UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

CURSO DE BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

Gabriel Amancio da Silva

COMO GARANTIR QUE UM CLUSTER KUBERNETES POSSUI COBERTURA DE FALHAS CONTINUAMENTE NA CLOUD? O USO DE CHAOS ENGINEERING NA ESTEIRA DE ENTREGA CONTÍNUA

RECIFE

2022

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Gabriel Amancio da Silva

COMO GARANTIR QUE UM CLUSTER KUBERNETES POSSUI COBERTURA DE FALHAS CONTINUAMENTE NA CLOUD? O USO DE CHAOS ENGINEERING NA ESTEIRA DE ENTREGA CONTÍNUA

Monografia apresentada ao Centro de Informática (CIN) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para conclusão do Curso de Ciências da Computação, orientada pelo professor Vinicius Cardoso Garcia.

RECIFE
2022

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Silva, Gabriel Amancio da.
Como garantir que um cluster Kubernetes possui cobertura de falhas continuamente na Cloud? : O uso de Chaos Engineering na esteira de entrega contínua / Gabriel Amancio da Silva. - Recife, 2022.
46 : il., tab.

Orientador(a): Vinicius Cardoso Garcia
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2022.

1. DevOps. 2. Pipeline de delivery. 3. Chaos Engineering. I. Garcia, Vinicius Cardoso . (Orientação). II. Título.

000 CDD (22.ed.)

AGRADECIMENTOS

Agradeço a minha família, que foi minha base em toda a minha trajetória e principalmente a minha mãe, eu vi seus sacrifícios de perto para me fazer chegar até aqui e jamais esquecerei disso. Tudo que faço é para compensar tudo isso. Amo você mãe, amo vocês todos.

Também agradeço a minha companheira que não me permitiu desanimar e me deu forças quando eu não tinha. Te amo Mayhhara.

E por fim, agradeço aos meus amigos que tornaram toda essa caminhada mais leve, amo vocês!

"Só por que alguma coisa não faz o que você planejou que ela fizesse não quer dizer que ela seja inútil."
Thomas Edison

RESUMO

Como criar um software resiliente é uma pergunta chave no mundo de desenvolvimento atual. É cada vez mais comum casos de vazamento de dados, sistemas fora do ar por horas, erros e falhas inesperadas. Uma das possíveis soluções a serem exploradas na prevenção desses problemas é o Chaos Engineering, área que tem crescido nos últimos anos e que possui o claro objetivo de melhorar a qualidade e resiliência dos serviços ao qual é integrado. Com uma metodologia focada na criação de hipóteses e experimentos para validar as mesmas, ele funciona como uma ótima ferramenta de design preventivo de software. No presente trabalho, foram realizados experimentos fazendo uso do Litmus Chaos, plataforma que fornece experimentos voltados para o Chaos Engineering, com o objetivo de colocar à prova o ambiente no qual a aplicação teste estava sendo disponibilizada. Os resultados apresentados mostram que a abordagem de uso do Chaos Engineering na esteira de entrega contínua deve ser levada em conta.

Palavras-chave: DevOps, Chaos Engineering, Pipeline de delivery

ABSTRACT

How to build resilient software is a key question in today's development world. It is increasingly common cases of data leakage, systems down for hours, bugs and unexpected crashes. One of the possible solutions to be explored in the prevention of these problems is Chaos Engineering, an area that has grown in recent years and which has the clear objective of improving the quality and resilience of the services to which it is integrated. With a methodology focused on creating hypotheses and experiments to validate them, it works as a great preventive software design tool. In this work, experiments were made using Litmus Chaos, a platform that provides experiments aimed at Chaos Engineering, in order to test the environment in which the test application was available. The results presented show that the approach of using Chaos Engineering in the Entrega Contínua approach should be taken into consideration.

Keywords: DevOps, Chaos Engineering, Delivery Pipeline

Sumário

Lista de Figuras

LISTA DE TABELAS

TABELA DE SIGLAS

Sigla	Significado	Página
LIB	Library	30
API	Application Programming Interface	29,30
AWS	Amazon Web Services	29
TI	Tecnologia da Informação	21
CE	Chaos Engineering	28

1. Introdução

Atualmente, o ambiente cloud tem dominado o mercado de TI, causando um grande movimento de migração das aplicações para o mesmo. A cloud tem se tornado também a plataforma principal para novos serviços da era digital, sendo esperado que até 2023 40% de todos os workloads estejam sendo disponibilizados nela [1]. Entretanto, boa parte das aplicações on-premise não estão prontas para explorar os benefícios que a cloud promete oferecer como, por exemplo, escalabilidade e alta disponibilidade [25]. Por questões arquiteturais e de alinhamento estratégico com o negócio, realizar essa migração - ou evolução, adaptação - não é algo tão simples de ser feito. Uma das alternativas para essa adaptação é o uso de microsserviços [31], que facilita os ganhos com a cloud pelo fato de a escalabilidade estar diretamente ligada ao seu modelo arquitetural [25].

Entretanto, apenas fazer o uso de microsserviços não irá garantir com que a aplicação esteja extraindo a melhor performance possível na cloud, uma das maneiras de buscar garantir isso é fazendo o uso de virtualização [32], técnica que busca fazer a criação de uma imagem virtual ou "versão abstrata" de alguma aplicação, seja ela um servidor, sistema operacional e afins, para que este seja utilizado em diversas máquinas ao mesmo tempo, sendo o seu principal objetivo gerenciar a demanda de recursos para que a computação tradicional seja mais escalável [33]. Uma das principais maneiras de se atingir a virtualização é fazendo o uso de contêineres, que é uma técnica de virtualização em que a imagem faz uso direto do sistema operacional da máquina que está rodando a aplicação.

Com a aplicação funcionando em microsserviços e fazendo uso de contêineres surge um novo desafio, lidar com inúmeros contêineres que estão fazendo uso da infraestrutura cloud. Dada essa necessidade, por diversas vezes se fazem uso de orquestradores de contêineres, buscando gerenciar da melhor maneira os recursos demandados por esses microsserviços, porém, como dito anteriormente, essa adaptação arquitetural não é algo tão trivial, assim como as validações e garantias necessárias para um bom uso dos orquestradores apresenta diversos desafios [26]. É importante lembrar que, quando se fala de gerenciar recursos na cloud, cada recurso tem um custo e esse custo é esperado que aumente cada vez mais, dado que o investimento das empresas vem crescendo no setor, sendo esperado atingir um aumento de 28% gasto na cloud neste ano [2]. Sendo assim, um recurso mal gerenciado impacta diretamente em uma perda considerável de dinheiro.

Nos últimos anos, dentre os orquestradores de contêineres um se destacou e vem tomando grande parte do mercado, o Kubernetes [27]. Ele que, dentre as grandes empresas, possui uma margem de uso no ambiente de produção de 59% [28]. Mesmo sendo open-source, o custo gerado pelo uso do Kubernetes para uma empresa é considerável quando se trata da infraestrutura necessária para utilizá-lo [29]. Sendo assim, o impacto causado pela falta de validação em um processo no qual a aplicação faz uso de um cluster Kubernetes pode ser grande não apenas na experiência do usuário da aplicação, mas também no financeiro desta empresa. Dito isso, um questionamento válido é como validar a garantia neste processo? Essa pergunta guia esse projeto, que busca responder com o uso de Chaos Engineering [29] de maneira que seu uso automatizado, antes da disponibilização no ambiente de produção, possa evitar impactos negativos não previstos [18].

1.1. Motivação

Atualmente, resiliência tem sido a palavra chave quando se fala de aplicações na cloud. Não criar o projeto pensando em uma arquitetura pronta para lidar com o inesperado é sinônimo de impacto financeiro negativo por problemas nos serviços atingidos. E quando se fala em cloud, basta um provedor ter problemas e inúmeros serviços saem do ar, como ocorreu diversas vezes com a AWS em 2021 [3].

Uma das possíveis soluções a serem exploradas na prevenção desses problemas é o Chaos Engineering, área que tem crescido nos últimos anos e que possui o claro objetivo de melhorar a qualidade e resiliência dos serviços ao qual é integrado. Com uma metodologia focada na criação de hipóteses e experimentos para validar as mesmas, ele funciona como uma ótima ferramenta de design preventivo de software, onde agrega no controle de qualidade de toda uma infraestrutura a ser validado pelos seus experimentos, possibilitando a criação de planos de ação a partir das comprovações, ou não, de hipóteses que foram testadas.

Por exemplo, como posso garantir que o cluster está preparado para uma troca de pods por sobrecarga de acessos? Com essa pergunta criamos a seguinte hipótese: Dado que o número de acessos no serviço aumente exponencialmente, o balanceador de carga deverá redistribuir os acessos, fazendo com que o serviço não fique fora do ar. A partir disso, é criado

um experimento que força essa situação, e que de acordo com o seu resultado, caso a hipótese não seja confirmada mostra que o sistema está com potenciais problemas que devem ser resolvidos antes de ir para um ambiente de produção. Sendo assim, o problema a ser explorado neste trabalho será:

Como o uso de Chaos Engineering pode colaborar na garantia sob a cobertura de falhas em um cluster Kubernetes ?

1.2. Objetivos

Sendo assim, podemos definir os seguintes objetivos:

- Definir indicadores relevantes para medir eficiência do uso do Chaos Engineering na esteira de entrega contínua
- Avaliar utilização do Chaos Engineering em projeto piloto com indicadores definidos

1.3. Estrutura

O projeto está construído essencialmente nos próximos 5 capítulos, sendo eles:

- Fundamentação teórica: Possui o objetivo de construir a base de conhecimento necessária para se compreender o projeto e suas referências, trazendo os principais tópicos que guiam o trabalho e destrinchando-os.
- Implementação: Busca apresentar a construção do projeto, o que foi utilizado e o porquê, possibilitando ao leitor compreender as decisões tomadas acerca do trabalho.
- Experimentos: Capítulo que foca no entendimento da criação e realização dos experimentos, mostrando, com a ajuda da metodologia GQM [34,35], o que foi definido, o porquê, o que foi coletado e entendendo quais indícios os resultados apresentados nos fornece.
- Conclusão: Seção que busca sintetizar tudo o que foi construído e fornecer uma análise macro do que pode ser extraído do resultado dos experimentos, além de mencionar as problemáticas reconhecidas no trabalho em os possíveis trabalhos futuros.
- Referências: Seção final que apresenta todas as referências utilizadas para embasar o projeto teoricamente.

2. Fundamentação Teórica

Neste capítulo serão apresentados e discutidos os conceitos teóricos que embasam o presente trabalho.

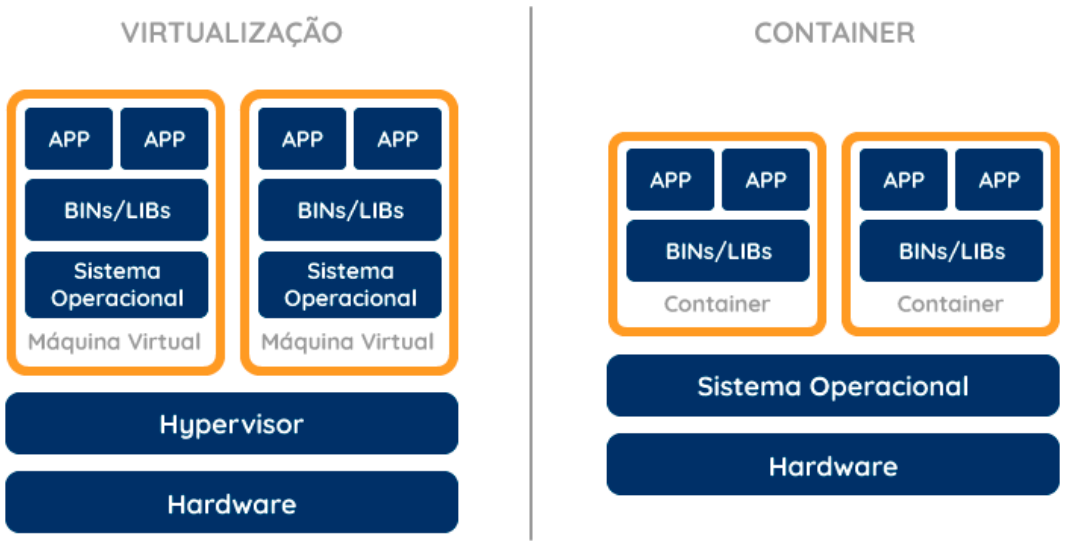
2.1. Virtualização

Virtualização é uma prática altamente utilizada pelas empresas, e que ganhou ainda mais força com o crescimento da computação em nuvem. Ela consiste em promover a criação de uma imagem virtual de uma aplicação, fazendo com que essa imagem possa ser usada em diferentes máquinas e ambientes com a mesma consistência. Garantindo assim que o software tenha a garantia de se comportar igual independentemente de onde seja disponibilizado [32, 42].

Dentro da virtualização, temos diferentes maneiras para a criação e gerenciamento dessas imagens criadas. Uma delas é o uso de um Maquinas virtuais, que é uma representação de um computador real, porém em software. Com isso, é possível simular um novo dispositivo na sua própria máquina com o seu próprio sistema operacional, mas que não irá se comunicar com o hardware. Para existir a comunicação com o hardware, existe uma camada chamada Hypervisor, que é responsável por fazer o gerenciamento de recursos como podemos ver na Figura 1 [42].

Uma outra maneira de realizar a virtualização é por meio de containers. Diferente da máquina virtual que roda o seu próprio sistema operacional, o container simula um sistema operacional e faz uso de tudo que for possível da máquina hospedeira, tendo um melhor aproveitamento de recursos e tornando todo o processo mais leve, como está sendo mostrado na Figura 1 [42].

Figura 1 [43] compara a virtualização com Máquina virtual e Container

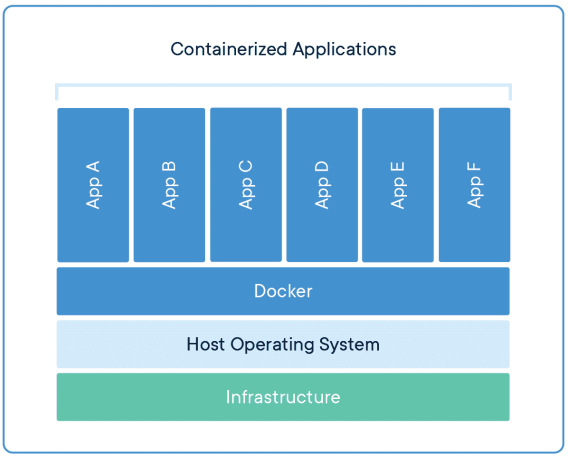


Fonte: [43]

2.1.1. Docker

Ao falar sobre Container, é quase que inevitável falar sobre Docker [43], ferramenta voltada para o desenvolvimento, implantação e disponibilização remota altamente usada em ambientes cloud [42]. Docker permite a criação de Docker images, que são as instâncias do container o qual o Docker estará rodando. Essas imagens possuem o código das aplicações, o runtime binário do próprio docker e os arquivos de configuração. A partir dessas imagens, o container é criado e implantado dentro do Docker, que funciona sob a camada do sistema operacional da máquina do hospedeiro, como é mostrado na Figura 2 [43, 42].

Figura 2 mostra a estrutura de funcionamento de containers com Docker



Fonte: [43, 42]

2.1.2. Kubernetes

No ambiente de produção, é comum se ter diversos containers funcionando, inclusive para uma mesma aplicação, tornando o gerenciamento complexo mesmo com a utilização do Docker. Além do gerenciamento se tornar complicado, existe também o fator escalabilidade que é buscado mas não é fácil de ser atingido apenas com o uso simples de containers. Para isso, são utilizadas ferramentas de orquestração de containers, sendo a mais popular chamada de Kubernetes [27, 28].

Kubernetes é uma aplicação open source, criada com o objetivo de promover o gerenciamento de aplicações conteinerizadas, além de lidar também com os serviços ao redor dos containers, como a própria comunicação entre eles [45].

Dentro da estrutura do Kubernetes existem uma vasta quantidade de componentes, o componente que será mencionado e focado nesse projeto é chamado de Pod. Pod é uma unidade que possui a capacidade de agrupar contêineres, sendo este a unidade mais atômica do Kubernetes. O Pod é responsável por compartilhar recursos entre os contêineres, como memṕoria, rede e informações sobre a execução dos contêineres contidos nele [27].

O design do Kubernetes permite a automação da implantação das aplicações, além de configurações de escalabilidade nativos, fazendo com que possa ser configurado de acordo com o acesso dos usuários, por exemplo, evitando que com o aumento do acesso à sua aplicação, ela não lide com a demanda. Com essa vasta parametrização, Kubernetes torna mais prático o processo de gerenciar o ambiente de produção, favorecendo assim a qualidade de a garantia dos serviços conteinerizados que estão sendo disponibilizados [45].

2.2. DevOps

Quando falamos sobre DevOps, podemos afirmar que se trata de um conjunto de práticas destinadas a reduzir o período entre uma alteração em um sistema e a mudança a ser colocada em produção, garantindo alta qualidade [9].

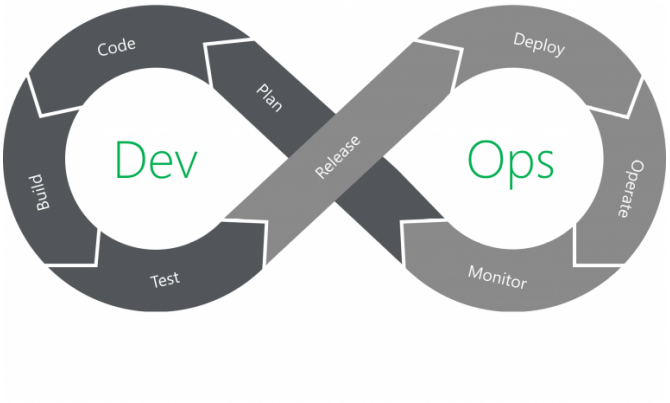
O surgimento do DevOps veio em conjunto com as metodologias ágeis, como o Scrum [6], do qual se utilizou bastante como base na sua criação. O DevOps vem pregar a

colaboração entre times, além da ênfase em automatização e do uso de ferramentas que auxiliam a comunicação dinâmica entre times da área de operações e da área de desenvolvimento.

Humble e Molesky [10], buscaram definir 4 princípios para o DevOps:

- Automação: DevOps se baseia em uma automação completa de build, deploy e testes buscando alcançar tempos curtos em suas entregas, provendo assim uma atualização constante e expressa que também possibilita o feedback constante dos usuários finais.
- Cultura: Para que o DevOps possa ser realmente implementado, é necessária uma mudança cultural para que a corresponsabilidade seja absorvida pelo time para que isso possibilite uma real entrega de software com alta qualidade. Com isso, a responsabilidade para que tudo funcione é de todos, todos são responsáveis pelo código escrito.
- Métricas: Com a entrega constante, o acompanhamento contínuo também vem em conjunto, e isso possibilita a criação de métricas e objetivos, visando a melhoria contínua do software baseado em dados que ele mesmo fornece e que, graças ao monitoramento contínuo, pode ser coletado e analisado para melhorias assertivas.
- Compartilhamento: O compartilhamento deve ser algo geral e em todos os níveis. Desde compartilhar conhecimento, sobre features por exemplo, ou até mesmo compartilhar ferramentas e descobertas em potencial. Isso potencializa a integração da equipe, fortifica a cultura e incentiva a colaboração entre as áreas.

A Figura 3 [22] representa um pipeline padrão de DevOps.



Fonte: [22]

Na fig. 3 é mostrado e ciclo infinito utilizado no devops, este ciclo é composto por 8 etapas, sendo elas:

- Plan: Fase de planejamento de um projeto

- Code: Fase de codificação do projeto
- Built: Fase de construção da versão codificada do projeto
- Test: Fase de teste e validação do projeto
- Release: Fase de controle para o lançamento do projeto em diversos ambientes
- Deploy: Fase de implantação do projeto no ambiente principal
- Operate: Fase de entrega do projeto implantado ao cliente
- Monitor: Fase de monitoramento e observação do projeto, para análise de erros e percepção de potenciais melhorias, para assim voltar ao ponto de planejamento

Compreendendo este ciclo que resume o que prega a cultura DevOps, podemos associar DevOps em uma área de pesquisa conhecida como Engenharia de Software Contínua, Área esta que foca no desenvolvimento, implantação e obtenção de feedback de maneira rápida e eficiente do software do cliente [11, 12].

2.2.1. Integração Contínua

Integração Contínua consiste em uma prática de desenvolvimento bem estabelecida dentro da indústria de desenvolvimento de software, onde os membros de uma equipe realizam as integrações e junções constantes de todo trabalho desenvolvido, sendo isso feito até mais de uma vez em um único dia. Essa prática permite que as empresas tenham releases bem mais frequentes e curtos, melhorando assim a produtividade e também a qualidade do que foi produzido.

Fitzgerald e Stol definem as atividades de Integração Contínua da seguinte maneira:

> Um processo normalmente disparado automaticamente que inclui etapas conectadas entre si, como compilar código, executar testes de unidade e aceitação, validar a cobertura de código, verificar a conformidade com o padrão de código e criar pacotes de implantação. Embora alguma forma de automação seja comum, a frequência também é relevante, pois deve ser regular o suficiente para garantir um feedback rápido aos desenvolvedores. Finalmente, qualquer falha de integração contínua também é um evento importante que pode ter uma série de cerimônias e artefatos altamente visíveis para ajudar a garantir que os problemas que levam a falhas de integração sejam resolvidos o mais rápido possível pelos responsáveis. [12]

Além disso, essa etapa agrega também o processo de criação da nova versão do projeto e teste das aplicações de forma automatizada, tornando essa prática ainda mais completa. [13].

2.2.2. Entrega Contínua

Entrega Contínua foca na garantia de que uma aplicação está sempre preparada para ir ao ambiente de produção após passar por diversas checagens automáticas de qualidade e testes.

A Entrega Contínua implementa um conjunto de práticas, por exemplo, a Integração Contínua, citado anteriormente, e a implementação de automação para entregar software automaticamente para um ambiente de produção. Essa prática promove uma série de vantagens, tais qual diminuição do risco de um deploy, redução de custos e obtenção de feedback dos usuários com maior frequência e rapidez [14].

2.2.3. Implantação Contínua

Implantação Contínua é uma prática que foca em garantir que cada pequena alteração passe pela pipeline definida e termine no ambiente de produção de maneira automatizada. Quando se fala de Entrega Contínua e Implantação Contínua, no meio acadêmico não se tem um consenso entre suas diferenças e definições [12], porém, o uso do Implantação Contínua implica diretamente que o Entrega Contínua também está sendo usado, mas o caminho contrário não é verdade [15].

A Implantação Contínua e o Entrega Contínua possuem uma diferença principal e reconhecida, que é o momento da entrega do software. Enquanto na Entrega Contínua a entrega é constante, fazendo com que o software sempre esteja em um estado de entrega depois de qualquer mudança ter sido feita, entretanto, a entrega final não é automatizada, é uma entrega manual como uma decisão com viés de negócio. Porém, na Implantação Contínua esse processo é automatizado.

2.3. SRE: Site Reliability Engineering

Segundo Benjamin Treynor Sloss [16], SRE é o que acontece quando você pede para um Engenheiro de Software elaborar o design de um time de operações. Com o crescimento da cultura DevOps, não só a área de Dev precisou se readaptar, mas também a parte de Ops. Anteriormente, com as divisões das áreas, a comunicação entre um Desenvolvedor e um SysAdmin era rara, além de complicada. Com o incentivo para que as áreas cada vez mais se aproximassem, foi natural que surgisse uma área de interseção entre Dev e Ops, alguém que entendesse ambos os lados para que promovesse uma maior colaboração.

Tendo isso em mente, o Google iniciou a criação de seus times de SRE com o pensamento mencionado no início, causando uma mudança significativa nas estruturas comuns usadas anteriormente. Com isso, ao invés de um time inteiro de SysAdmins, os times de SRE começaram a ser formados também por Engenheiros de Software.

2.3.1. SRE vs DevOps

Quando falamos de SRE e DevOps, algumas confusões são criadas, principalmente por conta do mercado de trabalho que comumente intitula vagas para a mesma função com ambos os nomes. Entretanto, existem diferenças significativas em suas definições.

Quando falamos de DevOps, estamos falando sobre uma cultura, cultura essa proveniente do Scrum, visando uma mudança na formação de equipes e estrutura interna de uma empresa de Software. Enquanto SRE é focado no desenvolvimento de práticas métricas para melhorar e implementar a colaboração incentivada pelo DevOps [17].

Em resumo, SRE vem para garantir e promover a implementação da cultura do DevOps

2.4. Chaos Engineering

Chaos Engineering é uma estratégia emergente na indústria, com o objetivo de avaliar a resiliência de um sistema distribuído rodando experimentos nesse sistema, enquanto ele está em produção. Esses experimentos podem identificar fraquezas que podem levar a quebra do sistema caso seja deixada de lado [18].

Essa prática gira em torno de criar hipóteses e rodar experimentos ao seu redor, buscando confirmá-la ou recusá-la. Essa hipótese se refere ao comportamento esperado do sistema, em um cenário específico. Em um experimento, se injeta certa sobrecarga ou falhas, que são passíveis de acontecer em situações reais, para se observar o comportamento do sistema. Essa observação irá trazer uma nova perspectiva, além de insights sobre o estado do sistema dentro da situação injetada, podendo assim aferir se a hipótese foi aprovada ou não.

O argumento utilizado para validar o uso de CE é que para viver em um mundo não confiável, em que cada vez mais a complexidade dos sistemas cresce e está passível de acontecer qualquer falha seja no próprio sistema ou na infraestrutura que ele faz uso, seria fazer com que as falhas aconteçam de maneira controla e provocada intencionalmente, para assim se ter resultados dessas falhas e parâmetros para se analisar e auxiliar na busca por soluções que previnam esse tipo de acontecimento [41].

Uma grande entusiasta, e também precursora, da prática é a empresa Netflix, responsável por criar algumas ferramentas open source conhecidas como *Simian Army* [18, 41], voltadas para a injeção de falhas nos ambientes da AWS. A netflix apresentou a primeira ferramenta, chamada de *Chaos Monkey*, em 2010 com o objetivo de auxiliar na migração de seus serviços de uma infraestrutura física para o ambiente cloud [42]. Com o decorrer do tempo, o time de engenharia da Netflix expandiu ainda mais essas ferramentas lançando também o *Chaos Gorilla* e o *Chaos Kong,* ambas ferramentas para injeção de falhas de diferentes maneiras na AWS, e essas duas ferramentas se juntaram ao *Chaos Monkey* formando o *Simian Army* [41].

2.5. Four Golden Signals

Quando se fala em DevOps e SRE, é inevitável falar de monitoramento. Para avaliar a eficiência deste projeto, um dos objetivos foi a definição dos indicadores relevantes para medir a eficiência do uso do CE, que podemos considerar a etapa de monitoramento deste trabalho.

O time de SRE da Google define 4 métricas essenciais para um monitoramento decente [19], e serão estas métricas usadas como indicadores para o projeto. São elas:

- Latência: O tempo que se leva para realizar uma request
- Tráfico: Acompanhamento da demanda que o projeto está tendo, e o quanto a aplicação consegue lidar com o seu aumento

- Erros: A taxa de requests que falham
- Saturação: Medida que define o quão disponível está a máquina que hospeda o projeto

Esses são os *4 Golden Signals*, e eles serão usados como indicadores da eficiência do CE neste projeto. O uso dos Golden Signals proporciona um norte para as necessidades de monitoramento de qualquer projeto, de modo que o uso deles auxilia não só na metrificação do software enquanto monitorado, mas também na construção do monitoramento ao seu redor, tal como definições de ferramentas, métodos de coletas dos dados e afins [40].

2.6. Sumário do capítulo

Neste capítulo foram apresentados os principais conceitos que entornam este projeto. Tendo início pela virtualização, tecnologia que dá a base para a estrutura do ambiente em nuvem, mencionando Docker e Kubernetes, que são as ferramentas principais neste meio [42]. Em seguida, foi discutido o tópico de DevOps, práticas essenciais para se manter a qualidade nos ambientes de produção, trabalhando em conjunto com SRE. Além da apresentação dos *4 golden signals*, que servirá de base para as métricas do projeto, guiando assim a coleta de dados.

No Capítulo em seguida será abordado a implementação do projeto, fazendo uso de tecnologias, ferramentas e metodologias que possuem base nas apresentadas nesta fundamentação.

3. Implementação

Neste capítulo estará sendo explorado tudo que envolveu a implementação do projeto, passando por tecnologias, ferramentas, decisões de arquitetura e escolhas pessoais.

3.1. Tecnologias adotadas

O projeto possui 3 componentes básicos, sendo eles: Api, Cloud e Pipeline.

Diferentes tecnologias foram utilizadas em cada um desses componentes, com a intenção do melhor aproveitamento, além da praticidade para o desenvolvimento em si de cada uma dessas etapas.

Ao falar sobre API, será abordado todo o entorno do componente, partindo de linguagem, framework e arquitetura. Além disso, também será mostrado como a aplicação foi implementada dentro de um cluster Kubernetes, e o design final da aplicação com os recursos integrados entre si.

Ao falar sobre Cloud, será possível ver o design da solução kubernetes integrada com os recursos criados no ambiente Cloud escolhido, além de explorar como foi feita a integração e os motivos pelos quais cada componente foi escolhido.

Por fim, a Pipeline será abordada em 3 tópicos diferentes, sendo dois deles ferramentas que foram usadas na pipeline e por último sua estrutura final, dessa maneira, será demonstrado a motivação do uso de cada ferramenta e o conjunto final que se fez do resultado da integração dessas ferramentas.

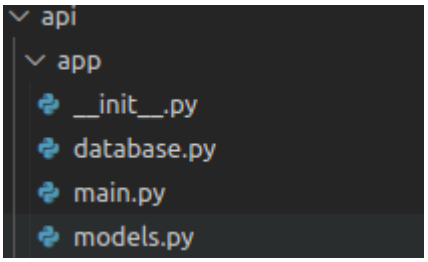
3.1.1. FastAPI

A API foi desenvolvida em FastAPI [8]. Tecnologia já mencionada anteriormente pela sua praticidade no uso e curva de desenvolvimento rápida. Para o projeto, o foco foi em desenvolver poucas rotas e com pouco processamento, como o objetivo não é testar a API em si, e sim a resiliência do ambiente em que ela está disponibilizada, não se fazia necessário criar algo complexo nesse componente.

Para este projeto, foi desenvolvida uma API que faz listas *to do*, que funciona como uma lista de tarefas a qual você marca quando tiver finalizado determinada atividade. A estrutura da api possui os seguintes componentes:

- Main: Arquivo responsável pela criação das rotas e controle central da api.
- Database: Arquivo responsável pela conexão com o banco de dados.
- Model: Arquivo responsável pela criação da estrutura de dados do *to do.*
- init: Arquivo python necessário para a linguagem reconhecer como um projeto.

Figura 4 mostra a estrutura final dos arquivo os quais compuseram a api

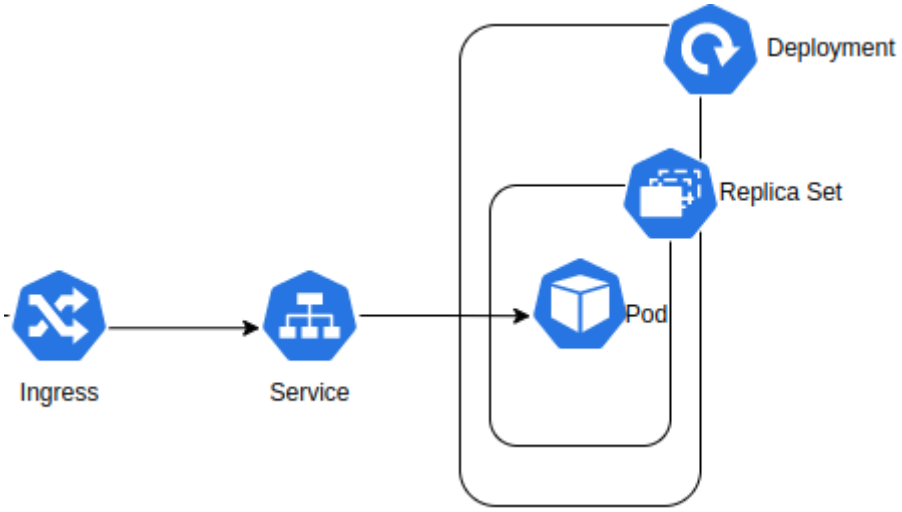


Fonte: imagem do autor

Como o objetivo deste trabalho é fazer os testes em um ambiente Kubernetes, a API deveria estar conteinerizada para ser implantada no cluster. Dado este motivo, foi utilizado Docker para fazer a geração da imagem. Esta imagem gerada permite que a usemos no container de um pod dentro do cluster kubernetes no qual a aplicação irá ser disponibilizada.

Para a aplicação desenvolvida, a arquitetura Kubernetes montada teve o objetivo de ser simplória, focando na demonstração do potencial do uso do CE em uma pipeline para validar a arquitetura e não na complexidade e exploração do Kubernetes. A Figura 5 ilustra a arquitetura criada para a aplicação.

Figura 5 ilustra arquitetura kubernetes deste projeto



Fonte: imagem do autor

Na Figura 5 conseguimos enxergar todos os componentes Kubernetes usados neste projeto e como interagem entre si. Primeiro temos o Deployment, componente responsável pelo gerenciamento de um grupo de contêineres e sua rede. No Deployment existe um processo chamado Replica Set, este processo se mantém observando todo e qualquer Pod contido nele, para que em caso de erro ou queda do pod, ele consiga recriar uma cópia o quanto antes, para evitar com que sua queda tenha impacto no acesso a aplicação [47]. Nesta aplicação foi definido que o número de pods que deveriam estar no ar era de apenas 1.

Na criação de um Pod só é possível acessá-lo via o IP criado para ele dentro do cluster Kubernetes no qual ele foi disponibilizado. Para conseguir ter o acesso externo, foram utilizados dois componentes: Service e Ingress. Como os Pods possuem um ciclo de vida efêmeros, ou seja, eles vêm e vão constantemente, se faz necessário alguma maneira que garanta acesso a aplicação disponibilizada pelos pods mesmo com suas idas e vindas [47]. O IP criado para o pod não é fixo dado este ciclo de vida, com isso, o Service é a solução para que se crie um acesso fixo aos pods, mesmo com as constantes mudanças.

Neste projeto, o objetivo era a análise a partir do uso externo ao cluster, dado que a intenção é avaliar o usuário final usando a aplicação e a sua experiência dentro dos possíveis problemas simulados com o CE. Por este motivo, a configuração utilizada no Service foi a de Load Balancer, opção voltada para prover o acesso externo do cluster ao pod. Já o Ingress, tal qual o deployment, é um mecanismo de gerenciamento e agrupamento de services, que visa facilitar a manipulação desses recursos [47]. O seu uso teve o objetivo de simplificar a

associação entre o Service e um DNS (Domain Name System) atribuído a aplicação, para facilitar os testes.

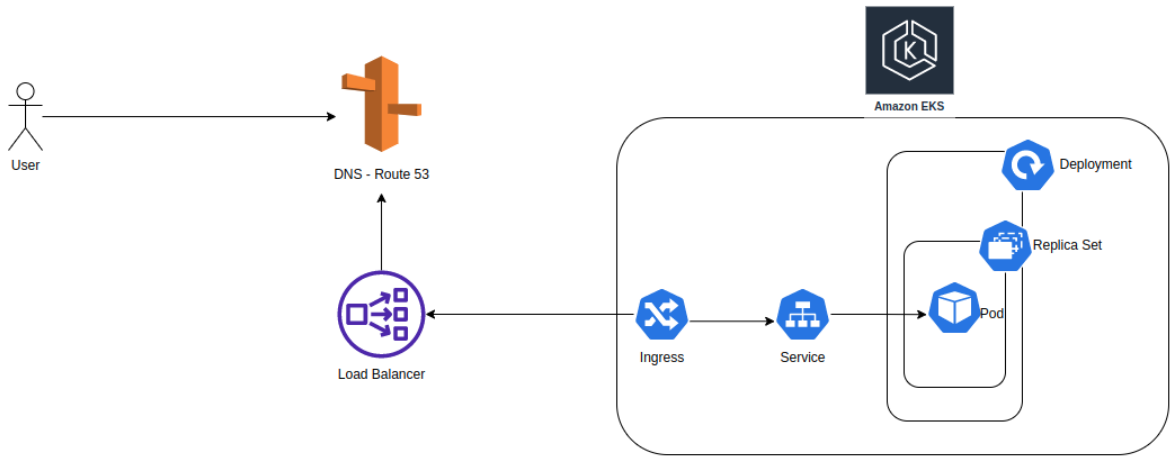
Com essa arquitetura se faz possível uma fácil integração com os recursos usados pelo Cloud Provider usado no projeto que será abordado no tópico seguinte.

3.1.2. AWS

A AWS é o Cloud Provider mais utilizado do mercado [20], isso foi o fator principal para sua escolha para esse projeto.

Foi utilizado o Terraform[21], que é uma ferramenta de IAC, para criar e gerenciar toda infraestrutura necessária para o projeto na AWS, otimizando o desenvolvimento e permitindo um maior controle de todos os recursos envolvidos. Sendo esses recursos respectivamente: Cluster EKS, Load Balancer e DNS no Route53.

Figura 6 representa a relação dos recursos da AWS com os do Kubernetes



Fonte: imagem do autor

Na Figura 6 conseguimos observar a estrutura AWS que funciona ao redor dos componentes Kubernetes criados para a aplicação. Todos os componentes estão implantados dentro do cluster EKS, serviço provido pela AWS que busca facilitar a execução do Kubernetes, por ser gerenciado pela própria AWS, sem necessidade do usuário fazer isto.

Como mencionado no tópico anterior, o Service da aplicação foi configurado para uso do Load Balancer, pois o objetivo era o acesso externo do usuário final. Sendo assim, foi

criado um recurso de Load Balancer que foi conectado com o Ingress da Aplicação. Neste Load Balancer foi associado uma regra de roteamento, para que haja a associação entre o DNS registrado no Route53 e a url disponibilizada pelo Load Balancer para acesso externo. Como foi dito anteriormente, o Ingress foi usado para facilitar essa associação do DNS com o Service, permitindo assim que após a configuração, o usuário consiga atingir o pod por meio deste DNS.

O uso do cluster EKS permitiu que houvesse uma integração de maneira simples com a ferramenta definida para a criação de pipelines, uma vez que a configuração desta integração é feita apenas pela configuração de uma chave de segurança.

3.1.3. Github Action

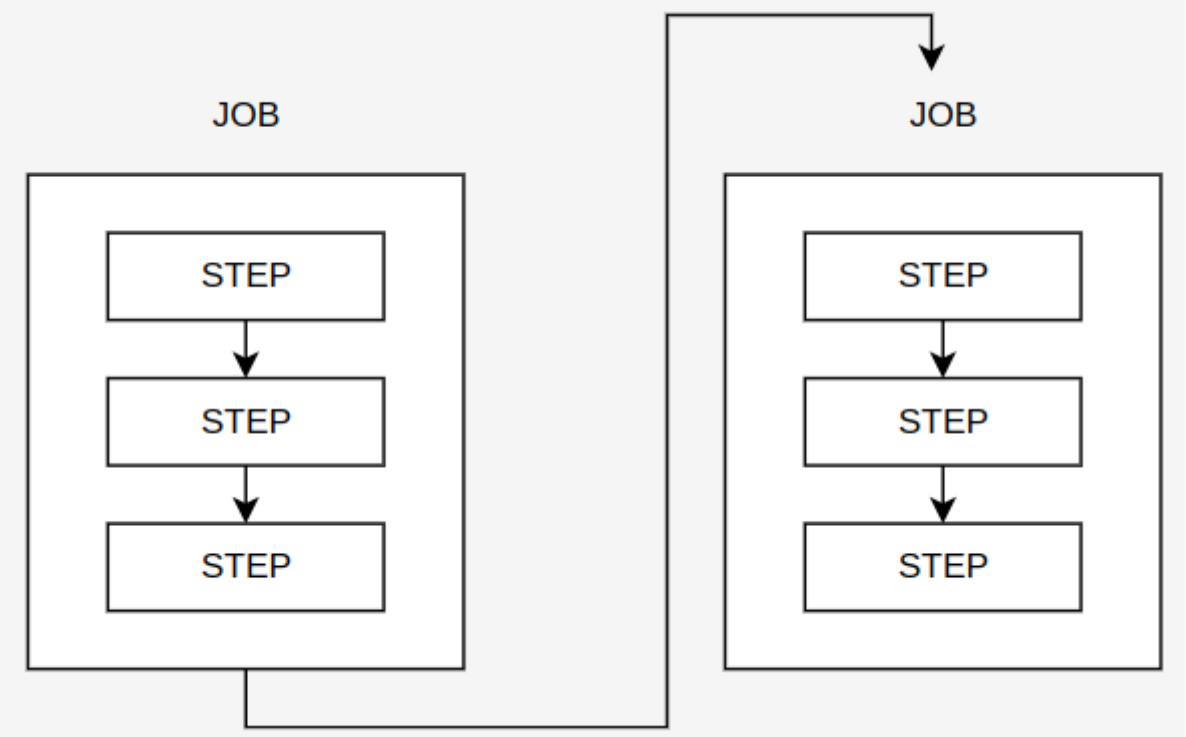
Github Action é uma ferramenta integrada ao Github [37], que permite fazer o uso de CI/CD em seus próprios repositórios [38]. Quando se trata de CI/CD e gerir pipelines, diversas ferramentas existem para este objetivo, uma amplamente usada é o Jenkins [39], ferramenta open source de grande popularidade no segmento. A opção de uso do Github Action se fez pela sua praticidade, dado que o projeto já estava no ambiente do repositório do Github, a configuração de uma outra ferramenta necessitaria de mais tempo.

Para compreender o funcionamento do Github Action é necessário entender sua estrutura de funcionamento que possui 2 principais estruturas:

- Steps: Steps são um conjunto de atividades com o propósito de executar alguma função, sendo ela a execução de um script ou a configuração de algum ambiente, por exemplo.
- Jobs: Jobs são um conjunto de Steps que deverão ser executados em paralelo ou em uma determinada sequência, como por exemplo ter um job que faz a configuração de um ambiente e após o término iniciar um job que realiza testes com o ambiente configurado pelo job anterior.

Na Figura 7 é possível observar uma representação da estrutura de steps e jobs em um CI/CD do Github Actions.

Figura 7 representa os componentes de um CI/CD no Github Action



Fonte: imagem do autor

O github actions provê inúmeros steps pré-prontos, criados pela comunidade que faz uso da ferramenta. Dentro do projeto foram utilizados alguns desses steps, para otimizar o processo de configuração, dado que, em uma pipeline é criado uma *vm* a qual irá executar todos os comandos definidos. Como será necessário fazer uso de *python, docker* e *kubernetes* neste projeto, é necessário configurar cada uma dessas ferramentas e serviços. Foram utilizados os seguintes steps pré-prontos:

- *actions/checkout@v3:* Step responsável por garantir que a *vm* está usando o projeto a partir da raiz da pasta no qual ele está contido.
- *kodermax/kubectl-aws-eks@master*: Step responsável por realizar a instalação e configuração do kubectl, ferramenta que permite fazer o uso de comandos Kubernetes em um cluster como o EKS.
- *docker-practice/actions-setup-docker@master*: Step responsável por realizar a instalação e configuração do docker na *vm.*
- *actions/setup-python@v2*: Step responsável por realizar a instalação e configuração do python na vm.

Além desses steps, existem outros que foram criados porém todos foram feitos a partir da necessidade do projeto. Cada um dos Jobs será detalhado posteriormente.

3.1.4. Litmus Chaos

O Litmus Chaos é uma plataforma Open Source de Chaos Engineering no ambiente Cloud. Eles proveem um hub de experimentos que facilita a implementação e incentiva a comunidade DevOps e SRE a compartilharem os seus, promovendo organicamente um maior número de usuários desta prática.

Além do Litmus Chaos, outras plataformas existem para o mesmo objetivo, como o Chaos Monkey e Chaos Kong da Netflix [18], entretanto, para utilizar essas alternativas uma configuração maior é necessária de ser realizada no ambiente, pois elas funcionam com outras ferramentas e não sozinhas. Este foi o motivo principal pelo qual foi escolhido o Litmus Chaos, por ser uma ferramenta de fácil configuração permite que maior tempo possa ser dedicado à análise e realização dos experimentos, ao invés da ambientação.

No hub, existem diversos experimentos que podem ser aplicados diretamente no cluster Kubernetes, para o projeto serão utilizados 2:

- pod-network-latency: Este experimento consiste em interromper a conectividade de rede dos pods do Kubernetes. O experimento pode injetar atrasos de rede aleatórios nos pods de réplica do aplicativo. Esse indo de encontro a métrica de latência,
- pod-cpu-hog: Esse experimento consiste em interromper o estado dos recursos do Kubernetes. O experimento pode injetar um pico de CPU em um nó em que o pod do aplicativo está agendado. Esse indo de encontro a métrica de Saturação.

Assim como algumas ferramentas mencionadas anteriormente, o Litmus Chaos também possui steps pré-prontos no github actions para uso direto [48], porém, no momento que este projeto foi desenvolvido eles apresentavam certa instabilidade, o que dificultou o uso e configuração. Com isso, foi decidido fazer um uso direto utilizando os arquivos *yaml*, que são usados para recursos kubernetes, configurando individualmente cada um desses experimentos e aplicando no cluster na etapa que fosse necessária. Dessa maneira, a ferramenta teve o comportamento estável e sem necessidade de configurações extras para contornar problemas.

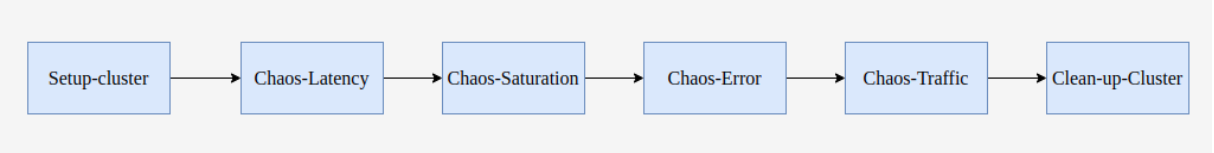
3.1.5. Estrutura da pipeline

Ao criar uma pipeline deve-se pensar em uma estrutura sequencial, onde cada uma de suas operações automatizadas possa ser considerada uma parte de entrega do software e também uma parte que garante a sua qualidade [36].

Cada pipeline deve se adaptar ao projeto no qual irá atuar, sendo cada passo configurado para isto. No projeto em questão, temos quatro necessidades básicas: Configuração do ambiente, realização dos experimentos, coleta de dados gerados nos experimentos e limpeza do ambiente após a realização dos experimentos.

Tendo isso definido, a pipeline para o projeto se mostrou eficiente tendo 6 passos. Sendo desses passos 1 responsável pela configuração do ambiente, 4 pelos experimentos e coleta de dados, o último responsável pela limpeza do ambiente após os experimentos. Na Figura 6 podemos observar a estrutura da pipeline, com cada um dos passos na sequência determinada.

Figura 8 representa a estrutura da pipeline final criada para o projeto



Fonte: imagem do autor

Ao observar cada um dos passos separadamente, as seguintes tarefas são realizadas::

- Setup-cluster: Etapa para realizar a instalação do Litmus Chaos no cluster e dos experimentos que serão realizados.
- Chaos-Latency: O primeiro experimento realizado é o Chaos injetado na latência. É utilizado o experimento do Litmus Chaos para causar uma variação brusca da latência no pod da aplicação. No momento que esse experimento se inicia, se dispara a coleta da latência, utilizando a ferramenta wrk[24], ferramenta utilizada para fazer benchmark HTTP, com ela é coletado a latência média da aplicação no período em que o experimento está ativo, após isso esse resultado é adicionado na planilha para melhor visualização. Após subir os dados para a planilha, o pod é resetado para ficar pronto para o próximo experimento.

- Chaos-Saturation: Segundo experimento realizado é o Chaos injetado no consumo de recursos da CPU do pod da aplicação. No momento que esse experimento se inicia, se dispara a coleta do tempo de resposta das requisições usando um script próprio em python e logo após o fim adicionando os dados na planilha. Após subir os dados para a planilha, o pod é resetado para ficar pronto para o próximo experimento.
- Chaos-Error: Terceiro experimento realizado é um teste de estresse em um só endpoint. Esse teste de estresse foi feito utilizando um script em python fazendo uso da lib *locust*, ferramenta que é voltada para load testing em APIs. Ao fim do experimento o resultado vai para planilha. Após subir os dados para a planilha, o pod é resetado para ficar pronto para o próximo experimento.
- Chaos-Traffic: Quarto e último experimento realizado é semelhante ao teste de estresse anterior, porém, ao invés do foco ser em apenas um endpoint, ele é feito de diversos endpoints ao mesmo tempo, também fazendo uso de um script python que faz uso do *locust*. Ao fim do experimento os resultados vão para planilha.
- Clean-up-Cluster: Ao fim dos experimentos, os recursos envolvidos são deletados.

3.2. Sumário do capítulo

Este capítulo teve como objetivo mostrar como foi pensado e executado a implementação do projeto desenvolvido neste trabalho e como as ferramentas foram integradas entre si.

Foi mostrado como a API desenvolvida foi disponibilizada em um cluster EKS, e como foi montada a arquitetura dos recursos do próprio kubernetes e também do Cloud Provider usado, a AWS. Além disso, também foi abordado as ferramentas usadas para a pipeline e parte dos experimentos do CE, sendo essas o Litmus Chaos e o Github Actions respectivamente. Foi apresentado cada um dos passos definidos na pipeline e o motivo para a existência de cada um.

Dito isso, no próximo capítulo será abordada a fase de experimentos que foram aplicados diretamente nos recursos apresentados neste capítulo. Será apresentado a construção de cada experimento e os dados coletados em sua execução. Além disso, será apresentada uma análise referente aos dados coletados em cada um dos experimentos.

4. Experimentos e Análise

Nesta seção será apresentado individualmente cada experimento e os dados coletados para fins de análise. Para cada experimento será apresentado a coleta de dados sem o experimento do Chaos estar em andamento e com o experimento em andamento, para haver uma comparação.

4.1. Experimentos

Para criação dos experimentos foi utilizada a metodologia GQM [34, 35], que traz uma estrutura top-down para proporcionar a criação de métricas direcionadas a um objetivo. Em cada um dos experimentos será apresentado a estrutura GQM que guiou sua construção.

4.1.1. CHAOS-LATENCY

Seguindo os *Golden Signals*, o primeiro a ser analisado foi a latência. Quando se fala de resiliência de aplicações de software de maneira direta se fala sobre a capacidade de um software falhar e, ainda assim, seguir funcionando [46]. Seguindo essa premissa, qual o objetivo de se analisar latência de uma aplicação no contexto de resiliência? A latência é responsável pelo tempo de processamento da informação que está sendo recebida pela aplicação. Sendo assim, uma latência estável é um indicativo de que a aplicação é capaz de responder ao usuário, mesmo em casos de erro, de maneira eficiente e não afetar a experiência do usuário. Dito isso, o objetivo deste experimento foi definido como: Garantir o tempo de resposta estável para o usuário final.

Para analisar o comportamento da aplicação dado um pico de latência é necessário provocar um problema na rede a qual a aplicação faz uso. Neste caso, foi usado o Litmus Chaos, mencionado no capítulo anterior, para ajudar na injeção da falha no cluster. O Litmus Chaos fornece alguns experimentos relacionados à rede, um deles é chamado pod-network-latency, já mencionado anteriormente. Com ele se faz possível inserir atrasos e interrupções na rede utilizada pelos pods.

Dado que já se tem o objetivo, o que deve ser provocado e como provocar esse acontecimento, para finalizar é necessário coletar o tempo de resposta da aplicação enquanto a

rede dos pods está sofrendo a injeção de falhas, assim será possível analisar se mesmo com as falhas a aplicação mantém o comportamento esperado, não tendo picos no tempo de resposta. Para isso foi utilizado a ferramenta wrk2, já mencionada no capítulo anterior. Com ela foi possível fazer a coleta do tempo de resposta no decorrer de 5 minutos para poder avaliar se houve, ou não, diferença no comportamento da aplicação.

Sendo assim, ao montar o GQM para este experimento, obtemos a seguinte definição:

- Goal: Garantir um tempo de resposta estável para o usuário final.
- Question: Em um aumento temporário da latência nos pods, o usuário irá ter um acréscimo do tempo de resposta?
- Metric: Tempo médio de resposta

Após essa definição, tudo foi executado e os seguintes dados foram gerados para análise:

Sem Chaos	0,051s	0,050s	0,051s	0,050s	0,052s
Com Chaos	1,05s	1,14s	1,14s	1,14s	1,14s

Cada coluna se refere a uma coleta diferente, cada uma dessas informações corresponde a latência média em um período de 1 minuto. É possível ver o aumento da latência gritante no momento em que o experimento está sendo executado.

4.1.2. CHAOS-SATURATION

Em seguida vamos à saturação. A saturação pode ser interpretada como diferentes métricas dentro dos *Golden Signals*, mas geralmente estão relacionadas aos recursos da máquina a qual está rodando a aplicação. Neste caso foi decidido usar a CPU como alvo dessa métrica, dado que a CPU tem relação direta com o processamento das informações em qualquer aplicação. Dado que ao disponibilizar uma aplicação em um cluster kubernetes um dos parâmetros passados para a definição de recursos é CPU, é interessante avaliar se o recurso alocado foi suficiente para a aplicação e sua demanda, sendo assim, o objetivo que se tem com isso é: Garantir que uma crescente no uso de memória não torne a aplicação lenta.

Para analisar como a aplicação responderá em um pico de consumo da CPU é necessário provocar esse pico. Para isso, novamente se fez uso do Litmus Chaos, que já fora

mencionado, para injetar o consumo exacerbado da CPU. Um dos experimentos fornecidos pelo Litmus Chaos é o pod-cpu-hog, que nos permite fazer com que a cpu da aplicação definida tenha um pico no seu consumo em determinado pod.

Após a definição do objetivo e da provocação necessária para gerar a falha, é necessário definir a métrica e coletar este dado. A CPU está ligada diretamente com o processamento de informações realizado pela aplicação. Quando se fala de processamento de informações, existe o tempo que a aplicação leva para entender o que está sendo requisitado para ela, como mencionado no tópico anterior esse tempo é a latência. Sendo assim, nesse experimento, será coletado o tempo médio de latência da aplicação no decorrer de 5 minutos de injeção de falha, pois o tempo de latência nos indica como a aplicação está se comportando no processamento das informações recebidas, em caso de subida no momento de um maior consumo do cpu pode ser um indício de que o CPU definido para aplicação não é suficiente e não é escalável.

Dito isso, ao montar o GQM para este experimento, obtemos a seguinte definição:

- Goal: Garantir que o pico de uso de memória não torne a aplicação lenta
- Question: Dado um pico de consumo de memória em parte dos pods, a latência da aplicação irá subir?
- Metric: Tempo médio de latência

Após essa definição, tudo foi executado e os seguintes dados foram gerados para análise:

Sem Chaos	0,486829262	0,40380679	0,397924899	0,395656465
Com Chaos	24,30273714	24,59412916	44,81050761	51,06785581

Cada linha se refere a uma coleta sequencial, ao final de uma requisição a outra logo foi executada e assim por diante, coletando o tempo de resposta de cada uma delas. Fica claro o aumento significativo do tempo de resposta da aplicação durante o experimento.

4.1.3. CHAOS-ERROR

O terceiro ponto analisado foi o sinal referente ao erro. A taxa de erros de uma aplicação web deve ser analisada e capturada de diferentes formas, indo de analisar o código

HTTP até checar o conteúdo da resposta da requisição [19]. Para analisar esse sinal neste trabalho foi decidido observar o código HTTP das requisições e a consistência dessas respostas, dado que com um alto índice de requisições idênticas para um endpoint, a resposta deve se manter constante preservando o comportamento da aplicação. Sabe-se que é improvável que as respostas sempre serão sucesso, tanto é que o Chaos Engineering se baseia neste fato, pois a falha é algo que não deve ser evitado mas sim ser preparado para lidar com ela [29]. Tendo isso em mente, o objetivo definido foi: Garantir confiabilidade do sistema mantendo uma taxa de erros http abaixo de 20% do total das requisições. Em diversos sistemas e projetos, a taxa de sucesso que se busca é por volta de 99%, esse valor busca ser definido com o usuário do sistema em questão, dado que esse valor é o indicador que o cliente julga aceitável perante o uso da aplicação. Como este projeto está usando uma aplicação simples, foi decidido deixar a taxa de sucesso mais baixa, levando em conta que o resultado ainda assim nos mostrará um indício do que se está comprovando ou não.

Para avaliar a porcentagem de erros da aplicação dentro do número de requisições é preciso criar antes o mecanismo que irá promover esses potenciais erros. Para isso foi utilizado a lib locust, ferramenta apresentada em um capítulo anterior, que nos auxilia na criação de scripts para obter um pico de requisições em endpoints definidos. Com isso, se faz possível avaliar a consistência das respostas da aplicação em um alto índice de requisições.

Sendo assim, foi realizada a coleta desses dados a partir de uma planilha, gerada pela própria lib locust, que permitiu fazer a análise dos dados coletados, dando foco na porcentagem dos erros no decorrer de 5 minutos com um número de requisições crescente.

Dito isso, ao montar o GQM para este experimento, se obtém a seguinte definição:

- Goal: Garantir confiabilidade do sistema tenha uma taxa de erros http abaixo de 20% do total das requisições
- Question: Em um pico de requisições, a aplicação continuará a responder com sucesso a maioria dessas?
- Metric: Porcentagem de erros http

Após essa definição, tudo foi executado e os seguintes dados foram gerados para análise:

Requests/s	Falhas/s	Total de Requests	Total de falhas
57.900.000	6.200.000	12755	2717
108.900.000	29.200.000	12900	2825
118.700.000	44.300.000	12919	2825
106.700.000	45.300.000	12960	2825

Cada linha se refere a um certo número de requisições realizadas para o mesmo endpoint, que foi crescendo de maneira sequencial controlado pelo script. É possível ver que ao chegar em um número alto de requisições por segundo, a aplicação começa a responder com erros uma certa porcentagem das requisições, ficando na média de 20% das respostas sendo erro, o que é um número muito significativo e no limite do objetivo definido.

4.1.4. CHAOS-TRAFFIC

Por último foi analisado o sinal de tráfego. O tráfego é um sinal que muda de acordo com o tipo de aplicação, dado que cada software tem sua demanda específica. No caso deste projeto, a aplicação é web, então a demanda é referente a requisições HTTP por segundo. A aplicação deve ser capaz de lidar com uma grande quantidade de requisições por segundo sem que isso impacte em sua performance. Neste caso, a percepção de mudança na performance por parte do usuário seria o tempo de resposta, dado que quanto mais lenta a aplicação, mais demorada a resposta chega no usuário. Dito isso, o objetivo definido para este experimento foi: Garantir que o aumento no número de requisições por segundo não impactem no tempo de resposta.

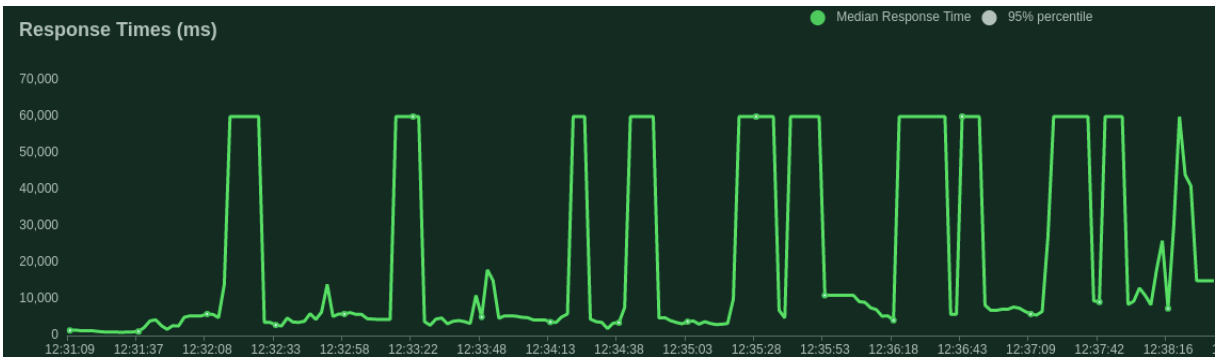
Para promover um pico de requisições foi feito o uso da lib locust, já mencionada anteriormente. Nela foi montado um script apontando para diferentes endpoints na aplicação em loops, para manter um alto índice de requisições por segundo. A locust já fornece uma interface que cria gráficos voltados para o tempo de resposta ao longo do teste, facilitando assim a análise dos dados coletados.

Dito isso, ao montar o GQM para este experimento, obtemos a seguinte definição:

- Goal: Garantir que o aumento no número de requisições não impacte no tempo de resposta
- Question: Em um pico de requisições, o usuário irá ter um acréscimo do tempo de resposta?
- Metric: Tempo médio de resposta

Após essa definição, tudo foi executado e os seguintes dados foram gerados para análise:

Figura 9 apresenta o gráfico do tempo de resposta do experimento



Fonte: imagem do autor

Na Figura 6 podemos observar a variação do tempo de resposta ao longo de 7 minutos. Nesse período de tempo o número de requisições por segundo e de usuários simulados subiram de maneira automatizada, como definido no script, atingindo um pico de 1000 usuários e 150 mil requisições por segundo. É claramente perceptível como o acréscimo no decorrer do tempo fez com que a média do tempo de resposta obtivesse uma variação constantemente, atingindo picos de 60000ms.

4.2. Análise

Baseado nos dados coletados em cada um dos experimentos conseguimos enxergar como essa aplicação de teste não estaria pronta para um ambiente de produção. Cada um dos experimentos do Chaos conseguiu mostrar com diferentes dados como essa aplicação, mesmo estando em um ambiente propício para escalabilidade, que é o Kubernetes, não está projetada para ser escalável.

Quando falamos de resiliência, nos referimos sempre a uma aplicação que consegue se manter disponível mesmo em ambientes conturbados, entretanto, para isso poder ser garantido é necessário realizar as devidas checagens antes de um possível deploy em produção. Com esta análise é possível perceber a relevância que o uso do Chaos Engineering pode proporcionar quando utilizado não de maneira isolada, mas sim como parte do processo incluído no próprio pipeline de testes da aplicação, para uma checagem e garantia mais robusta de qualidade.

5. Conclusões e Trabalhos Futuros

5.1. Contribuições

Ao fim deste trabalho é possível concluir que o uso do Chaos Engineering em uma pipeline de entrega contínua pode contribuir para a garantia de qualidade de aplicações que fazem uso do Kubernetes e de infraestruturas no ambiente da cloud. Foi possível enxergar as falhas da aplicação simulando cenários caóticos, mas possíveis, buscando validar a resiliência da aplicação antes de ir para um ambiente de produção.

Avaliando os 4 *Golden Signals* que a Google nos dá como pontos chave para um monitoramento, foi possível coletar informações de diferentes tipos que se complementam ao final dos experimentos, trazendo uma análise mais completa da aplicação em si. Com isso, podemos dizer que o uso de CE enriquece a esteira de entrega contínua.

5.2. Problemáticas

No decorrer do desenvolvimento do projeto, uma das principais complicações que foi enfrentado foi o uso de diferentes ferramentas para cada um dos cenários de Chaos. Esse acontecimento aumentou a complexidade da implementação além de dificultar a análise dos dados coletados, levando em conta que não vinham de uma fonte única e com a mesma estrutura.

Além disso, por ser uma aplicação muito simplória a qual está sendo utilizada para avaliar a infraestrutura na qual foi implementada, ela não foi projetada para fazer o melhor uso dos recursos do Kubernetes como sua premissa. Esse fato pode fazer com que alguns dos resultados causados pelos experimentos sejam desproporcionais, ainda que evidenciados, de um caso mais real.

5.3. Trabalhos futuros

Para trabalhos futuros, é interessante fazer a avaliação de uma aplicação mais robusta que faça uso de mais recursos que o Kubernetes fornece, e com isso fazer experimentos mais complexos validando outros pontos de avaliação e não apenas os 4 *Golden Signals*.

6. Bibliografia

[1] “Gartner predicts the futures of Cloud Computing and Edge Infrastructure”. Gartner. Disponível em: https://www.gartner.com/smarterwithgartner/gartner-predicts-the-future-of-cloud-and-edge-infrastructure. Acesso em 10 de fevereiro de 2022

[2] “Cloud Shift Impacts All IT Markets”. Gartner. Disponível em: https://www.gartner.com/smarterwithgartner/cloud-shift-impacts-all-it-markets. Acesso em 10 de fevereiro de 2022

[3] “Falha na nuvem gera instabilidade em Amazon, iFood e Disney+”. Folha de São Paulo. Disponível em: https://www1.folha.uol.com.br/tec/2021/12/falha-na-nuvem-gera-instabilidade-em-amazon-ifood-e-disney.shtml. Acesso em 14 de fevereiro de 2022

[4] “Top computer languages". Disponível em: https://statisticstimes.com/tech/top-computer-languages.php. Acesso em 15 de junho de 2022

[5] “Django makes it easier to build better web apps more quickly and with less code”. Disponível em: https://www.djangoproject.com. Acesso em 15 de junho de 2022.

[6] “Web development drop at a time”. Disponível em: https://flask.palletsprojects.com/en/2.2.x/. Acesso em 15 de junho de 2022.

[7] “O que é um microframework?”. Disponível em: https://www.treinaweb.com.br/blog/o-que-e-um-micro-framework. Acesso em 15 de junho de 2022.

[8] “FastAPI framework, high performance, easy to learn, fast to code, ready for production”. Disponível em: https://fastapi.tiangolo.com/. Acesso em 15 de junho de 2022.

[9] Len Bass, Ingo Weber, e Liming Zhu. 2015. DevOps: A Software Architect’s Perspective. Addison-Wesley, New York.

[10] Humble, J. , Molesky, J. , 2011. Why enterprises must adopt devops to enable Continuous Delivery. Cutter IT J. 24 (8), 6–12 .

[11] Jan Bosch. 2014. Continuous Software Engineering: An Introduction. Springer International Publishing, Cham, 3–13.

[12] Brian Fitzgerald e Klaas-Jan Stol. 2017. Continuous software engineering: A roadmap and agenda. Journal of Systems and Software 123 (jan 2017), 176–189.

[13] M. Leppänen et al., ‘‘The highways and country roads to Continuous Deployment,’’ IEEE Softw., vol. 32, no. 2, pp. 64–72, Mar. 2015.

[14] Lianping Chen. 2015. Continuous Delivery: Huge Benefits, but Challenges Too. IEEE Software 32, 2 (mar 2015), 50–54.

[15] J. Humble, e D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, 1st ed. Reading, MA, USA: Addison-Wesley, 2010.

[16] “Google’s Approach to Service Management: Site Reliability Engineering”. Disponível em: https://sre.google/sre-book/introduction/. Acesso em 12 de junho de 2022.

[17] “SRE vs DevOps: What 's The Difference?”. Disponível em: https://www.bmc.com/blogs/sre-vs-devops. Acesso em 14 de junho de 2022.]

[18] Basiri, A., Hochstein, L., Jones, N., Tucker, H. ‘‘Automating chaos experiments in production,’’ 2019, arXiv:1905.04648. [Online]. Disponível em: http://arxiv.org/abs/1905.04648. Acesso em 10 de junho de 2022.

[19] “Monitoring distributed systems”. Disponível em: https://sre.google/sre-book/monitoring-distributed-systems/. Acesso em 15 de junho de 2022.

[20] “Cloud infrastructure services vendor market share worldwide from 4th quarter 2017 to 1st quarter 2022”. Disponível em: https://www.statista.com/statistics/967365/worldwide-cloud-infrastructure-services-market-share-vendor/. Acesso em 26 de agosto de 2022.

[21] “Automate Infrastructure on Any Cloud”. Disponível em: https://terraform.io. Acesso em 10 de junho de 2022.

[22] “Criação de pipelines”. Disponível em: https://www.logicus.com.br/criacao-de-pipelines/. Acesso em 12 de junho de 2022.

[23] M. Shahin et al., “Continuous Integration, Delivery and Deployment”IEEE Softw., vol 5, pp. 3909-3943. Mar. 2017.

[24] “Modern HTTP benchmark tool”. Disponível em: https://github.com/wg/wrk. Acesso em 20 de agosto de 2022.

[25] Megargel, A., Shankararaman, V., Walker, D.K. (2020). “Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example. In: Ramachandran, M., Mahmood, Z. (eds) Software Engineering in the Era of Cloud Computing. Computer Communications and Networks. Springer”

[26] Tosatto, A., Ruiu, P., & Attanasio, A. (2015). Container-Based Orchestration in Cloud: State of the Art and Challenges. 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems.

[27] 2021. Kubernetes: Production-Grade Container Orchestration. Disponível em: https://kubernetes.io/. Acesso em 9 de setembro de 2022.

[28] “The true cost of Kubernetes: People, Time and Productivity”. Disponível em: https://www.koyeb.com/blog/the-true-cost-of-kubernetes-people-time-and-productivity. Acesso em 9 de setembro de 2022.

[29] “Principles of Chaos Engineering”. Disponível em: https://principlesofchaos.org/.Acesso em 9 de setembro de 2022.

[30] “Why Large Organizations Trust Kubernetes”. Disponível em: https://tanzu.vmware.com/content/blog/why-large-organizations-trust-kubernetes. Acesso em 9 de setembro de 2022.

[31] YaleYu, H. Silveira e M.Sundaram,“A microservice based reference architecture model in the context of enterprise architecture,” 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC).

[32] H. Khazaei, C. Barna, N. Beigi-Mohammadi e M. Litoiu, “Efficiency Analysis of Provisioning Microservices,” 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Luxembourg City, 2016, pp. 261-268.

[33] Malhotra L., Agarwal D and Jaiswal A., “Virtualization in Cloud Computing”, Journal of Information Technology and Software Engineering, 2014, 4:2.

[34] Basili, V. e Rombach, D. 1988. “The TAME project: Towards improvement-oriented software environments.” , IEEE Trans. Softw. Eng. 14, 6, 758 –773.

[35] Basili, V. 1994. “GQM approach has evolved to include models.”, IEEE Softw. 11, 1, 8.

[36] Rafal Leszko. 2017. Continuous delivery with Docker and Jenkins : delivering software at scale. Packt Publishing, Birmingham, UK.

[37] “Github: Where the world builds software”, Disponível em: https://github.com/, Acesso em 14 de setembro de 2022.

[38] A. Decan, T. Mens, P. R. Mazra e M. Golzadeh, “On the Use of GitHub Actions in Software Development Repositories”, 38th IEEE International Conference on Software Maintenance and Evolution (ICSME), Limassol, Cyprus, 2022

[39] “Build great things at any scale”, Disponível em: https://www.jenkins.io/, Acesso em 14 de setembro de 2022.

[40] “Golden Signals: Why are they so important to DevOps/SRE teams”, Disponível em: https://iamondemand.com/blog/the-golden-signals-why-theyre-so-important-to-devops-sre-teams/ , Acesso em 15 de setembro de 2022.

[41] “Chaos Engineering: the history, principles and practice”, Disponível em: https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-and-practice, Acesso em 15 de setembro de 2022.

[42] Agarwal A., Rao S. K. S. e Mahendra B. M, 2020, “Comprehensive review of virtualization tools”, International Research Journal of Engineering and Technology (IRJET), 7, 6.

[43] “Virtualização x Container”, Disponível em: https://www.funcao.com.br/2019/01/11/virtualizacao-x-container/, Acesso em 17 de setembro de 2022.

[44] “Develop Faster. Run Anywhere”, Disponível em: https://www.docker.com/, Acesso em 17 de setembro de 2022.

[45] Dewi, L. P., Noertjahyana, A., Palit, H. N. e Yedutun, K, 2019, “Server Scalability Using Kubernetes”, 4th Technology Innovation Management and Engineering Science International Conference (TIMES-iCON). doi:10.1109/times-icon47539.2019.9024501

[46] “Implementar aplicativos resilientes”, Disponível em: https://learn.microsoft.com/pt-br/dotnet/architecture/microservices/implement-resilient-applications/, Acesso em 22 de setembro de 2022.

[47] “Tutoriais | Kubernetes”, Disponível em: https://kubernetes.io/pt-br/docs/tutorials/, Acesso em 26 de setembro de 2022.

[48] “Github actions to trigger chaos on your review apps”, Disponível em: https://github.com/litmuschaos/github-chaos-actions, Acesso em 26 de setembro de 2022.