UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROJETO DE GRADUAÇÃO

DAYANE SILVA ERLACHER CASTRO

# ESCALONADORES AUTOMÁTICOS EM PODS COMO FERRAMENTA DE MITIGAÇÃO DE ATAQUES DDOS DESTINADOS A MICROSSERVIÇOS

VITÓRIA-ES

AGOSTO/2022

Dayane Silva Erlacher Castro

# ESCALONADORES AUTOMÁTICOS EM PODS COMO FERRAMENTA DE MITIGAÇÃO DE ATAQUES DDOS DESTINADOS A MICROSSERVIÇOS

Monografia do Projeto de Graduação da aluna Dayane Silva Erlacher Castro, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheira Eletricista.

Vitória-ES

Agosto/2022

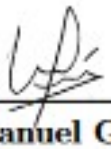
Dayane Silva Erlacher Castro

# ESCALONADORES AUTOMÁTICOS EM PODS COMO FERRAMENTA DE MITIGAÇÃO DE ATAQUES DDOS DESTINADOS A MICROSSERVIÇOS

Monografia do Projeto de Graduação da aluna Dayane Silva Erlacher Castro, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheira Eletricista.

Aprovado em 15 de agosto de 2022.

**COMISSÃO EXAMINADORA:**



**Msc. Víctor Manuel García Martínez**
Universidade Federal do Espírito Santo
Orientador

**Prof. Dr. Moisés Renato Nunes Ribeiro**
Universidade Federal do Espírito Santo
Coorientador

**Prof. Dr. Rodolfo da Silva Villaça**
Universidade Federal do Espírito Santo
Examinador

**Prof. Dr. João Henrique Gonçalves Medeiros Corrêa**
Universidade Federal do Ceará
Examinador

Vitória-ES

Agosto/2022

# AGRADECIMENTOS

Agradeço a Deus por me abençoar todos os dias da minha vida. Agradeço aos meus pais por sempre me incentivarem e fornecerem condições para que eu alcançasse meus objetivos. Agradeço a minha irmã por sempre me apoiar e me auxiliar em todos os meus desafios. Agradeço ao meu namorado pelo apoio e compressão nos momentos que me ausentei para que eu pudesse focar nos estudos e desenvolvimento de meus projetos.

Agradeço aos meus familiares por me apoiarem, torcerem e acreditarem em mim. Agradeço a 2Solve e aos meus amigos que trabalham comigo por todo o apoio. Agradeço aos meus amigos pelos tantos momentos compartilhados, bons e ruins, durante toda a nossa jornada na graduação.

Agradeço ao meu orientador, professor Msc. Víctor Manuel García Martínez, e meu coorientador Dr. Moisés Renato Nunes Ribeiro, por todo apoio, paciência, disponibilidade e atenção para o desenvolvimento e conclusão do projeto.

À banca examinadora pela aceitação do convite e pelo tempo investido para leitura e avaliação desse trabalho.

E finalmente, agradeço a Universidade Federal do Espírito Santo e a todos os professores, funcionários e aos meus amigos da empresa júnior CT junior que tive a oportunidade de conhecer na mesma. Cada um de vocês tem participação no que me tornei pessoal e profissionalmente.

# RESUMO

*Sites* e *e-commerces*, são exemplos de aplicações *web* que são hospedados na nuvem com a finalidade de serem acessíveis a todos na *Internet*. No entanto, há muitas pessoas mal-intencionadas. A segurança de infraestrutura é um tema de extrema importância nesse contexto. Há diversas espécies diferentes de ataques existentes, dentre elas, o ataque de negação de serviço distribuído (DDoS). O DDoS não tem como finalidade o roubo de dados pessoais ou bancários, este procura tornar a aplicação instável ou, até mesmo, indisponível, ocasionando em prejuízos financeiros. Tal ataque tem por característica o fato de ser dificilmente rastreável e de ainda não haver uma maneira de proteger a aplicação web deste ataque, restando então apenas formas de mitigá-lo. Este trabalho propõe avaliar o uso de plataformas de nuvem na mitigação de ataques destinados a aplicações baseadas em microsserviços. Para isto, foram utilizados *softwares* renomados para o desenvolvimento do *setup* de experimentos, definidos dois ataques para as experimentos e a avaliação de três opções de configuração de escalonadores do *Kubernetes* a serem utilizados como possibilidade de mitigação dos ataques. No contexto analisado foi observado que a utilização do Escalonador Automático vertical de *Pod* com limites definidos de CPU e memória trouxe uma maior vantagem se comparada as outras opções.

**Palavras-chave**: microsserviços; nuvem; *Kubernets*; DDoS; mitigação.

# ABSTRACT

Websites and e-commerces are examples of web applications that are hosted in the cloud with the purpose of being accessible to everyone on the Internet. However, there are many people who are malicious. Infrastructure security is an extremely important issue in this context. There are several types of existing attacks, among them, the attack of attacks Distributed Denial-of-Service (DDoS). DDoS is not intended to steal personal or bank data, this seeks to make the application unstable or even unavailable, causing financial losses. Such an attack is characterized by the fact that it is difficult to trace and that there is no way to protect the web application from this attack, leaving only ways to mitigate it. This work proposes to evaluate the use of cloud platforms in the mitigation of attacks aimed at applications based on microservices. For this, renowned softwares was used to develop the experiment setup, was defined two attacks for the experiments and evaluated three options for configuring Kubernetes schedulers to be used as a possibility of mitigating the attacks. In the analyzed context, it was observed that the use of the Kubernetes Vertical Pod Autoscaler (VPA) with defined limits of CPU and memory brought a greater advantage compared to the other options.



**Keywords**: microservices; cloud; Kubernetes; DDoS; mitigation.

# LISTA DE FIGURAS


Figura 1 – Diagrama demonstrativo comparando a arquitetura monolítica e a de microsserviços . . . . . 19
Figura 2 – Diagrama demonstrativo sob a estrutura da arquitetura monolítica e a de microsserviços . . . . . 20
Figura 3 – Diagrama representativo acerca da arquitetura de um node do *Kubernetes* 22
Figura 4 – Diagrama representativo acerca da arquitetura de um *Pod* do *Kubernetes* 23
Figura 5 – Diagrama representativo acerca da arquitetura de um serviço do *Kubernetes* . . . . . 23
Figura 6 – Diagrama comparativo entre os escalonadores HPA e VPA . . . . . 25
Figura 7 – Diagrama demonstrativo da funcionalidade do *ScaledObject* do KEDA . 26
Figura 8 – Diagrama demonstrativo do funcionamento de uma API REST . . . . 27
Figura 9 – Página *web* desenvolvida em PHP com vulnerabilidade de injeção de código . . . . . 29
Figura 10 – Página *web* desenvolvida em PHP sofrendo ataque de injeção de código 29
Figura 11 – Página *web* sofrendo ataque de injeção SQL através do *login* . . . . . 30
Figura 12 – Página *web* sofrendo ataque de injeção SQL através de um campo de busca . . . . . 31
Figura 13 – Diagrama representativo sobre o funcionamento do ataque de autenticação roubada . . . . . 32
Figura 14 – Diagrama representativo sobre o funcionamento do ataque MITM . . . 33
Figura 15 – Diagrama demonstrativo de como um ataque DDoS ocorre . . . . . 33
Figura 16 – Diagrama descritivo do modelo *Open Systems Interconnection* (OSI) . 34
Figura 17 – Diagrama demonstrativo a cerca do *setup* de experimentos projetado . 37
Figura 18 – Tabela *users* do banco de dados *PostgreSQL* . . . . . 38
Figura 19 – Diagrama demonstrativo a cerca da visão geral do módulo 1 . . . . . 45
Figura 20 – Diagrama demonstrativo acerca da visão geral do módulo 2 . . . . . 47
Figura 21 – Diagrama demonstrativo acerca da visão geral do módulo 3 . . . . . 48
Figura 22 – Captura da tela do *software Grafana* . . . . . 48
Figura 23 – Captura da tela do *software Grafana* na página de alertas . . . . . 49
Figura 24 – Exposição do uso de CPU da aplicação nos experimentos sem escalonamento . . . . . 51
Figura 25 – Exposição do uso de memória da aplicação nos experimentos sem escalonamento . . . . . 51
Figura 26 – Exposição do uso de CPU da aplicação sob o ataque *Raven-Storm* utilizando HPA HTPP . . . . . 53

Figura 27 – Exposição do uso de memória da aplicação sob o ataque *Raven-Storm* utilizando HPA HTPP . . . . . 53
Figura 28 – Avaliação do efeito da criação de um *Pod* sob o ataque *Raven-Storm* . 54
Figura 29 – Exposição do uso de CPU da aplicação sob o ataque *DDoS-Ripper* utilizando HPA métricas . . . . . 55
Figura 30 – Exposição do uso de memória da aplicação sob o ataque *DDoS-Ripper* utilizando HPA métricas . . . . . 55
Figura 31 – Avaliação do efeito da criação de um *Pod* sob o ataque *DDoS-Ripper* . 56
Figura 32 – Exposição do uso de CPU da aplicação sob o ataque *Raven-Storm* utilizando HPA métricas . . . . . 56
Figura 33 – Exposição do uso de memória da aplicação sob o ataque *Raven-Storm* utilizando HPA métricas . . . . . 57
Figura 34 – Exposição do uso de CPU da aplicação sob o ataque *Raven-Storm* com 1500 *treads* utilizando HPA métricas . . . . . 57
Figura 35 – Exposição do uso de memória da aplicação sob o ataque *Raven-Storm* com 1500 *treads* utilizando HPA métricas . . . . . 58
Figura 36 – Exposição do uso de CPU da aplicação sob o ataque *Raven-Storm* utilizando HPA métricas sob novo limite . . . . . 58
Figura 37 – Exposição do uso de memória da aplicação sob o ataque *Raven-Storm* utilizando HPA métricas sob novo limite . . . . . 59
Figura 38 – Avaliação do efeito de criação de *Pod* sob o ataque *Raven-Storm* . . . . 59
Figura 39 – Exposição do uso de CPU da aplicação sob o ataque *DDoS-Ripper* . . 60
Figura 40 – Exposição do uso de memória da aplicação sob o ataque *DDoS-Ripper* 61
Figura 41 – Exposição do uso de CPU da aplicação sob o ataque *Raven-Storm* utilizando VPA . . . . . 61
Figura 42 – Exposição do uso de memória da aplicação sob o ataque *Raven-Storm* utilizando VPA . . . . . 62
Figura 43 – Comparação dos escalonadores sob a aplicação atacada pelo *DDoS-Ripper* sob a perspectiva da CPU . . . . . 63
Figura 44 – Comparação dos escalonadores sob a aplicação atacada pelo *DDoS-Ripper* sob a perspectiva da memória . . . . . 63
Figura 45 – Comparação dos escalonadores sob a aplicação atacada pelo *Raven-Storm* sob a perspectiva da CPU . . . . . 64
Figura 46 – Comparação dos escalonadores sob a aplicação atacada pelo *Raven-Storm* sob a perspectiva da memória . . . . . 65

# LISTA DE TABELAS

Tabela 1 – Exposição dos valores médios e máximos de tempo de resposta dos experimentos no caso sem escalonamento . . . . . 52
Tabela 2 – Exposição dos valores médios e máximos de tempo de resposta dos experimentos no caso sob o ataque *Raven-Storm* utilizando HPA métricas 60
Tabela 3 – Comparação final dos valores médios e máximos de tempo de resposta dos experimentos sob o ataque *DDoS-Ripper* . . . . . 64
Tabela 4 – Comparação final dos valores médios e máximos de tempo de resposta dos experimentos sob o ataque *Raven-Storm* . . . . . 65

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| API | *Application Programming Interface* |
| CPU | *Central Processing Unit* |
| DDoS | *Distributed Denial-of-Service* |
| DNS | *Domain Name System* |
| DoS | *Denial-of-Service* |
| HPA | *Horizontal Pod Autoscaler* |
| HTTP | *Hypertext Transfer Protocol* |
| HTTPS | *Hypertext Transfer Protocol Secure* |
| IP | *Internet Protocol* |
| JSON | *JavaScript Object Notation* |
| JWT | JSON *Web Tokens* |
| k8s | *Kubernetes* |
| MITM | *Man-in-the-middle* |
| VM | Máquina virtual |
| OSI | *Open System Interconnection* |
| REST | *Representational State Transfer* |
| RPC | *Remote Procedure Call* |
| SAML | *Security Assertion Markup Language* |
| SMPC | *Simple Mail Transfer Protocol* |
| SOAP | *Simple Object Access Protocol* |
| SSL | *Secure Sockets Layer* |
| SQL | *Standard Query Language* |
| TCP | *Transmission Control Protocol* |

| | |
|---|---|
| TLS | *Transport Layer Security* |
| UFES | Universidade Federal do Espírito Santo |
| URI | *Uniform Resource Identifier* |
| VPA | *Vertical Pod Autoscaler* |
| XML | *Extensible Markup Language* |

# SUMÁRIO


**1 INTRODUÇÃO . . . . . 14**
**1.1 Apresentação do Tema e Justificativa . . . . . 14**
**1.2 Objetivo Geral . . . . . 16**
**1.3 Objetivo Específico . . . . . 16**
**1.4 Metodologia de pesquisa . . . . . 16**
**1.5 Organização da Monografia . . . . . 17**

**2 REFERENCIAL TEÓRICO . . . . . 18**
**2.1 Arquitetura de microsserviços . . . . . 18**
**2.2 Virtualização e contêineres . . . . . 19**
**2.3 *Kubernetes* (k8s) . . . . . 21**
2.3.1 Escalonadores . . . . . 24
**2.4 *Application Programming Interface* (API) . . . . . 26**
**2.5 Ataques às soluções baseadas em microsserviços . . . . . 28**
2.5.1 *API Injection Attacks* . . . . . 28
2.5.2 *Authentication Hijacking Attack* . . . . . 31
2.5.3 *Man-in-the-middle* (MITM) . . . . . 32
2.5.4 *Distributed Denial-of-Service* (DDoS) . . . . . 33

**3 METODOLOGIA . . . . . 36**
**3.1 Criação das imagens . . . . . 37**
3.1.1 Configuração dos microsserviços . . . . . 37
3.1.2 Configuração dos ataques . . . . . 39
**3.2 Configuração do ambiente no *Kubernetes* . . . . . 40**
3.2.1 Configuração dos microsserviços . . . . . 41
3.2.2 Configuração dos ataques . . . . . 43
**3.3 Configuração dos escalonadores no *Kubernetes* . . . . . 44**
3.3.1 *Horizontal Pod Autoscaling* (HPA) utilizando número de solicitações de acesso HTTP . . . . . 44
3.3.2 *Horizontal Pod Autoscaling* (HPA) utilizando as métricas memória e CPU . 46
3.3.3 *Vertical Pod Autoscaling* (VPA) utilizando as métricas memória e CPU . . 47
**3.4 Monitoramento com *prometheus* e *grafana* . . . . . 48**

**4 RESULTADOS E ANÁLISES . . . . . 50**
**4.1 *Horizontal Pod Autoscaling* (HPA) utilizando número de solicitações de acesso HTTP . . . . . 52**

4.2 *Horizontal Pod Autoscaling* (HPA) utilizando as métricas memória e CPU . . . . . 54
4.3 *Vertical Pod Autoscaling* (VPA) utilizando as métricas memória e CPU . . . . . 60
4.4 Análise . . . . . 62

5 CONCLUSÃO . . . . . 67

REFERÊNCIAS . . . . . 69

APÊNDICE A – BANCO DE DADOS . . . . . 72

APÊNDICE B – PROMETHEUS . . . . . 74

APÊNDICE C – SCALED OBJECT . . . . . 77

APÊNDICE D – HORIZONTAL POD AUTOSCALING UTILIZANDO MEMORIA E CPU . . . . . 78

APÊNDICE E – VERTICAL POD AUTOSCALING UTILIZANDO MEMORIA E CPU . . . . . 79

# 1 INTRODUÇÃO

## 1.1 Apresentação do Tema e Justificativa

Os hábitos populacionais foram drasticamente modificados no decorrer dos últimos anos. Com o surgimento da pandemia da COVID-19, grande parte das empresas necessitaram implementar diversas modificações em seus processos internos a fim de se adaptar a nova realidade. Inúmeros profissionais tiveram que trabalhar remotamente por mais de um ano, muitos desses que, possivelmente, nunca haviam trabalhado remotamente. Como consequência, houve uma intensificação no uso de serviços digitais. Isto porque uma grande fatia da população se mantia em isolamento social. Logo, serviços digitais como entrega de comida, *streaming* e redes sociais, aumentaram bastante suas demandas. Obedecendo o fluxo comum, com o aumento da demanda, houve também o aumento da oferta.

À medida que as empresas tornaram-se totalmente dependentes da qualidade das experiências digitais para se manterem competitivas, o uso da nuvem se tornou indispensável e os desenvolvedores sofreram uma intensa pressão para atender as novas demandas. O aproveitamento dos microsserviços nessa situação, tornou possível a adição de novos códigos, atualização de recursos e correção de *bugs* a taxas extraordinárias. Da mesma maneira, os programadores sofreram para gerenciar e dimensionar serviços para atender todas as expectativas de experiência dos usuários. Para auxiliar na solução desse problema, foi recorrido a tecnologias de orquestração de contêineres como o *Kubernetes* para gerenciar e priorizar cargas de trabalho. (MISTRETTA, 2022).

No entanto, ao utilizar a arquitetura baseada em microsserviços, utiliza-se também muitas APIs diferentes. Isto causa o aumento da superfície de ataque e torna a comunicação vulnerável à ameaças de segurança. Por este motivo, é necessário que todos os microsserviços estejam devidamente protegidos. Há diversos tipos de ataques as APIs e diversas possibilidades de aplicá-los. Existem aqueles que focam na leitura das informações, tais como, *Data Exposure* e *Unencrypted Communcations*. Os que interceptam e manipulam as informações, tais como, *Man-in-the-middle* (MITM). Os que roubam o código de autenticação e, com isso, conseguem livremente realizar requisições novas, tais como, *Authentication Hijacking Attack*. Aqueles que por meio de *scripts* consegue manipular as informações do servidor, tais como, *API Injection Attacks*. Dentre muitos outros.

A maioria dos ataques tem o enfoque nos dados, tanto os transmitidos na comunicação quanto os armazenados no banco de dados. Isto porque, geralmente, pessoas mal-intencionadas buscam informações sigilosas, tais como, dados bancários ou dados internos de empresas, para que seja possível solicitar um resgate dos dados, das duas formas

obtendo benefícios financeiros através dos atos. Nesse contexto, há diversas metodologias e tecnologias que podem ser utilizadas para tornar a comunicação mais robusta e menos vulnerável, buscando a segurança da integridade dos dados. No entanto, existem algumas espécies de ataque que não tem o foco nos dados, dentre elas, algumas tem o objetivo de tornar a aplicação lenta ou, até, a tornar indisponível, como por exemplo, o *Distributed Denial-of-Service* (DDoS). Devido suas especificações, não há soluções de segurança que protegem completamente a aplicação desse ataque, existem apenas diversas tecnologias e metodologias que o mitigam.

Por volta do segundo trimestre de 2020, iniciaram-se os aumentos nos registros dos número de ataques DDoS. Os chamados de resgate aumentaram quase um terço entre 2020 e 2021 e aumentaram 75% no quarto trimestre de 2021 em comparação com os três meses anteriores. (COOK, 2022). A *Microsoft* afirma que, no segundo semestre de 2021, mitigou uma média de 1955 ataques do tipo por dia, um aumento de 40% em relação aos seis primeiros meses do mesmo ano. Ainda, a empresa registrou em 2021 um recorde de pico de tráfego de 3,47 Tb/s. Essa operação foi direcionada a um cliente *Azure* na Ásia e envolveu cerca de 10 mil fontes diversas, alcançando a taxa de 340 milhões de pacotes por segundo. Em dezembro de 2021, dois outros ataques grandiosos foram detectados, novamente contra clientes Azure na Ásia. O primeiro foi um ataque que resultou em um pico de tráfego de 3,25 Tb/s; o segundo foi uma operação com pico de 2,55 Tb/s. (ALECRIM, 2022).

A *Kaspersky*, uma empresa tecnológica russa especializada na produção de *softwares* de segurança à *Internet*, afirmou que seu sistema de inteligência registrou uma média de 1406 ataques diários em janeiro e fevereiro de 2022. O pior dia de ataques foi 19 de janeiro de 2022, quando identificou 2250 ataques. (COOK, 2022). Em um contexto regional mais próximo, provedores de *Internet* do Espírito Santo (ES) apresentaram por cerca de 30 dias instabilidades no acesso à *Internet*. Por conta disto, foi apresentado uma denúncia à Comissão Parlamentar de Inquérito de crimes cibernéticos da Assembleia Legislativa. Um acontecimento surpreendente dado que "ora os empresários são incrédulos quanto a competência das autoridades em investigar crimes cibernéticos, ora as autoridades quando acionadas parecem insensíveis quanto ao tema.". (AYUB, 2022). Isto tem a possibilidade de ser explicado pelo fato de que dificilmente os atacantes são encontrados. Grande parte dos ataques do tipo DDoS dispõem da característica de serem dificilmente rastreáveis.

Portanto, este trabalho visa a busca de mais informações acerca do contexto apontado. Diante do exposto, em especial, no tangente ao ataque *Distributed Denial-of-Service* (negação de serviço distribuída, em tradução livre). Por fim, o presente trabalho visa desenvolver um cenário controlado, composto por uma aplicação *web* baseada em microsserviços formada por um *Back-End*, *Front-End* e um banco de dados. Com o objetivo de avaliar os diferentes

tipos de escalonamento do *Kubernetes* disponíveis como forma de mitigação de ataque DDoS de inundação HTTP (camada 7 do modelo OSI).

## 1.2 Objetivo Geral

O objetivo geral do projeto é avaliar o uso de escalonadores do *Kubernetes* na mitigação de ataques DDoS de inundação HTTP destinados à aplicações baseadas em microsserviços.

## 1.3 Objetivo Específico

Os objetivos específicos definidos neste trabalho, a fim de alcançar o objetivo geral, estão listados a seguir:

- Estudar as tecnologias de conteinerização e virtualização;
- Estudar a arquitetura para aplicações baseada em microsserviços;
- Estudar os principais ataques dirigidos a aplicações baseadas em microsserviços;
- Implantar uma aplicação *web* baseada em microsserviços usando *Kubernetes*;
- Configurar a plataforma de nuvem para mitigar ataques de negação de serviço distribuído (DDoS) de inundação de requisições HTTP;
- Simular ataques de negação de serviço distribuído (DDoS) dirigidos à aplicação implementada num cenário controlado;
- Avaliar os diferentes tipos de escalonamento de microsserviços disponíveis na nuvem como forma de mitigação de ataques.

## 1.4 Metodologia de pesquisa

No que diz respeito à classificação deste trabalho, conclui-se ser, sob a perspectiva de sua natureza, uma pesquisa aplicada. Prodanov (2013) diz que a pesquisa aplicada possui o objetivo de "gerar conhecimentos para aplicação prática dirigidos à solução de problemas específicos". Este trabalho visa implementar um estudo acerca de maneiras de mitigação de um ataque hacker específico que não possui uma solução de proteção definida na

comunidade. No que tange os seus objetivos, define-se como uma pesquisa explicativa, isto é, o presente trabalho busca através do estudo e experimentos classificar as tecnologias definindo os porquês das coisas e suas causas. Segundo Prodanov (2013), a pesquisa experimental caracteriza-se por manipular e controlar variáveis a fim de realizar um estudo da relação entre as causas e os efeitos de determinado fenômeno, assim como no presente trabalho.

## 1.5 Organização da Monografia

Além desta introdução, esta monografia é composta por outros quatro capítulos:

- O Capítulo 2 apresenta os aspectos relativos ao conteúdo teórico relevante para o trabalho e as tecnologias utilizadas;
- O Capítulo 3 apresenta a metodologia na qual foi desenvolvido todo o *setup* de experimentos;
- O Capítulo 4 apresenta os resultados e avaliações sobre os experimentos realizadas;
- O Capítulo 5 apresenta as considerações finais do trabalho.

# 2 REFERENCIAL TEÓRICO

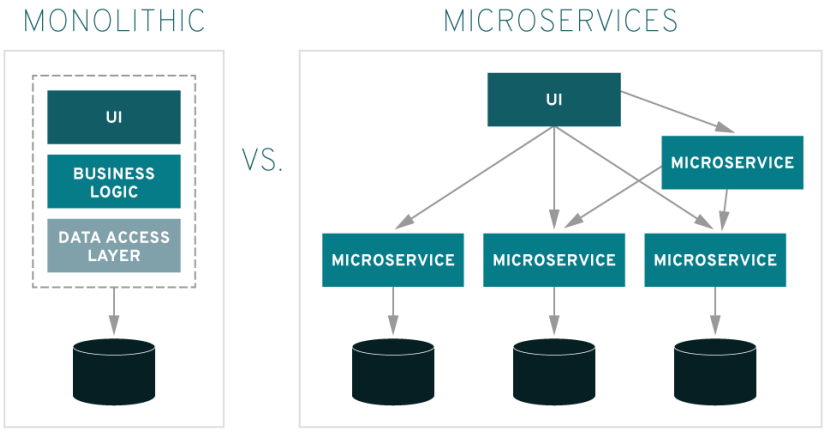
## 2.1 Arquitetura de microsserviços

Uma aplicação com arquitetura monolítica consiste em um único programa independente e autônomo, e, por isso, com a estrutura mais simples de desenvolver. Isto porque todo o código é um sistema singular cujo não é necessário organizar diversos módulos, composto apenas por um serviço de API e uma configuração. No entanto, o aumento da aplicação, consequentemente, o aumento do código é diretamente relacionado ao aumento da dificuldade na manutenção. Ademais, a cada modificação é necessário realizar um novo *deploy* de toda a aplicação, o que custa um tempo maior para a conclusão. Em termos leigos, *deploy* é a fase do ciclo de vida de um *software* que corresponde textualmente a passagem do *software* para a produção. Uma linha de código ou uma falha é capaz de inviabilizar toda a aplicação e nesta estrutura não há flexibilidade no que se refere à tecnologias, linguagens de programação ou bibliotecas adicionais, visto que, sendo somente um módulo, apenas uma poderá ser escolhida e o restante estará condicionado.

Enquanto isso, microsserviços são uma abordagem inovadora de arquitetura que consiste em criar aplicações desenvolvidas em vários serviços independentes que se comunicam entre si utilizando APIs. (DRAGONI et al., 2017). A Figura 1 demonstra a diferença entre as arquiteturas. Cada serviço é relacionado a uma funcionalidade ou finalidade específica, fazendo com que diferentes serviços acessem diferentes camadas. Ao surgir a necessidade de um serviço acessar a camada de outro, o ideal é que se comuniquem ao invés do segundo acessar a camada diretamente. Assim, a aplicação mantém-se modularizada e organizada. Diante disso, os testes e manutenção são simples, há mais agilidade ao realizar os *deploys* e a equipe de desenvolvedores adquire uma enorme flexibilidade no que se trata a divisão dos serviços, dado que cada equipe consegue trabalhar livremente sob cada serviço e escolher qualquer tecnologia, linguagem de programação ou biblioteca adicional.

Contudo, a arquitetura também possui suas desvantagens. Ao projetar é necessário despender uma fração de tempo maior na organização e planejamento dos serviços, pois, caso contrário, há grandes possibilidades de surgirem diversos problemas de modularização e necessidades de reescrita do código. A identificação de origem dos problemas é mais complexa por existirem mais rotas a serem verificadas. E o monitoramento do todo também é mais complexo, dado que na aplicação monolítica há apenas um serviço à monitorar e utilizando microsserviços, cada precisa ser monitorado separadamente. Porém, para a

Figura 1 – Diagrama demonstrativo comparando a arquitetura monolítica e a de microsserviços




Fonte: (REDHAT, 2021).

realização deste monitoramento de serviços, há tecnologias desenvolvidas especificamente com esta finalidade.

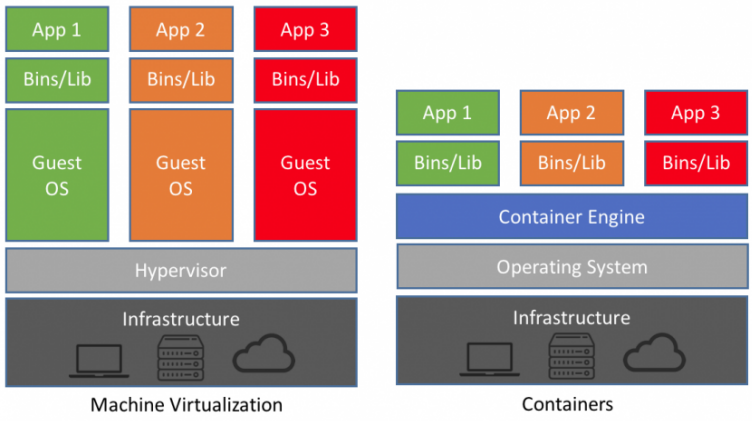
## 2.2 Virtualização e contêineres

Tanto o *software* monolítico quanto os microsserviços são capazes de serem executados em máquinas virtuais (VM) ou contêineres. As máquinas virtuais são computadores de *software* com a mesma funcionalidade que as máquinas físicas, executados na própria. Ou seja, em um computador físico é possível executar diversos outros computadores utilizando a virtualização, com o auxílio do *hypervisor*. O *hypervisor*, também conhecido como monitor da máquina virtual, é um *software* que as cria e gerencia. Este separa os recursos das respectivas máquinas físicas para que tais sejam capazes ser particionados e dedicados as virtuais. Quando o usuário emite uma instrução de máquina virtual que exige mais recursos do ambiente físico, o *hypervisor* retransmite a solicitação ao sistema físico e armazena as mudanças em cache. (REDHAT, 2020).

Os contêineres são uma abstração na camada do aplicativo que empacota código e dependências juntos. Estes ocupam menos espaço (*megabytes*) se comparado a máquinas virtuais (*gigabytes*), isto porque, cada VM contém seu próprio sistema operacional (identificado como "*Guest OS*" na Figura 2), o que possibilita a execução simultânea de várias funções com uso intenso de recursos. Os contêineres possuem sistema operacional compartilhado e não utilizam *hypervisor*, mais sim, os recursos do sistema e o *contêiner engine* para construir e pôr os ambientes em contêineres. Portanto, mesmo que as duas tecnologias sejam válidas a se utilizar junto as duas estruturas de organização de *software*, a conteine-

rização é a mais viável para a arquitetura de microsserviços, além de possuir a vantagem de portabilidade cuja todas as bibliotecas e dependências do projeto são encapsuladas em um arquivo baseado em código, no qual chamamos de imagem.

Figura 2 – Diagrama demonstrativo sob a estrutura da arquitetura monolítica e a de microsserviços




Fonte: (MENDONçA, 2019).

Dentre as opções, no projeto optou-se por utilizar *Docker* para aplicar os conceitos de conteinerização, por motivos de ser, o mais conhecido e possuir uma grande rede de suporte. Este é uma plataforma de código aberto desenvolvida na linguagem *Go* que fornece uma camada de abstração e automação para virtualização de sistemas operacionais utilizando a tecnologia de contêineres. Outro grande diferencial é a existência do *Docker Hub*, uma nuvem pública para armazenamento e compartilhamento de imagens. Nela tem-se 174 imagens oficiais das principais tecnologias utilizadas, dentre elas, *postgres*, *traefik*, *nginx*, *ubuntu*, *alpine*, *mongo*. (DOCKER, 2022).

Há pelo menos duas formas de criar uma imagem, por meio de um *dockerfile* e por meio do comando `docker commit`. O segundo é utilizado quando necessário construir uma imagem a partir de um contêiner existente. O *Dockerfile* é um documento de texto aplicado para montar uma imagem. Cada linha contém um comando interno e seus respectivos parâmetros. Abaixo situa-se um exemplo simples.

```
FROM node:latest
WORKDIR /var/www/
COPY . /var/www
RUN npm install
ENTRYPOINT npm start
EXPOSE 3000
```

Na primeira linha, tem-se o comando `FROM` que indica a origem da imagem. Toda imagem personalizada necessita ter como origem uma base. Por este motivo, é uma grande vantagem existir um repositório tão completo como o *Docker hub* a disposição. Em seguida, tem-se o parâmetro, seguido do nome da imagem, `node`. Em sequência, o símbolo "dois pontos" (:) e a versão `latest` (mais recente, em tradução livre).

Na próxima linha, tem-se o comando `WORKDIR` que define o diretório de trabalho, isto é, o diretório no qual vai-se iniciar o contêiner. Em seguida, o comando `COPY`, que copia os dados de um diretório para outro dentro do contêiner, visto que o primeiro parâmetro é o diretório local. Vale ressaltar que se o usuário deseja copiar os arquivos do diretório atual, basta especificar com o símbolo de "ponto final" (.).

Na quarta linha, tem-se o comando `RUN`. Esse especifica que os parâmetros são um comando a ser executado dentro do contêiner, no processo de criação da imagem. O `ENTRYPOINT`, quinta linha, é semelhante ao anterior, exceto pelo fato do comando só ser executado ao iniciar o contêiner. Por fim, `EXPOSE` define a porta que será empregue e exposta do contêiner. Há diversos outros comandos que podem ser manipulados na criação da imagem, tal como `ENV` que define uma variável de ambiente e `USER` que define o usuário a ser utilizado.

A criação de imagens é um fator importante para o projeto posto que é a base para iniciar um *setup* no *Kubernetes*.

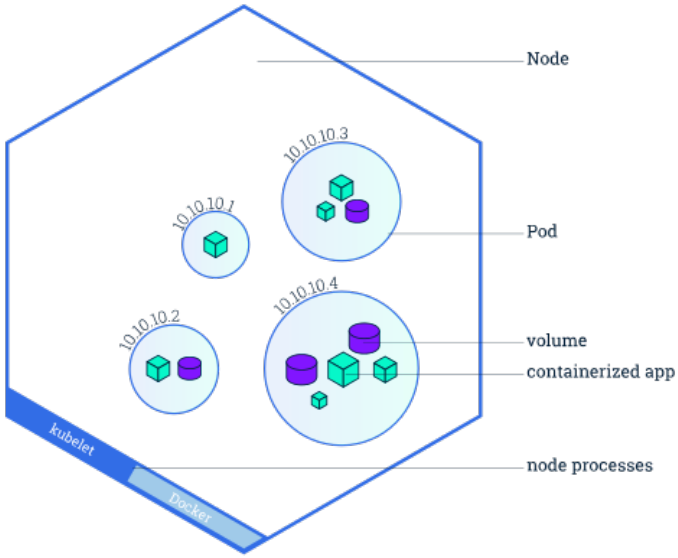
## 2.3 *Kubernetes* (k8s)

O *Kubernetes* (k8s) é uma poderosa ferramenta que automatiza a implantação e o gerenciamento de contêineres. O seu principal objetivo é resolver o processo de implantação automatizada de serviços. No ecossistema de microsserviços, é uma ferramenta extraordinária visto que a mesma auxilia e facilita muitos processos manuais de criação e gerenciamento dos microsserviços. Além de facilitar o escalonamento dos arranjos, tanto de forma manual quanto de forma automática, obedecendo parâmetros previamente configurados, dentre outras características.

Em termos leigos, um *cluster* k8s é um conjunto de nós (máquinas físicas) virtualmente conectados entre si e executando aplicativos em contêineres. O responsável por gerenciá-lo é chamado de *master*. Ele coordena todas as atividades, como agendamento, manutenção e dimensionamento de aplicativos, também lançando novas atualizações. (GANESAN, 2019).

O *node* (Fig. 3), ou em português, nó, é uma máquina de trabalho no *Kubernetes*, sendo possível ser uma máquina virtual ou física. Cada nó possui um *kubelet*, um agente para gerenciar os *pods* e os contêineres e se comunicar com o *master*. Um nó suporta diversos *pods*, considerando os recursos disponíveis.

Figura 3 – Diagrama representativo acerca da arquitetura de um node do *Kubernetes*



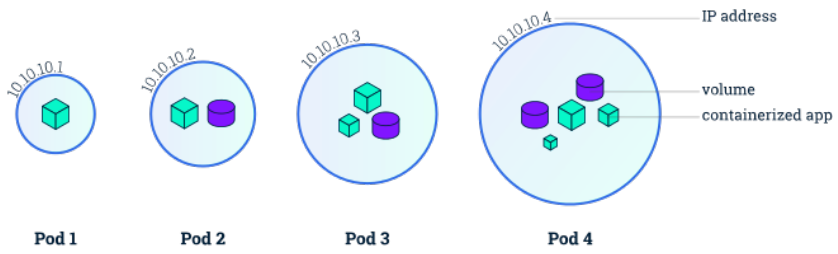

Fonte: (GANESAN, 2019).

Um *Pod* é a unidade de execução básica de um aplicativo k8s. Ele representa uma única instância de um aplicativo no *Kubernetes*, que consiste em um único contêiner ou em um pequeno número de contêineres fortemente acoplados que compartilham recursos. Dentre eles, volumes, um IP de rede exclusivo e configurações que controlam como os contêineres devem ser executados. Alguns destes são o *Deployment*, *Statefulset* e o *Daemonset*. (GANESAN, 2019). Normalmente, para criar um *Pod*, é criado uma imagem de contêiner, essa é enviada para um registro e na declaração de criação a mesma é referenciada.

O volume (visualizado na Figura 4) é apenas um armazenamento de dados acessível aos contêineres em um *Pod*. Independente da ação que ele sofra, o volume mantém os dados preservados. Há duas categorias relacionadas a volumes, o *Persistent Volume* (PV) e o *Persistent Volume Claim* (PVC), o PVC é uma reivindicação para a plataforma criar um PV e o PV é de fato o volume "físico".

O objetivo principal do *Deployment* (citado acima) é declarar o estado desejado do sistema, este é utilizado em diversos casos de uso, dentre eles, para aumentar de maneira simples o número solicitado de *pods* e recriar automaticamente algum que caso tenha sido encerrado.

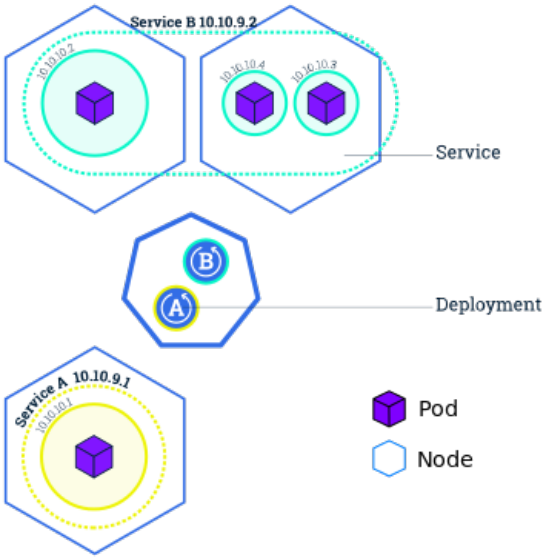
Figura 4 – Diagrama representativo acerca da arquitetura de um *Pod* do *Kubernetes*




Fonte: (GANESAN, 2019).

Um *service* (Fig. 5), ou em português, serviço, é uma abstração que define um conjunto lógico de *pods* e uma política para acessá-los. Representa um único *Domain Name System* (Sistema de Nomes de Domínio, em tradução livre, ou DNS) para um conjunto de *pods* e consegue balancear a carga entre eles. Há duas maneiras de se comunicar com um *Pod*, por meio do DNS definido na criação do serviço ou do IP relacionado.

Figura 5 – Diagrama representativo acerca da arquitetura de um serviço do *Kubernetes*




Fonte: (GANESAN, 2019).

Há, pelo menos, duas formas de iniciar um componente no *Kubernetes*. Executando um comando padrão (`kubectl run`) junto com os parâmetros, entre os quais, o nome do componente e a imagem são parâmetros obrigatórios. Ou por meio de um *YAML*, este é um formato de serialização de dados legíveis.

O *Kubernetes* é uma ferramenta bem completa. Acima foram citados os principais recursos e abaixo são citados os demais que foram utilizados para o desenvolvimento do projeto.

- ***Namespace*:** é um mecanismo de isolamento de um grupo de recursos dentro de um *cluster*;
- ***Cluster Role* e *Cluster Role Binding*:** são declarativos de permissões, sem declaração de *namespace*, vale ressaltar que as permissões são puramente aditivas (não há regras de exclusão);
- ***ServiceAccount*:** fornece uma identidade para processos executados em um *Pod*.
- ***Config Map*:** é um objeto da API usado para armazenar dados não-confidenciais em pares chave-valor, tal como a criação de variáveis de ambiente;
- ***Scaled Object* e *Horizontal Pod Autoscaler*:** são manifestos de configuração para a automatização de réplicas de *pods*;
- ***Vertical Pod Autoscaler*:** é um tipo de manifesto de configuração para a automatização de redimensionamento de *pods*;
- ***kubectl*:** é uma ferramenta que provê as funcionalidades de criar, ler, atualizar e remover os dados e componentes do *cluster*.

### 2.3.1 Escalonadores

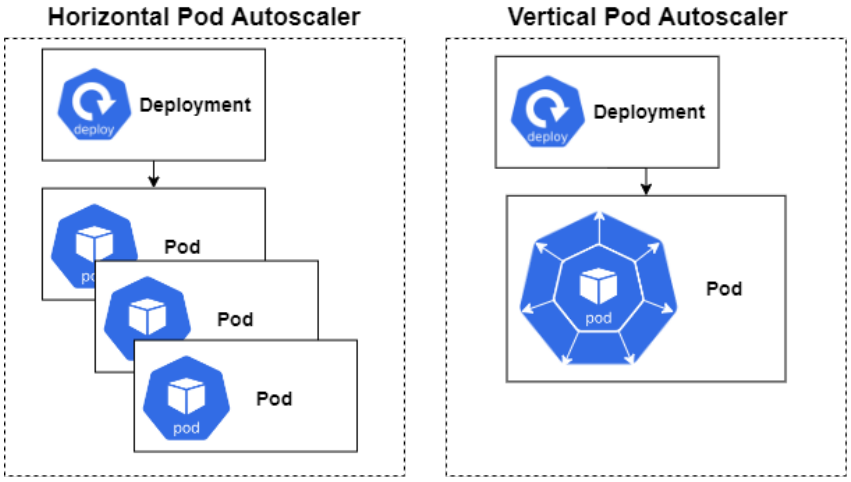
Há duas maneiras de realizar o escalonamento no *Kubernetes*, o *Horizontal Pod Autoscaler* (Escalonamento Horizontal do *Pod*, em tradução livre) e *Vertical Pod Autoscaler* (Escalonamento Vertical do *Pod*, em tradução livre). O escalonamento horizontal altera o número de réplicas de *Pods* do contexto, enquanto o vertical as métricas internas do *Pod*. Diferença exemplificada na Figura 6.

É possível realizar o escalonamento de modo manual na criação do *Deployment*, definindo no YAML o campo `replicas` (caso HPA) ou o `resources` (caso VPA) para o valor desejado. Ou quando o contexto está em execução, através de um simples comando. Exemplificando, o comando abaixo define um novo número de réplicas.

```
kubectl scale --replicas=<numero_replicas> deployment <nome_deployment>
```

No entanto, essas são ações manuais. O *Kubernetes* possibilita dimensionar automaticamente os *Pods* utilizando um valor de maneira declarativa, dados de CPU, memória ou, ainda, é possível integrar métricas personalizadas e fornecidas externamente.

Figura 6 – Diagrama comparativo entre os escalonadores HPA e VPA



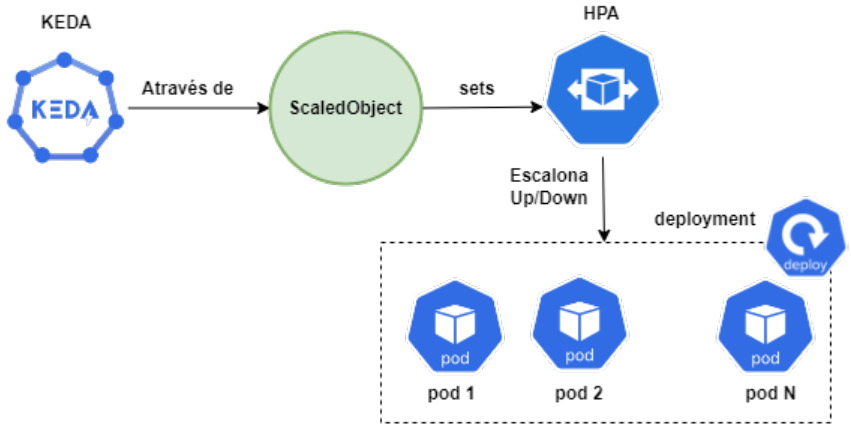

Fonte: Desenvolvido pelo próprio autor.

Para este projeto foi definido a utilização de três configurações de escalonadores, o HPA utilizando requisições HTTP, o HPA utilizando as métricas memória e CPU e o VPA utilizando as métricas memória e CPU. Apenas no caso do primeiro escalonador foi necessário a utilização de um aplicativo e um módulo externo, o *Prometheus* e o KEDA. O *Prometheus* é um aplicativo de código aberto utilizado para monitoramento e alerta de eventos. O mesmo armazena métricas em tempo real em um banco de dados de séries temporais. Para isso, foi utilizado para armazenar o número de solicitações de acesso HTTP enviado do *Back-End* da aplicação, ao receber uma requisição HTTP.

Enquanto o KEDA (*Kubernetes Event Driven Autoscaling*), um operador de código aberto do *Kubernetes* que se integra nativamente ao *Horizontal Pod Autoscaling*, para fornecer escalonamento automático refinado para cargas de trabalho orientadas a eventos. Esse suporta o conceito de *Scalers* que atuam como uma ponte entre o KEDA e o sistema externo do qual as métricas precisam ser buscadas. No caso deste trabalho, o *Scaler* escolhido é o *Prometheus*.

O *ScaledObject* (Fig. 7) é a definição de recurso personalizada que é utilizada para definir como o KEDA deve dimensionar seu aplicativo e quais são os gatilhos. (KEDA, 2022). Exemplificando, quando o número de solicitações é menor que o limite definido, o KEDA consegue dimensionar a implantação para o mínimo de réplicas definido. No decorrer do tempo, quando o limite é ultrapassado, o KEDA consegue direcionar esses dados para o HPA para impulsionar a expansão.

Figura 7 – Diagrama demonstrativo da funcionalidade do *ScaledObject* do KEDA




Fonte: Desenvolvido pelo próprio autor.

## 2.4 *Application Programming Interface* (API)

*Application Programming Interface* (API), do português, Interface de Programação de Aplicativos é um conjunto de regras definidas que explicam como os computadores ou aplicativos se comunicam entre si, atuando como uma camada intermediária que processa a transferência de dados entre sistemas com facilidade e segurança. (EDUCATION, 2020). Há diferentes protocolos de API, para que cada atue de acordo com as necessidades e objetivos do sistema. Dentre os quais, os principais são o SOAP (do inglês, Simple Object Access Protocol), RPC (do inglês, Remote Procedure Call) e REST.

O protocolo SOAP (em português, Protocolo Simples de Acesso a Objetos) é um protocolo construído em um formato de arquivo universal usado para criar documentos com dados organizados, conhecido como XML (do inglês, Extensible Markup Language). Este é um protocolo dito neutro, isto é, ele pode operar por diferentes protocolos de comunicação, como HTTP, SMTP, TCP, dentre outros. Além de ser conhecido por sua extensibilidade e independência de estilos de programação específicos, características que facilitam a adição de funcionalidades ao código. No entanto, o protocolo tem regras e procedimentos bastante rígidos, que requerem atenção especial do programador e era mais popular no passado. (IUGU, 2022).

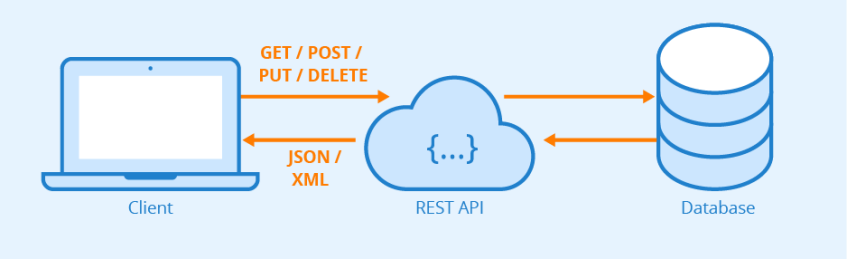
O RPC (em português, Chamadas de Procedimento Remoto) oferece um método simples para solicitar uma operação computacional de outro sistema, isto é, o protocolo envia parâmetros e recebe o resultado da operação. Este pode ser escrito em dois formatos diferentes, o JSON (JSON-RPC) ou XML (XML-RPC). O XML-RPC é mais antigo que o SOAP, contudo mais simplificado e relativamente leve, visto que usa uma largura de banda mínima. (EDUCATION, 2020).

No protocolo REST (em português, Transferência Representacional de Estado) a interface deve obedecer a certas restrições arquitetônicas, dentre elas, clientes e servidores trocam dados utilizando os verbos HTTP, tais como, `POST`, `GET`, `PUT` e `DELETE`. As APIs REST (também conhecida como API RESTful) também conseguem armazenar dados em cache para APIs lentas. Para isso, entretanto, a solicitação deve expressar uma permissão de armazenamento em cache. (IUGU, 2022).

O diagrama abaixo (Fig. 8) demonstra de maneira simples de qual modo a API REST funciona, enquanto o código abaixo, uma solicitação real. A solicitação é composta pelo verbo HTTP (`GET`), precedido do caminho da API solicitada (`/api/users/getAll`), do protocolo sequencial `HTTP/1.1` e do *host*, ou melhor, o IP (`192.168.15.27`) e porta (`8080`) do servidor de destino.

```
GET /api/users/getAll HTTP/1.1
Host: 192.168.15.27:8080
```

Figura 8 – Diagrama demonstrativo do funcionamento de uma API REST




Fonte: (NAEEM, 2020).

Em geral, utiliza-se API REST para a comunicação entre microsserviços, visto que por meio de sua padronização, a sua construção é simplificada e terceiros conseguem com maior facilidade entender. As principais vantagens em usufruir são a compatibilidade do navegador e escalabilidade, além das citadas anteriormente. Porém, o ideal não é empregar unicamente o HTTP nas requisições, pois, o protocolo possui poucos requisitos de segurança, o que aumenta a vulnerabilidade da aplicação podendo propiciar ocorrências de diversos ataques, tais como, ataques de injeção (do inglês, injection attacks), ataques de repetição (do inglês, *replay attacks*), homem no meio (do inglês, *man-in-the-middle*), dentre outros. (SILVA, 2017).

Uma maneira simples de aumentar a segurança sob o tráfego de dados entre os microsserviços é criptografar os dados. Ao se utilizar o HTTPS, a segurança é maior posto que o tráfego de dados entre cliente e servidor não é exposto. Dentre os algoritmos de criptografia temos o SSL (do inglês, *Secure Sockets Layer*) e o TLS (do inglês, *Transport*

*Layer Security*). Ambos possuem normas específicas, porém não há grandes diferenças entre os dois. O SSL utiliza o algoritmo MAC (do inglês, *Message Authentication Code*). Enquanto o TLS têm algoritmos de criptografia mais fortes, como o HMAC (do inglês, *Hash-based for Message Authentication Code*) e, é capaz de operar em diferentes portas.

O Newman (2015) em seu livro sugere que seja considerado que cada microsserviço tenha seu próprio conjunto de credenciais, tais como, SAML (do inglês, *Security Assertion Markup Language*) ou *OpenID Connect*. Além de sugerir a manipulação de chaves API (do inglês, API keys) que delimitam as solicitações do usuário pelo perfil do mesmo. Neste contexto, é possível aplicar o conceito de defesa em profundidade (do inglês, *Defense in depth*) criando um *gateway* API para centralizar todas as solicitações e suas respectivas credenciais.

O *gateway* deve ser o único ponto pertencente ao DMZ (do inglês, *Demilitarized Zone*) e é o ponto que deve criptografar e descriptografar os dados. Essa ação auxilia na proteção contra ataques *man-in-the-middle*, *eavesdropping* (do português, espionagem) e *tampering* (do português, adulteração). De maneira que não há formas de interceptar, editar, deletar ou adicionar dados na comunicação entre os dois pontos. (SILVA, 2017).

## 2.5 Ataques às soluções baseadas em microsserviços

Em um *software* baseado em microsserviços, cada um, individualmente, se comunica com outro por meio das APIs. Em comparação a aplicação monolítica, isto pode se torna uma vulnerabilidade. Dado que, ao utilizar a arquitetura citada, apenas a comunicação entre *Front-End* e *Back-End* é desenvolvida, enquanto na baseada em microsserviço, há diversas APIs expostas. Isto causa o aumento da superfície de ataque e torna a comunicação vulnerável a ameaças de segurança. Para superar as ameaças a segurança, é necessário que todos os microsserviços estejam devidamente protegidos. (KANJILAL, 2021). Nesta seção serão abordados alguns dos principais ataques sob o contexto de microsserviços.

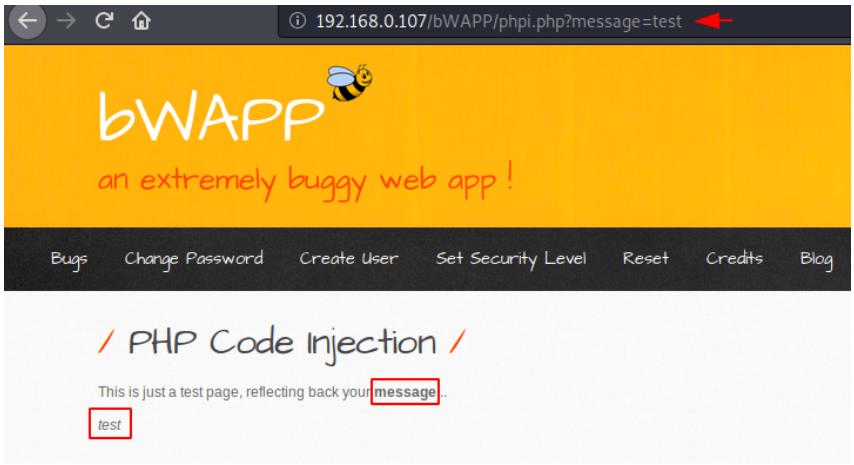
### 2.5.1 *API Injection Attacks*

*API Injection Attacks* (Ataques de injeção de API, em tradução livre) ocorrem quando dados não confiáveis são enviados para um intérprete como parte de um comando ou consulta. Dois dos tipos mais comuns são: SQL *injections* (SQLi) e *Code injections* (Injeções SQL e Injeções de código, em tradução livre).

Se os invasores detiverem o conhecimento da linguagem de programação aplicada em um aplicativo, é possível realizar a injeção de código por meio de campos de entrada de texto para forçar o servidor da *web* a executar as instruções desejadas. (VAADATA, 2022).

Exemplificando, a Figura 9 demonstra uma página *web* construída em PHP que dispõe de vulnerabilidade de injeção de código. Neste caso, é simples perceber o termo "php" e a mensagem exibida na tela estão expostos na URL do *site*.

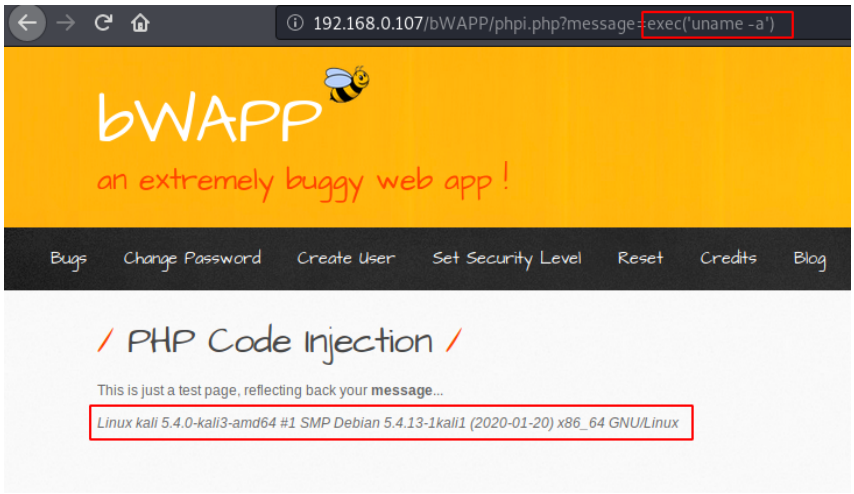
Figura 9 – Página *web* desenvolvida em PHP com vulnerabilidade de injeção de código




Fonte: (GOYAL, 2020).

A vulnerabilidade está relacionada ao PHP, dessa forma, o código a ser injetado necessita estar em PHP. Uma pessoa mal-intencionada consegue, facilmente, pesquisar alguns métodos da linguagem especificada e usufruir-los sob uma falha como essa, assim como é demonstrado na Figura 10.

Figura 10 – Página *web* desenvolvida em PHP sofrendo ataque de injeção de código

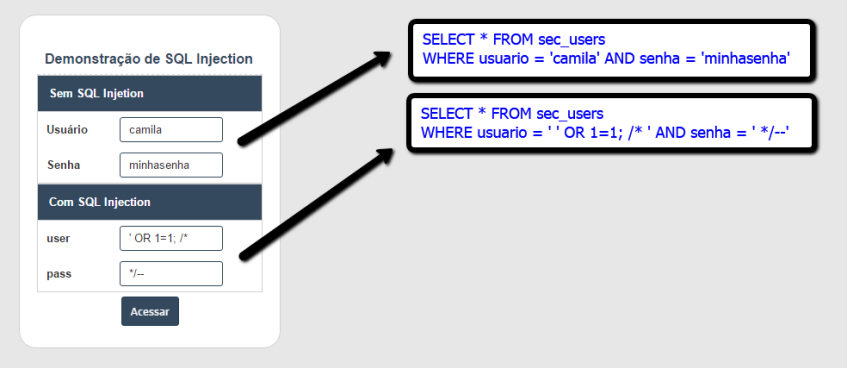



Fonte: (GOYAL, 2020).

Em um ataque de injeção SQL, o invasor injeta códigos para manipular comandos SQL, interagindo assim com o banco de dados por meio de consultas não intencionais. Essas falhas podem levar a manipulação, exclusão ou roubo de informações armazenadas. (VAADATA, 2022). O SQL é uma linguagem bem fácil de se compreender, logo é bem simples criar uma *query*, até mesmo para um leigo.

Uma situação possível de ocorrer nesse ataque é o caso de em uma aplicação existir um campo de busca ou, até mesmo, os campos do *login* sem validação dos dados de entrada. Nesta situação, uma pessoa mal-intencionada consegue se conectar sem ter as permissões necessárias (Fig. 11).

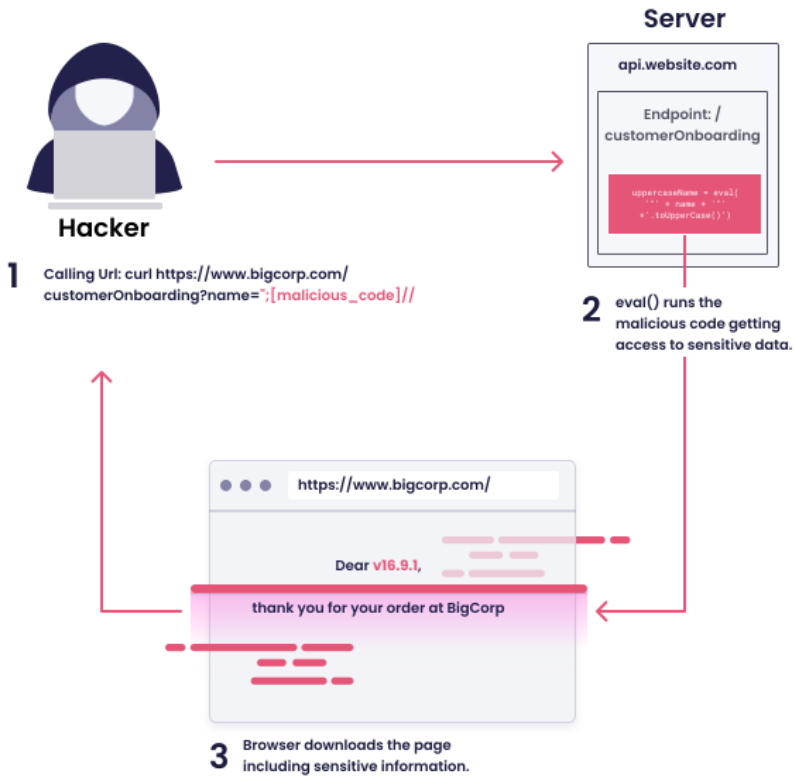
Figura 11 – Página *web* sofrendo ataque de injeção SQL através do *login*




Fonte: (MOREIRA, 2017).

Na primeira situação de exemplo apresentada, os campos de busca, o atacante consegue visualizar informações que não estão a sua disposição, utilizando a mesma estratégia do *login*. Nesse caso, é possível visualizar informações porque o campo é de busca e, naturalmente, a busca retornará informações e, nessa situação, há a possibilidade de retornar dados de outra tabela (Fig. 12).

Figura 12 – Página *web* sofrendo ataque de injeção SQL através de um campo de busca

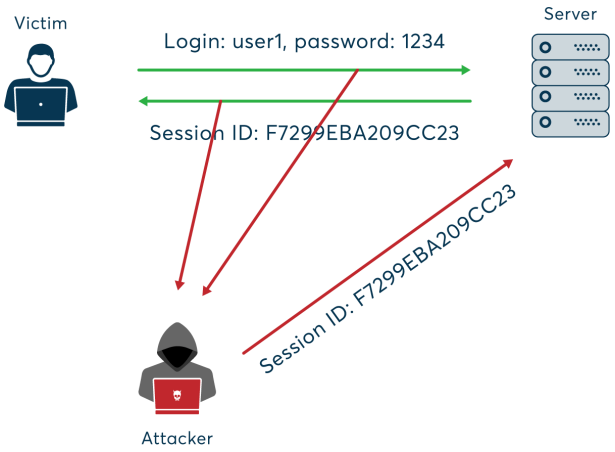



Fonte: (SNYK, 2022).

### 2.5.2 *Authentication Hijacking Attack*

APIs desenvolvidas com falhas na segurança, tal como, tráfego de dados entre cliente e servidor exposto, propiciam o *Authentication Hijacking Attack* (Ataque de Autenticação Roubada, em tradução livre). Há a possibilidade de os invasores tentarem contornar ou quebrar os métodos de autenticação que a API está utilizando. Um exemplo do ataque é exposto na Figura 13.

Figura 13 – Diagrama representativo sobre o funcionamento do ataque de autenticação roubada



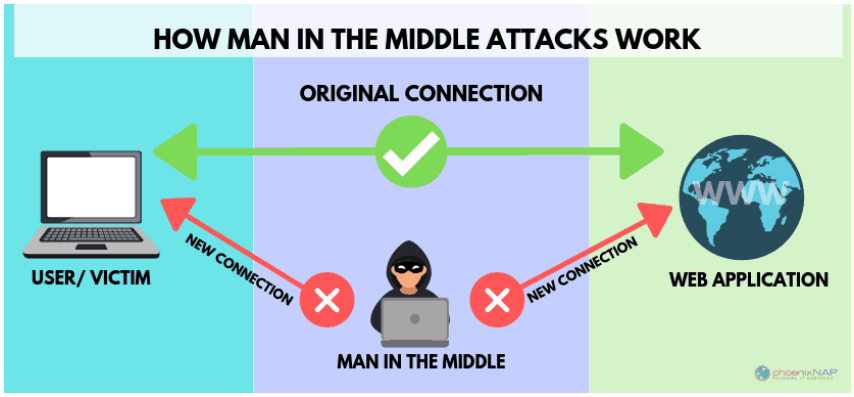

Fonte: (BANACH, 2019).

### 2.5.3 *Man-in-the-middle* (MITM)

Um ataque *Man-in-the-middle* (Homem No Meio, em tradução livre) é um tipo de ataque no qual um invasor discretamente se insere em uma comunicação ou transferência de dados entre um cliente e um servidor, um servidor e um servidor ou um cliente e um cliente (Fig. 14). Assim, agentes mal-intencionados conseguem acesso a dados confidenciais, como informações de identificação pessoal ou manipular a comunicação para introduzir um *malware*. (VAADATA, 2022).

"Transmissão de mensagens não assinadas ou criptografadas, problemas na configuração da sessão segura ou mesmo a utilização de criptografia SSL/TLS com a configuração incorreta, podem comprometer a segurança de APIs e tornar uma organização vulnerável a ataque Man-in-the-middle, comprometendo todas as mensagens com o cliente." (SENHASEGURA, 2022).

Figura 14 – Diagrama representativo sobre o funcionamento do ataque MITM

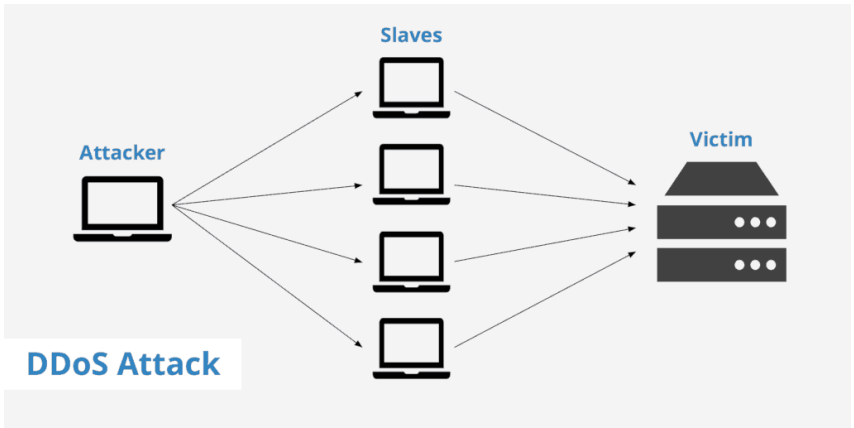



Fonte: (SECURITY, 2021).

### 2.5.4 *Distributed Denial-of-Service* (DDoS)

*Denial-of-Service* (DoS), em português, Negação de Serviço, é um tipo de ataque malicioso em que um atacante, sendo um computador ou um servidor, pertencente a própria autora (ou *hackeado*) ataca um alvo, esse tem a possibilidade de ser um servidor, um site ou uma empresa servidora de nuvem. Derivado do DoS, o *Distributed Denial-of-Service* (DDoS), em português, Negação de Serviço Distribuída, envolve diversas origens do ataque mantendo o foco em apenas um alvo, como demonstrado na Figura 15. O atacante controla diversos computadores ao redor do mundo, chamados de zumbis pela comunidade tecnológica. Isto deve-se ao fato de que os proprietários desses computadores não possuem o conhecimento de que sua máquina está infectada por um *malware* e está sendo controlada remotamente para efetuar o ataque. Um *malware* é qualquer *software* intencionalmente projetado para prejudicar ou explorar qualquer computador, servidor, ou rede de computadores. (MALWARE, 2021).

Figura 15 – Diagrama demonstrativo de como um ataque DDoS ocorre

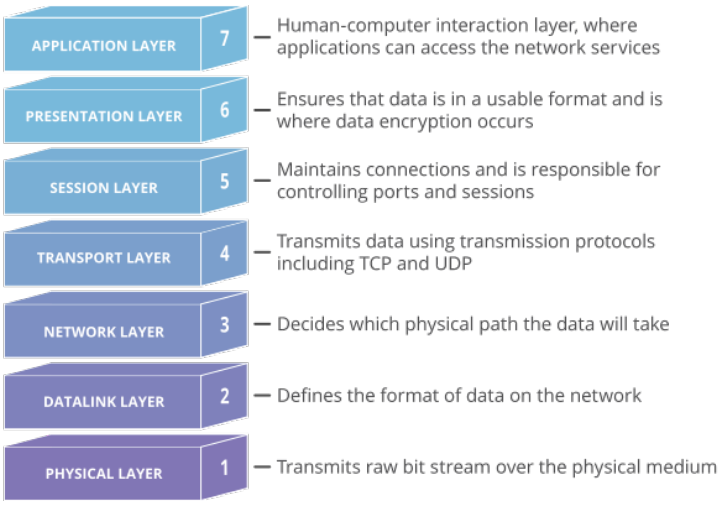



Fonte: (BIT2ME, 2022).

Diferente da maioria dos ataques conhecidos, os de negação de serviço não corrompem arquivos ou envolvem roubo de dados. O objetivo é sobrecarregar e esgotar recursos computacionais (como memória e processamento) e tornar o atacado indisponível para acessos comuns. Em geral a finalidade do ataque é causar danos financeiros ou ter um cunho político ou social. No caso de sucesso, dependendo da grandeza e do foco, esses danos podem ser milionários ou podem causar uma propaganda negativa, se a vítima for uma empresa provedora de nuvem. A grande dificuldade em mitigar o ataque é o fato de que enquanto há muitas solicitações atacantes, dentre elas, há inúmeros usuários comuns. Além de, comumente, o atacante utilizar os zumbis para efetuar o ataque, assim, não utilizando da própria máquina para a comunicação direta com o alvo.

Há diversas formas de aplicar o ataque, para classificá-los de uma forma melhor, em geral, utiliza-se o modelo *Open Systems Interconnection* (OSI). Na Figura 16 encontra-se uma descrição de cada camada do modelo.

Figura 16 – Diagrama descritivo do modelo *Open Systems Interconnection* (OSI)




Fonte: (FARE, 2022).

Embora exista uma segregação na definição de alguns tipos de ataques em relação ao modelo OSI, há também os que envolvem camadas em conjunto. Por exemplo, ataques nas camadas 3 e 4 normalmente são classificados como ataques da camada Infraestrutura, enquanto nas camadas 6 e 7, de Aplicação. No caso da primeira, a explicação é de que a utilização dos protocolos de camada 3 juntamente com os de transporte de camada 4 garantem o êxito no envio dos dados. (FARE, 2022). Esses ataques geralmente são grandes em volume e visam sobrecarregar a capacidade da rede ou dos servidores de aplicativos. Mas, felizmente, também são tipos de ataques que possuem assinaturas claras e são mais fáceis de detectar. (AMAZON, 2022).

Na camada de aplicação os ataques tendem a ser mais específicos e, por consequência,

mais sofisticados e menos comuns. Esses tendem a focar em partes específicas do *software*, fazendo com que não precisem de tanto volume quanto os de infraestrutura. Por exemplo, uma inundação de solicitações HTTP para uma página de login. (AMAZON, 2022).

# 3 METODOLOGIA

Para o desenvolvimento do projeto foram utilizados recursos próprios da autora, dentre eles, um computador de trabalho com sistema operacional *Windows* 10, processador *Intel*(R) *Core*(TM) i5-8265U CPU @ 1.60GHz, 1800 Mhz, 4 núcleos, 8 processadores lógicos e memória RAM de 16GB. Ainda, *Kubernetes* na versão 4.5.4 e *Docker* na versão 20.10.14. Também foram utilizados recursos do *datacenter* do laboratório NERDS (*Software Defined Networks Research Group*), especificamente um servidor *Dell* R230 (CPU E3-1240 v6 @ 3.70GHz, 8 núcleos e RAM 32 GB) rodando *Kubernetes* 4.5.4 sobre *Ubuntu Server* 22.04.

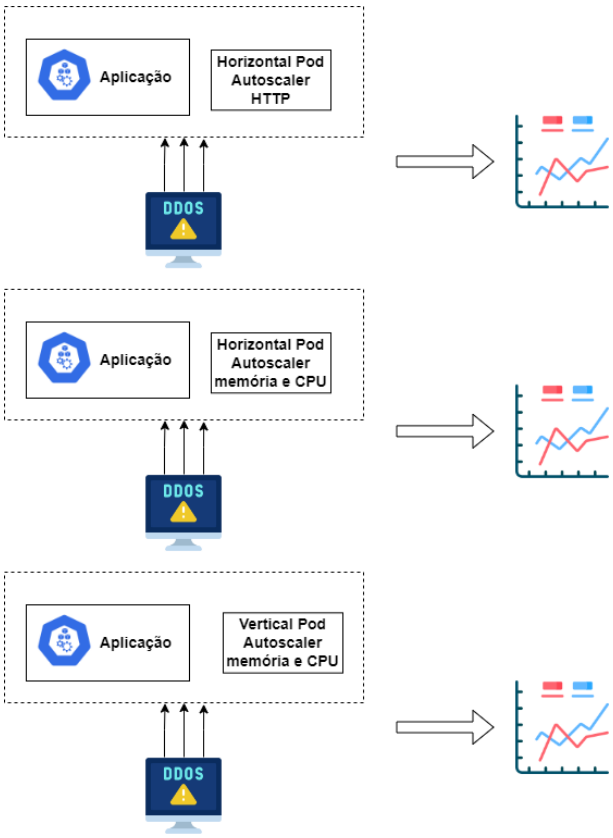
À princípio, foi necessária a realização de estudos acerca dos assuntos e tecnologias retratadas neste trabalho. As tecnologias de conteinerização e virtualização, arquitetura para aplicações baseada em microsserviços e os principais ataques dirigidos a aplicações baseadas em microsserviços.

Por questões de familiaridade da autora, a linguagem escolhida para o desenvolvimento do microsserviço do *Back-End* (serviço interno da aplicação que se comunica com o banco de dados) foi o *Node.js*[1] e para o *Front-End* (interface gráfica) foi o *framework React*[2]. O banco de dados foi o *postgreSQL*[3] por ser um banco SQL com boas referências a se utilizar com o *ubuntu*. O *Docker*[4] e o *Kubernetes*[5] foram escolhidos por terem um grande suporte na comunidade. E as demais tecnologias e bibliotecas foram selecionadas a critério da própria autora, sempre observando quais estão em destaque na comunidade.

Para os experimentos, primeiramente, foram criadas as imagens e lançadas ao *Docker Hub*. Posteriormente, foram configurados os ambientes no *Kubernetes* utilizando como base as imagens. E, por fim, foram realizados os experimentos. Foram sugeridos três opções de escalonadores a se utilizar junto à aplicação sob ataque e duas opções de ataque DDoS. Como demonstrado na Figura 17 para cada ataque.

[1] https://nodejs.dev/
[2] https://pt-br.reactjs.org/
[3] https://www.postgresql.org/
[4] https://www.docker.com/
[5] https://kubernetes.io/

Figura 17 – Diagrama demonstrativo a cerca do *setup* de experimentos projetado




Fonte: Desenvolvido pelo próprio autor.

## 3.1 Criação das imagens

### 3.1.1 Configuração dos microsserviços

O *Front-End* foi desenvolvido utilizando *React* na versão 18.1.0 e exposto na porta padrão (3000). Para a integração com a API, foi escolhido o *axios*[6]. O *Back-End* foi desenvolvido utilizando a linguagem de programação *Node.js* na versão 16.14.2. Para o desenvolvimento e configuração da API REST, foi aplicada a biblioteca *express*[7] e complementares. O microsserviço também possui autenticação de usuário, para tal foi utilizada a biblioteca *jsonwebtoken*[8]. Este módulo fornece *middleware express* para validação de JSON *Web Tokens* (JWTs). *Middleware* é um *software* que faz a mediação entre tecnologias, agindo como uma camada de comunicação dentre elas. E, por fim, o servidor foi disponibilizado na porta TCP 8080.

[6] https://www.npmjs.com/package/axios
[7] https://www.npmjs.com/package/express
[8] https://www.npmjs.com/package/jsonwebtoken

Uma forma interna de auxiliar na mitigação do ataque do tipo DDoS foi criar um limite de solicitações para cada IP, não há uma medida ideal para todas as aplicações e, há possibilidade de uma aplicação possuir diferentes valores ideais tal qual para cada rota. Para a definição da taxa é necessário realizar um estudo da aplicação em seu funcionamento normal, para que a taxa escolhida não limite usuários reais.

Para o experimento, os valores foram escolhidos a fim de limitar, mas não reprimir completamente os ataques realizados nos experimentos. Isto deve-se ao fato de todo o experimento ter sido realizado em apenas uma máquina física. Ou seja, com limitações de ataque. O código abaixo descreve a forma como foi feito.

```
const rateLimit = require("express-rate-limit");
const limiter = rateLimit({
    windowMs: 5 * 60 * 1000,
    max: 1000,
    standardHeaders: true,
legacyHeaders: false,
});
```

Posto que "`standardHeaders: true`" significa que a informação do limite da taxa de retorno estará no cabeçalho "`RateLimit`" e "`legacyHeaders`" desativa o cabeçalho "`X-RateLimit`". Vale ressaltar que a utilização dessa biblioteca não afeta o uso de escalonadores que será demonstrado adiante, o conteúdo exposto acima é apenas uma sugestão paralela.

Como o foco não era criar vários microsserviços, no banco de dados, foram criadas apenas tabelas para auxiliar na autenticação do usuário e uma tabela chamada *users* para simular possíveis dados pessoais de usuários. A tabela *users* está exposta na figura 18.

Figura 18 – Tabela *users* do banco de dados *PostgreSQL*

| | id [PK] integer | username text | user_password text | age integer | address text | cpf text |
|---|---|---|---|---|---|---|
| 1 | 20 | Laila | 873BHhdbmw7BE6l6W95S6... | 98 | Av. Fernando Ferrari, 51... | 123.456.789-10 |
| 2 | 18 | Fabricio | U61uN2mlq3Wo4lu7E8co3j... | 41 | Av. Fernando Ferrari, 51... | 123.456.789-10 |
| 3 | 17 | Ana | U2FsdGVkX1+UjDe5LpFuA3... | 6 | Av. Fernando Ferrari, 51... | 123.456.789-10 |
| 4 | 16 | Barbara | 5AF31Nçmcd45a32g132nA... | 7 | Av. Fernando Ferrari, 51... | 123.456.789-10 |

Fonte: Desenvolvido pelo próprio autor.

Para finalizar a construção da aplicação que sofrerá os ataques, foi criado uma imagem *Docker* de cada microsserviço para ser utilizada na criação do *Pod* do *Kubernetes*. No caso do *Front-End* e *Back-End* foram criados a partir de um *Dockerfile* (como o código abaixo).

```
FROM node:lts-alpine
ENV NODE_ENV=production
WORKDIR /usr/src/app
COPY . .
EXPOSE 3000
RUN chown -R node /usr/src/app
USER node
ENTRYPOINT npm start
```

Seguindo o *script* acima, sabe-se que a imagem do *Front-End* foi criada a partir de uma imagem *node* versão *lts-alpine*, que foi definido uma variável de ambiente, o diretório de trabalho, após foi copiado todo o diretório local para o diretório interno, a porta 3000 foi exposta, foram dadas permissões para o usuário *node*, foi definido o *node* como usuário de acesso e, por fim, ao iniciar o contêiner, o comando para iniciar a aplicação é executado. Vale ressaltar que o diretório foi completamente copiado porque era um pequeno serviço, logo, a criação foi rápida, mesmo copiando o `node_modules`. A imagem do *Back-End* é exatamente igual, exceto pelo fato de a porta ser a 8080.

A imagem do banco de dados foi criada a partir do comando `docker commit`. Utilizando uma imagem *postgre* na versão *alpine*, foi criado um contêiner, nele foi criado o banco de dados `database_pg`, suas tabelas e um usuário com permissões apenas de leitura. Após toda a configuração, foi realizado um `docker commit` do contêiner, criando a imagem.

### 3.1.2 Configuração dos ataques

Os ataques utilizados não foram desenvolvidos pela autora do trabalho, eles foram escolhidos em repositórios públicos com o foco na camada 7, ataques volumétricos DDoS desenvolvidos para sobrecarregar um servidor visado com solicitações HTTP. (CLOUDFARE, 2022). Os repositórios escolhidos foram o *DDoS-Ripper*[9] e o *Raven-Storm*[10]. Os dois ataques escolhidos foram escritos na linguagem de programação Python[11].

Enquanto o ataque *DDoS-Ripper* foi projetado para executar ataques de nível de aplicação, de caminho direto e indireto, em servidores da *Web*, o *Raven-Storm* foi projetado apenas para o caminho direto.

[9] https://github.com/palahsu/DDoS-Ripper
[10] https://github.com/Tmpertor/Raven-Storm
[11] https://www.python.org/

O *DDoS-Ripper* possui uma limitação no número de *threads*, as opções são 135 e 443, no caso deste projeto foi optado por 443. O ataque foi elaborado para executar dois vetores de ataque simultâneos, cada um consistindo em 443 *threads* (ou 135) independentes que carregam o servidor de destino com solicitações HTTP. O primeiro vetor de ataque é uma solicitação HTTP GET de caminho direto para o IP de destino, aproveitando um cabeçalho de agente de usuário aleatório escolhido de uma lista predefinida e um conjunto estático de cabeçalhos importados de uma lista definida no próprio código. O segundo modo de enviar requisições, chamado de caminho indireto, cria 443 (ou 135) tópicos que resultam em solicitações escolhidas aleatoriamente entre o site de validação de marcação *w3.org*[12] e no de compartilhamento do *Facebook*[13]. (RADWARE, 2022).

O *Raven-Storm* não possui limitação no número de *threads* e o valor utilizado foi o padrão definido pela aplicação 400. Esse, como dito anteriormente, executa ataques de nível de aplicativo de caminho direto. O ataque envia requisições URL para o servidor utilizando o cabeçalho a seguir.

```
{'User-Agent': choice(var.user_agents), "Connection": "keep-alive",
"Accept-Encoding": "gzip, deflate", "Keep-Alive": randint(110,120)}
```

Sendo que, `randint(110,120)` é uma função que resulta em um valor inteiro aleatório entre 110 e 120 e `choice(var.user_agents)` é uma função que resulta em um agente aleatório de uma lista definida no próprio código, tal como do *DDoS-Ripper*.

As imagens a serem utilizadas no *Kubernetes* foram criadas a partir da imagem oficial *Docker* do *ubuntu* em sua última versão. Foram instaladas as bibliotecas necessárias e clonado o repositório do *github*. Após toda a configuração, foi realizado um `docker commit` de cada contêiner, criando as duas imagens.

## 3.2 Configuração do ambiente no *Kubernetes*

Através do uso das imagens criadas do *Front-End*, *Back-End* e do banco de dados foi possível iniciar o ambiente para os experimentos. E, a partir das imagens criadas para os ataques foi possível configurar os *setups* atacantes. As duas seções abaixo descrevem com detalhes como foi feito a configuração.

[12] https://validator.w3.org/check?uri=<target server>
[13] https://www.facebook.com/sharer/sharer.php?u=<Ctarget server>|

### 3.2.1 Configuração dos microsserviços

Para a criação de todos os componentes no *Kubernetes* foram utilizados *scripts* YAMLs. Abaixo tem-se o utilizado para a criação do *Front-End*. (TENNAKOON, 2021). Em resumo, com este *script* foi criado um *Deployment* com apenas um *Pod* (`replicas: 1`), o nome dado foi `pg-front-day`, foi criado a partir da imagem `dayerlacher/projeto-graduacao:front`, especificada a porta `3000` e os limites de recursos `3Gi` de memória e `2` de CPU.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pg-front-day
spec:
  selector:
    matchLabels:
      run: pg-front-day
  replicas: 1
  template:
    metadata:
      labels:
        run: pg-front-day
    spec:
      containers:
      - name: pg-front-day
        image: dayerlacher/projeto-graduacao:front
        ports:
        - containerPort: 3000
        resources:
          limits:
            memory: 3Gi
            cpu: 2
          requests:
            memory: 70Mi
            cpu: "70m"
```

Para a criação do serviço, foi desenvolvido outro *script*. No qual foi nomeado como `pg-front-day` e exposta a porta `3000`.

```
apiVersion: v1
kind: Service
metadata:
  name: pg-front-day
  labels:
    run: pg-front-day
spec:
  ports:
  - port: 3000
  selector:
    run: pg-front-day
```

O *Back-End* foi criado da mesma maneira que o *Front-End*, o nome dado foi `pg-back-day` e a porta a 8080. Para a criação do banco de dados no ambiente do *Kubernetes*, além de criar o *Deployment* e o serviço, foram criados os volumes. O *Persistent Volume Claim* foi nomeado como `postgres-pvc` e requisitado `1Gi` de armazenamento. Enquanto, o *Persistent Volume* foi nomeado como `local-storage` e requisitado também `1Gi` de armazenamento. (ROSA, 2020).

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-storage
spec:
  storageClassName: manual
  capacity:
```

```
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/minikube/directory/structure/"
    type: DirectoryOrCreate
```

A criação do *Deployment* e do serviço seguem o mesmo formato anterior, exceto pelo fato de que neste caso há a necessidade de vincular o volume criado. Disponível para visualização no Apêndice A.

As portas mencionadas foram definidas sem critérios específicos e poderiam ser quaisquer outras que não estivessem sendo utilizada, da mesma forma que os nomes dados. A capacidade definida para o volume foi o padrão encontrado no tutorial do Rosa (2020). E a capacidade do *Deployment* foi escolhida pela própria autora e definida por meio de alguns experimentos iniciais, realizados antes de definir os parâmetros de ataque. No entanto, poderiam ser quaisquer outros.

### 3.2.2 Configuração dos ataques

Para fins de entendimento, vale ressaltar que no ambiente do *Kubernetes* a nomenclatura `Mi` relacionado a memória se refere a `MiB` e `1000m` de CPU é o mesmo que 1 CPU, que externamente equivale a:

- 1 vCPU AWS;
- 1 núcleo do GCP;
- 1 Azure vCore;
- 1 *Hyperthread* em um processador Intel bare-metal com *Hyperthreading*;

A arquitetura necessária na configuração dos ataques não necessitava ser grande. À vista disso, foram criados dois *Pods*, um para cada. O formato do YAML dos dois *Pods* são idênticos. No exemplo abaixo, tem-se que o nome escolhido foi `ddosripper` e a imagem base é `dayerlacher/projeto-graduacao:ripper`. Enquanto, no outro caso, o nome foi `ddosraven` e a imagem base `dayerlacher/projeto-graduacao:raven`.

```
apiVersion: v1
kind: Pod
metadata:
  name: ddosripper
  labels:
    app: ddosripper
spec:
  containers:
  - image: dayerlacher/projeto-graduacao:ripper
    command:
      - "sleep"
      - "604800"
    imagePullPolicy: IfNotPresent
    name: ddosripper
  restartPolicy: Always
```

Os nomes mencionados foram escolhidos sem critérios específicos e poderiam ser quaisquer outros. Os nomes das imagens foram os nomes dados quando elas foram enviadas para o *Docker Hub*. O comando `sleep` foi utilizado porque a base da imagem do ataque é o *ubuntu* e ao iniciar o ambiente é necessário manter uma seção interativa. Portanto, esse é um tempo suficiente para que o *Pod* não seja encerrado antes que a seção seja aberta.

## 3.3 Configuração dos escalonadores no *Kubernetes*

Neste trabalho foram desenvolvidas três opções de escalonamento na tentativa de mitigar os ataques DDoS: o HPA utilizando número de solicitações de acesso HTTP e os dois modos disponíveis (HPA e VPA) utilizando as métricas CPU e memória. As próximas seções descrevem com detalhes como foram configurados.

### 3.3.1 *Horizontal Pod Autoscaling* (HPA) utilizando número de solicitações de acesso HTTP

Para viabilizar o contador, no microsserviço do *Back-End*, foi importado a biblioteca `pro-client`[14] e criado o `Counter` do *Prometheus* no modo padrão (visto abaixo).

[14] https://www.npmjs.com/package/prom-client

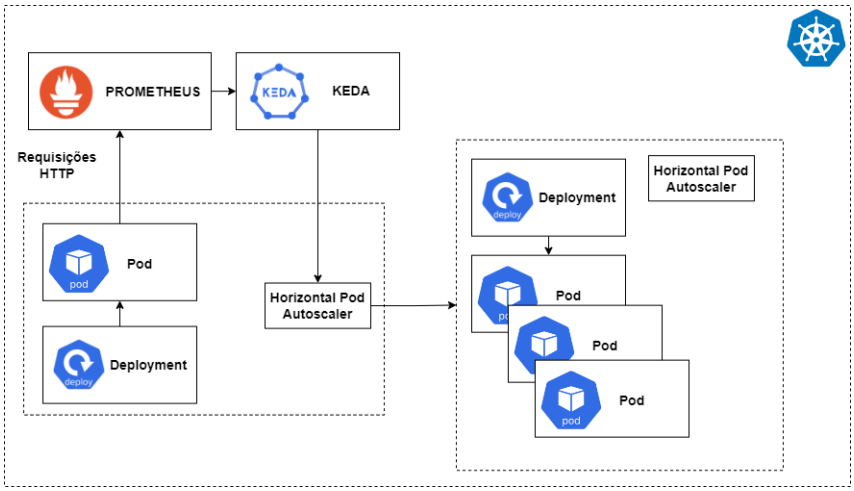
```
const Prometheus = require('prom-client');
const httpRequestsCounter = new Prometheus.Counter({
    name: "http_requests",
    help: "number of http requests",
})
```

Para a visualização do valor totalizado do contador, foi criado um *endpoint*, também no formato padrão, visualizado abaixo.

```
app.get('/metrics', async (req, res) => {
    res.set('Content-Type', Prometheus.register.contentType)
    res.end(await Prometheus.register.metrics())
})
```

O funcionamento desejado do módulo 1 é demonstrado na figura 19. O aplicativo expõe as métricas no formato *Prometheus*. O KEDA por meio de seus recursos e integração com o HPA, dimensiona automaticamente o aplicativo com base nas métricas de contagem de acesso HTTP.

Figura 19 – Diagrama demonstrativo a cerca da visão geral do módulo 1




Fonte: Desenvolvido pelo próprio autor.

Dentre os arquivos YAML para a configuração do *Prometheus*, foram criados: *ClusterRole*, *ServiceAccount*, *ClusterRoleBinding*, *ConfigMap*, *Deployment* e um *Service*. Todas as declarações estão no Apêndice B. Os três primeiros listados são utilizados para permitir que o módulo funcione corretamente, são dadas as permissões necessárias e é definido um nome para o *ServiceAccount*. Ademais, o *ConfigMap* define configurações do *Prometheus* e

foram vinculados volumes ao *Deployment*. Tanto para este quanto para o serviço, o nome escolhido foi *Prometheus* e a porta de acesso é a 9090.

O *ScaledObject* aplicado no projeto encontra-se no Apêndice C. Vale ressaltar algumas características, tais como, o nome dado foi `prometheus-scaledobject`, o alvo é o *Deployment* `pg-front-day`, ou seja, o *Front-End*. O KEDA pesquisará o alvo a cada 15 segundos (`pollingInterval`), o mínimo de réplicas definido foi 1 e o máximo 10. Ademais, o gatilho definido foi o *Prometheus*, a métrica o `access_frequency` e a consulta PromQL foi definida como `sum(rate(http_requests[2m]))`. Todos os valores foram utilizados com base no tutorial que foi utilizado como referência e sem requisitos bem definidos.

A consulta especificada retorna o valor agregado da taxa por segundo de solicitações HTTP, conforme medido nos últimos dois minutos. Ou seja, a cada 15 segundos é calculado a média das requisições nos últimos 2 minutos. O *threshold* (limite, em tradução livre) foi definido como 3. Isso significa que o passo é 3, ou seja, se o valor retornado é menor que 3, o número de *Pods* será o mínimo definido. E haverá um acrescimento no número de *Pods* a cada múltiplo de 3, exceto pelo primeiro múltiplo. Exemplificando, se o retorno da PromQL estiver entre, 12 e 14, o número de *Pods* será 4. (GUPTA, 2019).

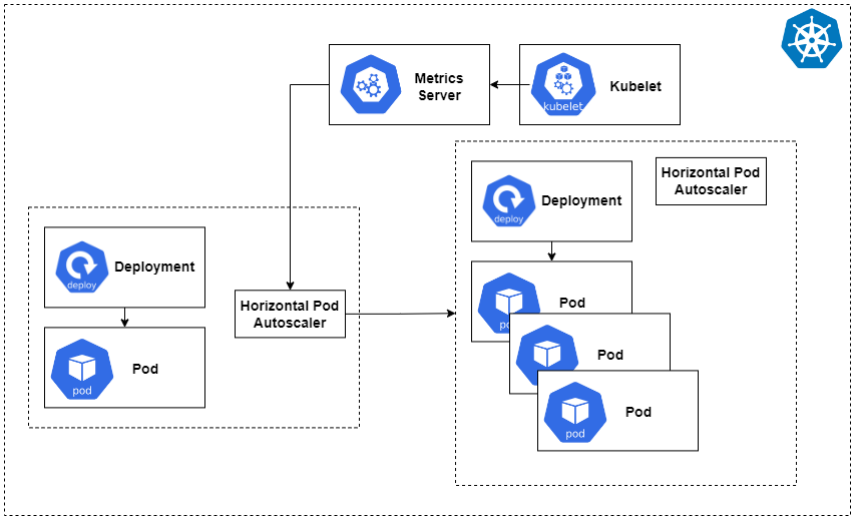
O módulo 1 demonstrado nessa seção é baseado no tutorial desenvolvido por Gupta (2019).

### 3.3.2 *Horizontal Pod Autoscaling* (HPA) utilizando as métricas memória e CPU

O módulo 2 foi desenvolvido com o propósito de através das métricas internas do *Pod* e suas réplicas, redimensionar horizontalmente o sistema. Diferente do módulo anterior, a configuração realizada quando se trata das métricas do servidor foi bem menor. Como as métricas pertencem ao próprio servidor, não é necessário o uso de módulo externo ou banco de dados. A Figura 20 demonstra de maneira explícita o funcionamento.

Há um recurso chamado `HorizontalPodAutoscaler` com diversas configurações que possibilitam o objetivo buscado. O arquivo em formato YAML encontra-se no Apêndice D. Os pontos de destaques são: o nome dado ao recurso criado foi `hpa-cpu`, o alvo foi o *Deployment* `pg-front-day`, o número mínimo de réplicas foi definido como 1 e o máximo como 5. Quanto aos parâmetros escolhidos para as métricas, foi escolhido `70m` para CPU e `120Mi` para memória. Ou seja, esse HPA tinha dois gatilhos e o primeiro a ser disparado desencadeava a criação de réplica. Os valores foram definidos por meio de alguns experimentos iniciais. No entanto, poderiam ser quaisquer outros.

Figura 20 – Diagrama demonstrativo acerca da visão geral do módulo 2




Fonte: Desenvolvido pelo próprio autor.

Ao definir valores muito pequenos, os gatilhos são acionados mais rapidamente e ao definir parâmetros maiores, eles são acionados posteriormente. Neste caso, para definir as métricas, depende do contexto e do limite disponível de métricas. Vale ressaltar que no caso de métricas muito baixas, o *Pod* não se inicia com o consumo de métricas zerado, logo, caso o limite definido seja muito baixo, a criação de um *Pod* pode ocasionar na criação de um segundo.

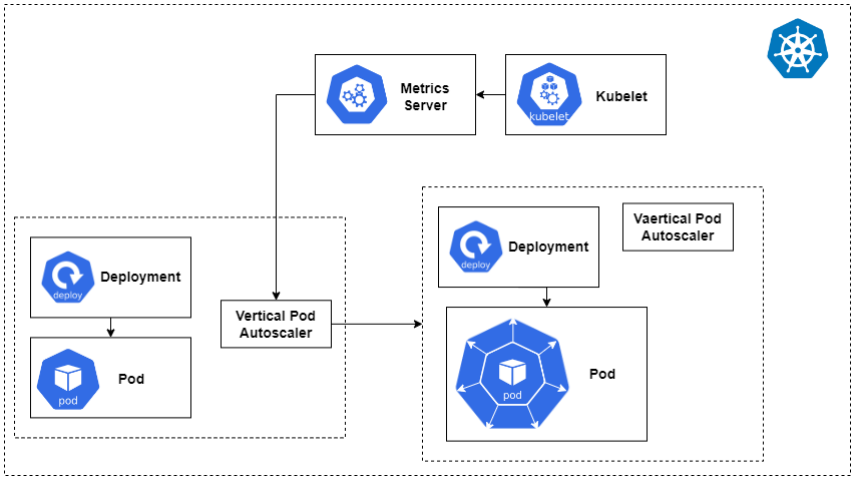
O módulo 2 demonstrado nessa seção é baseado no tutorial desenvolvido por Jakkula (2020).

### 3.3.3 *Vertical Pod Autoscaling* (VPA) utilizando as métricas memória e CPU

O módulo 3 foi desenvolvido com o propósito de através das métricas internas do *Pod* e suas réplicas, redimensionar verticalmente o sistema, ou seja, aumentar as métricas do *Pod*. Semelhante ao anterior, como as métricas pertencem ao próprio servidor, não é necessário o uso de módulo externo ou banco de dados. A figura 21 demonstra explicitamente o funcionamento.

Há um recurso chamado `VerticalPodAutoscaler` com diversas configurações que possibilitam o objetivo buscado. O arquivo em formato YAML encontra-se no Apêndice E. Vale ressaltar que o nome dado ao recurso criado foi `vpa`, o alvo foi o *Deployment* `pg-front-day`, os limites máximos de CPU e memória foram definidos como `4` e `5Gi`, respectivamente. E os parâmetros iniciais são `70m` de CPU e `70Mi` de memória. Os valores foram definidos por meio de alguns experimentos iniciais. No entanto, poderiam ser quaisquer outros.

Figura 21 – Diagrama demonstrativo acerca da visão geral do módulo 3




Fonte: Desenvolvido pelo próprio autor.

## 3.4 Monitoramento com *prometheus* e *grafana*

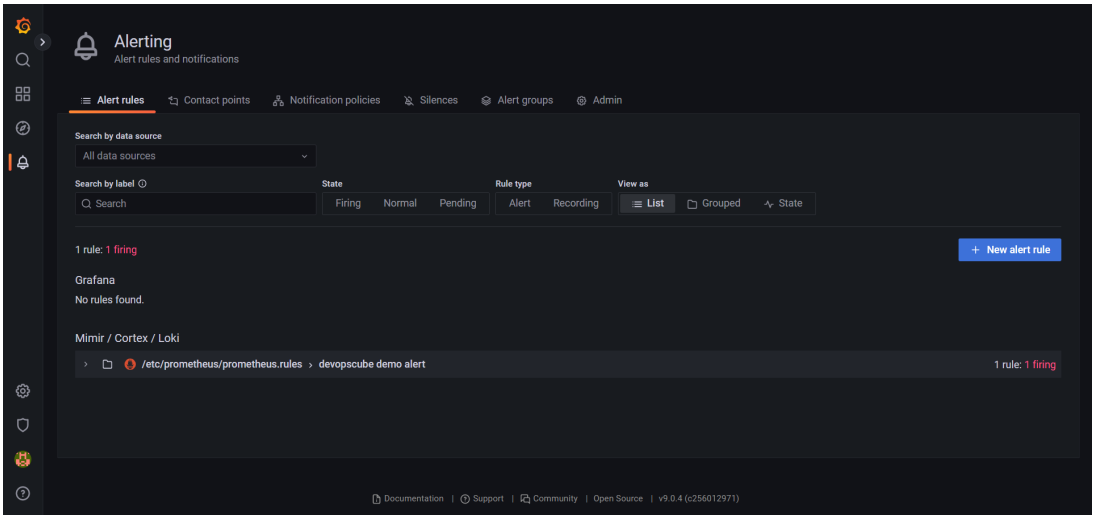
Há também a possibilidade de monitoramento do *Kubernetes*, essa opção não influencia no ataque, mas concede a oportunidade de análise geral e a possibilidade de descobrir algo incomum ocorrendo com a aplicação, tal como um ataque. Ao desfrutar da facilidade de integração entre o *Prometheus* e o *Grafana*, montamos um painel de monitoramento. *Grafana* é uma aplicação *web* gratuita utilizada para a visualização de dados e informações através de tabelas e gráficos variados. Essa se comunica com diversas possíveis fontes de dados, dentre elas, o *Prometheus* e o *InfluxDB*, um banco de dados de série temporais. Foi utilizado como base para a criação e configuração o tutorial desenvolvido por Alves (2020). A figura 22 expõe a visualização do painel.

Figura 22 – Captura da tela do *software Grafana*

Fonte: Desenvolvido pelo próprio autor.

Dentre as vantagens em se utilizar o *Grafana*, estão as funcionalidades de alerta. Facilmente é possível criar uma regra de alerta, definindo um conjunto de critérios de avaliação que determina se um alerta será acionado. "A regra consiste em uma ou mais consultas e expressões, uma condição, a frequência de avaliação e, opcionalmente, a duração na qual a condição é atendida."(GRAFANA, 2022). Há a possibilidade de criação de diversos alertas diferentes, não há limite. A figura 23 exibe como é a página de alertas.

Figura 23 – Captura da tela do *software Grafana* na página de alertas




Fonte: Desenvolvido pelo próprio autor.

# 4 RESULTADOS E ANÁLISES

Para preparar o ambiente de experimento foram executados os arquivos YAMLs descritos em metodologia, utilizando o comando padrão `kubectl`. Dessa forma, foram executados os arquivos da aplicação (*Back-End* e *Front-End*), banco de dados e os dois ataques. Vale ressaltar que o comando abaixo foi utilizado para criar todos os componentes através dos arquivos YAMLs descritos na metodologia.

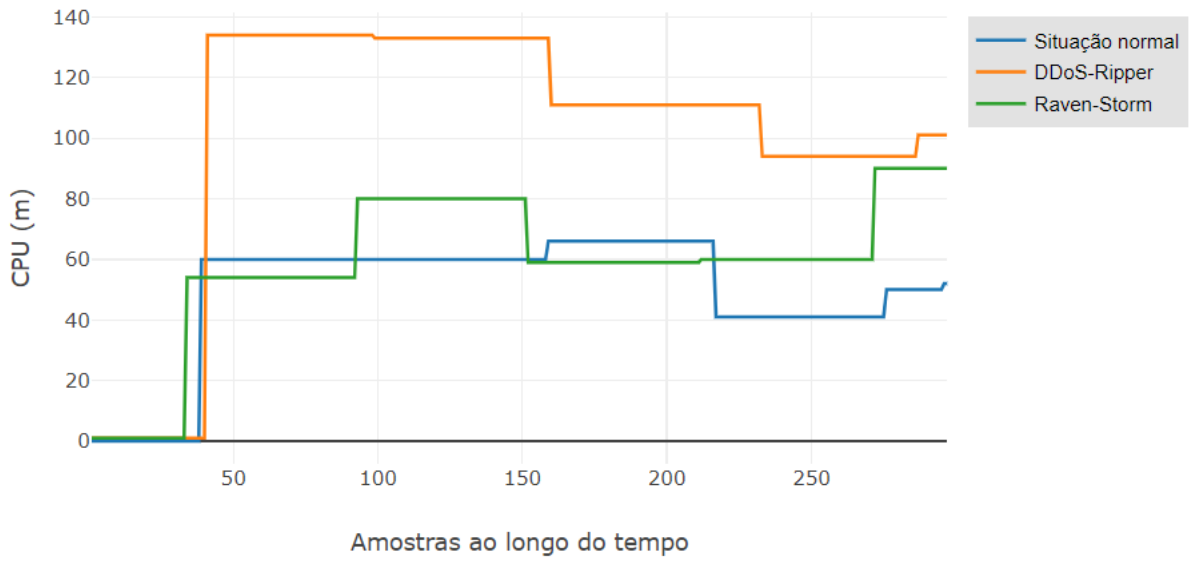
```
kubectl create -f arquivo.yaml
```

Para a realização dos experimentos, foi definido, por critérios da própria autora, o tempo de 5 minutos para cada, sendo que, todos os gráficos gerados nos experimentos começam juntamente com o ataque. Foi definido a coleta dos parâmetros de memória e de CPU a cada 1 segundo e a cada 0,1 segundos a coleta do tempo de resposta da aplicação a uma requisição. Esse tempo foi coletado a partir de requisições HTTP GET simulando usuários comuns da aplicação, realizando a autenticação do usuário e posteriormente a requisição. Todos os dados foram salvos em arquivos de texto. Os gráficos foram criados por meio da biblioteca *ploty*[1]. Foi optado por obter as métricas por meio de *scripts* desenvolvidos pela autora, por este motivo os gráficos não foram gerados pelo Grafana.

À principio foi realizado alguns experimentos preliminares com o objetivo de definir o consumo da aplicação em uma situação comum e nas duas situações de ataque. Isto porque a aplicação foi construída para este projeto, logo foi necessário descobrir o perfil de consumo. Os ataques não foram desenvolvidos pela autora, então precisaram ser testados com o intuito de obter também o perfil. Os perfis foram utilizados a fim de definir valores escolhidos de métricas. Para esses experimentos foram definidos valores altos de limites a fim de que a sobrecarga do *Pod* não fosse um problema. As figuras 24 e 25 representam, respectivamente, o comportamento da CPU e memória da aplicação no experimento inicial.

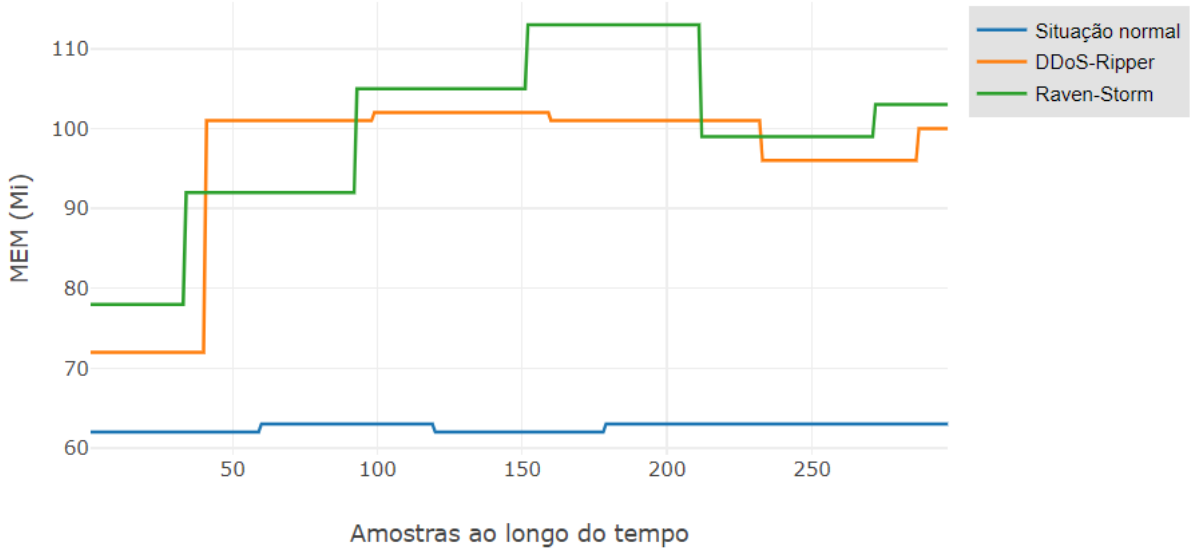
[1] https://plotly.com/javascript/

Figura 24 – Exposição do uso de CPU da aplicação nos experimentos sem escalonamento




Fonte: Desenvolvido pela própria autora.

Figura 25 – Exposição do uso de memória da aplicação nos experimentos sem escalonamento




Fonte: Desenvolvido pela própria autora.

Nessas figuras é possível visualizar a diferença entre os ataques, enquanto o ataque *DDoS-Ripper* consome mais CPU, o *Raven-Storm* consome um volume maior de memória na maior parte do tempo. Considerando tais cenários, a tabela 1 expõe as os tempos de resposta médios e máximos encontradas para as requisições HTTP.

Na Tabela 1 percebe-se que o ataque *DDoS-Ripper* possui o tempo médio quase 14 vezes maior se comparado a situação sem ataque, enquanto o *Raven-Storm* apenas 7 vezes.

Tabela 1 – Exposição dos valores médios e máximos de tempo de resposta dos experimentos no caso sem escalonamento

| # | Tempo médio de resposta [s] | Tempo máximo de resposta [s] |
|---|---|---|
| Situação sem ataque | 0,06 | 0,51 |
| Ataque *DDoS-Ripper* | 0,83 | 8,06 |
| Ataque *Raven-Storm* | 0,43 | 1,95 |

No caso do tempo máximo, essa diferença sobe para quase 16 vezes, ao passo que para o segundo ataque o valor se mantém 7 vezes maior. No contexto geral, é perceptível a potência do ataque *DDoS-Ripper* em relação ao segundo.

Vale ressaltar que, nas próximas figuras, as situações descritas como "Sem escalonamento" se referem aos casos sob ataque, mas em que a aplicação está superdimensionada. Isto porque ao definir os valores padrões como limites da aplicação e efetuar os ataques, os limites são ultrapassados e a aplicação torna-se indisponível. Portanto, a aplicação foi superdimensionada apenas para a obtenção dos dados considerados normais na situação de ataque, isso para avaliarmos os efeitos dos escalonamentos.

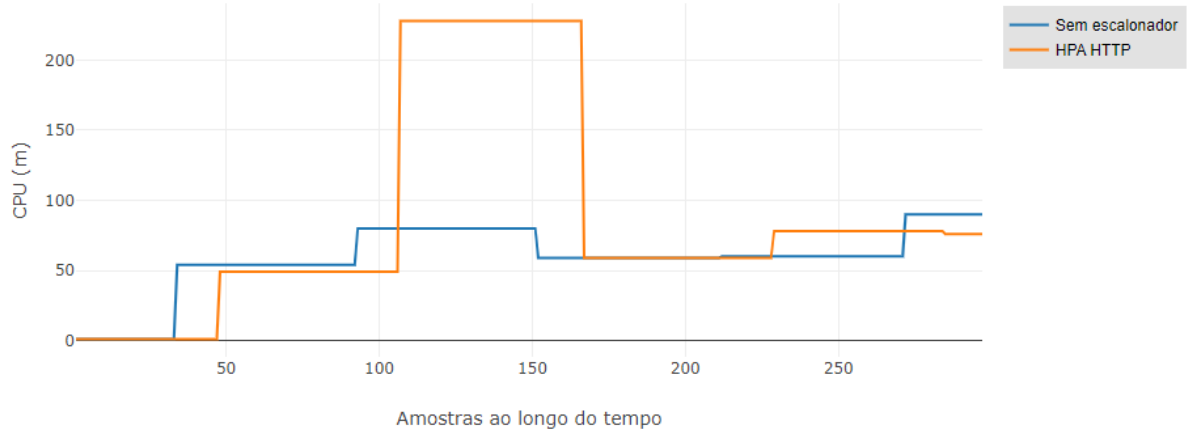
## 4.1 *Horizontal Pod Autoscaling* (HPA) utilizando número de solicitações de acesso HTTP

Com o propósito de tentar mitigar o ataque DDoS usando a própria infraestrutura de nuvem, o primeiro escalonador testado foi o HPA utilizando número de solicitações de acesso HTTP. O processo de configuração deste módulo foi iniciado instalando o KEDA (código abaixo). Posteriormente, foram executados os arquivos YAML do *Prometheus* e do *ScaledObject*.

```
kubectl apply -f https://github.com/kedacore/keda/releases/download/
v2.7.1/keda-2.7.1.yaml
```

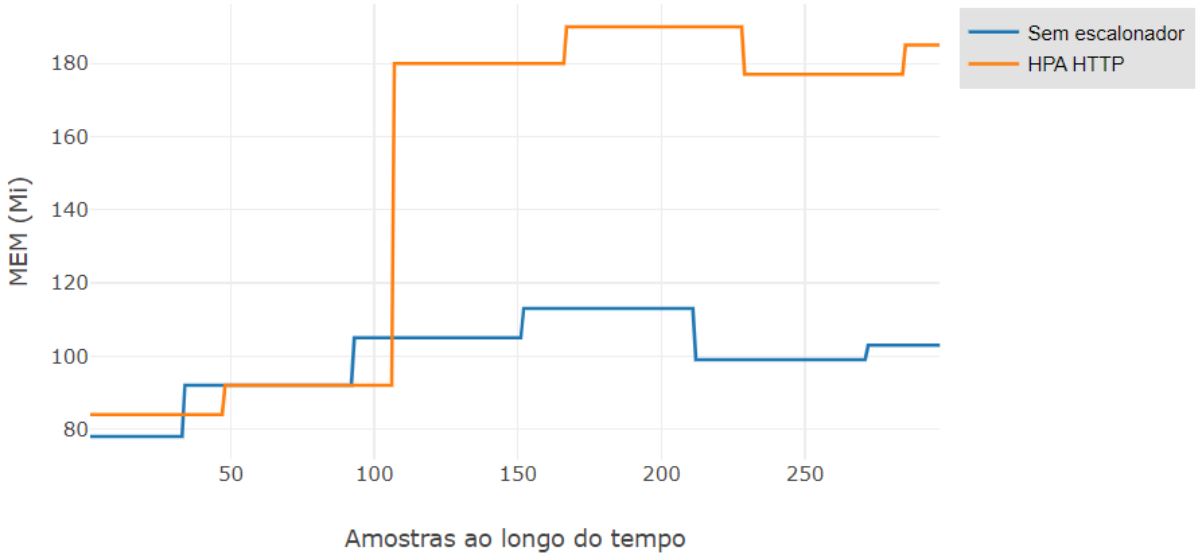
Após todo o ambiente ter sido configurado e executado, foi realizado o experimento apenas com o ataque *Raven-Storm*. Isto porque o contador foi desenvolvido com a finalidade de contabilizar as requisições válidas, ou seja, foi configurado apenas para caminhos criados e no caso do ataque *DDoS-Ripper* as solicitações GET são aleatórias. Nas figuras 26 e 27 encontram-se os resultados do experimento em comparação com os resultados do experimento do ataque sem o escalonamento.

Figura 26 – Exposição do uso de CPU da aplicação sob o ataque *Raven-Storm* utilizando HPA HTPP




Fonte: Desenvolvido pela própria autora.

Figura 27 – Exposição do uso de memória da aplicação sob o ataque *Raven-Storm* utilizando HPA HTPP



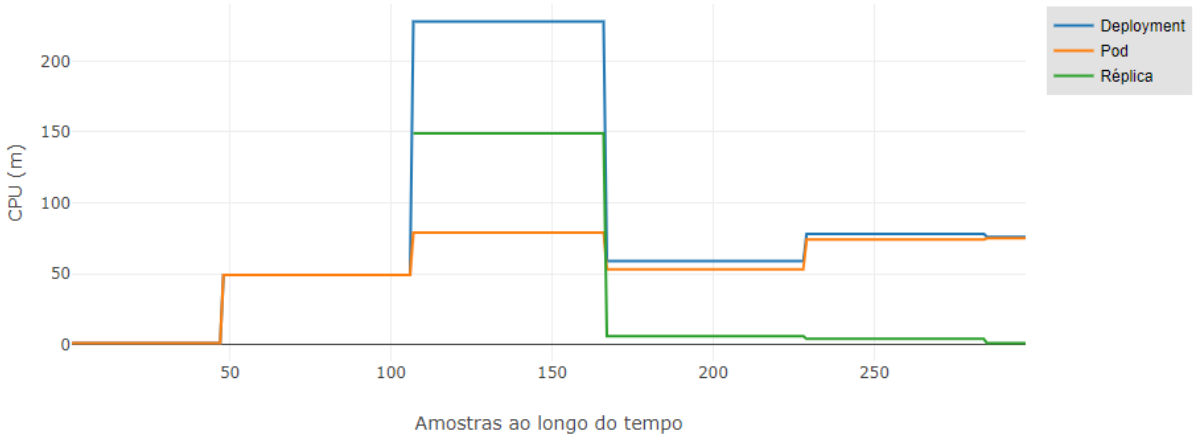

Fonte: Desenvolvido pela própria autora.

Analisando as figuras, nota-se que no que tange a CPU os valores finais da configuração com HPA HTTP foram um pouco menores se comparado a configuração sem escalonamento. Porém, no que tange a memória, a situação foi contrária e o valor se aproxima do dobro. Ainda, nota-se nas duas figuras um grande salto na utilização das métricas por volta da amostra 106, que é bem próximo de 2 minutos de experimento. Acredita-se que esse efeito foi visualizado porque a métrica utilizada para a avaliação envolvia a média das requisições nos últimos 2 minutos. Logo, anteriormente ao ataque, o número de solicitações era normal. Quando foi iniciado o ataque, começaram a subir drasticamente, mas a média, por ser calculada no período de 2 minutos, aumentou em um ritmo mais lento. Quão mais próximo

de 2 minutos de experimento, mais a média se tornava maior. Portanto, exatamente nesse momento, foi criado o segundo *Pod*.

No momento de criação de um *Pod* ele inicia-se com o consumo relativamente alto se comparado aos valores tratados neste trabalho, por ter uma arquitetura bem pequena. Na figura 28 é possível visualizar esse efeito. Vale ressaltar que nessa figura o "*Deployment*" se refere ao valor total (soma da métricas entre todas as réplicas, valor visualizado nas duas figuras anteriores), "*Pod*" se refere ao primeiro criado e, por fim, a "réplica" é a réplica criada.

Figura 28 – Avaliação do efeito da criação de um *Pod* sob o ataque *Raven-Storm*




Fonte: Desenvolvido pela própria autora.

Os valores médio e máximo de tempo de resposta encontrados foram 0,40 e 3,91 segundos, respectivamente. Enquanto os valores no caso sem escalonador eram 0,43 e 1,95 segundos. Os valores médios encontrados foram bem próximos, enquanto os máximos tiveram uma considerável diferença. No entanto, os maiores valores se referem ao momento próximo a criação do *Pod*. Se fosse excluído esse trecho de dados, os valores seriam 0,36 e 1,76 segundos. Evidenciando o efeito da criação do *Pod* na perspectiva externa.

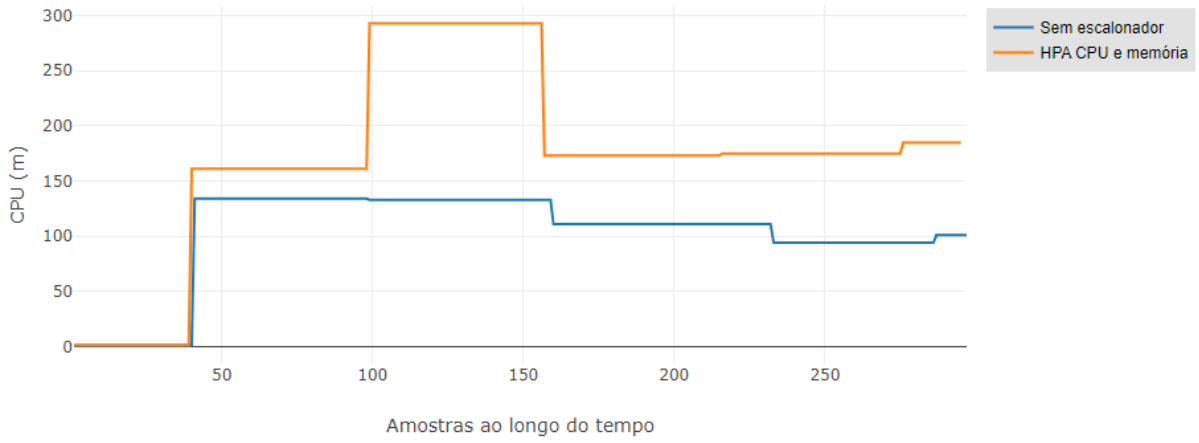
Vale ressaltar que em todo o projeto não foi abordado o balanceamento de cargas, assim sendo, foi utilizado o balanceamento padrão do *Kubernetes*.

## 4.2 *Horizontal Pod Autoscaling* (HPA) utilizando as métricas memória e CPU

Nesse experimento foi realizado o ataque *DDoS-Ripper* e, em seguida, o ataque *Raven-Storm*. Como citado anteriormente, esse experimento não necessitou de muitas configurações.

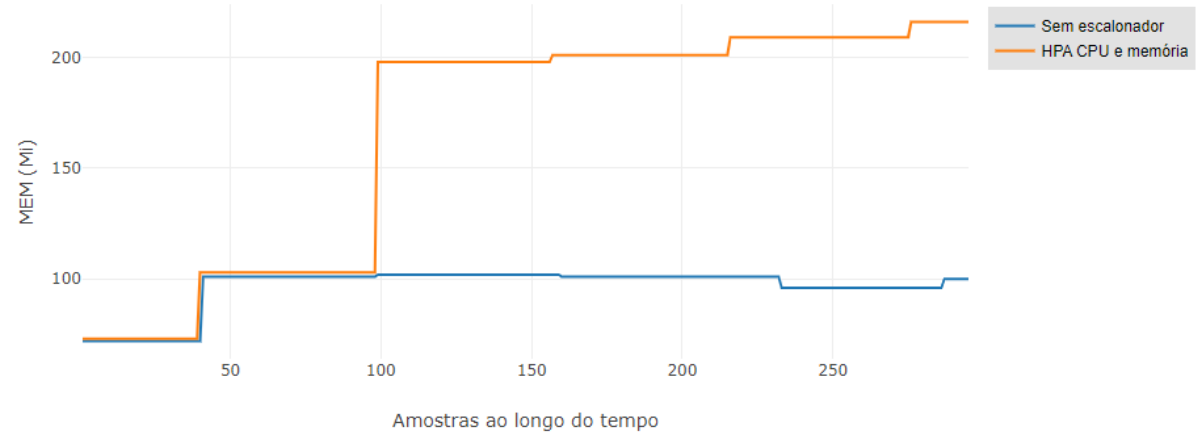
Para aprontá-lo bastou executar o comando apresentado anteriormente ao arquivo YAML (Apêndice D). Os resultados gerados no primeiro experimento se apresentam nas Figuras 29 e 30.

Figura 29 – Exposição do uso de CPU da aplicação sob o ataque *DDoS-Ripper* utilizando HPA métricas




Fonte: Desenvolvido pela própria autora.

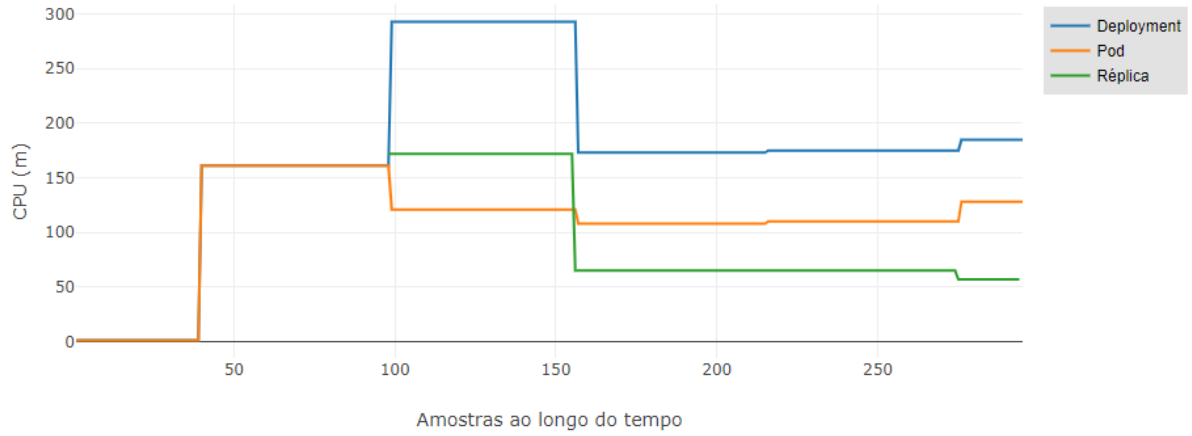
Figura 30 – Exposição do uso de memória da aplicação sob o ataque *DDoS-Ripper* utilizando HPA métricas




Fonte: Desenvolvido pela própria autora.

Analisando as figuras, nota-se que após a criação do *Pod* a memória não reduziu e as duas métricas se mantiveram por volta do dobro da referência. Ainda, nota-se um grande salto na utilização das métricas por volta da amostra 89. Como explicitado abaixo (Fig. 31), esse foi exatamente o momento em que o segundo *Pod* foi criado.

Figura 31 – Avaliação do efeito da criação de um *Pod* sob o ataque *DDoS-Ripper*

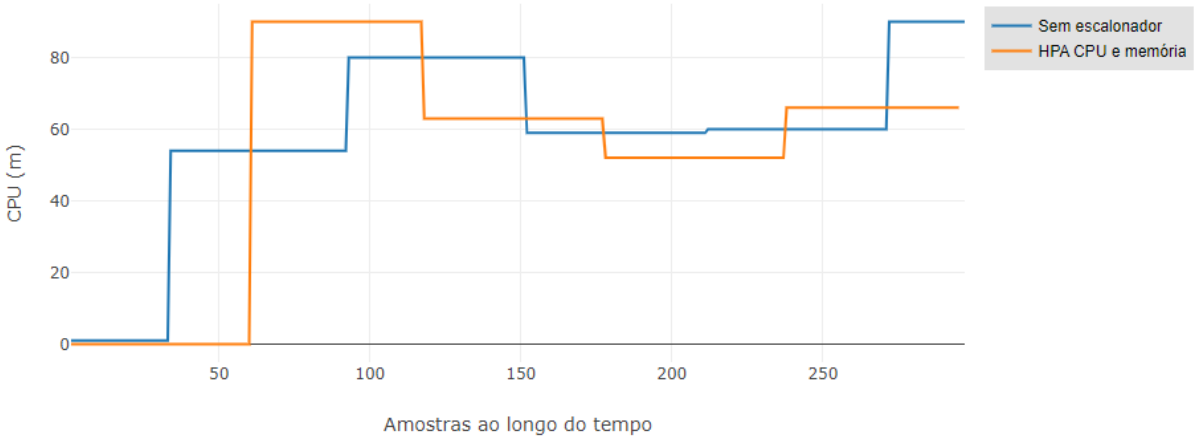



Fonte: Desenvolvido pela própria autora.

Os tempos computados foram 0,29 e 1,82 segundos, médio e máximo, respectivamente. Enquanto na situação sem escalonamento, foram 0,83 e 8,06 segundos. Ou seja, mesmo que as métricas internas estejam dobradas, nas externas houveram uma redução.

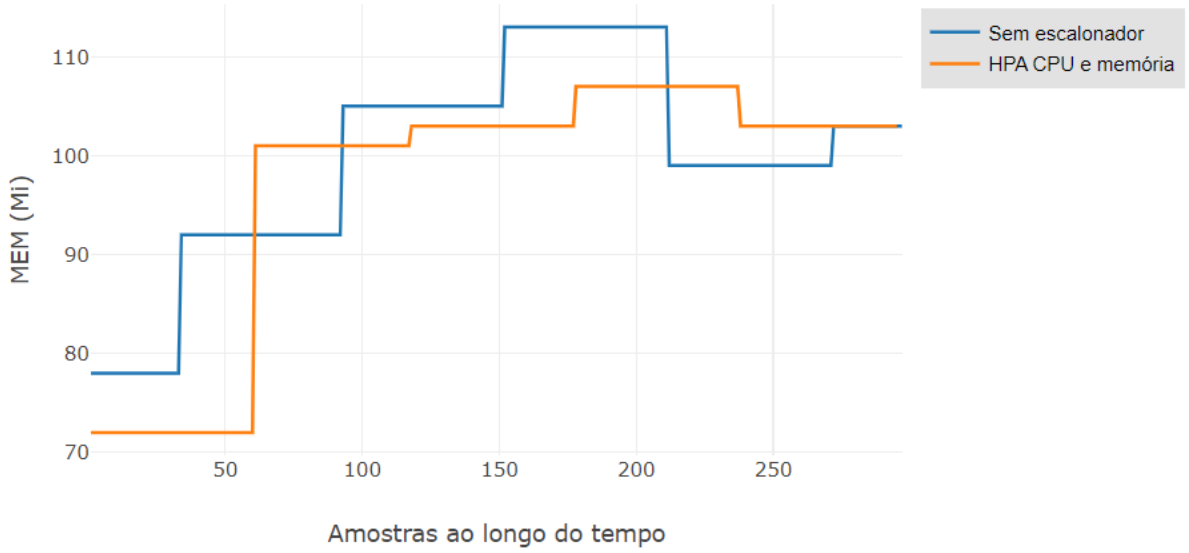
Após, foi realizado o ataque *Raven-Storm* (Fig. 32 e 33).

Figura 32 – Exposição do uso de CPU da aplicação sob o ataque *Raven-Storm* utilizando HPA métricas




Fonte: Desenvolvido pela própria autora.

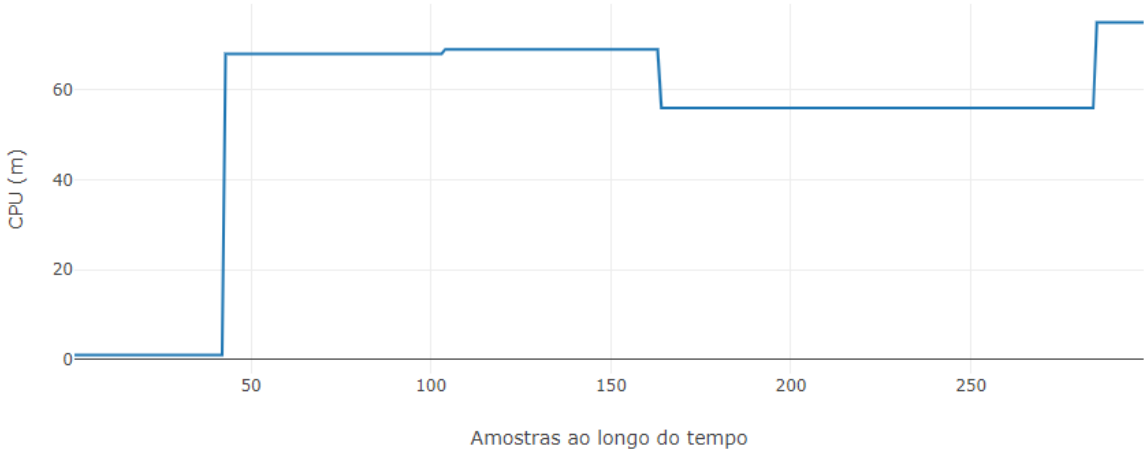
Figura 33 – Exposição do uso de memória da aplicação sob o ataque *Raven-Storm* utilizando HPA métricas




Fonte: Desenvolvido pela própria autora.

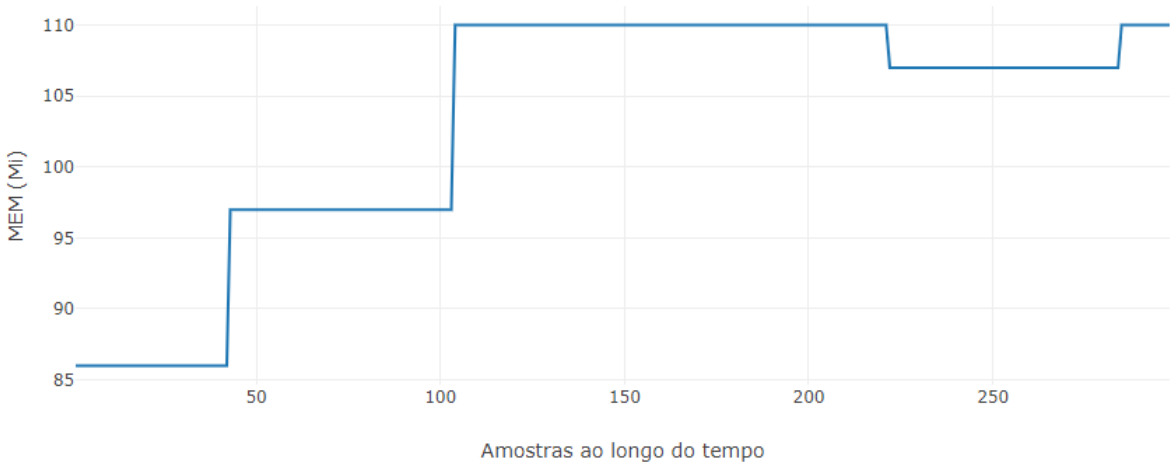
Com base nos gráficos de resultados, nota-se que o limite do HPA não foi atingido. Com isso, foi avaliado a possibilidade de aplicar um ataque com mais *threads*, anteriormente foi utilizado o padrão definido pelo desenvolvedor (400) e nesse experimento foi aumentado para 1500. Os resultados se apresentam nas Figuras 34 e 35.

Figura 34 – Exposição do uso de CPU da aplicação sob o ataque *Raven-Storm* com 1500 *treads* utilizando HPA métricas




Fonte: Desenvolvido pela própria autora.

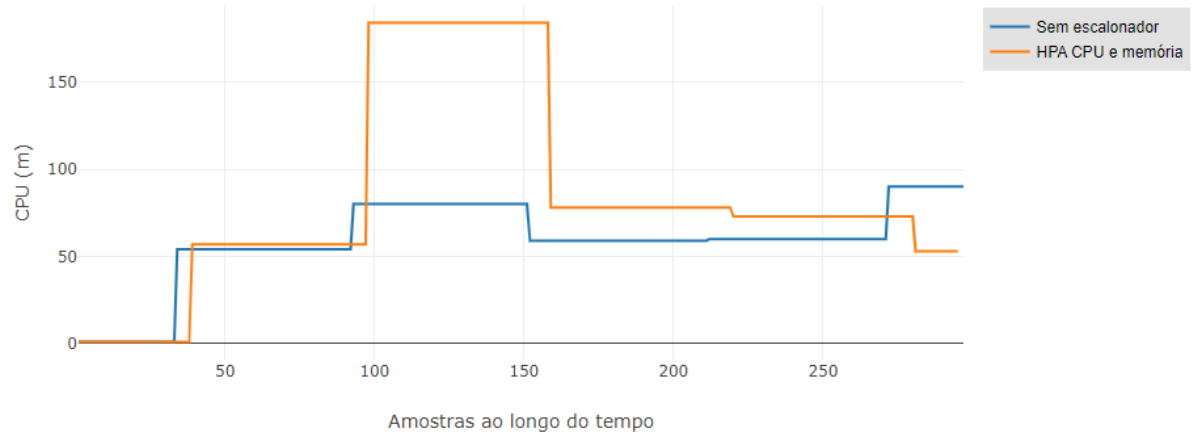
Figura 35 – Exposição do uso de memória da aplicação sob o ataque *Raven-Storm* com 1500 *treads* utilizando HPA métricas




Fonte: Desenvolvido pela própria autora.

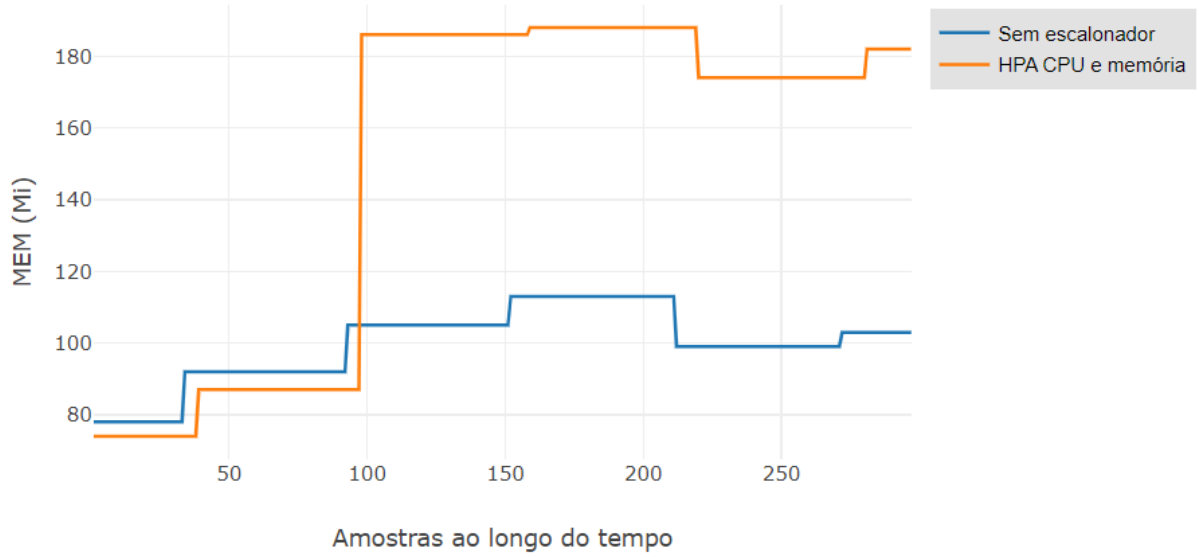
Nota-se a partir da figura acima que, da mesma forma, os limites HPA não foram atingidos. Logo para fins de estudos foi reduzido o valor de limite definido de `140 m` para `70 m`. As Figuras 36 e 37 expõem os resultados.

Figura 36 – Exposição do uso de CPU da aplicação sob o ataque *Raven-Storm* utilizando HPA métricas sob novo limite




Fonte: Desenvolvido pela própria autora.

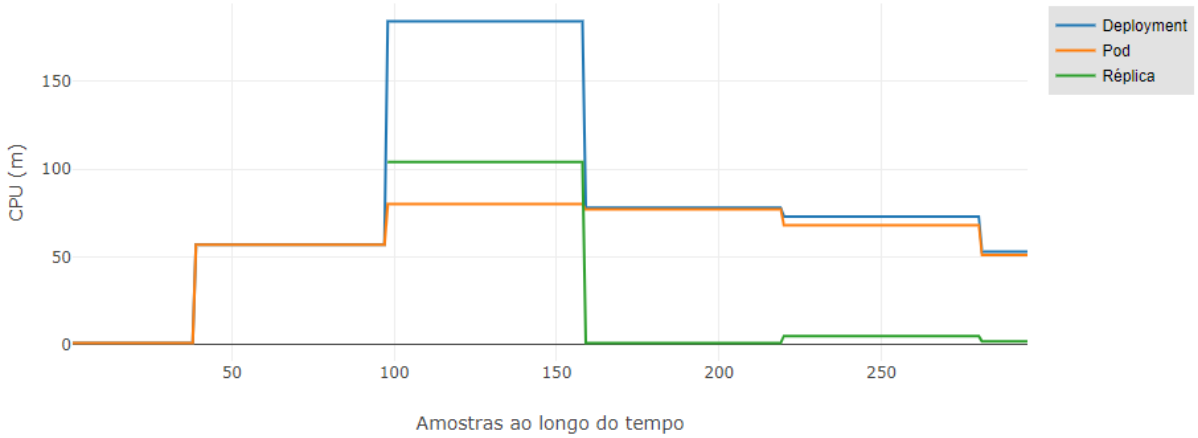
Figura 37 – Exposição do uso de memória da aplicação sob o ataque *Raven-Storm* utilizando HPA métricas sob novo limite




Fonte: Desenvolvido pela própria autora.

Analisando os gráficos, nota-se que no caso com escalonador, o valor da CPU no fim do tempo de experimento era menor. Porém, o valor da memória ultrapassou o dobro do valor no caso sem escalonador. Ainda, nota-se um grande salto na utilização das métricas, por volta da amostra 96, nos dois casos. Ao reduzir o limite, era esperado que quando a primeira réplica fosse criada e atingisse os valores de gatilhos escolhidos. Isto porque na inicialização do *Pod* as métricas são altas, e a memória pode atingir o valor definido como limite. Mas, o esperado não ocorreu, apenas uma réplica foi criada (Fig. 38). Existem outras configurações para evitar esses casos, mas neste projeto elas não foram aplicadas.

Figura 38 – Avaliação do efeito de criação de *Pod* sob o ataque *Raven-Storm*




Fonte: Desenvolvido pela própria autora.

Os tempos computados durante os experimentos com escalonamento horizontal foram resumidas na Tabela 2 na ordem apresentada anteriormente. Quanto ao tempo, os valores

não sofreram grande interferência, os valores médios se mantiveram próximos e quanto ao valores máximos houve apenas uma grande divergência, no caso com mais *threads*.

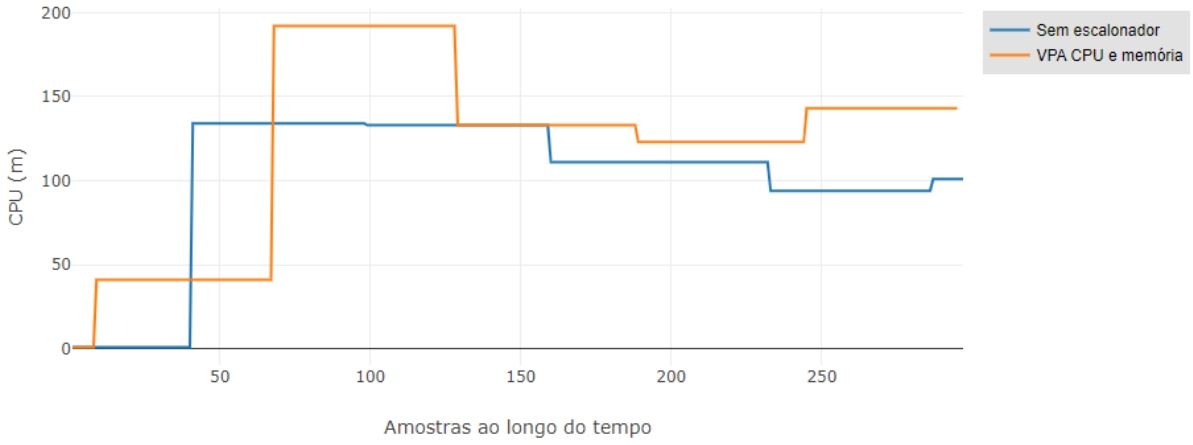
Tabela 2 – Exposição dos valores médios e máximos de tempo de resposta dos experimentos no caso sob o ataque *Raven-Storm* utilizando HPA métricas

| # | Tempo médio de resposta [s] | Tempo máximo de resposta [s] |
|---|---|---|
| Sem escalonamento | 0,43 | 1,95 |
| *Raven-Storm* | 0,22 | 2,15 |
| *Raven-Storm* com 1500 *threads* | 0,40 | 3,68 |
| *Raven-Storm* com HPA sob novo limite | 0,46 | 2,39 |

## 4.3 *Vertical Pod Autoscaling* (VPA) utilizando as métricas memória e CPU

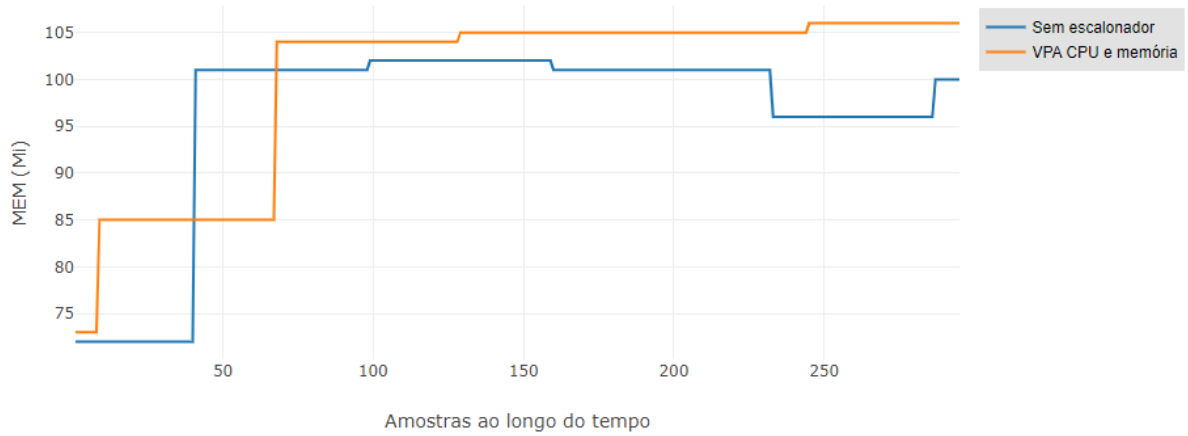
O último experimento foi realizado com as mesmas métricas do experimento 2, mas com o escalonamento vertical. A configuração feita para prepará-lo para o experimento foi semelhante a do experimento 2, apenas foi executado o arquivo YAML (Apêndice E). Os resultados gerados no experimento sob o ataque *DDoS-Ripper* se apresentam nas Figuras 39 e 40.

Figura 39 – Exposição do uso de CPU da aplicação sob o ataque *DDoS-Ripper*




Fonte: Desenvolvido pela própria autora.

Figura 40 – Exposição do uso de memória da aplicação sob o ataque *DDoS-Ripper*

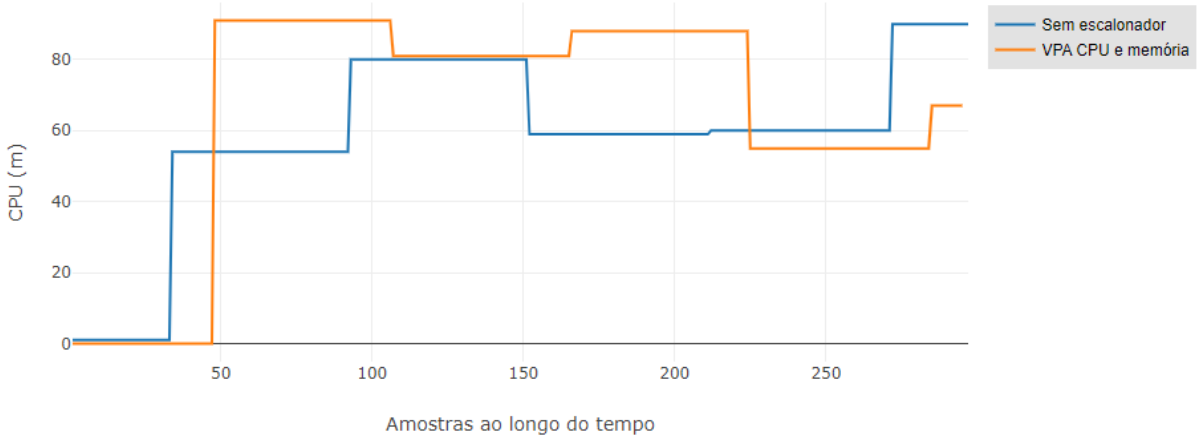



Fonte: Desenvolvido pela própria autora.

Os comportamentos apresentados bem próximos ao sem escalonamento, como nesse caso não existe a criação de um novo *Pod* não vemos um salto como nas situações anteriores, notamos que ocorre uma pequena elevação no início da simulação e outra por volta de 70, a segunda bem semelhante ao salto próximo de 40 da referência. No entanto, os valores se mantém por todo o gráfico bem próximos. Os valores encontrados de tempo de resposta foram a média de 0,52 segundos e a máxima 2,33 segundos. Enquanto, no caso sem escalonamento eram 0,83 e 8,06 segundos. Notando assim, uma redução nas métricas externas.

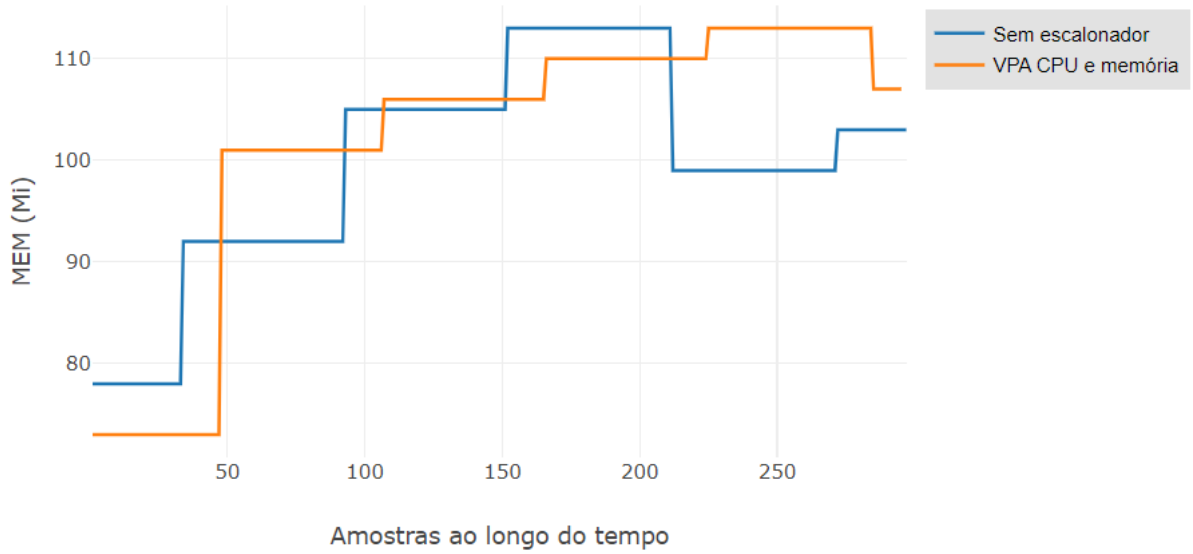
Após a realização do ataque *Raven-Storm*, obtivemos os seguintes resultados apresentados nas Figuras 41 e 42.

Figura 41 – Exposição do uso de CPU da aplicação sob o ataque *Raven-Storm* utilizando VPA




Fonte: Desenvolvido pela própria autora.

Figura 42 – Exposição do uso de memória da aplicação sob o ataque *Raven-Storm* utilizando VPA




Fonte: Desenvolvido pela própria autora.

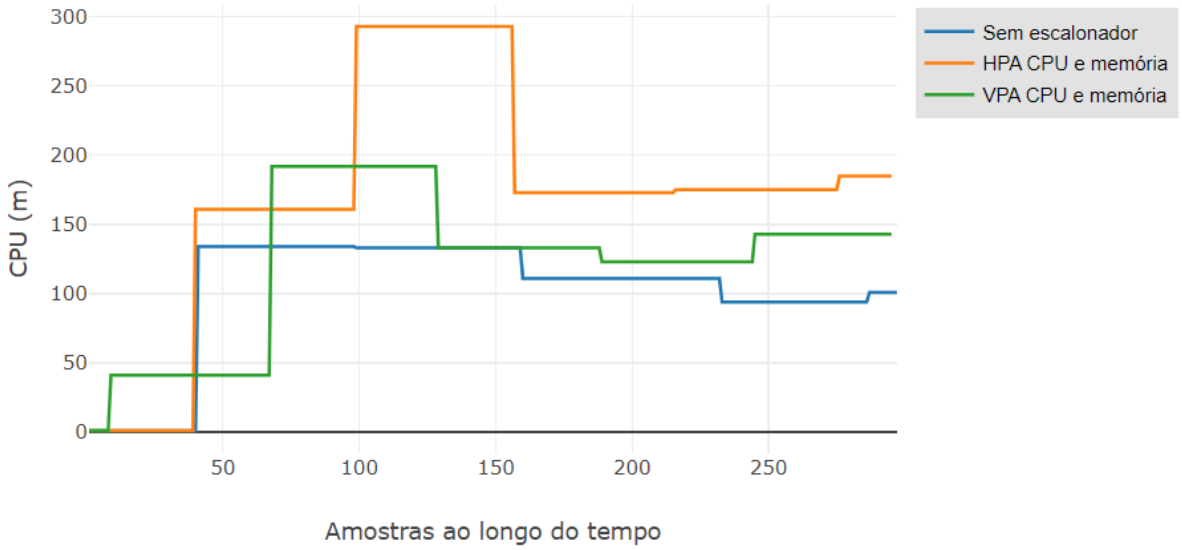
Novamente os resultados foram bem próximos. No entanto, nesse caso, no que tange a CPU o valor dentre os últimos minutos foi menor que a referência. Enquanto, o valor da memória se manteve bem próximo. Os tempos foram de 0,25 e 1,05 segundos, médio e máximo, respectivamente. Enquanto, no caso sem escalonamento eram 0,43 e 1,95 segundos. Notando assim, uma redução nas métricas externas.

## 4.4 Análise

Nas duas primeiras opções notou-se que o grande impacto foi na criação dos *Pods*. Utilizando uma aplicação real e a analisando como se tratando de um microsserviço, encontra-se o consumo médio apenas da aplicação, sem clientes, de 84MB. Quando estimado 3000 novas requisições em 5 minutos, encontramos 115MB. O valor médio encontrado logo no inicio da vida do *Pod* foi de 74MB. Esse valor corresponde a 64% de 115MB. O tempo médio de estabilização para valores comuns é de um minuto. Portanto, nesse contexto, mesmo que no primeiro caso a CPU se manteve menor que a situação sem escalonador, essas duas opções não se mostraram viáveis no que tange microsserviços. No entanto, se tratando de aplicações monolíticas, o consumo da criação do *Pod* será bem inferior ao consumo na aplicação, possivelmente, se tonando viável.

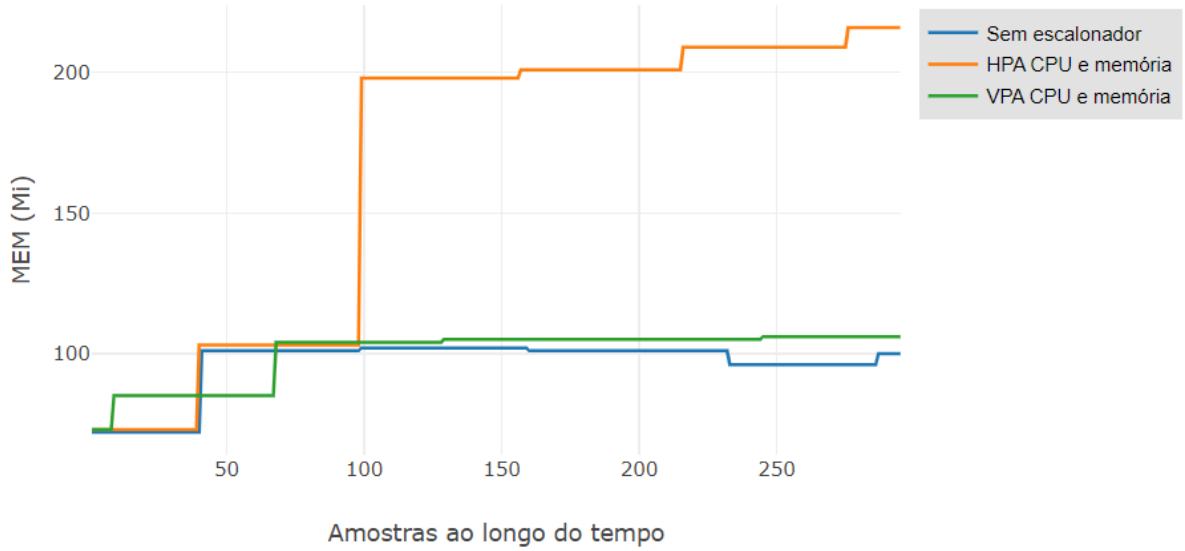
Como não foi possível a utilização dos dois ataques na primeira opção de escalonador, as Figuras 43 e 44 sob o ataque *DDoS-Ripper* são visualizados apenas dois escalonadores.

Figura 43 – Comparação dos escalonadores sob a aplicação atacada pelo *DDoS-Ripper* sob a perspectiva da CPU




Fonte: Desenvolvido pela própria autora.

Figura 44 – Comparação dos escalonadores sob a aplicação atacada pelo *DDoS-Ripper* sob a perspectiva da memória




Fonte: Desenvolvido pela própria autora.

Neste caso, observa-se que enquanto o VPA se manteve mais próximo da referência, o HPA, após a criação do *Pod*, se manteve levemente mais afastado, no caso da CPU e bastante maior, no caso da memória.

A métrica tempo de resposta é de extrema importância no contexto tratado neste projeto. Isto porque essa é a métrica que mais define qual será a percepção de novos clientes sob uma aplicação em ataque. A Tabela 3 demonstra os valores sob a perspectiva do ataque

nas *DDoS-Ripper*.

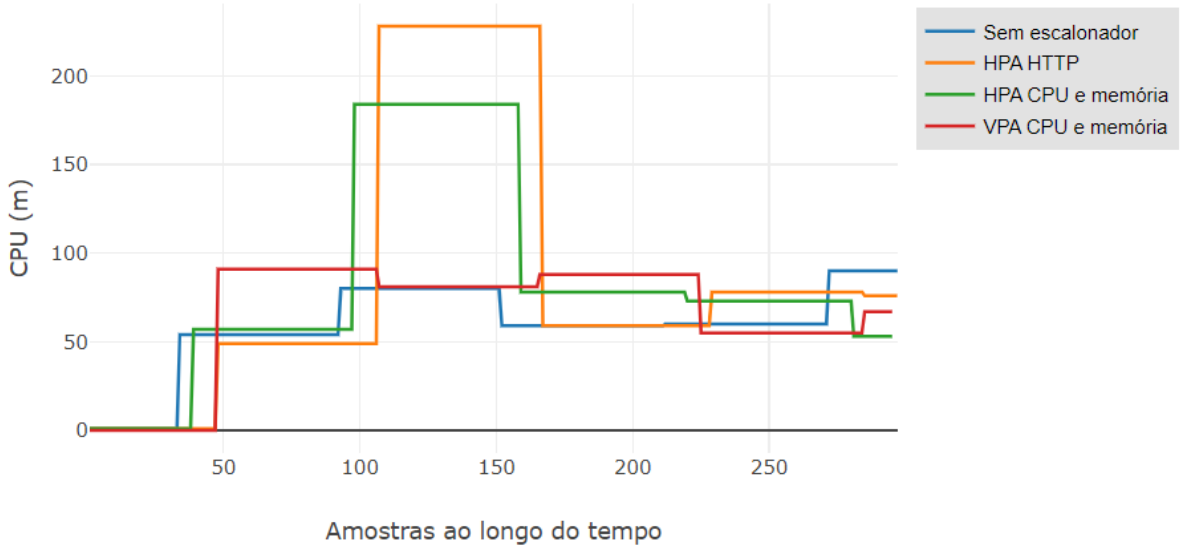
Tabela 3 – Comparação final dos valores médios e máximos de tempo de resposta dos experimentos sob o ataque *DDoS-Ripper*

| # | Tempo médio de resposta [s] | Tempo máximo de resposta [s] |
|---|---|---|
| Sem ataque | 0,06 | 0,51 |
| Sem escalonador | 0,83 | 8,06 |
| HPA CPU | 0,29 | 1,82 |
| VPA | 0,52 | 2,33 |

Nela, nota-se que os menores valores encontrados, descontando o caso sem ataque, é o caso do HPA com base nas métricas memória e CPU. E, em segundo, o escalonador vertical. Portanto, no caso do ataque *DDoS-Ripper*, o escalonador com os melhores resultados no que tange a métrica interna foi o VPA, enquanto no que tange a métrica externa foi o HPA. Porém, a diferença entre seus tempos médios é de 0,23 segundos, um tempo quase imperceptível ao usuário da aplicação.

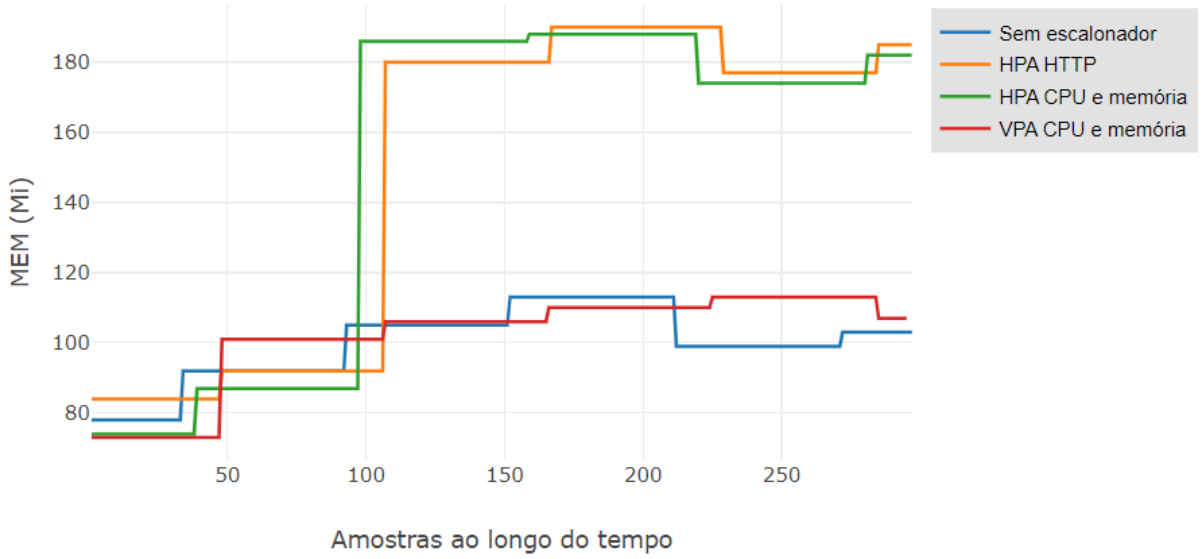
As Figuras 45 e 46 demonstram o efeito sob o ataque *Raven-Storm*.

Figura 45 – Comparação dos escalonadores sob a aplicação atacada pelo *Raven-Storm* sob a perspectiva da CPU




Fonte: Desenvolvido pela própria autora.

Figura 46 – Comparação dos escalonadores sob a aplicação atacada pelo *Raven-Storm* sob a perspectiva da memória




Fonte: Desenvolvido pela própria autora.

Nota-se que nas duas imagens, o único escalonador que manteve, sob as duas métricas, o consumo da aplicação estabilizado foi o vertical. Isto porque além de não ocorrer os picos na criação de novos *Pods* como as duas primeiras opções, o valor se manteve bem próximo a referência. Apesar do pico na criação dos *Pods*, o consumo de CPU nos outros dois casos se estabilizou dentro dos 5 minutos. Porém, a memória consumida foi de quase o dobro da referência.

No caso do ataque *Raven-Storm*, percebe-se na Tabela 4, o escalonador com os menores valores foi o vertical, seguido do caso HPA HTTP. No entanto, esse ainda foi maior que a referência.

Tabela 4 – Comparação final dos valores médios e máximos de tempo de resposta dos experimentos sob o ataque *Raven-Storm*

| # | Tempo médio de resposta [s] | Tempo máximo de resposta [s] |
|---|---|---|
| Sem ataque | 0,06 | 0,51 |
| Sem escalonador | 0,43 | 1,95 |
| HPA HTTP | 0,40 | 3,91 |
| HPA CPU | 0,46 | 2,39 |
| VPA | 0,25 | 1,05 |

Nas duas tabelas, nota-se que os valores de tempo máximo quanto utilizado HPA sempre foram superiores as referências (sem escalonador). Isto porque na criação do *Pod*, como explicitado anteriormente, ocorre um aumento das métricas memória e CPU. Consequentemente, nesse período, o tempo também aumenta.

Vale ressaltar que em uma situação em que a aplicação não possui escalonamento e tem recursos definidos conforme demanda normal de clientes, provavelmente, caso ocorra um ataque DDoS, esse causaria grandes instabilidades ou, até mesmo, a tornaria indisponível, causando um enorme prejuízo financeiro. Para contornar essa situação, há a possibilidade de manter os recursos superiores ao consumo comum (superdimensionar). Porém, nesse caso, o custo de manter a aplicação seria maior. E, esse custo adicional seria, em geral, apenas um custo, assim, somente sob a possibilidade de ocorrer o ataque, poderia ser considerado um uso válido. Nesse contexto, a melhor situação apresentada é a utilização de algum escalonador que redimensionariam a aplicação conforme a demanda.

# 5 CONCLUSÃO

O uso de escalonadores automáticos apresenta-se como uma maneira de mitigar ataques de negação de serviço distribuído. Este trabalho analisou três propostas de escalonamento à partir de três métricas conhecidas e importantes. Todo o ambiente experimental foi implementado no *Kubernetes*, a fim de ser aproveitado as funcionalidades de escalonamento presentes no mesmo. O cenário desenvolvido para os experimentos foi uma aplicação *web* baseada em microsserviços e um banco de dados SQL. Os ataques escolhidos possuem foco na camada 7 da tabela OSI.

No presente trabalho foi possível compreender o comportamento e características a se considerar no que tange a utilização dos escalonadores. O *Kubernetes* tem dois tipos, o escalonador horizontal (HPA) e o vertical (VPA). Visto que o HPA aumenta o número de *Pods* vinculados a um *Deployment* e o VPA aumenta a capacidade desse *Pods*. Foram propostos três configurações de escalonadores. No primeiro caso, o HPA em conjunto com o KEDA capturando o valor agregado da taxa de solicitações HTTP por segundo, conforme medido nos últimos dois minutos. E o limite definido de réplicas foi 10.

No segundo caso, foi desenvolvido um HPA, os limites definidos foram as métricas CPU e memória e o número máximo de 5 réplicas. E o terceiro, o VPA, os limites foram definidos a partir de um valor máximo de CPU e memória. Os parâmetros de analise escolhidos foram CPU, memória e os tempos médios e máximos de resposta da requisição HTTP.

No contexto de microsserviços foi observado que a utilização do VPA trouxe mais vantagem se comparada as outras opções. Isto porque na criação de um novo *Pod* o consumo de métricas para as configurações inicias foram bem superiores, se comparado o consumo comum do *Deployment*. Essa configuração inicial também atingiu a funcionalidade da aplicação, fato percebido ao analisar os tempos médios e máximos de resposta da requisição HTTP médias e máximas. Nesses casos, os maiores tempos médios e máximos de resposta foram encontrados logo a após a criação do *Pod*, no mesmo momento em que o cossumo de CPU e memória estavam altos.

Além das opções de escalonamento foram apresentados duas opções adicionais, a primeira seria a limitação de requisições HTTP por um único IP. Essa é uma opção que não limita a utilização de outras formas de mitigação e é bastante simples de implementar. A segunda opção não é uma opção de mitigação e, sim, de monitoramento. O uso do *Prometheus* em conjunto com o *Grafana* tem o grande potencial de ser um modelo visual de monitoramento

e alerta para casos de ataque.

A realização de uma análise econômica sob o contexto apresentado é interessante, no que tange um trabalho futuro. O cenário apresentado é bem amplo, sendo possível a expansão desse estudo para os ataque destinados as demais camadas da tabela OSI. Desta maneira, tornando possível a definição de uma taxa de eficacia geral sob cada método de mitigação. Existem também a possibilidade de estudo de outros métodos de mitigação, para que ocorra uma análise sob a perspectiva geral e sob a eficacia de métodos em conjunto. Outro experimento interessante é a realização do ataque até tornar o serviço indisponível e, nesse momento, acionar o escalonamento para tornar o serviço novamente disponível.

# REFERÊNCIAS

ALECRIM, E. Microsoft barra ataque DDoS com tráfego recorde de 3,47 Tb/s contra o alvo. 2022. Disponível em: <https://tecnoblog.net/noticias/2022/01/28/microsoft-barra-ataque-ddos-com-trafego-recorde-de-347-tb-s-contra-o-alvo/>. Acesso em: 25 jul. 2022. Citado na página 15.

ALVES, F. Monitorando Kubernetes com Prometheus e Grafana. 2020. Disponível em: <https://medium.com/linux-world-open-source/monitorando-kubernetes-com-prometheus-e-grafana-80c86222ed28>. Acesso em: 23 jul. 2022. Citado na página 48.

AMAZON. O que é um ataque DDoS? 2022. Disponível em: <https://aws.amazon.com/pt/shield/ddos-attack-protection/>. Acesso em: 17 jul. 2022. Citado 2 vezes nas páginas 34 e 35.

AYUB, T. Onda de ataques cibernéticos DDoS deixa cidades do ES sem internet. 2022. Disponível em: <https://tecnologia.ig.com.br/colunas/internet-sem-aspas/2022-07-07/ataques-ddos-espirito-santo-internet.html>. Acesso em: 25 jul. 2022. Citado na página 15.

BANACH, Z. What Is Session Hijacking: Your Quick Guide to Session Hijacking Attacks. 2019. Disponível em: <https://www.invicti.com/blog/web-security/session-hijacking/>. Acesso em: 20 jul. 2022. Citado na página 32.

BIT2ME. O que são ataques DoS? 2022. Disponível em: <https://academy.bit2me.com/>. Acesso em: 17 jul. 2022. Citado na página 33.

CLOUDFARE. ataque de inundação de HTTP. 2022. Disponível em: <https://www.cloudflare.com/pt-br/learning/ddos/http-flood-ddos-attack/>. Acesso em: 16 ago. 2022. Citado na página 39.

COOK, S. Mais de 20 estatísticas e fatos de ataques DDoS para 2018-2022. 2022. Disponível em: <https://www.comparitech.com/>. Acesso em: 25 jul. 2022. Citado na página 15.

DOCKER. Dockeer Hub. 2022. Disponível em: <https://hub.docker.com/>. Acesso em: 17 jul. 2022. Citado na página 20.

DRAGONI, N.; GIALLORENZO, S.; LAFUENTE, A. L.; MAZZARA, M.; MONTESI, F.; MUSTAFIN, R.; SAFINA, L. Microservices: Yesterday, today, and tomorrow. In: _____. Present and Ulterior Software Engineering. Cham: Springer International Publishing, 2017. p. 195–216. ISBN 978-3-319-67425-4. Citado na página 18.

EDUCATION, I. C. Interface de programação de aplicativos (API). 2020. Disponível em: <https://www.ibm.com/cloud/learn/api>. Acesso em: 19 jul. 2022. Citado na página 26.

FARE, C. Como funcionam os ataques DDoS de camada 3? | L3 DDoS. 2022. Disponível em: <https://www.cloudflare.com/pt-br/learning/ddos/layer-3-ddos-attacks/>. Acesso em: 17 jul. 2022. Citado na página 34.

GANESAN, N. Kubernetes. 2019. Disponível em: <https://faun.pub/kubernetes-intro-45017a8da8c6>. Acesso em: 17 jul. 2022. Citado 3 vezes nas páginas 21, 22 e 23.

GOYAL, S. PHP Code Injection – Attacks and Mitigation. 2020. Disponível em: <https://secnhack.in/php-code-injection-attacks-and-mitigation/>. Acesso em: 20 jul. 2022. Citado na página 29.

GRAFANA. Create and manage Grafana Alerting rules. 2022. Disponível em: <https://grafana.com/docs/grafana/latest/alerting/alerting-rules/>. Acesso em: 23 jul. 2022. Citado na página 49.

GUPTA, A. Autoscaling Kubernetes apps with Prometheus and KEDA. 2019. Disponível em: <https://itnext.io/tutorial-auto-scale-your-kubernetes-apps-with-prometheus-and-keda-c6ea460e4642>. Acesso em: 21 jul. 2022. Citado na página 46.

IUGU, R. Protocolos de API: o que são e quais são eles? 2022. Disponível em: <https://www.iugu.com/iugu4devs/blog/protocolos-de-api>. Acesso em: 19 jul. 2022. Citado 2 vezes nas páginas 26 e 27.

JAKKULA, V. Horizontal Pod Autoscaler(HPA) based on CPU and Memory. 2020. Disponível em: <https://faun.pub/kubernetes-horizontal-pod-autoscaler-hpa-bb789b3070e4>. Acesso em: 21 jul. 2022. Citado na página 47.

KANJILAL, J. Security Challenges and Solutions for Microservices Architecture. 2021. Disponível em: <https://www.developer.com/security/security-solutions-microservices/>. Acesso em: 20 jul. 2022. Citado na página 28.

KEDA. Kubernetes Event-driven Autoscaling. 2022. Disponível em: <https://keda.sh/>. Acesso em: 21 jul. 2022. Citado na página 25.

MALWARE. 2021. Disponível em: <https://pt.wikipedia.org/wiki/Malware>. Acesso em: 17 jul. 2022. Citado na página 33.

MENDONçA, P. A. S. de. Virtualização, Containers e Docker. 2019. Disponível em: <https://pauloandresoares.com/2019/10/29/virtualizacao-containers-e-docker.html>. Acesso em: 03 jul. 2022. Citado na página 20.

MISTRETTA, A. CRESCENTE A DEMANDA POR APM, CLOUD E MICROSSERVIÇOS. 2022. Disponível em: <https://www.trescon.com.br/crescente-a-demanda-por-apm-cloud-e-microsservicos/>. Acesso em: 21 jul. 2022. Citado na página 14.

MOREIRA, C. SQL Injection: Injetando Dados a partir de Inputs. 2017. Disponível em: <https://www.scriptcaseblog.net/pt/development-pt/sql-injection-injetando-dados-a-partir-de-inputs/>. Acesso em: 20 jul. 2022. Citado na página 30.

NAEEM, T. Definição de API REST: Noções básicas de APIs REST. 2020. Disponível em: <https://www.astera.com/>. Acesso em: 19 jul. 2022. Citado na página 27.

NEWMAN, S. Building Microservices. 5. ed. [S.l.]: Copyright, 2015. Citado na página 28.

PRODANOV, E. C. de Freitas e C. C. Metodologia do trabalho científico [recurso eletrônico] : métodos e técnicas da pesquisa e do trabalho acadêmico. 2. ed. Novo Hamburgo: Universidade Feevale: Feevale, 2013. Citado 2 vezes nas páginas 16 e 17.

RADWARE. Radware Threat Advisory. 2022. Disponível em: <https://www.radware.com/getattachment/527adc01-4d91-4366-ac2c-562a4f5d5d7a/Alert-OpsBedilReloaded.pdf.aspx>. Acesso em: 26 jul. 2022. Citado na página 40.

REDHAT. Containers e máquinas virtuais (VMs). 2020. Disponível em: <https://www.redhat.com/pt-br/topics/containers/containers-vs-vms>. Acesso em: 03 jul. 2022. Citado na página 19.

REDHAT. O que são os microsserviços? 2021. Disponível em: <https://www.redhat.com/pt-br/topics/microservices/what-are-microservices>. Acesso em: 03 jul. 2022. Citado na página 19.

ROSA, M. S. Kubernetes: Banco PostgreSQL. 2020. Disponível em: <https://medium.com/bemobi-tech/kubernetes-banco-postgresql-2e799bfbc6ae>. Acesso em: 19 jul. 2022. Citado 2 vezes nas páginas 42 e 43.

SECURITY, C. A Complete Guide to Man in The Middle Attack (MitM). 2021. Disponível em: <https://wallstreetinv.com/cyber-security/man-in-the-middle-attack-mitm/>. Acesso em: 20 jul. 2022. Citado na página 33.

SENHASEGURA. Os desafios para garantir a segurança de APIs. 2022. Disponível em: <https://senhasegura.com/pt-br/os-desafios-para-garantir-a-seguranca-de-apis/>. Acesso em: 20 jul. 2022. Citado na página 32.

SILVA, R. C. da. Best Practices to Protect Your Microservices Architecture. 2017. Disponível em: <https://medium.com/@rcandidosilva/best-practices-to-protect-your-microservices-architecture-541e7cf7637f>. Acesso em: 12 set. 2021. Citado 2 vezes nas páginas 27 e 28.

SNYK. Code injection. 2022. Disponível em: <https://learn.snyk.io/lessons/malicious-code-injection/javascript/>. Acesso em: 20 jul. 2022. Citado na página 31.

TENNAKOON, R. Deploying a Complete Node.js Application in Kubernetes. 2021. Disponível em: <https://levelup.gitconnected.com/deploying-a-complete-node-js-application-in-kubernetes-d747986e1e61>. Acesso em: 19 jul. 2022. Citado na página 41.

VAADATA. How to strengthen the security of your APIs to counter the most common attacks? 2022. Disponível em: <https://www.developer.com/security/security-solutions-microservices/>. Acesso em: 20 jul. 2022. Citado 3 vezes nas páginas 29, 30 e 32.

# APÊNDICE A – BANCO DE DADOS

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  selector:
    matchLabels:
      app: postgres
  replicas: 1
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:10.4
          imagePullPolicy: "IfNotPresent"
          ports:
          - containerPort: 5432
          resources:
            limits:
              memory: 1Gi
              cpu: 2
            requests:
              memory: 512Mi
              cpu: 1
          volumeMounts:
            - mountPath: /var/lib/postgresql/data
              name: postgresdb
      volumes:
        - name: postgresdb
          persistentVolumeClaim:
            claimName: postgres-pvc
---
```

```
kind: Service
apiVersion: v1
metadata:
  name: postgres
spec:
  selector:
    app: postgres
  ports:
    - port: 5432
  type: ClusterIP
```

# APÊNDICE B – PROMETHEUS

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
  - services
  verbs: ["get", "list", "watch"]
- nonResourceURLs: ["/metrics"]
  verbs: ["get"]
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: prom-conf
  labels:
```

```
    name: prom-conf
data:
  prometheus.yml: |-
    global:
      scrape_interval: 5s
      evaluation_interval: 5s
    scrape_configs:
      - job_name: 'pg-setup-day'

        kubernetes_sd_configs:
        - role: service
        relabel_configs:
        - source_labels: [__meta_kubernetes_service_label_run]
          regex: pg-setup-day
          action: keep
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus
spec:
  replicas: 1
  selector:
      matchLabels:
        app: prometheus
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      containers:
        - name: prometheus
          image: prom/prometheus
          args:
            - "--config.file=/etc/prometheus/prometheus.yml"
            - "--storage.tsdb.path=/prometheus/"
          ports:
            - containerPort: 9090
          volumeMounts:
```

```
            - name: prometheus-config-volume
              mountPath: /etc/prometheus/
            - name: prometheus-storage-volume
              mountPath: /prometheus/
      volumes:
        - name: prometheus-config-volume
          configMap:
            defaultMode: 420
            name: prom-conf

        - name: prometheus-storage-volume
          emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: prometheus
spec:
  ports:
  - port: 9090
    protocol: TCP
  selector:
    app: prometheus
```

# APÊNDICE C – SCALED OBJECT

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: prometheus-scaledobject
  namespace: default
spec:
  scaleTargetRef:
    kind: Deployment
    name: pg-front-day
  pollingInterval: 15
  minReplicaCount: 1
  maxReplicaCount: 10
  triggers:
  - type: prometheus
    metadata:
      serverAddress: http://prometheus.default.svc.cluster.local:9090
      metricName: access_frequency
      threshold: '3'
      query: sum(rate(http_requests[2m]))
```

# APÊNDICE D – HORIZONTAL POD AUTOSCALING UTILIZANDO MEMORIA E CPU

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-cpu
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: pg-front-day
  minReplicas: 1
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: AverageValue
          averageValue: 140m
    - type: Resource
      resource:
        name: memory
        target:
          type: AverageValue
          averageValue: 120Mi
```

# APÊNDICE E – VERTICAL POD AUTOSCALING UTILIZANDO MEMORIA E CPU

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa
  namespace: default
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: pg-front-day
  updatePolicy:
    updateMode: "Auto"
  resourcePolicy:
    containerPolicies:
      - containerName: pg-front-day
        maxAllowed:
          cpu: 4
          memory: 5Gi
        controlledResources: ["cpu", "memory"]
```