UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
PROGRAMA DE RESIDÊNCIA EM TECNOLOGIA DA INFORMAÇÃO

Implantação de Orquestrador de Containers para melhoria do ciclo de vida das aplicações na JFRN e avanço da cultura DevOps

Mariana Fonseca dos Santos

Natal-RN, Brasil
2023

Mariana Fonseca dos Santos

Implantação de Orquestrador de Containers para melhoria do ciclo de vida das aplicações na JFRN e avanço da cultura DevOps

Trabalho de Conclusão de Curso apresentado ao Programa de Residência em Tecnologia da Informação do Instituto Metrópole Digital da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do título de Especialista em Tecnologia da Informação. Área de Concentração: Infraestrutura e Redes

Orientador: Marcos César Madruga Alves Pinheiro

Natal-RN, Brasil
2023

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Santos, Mariana Fonseca dos.
Implantação de orquestrador de containers para melhoria do ciclo de vida das aplicações na JFRN e avanço da cultura DevOps / Mariana Fonseca dos Santos. - 2023.
52 f.: il.

Trabalho de Conclusão de Curso - TCC (graduação) - Universidade Federal do Rio Grande do Norte, Instituto Metrópole Digital, Programa de Residência em Tecnologia da Informação. Natal, RN, 2023.
Orientação: Prof. Dr. Marcos César Madruga Alves Pinheiro.

1. Desenvolvimento de software - TCC. 2. Kubernetes - TCC. 3. Container - TCC. 4. Orquestração - TCC. I. Pinheiro, Marcos César Madruga Alves. II. Título.

RN/UF/BCZM CDU 004.4

Elaborado por Fernanda de Medeiros Ferreira Aquino - CRB-15/301

Mariana Fonseca dos Santos

Implantação de Orquestrador de Containers para melhoria do ciclo de vida das aplicações na JFRN e avanço da cultura DevOps

Trabalho de Conclusão de Curso apresentado ao Programa de Residência em Tecnologia da Informação do Instituto Metrópole Digital da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do título de Especialista em Tecnologia da Informação. Área de Concentração: Infraestrutura e Redes

Trabalho aprovado. Natal-RN, Brasil, 31 de outubro de 2023:

Prof. Dr. Marcos César Madruga Alves Pinheiro
Orientador

Prof. MSc André Luiz da Silva Solino
Examinador

Prof. MSc Wellington Silva de Souza
Examinador

Natal-RN, Brasil
2023

Agradecimentos

Início agradecendo a Deus e a Nossa Senhora, aos quais recorri diversas vezes, não pedindo uma solução pronta, mas que me dessem forças e me capacitasse durante todo o trabalho.

Aos meus pais, Cristina e Manoel, que sempre me apoiam, às vezes sem nem entender, e são pilares inabaláveis aos quais posso me sustentar.

À minha tia Ana e prima Camila, assim como aos demais familiares que estão ao meu lado a todo momento.

Meus colegas de residência, Emilly e Yan, que foram essenciais na minha jornada na Residência e no desenvolvimento deste trabalho, os quais pude compartilhar os apertos e as felicidades.

Os residentes Kevin, Weverson, Madson e Humberto, mesmo sendo de outro time, pudemos encontrar pontos em comum e sempre foram prestativos.

A toda equipe do NTIC da JFRN, em especial aos servidores da Infra: Murillo, Chico, Valdez, Wellington e demais. Seus conhecimentos e paciência conosco durante todo o período da residência com certeza foram imprescindíveis para que finalizássemos com excelência e evoluíssemos como profissionais.

Ao professor Madruga, que já era um orientador desde nosso primeiro contato. Sua enorme paciência e ainda maior sabedoria, e sua facilidade em passar seus conhecimentos sempre em sua humildade, me trouxeram uma maior vontade de evoluir como profissional e voltar a gostar das atividades acadêmicas.

“Ser ruim em alguma coisa, é o primeiro passo pra ser bom em alguma coisa.”

(Hora de Aventura)

Resumo

A implantação de um orquestrador de *containers* é uma iniciativa essencial para otimizar o ciclo de vida das aplicações e impulsionar a cultura DevOps nas organizações. No cenário tecnológico atual, as práticas ágeis e a automação de processos tornaram-se imperativas para garantir a eficiência e a agilidade no desenvolvimento de *software*. Este trabalho tem como objetivo implantar um orquestrador de *containers* na JFRN, por meio da utilização de um *cluster* Kubernetes em conjunto com o *deploy* de uma aplicação conteinerizada desenvolvida pela instituição. Para isso, o o desenvolvimento é iniciado por um breve estudo comparativo entre plataformas de orquestração de *containers* conhecidas, o Docker Swarm e Kubernetes, a fim de justificar a escolha do Kubernetes. São explorados os desafios da criação do *cluster* e a migração da aplicação para a nova arquitetura, e realizados experimentos para a simulação da escalabilidade automática a partir de uma demanda gerada e o tratamento de falhas pelo *cluster*. A análise comparativa inicial auxiliou na formação de um embasamento para justificar a escolha pelo uso do Kubernetes. Os resultados dos experimentos realizados corroboram para a solidificação das vantagens de se utilizar um orquestrador de containers, ao entregar uma infraestrutura resiliente, automatizada, escalável e segura disponibilizando ao usuário final uma aplicação mais confiável.

Palavras-chave: Container. Kubernetes. Orquestração.

Abstract

Deploying a container orchestrator is an essential initiative to optimize the life cycle of applications and boost the DevOps culture in organizations. In today's technological scene, agile practice and process automation have become imperative to ensure efficiency and agility in software development. The goal of this work is to deploy a container orchestrator in JFRN, by making use of a Kubernetes cluster in collaboration with the deployment of a container-based application developed by the institution. To be able to achieve these goals, the work starts with a brief comparative study between known container orchestration platforms, Docker Swarm and Kubernetes, to justify the Kubernetes use choice. The challenges of cluster creation and migrating the application to the new way of deployment, are explored. For the simulation of auto-scaling from a generated demand and how the cluster will handle failure, an experiment will be made. The initial comparative analyses assisted in the development of a basis to justify the Kubernetes use choice. The experiment results contributed to solidifying the advantages of using a container orchestrator, with being able to deliver a resilient, automated, scalable, and secure infrastructure that gives the end user a much more trusted application.

Keywords: Container. Kubernetes. Orchestration..

Lista de ilustrações

Lista de abreviaturas e siglas

CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
HPA	Horizontal Pod Autoscaling
IMD	Instituto Metrópole Digital
JFRN	Justiça Federal do Rio Grande do Norte
VMM	Virtual Machine Manager
UFRN	Universidade Federal do Rio Grande do Norte
SO	Sistema Operacional
TI	Tecnologia da Informação

Sumário

1 Introdução

Na era da transformação digital, as organizações buscam constantemente adaptar-se às crescentes demandas por serviços mais ágeis, escaláveis e resilientes. No centro desse movimento está a necessidade de infraestruturas de TI flexíveis que possam acomodar o rápido desenvolvimento e implantação de aplicações. A Justiça Federal do Rio Grande do Norte (JFRN) não é exceção a esta tendência.

A busca pela excelência na entrega de software é respaldada por uma série de ferramentas e práticas, das quais o DevOps é um pilar fundamental. A integração contínua, a entrega contínua e a automação de infraestrutura são conceitos centrais para a cultura DevOps, que busca encurtar os ciclos de desenvolvimento e implementação, promovendo a colaboração entre equipes de desenvolvimento e operações.

Nesse contexto, a utilização de orquestradores de containers como Kubernetes se destaca como uma solução poderosa para a implantação de aplicações de forma consistente, escalável e altamente disponível.

A necessidade de uma abordagem mais ágil e eficiente no desenvolvimento de software também é corroborada por Smith (2017), que enfatiza a importância da entrega contínua para automatizar o processo de implantação e garantir que as mudanças sejam entregues rapidamente e com qualidade.

A importância da orquestração de containers na gestão de aplicações é corroborada por Gill et al. (2018), que discutem como o DevOps pode ser aplicado a sistemas de gerenciamento de informações. A automação e a orquestração são componentes-chave para a eficácia desses sistemas.

Este trabalho tem como objetivo explorar a implantação de um orquestrador de *containers* na Justiça Federal do Rio Grande do Norte (JFRN), para facilitar e padronizar o gerenciamento do ciclo de vida de aplicações composta por microsserviços conteinerizados, por meio da utilização de um *cluster* Kubernetes em conjunto ao deployment teste de uma aplicação desenvolvida pela Seção de Sistemas da Justiça Federal do Rio Grande do Norte.

Para alcançar esses objetivos, é necessário iniciar com uma breve avaliação da escolha entre os dois motores de orquestração de *containers* que podem ser comparados na mesma categoria, o Docker Swarm e o Kubernetes, justificando assim o motivo de optar pelo Kubernetes neste trabalho. Será explorado os desafios para o provisionamento do Kubernetes na infraestrutura da JFRN em conjunto as adaptações necessárias para migração da aplicação Hígia do seu modelo convencional em único host. Por fim, serão realizados experimentos a fim de testar a resiliência e capacidade de escalabilidade do *cluster* Kubernetes.

Essa pesquisa se justifica pois, na era atual de transformação digital, as organizações estão constantemente em busca de soluções que ofereçam maior eficiência, escalabilidade e

resiliência em suas operações de TI. A conteinerização, e consequentemente, a orquestração de containers, surge como uma abordagem promissora para atender a essas necessidades.

1.1 Organização do trabalho

Esta seção apresenta como está organizado o trabalho, sendo descrito, portanto, do que trata cada capítulo.

No Capítulo 2 são abordados os principais conceitos teóricos que dão base para o desenvolvimento do trabalho e é apresentado o estudo comparativo para a escolha do orquestrador de *containers*;

No Capítulo 3 são descritos os passos necessários para desenvolvimento do trabalho com a criação da infraestrutura, criação e configuração do *cluster* e adaptação da aplicação alvo;

No Capítulo 4 são descritos os experimentos que serão feitos e o que é esperado de cada um;

No Capítulo 5 as execuções e resultados dos experimentos são explorados e discutidos;

No Capítulo 6 são expostas as considerações finais sobre o trabalho e apresentadas propostas de trabalhos futuros.

2 Referêncial Teórico

Neste capítulo são apresentados os principais conceitos que dão a base teórica para o desenvolvimento do trabalho e entendimento da arquitetura utilizada.

2.1 Virtualização

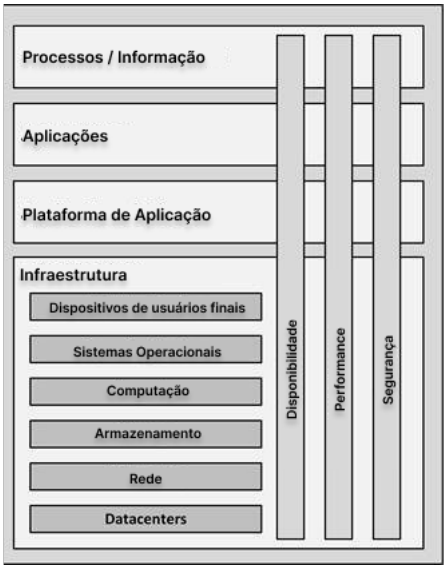
A infraestrutura de Tecnologia da Informação (TI) em suas décadas iniciais era relativamente simples. Ao passo que aplicações foram se desenvolvendo e tornando-se mais complexas e avançadas, o hardware base para essas aplicações ficaram mais rápidos.

Porém, em anos mais recentes essa infraestrutura se tornou mais complicada e Laan (2017) relaciona ao fato do surgimento novos tipos de aplicações como o comércio digital, *Big Data*, Internet das Coisas e entre outras, que requerem um maior nível de sofisticação da infraestrutura para entregar os serviços de forma mais rápida, segura, altamente escaláveis e disponíveis 24/7.

As organizações e empresas precisam se adaptar a esse modelo mais ágil e flexível de entrega de serviços, para isso, uma definição de arquitetura de infraestrutura é preciso.

Laan (2017) propõe um modelo composto por "blocos de construção" separados em diferentes categorias como o gerenciamento de processos e informações, gerenciamento de aplicações e a própria infraestrutura, como ilustrado na Figura 1. A virtualização faz parte do escopo de Compute (computação) dentro da Infraestrutura.

Figura 1 – Ilustração do modelos de infraestrutura e seus blocos



Fonte: Laan (2017) e modificado pelo autor

Soyinka (2015) lista algum dos motivos técnicos e não técnicos pelos quais as empresas acham a adoção da virtualização um formato atrativo dentro de sua infraestrutura, sendo eles:

- melhoria na disponibilidade dos servidores e aplicações, reduzindo o tempo em que servidores ficam indisponíveis;
- oferece um melhor suporte a multi plataformas;
- possibilidade de entregar um ótimo ambiente para testes e resoluções de problemas sem afetar o que já está funcionando;
- economia de custos na compra e manutenção de hardware;
- visível retorno no investimento da implantação de virtualização;
- redução do impacto ambiental pela redução do número de servidores físicos utilizados;

A virtualização não é um conceito recente. Os seus primeiros passos podem ser datados no início da década de 1960, quando o cientista inglês Christopher Strachey publicou seu artigo "Time Sharing Processing in Large Fast Computers". Neste artigo, Strachey tratava da multiprogramação em tempo compartilhado, método onde múltiplos usuários com diferentes programas interagem quase que simultaneamente com a CPU. (The Editors of Encyclopedia Britannica, 2021)

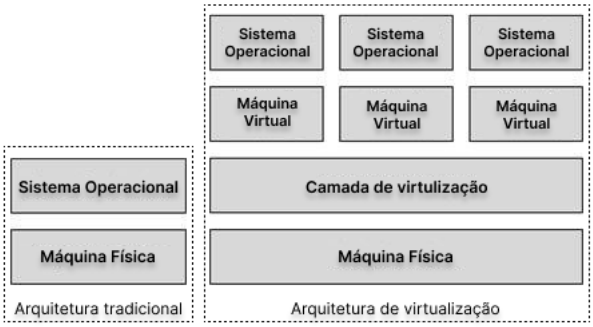
Ainda na década de 1960, teve seu desenvolvimento e implementação impulsionados com a introdução das máquinas virtuais (VM) pela IBM em seu sistema experimental M44/44X. Laan (2017) então complementa que desde 1972, todos os *mainframes* têm utilizado a virtualização a partir do lançamento do primeiro sistema operacional (SO) de máquina virtual, o VM/370.

De forma simplificada, em um modelo mais convencional da arquitetura de computadores, o sistema operacional interage diretamente com o *hardware*. Laan (2017) conceitua virtualização como a "introdução de uma camada de abstração entre o hardware do computador físico e o sistema operacional que está usando este hardware". A diferença entre o modelo convencional e virtualização pode ser observada com a Figura 2.

Segundo Silberschatz, Galvin e Gagne (2021), a implementação da virtualização envolve diversos componentes, como:

- O ***host***, ou hospedeiro, o sistema fundamental de *hardware* que irá executar as máquinas virtuais;
- O ***Virtual Machine Manager*** (VMM), ou ***hypervisor***, responsável por criar executar as máquinas virtuais ao disponibilizar uma interface idêntica ao *host*;

Figura 2 – Ilustração dos modelos de arquitetura to tipo tradicional e virtualização



Fonte: Laan (2017) e modificado pelo autor

Na opinião de Veras (2011), para entender a virtualização é necessário "entender que ela consiste em estender ou substituir um recurso ou uma interface existente por outro, de modo a imitar um comportamento."e isso é realizado pela camada de *software*, o *hypervisor*, que entrega um ambiente completo em que é possível a coexistência de um sistema operacional e diversos processos. Dessa forma, um único *hardware* consegue executar diversas máquinas virtuais, cada um com seu sistema operacional (chamado de *guest*, ou hóspede). Os *hypervisors* são categorizados em tipos, Veras (2011) define os mais notáveis em tipo 1 e tipo 2.

O tipo 1 executa diretamente no *hardware* do servidor hospedeiro e as máquinas virtuais são postas sobre ele. Seu principal papel é compartilhar os recursos do *hardware* com as máquinas virtuais, dando a ilusão que esses recursos são privativos de cada uma. A Figura 3a ilustra a hierarquia de camadas do tipo 2. São referência nessa categoria o VMware ESXi Server e o Hyper-V da Microsoft. (VERAS, 2011)

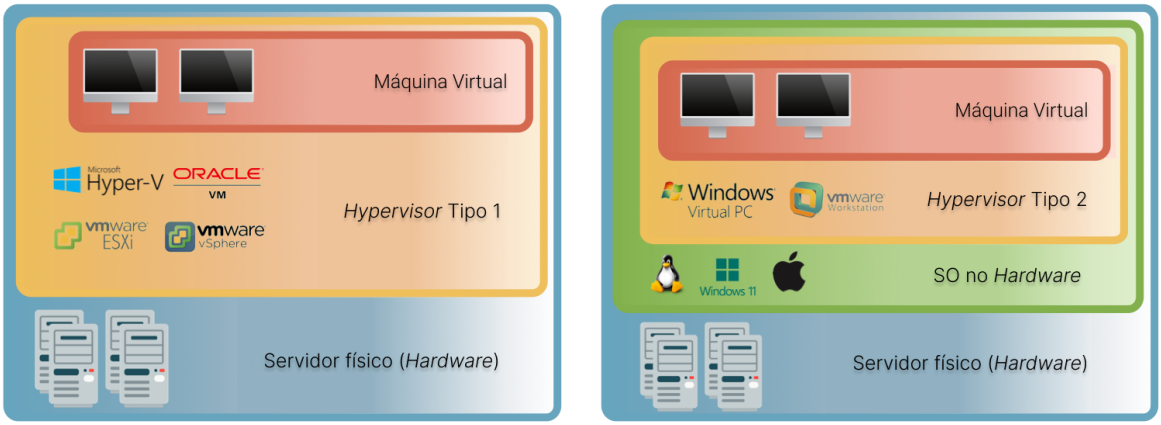
O tipo 2 são aplicações que executam sobre um sistema operacional mas disponibilizam características de uma VMM para esse SO hospedeiro. A Figura 3b ilustra a hierarquia de camadas do tipo 1. São exemplos desse tipo de aplicação o Oracle VirtualBox, VMware Workstation e Java Virtual Machine (JVM).

2.2 Containers

Por décadas, a adoção de máquinas virtuais (VMs) tem sido amplamente reconhecida por seu papel na otimização do uso de recursos de TI. No entanto, é notável que, ao encapsular um servidor completo em uma máquina virtual, a demanda de CPU e RAM necessária para que uma VM funcione de maneira eficiente pode sobrecarregar o servidor físico hospedeiro, resultando na limitação da quantidade de VMs que podem ser criadas. (VMWARE, 2022)

O uso da tecnologia de *containers*, que se popularizou com a plataforma Docker,

Figura 3 – Tipos de *hypervisors*



.

(a) Tipo 1 (b) Tipo 2

Fonte: Simic (2019) e modificado pelo autor

aparece como resposta à necessidade de uma maneira de implantar e executar aplicações de forma a melhor utilizar os recursos disponíveis, encapsulando as aplicações e suas dependências sem que haja interferência e incompatibilidade de outras aplicações executando no mesmo *host*. Capacita aos times de Operação uma maior agilidade e equipara seu ciclo de lançamento das aplicações ao ciclo de constantes mudanças feitas pelos times de Desenvolvimento.

Container não é um conceito novo e nem criado pelo Docker, é um termo que passou a ser utilizado com o lançamento do Sistema Operacional *Sun's Solaris 10* e os *Solaris containers*, em 2005, sendo uma evolução do conceito de *jail* (cadeia ou jaula) explorado nos sistemas tipo UNIX, para definir um ambiente tipo "jaula" onde os programas "enjaulados" executados nesse ambiente, tem limitações quanto aos recursos que podem acessar. (NICKOLOFF; KUENZLI, 2019).

Em comparação a VMs, um *container* é mais leve e não possui o mesmo *overhead* de recursos, possibilitando assim a execução de um maior número de aplicações em um mesmo *hardware*. (LUKSA, 2018).

O funcionamento de um *container* se difere de uma VM na forma como as aplicações executadas por eles fazem chamadas de sistemas. Um *container* faz todas as chamadas diretamente do *kernel* ao SO do *host*, diferente de uma VM, onde a aplicação faz chamadas ao *kernel* do SO do *host* hospedeiro, e então o *kernel* realiza as instruções na CPU do *host* pelo *hypervisor*. (NICKOLOFF; KUENZLI, 2019)

O isolamento de processos realizado pelo *container* se baseia em dois mecanismos importantes de sistemas tipo Linux: *Linux Namespaces* e *Linux Control Groups* (*cgroups*). Em resumo, podemos definir cada um:

- ***Linux Namespaces***: trabalha para que cada processo somente enxergue sua parte

do sistema (arquivos, processos, interface de rede, *host*name e etc).

- ***Linux Control Groups*** (*cgroups*): responsável por limitar o quanto de recursos um processo pode utilizar. Por exemplo: CPU, memória, largura de banda e entre outros.

Por mais poderosos que os *containers* sejam, eles não são uma solução por si só, e para serem usados efetivamente é necessário estarem inseridos em uma plataforma que forneça ferramentas para possibilitar a criação, implantação, execução, orquestração e segurança das aplicações em *containers*. (DUNCAN; OSBORNE, 2018)

As plataformas de *containers* podem existir de várias formas, como:

1. *Container Engine* (Motor de container), que tem como exemplo o Docker, LXC (Linux *containers*) e rkt;
2. *Container Orchestration* (Orquestrador de container), que tem como referências Kubernetes e Docker Swarm;
3. Plataformas Gerenciadas de Container, geralmente voltadas para soluções em nuvem como a Amazon EKS eGoogle Kubernetes Engine (GKE). (AQUA, 2021)

Container Engine (Motor de container), que tem como exemplo o Docker, LXC (Linux *containers*) e rkt. Container Orchestration (Orquestrador de container), que tem como referências Kubernetes e Docker Swarm. Plataformas Gerenciadas de Container, geralmente voltadas para soluções em nuvem como a Amazon EKS eGoogle Kubernetes Engine (GKE). (AQUA, 2021)

2.2.1 Docker

O Docker é uma plataforma de código aberto que se tornou um dos principais impulsionadores da popularização da tecnologia de *container*. Desenvolvido pela Docker, Inc., ele fornece uma maneira eficiente de empacotar, distribuir e executar aplicativos em ambientes isolados chamados *container*. Cada *container* é uma instância leve e independente que inclui não apenas o código do aplicativo, mas também todas as dependências necessárias para sua execução.

Essa abordagem resolve muitos dos desafios associados à implantação de aplicativos em diferentes ambientes, garantindo que as aplicações funcionem da mesma forma em qualquer lugar. De acordo com Caballer et al. (2018), o Docker tornou-se uma peça fundamental da arquitetura de microsserviços, na qual aplicativos são divididos em componentes menores que podem ser implantados em contêineres independentes. Essa abordagem permite uma maior flexibilidade e escalabilidade no desenvolvimento e na implantação de aplicativos.

O Docker tem uma comunidade ativa e uma ampla base de usuários, tornando-se uma das principais ferramentas para o desenvolvimento e implantação de aplicativos em contêineres. Com sua crescente adoção, ele está moldando a forma como as equipes de desenvolvimento e operações colaboram na entrega de software de maneira mais rápida e eficiente.

Definida como "A plataforma de execução de *container* padrão da indústria.", a *Docker Engine* é peça fundamental do ecossistema Docker. Utiliza uma arquitetura do tipo cliente-servidor, que realiza as tarefas e fluxos responsáveis pela construção, envio e execução de aplicativos em *container* usando os componentes e serviços do Docker. (MELL, 2022)

A *Docker Engine* é composta pelo Docker *daemon*, um processo chamado *dockerd* executado no servidor e que é responsável por escutar as requisições da Docker API e gerenciar os objetos Docker. A interação com o *dockerd* é feita pelo Docker *client*, a interface de linha de comando do lado do cliente, chamada *docker*. (DOCKER, 2020)

É preciso definir os objetos Docker e outros componentes que fazem parte do ecossistema e possibilitam o empacotamento, distribuição e execução das aplicações conteinerizadas. De interesse para este trabalho são eles: imagens, *container*, volumes e *registries*.

A imagem é um "pacote" binário que encapsula todos os arquivos necessários para a execução da aplicação no *container* e pode ser disponibilizada somente localmente ou armazenada em um repositório para ser compartilhada.

Um *container* é a instância em execução de uma imagem Docker, sendo um processo isolado tanto do *host* quanto de outros processos. Também só pode acessar os recursos alocados a ele. Tem como característica ser efêmero, ou seja, ao ser destruído por qualquer motivo, seus dados e modificações que possam ter sido feitas não persistem na memória, a menos que estejam associados a um volume. Essa característica pode ser vista como um benefício, pois traz agilidade na resolução de problemas e não deixa "lixo" no sistema operacional do *host*.

Os volumes, são uma maneira de armazenar e persistir dados que podem ser usados e gerados pelo *container* durante sua execução.

O *Docker Registry* é um repositório onde são armazenadas as imagens criadas e facilita o compartilhamento delas entre pessoas e computadores, podendo ser um repositório público como o *Docker Hub* ou privado para acesso somente dentro de uma empresa, por exemplo.

2.3 Orquestração de Container

Quando se pensa em criar aplicações robustas, que possuam alta disponibilidade e são escaláveis, é necessário que seja possível lançar essas aplicações conteinerizadas

entre múltiplos servidores. Apesar do formato de *container* Docker prover uma forma interessante de se construir aplicações, ele acaba se limitando ao *deploy* em apenas um único *host* e não oferece por si só o que seria necessário para implantar e gerenciar diversos *containers* que trabalham em conjunto e escalam com o crescimento da demanda.

Com esse cenário e a crescente evolução na utilização de *containers*, é introduzido o que se chama de orquestração de *containers*, uma maneira de automatizar o processo de provisionamento, implantação e configuração de rede. Isso proporciona escalabilidade, disponibilidade, gerenciamento de recursos, balanceamento de carga e gerenciamento do ciclo de vida dos *containers*. (REDHAT, 2022)

Os principais competidores do mercado de orquestração de *containers* são: Kubernetes, desenvolvido pela Google e Docker Swarm, que nativamente faz parte do "ecossistema" Docker.

2.3.1 Docker Swarm

O *Docker Swarm* foi introduzido no Docker 1.12 em 2016. Desde então se tornou uma ferramenta interessante para gerenciar e orquestrar *containers* em ambientes de produção. Ele permite a criação e o gerenciamento de *clusters* de *containers* Docker, simplificando a implantação, escalabilidade e alta disponibilidade de aplicativos em *containers*.

Com o *Docker Swarm*, é possível distribuir *containers* em diversos nós de maneira eficiente, monitorar o estado dos serviços e garantir que as aplicações estejam sempre disponíveis, mesmo em caso de falhas em algum nó do cluster. É uma solução útil para implementações simples de orquestração de *containers*, especialmente para equipes que já estão familiarizadas com o Docker, oferecendo uma transição suave para ambientes de *containers* em escala.

Seu funcionamento é baseado em múltiplas instâncias do *Docker Engine*, distribuídas entre diversos *hosts* formando um *cluster* trabalhando em "modo *swarm*". A estes *hosts* podem ser atribuídos a papéis de *manager* (gerente) ou *worker* (trabalhador), podendo assumir os dois papéis ao mesmo tempo ou só um, e o *deploy* de um *container* passa agora a ser chamado de *service* (serviço). (DOCKER, 2016)

Ao se criar um *service*, é definido um estado esperado de réplicas, rede, recursos computacionais e armazenamento alocados, portas a serem expostas e entre outras configurações. O nó *manager* se responsabiliza e trabalha para que esse estado seja mantido.

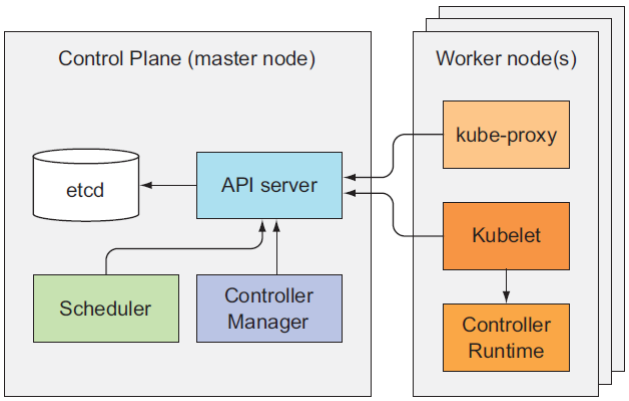
2.3.2 Kubernetes

Kubernetes é uma plataforma de código aberto para orquestração de *containers* entre múltiplos *hosts*, que tem como objetivo gerenciar serviços e aplicações conteinerizadas, facilitando tanto a automação como a configuração declarativa, disponibilizando descoberta de serviço, balanceamento de carga, automatização de novos lançamentos e

reversão, gerenciamento de recursos, a capacidade de "autocura", gerenciamento de secrets e configurações. (KUBERNETES, 2019a)

Existem uma série de componentes que constroem e fazem o Kubernetes na visão da arquitetura. Partindo do princípio básico que ao fazer o *deploy* do Kubernetes, tem-se um *cluster* que é composto por um grupo de máquinas (*hosts*) que podem ser categorizados em nós do *Control Plane* (referido comumente como nós master) e nós *workers*, que possuem componentes que os diferenciam em relação ao seu papel dentro do *cluster*. Essa arquitetura é ilustrada na Figura 4, exemplificando a organização e seus componentes, que serão explorados nas próximas seções.

Figura 4 – Diagrama com os componentes do Control Plane e nós Worker



Fonte: Luksa (2018)

Mudando para a visão das aplicações que são executadas no Kubernetes, essas que são denominadas *workloads* (cargas de trabalho), temos um conjunto de objetos e recursos do Kubernetes para o gerenciamento desses *workloads*. A base desses objetos é o *Pod*, a menor unidade escalável e implantável dentro do Kubernetes, onde um ou mais *containers* são executados. De forma que a quantidade de aplicações cresce, se torna impraticável gerenciar as configurações individuais de cada *Pod* e seu ciclo de vida, dessa forma, são introduzidos outros objetos responsáveis por esse gerenciamento como o *Service*, *Deployment*, *ConfigMap*.

2.3.2.1 Control Plane

O *Control Plane* é a parte responsável por controlar e fazer funcionar o cluster, sendo formado por um conjunto de componentes que irão realizar decisões globais dentro do cluster, sendo esses o Kubernetes API Server, o Scheduler, o Controller Manager e etcd. Esses componentes possibilitam tarefas como o gerenciamento de pods, realizar o controle de replicação, gerenciar services, nós e outros componentes do ambiente Kubernetes.

O *Kubernetes API Server* (*kube-apiserver*) é componente principal do *Control Plane* por ser responsável por expor a API do Kubernetes, garantindo a comunicação dos demais componentes já que eles não se comunicam diretamente. Servindo de *front-end* para que os diversos componentes tanto do *Control Plane* como nós *worker* possam realizar suas tarefas fazendo chamadas à essa API, realizando a validação e configuração de informações para os objetos da API, como *Pods* e *Services*. (RENSIN, 2015)

O *etcd* é uma espécie de banco de dados consistente e de alta disponibilidade, no formato chave-valor, utilizado para armazenar todos os dados do cluster Kubernetes.

O *Scheduler* (*kube-scheduler*) é um processo que atribui *Pods* aos Nós, determinando se um nó é válido ou não para receber um *Pod* que esteja na *scheduling queue* (fila de agendamento) de acordo com as limitações e recursos disponíveis.

O *Controller Manager* (*kube-controller-manager*) realiza funções a nível de *cluster* como a replicação de componentes, monitorar os nós *workers*, tratar falhas dos nós em geral e etc.

2.3.2.2 Nó Worker

Assim como o *Control Plane*, o Nó *Worker* possui elementos específicos que o caracterizam e rodam em todos os nós *worker* que compõem o *cluster*, sendo responsável por efetivamente executar a aplicações implantadas. Esses elementos são o *kubelet*, *Kubernetes Service Proxy* e o *container runtime*.

Sendo o agente primário que executa dentro um nó *worker*, o *kubelet* é responsável por garantir que os *containers* estão rodando dentro de um *Pod* da forma correta e saudável, fazendo isso a partir de uma abstração chamada *PodSpec* que é um arquivo do tipo YAML ou JSON que descreve um *pod*.

O *Kubernetes Service Proxy* (*kube-proxy*) é um *proxy* de rede do Kubernetes que roda em cada nó e mantém regras de rede dentro de cada nó e possibilita que conexões de rede sejam estabelecidas para os *Pods* a partir de sessões de rede dentro e fora do *cluster*.

Por fim, o *container runtime* é uma peça chave dentro do Kubernetes para que os *containers* sejam executados de forma efetiva, sendo responsável por gerenciar a execução e o ciclo de vida dos *containers* dentro do ambiente.

2.3.2.3 Pod

Em uma orquestração de *containers*, como o Kubernetes, o "*Pod*" é um conceito fundamental. De acordo com Kubernetes (2019b), um *Pod* é a unidade mais simples em que as cargas de trabalho são colocadas. Ele pode conter um ou mais *containers*, sendo esses *containers* compartilhando o mesmo espaço de rede e armazenamento, o que possibilita que eles se comuniquem facilmente e acessem os mesmos volumes.

O conceito de *Pod* é fundamental para entender como o Kubernetes lida com a orquestração de *containers*. Beltre et al. (2019) destaca que, ao agrupar *containers* em

Pods, o Kubernetes oferece uma abordagem flexível para o gerenciamento de aplicativos, permitindo que vários *containers* trabalhem juntos de maneira coesa, compartilhando recursos e informações.

2.3.2.4 Deployments

Em Kubernetes, um *Deployment* é um recurso que permite gerenciar de forma declarativa o estado desejado de um *Pod*, sendo possível realizar as criações, alterações ou novas implantações para os pod's. Através dele é possível escalar as aplicações executando dentro do *cluster*, podendo aumentar sua capacidade ou diminuir dependendo do estado da aplicação. (KUBERNETES, 2019c)

Na criação de um *Deployment* são definidos os parâmetros de implementação ou atualização, dos quais incluem: número de *pods* que devem ser criados, número mínimo que deve estar disponível, entre outros.

2.3.2.5 Services

No Kubernetes, *services* (serviços) são uma maneira abstrata de expor um aplicativo em execução em um conjunto de *pods*, implementando-os como um serviço de rede. (KUBERNETES, 2017)

Os *pods* são mortais, eles nascem e caso morram outros são colocados em seu lugar, devido a esse fator, para manter os processos ali executados de maneira consistentes são definidos os serviços. Esses serviços podem ser expostos de três formas diferentes, que no Kubernetes são denominados *ServiceTypes*, sendo eles:

- ***ClusterIp***: Expõe o serviço em um IP interno do *cluster*. Ao optar por esse *ServiceType* o serviço ficará acessível apenas de dentro do *cluster*;
- ***NodePort***: Expõe o serviço no IP de cada nó *worker* em uma porta estática;
- ***LoadBalancer***: Expõe o serviço externamente usando o balanceador de carga disponível na arquitetura;

2.3.2.6 ConfigMap

Ao implantar aplicativos em ambientes complexos, é essencial gerenciar variáveis de configuração, *secrets* e outros parâmetros que podem variar de ambiente para ambiente, como desenvolvimento, teste e produção.

O Kubernetes oferece uma solução eficaz para esse desafio por meio do recurso chamado *ConfigMap*. Um *ConfigMap* é um objeto Kubernetes projetado para armazenar dados de configuração em formato de pares chave-valor, que pode ser montado como volumes em *containers*.

De acordo com Kubernetes (2020a), os *ConfigMaps* permitem que os desenvolvedores isolem as configurações de seus aplicativos do código-fonte e forneçam uma maneira de atualizar essas configurações sem precisar recriar *containers* ou modificar os arquivos de configuração no sistema de arquivos dos contêineres.

2.4 Kubernetes vs *Docker Swarm*

A escolha entre Kubernetes e Docker Swarm para a orquestração de containers é uma decisão importante para equipes de desenvolvimento e operações. Ambas as soluções têm seus méritos, mas para justificar a escolha do Kubernetes, é fundamental realizar um comparativo entre essas duas opções.

De acordo com ThinkSys (2022), o Docker Swarm é uma ferramenta de orquestração de containers desenvolvida pela Docker, Inc. Ele é projetado para ser simples de configurar e usar, o que o torna uma escolha atraente para equipes que desejam começar rapidamente. No entanto, essa simplicidade também pode limitar sua capacidade de escalabilidade e personalização em ambientes de produção mais complexos.

Por outro lado, o Kubernetes, como destacado por Beltre (2019), é uma plataforma mais robusta e escalável. Ele fornece uma arquitetura flexível que permite a gestão eficaz de containers em escala, tornando-o ideal para implantações em grande escala. A facilidade de escalabilidade é uma das principais razões pelas quais muitas organizações escolhem o Kubernetes.

O suporte de comunidade também é um fator a ser considerado. Conforme Microsoft (2020) aponta, o Kubernetes possui uma comunidade ativa e uma ampla base de usuários, o que resulta em suporte constante, atualizações regulares e um vasto ecossistema de ferramentas de terceiros. O Docker Swarm, embora seja uma solução válida, não desfruta da mesma quantidade de contribuições e desenvolvimento de terceiros.

Além disso, a capacidade de gestão de microsserviços, de acordo com Castillo (2022), é um ponto forte do Kubernetes. Ele oferece recursos avançados para implantações de microsserviços, incluindo balanceamento de carga, recuperação automática e escalabilidade automática. Isso é fundamental para organizações que adotam arquiteturas de microsserviços.

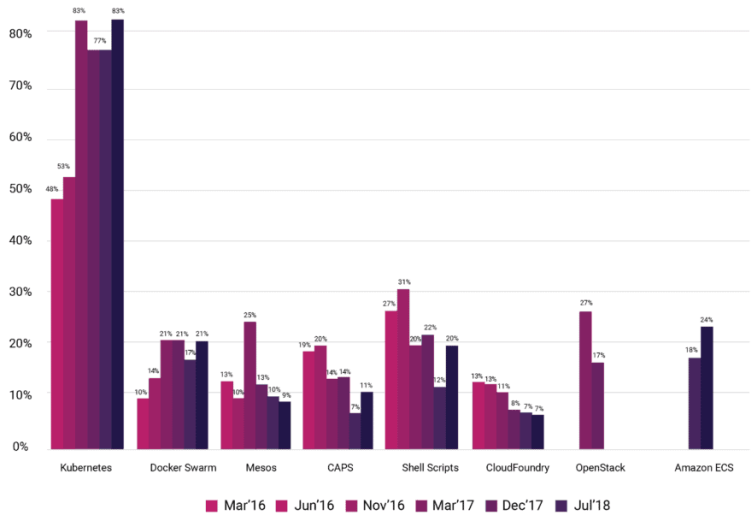
Outro aspecto a ser considerado é a flexibilidade do Kubernetes e seus modelos de implantação. Conforme Gordon (2022) destaca, o Kubernetes fornece uma ampla gama de recursos que podem ser personalizados para atender às necessidades específicas de um aplicativo ou organização. Isso permite maior flexibilidade na implantação e gestão de containers.

A CNCF (2015), realiza pesquisas anuais para melhor entender o estado em que se encontra a adoção de orquestradores de containers de forma geral.

Em sua pesquisa de 2018, Barnard (2018) mostra que entre os participantes da

pesquisa o Docker Swarm tinha uma porcentagem de adoção de 21%, contra 83% do Kubernetes, que já vinha liderando desde pesquisas anteriores (anos de 2016 e 2018), como visto no gráfico da Figura 5. O crescimento da adoção e solidificação do Kubernetes como líder no mercado de orquestração de *containers* é fomentado nas pesquisas dos anos seguintes.

Figura 5 – Gráfico de adoção de orquestradores de *containers* entre 2016 e 2018



Fonte: (BARNARD, 2018)

A CNCF (2022) considera 2021 o ano em que o Kubernetes atravessou o "abismo" se tornando a tecnologia convencional, onde 96% dos entrevistados estavam usando ou pelo menos avaliando o uso do Kubernetes. A partir das pesquisas de 2019 realizadas pela CNCF, não foram encontrados registros sobre o Docker Swarm, podendo se deduzir que sua adoção é irrisória se comparada ao Kubernetes.

Outras empresas de importância do mercado também realizam e divulgam suas pesquisas sobre o assunto. A Datadog (2022), uma conhecida solução de monitoramento e segurança de aplicações, em sua pesquisa de 2022 baseada na avaliação de 1.5 bilhões de containers executados por seus clientes, constata a popularidade do Kubernetes com uma adoção de mais de 50%.

Em resumo, embora o Docker Swarm possa ser uma escolha adequada para equipes que desejam uma solução de orquestração de *containers* simples e rápida, o Kubernetes se destaca quando se trata de escalabilidade, suporte e adoção da comunidade, gestão de microsserviços e flexibilidade. Portanto, ao comparar essas duas opções, a escolha do Kubernetes se justifica, especialmente para organizações que planejam implantações em grande escala e desejam um sistema de orquestração de *containers* altamente flexível e personalizável.

2.5 Rancher

O processo de iniciar e manter um ou várias instâncias de *clusters* Kubernetes traz um desafio por si só. Com sua crescente adoção, surge a necessidade de ferramentas que auxiliem e facilitem toda a gestão do ciclo de vida desses *clusters* e assim o que é chamado de plataformas de gerenciamento de *containers*.

O Rancher se apresenta como uma dessas soluções. Segundo Rancher (2022), o Rancher é uma plataforma *open-source* para implantar e executar *clusters* em qualquer provedor, desde nuvens públicas à *on premise*. Se torna possível gerenciar *clusters* já existentes como também criar novos em um só lugar.

A plataforma oferece diversos recursos e melhorias sobre o Kubernetes "tradicional", sempre visando centralizar e facilitar a interação com o orquestrador de *containers*.

Rancher (2022) cita que uma de suas melhorias mais significativas é a centralização da autenticação e a utilização de RBAC (Controle Baseado em Função) para todos os *clusters*, trazendo maior capacidade de controle de acesso.

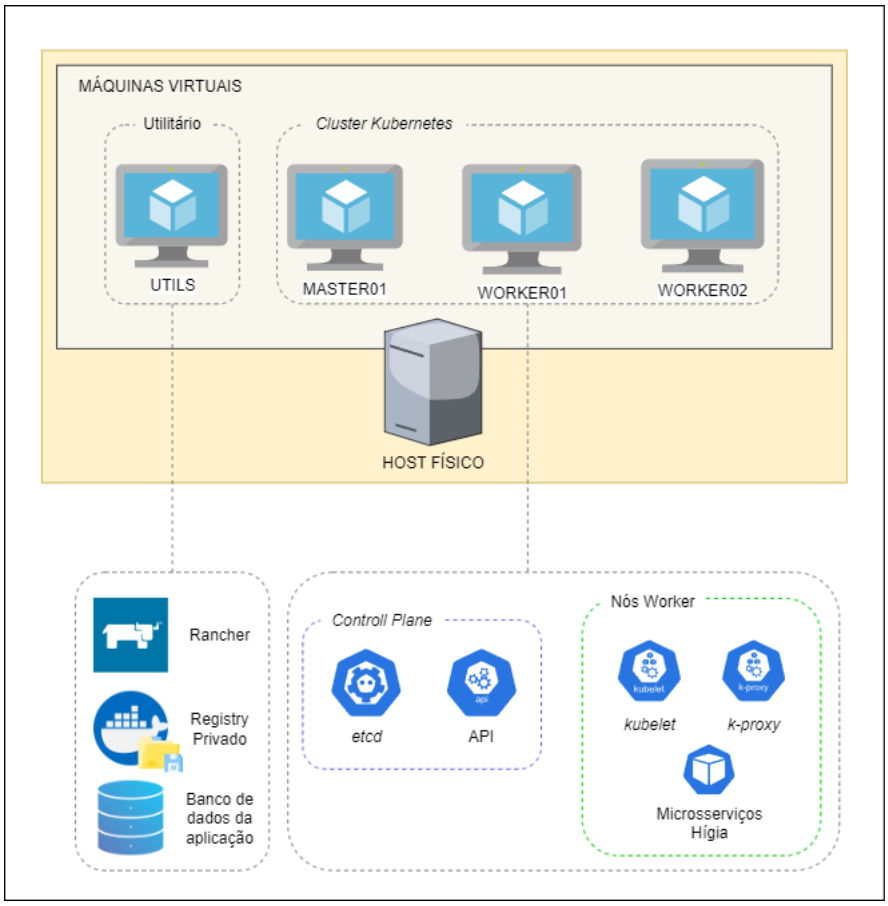
Outro ponto forte é o monitoramento e configuração de alertas para os *clusters* e seus recursos. Também traz a facilitação no envio dos logs a fontes externas.

3 Desenvolvimento

Este capítulo irá apresentar o desenvolvimento da solução proposta para a implementação do protótipo do *cluster* Kubernetes, desde o detalhamento da infraestrutura criada utilizando máquinas virtuais até o provisionamento do *cluster* com a plataforma Rancher. Também serão apresentadas as ações que foram necessárias para adaptar e transferir a aplicação desenvolvida pela JFRN, o Hígia, partindo do entendimento do seu *deploy* comum em único *host* utilizando Docker para se chegar à sua implantação em um modelo de *cluster*.

Na Figura 6 temos uma visão da infraestrutura montada, composta por um conjunto de máquinas virtuais onde cada uma tem seu papel bem definido como a VM UTILS que irá hospedar serviços gerais como a plataforma Rancher, o *Registry* privado e o banco de dados utilizado pelos microsserviços, que ficará de fora da estrutura do *cluster*.

Figura 6 – Imagem da visão geral da infraestrutura criada



Fonte: Produzido pelo autor

As VMs MASTER01, WORKER01 e WORKER 02 são os nós que compõem o

cluster. São ilustrados alguns dos principais componentes que fazem parte de cada uma, como o *etcd* e *API Server* no nó MASTER. Já nos nós WORKER além dos componentes *kubelet* e *k-proxy* é ilustrado onde as aplicações conteinerizadas, como os microsserviços da aplicação Hígia, irão de fato executar dentro dos *Pods* distribuídos entre eles.

3.1 Criação da infraestrutura

Para a realização deste trabalho e criação do protótipo do *cluster* Kubernetes, foi escolhida a abordagem de implantação *on-premise*, ou seja, utilizar os próprios recursos e infraestrutura como servidor, rede e armazenamento da própria JFRN, em detrimento de se utilizar alguma nuvem pública ou privada que já ofereçam os recursos de criação/hospedagem de um *cluster*.

Dessa forma, uma pequena infraestrutura de máquinas virtuais foi criada para o provisionamento do *cluster* Kubernetes e hospedagem da ferramenta de gerenciamento do *cluster*, Rancher. É necessário comentar que o *host* físico atrelado à essa infraestrutura possuía limitações de recursos, sendo uma informação importante para a escolha da distribuição do Kubernetes a ser utilizada.

Partindo das máquinas virtuais, utilizando o software de virtualização da *VMWare*, o ESXi 6.5, foram criadas quatro máquinas com o Sistema Operacional Debian 11.

A escolha do sistema operacional (SO) deu-se por ele ser mais enxuto e, assim, mais adequado ao ambiente de testes, que possui limitações de recursos. Para o ambiente de produção, seria necessário adequar-se ao que já é praticado na infraestrutura da JFRN, que utiliza RHEL (Red Hat Enterprise Linux).

Essas máquinas podem ser subdivididas em dois grupos: utilitário/infraestrutura e o nós do *cluster*.

- **Grupo utilitário/infraestrutura**: uma máquina designada para o *deploy* dos *containers* do Rancher, do *Registry* privado de imagens e o banco de dados da aplicação alvo. Para isso será utilizado o Docker. A máquina possui as seguintes especificações de recursos computacionais e armazenamento: 2 vCPU, 4 Gb de RAM e 150 Gb de disco.

- **Nós do *cluster***: três máquinas que formam o *cluster* propriamente dito, sendo divididas em: uma máquina para ser o nó MASTER e duas máquinas para serem os nós WORKER. Todas possuem as mesmas configurações de recursos computacionais e armazenamento, sendo elas: 2 vCPU, 3 Gb de RAM e 15 Gb de disco.

3.2 Provisionamento do *cluster*

Com as máquinas virtuais disponíveis, o próximo passo é a criação do *cluster*, fazendo a instalação do Kubernetes e partes necessárias.

A abordagem aplicada, foi a de se utilizar a ferramenta Rancher, um *software open source* que nos permite tanto criar como gerenciar múltiplos *clusters* em diferentes plataformas, seja ele em nuvem ou local.

O *deploy* do Rancher pode ser feito de diversas formas, utilizando a linha de comando com o *Helm Charts*, principal gerenciador de pacotes do Kubernetes, que possibilita o que é chamado de "*cluster* pronto para produção", garantindo a alta disponibilidade do Rancher Server ao ser implantado em mais de um nó. Alternativamente, o *deploy* pode ser realizado escolhendo implantar em um único nó a partir de uma imagem Docker distribuída e mantida pelo próprio Rancher. Essa é uma maneira recomendada para fins de testes e estudos, pois oferece maior facilidade e agilidade no *deploy* do Rancher Server. No entanto, possui a desvantagem de não possibilitar a alta disponibilidade, uma vez que está em um único nó.

Vale ressaltar que o Rancher não é o próprio Kubernetes, mas sim uma plataforma que visa facilitar o gerenciamento, administração e manutenção de *clusters* Kubernetes trazendo uma interface gráfica intuitiva e que concentre as principais necessidades em um só lugar.

É necessário também a escolha da distribuição do Kubernetes a ser utilizada como motor do *cluster*. Ao utilizar o próprio recurso de criação do *cluster* do Rancher, ele dá duas opções, o k3s e RKE (*Rancher Kubernetes Engine*), ambas criadas por si e certificadas pelo *Cloud Native Computing Foundation* (CNCF).

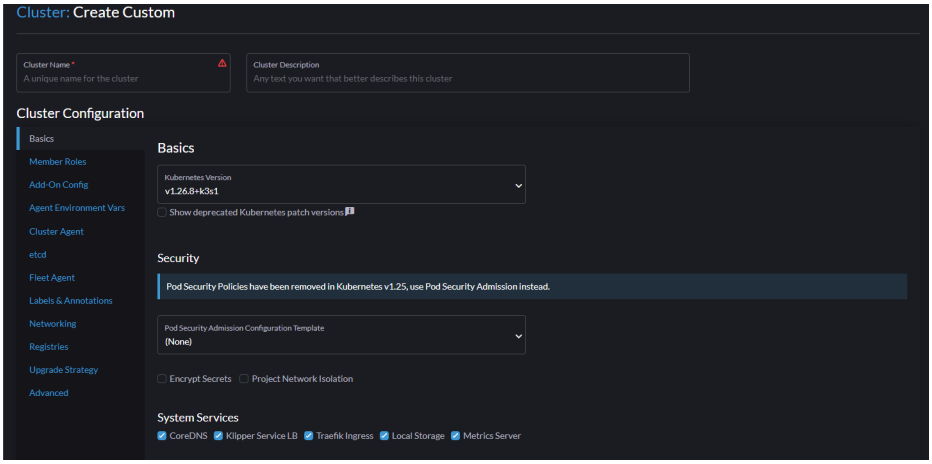
O k3s, na versão *v1.26.6+k3s1*, foi escolhido por ser uma distribuição leve do Kubernetes e ideal para ambientes que possuam limitações de recursos, como é o caso de nossa infraestrutura.

O processo de criação do *cluster* propriamente dito é simples, já que utilizando o recurso do Rancher a instalação e configuração de cada nó é transparente ao usuário. Na Figura 7, temos a página inicial para criação. Esta página oferece diversas opções para customização e configuração do *cluster*. Nela é selecionada a versão da distribuição que decidimos anteriormente. Nenhuma outra alteração foi realizada nessa etapa.

Em seguida temos a tela para selecionarmos o "papel" do nó, visto na Figura 8. Essa seleção é feita com a checagem dos componentes, lembrando que o nó MASTER, necessita do *Control Plane* e o *etcd*. A depender das opções selecionadas, o comando gerado no passo 2 é diferente, e deve ser copiado e executado no nó, de acordo com seu papel.

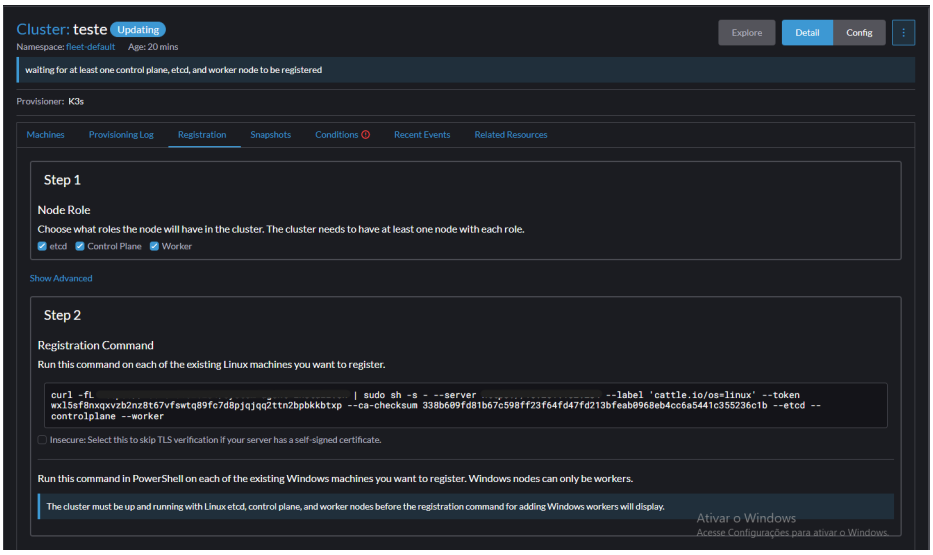
Por fim, após a finalização da criação é possível ter uma visão geral do *cluster*, Figura 9, onde são mostradas informação importantes como o estado de recursos computacionais, quantidade de nós ativos e *Deployments*, o bom funcionamento dos elementos *etcd*, *Scheduler* e *Controller Manager*.

Figura 7 – Criação do *cluster* a partir do Rancher



Fonte: Produzido pelo autor

Figura 8 – Etapa em que é selecionado o papel do nó e gerado o comando a ser executado em cada nó



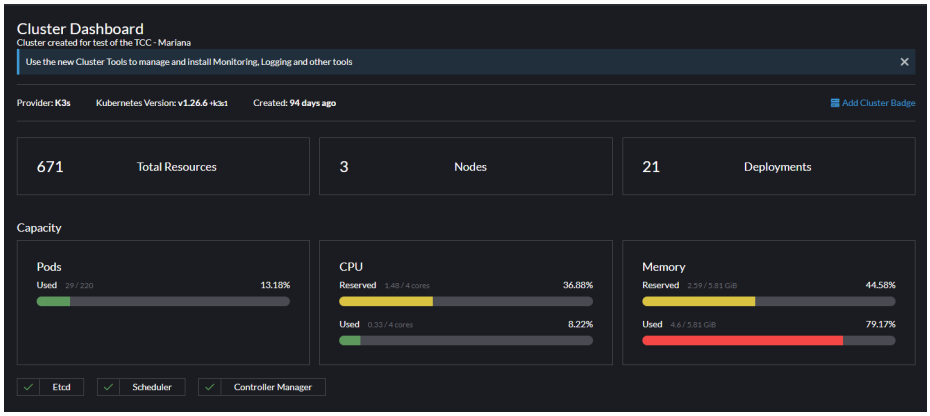
Fonte: Produzido pelo autor

3.3 Porte da aplicação Hígia para o Kubernetes

A transição do *deploy* comum utilizando o Docker para uma ferramenta de orquestração como o Kubernetes, requer adaptações significativas. Os arquivos de configuração de implantação, o objeto *Deployment* em Kubernetes, por exemplo, são mais detalhados e abrangentes do que os arquivos *Docker Compose*. Além disso, as práticas recomendadas para definir serviços, volumes, variáveis de ambiente e escalabilidade diferem consideravelmente.

É crucial entender e implementar essas adaptações para aproveitar ao máximo os

Figura 9 – Visão geral do *Cluster* no Rancher



Fonte: Produzido pelo autor

benefícios do Kubernetes e garantir uma implantação eficiente de aplicativos em *containers*.

Esta seção explora as observações feitas sobre o estado atual da organização do projeto e o *deploy* comum em Docker da aplicação escolhida, o Hígia, e as mudanças que foram necessárias para seu funcionamento em Kubernetes.

3.3.1 Hígia

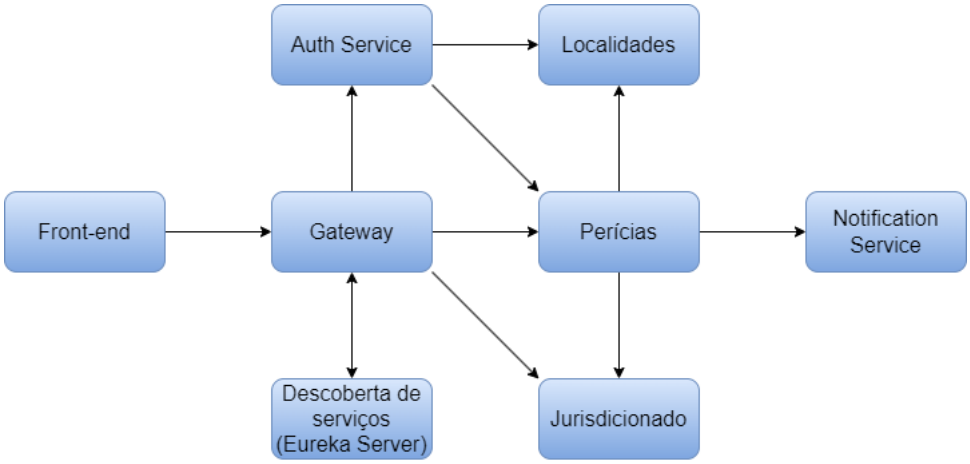
Um dos objetivos do trabalho é realizar um caso de teste, utilizando uma aplicação já conteinerizada e desenvolvida pela JFRN, fazendo seu *deploy* no protótipo do *cluster* Kubernetes criado. Dessa forma, verificar seu comportamento, desempenho e as adaptações necessárias entre seu *deploy* já utilizado em produção e o realizado no Kubernetes.

A aplicação escolhida é o "Hígia – Sistema Web de Informatização de formulários de perícias", definido como "um sistema capaz de centralizar a criação e preenchimento dos formulários de perícia, que por sua vez auxiliará os servidores públicos envolvidos neste processo.". Desenvolvido em framework *Java Spring Boot* e *Angular*, o sistema utiliza a arquitetura de microsserviços, apresentada na Figura 10.

O armazenamento de dados é feito utilizando um sistema de banco de dados, onde os microsserviços de *Auth Service*, Localidades, Perícias e Jurisdicionado salvam e consultam seus dados em seus respectivos bancos.

O processo de adaptação da aplicação começou com o entendimento do projeto como um todo, partindo de quais microsserviços fazem parte da aplicação e a comunicação/dependência entre cada um, os arquivos que compõem o projeto para a geração dos artefatos e imagens de cada microsserviço e o entendimento da esteira CI/CD já utilizada para o *deploy* em ambiente de testes.

Figura 10 – Diagrama de microsserviços do Hígia



Fonte: Produzido pelo autor

3.3.2 Arquivos e esteira CI/CD

Cada microsserviço tem seu respectivo repositório no *GitLab*, a plataforma de escolha da JFRN para o versionamento de código e gerenciamento de projetos. Estes repositórios estão estruturados de forma semelhante entre projetos desenvolvidos com os *frameworks Angular* e *Java SpringBoot*. O projeto é composto por diversos arquivos, os quatro tipos que mais interessam para o desenvolvimento deste trabalho, são:

- ***src/***: diretório onde está o código-fonte e configurações do projeto;
- ***docker-compose.yaml*** e ***Dockerfile***: arquivos envolvidos na criação da imagem e do *container* correspondente ao projeto, bem como configurações de portas à serem expostas e variáveis de ambiente;
- ***.gitlab-ci.yaml***: Arquivo responsável pela definição e descrição das etapas da esteira CI/CD.

Como ilustrado na Figura 11, a esteira CI/CD consiste em cinco estágios com os seguintes papeis:

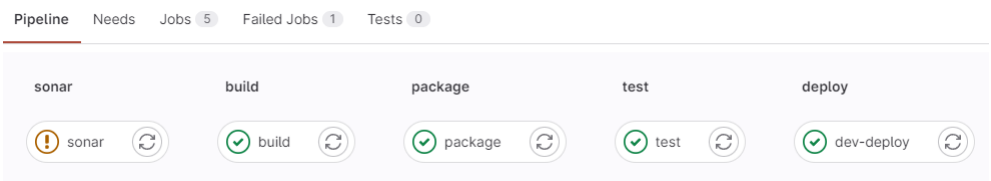
1. ***sonar***: escaneamento em busca de vulnerabilidades e má prática de código, utilizando a ferramenta *SonarQube*;
2. ***build***: compilação do código fonte dos projetos baseados em *Spring Boot* utilizando a ferramenta *Apache Maven*. *Build* dos projetos *Angular* com o gerenciador de pacotes *npm*;

3. ***package***: feito o empacotamento dos artefatos gerados na etapa de *build*, os organizando em um diretório junto aos arquivos *Dockerfile*, *docker-compose.yaml* e *.ENV*;

4. ***test***: para os projetos desenvolvidos com o *framework Sprint Boot*, essa etapa realiza testes unitários no código utilizando a biblioteca *JUnit*;

5. ***deploy***: no caso dos ambientes de testes e homologação se as etapas anteriores foram finalizadas com sucesso, ou apenas avisos, o *deploy* ao *host* que hospeda a aplicação é feito de forma automática. Existe uma sequência de comandos que se inicia com a cópia para o servidor de destino do pacote gerado na etapa de *package* e executado remotamente os comandos Docker sobre o *docker-compose.yaml* e assim gerar a instância do container. Para o ambiente de produção, esta etapa é executada manualmente.

Figura 11 – Exemplo de esteira CI/CD em projetos no *GitLab*



Fonte: Produzido pelo autor

3.3.3 Adaptações

A partir da análise do arquivo de descrição da esteira CI/CD, é notado na forma em que é feito o *deploy* com a esteira de testes, que a imagem do *container* é criada a tempo de execução da esteira e não é armazenada em um *Registry*. Este é o primeiro ponto a ser adaptado, já que quando o objeto de *DEPLOYMENT* para cada microsserviço for configurado, é necessário passar a imagem correta. Todos os nós *WORKER* aos quais o *workload* for distribuído precisam ter acesso a essa imagem.

Para resolver esse problema, foram gerados os artefatos de cada microsserviço e por já possuírem seus Dockerfile, foi feita a construção de cada imagem e realizado o *push* para o *Registry* criado na infraestrutura deste trabalho. Dessa forma, os nós passam a ter acesso a essa imagem quando necessário. A Figura 12 ilustra o acesso via web do *Registry*, onde é possível ter a listagem das imagens que ele tem armazenado.

As próximas adaptações são sobre o arquivo *docker-compose.yaml* já que esse arquivo não é utilizado pelo Kubernetes, é preciso fazer uma análise de seus componentes para melhor adaptar.

Figura 12 – Imagem do *Registry* com todas as imagens geradas na fase de adaptação

```
// 20231021171609
//[redacted]/v2/_catalog

{
  "repositories": [
    "ang-higia-site",
    "app-jurisdicionado",
    "sboot-app-jurisdicionado",
    "sboot-auth-server",
    "sboot-core-localidades-service",
    "sboot-eureka-server",
    "sboot-higia-formularios",
    "sboot-higia-gateway",
    "sboot-localidades-api",
    "sboot-pagina-setorial-api"
  ]
}
```

Fonte: Produzido pelo autor

Utilizando como exemplo o *docker-compose.yaml* do microsserviço de *Gateway*, é ilustrado na Figura 13a, temos a definição da imagem a ser utilizada, quantidade de réplicas, limites de recursos como CPU e Memória e a indicação do arquivo com as variáveis de ambiente. Em alguns casos tem-se portas a serem expostas. Para traduzir ao Kubernetes, podemos desmembrar em objetos *DEPLOYMENT* e *SERVICE*, e o arquivo .env transformado em *CONFIGMAP*.

Figura 13 – Exemplificação de erquivos docker-compose.yaml do Hígia

```yaml
version: '3.8'

services:
  gateway-higia-service:
    container_name: sboot-gateway-higia-service
    image: sboot-gateway-higia-service:2.0
    restart: always
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "9002:9002"
    environment:
      - SPRING_CLOUD_CONFIG_FAIL-FAST=true
    deploy:
      mode: replicated
      replicas: 1
      resources:
        limits:
          cpus: '0.50'
          memory: 1GB
    env_file:
      - .env

networks:
  default:
    name: microservices_network
    external: true
```

(a) Arquivo *docker-compose.yaml* do microsserviço de *Gateway*

```yaml
version: '3.8'

services:
  atom-pericia-service:
    container_name: sboot-atom-pericia-service
    image: sboot-atom-pericia-service:2.0
    restart: always
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      - SPRING_CLOUD_CONFIG_FAIL-FAST=true
    deploy:
      mode: replicated
      replicas: 1
      resources:
        limits:
          cpus: '0.25'
          memory: 1GB
    env_file:
      - .env

networks:
  default:
    name: microservices_network
    external: true
```

(b) Arquivo *docker-compose.yaml* do microsserviço de Perícias

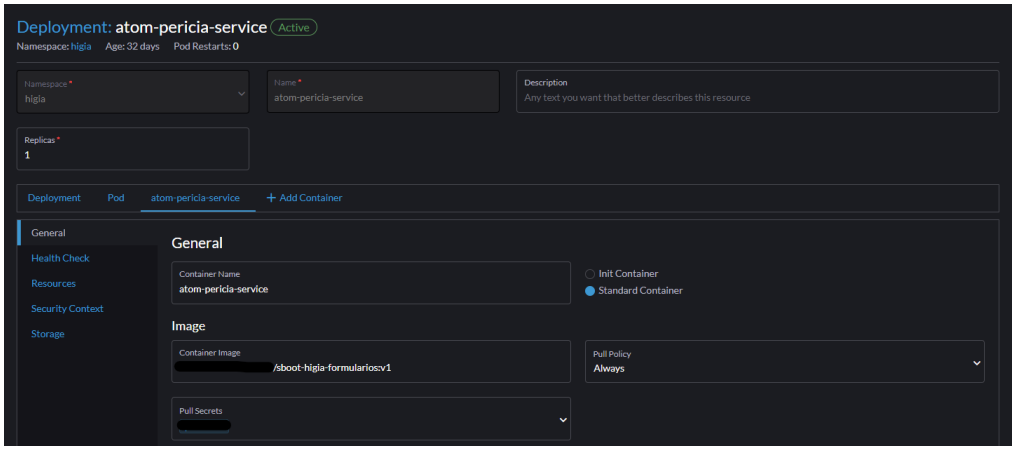
Fonte: Produzido pelo autor

A criação do arquivo que descreve o objeto *DEPLOYMENT* foi realizada de maneira similar entre os microsserviços, aproveitando os recursos da interface gráfica da plataforma Rancher.

Utilizando como exemplo o microsserviço de Perícias, seu *docker-compose.yaml* é ilustrado na Figura 13b. Na página de criação do *DEPLOYMENT*, Figura 14, ao início definimos um nome para identificá-lo e selecionamos o *namespace* a qual ele irá fazer parte.

Da mesma forma de seu *docker-compose.yaml* podemos dar um nome ao *container* que irá executar dentro *Pod*, indicamos a imagem a ser utilizada, que neste trabalho são as que criamos no início desta seção, e a quantidade de réplicas esperadas.

Figura 14 – Página de criação de um *Deployment*



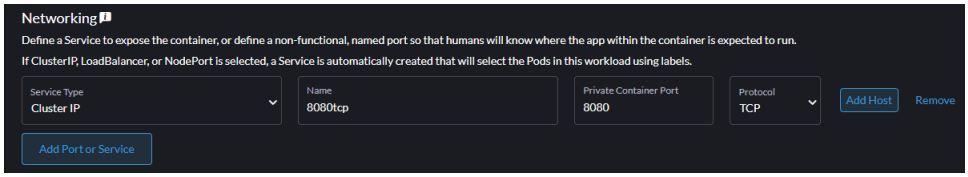
Fonte: Produzido pelo autor

É possível configurar o *SERVICE* associado ao *DEPLOYMENT* na mesma página, facilitando o processo e diminuindo as chances de erros. Com exceção dos microsserviços responsáveis pela Descoberta de Serviços e o *Gateway*, que possuíam suas particularidades e eram melhor configurados em um menu mais completo e separado, todos os outros *SERVICES* foram configurados aproveitando a facilidade mencionada anteriormente.

De acordo com a forma com que o microsserviços se comunicam a partir do microsserviço responsável pela Descoberta de Serviços, foi necessário que um *SERVICE* do tipo *ClusterIP* expondo a porta 8080 fosse criada para cada um deles para assim garantir essa comunicação interna, como mostrado na Figura 15.

Na sequência, são descritas as variáveis de ambiente e é possível selecionar o *CONFIGMAP*. O arquivo do tipo *ConfigMap* é criado em outro menu dentro do Rancher, nele é configurado o nome e o *namespace* ao qual ele está vinculado. As variáveis de ambiente são listadas no tipo chave-valor. Essa separação em outro menu facilita o gerenciamento e a manutenção de todos os ConfigMaps criados, e é possível ter uma visão geral e organizada dos arquivos de configuração que existem no ambiente.

Figura 15 – Criação do *SERVICE* do tipo *CluterIP* na página de *Deployment*



Fonte: Produzido pelo autor

Sobre as particularidades mencionadas anteriormente, nos microsserviços de *Auth Service*, Localidades, Perícias, Jurisdicionado e *Gateway* são passados por variáveis de ambiente o *host* (IP ou URL) e a porta em que o microsserviço de Descoberta de Serviços (Eureka) recebe as requisições. De forma semelhante, é configurado no código fonte do microsserviço de *front-end* da aplicação, o *host* (IP ou URL) do microsserviço de *Gateway*.

Dessa forma, foi identificado que era necessário minimamente a fixação de um endereço IP para os *SERVICES* da Descoberta de Serviços e do *Gateway*. Uma das maneiras de se realizar tal configuração é a utilização de um *SERVICE* do tipo *LoadBalancer*.

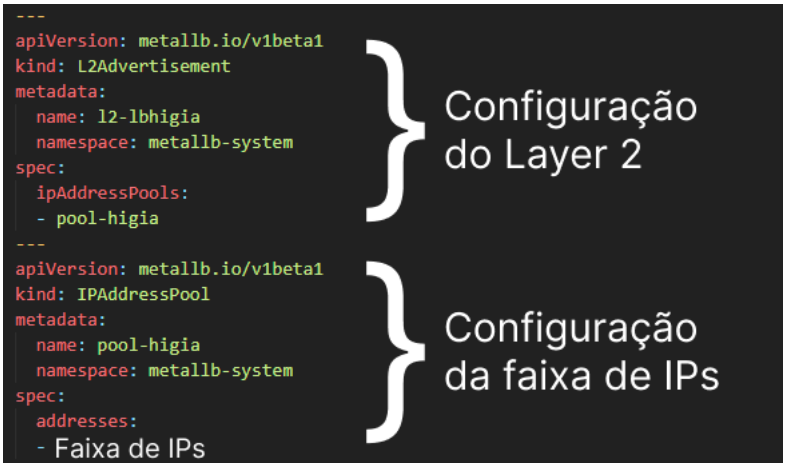
Soluções de *cluster* na nuvem oferecem por si só o recurso de *Load Balancer*, mas por se tratar de um *cluster on-premise* foi necessário a implantação do MetalLB (2023), solução popular de *Load Balancer* para *clusters* Kubernetes *bare-metal*.

Sua instalação foi facilmente feita utilizando o recurso próprio da Rancher, o *Apps*, que funciona como um repositório de aplicativos disponíveis e compatíveis que possam ser instalados no *cluster*. Após a instalação, foi preciso configurar uma faixa de IPs que estará disponível ao MetalLB e isso é feito com o tipo recurso customizado *IPAddressPool*, como ilustrado na Figura 16, em um arquivo de configuração do tipo YAML indicamos o tipo do recurso, definido um nome e selecionado o *namespace* do MetalLB e definir a faixa de IPs.

Uma vez feita essa configuração e os IPs são alocados a um serviço, eles precisam ser anunciados na rede e para isso existem diversos protocolos que podem ser escolhidos. Seguindo a recomendação da forma mais simples de implementação, foi escolhido o modo *Layer 2*, que não precisa de configurações específicas de protocolo só o IP é necessário. Ainda na Figura 16, o mesmo arquivo de configuração criado anteriormente, definimos o tipo de recurso a ser utilizado, descrito um nome e selecionado o namespace. Por fim, utilizamos o seletor para associar esse recurso com a faixa de IPs alvo.

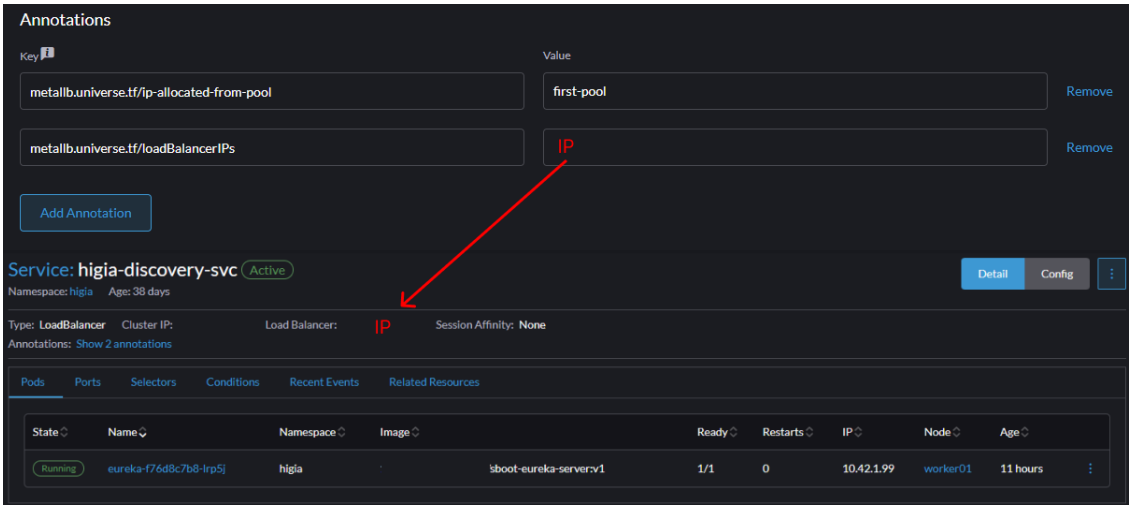
Após a etapa de configuração do MetalLB, voltamos aos *SERVICES*. Para que um IP seja atribuído ao Serviço é preciso adicionar uma anotação no arquivo de descrição do *SERVICE*. De acordo com a documentação do MetalLB, existem diferentes tipos de anotação que podem ser feitas, para a atribuição de um IP fixo pode-se ser utilizado a anotação "*metallb.universe.tf/loadBalancerIPs*" em seguida do IP escolhido. A Figura 17 ilustra a configuração e a visualização do IP atribuído depois de aplicadas as alterações.

Figura 16 – Configuração de recursos do *MetalLB*



Fonte: Produzido pelo autor

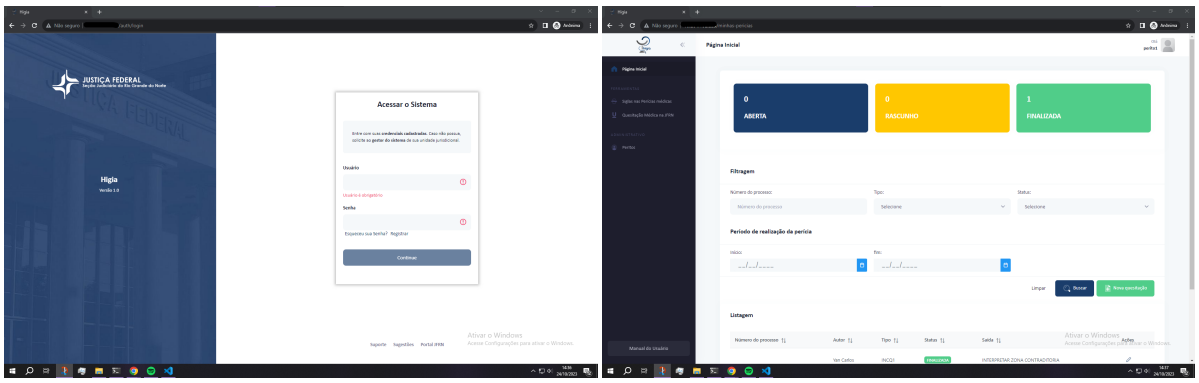
Figura 17 – Configuração e visualização do IP fixo atribuído a um Serviço



Fonte: Produzido pelo autor

O microsserviço do front-end do Hígia não tem as mesmas peculiaridades que a Descoberta de Serviços e o *Gateway*, mas como se trata da porta de entrada para o uso da aplicação pelos os usuários e a fim de gerar um ambiente mais próximo da realidade foi utilizado os mesmo métodos anteriores para fixar um IP para o serviço.

Figura 18 – Telas de *login* e visão do usuário do Hígia



(a) Tela de *login* (b) Painel Inicial

Fonte: Produzido pelo autor

4 Experimentos

Neste capítulo serão abordadas experimentações a fim de prova de conceito sobre dois principais recursos do Kubernetes: escalabilidade horizontal automática e a capacidade de autocura do *cluster*.

Será feita a configuração do *Horizontal Pod Autoscaling* (HPA) em um serviço alvo, o *front-end* do Hígia, fazendo um teste de carga para simular um elevado número de requisições realizadas à esse serviço, especula-se que de acordo com a demanda o serviço escale automaticamente como esperado na configuração feita.

E para a capacidade de autocura, será simulado o mal funcionamento de um dos nós *WORKER* e espera-se que os *Pods* que estejam nesse nó sejam automaticamente realocados em outro nó disponível.

4.1 Escala automática de serviço

Um dos pontos fortes de se escolher um orquestrador de *containers* como o Kubernetes, é ter disponível uma gama de recursos que simplifica a capacidade de automatizar a escalabilidade das aplicações, ou seja o número de instâncias disponíveis, para atender a demanda elástica e de forma geral não padronizada ao quais essas aplicações são submetidas.

Para solucionar tal problema, o Kubernetes disponibiliza nativamente um recurso chamado *Horizontal Pod Autoscaling* (HPA) que automaticamente escala o número de réplicas de pods de um *Deployment* de acordo com a configuração feita.

Luksa (2018) explica que esse recurso funciona em três etapas: coleta constante das métricas dos pods, cálculo da quantidade de *pods* necessários para levar as métricas ao mais próximo possível do valor desejado e por fim a atualização do campo de réplicas do recurso alvo (*Deployment*).

O HPA foi escolhido para a realização deste experimento que tem como objetivo testar o *DEPLOYMENT* do microsserviço de *Front-end* do Hígia, levantando o questionamento se a partir de um teste de carga as requisições geradas irão fazer com que o limite de recurso de CPU configurado seja atingido, dessa forma acionando o HPA e a quantidade de réplicas irá escalar automaticamente para comportar a demanda.

Para Luksa (2018), uma das métricas mais importantes para se basear o escalonamento automático é o consumo de CPU pelos processos que executam dentro dos *Pods*, visto que o alto consumo desse recurso pode gerar problemas, uma vez que os processos estarão sobrecarregados e não atenderão mais à demanda, justificando assim a escolha dessa métrica para a realização do experimento.

4.1.1 Ajuste de configuração do *Deployment*

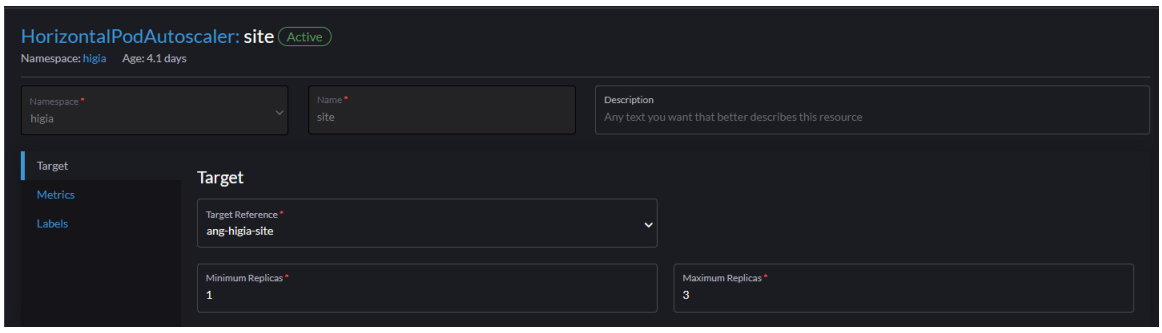
Luksa (2018) explica que para que para o escalonamento automático aconteça, todos os *pods* criados pelo *Deployment* possuem os *resources requests* de CPU especificados. Quando o *Deployment* do microsserviço do *Front-end* foi primeiramente criado, essa configuração não foi realizada, dessa forma ele será revisitado para que seja atendida essa primeira necessidade.

Não é preciso destruir o arquivo inicial, apenas alterá-lo. Na aba de *Resources* do *Deployment* correspondente, são definidos os resources *requests* de CPU para 25 mCPUs e de Memória para 500 MiB. Esses valores foram escolhidos com base no que já era configurado no *docker-compose.yaml*, reduzindo em 50% devido a limitação de recursos da infraestrutura, mas que ainda seja possível comportar o aumento do número de réplicas durante o experimento.

4.1.2 Configuração do HPA e ferramenta de teste de carga

A criação do objeto HPA é feita de forma simples utilizando a interface gráfica do Rancher (Figura 19). Da mesma forma do objeto *Deployment*, inicialmente é escolhido o *namespace* a que ele estará associado e também é definido um nome. Na aba de *Target* (alvo), na opção de *Target References* são listados todos os *Deployments* disponíveis, nesse caso é escolhido o referente ao microsserviço do *Front-end*. Em seguida são definidas as quantidades mínimas e máximas de réplicas, novamente, para fim deste experimento e em respeito a limitação de recursos da infraestrutura foram escolhidos a quantidade mínima de uma réplica e máximo de três.

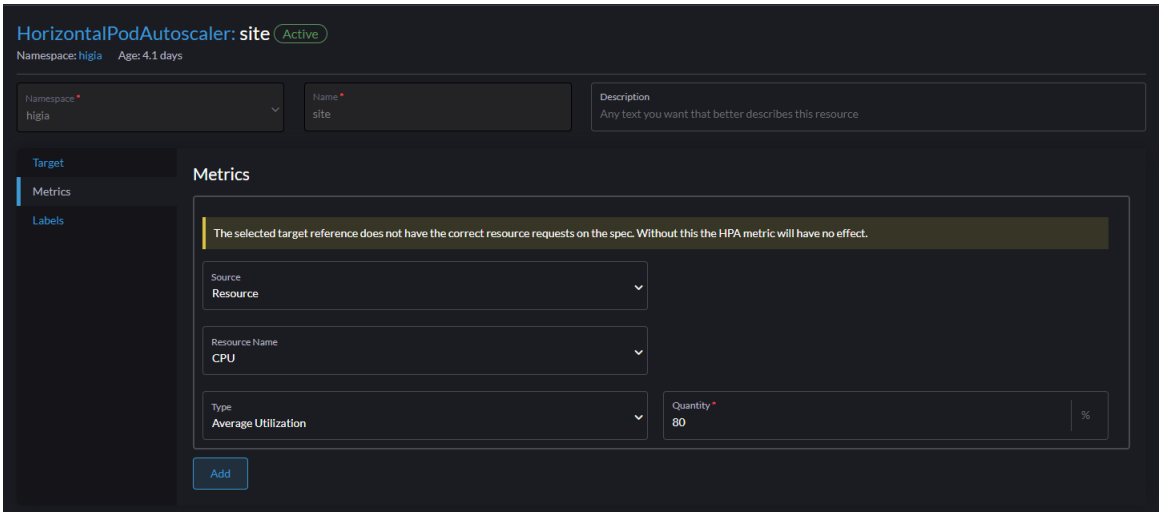
Figura 19 – Configuração do HPA



Fonte: Produzido pelo autor

Já na aba de *Metrics*, Figura 20, são definidas a fonte das métricas a serem monitoradas, o nome, tipo e quantidade. Para a fonte foi escolhido o *Resource* (recurso), o nome é o CPU, o tipo é *Average Utilization* (utilização média) e a quantidade em 80%.

Figura 20 – Configuração das métricas



Fonte: Produzido pelo autor

A própria página do HPA criado oferece recursos para monitorar as métricas atuais de acordo com a utilização dos *pods* e o estado da quantidade de réplicas ativas. Mas para uma melhor visualização, foi instalada utilizando o recurso de *Apps* do Rancher uma *stack* de monitoramento composta pelos *softwares* Prometheus e Grafana. Com esse tipo de instalação "pronta" é possível coletar diversas métricas sobre o *cluster* inteiro e também a plotagem de gráficos sem a necessidade de gastar muito tempo com a configuração. Para esse experimento, podemos filtrar uma métrica sobre as réplicas com estado do tipo *ready* (pronto) de um *Deployment* e especificar qual. Com essa filtragem e atrelado a plotagem de um gráfico, podemos acompanhar o escalonamento das réplicas em relação ao tempo.

Para a realização do teste de carga, foi utilizado a ferramenta de código-aberto *Apache JMeter*, em concordância com práticas já adotadas pela Sessão de Sistemas da JFRN, além de sua vasta adoção no mercado.

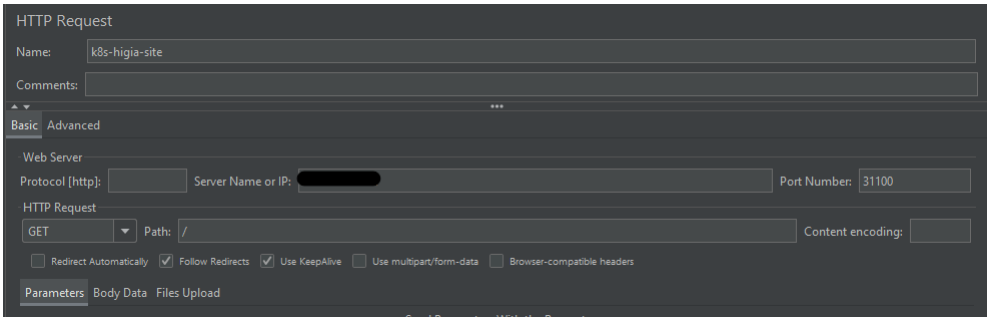
A configuração do plano de teste iniciou-se com a criação de um *Thread Group*, um grupo de *threads* que irão executar em um mesmo cenário. A Figura 21 mostra os campos a serem definidos, quantidade de usuários virtuais que corresponde a quantidade de *threads*, o tempo que cada thread espera para iniciar (*ramp up period*) e por fim o número de iterações por usuário (*loop count*). Neste teste são 500 usuários, cada *thread* leva 2s para iniciar e a tarefa é iterada 20 vezes por usuário.

Ao *Thread Group* é adicionado um *Sampler* que é responsável por dizer ao JMeter que ele deve enviar requisições a um servidor. Pelo microsserviço do teste se tratar de um *front-end* de uma aplicação Web, é escolhido o tipo *HTTP Request* e o método GET. É preciso informar o nome do servidor ou IP, a porta e o caminho da aplicação. A Figura 22 ilustra esta etapa da configuração.

Figura 21 – Criação do *Thread Group*

Fonte: Produzido pelo autor

Figura 22 – Configuração do *Sampler* do tipo *HTTP Request* no JMeter



Fonte: Produzido pelo autor

4.2 Recuperação de falha do *cluster*

A capacidade de autocura de um *cluster* Kubernetes é uma dos seus principais benefícios quando está se decidindo pela ferramenta de orquestração. Gcore (2023) explica que esse recurso trabalha para garantir que o estado atual e o estado desejado do *cluster* estejam sincronizados, e assim que mudanças são identificadas, desde atualização de configuração a falha de um dos nós, uma série de mecanismos trabalham para recuperar ao estado desejado.

O experimento a ser realizado visa testar essa capacidade de autocura simulando uma falha mais severa em um nó *WORKER*, onde seus recursos ficarão indisponíveis por um certo período de tempo. É esperado que com esse experimento seja observado que os *pods* que estavam no nó em falha sejam realocados a um nó saudável.

De acordo com Garod (2018), existem estágios até que se chegue no novo agenda-

mento (*scheduler*) dos *pods*. O nó *MASTER* envia sinais a cada 5s para verificar o estado de um nó *WORKER*, que tem um período de 40s para responder ao sinal. Se após esse tempo ele não responder, é marcado com o *status NotReady*. Ao se manter nesse estado por 5 minutos, os *pods* deste nó são destruídos.

Para o experimento, foi escolhido o nó WORKER02 e a falha a ser simulada será feita com o simples desligamento da máquina virtual correspondente, a mantendo desligada pelo tempo mínimo de 5 minutos.

5 Resultados

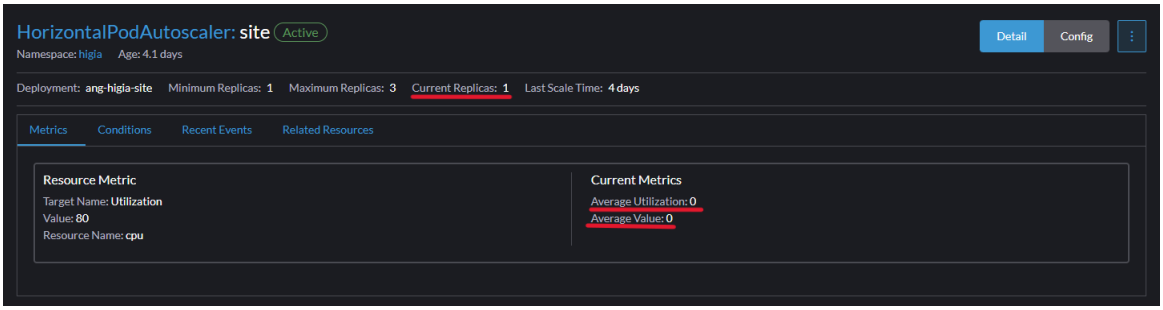
Este capítulo tem como objetivo avaliar o sucesso da implementação do cluster Kubernetes e o porte da aplicação Hígia descritos no Capítulo 3 em conjunto aos resultados obtidos sobre a execução dos experimentos definidos no Capítulo 4.

5.1 Resultados dos experimentos

5.1.1 Escala automática de serviço

O experimento da Escala automática de serviço foi realizado com a execução do plano de teste no JMeter de acordo com as configurações feitas na seção 4.1. O cenário inicial é ilustrado na Figura 23, onde temos os valores de utilização média zerados, ou seja, as requisições ainda não foram geradas e o número de réplicas atuais está no mínimo de 1.

Figura 23 – Estado do HPA antes do início do experimento

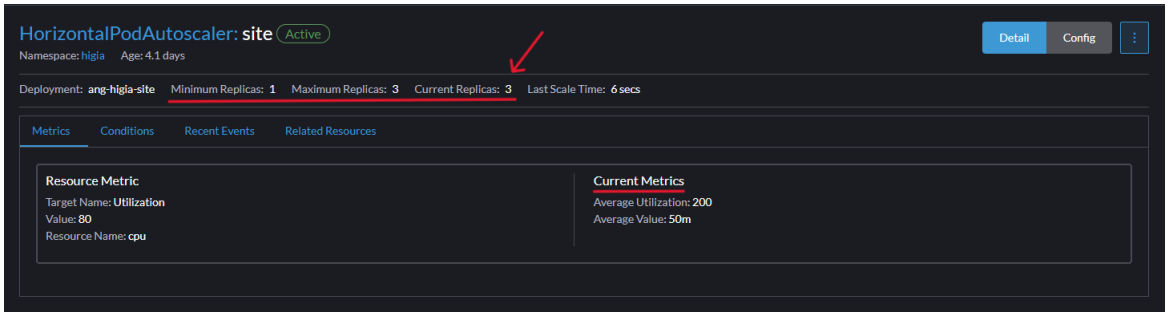


Fonte: Produzido pelo autor

A execução do plano de teste levou cerca de 3min50s para ser completada e durante esse tempo foi possível acompanhar na página do HPA, Figura 24, o aumento da média de utilização do recurso CPU e o estado da quantidade de réplicas. Como esperado, ao atingir o limite de 80% de utilização do CPU novas réplicas são geradas e atingem o número máximo de 3 que foi configurado.

A escala do número de réplicas antes, durante e depois da execução da carga de requisições é melhor observada na Figura 25 com gráfico gerado pela *stack* de monitoramento e de acordo com a filtragem mencionada anteriormente. Foi observado durante o experimento que o *cluster* levou em cerca de 5 minutos para retornar automaticamente à quantidade mínima de réplicas após a finalização do teste e consequentemente a utilização média do CPU ficou abaixo do limite imposto.

Figura 24 – Estado do HPA durante o experimento



Fonte: Produzido pelo autor

Figura 25 – Gráfico do número de réplicas do serviço de Front-end do Hígia em relação ao tempo

Fonte: Produzido pelo autor

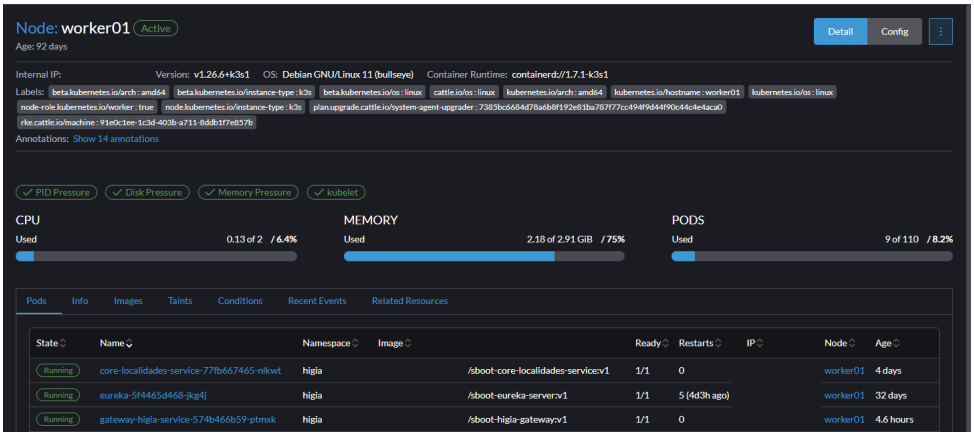
Conclui-se que o experimento foi bem sucedido, pois foi possível responder ao questionamento e expectativa levantados no Capítulo 4 sobre a automatização da escalabilidade de serviços utilizando o próprio recurso do Kubernetes. Temos que o teste de carga foi feito de tal forma que o limite de recurso descrito para o serviço alvo foi atingido e assim acionando o HPA, que de forma automática replicou os pods de forma a atender a demanda.

5.1.2 Recuperação de falha do nó pelo *cluster*

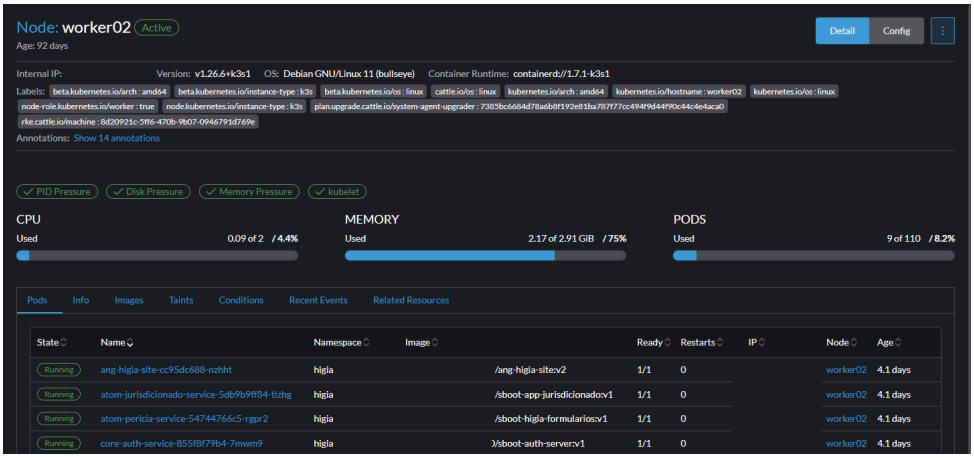
Este experimento tinha como objetivo testar e exemplificar de forma prática a capacidade de autocura de um *cluster* Kubernetes a partir do cenário definido na seção 4.2 deste trabalho.

As figuras Figura 26a e Figura 26b mostram o estado inicial dos nós WORKER01 e WORKER02, respectivamente. Nas imagens é possível ver o estado de cada um como ativo e seus componentes em boas condições. É notável observar a listagem de pods em cada um, sendo três pods no nó WORKER 01 e quatro pods no WORKER02. Essa informação é importante para comparação final quando os pods forem realocados.

Figura 26 – Nós *Workers* antes do ínicio do teste



(a) Nó WORKER01

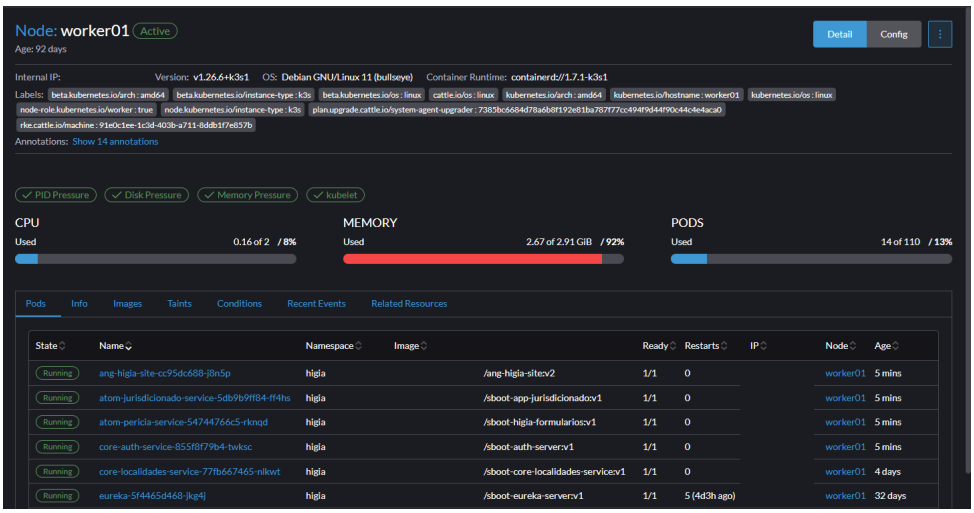


(b) Nó WORKER02

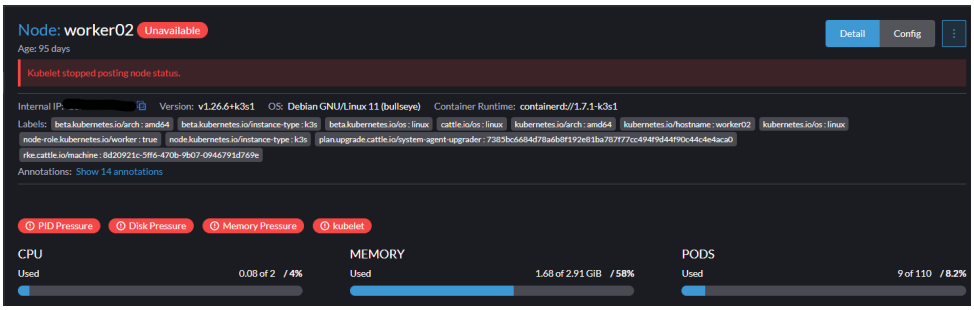
Fonte: Produzido pelo autor

O teste é iniciado fazendo o simples desligamento da máquina virtual que é *host* do nó WORKER2 e cerca de 5 minutos são aguardados de acordo com o definido na seção 4.2. Após esse tempo, ao verificarmos o estado do nó ele se encontra como indisponível (fig:w2-fim). E pela Figura 27a, observa-se que a quantidade de *pods* aumentou no WORKER01, indicando assim que a migração para recuperar a falha do nó WORKER02 ocorreu automaticamente e como esperado.

Figura 27 – Nós *Workers* ao fim do teste



(a) Nó WORKER01



(b) Nó WORKER02

Fonte: Produzido pelo autor

6 Considerações finais

A capacidade de encapsular aplicações e suas dependências em *containers* isolados proporciona flexibilidade e agilidade no desenvolvimento, teste e implantação de software. No entanto, com a crescente adoção de *containers*, torna-se crucial gerenciá-los de maneira eficiente, especialmente em ambientes que envolvem múltiplos *containers* operando simultaneamente. Ferramentas de orquestração, como o Kubernetes, prometem solucionar os desafios associados ao gerenciamento desses *containers*.

O estudo do Kubernetes, sua aplicabilidade e suas vantagens em relação a outras ferramentas se torna fundamental para que as organizações possam tomar decisões conscientes sobre a adoção e implementação de soluções de orquestração. Além disso, à medida que a cultura DevOps continua a ganhar relevância, entender como ferramentas como o Kubernetes se encaixam nesse ecossistema pode oferecer insights valiosos para profissionais e organizações que buscam aprimorar seus ciclos de desenvolvimento e operação.

O desenvolvimento do presente trabalho possibilitou uma análise comparativa entre o Kubernetes e Docker Swarm, servindo de embasamento inicial para justificar a escolha pelo uso do Kubernetes como orquestrador de *containers*, trazendo principalmente a visão de adoção de mercado onde as pesquisas realizadas pela CNCF e Datadog consolidam o Kubernetes como o líder nessa categoria.

Outra importante contribuição foi a constatação que é sim possível migrar uma aplicação desenvolvida pela JFRN, o Hígia, de seu modelo de único *host* e sem replicação de serviços para um modelo utilizando o *cluster* Kubernetes, que é capaz de entregar uma infraestrutura automatizada, resiliente, escalável e segura podendo assim disponibilizar ao usuário final um sistema mais confiável.

Os experimentos realizados também corroboram para afirmação anterior, uma vez que foi possível testar e comprovar que ao se aplicar uma carga grande o suficiente, simulando uma crescente utilização pelos usuários, um serviço consegue se replicar sozinho na tentativa de melhor atender a demanda. Em complemento, foi testada a capacidade de autocura do cluster ao se simular a falha de um nó e tendo como resultado realocação dos serviços comprometidos à um nó saudável.

Conclui-se que este trabalho conseguiu demonstrar a implantação de uma solução em formato de um protótipo de um *cluster* Kubernetes em conjunto ao deploy de uma aplicação escolhida, dando início a mais um processo de modernização do ciclo de desenvolvimento e infraestrutura da JFRN.

6.1 Trabalhos futuros

Um aspecto importante que este trabalho não conseguiu abordar foi a utilização de uma ferramenta que incorpore à uma esteira CI/CD todo o novo processo para se implantar uma aplicação utilizando o Kubernetes. Assim, uma sugestão de trabalho futuro é o estudo e implementação ao CI/CD do próprio GitLab, a ferramenta já utilizada na JFRN, ou novas ferramentas conceituadas do mercado. Alliel (2022) indica: Jenkins X, GitHub Actions, ArgoCD e entre outras.

Outra vertente de exploração seria das Plataformas Gerenciáveis do Kubernetes. Este trabalho utilizou o Rancher para todo o gerenciamento do *cluster* mas ele possui suas limitações principalmente na integração com a ferramenta atual de CI/CD da JFRN. A busca por esse tipo de plataforma se dá pela complexidade e a curva de aprendizado para se utilizar o Kubernetes por si só. Uma dessas plataformas gerenciáveis que não atende somente à nuvem é o OpenShift da Red Hat, que entraria em concordância com o ambiente da JFRN.

Referências

ALLIEL, M. *CI/CD Pipelines for Kubernetes: Best Practices and Tools*. 2022. Komodor. Disponível em: <https://komodor.com/blog/ci-cd-pipelines-for-kubernetes-best-practices-and-tools/>. Acesso em: 24 out 2023. Citado na página 48.

AQUA. *Container Platforms: 6 Best Practices and 15 Top Solutions*. 2021. Cloud Native Wiki. Disponível em: <https://www.aquasec.com/cloud-native-academy/container-platforms/container-platforms-6-best-practices-and-15-top-solutions/>. Acesso em: 6 out 2023. Citado na página 17.

BARNARD, K. *CNCF Survey: Use of cloud native technologies in production has grown over 200%*. 2018. Disponível em: <https://www.cncf.io/blog/2018/08/29/cncf-survey-use-of-cloud-native-technologies-in-production-has-grown-over-200-percent/>. Acesso em: 5 out 2023. Citado 2 vezes nas páginas 23 e 24.

BELTRE, A. M. et al. *Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms*. 2019. 11-20 p. Citado na página 21.

CABALLER, M. et al. *Orchestrating Complex Application Architectures in Heterogeneous Clouds*. 2018. 3-18 p. Disponível em: <https://doi.org/10.1007/s10723-017-9418-y>. Acesso em: 20 out 2023. Citado na página 17.

CASTILLO, A. F. *Why is Kubernetes important for Microservices*. 2022. Disponível em: <https://cloudcomputingtechnologies.com/why-is-kubernetes-important-for-microservices/>. Acesso em: 23 out 2023. Citado na página 23.

CNCF. *Cloud Native Computing Foundation*. 2015. Cloud Native Computing Foundation. Disponível em: <https://www.cncf.io/>. Acesso em: 13 set 2023. Citado na página 23.

CNCF. *CNCF Annual Survey 2021*. 2022. Cloud Native Computing Foundation. Disponível em: <https://www.cncf.io/reports/cncf-annual-survey-2021/>. Acesso em: 14 out 2023. Citado na página 24.

DATADOG. *9 Insights on Real-World Container Use*. 2022. Disponível em: <https://www.datadoghq.com/container-report/>. Acesso em: 5 out 2023. Citado na página 24.

DOCKER. *Swarm mode key concepts*. 2016. Disponível em: <https://docs.docker.com/engine/swarm/key-concepts/>. Acesso em: 27 set 2023. Citado na página 19.

DUNCAN, J.; OSBORNE, J. *OpenShift in Action*. 1. ed. Shelter Island: Manning, 2018. Citado na página 17.

GAROD, M. *The Curious Case of Failing Over in Kubernetes*. 2018. Medium. Disponível em: <https://mgarod.medium.com/the-curious-case-of-failing-over-in-kubernetes-fcd16bc9a94d#:~:text=If%20the%20node%20ever%20fails,that%20node%20will%20be%20deleted.> Acesso em: 24 out 2023. Citado na página 41.

GCORE. *Kubernetes. Replication and self-healing*. 2023. Gcore. Disponível em: <https://gcore.com/learning/kubernetes-and-self-healing-micro-services/>. Acesso em: 24 out 2023. Citado na página 41.

GILL, A. Q. et al. *DevOps for information management systems*. Emerald Publishing Limited, 2018. 122-139 p. Disponível em: <https://doi.org/10.1108/VJIKMS-02-2017-0007>. Acesso em: 20 out 2023. Citado na página 11.

GORDON, A. W. *Is Kubernetes right for me? Choosing the best deployment platform for your business*. 2022. DigitalOcean. Disponível em: <https://www.digitalocean.com/blog/is-kubernetes-right-for-me>. Acesso em: 23 out 2023. Citado na página 23.

KUBERNETES. *Service*. 2017. Disponível em: <https://kubernetes.io/docs/concepts/services-networking/service/>. Acesso em: 8 ago 2023. Citado na página 22.

KUBERNETES. *Overview*. 2019a. Disponível em: <https://kubernetes.io/docs/concepts/overview/>. Acesso em: 27 set 2023. Citado na página 20.

KUBERNETES. *Pods*. 2019b. Disponível em: <https://kubernetes.io/docs/concepts/workloads/pods/>. Acesso em: 27 set 2023. Citado na página 21.

KUBERNETES. *Deployments*. 2019c. Disponível em: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>. Acesso em: 27 set 2023. Citado na página 22.

KUBERNETES. *ConfigMaps*. 2020a. Disponível em: <https://kubernetes.io/docs/concepts/configuration/configmap/>. Acesso em: 22 ago 2023. Citado na página 23.

LAAN, S. *IT Infrastructure Architecture – Infrastructure Building Blocks and Concepts*. 3. ed. [S.l.]: Lulu Press Inc, 2017. Citado 3 vezes nas páginas 13, 14 e 15.

LUKSA, M. *Kubernetes in Action*. 1. ed. Shelter Island: Manning, 2018. Citado 4 vezes nas páginas 16, 20, 38 e 39.

MELL, E. *Docker Engine*. 2022. TechTarget. Disponível em: <https://www.techtarget.com/searchitoperations/definition/Docker-Engine>. Acesso em: 6 out 2023. Citado na página 18.

METALLB. *METALLB*. 2023. MetalLB. Disponível em: <https://metallb.universe.tf/>. Acesso em: 31 ago 2023. Citado na página 35.

NICKOLOFF, J.; KUENZLI, S. *Docker in Action*. 2. ed. Shelter Island: Manning, 2019. Citado na página 16.

RANCHER. *What is Rancher?* 2022. Disponível em: <https://ranchermanager.docs.rancher.com/>. Acesso em: 13 set 2023. Citado na página 25.

REDHAT, I. *What is container orchestration?* 2022. Disponível em: <https://www.redhat.com/en/topics/containers/what-is-container-orchestration>. Acesso em: 25 jul 2023. Citado na página 19.

RENSIN, D. *Kubernetes: Scheduling the Future at Cloud Scale*. 1. ed. Sebastopol, CA: O'Reilly Media, 2015. Citado na página 21.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating System Concepts*. 10. ed. [S.l.]: Wiley, 2021. Citado na página 14.

SIMIC, S. *What is a Hypervisor? Types of Hypervisors 1 & 2*. 2019. PhoenixNAP. Disponível em: <https://phoenixnap.com/kb/what-is-hypervisor-type-1-2>. Acesso em: 23 out 2023. Citado na página 16.

SMITH, S. *Practical continuous deployment: a guide to automated software delivery*. 2017. ATLASSIAN Blog. Disponível em: <https://www.atlassian.com/blog/continuous-delivery/practical-continuous-deployment>. Acesso em: 23 out 2023. Citado na página 11.

SOYINKA, W. *Linux Administration: A Beginner's Guide, Seventh Edition*. 7. ed. [S.l.]: McGraw Hill, 2015. Citado na página 14.

The Editors of Encyclopedia Britannica. time-sharing. In: _____. *Encyclopedia Britannica*. [S.l.: s.n.], 2021. Citado na página 14.

THINKSYS. *Docker Swarm vs. Kubernetes: Comparison 2023*. 2022. Disponível em: <https://thinksys.com/devops/docker-swarm-vs-kubernetes-comparison/>. Acesso em: 23 out 2023. Citado na página 23.

VERAS, M. *Virtualização. Componente Central do Datacenter*. 1. ed. Rio de Janeiro: Brasport, 2011. Citado na página 15.

VMWARE. *Why use containers vs. VMs?* 2022. VMware. Disponível em: <https://www.vmware.com/br/topics/glossary/content/vms-vs-containers.html>. Acesso em: 5 out 2023. Citado na página 15.