1'
 for i in range(len(sysctl)):
    try:
       sysctl[i].replace(f'#{ipv4}', ipv4)
       sysctl[i].replace(f'#{ipv6}', ipv6)
    except:
       pass
   # Rewrite sysctl.conf file with changes
 with open('/etc/sysctl.conf', 'w') as file:
    file.write(''.join(sysctl))


 # Apply changes
 run_cmd('sysctl --system')


 cidr = config['net_fix']['ip_saddr']
 internet_ip = config['net_fix']['forward_ip']
 ethernet = config['net_fix']['ethernet']
 # Define nat rules
 nat = f"""table ip nat {{
 chain postrouting {{
       type nat hook postrouting priority srcnat; policy accept;
       ip saddr {cidr} oif "{ethernet}" snat to {internet_ip};
 }}
}}"""
 nat_file = 'nat.rules'
 # Save file
 with open(nat_file, 'w+') as file:
    file.write(nat)

 # Apply nat rules
 run_cmd(f'nft -f {nat_file}')


def juju_bootstrap():
 """
 Bootstrap juju on MaaS node
 """
 bs_series = config['juju']['bootstrap']['series']
```

```
    tags = config['juju']['bootstrap']['tags']
    clouds = list(config['juju']['clouds'].keys())
    run_cmd(f'juju bootstrap --bootstrap-series={bs_series} --constraints
tags={tags} {clouds[0]}')


    # Await juju node to be deployed (status 6)
    # Ignoring nodes that are ready (status 4)
    await_nodes_status(6, 4)


def create_model():
    """
    Add new model to juju
    """


    series = config['juju']['model']['series']
    name = config['juju']['model']['name']


    run_cmd(f'juju add-model --config default-series={series} {name}')


def deploy_bundle():
    """
    Deploy openstack bundle
    """
    username = config['maas']['jesus']


    # Get compute nodes id
    nodes_id = get_nodes_id(lambda x: x.find('compute') != -1)


    # Allocate nodes so they won't be used during deploy
    # They will be used separately to deploy nova
    for i in nodes_id:
        run_cmd(f'maas {username} machines allocate system_id={i}')


    bundle = config['juju']['bundle']
```

```
 bundle = yaml.safe_load(bundle)
  run_cmd(f'juju deploy ./{bundle}')



def unseal_vault():
 vault = bash_to_string("juju status --format json | jq '.applications
| .vault.units'")
 vault = json.loads(vault)
 for vault_unit in vault.values():
   vault_ip = vault_unit['public-address']
   ssh_bash = subprocess.Popen(['ssh', '-tt', '-i',
'~/.local/share/juju/ssh/juju_id_rsa', vault_ip],
                               stdin=subprocess.PIPE,
                               stdout=subprocess.PIPE,
                               )
   ssh_bash.stdin.write('export VAULT_ADDR=http://127.0.0.1:8200 \n')
   ssh_bash.stdin.write('vault operator init --key-shares=1
--key-threshold=1 \n')
   ssh_bash.stdin.close()
   for line in ssh_bash.stdout.readlines():
     if line.find('Key') != -1:
       key = line.split(':')[1].strip()
       run_cmd(f'echo {key} > vault_key')
     elif line.find('Token') != -1:
       token = line.split(':')[1].strip()
       run_cmd(f'echo {vault} > vault_token')
   ssh_bash.stdin.write(f'vault operator unseal {key}\n')
   ssh_bash.stdin.close()
   run_cmd(f'juju run vault/leader authorize-charm token={token}')
   run_cmd('juju run vault/leader generate-root-ca')



def juju():
  """
  Juju steps (installation and configuration)
  """
  print('Installing juju...')
  start = time.time()
  juju_install()
  print(f'time elapsed installing juju {(time.time() - start):.3f}')
   print('Configuring juju...')
  start = time.time()
```

```
  juju_config()
  print(f'time elapsed Configuring juju {(time.time() - start):.3f}')



  print('Resolving internet issues...')
  start = time.time()
  resolve_internet_issues()
  print(f'time elapsed resolving {(time.time() - start):.3f}')
   print('Bootstrapping juju...')
  start = time.time()
  juju_bootstrap()
  print(f'time elapsed bootstrapping {(time.time() - start):.3f}')



  print('Creating juju model openstack...')
  start = time.time()
  create_model()
  print(f'time elapsed to create model {(time.time() - start):.3f}')



  print('Deploying juju openstack bundle...')
  start = time.time()
  deploy_bundle()
  await_services()
  print(f'time elapsed to deploy bundle {(time.time() - start):.3f}')

  print(Unsealing vault...')
  start = time.time()
  unseal_vault()
  await_services()
  print(f'time elapsed to unseal vault {(time.time() - start):.3f}')

art = ''''''



if __name__ == "__main__":
 # example: python3 maas.py FUNCAO CONF_FILE
 if len(argv) > 2:
   # Load yaml config file
   with open(argv[2], 'r') as file:
     config = yaml.safe_load(file)
```

```
if argv[1] == 'main':
  main()
elif argv[1] == 'bash_to_string':
  bash_to_string(argv[3])
elif argv[1] == 'fetch_pxelinux':
  fetch_pxelinux()
elif argv[1] == 'run_cmd':
  run_cmd(argv[3])
elif argv[1] == 'apt_install':
  apt_install(argv[3])
elif argv[1] == 'clean_postgres':
  clean_postgres()
elif argv[1] == 'delete_maas':
  delete_maas()
elif argv[1] == 'maas_create_admin':
  maas_create_admin()
elif argv[1] == 'ip_range':
  ip_range()
elif argv[1] == 'maas_ssh_key':
  maas_ssh_key()
elif argv[1] == 'first_time_config':
  first_time_config()
elif argv[1] == 'get_nodes_id':
  get_nodes_id()
elif argv[1] == 'await_nodes_status':
  if len(argv) > 4:
    await_nodes_status(argv[3], argv[4])
  else:
    await_nodes_status(argv[3])
elif argv[1] == 'commission_node':
  commission_node()
elif argv[1] == 'install_dependencies':
  install_dependencies()
elif argv[1] == 'installation':
  installation()
elif argv[1] == 'database_configuration':
  database_configuration()
elif argv[1] == 'subnet_id':
  subnet_id(argv[3])
elif argv[1] == 'set_gateway_ip':
  set_gateway_ip()
```

```
    elif argv[1] == 'create_lxd_bridge':
        create_lxd_bridge()
    elif argv[1] == 'await_http_server':
        await_http_server()
    elif argv[1] == 'await_image_import':
        await_image_import()
    elif argv[1] == 'ipmi_reset':
        ipmi_reset()
    elif argv[1] == 'create_compute_bridge':
        create_compute_bridge()
    elif argv[1] == 'lxd_init':
        lxd_init()
    elif argv[1] == 'create_vm_host':
        create_vm_host()
    elif argv[1] == 'create_vm':
        create_vm()
    elif argv[1] == 'maas':
        maas()
    elif argv[1] == 'juju_install':
        juju_install()
    elif argv[1] == 'juju_config':
        juju_config()
    elif argv[1] == 'resolve_internet_issues':
        resolve_internet_issues()
    elif argv[1] == 'juju_bootstrap':
        juju_bootstrap()
    elif argv[1] == 'create_model':
        create_model()
    elif argv[1] == 'juju':
        juju()
    elif argv[1] == 'deploy_bundle':
        deploy_bundle()
    else:
        comandos = ['bash_to_string', 'fetch_pxelinux', 'run_cmd',
'apt_install', 'clean_postgres', 'delete_maas', 'maas_create_admin',
'ip_range', 'maas_ssh_key', 'first_time_config', 'get_nodes_id',
'await_nodes_status', 'commission_node', 'install_dependencies',
'installation', 'database_configuration', 'subnet_id',
'set_gateway_ip', 'create_lxd_bridge', 'await_http_server',
'await_image_import', 'ipmi_reset', 'create_compute_bridge',
'lxd_init', 'create_vm_host', 'create_vm', 'maas', 'juju_install',
'juju_config', 'resolve_internet_issues', 'juju_bootstrap',
```

```
'create_model', 'juju', 'deploy_bundle', 'main']
    print(f"{argv[1]} inexistente, Comandos existentes:\n")
    for i in comandos:
        print(i)
 else:
    raise Exception('Necessario passar o arquivo de configuração
yaml. Ex:\npython3 maas.py main arquivo.yaml')
```

C. ARQUIVO MAAS.YAML

```
postgres:
 database: 'maas'
 username: 'maas'
 password: 'some_password'
 version: 'postgresql-14'
ssh:
 maas_key: 'maaskey'
maas:
 channel: '3.4'
 server: 'http://127.0.1.1:5240/MAAS'
 username: 'jesus'
 password: 'some_password'
 email: 'gabriel.jdantas1@gmail.com'
 dhcp:
   subnet: '10.42.0.0/16'
   cidr: '10.42.0.0/16'
   gateway: '10.42.0.1'
   # Ip for dhcp server
   ip_range:
     # Starting ip
     start: '10.42.191.255'
     # Amount of ips will be used
     size: 64
     # This means that 10.42.xx.- 64 ips on xx will be used from xx.0 to
xx.255
 dns: 'x.x.x.x'
 image_release: 'jammy'
 nodes_amount: 2
lxd:
```

```
 bridge: 'br0'
 password: 'some_password'
 init_file_name: 'lxd.yaml'
 init_file:
   config:
    core.https_address: '[::]:8443'
    core.trust_password: "{password}"
   networks: []
   storage_pools:
    - config: {{}}
    description: ""
    name: default
    driver: dir
   profiles:
    - config: {{}}
    description: ""
    devices:
      eth0:
        name: eth0
        nictype: bridged
        parent: br0
        type: nic
      root:
        path: /
        pool: default
        type: disk
    name: default
   projects: []
   cluster: null
vm_host:
 name: 'lxd-controller'
 type: 'lxd'
 project: 'tst-project'
compute:
 bridge:
   name: 'br-ex'
   type: 'ovs'
   link_mode: 'AUTO'
vm:
 juju:
   hostname: 'juju-machine'
   cpus: 2
   ram: 4
   storage: 50
   tag: 'juju'
juju:
 bundle: 'bundle.yaml'
```

```
 channel: '3.1'
 clouds:
   maas-one:
     type: maas
     auth-types: [oauth1]
     endpoint: http://10.42.0.1:5240/MAAS
 credentials:
   maas-one:
     anyuser:
     auth-type: oauth1
     maas-oauth:
 bootstrap:
   series: 'jammy'
   tags: 'juju'
   name: 'maas-controller'
 model:
   series: 'jammy'
   name: 'openstack'
net_fix:
 # Local ips
 ip_saddr: '10.42.0.0/16'
 # Ip where internet request will be forwarded
 forward_ip: x.x.x.x
 ethernet: 'eno2'
```