

Relatório de Laboratório - Sistemas Distribuídos

Rodrigo Sales Araújo Teixeira
Abraão Lenon Moreira de Oliveira
Matheus Linhares Ferreira Gomes
Universidade de Fortaleza (Unifor)

3 de julho de 2025

Sumário

1	Especificações do Laboratório	1
1.1	Especificações das Tarefas	1
1.1.1	Tarefa 1 - Configuração do Ambiente	1
1.1.2	Tarefa 2 - Implantação da Aplicação	1
1.1.3	Tarefa 3 - Teste de Desempenho	1
1.1.4	Tarefa 4 - Teste de Resiliência	2
1.1.5	Tarefa 5 - Teste de Desempenho com Escalonamento Automático	2
1.2	Entregáveis	3
1.2.1	Entrega Parcial 1 - Configuração e Implantação	3
1.2.2	Entrega Parcial 2 - Teste de Desempenho e Análise de Overhead.	3
1.2.3	Entrega Parcial 3 - Testes de Resiliência e Escalonamento Automático.	3
1.2.4	Entrega Final - Relatório Consolidado e Repositório Completo.	4
2	Ambiente de Operação	5
2.1	Ambiente Operacional	5
2.2	Repositório Git	5
3	Instalações	6
3.1	Instalação do Docker	6
3.2	Instalação do <i>kubectl</i>	6
3.3	Instalação do Kind	7
3.4	Instalação do Istio	7
3.5	Instalação do Skaffold	7
3.6	Instalação do K9S	8
3.7	Instalação do Kustomize	8
3.8	Implantação da Aplicação Online Boutique	8
3.9	Instalação do Locust	11
4	Testes de Carga	13
4.1	Realização dos Testes	13
4.2	Resultados Obtidos	15
4.3	Análise dos Resultados dos Testes de Carga	16
4.3.1	Tempo Médio de Resposta	16
4.3.2	Taxa de Erro das Requisições	17
4.3.3	Throughput (Requisições por Segundo)	17
4.3.4	Análise Gráfica	18
4.3.5	Análise dos Arquivos de Log (Exploratória)	18
4.3.6	Considerações Finais	18

5	Teste de Resiliência	19
5.1	Cenário de teste	19
5.2	Arquivos de injeção de falha controlada	20
5.3	Resultados Obtidos	21
5.4	Análise dos Resultados	21
5.4.1	Análise das Métricas de Resiliência	21
5.5	Conclusão	22
6	Teste de Escalonamento Automático usando HPA	23
6.1	Cenário de Teste	23
6.2	Arquivos utilizados	24
6.3	Resultados Obtidos	25
6.4	Análise das Métricas de Desempenho – Com e Sem HPA	26
6.5	Conclusão	26

Resumo

Este documento técnico descreve o desenvolvimento, execução e consolidação das atividades práticas realizadas no âmbito do Laboratório de Sistemas Distribuídos do Programa de Pós Graduação em Informática Aplicada da Universidade de Fortaleza(Unifor). O foco principal foi a implementação e análise de uma arquitetura de microsserviços operando sob a plataforma Kubernetes, com ênfase no uso do Istio como service mesh.

Para a realização dos experimentos, foi empregada a aplicação Online Boutique, a qual foi implantada em dois cenários distintos: um com injeção automática de sidecars pelo Istio ativada, e outro sem esse recurso. O trabalho foi estruturado em cinco fases: (1) preparação do ambiente experimental, (2) implantação da aplicação, (3) execução de testes de desempenho para medir o impacto do Istio, (4) avaliação da resiliência por meio da injeção de falhas controladas, e (5) testes de escalabilidade utilizando o Horizontal Pod Autoscaler (HPA) do Kubernetes.

Durante a execução das tarefas, foram empregadas ferramentas como o Locust para geração de carga, além de utilitários como Kind, Skaffold, Kustomize, K9S e Istioctl para gerenciamento e observabilidade do sistema. Os resultados evidenciaram que, embora o Istio introduza um certo overhead, ele também agrega melhorias consideráveis. O uso do HPA demonstrou ser eficaz na adaptação a picos de demanda.

Capítulo 1

Especificações do Laboratório

1.1 Especificações das Tarefas

1.1.1 Tarefa 1 - Configuração do Ambiente

- Configurem um repositório Git compartilhado para o projeto.
- Instalem e configurem a distribuição Kubernetes local escolhida em suas máquinas (ou em uma máquina compartilhada pela equipe). Certifiquem-se de alocar recursos suficientes (CPU/RAM).
- Instalem o Istio no cluster Kubernetes, utilizando o perfil de instalação demo ou default. Verifiquem a instalação.

1.1.2 Tarefa 2 - Implantação da Aplicação

- Obtenham os manifestos de implantação da aplicação Online Boutique.
- Implantação base: implantem a aplicação sem a injeção automática de sidecars do Istio (ou seja, em um namespace sem o rótulo istio-injection=enabled). Verifiquem se todos os serviços estão rodando e se a aplicação está acessível.
- Implantação com Istio: habilitem a injeção automática de sidecars do Istio para um novo namespace (e.g., online-boutique-istio) e implantar a aplicação novamente neste namespace. Verifiquem se os sidecars foram injetados (`kubectl get pods -n <namespace> -o wide` deve mostrar 2/2 containers por pod) e se a aplicação continua acessível.

1.1.3 Tarefa 3 - Teste de Desempenho

- Configurem a ferramenta de geração de carga escolhida (Locust ou k6). Criem um script de teste que simule a interação de usuários com a loja online (e.g., navegar por produtos, adicionar ao carrinho, finalizar compra). Dica: a aplicação Online Boutique já vem com um script para realizar testes de carga com o Locust.
- Teste sem Istio: executem o teste de carga contra a versão da aplicação sem os sidecars do Istio (implantação base). Coletam métricas como latência média/percentil 95/99 e vazão (requisições por segundo).

- Teste com Istio: executem o mesmo teste de carga, com a mesma intensidade, contra a versão da aplicação com os sidecars do Istio (implantação com Istio). Coletem as mesmas métricas.
- Análise de overhead: comparem os resultados dos dois testes. Analisem e quantifiquem o overhead (diferença) de desempenho (latência e/ou vazão) introduzido pelo Istio. Discutam possíveis causas para o overhead observado.

1.1.4 Tarefa 4 - Teste de Resiliência

- Utilizando os recursos de `VirtualService` e/ou `DestinationRule` do Istio, configurem regras de injeção de falhas.
- Injeção de atraso: injetem um atraso significativo (e.g., 2 segundos) nas respostas de um serviço interno crítico, mas não essencial para a funcionalidade básica (e.g., recommendation ou ad). Observem (manualmente ou via logs/métricas) como a aplicação se comporta. A interface do usuário ainda funciona? O desempenho degrada gradualmente?
- Injeção de erro: injetem erros HTTP (e.g., 503 Service Unavailable) em uma porcentagem das requisições (e.g., 25% e 50%) para outro serviço (e.g., productcatalog). Observem o comportamento da aplicação. Ela consegue lidar com falhas parciais? Descrevam os mecanismos de resiliência (ou a falta deles) observados.
- Documentem as configurações do Istio utilizadas e os comportamentos observados em cada cenário de falha.

1.1.5 Tarefa 5 - Teste de Desempenho com Escalonamento Automático

- Configurem o *Horizontal Pod Autoscaler* (HPA) do Kubernetes para um ou mais serviços que sejam gargalos potenciais sob carga (e.g., frontend, productcatalog, checkout). Definam métricas de alvo (e.g., utilização de CPU em 70%). Certifiquem-se que os requisitos de recursos (CPU/memória) estão definidos nos manifestos de implantação dos serviços alvos para o HPA funcionar corretamente.
- Teste com HPA: executem um teste de carga (usando Locust/k6) com intensidade crescente ou sustentada que seja suficiente para disparar o escalonamento automático. Monitorem o número de pods do(s) serviço(s) com HPA. Coletem métricas de desempenho (latência, vazão) durante o teste.
- Análise dos resultados: comparem o desempenho (latência, vazão) sob carga com o HPA habilitado versus um cenário com um número fixo de réplicas (pode ser o resultado do teste de desempenho com Istio, se a carga for comparável, ou um novo teste de controle). Analisem a eficácia do HPA em manter o desempenho e lidar com a variação de carga. Discutam os limites e desafios do escalonamento automático.

1.2 Entregáveis

1.2.1 Entrega Parcial 1 - Configuração e Implantação

- **Foco:** Tarefas 1 e 2
- **Entregáveis:**
 - Relatório Preliminar (2–3 páginas), incluindo:
 - * Formação da equipe e link para o repositório Git criado.
 - * Evidência do sucesso (e.g., screenshots, logs) na instalação e configuração do ambiente Kubernetes local (incluir versão, recursos alocados).
 - * Evidência do sucesso (e.g., screenshots, logs) na instalação do Istio (incluir versão, perfil utilizado).
 - * Evidência do sucesso (e.g., screenshots, logs) na implantação da aplicação Online Boutique nos dois cenários (sem e com injeção do sidecar Istio).
 - Repositório Git atualizado com a estrutura inicial e quaisquer scripts/manifests básicos utilizados.

1.2.2 Entrega Parcial 2 - Teste de Desempenho e Análise de Overhead.

- **Foco:** Tarefa 3
- **Entregáveis:**
 - Uma seção atualizada do relatório descrevendo:
 - * Metodologia do teste de desempenho (ferramenta escolhida, script de teste, intensidade da carga, duração).
 - * Resultados dos testes de desempenho (tabelas/gráficos comparando latência e vazão com e sem Istio).
 - * Análise preliminar do *overhead* de desempenho introduzido pelo Istio.
 - Repositório Git atualizado contendo os scripts de teste de carga utilizados e os manifestos relevantes da aplicação (se modificados).

1.2.3 Entrega Parcial 3 - Testes de Resiliência e Escalonamento Automático.

- **Foco:** Tarefas 4 e 5
- **Entregáveis:**
 - Uma seção atualizada do relatório descrevendo:
 - * Metodologia dos testes de injeção de falhas (configurações do Istio, cenários testados).
 - * Observações e análise do comportamento da aplicação sob falha injetada.
 - * Configuração do HPA (manifestos YAML).

- * Metodologia do teste de desempenho com HPA (carga aplicada).
- * Resultados do teste com HPA (gráficos mostrando número de pods ao longo do tempo, métricas de desempenho sob carga).
- * Análise preliminar da eficácia do HPA.
- Repositório Git atualizado contendo os manifestos YAML do Istio para injeção de falhas, os manifestos do HPA e quaisquer outros artefatos relevantes.

1.2.4 Entrega Final - Relatório Consolidado e Repositório Completo.

- **Foco:** Integração, refinamento e conclusões
- **Entregáveis:**
 - **Relatório Técnico Final:** versão completa e revisada do relatório, integrando todas as seções anteriores (Introdução, Configuração, Metodologias, Resultados, Análise e Discussão aprofundada comparando todos os experimentos, conclusões gerais, e dificuldades).
 - **Repositório Git Final:** Link para o repositório Git finalizado, contendo todo o código, scripts, manifestos YAML, e um arquivo `README.md` explicando como replicar os experimentos.

Capítulo 2

Ambiente de Operação

2.1 Ambiente Operacional

Sistema Pop_OS 22.04 LTS (x86_64).

Recursos 16 GB RAM, 8 CPU, 512 GB SSD.

Container Runtime Docker Docker version 28.1.1, build 4eba377 (DOCKER, 2025).

Cluster Kind version 0.27.0 (THE KUBERNETES AUTHORS, 2025a).

Scaffold v2.15.0 (AN OPEN SOURCE PROJECT FROM GOOGLE., 2025).

Kustomize v5.6.0 (THE KUBERNETES AUTHORS, 2025c).

locust v2.37.4 (HEYMAN; HOLMBERG; BALDWIN, 2025)

K9S v0.50.4 (THE K9S AUTHORS, 2025)

Istio client version: 1.25.2 control plane version: 1.25.2 data plane version: 1.25.2 (12 proxies) (THE ISTIO AUTHORS, 2025).

2.2 Repositório Git

- Repositório do projeto:
<https://github.com/uuur9tgve84nrandomorgxvtk9932kk/cluster-kind>

Capítulo 3

Instalações

A seguir, são descritos os procedimentos realizados para configurar o ambiente necessário à execução dos experimentos do laboratório.

3.1 Instalação do Docker

1. Adicionar repositório Docker CE e instalar runtime:

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -
o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/
keyrings/docker.asc] https://download.docker.com/linux/
ubuntu \
$(. /etc/os-release && echo "${UBUNTU_CODENAME:-
$VERSION_CODENAME}") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update
```

3.2 Instalação do *kubectl*

1. Baixar binário compatível (v1.32.0) (THE KUBERNETES AUTHORS, 2025b):

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.
k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
```

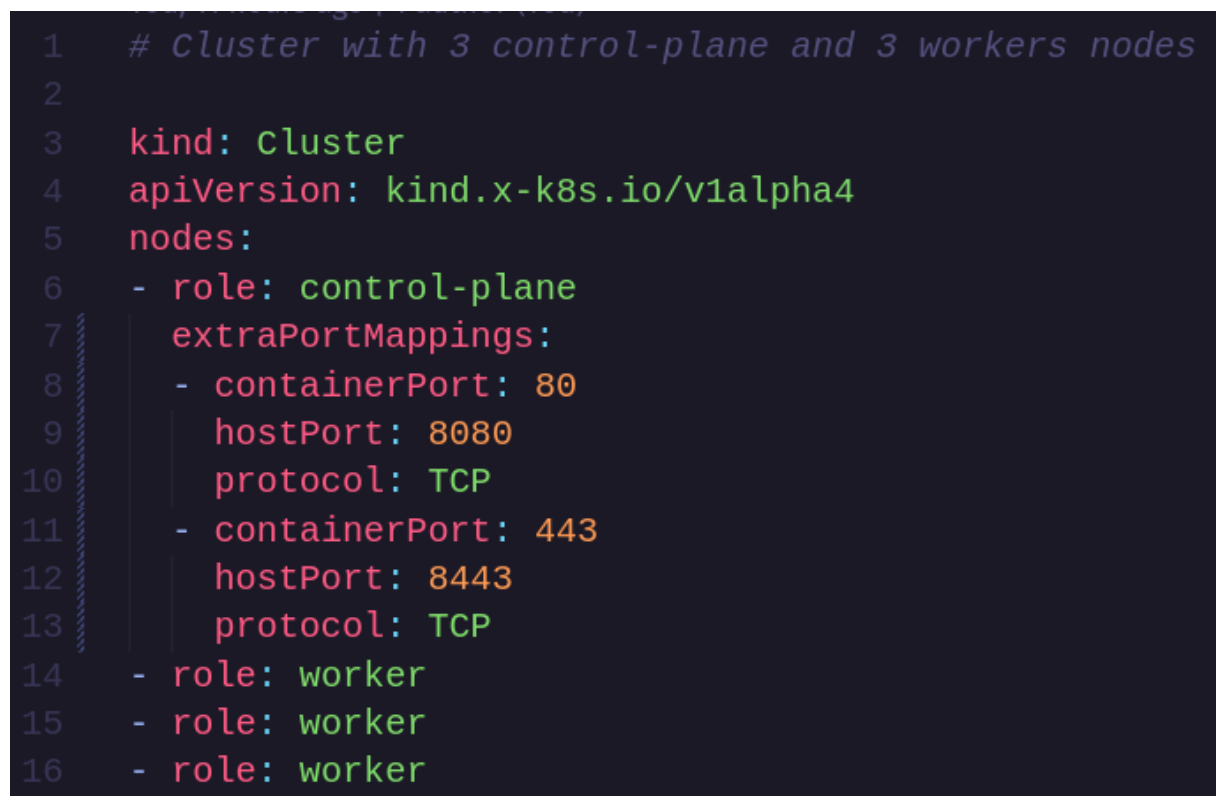
3.3 Instalação do Kind

1. Baixar Kind v0.29.0 (THE KUBERNETES AUTHORS, 2025a):

```
[ $(uname -m) = x86_64 ] && curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.29.0/kind-linux-amd64
chmod +x ./kind
sudo mv ./kind /usr/local/bin/kind
```

2. Criar cluster Kind:

```
kind create cluster --config kind-config.yaml
```



```
1  # Cluster with 3 control-plane and 3 workers nodes
2
3  kind: Cluster
4  apiVersion: kind.x-k8s.io/v1alpha4
5  nodes:
6  - role: control-plane
7    extraPortMappings:
8      - containerPort: 80
9        hostPort: 8080
10       protocol: TCP
11      - containerPort: 443
12        hostPort: 8443
13        protocol: TCP
14  - role: worker
15  - role: worker
16  - role: worker
```

Figura 3.1: Evidência: Arquivo de configuração do Kind (kind-config.yaml).

3.4 Instalação do Istio

1. Download e instalação do Istioctl v1.25.2:

```
curl -L https://istio.io/downloadIstio | sh -
export PATH="$PATH:$HOME/istio-1.25.2/bin"
istioctl install --set profile=minimal -y
```

3.5 Instalação do Skaffold

1. Download e instalação do Skaffold v2.15.0:

```
curl -Lo skaffold https://storage.googleapis.com/skaffold/
releases/latest/skaffold-linux-amd64 && \
sudo install skaffold /usr/local/bin/
```

3.6 Instalação do K9S

1. Download e instalação do K9S v0.50.4:

```
# Via LinuxBrew
brew install derailed/k9s/k9s
# Via PacMan
pacman -S k9s
```

3.7 Instalação do Kustomize

1. Download e instalação do Kustomize v5.6.0:

```
curl -s "https://raw.githubusercontent.com/kubernetes-sigs/
kustomize/master/hack/install_kustomize.sh" | bash
```

3.8 Implantação da Aplicação Online Boutique

1. Clonar repositório e implantar versão padrão (GOOGLE CLOUD PLATFORM, 2025):

```
git clone --recurse-submodules https://github.com/
uue9tgve84nrandomorgxvtk9932kk/cluster-kind
kubectl create namespace online-boutique
cd cluster-kind/microservices-demo/
skaffold run -n online-boutique
```

Context: kind-kind
Cluster: kind-kind
User: kind-kind
K8s Rev: v0.50.4 v0.50.6
K8s Rev: v1.32.2
CPU: n/a
MEM: n/a

<0> all
<1> kube-system
<2> default
<3> ingress-nginx
<4> online-boutique

<a> Attach
<ctrl-d> Delete
<d> Describe
<e> Edit
<shift-j> Jump Owner

<ctrl-k> Kill
<l> Logs
<p> Logs Previous
<shift-f> Port-Forward
<z> Sanitize
<s> Shell

<o> Show Node
<f> Show PortForward
<t> Transfer
<y> YAML



NAMEs	PF	READY	STATUS	RESTARTS	IP	NODE	AGE
adservice-784c8769b6-lrg76	•	1/1	Running	1	10.244.1.5	kind-worker	10h
cartservice-bbcf78958-dc45w	•	1/1	Running	1	10.244.2.12	kind-worker3	10h
checkoutservice-64f9656776-j65j7	•	1/1	Running	1	10.244.2.2	kind-worker3	10h
currencyservice-f59f4899f-4v5cs	•	1/1	Running	2	10.244.2.3	kind-worker3	10h
emailservice-9c9b65755-dhwz	•	1/1	Running	1	10.244.1.9	kind-worker	10h
frontend-8882b4f9b-40crq	•	1/1	Running	1	10.244.1.10	kind-worker	10h
loadgenerator-57c8d8d66-fp4bx	•	1/1	Running	0	10.244.3.4	kind-worker2	7h10m
payment-service-8677cfc5c5-68mr7	•	1/1	Running	2	10.244.2.4	kind-worker3	10h
productcatalogservice-775f78dbc9-q2nwp	•	1/1	Running	1	10.244.1.2	kind-worker	10h
recommendationservice-554b554cd6-4t9sj	•	1/1	Running	1	10.244.2.5	kind-worker3	10h
redis-cart-554c549f8d-g57vq	•	1/1	Running	1	10.244.1.8	kind-worker	10h
shippingservice-bb6fb9759-f282v	•	1/1	Running	1	10.244.3.2	kind-worker2	10h

<pod>

Viewing v1/pods in namespace 'online-boutique'...

Figura 3.2: Evidência: Instalação da aplicação Online Boutique.

2. Configuração do Ingress para versão sem Istio:

```
kubectl apply -f https://kind.sigs.k8s.io/examples/ingress/
  deploy-ingress-nginx.yaml

kubectl apply -n online-boutique -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend-ingress
spec:
  rules:
  - http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: frontend
          port:
            number: 80
EOF
```

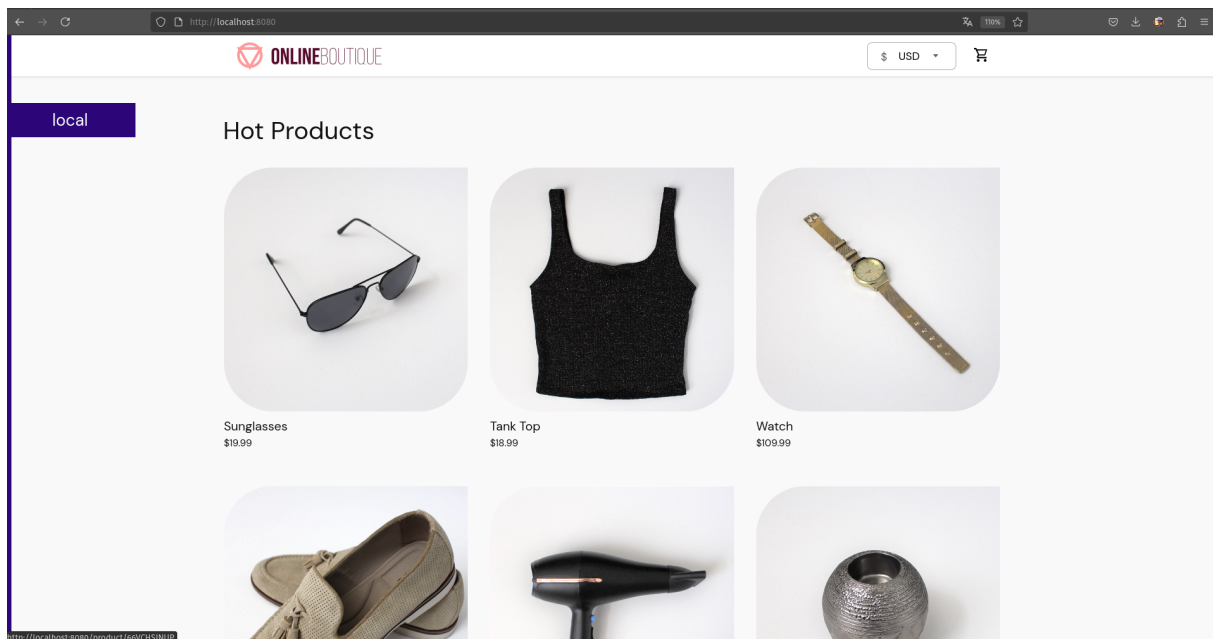


Figura 3.3: Evidência: Acesso via navegador à aplicação Online Boutique (sem Istio).

3. Implantar versão com Istio:

```
kubectl label namespace default istio-injection=enabled
cd cluster-kind/microservices-demo/
kustomize edit add component components/service-mesh-istio
kubectl apply -k .
```

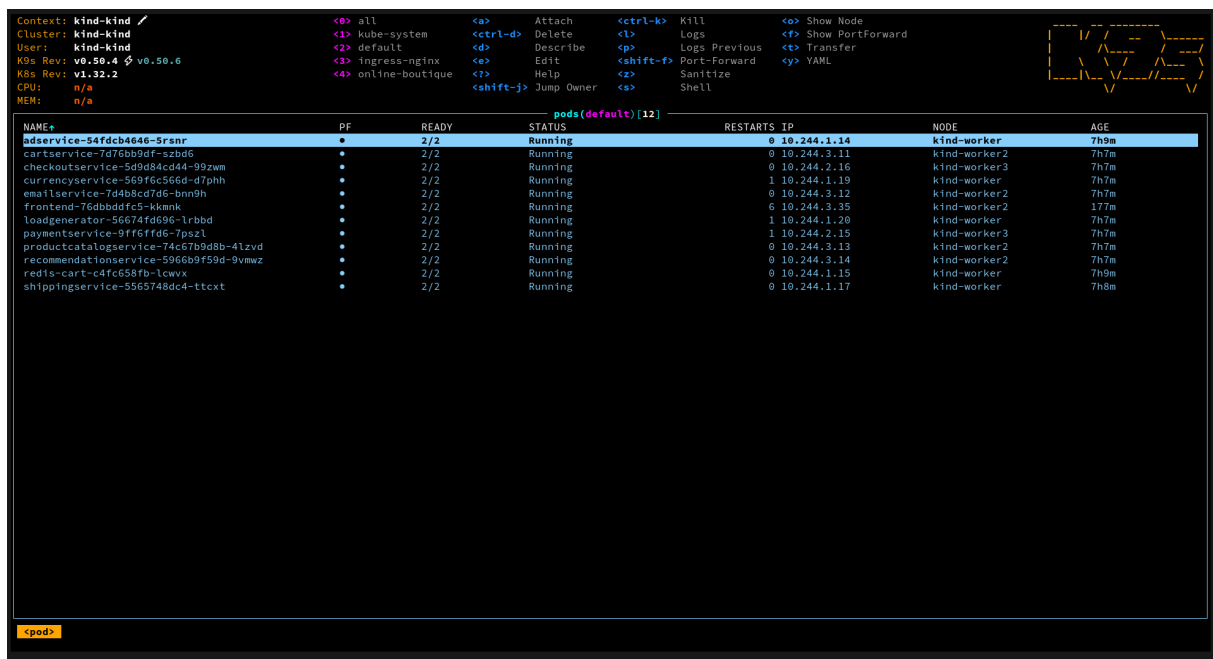


Figura 3.4: Evidência: Aplicação Online Boutique com Istio.

4. Configuração do Ingress com Istio:

```
kubectl apply -n default -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend-ingress
spec:
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend
            port:
              number: 80
EOF
```

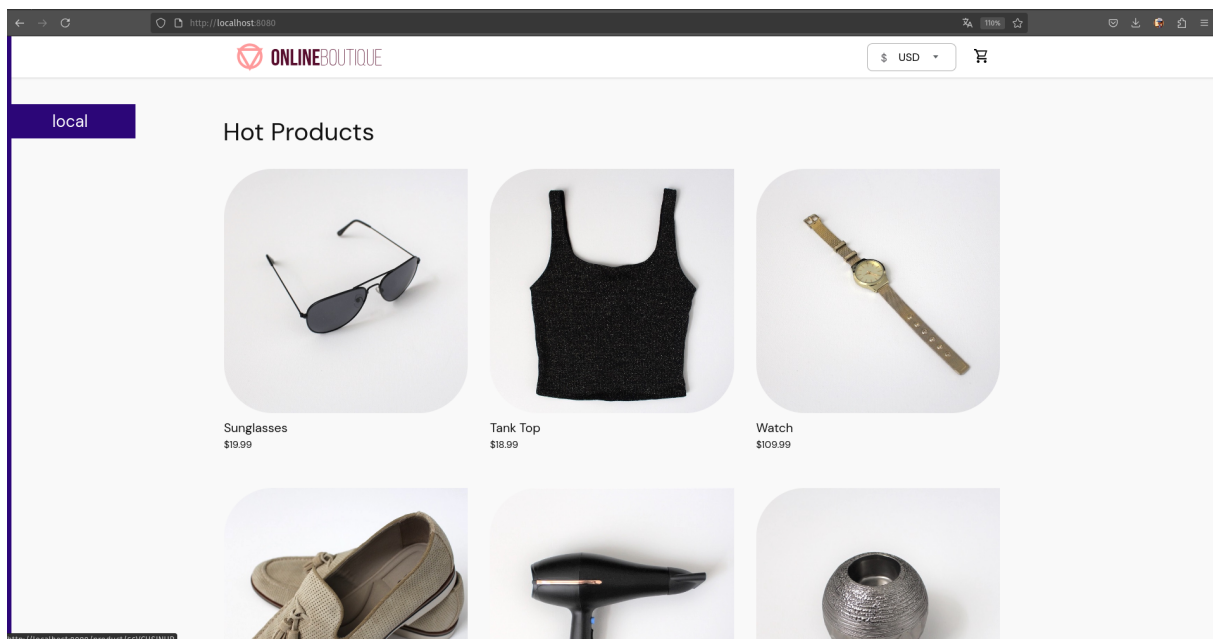


Figura 3.5: Evidência: Acesso via navegador à aplicação Online Boutique (com Istio).

3.9 Instalação do Locust

1. Criação do ambiente virtual:

```
python3 -m venv venv
source venv/bin/activate
```

2. Instalação do Locust:

```
pip install locust
```

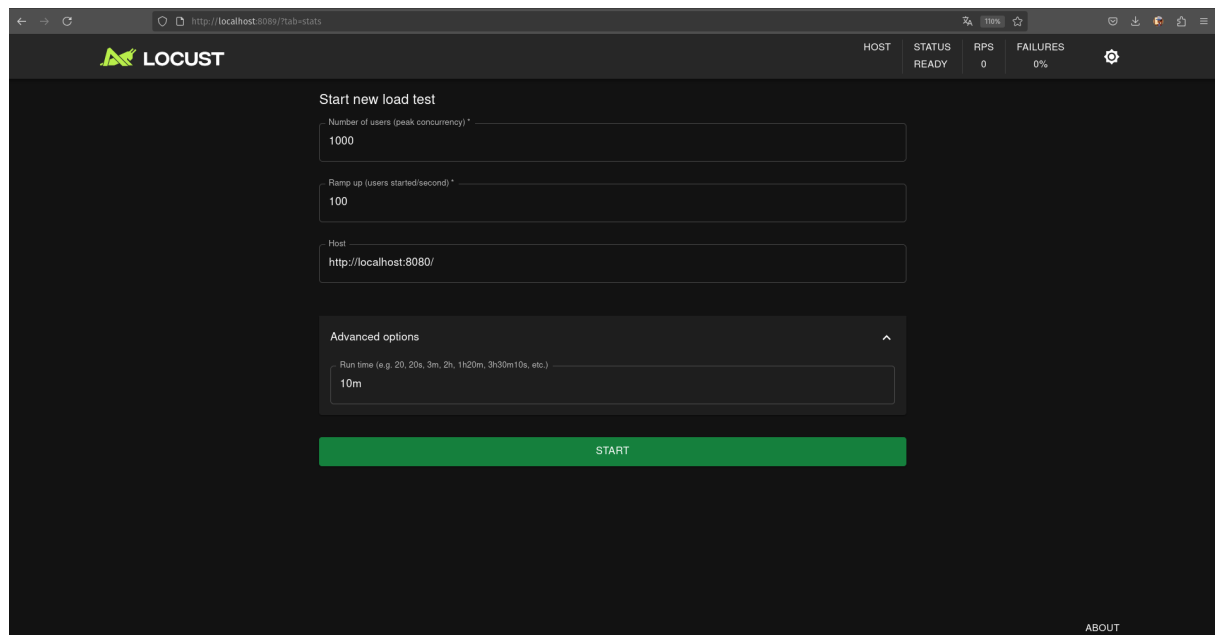


Figura 3.6: Evidência: Instalação do Locust.

Capítulo 4

Testes de Carga

4.1 Realização dos Testes

Os testes de carga foram realizados utilizando a ferramenta *Locust*, a partir do seguinte procedimento:

1. Acessar o diretório do script Locust:
`cluster-kind/microservices-demo/src/loadgenerator`
2. Executar o comando **locust** a partir deste diretório.
3. Foram realizados os seguintes cenários de testes:
 - (a) Máximo de 1000 usuários, com taxa de crescimento de 100 usuários/s.
 - (b) Máximo de 1250 usuários, com taxa de crescimento de 100 usuários/s.
 - (c) Máximo de 1500 usuários, com taxa de crescimento de 100 usuários/s.
4. Cada cenário foi executado 10 vezes para as duas versões da aplicação (com e sem Istio).

O script utilizado para simulação dos testes está descrito abaixo:

Listing 4.1: Script locust.py utilizado nos testes de carga.

```
1 #!/usr/bin/python
2
3 import random
4 from locust import FastHttpUser, TaskSet, between
5 from faker import Faker
6 import datetime
7 fake = Faker()
8
9 products = [
10     '0PUK6V6EVO', '1YMWWN1N4O', '2ZYFJ3GM2N', '66VCHSJNUP',
11     '6E92ZMYFZ', '9SIQT8TOJO', 'L9ECAV7KIM', 'LS4PSXUNUM', '
12     OLJCESPC7Z'
13 ]
```

```

14 def index(l):
15     l.client.get("/")
16
17 def setCurrency(l):
18     currencies = ['EUR', 'USD', 'JPY', 'CAD', 'GBP', 'TRY']
19     l.client.post("/setCurrency", {'currency_code': random.choice(
20         currencies)})
21
22 def browseProduct(l):
23     l.client.get("/product/" + random.choice(products))
24
25 def viewCart(l):
26     l.client.get("/cart")
27
28 def addToCart(l):
29     product = random.choice(products)
30     l.client.get("/product/" + product)
31     l.client.post("/cart", {
32         'product_id': product,
33         'quantity': random.randint(1,10)})
34
35 def empty_cart(l):
36     l.client.post('/cart/empty')
37
38 def checkout(l):
39     addToCart(l)
40     current_year = datetime.datetime.now().year+1
41     l.client.post("/cart/checkout", {
42         'email': fake.email(),
43         'street_address': fake.street_address(),
44         'zip_code': fake.zipcode(),
45         'city': fake.city(),
46         'state': fake.state_abbr(),
47         'country': fake.country(),
48         'credit_card_number': fake.credit_card_number(card_type="
49             visa"),
50         'credit_card_expiration_month': random.randint(1, 12),
51         'credit_card_expiration_year': random.randint(current_year,
52             current_year + 70),
53         'credit_card_cvv': f"{random.randint(100, 999)}",
54     })
55
56 def logout(l):
57     l.client.get('/logout')
58
59 class UserBehavior(TaskSet):
60     def on_start(self):
61         index(self)
62         tasks = {
63             index: 1,
64             setCurrency: 2,

```

```

62     browseProduct: 10,
63     addToCart: 2,
64     viewCart: 3,
65     checkout: 1
66 }
67
68 class WebsiteUser(FastHttpUser):
69     tasks = [UserBehavior]
70     wait_time = between(1, 10)

```

4.2 Resultados Obtidos

A figura a seguir apresenta um exemplo dos resultados gerados após a execução dos testes:

Usuários		Tempo Médio Sem Istio (ms)	Taxa de Erro Sem Istio (%)	Throughput Sem Istio (req/s)	Tempo Médio Com Istio (ms)	Taxa de Erro Com Istio (%)	Throughput Com Istio (req/s)
0	1000	4556.50 ± 743.11	16.12%	2195.39	5175.63 ± 662.04	2.61%	2059.51
1	1250	5134.29 ± 870.63	19.09%	2574.08	5259.94 ± 883.54	17.90%	2543.56
2	1500	4434.32 ± 972.71	30.97%	3322.18	5074.59 ± 1065.67	28.25%	3099.17

Figura 4.1: Tabela de resultados obtidos a partir dos testes de carga.

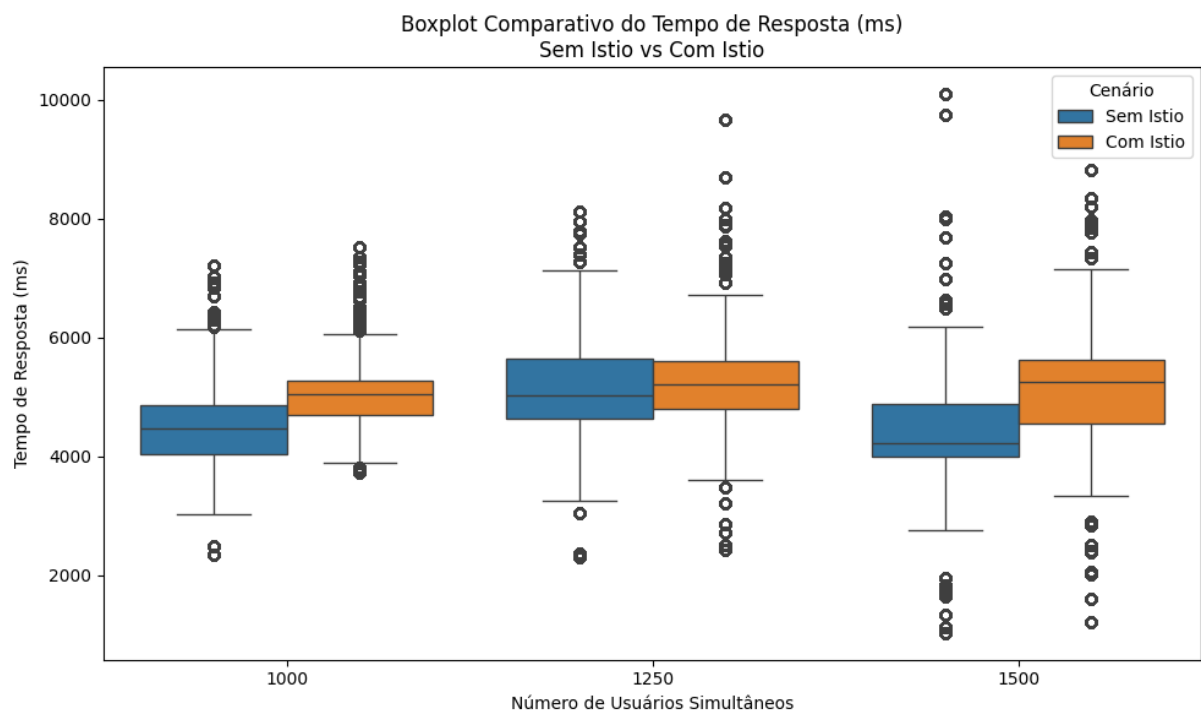


Figura 4.2: Enter Boxplot comparativo do tempo de resposta (ms)

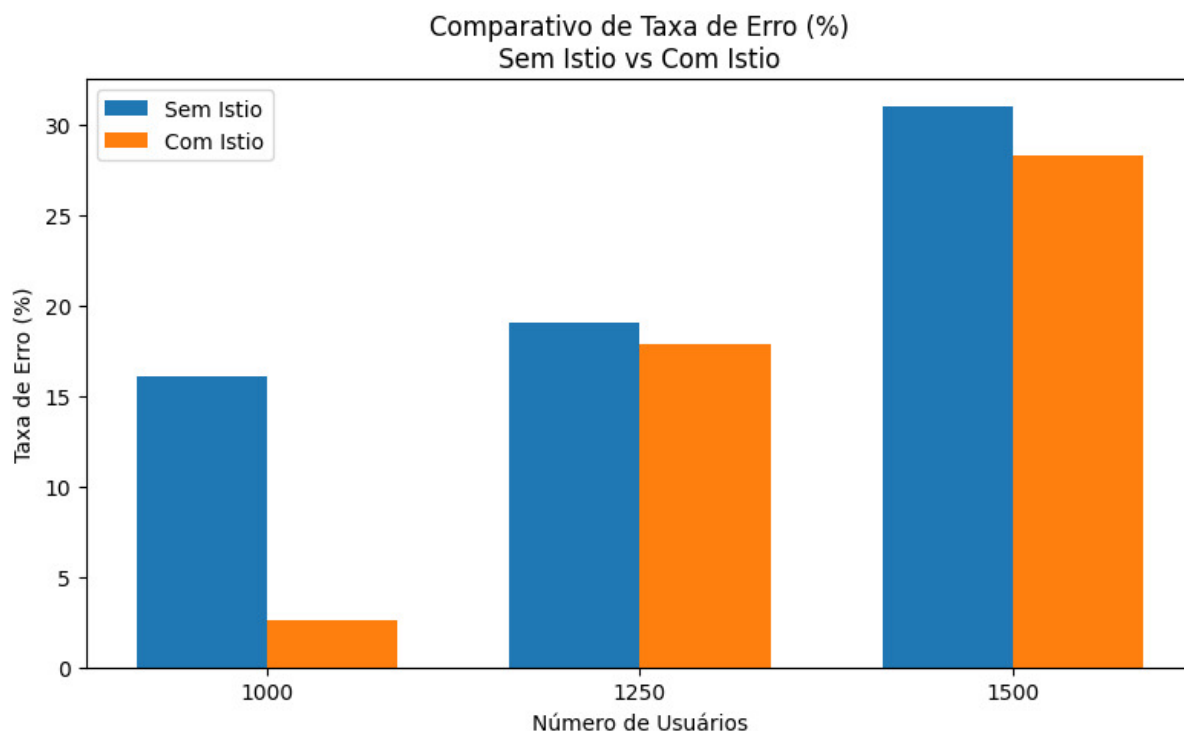


Figura 4.3: Comparativo de taxa de erro (%)

4.3 Análise dos Resultados dos Testes de Carga

Os testes de carga realizados no sistema *Online Boutique* tiveram como objetivo avaliar o comportamento da aplicação sob diferentes níveis de estresse, considerando dois cenários principais: **com Istio** e **sem Istio**. Para isso, foram conduzidas 10 execuções independentes para cada carga de usuários (1000, 1250 e 1500 usuários simultâneos), totalizando 60 execuções controladas.

A ferramenta utilizada para a simulação de carga foi o **Locust**, que gerou métricas fundamentais como **tempo médio de resposta**, **taxa de erro** e **throughput** (requisições por segundo). Além dos dados agregados exibidos em tabelas, também foram gerados arquivos detalhados de log (.csv) contendo informações por endpoint, analisadas individualmente como parte do processo exploratório.

4.3.1 Tempo Médio de Resposta

A primeira métrica avaliada foi o tempo médio de resposta das requisições, com e sem o uso do Istio. Os valores obtidos (em milissegundos) estão resumidos abaixo:

Nº de Usuários	Sem Istio	Com Istio
1000	4556,50 ± 743,11	5175,63 ± 662,04
1250	5134,29 ± 870,63	5259,94 ± 883,54
1500	4434,32 ± 972,71	5074,59 ± 1065,67

Observa-se que, em todos os cenários, o uso do Istio introduziu uma sobrecarga média de aproximadamente 500 a 700 milissegundos no tempo de resposta. Essa diferença é

explicada pela inserção de *sidecars* no tráfego de rede, que adicionam camadas de proxy e roteamento interno.

O comportamento do tempo médio ao aumentar a carga é: sem Istio, o tempo médio tende a diminuir levemente ao se aproximar de 1500 usuários, o que pode indicar saturação de falhas (muitas requisições falhando rapidamente). Já com Istio, os tempos se mantêm mais estáveis e altos, com crescimento moderado sob carga.

4.3.2 Taxa de Erro das Requisições

A taxa de erro representa a proporção de requisições que não foram concluídas com sucesso. Os dados obtidos são os seguintes:

Nº de Usuários	Taxa de Erro (%) - Sem Istio	Taxa de Erro (%) - Com Istio
1000	16,12%	2,61%
1250	19,09%	17,90%
1500	30,97%	28,25%

Este foi o indicador que mais evidenciou as vantagens da utilização do Istio. Com 1000 usuários, a taxa de erro com Istio foi quase 6 vezes menor do que no cenário sem a malha de serviço. Isso demonstra um benefício claro do Istio em ambientes com carga moderada, atuando de forma eficaz na estabilização do tráfego, gerenciamento de conexões e balanceamento inteligente de carga.

Contudo, à medida que a carga se aproxima de 1500 usuários, essa vantagem diminui: ambos os cenários apresentam taxas de erro próximas de 30%. Isso sugere que, a partir de determinado ponto de saturação, os recursos computacionais ou o próprio cluster passam a ser o gargalo, não apenas a presença ou ausência do Istio.

4.3.3 Throughput (Requisições por Segundo)

A métrica de throughput indica o volume médio de requisições processadas por segundo durante os testes. Os resultados são apresentados abaixo:

Nº de Usuários	Throughput Sem Istio (req/s)	Throughput Com Istio (req/s)
1000	2195,39	2059,51
1250	2574,08	2543,56
1500	3322,18	3099,17

Conforme esperado, o throughput foi consistentemente maior no cenário sem Istio, com diferenças de até 200 requisições por segundo nos testes com 1500 usuários. Esse comportamento é coerente com a sobrecarga introduzida pelo Istio, que reprocessa cada requisição por meio dos *proxies sidecar* e executa políticas adicionais de rede, segurança e observabilidade.

Entretanto, a diferença de throughput entre os cenários diminui proporcionalmente com o aumento da carga, o que demonstra que o Istio consegue escalar razoavelmente bem, mesmo sob estresse elevado.

4.3.4 Análise Gráfica

Boxplot dos Tempos de Resposta

O gráfico boxplot dos tempos de resposta, segmentado por número de usuários, evidencia que:

- Os tempos de resposta no cenário com Istio apresentam maior mediana e mais variabilidade (whiskers mais longos e mais *outliers*);
- A presença de *outliers* muito altos (acima de 10.000 ms) em ambos os cenários é comum, mas mais frequente com o Istio, especialmente sob maior carga.

Gráfico de Barras da Taxa de Erro

O gráfico de barras deixa clara a expressiva vantagem do Istio com 1000 usuários, com uma taxa de erro bastante reduzida. À medida que a carga aumenta, a diferença entre os cenários se torna menor, até praticamente se igualar com 1500 usuários.

4.3.5 Análise dos Arquivos de Log (Exploratória)

A análise dos arquivos `requests.csv` de cada teste revelou que os endpoints mais acessados e críticos em termos de latência e falhas foram:

- GET /
- GET /cart
- POST /cart
- POST /cart/checkout
- GET /product/{id}

Requisições de escrita, como POST /cart e POST /cart/checkout, apresentaram os maiores índices de falha, especialmente no cenário sem Istio. Já as requisições do tipo GET tendem a ter menor variabilidade de tempo de resposta, mas enfrentam picos consideráveis em alguns testes (tempo máximo acima de 60 segundos em casos isolados).

O campo de percentis (66%, 75%, 95%, 99%) nos arquivos reforça a conclusão de que a maioria das requisições é concluída em menos de 10 segundos, mas há picos severos que impactam a média.

4.3.6 Considerações Finais

A partir dos resultados observados, é possível concluir que o uso do Istio traz benefícios importantes de controle de tráfego, confiabilidade e robustez, sobretudo em cenários com carga moderada (até 1250 usuários). O sistema apresentou menores taxas de erro e maior estabilidade sob carga média, mesmo com pequena penalização de desempenho (tempo de resposta e throughput).

Todavia, em cargas muito elevadas, os benefícios do Istio tendem a se diluir, exigindo configurações avançadas de escalabilidade no cluster, ajustes finos de política de tráfego, e provisionamento adequado de recursos.

O uso de *service mesh* como o Istio, portanto, deve ser planejado estrategicamente com base no perfil esperado de tráfego da aplicação e na capacidade de infraestrutura disponível, ponderando os ganhos de resiliência frente ao custo computacional e de latência adicional.

Capítulo 5

Teste de Resiliência

5.1 Cenário de teste

Nesse teste foram adicionados 3 arquivos de VirtualService do istio. Foram considerados dois tipos de falhas:

- **Injeção de atraso:** um tempo de espera fixo de 2 segundos foi inserido nas respostas do serviço `recommendation`.
- **Injeção de erro:** respostas HTTP 503 (*Service Unavailable*) foram retornadas de forma artificial para 25% e 50% das requisições direcionadas ao serviço `productcatalog`.

A ferramenta utilizada foi o Locust, onde mantemos uma taxa fixa de 1000 usuários, com taxa de crescimento de 100 usuários/s, com as taxas de 25% e 50% e usamos os dados coletados no capítulo de teste de carga. Os dados coletados foram analisados de forma exploratória entre os cenários.

5.2 Arquivos de injeção de falha controlada

```
1  apiVersion: networking.istio.io/v1beta1
2  kind: VirtualService
3  metadata:
4    name: productcatalog-abort-25
5    namespace: default
6  spec:
7    hosts:
8      - productcatalogservice.default.svc.cluster.local
9    http:
10     - fault:
11       abort:
12         httpStatus: 503
13         percentage: { value: 25 }
14       route:
15         - destination:
16           host: productcatalogservice.default.svc.cluster.local
```

Figura 5.1: Arquivo VirtualService para 25% de falha productcatalogservice

```
1  apiVersion: networking.istio.io/v1beta1
2  kind: VirtualService
3  metadata:
4    name: productcatalog-abort-50
5    namespace: default
6  spec:
7    hosts:
8      - productcatalogservice.default.svc.cluster.local
9    http:
10     - fault:
11       abort:
12         httpStatus: 503
13         percentage: { value: 50 }
14       route:
15         - destination:
16           host: productcatalogservice.default.svc.cluster.local
```

Figura 5.2: Arquivo VirtualService para 50% de falha productcatalogservice


```

1  you, 1 second ago | I authored (you)
2  apiVersion: networking.istio.io/v1beta1
3  kind: VirtualService
4  metadata:
5    name: recommendation-delay
6    namespace: default
7  spec:
8    hosts:
9      - recommendation-service.default.svc.cluster.local
10   http:
11     - fault:
12       delay:
13         fixedDelay: 2s # atraso
14         percentage: { value: 100 }
15     route:
16       - destination:
17         host: recommendation-service.default.svc.cluster.local

```

Figura 5.3: Arquivo VirtualService para delay de 2s recommendation-service

5.3 Resultados Obtidos

A Tabela 5.1 apresenta os dados resumidos de desempenho obtidos nas duas execuções: uma com falhas induzidas via Istio e outra sem injeção de falhas explícita.

Tabela 5.1: Métricas de Resiliência – Injeção de Falhas vs. Execução Base

Cenário	Reqs	Falhas	Erro (%)	Avg (ms)	Mediana (ms)	Máx (ms)	RPS
Com falhas	65 462	21 963	33,4%	4 658,46	4 900	16 564	109.09
Sem falhas	62 836	7 485	11,91%	5 049,07	5 100	18 377	104.71

5.4 Análise dos Resultados

5.4.1 Análise das Métricas de Resiliência

A Tabela 5.1 apresenta uma comparação entre dois cenários distintos da aplicação *Online Boutique*: um com injeção de falhas controladas e outro representando a execução base, sem falhas. O objetivo dos testes foi avaliar o impacto da resiliência na performance do sistema, observando como a aplicação se comporta sob condições adversas.

Volume de Requisições e Taxa de Erros

No cenário com falhas, foram processadas 65 462 requisições, das quais 21 963 resultaram em falha, o que corresponde a uma taxa de erro de 33,4%. Já na execução base, sem injeção de falhas, foram registradas 62 836 requisições com 7 485 falhas, resultando em uma taxa de erro significativamente menor, de 11,91%.

Embora o aumento da taxa de erro em presença de falhas seja esperado, é relevante destacar que, mesmo sob falhas induzidas, cerca de 66% das requisições foram concluídas com sucesso. Isso sugere que mecanismos de tolerância a falhas, como *circuit breakers* ou *retries*, podem estar parcialmente funcionando.

Tempo de Resposta Médio e Mediano

De maneira interessante, o tempo médio de resposta no cenário com falhas foi menor (4 658,46 ms) do que no cenário sem falhas (5 049,07 ms). A mediana também seguiu esse padrão, com 4 900 ms para o cenário com falhas e 5 100 ms para o cenário base. Essa

diferença pode ser explicada pelo comportamento das requisições que falham rapidamente, sendo interrompidas antes de percorrer todo o fluxo normal de execução, o que reduz o tempo total de processamento.

Tempo Máximo de Resposta

No que diz respeito ao tempo máximo de resposta, observou-se um valor maior no cenário sem falhas (18 377 ms) em comparação ao cenário com falhas (16 564 ms). Isso sugere que, na ausência de falhas, algumas requisições chegaram próximas ao limite de tempo permitido (timeout), enquanto com falhas essas requisições possivelmente foram encerradas antecipadamente por mecanismos de proteção.

Throughput (RPS)

O throughput, medido em requisições por segundo (RPS), também foi levemente superior no cenário com falhas (109,09 req/s) em relação ao cenário sem falhas (104,71 req/s). Esse resultado é coerente com a presença de requisições que falham rapidamente, permitindo ao sistema liberar recursos mais rapidamente e atender um volume maior por unidade de tempo.

5.5 Conclusão

A análise dos resultados evidencia que o sistema apresentou um comportamento parcialmente resiliente frente à injeção de falhas. Mesmo com o aumento significativo da taxa de erro para 33,4%, a aplicação foi capaz de manter níveis aceitáveis de desempenho em termos de tempo médio de resposta e throughput, indicando que falhas estão sendo tratadas de forma rápida e, possivelmente, superficial por mecanismos como *timeouts*, rejeições imediatas ou interrupções de fluxo via *circuit breakers*.

A redução dos tempos médios e medianos no cenário com falhas sugere que muitas requisições são encerradas antes de completar seu ciclo normal, o que diminui o tempo total de processamento, mas impacta negativamente a taxa de sucesso. O throughput levemente superior nesse cenário reforça a hipótese de que o sistema está liberando recursos rapidamente em caso de erro, permitindo maior volume de requisições por segundo — ainda que com menor eficácia.

Entretanto, o aumento de mais de 20 pontos percentuais na taxa de erro em relação ao cenário base revela que os mecanismos de resiliência existentes ainda são limitados em termos de abrangência e profundidade. Ausência de *fallbacks* funcionais, replicação insuficiente de serviços críticos ou ausência de cache local podem estar entre as causas.

Portanto, conclui-se que o sistema demonstra resiliência básica, mas que há margem significativa para evolução. Estratégias como replicação horizontal, degradação graciosa de funcionalidades não críticas, aplicação de *retries* com limites progressivos e utilização de mecanismos de observabilidade para detecção proativa de falhas são recomendadas para ampliar a robustez e garantir continuidade de serviço mesmo em cenários de falha parcial ou intermitente.

Capítulo 6

Teste de Escalonamento Automático usando HPA

6.1 Cenário de Teste

Nessa etapa do laboratorio foi criado um cenario para analise do Horizontal Pod Autoscaler (HPA), workload nativo do Kubernetes (THE KUBERNETES AUTHORS, 2025b) capaz de aumentar o numero de replicas de pods de acordo com uma "meta" de consumo de algum recurso, especificamente memoria ou cpu, alterando diretamente os deployments ou workloads parecidos.

Diante desse cenário, configuramos HPA para 4 serviços que mais demandam recursos da aplicação Online Boutique, sendo eles:

- **Frontend**
- **Checkout Service**
- **Product Catalog Service**
- **Recommendation Service**

E aplicamos uma meta de 70% de cpu com um número de réplicas inicial de 2 pods chegando a um numero maximo de replicas de 6 pods.

A carga foi gerada por meio da ferramenta **Locust** (HEYMAN; HOLMBERG; BALDWIN, 2025; DEVELOPERS, 2025), simulando 1250 usuários acessando simultaneamente com uma variação de requisições por segundo. O número de réplicas foi monitorado via cli do kubernetes (kubectl), onde conseguimos visualizar alteração do numero e o consumo de cpu dos pods.

Para fins de análise dois cenários foram comparados:

- Com HPA habilitado.
- Com número fixo de réplicas.

6.2 Arquivos utilizados

```
1  apiVersion: autoscaling/v2
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: checkout-hpa
5  spec:
6    scaleTargetRef:
7      apiVersion: apps/v1
8      kind: Deployment
9      name: checkoutservice
10   minReplicas: 2
11   maxReplicas: 6
12   metrics:
13   - type: Resource
14     resource:
15       name: cpu
16       target:
17         type: Utilization
18         averageUtilization: 70
19
```

Figura 6.1: Arquivo HPA Checkout Service

```
1  apiVersion: autoscaling/v2
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: frontend-hpa
5  spec:
6    scaleTargetRef:
7      apiVersion: apps/v1
8      kind: Deployment
9      name: frontend
10   minReplicas: 2
11   maxReplicas: 6
12   metrics:
13   - type: Resource
14     resource:
15       name: cpu
16       target:
17         type: Utilization
18         averageUtilization: 70
19
```

Figura 6.2: Arquivo HPA Frontenf

```

1  apiVersion: autoscaling/v2
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: recomendation-hpa
5  spec:
6    scaleTargetRef:
7      apiVersion: apps/v1
8      kind: Deployment
9      name: recommendationservice
10   minReplicas: 2
11   maxReplicas: 6
12   metrics:
13   - type: Resource
14     resource:
15       name: cpu
16       target:
17         type: Utilization
18         averageUtilization: 70

```

Figura 6.4: Arquivo HPA Recommendation Service

```

1  apiVersion: autoscaling/v2
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: productcatalog-hpa
5  spec:
6    scaleTargetRef:
7      apiVersion: apps/v1
8      kind: Deployment
9      name: productcatalogservice
10   minReplicas: 2
11   maxReplicas: 6
12   metrics:
13   - type: Resource
14     resource:
15       name: cpu
16       target:
17         type: Utilization
18         averageUtilization: 70

```

Figura 6.3: Arquivo HPA Product Catalog Service

6.3 Resultados Obtidos

A Tabela 6.1 apresenta os dados resumidos dos testes de desempenho com e sem o uso do HPA.

Tabela 6.1: Métricas de Desempenho – Com e Sem HPA

Cenário	Reqs	Falhas	Erro (%)	Avg (ms)	Mediana (ms)	Máx (ms)	RPS
Com HPA	116 685	29	0,025%	1 875.24	1 200	14 503	194.43
Sem HPA	83 370	35 483	42,56%	4 469.62	4 700	29 381	138.94

6.4 Análise das Métricas de Desempenho – Com e Sem HPA

A Tabela 6.1 apresenta um comparativo entre dois cenários de execução da aplicação *Online Boutique*: um com utilização de *Horizontal Pod Autoscaler* (HPA) e outro sem qualquer mecanismo automático de escalonamento.

Volume de Requisições e Taxa de Erro

No cenário com HPA, foram processadas 116 685 requisições com apenas 29 falhas, resultando em uma taxa de erro de apenas 0,025%. Em contraste, o cenário sem HPA processou 83 370 requisições, das quais 35 483 falharam — uma taxa de erro alarmante de 42,56%. Esses dados evidenciam que o uso do HPA teve impacto direto na robustez da aplicação, permitindo que o sistema absorvesse a carga de forma estável, enquanto no cenário sem escalonamento houve colapso parcial na capacidade de atendimento.

Tempos de Resposta

O tempo médio de resposta no cenário com HPA foi significativamente menor: 1 875,24 ms contra 4 469,62 ms no cenário sem HPA. A mediana seguiu a mesma tendência, com 1 200 ms (com HPA) contra 4 700 ms (sem HPA). Isso indica que, além de reduzir falhas, o escalonamento automático proporcionou uma experiência de resposta muito mais rápida para o usuário final.

Tempo Máximo de Resposta

O tempo máximo de resposta também foi drasticamente reduzido com o uso do HPA: 14 503 ms contra 29 381 ms no cenário sem escalonamento. Isso mostra que, mesmo nos piores casos, o HPA contribuiu para mitigar picos de latência severa, distribuindo a carga entre múltiplas réplicas de forma mais eficaz.

Throughput (RPS)

Por fim, o throughput médio (medido em requisições por segundo) foi superior com HPA: 194,43 req/s frente a 138,94 req/s sem HPA. Esse aumento de cerca de 40% no volume de requisições processadas por segundo reforça que o sistema foi capaz de escalar horizontalmente de maneira eficiente para lidar com a demanda.

6.5 Conclusão

A análise comparativa deixa claro que o uso do *Horizontal Pod Autoscaler* resultou em melhorias expressivas nos principais indicadores de desempenho e confiabilidade do sis-

tema. A redução drástica na taxa de erro, associada a menores tempos de resposta e maior throughput, demonstra que o HPA é uma ferramenta essencial para garantir elasticidade e qualidade de serviço em aplicações baseadas em microsserviços que operam sob carga variável. A ausência desse recurso, por outro lado, compromete diretamente a capacidade do sistema de atender seus usuários com estabilidade e desempenho adequados.

Referências

AN OPEN SOURCE PROJECT FROM GOOGLE. **Scaffold: Container and Kubernetes Development**. [S.l.: s.n.], 2025. <https://skaffold.dev/>. Acesso em 26 jun 2025.

DEVELOPERS, Locust. **Locust - Scalable Load Testing in Python**. [S.l.: s.n.], 2025. <https://github.com/locustio/locust>. Acessado em: 26 jun 2025.

DOCKER, Inc. **Docker Engine & Docker Desktop**. [S.l.: s.n.], 2025. <https://docs.docker.com/>. Acesso em 26 jun 2025.

GOOGLE CLOUD PLATFORM. **Online Boutique (Microservices Demo)**. [S.l.: s.n.], 2025. <https://github.com/GoogleCloudPlatform/microservices-demo>. Acesso em 26 jun 2025.

HEYMAN, Jonatan; HOLMBERG, Lars; BALDWIN, Andrew. **Locust: An Open Source Load Testing Tool**. [S.l.: s.n.], 2025. <https://locust.io/>. Versão 2.37.4. Acessado em: 26 jun 2025.

THE ISTIO AUTHORS. **Istio Service Mesh**. [S.l.: s.n.], 2025. <https://istio.io/>. Acesso em 26 jun 2025.

THE K9S AUTHORS. **K9S: Kubernetes CLI To Manage Your Clusters In Style!** [S.l.: s.n.], 2025. <https://k9scli.io/>. Acesso em 30 jun 2025.

THE KUBERNETES AUTHORS. **Kind: Run Kubernetes Locally**. [S.l.: s.n.], 2025. <https://kind.sigs.k8s.io/>. Acesso em 26 jun 2025.

_____. **Kubernetes: Production-Grade Container Orchestration**. [S.l.: s.n.], 2025. <https://kubernetes.io/>. Acesso em 26 jun 2025.

_____. **Kustomize: Kubernetes native configuration management**. [S.l.: s.n.], 2025. <https://kustomize.io/>. Acesso em 26 jun 2025.