

Relatório - Monitoramento e Observabilidade em Cluster Kubernetes

- Universidade de Brasília
- Disciplina: Programação para Sistemas Paralelos e Distribuídos (PSPD)
- Professor: Fernando W. Cruz
- Turma: 2
- Data da entrega: 07/12/2025
- Link do vídeo de apresentação: <https://www.youtube.com/watch?v=OAVnJvgpjgg>
- Integrantes
 - Guilherme Westphall de Queiroz - 211061805
 - Lucas Martins Gabriel - 221022088

Sumário

[Sumário](#)

[Introdução](#)

[Metodologia](#)

[Configurar Ambiente \(Experiência de Montagem do Kubernetes em Modo Cluster\)](#)

[Escolha das Ferramentas](#)

[Instalação](#)

[Kubectl](#)

[Kind](#)

[Helm](#)

[Prometheus](#)

[Criando o cluster](#)

[Acessando o Prometheus](#)

[API gRPC da aplicação](#)

[Testes](#)

[Configuração adicional](#)

[Metrics-server](#)

[Arquivos YAML para o HAP \(Horizontal Pod Autoscaling\)](#)

[Testes de carga com k6 no cluster Kubernetes](#)

[Monitoramento com Prometheus](#)

[Throughput médio do serviço](#)

[Throughput por endpoint](#)

[Latência média](#)

[Latência P95](#)

[Uso de CPU](#)

[Uso de memória](#)

[Conclusão](#)

[Dificuldades encontradas](#)

[Anexos](#)

[Réplica Stub](#)

[Réplica Geral](#)

[4 Workers](#)

[4 Workers + 3 Rélicas](#)
[Autoscaling \(HPA\)](#)
[Monitoramento HPA](#)
[Referências](#)

Introdução

Para viabilizar os experimentos de desempenho e observabilidade, a aplicação bancária baseada em microserviços foi totalmente conteinerizada e implantada em um cluster Kubernetes dedicado. Cada serviço – `client`, `account`, `stub` (API Gateway) e `postgres` – possui seu próprio `Dockerfile`, permitindo empacotar código, dependências e configuração em imagens isoladas. Essas imagens são utilizadas pelos manifests presentes na pasta `k8s/`, que definem os objetos de `Deployment` e `Service` responsáveis por instanciar os pods de aplicação e expô-los dentro do cluster.

O cluster Kubernetes foi criado com o `kind`, utilizando um arquivo `cluster.yaml` que descreve um plano de controle e múltiplos nós de trabalho (workers). Essa configuração permite simular um ambiente distribuído real, com a aplicação sendo distribuída entre os workers e exposta externamente por meio de um `NodePort`, mapeando a porta 30080 do host para o serviço `stub-service`. Após a criação do cluster, os manifests da aplicação e do banco de dados foram aplicados sequencialmente, resultando em um ambiente consistente para todos os cenários de teste.

Sobre esse ambiente, foram conduzidos testes de carga com a ferramenta `k6`, que simula usuários virtuais exercitando um fluxo completo da aplicação (criação de cliente, criação de conta, login, transações e exclusão). A partir de uma configuração base (baseline), variámos o número de réplicas dos microserviços e a quantidade de nós workers do cluster, sempre mantendo a mesma carga (100 VUs por 10 minutos) para permitir comparações diretas entre os cenários.

O monitoramento foi estruturado com **Prometheus**. O Prometheus foi instalado no namespace `monitoring` via Helm, configurado para coletar métricas dos serviços do namespace da aplicação, em especial do `stub-service`, que expõe contadores de requisições, histogramas de latência e métricas de uso de recursos. Através da UI da ferramenta, é possível acompanhar em tempo real throughput, latência (média e P95), consumo de CPU e memória durante e após a execução dos testes de carga. Dessa forma, o conjunto Kubernetes + kind + k6 + Prometheus fornece uma visão integrada do comportamento da aplicação sob diferentes configurações de contêineres e de cluster.

Metodologia

Todo o desenvolvimento do trabalho foi realizado em **par**, pelos dois integrantes do grupo, adotando uma abordagem de **pair programming** em chamadas pelo Discord. Em vez de dividir o projeto em partes isoladas, ambos participaram ativamente de todas as etapas, compartilhando a mesma tela e discutindo as decisões técnicas em tempo real. Nesse contexto, o grupo passou pela montagem do ambiente (configuração do kind, criação do cluster Kubernetes multi-nó e preparação das imagens Docker dos microserviços), pelo deploy da aplicação e do banco de dados no cluster (incluindo ajustes nos arquivos de Deploy e Service), pela configuração do monitoramento com Prometheus (definição das métricas e queries de interesse), pela implementação e execução dos testes de carga com k6 (bem como a análise dos resultados obtidos) e, por fim, pela elaboração do relatório e do material de apresentação, sempre usando.

Configurar Ambiente (Experiência de Montagem do Kubernetes em Modo Cluster)

Escolha das Ferramentas

Para a execução do trabalho, inicialmente consideramos a possibilidade de montar um **cluster físico real**, distribuído em múltiplas máquinas. Porém, por limitações de infraestrutura, não foi viável configurar e manter um ambiente desse tipo. Diante disso, optamos pelo uso do **kind (Kubernetes in Docker)**, que permite criar um cluster

Kubernetes completo em um único host, com múltiplos nós lógicos (control-plane e workers) definidos em arquivo de configuração. No contexto do nosso ambiente, o kind foi a solução que melhor conciliou a necessidade de **simular um cluster multinode**, com facilidade de recriação dos cenários (basta destruir e recriar o cluster) e repetibilidade dos testes, algo essencial para comparações de desempenho entre diferentes configurações de réplicas e workers.

Para os testes de carga, escolhemos a ferramenta **k6**. Dois fatores principais motivaram essa escolha: a facilidade de uso, já que basta definir um script em JavaScript descrevendo o cenário de teste (fluxo de criação de cliente, conta, login, transações, etc.) e a quantidade de usuários virtuais; e o fato de que o grupo já havia utilizado o k6 em um trabalho anterior, o que reduziu a curva de aprendizado e permitiu focar mais na análise dos resultados do que na configuração da ferramenta em si.

Por fim, para monitoramento e observabilidade, adotamos o Prometheus, que é hoje o padrão de fato no ecossistema Kubernetes. A ferramenta foi utilizada para coletar métricas de aplicação e infraestrutura (requisições, latência, CPU, memória). Essa ferramente se integrou bem ao ambiente montado com kind e ao fluxo de testes com k6, fornecendo a base de dados necessária para as análises apresentadas no relatório.

Instalação

A instalação do ambiente foi relativamente tranquila: em geral, conseguimos seguir praticamente **apenas a documentação oficial** das ferramentas (`kubectl`, kind, Helm e Prometheus) sem grandes dificuldades adicionais. A maior parte do trabalho consistiu em executar os comandos de instalação, validar as versões e, em seguida, aplicar os manifests Kubernetes do projeto. As únicas exceções relevantes foram:

1. a necessidade de **ajustar o Stub** para expor um endpoint de métricas compatível com o Prometheus;
2. um problema com a **pilha Grafana+Prometheus** instalada via Helm, que impediu o uso direto do Prometheus embutido no Grafana. Na prática, só conseguimos realizar o monitoramento usando a instância "simples" do Prometheus (`prometheus-server` no namespace `monitoring`), acessando sua interface nativa e consultando as métricas diretamente por lá.

A seguir detalhamos os principais passos de instalação e configuração.

Kubectl

Para instalar o `kubectl`, que é a ferramenta de linha de comando para interagir com o cluster Kubernetes, executamos os seguintes comandos no terminal:

```
sudo apt update
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Verificar se foi instalado com sucesso:

```
kubectl version --client
```

Kind

A seguir, detalhamos os principais comandos utilizados para instalar o Kind e verificar sua versão instalada:

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/latest/kind-linux-amd64
chmod +x ./kind
sudo mv ./kind /usr/local/bin/kind
```

```
kind --version
```

Helm

O Helm é o gerenciador de pacotes do Kubernetes, facilitando a instalação e configuração de aplicações complexas como o Prometheus. Com ele, é possível definir charts (pacotes) contendo todos os recursos necessários e gerenciar versões de forma simplificada. O seguinte comando instala o Helm:

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```

Testar se o Helm foi instalado com sucesso:

```
helm version
```

Prometheus

Adicionamos o Prometheus ao ambiente de monitoramento instalando-o via Helm no namespace `monitoring`.

```
kubectl create namespace monitoring
```

```
helm install prometheus prometheus-community/prometheus \
--namespace monitoring
```

Após subir o stack, configuramos o `stub-service` para expor métricas HTTP em um endpoint específico (por exemplo, `/metrics`), de forma que o Prometheus pudesse fazer *scrape* das métricas de throughput, latência, CPU e memória. Embora tenhamos tentado integrar essa instância com o Grafana instalado via Helm, enfrentamos problemas de acesso na versão empacotada, e a solução prática adotada foi utilizar a **interface nativa do Prometheus** para executar as queries e gerar os gráficos usados na análise.

Para integrar o Prometheus na aplicação gRPC, foi necessário a criação de um novo serviço e a exposição em um endpoint da aplicação `/metrics`:

```
# metrics.py

from prometheus_client import Counter, Histogram, start_http_server

HTTP_REQUESTS_TOTAL = Counter(
    "http_requests_total",
    "Total de requisições HTTP recebidas pelo stub",
    ["method", "endpoint", "status"]
)

HTTP_REQUEST_DURATION_SECONDS = Histogram(
    "http_request_duration_seconds",
    "Latência das requisições HTTP no stub",
    ["method", "endpoint"],
    buckets=[0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 2, 5]
)

def start_metrics_server(port: int = 8001):
```

```
start_http_server(port)
print(f"Prometheus metrics server rodando em :{port}/metrics")
```

Criando o cluster

O cluster Kubernetes foi criado com o kind a partir do arquivo `cluster.yaml` definido na pasta `k8s`, contendo um nó de controle e dois nós de trabalho, com mapeamento de porta para expor o `stub-service` via NodePort:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
    extraPortMappings:
      - containerPort: 30080
        hostPort: 30080
        protocol: TCP
  - role: worker
  - role: worker
```

Comando para criar o cluster com o Kind:

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

Verificar se os nodes definidos no arquivo foram criados:

```
kubectl get nodes
```

Aplicar os deployments da pasta `k8s`:

- `namespace.yaml`
- `secrets-dev.yaml`
- `postgres.yaml`
- `client_server.yaml`
- `account_server.yaml`
- `stub.yaml`

Exemplo:

```
kubectl apply -f k8s/namespace.yaml
```

Acessando o Prometheus

Para acessar a interface web do Prometheus, primeiro verificamos a porta exposta pelo serviço `prometheus-server` no namespace `monitoring`:

```
kubectl get svc -n monitoring
```

retornando algo como:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
prometheus-alertmanager	ClusterIP	10.96.90.182	<none>	9093/TCP	10m
prometheus-alertmanager-headless	ClusterIP	None	<none>	9093/TCP	10m
prometheus-kube-state-metrics	ClusterIP	10.96.130.69	<none>	8080/TCP	10m
prometheus-prometheus-node-exporter	ClusterIP	10.96.9.161	<none>	9100/TCP	10m
prometheus-prometheus-pushgateway	ClusterIP	10.96.191.6	<none>	9091/TCP	10m
prometheus-server	ClusterIP	10.96.106.219	<none>	80/TCP	10m

```
kubectl port-forward -n monitoring svc/prometheus-server 9090:<PORT>
```

A partir daí, as queries foram executadas diretamente na interface web do Prometheus (<http://localhost:9090>), servindo de base para os gráficos e análises apresentados nas seções de resultados e monitoramento.

API gRPC da aplicação

A aplicação bancária foi construída sobre uma arquitetura de microserviços, em que a **comunicação principal entre os serviços é feita via gRPC**. Nesse contexto, o gRPC funciona como uma camada de RPC de alto desempenho, baseada em HTTP/2 e mensagens serializadas em **Protocol Buffers (Protobuf)**, garantindo chamadas rápidas e tipadas entre os servidores internos.

Dois microserviços principais expõem APIs gRPC:

- **Client Server (Serviço A - porta 50051)**: implementa o `ClientService`, responsável por CRUD de clientes e autenticação (login), persistindo os dados em um banco PostgreSQL via Prisma.
- **Account Server (Serviço B - porta 50052)**: implementa o `AccountService`, responsável por CRUD de contas, operações de envio de dinheiro (`SendMoney`) e listagem de transações, também com PostgreSQL e Prisma.

Para consumo externo, o sistema expõe um **API Gateway (Serviço P - Stub)** em Python com FastAPI. O Stub oferece endpoints HTTP/REST públicos (porta 30080 via NodePort) e, ao receber uma requisição, traduz os dados para as mensagens Protobuf e encaminha a chamada via gRPC para os serviços internos (`client` e `account`). Dessa forma, clientes externos interagem com uma API HTTP simples, enquanto o backend mantém os benefícios de desempenho e tipagem do gRPC.

Testes

Configuração adicional

Para o teste de autoscaling, foi necessário instalar/configurar o seguinte:

Metrics-server

O **metrics-server** serve para fornecer ao Kubernetes as métricas **de CPU e memória em tempo real** dos pods e nodes — e ele é obrigatório para o HPA funcionar

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

Para testar:

```
kubectl get apiservices | grep metrics
kubectl top nodes
```

```
kubectl top pods
```

Arquivos YAML para o HAP (Horizontal Pod Autoscaling)

Como o objetivo foi testar o autoscaling para todos os serviços (`stub`, `client`, `account`), foi necessário a definição adicional de 3 arquivos YAML para controlar o autoscaling de cada serviço:

- `hpa-stub-cpu.yaml` :

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: stub-hpa-cpu
  namespace: t3
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: stub
  minReplicas: 1
  maxReplicas: 3
  metrics:
    - type: Resource
      resource:
        name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

- `hpa-client-cpu.yaml` :

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: clientserver-hpa
  namespace: t3
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: clientserver
  minReplicas: 1
  maxReplicas: 3
  metrics:
    - type: Resource
      resource:
        name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

- `hpa-account-cpu.yaml` :

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: accountserver-hpa
  namespace: t3
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: accountserver
  minReplicas: 1
  maxReplicas: 3
  metrics:
    - type: Resource
      resource:
        name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Além disso foi necessário fazer uma modificação nos arquivos `stub.yaml`, `client_server.yaml` e `account_server.yaml`, adicionando a seguinte configuração:

```
spec:
  containers:
    ...
    image: ...
    ports:
    ...
    resources:
      requests:
        cpu: "100m"
        memory: "128Mi"
      limits:
        cpu: "300m"
        memory: "256Mi"
```

Essa configuração YAML define os **limites e requisições de recursos (CPU e memória)** que cada container pode usar no Kubernetes. Ela é essencial para o funcionamento do **HPA (Horizontal Pod Autoscaler)** e para garantir uma boa distribuição de recursos no cluster.

- **resources.requests**: Define a **quantidade mínima garantida** de recursos que o Kubernetes reservará para o pod
 - `cpu: "100m"` → Solicita 100 milicores (0,1 core de CPU). É o mínimo que o pod precisa para funcionar.
 - `memory: "128Mi"` → Solicita 128 MiB de memória RAM como mínimo garantido.
- **resources.limits**: Define o **limite máximo** de recursos que o container pode consumir
 - `cpu: "300m"` → O container não poderá usar mais que 300 milicores (0,3 core).

- `memory: "256Mi"` → Limite máximo de 256 MiB de memória. Se o pod tentar usar mais, pode ser terminado (OOMKilled).

O HPA monitora o uso percentual de CPU com base nos valores definidos em `requests`. Por exemplo, se o HPA estiver configurado para escalar quando atingir 50% de CPU, ele compara o uso atual com os `100m` requisitados. Sem essa definição, o HPA não consegue calcular a utilização e, portanto, não funciona.

Além disso, essa configuração ajuda o Kubernetes a saber exatamente quantos recursos cada pod precisa e consegue distribuí-los de forma eficiente entre os nós.

Após a definição dos arquivos citados acima, basta aplicar os arquivos HPA:

```
kubectl apply -f k8s/hpa-stub-cpu.yaml
kubectl apply -f k8s/hpa-client-cpu.yaml
kubectl apply -f k8s/hpa-account-cpu.yaml
```

Testes de carga com k6 no cluster Kubernetes

Os testes de carga foram realizados com a ferramenta **k6**, simulando **100 usuários virtuais (VUs)** durante **10 minutos** em todos os cenários. Cada usuário executa um fluxo completo de uso do sistema, incluindo criação de cliente, criação de conta, login, realização de transação e exclusão, de forma a exercitar todos os serviços envolvidos (stub, client, account e banco de dados Postgres).

O cluster Kubernetes foi configurado com **1 nó master** e um número variável de **workers**, entre os quais foram distribuídos quatro serviços: `stub-service`, `account-service`, `client-service` e `postgres`. Para cada execução, foram alterados apenas dois tipos de parâmetros:

- 1. Número de réplicas por serviço** (via `Deployment`), e
- 2. Número de nós workers** no cluster.

Foram avaliados cinco cenários:

- **Baseline:** 2 workers e 1 master, com **1 réplica de cada serviço** (stub, account, client e postgres).
- **Réplica Stub:** 2 workers e 1 master, com **3 réplicas apenas do stub**, mantendo account, client e postgres com 1 réplica.
- **Réplica dos serviços:** 2 workers e 1 master, com **3 réplicas para stub, account e client**, mantendo o Postgres com 1 réplica.
- **4 workers:** 4 workers e 1 master, com **1 réplica de cada serviço**.
- **4 workers e 3 réplicas** – 4 workers, `stub`, `client` e `account` com 3 réplicas cada (teste adicional).
- **HPA (Autoscaling) baseado em CPU** – 2 workers, `stub`, `client` e `account` inicialmente com 1 réplica

Veja a tabela abaixo:

Cenário	Workers / Rápidas (stub/client/account)	Throughput (http_reqs/s)	Latência	p95	http_req_failed	checks_failed
Baseline	2w - 1x stub, 1x account, 1x client	301 req/s	231 ms	711 ms	0%	0%
Replica Stub	2w - 3x stub , 1x account, 1x client	670 req/s	48 ms	133 ms	10,14%	12,62%
Replica 3xServiços	2w - 3x stub, 3x account, 3x client	670 req/s	48 ms	142 ms	10,14%	≈12%
4 Workers	4w - 1x stub, 1x account, 1x client	199 req/s	396 ms	1,37 s	0%	0%

4 Workers + 3xServiços	4w - 3x stub, 3x account, 3x client	333,77 req/s	195,92 ms	611,58 ms	0%	0%
Autoscaling	2w - (1-n)x stub, (1-n)x account, (1-n)x client	125,31 req/s	770 ms	2,92 s	1,47%	1,52%

Em todos os casos, analisaram-se principalmente: **taxa de requisições por segundo (throughput)**, **latência média e p95** e **taxa de falhas HTTP/checks** reportadas pelo k6.

No cenário **baseline**, o sistema apresentou um comportamento considerado “saudável”, com throughput em torno de 300 requisições por segundo, latência média na ordem de centenas de milissegundos e **ausência de falhas** (0% de `http_req_failed` e checks de negócio todos bem-sucedidos). Esse cenário foi adotado como **referência** de correção funcional.

Ao aumentar apenas o número de réplicas do **stub**, o throughput praticamente dobrou (cerca de 670 requisições por segundo) e a latência caiu significativamente (média em torno de dezenas de milissegundos). Porém, esse ganho de desempenho veio acompanhado de cerca de **10% de falhas HTTP** e de uma proporção relevante de checks de negócio falhando, especialmente nas operações de transação. Isso indica que, ao remover o stub como gargalo, a carga passou a pressionar mais fortemente os serviços downstream (account, client e, sobretudo, o Postgres), que não foram escalados na mesma proporção e passaram a **não conseguir atender corretamente** todas as requisições.

No cenário de **réplicas geral**, em que **stub, account e client** foram escalados simultaneamente para 3 réplicas, o comportamento global manteve-se muito próximo ao de **stubrep_heavy**: throughput elevado e baixa latência, porém com **taxa de erro semelhante**. Isso sugere que, nesse ponto, o gargalo principal já não se encontra mais na camada de aplicação, mas sim em um recurso **compartilhado e não escalado**, muito provavelmente o banco de dados Postgres (ou seu pool de conexões/IO). Assim, aumentar apenas o número de pods de aplicação não foi suficiente para reduzir a taxa de falhas.

O cenário com **4 workers sem réplicas**, em que se aumentou o número de **workers** de 2 para 4 mantendo **1 réplica de cada serviço**, observou-se uma **queda de throughput** em relação ao baseline e um aumento da latência, embora **sem introdução de falhas** (0% de erros). Esse resultado mostra que **apenas adicionar nós ao cluster**, sem ajustar o número de réplicas dos serviços nem tratar os gargalos reais (como o banco), não traz benefício direto para a performance da aplicação e pode até introduzir overhead adicional de comunicação e agendamento entre nós.

No cenário **4 workers e 3 réplicas de cada serviço** `stub`, `client` e `account`, observa-se um compromisso intermediário: throughput maior que o baseline (≈ 334 req/s) e latências inferiores às do baseline, com **0% de falhas** nas requisições. Contudo, durante esse teste houve **instabilidades nos containers**, com alguns pods ficando temporariamente em estado `down`, provavelmente devido à sobrecarga do host físico (CPU/memória). Assim, embora os números sejam promissores, esse cenário deve ser interpretado com cautela, pois expõe o limite de capacidade da infraestrutura disponível.

Por fim, testamos um cenário com **autoscaling via HPA** aplicado simultaneamente aos três serviços de aplicação (`stub`, `client` e `account`). Nesse caso, o comportamento foi o pior de todos os cenários avaliados: o throughput médio caiu para cerca de **125 req/s**, a latência média subiu para aproximadamente **770 ms** e o p95 aproximou-se de **3 segundos**, com cerca de **1,5% de falhas HTTP e 1,5% de checks** quebrados. Durante a execução, foi possível observar diversos erros operacionais (pods caindo e subindo, conexões sendo resetadas, demora na estabilização do cluster), indicando que, nas condições de hardware disponíveis, o HPA acabou gerando um ambiente instável, com o cluster “correndo atrás” da carga em vez de conseguir acompanhá-la de forma suave.

De forma geral, os testes evidenciam que:

- A escalabilidade obtida via Kubernetes depende tanto da **configuração das réplicas dos serviços** quanto do **dimensionamento de recursos compartilhados**, em especial o banco de dados.
- Escalar apenas a borda (stub) ou somente a camada de aplicação pode melhorar métricas superficiais de desempenho, mas **comprometer a corretude** se o backend não acompanhar.

- Aumentar apenas o número de nós do cluster, por sua vez, não resolve gargalos lógicos da aplicação e não garante, por si só, melhor throughput.

Esses resultados reforçam a importância de tratar a aplicação como um todo ao propor estratégias de escalonamento, combinando réplicas, recursos e observabilidade para identificar e aliviar os verdadeiros gargalos.

Monitoramento com Prometheus

Para complementar os testes de carga com o k6, o sistema foi instrumentado com métricas Prometheus expostas pelo `stub-service`. A partir dessas métricas foram definidas consultas (queries) para acompanhar, em tempo real, o comportamento do sistema durante o cenário **baseline**. A seguir são descritas as principais queries e a interpretação dos gráficos correspondentes (todos os gráficos são da Baseline para simplificar o processo de exemplificação). As imagens dos outros cenários foram adicionadas nos Anexos

Throughput médio do serviço

Query (TP):

```
sum(
  rate(http_requests_total{
    namespace="t3",
    service="stub-service",
  }[15m])
)
```

Essa consulta soma a taxa de variação (`rate`) do contador de requisições HTTP do `stub-service` em uma janela deslizante de 15 minutos, resultando em uma estimativa de **requisições por segundo** atendidas pelo serviço.

No gráfico correspondente, observa-se que o throughput sai de zero, cresce rapidamente quando o teste inicia e se estabiliza em um patamar praticamente constante, indicando que o `stub-service` consegue sustentar a carga do teste baseline sem quedas abruptas de vazão.



Throughput por endpoint

Query (TP endpoint):

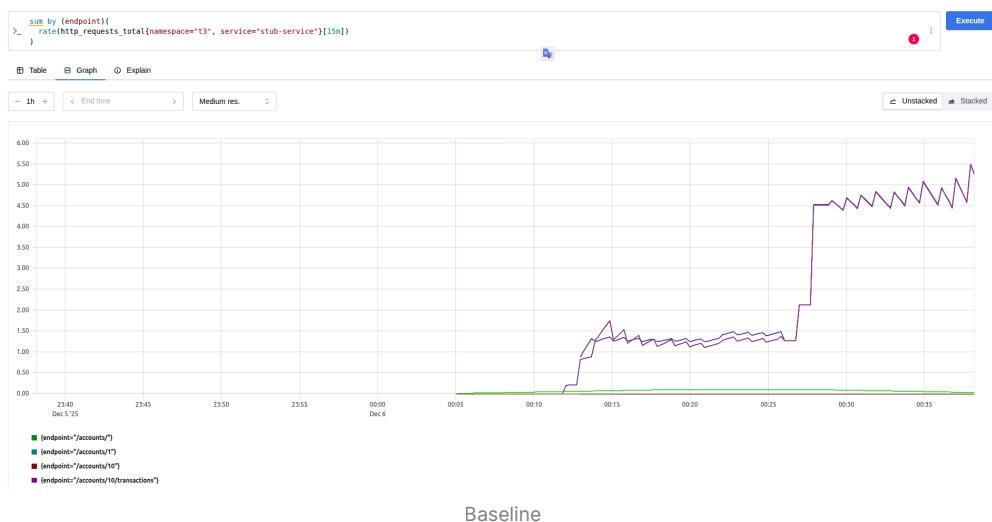
```

sum by (endpoint)(
rate(http_requests_total{
    namespace="t3",
    service="stub-service"
}[15m])
)

```

Essa query decompõe o throughput por valor do rótulo `endpoint`, permitindo identificar quais rotas concentram a maior parte das requisições.

No gráfico, o endpoint de **transações** (`/accounts/10/transactions`) domina claramente o volume de requisições, enquanto as rotas de criação e consulta de contas aparecem com valores bem menores. Isso corrobora o desenho do cenário de teste, em que cada usuário realiza várias operações de transação após a criação da conta.



Latência média

Query (Latência média – forma conceitual):

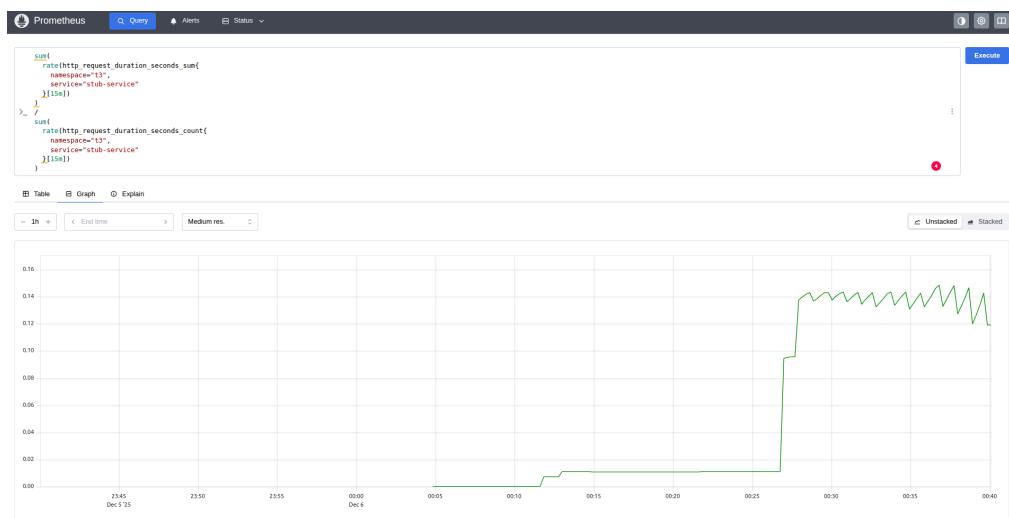
```

sum(
rate(http_request_duration_seconds_sum{
    namespace="t3",
    service="stub-service"
}[15m])
) / sum(
rate(http_request_duration_seconds_count{
    namespace="t3",
    service="stub-service"
}[15m])
)

```

A ideia dessa consulta é dividir a soma das durações das requisições pela quantidade de requisições na mesma janela, obtendo a **latência média em segundos**.

O gráfico mostra um aumento inicial quando o teste começa, seguido de um patamar relativamente estável na ordem de centenas de milissegundos. Não há explosões de latência ao longo do tempo, o que indica que, no baseline, o serviço atende a carga mantendo uma média de resposta consistente.



Latência P95

Query (P95 – forma conceitual):

```
histogram_quantile(
  0.95,
  sum by (le)
  rate(http_request_duration_seconds_bucket{
    namespace="t3",
    service="stub-service"
  }[15m])
)
```

Essa query usa os buckets de histograma de duração das requisições para calcular o **percentil 95 de latência (P95)**, isto é, o tempo abaixo do qual 95% das requisições são concluídas.

No gráfico, o P95 cresce até um patamar em torno de meio segundo, mantendo-se estável durante o teste. Isso mostra que a cauda de latência, embora maior que a média, permanece controlada, sem degradação progressiva ao longo dos 10 minutos de carga.



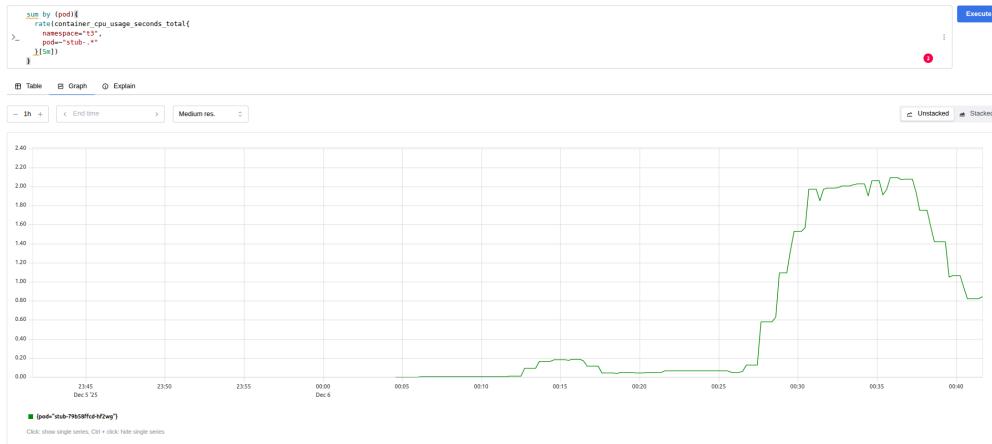
Uso de CPU

Query (CPU):

```
sum by (pod)(
  rate(container_cpu_usage_seconds_total{
    namespace="t3",
    pod=~"stub-.*"
  }[5m])
)
```

Essa consulta calcula, por pod do `stub-service`, a taxa de consumo de CPU em segundos de CPU por segundo, equivalente ao número de vCPUs efetivamente ocupadas.

O gráfico mostra que, durante o teste, o consumo de CPU sobe de praticamente zero para um patamar elevado e relativamente estável (aproximadamente 1-2 vCPUs), sem crescimento descontrolado. Isso indica que o serviço trabalha próximo da capacidade, mas sem sinais claros de saturação no cenário baseline.



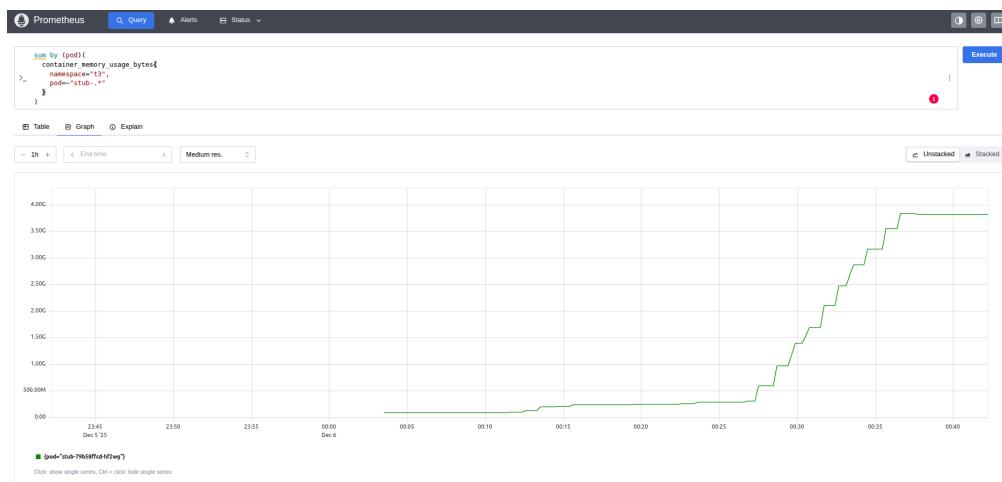
Uso de memória

Query (Memória):

```
sum by (pod)(  
    container_memory_usage_bytes{  
        namespace="t3",  
        pod=~"stub-.*"  
    }  
)
```

Essa query soma o uso de memória em bytes por pod do `stub-service`, permitindo acompanhar a evolução do **consumo de memória RAM** ao longo do teste.

No gráfico observa-se um crescimento gradual da memória utilizada, seguido de estabilização em um novo patamar. Esse comportamento é típico de aplicações que alocam mais heap e cache sob carga até atingir um ponto de equilíbrio. No baseline, não há indicação de crescimento indefinido (que sugeriria vazamento de memória), mas o valor final observado deve ser considerado no dimensionamento de recursos dos pods.



Conclusão

Os testes de carga com k6, utilizando 100 usuários virtuais por 10 minutos em cada cenário, mostraram na prática como decisões de escalonamento afetam o comportamento do sistema. O cenário baseline serviu como referência de correção, com throughput estável, latências moderadas e ausência de erros. Ao aumentar apenas as réplicas do stub (borda), obtivemos um ganho expressivo em throughput e redução de latência, mas às custas de uma taxa significativa de falhas, evidenciando que o gargalo foi deslocado para os serviços downstream e, principalmente, para o banco de dados. Mesmo escalando todos os serviços de aplicação, o problema não foi resolvido, reforçando o papel central de recursos compartilhados como o Postgres. Por outro lado, aumentar somente o número de workers, sem ajustar réplicas e sem tratar o backend, não trouxe benefícios de desempenho e chegou a degradar o throughput e a latência, mostrando que "mais nós" não significa, por si só, mais capacidade útil.

O monitoramento com Prometheus complementou essa análise, permitindo correlacionar métricas de throughput, latência (média e P95), CPU e memória com cada cenário de teste. Assim, foi possível enxergar o sistema de forma mais completa, identificando não apenas "se ficou mais rápido ou mais lento", mas também onde estavam os gargalos e como os recursos estavam sendo consumidos.

Em síntese, o trabalho permitiu consolidar três aprendizados principais: (i) a importância de entender a aplicação como um todo ao planejar estratégias de escalonamento, considerando tanto as réplicas quanto os recursos compartilhados; (ii) o papel do Kubernetes como plataforma de experimentação, facilitando a comparação entre

diferentes configurações de cluster e serviços; e (iii) o valor de uma boa observabilidade (Prometheus) para transformar testes de carga em conhecimento concreto sobre o comportamento e os limites da aplicação.

Dificuldades encontradas

Apesar de termos conseguido concluir todas as etapas propostas, o desenvolvimento do trabalho esbarrou em algumas dificuldades práticas que impactaram bastante o ritmo e a experiência dos testes.

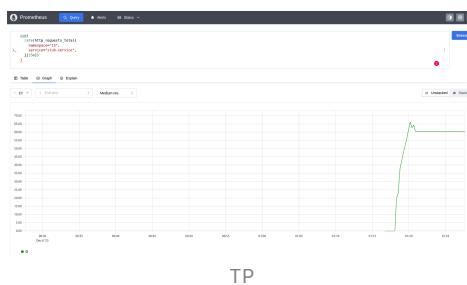
A primeira e mais marcante foi a **demora para buildar o ambiente e executar os testes a cada mudança de parâmetro**. Toda alteração de configuração (número de réplicas, workers, ajustes nos manifests, etc.) implicava recravar ou atualizar o cluster, garantir que todos os pods estivessem saudáveis, rodar novamente o k6 por 10 minutos e só então coletar e analisar os resultados. Esse ciclo é naturalmente lento e altamente repetitivo, o que tornava a experimentação mais cansativa e exigia bastante disciplina para manter anotações consistentes de cada cenário.

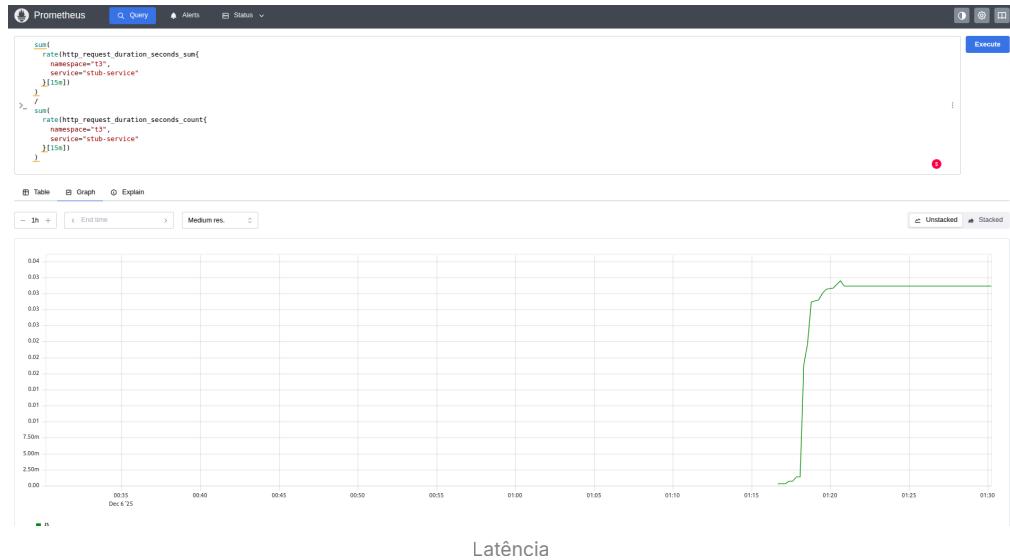
Em segundo lugar, a própria **execução pesada do ambiente** se mostrou um desafio. Rodar um cluster Kubernetes multinode via kind, com múltiplos microserviços e um banco PostgreSQL, ao mesmo tempo em que o k6 aplicava uma carga de 100 usuários virtuais por 10 minutos, exigiu muitos recursos de CPU e memória da máquina. Em vários momentos o computador chegou a travar ou reiniciar, interrompendo os testes no meio e obrigando a reconstruir o cluster e repetir todo o processo. Isso aumentou o tempo total de execução do trabalho e limitou a quantidade de cenários que conseguimos explorar com estabilidade.

Por fim, houve a **dificuldade em montar um cluster “real” com múltiplas máquinas físicas**. Ficou claro para nós que executar todos os testes localmente, em um único host, não é o cenário ideal para observar o impacto “real” de diferentes configurações de cluster, principalmente em termos de rede e isolamento de recursos. No entanto, viabilizar um ambiente distribuído com várias máquinas físicas (ou mesmo VMs dedicadas) estava fora do nosso alcance neste contexto, tanto por limitações de infraestrutura quanto de tempo. Por isso, tivemos que aceitar o compromisso de usar o kind como aproximação prática de um cenário multinode, cientes de que alguns resultados são influenciados pelas restrições do ambiente local.

Anexos

Réplica Stub





Latência



P95



CPU



Memória

Réplica Geral



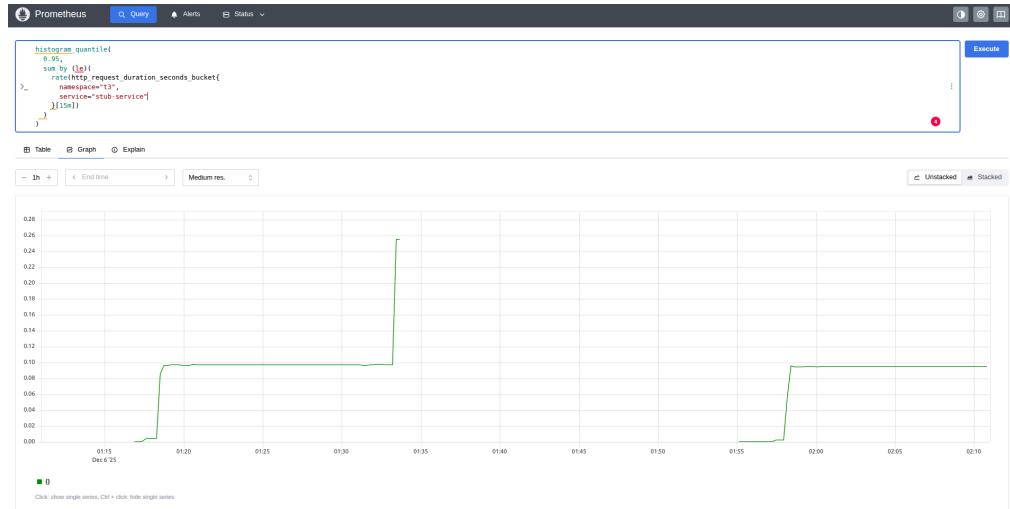
TP



TP end



Latência



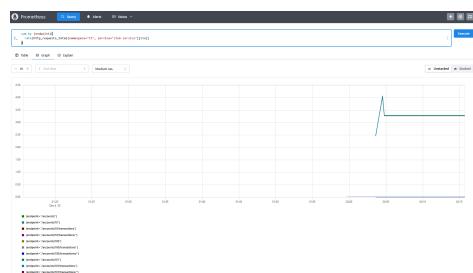
P95

Não conseguimos pegar a métrica de **CPU** e de **Memória** para este cenário de teste, por alguma razão o **Prometheus** retornava **empty value**.

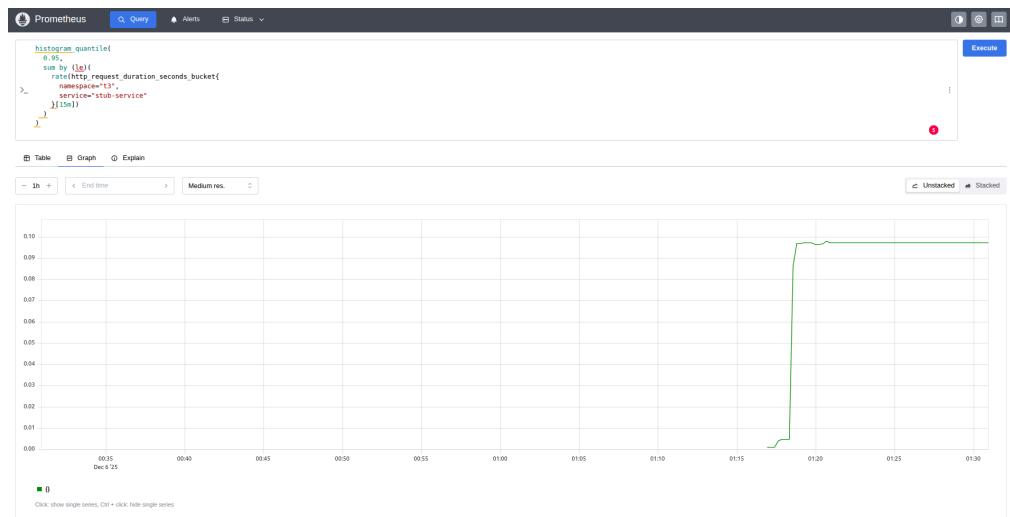
4 Workers



TP

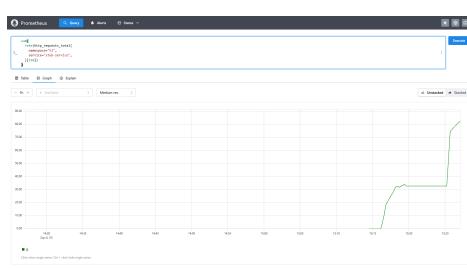


TP end

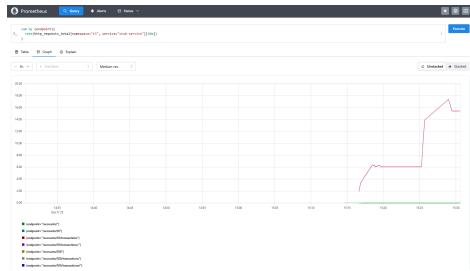


Não conseguimos pegar a métrica de **CPU** e de **Memória** para este cenário de teste, por alguma razão o **Prometheus** retornava `empty value`.

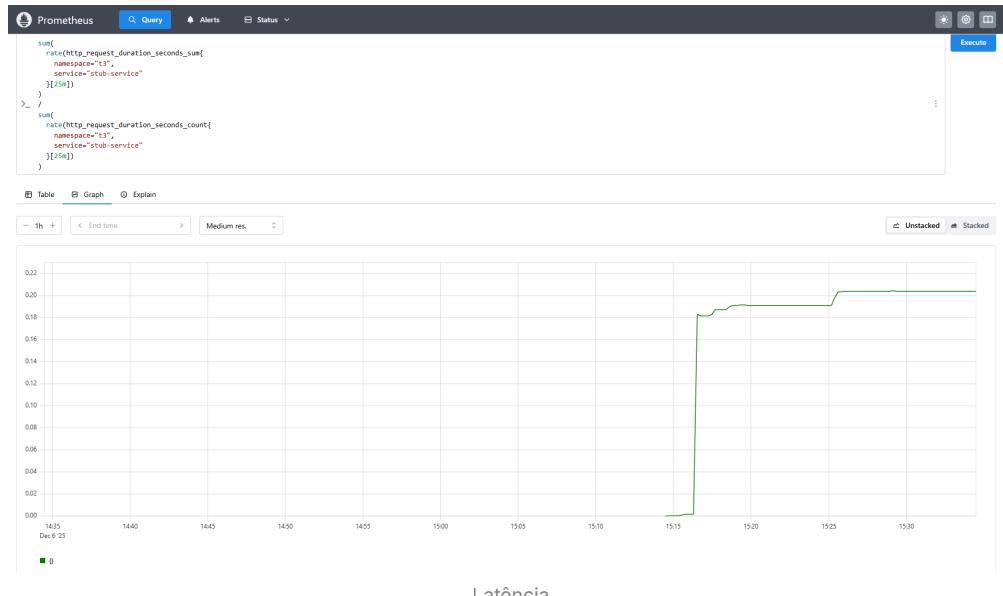
4 Workers + 3 Réplicas



TP



TP end



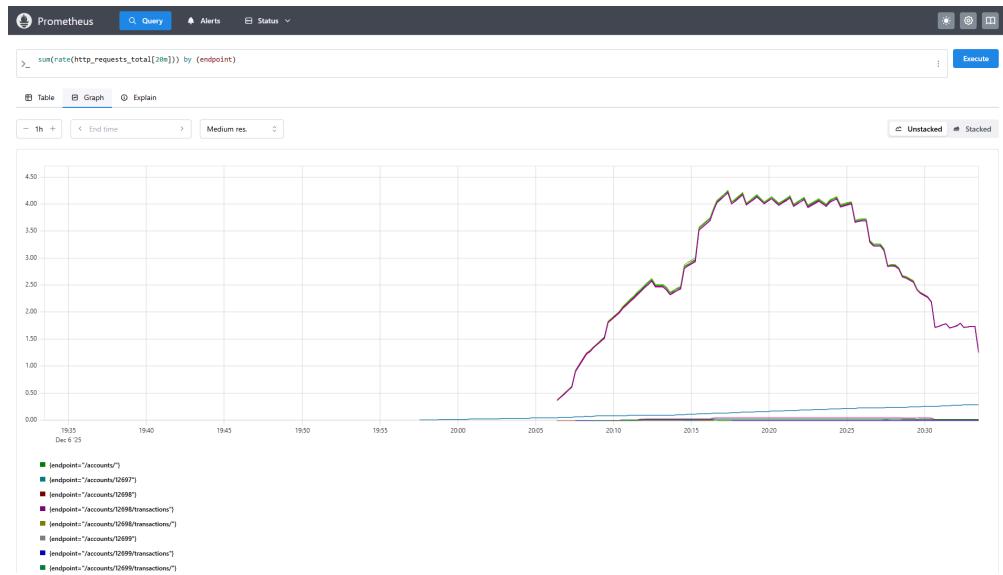
Latência



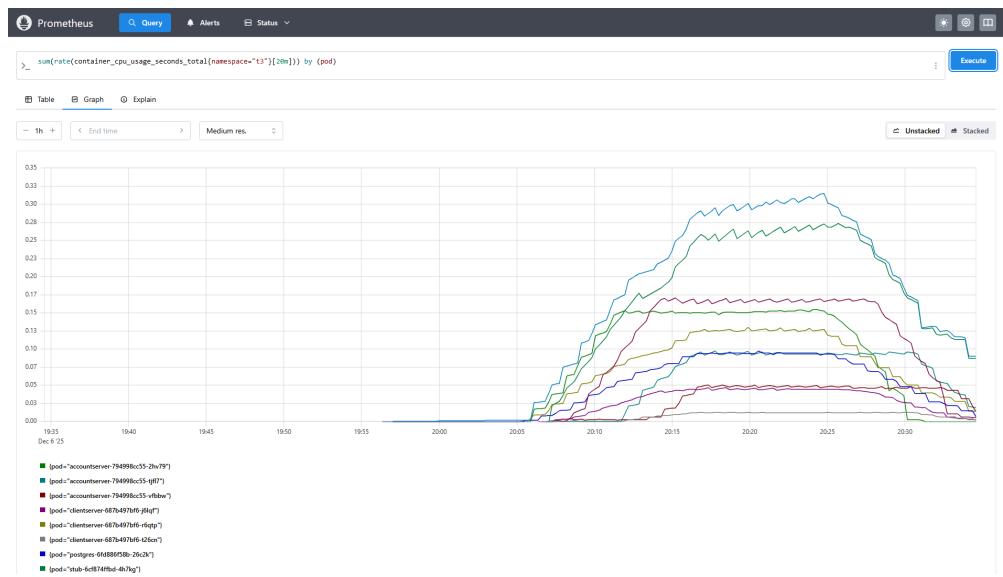
P95

Não conseguimos pegar a métrica de **CPU** e de **Memória** para este cenário de teste, por alguma razão o **Prometheus** retornava `empty value`.

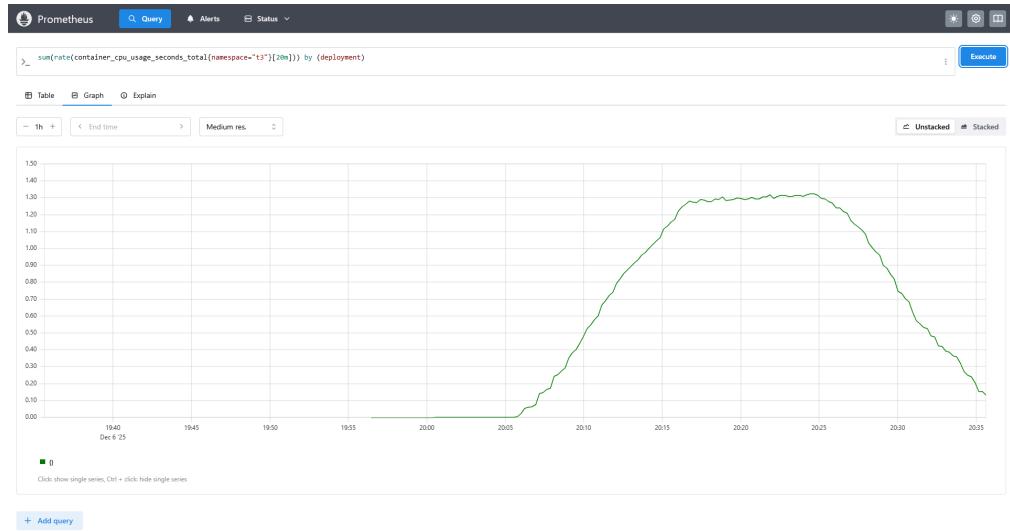
Autoscaling (HPA)



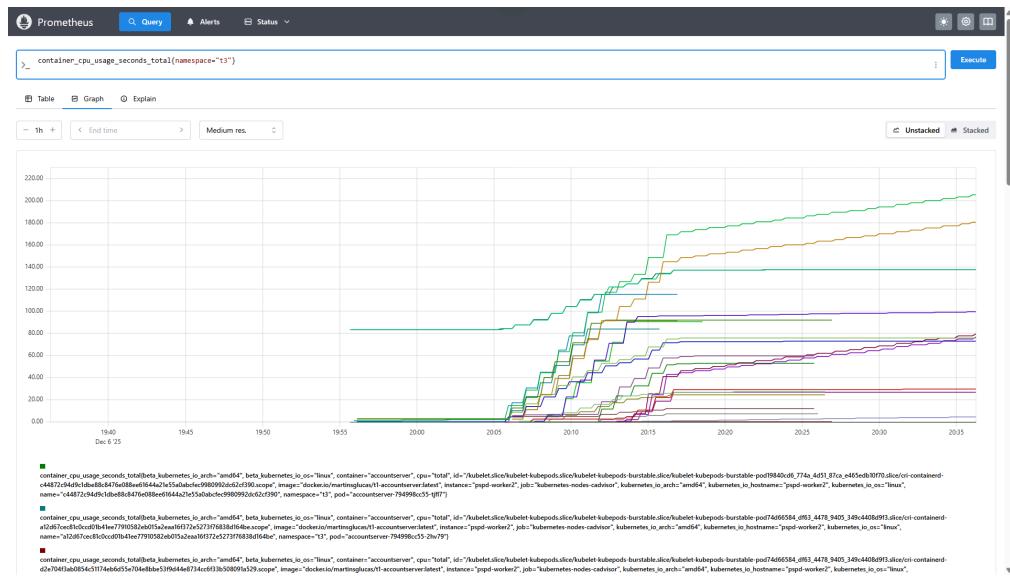
Request por endpoint



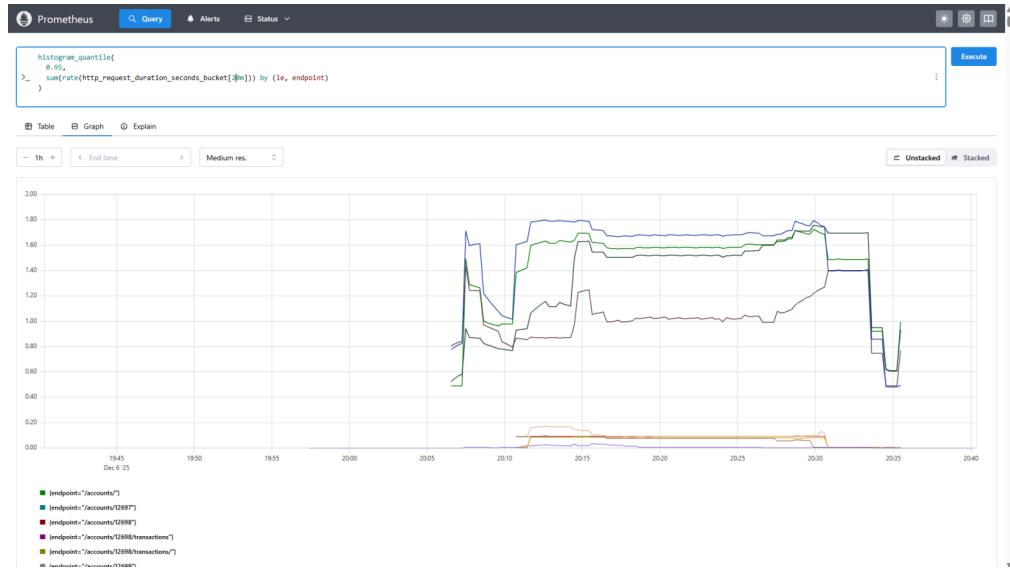
CPU por pod



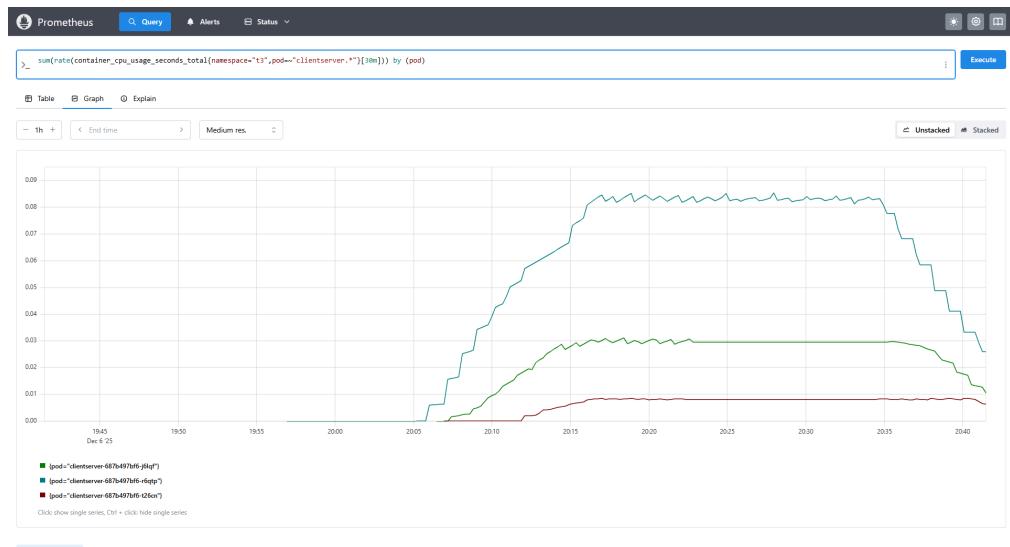
CPU por deployment



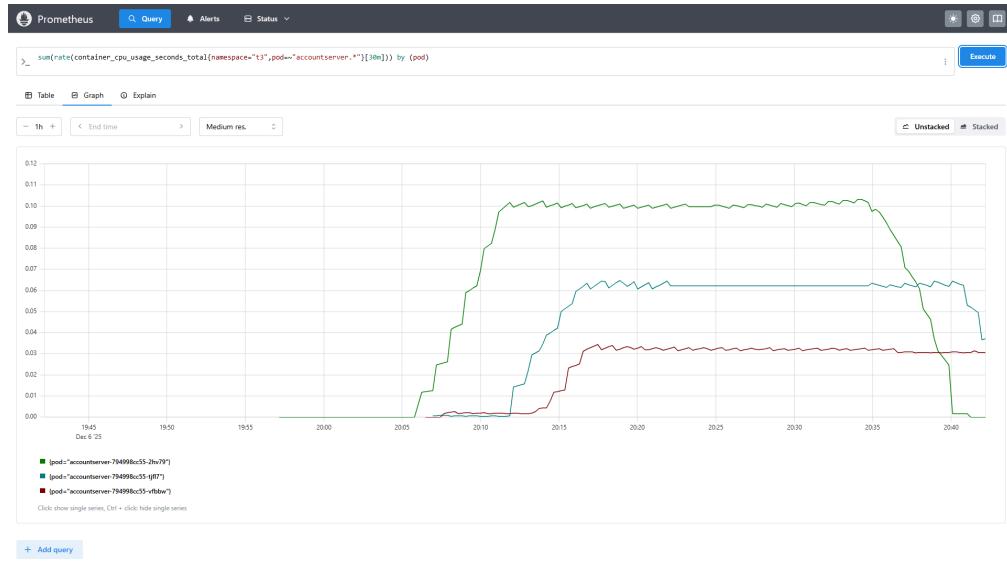
CPU pedida VS. CPU usada



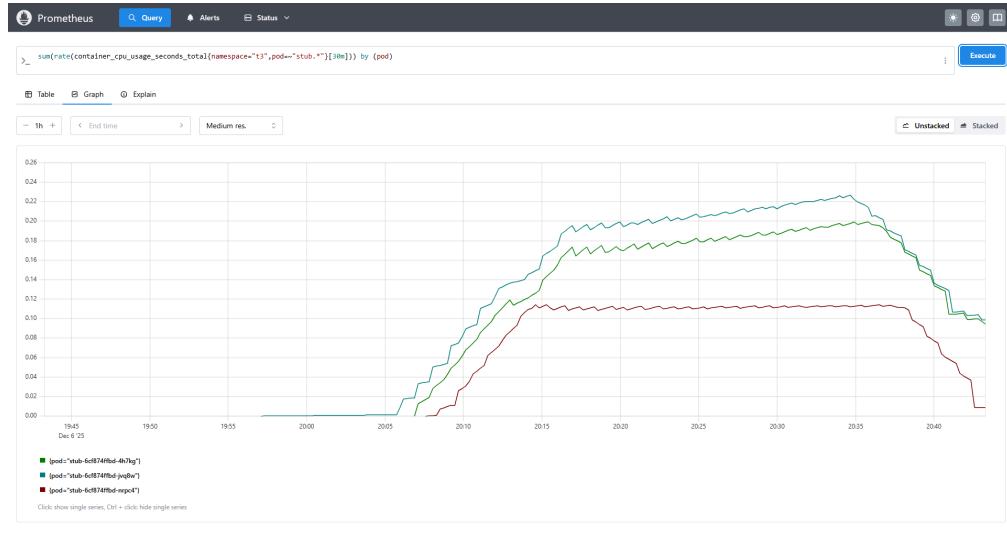
P95



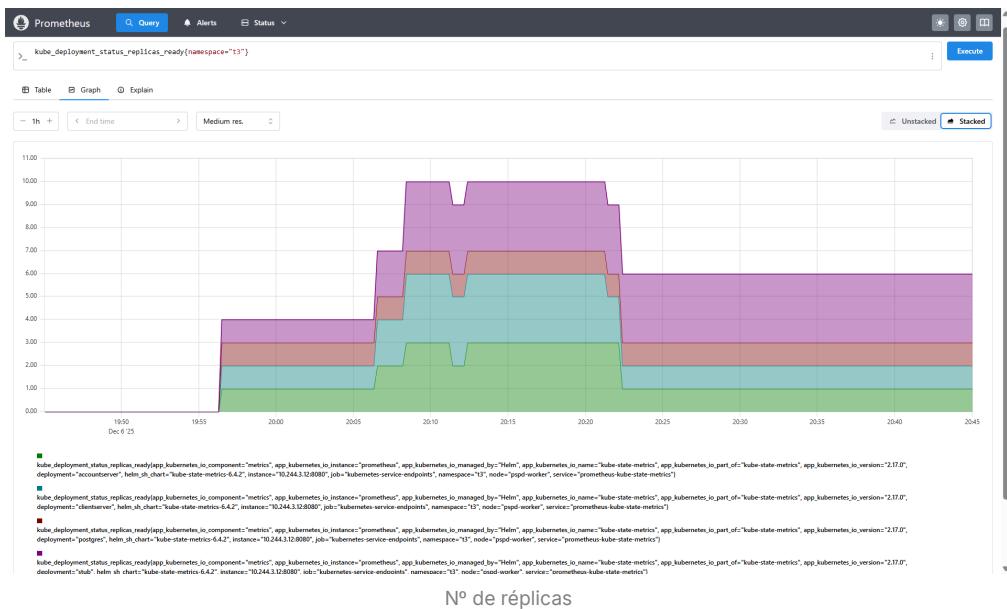
CPU do client gRPC



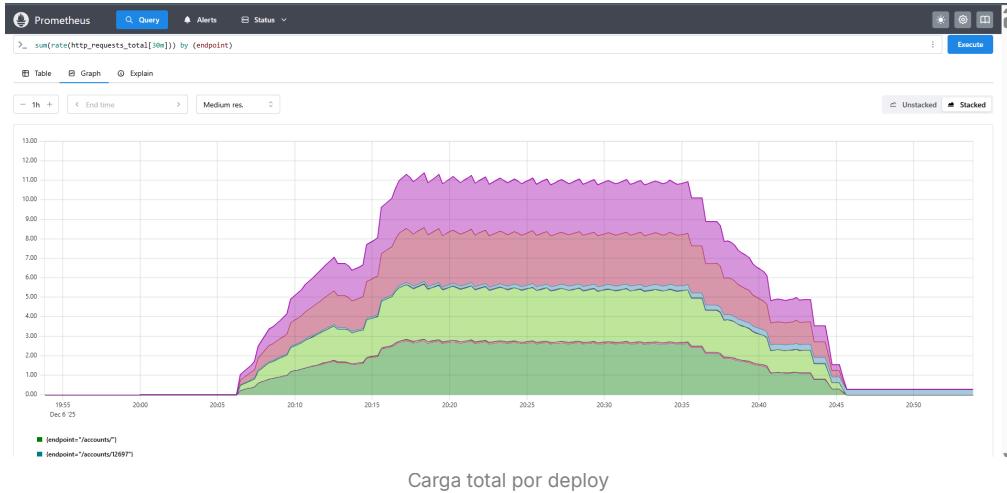
CPU do account



CPU do stub



Nº de réplicas



Carga total por deploy

Monitoramento HPA

Além do monitoramento das métricas pelo Prometheus, o cenário do HPA também teve monitoramento dos pods, da CPU, ... através dos comandos:

```
# mostrar escala dos pods
kubectl get pods -n t3 -w
```

```
# mostrar decisão do autoscaler
kubectl get hpa -n t3 -w
```

```
# mostrar consumo de CPU
watch -n 2 "kubectl top pod -n t3 --containers"
```

As imagens a seguir mostram alguns momentos durante a execução do teste de carga:

```

lucasmartins@LUCASMARTIN:~$ kubectl get hpa -n t3 -w
NAME          REFERENCE   TARGETS
accountserver-hpa Deployment/accountserver   cpu: 1%50%
clientserver-hpa Deployment/clientserver    cpu: 1%50%
stub-hpa-cpu   Deployment/stub            cpu: 3%50%
               MINPODS   MAXPODS   REPLICAS   AGE
accountserver-hpa      1         3        10m   1d
clientserver-hpa       1         3        10m   1d
stub-hpa-cpu          1         3        60m   1d

lucasmartins@LUCASMARTIN:~$ kubectl get pods -n t3 -w
NAME                               STATUS   AGE
accountserver-794998cc55-2hv79   1/1    Running   0   12m
clientserver-687b497bf6-r6gtp   1/1    Running   0   12m
postgres-6fd86f58b-26c2k       1/1    Running   0   134m
stub-6cf874ffbd-jvq8w          1/1    Running   0   12m

Every 2.0s: kubectl top pod -n t3 --containers           LUCAS MARTINS: Sat Dec 6 17:05:02 2025
POD                           NAME          CPU(cores)   MEMORY(bytes)
accountserver-794998cc55-2hv79 accountserver   1m          21Mi
clientserver-687b497bf6-r6gtp   clientserver   1m          21Mi
postgres-6fd86f58b-26c2k       postgres      1m          30Mi
stub-6cf874ffbd-jvq8w          stub          3m          39Mi

```

Início da execução do teste

```

lucasmartins@LUCASMARTIN:~$ kubectl get hpa -n t3 -w
accountserver-hpa Deployment/accountserver   cpu: 189%5
0% 1         3         3        12m
clientserver-hpa Deployment/clientserver    cpu: 82%50
% 1         3         3        12m
stub-hpa-cpu   Deployment/stub            cpu: 206%5
0% 1         3         3        62m
accountserver-hpa Deployment/accountserver   cpu: 94%50
% 1         3         3        12m
stub-hpa-cpu   Deployment/stub            cpu: 203%5
0% 1         3         3        62m
clientserver-hpa Deployment/clientserver    cpu: 40%50
% 1         3         3        13m
accountserver-hpa Deployment/accountserver   cpu: 114%5
0% 1         3         3        13m
stub-hpa-cpu   Deployment/stub            cpu: 158%5
0% 1         3         3        63m
clientserver-hpa Deployment/clientserver    cpu: 71%50
% 1         3         3        13m

lucasmartins@LUCASMARTIN:~$ kubectl get pods -n t3 -w
Every 2.0s: kubectl top pod -n t3 --containers           LUCAS MARTINS: Sat Dec 6 17:07:15 2025
POD                           NAME          CPU(cores)   MEMORY(bytes)
stub-6cf874ffbd-4h7kg        ContainerCreating 0          1s
accountserver-794998cc55-tjfl7 0/1    Pending     0   0s
accountserver-794998cc55-tjfl7 0/1    Pending     0   1s
accountserver-794998cc55-tjfl7 0/1    Pending     0   1s
accountserver-794998cc55-tjfl7 0/1    Pending     0   4s
stub-6cf874ffbd-4h7kg        0/1    Running    0   20s
clientserver-687b497bf6-t26cn 1/1    Running    0   10s
accountserver-794998cc55-tjfl7 0/1    Init:0/1   0   10s
stub-6cf874ffbd-4h7kg        1/1    Running    0   15s
accountserver-794998cc55-tjfl7 0/1    PodInitializing 0   17s
accountserver-794998cc55-tjfl7 0/1    Running    0   19s
accountserver-687b497bf6-j6lqf 0/1    Pending     0   30s
clientserver-687b497bf6-j6lqf 0/1    Init:0/1   0   55s
clientserver-687b497bf6-j6lqf 0/1    PodInitializing 0   58s
accountserver-794998cc55-vfbbw 0/1    Init:0/1   0   94s
accountserver-794998cc55-vfbbw 0/1    PodInitializing 0   101s
clientserver-687b497bf6-j6lqf 0/1    Running    0   105s
stub-6cf874ffbd-npc4        0/1    Running    0   99s
accountserver-794998cc55-vfbbw 0/1    Running    0   104s

Every 2.0s: kubectl top pod -n t3 --containers           LUCAS MARTINS: Sat Dec 6 17:07:15 2025
POD                           NAME          CPU(cores)   MEMORY(bytes)
accountserver-794998cc55-2hv79 accountserver   168m        89Mi
accountserver-794998cc55-tjfl7  accountserver   1m          19Mi
clientserver-687b497bf6-r6gtp   clientserver   142m        60Mi
clientserver-687b497bf6-t26cn  clientserver   1m          19Mi
postgres-6fd86f58b-26c2k       postgres      47m          61Mi
stub-6cf874ffbd-4h7kg        stub          129m        47Mi
stub-6cf874ffbd-jvq8w          stub          188m        77Mi

```

2 min de execução, cada serviço, exceto postgres, com 2 réplicas

```

lucasmartins@LUCASMARTIN ~ + - x lucasmartins@LUCASMARTIN ~ + - x
stub-hpa-cpu Deployment/stub cpu: 289%/5
% 1 3 3 6m
clientserver-hpa Deployment/clientserver cpu: 51%/50
% 1 3 3 16m
accountserver-hpa Deployment/accountserver cpu: 91%/50
% 1 3 3 16m
stub-hpa-cpu Deployment/stub cpu: 296%/5
% 1 3 3 6m
clientserver-hpa Deployment/clientserver cpu: 55%/50
% 1 3 3 17m
accountserver-hpa Deployment/accountserver cpu: 88%/50
% 1 3 3 17m
stub-hpa-cpu Deployment/stub cpu: 279%/5
% 1 3 3 6m
clientserver-hpa Deployment/clientserver cpu: 68%/50
% 1 3 3 17m
accountserver-hpa Deployment/accountserver cpu: 95%/50
% 1 3 3 17m
| lucasmartins@LUCASMARTIN ~ + - x lucasmartins@LUCASMARTIN ~ + - x
accountserver-794998cc55-tjfl7 0/1 Init:0/1 0 18s
stub-6cf874fffd-nrpb4 0/1 Running 0 15s
accountserver-794998cc55-tjfl7 0/1 PodInitializing 0 17s
accountserver-794998cc55-tjfl7 0/1 Running 0 19s
accountserver-794998cc55-tjfl7 1/1 Running 0 30s
clientserver-687b497bf6-j6lqf 0/1 Init:0/1 0 55s
clientserver-687b497bf6-j6lqf 0/1 PodInitializing 0 58s
accountserver-794998cc55-vrbbw 0/1 Init:0/1 0 94s
accountserver-794998cc55-vrbbw 0/1 PodInitializing 0 101s
clientserver-687b497bf6-j6lqf 0/1 Running 0 105s
clientserver-687b497bf6-j6lqf 0/1 Running 0 99s
accountserver-794998cc55-vrbbw 0/1 Running 0 104s
clientserver-687b497bf6-j6lqf 1/1 Running 0 116s
stub-6cf874fffd-nrpb4 1/1 Running 0 101s
accountserver-794998cc55-vfbww 1/1 Running 0 115s
accountserver-794998cc55-2hv79 0/1 OOMKilled 0 18s
accountserver-794998cc55-2hv79 0/1 Running 1 (3s ago) 18m
accountserver-794998cc55-2hv79 1/1 Running 1 (14s ago) 18m
| lucasmartins@LUCASMARTIN ~ + - x lucasmartins@LUCASMARTIN ~ + - x
Every 2.0s: kubectl top pod -n t3 --containers LUCASMARTIN: Sat Dec 6 17:11:25 2025
POD NAME CPU(cores) MEMORY(bytes)
accountserver-794998cc55-tjfl7 accountserver 23m 25Mi
accountserver-794998cc55-vrbbw accountserver 1m 19Mi
clientserver-687b497bf6-j6lqf clientserver 71m 45Mi
clientserver-687b497bf6-j6lqf 131m 67Mi
clientserver-687b497bf6-t26cn clientserver 19m 22Mi
postgres-6fd886f58b-26c2k postgres 108m 800Mi
stub-6cf874fffd-uh7kg stub 27m 169Mi
stub-6cf874fffd-jvq8w stub 28m 192Mi
stub-6cf874fffd-nrpb4 stub 270m 140Mi
| lucasmartins@LUCASMARTIN ~ + - x lucasmartins@LUCASMARTIN ~ + - x
17:11 06/12/2025

```

mátedade da execução, cada serviço com 3 réplicas

```

lucasmartins@LUCASMARTIN ~ + - x lucasmartins@LUCASMARTIN ~ + - x
accountserver-hpa Deployment/accountserver cpu: 83%/50
% 1 3 3 18m
stub-hpa-cpu Deployment/stub cpu: 275%/5
% 1 3 3 6m
clientserver-hpa Deployment/clientserver cpu: 74%/50
% 1 3 3 18m
accountserver-hpa Deployment/accountserver cpu: 99%/50
% 1 3 3 18m
stub-hpa-cpu Deployment/stub cpu: 274%/5
% 1 3 3 6m
clientserver-hpa Deployment/clientserver cpu: 72%/50
% 1 3 3 18m
accountserver-hpa Deployment/accountserver cpu: 98%/50
% 1 3 3 18m
| lucasmartins@LUCASMARTIN ~ + - x lucasmartins@LUCASMARTIN ~ + - x
clientserver-687b497bf6-j6lqf 0/1 Init:0/1 0 55s
clientserver-687b497bf6-j6lqf 0/1 PodInitializing 0 58s
accountserver-794998cc55-vrbbw 0/1 Init:0/1 0 94s
accountserver-794998cc55-vrbbw 0/1 PodInitializing 0 101s
clientserver-687b497bf6-j6lqf 0/1 Running 0 105s
clientserver-687b497bf6-j6lqf 0/1 Running 0 99s
accountserver-794998cc55-vrbbw 0/1 Running 0 104s
clientserver-687b497bf6-j6lqf 1/1 Running 0 116s
stub-6cf874fffd-nrpb4 1/1 Running 0 101s
accountserver-794998cc55-vfbww 1/1 Running 0 115s
accountserver-794998cc55-2hv79 0/1 OOMKilled 0 18s
accountserver-794998cc55-2hv79 0/1 Running 1 (3s ago) 18m
accountserver-794998cc55-2hv79 1/1 Running 1 (14s ago) 18m
stub-6cf874fffd-jvq8w 0/1 OOMKilled 0 19m
stub-6cf874fffd-uh7kg 0/1 OOMKilled 0 6m55s
stub-6cf874fffd-uh7kg 0/1 Running 1 (3s ago) 19m
stub-6cf874fffd-jvq8w 0/1 Running 1 (3s ago) 20m
stub-6cf874fffd-nrpb4 1/1 Running 1 (14s ago) 20m
| lucasmartins@LUCASMARTIN ~ + - x lucasmartins@LUCASMARTIN ~ + - x
Every 2.0s: kubectl top pod -n t3 --containers LUCASMARTIN: Sat Dec 6 17:12:47 2025
POD NAME CPU(cores) MEMORY(bytes)
accountserver-794998cc55-2hv79 accountserver 1m 19Mi
accountserver-794998cc55-tjfl7 accountserver 285m 182Mi
accountserver-794998cc55-vrbbw accountserver 1m 20Mi
clientserver-687b497bf6-j6lqf clientserver 66m 56Mi
clientserver-687b497bf6-j6lqf 38m 68Mi
clientserver-687b497bf6-t26cn clientserver 32m 33Mi
postgres-6fd886f58b-26c2k postgres 97m 91Mi
stub-6cf874fffd-nrpb4 stub 291m 160Mi
| lucasmartins@LUCASMARTIN ~ + - x lucasmartins@LUCASMARTIN ~ + - x
17:12 06/12/2025

```

OOMKilled por usar memória além do limite

```

iteration_duration.....: avg=7.61s min=809.91μs max=10.02s p(90)=15.01s p(95)=25.92s
91μs min=99s max=208s vus.....: 8630 13 024355/s
vus_max.....: 200 min=200 max=200
NETWORK
data_received.....: 17 MB 25 kB/s
data_sent.....: 12 MB 17 kB/s

running (11-12.9s) 8630/200 VUs, 8630 complete and 1 interrupted iterations
default ✓ [=====] 8630/200 VUs 11m0s
lucasmartins@LUCAS MARTIN:~/pspd/t3-cluster/tests$ |

```

	NAME	CPU(cores)	MEMORY(bytes)
accountserver-794998cc55-2hv79	accountserver	1m	20Mi
accountserver-794998cc55-tjfl7	accountserver	62m	129Mi
accountserver-794998cc55-vfbw	accountserver	19m	50Mi
clientserver-687b497bf6-j6lqf	clientserver	1m	27Mi
clientserver-687b497bf6-r6tp	clientserver	67m	68Mi
clientserver-687b497bf6-t26cn	clientserver	5m	30Mi
postgress-6fd86f58b-26c2k	postgress	51m	109Mi
stub-6cf874ffbd-4h7kg	stub	8m	110Mi
stub-6cf874ffbd-jvq8w	stub	236m	115Mi
stub-6cf874ffbd-nrpc4	stub	3m	48Mi

Finalização do teste

```

clientserver-687b497bf6-j6lqf 0/1 Running 0 105s
stub-6cf874ffbd-nrpc4 0/1 Running 0 90s
accountserver-794998cc55-vfbw 0/1 Running 0 190s
clientserver-687b497bf6-j6lqf 1/1 Running 0 116s
stus-6cf874ffbd-nrpc4 1/1 Running 0 101s
accountserver-794998cc55-vfbw 0/1 Running 0 115s
accountserver-794998cc55-2hv79 0/1 OOMKilled 0 18s
accountserver-794998cc55-tjfl7 0/1 Running 0 99s
accountserver-794998cc55-vfbw 0/1 Running 0 190s
clientserver-687b497bf6-j6lqf 1/1 Running 0 116s
stus-6cf874ffbd-nrpc4 1/1 Running 0 101s
accountserver-794998cc55-vfbw 0/1 Running 0 115s
accountserver-794998cc55-2hv79 0/1 OOMKilled 0 18s
accountserver-794998cc55-tjfl7 0/1 Running 0 18s
accountserver-794998cc55-vfbw 0/1 OOMKilled 0 18s
clientserver-687b497bf6-j6lqf 0/1 OOMKilled 0 19s
stus-6cf874ffbd-4h7kg 0/1 Running 0 6m55s
stus-6cf874ffbd-jvq8w 1/1 Running 1 (3s ago) 6m57s
stus-6cf874ffbd-4h7kg 0/1 Running 1 (3s ago) 20m
stus-6cf874ffbd-jvq8w 1/1 Running 1 (14s ago) 7m8s
stus-6cf874ffbd-nrpc4 0/1 OOMKilled 0 8m2s
stus-6cf874ffbd-nrpc4 0/1 Running 1 (3s ago) 8m4s
stus-6cf874ffbd-nrpc4 0/1 Running 1 (14s ago) 8m4s

```

Réplicas encerradas

Referências

- Kubernetes Documentation. Disponível em: <https://kubernetes.io/docs/>.
- kind – Kubernetes IN Docker. Disponível em: <https://kind.sigs.k8s.io/>.
- kubectl – Command-line tool for Kubernetes. Disponível em: <https://kubernetes.io/docs/reference/kubectl/>.
- Helm – The Kubernetes Package Manager. Disponível em: <https://helm.sh/docs/>.
- Prometheus – Monitoring System & Time Series Database. Disponível em: <https://prometheus.io/docs/>.
- k6 – Load Testing for Developers. Disponível em: <https://grafana.com/docs/k6/latest/>.
- Docker Documentation. Disponível em: <https://docs.docker.com/>.