



**INSTITUTO
FEDERAL**
Brasília

Instituto Federal de Educação, Ciência e Tecnologia de Brasília, campus Taguatinga,
Campus Taguatinga

**PROPOSTA DE IMPLEMENTAÇÃO DE NUVEM PRIVADA NO IFB: UTILIZAÇÃO
EFICIENTE DE RECURSOS COMPUTACIONAIS E FOMENTO À PESQUISA E
ENSINO DA COMPUTAÇÃO EM NUVEM**

Por

MATHEUS FERNANDES BEZERRA

Trabalho de Graduação

BRASÍLIA/2024

Matheus Fernandes Bezerra

**PROPOSTA DE IMPLEMENTAÇÃO DE NUVEM PRIVADA NO IFB:
UTILIZAÇÃO EFICIENTE DE RECURSOS COMPUTACIONAIS E
FOMENTO À PESQUISA E ENSINO DA COMPUTAÇÃO EM NUVEM**

*Trabalho apresentado ao Curso de Ciência da Computação
do Instituto Federal de Educação, Ciência e Tecnologia de
Brasília, campus Taguatinga, como requisito parcial para
obtenção do grau de Bacharel em Ciência da Computação.*

Orientador: Dr. Daniel Saad Nogueira Nunes

BRASÍLIA
2024

Matheus Fernandes Bezerra

Proposta de Implementação de Nuvem Privada no IFB: Utilização Eficiente de Recursos Computacionais e Fomento à Pesquisa e Ensino da Computação em Nuvem/ Matheus Fernandes Bezerra. – BRASÍLIA, 2024-

97 p. : il. (algumas color.) ; 30 cm.

Orientador Dr. Daniel Saad Nogueira Nunes

Trabalho de Graduação – Instituto Federal de Educação, Ciência e Tecnologia de Brasília, *campus* Taguatinga,, 2024.

1. Palavra-chave1. 2. Palavra-chave2. I. Orientador. II. Universidade xxx. III. Faculdade de xxx. IV. Título

CDU 004

Trabalho de Graduação apresentado por **Matheus Fernandes Bezerra** ao curso de Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia de Brasília, *campus* Taguatinga, sob o título **Proposta de Implementação de Nuvem Privada no IFB: Utilização Eficiente de Recursos Computacionais e Fomento à Pesquisa e Ensino da Computação em Nuvem**, orientado pelo **Prof. Dr. Daniel Saad Nogueira Nunes** e aprovado pela banca examinadora formada pelos professores:

Prof. Dr. Fabiano Cavalcanti Fernandes
IFB

Prof. Me. Diego Martins de Oliveira
IFB

Prof. Dr. Daniel Saad Nogueira Nunes
IFB

BRASÍLIA
2024

Resumo

Este trabalho traz um projeto de implementação de uma nuvem privada no Instituto Federal de Brasília (IFB) utilizando a plataforma *open-source* OpenStack, com foco na implantação de um *Cluster* modular por meio do *Kolla-Ansible*, que efetuar a containerização dos serviços do OpenStack. A arquitetura organiza os nós em funções dedicadas para *Compute*, *Storage* e *Controller*, garantindo alocação eficiente de recursos. Embora limitações de hardware tenham impedido a implementação completa, um ambiente de teste demonstrou com sucesso a integração dos serviços e o potencial de escalabilidade do *cluster* com recursos adicionais no campus de Taguatinga.

O projeto utiliza três nós: um nó principal que executa todos os serviços e faz o controle de rede e dois nós especializados para grupos de serviços específicos. Esse *design* simplifica a adição de novos nós ao *cluster*, assegurando isolamento e controle dos dados. Além disso, também mostramos a integração do *Hierarchical Multitenancy* no OpenStack proporciona isolamento lógico avançado entre projetos para possíveis novas pesquisas, e os *templates HEAT* para a automação de infraestrutura como código (IaC), promovendo maior organização e gerenciamento de recursos.

O estudo destaca a viabilidade de, em um futuro próximo, implementar uma nuvem privada no IFB, o que melhoraria o controle da infraestrutura e forneceria recursos valiosos para as pesquisas dos alunos. Essa solução não apenas atende às possíveis necessidades institucionais, mas também estabelece uma base escalável para crescimento futuro, alinhando-se às demandas de ambientes educacionais.

Palavras-chave: Computação em Nuvem, Arquitetura de Nós, OpenStack, Nuvem privada, Virtualização, Kolla-Ansible

Abstract

This work presents a project for the implementation of a private cloud at the Federal Institute of Brasília (IFB) using the open-source platform OpenStack. The focus is on deploying a modular cluster through Kolla-Ansible, which containerizes OpenStack services. The architecture organizes nodes into dedicated roles for Compute, Storage, and Controller, ensuring efficient resource allocation. Although hardware limitations prevented a full implementation, a test environment successfully demonstrated the integration of services and the potential for cluster scalability with additional resources at the Taguatinga campus.

The project utilizes three nodes: a main node that runs all services and manages network control, and two specialized nodes for specific service groups. This design simplifies the addition of new nodes to the cluster, ensuring data isolation and control. Furthermore, the integration of Hierarchical Multitenancy in OpenStack provides advanced logical isolation between projects for potential future research, and HEAT templates for Infrastructure as Code (IaC) automation promote greater organization and resource management.

The study highlights the feasibility of implementing a private cloud at IFB in the near future, which would improve infrastructure control and provide valuable resources for student research. This solution not only meets potential institutional needs but also establishes a scalable foundation for future growth, aligning with the demands of educational environments.

Keywords: Cloud Computing, Node Architecture, OpenStack, Private Cloud, Virtualization, Kolla-Ansible

Lista de Figuras

3.1	Anéis de segurança e suas aplicações. Extraído de (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016), a figura ilustra a separação lógica dos anéis de proteção em sistemas x86. O anel mais privilegiado (anel 0) pode acessar os anéis superiores (como o anel 3), mas o inverso não é permitido — ou seja, o anel 3 não pode acessar diretamente o anel 0.	33
3.2	Tradução de binários em <i>hypervisores</i> . A figura ilustra a separação em anéis de proteção, com o <i>hypervisor</i> operando no anel 0 e o sistema operacional convidado no anel 1. O <i>hypervisor</i> intercepta e reescreve binários críticos antes da execução, realizando a emulação e garantindo a segurança ao evitar o acesso direto ao hardware.	34
3.3	<i>Hypervisor</i> do tipo 1. A imagem ilustra a operação direta com o hardware, onde não há camadas intermediárias entre o <i>hypervisor</i> e o <i>hardware</i> , aumentando a performance.	35
3.4	<i>hypervisor</i> do tipo 2. A imagem mostra a necessidade de um sistema operacional hospedeiro para o funcionamento do <i>hypervisor</i>	36
3.5	Virtualização completa nos anéis de segurança. A figura demonstra o <i>Guest OS</i> operando no anel 1 e o VMM no anel 0, com o <i>hypervisor</i> realizando a emulação.	37
3.6	Paravirtualização nos anéis de segurança. A figura mostra como o sistema operacional convidado modificado realiza chamadas críticas diretamente para o <i>hypervisor</i> através de hypercalls, tornando o tratamento mais eficiente e reduzindo a necessidade de emulação.	37
3.7	Virtualização assistida por hardware nos anéis de segurança. A figura mostra o <i>Guest OS</i> operando no anel 0 e o <i>hypervisor</i> no anel -1, permitindo um gerenciamento eficiente do hardware e reduzindo a necessidade de emulação por software.	38
3.8	Interface para drivers do <i>libvirt</i> . A figura ilustra como o <i>libvirt</i> se conecta a diferentes <i>hypervisores</i> através de drivers específicos, permitindo o gerenciamento padronizado de componentes de virtualização, como rede, armazenamento e dispositivos.	41
3.9	Arquitetura do <i>hypervisor QEMU/KVM</i> . A figura ilustra a interação entre o QEMU e o KVM, mostrando o gerenciamento de vCPUs, E/S e memória, além das transições entre o modo do host e do convidado para virtualização assistida por hardware.	43

3.10	Arquitetura SR-IOV. A figura ilustra como uma única NIC física pode apresentar múltiplas interfaces lógicas (<i>Virtual Functions</i> - VFs), permitindo que cada VM tenha acesso direto ao hardware, enquanto a <i>Physical Function</i> (PF) gerencia os recursos compartilhados.	45
3.11	Comparação de arquitetura entre containers e VMs. A figura mostra como os containers compartilham o <i>kernel</i> do host, enquanto as VMs utilizam um <i>hypervisor</i> e requerem emulação de hardware, resultando em maior <i>overhead</i> para VMs.	46
3.12	Mapa dos componentes do OpenStack. A figura apresenta os principais serviços do OpenStack, destacando suas áreas de atuação, como <i>Compute</i> , <i>Network</i> e <i>Storage</i> , e ilustrando como esses serviços interagem para fornecer uma solução de nuvem completa.	51
3.13	Arquitetura conceitual do OpenStack. A figura apresenta os principais componentes do OpenStack e suas respectivas funções, como <i>Nova</i> para computação, <i>Neutron</i> para redes e <i>Cinder</i> para armazenamento em blocos, ilustrando a interação desses serviços em uma solução de nuvem integrada.	53
3.14	Estrutura do Nova. A figura mostra os principais subcomponentes do serviço Nova, incluindo <i>nova-api</i> , <i>nova-scheduler</i> , <i>nova-conductor</i> , <i>nova-compute</i> e <i>nova-placement</i> , e como eles interagem para gerenciar o ciclo de vida das instâncias.	55
3.15	Fluxo de trabalho do Neutron. A figura ilustra os principais componentes do Neutron, como <i>neutron-server</i> , agentes de DHCP, L3, metadados e balanceamento de carga, demonstrando como eles colaboram para fornecer serviços de rede em ambientes OpenStack.	58
3.16	Autenticação de serviços e usuários no Keystone. A figura demonstra como o Keystone gerencia a autenticação e autorização de usuários e serviços no OpenStack, emitindo tokens e verificando permissões para garantir acesso seguro aos recursos.	60
3.17	Arquitetura do Cinder. A figura apresenta os principais componentes do serviço Cinder, incluindo <i>cinder-api</i> , <i>cinder-scheduler</i> , <i>cinder-volume</i> , <i>cinder-backup</i> , e <i>cinder-client</i> , ilustrando como eles colaboram para fornecer armazenamento em bloco persistente e escalável no OpenStack.	62
3.18	Estrutura do Kubernetes. A figura apresenta os principais componentes do Kubernetes, como nós, pods, plano de controle (<i>Control Plane</i>), <i>Scheduler</i> , <i>Kubelet</i> e serviços, ilustrando como eles interagem para orquestrar aplicações distribuídas de forma eficiente.	68

4.1	Requisitos de hardware recomendados para a criação de um Nó OpenStack. A figura apresenta as especificações mínimas de CPU, memória e armazenamento necessárias para um nó OpenStack simples, com recomendações para ambientes de produção.	73
4.2	Arquitetura geral do <i>cluster</i> OpenStack no IFB, mostrando a distribuição dos nós: Controller , responsável pelos serviços essenciais; Compute , para criação de instâncias; e Storage , dedicado ao armazenamento. A figura ilustra a relação funcional e as conexões entre os componentes.	73
4.3	Containers de base do serviço OpenStack executando no nó principal do <i>cluster</i> após efetuar o <i>deploy</i> com o Kolla-ansible, mostrando de estão saudáveis e quando foram iniciados	81
4.4	Imagem retirada do terminal do nó principal do cluster	81
4.5	Interface do Horizon mostrando a configuração e gerenciamento de projetos, local onde também é configurado todos os detalhes relacionados aos projetos. .	82
4.6	Interface do Horizon mostrando usuários e contas de serviço configurados pelo Keystone.	83
4.7	Interface do Horizon mostrando as <i>roles</i> padrões do OpenStack para controle de contas de serviço.	84
4.8	Visualização da topologia de rede no <i>dashboard</i> Horizon, mostrando as conexões entre instâncias, roteadores e sub-redes.	86

Lista de Tabelas

2.1	Comparação de Ferramentas de Nuvem	25
2.2	Comparação da atividade das comunidades dos projetos open-source	28

Sumário

1	Introdução	19
1.1	Justificativa	20
1.2	Objetivos	21
2	Revisão da Literatura	23
2.1	Mapeamento Sistemático	23
2.1.1	Metodologia	23
2.1.2	Perguntas de Pesquisa	23
2.1.3	Critérios de Exclusão	24
2.2	Análise dos Artigos Seleccionados	24
2.3	Outras Motivações para a Escolha do OpenStack	28
2.3.1	Utilização em Instituições de Ensino	28
3	Fundamentação Teórica	31
3.1	Introdução a Virtualização	31
3.1.1	Anéis de segurança na arquitetura x86_64	32
3.1.2	<i>Hypervisor</i> ou VMM	33
3.1.3	<i>Hypervisor</i> Tipo 1 (Nativo ou Bare Metal)	34
3.2	Virtualizações	35
3.2.1	Virtualização Completa (<i>Full Virtualization</i>)	36
3.2.2	Paravirtualização	36
3.2.3	Virtualização Assistida por Hardware	38
3.2.4	Considerações sobre os tipos de virtualização	39
3.3	Importância das Tecnologias de Virtualização para VMs	39
3.3.1	Libvirt - API e VMs	40
3.3.2	KVM - <i>Kernel-based Virtual Machine</i>	41
3.3.3	VMX para o KVM	43
3.3.4	Alocação de CPU	44
3.3.5	Alocação de Memória	44
3.3.6	Virtualização na Placa de Rede	44
3.4	Containers	45
3.4.1	Namespaces	46
3.4.2	Cgroups	48
3.4.3	Docker	48
3.4.4	Linux Containers - LXC	49
3.5	Introdução ao OpenStack	50
3.6	Arquitetura Geral	50

3.6.1	Modelo de Serviço	51
3.6.2	Principais Componentes	51
3.7	Componentes do OpenStack	53
3.7.1	Nova (Compute)	54
3.7.2	Arquitetura do Nova	54
3.7.3	Neutron (Networking)	56
3.7.4	Keystone (Identity Service)	58
3.7.5	Interação KeyStone com outros componentes	59
3.7.6	Cinder (Block Storage)	60
3.7.7	Swift (Object Storage)	61
3.7.8	Glance (<i>Image Service</i>)	63
3.7.9	Heat (<i>Infrastructure as Code</i>)	64
3.7.10	Trove (Database as a Service)	65
3.7.11	Recapitulando	66
3.8	Orquestração de VMs	67
3.9	Orquestração de Containers	67
3.9.1	Estrutura do Kubernetes	68
4	Projeto de Implementação do OpenStack no IFB	71
4.1	Kolla	71
4.2	Hardware	72
4.3	Arquitetura Geral	73
4.4	Controller	74
4.4.1	Ambiente de Teste com VMs	74
4.4.2	Configurações do Host OpenStack	75
4.4.3	Configuração de Rede	76
4.4.4	Configuração de Rede Externa	76
4.4.5	SSH Entre os Nós	76
4.4.6	Containers do Cluster	77
4.4.7	Docker Register	77
4.4.8	Configuração Multinode	78
4.5	Segundo Nó (Storage)	79
4.6	Terceiro Nó (Compute)	80
4.7	<i>Deploy</i> da Aplicação	80
4.8	Por Dentro do OpenStack	81
4.8.1	OpenStack-Client	81
4.8.2	Projetos	82
4.8.3	Usuários	83
4.8.4	Políticas de Acesso para Serviços e Usuários	83

4.8.5	Rede	84
4.8.6	Arquitetura de Separação de Recursos para o Campus	85
4.8.7	Debug e Resolução de Problemas	86
4.9	Migração e Atualizações Futuras	87
4.9.1	Atualização de containeres	87
4.10	Análise do Cluster	88
4.11	Implementação e Utilização de Recursos	88
4.11.1	Configuração para a utilização de IaC	89
4.11.2	Criação de Instâncias para Alunos em Sala de Aula	89
4.11.3	Criação de Projetos para Alunos de Pesquisa	91
5	Conclusão	93
5.1	Panorama Geral e Viabilidade do Projeto	93
5.1.1	Principais Resultados Alcançados	93
5.2	Alinhamento com os Objetivos do Estudo	93
5.3	Manutenção, Evolução e Monitoramento	94
5.4	Trabalhos Futuros	94
	Referências	95

1

Introdução

O primeiro método de compartilhamento de recursos precedente à computação em nuvem foi o *time sharing* (Tempo Compartilhado), começou entre 1950 e 1960, quando a ideia inicial foi comprar máquinas potentes e manter rotinas de tempo compartilhado, com outros programas acessando a máquina simultaneamente, cada programa recebia um tempo e era capaz de utilizar os recursos da máquina durante o período, implementado no MIT em 1963 e utilizado até 1973 (IBM, 2024a). Em 1961, John McCarthy avançou na ideia e escreveu sobre como a força computacional deveria ser vendida como um serviço público, uma “*commodity*” (ARUTYUNOV, 2012, History Cloud), o modelo de pagamento apenas pelo uso, semelhante ao que ocorre com serviços essenciais, como água e energia elétrica, ele acrescentou que essa seria uma nova indústria de grande relevância no futuro (QIAN et al., 2009).

Com o avanço da tecnologia, a popularização do acesso à banda larga e a redução dos custos dos computadores residenciais (QIAN et al., 2009), surgiu uma nova onda de sites “.com”. O número de soluções de *SaaS* (*Software as a Service*) no mercado cresceu exponencialmente (STATS, 2018), aumentando significativamente a demanda por infraestrutura capaz de suportar esses serviços. A necessidade de adquirir hardware e montar servidores para hospedar sites, mesmo os considerados ‘simples’, fez com que a ideia de comercializar força computacional se tornasse cada vez mais atrativa. Dessa forma, o conceito de *IaaS* (*Infrastructure as a Service*) na nuvem foi se desenvolvendo, permitindo que as pessoas pagassem apenas pelo uso da infraestrutura de terceiros conforme suas necessidades, eliminando a necessidade de adquirir todo o hardware e personalizar servidores por conta própria.

A computação em nuvem evoluiu a passos lentos até a chegada da AWS em 2006. A AWS contribuiu significativamente para o aumento na quantidade de pessoas adquirindo infraestruturas no formato *IaaS* e foi uma das principais difusoras dessa tecnologia, iniciando com a venda de armazenamento como serviço **S3** (AWS, 2006). Porém, foi apenas em 2010 que tivemos a entrada de outras grandes empresas no mercado, como o Google Cloud e o Azure. Em 2008 foi criado o LXC (Linux Containers Projects) pela IBM (HILDRED, 2015), esse seria a base para a criação de novas tecnologias como o Docker que iriam ser utilizados para criação de microsserviços em nuvem.

Em 2013, o lançamento da tecnologia Docker (DOCKER, 2024) revolucionou o uso de

containeres, otimizando o gerenciamento de recursos e permitindo a execução leve e isolada de aplicações, o que levou o Google a migrar todos os microsserviços para containeres (HILDRED, 2015). Já em 2014, o Google lançou o Kubernetes, uma tecnologia fundamental para a automação e escalabilidade na orquestração de containeres, impulsionando a adoção de serviços em nuvem e acelerando a migração das infraestruturas empresariais (AWS, 2014). Desde então, o uso de serviços em nuvem tem crescido rapidamente entre as empresas [(DELTA, 2024); (DATAMATION, 2017)].

Embora a computação em nuvem pública tenha democratizado o acesso a infraestruturas escaláveis, sua dependência de provedores externos levanta preocupações estratégicas para organizações com demandas específicas de segurança como bancos. Como alternativa, surgiram as nuvens privadas, controladas internamente pelas próprias empresas, que se apresentam como uma estratégia viável para evitar a dependência de infraestrutura de terceiros, mantendo os benefícios característicos das nuvens públicas. Ao adotar uma nuvem privada, as organizações podem configurar e gerenciar sua própria infraestrutura com as mesmas arquiteturas utilizadas em nuvens públicas, garantindo a flexibilidade necessária para alocar recursos de maneira automática ou personalizada, de acordo com suas necessidades. Além disso, o controle direto sobre dados e segurança, aspectos cruciais para muitas corporações, permanece integralmente no ambiente corporativo, promovendo maior tranquilidade e redução de riscos (SUSNJARA, 2013).

Seguindo a ideia dos fornecedores de IaaS, surge um questionamento: como promover algo similar no Instituto Federal de Brasília de forma que os alunos e os demais membros da comunidade acadêmica possam utilizar a infraestrutura de maneira simples e escalável? Além disso, como a instituição pode aproveitar ao máximo o hardware remanescente, transformando-o em uma infraestrutura de nuvem que não apenas suporte as necessidades acadêmicas e administrativas, mas também fomenta estudos e pesquisas sobre essa tecnologia no campus? A criação de uma nuvem privada mantida pela própria faculdade pode ser a solução, permitindo que recursos ociosos sejam reutilizados de forma eficiente, proporcionando um ambiente seguro e de baixo custo para todos os envolvidos.

1.1 Justificativa

A evolução da computação em nuvem revolucionou a maneira como recursos computacionais são consumidos, tornando-se uma ferramenta essencial para a competitividade e eficiência de empresas modernas. Atualmente, segundo pesquisas, cerca de 90% das empresas utilizam alguma forma de serviço em nuvem, sejam eles privados, híbridos ou públicos, com benefícios econômicos significativos que podem representar até um aumento de 11.2% na receita (CLOUD, 2023, 2021). Essa adesão massiva é impulsionada pela flexibilidade, escalabilidade e redução de custos proporcionada pela computação em nuvem.

Apesar das vantagens, a dependência da infraestrutura de terceiros ao utilizar nuvens públicas apresenta desafios, especialmente como segurança e controle sobre dados sensíveis e

custos dependendo da aplicação (SUSNJARA, 2013). Por isso, a adoção de nuvens privadas, que permite às empresas configurar e gerenciar sua própria infraestrutura, tem ganhado destaque como uma alternativa estratégica. Neste contexto, surge a necessidade de analisar como instituições acadêmicas podem implementar nuvens privadas para aproveitar ao máximo seus recursos computacionais e oferecer aos alunos um ambiente prático e seguro para aprendizagem e pesquisa.

Diante desse cenário, a presente pesquisa busca justificar a implantação e implementar uma nuvem privada em um ambiente acadêmico, com foco em proporcionar um ambiente seguro e de baixo custo, aproveitando hardware ocioso e oferecendo serviços de maneira simples e escalável para os alunos. Isso possibilita não apenas o uso eficiente da infraestrutura disponível, mas também fomenta estudos e pesquisas na área de computação em nuvem, capacitando os estudantes e promovendo o desenvolvimento de novas tecnologias.

1.2 Objetivos

O objetivo geral deste projeto é propor uma arquitetura para implementação de uma nuvem privada no campus do Instituto Federal de Brasília (IFB), utilizando os recursos computacionais já existentes e gerida pela própria instituição. Essa implementação servirá como prova de conceito e permitirá que alunos, professores e funcionários tenham acesso a uma plataforma de computação escalável, segura e eficiente, capaz de atender às diversas demandas acadêmicas e administrativas.

Com a implementação desse projeto espera-se alcançar os seguintes benefícios:

- **Otimização do Uso de Recursos:** Aproveitar o hardware remanescente e ocioso da instituição, transformando-o em uma infraestrutura de nuvem capaz de suportar a execução de aplicações acadêmicas, projetos de pesquisa e atividades administrativas, reduzindo custos operacionais e aumentando a eficiência.
- **Fomento à Pesquisa e Inovação:** Criar um ambiente que facilite o estudo e a pesquisa sobre tecnologias de computação em nuvem, permitindo que os alunos desenvolvam projetos práticos utilizando a infraestrutura de nuvem privada, alinhando a teoria aprendida em sala de aula com a prática.
- **Segurança e Controle de Dados:** Garantir que os dados sensíveis da instituição sejam mantidos em um ambiente seguro e controlado, minimizando os riscos associados ao uso de nuvens públicas e atendendo às exigências de conformidade com políticas de segurança da informação.
- **Facilitação do Acesso a Serviços de Computação:** Proporcionar um ambiente de fácil acesso e uso para todos os membros da comunidade acadêmica, com a possibilidade de escalabilidade conforme as necessidades individuais de cada usuário, garantindo um serviço confiável e de alta disponibilidade.

- **Redução de Dependência de Infraestruturas Externas:** Diminuir a dependência da instituição em relação a serviços de nuvem pública, promovendo a autossuficiência e a autonomia tecnológica, o que pode resultar em economia financeira e maior controle sobre os recursos utilizados.
- **Capacitação de Alunos e Colaboradores:** Oferecer treinamentos sobre o uso da infraestrutura de nuvem privada, capacitando alunos e colaboradores a utilizarem essa tecnologia de forma eficiente, fomentando o desenvolvimento de competências que são altamente demandadas no mercado de trabalho atual.

2

Revisão da Literatura

Com o objetivo de realizar um levantamento de dados sobre o contexto do desenvolvimento de uma nuvem privada, foi aplicado o método de mapeamento sistemático, conforme as diretrizes propostas por KITCHENHAM et al. (2010).

2.1 Mapeamento Sistemático

O objetivo deste Mapeamento Sistemático é identificar qual o melhor serviço para implementar uma nuvem privada no campus do Instituto Federal de Brasília (IFB). Para responder a essa questão, será realizada uma busca por dados comparativos entre os principais serviços disponíveis atualmente, incluindo: OpenNebula, Apache CloudStack, RedHat OpenShift e OpenStack. A escolha desses serviços foi baseada em sua popularidade no GitHub, no engajamento de suas comunidades ativas e no suporte oferecido a tecnologias modernas. Esses critérios serão explorados detalhadamente ao longo do texto.

2.1.1 Metodologia

A metodologia adotada envolve a comparação de diferentes estudos e artigos que destacam as vantagens e desvantagens de cada serviço. Além disso, serão analisadas as tecnologias e funcionalidades oferecidas por cada solução, visando identificar qual delas melhor atende aos requisitos do ambiente do IFB.

2.1.2 Perguntas de Pesquisa

As seguintes perguntas de pesquisa foram formuladas para orientar este estudo:

1. **RQ1:** As instituições de ensino superior possuem nuvens privadas? Se sim, quais serviços são utilizados?
2. **RQ2:** Qual serviço oferece um ambiente mais configurável e escalável?

3. **RQ3:** Como as plataformas OpenStack, OpenNebula, Apache CloudStack e OpenShift se comparam em termos de funcionalidades, escalabilidade, segurança e suporte comunitário para atender às necessidades?

2.1.3 Critérios de Exclusão

Os seguintes critérios de exclusão foram estabelecidos para a seleção dos artigos:

- **Falta de relevância:** Artigos que não abordem diretamente a comparação entre diferentes plataformas de nuvem privada, ou que tratem de temas fora do escopo da pesquisa, como nuvens públicas ou híbridas, serão excluídos.
- **Falta de evidências empíricas:** Estudos que não apresentem dados empíricos ou práticos para suportar suas conclusões, como análises baseadas apenas em teoria ou opiniões, serão considerados irrelevantes para o propósito da comparação.
- **Data de publicação:** Artigos publicados antes de 2012 serão excluídos, visto que tecnologias de nuvem evoluíram significativamente a partir desse período, e estudos mais antigos podem não refletir o estado atual das plataformas.
- **Escopo geográfico ou institucional limitado:** Artigos focados em implementações muito específicas de nuvem privada, que não apresentem generalizações aplicáveis para o contexto acadêmico ou institucional de grande porte, serão excluídos.
- **Metodologia não definida:** Artigos que não apresentem uma metodologia clara para a comparação de plataformas, ou que falhem em descrever seus métodos de coleta de dados e análise, serão excluídos por não atenderem aos padrões de rigor científico.
- **Foco em outras tecnologias:** Artigos cujo foco principal seja em outras tecnologias de virtualização ou automação, e não diretamente em plataformas de nuvem privada como OpenStack, serão excluídos.
- **Indisponibilidade de texto completo:** Artigos cujo texto completo não esteja disponível para consulta ou que apenas apresentem resumos ou resenhas sem detalhes suficientes para uma avaliação aprofundada.

2.2 Análise dos Artigos Selecionados

VOGEL et al. (2016) apresentam uma comparação abrangente dos serviços disponíveis nas principais soluções de Infraestrutura como Serviço (*IaaS*) e Plataforma como Serviço (*PaaS*) atualmente utilizadas: OpenNebula, CloudStack e OpenStack. Adicionalmente, incluímos o OpenShift na comparação, conforme documentado em REDHAT (2024). A Tabela 2.1 detalha as funcionalidades e recursos oferecidos por cada plataforma em diferentes camadas, como

abstração de recursos, serviço principal, suporte, gerenciamento, segurança, controle e serviços de valor agregado.

Essa comparação tem como objetivo auxiliar na compreensão das diferenças e semelhanças entre essas soluções, permitindo uma avaliação mais informada na seleção da plataforma que melhor atenda às necessidades específicas de implementação, escalabilidade, segurança e gerenciamento em ambientes de nuvem.

Tabela 2.1: Comparação de Ferramentas de Nuvem

Camada	OpenNebula	CloudStack	OpenStack	OpenShift
Abstração de Recursos				
Computação	Oned	Libcloud	Nova	Kubernetes
Armazenamento	Interno	Interno	Swift/Cinder	Persistent Volumes
Volume	Interno	Interno	Nova-Volume	Kubernetes Volumes
Rede	Virtual Network Manager	Interno	Neutron/Nova Network	Kubernetes Networking
Serviço Principal				
Serviço de Identidade	IAM plugin	IAM API	Keystone	OAuth, LDAP
Agendamento	Scheduler	Interno	Nova-scheduler	Kubernetes Scheduler
Repositório de Imagens	Interno	Interno	Glance	OpenShift Registry
Cobrança e Faturamento	Interno	CloudStack Usage	Ceilometer	Não embutido
Registro	Interno	Interno	Interno	EFK Stack
Suporte				
Barramento de Mensagens	Interno/RabbitMQ	Interno/RabbitMQ	RabbitMQ	Não aplicável
Banco de Dados	SQLite/MySQL	MySQL	MySQL/Galera/MariaDB/MongoDB	etcd
Serviço de Transferência	Interno	Interno	Nova Object Store/Cinder	Não aplicável
Gerenciamento				
Gerenciamento de Recursos	Interno	Interno	Nova	Kubernetes
Continua na próxima página				

Tabela 2.1 – continuação da página anterior

Camada	OpenNebula	CloudStack	OpenStack	OpenShift
Gerenciamento de Federação	Não disponível	Não disponível	Não disponível	OpenShift Federation
Gerenciamento de Elasticidade	Auto-scaling	Elastic Load Balancing	Elastic Recheck	HPA
Gerenciamento de Usuários/Grupos	Interno	Interno	Interno	RBAC
Definição de SLA	Não disponível	Não disponível	Não disponível	Não embutido
Monitoramento	Probe/SSH/OneGate	Externo	Externo	Prometheus, Grafana
Ferramentas de Gerenciamento				
Ferramentas CLI	OpenNebula CLI	CloudMonkey	OpenStack CLI	OpenShift CLI (oc)
APIs	Nuvem pública e Plugins	Nuvem pública e Plugins	Nuvem pública e Plugins	REST APIs
Painel	Sunstone (Admin UI, User UI)	Admin UI	Horizon (Admin UI)	OpenShift Web Console
Orquestrador	OneFlow	CloudStack Cookbook	Heat	Kubernetes
Segurança				
Autenticação	Basic Auth/OpenNebula Auth/x.509/LDAP	SAML/LDAP	LDAP /Tokens/X.509/HTTPD	OAuth, LDAP
Autorização	Auth driver	SAML	Keystone	RBAC
Grupos de Segurança	Interno	Interno	Interno	Network Policies
Single Sign-On	Não disponível	Externo	Não disponível	Provedores OAuth
Monitoramento de Segurança	Externo	Externo	Externo	Compliance Operator, Falco
Controle				
Aplicação de SLA	Não disponível	Não disponível	Não disponível	Não embutido
Monitoramento de SLA	Não disponível	Não disponível	Não disponível	Não embutido
Medição	Externo	Plugin de Uso	Ceilometer	OpenShift Metering
Controle de Políticas	Não disponível	Não disponível	Não disponível	Open Policy Agent
Serviço de Notificação	Não disponível	Interno	Não disponível	Não embutido
Continua na próxima página				

Tabela 2.1 – continuação da página anterior

Camada	OpenNebula	CloudStack	OpenStack	OpenShift
Orquestração	Interno	Interno	Interno	Kubernetes
Serviços de Valor Agregado				
Zonas de Disponibilidade	Interno	Interno	Interno	Suporta multi-zonas
Alta Disponibilidade	Externo	Externo	Externo	Configurações de HA
Suporte Híbrido	Microsoft Azure	Amazon EC2/IBM	HP Helion/Amazon EC2/IBM	Implantações híbridas
Migração ao Vivo	Interno	Interno	Interno	Migração de aplicações
Suporte à Portabilidade	Não disponível	Não disponível	Não disponível	Portabilidade de contêineres
Contextualização de Imagem	One-context	Não disponível	Não disponível	Source-to-Image (S2I)
Suporte a Aplicações	Não disponível	Não disponível	Não disponível	Implantação via Kubernetes

Após a análise da tabela 2.1, percebemos que, de forma geral, o OpenStack apresenta serviços mais modularizados e com maior potencial de integração. Isso torna a solução mais configurável e escalável, o que é especialmente vantajoso, considerando que os serviços necessários em um campus voltado à pesquisa podem variar significativamente.

WEN et al. (2012) apresentam uma comparação detalhada entre as plataformas de gerenciamento de nuvem OpenStack e OpenNebula. Elas são comparadas sob diversos aspectos, incluindo origem, arquitetura, *hypervisor* entre outros detalhes. E destaca o OpenStack em relação à escalabilidade do serviço e os componentes existentes dentro dele, mas finaliza explicando que vai depender de como você vai utilizar esse serviço em nuvem para efetuar a escolha certa.

SHAHZADI et al. (2017) apresentam uma análise comparativa do desempenho de diferentes plataformas de nuvem de código aberto focadas em IaaS. O estudo examina aspectos como arquitetura, modelos de nuvem suportados, compatibilidade com *hypervisors* e linguagens de programação. A comparação destaca o desempenho e a escalabilidade de cada plataforma, oferecendo recomendações para pesquisadores sobre a escolha adequada de ferramentas de validação em ambientes de teste. A análise conclui que o OpenStack demonstra desempenho superior em cenários de teste, especialmente ao comparar implantações com o *Native OpenStack Approach*.

KUMAR et al. (2014) apresentam uma comparação abrangente entre as principais plataformas de nuvem de código aberto, incluindo OpenStack, Eucalyptus, CloudStack e OpenNebula. O estudo aborda aspectos como origem, arquitetura, compatibilidade com *hypervisors*, suporte a APIs e capacidade de migração de VMs. A análise destaca o OpenStack como uma solução

robusta e escalável, com suporte comunitário amplo e uma arquitetura modular que facilita a integração de diversos serviços. No entanto, conclui que a escolha ideal entre as plataformas depende dos requisitos específicos do ambiente, como suporte a APIs da AWS, integração com sistemas existentes e casos de uso para nuvens públicas, privadas ou híbridas.

2.3 Outras Motivações para a Escolha do OpenStack

Ao considerar uma arquitetura de nuvem privada completa, o OpenShift é comumente utilizado em conjunto com outros serviços de nuvem devido ao seu foco no provisionamento e na orquestração de contêineres. Frequentemente, ele complementa as funcionalidades das soluções de nuvem privada (IBM, 2022).

Como todos esses projetos são de código aberto (*open-source*), é importante analisar a atividade das comunidades que os mantêm. Dentre eles, o OpenStack se destaca por possuir uma comunidade extremamente ativa e numerosa.

Tabela 2.2: Comparação da atividade das comunidades dos projetos open-source

Projeto	Contribuidores	Estrelas
OpenStack	2.640	5.300
Apache CloudStack	415	2.000
OpenNebula	162	1.200
OpenShift	534	8.500

Ao comparar a comunidade do OpenStack com a do OpenNebula, percebe-se que, apesar de o OpenNebula ter sido lançado antes, em 2011, a comunidade do OpenStack já era maior e contava com grandes empresas contribuindo para seu desenvolvimento (WEN et al., 2012).

2.3.1 Utilização em Instituições de Ensino

Ao pesquisar a infraestrutura de grandes instituições de ensino, observa-se que muitas delas utilizam tecnologias de nuvem privada. Abaixo estão alguns exemplos:

- **Harvard:** Utiliza serviços específicos do OpenStack para a criação de um *Cloud Dataverse* (CROSAS, 2017).
- **Universidade da Califórnia:** Possui uma nuvem privada com OpenStack que utiliza entre 5.000 núcleos de CPU e 50 TB de armazenamento (HANUKAEV, 2024).
- **Cambridge:** Mantém uma nuvem privada com OpenStack com até 2.000 núcleos de CPU e 20 TB de armazenamento (HANUKAEV, 2024).
- **UCLouvain:** Utiliza o OpenNebula para fornecer serviços em nuvem para seus alunos (MALENGREAU, 2018).

- **USP:** Adota o Apache CloudStack para prover serviços em nuvem aos seus alunos (SIRETT, 2018).

Outro ponto que levou à escolha do OpenStack é sua superioridade em resiliência e escalabilidade quando comparado aos concorrentes (VOGEL et al., 2016). Além disso, o OpenStack apresenta uma fragmentação mais eficiente dos componentes. Cada serviço listado na Tabela 2.1 possui um componente específico dentro do OpenStack, dedicado a garantir seu funcionamento adequado (VOGEL et al., 2016).

3

Fundamentação Teórica

Neste capítulo, abordaremos tópicos fundamentais relacionados à virtualização e aos serviços de computação em nuvem de forma geral, proporcionando uma compreensão mais aprofundada de como funciona tecnologias e de qual forma são utilizadas, essencial para entender como funciona o OpenStack internamente.

3.1 Introdução a Virtualização

Para começarmos a falar sobre a virtualização, precisamos entender o que ela é, para que serve e quais são suas aplicações práticas. A virtualização nada mais é do que a criação virtual de um serviço computacional sobre hardwares e softwares reais (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016). Quando falamos de real e virtual, devemos entender que o real é o que está efetivamente executando na máquina principal, no computador raiz, enquanto o virtual será originado a partir dele, como um convidado dentro do software/hardware real. Por isso, podemos denominar de ‘Serviço Computacional Convidado’ (*Guest Computing Service*) ou, como geralmente estamos lidando com sistemas operacionais, é comum chamá-los de ‘SO Convidado’ (*Guest OS*) ou até mesmo ‘VM Convidada’ (*Guest VM*). A maneira como essa VM (Máquina Virtual) é hospedada no software principal depende de como ela vai ser criada (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016), falaremos mais sobre criação de VMs.

É fundamental entender as aplicações práticas da virtualização, já que utilizamos a todo momento. As aplicações de virtualização são diversas e abrangem diferentes aspectos da tecnologia. Vamos explorar algumas dessas aplicações como mencionadas por CHIRAMMAL; MUKHEDKAR; VETTATHU (2016):

- **Consolidação de servidores:** A virtualização permite consolidar servidores, economizando energia e espaço físico. Ao reduzir o número de servidores físicos, há também menos componentes de rede e racks, resultando em economia de recursos e maior utilização de hardware.
- **Isolamento de serviços:** A virtualização facilita o isolamento de serviços, evitando a necessidade de servidores dedicados para cada aplicação e reduzindo custos com melhor uso dos recursos disponíveis.

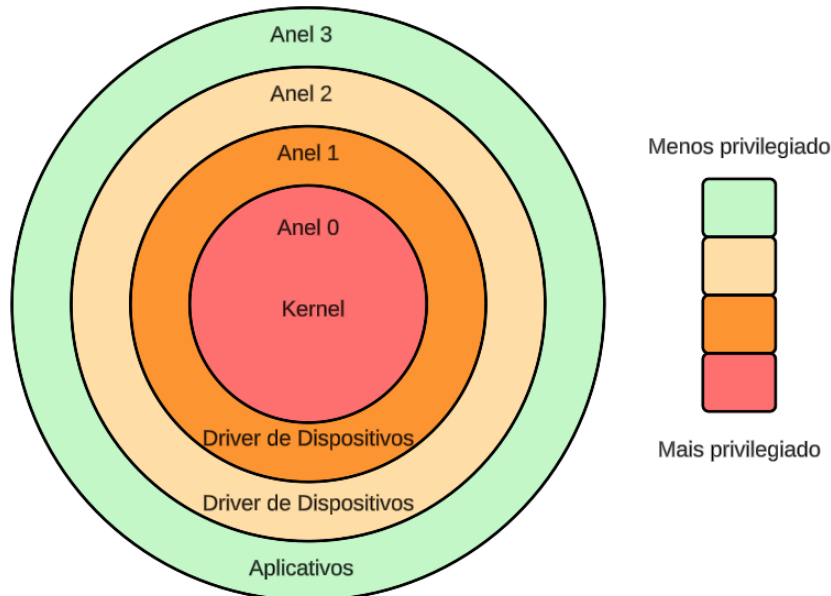
- **Provisionamento rápido de servidores:** Com virtualização, máquinas virtuais podem ser criadas rapidamente a partir de imagens ou *snapshots*, sem precisar configurar manualmente recursos físicos, agilizando o processo.
- **Recuperação de desastres:** A virtualização simplifica a recuperação de desastres, com *snapshots* atualizados que podem ser reimplantados rapidamente. Técnicas de migração de VMs aumentam a flexibilidade nesse processo.
- **Balanceamento dinâmico de carga:** A virtualização permite mover VMs sobrecarregadas para servidores subutilizados, de acordo com políticas estabelecidas, otimizando o uso dos recursos.
- **Ambiente de desenvolvimento e teste mais rápido:** A virtualização facilita a criação de ambientes de teste temporários de forma rápida e segura, com a possibilidade de restaurar rapidamente imagens em caso de problemas.
- **Maior confiabilidade e segurança:** A virtualização cria uma camada de abstração que protege o sistema contra falhas físicas e permite configurar ambientes isolados, aumentando a segurança da infraestrutura.
- **Independência do hardware:** A virtualização elimina o *lock-in* de hardware, permitindo mais flexibilidade na escolha de equipamentos e garantindo maior continuidade operacional.

3.1.1 Anéis de segurança na arquitetura x86_64

Neste projeto, vamos focar na orquestração de VMs com a arquitetura x86-64, que é a mais utilizada no mercado entre os diferentes serviços nuvem (COUTERPOINT, 2023). Essa arquitetura apresenta uma forma interessante de segurança, separando as aplicações em anéis de segurança. Ao todo, temos quatro anéis, sendo o anel 0 o mais privilegiado, onde fica o kernel, que realiza funções críticas ao hardware, e o anel 3 o menos privilegiado. Na Figura 3.1 podemos visualizar melhor como funciona a separação dos anéis, tendo o kernel direto no anel 0, drivers que precisam de acesso significativo ao hardware no anel 1 e 2 e no anel 3 as aplicações normais de usuários, que não precisam de acesso direto ao hardware.

Essa lógica de segurança em formato de anéis ajuda na virtualização e no controle dos *hypervisores*, os quais explicaremos logo em seguida. Os anéis de nível mais baixo, anel 0, têm acesso aos anéis de nível mais alto, anel 3, mas não vice-versa. O SO realiza a segmentação, um método utilizado para definir diferentes regiões da memória com níveis de privilégio distintos. No Linux e na maioria dos sistemas operacionais modernos, o que mais importa é o anel 3 (*user mode*), menos privilegiado, e o anel 0 (*kernel mode*), mais privilegiado (TANENBAUM; BOS, 2015). Os demais anéis também são utilizados para armazenar drivers que precisam de mais privilégio que o modo usuário mas menos do que o kernel.

Figura 3.1: Anéis de segurança e suas aplicações. Extraído de (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016), a figura ilustra a separação lógica dos anéis de proteção em sistemas x86. O anel mais privilegiado (anel 0) pode acessar os anéis superiores (como o anel 3), mas o inverso não é permitido — ou seja, o anel 3 não pode acessar diretamente o anel 0.



A interação entre o hardware e o sistema operacional na arquitetura x86-64 é fundamental para garantir a segurança e integridade do sistema. Cada anel de privilégio possui um conjunto específico de instruções que pode executar, e o hardware implementa mecanismos que restringem o acesso a essas instruções conforme o nível de privilégio. Por exemplo, o código que roda no anel 3, onde estão as aplicações de usuário, não pode executar diretamente instruções privilegiadas que interagem com o hardware, como manipular registros de controle ou acessar diretamente dispositivos de E/S (entrada/saída) (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016).

Além disso, o hardware utiliza tabelas de descritores de segmentos e tabelas de páginas para garantir que cada anel tenha acesso apenas às porções de memória designadas para seu nível de privilégio. Esses mecanismos previnem, por exemplo, que um código malicioso em anel 3 tente acessar diretamente áreas de memória que pertencem ao kernel ou a outras aplicações. Dessa forma, a combinação de segmentação, gerenciamento de memória e controle de instruções pelo hardware e SO forma uma barreira eficaz contra violações de segurança (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016).

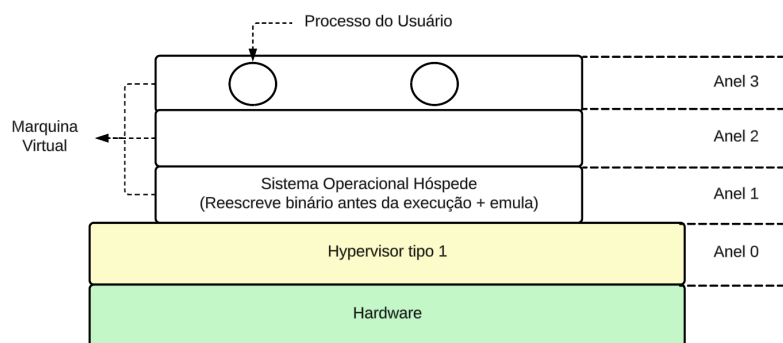
3.1.2 Hypervisor ou VMM

Ao lidarmos com máquinas virtuais (VMs), é essencial contar com um serviço que gerencie essas instâncias de maneira eficiente, permitindo que possamos iniciar, excluir e replicar nossas máquinas de forma organizada. Esse serviço é desempenhado pelo *hypervisor*, cujo papel é fornecer suporte a múltiplas cópias do hardware físico (TANENBAUM; BOS, 2015). Nos hypervisores modernos, é possível visualizar detalhes de cada VM em execução, como

logs, utilização de recursos, tráfego de rede, entre outros aspectos. Além disso, o *hypervisor* é responsável por definir como a VM será construída e como interagirá com o sistema operacional principal (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016).

O *hypervisor* desempenha um papel crucial na tradução de chamadas de sistema (syscalls). Quando uma VM executa uma função crítica, não é viável conceder acesso irrestrito ao hardware, devido a questões de escalabilidade e segurança (TANENBAUM; BOS, 2015). Para resolver esse problema, o *hypervisor* realiza modificações nas chamadas críticas, permitindo que a VM funcione como um sistema operacional principal, conforme ilustrado na Figura 3.2 onde temos os serviços separados em anéis com o *hypervisor* no anel 0 e no anel 1 o sistema operacional reescrevendo os binários antes da execução e fazendo a emulação. Dessa forma, o *hypervisor* impede que a VM tenha acesso irrestrito ao hardware. Por exemplo, quando a VM tenta acessar diretamente um recurso específico, como a memória, o *hypervisor* intercepta essa solicitação e a traduz, garantindo a segurança e o controle. Assim, a VM só consegue visualizar o espaço de memória que lhe foi alocado.

Figura 3.2: Tradução de binários em *hypervisores*. A figura ilustra a separação em anéis de proteção, com o *hypervisor* operando no anel 0 e o sistema operacional convidado no anel 1. O *hypervisor* intercepta e reescreve binários críticos antes da execução, realizando a emulação e garantindo a segurança ao evitar o acesso direto ao hardware.



Fonte: Adaptado de Modern Operating Systems (TANENBAUM; BOS, 2015).

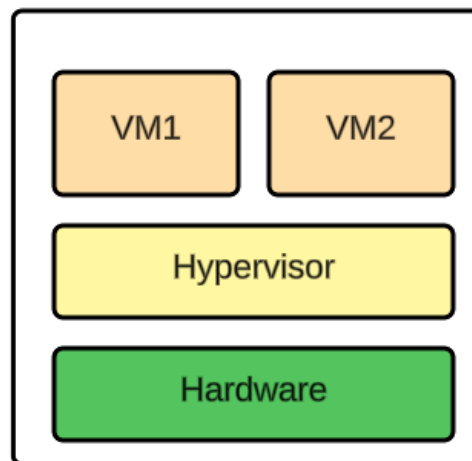
As arquiteturas de *hypervisores* também podem variar bastante de um sistema operacional para outro. No caso do Windows, o *hypervisor* mais utilizado é o Hyper-V, que também é a solução adotada pela Azure para a implementação de sua infraestrutura como serviço (IaaS). No entanto, nesta pesquisa, nosso foco será no sistema operacional Linux, devido à sua predominância no mercado de servidores (INSIGHTS, 2024) e ao fato de ser *open-source* (LEE, 2024). Especificamente, abordaremos o *hypervisor KVM (Kernel-based Virtual Machine)*, que é amplamente utilizado nos principais provedores de IaaS, como AWS e Google.

3.1.3 *Hypervisor* Tipo 1 (Nativo ou Bare Metal)

A Figura 3.3 ilustra o *hypervisor* tipo 1, que é o mais utilizado quando estamos falando de IaaS, o motivo é que ele opera junto ao hardware, as chamadas críticas são feitas diretas no

kernel do OS aumentando sua performance (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016), um exemplo notável é o *hypervisor KVM* do Linux. Ele consegue transformar kernel do Linux em um *hypervisor* fazendo com que a VM seja vista como um processo dentro do sistema, como podemos ver na imagem não temos camadas intermediárias para rodar o *hypervisor*, ele roda junto ao *hardware*.

Figura 3.3: *Hypervisor* do tipo 1. A imagem ilustra a operação direta com o hardware, onde não há camadas intermediárias entre o *hypervisor* e o *hardware*, aumentando a performance.



Fonte: Adaptado de (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016).

Perceba que no *bare metal* não temos a ideia de um *host OS*, o *hypervisor* está direto com o hardware porque é executado junto ao kernel.

***hypervisor* Tipo 2 (Hospedeiro)**

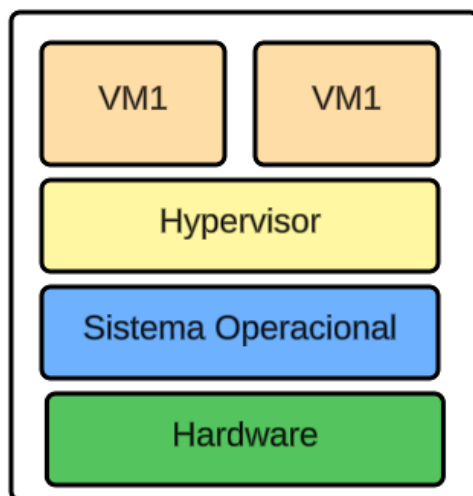
O *hypervisor* tipo 2, mostrado na figura 3.4, requer um sistema operacional host principal para funcionar, sendo útil pela facilidade de manuseio e instalação. No entanto, ele não é a melhor opção quando se necessita de alta performance e escalabilidade. Isso se deve, claramente, às camadas adicionais de software que ele introduz. Diferentemente de um *hypervisor* bare metal, o *hypervisor* hospedeiro gera uma sobrecarga maior por operar dentro de um sistema operacional, resultando em chamadas de sistema intermediadas pelo *host OS*. Essa configuração cria uma camada de abstração adicional, que pode reduzir a eficiência nas operações críticas.

Neste caso o *hypervisor* é visto de fato como um serviço a parte que está executando no sistema operacional e a VM será vista como um filho desse serviço.

3.2 Virtualizações

Existem três grandes formas de efetuar a virtualização por meio do *hypervisor*, cada uma utilizando características específicas que podem ser aplicadas em diferentes cenários.

Figura 3.4: *hypervisor* do tipo 2. A imagem mostra a necessidade de um sistema operacional hospedeiro para o funcionamento do *hypervisor*.



Fonte: Adaptado de (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016).

3.2.1 Virtualização Completa (*Full Virtualization*)

Neste método, o *hypervisor* emula o hardware para permitir que o sistema operacional convidado (*Guest OS*) opere como se estivesse em uma máquina física independente como mostrado na figura 3.5. Não há necessidade de modificações no sistema operacional convidado, pois ele não sabe que está sendo virtualizado. Praticamente todas as funções são traduzidas pelo *hypervisor*, evitando chamadas diretas da VM para o kernel do host. O *Guest OS* roda no anel 1, enquanto a máquina virtual (*Virtual Machines Monitor* - VMM) roda no anel 0 (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016).

Esse estilo de virtualização utiliza o mecanismo *Trap and Emulate* (Armadilha e Emulação). Quando o *Guest OS* tenta acessar um endereço de memória indevido, o acesso é capturado e emulado pelo *hypervisor* (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016), o que pode gerar uma sobrecarga maior no sistema.

3.2.2 Paravirtualização

Nesse caso, o *Guest OS* é modificado para interagir diretamente com o *hypervisor*, resultando em um desempenho melhor em comparação à virtualização completa. As instruções sensíveis são removidas, e, em vez disso, o sistema operacional realiza chamadas diretas ao *hypervisor* para operações como E/S e mudanças em registros críticos, similar a como programas de aplicação fazem no sistema Linux (TANENBAUM; BOS, 2015).

A figura 3.6 mostra como o SO convidado modificado realiza chamadas críticas diretamente para o *hypervisor* através de *hypercalls*. Como o SO foi adaptado para trabalhar nesse ambiente, o tratamento de chamadas sensíveis se torna mais eficiente, reduzindo a necessidade de simulação todas as instruções.

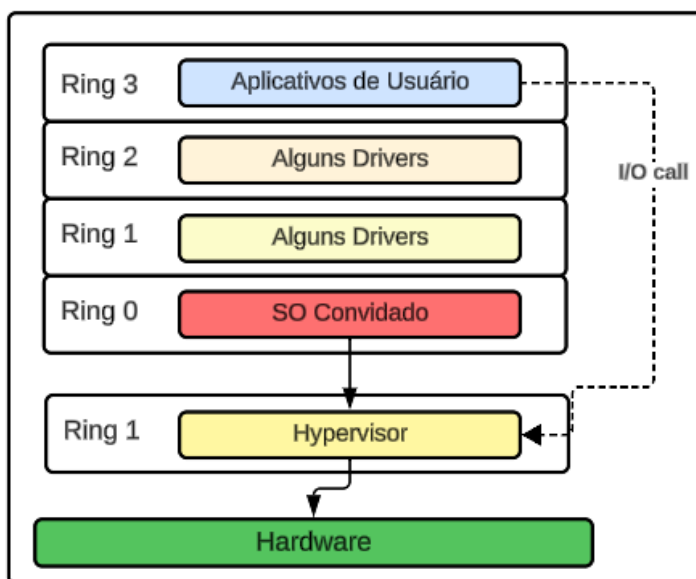
O *Guest OS* opera no anel zero, aproveita as modificações nas chamadas críticas e evitar ações indevidas para reduzir a sobrecarga tornando mais performático que a virtualização completa.

3.2.3 Virtualização Assistida por Hardware

Geralmente é alcançado uma maior performance com a virtualização assistida por hardware, que utiliza recursos nativos do processador, como Intel VT-x ou AMD-V, para executar funções de virtualização diretamente no hardware. Além disso, funções nativas do kernel Linux podem ser utilizadas para otimizar o controle das VMs (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016). Quando uma instrução sensível é executada, o hardware captura a operação e a passa ao *hypervisor* para emulação (TANENBAUM; BOS, 2015).

A figura 3.7 apresenta uma visão esquemática dos diferentes anéis de privilégio utilizados na virtualização assistida por hardware. Nela, o *Guest OS* opera no anel zero, o mesmo nível de privilégio em que um sistema operacional tradicional funcionaria em uma máquina física, enquanto o *hypervisor* assume um papel central no anel -1, criado especificamente para gerenciar diretamente o acesso ao hardware. Essa estrutura permite que o *hypervisor* intercepte chamadas sensíveis e controle os recursos de forma eficiente, enquanto o sistema convidado e suas aplicações continuam a operar como se tivessem controle completo sobre o hardware subjacente.

Figura 3.7: Virtualização assistida por hardware nos anéis de segurança. A figura mostra o *Guest OS* operando no anel 0 e o *hypervisor* no anel -1, permitindo um gerenciamento eficiente do hardware e reduzindo a necessidade de emulação por software.



Fonte: Adaptado de (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016).

A virtualização assistida por hardware se destaca pela eficiência porquê o processador fica responsável por lidar diretamente com operações sensíveis, eliminando a necessidade de

emular essas instruções no software. Essa abordagem reduz a latência associada às *traps* e minimiza o impacto no desempenho causado pela invalidação de caches e TLBs. O uso de anéis de privilégio dedicados, como o anel -1 para o *hypervisor*, permite que os sistemas convidados operem de forma isolada e com alta performance, enquanto o *hypervisor* gerencia as interações críticas com o hardware de maneira otimizada (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016).

3.2.4 Considerações sobre os tipos de virtualização

Não são apenas as funções críticas que são traduzidas pelo *hypervisor*. Utilizando a tecnologia VT (Virtualization Technology), o hardware gera uma grande quantidade de interrupções, conhecidas como *traps*. Essas *traps* podem prejudicar significativamente o desempenho do sistema, pois invalidam o cache da CPU e os TLBs (*Translation Lookaside Buffers*), que são essenciais para o rápido acesso à memória. Quando o cache e os TLBs são invalidados, a CPU precisa recarregar essas informações, o que aumenta a latência e reduz a eficiência geral. Para mitigar esses efeitos negativos, o *hypervisor* captura e traduz algumas funções não críticas. Isso é feito para reduzir a frequência das *traps* e, consequentemente, diminuir o impacto sobre o desempenho. Ao gerenciar melhor quais funções precisam ser interceptadas e traduzidas, o *hypervisor* pode manter um equilíbrio entre a funcionalidade e a eficiência do sistema (TANENBAUM; BOS, 2015).

Lembrando que essas 3 são formas gerais de virtualização, como serão feitas as traduções binárias das funções críticas a emulação entre outros detalhes vai depender do *hypervisor* que está sendo utilizado. Nenhuma instrução sensível emitida pelo sistema operacional hóspede jamais é executada diretamente pelo verdadeiro hardware. Elas são transformadas em chamadas pelo *hypervisor*, que então as emula (TANENBAUM; BOS, 2015).

Como estamos falando que computação em nuvem nosso foco vai ser o *hypervisor* do tipo 1 e com a virtualização assistida pelo hardware utilizado pelo OpenStack, o motivo dessa escolha é pelos detalhes que falamos a cima, precisamos do máximo de performance possível e alta escalabilidade, o OpenStack aceita vários modelos de *hypervisors* mas por padrão é utilizado o QEMU/KVM (OPENSTACK, 2024a).

3.3 Importância das Tecnologias de Virtualização para VMs

Tecnologias como o KVM (*Kernel-based Virtual Machine*) e ferramentas como o Libvirt proporcionam uma base robusta para a criação, gerenciamento e escalabilidade de VMs, enquanto a virtualização assistida por hardware e técnicas de alocação de recursos (CPU, memória e rede) garantem que as VMs operem com alto desempenho. Estas tecnologias são indispensáveis para qualquer infraestrutura moderna que dependa de VMs para suportar suas operações e demandas em ambientes complexos e de larga escala.

3.3.1 Libvirt - API e VMs

O *libvirt* é amplamente utilizado em diversos serviços que gerenciam máquinas virtuais (VMs), sendo uma biblioteca essencial para simplificar a criação e administração dessas VMs. Ele oferece uma interface unificada e de fácil utilização para diferentes tecnologias de virtualização, como o QEMU, que, por sua vez, utiliza o KVM para aproveitar a aceleração por hardware. Além disso, o *OpenStack*, por meio do serviço *NOVA*, utiliza o *libvirt* para a orquestração de VMs (OPENSTACK, 2024a); *libvirt guide*.

Um exemplo de criação de uma máquina virtual utilizando o *libvirt* com QEMU-KVM pode ser feito através do comando `virt-install` diretamente no terminal, passando os parâmetros de configuração desejados. Esses parâmetros são convertidos em um arquivo XML que descreve as especificações da VM. Esse arquivo, geralmente localizado em `/etc/libvirt/qemu`, pode ser editado manualmente conforme necessário. O QEMU, em conjunto com o KVM, utiliza essas informações para configurar e iniciar a máquina virtual de maneira otimizada. Por padrão, se o KVM estiver disponível no sistema, ele será utilizado para melhorar o desempenho da VM.

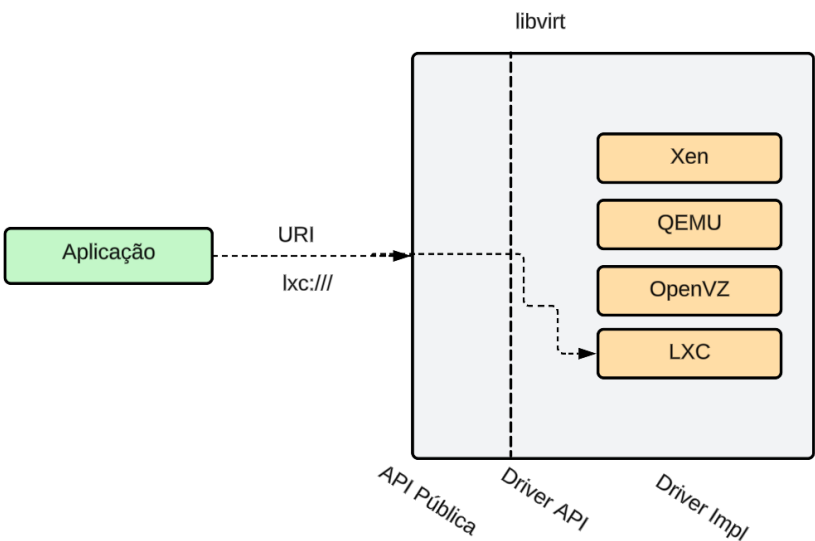
O *libvirt* é, na verdade, um conjunto de APIs chamado “Virtualization API”, que permite o gerenciamento padronizado de várias plataformas de virtualização, como KVM, QEMU, Xen, VMware, Hyper-V, e LXC. Essas APIs fornecem uma camada de abstração, permitindo que diferentes *hypervisors* sejam controlados de maneira uniforme e consistente (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016). Isso torna o *libvirt* uma ferramenta central em ambientes de virtualização complexos e escaláveis.

Como mostrado da figura 3.8, ele utiliza dos *driver* de cada *hypervisor*, os *drivers* são descobertos durante o processo de conexão com a API. Cada *driver* tem uma API de registro que carrega as referências de função específicas do *driver* para as APIs *libvirt* chamarem, assim como os respectivos protocolos utilizados por essas APIs. A arquitetura do *driver* também é usada para dar suporte a outros componentes de virtualização, como armazenamento, *pools* de armazenamento, dispositivo host, rede, interfaces de rede e filtros de rede (LIBIRT, 2024). Um exemplo simples de XML configurado para 1 vCPU utilizando o KVM e o QEMU, armazenando a imagem no arquivo do tipo QCOW2 pode ser visto em Código 1.

O código 1 contém as informações necessárias para configurar a VM, incluindo a memória, quantidade de vCPUs, imagem de carregamento, tipo de arquivo, entre outras configurações.

Naturalmente, para salvar o armazenamento das VMs, utilizamos o arquivo QCOW2 (*QEMU Copy On Write version 2*). Ele possui diversas funcionalidades que ajudam no manuseio das imagens e na escalabilidade. Sua estrutura foi desenhada pelo QEMU pensando no manuseio de VMs, por isso inclui recursos como *snapshots*, compressão, encriptação e redimensionamento dinâmico.

Figura 3.8: Interface para drivers do *libvirt*. A figura ilustra como o *libvirt* se conecta a diferentes hipervisores através de drivers específicos, permitindo o gerenciamento padronizado de componentes de virtualização, como rede, armazenamento e dispositivos.



Fonte: Adaptado de (LIBIRT, 2024).

```
1  <domain type='kvm'>
2    <name>example</name>
3    <memory unit='KiB'>1048576</memory>
4    <vcpu placement='static'>1</vcpu>
5    <os>
6      <type arch='x86_64' machine='pc-i440fx-2.9'>hvm</type>
7      <boot dev='hd' />
8    </os>
9    <devices>
10     <disk type='file' device='disk'>
11       <driver name='qemu' type='qcow2' />
12       <source file='/var/lib/libvirt/images/example.qcow2' />
13       <target dev='vda' bus='virtio' />
14       <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0' />
15     </disk>
16     <interface type='network'>
17       <mac address='52:54:00:6b:3c:58' />
18       <source network='default' />
19       <model type='virtio' />
20       <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0' />
21     </interface>
22   </devices>
23 </domain>
```

Código 1: Exemplo de configuração em XML para uma VM utilizando KVM e QEMU. O arquivo define as especificações da máquina virtual, incluindo uma vCPU, memória, armazenamento em formato QCOW2, e outras configurações essenciais para a inicialização e operação da VM.

3.3.2 KVM - Kernel-based Virtual Machine

O *Kernel-based Virtual Machine (KVM)*, como comentado acima, interage com o kernel para formar um poderoso hipervisor tipo 1 para sistemas operacionais baseados em Linux, amplamente utilizado em serviços de nuvem. Ele permite a orquestração de VMs com funções

integradas diretamente ao kernel e oferece virtualização assistida por hardware.

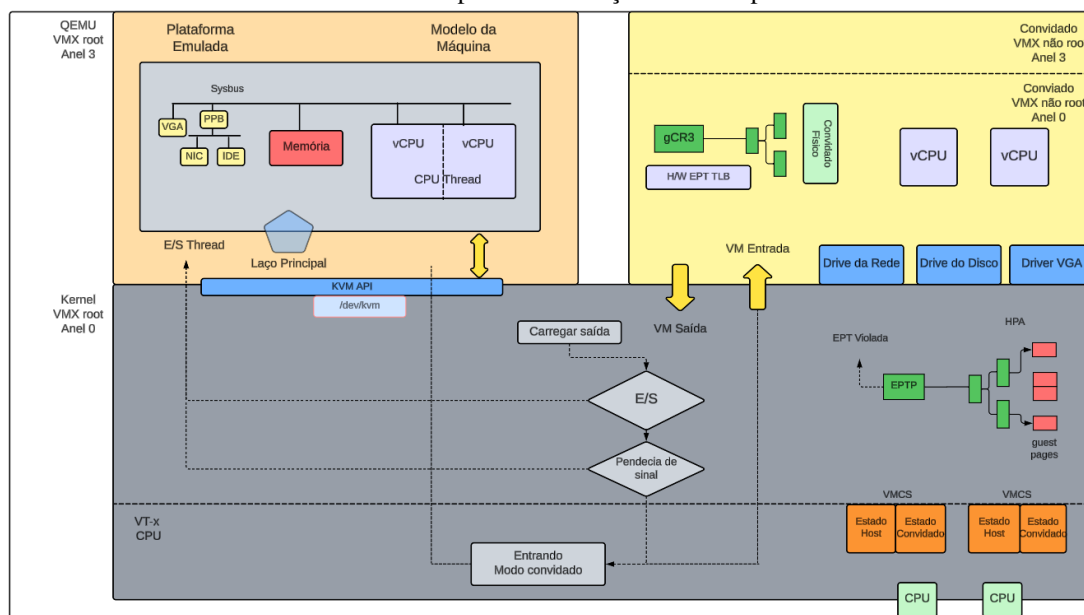
O KVM não funciona como um serviço independente, ele precisa estar integrado ao kernel do Linux e depende da assistência de hardware para virtualização (VT-X), utilizando módulos específicos como os encontrados em *kvm-intel.ko*. Além disso, o KVM também precisa do **QEMU**, que fornece os binários necessários para a construção e gerenciamento das VMs. Juntos, KVM e QEMU formam um hipervisor robusto, capaz de controlar várias VMs, oferecendo uma solução de virtualização escalável e eficiente (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016).

De forma geral, o módulo de kernel do KVM é responsável por diversas funções críticas (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016):

- **Gestão de CPUs Virtuais (vCPUs):** O KVM gerencia vCPUs, que são *threads* POSIX criadas para cada VM, permitindo a execução de instruções diretamente no kernel do sistema operacional.
- **Gerenciamento de Memória:** Utiliza *EPT (Extended Page Tables)* ou *NPT (Nested Page Tables)* para gerenciar a memória alocada para cada VM, assegurando o mapeamento eficiente entre a memória virtual da VM e a memória física do host.
- **Manuseio de Threads de I/O:** Facilita a comunicação entre dispositivos virtuais e o hardware físico, interceptando e encaminhando operações de I/O.
- **Interrupções e Exceções:** Lida com interrupções e exceções geradas por chamadas críticas, utilizando virtualização assistida por hardware para emular respostas apropriadas.
- **Controle e Monitoramento:** Expondo uma interface através de *ioctl*s, o KVM permite a configuração, controle e monitoramento das VMs.
- **Segurança e Isolamento:** Garante a segurança e o isolamento das VMs, controlando o acesso a cada espaço de memória e evitando interferências entre as VMs.

Na figura 3.9, é ilustrada a arquitetura do *hypervisor QEMU/KVM*, mostrando como o QEMU e o KVM interagem para emular uma virtualização, gerenciar vCPUs e realizar transições entre os modos do host e do convidado. Na seção superior, o modelo de máquina virtual exhibe componentes como o subsistema de E/S e os *threads* da CPU responsáveis pela execução das instruções da VM. Já a seção inferior destaca o papel do KVM, que utiliza APIs específicas e tabelas de páginas estendidas (*EPT*) para gerenciar memória e executar VMs em modo convidado, enquanto monitora eventos de saída (*VM Exit*) e lida com interrupções, entradas/saídas e violações de página. Essa integração entre QEMU e KVM resulta em uma infraestrutura robusta para virtualização assistida por hardware, assegurando desempenho e isolamento entre as VMs.

Figura 3.9: Arquitetura do *hypervisor QEMU/KVM*. A figura ilustra a interação entre o QEMU e o KVM, mostrando o gerenciamento de vCPUs, E/S e memória, além das transições entre o modo do host e do convidado para virtualização assistida por hardware.



Fonte: Adaptado de <https://wiki.qemu.org/Documentation/Architecture>.

3.3.3 VMX para o KVM

O **VMX** é substancial para o **KVM**, que utiliza as funções **VM*** disponibilizadas pela tecnologia de hardware assistido. Essas funções são usadas para o controle da VM e são disponibilizadas pelo **KVM** para que outros serviços de mais alto nível possam acessá-las através dos *ioctls* (*input/output control*) (WIKIPEDIA, 2024), que ficam em */dev/kvm*. Alguns exemplos de funções disponibilizadas pelo hardware são:

- **VMREAD**: Lê o conteúdo de um campo de controle específico da VM.
- **VMWRITE**: Escreve um valor em um campo de controle específico da VM.
- **VMCLEAR**: Limpa a estrutura da VM.
- **VMPTRLD**: Carrega o ponteiro para a estrutura da VM.
- **VMLAUNCH**: Inicia a execução da VM.
- **VMRESUME**: Retoma a execução da VM após uma interrupção.

Essas funções interagem com o KVM através de chamadas ao *ioctl*. O **KVM** fornece uma interface que os usuários podem utilizar para configurar e controlar VMs, e essa interface faz uso das funções **VMX** para manipular diretamente o estado e a execução das VMs no nível do hardware (ZABALJÁUREGUI, 2008). Existem diversas outras chamadas que são utilizadas.

Essa relação é de extrema importância para o *QEMU-KVM*, pois o *QEMU* utiliza as chamadas *ioctl*s disponibilizadas na interface do KVM para se comunicar com o hardware de maneira mais eficiente.

3.3.4 Alocação de CPU

Como comentado, uma vCPU, que é a CPU da VM, é implementada como uma *thread* POSIX. Isso significa que ela funciona como uma conexão ao CPU físico, com funções que ajudam a controlá-la. Essas *threads* são semelhantes a qualquer outra *thread* no sistema operacional *host* em termos de execução (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016).

Como cada vCPU será processada depende de como o escalonador do Linux irá tratá-la. O escalonador do Linux gerencia a alocação de tempo de CPU para todas as *threads*, incluindo aquelas que representam vCPUs. Em geral, podemos esperar uma média de 2 *threads* vCPU para cada núcleo físico da CPU, embora isso possa variar dependendo da aplicação e do hardware utilizado (TECHTARGET, 2023).

3.3.5 Alocação de Memória

A alocação de memória é uma parte crítica no processo de criação e gerenciamento de máquinas virtuais (VMs). Cada VM precisa de sua própria área de memória para operar, e o gerenciamento dessa memória é essencial para garantir que o sistema *host* possa suportar múltiplas VMs simultaneamente sem sobrecarregar os recursos disponíveis. De forma geral, cada VM possui suas próprias tabelas de páginas que mapeiam endereços de memória virtual para endereços de memória física (TANENBAUM; BOS, 2015).

No contexto de sistemas como *QEMU/KVM*, por exemplo, a alocação de memória envolve a leitura de um arquivo de configuração (como um XML) que especifica a quantidade de memória necessária para a VM. A memória especificada é então reservada no espaço de endereço do processo, e o *hypervisor* (textitKVM, neste caso) é responsável por criar as páginas de memória necessárias para traduzir os endereços de memória virtual da VM para a memória física do *host* (CHIRAMMAL; MUKHEDKAR; VETTATHU, 2016).

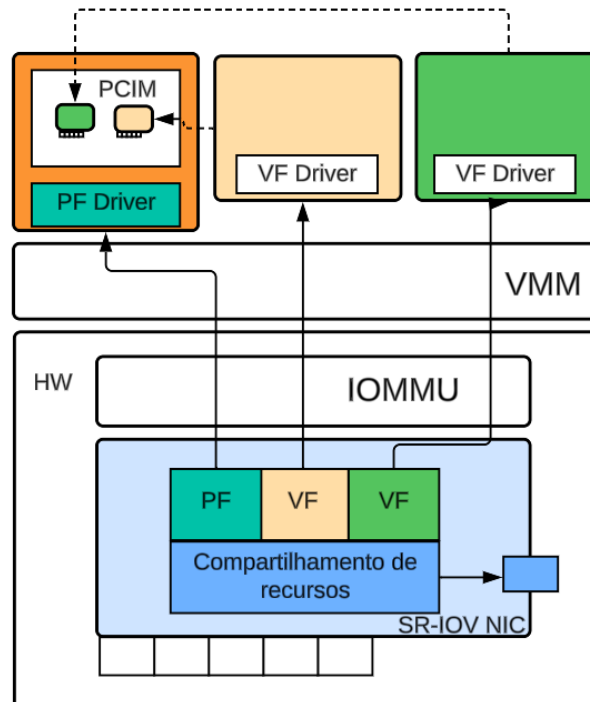
3.3.6 Virtualização na Placa de Rede

Para que todas as VMs de uma máquina tenham acesso a uma rede, precisamos de três componentes principais: *Hypervisor*, NICs virtuais (*Network Interface Controller*) e os Switches Virtuais. Cada VM possui sua própria vNIC com endereço MAC (*Media Access Control*) e IP (*Internet Protocol*) próprios.

O *switch* virtual (vSwitch) é um componente essencial que permite a comunicação entre as vNICs e a NIC real. Ele funciona como um *switch* normal, encaminhando pacotes entre as vNICs e a rede física, permitindo que todas consigam se comunicar.

O **SR-IOV** (*Single Root I/O Virtualization*) é outra tecnologia importante para a virtualização. Ele permite que uma única NIC física apresente múltiplas interfaces lógicas, como mostrado na figura 3.10. Cada interface lógica é chamada de VF (*Virtual Function*). Naturalmente, cada VF é atribuída a uma VM, permitindo acesso direto ao hardware da NIC, melhorando a performance (DONG et al., 2012), enquanto a PF (*Physical Function*) compartilha recursos com as VFs.

Figura 3.10: Arquitetura SR-IOV. A figura ilustra como uma única NIC física pode apresentar múltiplas interfaces lógicas (*Virtual Functions* - VFs), permitindo que cada VM tenha acesso direto ao hardware, enquanto a *Physical Function* (PF) gerencia os recursos compartilhados.



Fonte: Adaptado de (DONG et al., 2012).

Junto a todas essas tecnologias, utilizamos também algoritmos para efetuar o balanceamento de carga, como *Round-Robin*, *Least Connections* e *Weighted Fair Queuing*, usados para distribuir o tráfego de maneira eficiente entre várias VMs (DONG et al., 2012).

3.4 Containers

container é uma forma leve de virtualização que permite isolar processos e seus recursos em um ambiente controlado e consistente. Os containers compartilham o mesmo *kernel* do sistema operacional, mas são isolados uns dos outros e do sistema *host*, proporcionando uma maneira eficiente de executar aplicações com suas dependências em um ambiente previsível.

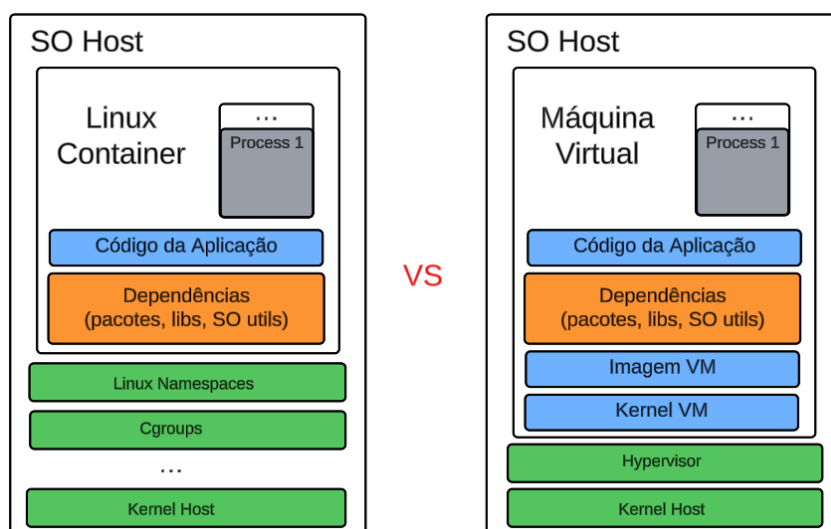
O **LXC** (*Linux Containers*) foi a primeira grande implementação dos containers como conhecemos hoje em dia (IBM, 2024a), um serviço isolado executando com uma certa limitação de hardware e de fácil criação. Esse novo sistema serviu de inspiração para o desenvolvimento

de grandes serviços usados na indústria hoje em dia, como o **Docker**.

De forma geral, os containers são criados utilizando tecnologias existentes dentro do sistema operacional. No caso dos sistemas baseados em *Unix*, utilizam-se as tecnologias de *namespaces* e *cgroups*. Iremos destacar melhor como cada uma dessas tecnologias funciona, mas, basicamente, utilizamos essas tecnologias para fazer a limitação de recursos e separação de acesso lógico.

Quando comparamos a infraestrutura necessária para executar uma VM ou um container, percebemos que, de fato, ocorre uma diminuição de *overhead*. A VM precisa de emulação de hardware, enquanto o container é apenas software (VELICHKO, 2022), como mostrado na figura 3.11. No caso dos containers, eles também compartilham o mesmo *host kernel*, o que gera maior facilidade na hora de escalar a quantidade de containers.

Figura 3.11: Comparação de arquitetura entre containers e VMs. A figura mostra como os containers compartilham o *kernel* do host, enquanto as VMs utilizam um *hypervisor* e requerem emulação de hardware, resultando em maior *overhead* para VMs.



Fonte: Adaptado de (VELICHKO, 2022).

3.4.1 Namespaces

Atualmente, o Linux implementa sete tipos diferentes de *namespaces*. O propósito de cada *namespace* é encapsular um recurso de sistema global específico em uma abstração que faz parecer aos processos dentro do *namespace* que eles têm sua própria instância isolada do recurso global. Um dos objetivos gerais dos *namespaces* é dar suporte à implementação de containers, uma ferramenta para virtualização leve que fornece a um grupo de processos a ilusão de que eles são os únicos processos no sistema (KERRISK, 2013).

Existem *namespaces* essenciais no Linux, cada um com uma tarefa importante no sistema operacional para garantir segurança e isolamento:

- **PID (Identificador de Processo):** Isola os IDs de processos. Dessa forma, processos dentro do *namespace* PID não podem ver processos de fora dele. Isso significa que processos com *namespaces* diferentes podem ter o mesmo PID, porém não têm visibilidade ou interferência uns com os outros. Funciona como um serviço hierárquico: o processo pai tem acesso ao filho, mas isso também pode ser alterado, isolando completamente o processo filho.
- **NET (Rede):** Isola interfaces de rede, roteadores, tabelas de roteamento, *firewalls* e *sockets* de rede, permitindo um isolamento completo da rede. Isso significa que cada *namespace* de rede pode ter suas próprias interfaces de rede, rotas, regras de *firewall*, etc., sem interferir nos *namespaces* de rede de outros processos. É frequentemente usado para criar ambientes de rede virtualizados, como containers com suas próprias pilhas de rede.
- **MNT (Montagem de Sistemas de Arquivos):** Isola a árvore de sistemas de arquivos montados. Dessa forma, os processos dentro de um *namespace* de montagem têm uma visão diferente dos sistemas de arquivos montados em comparação aos processos fora desse *namespace*. Isso permite ter diferentes sistemas de arquivos ou pontos de montagem visíveis para diferentes processos. Por exemplo, pode-se montar um diretório somente leitura dentro de um container sem afetar o resto do sistema.
- **UTS (Identificação do Sistema):** Isola o nome do *host* e o domínio do sistema. Isso permite que cada *namespace* tenha seu próprio nome de *host* e domínio, possibilitando a criação de ambientes isolados que podem parecer sistemas diferentes para os processos dentro deles. É útil para ambientes de teste e desenvolvimento onde diferentes configurações de nome de *host* são necessárias.
- **IPC (Comunicação entre Processos):** Isola recursos de IPC, como filas de mensagens, semáforos e memória compartilhada. Isso garante que os processos dentro de um *namespace* de IPC não possam acessar os recursos de IPC fora dele. Esse isolamento é essencial para containers e outros ambientes de virtualização, onde os recursos de IPC precisam ser isolados para garantir a segurança e a estabilidade.
- **USER (Identificação do Usuário):** Isola os IDs de usuário e grupos. Isso permite que processos dentro de um *namespace* de usuário tenham uma visão diferente dos IDs de usuário e grupos comparados aos processos fora do *namespace*. É particularmente útil para permitir que processos dentro de containers rodem como *root* dentro do container, mas como um usuário não privilegiado no sistema *host*, aumentando a segurança.
- **CGROUP (Grupos de Controle):** Isola a hierarquia de *cgroups*, usados para limitar e isolar o uso de recursos como CPU, memória, disco e rede. Cada *namespace* de

cgroup pode ter suas próprias regras e limitações, permitindo a alocação e gerenciamento de recursos de forma eficiente e segura. É uma ferramenta essencial para a gestão de recursos em ambientes de containers e virtualização, garantindo que um container não possa monopolizar os recursos do sistema.

3.4.2 Cgroups

Os *cgroups*, como mencionado anteriormente, são essenciais para o gerenciamento de recursos da máquina. Eles permitem definir quanto cada processo pode utilizar, complementando o isolamento oferecido pelos *namespaces*. Enquanto os *namespaces* oferecem isolamento lógico, os *cgroups* garantem o isolamento de recursos. Na prática, containers como Docker e LXC possuem funcionalidades adicionais, mas a base deles é o isolamento lógico e de recursos.

Podemos separar diferentes recursos existentes na máquina utilizando o PID do processo. Sempre que um novo processo é iniciado, o sistema operacional atribui um PID, essencial para a identificação no contexto apropriado. Existe um diretório padrão para separar os *cgroups*: `/sys/fs/cgroup/resource`, onde *resource* representa o recurso a ser isolado.

Para criar um *cgroup* específico, utilizamos o diretório padrão e adicionamos uma nova pasta com o nome do *cgroup*, por exemplo: `/sys/fs/cgroup/newgroup/resource`. Dentro desta pasta, configuramos como queremos limitar os recursos. Após essa configuração, basta mover o PID para dentro desse novo *cgroup* na pasta `/tasks`. Nos novos sistemas operacionais baseados em Linux, é utilizado o *cgroup v2*, que altera um pouco a organização dos diretórios; por exemplo, o PID deve ser movido para `/newgroup/cgroup.procs`. O Código 2 mostra um exemplo de como poderíamos baixar o OS Debian simples apenas com os arquivos necessários para executar o sistema e isolá-lo.

Após a criação feita no Código 2, você já pode executar o serviço isolado como um container. Os containers executados por LXC ou Docker possuem diversas outras *features* que ajudam em uma boa integração para microserviços e automatização de *deploy*, isso é apenas um exemplo.

3.4.3 Docker

O **Docker** é uma plataforma amplamente utilizada para criar, implantar e gerenciar containers, especialmente em contextos de microserviços e escalabilidade horizontal. Ele facilita a criação de ambientes isolados e consistentes para a execução de aplicativos, utilizando a tecnologia de containers junto com diversas outras funcionalidades (DOCKER, 2024). O Docker foi desenvolvido com base na tecnologia **Linux Containers (LXC)**.

Uma das principais inovações do Docker foi a implementação do conceito de imagens dentro dos containers, o que simplificou a cópia e reconstrução de ambientes. Imagens de containers são configurações pré-definidas que podem ser utilizadas para criar ambientes de execução idênticos em qualquer lugar (DOCKER, 2024). Quando uma imagem é copiada e

```
1 # Baixando o SO simplificado stable.
2 debootstrap stable path_debian http://deb.debian.org/debian/
3
4 # Acessando o bash e isolando os namespaces
5 unshare --mount --pid --net --user --map-root-user --fork --uts --ipc chroot path_debian
6 ↪ /bin/bash
7
8 # Dentro do debian faça os pontos de montagem do debian
9 mount -t proc proc /proc
10 mount -t sysfs sys /sys
11 mount -t tmpfs tmp /tmp
12
13 # Crie os arquivos de configuração do group
14 # limitando a 50% do uso da CPU
15 echo "50000 100000" | sudo tee /sys/fs/cgroup/my_cgroup/cpu.max
16
17 # limitando a 100MB de memória
18 echo "100M" | sudo tee /sys/fs/cgroup/my_cgroup/memory.max
19
20 # movendo o PID no processo /bin/bash para dentro do cgroup
21 echo <PID> | sudo tee /sys/fs/cgroup/my_cgroup/cgroup.procs
22
23 # Para visualizar se está realmente sendo limitado você pode olhar os arquivos .stat e .events
24 cat /sys/fs/cgroup/my_cgroup/memory.events
25 cat /sys/fs/cgroup/my_cgroup/cpu.stat
```

Código 2: Exemplo de script para baixar uma versão mínima do sistema operacional Debian, configurá-lo de forma isolada utilizando *cgroups* e *namespaces*, e executar processos dentro desse ambiente isolado, simulando um container.

executada, o Docker constrói um ambiente isolado que replica exatamente a configuração daquela imagem. Essa funcionalidade é extremamente útil para desenvolvedores que precisam garantir a replicação precisa de ambientes de desenvolvimento, teste ou produção.

3.4.4 Linux Containers - LXC

O **Linux Containers (LXC)** é uma tecnologia de virtualização a nível de sistema operacional que permite a criação de múltiplas instâncias isoladas de ambientes de usuários em um único *host* Linux. Enquanto o Docker popularizou o conceito de containers, o LXC é uma solução mais antiga que oferece uma abordagem semelhante, mas com algumas diferenças em termos de implementação e casos de uso (REDHAT, 2023).

O LXC funciona utilizando *namespaces* do *kernel* Linux, que fornecem isolamento para processos, além de *cgroups*, que gerenciam a limitação de recursos como CPU, memória e I/O. Isso permite que cada container LXC tenha seu próprio sistema de arquivos, rede, processos e até mesmo dispositivos. Diferente do Docker, que é mais orientado a aplicações, o LXC permite a criação de containers que podem funcionar como sistemas operacionais completos, tornando-se uma solução interessante para ambientes que exigem maior flexibilidade e controle sobre o sistema (MACHADO, 2017).

Em termos de utilização, o LXC é frequentemente gerenciado por ferramentas como o *lxd*, que fornece uma interface mais amigável e funcionalidades avançadas para administrar

containers em larga escala, incluindo suporte a redes complexas, armazenamento em disco e operações de migração ao vivo.

O LXC é extremamente útil para virtualização, mas não oferece uma experiência tão boa para os desenvolvedores. A tecnologia Docker oferece mais do que a habilidade de executar containers: ela também facilita o processo de criação e construção de containers, o envio e o controle de versão de imagens, entre outros (REDHAT, 2023).

3.5 Introdução ao OpenStack

O OpenStack é uma plataforma de código aberto composta por uma série de softwares projetados para gerenciar infraestruturas virtualizadas. Desenvolvido com o apoio de uma ampla comunidade de colaboradores, que inclui grandes empresas de tecnologia como VMWare, IBM, Cisco, Red Hat, Canonical, entre outras (AMORIM, 2015), o OpenStack se destaca por ser um projeto *open-source* altamente colaborativo.

Iniciado em 2010 pela Rackspace e NASA, o OpenStack tinha como objetivo criar uma plataforma de computação em nuvem de código aberto que fosse fácil de usar e implementar, além de interoperável entre diferentes implementações e adaptável a diversas escalas (OPENSTACK, 2024a)

Funcionando como um sistema operacional em nuvem, o OpenStack gerencia grandes conjuntos de recursos de computação, armazenamento e rede em um datacenter (OPENSTACK, 2024b). Além de sua função principal como Infraestrutura como Serviço (IaaS), o OpenStack incorpora componentes adicionais que proporcionam orquestração, gerenciamento de falhas e serviços, expandindo suas capacidades para atender às necessidades complexas de ambientes de nuvem.

3.6 Arquitetura Geral

A arquitetura do OpenStack é composta por uma coleção de serviços modulares interconectados, que em conjunto possibilitam o gerenciamento e a automação de recursos de computação, rede e armazenamento. Essa modularidade permite que os administradores personalizem a solução de acordo com as demandas específicas de seus ambientes, integrando apenas os componentes necessários.

Essa flexibilidade faz do OpenStack uma solução escalável, adequada tanto para pequenas nuvens privadas quanto para grandes data centers públicos. A Figura 3.12 ilustra como os diferentes componentes do OpenStack se integram para formar uma solução completa de infraestrutura de nuvem.

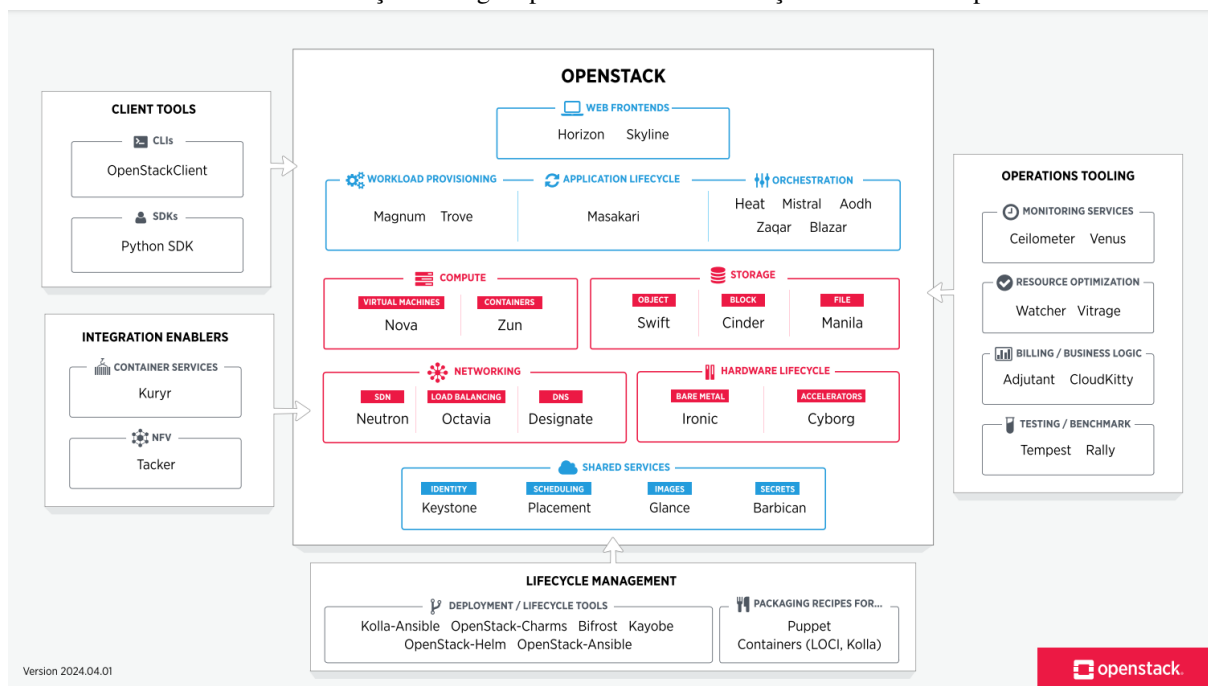
3.6.1 Modelo de Serviço

O OpenStack segue, como comentado acima, o modelo de IaaS, onde recursos de computação, armazenamento e redes são disponibilizados aos usuários sob demanda. Esses recursos são gerenciados por meio de APIs (*Application Programming Interfaces*), permitindo que os usuários criem, configurem e administrem sua infraestrutura de maneira programática.

Esse modelo de serviço oferece flexibilidade significativa, permitindo que os recursos sejam provisionados e escalados conforme necessário, sem a necessidade de intervenção manual. O OpenStack automatiza muitas das tarefas tradicionais de gerenciamento de infraestrutura, como alocação de recursos, monitoramento de desempenho e aplicação de políticas de segurança, facilitando a administração e garantindo alta disponibilidade e eficiência dos recursos.

Na Figura 3.12, é possível observar os principais componentes oferecidos pelo OpenStack e suas respectivas áreas de interação. Por exemplo, Neutron, Octavia e Designate atuam na área de *Network*, enquanto Nova e Zun estão relacionados à *Compute*, entre outros.

Figura 3.12: Mapa dos componentes do OpenStack. A figura apresenta os principais serviços do OpenStack, destacando suas áreas de atuação, como *Compute*, *Network* e *Storage*, e ilustrando como esses serviços interagem para fornecer uma solução de nuvem completa.



Fonte: <https://www.openstack.org/software>.

3.6.2 Principais Componentes

A arquitetura do OpenStack, como comentado, é composta por vários componentes interconectados, cada um desempenhando uma função específica dentro do ambiente de nuvem. A Figura 3.13 mostra a arquitetura conceitual do OpenStack, destacando como esses componentes

colaboram para fornecer uma solução integrada e escalável. A seguir, são descritos os principais componentes do OpenStack:

- **Nova (Compute):** Responsável pela criação e gerenciamento de instâncias de máquinas virtuais (VMs) e containeres. Similar ao EC2 da AWS, o Nova oferece um serviço robusto para provisionamento de instâncias, integrando-se com outros componentes como Neutron e Cinder para garantir escalabilidade e eficiência (NOVA, 2024).
- **Neutron (Networking):** O Neutron é o serviço de rede do OpenStack, fornecendo conectividade como um serviço. Ele gerencia redes, sub-redes, roteadores, balanceadores de carga e firewalls, permitindo a criação e configuração de redes virtuais que conectam as instâncias gerenciadas pelo Nova (NEUTRON, 2024).
- **Keystone (Identity Service):** O Keystone é o serviço de identidade e autenticação do OpenStack. Ele gerencia usuários e permissões, fornecendo uma estrutura centralizada de autenticação para todos os serviços do OpenStack e auxiliando na aplicação de políticas de acesso.
- **Cinder (Block Storage):** O Cinder gerencia volumes de armazenamento persistente, que podem ser anexados e desanexados de instâncias conforme necessário. Esses volumes são ideais para armazenamento de dados que requerem acesso rápido e alta durabilidade.
- **Swift (Object Storage):** O Swift é o sistema de armazenamento de objetos, projetado para armazenar e recuperar grandes quantidades de dados não estruturados. Ele armazena dados em containeres distribuídos, oferecendo uma solução escalável e resiliente, ideal para arquivos estáticos, backups e outros dados que não requerem acesso frequente, funciona muito parecido ao S3 da AWS.
- **Glance (Image Service):** O Glance gerencia as imagens de disco usadas para inicializar instâncias. Ele permite o upload, armazenamento e recuperação de imagens diretamente pelo Horizon ou por APIs, suportando diversos formatos de imagem.
- **Horizon (Dashboard):** Horizon é a interface gráfica baseada na web do OpenStack. Ela permite aos usuários e administradores gerenciar todos os serviços disponíveis de maneira intuitiva, incluindo a criação de instâncias, configuração de redes e monitoramento de recursos.
- **Heat (Orchestration):** Heat é o serviço de orquestração que automatiza o provisionamento e gerenciamento de infraestrutura usando templates YAML. Com Heat,

3.7.1 Nova (Compute)

O **Nova** é o serviço de computação do OpenStack, responsável por gerenciar o ciclo de vida das instâncias de máquinas virtuais (VMs) e containeres. Atuando como o motor de computação na nuvem, ele permite que os usuários provisionem e escalem recursos de maneira eficiente. Para provisionar uma instância funcional e acessível, o Nova interage com outros serviços do OpenStack, como Glance, Neutron, Keystone e Placement.

As principais funcionalidades do Nova incluem:

- **Provisionamento de Instâncias:** Criação de instâncias de VMs com diferentes configurações de CPU, memória e armazenamento, atendendo a diversas necessidades de computação.
- **Suporte a containeres:** Execução de containeres, proporcionando uma camada adicional de flexibilidade e eficiência.
- **Escalabilidade:** Administração de grandes quantidades de instâncias, distribuídas em diversos hosts e regiões, com alta escalabilidade.
- **Integração com Outros Componentes:** Interação transparente com serviços como Neutron para gerenciamento de rede e Cinder para armazenamento em blocos, garantindo alocação coordenada e eficiente dos recursos.

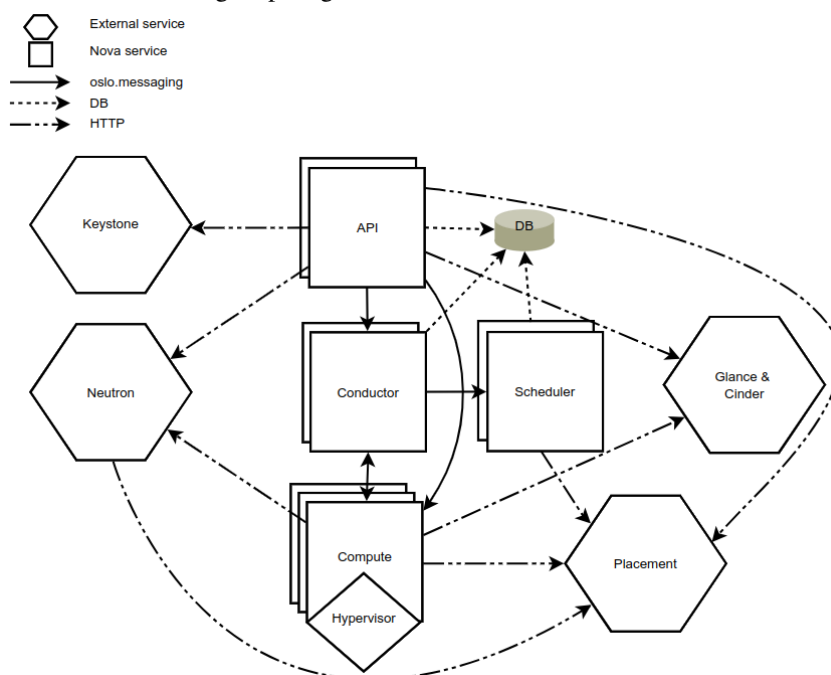
3.7.2 Arquitetura do Nova

A arquitetura do Nova é composta por vários subcomponentes que colaboram para gerenciar o ciclo de vida das instâncias (NOVA, 2024). Os principais subcomponentes incluem:

- **nova-api:** Recebe as requisições dos usuários (via API ou Dashboard) e encaminha-as para os demais subcomponentes.
- **nova-scheduler:** Decide em qual nó de computação a instância será criada, consultando o serviço Placement para verificar a disponibilidade de recursos.
- **nova-conductor:** Atua como intermediário entre o nova-compute e o banco de dados, ajudando a minimizar a carga nos nós de computação.
- **nova-compute:** Gerencia a criação, modificação e destruição das instâncias em um nó de computação, interagindo com o *hypervisor* para executar as operações necessárias.
- **nova-placement:** Mantém informações sobre os recursos disponíveis e utilizados no cluster, auxiliando o nova-scheduler na tomada de decisões.

O processo de criação de uma instância no Nova segue estas etapas:

Figura 3.14: Estrutura do Nova. A figura mostra os principais subcomponentes do serviço Nova, incluindo *nova-api*, *nova-scheduler*, *nova-conductor*, *nova-compute* e *nova-placement*, e como eles interagem para gerenciar o ciclo de vida das instâncias.



Fonte: (OPENSTACK, 2024a).

1. **Recepção da Requisição:** O usuário envia uma solicitação para criar uma nova instância através da API do OpenStack.
2. **Processamento Inicial:** O **nova-api** recebe a requisição, cria uma entrada no banco de dados e a coloca em uma fila para processamento pelo **nova-scheduler** (utilizando o RabbitMQ, um serviço popular de mensageria).
3. **Escolha do Nó de Computação:** O **nova-scheduler** lê a requisição da fila, consulta o **nova-placement** para verificar a disponibilidade de recursos e seleciona o nó mais adequado aplicando filtros e políticas.
4. **Intermediação do Conductor:** O **nova-conductor** envia as instruções necessárias ao **nova-compute** do nó selecionado.
5. **Interação com o hypervisor:** O **nova-compute** se comunica com o *hypervisor* (como QEMU-KVM) para criar a instância, configurando a rede via Neutron e anexando volumes de armazenamento via Cinder, se necessário.
6. **Inicialização da Instância:** Após a criação, a instância é iniciada, e o **nova-compute** atualiza o estado da instância no banco de dados.

Na escolha do nó de computação, o **nova-scheduler** aplica diversos filtros para garantir a alocação eficiente das instâncias. Alguns dos principais filtros incluem:

- **AggregateIoOpsFilter:** Filtra nós com base na carga de operações de I/O.
- **AggregateNumInstancesFilter:** Limita o número de instâncias por nó.
- **ComputeCapabilitiesFilter:** Avalia as capacidades de computação dos nós.
- **RamFilter:** Verifica a quantidade de memória RAM disponível.
- **DiskFilter:** Considera a disponibilidade de espaço em disco.

Além da criação de instâncias, o Nova gerencia outras operações do ciclo de vida, como:

- **Escalabilidade:** Permite a escala horizontal de instâncias, assegurando o crescimento da infraestrutura conforme necessário.
- **Migração:** Suporta a migração de instâncias entre nós de computação para manutenção ou balanceamento de carga.
- **Redimensionamento:** Permite o ajuste de recursos alocados a uma instância existente.
- **Monitoramento e Recuperação:** Monitora o estado das instâncias e realiza ações corretivas em caso de falhas.

3.7.3 Neutron (Networking)

O **Neutron** é o serviço de gerenciamento e criação de rede, responsável por fornecer conectividade como um serviço entre os componentes do ambiente de nuvem. Ele permite que os usuários criem e gerenciem redes virtuais, sub-redes, roteadores, balanceadores de carga, firewalls e outros recursos de rede de maneira programática, garantindo que as instâncias possam se comunicar entre si e com redes externas.

O Neutron oferece uma ampla gama de funcionalidades que permitem a configuração e gerenciamento de redes de forma flexível e escalável:

- **Criação de Redes Virtuais:** Permite a criação de redes virtuais que conectam as instâncias de VMs, proporcionando isolamento entre diferentes grupos de usuários ou projetos.
- **Gerenciamento de Sub-redes:** Facilita a criação de sub-redes dentro das redes virtuais, com suporte para alocação automática de endereços IP (DHCP) e configuração de gateway.
- **Roteamento e NAT:** Suporta a configuração de roteadores para interconectar sub-redes e fornecer acesso externo às instâncias, incluindo suporte para NAT (Network Address Translation).

- **Segurança de Rede:** Inclui a configuração de firewalls e grupos de segurança que controlam o tráfego de entrada e saída da rede, garantindo um ambiente seguro.
- **Balanceamento de Carga:** Oferece serviços de balanceamento de carga para distribuir o tráfego entre múltiplas instâncias, melhorando a disponibilidade e o desempenho dos aplicativos.

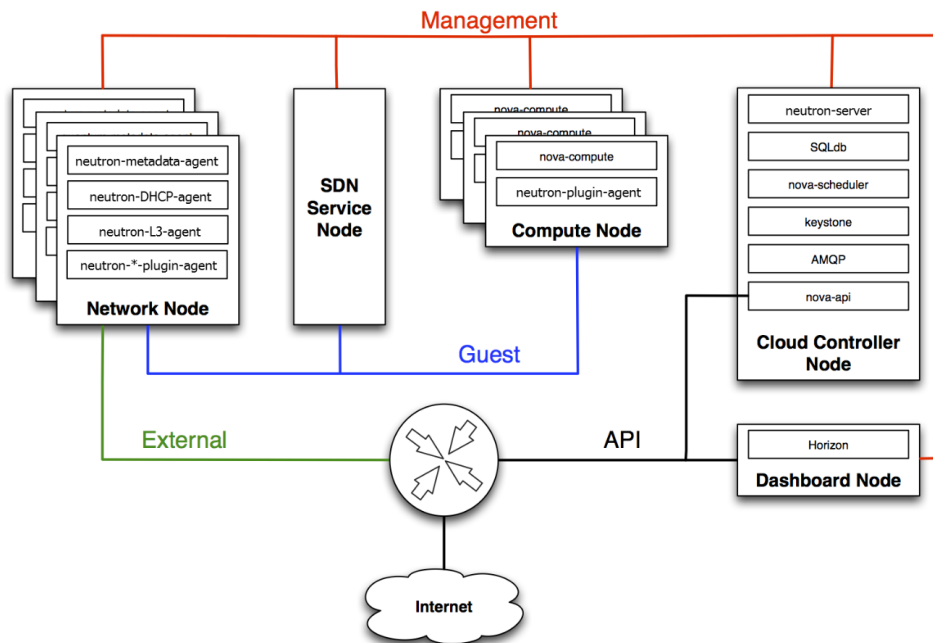
Assim como o Nova a arquitetura do Neutron é composta por diversos componentes que colaboram para gerenciar a rede em um ambiente OpenStack. Os principais componentes incluem:

- **neutron-server:** Componente central que expõe as APIs do Neutron e orquestra as operações de rede, interagindo com o banco de dados e os agentes de rede.
- **neutron-plugin:** Abstrai a interação com diferentes tecnologias de rede (como Open vSwitch, Linux Bridge, etc.), permitindo que o Neutron seja compatível com uma variedade de backends.
- **neutron-dhcp-agent:** Gerencia os serviços DHCP para as sub-redes, garantindo que as instâncias recebam endereços IP automaticamente.
- **neutron-l3-agent:** Responsável pelo roteamento de tráfego entre sub-redes e entre a rede interna e externa, implementando funcionalidades de roteamento e NAT.
- **neutron-metadata-agent:** Facilita o acesso das instâncias aos metadados necessários para sua configuração, como as chaves SSH ou dados de inicialização.
- **neutron-lbaas-agent:** Gerencia as operações de balanceamento de carga, criando e configurando balanceadores de carga conforme necessário.

O processo de criação de uma rede e suas operações no Neutron envolvem várias etapas, que podem ser descritas da seguinte maneira:

1. **Criação de Rede:** O usuário solicita a criação de uma rede virtual via API do Neutron, que é processada pelo **neutron-server**.
2. **Configuração de Sub-rede:** O usuário define uma sub-rede dentro da rede criada, configurando parâmetros como o intervalo de endereços IP, gateway e servidor DHCP.
3. **Configuração de Segurança:** O usuário configura grupos de segurança e regras de firewall para controlar o tráfego permitido a conexão para as instâncias conectadas à rede.
4. **Roteamento e NAT:** Se necessário, o usuário cria um roteador para interligar sub-redes ou conectar a rede à internet, configurando NAT para que as instâncias possam acessar redes externas.

Figura 3.15: Fluxo de trabalho do Neutron. A figura ilustra os principais componentes do Neutron, como *neutron-server*, agentes de DHCP, L3, metadados e balanceamento de carga, demonstrando como eles colaboram para fornecer serviços de rede em ambientes OpenStack.



Fonte: (OPENSTACK, 2024a).

5. **Balanceamento de Carga:** Para distribuir o tráfego entre várias instâncias, o usuário pode configurar um serviço de balanceamento de carga, que será gerenciado pelo **neutron-lbaas-agent**. Não é necessário criar diretamente um balanceador de carga

A maneira como ela vai ser configurada dentro do OpenStack vai depender totalmente da necessidade do projeto.

3.7.4 Keystone (Identity Service)

O **Keystone** é o serviço de identidade do OpenStack, responsável por autenticar e autorizar usuários e serviços que interagem dentro da nuvem. Ele desempenha um papel central na segurança e no gerenciamento de acesso, garantindo que apenas usuários e serviços autenticados possam acessar os recursos disponíveis no OpenStack.

O Keystone oferece várias funcionalidades essenciais para a verificação de identidade e controle de acesso:

- **Autenticação e Autorização:** O Keystone valida as credenciais dos usuários e serviços, emitindo tokens que são utilizados para autorizar o acesso aos outros componentes do OpenStack. Esses tokens carregam informações sobre as permissões do usuário ou serviço, garantindo que eles possam acessar apenas os recursos para os quais têm permissão.

- **Gerenciamento de Identidades:** O Keystone gerencia os usuários, grupos, projetos e domínios dentro do OpenStack. Ele permite a criação e atribuição de papéis (*Roles*), que determinam o nível de acesso e as permissões que cada usuário ou serviço tem dentro de um projeto ou domínio específico.
- **Serviço de Catálogo:** O Keystone mantém um catálogo dos serviços disponíveis no OpenStack, dentro do Horizon podemos ver as respectivas APIs e endpoints. Isso permite que os usuários e serviços descubram e interajam com outros componentes de maneira centralizada e eficiente.

O Keystone funciona como uma camada de segurança intermediária que se integra com todos os outros serviços do OpenStack. Quando um usuário ou serviço precisa acessar um recurso, ele primeiro se autentica no Keystone, que verifica as credenciais fornecidas (por exemplo, nome de usuário e senha, ou token). Se a autenticação for bem-sucedida, o Keystone emite um token que o usuário ou serviço pode usar para acessar outros componentes, como Nova, Neutron, ou Glance.

Cada vez que uma solicitação é feita a um serviço do OpenStack, o token é enviado como parte da requisição. O serviço verifica o token com o Keystone para garantir que ele seja válido e que o usuário ou serviço tenha as permissões necessárias para realizar a operação solicitada. Se o token for válido e as permissões forem adequadas, o serviço processa a requisição.

3.7.5 Interação KeyStone com outros componentes

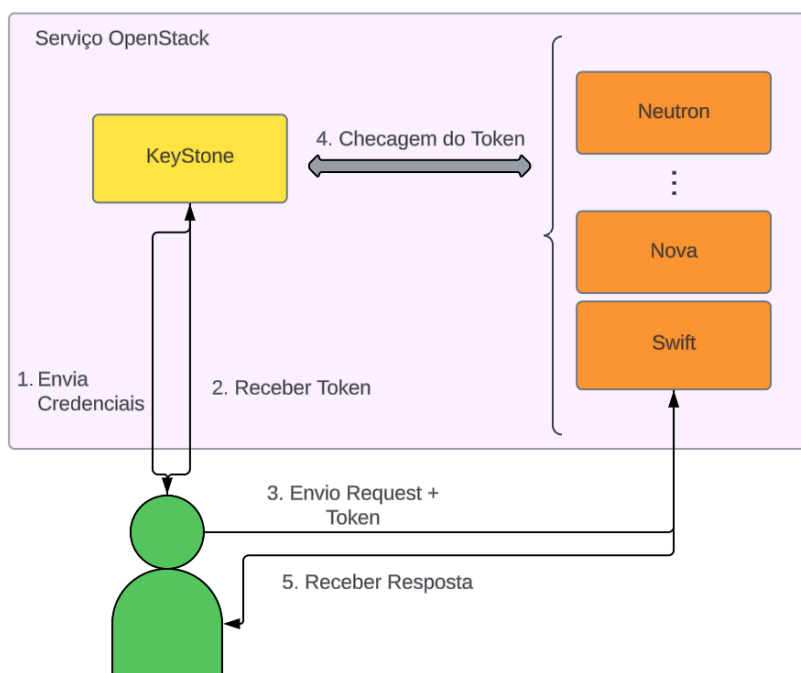
O Keystone é essencial para a operação segura e eficiente de toda a infraestrutura do OpenStack. Todos os serviços, como Nova, Neutron, Glance, Cinder, e Swift, dependem do Keystone para autenticação e autorização. Quando um serviço precisa verificar as permissões de um usuário, ele consulta o Keystone, o que garante que todas as operações dentro do OpenStack estejam protegidas por um controle de acesso robusto (DA SILVA et al., 2018).

Além disso, o catálogo de serviços mantido pelo Keystone é fundamental para que os usuários e serviços descubram quais APIs e endpoints estão disponíveis, facilitando a comunicação e interação dentro do ambiente de nuvem.

Por fim, a capacidade do Keystone de integrar-se com sistemas de autenticação externos, através de federação e SSO, torna-o altamente versátil e adaptável a diferentes ambientes corporativos, garantindo uma experiência de usuário coesa e segura em organizações de todos os tamanhos.

Este papel central do Keystone na segurança e no gerenciamento de acesso faz dele um componente crítico no ecossistema OpenStack, assegurando que apenas usuários e serviços autorizados possam operar dentro da nuvem.

Figura 3.16: Autenticação de serviços e usuários no Keystone. A figura demonstra como o Keystone gerencia a autenticação e autorização de usuários e serviços no OpenStack, emitindo tokens e verificando permissões para garantir acesso seguro aos recursos.



Fonte: Adaptado de (DA SILVA et al., 2018).

3.7.6 Cinder (Block Storage)

O **Cinder** é o serviço de armazenamento em bloco, responsável por gerenciar volumes de armazenamento que podem ser anexados a instâncias de máquinas virtuais. Ele fornece uma solução flexível e escalável para armazenar dados de maneira persistente, permitindo que os volumes sejam criados, gerenciados e conectados ou desconectados de instâncias conforme necessário (ROSADO; BERNARDINO, 2014).

O armazenamento em bloco divide os dados em blocos e os armazena em partes separadas cada um com identificador exclusivo, dentro do sistema cada bloco é tratado como um disco individual, utilizamos dele para dar armazenamento a instância. Essa forma deixa fácil a duplicação do armazenamento para cópias ou recuperar dados (IBM, 2024b).

As principais funcionalidades do Cinder incluem:

- **Criação e Gerenciamento de Volumes:** Permite que os usuários criem volumes de armazenamento que podem ser anexados a instâncias para uso como discos adicionais. Esses volumes podem ser criados a partir de imagens, snapshots, ou de forma vazia.
- **Snapshots:** O Cinder suporta a criação de snapshots de volumes, permitindo que os dados sejam salvos em um ponto no tempo para backup ou clonagem futura.
- **Tipos de Armazenamento:** Oferece a possibilidade de definir diferentes tipos de armazenamento, permitindo a escolha entre diversos backends de armazenamento

com diferentes características de desempenho e custo.

- **Expansão de Volumes:** Volumes existentes podem ser expandidos conforme necessário, sem a necessidade de desconectar o volume da instância.
- **Conectividade Multi-backend:** Suporta múltiplos backends de armazenamento, como LVM, NFS, Ceph, e outros, proporcionando flexibilidade na escolha do hardware e arquitetura de armazenamento.

A arquitetura geral do Cinder se divide em:

- **Cinder-API:** Exposição da interface RESTful para gerenciamento de volumes.
- **Cinder-Scheduler:** Responsável por determinar qual backend de armazenamento deve ser usado para criar um volume.
- **Cinder-Volume:** Gerenciamento direto de volumes nos backends de armazenamento.
- **Cinder-Backup:** Criação e gerenciamento de backups de volumes em armazenamento secundário.
- **Cinder-Client:** Ferramenta de linha de comando para interagir com a API do Cinder.

O Cinder é crucial para aplicações que exigem armazenamento de dados persistente e de alto desempenho, como bancos de dados e sistemas de arquivos distribuídos. Sua capacidade de integrar-se com diversos tipos de armazenamento e fornecer snapshots e backups automáticos torna-o um componente vital para a infraestrutura de nuvem do OpenStack por se ligar as instâncias (CINDER, 2024).

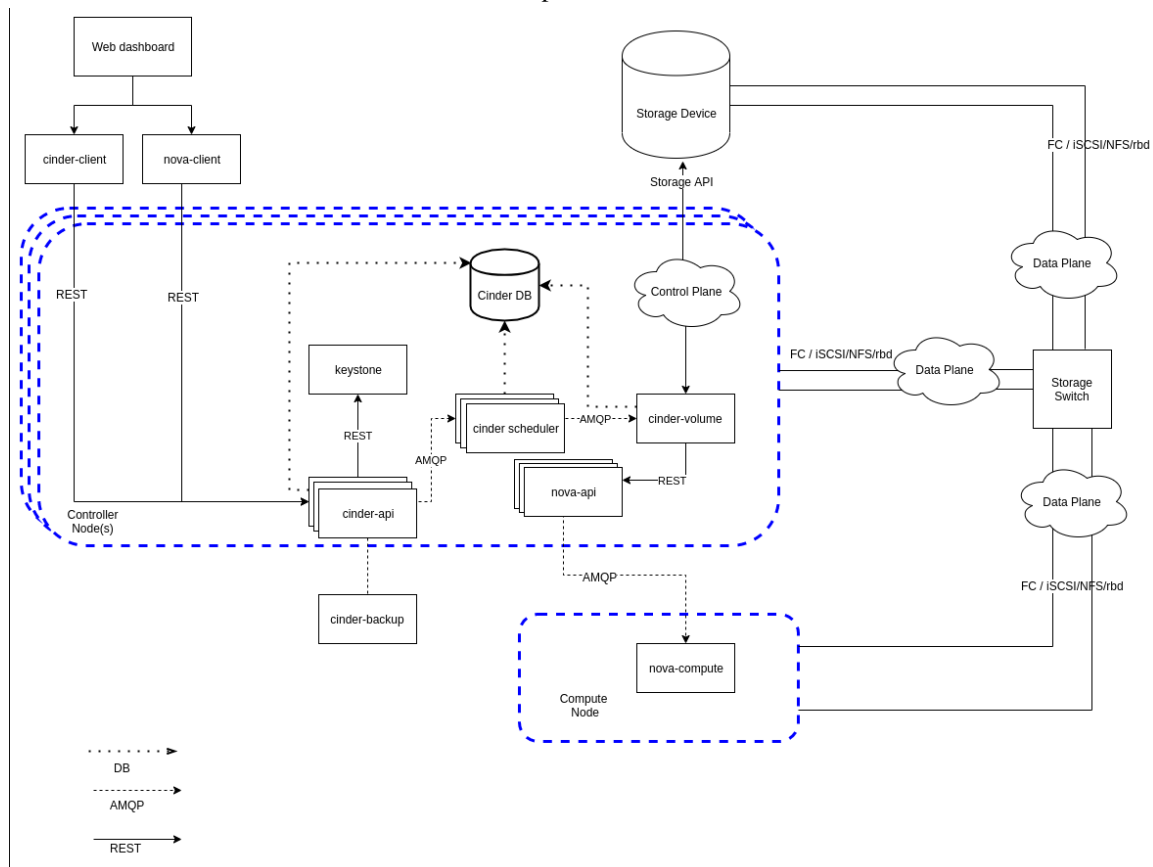
3.7.7 Swift (Object Storage)

O **Swift** é o serviço de armazenamento de objetos, projetado para armazenar e recuperar grandes volumes de dados não estruturados de forma escalável e redundante. Diferente do armazenamento em bloco, o Swift armazena dados como objetos, que são agrupados em containeres e acessados via uma API RESTful, sem a necessidade de gerenciar um sistema de arquivos tradicional (SWITFT, 2024), segue a mesma ideia do S3 AWS, assim você consegue salvar todos os arquivos e acessar eles pelo ID de forma eficiente via API.

O armazenamento de objetos encapsula dados em unidades chamadas objetos, que incluem tanto os dados quanto seus metadados e atributos. Essa abordagem permite um gerenciamento mais flexível e escalável dos dados, distribuindo a carga de trabalho entre dispositivos de armazenamento inteligentes, conhecidos como *Object-based Storage Devices (OSDs)*, que oferecem alta performance e segurança em ambientes de computação distribuída" (PANASAS, 2007).

As principais funcionalidades do Swift incluem:

Figura 3.17: Arquitetura do Cinder. A figura apresenta os principais componentes do serviço Cinder, incluindo *cinder-api*, *cinder-scheduler*, *cinder-volume*, *cinder-backup*, e *cinder-client*, ilustrando como eles colaboram para fornecer armazenamento em bloco persistente e escalável no OpenStack.



Fonte: (CINDER, 2024).

- **Armazenamento Escalável de Objetos:** Permite armazenar uma quantidade ilimitada de dados distribuídos entre múltiplos servidores, garantindo escalabilidade horizontal.
- **Redundância e Replicação:** Implementa replicação automática dos dados para assegurar alta disponibilidade e durabilidade, mesmo em caso de falhas de hardware.
- **Acesso via API RESTful:** Os dados podem ser acessados e gerenciados através de uma API RESTful, facilitando a integração com aplicações e serviços web.
- **Gerenciamento de containeres:** Organiza os objetos em containeres, permitindo que os usuários agrupem dados logicamente e definam políticas de acesso específicas.
- **Versionamento e Auditoria:** Suporta versionamento de objetos e auditoria de acessos, garantindo controle sobre as mudanças e acesso aos dados.

O Swift é ideal para armazenar grandes quantidades de dados não estruturados, como backups, arquivos multimídia, e logs. Sua arquitetura distribuída e resiliente o torna uma escolha popular para ambientes que exigem armazenamento de longa duração e alta disponibilidade.

3.7.8 Glance (*Image Service*)

O **Glance** é o serviço de gerenciamento de imagens, projetado para armazenar, descobrir e recuperar imagens de disco de máquinas virtuais. Ele permite que os usuários façam upload de imagens de sistemas operacionais e outros discos, que podem ser usados para instanciar novas máquinas virtuais. As imagens podem ser armazenadas em uma variedade de backends, como Swift, Ceph, ou em sistemas de arquivos locais.

As principais funcionalidades do Glance incluem:

- **Gerenciamento de Imagens:** Facilita o upload, armazenamento e compartilhamento de imagens de disco, que podem ser usadas para criar novas instâncias de VMs.
- **Conversão de Formatos:** Glance pode converter imagens entre diferentes formatos de disco, proporcionando maior compatibilidade e flexibilidade na utilização das imagens.

O Glance é essencial para o provisionamento de instâncias no OpenStack, permitindo que as imagens de sistemas operacionais e discos sejam gerenciadas de forma centralizada e eficiente. Sua integração com outros componentes do OpenStack, como Nova e Cinder, torna-o um elemento crucial para a operação e escalabilidade da nuvem.

3.7.9 Heat (*Infrastructure as Code*)

O **Heat** é o serviço de orquestração de recursos, projetado para permitir o gerenciamento automatizado e coordenado de infraestruturas complexas. Utilizando templates escritos em YAML, o Heat facilita a criação e o gerenciamento de pilhas de recursos, como instâncias de computação, volumes de armazenamento, redes e outros serviços do OpenStack, de forma declarativa e repetível (HEAT, 2024).

O Heat permite que orquestremos toda a infraestrutura de uma organização específica com templates, aceita também o Terraform e o CloudFormation, dois serviços famosos de IaC, a ideia de utilizar esses templates é basicamente configurar um ambiente e não precisar ficar refazendo ele caso necessário, você tem todo controle no template e sabe exatamente quais passos foram feitos para construção ou alocação de determinado recurso.

As principais funcionalidades do Heat incluem:

- **Orquestração de Pilhas de Recursos:** Gerencia a criação, atualização e exclusão de conjuntos de recursos interdependentes como uma única unidade, chamada de pilha (*stack*).
- **Infraestrutura como Código:** Os usuários podem definir a infraestrutura em templates YAML, que descrevem os recursos e as relações entre eles, permitindo a automação de operações e a reprodutibilidade dos ambientes.
- **Automação de Provisionamento:** Suporta a automação de tarefas repetitivas e complexas, como o escalonamento de instâncias de computação e o provisionamento de redes e volumes.
- **Suporte a Vários Templates:** Além dos templates nativos do Heat (HOT - Heat Orchestration Template), também suporta templates no formato AWS CloudFormation, proporcionando flexibilidade na definição dos recursos.

Arquitetura geral Heat:

- **Heat-engine:** É o componente central responsável pelo gerenciamento das pilhas de recursos (*stacks*). O Heat Engine interpreta os templates, cria e gerencia os recursos descritos nos templates, e lida com as operações de ciclo de vida das pilhas, como criação, atualização e exclusão.
- **Heat-api:** Fornece uma interface RESTful que permite aos usuários interagir com o serviço Heat. Através da Heat API, os usuários podem enviar templates, criar e gerenciar pilhas, e consultar o estado das pilhas e recursos.
- **Heat-api-cfn:** Implementa uma API compatível com AWS CloudFormation, permitindo que os usuários usem templates no formato AWS CloudFormation para definir

e gerenciar seus recursos. Isso oferece uma compatibilidade adicional para aqueles que estão familiarizados com a infraestrutura como código.

- **Python-heatclient:** É a ferramenta de linha de comando que permite interagir com o serviço Heat. Os usuários podem usar o Heat Client para executar comandos que criam, atualizam, deletam e listam pilhas e recursos no OpenStack.

O Heat é ideal para automatizar a implementação e o gerenciamento de infraestruturas complexas e repetitivas em ambientes de nuvem, garantindo consistência, escalabilidade e eficiência operacionais. Sua capacidade de orquestrar diversos serviços do OpenStack em conjunto torna-o uma ferramenta essencial para DevOps e equipes de operações que buscam simplificar a gestão de infraestrutura na nuvem.

3.7.10 Trove (Database as a Service)

O **Trove** é o serviço de banco de dados como serviço (DBaaS) do OpenStack, projetado para simplificar o gerenciamento e a operação de bancos de dados relacionais e não relacionais em ambientes de nuvem. O Trove permite aos usuários provisionar, configurar, operar e escalar instâncias de banco de dados com facilidade, utilizando uma interface unificada e integrada ao ecossistema OpenStack (TROVE, 2024).

O Trove possibilita o gerenciamento de múltiplos bancos de dados através de uma API RESTful, oferecendo suporte para diversos tipos de bancos de dados, como MySQL, PostgreSQL, MongoDB, entre outros. Isso permite que as organizações mantenham uma infraestrutura de banco de dados flexível e escalável, sem a complexidade tradicional associada à administração de bancos de dados em larga escala.

As principais funcionalidades do Trove incluem:

- **Provisionamento Automatizado de Bancos de Dados:** Facilita a criação de instâncias de banco de dados de forma automatizada, permitindo o provisionamento rápido e eficiente de bancos de dados para diferentes aplicações.
- **Gestão Simplificada:** Oferece ferramentas para gerenciar facilmente backups, restaurações, atualizações de software e monitoramento de desempenho, simplificando a administração de bancos de dados.
- **Suporte a Múltiplos Motores de Banco de Dados:** O Trove suporta uma variedade de motores de banco de dados, tanto relacionais quanto não relacionais, proporcionando flexibilidade para diferentes necessidades de aplicação.
- **Escalabilidade:** Permite a escalabilidade vertical e horizontal das instâncias de banco de dados, garantindo que os recursos possam ser ajustados conforme a demanda aumenta.

Arquitetura geral do Trove:

- **Trove-taskmanager:** Responsável por gerenciar as operações assíncronas no Trove, como a criação de instâncias, backups e restaurações. Ele coordena essas tarefas em segundo plano para garantir que sejam executadas de forma eficiente.
- **Trove-api:** Fornece a interface RESTful através da qual os usuários podem interagir com o serviço Trove. Através da Trove API, os usuários podem criar, configurar e gerenciar instâncias de banco de dados.
- **Trove-conductor:** Atua como um intermediário entre o Trove-taskmanager e o banco de dados de controle do OpenStack, garantindo a integridade e a sincronização dos dados de gerenciamento.
- **Trove-guestagent:** Instalado dentro de cada instância de banco de dados provisionada pelo Trove, o Trove-guestagent é responsável por executar as operações específicas do banco de dados, como backups, restaurações e mudanças de configuração, comunicando-se com o Trove-taskmanager para realizar essas tarefas.
- **Python-troveclient:** É a ferramenta de linha de comando que permite aos usuários interagir diretamente com o serviço Trove. O Trove Client permite a execução de comandos para criar, gerenciar e excluir instâncias de banco de dados.

O Trove é ideal para organizações que desejam simplificar o gerenciamento de bancos de dados na nuvem, permitindo uma administração eficiente e escalável. Com o Trove, as equipes de operações podem automatizar o provisionamento e a manutenção de bancos de dados, garantindo um desempenho consistente e reduzindo a complexidade operacional.

3.7.11 Recapitulando

Como vimos, o OpenStack é composto por diversos componentes que interagem entre si para formar uma infraestrutura como serviço (IaaS) robusta e escalável. Exploramos em detalhe alguns dos componentes mais importantes, como Nova, Neutron e Keystone, que são fundamentais para o funcionamento do OpenStack e serão amplamente utilizados na implementação do nosso projeto.

O OpenStack continua a evoluir, ganhando popularidade e recebendo novas funcionalidades a cada ano. Por essa razão, a análise dos componentes foi abordada de forma geral, destacando aspectos arquiteturais e operacionais essenciais. É importante ressaltar que, embora tenhamos focado nos componentes mais amplamente utilizados, o OpenStack oferece uma gama ainda maior de serviços que podem ser personalizados para atender a necessidades específicas.

3.8 Orquestração de VMs

A orquestração de VMs torna-se essencial em ambientes onde a quantidade de instâncias supera a capacidade de gerenciamento manual. Inicialmente, quando lidamos com poucas instâncias, o uso de um *hypervisor* para controlar as máquinas virtuais pode ser suficiente. Um *Virtual Machine Monitor (VMM)* pode monitorar logs, verificar o uso de recursos e reiniciar instâncias quando necessário. Entretanto, à medida que a infraestrutura se expande, o gerenciamento de dezenas ou centenas de VMs se torna inviável sem uma camada de orquestração.

Aqui entra o OpenStack, que proporciona uma plataforma escalável e flexível para gerenciar grandes quantidades de recursos computacionais. Ele automatiza tarefas como a criação de novas instâncias, a distribuição de recursos entre elas, e a recuperação de falhas, tudo isso através de uma interface centralizada. Para ilustrar, enquanto uma pequena empresa poderia gerenciar manualmente poucas instâncias, uma grande organização com picos de demanda e infraestrutura complexa, como uma empresa de e-commerce de grande porte, exigiria orquestração automática para manter a continuidade do serviço em momentos de alta carga, como em eventos sazonais.

A orquestração de VMs não se resume apenas à criação e destruição de instâncias, mas envolve uma série de processos automatizados, como provisionamento, escalabilidade automática, e monitoramento contínuo. Esses processos garantem que a infraestrutura esteja sempre otimizada e pronta para atender às demandas crescentes, minimizando o tempo de inatividade e maximizando a eficiência dos recursos.

3.9 Orquestração de Containers

A orquestração de containers segue uma lógica semelhante à de VMs, porém com um nível de granularidade e agilidade muito maior. Containers são instâncias leves que compartilham o mesmo sistema operacional do host, o que permite que sejam inicializados rapidamente e consumam menos recursos em comparação com VMs completas. No entanto, com a proliferação de containers em um ambiente de produção, gerenciá-los manualmente também se torna inviável. É aí que entra o Kubernetes, uma das ferramentas mais utilizadas para orquestração de containers.

O Kubernetes permite gerenciar de forma eficiente centenas ou até milhares de containers, distribuindo a carga de trabalho entre diferentes nós e garantindo que os serviços continuem funcionando, mesmo que algum nó falhe. Ele utiliza um sistema baseado em declarações em um arquivo *.yaml*, onde o administrador define o estado desejado da aplicação (quantidade de containers, recursos utilizados, etc.), e o Kubernetes se encarrega de garantir que esse estado seja mantido.

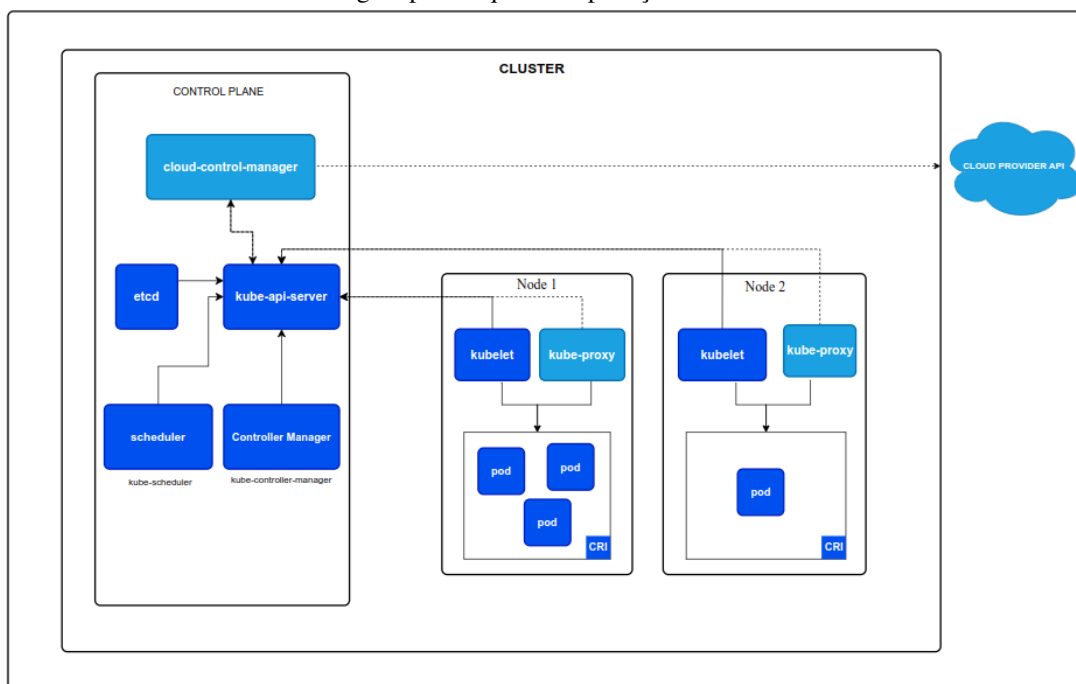
Além disso, o Kubernetes facilita o escalonamento de aplicações, monitorando o desempenho dos containers e adicionando ou removendo instâncias conforme necessário. Isso é particularmente útil em arquiteturas baseadas em microserviços, onde diferentes partes de uma aplicação podem ser escaladas de maneira independente, melhorando a eficiência e o uso de

recursos (KUBERNETS, 2024).

3.9.1 Estrutura do Kubernetes

O Kubernetes é composto por vários componentes que, juntos, formam um sistema robusto de orquestração como podemos ver na figura 3.18. Os principais elementos da arquitetura do Kubernetes incluem:

Figura 3.18: Estrutura do Kubernetes. A figura apresenta os principais componentes do Kubernetes, como nós, pods, plano de controle (*Control Plane*), *Scheduler*, *Kubelet* e serviços, ilustrando como eles interagem para orquestrar aplicações distribuídas de forma eficiente.



Fonte: <https://kubernetes.io/docs/concepts/architecture/>.

- **Nó (Node):** Cada máquina física ou virtual dentro do cluster do Kubernetes é chamada de nó. Um nó pode conter um ou mais containers.
- **Pods:** O pod é a menor unidade de execução no Kubernetes e pode conter um ou mais containers. Todos os containers dentro de um pod compartilham o mesmo endereço IP e espaço de rede, por padrão cada pod possui apenas um container.
- **Control Plane:** A camada de controle central que gerencia os nós do cluster e garante que o estado desejado das aplicações seja mantido.
- **Scheduler:** Responsável por alocar pods nos nós disponíveis de acordo com os requisitos de recursos e políticas definidas.
- **Kubelet:** Um agente que roda em cada nó do cluster e garante que os containers estejam executando conforme o especificado pelo Kubernetes.

- **Service:** Um serviço que expõe um conjunto de pods como uma única entidade, facilitando o balanceamento de carga e a descoberta de serviços.

Com essa arquitetura, o Kubernetes permite que as empresas implementem e gerenciem aplicações distribuídas de forma eficiente, garantindo alta disponibilidade, resiliência e facilidade de manutenção. Também pode ser feita a integração a nossa nuvem privada utilizando o componente Magnum do OpenStack.

4

Projeto de Implementação do OpenStack no IFB

Todos os detalhes apresentados neste desenvolvimento foram baseados na documentação oficial do OpenStack (OPENSTACK, 2024a). Algumas configurações específicas relacionadas às VMs foram obtidas em discussões sobre problemas registrados no *launchpad bugs (OpenStack)*.

A finalidade deste capítulo é detalhar o processo de implementação do OpenStack no campus do Instituto Federal de Brasília, apresentando a arquitetura proposta, as configurações realizadas nos diferentes nós (Controller, Storage e Compute), e os ajustes necessários para atender às demandas específicas da instituição. Além disso, são abordadas estratégias de automação, criação de projetos para diferentes perfis de usuários e as considerações para futuras atualizações e migrações, garantindo a escalabilidade e a eficiência do ambiente implementado.

4.1 Kolla

Para construir nosso *cluster* OpenStack, é necessário definir como o serviço será colocado em produção e como as máquinas serão distribuídas dentro do *cluster*. Optamos pelo uso do *Kolla-Ansible* devido à sua organização, facilidade de configuração e capacidade de automação.

O Kolla utiliza containeres Docker para executar os serviços de forma isolada, proporcionando maior controle sobre cada componente. Essa abordagem permite escalar serviços, alterar imagens, adicionar configurações específicas e realizar modificações sem impactar os demais serviços do *cluster*. A utilização de containeres também facilita a manutenção e atualizações, uma vez que cada serviço opera de maneira independente.

Além disso, o Kolla-Ansible permite definir em qual nó cada serviço será executado, organizando grupos de controle que atribuem os containeres aos nós correspondentes. Por padrão, o Nó principal (**Controller**) executa todos os serviços essenciais, enquanto os demais Nós recebem serviços específicos de acordo com suas funções, como *Compute* ou *Storage*. No entanto, também é possível configurar qualquer Nó para executar todos os serviços, dependendo da necessidade.

O OpenStack utiliza o **HAProxy** para realizar o balanceamento de carga entre os diferentes Nós, garantindo melhor distribuição do tráfego e alta disponibilidade. Isso elimina a necessidade de configurações adicionais de balanceamento, desde que os Nós estejam na mesma

rede.

O Kolla oferece diferentes versões para organizar e gerenciar as imagens Docker de forma eficiente. No nosso caso, optamos pelo lançamento mais estável disponível no momento desta pesquisa, “ubuntu-noble”. Essa escolha foi feita porque versões mais recentes nem sempre possuem todas as imagens *builded*, o que pode levar à ausência de imagens importantes, como o *Cinder* executando a versão “rocky”. Para configurar a *release* desejada, basta alterar a variável `openstack_tag` no arquivo `globals.yaml` e executar o *deploy*.

No Linux, o Python é essencial para o funcionamento de serviços do sistema, e alterar pacotes ou versões globalmente pode causar problemas. Por isso, é recomendável usar ambientes virtuais para isolar dependências de projetos, evitando interferências no sistema ou em outros projetos. Ferramentas como o `pyenv` permitem instalar e gerenciar versões específicas do Python de forma segura, criando ambientes virtuais dedicados sem afetar a versão padrão do sistema. Essa abordagem protege o sistema operacional e mantém um ambiente de desenvolvimento consistente. Para a instalação do kolla vamos utilizar o ‘python venv’ e dentro dele instalar os pacotes essenciais do kolla como recomendado em *Kolla Ansible documentation*.

Ao final do desenvolvimento, explicaremos como organizar nosso *cluster* para facilitar futuras migrações e atualizações. Como os serviços estão isolados em containeres, as atualizações tornam-se mais simples, permitindo a substituição individual de cada imagem sem impacto significativo nos demais serviços. Além disso, o Kolla garante uma retrocompatibilidade entre os containeres, tornando a troca entre versões mais estável e segura.

4.2 Hardware

O objetivo principal é aproveitar o hardware remanescente para construir uma nuvem privada, com a possibilidade de, no futuro, expandir essa infraestrutura para uma nuvem completa. Essa expansão permitiria distribuir todo o hardware disponível de forma eficiente, beneficiando diretamente os alunos do campus. Além disso, será demonstrado adiante como essa abordagem simplifica a distribuição de poder computacional e oferece maior visibilidade sobre a utilização dos recursos em cada projeto.

Para a criação de um nó (Nó) simples do OpenStack, recomenda-se uma CPU com 24 núcleos, 24 GB ou mais de memória RAM e 4 discos de 500 GB (7200 RPM) (OPENSTACK, 2024a), conforme ilustrado na Figura 4.1. Como o cluster é composto por 3 nós, esses recursos precisariam ser triplicados para atender às demandas do ambiente.

Devido às limitações do hardware disponível, essa configuração não é viável. Portanto, será criado apenas para fins de demonstração e apresentaremos a arquitetura do cluster, como deveria ser feito em servidores em produção, a forma como vamos estruturar essa arquitetura deixa simples a adição e substituição de mais nós.

Figura 4.1: Requisitos de hardware recomendados para a criação de um Nó OpenStack. A figura apresenta as especificações mínimas de CPU, memória e armazenamento necessárias para um nó OpenStack simples, com recomendações para ambientes de produção.

Criteria	Mínimo	Recomendado
CPU	4 core @ 2.4 GHz	24 core @ 2.67 GHz
RAM	8 GB	24 GB or more
HDD	2 x 500 GB (7200 rpm)	4 x 500 GB (7200 rpm)
RAID	Software RAID-1 (use mdadm, pois melhora o desempenho de leitura quase duas vezes)	Hardware RAID-10

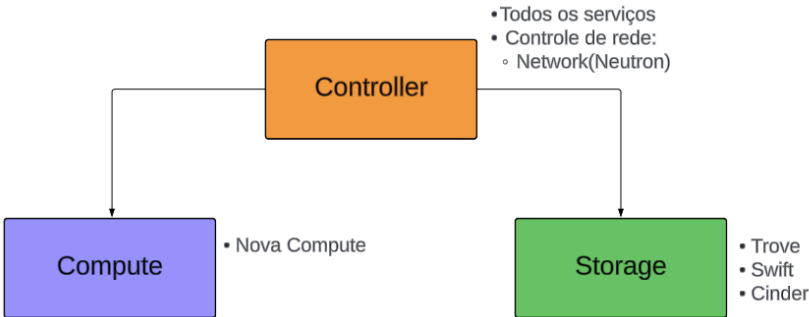
Fonte: https://docs.openstack.org/murano/rocky/admin/deploy_murano/prerequisites.html.

4.3 Arquitetura Geral

Para implementar uma nuvem privada no campus do IFB, o *cluster* OpenStack será composto por três nós: um nó principal (**Controller**), responsável pelos serviços essenciais, e dois nós adicionais dedicados, respectivamente, à criação de instâncias (*Compute*) e ao armazenamento (*Storage*). Essa divisão funcional segue o princípio de modularidade, conferindo escalabilidade e organização à arquitetura.

O **Controller** gerencia os demais nós do *cluster*, centralizando a criação e administração de recursos. As configurações principais do OpenStack são definidas no arquivo `globals.yaml`, localizado em `/etc/kolla/globals.yaml`, que especifica os serviços executados, interfaces de rede e alocações de IP, incluindo o Neutron. Configurações adicionais, como a relação entre os nós, são feitas no arquivo `multinode`, encontrado em `venv/share/kolla-ansible/ansible/inventory/multinode`.

Figura 4.2: Arquitetura geral do *cluster* OpenStack no IFB, mostrando a distribuição dos nós: **Controller**, responsável pelos serviços essenciais; **Compute**, para criação de instâncias; e **Storage**, dedicado ao armazenamento. A figura ilustra a relação funcional e as conexões entre os componentes.



Apesar de simples, essa arquitetura atende a todos os requisitos que comentamos sobre o campus, permitindo a criação de usuários, alocação de projetos e gerenciamento automatizado de recursos com agilidade. Além disso, o monitoramento detalhado possibilita tomadas de decisão informadas e a adição de recursos específicos para atender a demandas crescentes, garantindo

flexibilidade e desempenho.

4.4 Controller

Devido às limitações do hardware, composto por servidores antigos com 16 GB de RAM e 120 GB de armazenamento total por nó, a configuração busca otimizar os recursos disponíveis. Essa capacidade é muito inferior à recomendada, mas é suficiente para executar o OpenStack em um ambiente de testes.

Uma limitação significativa é a ausência de discos dedicados exclusivamente a serviços de armazenamento como o **Swift** e o **Cinder**, que requerem volumes lógicos preferencialmente alocados em discos separados. No entanto, nesta configuração, todos os serviços operam em conjunto no mesmo ambiente.

4.4.1 Ambiente de Teste com VMs

Também foram realizados testes em VMs para compreender a arquitetura, sua conectividade e o gerenciamento. Esses testes se mostraram eficientes facilitando o controle e os ajustes necessários, proporcionando um ambiente flexível para experimentação. A seguir, é apresentada uma explicação simplificada sobre como essa arquitetura de teste foi configurada.

Para garantir estabilidade, as VMs foram configuradas com 18 GB de RAM, adequadas para suportar serviços críticos, como o sistema de filas. O nó **Controller** utiliza 4 vCPUs, enquanto os nós de *compute* e *storage* operam com 3 vCPUs cada. Apesar de ocasionais momentos de lentidão, essa configuração atende à demonstração proposta.

O **Controller** foi equipado com 120 GB de armazenamento para maior estabilidade, enquanto os demais nós utilizam 60 GB, suficientes para o propósito do teste. Em ambientes de produção, o nó de *storage* demandaria uma maior capacidade de armazenamento para gerenciar grandes volumes de dados, e o nó de *compute* necessitaria de mais memória RAM e processadores mais robustos para múltiplas instâncias simultâneas, a quantidade exata vai depender diretamente da quantidade de infraestrutura que será provida pelo campus.

Adicionalmente, dois discos extras foram alocados ao **Controller** para gerenciar *storage*: um dedicado ao **Swift (Object Storage)** e outro ao **Cinder (Block Storage)**. A configuração desses discos é realizada no `globals.yaml`, sendo necessário criar os volumes lógicos previamente.

Embora o **Controller** gerencie serviços de armazenamento, um nó dedicado ao *storage* pode ser adicionado ao *cluster* para balancear a carga. O OpenStack distribui os dados automaticamente entre os nós disponíveis, permitindo escalabilidade conforme necessário, com adição eficiente de novos nós para atender ao crescimento das demandas.

4.4.2 Configurações do Host OpenStack

Como mencionado anteriormente, é fundamental compreender o papel dos volumes lógicos utilizados no *cluster*, pois eles simplificam significativamente o gerenciamento de espaço. Com o uso de volumes lógicos, alterações como adicionar, remover ou modificar *HDs* podem ser realizadas de forma transparente, sem a necessidade de alterar as configurações do *cluster* OpenStack. Toda a manipulação ocorre diretamente nos volumes lógicos, garantindo maior flexibilidade e praticidade na administração dos recursos.

O nó **Controller** exige pelo menos duas interfaces de rede: uma para conexão interna e outra para conexão externa. No entanto, como não há uma conexão externa disponível para provisionar IPs públicos, utilizaremos às duas interfaces de forma diferente. Uma delas será dedicada ao Neutron, mas sem suporte para alocar IPs públicos diretamente às instâncias. Ainda assim, será possível utilizar o gateway padrão do *switch* tendo acesso à internet e sendo possível que todos os usuários da rede interna acessem e utilizem as instâncias normalmente.

No caso da implementação em VMs, utilizaremos também duas interfaces porem configuradas para executar em modo `bridge`:

- A primeira interface será configurada em modo *bridge* com o host para permitir a comunicação interna.
- A segunda interface também será em modo *bridge*, dedicada ao serviço **Neutron**, responsável por prover *Network as a Service* (NaaS).

Os endereços IP estáticos para essas interfaces devem ser definidos no `netplan`. É essencial configurar IPs fixos para os nós, pois no arquivo `multinode` é necessário especificar o IP ou o DNS do nó que será utilizado nos grupos de controle. Além disso, é necessário reservar um endereço IP exclusivo na rede para o **Kolla Internal VIP Address**, utilizado na comunicação interna entre os serviços do OpenStack. A configuração deve ser como a mostrada em Código 3.

```
1 kolla_base_distro: "ubuntu"
2
3 network_interface: "enp0s3"
4 neutron_external_interface: "enp0s8"
5 kolla_internal_vip_address: "192.168.68.199" # esse IP tem que estar desalocado
6 kolla_external_vip_address: "192.168.68.199"
```

Código 3: Exemplo de configuração do endereço VIP interno e externo do Kolla no arquivo `globals.yaml`. Essa configuração inclui a definição das interfaces de rede (`network_interface` e `neutron_external_interface`) e o `kolla_internal_vip_address`, que deve ser um endereço IP exclusivo e não alocado, utilizado para a comunicação interna entre os serviços do OpenStack.

O parâmetro `kolla_base_distro` é utilizado para especificar a distribuição do sistema operacional em que os serviços do OpenStack serão executados. Essa configuração

auxilia o OpenStack a determinar quais pacotes devem ser instalados e como gerenciá-los dentro do *cluster*.

4.4.3 Configuração de Rede

A configuração da rede no OpenStack é um dos aspectos mais críticos para o funcionamento de um *cluster*, pois garante a conectividade entre os nós e permite que as instâncias interajam com redes internas e externas. Uma rede bem configurada é essencial para a comunicação entre serviços, o acesso a recursos externos.

4.4.4 Configuração de Rede Externa

A configuração de uma rede externa no OpenStack é essencial para permitir o acesso a recursos fora do *cluster* e para possibilitar a comunicação das instâncias com redes públicas ou corporativas. Esse processo inclui a definição da interface de rede física (ou virtual, no caso de VMs) responsável pela conexão externa, bem como a configuração do *gateway* e do roteamento de tráfego. Após essa etapa, é necessário criar a rede externa e a sub-rede correspondente (*subnet pool*), definindo a faixa de IPs públicos disponíveis, utilizando o Horizon ou a *CLI*.

Antes de iniciar a configuração, os seguintes pré-requisitos devem ser atendidos:

- Identificar e definir a interface de rede a ser utilizada para a conexão ao meio externo.
- Alocar previamente os endereços IP públicos e a máscara de rede correspondente.
- Configurar o *gateway* padrão e validar as regras do *firewall*, garantindo que o tráfego necessário esteja permitido.

No caso de testes realizados em máquinas virtuais, é necessário configurar uma interface de rede no modo *bridge*. Essa interface deve ser a mesma especificada no arquivo de configuração `globals.yaml` para o Neutron. Essa configuração garante que o tráfego da interface `br-ex` seja corretamente encaminhado para a rede externa, permitindo o acesso à internet pelas instâncias.

Além disso, independentemente da forma de *deployment*, se você estiver utilizando o **OVS** para controle de rede, padrão utilizado pelo OpenStack, é crucial verificar se os parâmetros `local_ip` e `bridge_mappings` estão configurados corretamente no arquivo `openvswitch_agent.ini`, como no Código 4.

4.4.5 SSH Entre os Nós

A configuração do acesso SSH é necessária na conexão dos nós, pois permite que o host principal gerencie e execute comandos em todo o *cluster*.

Para configurar o acesso SSH, é necessário criar uma chave SSH no **controller** e garantir que ele tenha acesso com permissões de *root* a todos os outros nós do *cluster*. Essa configuração

```
1 [ovs]
2 bridge_mappings = physnet1:br-ex
3 local_ip = <controller-ip>
```

Código 4: Configuração do arquivo `openvswitch_agent.ini` para habilitar o suporte à rede no cluster utilizando o OpenvSwitch (OVS). Os parâmetros `bridge_mappings` e `local_ip` são essenciais para associar as redes físicas às pontes virtuais e definir o IP local do nó no *cluster*.

elimina a necessidade de fornecer senhas para executar comandos com privilégios elevados, simplificando a administração e automação.

O processo envolve copiar a chave pública gerada no **controller** para cada nó utilizando o comando `ssh-copy-id`. Após isso, o usuário configurado geralmente já terá acesso como *sudo*. Caso contrário, será necessário acessar o nó, editar o arquivo de permissões utilizando o comando `visudo`, e adicionar o usuário do **controller** à lista de usuários com permissões de *root*. Essa configuração garante que o **controller** possa gerenciar os nós de forma segura e com o acesso necessário.

4.4.6 Containers do Cluster

Dentro do Openstack a configuração do **Docker Compose** feito pelo `kolla-ansible` utiliza da tag `'restart.always'` isso garante que se o container sofre com algum travamento e for deletado automaticamente o docker vai tentar subir outro container igual para substituí-lo. Isso é de extrema importância para manter a saúde do *cluster* e garantir que todos os serviços vão continuar habilitados.

Essa política também é seguida pelas configurações de saúde do *cluster* chamado de *healthcheck* garantindo que os containers sejam automaticamente recriados se forem removidos ou pararem, nesse caso até se o nó sofrer com um *reboot* automaticamente, após a inicialização, ocorrerá a recriação dos containers. Essas configurações podem ser vistas dentro de `'/etc/kolla/docker-compose'`. Para isso funcionar o serviço do docker precisa estar executando se for parado forçadamente os containers também serão destruídos.

4.4.7 Docker Register

Antes de irmos para a criação do próximo nó é necessário criar um serviço de registro no *controller*, ele basicamente vai ser responsável por fornecer as imagens dos serviços para os outros nós, dessa forma garantimos a consistência das imagens em todo o *cluster*. Pode ser criado mais de um registro dentro do *cluster*, assim como pode ter vários *controllers*. No nosso caso apenas um já resolve o problema. No `globals.yaml` deve ser adicionado as variáveis como mostrado no Código 5.


```
1 docker_registry: <IP-CONTROLLER>:<port>
2 docker_registry_insecure: yes
```

Código 5: Configuração do `globals.yaml` para definir o Docker registry interno, especificando as variáveis necessárias para que o registro forneça as imagens de forma consistente aos demais nós do cluster.

Após subir o Docker Registry e enviar as imagens para ele, é necessário alterar o arquivo `/etc/docker/daemon.json` no Docker para direcionar o *pull* das imagens primeiramente ao serviço de registro configurado no **Controller**. Isso garante que, ao adicionar novos nós ao *cluster*, eles busquem as imagens diretamente do registro interno, assegurando a consistência das imagens. Além disso, essa configuração facilita a propagação de novas imagens no *cluster*, tornando o processo mais eficiente e controlado. Nas configurações do Docker no arquivo `daemon.json` deve se adicionar Código 6.

```
1 {
2   "insecure-registries": ["http://<IP-CONTROLLER>:<port>"]
3 }
```

Código 6: Configuração do arquivo `daemon.json` para incluir o registro interno no Docker como uma rota insegura (*http*), permitindo que os nós do *cluster* realizem o *pull* das imagens diretamente do Docker registry interno.

4.4.8 Configuração Multinode

Para finalizar, é necessário ajustar o arquivo de configuração do Kolla-Ansible, chamado **multinode**, para especificar de forma clara quais serviços estão associados a cada nó do cluster. Nesta configuração, os serviços são organizados em grupos, como o **[Controller]**. No nosso caso, adicionamos o segundo nó ao grupo **[Storage]**. Essa definição no arquivo **multinode** garante que o **Controller** saiba exatamente onde cada serviço deve ser instalado e como acessar os demais nós. O arquivo **multinode** deve seguir o formato ilustrado no Código 7.

Caso seja necessário criar variáveis de ambiente ou configurar caminhos padrões do openstack, como o caminho do **python** que será utilizado, pode-se alterar o arquivo **ansible.cfg** naturalmente encontrado em `` /ansible/ ``, caso não tenha sido criado pelo **deployment** do *multinode*, não tem problema criar o arquivo direto em `` /.ansible.cfg ``.

Para verificar se o OpenStack está utilizando o arquivo de configuração correto, pode-se executar o comando ``ansible -version``:

```

1 [all:vars]
2 # criação de variáveis de ambiente para rodar o multinode
3 [all]
4 192.168.68.101 ansible_ssh_user=controller ansible_become=True
5 ↪ ansible_private_key_file=~/.ssh/id_rsa
6 192.168.68.103 ansible_ssh_user=storage ansible_become=True
7 ↪ ansible_private_key_file=~/.ssh/id_rsa
8 192.168.68.104 ansible_ssh_user=compute ansible_become=True
9 ↪ ansible_private_key_file=~/.ssh/id_rsa
10 [control]
11 192.168.68.101 ansible_ssh_user=controller ansible_become=True
12 ↪ ansible_private_key_file=~/.ssh/id_rsa
13 [network]
14 192.168.68.101 ansible_ssh_user=controller ansible_become=True
15 ↪ ansible_private_key_file=~/.ssh/id_rsa
16 [compute]
17 192.168.68.101 ansible_ssh_user=controller ansible_become=True
18 ↪ ansible_private_key_file=~/.ssh/id_rsa
19 192.168.68.104 ansible_ssh_user=compute ansible_become=True
20 ↪ ansible_private_key_file=~/.ssh/id_rsa
21 [monitoring]
22 192.168.68.101 ansible_ssh_user=controller ansible_become=True
23 ↪ ansible_private_key_file=~/.ssh/id_rsa
24 [storage]
25 192.168.68.101 ansible_ssh_user=controller ansible_become=True
26 ↪ ansible_private_key_file=~/.ssh/id_rsa
27 192.168.68.103 ansible_ssh_user=storage ansible_become=True
28 ↪ ansible_private_key_file=~/.ssh/id_rsa

```

Código 7: Configuração do arquivo multinode separando os grupos de controle entre os nós.

4.5 Segundo Nó (Storage)

Este nó será dedicado exclusivamente ao armazenamento, utilizando os serviços **Cinder** e **Swift**, cada um alocado em *volume groups* separados. Inicialmente, cada *volume group* (*cinder-volumes* e *swift-volumes*) será configurado com um único disco. Essa abordagem oferece flexibilidade para futuras expansões, permitindo adicionar novos discos ao *volume group* sem complexidade significativa.

A criação inicial dos *volume groups* é realizada manualmente no **Controller**, que também executa todos os serviços essenciais. Dessa forma, é necessário alocar previamente os discos destinados ao **Swift** e ao **Cinder** no **Controller** antes de configurar o nó de *storage*. Após essa etapa, é possível automatizar a criação de volumes e a configuração dos discos, reduzindo a necessidade de intervenção manual nos nós. Caso novos discos sejam adicionados após a implantação inicial, eles podem ser incluídos manualmente nos *volume groups* do nó de *storage* de forma simples e eficiente.

Uma vez configurado o **Controller**, é essencial garantir que ele tenha acesso ao nó de *storage*. Isso requer a configuração das chaves SSH para permitir o acesso seguro e direto do **Controller** ao nó de *storage*, eliminando a necessidade de autenticação manual. Além disso, é necessário atualizar o arquivo de inventário `/etc/kolla/inventory/multinode` com o endereço IP ou DNS do nó de *storage*, garantindo que os serviços do OpenStack possam localizá-lo corretamente.

Com o nó de *storage* configurado, o **Cinder** gerenciará volumes para instâncias e o **Swift** será responsável pelo armazenamento de objetos, permitindo que o *cluster* atenda a diferentes demandas de armazenamento de maneira eficiente e escalável.

4.6 Terceiro Nó (Compute)

Este nó será dedicado exclusivamente à criação de máquinas virtuais (*instances*), desempenhando o papel equivalente ao EC2 em provedores de nuvem pública. Ele utilizará os recursos alocados, como CPU, memória e armazenamento local, para provisionar máquinas virtuais com base nos *flavors* definidos pelo usuário da organização.

Os *flavors* representam as especificações das máquinas virtuais, como quantidade de CPUs, memória RAM e tamanho do disco. Esses recursos são consumidos diretamente do nó **Compute**, que é responsável por executar as instâncias e garantir sua performance, isolamento e conectividade.

A configuração do nó **Compute** é realizada pelo Kolla-Ansible, utilizando o arquivo de inventário `/etc/kolla/inventory/multinode`. Neste arquivo, o nó **Compute** deve ser listado sob o grupo **[compute]** para que os serviços necessários, como o *Nova Compute*, sejam implantados corretamente. Além disso, o **Controller** gerencia as tarefas administrativas relacionadas à criação, inicialização e destruição das máquinas virtuais, enquanto o nó **Compute** executa as operações físicas.

A comunicação entre o **Controller** e o nó **Compute** é feita de forma segura e direta. Seguindo a mesma ideia do outro nó, precisamos configurar a chave SSH e adicionar o DNS ou IP no arquivo **multinode**

Após a implantação inicial, o nó **Compute** estará pronto para receber solicitações de criação de máquinas virtuais. Ele será responsável por alocar os recursos conforme especificado nos *flavors*, garantindo a escalabilidade e eficiência do *cluster*. Caso novos nós **Compute** sejam adicionados ao *cluster*, o balanceamento de carga entre os nós será automaticamente gerenciado pelo OpenStack, otimizando o uso de recursos.

4.7 Deploy da Aplicação

Após configurar a rede e os arquivos mencionados, o *deploy* dos serviços pode ser realizado. Esse processo cria múltiplos containeres que executam os serviços padrão do OpenStack, conforme ilustrado na Figura 4.4. O acesso ao painel do OpenStack (**Horizon**) pode ser feito por meio do endereço `http://kolla_internal_vip_address/dashboard`. A partir desse painel, é possível configurar todos os serviços de IaaS. Detalhes adicionais sobre a configuração dos serviços serão apresentados nas próximas seções.

Os containeres exibidos, ou a maioria deles, devem estar visíveis em todos os nós do *cluster*. Cada container possui a marcação *health*, indicando que está em execução sem erros

Figura 4.3: Containers de base do serviço OpenStack executando no nó principal do *cluster* após efetuar o *deploy* com o Kolla-ansible, mostrando de estão saudáveis e quando foram iniciados

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
9d6af125930a	192.168.68.101:4890/openstack.kolla/mariadb-server:master-ubuntu-noble	"dumb-init -- kolla..."	13 hours ago	Up 13 hours (healthy)		mariadb
4d6da11755b5	192.168.68.101:4890/openstack.kolla/proxyysql:master-ubuntu-noble	"dumb-init --single..."	15 hours ago	Up 15 hours (healthy)		proxyysql
b7436415b6c9	192.168.68.101:4890/openstack.kolla/haproxy:master-ubuntu-noble	"dumb-init --single..."	15 hours ago	Up 15 hours (healthy)		haproxy
c9e8025eb995	192.168.68.101:4890/openstack.kolla/horizon:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 11 hours (healthy)		horizon
568628dca41	192.168.68.101:4890/openstack.kolla/heat-engine:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 11 hours (healthy)		heat_engine
68428f9f20b4	192.168.68.101:4890/openstack.kolla/heat-api-cfn:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 11 hours (healthy)		heat_api_cfn
6db25b70e669	192.168.68.101:4890/openstack.kolla/heat-api:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 11 hours (healthy)		heat_api
1043a95593bd	192.168.68.101:4890/openstack.kolla/neutron-metadata-agent:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 11 hours (healthy)		neutron_metadata_agent
5ac866dc5b72	192.168.68.101:4890/openstack.kolla/neutron-l3-agent:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 11 hours (healthy)		neutron_l3_agent
0be5f6a8356f	192.168.68.101:4890/openstack.kolla/neutron-dhcp-agent:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		neutron_dhcp_agent
b04359939208	192.168.68.101:4890/openstack.kolla/neutron-openvswitch-agent:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		neutron_openvswitch_agent
e335d68f5e75	192.168.68.101:4890/openstack.kolla/neutron-server:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		neutron_server
0dd423879716	192.168.68.101:4890/openstack.kolla/nova-compute:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		nova_compute
abd116fec181	192.168.68.101:4890/openstack.kolla/nova-libvirt:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 16 hours (healthy)		nova_libvirt
42ecbce8f9d2	192.168.68.101:4890/openstack.kolla/nova-ssh:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 16 hours (healthy)		nova_ssh
95c3899c387d	192.168.68.101:4890/openstack.kolla/nova-novncproxy:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		nova_novncproxy
fb7ef637da4a	192.168.68.101:4890/openstack.kolla/nova-conductor:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		nova_conductor
c04249001538	192.168.68.101:4890/openstack.kolla/nova-api:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		nova_metadata
b0e3da79c4b	192.168.68.101:4890/openstack.kolla/nova-scheduler:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		nova_api
5c59b86b9dc5	192.168.68.101:4890/openstack.kolla/nova-scheduler:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		nova_scheduler
8d8b9d75024e	192.168.68.101:4890/openstack.kolla/placement-api:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		placement_api
9c943ae4d393	192.168.68.101:4890/openstack.kolla/glance-api:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		glance_api
f81842186c6d	192.168.68.101:4890/openstack.kolla/keystone:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		keystone
8a91ab3ad0d3	192.168.68.101:4890/openstack.kolla/keystone-fernet:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		keystone_fernet
804dbf8de0b9	192.168.68.101:4890/openstack.kolla/keystone-ssh:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 16 hours (healthy)		keystone_ssh
2b711fffbe78	192.168.68.101:4890/openstack.kolla/rabbitmq:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 12 hours (healthy)		rabbitmq
bebb21212918	192.168.68.101:4890/openstack.kolla/memcached:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 16 hours (healthy)		memcached
0ce6a5b64147	192.168.68.101:4890/openstack.kolla/keepalived:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 16 hours		keepalived
07407cd572bf	192.168.68.101:4890/openstack.kolla/cron:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 16 hours		cron
7072836372ad	192.168.68.101:4890/openstack.kolla/kolla-toolbox:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 16 hours		kolla_toolbox
6375bcd9f9cc	192.168.68.101:4890/openstack.kolla/fluentd:master-ubuntu-noble	"dumb-init --single..."	4 days ago	Up 16 hours		fluentd
62f81d3c026f	registry:2	"/entrypoint.sh /etc..."	5 days ago	Up 16 hours		registry

Figura 4.4: Imagem retirada do terminal do nó principal do cluster

detectados.

4.8 Por Dentro do OpenStack

Após configurado o *cluster*, é fundamental entender como o OpenStack organiza e gerenciar seus recursos. O OpenStack funciona por meio de **projetos**, que podem conter diversos **usuários**, e cada usuário tem permissões específicas definidas pelas *roles*.

A seguir, exploraremos os principais conceitos e configurações relacionados aos projetos, usuários, políticas de acesso, cotas e rede, detalhando suas funcionalidades e como interagem no ecossistema OpenStack, todos os comandos mostrados podem ser feitos direto pelo Horizon.

4.8.1 OpenStack-Client

Para utilizar a maioria dos comandos descritos nesta seção, é necessário instalar o pacote `python-openstackclient`. Isso pode ser feito utilizando o gerenciador de pacotes `pip`. Após a instalação, você deve realizar o *login* configurando as variáveis de ambiente listadas abaixo.

Por padrão, as únicas alterações necessárias serão no valor de `VIP_IP` (o endereço IP configurado) e na senha gerada automaticamente. O endereço IP pode ser consultado no arquivo `globals.yaml`, enquanto a senha pode ser obtida utilizando o comando da Figura 8.

```
1 sudo cat /etc/kolla/passwords.yml | grep "keystone_admin_password"
```

Código 8: Comando utilizado para recuperar a senha gerada automaticamente para a conta admin no arquivo de configuração do Kolla (`passwords.yml`). Essa senha será utilizada para configurar o acesso ao OpenStack.

Após obter essas informações, é possível criar o arquivo `admin_variables.sh`, que conterá as variáveis de ambiente necessárias para acessar o *admin*. Após criar esse arquivo com as variáveis demonstradas na Figura 9, basta executá-lo com o comando `source admin_variables.sh`

```

1 export OS_AUTH_URL=http://<VIP_IP>:5000/v3
2 export OS_PROJECT_NAME=admin
3 export OS_USERNAME=admin
4 export OS_PASSWORD=<GENERATED_PASSWORD>
5 export OS_PROJECT_DOMAIN_NAME=Default
6 export OS_USER_DOMAIN_NAME=Default
7 export OS_REGION_NAME=RegionOne
8 export OS_INTERFACE=public
9 export OS_IDENTITY_API_VERSION=3

```

Código 9: Exemplo de configuração das variáveis de ambiente necessárias para acessar o admin do OpenStack utilizando o Terminal. Inclui definições do endpoint (`OS_AUTH_URL`), credenciais de autenticação, e outros parâmetros essenciais.

4.8.2 Projetos

Os projetos (também chamados de *tenants*) agrupam recursos como instâncias, redes e volumes. Cada projeto possui isolamento de recursos, evitando interferência de um projeto sobre outro. Além disso, é possível definir cotas para limitar o uso de instâncias, redes e outros recursos. Cada projeto pode ter usuários com papéis bem definidos, garantindo um controle de acesso eficaz.

O OpenStack também permite personalizar quais serviços um projeto pode acessar, como *Swift*, *Glance*, entre outros. Na figura 4.5, é exibida a interface do *dashboard* Horizon, onde podem ser realizadas as configurações relacionadas a projetos, incluindo sua criação e atribuição de permissões.

Figura 4.5: Interface do Horizon mostrando a configuração e gerenciamento de projetos, local onde também é configurado todos os detalhes relacionados aos projetos.

Name	Description	Project ID	Domain Name	Enabled	Actions
Deployment	development project.	32f6c2d5e6c454791f35363e278900f	Default	Yes	Manage Members
service		5c55b631ad8b47b1a13e8218358148bf	Default	Yes	Manage Members
admin	Bootstrap project for initializing the cloud.	a369aef83ad41e2bb2f1432f637f117d	Default	Yes	Manage Members

Fonte: *Dashboard* Horizon do OpenStack.

Dentro dos projetos utilizamos cotas. As cotas delimitam o número de recursos que cada projeto pode consumir, como instâncias, volumes e redes. Embora sejam úteis para evitar o uso abusivo de recursos, o OpenStack não garante previamente que o *cluster* tenha a capacidade de suprir todas as cotas aprovadas. Caso um projeto tente criar recursos além da capacidade disponível no *cluster*, erros de criação podem ocorrer, e nós sobrecarregados podem apresentar *reboots* inesperados.

4.8.3 Usuários

Os usuários são entidades autenticadas pelo serviço **Keystone**, que gerencia identidades e permissões. Eles podem ser criados tanto pela interface de linha de comando (CLI) quanto pela interface gráfica (Horizon). Cada usuário está associado a um ou mais projetos, com *roles* específicas que definem suas permissões no ambiente.

O Keystone oferece suporte a diferentes métodos de autenticação, como LDAP ou banco de dados interno, permitindo integração com sistemas externos. Na figura 4.6, é mostrado como usuários e contas de serviço são configurados de forma similar, diferenciando-se apenas pelas políticas de segurança atribuídas.

Figura 4.6: Interface do Horizon mostrando usuários e contas de serviço configurados pelo Keystone.

Identity / Users

Users

Displaying 8 items

User Name

Filter

+ Create User

Delete Users

<input type="checkbox"/>	User Name	Description	Email	User ID	Enabled	Domain Name	Actions
<input type="checkbox"/>	admin	-		aa7612cdf7fc4f4caa455e5c3de22676	Yes	Default	Edit
<input type="checkbox"/>	heat_domain_admin	-		3b6bf23a2a19460091696ca4027cae64	Yes	-	Edit
<input type="checkbox"/>	neutron	-		4da1da09ca1b484d8fec2614bdebd399	Yes	Default	Edit
<input type="checkbox"/>	heat	-		b0a6f88f13cd4216b7baeba92e7c57c4	Yes	Default	Edit
<input type="checkbox"/>	user-bezerra	-	bezerra.fernandes69@gmail.com	e710014ee45d437d8f16bb651416b62f	Yes	Default	Edit
<input type="checkbox"/>	glance	-		3c1b9f8b680f4647b4b7fbb7ee477098	Yes	Default	Edit
<input type="checkbox"/>	placement	-		3e9da026f4324b35a05d921eb21b9998	Yes	Default	Edit
<input type="checkbox"/>	nova	-		32b685f74b11448eafc6b6f0abc5402c	Yes	Default	Edit

Displaying 8 items

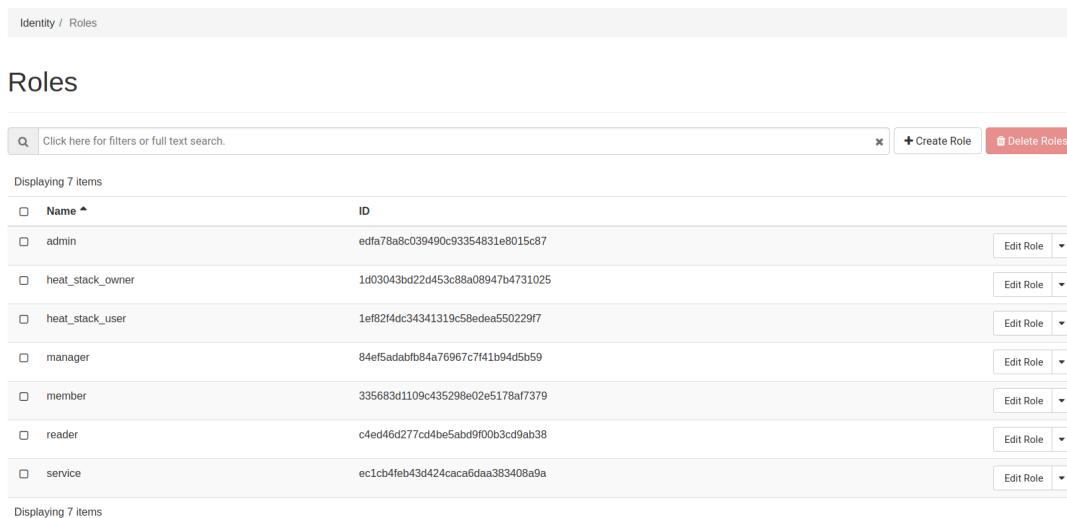
Fonte: *Dashboard* Horizon do OpenStack.

4.8.4 Políticas de Acesso para Serviços e Usuários

As políticas de acesso, definidas em arquivos de configuração (JSON ou YAML), determinam quais operações cada *role* pode executar nos serviços, como Nova, Neutron e Cinder. Isso viabiliza um controle granular, permitindo personalizar ações como criação de instâncias e gerenciamento de redes.

Em implantações como Kolla-Ansible, esses arquivos podem ser gerenciados centralmente para simplificar a administração. Na figura 4.7, são mostradas as *roles* padrões criadas pelo OpenStack para contas de serviço.

Figura 4.7: Interface do Horizon mostrando as *roles* padrões do OpenStack para controle de contas de serviço.



Name	ID	Action
admin	edfa78a8c039490c93354831e8015c87	Edit Role
heat_stack_owner	1d03043bd22d453c88a08947b4731025	Edit Role
heat_stack_user	1ef82f4dc34341319c58ede550229f7	Edit Role
manager	84ef5adabfb84a76967c7f41b94d5b59	Edit Role
member	335683d1109c435298e02e5178af7379	Edit Role
reader	c4ed46d277cd4be5abd9f00b3cd9ab38	Edit Role
service	ec1cb4feb43d424caca6daa383408a9a	Edit Role

Fonte: *Dashboard* Horizon do OpenStack.

4.8.5 Rede

O serviço **Neutron** gerencia a criação de redes e *subnets* virtuais (ou VPCs), onde se definem regras de firewall, roteamento e conexões entre instâncias. Cada projeto pode manter suas próprias redes isoladas, garantindo segurança e organização.

A visualização da topologia de rede no Horizon (ou via *CLI*) exibe como cada elemento (instâncias, roteadores e sub-redes) está interligado, ajudando a identificar rotas e possíveis pontos de falha. Após as configurações iniciais, é possível criar uma rede externa utilizando os comandos direto no terminal. No Código 10 criamos a *network* externa e definimos uma *subnet* para alocar IPs no intervalo especificado em **–allocation-pool**, além de configurarmos o *gateway* de saída.

Também criamos uma rede interna Código 11, que será utilizada pelas instâncias. Enquanto a rede externa é compartilhada entre projetos, as redes internas são isoladas para cada projeto, sendo criadas a partir de contas administradoras de projetos.

Para verificar se as conexões foram configuradas corretamente, podemos acessar a área *Network Topology*. Essa seção apresenta uma visualização gráfica de como a *Network* está estruturada, permitindo identificar as conexões entre instâncias, roteadores e sub-redes. A figura 4.8 ilustra essa visualização dentro do *dashboard* Horizon.

```
1 openstack network create \  
2   --provider-network-type flat \  
3   --provider-physical-network physnet1 \  
4   --external external-network \  
5 \  
6 openstack subnet create \  
7   --network external-network \  
8   --subnet-range 192.168.1.0/24 \  
9   --allocation-pool start=192.168.1.100,end=192.168.1.200 \  
10  --gateway 192.168.1.1 \  
11  --no-dhcp external-subnet
```

Código 10: Comandos para criar uma rede externa no OpenStack. A configuração define o tipo de rede (*flat*), associa a rede ao provedor físico (*physnet1*), e configura uma sub-rede com intervalo de endereços alocados, gateway e desativação do DHCP.

```
1 openstack network create internal-network \  
2 openstack subnet create \  
3   --network internal-network \  
4   --subnet-range 10.0.0.0/24 \  
5   internal-subnet
```

Código 11: Comandos para criar uma rede interna no OpenStack. Redes internas são isoladas por projeto e utilizadas pelas instâncias. A configuração inclui a criação da rede e de uma sub-rede associada, com as definições de faixa de IPs, gateway e outras propriedades específicas.

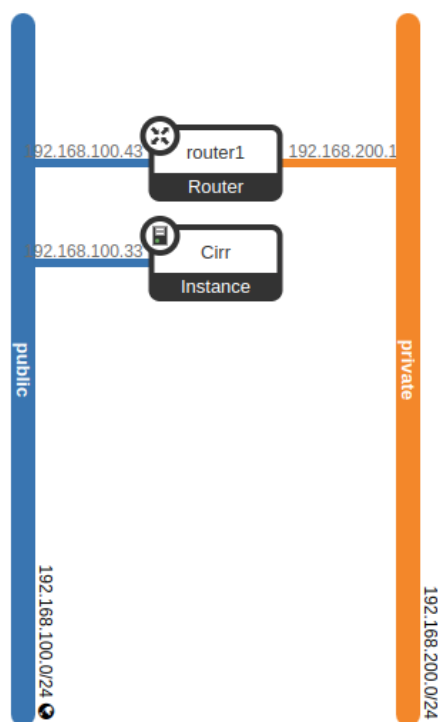
4.8.6 Arquitetura de Separação de Recursos para o Campus

O OpenStack oferece diversas formas de organizar e separar projetos e usuários para atender a diferentes necessidades. No contexto de uma instituição de ensino superior, como um campus universitário, é comum a necessidade de segmentar recursos entre departamentos, e dentro desses, entre seus respectivos usuários. Uma abordagem eficiente para essa finalidade é a utilização da *Hierarchical Multitenancy*.

Essa arquitetura permite a criação de uma hierarquia de projetos, em que subprojetos são vinculados a um projeto principal. Cada projeto principal pode representar um departamento, como Engenharia, Ciências da Computação ou Administração, enquanto os subprojetos podem ser utilizados para atividades específicas, como Trabalhos de Conclusão de Curso (TCCs), pesquisas acadêmicas ou laboratórios de ensino.

A utilização dessa arquitetura torna-se ainda mais poderosa quando combinada com os templates do Heat. Esses templates permitem a criação automatizada de todos os componentes necessários para iniciar uma nova pesquisa ou atividade, otimizando o provisionamento de recursos e reduzindo o esforço operacional. A seguir, serão apresentados detalhes sobre os templates e suas aplicações práticas.

Figura 4.8: Visualização da topologia de rede no *dashboard* Horizon, mostrando as conexões entre instâncias, roteadores e sub-redes.



Fonte: *Dashboard* Horizon do OpenStack.

4.8.7 Debug e Resolução de Problemas

Devido à complexidade e constante desenvolvimento do OpenStack, erros são situações frequentes e podem exigir diferentes abordagens de depuração. A análise inicial deve sempre incluir a verificação dos arquivos de log, geralmente localizados em `/var/log/kolla/`, que fornecem informações detalhadas sobre o comportamento dos serviços e possíveis falhas. Adicionalmente, habilitar o modo de depuração nos serviços pode facilitar a identificação de problemas. Isso pode ser feito configurando a variável `DEBUG=True` no arquivo de configuração do serviço em questão.

Quando o problema envolve um erro mais profundo, como falhas em componentes do código, recomenda-se o uso do **DevStack**. O DevStack é uma ferramenta leve e altamente configurável, ideal para criar um ambiente de desenvolvimento que replica uma instalação do OpenStack em um único nó. Ele permite a execução de serviços do OpenStack em modo debug, facilitando o acesso a logs detalhados e a inserção de pontos de interrupção no código. Esse ambiente é ideal para testar alterações e investigar problemas sem afetar a instalação principal.

Para realizar uma depuração interativa com o DevStack, pode-se utilizar o depurador *pdb* do Python. Insere-se a seguinte linha no código no ponto onde deseja interromper a execução:

```
import pdb; pdb.set_trace().
```

Após ajustar o código, basta reiniciar o serviço no ambiente DevStack para realizar a

análise interativa. Essa abordagem é vantajosa, pois permite isolar problemas e testar soluções de forma controlada antes de aplicá-las em um ambiente de produção.

Embora a maioria dos erros possa ser resolvida ajustando configurações ou corrigindo permissões, erros de código requerem conhecimento aprofundado sobre o funcionamento interno dos serviços do OpenStack e suas interações. Nesses casos, o uso do DevStack como ambiente de desenvolvimento é altamente recomendado, pois proporciona flexibilidade e controle total sobre os serviços e suas dependências.

4.9 Migração e Atualizações Futuras

No contexto de software, atualizações e correções de erros são lançadas regularmente, e o OpenStack não é uma exceção. É fundamental manter o ambiente atualizado para aproveitar novos recursos (*features*) e melhorias de desempenho, além de corrigir vulnerabilidades de segurança. Antes de realizar qualquer atualização, é de extrema importância garantir que todos os backups necessários estejam disponíveis, permitindo a restauração do sistema em caso de falhas.

4.9.1 Atualização de containeres

Como mencionado anteriormente, o Kolla-Ansible isola os serviços do OpenStack em containeres Docker. Essa arquitetura permite atualizar os containeres de forma individual. No entanto, para garantir que todos os serviços sejam devidamente sincronizados, se comuniquem corretamente e, quando necessário, sejam reconfigurados automaticamente, recomenda-se realizar uma atualização completa utilizando como no Código 12:

```
1 source path/venv/bin/activate
2 pip install --upgrade git+https://opendev.org/openstack/kolla-ansible@master
3 kolla-ansible pull
4 kolla-ansible prechecks
5 kolla-ansible upgrade
6 kolla-ansible deploy
```

Código 12: Comando para atualizar todos os serviços do OpenStack gerenciados pelo Kolla-Ansible. Essa abordagem garante que os containeres sejam sincronizados, configurados corretamente e reconfigurados automaticamente quando necessário.

Esse procedimento garante que os containeres sejam atualizados para as versões mais recentes das imagens, aplicando automaticamente quaisquer modificações *necessárias nas configurações e garantindo a continuidade dos serviços.

Em alguns casos, quando são lançadas novas *features* ou alterações significativas nas configurações, é necessário atualizar o arquivo `globals.yaml`. Esse arquivo atualizado pode ser encontrado no diretório de exemplos `../kolla-ansible/etc_examples/kolla/`.

Após obter o novo arquivo, é necessário incorporar manualmente as configurações existentes para preservar a compatibilidade com o ambiente atual.

4.10 Análise do Cluster

Com os três nós configurados e operacionais, utilizamos serviços nativos do OpenStack para realizar a análise do cluster. Os serviços utilizados incluem:

- **Gnocchi:** Responsável por coletar e armazenar métricas relacionadas ao desempenho e utilização do cluster, como uso de CPU, memória e armazenamento.
- **Aodh:** Utilizado para a configuração de alarmes. Este serviço nos permite definir ações automatizadas caso ocorra alguma falha.
- **Panko:** Registro e armazenamento de logs de eventos, mostrando exatamente o que foi feito e por qual pessoa.
- **Ceilometer:** Um dos principais serviços do OpenStack para medição de dados de uso. Ele fornece uma visão abrangente dos recursos consumidos, contribuindo para relatórios detalhados e para a gestão eficiente do cluster.

A integração desses serviços permite não apenas monitorar o desempenho e a utilização do *cluster*, mas também detectar anomalias e registrar eventos críticos. Essa abordagem facilita a manutenção preventiva e corretiva no ambiente OpenStack.

Além disso, o **Ceilometer** permite integração com outros serviços de monitoramento amplamente utilizados, como o Grafana. Essa conexão possibilita a criação de *dashboards* personalizados para visualizar métricas em tempo real, identificar tendências de uso e acompanhar o desempenho do cluster de maneira gráfica e centralizada.

Por fim, a análise do *cluster* é um passo essencial para garantir que a infraestrutura opere com alta disponibilidade, desempenho consistente e segurança, características fundamentais para atender às demandas de um ambiente acadêmico e administrativo.

4.11 Implementação e Utilização de Recursos

O Heat é o serviço de orquestração do OpenStack que permite automatizar o gerenciamento de infraestrutura por meio de templates declarativos escritos em YAML, naturalmente esse tipo de serviço é chamado de IaC (*Infrastructure as Code*). Com ele, você pode definir, provisionar e gerenciar recursos, como instâncias, volumes, redes e até aplicações inteiras, de forma padronizada e repetível. Esse serviço é especialmente útil para configurar ambientes de trabalho para novos estudantes, permitindo que iniciem projetos com rapidez e eficiência. Além disso, o mesmo template pode ser facilmente ajustado para adicionar mais recursos, conforme a necessidade do projeto.

Os próprios alunos também podem utilizar templates do Heat para gerenciar a infraestrutura de seus projetos. Isso facilita a demonstração de quais recursos foram utilizados e como o ambiente pode ser replicado de forma simples e eficiente. Essa abordagem promove organização e consistência no uso dos recursos de infraestrutura, otimizando o aprendizado e a colaboração em projetos acadêmicos.

4.11.1 Configuração para a utilização de IaC

É possível que o serviço Heat não tenha permissões diretas para acessar e manipular o *cluster*. Para contornar essa limitação, normalmente criamos um arquivo de configuração específico para o Heat em `/etc/kolla/config/heat.conf`, utilizando a configuração como mostrado no Código 13.

É necessário manter esses arquivos de configuração devidamente protegidos, pois eles geralmente contêm credenciais sensíveis. Essa prática deve ser aplicada a todos os arquivos localizados em `/etc/kolla/`, garantindo a segurança do ambiente e prevenindo acessos não autorizados.

```
1 [database]
2 connection = mysql+pymysql://heat:<HEAT_DB_PASSWORD>@<DATABASE_HOST>/heat
3
4 [keystone_authtoken]
5 www_authenticate_uri = http://<KEYSTONE_HOST>:5000
6 auth_url = http://<KEYSTONE_HOST>:5000
7 memcached_servers = <MEMCACHED_HOST>:11211
8 auth_type = password
9 project_domain_name = default
10 user_domain_name = default
11 project_name = service
12 username = heat
13 password = <HEAT_SERVICE_PASSWORD>
```

Código 13: Exemplo de configuração do arquivo `heat.conf`, localizado em `/etc/kolla/config/`. Esse arquivo define as permissões e parâmetros necessários para o serviço Heat gerenciar a infraestrutura no *cluster*.

4.11.2 Criação de Instâncias para Alunos em Sala de Aula

Uma aplicação prática do *cluster* em um ambiente acadêmico é sua integração com as atividades em sala de aula. Considere o seguinte cenário: você está ministrando uma aula sobre processamento de imagens e deseja disponibilizar uma instância para cada aluno. Essas instâncias podem ser configuradas previamente com um ambiente virtual e todos os pacotes necessários do python já instalados, garantindo que os alunos possam se concentrar exclusivamente nos objetivos da aula.

Para alcançar esse objetivo, é possível utilizar um único template do Heat. Aplicando o conceito de `HierarchicalMultitenancy`, criamos um subprojeto denominado

aula-proc-dig dentro do projeto principal. Nesse subprojeto, as instâncias dos alunos serão provisionadas de forma organizada Código 14.

```
1 heat_template_version: 2021-04-16
2
3 description: Template para criação de instâncias para alunos em sala de aula.
4
5 parameters:
6   student_count:
7     type: number
8     description: Número de alunos (instâncias a serem criadas)
9     default: 10
10  image_id:
11    type: string
12    description: ID da imagem com ambiente pré-configurado
13  flavor:
14    type: string
15    description: Tipo de instância (flavor)
16    default: m1.small
17  network_id:
18    type: string
19    description: ID da rede para as instâncias
20
21 resources:
22   student_instances:
23     type: OS::Heat::ResourceGroup
24     properties:
25       count: { get_param: student_count }
26       resource_def:
27         type: OS::Nova::Server
28         properties:
29           name:
30             str_replace:
31               template: "aluno-INSTANCE"
32               params:
33                 INSTANCE: "%index%"
34             image: { get_param: image_id }
35             flavor: { get_param: flavor }
36             networks:
37               - network: { get_param: network_id }
38
39 outputs:
40   instances_info:
41     description: Informações das instâncias criadas
42     value: { get_attr: [student_instances, show] }
```

Código 14: Exemplo de template Heat para provisionamento automatizado de instâncias no subprojeto aula-proc-dig, configuradas para atividades de processamento de imagens.

4.11.3 Criação de Projetos para Alunos de Pesquisa

Para atender alunos que estão iniciando projetos de pesquisa, como CNPq ou Trabalhos de Conclusão de Curso (TCC), é possível criar um template dedicado. Também seguindo o conceito de `HierarchicalMultitenancy`, podemos provisionar uma rede e uma instância isolada para cada aluno, permitindo que eles configurem o ambiente de acordo com suas necessidades específicas. Além disso, essa abordagem facilita a escalabilidade, possibilitando a alocação de recursos adicionais de maneira ágil e controlada, caso seja necessário Código 15.

```
1 heat_template_version: 2021-04-16
2
3 description: Template para provisionamento de projetos individuais de pesquisa, incluindo rede
4   ↪ e instância isolada.
5
6 parameters:
7   project_name:
8     type: string
9     description: Nome do projeto do aluno
10    default: "projeto-pesquisa"
11  image_id:
12    type: string
13    description: ID da imagem com ambiente pré-configurado
14  flavor:
15    type: string
16    description: Tipo de instância (flavor)
17    default: m1.small
18  network_cidr:
19    type: string
20    description: CIDR para a rede privada do projeto
21    default: "192.168.100.0/24"
22  gateway_ip:
23    type: string
24    description: Gateway IP para a rede privada
25    default: "192.168.100.1"
```

Código 15: Parte 1: Exemplo de template Heat para provisionamento de projetos individuais de pesquisa, incluindo uma rede e uma instância configurável, com suporte à escalabilidade de recursos, continuação em Código 16.

Esses templates foram projetados para atender às necessidades específicas dos alunos em atividades acadêmicas, mas a abordagem é flexível e pode ser adaptada para qualquer membro do campus que necessite de infraestrutura computacional. Professores, pesquisadores e equipes administrativas podem solicitar recursos personalizados, como instâncias, redes ou ambientes configurados, que serão provisionados de forma automatizada e organizada por meio de templates do Heat.

```

26 resources:
27   project_network:
28     type: OS::Neutron::Net
29     properties:
30       name: { get_param: project_name }
31
32   project_subnet:
33     type: OS::Neutron::Subnet
34     properties:
35       network_id: { get_resource: project_network }
36       cidr: { get_param: network_cidr }
37       gateway_ip: { get_param: gateway_ip }
38       dns_nameservers:
39         - 8.8.8.8
40         - 8.8.4.4
41
42   project_router:
43     type: OS::Neutron::Router
44     properties:
45       name: { str_replace: { template: "{project_name}-router", params: { project_name: {
46         ↪ get_param: project_name } } } }
47
48   router_interface:
49     type: OS::Neutron::RouterInterface
50     properties:
51       router_id: { get_resource: project_router }
52       subnet_id: { get_resource: project_subnet }
53
54   student_instance:
55     type: OS::Nova::Server
56     properties:
57       name: { str_replace: { template: "{project_name}-instance", params: { project_name: {
58         ↪ get_param: project_name } } } }
59       image: { get_param: image_id }
60       flavor: { get_param: flavor }
61       networks:
62         - network: { get_resource: project_network }
63
64 outputs:
65   network_info:
66     description: Informações da rede criada para o projeto
67     value:
68       network_name: { get_resource: project_network }
69       subnet_cidr: { get_param: network_cidr }
70
71   instance_info:
72     description: Informações da instância criada para o aluno
73     value:
74       instance_name: { get_attr: [student_instance, name] }
75       instance_ip: { get_attr: [student_instance, first_address] }

```

Código 16: Parte 2: Criação dos recursos do necessário para rodar as instâncias com a imagem escolhida em Código 15.

5

Conclusão

5.1 Panorama Geral e Viabilidade do Projeto

A proposta de implementação de uma nuvem privada no Instituto Federal de Brasília (IFB), detalhada ao longo deste trabalho, demonstrou-se promissora e possível para atender às demandas acadêmicas e administrativas da instituição. Baseada no OpenStack e utilizando *Kolla-Ansible* para a containerização dos serviços, a arquitetura aqui descrita destaca-se pela flexibilidade e escalabilidade, podendo se adaptar a cenários de maiores dimensões e complexidades. Este trabalho apresentou uma análise detalhada das tecnologias envolvidas, destacando o OpenStack como a solução mais adequada para atender às demandas específicas de escalabilidade.

5.1.1 Principais Resultados Alcançados

Devido às restrições de hardware, o projeto foi desenvolvido em um ambiente de demonstração, com *clusters* e *volumes* reduzidos. Mesmo assim, comprovou-se a viabilidade do modelo conceitual, evidenciando como os serviços (rede, armazenamento e orquestração) interagem e podem ser configurados de forma modular. A utilização de técnicas como *Hierarchical Multitenancy* e *Heat templates* mostrou-se eficaz para organizar projetos e pesquisas, permitindo o gerenciamento de recursos e facilitando a expansão conforme a aquisição de novos nós, junto a adição fácil ao cluster, apenas adicionando o IP ou DNS ao grupo de controle.

5.2 Alinhamento com os Objetivos do Estudo

Em relação aos objetivos estabelecidos no início da pesquisa, demonstrou-se que eles podem ser alcançados com o cluster OpenStack, tais como:

- **Otimização do Uso de Recursos:** Com o OpenStack, é possível compartilhar a mesma máquina para várias instâncias, eliminando a necessidade de dedicar um servidor exclusivo para cada aluno.

- **Fomento à Pesquisa e Inovação:** A infraestrutura proporcionada permite incentivar novas pesquisas, oferecendo de maneira rápida recursos para os alunos, podendo escalar de acordo com a complexidade da pesquisa.
- **Segurança e Controle de Dados:** Com o OpenStack, há controle total sobre os dados que transitam na nuvem, eliminando a necessidade de utilizar serviços externos para armazenamento.
- **Facilitação do Acesso a Serviços de Computação:** Utilizando templates como o Heat, é possível provisionar de maneira ágil a infraestrutura necessária para pesquisas.
- **Redução da Dependência de Infraestruturas Externas:** Não será mais necessário depender de infraestruturas externas, como o Google Colab. Em casos que exigem o uso de placas de vídeo, também é possível integrá-las ao Cluster OpenStack, desde que o hardware necessário seja adquirido no futuro.

5.3 Manutenção, Evolução e Monitoramento

Quanto à manutenção e à evolução do ambiente, a adoção de containers isolados reforçou a facilidade de atualizações, garantindo um processo menos suscetível a falhas e simplificando o gerenciamento de cada componente do OpenStack. Além disso, as ferramentas de monitoramento como (*Ceilometer*) possibilitam acompanhar o desempenho e o uso dos recursos em tempo real, fornecendo dados necessários para futuras otimizações e tomadas de decisão, além disso o próprio *Kolla-Ansible* nos entrega os novos containers já pronto com as novas versões e naturalmente com retrocompatibilidade, caso seja necessário uma atualização rápida.

5.4 Trabalhos Futuros

Como trabalho futuro, a partir do momento em que tivermos mais recursos de hardware, poderemos efetuar a construção de um cluster robusto, capaz de suprir as necessidades de infraestrutura no IFB. Adicionalmente, será possível explorar mais os templates Heat e as diversas formas de utilização da arquitetura para os membros do campus, juntamente com a divulgação desse novo serviço para a comunidade do IFB. Acredita-se que, ao disponibilizar uma infraestrutura de nuvem privada robusta, o campus poderá impulsionar projetos de ensino e pesquisa, abrindo espaço para inovações que contribuam significativamente para o desenvolvimento tecnológico.

- AMORIM, B. **Arquitetura OpenStack**. 2015.
- ARUTYUNOV, V. Cloud computing: its history of development, modern state, and future considerations. **Scientific and Technical Information Processing**, [S.l.], v.39, p.173–178, 2012.
- AWS. **Amazon Web Services Launches**. 2006.
- AWS. **Companies that migrated to AWS**. 2014.
- CHIRAMMAL, H.; MUKHEDKAR, P.; VETTATHU, A. **Mastering KVM Virtualization**. [S.l.]: Packt Publishing, 2016.
- CINDER, O. **Cinder Documentação**. 2024.
- CLOUD, B. G. **\$414 Billion in Profits can be Gained by Using Cloud for Business Growth**: infosys research. 2021.
- CLOUD, S. **Cloud Computing Statistics That Will Blow Your Mind**. 2023.
- COUTERPOINT, T. **Data Center CPU Market**: amd surpasses intel in share growth. 2023.
- CROSAS, M. **Cloud Dataverse**. 2017.
- DA SILVA, C. E. et al. Self-adaptive authorisation in OpenStack cloud platform. **Journal of Internet Services and Applications**, [S.l.], v.9, p.1–17, 2018.
- DATAMATION. **Cloud Computing Adoption**. 2017.
- DELTA, E. **How Many Companies Use Cloud Computing in 2024**. 2024.
- DOCKER. **Docker Documentation**. 2024.
- DONG, Y. et al. High performance network virtualization with SR-IOV. **Journal of Parallel and Distributed Computing**, [S.l.], v.72, n.11, p.1471–1480, 2012.
- HANUKAEV. **Common Use Cases & Benefits of OpenStack Technology for Universities**. Accessed: 2024-10-02, <https://openmetal.io/resources/blog/common-use-cases-benefits-of-openstack-technology-for-universities/>.
- HEAT, O. **Heat Documentação**. 2024.
- HILDRED, T. **The History of Containers**. 2015.
- IBM. **IBM Cloud Private with OpenShift**. Accessed: 2024-10-03, <https://www.ibm.com/docs/en/cloud-private/3.1.2?topic=environments-cloud-private-openshift>.
- IBM. **Time Sharing**. 2024.
- IBM. **O que é o Armazenamento em Bloco**. 2024.

- INSIGHTS, F. B. **Server Operating System Market Volume, Share & Industry Analysis**. Accessed: 2024-11-23, <https://www.fortunebusinessinsights.com/server-operating-system-market-106601>.
- KERRISK, M. **Namespaces in operation, part 1: namespaces overview**. 2013.
- KITCHENHAM, B. et al. Systematic literature reviews in software engineering—a tertiary study. **Information and software technology**, [S.l.], v.52, n.8, p.792–805, 2010.
- KUBERNETS. **Documentação Kubernetes**. 2024.
- KUMAR, R. et al. Open source solution for cloud computing platform using OpenStack. **International Journal of Computer Science and Mobile Computing**, [S.l.], v.3, n.5, p.89–98, 2014.
- LEE, K. **Why Use Linux for Servers in Enterprise Environments**. 2024.
- LIBIRT. **Libvirt Documentation**. Accessed: 2024-10-16, <https://libvirt.org/api.html>.
- MACHADO, T. L. D. **Linux Containers**. 2017.
- MALENGREAU, F. **UCLouvain Case Study VDI for 37,000 students with OpenNebula**. Accessed: 2024-10-02, <https://opennebula.io/blog/experiences/uclouvain-case-study-vdi-for-37000-students-with-opennebula/>.
- NEUTRON, O. **Neutron Architecture**. 2024.
- NOVA, O. **Nova Architecture**. 2024.
- OPENSTACK. **Documentacao Openstack**. 2024.
- OPENSTACK. **OpenStack Software**. 2024.
- PANASAS, I. **Object Storage Architecture White Paper**. Accessed: 2024-08-31, White paper.
- QIAN, L. et al. Cloud computing: an overview. In: CLOUD COMPUTING: FIRST INTERNATIONAL CONFERENCE, CLOUDCOM 2009, BEIJING, CHINA, DECEMBER 1-4, 2009. PROCEEDINGS 1. **Anais...** [S.l.: s.n.], 2009. p.626–631.
- REDHAT. **O que é Docker**. 2023.
- REDHAT. **OpenShift Documentation**. Accessed: 2024-10-16, https://docs.openshift.com/container-platform/4.17/welcome/learn_more_about_openshift.html.
- ROSADO, T.; BERNARDINO, J. An overview of openstack architecture. In: OF THE 18TH INTERNATIONAL DATABASE ENGINEERING & APPLICATIONS SYMPOSIUM. **Proceedings...** [S.l.: s.n.], 2014. p.366–367.
- SHAHZADI, S. et al. Infrastructure as a service (IaaS): a comparative performance analysis of open-source cloud platforms. In: IEEE 22ND INTERNATIONAL WORKSHOP ON COMPUTER AIDED MODELING AND DESIGN OF COMMUNICATION LINKS AND NETWORKS (CAMAD), 2017. **Anais...** [S.l.: s.n.], 2017. p.1–6.

- SIRETT, G. **University of São Paulo move to open source Apache CloudStack to power Latin Americas largest educational cloud | Case Studies**. Accessed: 2024-10-02, <https://www.shapeblue.com/university-of-sao-paulo-move-to-open-source-apache-cloudstack-to-power>
- STATS, I. L. **Total number of Websites**. 2018.
- SUSNJARA, S. **The advantages and disadvantages of private cloud**. 2013.
- SWITFT, O. **Swift Documentação**. 2024.
- TANENBAUM, A.; BOS, H. **Modern Operating Systems, 4th Edition**. [S.l.]: Pearson Higher Education, 2015.
- TECHTARGET. **TechTarget what is vCPU**. 2023.
- TROVE, O. **Trove Documentação**. 2024.
- VELICHKO, I. **What Is a Standard Container: diving into the oci runtime spec**. 2022.
- VOGEL, A. et al. Private IaaS clouds: a comparative analysis of opennebula, cloudstack and openstack. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED, AND NETWORK-BASED PROCESSING (PDP), 2016. **Anais...** [S.l.: s.n.], 2016. p.672–679.
- WEN, X. et al. Comparison of open-source cloud management platforms: openstack and opennebula. In: INTERNATIONAL CONFERENCE ON FUZZY SYSTEMS AND KNOWLEDGE DISCOVERY, 2012. **Anais...** [S.l.: s.n.], 2012. p.2457–2461.
- WIKIPEDIA. **Ioctl — Wikipedia, The Free Encyclopedia**. [Online; accessed 21-June-2024], <http://en.wikipedia.org/w/index.php?title=Ioctl&oldid=1219949704>.
- ZABALJÁUREGUI, M. Hardware assisted virtualization intel virtualization technology. **Unpublished Student Thesis**, [S.l.], 2008.