Análise de Desempenho e Escalabilidade de uma Arquitetura de Microsserviços: Um Estudo de Caso do Spring PetClinic com Locust

Áurea Letícia Carvalho Macedo
Universidade Federal do Piauí (UFPI)
Picos, PI, Brasil
aurea.macedo@ufpi.edu.br

Luis Gustavo Luz de Deus Ramos
Universidade Federal do Piauí (UFPI)
Picos, PI, Brasil
luis.ramos@ufpi.edu.br

Viviany da Silva Araujo
Universidade Federal do Piauí (UFPI)
Picos, PI, Brasil
viviany.araujo@ufpi.edu.br

***Resumo*—Arquiteturas de microsserviços são um padrão dominante para construir aplicações escaláveis e resilientes. No entanto, a complexidade de rede e a comunicação entre serviços introduzem desafios de desempenho que precisam ser quantificados. Este artigo apresenta uma avaliação de desempenho da aplicação de microsserviços Spring PetClinic. A ferramenta de teste de carga Locust foi utilizada simulando três cenários distintos de carga: Leve (50 usuários), Moderado (100 usuários) e Pico (200 usuários), executando 5 repetições de cada para garantir validade estatística. Métricas chave de desempenho, como tempo de resposta, throughput (requisições por segundo) e taxa de sucesso, foram coletadas e analisadas, descartando períodos de aquecimento. Os resultados indicam que o sistema entra em saturação no cenário moderado, onde a taxa de sucesso caiu de 100% (Leve) para $73{,}52\%$ e o tempo médio de resposta aumentou 1.400%, de $283\,\mathrm{ms}$ para $4.256\,\mathrm{ms}$. O cenário de pico resultou em colapso do sistema ($2{,}83\%$ de sucesso), indicando um gargalo de CPU no serviço `customers-service`, que atingiu mais de 600% de uso. Esta análise fornece insights sobre o comportamento da aplicação sob estresse e ajuda a identificar limites de escalabilidade.**

***Index Terms*—Teste de Desempenho, Spring Boot, Spring PetClinic, Locust, Escalabilidade, Microsserviços, Gargalo de CPU.**

I. Introdução

A adoção de arquiteturas de microsserviços revolucionou o desenvolvimento de software, permitindo que grandes aplicações sejam decompostas em serviços menores, independentes e de fácil manutenção [1]. Essa abordagem oferece vantagens significativas em termos de escalabilidade, resiliência e *deploy* independente. Além disso, ela também introduz uma complexidade considerável na comunicação inter-serviços e no monitoramento de desempenho ponta-a-ponta [2].

O projeto Spring PetClinic [3], mantido pela comunidade Spring, serve como uma aplicação de referência padrão para demonstrar e comparar diferentes pilhas de tecnologia. Sua versão de microsserviços é um alvo ideal para estudos de desempenho, pois apresenta uma implementação realista de serviços desacoplados. Componentes como API Gateway, Clientes, Veterinários, Visitas se comunicam para cumprir às solicitações do usuário.

Compreender o comportamento de microsserviços sob diferentes cargas é essencial para garantir confiabilidade e eficiência. A incapacidade de identificar gargalos de desempenho pode resultar em degradação da experiência do usuário, falhas em cascata nos serviços interdependentes e aumento significativo dos custos operacionais. Nesse contexto, avaliações de desempenho controladas permitem quantificar métricas-chave, oferecendo insights valiosos para otimização e dimensionamento adequado da infraestrutura.

O objetivo deste trabalho é medir e analisar o desempenho básico da aplicação Spring PetClinic Microservices sob cargas de usuário controladas. Para isso, utilizamos a ferramenta de teste de carga baseada em Python [4], Locust [5], simulando três cenários de tráfego: leve, moderado e pico. Dessa forma, foi possível quantificar o impacto da carga nas principais métricas de desempenho e identificar os limites de escalabilidade da configuração padrão da aplicação.

II. Sistema e Metodologia

Esta seção descreve a aplicação-alvo utilizada neste estudo, o ambiente de teste e a metodologia adotada para a avaliação de desempenho. Inicialmente, apresentamos a arquitetura da aplicação Spring PetClinic Microservices. Em seguida, detalhamos o ambiente de execução, as ferramentas e parâmetros utilizados para a simulação de carga. Por fim, apresentamos os cenários de teste, os *endpoints* selecionados para avaliação e as métricas monitoradas, fornecendo uma visão completa do procedimento adotado.

A. Sistema Sob Teste (SUT)

O sistema sob teste é a versão de microsserviços do Spring PetClinic [3]. A arquitetura é composta por vários serviços principais, executados localmente via `docker-compose`, que se comunicam entre si para atender às solicitações dos usuários. Foi alocado um limite de $2\,\mathrm{GiB}$ de memória RAM para cada serviço principal no arquivo `docker-compose.yml` para evitar falhas por OOM (*Out Of Memory*). Entre os serviços, destacam-se:

- **API Gateway:** Ponto de entrada único para todas as requisições externas, roteando o tráfego para os serviços apropriados.

- **Customers Service:** Gerencia informações de donos (owners) e seus animais (pets).
- **Vets Service:** Gerencia informações sobre os veterinários.
- **Visits Service:** Gerencia os registros de visitas dos animais.
- **Database (MySQL/HSQLDB):** Banco de dados utilizado pelos serviços para persistência de dados.

O teste de carga foi direcionado ao API Gateway, simulando o tráfego de usuários reais e garantindo que todas as requisições fossem roteadas corretamente para os serviços correspondentes. O *mix* de requisições do plano de teste foi projetado para exercitar os principais *endpoints* da aplicação, priorizando as operações mais frequentes e representativas do uso típico do sistema, conforme descrito abaixo:

- 40%: `GET /api/customer/owners` (Listar donos)
- 30%: `GET /api/customer/owners/{id}` (Buscar dono)
- 20%: `GET /api/vet/vets` (Listar veterinários)
- 10%: `POST /api/customer/owners` (Cadastrar dono)

B. Metodologia de Teste

Os testes foram conduzidos usando o Locust [5], uma ferramenta que permite definir o comportamento do usuário em código Python e gerar carga controlada sobre os *endpoints* da aplicação. Essa abordagem possibilita mensurar métricas de desempenho relevantes, como tempo médio e máximo de resposta, *throughput*, número total de requisições, taxa de sucesso e ocorrência de erros, fornecendo uma visão abrangente do comportamento do sistema sob diferentes condições de estresse.

1) Cenários de Carga: Para avaliar o desempenho do sistema, foram definidos três cenários de carga representativos de diferentes níveis de utilização: leve, moderado e pico. O cenário leve simula uma situação de baixa demanda, o moderado representa o uso típico esperado e o pico corresponde a um nível extremo de tráfego que testa os limites da infraestrutura. Cada cenário foi executado 5 vezes para garantir a consistência estatística dos resultados, permitindo calcular a média e o desvio padrão das métricas coletadas. A Tabela I apresenta os parâmetros de cada cenário.

Tabela I
PARÂMETROS DOS CENÁRIOS DE TESTE DE CARGA

Parâmetro	Leve	Médio	Pico
Usuários	50	100	200
Taxa (spawn)	5	10	20
Duração	10 min	10 min	5 min
Aquecimento	1 min	1 min	30 s

2) Coleta e Análise de Métricas: A execução automatizada da bateria de testes foi realizada por meio de scripts em PowerShell, garantindo a repetibilidade das 5 execuções de cada cenário. Para a análise, os dados gerados pelo Locust nos arquivos `_stats_history.csv` foram processados utilizando um script Python [4] baseado em Pandas [6]. Assim, foi possível filtrar o período inicial de aquecimento (*warm-up*) conforme definido na Tabela I e focar apenas no comportamento do sistema sob carga estável. As métricas coletadas têm como objetivo fornecer uma visão abrangente do desempenho do sistema:

- **Tempo Médio de Resposta (ms):** O tempo médio para uma requisição ser completada.
- **Tempo Máximo de Resposta (ms):** O pior tempo de resposta observado.
- **Throughput (Req/s):** O número de requisições que o sistema processou por segundo.
- **Taxa de Sucesso (%):** A porcentagem de requisições que retornaram sem erros (HTTP 2xx).
- **Uso de Recursos:** Uso de CPU e Memória dos contêineres, coletado manualmente via `docker stats`.

III. RESULTADOS E ANÁLISE

Esta seção apresenta os resultados consolidados das execuções de cada cenário de teste, considerando as médias e o desvio padrão das principais métricas de desempenho. O objetivo é analisar o comportamento do sistema sob diferentes níveis de carga e identificar o ponto em que ocorre a saturação dos serviços. Essa análise permite compreender o limite operacional da aplicação e os impactos da concorrência entre os microsserviços.

A. Visão Geral das Métricas de Desempenho

A Tabela II apresenta as principais métricas obtidas, destacando a variação do desempenho à medida que a carga de usuários aumenta. Observa-se que o sistema mantém estabilidade no cenário leve, com 100% de sucesso e tempo médio de resposta de apenas 283 ms. No cenário médio, há uma queda significativa no desempenho, com a taxa de sucesso reduzida para 73,5% e o tempo médio ultrapassando 4,3 s. Já no cenário pico, o sistema entra em colapso, apresentando apenas 2,8% de sucesso e atingindo o tempo máximo configurado de 10 s em diversas requisições.

Tabela II
RESUMO CONSOLIDADO DAS MÉTRICAS DE DESEMPENHO (MÉDIA DE 5 EXECUÇÕES)

Métrica	Leve (50 usuários)	Médio (100 usuários)	Pico (200 usuários)
Throughput (Req/s)	22,21	16,97	85,31
Tempo Médio (ms)	282,53	4.256,12	335,48
Tempo Máximo (ms)	2.286	10.000	10.000
Taxa de Sucesso (%)	100,00	73,52	2,83
Total Req (por exec.)	11901	9110	22878
Total Erros (global)	0	15519	111157

Na Fig. 1, observa-se um aumento acentuado no tempo médio de resposta no cenário médio, atingindo 4,3 s, em contraste com o cenário leve (282,5 ms) e o pico (335,5 ms). Esse comportamento indica instabilidade sob carga intermediária, possivelmente associada à saturação de recursos.

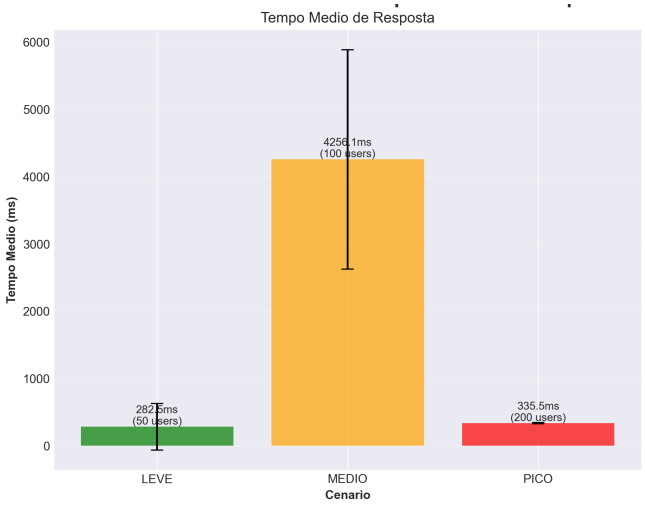


Figura 1. Tempo médio de resposta nos três cenários (Leve, Médio e Pico).

A Fig. 2 mostra o throughput médio por cenário. O médio apresentou o maior valor numérico (85,3 req/s), seguido dos cenários leve (22,2 req/s) e médio (17,0 req/s). Entretanto, o alto throughput no cenário de pico é ilusório, pois resulta do grande volume de requisições com falha que retornam rapidamente.

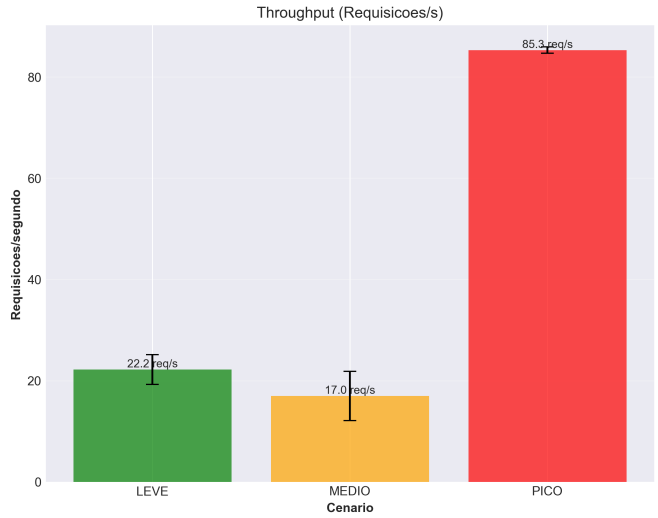


Figura 2. Throughput (taxa de processamento) comparativo entre os cenários.

Na Fig. 3, a taxa de sucesso manteve-se em 100% no cenário leve, atendendo plenamente ao SLA (linha tracejada azul). Por outro lado, o cenário médio apresentou forte oscilação, com intervalo de confiança entre 40% e valores superiores a 100%, refletindo comportamento instável e fora da faixa representada no gráfico.

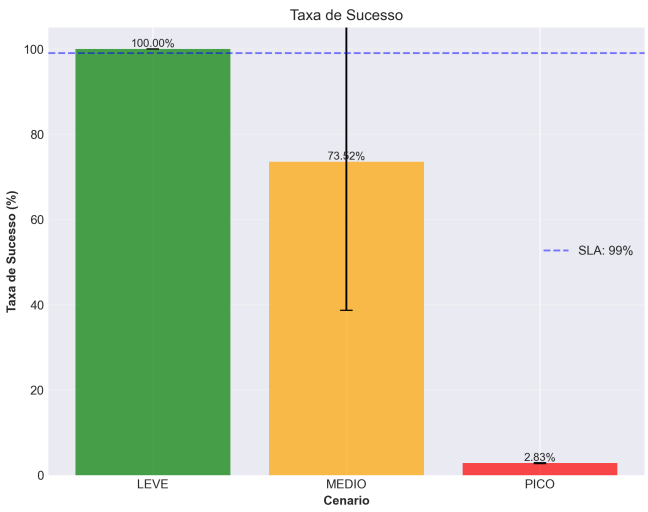


Figura 3. Taxa de sucesso das requisições em cada cenário.

Por fim, a Fig. 4 mostra que o tempo máximo de resposta atingiu o limite de timeout de 10 s nos cenários médio e pico, enquanto o cenário leve manteve um valor significativamente inferior (2.286 ms).

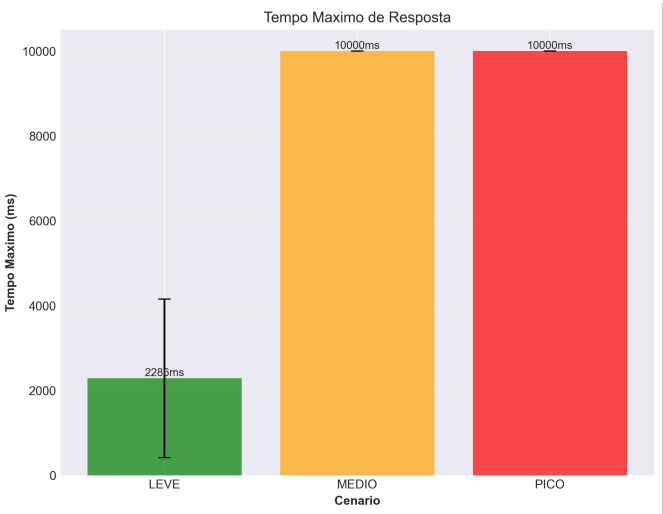


Figura 4. Tempo máximo de resposta observado nos testes.

B. Análise Detalhada por Métrica e Variabilidade

Abaixo é apresentando box plots que detalham a variabilidade estatística das métricas ao longo das 5 execuções de cada cenário.

1) Tempo de Resposta: O tempo médio de resposta foi de 283 ms no leve (Fig. 1), com baixa variabilidade (Fig. 5). No médio, aumentou significamente para 4.256 ms (4,3 s), com enorme variabilidade (caixa de 3.000 a 5.500 ms), indicando instabilidade crítica. O tempo máximo atingiu 10 s (Fig. 4). O valor de 335 ms na Fig. 1 no pico é enganoso devido às falhas rápidas.

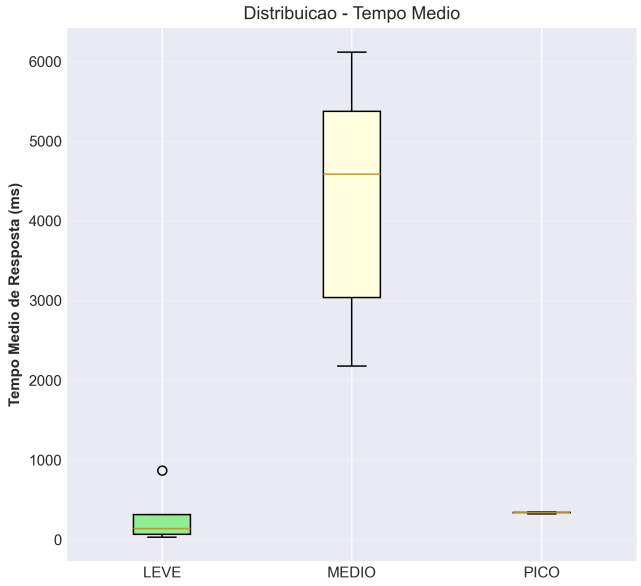


Figura 5. Distribuição do tempo médio de resposta (ms) nas 5 execuções de cada cenário.

2) Throughput: No cenário leve, o throughput médio foi de 22,2 req/s (Fig. 2). O box plot correspondente (Fig. 6) mostra baixa variabilidade (caixa entre 21 e 24 req/s), indicando estabilidade. No cenário médio, o throughput caiu para 16,97 req/s, com alta variabilidade (caixa entre 12 e 20 req/s), confirmando a saturação e instabilidade do sistema. O valor de 85,31 req/s no pico é artificial, refletindo falhas rápidas, como mostra a concentração no box plot.

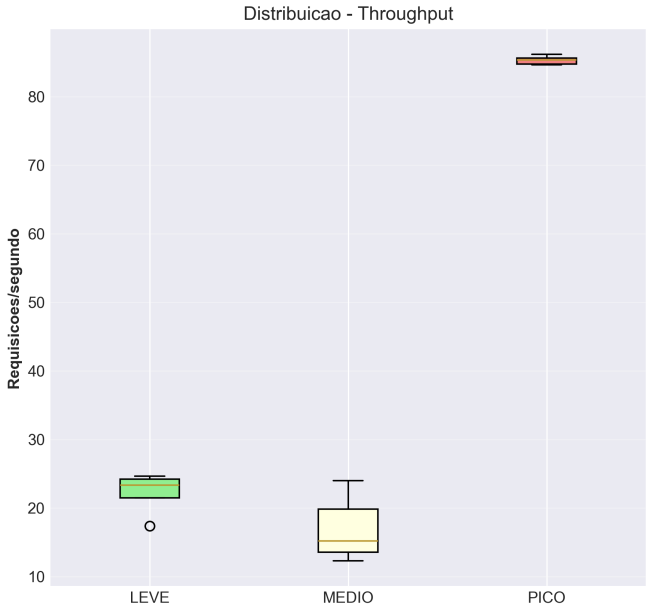


Figura 6. Distribuição do throughput (requisições por segundo) nas 5 execuções de cada cenário.

3) Taxa de Sucesso: A taxa de sucesso foi de 100% no leve (Fig. 3 e 7). No médio, caiu para 73,52%, com variabilidade extrema (Fig. 7, caixa de 40% a 100%). Isso demonstra que o sistema estava operando no limiar do colapso, com resultados muito inconsistentes entre as execuções. No pico, o colapso foi consistente, com sucesso médio de apenas 2,83% e baixa variabilidade.

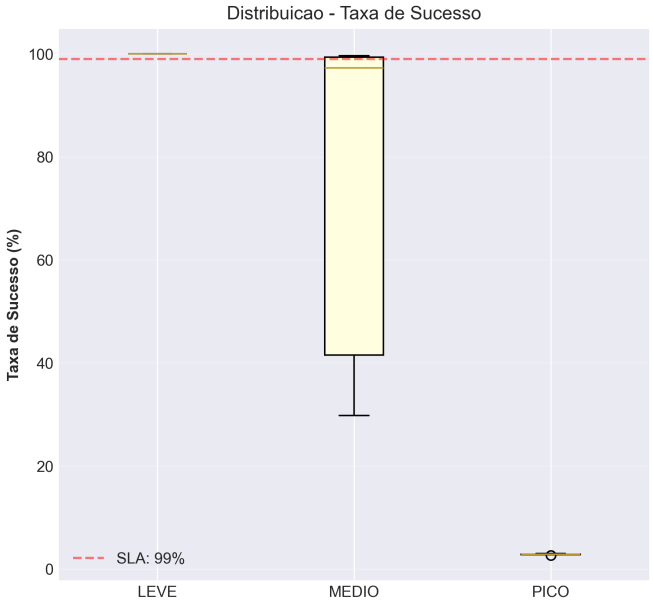


Figura 7. Distribuição da taxa de sucesso (%) nas 5 execuções de cada cenário.

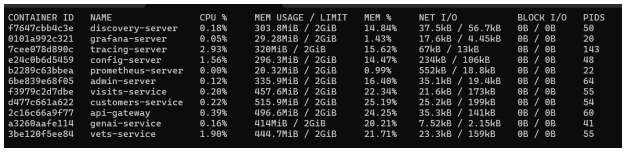
C. Identificação do Gargalo: Utilização de Recursos

O monitoramento da utilização de recursos via `docker stats` foi essencial para identificar a causa raiz da degradação observada nos testes de carga. Enquanto a memória permaneceu abaixo de 50% do limite de 2 GiB em todos os cenários (Tabela III), a CPU foi o principal fator limitante do desempenho do sistema.

Tabela III
USO DE RECURSOS DO CONTÊINER `customers-service`.

Cenário	CPU (%)	MEM (%) de 2GB
Ocioso (Baseline)	0,22	25,19
Leve	69,82	28,48
Médio	615,62	48,37
Pico	651,89	49,66

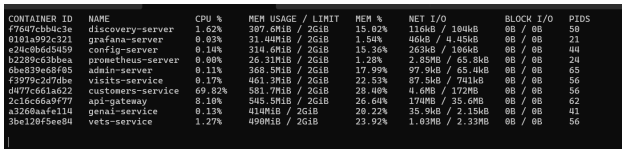
No cenário ocioso, o `customers-service` apresentou apenas 0,22% de uso de CPU, conforme mostrado na Figura 8. Esse comportamento evidencia que o sistema não apresenta sobrecarga significativa em repouso e que os serviços permanecem estáveis em estado ocioso, mantendo consumo mínimo de recursos.



CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
f7647cbb4c3e	discovery-server	0.18%	303.8MiB / 2GiB	14.84%	37.5kB / 56.7kB	0B / 0B	50
0101a992c321	grafana-server	0.05%	29.28MiB / 2GiB	1.43%	17.6kB / 4.45kB	0B / 0B	20
7cee078d899c	tracing-server	2.93%	320MiB / 2GiB	15.62%	67kB / 13kB	0B / 0B	143
e24c0b6d5459	config-server	1.56%	296.3MiB / 2GiB	14.47%	234kB / 106kB	0B / 0B	48
b2289c63bbea	prometheus-server	0.00%	20.32MiB / 2GiB	0.99%	552kB / 18.8kB	0B / 0B	22
6be839e68f05	admin-server	0.12%	335.9MiB / 2GiB	16.40%	35.1kB / 19.4kB	0B / 0B	64
f3979c2d7dbe	visits-service	0.20%	457.6MiB / 2GiB	22.34%	21.6kB / 173kB	0B / 0B	55
d477c661a622	customers-service	0.22%	515.9MiB / 2GiB	25.19%	25.2kB / 199kB	0B / 0B	54
2c16c66a9f77	api-gateway	0.39%	496.6MiB / 2GiB	24.25%	35.3kB / 141kB	0B / 0B	60
a3268aafe114	genai-service	0.16%	414MiB / 2GiB	20.21%	7.52kB / 2.15kB	0B / 0B	41
3be120f5ee84	vets-service	1.90%	444.7MiB / 2GiB	21.71%	23.3kB / 159kB	0B / 0B	55

Figura 8. Estado ocioso (Baseline) — consumo de 0,22% de CPU no contêiner principal.

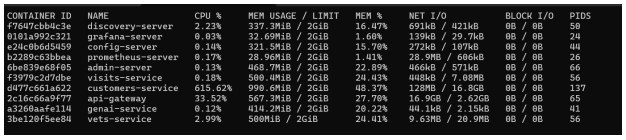
A Figura 9 apresenta o comportamento do sistema no cenário leve. O serviço `customers-service` atinge aproximadamente 69,82% de utilização de CPU, indicando atividade moderada e boa distribuição da carga entre os núcleos disponíveis. Não há sinais de contenção nem degradação perceptível de desempenho, caracterizando operação estável e saudável.



CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
f7647cbb4c3e	discovery-server	1.62%	307.6MiB / 2GiB	15.02%	116kB / 104kB	0B / 0B	50
0101a992c321	grafana-server	0.03%	31.44MiB / 2GiB	1.54%	46kB / 4.45kB	0B / 0B	21
e24c0b6d5459	config-server	0.14%	314.6MiB / 2GiB	15.36%	263kB / 106kB	0B / 0B	44
b2289c63bbea	prometheus-server	0.00%	26.31MiB / 2GiB	1.28%	2.85MB / 65.8kB	0B / 0B	24
6be839e68f05	admin-server	0.11%	368.5MiB / 2GiB	17.99%	97.9kB / 65.4kB	0B / 0B	65
f3979c2d7dbe	visits-service	0.17%	461.3MiB / 2GiB	22.53%	87.5kB / 741kB	0B / 0B	56
d477c661a622	customers-service	69.82%	581.7MiB / 2GiB	28.40%	4.6MB / 172MB	0B / 0B	56
2c16c66a9f77	api-gateway	0.18%	545.5MiB / 2GiB	26.64%	179MB / 35.6MB	0B / 0B	62
a3268aafe114	genai-service	0.13%	414MiB / 2GiB	20.22%	35.9kB / 2.15kB	0B / 0B	41
3be120f5ee84	vets-service	1.27%	490MiB / 2GiB	23.92%	1.03MB / 2.33MB	0B / 0B	56

Figura 9. Cenário LEVE — consumo de 69,82% de CPU no contêiner `customers-service`.

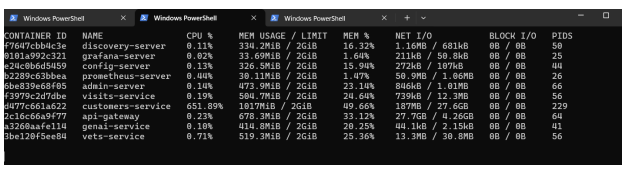
A Figura 10 demonstra o início da saturação do serviço principal. Com aproximadamente 615,62% de uso de CPU, o contêiner tenta utilizar mais de seis núcleos simultaneamente, evidenciando o limite da escalabilidade horizontal do serviço. Esse comportamento caracteriza o surgimento do gargalo de processamento, com indícios de contenção de threads e aumento progressivo da latência nas requisições.



CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
f7647cbb4c3e	discovery-server	2.23%	337.3MiB / 2GiB	16.47%	691kB / 421kB	0B / 0B	50
0101a992c321	grafana-server	0.03%	32.69MiB / 2GiB	1.60%	139kB / 29.7kB	0B / 0B	24
e24c0b6d5459	config-server	0.14%	321.5MiB / 2GiB	15.70%	272kB / 107kB	0B / 0B	44
b2289c63bbea	prometheus-server	0.17%	28.96MiB / 2GiB	1.41%	28.9MB / 606kB	0B / 0B	26
6be839e68f05	admin-server	0.13%	468.7MiB / 2GiB	22.89%	466kB / 571kB	0B / 0B	66
f3979c2d7dbe	visits-service	0.18%	500.4MiB / 2GiB	24.43%	448kB / 7.08MB	0B / 0B	56
d477c661a622	customers-service	615.62%	990.6MiB / 2GiB	48.37%	128MB / 16.8GB	0B / 0B	137
2c16c66a9f77	api-gateway	33.52%	567.3MiB / 2GiB	27.70%	16.9GB / 2.62GB	0B / 0B	65
a3268aafe114	genai-service	0.12%	414.2MiB / 2GiB	20.22%	44.1kB / 2.15kB	0B / 0B	41
3be120f5ee84	vets-service	2.99%	500MiB / 2GiB	24.41%	9.63MB / 20.9MB	0B / 0B	56

Figura 10. Cenário MÉDIO — consumo de 615,62% de CPU no contêiner `customers-service`.

Por fim, a Figura 11 representa o cenário de carga máxima, no qual o sistema atinge o limite total de utilização de CPU. Com cerca de 651,89% de uso, o contêiner entra em estado de contenção severa, resultando em degradação generalizada do throughput e aumento abrupto no tempo de resposta. Esse comportamento confirma a saturação completa do microsserviço `customers-service`, identificado como o principal gargalo da arquitetura.



CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
f7647cbb4c3e	discovery-server	0.11%	334.2MiB / 2GiB	16.32%	1.16MB / 681kB	0B / 0B	50
0101a992c321	grafana-server	0.02%	33.69MiB / 2GiB	1.64%	211kB / 50.8kB	0B / 0B	25
e24c0b6d5459	config-server	0.13%	326.5MiB / 2GiB	15.94%	272kB / 107kB	0B / 0B	44
b2289c63bbea	prometheus-server	0.44%	30.11MiB / 2GiB	1.47%	50.9MB / 1.86MB	0B / 0B	26
6be839e68f05	admin-server	0.14%	473.9MiB / 2GiB	23.14%	846kB / 1.01MB	0B / 0B	66
f3979c2d7dbe	visits-service	0.19%	504.7MiB / 2GiB	24.64%	739kB / 12.3MB	0B / 0B	56
d477c661a622	customers-service	651.89%	1017MiB / 2GiB	49.66%	187MB / 27.6GB	0B / 0B	229
2c16c66a9f77	api-gateway	0.23%	678.3MiB / 2GiB	33.12%	27.7GB / 4.26GB	0B / 0B	64
a3260aafe114	genai-service	0.10%	414.8MiB / 2GiB	20.25%	44.1kB / 2.15kB	0B / 0B	41
3be120f5ee84	vets-service	0.71%	519.3MiB / 2GiB	25.36%	13.3MB / 30.8MB	0B / 0B	56

Figura 11. Cenário PICO — consumo de 651,89% de CPU no contêiner `customers-service`.

A exaustão de CPU no `customers-service` explica diretamente os altos tempos de resposta, a queda de throughput e a redução da taxa de sucesso observados nos cenários médio e pico. O serviço torna-se sobrecarregado a ponto de não conseguir processar as requisições a tempo, ocasionando *timeouts* em cascata e degradação global do sistema.

IV. Conclusão

Este estudo realizou uma análise de desempenho da aplicação de microsserviços Spring PetClinic usando testes de carga automatizados com Locust. A metodologia de três cenários (Leve, Médio, Pico), executada 5 vezes cada, permitiu uma avaliação estatística robusta do comportamento do sistema.

Os resultados demonstram que a aplicação é estável sob a carga Leve (50 usuários), com 100% de sucesso e tempo de resposta de 283 ms. No entanto, o sistema atinge um ponto de saturação abrupto no cenário médio (100 usuários). O gargalo foi identificado no `customers-service`, cuja CPU saturou em mais de 600% (Fig. 10), causando uma queda na taxa de sucesso para 73,52% e um aumento de 1.400% no tempo de resposta (4,2 s). O cenário pico (200 usuários) levou o sistema ao colapso total (2,83% de sucesso), confirmando que a arquitetura não escala além de 50 usuários sem otimização ou alocação de mais recursos de CPU.

A. Limitações e Trabalhos Futuros

Uma limitação deste estudo é o ambiente de teste. Os testes foram executados localmente em uma única máquina, o que significa que o cliente (Locust) e o servidor (Docker) competiram pelos mesmos recursos de CPU e rede. Embora tenhamos alocado 2 GiB de RAM por serviço (o que resolveu o gargalo de memória identificado em testes preliminares), o gargalo de CPU foi atingido rapidamente, o que pode ser exacerbado pela competição de recursos.

Como trabalhos futuros, sugere-se a repetição dos testes em um ambiente de nuvem (como Kubernetes no GKE ou EKS) com alocação dedicada de CPU por contêiner para analisar o impacto do auto-scaling e determinar o limite real de processamento do `customers-service`. Além disso, uma análise mais profunda usando ferramentas de APM (Application Performance Monitoring) como o Prometheus (já incluído no PetClinic) poderia identificar gargalos específicos em consultas de banco de dados ou métodos Java que causam o uso intensivo de CPU neste serviço.

Referências

[1] S. Newman, "Building Microservices: Designing Fine-Grained Systems," O'Reilly Media, 2015.

[2] C. Richardson, "Microservices Patterns: With examples in Java," Manning Publications, 2018.

[3] Spring Community, "Spring PetClinic Microservices," GitHub Repository. [Online]. Available: https://github.com/spring-petclinic/spring-petclinic-microservices

[4] Python Software Foundation, "Documentação do Python 3," [Online]. Available: https://docs.python.org/pt-br/3

[5] Locust.io, "Locust - An open source load testing tool," [Online]. Available: https://locust.io/

[6] Pandas Development Team, "Pandas Documentation," [Online]. Available: https://pandas.pydata.org/docs