Aprendizagem de Máquina Nativa à Nuvem: Provendo Alta Disponibilidade ao Sistema SPIRA com Kubernetes

Vitor Guidi

Proposta de Trabalho de Conclusão de Curso
apresentado à disciplina
MAC0499
(Trabalho de Formatura Supervisionado)

Orientador: Prof. Dr. Alfredo Goldman
Coorientador: Me. Renato Cordeiro Ferreira

São Paulo, Abril de 2023

Sumário

Abstract

GUIDI, V. **Cloud-Native Machine Learning: Enabling High-Availability in the SPIRA system with Kubernetes**. Monograph (Capstone Project) – Institute of Matematics and Statistics, University of São Paulo, São Paulo, 2023.

SPIRA is a research project at Universidade de São Paulo whose goal is to detect respiratory insufficiency via voice by using Machine Learning. In a previous research project, Tamae developed a set of microservices to enable SPIRA being used in hospitals. However, due to time constraints, all services were provisioned as a single replica at a single physical machine. This deployment strategy is unreliable, since the whole system may become unavailable if a single service crashes, or if the physical machine underneath goes offline. The goal of this research was to enable high-availability for SPIRA by migrating all of its business and infrastructure microservices to Kubernetes – currently, the most popular container orchestrator in the industry. It proposes a new infrastructure architecture that ensures a degree of resilience for all services, which may be deployed in a cluster of many machines. Finally, to verify the high avaibality, this research used chaos testing to ensure the system keeps responding and do not loose data even when different services are turned off randomly.

Keywords: Machine Learning Engineering, Kubernetes, Cloud Native, Chaos Tests, High Availability, Operators, Machine Learning Infrastructure, MLOps.

Resumo

GUIDI, V. **Aprendizagem de Máquina Nativa à Nuvem: Provendo Alta Disponibilidade ao sistema SPIRA com Kubernetes**. Monografia (Trabalho de Formatura Supervisionado) – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

O SPIRA é um projeto de pesquisa na Universidade de São Paulo, cujo propósito é detectar insuficiência resporatória por meio de análise de voz, utilizando aprendizado de máquina. Em um projeto anterior, Tamae desenvolveu um conjunto de microsserviços para possibilitar que o SPIRA seja utilizado em hospitais. Entretanto, por limitações de tempo, todos os microsserviços foram provisionados como uma única réplica, dentro de uma única máquina virtual. Esta estratégia de provisionamento não provê alta disponibilidade, já que o serviço inteiro pode ficar indisponível se uma réplica de um dos microsserviços parar de funcionar, ou se a máquina hospedeira ficar indisponível. O objetivo desta pesquisa é prover alta disponibilidade para os microsserviços do SPIRA, por meio da migração para o Kubernetes – atualmente, o orquestrador de contêineres mais popular da indústria. Uma nova arquitetura de provisionamento foi proposta, que pode ser implantada em um conjunto de máquinas ao invés de uma única réplica. Finalmente, para verificar que de fato foi adicionada alta disponibilidade, esta pesquisa utilizou testes de caos para garantir que o sistema continue respondendo e não haja perda de dados quando diferentes réplicas dos microsserviços são desligadas aleatoriamente.

Keywords: Engenharia de Aprendizado de Máquina, Kubernetes, Sistemas Nativos à Nuvem, Testes de Caos, Alta Disponibilidade, Operadores, Infraestrutura de Aprendizado de Máquina, MLOps.

Capítulo 1

Introdução

Com o advento da computação em nuvem, houve uma revolução na forma de disponibilizar aplicações. A nuvem oferece infraestrutura como serviço, evitando assim a necessidade de adquiri-la e operá-la fisicamente. Ademais, só se paga pelo que realmente é usado, e os recursos adquiridos podem ser mobilizados ou desmobilizados a qualquer momento.

Para que uma aplicação tire o máximo proveito desse novo paradigma de infraestrutura, há uma série de boas práticas da indústria. Aplicações que seguem tais práticas são geralmente chamadas de NATIVAS À NUVEM (em ingles, *cloud native*). Segundo a AWS[1], pioneira da computação em nuvem, aplicações nativas à nuvem são compostas por microsserviços conteinerizados, que expõem APIs para consumo.

O SPIRA é um projeto de pesquisa da Universidade de São Paulo criado durante a pandemia de COVID-19 para detectar insuficiência respiratória por meio de análise de fala. Em seu trabalho de formatura, Tamae criou um conjunto de microsserviços para prover uma API de inferência para o sistema SPIRA (Tamae, 2022).

Os microsserviços do SPIRA foram escritos utilizando arquitetura hexagonal (Martin, 2017), seguindo os princípios de aplicações nativas à nuvem. Entretanto, foram implantados em uma única máquina física, o que traz risco de indisponibilidade.

Em uma pesquisa de iniciação científica[2] realizada na Universidade de São Paulo, uma aplicação composta de múltiplos microsserviços foi provisionada no Kubernetes, obtendo assim alta disponibilidade. Dado o sucesso da pesquisa em questão, essa metodologia se mostrou viável para o caso de uso do SPIRA.

Esta pesquisa visa prover alta disponibilidade para a aplicação `SPIRA`, consolidando os resultados dos dois trabalhos citados anteriormente. Para tanto, o projeto visa implantar a aplicação no orquestrador de contêineres Kubernetes, possibilitando a existência de réplicas dos serviços em máquinas distintas.

O Capítulo 2 apresenta os conceitos fundamentais de computação em nuvem, o Kubernetes e suas primitivas. O Capítulo 3 introduz a arquitetura do sistema `SPIRA`, originalmente desenvolvida em Tamae. O Capítulo 4 apresenta a nova arquitetura do SPIRA no Kubernetes, e os desafios encontrados para provisioná-la. O Capítulo 5 apresenta estratégias de teste empregadas durante a pesquisa. O Capítulo 6 sumariza os resultados obtidos e trabalhos futuros. Finalmente, o Apêndice A instrui como provisionar o `SPIRA`, e o Apêndice B demonstra como reproduzir os resultados obtidos.

[1] Amazon Web Services: what is cloud native?
[2] Cluster de Kubernetes em Raspberry Pi

Capítulo 2

Sistemas Nativos à Nuvem

O ecossistema nativo à nuvem se baseia em abstrações na camada de infraestrutura e de aplicação, que atuam como os blocos básicos de construção de sistemas. A nova arquitetura do SPIRA (apresentada no Capítulo 4), em particular, se vale de abstrações fornecidas pelo Kubernetes para provisionar seus microsserviços de infraestrutura e de negócio.

A Seção 2.1 apresenta os conceitos básicos de computação em nuvem. A seção Seção 2.2 introduz o Kubernetes e seu ecossistema, discutindo múltiplas distribuições de Kubernetes, em particular a escolha do KinD nesta pesquisa. Por fim, a Seção 3.1 lista as dependências de infraestrutura do SPIRA.

2.1 Conceitos de Computação em Nuvem

Uma aplicação nativa à nuvem é composta por um ou mais **microsserviços**, que expõem suas funcionalidades por meio de uma API. Tais microsserviços geralmente residem em **contêineres**, que por sua vez consomem frações de recursos em **máquinas virtuais**. Para administrar o ciclo de vida dos microsserviços, e garantir que eles interajam corretamente como um sistema coeso, são empregados **orquestradores de contêineres**.

Esta seção esclarece o que são e qual é o propósito de cada componente mencionado.

2.1.1 APIs

Uma API[1] é uma interface pela qual um cliente solicita a execução de alguma tarefa computacional para um servidor, por meio de um protocolo de comunicação predefinido.

No contexto do sistema SPIRA, uma API segue o padrão REST[2], pressupõe que a comunicação ocorre pelo protocolo HTTP[3], e entrega resultados no formato JSON[4].

2.1.2 Microsserviços

Microsserviços (Richardson, 2018), são coleções de serviços desacoplados entre si, com responsabilidades bem definidas. O termo foi originalmente cunhado pela Amazon[5], que descreve as equipes

[1] What is an API?
[2] Architectural Styles and the Design of Network-based Software Architectures, by Roy Fielding
[3] RFC 2616: Hypertext Transfer Protocol – Http 1.1
[4] The JavaScript Object Notation (JSON) Data Interchange Format – RFC 8259
[5] Amazon

responsáveis por um microsserviço como *two pizza teams*[6](times de duas pizzas, em inglês): equipes pequenas, que podem ser alimentados com apenas duas pizzas, responsáveis por um escopo restrito da regra de negócio total da empresa, de forma a diminuir a carga cognitiva dos desenvolvedores, evoluir, e provisionar seus microsserviços de forma independente das demais equipes.

Sendo assim, cada time precisa estar ciente apenas dos contratos com os demais serviços com que interagem, não dos requisitos de negócio e dos detalhes de implementação do sistema inteiro.

2.1.3 Máquinas Virtuais

Máquinas virtuais[7] são uma forma de executar múltiplos sistemas operacionais no mesmo *hardware*. O isolamento de recursos é implementado por meio de um *Hypervisor*[8]. Uma característica importante de máquina virtuais é que há custo extra (*overhead*) na execução de múltiplos sistemas operacionais que compartilham o mesmo *hardware*.

No contexto de aplicações nativas à nuvem, máquinas virtuais servem dois propósitos: oferecer um sistema operacional para o usuário, sem que seja necessário a aquisição de máquinas físicas; e possibilitar que múltiplos usuários compartilhem a mesma máquina hospedeira (em diferentes instâncias do sistema operacional), permitindo assim economia de escala.

2.1.4 Contêineres

Contêineres[9] são uma abstração com origem no projeto Linux. Tal abstração possibilita que aplicações executem compartilhando o mesmo *kernel* do sistema operacional de uma máquina – garantindo o isolamento de recursos e das dependências de software – e diminuindo assim o consumo de recursos de *hardware*.

No livro *Cloud Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment* (Garrison and Nova, 2017), outro motivo para que aplicações nativas à nuvem sejam provisionadas em contêineres é que o ferramental e os processos estão atualmente mais maduros em relação às máquinas virtuais.

2.1.5 Orquestradores de Contêineres

Orquestradores de contêineres[10] são ferramentas que administram o ciclo de vida de múltiplos contêineres. As principais funcionalidades que orquestradores de contêineres oferecem são:

- escalonar dinamicamente contêineres entre as múltiplas máquinas,
- monitorar a saúde de um contêiner,
- provisionar substitutos para reestabelecer o funcionamento do sistema em caso de falha,
- prover primitivas de rede que abstraia a comunicação entre múltiplos contêineres, na mesma máquina ou em máquinas diferentes, e

[6] Two pizza teams
[7] What are virtual machines?
[8] Hypervisor
[9] What are containers?
[10] What is container orchestration? (Red Hat)

- prover elasticidade, de forma que o número de contêineres de um microsserviço se adapte dinamicamente ao tráfego recebido.

Durante seu trabalho de formatura, Tamae provisionou o SPIRA por meio da ferramenta Docker Compose[11]. Embora ele seja um orquestrador de contêineres, é limitado porque se restringe a somente uma máquina, e não oferece elasticidade para adaptar o número de réplicas à demanda.

2.2 Kubernetes e seu Ecossistema

segundo pesquisa realizada pela empresa Stack Overflow[12], o Kubernetes[13] é o orquestrador de contêineres mais usado por desenvolvedores. Por tal razão, foi empregado nesta pesquisa.

Dado um conjunto – ou, em inglês, ***cluster*** – de máquinas, o Kubernetes aloca contêineres nas máquinas individuais, e abstrai a comunicação entre os microsserviços. Maiores detalhes podem ser consultados na documentação do *Kubernetes*[14].

Da mesma forma que o sistema operacional oferece abstrações para o programador interagir com o hardware de uma máquina, o Kubernetes oferece abstrações para o desenvolvedor hospedar seus contêineres em um *cluster* de computadores.

O Kubernetes é dividido em duas partes: o **plano de controle**, e o **plano de dados**. O plano de controle é responsável por decidir o estado futuro do *cluster*, a partir do estado desejado dos recursos definido pelo usuário. O plano de dados é responsável por materializar as decisões do plano de controle nos contêineres e máquinas físicas.

Para hospedar uma aplicação, o Kubernetes oferece alguns objetos primitivos. O plano de controle pode ser estendido por meio de recursos e controladores customizados, de forma a orquestrar aplicações com ciclo de vida complexo reutilizando as primitivas da ferramenta.

2.2.1 Recursos

De forma simplificada, o Kubernetes funciona por meio de controladores que executam indefinidamente um **laço de reconciliação**: observam o estado atual de um recurso, comparam com o estado desejado, e tomam as ações necessárias para que o estado desejado se realize.

Um **recurso** é algo a ser gerenciado pelo Kubernetes. Recursos são representados no formato YAML[15]. Por meio desses arquivos, os usuários da ferramenta definem o estado desejado das aplicações. O Kubernetes toma ações que façam o estado atual convergir ao estado desejado pelo usuário.

2.2.2 Recursos nativos do Kubernetes

Dentre os recursos nativos do Kubernetes, os utilizados nessa pesquisa são os seguintes:

- NAMESPACES: são divisões lógicas do *cluster*. Permitem segregar objetos por domínio de negócio, facilitando a administração do *cluster* e a orquestração de aplicações.

- NODES: são máquinas (físicas ou virtuais) responsáveis por hospedar conjuntos de contêineres.

[11] Docker Compose
[12] Stack Overflow: 2023 most popular technologies
[13] Kubernetes
[14] Kubernetes docs
[15] YAML

- PODS: são uma coleção de contêineres. Constituem a unidade atômica de uma aplicação.
- DEPLOYMENTS: são uma interface que encapsula múltiplos PODS anônimos, usualmente empregado para aplicações sem estado. Oferecem a capacidade de manter múltiplas réplicas de um POD, além de possibilitar diferentes políticas de atualização de versão.
- DAEMON SETS: são responsáveis por manter um POD de uma aplicação em cada máquina do *cluster*.
- STATEFUL SETS: são uma interface que encapsula múltiplos pods com identidade fixa. Podemos garantir que réplicas com a mesma identidade sempre manterão o mesmo estado após os PODS serem recriados ou realocados no *cluster*, e responderão aos mesmo *endpoints*. Isso facilita o provisionamento de bancos de dados dentro do Kubernetes.
- SERVICES: são uma abstração de rede. Oferecem um *endpoint* pelo qual comunicação de rede pode ser estabelecida com o conjunto de PODS que forma uma aplicação.
- PERSISTENT VOLUMES: são unidades atômicas de armazenamento persistente. Podem ser montados no sistema de arquivo dos PODS.
- CONFIGMAPS: são estruturas que armazenam dados, usualmente de configuração. Facilitam o reúso de configuração para múltiplos PODS.
- SECRETS: são responsáveis por armazenar dados, obfuscados em base64[16]. São similares aos CONFIGMAPS, enquanto facilitam reuso de dados configuração para múltiplos PODS.
- CRDS: são recursos customizados, criados por terceiros, que estendem a API do Kubernetes para orquestrar o ciclo de vida de aplicações arbitrárias. Geralmente são acompanhados de controladores, formando assim o padrão de projeto *Operator*[17].

O entendimento dos componentes primitivos do Kubernetes permitirá a compreensão da estratégia de provisionamento do SPIRA, discutida no Capítulo 4.

2.2.3 Plano de controle

O plano de controle do Kubernetes é responsável por observar o estado atual do *cluster* e tomar decisões para convergir para o estado desejado pelo usuário. O estado do *cluster* é o estado do conjunto de objetos nele contido.

O usuário precisa uma forma de declarar estado. A API do Kubernetes é a interface que nos permite declarar o estado desejado de um *cluster*. Usualmente, os objetos são representados em formato YAML ou JSON, e publicados para o plano de controle do *cluster*.

Internamente, o Kubernetes manipula os recursos por meio de uma API REST[18]. Cada recurso tem um controlador associado, que irá seguir sua lógica de reconciliação para garantir que o estado atual do *cluster* avance em direção ao estado desejado, declarado nos objetos pelo usuário.

O plano de controle é composto pelos seguintes elementos:

[16] base64
[17] Operator pattern
[18] Kubernetes API

- API SERVER: é responsável por servir as requisições HTTP de criação, atualização e remoção dos objetos do Kubernetes. Além disso, publica os eventos para quem se inscreveu a notificações do ciclo de vida dos diferentes tipos de objetos.

- ETCD: é responsável por armazenar os metadados do *cluster*. É consumido principalmente pelo API SERVER para consultar, persistir, e alterar o estado dos objetos.

- SCHEDULER: é responsável por decidir em quais NODES do *cluster* os PODS deverão ser alocados. Possui políticas padrão de escalonamento, e pode ser estendido com políticas customizadas pelo usuário.

- CONTROLLER MANAGER: é o componente que se inscreve nos eventos emitidos pelo API SERVER, no que toca ao ciclo de vida dos objetos primitivos. Ele contém a lógica de reconciliação: dado um evento do ciclo de vida dos objetos, observa o estado atual do *cluster* e toma uma decisão que permita avançar o estado atual para o estado desejado pelo usuário.

2.2.4 Plano de dados

O plano de dados é responsável por implementar as decisões do plano de controle nas máquinas físicas do *cluster*. Mais precisamente, é responsável por escalonar os PODS nas máquinas definidas pelo SCHEDULER, e por implementar as abstrações de rede.

Durante o desenvolvimento do Kubernetes, um princípio de *design* foi adotado[19] para diminuir o atrito na migração aplicações de máquinas virtuais para contêineres. Tal princípio impõe alguns requisitos sobre a forma com que a comunicação sobre a rede deve funcionar:

- cada POD deve ter um IP único no *cluster*,

- contêineres no mesmo POD devem ser capazes de se comunicar livremente com a interface *loopback*[20] do POD, e

- PODS em diferentes máquinas do *cluster* devem ser capazes de se comunicar por meio de seus IPs únicos sem o uso do protocolo NAT[21].

O padrão que define os requisitos acima é denominado *Container Network Interface* (interface de rede de contêiner, em inglês). Existem várias implementações diferentes de CNI, entre elas o Calico[22].

Os componentes que realizam as tarefas do plano de dados são os seguintes:

- KUBE-PROXY: é um *daemon*[23] responsável por monitorar o ciclo de vida de objetos do tipo SERVICE, e alterar as configurações de rede da máquina de forma que os PODS possam se comunicar com o DNS de tais objetos pelo contrato de rede do Kubernetes (PODS com IP único no *cluster*). Geralmente, isso é realizado por meio de configuração de *iptables*[24] no Linux.

[19] Kubernetes networking design principles
[20] Loopback interface
[21] Network Address Translation
[22] Calico
[23] Linux Daemons
[24] iptables

- KUBELET: é um *daemon* presente em cada máquina do *cluster*. É responsável por materializar os PODS esperados pelo API SERVER em sua máquina, após decisão do SCHEDULER. Faz isso por meio de uma *Container Runtime Interface*[25] (interface de execução de contêineres, em inglês), uma abstração pela qual se comunica com um *container runtime* (por exemplo, o *Docker*).

2.2.5 Helm Charts

Um HELM CHART[26] é uma abstração empregada para representar aplicações complexas, compostas por múltiplos recursos de Kubernetes. Por meio de uma *template engine*[27], permite a modificação de parâmetros de uma aplicação, e reúso de recursos entre diferentes projetos.

Nesta pesquisa, os HELM CHARTS foram utilizados no Capítulo 4 para o provisionamento de alguns dos componentes de infraestrutura. Em particular, o NATS e o MLFlow.

2.2.6 Operadores

Um operador é um ponto de extensão do Kubernetes, que permite gerenciar aplicações complexas a partir da composição de recursos nativos e da extensão da API do *Kubernetes*.

Os operadores são formados por dois componentes: Definições de Recurso Customizadas [28](em inglês, *Custom Resource Definitions*, CRD), que representam o recurso, e um controlador, que executa um laço de reconciliação.

Segundo Garrison and Nova, no livro *Cloud Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment*, um laço de reconciliação é um padrão de projeto no qual o estado atual do *cluster*, e tomam-se ações incrementais de forma que o estado atual convirja para o estado declarado pelo usuário.

Nesta pesquisa, os operadores foram usados no Capítulo 4 para provisionar algumas das dependências do sistema `SPIRA` dentro do Kubernetes. Em particular, o MinIO, o PostgreSQL, e o MongoDB.

2.2.7 Distribuições de Kubernetes

Há várias implementações concretas de Kubernetes disponíveis, que se dividem em dois grandes grupos: distribuições para desenvolvimento local, e distribuições para ambientes de produção. Para ambientes de produção, podemos ter distribuições de Kubernetes em nuvem, ou para *data centers* privados (também conhecido como provisionamento *on premise*).

Distribuições de Kubernetes para desenvolvimento local incluem ferramentas como KinD[29] (*Kubernetes in Docker*), e o MiniKube[30]. Dentre as distribuições de Kubernetes para ambiente em nuvem, podemos encontrar oferecimentos dos grandes provedores: o EKS[31], da Amazon; o AKS[32], da Microsoft; e o GKE[33], da Google.

[25] Container Runtime Interface
[26] Helm charts
[27] Templating engines
[28] Custom Resource Definitions
[29] KinD
[30] MiniKube
[31] Elastic Kubernetes Service
[32] Azure Kubernetes Service
[33] Google Kubernetes Engine

Como o escopo desta pesquisa não inclui a migração do SPIRA para um provedor de computação em nuvem, os serviços comerciais descritos não foram utilizados.

Quanto às distribuições de desenvolvimento local, o KinD foi escolhido porque com ele é possível simular um *cluster* com múltiplas máquinas, cada qual correspondendo a um contêiner Docker[34]. O MiniKube foi descartado, porque não oferece essa mesma capacidade.

[34] Docker

Capítulo 3

Arquitetura Original do Sistema SPIRA

Na implementação realizada por Tamae em 2022, o SPIRA foi hospedado em uma única máquina física. As seções seguintes apresentam os microsserviços constituintes do sistema SPIRA, o fluxo de inferência, a maneira como o sistema foi provisionado originalmente, e as limitações da solução que justificam as mudanças arquiteturais descritas no Capítulo 4.

3.1 Microsserviços de Infraestrutura

Como discutido na Seção 2.1, aplicações nativas à nuvem são compostas de múltiplos microsserviços. Tais microsserviços podem ser responsáveis por executar alguma lógica de negócio, ou prover serviços de infraestrutura.

Os componentes de lógica de negócio do SPIRA são responsáveis, principalmente, por cadastro e autenticação de usuários, armazenamento de metadados, e processamento de inferências a partir de áudios coletados dos pacientes. Ambos serão discutidos em mais detalhe no Capítulo 4.

Já os componentes de infraestrutura, são responsáveis por duas grandes funções: persistência de dados, e comunicação assíncrona entre serviços. Em particular, os componentes de infraestrutura do SPIRA são os seguintes:

- **MongoDB**[1]: O MongoDB é um banco de dados orientado a documentos. Também é utilizado no SPIRA para armazenar os dados do microsserviços da aplicação.
- **MLFlow**[2]: O MLFlow é um gerenciador do ciclo de vida de modelos de aprendizado de máquina. É utilizado no SPIRA para disponibilizar modelos de inferência.
- **PostgreSQL**[3]: O PostgreSQL é um banco de dados relacional. É utilizado para armazenar metadados do MLFlow.
- **MinIO**[4]: O MinIO é um sistema de armazenamento de arquivos, desenvolvido para ter uma API idêntica ao Amazon S3[5]. É utilizada para armazenar gravações dos pacientes, que serão submetidas para análise e usada pelo MLFlow para armazenar modelos de aprendizado de máquina.

[1] MongoDB
[2] MLFlow
[3] PostgreSQL
[4] MinIO
[5] S3

- **NATS**[6]: O NATS é um sistema de mensageria. É utilizado para comunicação assíncrona entre os microsserviços do SPIRA.

3.2 Microsserviços de Negócio

A principal responsabilidade dos microsserviços de negócio do `SPIRA` é executar o fluxo de inferência. O usuário se cadastra, acessa o *front-end* da aplicação, preenche um formulário, e envia amostras de voz, gerando assim uma requisição de inferência. Por fim, a inferência é processada, e o resultado fica disponível para visualização.

Os microsserviços de negócio são os seguintes:

- `SPIRA API Server`: é responsável pelo cadastro, autorização e autenticação de usuários, cadastro de modelos de inferência, e cadastro das inferências. Quando uma inferência é criada, uma notificação é enviada para posterior processamento por meio do NATS. Quando uma notificação de conclusão da inferência é então recebida pelo `SPIRA API Server`, que atualiza seu estado para completo e deixa o resultado disponível para consulta pelo usuário.
- `SPIRA Inference Server`: é responsável por executar uma inferência, quando solicitado via NATS. Dada uma inferência, coleta seus dados do MinIO e do MongoDB, carrega um modelo do MLFlow, executa a inferência, e disponibiliza a resposta de forma assíncrona por meio do NATS.

O fluxo de inferência está representado na Figura 3.1, que apresenta os microsserviços do `SPIRA` como atores num diagrama de sequência, mostrando quais operações e em qual ordem são executadas durante a criação e o processamento de uma inferência:

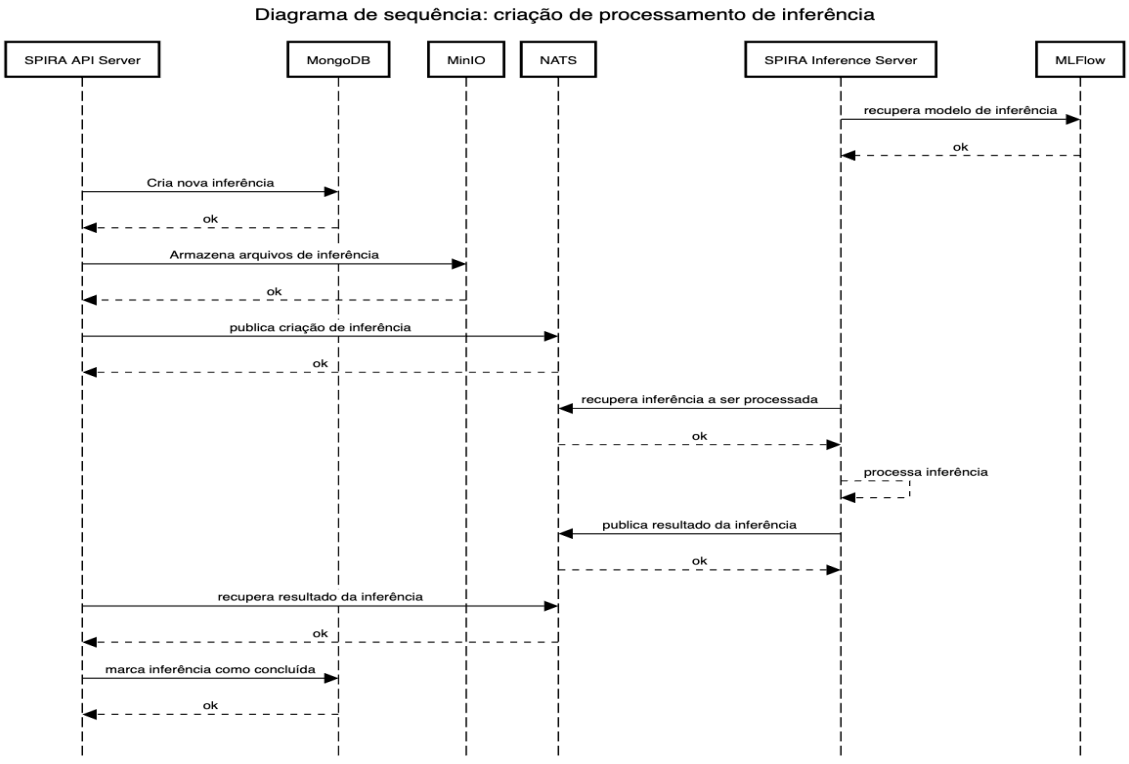

Figura 3.1: *Fluxo de inferência do `SPIRA`*

[6] NATS

3.3 Provisionamento do sistema

Na pesquisa de Tamae, todos os microsserviços do SPIRA foram provisionados em uma máquina física na Universidade de São Paulo, por meio do Docker Compose (introduzido na Subseção 2.1.5). Um *daemon* do Linux foi registrado, para garantir que todos os microsserviços sejam executados quando a máquina for iniciada. O sistema, por sua vez, fica exposto para uso por meio de redirecionamento *DNS*.

Os componentes do sistema provisionado em Docker Compose estão expostos na Figura 3.2:

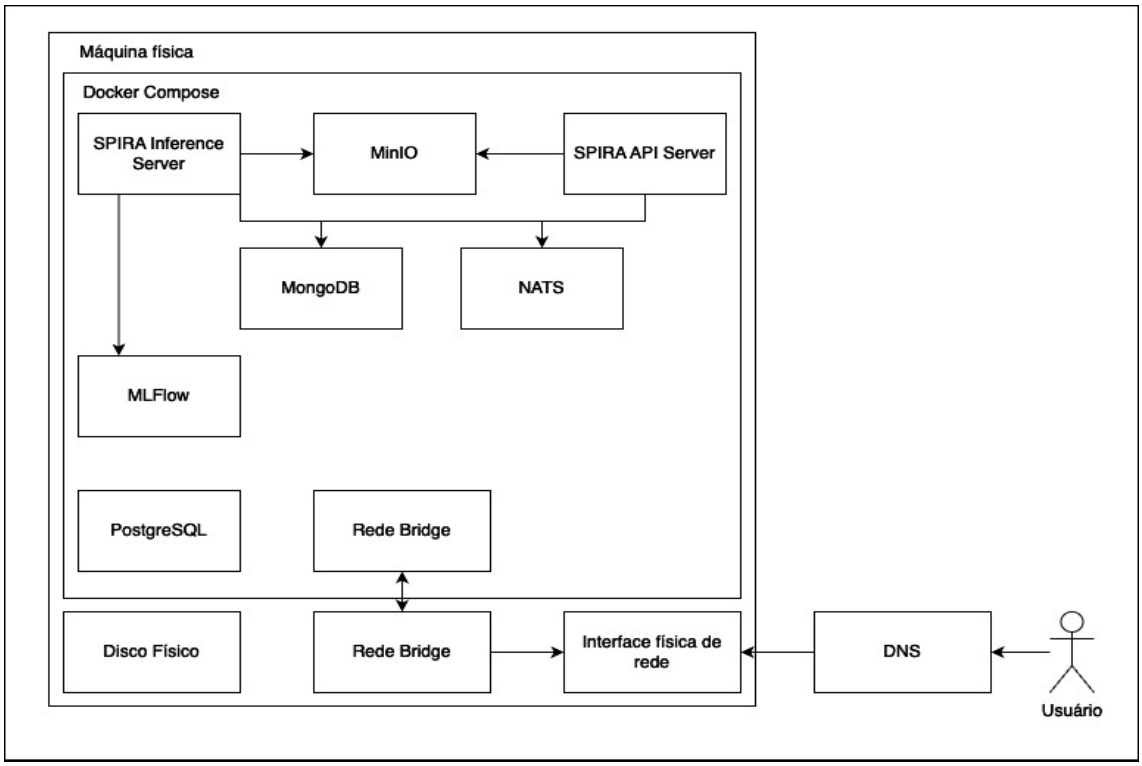

Figura 3.2: *Provisionamento original do SPIRA em Docker Compose*

3.4 Limitações da Arquitetura Original

Na arquitetura desenvolvida em Tamae, todos os microsserviços estão hospedados somente em uma máquina física. Ademais, cada microsserviço tem apenas uma réplica. Caso a máquina ou um componente crítico do SPIRA (por exemplo, o MongoDB) fique indisponível, seja por morte do processo ou qualquer outro fator, a aplicação também fica indisponível.

O Capítulo 4 apresenta a solução desenvolvida nesta pesquisa, para superar as limitações da solução original.

Capítulo 4

Arquitetura do SPIRA no Kubernetes

De modo a superar as limitações de resiliência apresentadas no Capítulo 3, esta pesquisa se propôs a transferir o SPIRA para o Kubernetes, de modo a executarem em mais de uma máquina, e ter mais de uma réplica por serviço. Isso elimina os pontos únicos de falha, e permite que usemos as funcionalidades de *self-healing* do Kubernetes (principalmente o reescalonamento de um POD) para tornar a aplicação resiliente.

A infraestrutura provisionada utilizou operadores e HELM CHARTS disponibilizados por empresas desenvolvedoras e comunidades de usuários. Com base neles, foi possível executar todos os fluxos da aplicação num *cluster* Kubernetes.

A partir deste ponto, o termo **arquitetura original** denominará a forma de provisionamento em Docker Compose, adotada por Tamae. Em contrapartida, o termo **arquitetura nativa à nuvem** denominará a forma de provisionamento do SPIRA em Kubernetes desenvolvida nesta pesquisa.

4.1 Simulação de *Cluster* em Ambiente de Desenvolvimento Local

Para provisionar um *cluster* de Kubernetes, foi empregado o KinD, uma distribuição de Kubernetes para desenvolvimento local, introduzido na seção Subseção 2.2.7. Nesta pesquisa, foi provisionado um NODE como plano de controle, e três NODES como plano de dados. Uma vez provisionado, o *cluster* pôde ser acessado por meio da ferramenta *kubectl*[1].

Durante o processo de provisionamento, o único problema que afetou o funcionamento do KinD foi a saturação de alguns limites do sistema operacional. Em particular, os limites relativos ao subsistema *inotify*[2] do Linux, responsável por gerar notificações de mudança no sistema de arquivos. A solução encontrada foi aumentar esses limites, como sugerido num tópico de suporte da SUSE[3].

4.2 Microsserviços de Infraestrutura

Esta seção apresentará a forma de provisionamento nativa à nuvem dos microsserviços de infraestrutura do SPIRA.

[1] kubectl CLI
[2] inotify
[3] SUSE: how to increase inotify limits

4.2.1 MinIO

O MinIO é um sistema de armazenamentos de objetos, com API compatível com o Amazon S3. No sistema SPIRA, ele é responsável por armazenar os áudios das inferências. Para provisioná-lo no Kubernetes, empregou-se o operador oficial da solução[4].

Uma vez implantado no *cluster*, o operador pode criar *tenants*: unidades funcionais do MinIO. A cada *tenant* é associado um objeto SERVICE, que possibilita o acesso ao serviço. Para o SPIRA, como se usa a solução somente para armazenar áudios, um único *tenant* foi suficiente.

Os componentes do Kubernetes provisionados após a criação de um *tenant* são os seguintes:

- um DEPLOYMENT para o operador, que gera dois PODS;
- um SERVICE para permitir comunicação com o operador;
- um DEPLOYMENT do console, que habilita uma interface gráfica para interagir com o operador;
- um SERVICE para permitir acesso ao *console*;
- um SERVICE para permitir compatibilidade com o serviço STS da AWS, de forma a garantir compatibilidade com as APIs originais da Amazon (não usado na pesquisa);
- um STATEFUL SET, que gera três PODS do MinIO, garantindo alta disponibilidade; e
- dois PERSISTENT VOLUME CLAIMS para cada POD, para garantir tolerância a falha no armazenamento dos dados.

A forma de provisionamento do MinIO no Kubernetes supera a deficiência da arquitetura original. Ao oferecer redundância no armazenamento, a aplicação previne perda de dados. Por ter três réplicas, tolera a perda de uma delas enquanto continua disponível recebendo escritas. Pode servir somente leituras se duas instâncias do serviço estiverem indisponíveis.

Houve duas dificuldades principais ao provisionar o MinIO no Kubernetes.

A primeira foi o fato do modo padrão de comunicação ser via HTTPS[5]. Isso força que as aplicações que usam o MinIO a ter certificados TLS[6] emitidos pelo próprio Kubernetes, o que aumentaria a complexidade da operação. Para contornar essa complexidade, o *tenant* original foi recriado, para aceitar o protocolo HTTP como método de comunicação com as aplicações. Isso evitou a necessidade de lidar com emissão de certificados dentro do Kubernetes.

A segunda foi o fato de aplicações em NAMESPACE diferente de um SERVICE precisarem usar a forma completa do DNS desse objeto. Após um longo processo de depuração, por simplicidade, o *tenant* foi provisionado no mesmo NAMESPACE das aplicações, contornando assim a necessidade de usar o endereço DNS[7] completo dos serviços.

[4] MinIO Operator
[5] HTTPS
[6] TLS
[7] DNS

4.2.2 NATS

O NATS é um sistema de mensageria projetado para ser nativo ao Kubernetes. Ele possibilita a comunicação assíncrona entre microsserviços. Nesta pesquisa, o provisionamento foi feito por meio do HELM CHART oficial[8]. Seus componentes são os seguintes:

- um SERVICE, para comunicação dos PODS de aplicação com a ferramenta,
- um STATEFULSET com três PODS, para garantir alta disponibilidade por meio de redundância, e
- um DEPLOYMENT de um POD, para permitir depuração (não utilizado nesta pesquisa).

Ao ter várias réplicas, a ferramenta pode publicar mensagens ainda que uma réplica esteja indisponível. Ademais, garante também leitura de mensagens quando até duas réplicas estiverem indisponíveis.

Como o NATS foi projetado para ser nativo à nuvem, não houve dificuldade em seu provisionamento.

4.2.3 MongoDB

O MongoDB é um banco de dados orientado a documentos, eventualmente consistente. No SPIRA, é utilizado para armazenar dados dos usuários e das inferências. Na arquitetura original, só havia uma réplica, o que poderia acarretar em indisponibilidade do SPIRA caso o banco de dados ficasse indisponível.

Na arquitetura nativa à nuvem, o problema de indisponibilidade foi resolvido ao oferecer uma implementação com três réplicas. Isso permite que a ferramenta aceite escritas quando uma réplica estiver indisponível, e sirva somente leituras quando duas réplicas estiverem indisponíveis.

Nesta pesquisa, foi empregado o *MongoDB Community Operator*[9]. Os componentes do Kubernetes provisionados para o MongoDB são os seguintes:

- um DEPLOYMENT de um POD para o operador da ferramenta,
- um STATEFUL SET de três PODS, de garantir alta disponibilidade,
- um PERSISTENT VOLUME CLAIM para cada POD, e
- um SERVICE que permite que as aplicações se comuniquem com a ferramenta.

O único contratempo enfrentado durante o provisionamento do MongoDB foi que, por padrão, a instrumentação é desabilitidada. Para habilitar a instrumentação, foi necessário adicionar campos no CRD do MongoDB, como demonstrado neste commit[10]. O CRD do MongoDB, utilizado para provisionar a ferramenta, pode ser encontrada no repositório desta pesquisa[11]. As métricas disponíveis para o MongoDB podem ser encontradas na documentação oficial[12]

[8] NATS helm chart
[9] MongoDB community operator
[10] Commit adicionando instrumentação no MongoDB
[11] MongoDB CRD
[12] Exported MongoDB metrics

4.2.4 MLFlow

O MLFlow é um sistema de gerenciamento do ciclo de vida de modelos de aprendizado de máquina. No SPIRA, ele é utilizado para versionar e armazenar os modelos de inferência. As instâncias do `SPIRA Inference Service`, no processo de inicialização, se comunicam com o MLFlow para obter os modelos que realizarão a inferência, que por sua vez ficam armazenados em memória enquanto a réplica estiver ativa.

Esta pesquisa empregou um HELM CHART[13] disponibilizado pela comunidade. Seus componentes são os seguintes:

- um DEPLOYMENT com três PODS, para prover alta disponibilidade,
- um SERVICE que permite comunicação das aplicações com o MLFlow, e
- referência a um banco de dados relacional, utilizado para armazenar metadados.

Em comparação à implementação original, o MLFlow deixa de ser um ponto único de falha. A garantia da permanência dos dados se dá ao esquema de redundância e replicação do banco de dados relacional escolhido. Nesta pesquisa, foi empregado PostgreSQL, cuja estratégia de resiliência é discutida na seção seguinte.

4.2.5 PostgreSQL

O PostgreSQL é um banco de dados relacional. Sua função no SPIRA é armazenar os metadados do MLFlow. Na arquitetura original, foi implantado com somente uma réplica, o que causava o risco do MLFlow não inicializar por não ter acesso a seus metadados. Consequentemente, isso poderia implicar na falha ao inicializar o SPIRA, ou impedir que novos PODS de MLFlow fossem escalonados para substituir PODS que falharam.

Para mitigar esse risco, a mesma estratégia das demais ferramentas foi utilizada: um STATEFUL SET com três réplicas. Nesta pesquisa, foi empregado o operador da Zalando[14]. Ele implementa alta disponibilidade pela cópia síncrona entre uma réplica que recebe escritas e as demais que servem somente leitura. Os componentes criados pelo Kubernetes para o provisionamento do PostgreSQL são os seguintes:

- um STATEFUL SET de três PODS, para garantir alta disponibilidade por meio de replicação;
- um PERSISTENT VOLUME CLAIM para cada POD do STATEFUL SET, de forma a permitir a persistência dos dados no banco;
- um DEPLOYMENT de um POD, para o operador da ferramenta;
- um DEPLOYMENT de um POD, para a interface gráfica do operador da ferramenta; e
- um SERVICE que permite que as aplicações se comuniquem com a ferramenta.

[13] MLFlow Helm Chart
[14] Zalando Postgres Operator

Na arquitetura original, o banco de dados relacional empregado pelo MLFlow é o MySQL[15]. Numa primeira iteração, o MySQL foi mantido, e provisionado por meio do operador oficial[16]. Porém, houve dificuldades com a ferramenta: ao reiniciar o computador hospedeiro do KinD, os PODS do MySQL eram incapazes de iniciar. Isso impossibilitava, consequentemente, a inicialização dos PODS de MLFlow, impedindo assim a inicialização do SPIRA como um todo.

Em razão disso, o banco de dados relacional foi substituído pelo PostgreSQL, que não apresentou o mesmo problema. O CRD do PostgreSQL utilizado para provisionar a ferramenta pode ser encontrada no repositório da pesquisa[17]. O mesmo vale para o CRD do MySQL[18].

4.2.6 Chaos Mesh

O Chaos Mesh é uma ferramenta que permite testes de caos dentro *cluster* Kubernetes, descritos em mais detalhes no Apêndice B. Para esta pesquisa, a ferramenta foi utilizada de forma a encerrar periodicamente PODS dos componentes do SPIRA, tanto de infraestrutura quanto de aplicação. Os componentes criados pelo Kubernetes para o provisionamento do Chaos Mesh são os seguintes:

- um DEPLOYMENT de um POD para o *chaos controller manager*, responsável por fazer o escalonamento de experimentos de caos;
- um DAEMON SET de um POD para o *chaos daemon*, responsável por interagir com a máquina física e gerar perturbações (por exemplo, aumentar o uso de CPU de um POD);
- um DEPLOYMENT de um POD do *chaos dashboard*, uma interface gráfica pela qual podemos criar e administrar experimentos de caos; e
- um DEPLOYMENT de um POD do *chaos dns server*, que interceptar chamadas de DNS e injetar falhas durante um experimento de caos.

A única dificuldade enfrentada no provisionamento da ferramenta foi o fato de os experimentos de caos serem criados por meio de interface gráfica. Para melhor reprodutibilidade, foi buscada uma forma de criar experimentos de forma programática, como apresentado no Apêndice B.

Após um processo de investigação, notou-se que, ao criar um experimento de caos, um CRD do Kubernetes era gerado no NAMESPACE onde a ferramenta foi provisionada. Ademais, como os experimentos podem ser representados como arquivos YAML, foi possível registrá-los em sistema de controle de versão[19], facilitando a reprodução programática dos experimentos.

4.2.7 Prometheus e Grafana

O Prometheus é um banco de dados de séries temporais. Sua função é armazenar e permitir consultas sobre métricas. Seus três principais tipos de métricas são os seguintes:

- *Gauge*: grandezas que podem ser incrementadas ou decrementadas ao decorrer do tempo. São úteis, por exemplo, para medir o número de requisições concorrentes processadas por um servidor HTTP, num intervalo de tempo pré-fixado.

[15] MySQL
[16] MySQL Operator
[17] CRD do PostgreSQL
[18] CRD do MySQL
[19] SPIRA chaos experiment files

- *Counter*: grandezas que somente crescem ou se mantém constantes ao longo do tempo. São úteis, por exemplo, para medir a quantidade acumulada de requisições atendidas por um servidor HTTP desde o momento em que foi iniciado.

- *Histogram*: grandezas contabilizadas em percentis. São úteis, por exemplo, para medir os percentis de latência de uma aplicação.

O Grafana, por sua vez, é uma interface gráfica que consome fontes de métricas e permite a criação de painéis para acompanhar a saúde dos sistemas. Nesta pesquisa, o Grafana foi provisionado junto ao Prometheus, permitindo a criação de painéis para acompanhar a saúde das aplicações de infraestrutura.

As aplicações de infraestrutura, por sua vez, foram instrumentadas de forma que o Prometheus fosse capaz de, periodicamente, realizar requisições HTTP a uma rota específica. Dessa forma, as métricas disponibilizadas pelas aplicações de infraestrutura (exceto o MLFlow, que não é instrumentado) são coletadas de forma automática. Mais detalhes de como a instrumentação foi realizada podem ser encontrados no Apêndice A.

Quanto aos microsserviços de negócio do `SPIRA` (`SPIRA API Server` e `SPIRA Inference Server`), esta pesquisa não os instrumentou. Para tal, é necessário editar o código das aplicações em questão, disponibilizando uma rota de métricas que possa ser interpretada pelo Prometheus. Tal tarefa poderá ser realizada como trabalho futuro.

Os componentes criados pelo Kubernetes para o provisionamento do Prometheus e Grafana são os seguintes:

- um DAEMON SET de um POD do Node Exporter, para coletar métricas das máquinas constituintes do *cluster*;

- um DEPLOYMENT de um POD do Grafana, para disponibilizar os painéis das aplicações;

- um SERVICE para o Grafana, para permitir o acesso a sua interface gráfica;

- um STATEFUL SET de um POD do Prometheus, para permitir o armazenamento e a obtenção de métricas;

- um SERVICE, para possibilitar acesso ao Prometheus;

- um DEPLOYMENT de um POD para o Kube State Metrics[20], para exportar métricas sobre os objetos do Kubernetes no *cluster* (PODS, DEPLOYMENTS, etc.);

- um DEPLOYMENT de uma réplica para o Grafana Alert Manager[21], que possibilita a criação de alarmes caso a saúde de alguma aplicação deteriore (não utilizado nessa pesquisa); e

- um DEPLOYMENT de um POD para o Prometheus OPERATOR, que gerencia o ciclo de vida das aplicações mencionadas acima.

[20] Kube State Metrics
[21] Grafana Alert Manager

4.3 Microsserviços de Negócio

Esta seção apresentará a forma de provisionamento nativa à nuvem dos microsserviços de negócio do SPIRA.

4.3.1 SPIRA API Server

O SPIRA API Server é o microsserviço do SPIRA responsável pela criação de usuário, autenticação, criação de inferências, dentre outras responsabilidades (Tamae, 2022). Uma decisão de projeto que facilitou o provisionamento da aplicação no Kubernetes foi o fato de consumir seus parâmetros de configuração por meio de variáveis de ambiente, como sugerido nos princípios da *Twelve-Factor App*[22].

Em razão disso, não foi necessário alterar o código para a aplicação funcionar no Kubernetes. Bastou criar ConfigMaps[23] e Secrets[24], que guardam principalmente:

- credenciais padrão do *SPIRA API Server*,
- credenciais para acesso ao MINIO,
- credenciais de acesso ao MongoDB,
- parâmetros de configuração para a autenticação JWT (Tamae, 2022), e
- *endpoints* DNS para acesso aos serviços de infraestrutura, gerados nas etapas anteriores do presente capítulo.

Em seguida, foi criado um arquivo YAML[25] definindo um objeto do tipo Deployment, que referencia os Secrets e ConfigMaps para carregar as variáveis de ambiente nos pods da aplicação. Ademais, no mesmo arquivo YAML, é também definido um objeto do tipo Service, que expõe a aplicação para outros pods no *cluster*.

A maior complexidade durante o processo de provisionamento da aplicação foi que, quando implantado, o MongoDB não possui o usuário padrão utilizado pelo SPIRA para acessar suas tabelas. Para contornar esse problema, foi empregado um *init container*[26], que cria o usuário esperado por meio de um *script*[27].

4.3.2 SPIRA Inference Server

O SPIRA Inference Server é o serviço responsável por consumir solicitações de inferência em um tópico do NATS, processá-las, e enviar o resultado do processamento para um outro tópico.

Por construção, os Secrets e ConfigMaps empregados no *SPIRA API Server* são reutilizáveis pelo *SPIRA Inference Server*. Em razão disso, o processo de provisionamento de ambas as aplicações é semelhante.

[22] Twelve-Factor App
[23] SPIRA configmaps
[24] Spira secrets
[25] SPIRA API Server deployment file
[26] Init containers
[27] SPIRA API Server init script

A maior dificuldade encontrada no provisionamento do `SPIRA Inference Server` foi que, para a aplicação conseguir iniciar, o modelo a ser utilizado deve estar registrado no MLFlow e no MongoDB, e alguns *buckets* no MinIO devem existir previamente. Para contornar esse problema, a solução empregada foi similar à adotada no `SPIRA API Server`: usa um *init container* que inicializa as dependências descritas por meio de um *script*[28].

A forma de provisionamento e exposição da aplicação também é semelhante ao `SPIRA API Server`: foi criado um arquivo YAML[29], que define um objeto SERVICE e um objeto DEPLOYMENT.

4.4 Conclusão

O provisionamento de um *cluster Kubernetes* via KinD permitiu o provisionamento de todos os microsserviços do `SPIRA`, tanto na camada de infraestrutura quanto na camada de negócio. A implantação do PostgreSQL, do MLFlow e do NATS possibilitou provisionar o `SPIRA Inference Server`, ao passo que o provisionamento do MongoDB e do MinIO possibilitou provisionar o `SPIRA API Server` - que foi exposto por meio de um objeto do tipo SERVICE para aplicações dentro do *cluster*. Adicionalmente, foram provisionados o Prometheus e o Grafana, permitindo assim coletar métricas e criar painéis para acompanhar a saúde dos sistemas de infraestrutura. Finalmente, foi provisionada a ferramenta Chaos Mesh.

O Capítulo 5 discutirá a estratégia de testes adotada, de modo a validar que os fluxos da aplicação se mantiveram funcionais, e que, além disso, o sistema ganhou alta disponibilidade, cumprindo assim o objetivo desta pesquisa.

[28] SPIRA Inference Server init script
[29] SPIRA Inference Service deployment file

Capítulo 5

Testes

A implantação do SPIRA no ambiente Kubernetes se deu em etapas. Primeiro, foi criado o *cluster* KinD, com uma máquina de plano de controle, e três máquinas de plano de dados. As dependências foram provisionadas por meio de seus respectivos operadores e HELM CHARTS. Por fim, foram criados os DEPLOYMENTS e SERVICES dos microsserviços de negócio (`SPIRA Inference server` e `SPIRA API server`).

Uma vez que os microsserviços estavam sendo executados no *cluster*, foi necessário verificar que os microsserviços de negócio conseguiam se comunicar com as dependências de infraestrutura. Isso foi feito tomando como base os testes de integração previamente implementados (Tamae, 2022).

Por fim, o fluxo ponta a ponta da aplicação foi verificado por meio de um teste manual, e a alta disponibilidade foi verificada por meio de um teste de caos com a ferramenta Chaos Mesh.

5.1 Testes de Integração

Segundo Richardson, no livro *Microservices Patterns*, **testes de integração** são responsáveis por testar a comunicação de um microsserviço com suas dependências, sejam microsserviços de infraestrutura ou de negócio. Como podemos controlar o DNS dos componentes de infraestrutura nos testes de integração, bastava editar as configurações apropriadas para que os testes de integração desenvolvidos por Tamae fossem reaproveitados no ambiente Kubernetes.

Um complicador, porém, foi o processo de compilação do `SPIRA Inference Server` e do `SPIRA API Server`. Os artefatos de teste não eram copiados para a imagem de produção. Para resolver tal problema, foram criadas versões de teste da imagem (que mantinham os arquivos de teste), permitindo assim a execução dos testes de integração.

De posse dessas imagens, foi possível criar PODS contendo os artefatos de teste. Para o `SPIRA API Server`, foi criado um arquivo YAML[1] definindo um POD capaz de executar os testes de integração. O mesmo foi feito para o `SPIRA Inference Server`[2].

Posteriormente, foi possível estabelecer um ciclo de testes. Iterativamente, foram construídos e validados os CONFIGMAPS e SECRETS da aplicação, como discutido no Capítulo 4. Eventualmente, todos os testes de integração passaram e a nova arquitetura foi validada. Para mais detalhes em como reproduzir os testes de integração, consulte o Apêndice B.

[1] SPIRA API Integration test pod CRD
[2] SPIRA Inference Server integration tests image

5.2 Testes Ponta a Ponta

Não basta que todos os componentes sejam capazes de se comunicar: é necessário verificar que, no ambiente Kubernetes, o fluxo de inferência se manteve funcional. Para tanto, foram adotados **testes ponta a ponta**, que segundo o livro *Microservices Patterns: With examples in Java*, consistem em provisionar o sistema completo e testar sua corretude. Nesta pesquisa, não foram desenvolvidos testes ponta a ponta automatizados. Após o provisionamento da aplicação, o fluxo de inferência foi testado manualmente.

Como o único componente do `SPIRA` acessado de forma externa ao sistema é o `SPIRA API Server`, havia duas opções: implementar um INGRESS CONTROLLER[3] no *cluster*, que possibilitaria expor o SERVICE do `SPIRA API Server` para a Internet, ou utilizar o *port-forward*[4] para expor, temporariamente, a aplicação em execução local.

Por simplicidade, foi empregado o `kubectl port-forward` em conjunção com o ngrok[5]. Isso possibilitou que o `SPIRA API Server` ficasse exposto na máquina local por meio de uma porta arbitrária. O ngrok, por sua vez, disponibiliza uma conexão HTTPS pública na Internet que realiza um túnel com a porta exposta na máquina local.

Uma vez que o Netlify apontava para o *back-end* do SPIRA no Kubernetes, foi possível realizar o fluxo de inferência: autenticar na página de `login`, criar uma inferência, e fazer *upload* dos arquivos. Isso gerou uma requisição de inferência pelo `SPIRA API Server`, recebida via NATS pelo `SPIRA Inference Server` e processada. Finalmente, a inferência foi registrada como concluída no MongoDB.

Para verificar que o fluxo foi concluído, uma chamada HTTP foi realizada via cURL[6] para o `SPIRA API Server`. Para mais detalhes de como reproduzir o fluxo ponta a ponta, consulte o Apêndice B.

5.3 Testes de Caos

Validado o fluxo ponta a ponta da aplicação, resta verificar que de fato o Kubernetes provê alta disponibilidade ao sistema, o que não era possível na arquitetura original. Para isso, foram realizados testes de caos empregando a ferramenta Chaos Mesh, cujo provisionamento foi discutido no Capítulo 4. Segundo Garrison and Nova, no livro *Cloud Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment*, **testes de caos** são aqueles em que disrupções são provocadas intencionalmente na infraestrutura de sistemas de produção, de modo a expor modos de falha que não se manifestam em situações normais.

O experimento em si utilizou *schedules*[7] do Chaos Mesh, que rodavam continuamente no *cluster* Kubernetes. Para cada microsserviço, tanto de infraestrutura quanto de negócio, a cada minuto era destruído um POD, escolhido de forma arbitrária entre todos os PODS existentes no momento. Os CRDs dos *schedules* em questão podem ser encontrados no repositório da pesquisa[8].

Para cada inferência cadastrada no sistema a receber um status HTTP 200 (OK) como resposta

[3] Ingress controllers
[4] `kubectl port-forward`
[5] ngrok
[6] curl
[7] Chaos Mesh schedules documentation
[8] SPIRA Chaos Mesh schedules

do SPIRA API Server, a inferência em questão deveria ser eventualmente processada e marcada como completa. Se tais condições fossem verdade, o experimento seria dado como concluído com sucesso.

Havia, porém, uma limitação no código do SPIRA API Server: a rota de criação de inferência não devolvia o identificador da inferência na resposta. Sem isso, era impossível saber se as criações de inferência respondidas com sucesso seriam de fato processadas. Para contornar a situação, a implementaç˜ao foi alterada e para devolver o identificador da inferência[9].

Cumprido esse requisito, foi possível iniciar o teste de caos. Foi construído um *script* Python[10] para enviar requisições ao SPIRA API Server enquanto os PODS eram constantemente encerrados. Esse *script* foi executado num POD[11] construído para esse propósito. Para cada chamada respondida com sucesso, o *script* escrevia o identificador devolvido em um arquivo de texto.

Após um intervalo de tempo arbitrário, as chamadas HTTP de criação de inferência são encerradas. Nesse momento, um segundo *script*[12] é executado, cujo propósito é consumir todos os identificadores de inferência que foram criados com sucesso durante o teste. Em seguida, fazia uma única chamada HTTP para o SPIRA Inference Server, para descobrir o estado de todas as inferências realizadas desde o início da aplicação. Sendo assim, o *script* pode determinar se todas as inferências criadas durante o teste de caos foram processadas com sucesso.

No experimento realizado, o *script* executou por 1h. Ao final, todos os identificadores de inferência registrados constavam como processadas. Isso demonstra que, mesmo com o encerramento constante de PODS dos microsserviços de negócio e infraestrutura, o SPIRA respondia e finalizava as inferências com sucesso. **Portanto, o objetivo da pesquisa foi concluído com sucesso: a aplicação de fato obteve alta disponibilidade**.

Para reproduzir os testes mencionados neste capítulo, consulte o Apêndice B.

[9] Commit to return inference ID on inference creation
[10] SPIRA Inference flood script
[11] SPIRA API flood pod
[12] SPIRA chaos test verification script

Capítulo 6

Conclusão

Esta pesquisa tinha como objetivo **prover alta disponibilidade para o sistema `SPIRA`**, por meio da migração do mesmo para uma arquitetura nativa à nuvem. Para tal, todos os microsserviços de infraestrutura e negócio foram adaptados para funcionar no orquestrador de contêineres Kubernetes.

O processo de migração foi possível pelo reúso dos testes de integração implementados previamente por Tamae, com alterações pontuais nas configurações de DNS e a criação de imagens de teste para uso na esteira de integração contínua. Após tais modificações, todos os componentes de infraestrutura e de negócio se comunicaram no *cluster* Kubernetes com sucesso.

O fluxo de inferência foi validado manualmente, por meio de um teste ponta a ponta, ao apontar o *front-end* servido pelo Netlify para o *back-end* executado no Kubernetes, com o auxílio da ferramenta ngrok e da funcionalidade de `port-forward` do próprio Kubernetes. Uma inferência foi criada e processada com sucesso, o que provou que o fluxo da aplicação se manteve funcional em ambiente nativo à nuvem.

Por fim, a alta disponibilidade foi demonstrada por meio de um experimento de caos, implementado com a ferramenta Chaos Mesh. PODS eram periodicamente eliminados, mas o Kubernetes foi capaz de escalonar novas réplicas saudáveis. Ademais, todas as requisições direcionadas ao *cluster* e registradas pelo `SPIRA API Server` foram processadas com sucesso.

Sendo assim, os objetivos desta pesquisa foram cumpridos. Como trabalhos futuros, há a possibilidade de:

- empacotar os objetos Kubernetes da solução nativa à nuvem do `SPIRA` como um HELM CHART para simplificar ainda mais o provisionamento do `SPIRA` em novos ambientes Kubernetes;
- provisionar o sistema `SPIRA` em um provedor de nuvem, para colher os benefícios de uma implementação nativa à nuvem em múltiplas máquinas;
- expandir os testes de caos para além do encerramento de PODS, incluindo corrupção e/ou remoção de volumes persistentes, bem como o encerramento periódico de máquinas do *cluster*; e
- instrumentar o `SPIRA API Server` e o `SPIRA Inference Server` para coletar métricas dos microsserviços de negócio, aproveitando o provisionamento de Prometheus e Grafana feito nesta pesquisa.

Apêndice A

Provisionando o SPIRA em *cluster* KinD

Este apêndice registra o processo de provisionamento dos microsserviçøs do SPIRA, tanto de negócio quanto de infraestrutura. Além disso, documenta como reproduzir os fluxos de teste: integração, ponta a ponta, e caos. Devido a dependências entre os componentes, há restrições na ordem em que os provisionamentos devem ser executados. Uma dessas possíveis ordens é apresentada aqui.

As seguintes seções tratam das etapas individuais de provisionamento. Inicialmente, é necessário criar um *cluster* Kubernetes, provisionado via KinD. Em seguida, é necessário provisionar os componentes de infraestrutura e, por fim, os componentes de negócio. Uma vez que todos estejam provisionados, é possível executar os fluxos de teste.

O procedimento aqui descrito foi executado em sistema operacional Ubuntu 22.04, com processador AMD Ryzen 3500U, e 32GB de RAM. O *kernel* Linux utilizado foi 6.2.0-34-generic, KinD versão 0.18.0, Kubernetes versão 1.26.3, e Docker versão 23.0.5.

Nos exemplos abaixo, comandos iniciados com `$` são executados por usuários regulares, comandos iniciados em `#` são executados por super-usuários, e comandos iniciados por `#` são executados em outro *terminal*.

A.1 KinD

O primeiro passo para ter um *cluster* Kubernetes para desenvolvimento local é a instalação do KinD. As instruções podem ser encontradas na documentação oficial da ferramenta[1].

Uma vez que o KinD esteja instalado, é necessário criar um *cluster*. Para tal, gere um arquivo de configuração, denominado *kind.yaml*, com um NODE de plano de controle, e três NODES de plano de dados:

```
1 kind: Cluster
2 apiVersion: kind.x-k8s.io/v1alpha4
3 nodes:
4 - role: control-plane
5 - role: worker
```

[1] KinD installation instructions

```
6 - role: worker
7 - role: worker
```

Para a criar o *cluster*, execute o seguinte comando:

```
$ kind create cluster --config kind.yaml --name spira
```

Se tudo der certo, devemos ver os NODES do *cluster* ao executar o seguinte comando:

```
$kubectl get nodes
NAME                  STATUS   ROLES           AGE   VERSION
spira-control-plane   Ready    control-plane   97s   v1.26.3
spira-worker          Ready    <none>          76s   v1.26.3
spira-worker2         Ready    <none>          76s   v1.26.3
spira-worker3         Ready    <none>          76s   v1.26.3
```

Por fim, devemos aumentar o limite de eventos de modificação no sistema de arquivos (*inotify*), para evitar os problemas enfrentados durante a pesquisa. Para tal, fora executados os seguintes comandos:

```
# sysctl fs.inotify.max_user_instances=8192
# sysctl fs.inotify.max_user_watches=524288
# sysctl -p
```

Note, porém, que os comandos acima devem ser executados a cada reinício da máquina de desenvolvimento. Alternativamente, aumente os limites em caráter permanente, seguindo as orientações encontradas no suporte da *Suse*[2].

Após executar as instruções acima, será possível acessar o *cluster* via `kubectl`[3].

Um detalhe importante: por simplicidade, **todos os microsserviços, tanto na camada de negócio quanto de infraestrutura, foram instaladas no NAMESPACE `spira`**.

A.2 MinIO

O provisionamento do MinIO se dá por meio do seu operador oficial, que por sua vez é instalado com a ferramenta *krew*. Suas instruções de instalação podem ser encontradas na documentação oficial[4].

Uma vez que o *krew* esteja instalado, instale o operador do MinIO executando os seguintes comandos:

```
$ kubectl krew update
$ kubectl krew install minio
$ kubectl minio init
$ kubectl minio proxy
```

[2] Inotify limit increase guide
[3] kubectl
[4] krew

Após executar os dois últimos comandos com sucesso, é esperado que os seguintes recursos tenham sido criados no NAMESPACE *minio-operator*:

```
namespace/minio-operator created
serviceaccount/minio-operator created
clusterrole.rbac.authorization.k8s.io/minio-operator-role created
clusterrolebinding.rbac.authorization.k8s.io/minio-operator-binding created
customresourcedefinition.apiextensions.k8s.io/tenants.minio.min.io created
customresourcedefinition.apiextensions.k8s.io/policybindings.sts.min.io created
service/operator created
service/sts created
deployment.apps/minio-operator created
serviceaccount/console-sa created
secret/console-sa-secret created
clusterrole.rbac.authorization.k8s.io/console-sa-role created
clusterrolebinding.rbac.authorization.k8s.io/console-sa-binding created
configmap/console-env created
service/console created
deployment.apps/console created
-----------------

To open Operator UI, start a port forward using this command:

kubectl minio proxy -n minio-operator
```

O comando `minio proxy -n minio-operator` deve imprimir uma mensagem no terminal, informando um *token* de acesso para a interface gráfica do MinIO. A saída do comando se parece com o seguinte:

```
Starting port forward of the Console UI.

To connect open a browser and go to http://localhost:9090

Current JWT to login: ...

Forwarding from 0.0.0.0:9090 -> 9090
```

De posse do *token* de acesso, crie um *tenant*, que nessa pesquisa apresenta as seguintes configurações:

- o nome do *tenant* deve ser spira,
- o tenant deve ser criado no namespace `spira`,
- três réplicas,

- uma cpu por réplica,
- 2Gi de RAM por réplica, e
- HTTPS/TLS desabilitado,

Uma vez criado, o *tenant* gerará algumas credenciais de acesso. Elas devem ser guardadas porque os valores compõem um dos SECRETS das aplicações de negócio do SPIRA, que devem ser atualizados em caso de criação inicial (ou recriação) do *tenant* de MinIO.[5].

Segue abaixo um exemplo do arquivo gerado com as credenciais:

```
{
    "url":"http://minio.spira.svc.cluster.local:80",
    "accessKey":"QgqzLCpDkd1M2tol",
    "secretKey":"M8jXhkLQhLTkrzzYUPB3OhozkIQ9cPFJ",
    "api":"s3v4",
    "path":"auto"
}
```

Por fim, é necessário criar os *buckets* de nome `mlflow` e `audio-files` manualmente. Caso contrário, as aplicações de negócio do SPIRA não funcionarão nos passos seguintes. Isso pode ser feito pela mesma interface gráfica empregada na criação dos *tenants*.

Ao final do processo, os seguintes recursos devem estar presentes no NAMESPACE SPIRA:

```
$ kubectl get all -n spira
NAME                  READY   STATUS    RESTARTS   AGE
pod/spira-pool-0-0    2/2     Running   0          14m
pod/spira-pool-0-1    2/2     Running   0          14m
pod/spira-pool-0-2    2/2     Running   0          14m

NAME                     TYPE           CLUSTER-IP     AGE
service/minio            LoadBalancer   10.96.234.46   14m
service/spira-console    LoadBalancer   10.96.203.34   14m
service/spira-hl         ClusterIP      None           14m

NAME                              READY   AGE
statefulset.apps/spira-pool-0     3/3     14m
```

A.3 MongoDB

O provisionamento do MongoDB se dá a partir do *MongoDB Community Operator*[6]. O operador, por sua vez, é instalado a partir da ferramenta Helm com os seguintes comandos:

[5] SPIRA MinIO secret CRD
[6] MongoDB Community Operator

```
$ helm repo add mongodb https://mongodb.github.io/helm-charts
$ helm install community-operator mongodb/community-operator \
--namespace=spira --create-namespace
```

Em caso de sucesso, são esperados os seguintes recursos no NAMESPACE `spira`:

```
NAME                                                READY   STATUS    AGE
mongodb-kubernetes-operator-695bd4bd9b-qffq6        1/1     Running   2m
NAME                                               READY   AVAILABLE   AGE
deployment.apps/mongodb-kubernetes-operator        1/1     1           2m15s
NAME                                                          READY   AGE
replicaset.apps/mongodb-kubernetes-operator-695bd4bd9b        1       2m15s
```

Uma vez que o operador esteja instalado, é necessário aplicar o CRD do MongoDB, comentando a parte de SERVICEMONITOR[7].

De posse do arquivo, execute o comando:

```
$ kubectl apply -f mongo.yml -n spira
```

Após a execução do comando acima, é esperada a presença dos seguintes recursos no NAMESPACE `spira`:

```
NAME                                          READY   STATUS    AGE
example-mongodb-0                             2/2     Running   92m
example-mongodb-1                             2/2     Running   55m
example-mongodb-2                             2/2     Running   37m
NAME                            TYPE           CLUSTER-IP   AGE
service/example-mongodb-svc     ClusterIP      None         93m
NAME                                       READY   AGE
statefulset.apps/example-mongodb           3/3     93m
statefulset.apps/example-mongodb-arb       0/0     93m
```

Para verificar que a aplicação foi provisionada com sucesso, execute um contêiner efêmero de depuração[8]. Uma vez que o arquivo esteja aplicado, é esperado que um POD tenha sido criado:

```
$ kubectl get pods -n spira
NAME                                          READY   STATUS    AGE
mongo-shell-debug-86557c77b-vnqgt             1/1     Running   22m
```

Por fim, é possível executar commandos dentro do POD e verificar o funcionamento da ferramenta:

```
$ kubectl apply -f mongo_dbg.yml -n spira
$ kubectl exec -it mongo-shell-debug-86557c77b-vnqgt -- /bin/bash
> mongosh $CONNECTION_URL
```

[7] MongoDB CRD
[8] MongoDB debug container

Uma vez dentro da *CLI* do MongoDB, podemos realizar operações normalmente:

```
$ kubectl exec -it mongo-shell-debug-86557c77b-vnqgt -- /bin/bash
> root@mongo-shell-debug-86557c77b-vnqgt:/# mongosh $CONNECTION_URL
Current Mongosh Log ID:        6551a4b0d4aabc99cda7596d
...

example-mongodb [primary] admin> show dbs
admin   172.00 KiB
config  184.00 KiB
```

A.4 PostgreSQL

Para a instalação do PostgreSQL, é necessário instalar a ferramenta Helm. Instruções de instalação podem ser encontradas na documentação oficial da ferramenta[9]. Uma vez que o Helm esteja instalado, o operador do PostgreSQL pode ser instalado com os seguintes comandos:

```
$helm repo add postgres-operator-charts \
https://opensource.zalando.com/postgres-operator/charts/postgres-operator

$ helm install postgres-operator postgres-operator-charts/postgres-operator

$ helm repo add postgres-operator-ui-charts \
https://opensource.zalando.com/postgres-operator/charts/postgres-operator-ui

$ helm install postgres-operator-ui \
postgres-operator-ui-charts/postgres-operator-ui
```

Com a execução dos comandos acima, é esperado que os seguintes recursos sejam criados:

```
NAME                                                READY          AGE
pod/postgres-operator-685c7577c4-p8jcp              1/1            2m49s
pod/postgres-operator-ui-6d5f555877-tvwws           1/1            2m5s
NAME                                                TYPE           AGE
service/postgres-operator                           ClusterIP      2m49s
service/postgres-operator-ui                        ClusterIP      2m5s
NAME                                                READY          AGE
deployment.apps/postgres-operator                   1/1            2m49s
deployment.apps/postgres-operator-ui                1/1            2m5s
NAME                                                READY          AGE
replicaset.apps/postgres-operator-685c7577c4        1              2m49s
replicaset.apps/postgres-operator-ui-6d5f555877     1              2m5s
```

[9] Helm install instructions

Uma vez que o operador esteja instalado, basta aplicar o CRD do PostgreSQL gerado nesta pesquisa. Para tal, considerando o arquivo *postgres.yml*[10] esteja presente no diretório local, basta executar:

```
$ kubectl apply -f postgres.yml -n spira
```

Após a execução do comando acima, é esperado que os seguintes recursos tenham sido criados:

```
NAME                                    READY   AGE
pod/spira-pg-cluster-0                  1/1     10m
pod/spira-pg-cluster-1                  1/1     8m56s
pod/spira-pg-cluster-2                  1/1     6m37s
NAME                                    TYPE           AGE
service/spira-pg-cluster                ClusterIP      10m
service/spira-pg-cluster-config         ClusterIP      8m50s
service/spira-pg-cluster-repl           ClusterIP      10m
NAME                                    READY          AGE
statefulset.apps/spira-pg-cluster       3/3            10m
```

Por fim, crie manualmente o banco *metadata*, que é necessário na inicialização do MLFlow. Para tal, usamos os seguintes comandos:

```
# apt install postgresql-client-common
# apt-get install postgresql-client
$ export PGPASSWORD=$(kubectl get secret  \
postgres.spira-pg-cluster.credentials.postgresql.acid.zalan.do \
-o 'jsonpath={.data.password}' | base64 -d) \
$ export PGMASTER=$(kubectl get pods -o jsonpath={.items..metadata.name} -l \
application=spilo,cluster-name=spira-pg-cluster,spilo-role=master -n spira)
$ export PGSSLMODE=require
$ kubectl port-forward $PGMASTER 6432:5432 -n spira
```

Isso fará o PostgreSQL ser exposto na porta 6432. Por fim, em outro terminal, execute o login, utilizando a senha armazenada na variável de ambiente `PGPWASSWORD`:

```
$ psql -U postgres -h localhost -p 6432
...
postgres=# create database spirametadata;
```

Por fim, a instalação da ferramenta está completa e validada.

A.5 NATS

A instalação do NATS também se dá por meio de um Helm Chart. A única complicação é que, para que a coleta de métricas via Prometheus funcione, é necessário sobrescrever uma parte do Helm Chart. Para tal, é necessário ter o arquivo de *override* no diretório local[11].

[10] PostgreSQL CRD
[11] Nats instrumentation override

De posse do arquivo, basta executar:

```
$ helm repo add nats https://nats-io.github.io/k8s/helm/charts/
$ helm install my-nats nats/nats --values nats_scrape_config_override.yaml
```

A execução dos comandos acima deve ter criado os seguintes recursos:

```
NAME                                  READY   STATUS    AGE
pod/my-nats-0                         2/2     Running   8m43s
pod/my-nats-1                         2/2     Running   24m
pod/my-nats-2                         2/2     Running   24m
pod/my-nats-box-547fd47df-j4jvz       1/1     Running   24m
NAME                                  TYPE           AGE
service/my-nats                       ClusterIP      24m
service/my-nats-headless              ClusterIP      24m
NAME                                  READY   AGE
deployment.apps/my-nats-box           1/1     24m
NAME                                  READY   AGE
replicaset.apps/my-nats-box-547fd47df 1       24m
NAME                                  READY   AGE
statefulset.apps/my-nats              3/3     24m
```

Para verificar que a instalação ocorreu com sucesso, basta executar:

```
$ kubectl exec -n spira -it deployment/my-nats-box -- /bin/sh -l
> nats sub test & nats pub test hi
                          _                  _
 _ __    __ _| |_ ___       | |__   _____  __
| '_ \ / _` | __/ __|_____| '_ \ / _ \ \/ /
| | | | (_| | |_\__ \_____| |_) | (_) >  <
|_| |_|\__,_|\__|___/     |_.__/ \___/_/\_\

nats-box v0.13.8
my-nats-box-547fd47df-j4jvz:~# nats sub test & nats pub test hi
05:34:43 Subscribing on test
05:34:43 Published 2 bytes to "test"
```

A.6 MLFlow

O MLFlow também é instalado por meio de um Helm Chart. Há, porém, alguns parâmetros em suas dependências que dependem de provisionamentos anteriores: as credenciais do MinIO, geradas na Seção A.2, e as credenciais do PostgreSQL, geradas na Seção A.4.

As credenciais do PostgreSQL podem ser obtidas com os seguintes comandos, executados a partir do namespace `spira`.

```
$ export MINIO_ACCESS_KEY_ID=QgqzLCpDkd1M2tol
$ export MINIO_SECRET_ACCESS_KEY=M8jXhkLQhLTkrzzYUPB3OhozkIQ9cPFJ
$ export PGPASSWORD=$(kubectl get secret \
postgres.spira-pg-cluster.credentials.postgresql.acid.zalan.do -o \
'jsonpath={.data.password}' | base64 -d)
$ export PGUSER=$(kubectl get secret \
postgres.spira-pg-cluster.credentials.postgresql.acid.zalan.do -o \
'jsonpath={.data.username}' | base64 -d)
```

De posse dessas constantes, provisione o MLFlow com o seguinte comando:

```
$ helm upgrade --install mlflow community-charts/mlflow \
  --set replicaCount=3 \
  --set backendStore.databaseMigration=true \
  --set backendStore.postgres.enabled=true \
  --set backendStore.postgres.host=spira-pg-cluster \
  --set backendStore.postgres.port=5432 \
  --set backendStore.postgres.database=spirametadata \
  --set backendStore.postgres.user=$PGUSER \
  --set backendStore.postgres.password=$PGPASSWORD \
  --set artifactRoot.s3.enabled=true \
  --set artifactRoot.s3.bucket=mlflow \
  --set artifactRoot.s3.awsAccessKeyId=$MINIO_ACCESS_KEY_ID \
  --set artifactRoot.s3.awsSecretAccessKey=$MINIO_SECRET_ACCESS_KEY \
  --set extraEnvVars.MLFLOW_S3_ENDPOINT_URL=minio:80 \
  --set serviceMonitor.enabled=true
```

A execução do comando acima deve criar os seguintes recursos:

```
NAME                                    READY           AGE
pod/mlflow-8455675bf9-chh5v             1/1             6m29s
pod/mlflow-8455675bf9-dm7qg             1/1             6m29s
pod/mlflow-8455675bf9-sndgn             1/1             6m29s
NAME                                    TYPE            AGE
service/mlflow                          ClusterIP       6m29s
NAME                                    READY           AGE
deployment.apps/mlflow                  3/3             6m29s
NAME                                    READY           AGE
replicaset.apps/mlflow-8455675bf9       3/3             6m29s
```

Para verificar que a aplicação de fato funciona, faça um *port-forward* e redirecione o SERVICE da aplicação para uma porta local. Por fim, acesse o endereço abaixo e verifique se a interface gráfica da ferramenta é exibida:

```
$ kubectl port-forward service/mlflow 5000:5000
```

A.7 SPIRA API Server

Uma vez que todos os componentes de infraestrutura estejam provisionados, é possível provisionar as os microsserviços de negócio. Para tal, basta aplicar os SECRETS[12], CONFIGMAPS[13] e o DEPLOYMENT[14].

Há, porém, um detalhe: um dos SECRETS contém as credenciais do MinIO, geradas cada vez que o *tenant* é criado. Dito isso, é necessário resgatar as credenciais discutidas na Seção A.2, e editar o SECRET apropriado[15].

Note que as credenciais estão representadas em *base64*. Portanto, é necessário fazer a conversão:

```
$MINIO_ACCESS_KEY_ID=<valor salvo anteriormente>
$MINIO_ACCESS_KEY_ID_BASE_64=$(echo -n $MINIO_ACCESS_KEY_ID | base64)
$echo $MINIO_ACCESS_KEY_ID_BASE_64
UWdxekxDcERrZDFNMnRvbA==
...
$MINIO_SECRET_ACCESS_KEY=<valor salvo anteriormente>
$MINIO_SECRET_ACCESS_KEY_BASE_64=$(echo -n $MINIO_SECRET_ACCESS_KEY | base64)
$echo $MINIO_SECRET_ACCESS_KEY_BASE_64
TThqWGhrTFFoTFRrcnp6WVVQQjNPaG96a0lROWNQRko=
```

Após isso, edite o recurso que, dadas as credenciais apresentadas acima, deve ter o seguinte formato:

```
apiVersion: v1
kind: Secret
metadata:
  name: spira-minio-tenant-secret
type: Opaque
data:
  minio_access_key: UWdxekxDcERrZDFNMnRvbA==
  minio_secret_key: TThqWGhrTFFoTFRrcnp6WVVQQjNPaG96a0lROWNQRko=
  AWS_ACCESS_KEY_ID: UWdxekxDcERrZDFNMnRvbA==
  AWS_SECRET_ACCESS_KEY: TThqWGhrTFFoTFRrcnp6WVVQQjNPaG96a0lROWNQRko=
```

O Kubernetes permite a aplicação em massa de recursos. Isso pode ser usado para aplicar todos os arquivos que compõem o `SPIRA API Server`:

```
$ git clone https://github.com/vitorguidi/mac0499.git
$ cd mac0499/spira_k8s_files/spira
$ kubectl apply -f configmaps/ -n spira
$ kubectl apply -f secrets/ -n spira
$ kubectl apply -f api_server_deployment.yml
```

[12] SPIRA secrets
[13] SPIRA secrets
[14] SPIRA API Server deployment
[15] MinIO secret

Por fim, após aplicar todos os arquivos, os seguintes recursos devem estar presentes:

```
NAME                                              READY          AGE
pod/spira-api-server-5579695cdb-ck7hz             1/1            29m
pod/spira-api-server-5579695cdb-l4j45             1/1            10m
pod/spira-api-server-5579695cdb-sm5td             1/1            10m
NAME                                              TYPE           AGE
service/spira-api-server                          LoadBalancer   29m
NAME                                              READY          AGE
deployment.apps/spira-api-server                  3/3            29m
NAME                                              READY          AGE
replicaset.apps/spira-api-server-5579695cdb       3              29m
```

Uma vez aplicados os recursos, os PODS da aplicação executarão seus contêineres de inicialização, e a aplicação estará disponível.

A.8 SPIRA Inference Server

Uma vez que os SECRETS e CONFIGMAPS já foram aplicados na seção anterior, basta aplicar o DEPLOYMENT[16]. O processo de inicialização é idêntico ao `SPIRA API Server`.

```
$ git clone https://github.com/vitorguidi/mac0499.git
$ cd mac0499/spira_k8s_files/spira
$ kubectl apply -f inf_server_deployment.yml
```

Uma vez aplicados os arquivos, é esperado que os seguintes recursos estejam presentes no *cluster*:

```
NAME                                              READY          AGE
pod/spira-inf-server-674bbb47db-42qff             1/1            29m
pod/spira-inf-server-674bbb47db-bg8q2             1/1            10m
pod/spira-inf-server-674bbb47db-qtdzj             1/1            10m
NAME                                              TYPE           AGE
service/spira-inf-server                          LoadBalancer   29m
NAME                                              READY          AGE
deployment.apps/spira-inf-server                  3/3            29m
NAME                                              READY          AGE
replicaset.apps/spira-inf-server-674bbb47db       3              29m
```

Porém, há um detalhe: como o registro do modelo no MLFlow ocorre durante a inicialização dos PODS, não haverá nenhum modelo registrado com o rótulo de produção. Para tal, é necessário acessar a interface gráfica do MLFlow e registrar algum modelo de nome *pyfunc-test-model* como produção. Para acessar o MLFlow, siga as instruções descritas na Seção A.6.

[16] SPIRA Inference Server deployment

A.9 Prometheus e Grafana

Para provisionar o Prometheus e o Grafana, primeiro é necessário realizar uma intervenção manual no provisionamento do MinIO, de forma a habilitar a exposição de métricas. Por esse motivo, obtenha as credenciais discutidas na Seção A.2.

Inicialmente, instale a ferramenta *MinIO client*. Instruções para tal podem ser encontradas na documentação oficial[17].

Para provisionar a ferramenta, retome as credenciais do *MinIO*, como discutido em Seção A.2. De posse das credenciais, execute os seguintes comandos:

```
$ kubectl run -i --tty minio-debug --image=alpine
$ wget https://dl.min.io/client/mc/release/linux-amd64/mc
$ chmod +x mc
$ ./mc config host add spira http://minio:80 <minio access key salvo anteriormente>
<minio secret key salvo anteriormente> --insecure
$ mc admin prometheus generate spira
```

Isso deve gerar a seguinte saída, indicando que o *MinIO client* foi capaz de registrar o *tenant* de *MinIO* criado na pesquisa:

```
mc: Configuration written to `/root/.mc/config.json`. \
    Please update your access credentials.
mc: Successfully created `/root/.mc/share`.
mc: Initialized share uploads `/root/.mc/share/uploads.json` file.
mc: Initialized share downloads `/root/.mc/share/downloads.json` file.
Added `spira` successfully.
```

Em seguinda, execute outro comando:

```
$ ./mc admin prometheus generate spira
```

Uma vez executado, será gerado um arquivo para as configurações de *scrape* de Prometheus. Instruções para o provisionamento do Prometheus com a configuração alterada podem ser encontradas neste repositório[18].

```
scrape_configs:
- job_name: minio-job
  bearer_token: <bearer token gerado ao executar o comando mc admin generate>
  metrics_path: /minio/v2/metrics/cluster
  scheme: http
  static_configs:
  - targets: ['minio:80']
```

O próximo passo consiste em transformar o conteúdo da saída acima em *base64*, de forma a encaixá-lo em um SECRET que será utilizado posteriormente no provisionamento do Prometheus.

[17] Minio Client documentation
[18] Prometheus additional scrape configs

Para gerar o arquivo apropriado, como as credenciais do MinIO mudam a cada instalação de *tenant*, atualize o SECRET da seguinte forma, tomando o cuidado de remover a primeira linha no conteúdo da saída anteriormente apresentada:

```
$ cat << EOF >> minio_scraping_raw_data.yaml
- job_name: minio-job
  bearer_token: <bearer token gerado ao executar o comando mc admin generate>
  metrics_path: /minio/v2/metrics/cluster
  scheme: http
  static_configs:
  - targets: ['minio:80']
EOF
$ base64 < minio_scraping_raw_data.yaml > prometheus-additional.yaml
$ kubectl create secret generic additional-scrape-configs \
    --from-file=prometheus-additional.yaml --dry-run \
        -oyaml > additional-scrape-configs.yaml
```

Após executar os comandos acima, é esperado que o arquivo *additional-scrape-configs.yml* apresente conteúdo similar ao que pode ser encontrado no repositório da pesquisa[19]. Por fim, aplicamos o arquivo com o seguinte comando:

```
$ kubectl apply -f additional-scrape-configs.yaml -n spira
```

Uma que o SECRET acima esteja aplicado, crie o seguinte arquivo, que também está disponível no repositório da pesquisa[20]:

```
cat << EOF >> prometheus_scrape_config_override.yaml
prometheus:
 prometheusSpec:
   additionalScrapeConfigsSecret:
     enabled: true
     name: additional-scrape-configs
     key: prometheus-additional.yaml
EOF
```

Por fim, todos os requisitos para instalar o Prometheus e o Grafana estão satisfeitos. Basta então executar os seguintes comandos:

```
$ helm repo add prometheus-community \
    https://prometheus-community.github.io/helm-charts
$ helm repo update
$ helm install kube-prom prometheus-community/kube-prometheus-stack \
    --values prometheus_scrape_config_override.yaml -n spira
```

[19] SPIRA scraping secrets
[20] Prometheus scrape config overrides

O processo de provisionamento é um tanto lento. Uma vez que os PODS de Prometheus e Grafana estiverem disponíveis, execute os seguintes comandos para ter acesso à interface gráfica das ferramentas e verificar que tudo correu bem:

```
$ kubectl port-forward svc/kube-prom-kube-prometheus-prometheus 8080:9090
$ kubectl port-forward svc/kube-prom-grafana 8090:80
```

Após a execução dos comandos acima, é esperado que os seguintes recursos tenham sido criados:

```
NAME                                                              READY
pod/alertmanager-kube-prom-kube-prometheus-alertmanager-0         2/2
pod/kube-prom-grafana-6cd879bd6f-6dnnf                            3/3
pod/kube-prom-kube-prometheus-operator-6b9c7fc588-zvxqq           1/1
pod/kube-prom-kube-state-metrics-84c7f6688f-8vg2p                 1/1
pod/prometheus-kube-prom-kube-prometheus-prometheus-0             2/2
NAME                                                              TYPE
service/alertmanager-operated                                     ClusterIP
service/kube-prom-grafana                                         ClusterIP
service/kube-prom-kube-state-metrics                              ClusterIP
service/kube-prom-prometheus-node-exporter                        ClusterIP
NAME                                                              READY
daemonset.apps/kube-prom-prometheus-node-exporter                 4
NAME                                                              READY
deployment.apps/kube-prom-grafana                                 1/1
deployment.apps/kube-prom-kube-prometheus-operator                1/1
deployment.apps/kube-prom-kube-state-metrics                      1/1
NAME                                                              READY
replicaset.apps/kube-prom-grafana-6cd879bd6f                      1
replicaset.apps/kube-prom-kube-prometheus-operator-6b9c7fc588     1
replicaset.apps/kube-prom-kube-state-metrics-84c7f6688f           1
NAME                                                                   READY
statefulset.apps/alertmanager-kube-prom-kube-prometheus-alertmanager   1/1
statefulset.apps/prometheus-kube-prom-kube-prometheus-prometheus       1/1
```

Para habilitar a instrumentação do MongoDB, reaplique seu CRD, descomentando o recurso de SERVICEMONITOR[21].

Por fim, para acessar as interfaces gráficas das ferramentas, basta executar os seguintes comandos e realizar o acesso pelo navegador:

```
$ kubectl port-forward svc/kube-prometheus-stack-prometheus 8080:9090
$ kubectl port-forward svc/kube-prometheus-stack-grafana 8090:80
```

[21] Mongo CRD

A.10 Chaos Mesh

O provisionamento do Chaos Mesh se dá por meio da ferramenta Helm. Para tal, basta executar os seguintes comandos:

```
$ helm repo add chaos-mesh https://charts.chaos-mesh.org
$ helm install chaos-mesh chaos-mesh/chaos-mesh -n=spira --version 2.6.1
```

Após executar os comandos acima, é esperado que os seguintes recursos tenham sido criados:

```
NAME                                                READY   AGE
pod/chaos-controller-manager-66684d7685-7f5lg       1/1     5m16s
pod/chaos-controller-manager-66684d7685-h45j5       1/1     5m16s
pod/chaos-controller-manager-66684d7685-nslv6       1/1     5m16s
pod/chaos-daemon-ff8cs                              1/1     5m16s
pod/chaos-daemon-pgjt9                              1/1     5m16s
pod/chaos-daemon-tq9mx                              1/1     5m16s
pod/chaos-dashboard-8fc854f8d-2gpmn                 1/1     5m16s
pod/chaos-dns-server-76d86f84d7-84qmq               1/1     5m16s
NAME                                          TYPE          AGE
service/chaos-daemon                          ClusterIP     5m16s
service/chaos-dashboard                       NodePort      5m16s
service/chaos-mesh-controller-manager         ClusterIP     5m16s
service/chaos-mesh-dns-server                 ClusterIP     5m16s
NAME                                               READY   AGE
daemonset.apps/chaos-daemon                        3       5m16s
NAME                                               READY   AGE
deployment.apps/chaos-controller-manager           3/3     5m16s
deployment.apps/chaos-dashboard                    1/1     5m16s
deployment.apps/chaos-dns-server                   1/1     5m16s
NAME                                                     READY   AGE
replicaset.apps/chaos-controller-manager-66684d7685      3       5m16s
replicaset.apps/chaos-dashboard-8fc854f8d                1       5m16s
replicaset.apps/chaos-dns-server-76d86f84d7              1       5m16s
```

Para verificar que tudo correu bem, basta executar um *port-forward* e acessar a interface gráfica da ferramenta:

```
$ kubectl port-forward service/chaos-dashboard 8090:2333
```

Apêndice B

Executando os testes do **SPIRA** no Kubernetes

Este apêndice tem como objetivo registrar o processo de testes do SPIRA, de forma a permitir a reprodução do provisionamento. Além disso, busca apresentar evidência que, de fato, o sistema foi provisionado no Kubernetes, que o fluxo de inferência se mantém funcional, e que a aplicação ganhou alta disponibilidade.

Para executar os passos descritos aqui, considera-se que as instruções no Apêndice A já foram executadas. As seguintes seções descrevem o processo de reprodução dos testes de integração, ponta a ponta, e de caos.

Nos exemplos abaixo, comandos iniciados com `$` são executados por usuários regulares, comandos iniciados em `#` são executados por super-usuários, e comandos iniciados por `#` são executados em outro *terminal*.

B.1 Testes de Integração

O propósito dos testes de integração é verificar se as aplicações de negócio do SPIRA são capazes de se comunicar corretamente com os componentes de infraestrutura, e realizar as operações necessárias em cada um deles.

Para o `Spira API Server`, crie o POD com a imagem contendo os artefatos do teste de integração[1] executando os seguintes comandos:

```
$ git clone https://github.com/vitorguidi/mac0499.git
$ cd mac0499/spira_k8s_files/spira/debug
$ kubectl apply -f spira_api_test_pod.yml
```

Espera-se que o POD apropriado tenha sido criado:

```
NAME                    READY   STATUS    RESTARTS      AGE
spira-api-test-pod      1/1     Running   0             47m
```

Execute então os seguintes comandos:

[1] SPIRA API Server integration test image

```
$ kubectl exec -it spira-api-test-pod -- /bin/sh
$ export PYTHONPATH=$PYTHONPATH:.
$ pip install "pymongo[srv]"
$ py.test tests/integration_tests/
```

É esperado o resultado positivo dos testes, como mostrado abaixo:

```
============================= test session starts ==============================
platform linux -- Python 3.8.16, pytest-7.1.2, pluggy-1.0.0
rootdir: /app, configfile: pyproject.toml
plugins: asyncio-0.19.0
asyncio: mode=strict
tests/integration_tests/connections/database/test_1_mongo_conn.py .....
tests/integration_tests/connections/database/test_1_mongo_inserts.py ....
tests/integration_tests/connections/database/test_1_mongo_queries.py ....
tests/integration_tests/connections/database/test_2_adapter.py ..........
tests/integration_tests/connections/message_service/test_adapter.py ...
tests/integration_tests/connections/simple_storage/test_1_minio_bucket.py ..
tests/integration_tests/connections/simple_storage/test_2_minio_adapter.py ..
tests/integration_tests/endpoints/test_inference.py ............
tests/integration_tests/endpoints/test_model.py ........
tests/integration_tests/endpoints/test_result.py ...
[90%]
tests/integration_tests/endpoints/test_user.py ......
[100%]
============================== 61 passed in 10.66s ==============================
```

A execução dos testes de integração para o `SPIRA Inference Server` são similares. A única diferença é o emprego de uma imagem diferente[2]:

```
$ git clone https://github.com/vitorguidi/mac0499.git
$ cd mac0499/spira_k8s_files/spira/debug
$ kubectl apply -f spira_inference_test_pod.yml
```

Espera-se que o POD apropriado tenha sido criado:

```
NAME                          READY   STATUS    RESTARTS      AGE
spira-inference-test-pod      1/1     Running   0             47m
```

Execute então os seguintes comandos:

[2] Spira Inference Server integration tests image

```
$ kubectl exec -it spira-api-test-pod -- /bin/sh
$ export PYTHONPATH=$PYTHONPATH:.
$ pip install "pymongo[srv]"
$ py.test tests/integration_tests/
```

É esperado o resultado positivo dos testes, como mostrado abaixo:

```
============================================================================
test session starts ========================================================
platform linux -- Python 3.8.16, pytest-7.1.2, pluggy-1.0.0
rootdir: /app, configfile: pyproject.toml
plugins: asyncio-0.19.0
asyncio: mode=strict
collected 7 items

tests/integration_tests/adapters/message_service/test_adapter.py ...
tests/integration_tests/adapters/simple_storage/test_1_connection.py ..
tests/integration_tests/adapters/simple_storage/test_2_adapter.py ..
============================================================================
7 passed in 1.32s
============================================================================
```

A execução dos testes acima demonstra, portanto, que os componentes de aplicação conseguem se comunicar corretamente com os componentes de infraestrutura.

B.2 Testes Ponta a Ponta

O teste ponta a ponta consiste em apontar o *front end*[3] do SPIRA para o *back-end* provisionado em Kubernetes durante essa pesquisa.

Para a execução do teste, é necessário expor o `SPIRA API Server` publicamente para a Internet. A solução encontrada nesta pesquisa foi empregar o ngrok[4], cujas instruções de instalação se encontram no site da ferramenta.

Para expor a aplicação, há duas etapas: expor o SERVICE do *SPIRA API Server* localmente na máquina hospedeira do KinD, e em seguida deixar a porta em questão disponível publicamente na Internet.

Para a primeira etapa, basta executar os seguintes comandos:

```
$ kubectl port-forward service/spira-api-server 5050:80
$ ngrok http 5050
```

Isso deve gerar uma saída similar ao texto a seguir, contendo o endereço público da aplicação:

```
ngrok
```

[3] front-end
[4] ngrok

```
Build better APIs with ngrok. Early access: ngrok.com/early-access
Session Status                online
Account                       vitorguidi@gmail.com (Plan: Free)
Version                       3.3.1
Region                        South America (sa)
Latency                       -
Web Interface                 http://127.0.0.1:4040
Forwarding
https://b026-2804-214-85c4-c570-4ea8-ed6b-fa75-ed6e.ngrok-free.app
->
http://localhost:5050
Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Uma vez que a aplicação esteja exposta publicamente, é necessário apontar o *front-end* no Netlify[5] para o endereço gerado via *ngrok*. Para tal, altere as seguintes variáveis de ambiente para o endereço gerado pelo *ngrok*:

- VUE_APP_BACKEND_URL
- VUE_APP_INFERENCE_BACKEND_URL

Há, porem, um detalhe: o *ngrok* intercepta as requisições e gera um redirecionamento, deixando o usuário ciente de que a página é servida por meio dela. Isso quebra o fluxo de usuário, o que tornou necessário contornar o problema, empregando o navegador Firefox e a extensão *User Agent Switcher and Manager*[6].

A forma de contornar o problema foi usar a extensão em questão e injetar um *user-agent* que não referencie um navegador, como sugerido num tópico de suporte encontrado no GitHub[7]. Em particular, foi escolhido o *user-agent* do Instagram para Android, de forma arbitrária, que fez o fluxo voltar a funcionar.

Finalmente, a página da aplicação[8] pode ser acessada com as credenciais definidas no SECRET[9] de autenticação do `SPIRA API Server`. O usuário e senha são, respectivamente, *username* e *password*.

Feito tudo isso, é possível cadastrar uma inferência. Há uma página que permite verificar os resultados. Para o experimento em particular, é esperado que a inferência criada apresente o estado *completed*.

B.3 Testes de Caos

Os testes de caos consistem em aplicar CRDs da ferramenta Chaos Mesh, que desligarão, a cada minuto, um POD aleatório de cada um dos componentes do `SPIRA`.

Para tanto, execute os seguintes comandos:

[5] Spira Netlify Frontend
[6] User agent switcher and manager, firefox extension
[7] ngrok redirect user-agent issue
[8] SPIRA Frontend login page
[9] SPIRA API Server credentials

```
$ git clone https://github.com/vitorguidi/mac0499.git
$ cd mac0499/chaos/schedules
$ kubectl apply -f .
```

Após isso, os SCHEDULES do Chaos Mesh deverão ter sido criados. É esperado que os seguintes recursos estejam presentes:

```
kubectl get schedule
NAME                    AGE
kill-minio              8s
kill-mlflow             8s
kill-mongo              8s
kill-nats               8s
kill-pg                 8s
kill-spira-api-server   8s
kill-spira-inf-server   8s
```

Para verificar que, de fato, o experimento de caos funciona, é necessário garantir que os PODS de cada um dos componentes estão sendo recriados após serem encerrados. Para isso, compare a idade dos PODS em cada STATEFUL SET e DEPLOYMENT: espera-se observar diferença no campo *AGE* de, no mínimo, um minuto entre o mais recente a ser criado e os demais. Por exemplo, para o PostgreSQL:

```
NAME                                    STATUS              AGE
pod/spira-pg-cluster-0                  Running             82s
pod/spira-pg-cluster-1                  Running             2m21s
pod/spira-pg-cluster-2                  Running             21s
```

No exemplo acima, cada POD foi recriado com intervalo de um minuto, o que prova que de fato o experimetno de caos está sendo executado com sucesso.

Agora que os PODS estão sendo destruídos e recriados, é possível executar o experimento de fato: bombardear o `SPIRA API Server` com requisições de criação de inferência, anotar o identificador das inferências, e verificar se todas as inferências criadas foram executadas com sucesso ao final de um período arbitrário de tempo,.

Para facilitar o experimento, um POD, disponível no repositório da disciplina[10], deve ser criado dentro Kubernetes. A partir dele, devem ser executados dois *scripts*, como discutido em Seção 5.3: o primeiro[11] bombardeia a aplicação com requisições, escrevendo num arquivo de texto os identificadores das inferências cuja criação foi confirmada; e o segundo[12] consome o arquivo de texto, consulta o `SPIRA API Server`, e verifica se todas as requisições criadas foram processsadas com sucesso.

Para ter acesso a esse POD contendo os *scripts* dentro do *cluster*, execute os seguintes comandos:

[10] Python3 debug pod
[11] SPIRA API flood script
[12] SPIRA API chaos result checker

```
$ git clone https://github.com/vitorguidi/mac0499.git
$ cd mac0499/chaos/debug
$ kubectl apply -f debug_scripts_pod.yaml -n spira
$ kubectl exec -it python3-dbg-pod  -- /bin/bash
```

Em seguida, executamos o primeiro *script*, para criar as inferências e registrar seus identificadores. Os comandos abaixo serão executados dentro do POD de *debug*:

```
$ git clone https://github.com/vitorguidi/mac0499.git
$ cd mac0499/chaos/test_scripts
# apt install git
$ pip3 install requests
$ env SPIRA_INFERENCE_DUMP_FILE=inference_dump.txt python3 spira_api_flood.py
```

O primeiro *script* executa por um tempo arbitrário, devendo ser interrompido via SIGINT. Ao final da execução, execute o segundo *script* para verificar se todas as inferências criadas foram processadas com sucesso:

```
$ env CHAOS_LOG_FILE=inference_dump.txt python3 chaos_result_checker.py
```

Por fim, é esperado que todas as inferências criadas tenham sido processadas. O resultado esperado é similar ao seguinte:

```
{
    "failed_inference_acks": 0,
    "ack_and_completed_inferences": 21,
    "ack_and_not_completed_inferences": 0
}
```

Como não há inferências criadas e não processadas (ack_and_not_completed_inferences), o experimento foi um sucesso. No exemplo específico, não houve casos de inferências não processadas (failed_inference_acks), mas é aceitável que esse número não seja zero, dado que o contrato do SPIRA é que **todas as inferências criadas serão eventualmente processadas**.

Referências Bibliográficas

J. Garrison and K. Nova. **Cloud Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment**. O'Reilly Media, 2017. ISBN 9781491984277. URL https://books.google.com.br/books?id=zlk7DwAAQBAJ. 5, 9, 23

Robert C. Martin. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Robert C. Martin Series. Prentice Hall, Boston, MA, 2017. ISBN 978-0-13-449416-6. URL https://www.safaribooksonline.com/library/view/clean-architecture-a/9780134494272/. 3

C. Richardson. **Microservices Patterns: With examples in Java**. Manning, 2018. ISBN 9781617294549. URL https://books.google.com.br/books?id=UeK1swEACAAJ. 4, 22

Victor Tamae. Building an intelligent system to detect respiratory insufficiency, 2022. URL https://daitamae.github.io/MAC0499/Monograph.pdf. 1, 2, 3, 6, 11, 13, 14, 20, 22, 25