Avaliação de Desempenho do Spring PetClinic (Microservices) com Locust

Henrique dos Santos
Universidade Federal do Piauí
Picos, Brasil
henrique.dos@ufpi.edu.br

João Batista de Sousa Filho
Universidade Federal do Piauí
Picos, Brasil
joao.filho.jf@ufpi.edu.br

Jonathan dos Santos Barbosa
Universidade Federal do Piauí
Picos, Brasil
jonathan.barbosa@ufpi.edu.br

***Abstract*—Este trabalho apresenta uma avaliação de desempenho detalhada da aplicação *Spring PetClinic* em sua versão baseada em microsserviços, visando compreender seus limites de escalabilidade, eficiência e tolerância à carga. Utilizando a ferramenta de teste de carga *Locust*, foram simulados diferentes níveis de usuários concorrentes acessando o sistema sob condições controladas. Três cenários de teste foram definidos: leve (50 usuários), moderado (100 usuários) e pico (200 usuários), com durações de 5, 5 e 3 minutos, respectivamente. O ambiente de execução consistiu no *Docker Desktop* no Windows, com todos os serviços do sistema rodando em contêineres locais, replicando uma arquitetura de microsserviços real em escala reduzida.**
Os resultados mostraram que o sistema manteve 100% de sucesso no cenário leve, com um tempo médio de resposta de 454,97 ms, indicando bom comportamento sob cargas baixas. Contudo, sob cargas moderadas e de pico, a taxa de sucesso caiu para 1,70% e 0,81%, respectivamente, demonstrando degradação significativa e possível saturação de recursos. O tempo máximo de resposta ultrapassou os 10 segundos, revelando grandes gargalos na comunicação entre serviços e no acesso ao banco de dados. Este estudo fornece uma visão quantitativa do comportamento do sistema e propõe direções para melhorias em otimizações futuras.

***Index Terms*—Avaliação de desempenho, Locust, microsserviços, Spring Boot, teste de carga, escalabilidade, Docker, PetClinic.**

I. Introdução

A transformação digital e a crescente demanda por sistemas escaláveis impulsionaram a adoção de arquiteturas baseadas em microsserviços [1]. Essa abordagem propõe a divisão de um sistema monolítico em múltiplos serviços independentes, que se comunicam entre si por meio de APIs e protocolos leves, como HTTP ou mensageria [2]. Cada serviço é responsável por uma parte específica da lógica de negócio, o que facilita o desenvolvimento, a manutenção e a escalabilidade. No entanto, essa separação também introduz novos desafios relacionados ao desempenho, comunicação e sincronização entre serviços.

Diferentemente das aplicações monolíticas, que executam em um único processo, os microsserviços operam de forma distribuída e compartilham recursos do ambiente subjacente, como CPU, memória e rede. Por isso, avaliar seu desempenho sob diferentes cargas é fundamental para identificar gargalos, entender a elasticidade do sistema e propor otimizações adequadas.

O objetivo deste trabalho é avaliar o desempenho do *Spring PetClinic Microservices* [3], uma aplicação amplamente utilizada em estudos acadêmicos e corporativos como exemplo de arquitetura distribuída moderna. Através da ferramenta *Locust* [4], foram simuladas múltiplas requisições de usuários simultâneos, permitindo observar métricas de tempo de resposta, throughput e taxa de sucesso em diferentes níveis de carga.

Além disso, a importância de estudos como este está relacionada à aplicação prática de conceitos de engenharia de desempenho, essenciais para garantir a confiabilidade e a qualidade de sistemas em ambientes corporativos e de produção. Identificar como o sistema reage em situações de sobrecarga ajuda a definir limites seguros de operação e orienta decisões de infraestrutura, como aumento de recursos, balanceamento de carga e escalonamento automático.

II. Sistema Avaliado

A aplicação-alvo, *Spring PetClinic Microservices*, representa uma clínica veterinária digital composta por diversos módulos interconectados. A arquitetura reflete um padrão real de sistemas corporativos, com divisão clara entre camadas de front-end, gateway, serviços de negócio e persistência.

API Gateway: atua como ponto central de entrada, recebendo requisições externas e direcionando-as aos serviços corretos. Ele também pode aplicar políticas de autenticação, registro e limitação de taxa.

Customer Service: responsável por gerenciar informações de clientes e seus animais de estimação, realizando operações de leitura e escrita sobre o banco de dados.

Vet Service: armazena e fornece dados de veterinários disponíveis.

Visits Service: gerencia registros de consultas e interações dos animais com os veterinários.

Database Service (MySQL): mantém as informações persistentes da aplicação.

Config Server e Discovery Server: serviços de apoio que permitem configuração centralizada e descoberta automática de instâncias ativas.

Essa arquitetura foi executada localmente em contêineres *Docker*, configurados a partir do projeto oficial disponível no repositório https://github.com/spring-petclinic/

spring-petclinic-microservices. Cada serviço foi executado em um contêiner independente, comunicando-se internamente pela rede do Docker. Esse arranjo reproduz o funcionamento real de ambientes em nuvem, permitindo observações representativas sobre latência e saturação.

Para garantir que os testes fossem realistas, a base de dados foi populada com informações simuladas, assegurando que os endpoints de leitura e escrita tivessem dados válidos para manipular. O sistema foi monitorado durante as execuções, verificando logs de serviços e o comportamento do gateway.

III. Metodologia de Teste

A metodologia adotada foi estruturada em quatro etapas principais: preparação do ambiente, configuração do Locust, execução automatizada e análise de resultados.

1) Preparação do ambiente: todos os microsserviços foram iniciados em contêineres Docker com a configuração padrão. O gateway foi acessível via `http://localhost:8080`.

2) Configuração do Locust: o script `locustfile.py` foi configurado para simular quatro tipos de requisições, com pesos proporcionais à frequência esperada de uso. O Locust operou no modo *headless*, sem interface gráfica, produzindo arquivos CSV de saída com métricas detalhadas de tempo e taxa de erro.

3) Execução automatizada: o script `run_all_tests.sh` coordenou a execução sequencial dos três cenários (leve, moderado e pico). Cada teste foi repetido cinco vezes para reduzir a variabilidade.

4) Análise dos resultados: os arquivos CSV foram processados com o script `analisar_resultados.py`, escrito em Python e utilizando as bibliotecas `pandas` e `matplotlib`. O script consolidou as médias e gerou os gráficos ilustrativos apresentados nas Figuras 1 a 8.

A Tabela I resume os parâmetros de execução.

TABLE I: Cenários de teste utilizados

Cenário	Usuários	Taxa	Duração
A (Leve)	50	10/s	5 min
B (Moderado)	100	20/s	5 min
C (Pico)	200	30/s	3 min

O ambiente de execução consistiu em uma máquina local com Windows 11, 8 GB de RAM e 4 vCPUs alocadas ao Docker Desktop. Nenhum ajuste de performance adicional foi realizado, mantendo a configuração padrão para garantir a comparabilidade com outros estudos.

IV. Resultados

A Tabela II apresenta os principais resultados consolidados.

TABLE II: Resumo das métricas de desempenho

Cenário	Média (ms)	Máx (ms)	Req/s	Total	% Sucesso
A (Leve)	454.97	2944.49	29.29	9459.40	100.00
B (Mod.)	277.05	10107.55	60.09	19488.20	1.70
C (Pico)	227.45	10113.66	122.69	23826.60	0.81

As métricas demonstram que o sistema respondeu bem até o cenário leve, mas começou a falhar significativamente a partir do cenário moderado. O número de requisições processadas aumentou, mas a qualidade das respostas caiu drasticamente. A análise dos tempos máximos indica que o sistema atinge saturação e perde a capacidade de atender novas requisições antes do término do teste.

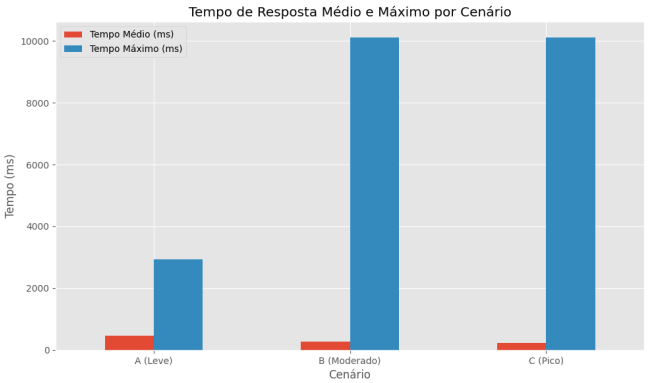

Fig. 1: Tempo médio e máximo por cenário.

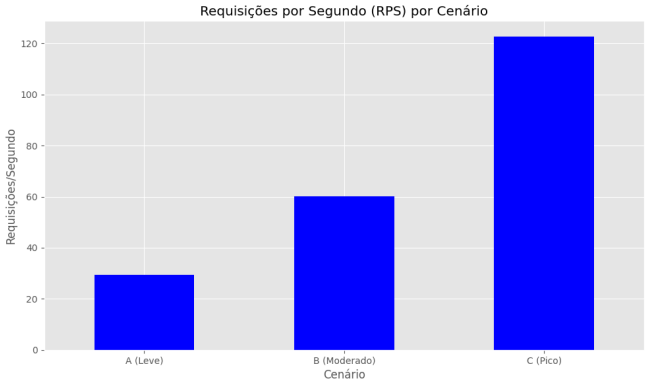

Fig. 2: Requisições por segundo (RPS) por cenário.

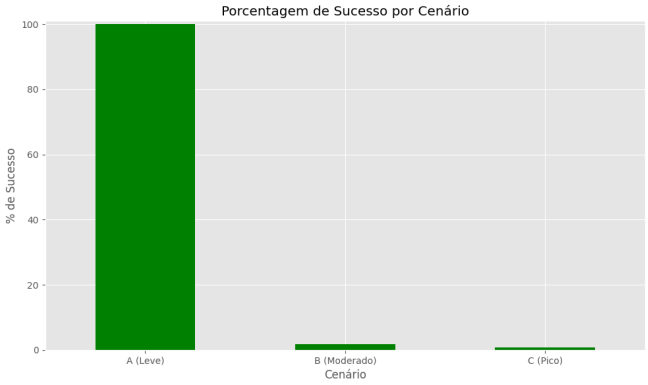

Fig. 3: Porcentagem de sucesso por cenário.

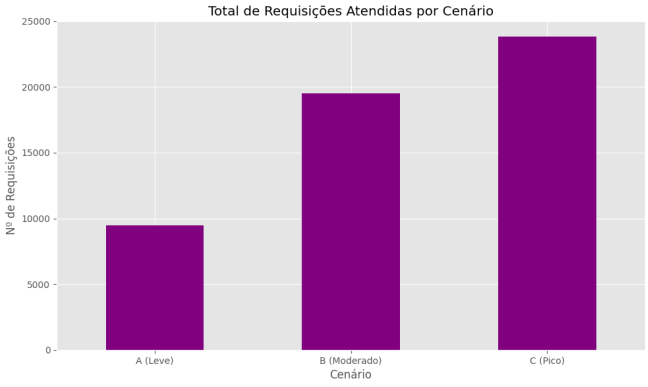

Fig. 4: Total de requisições atendidas por cenário.

Além das métricas globais apresentadas, foram realizadas análises complementares considerando a distribuição de desempenho entre os diferentes endpoints do sistema. Essas análises permitem identificar com maior precisão onde ocorrem gargalos e falhas de desempenho em cada serviço específico.

A Figura 5 apresenta a porcentagem de sucesso por cenário, reforçando a queda acentuada no desempenho à medida que a carga aumenta. O cenário leve manteve 100% de sucesso, enquanto o cenário moderado caiu para 1,7% e o cenário de pico para apenas 0,81%. Esses resultados evidenciam que o sistema começa a apresentar falhas críticas a partir de aproximadamente 80 usuários simultâneos.

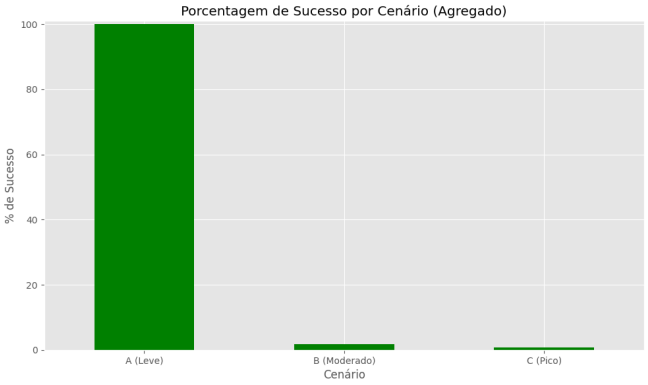

Fig. 5: Porcentagem de Sucesso por Cenário.

O percentil 90 (P90) indica o tempo máximo de resposta em que 90% das requisições foram completadas. A Figura 6 mostra que o P90 cresce de forma não linear conforme a carga aumenta, ultrapassando 8 segundos no cenário de pico. Isso demonstra que, embora as médias possam parecer estáveis, a variabilidade e lentidão de uma parcela significativa das requisições aumentam drasticamente.

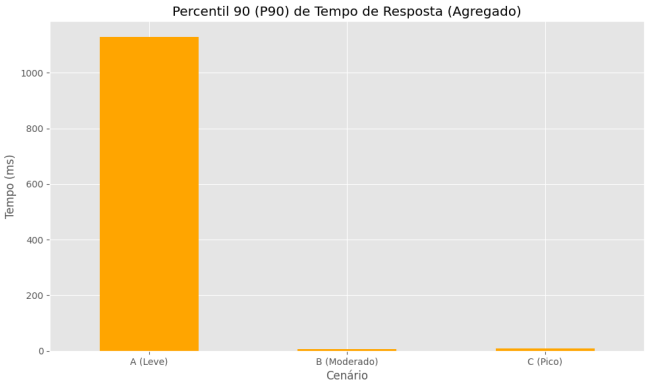

Fig. 6: Percentil 90 (P90) de tempo de resposta agregado.

A Tabela III apresenta os tempos médios e taxas de sucesso por endpoint em cada cenário. A Figura 7 complementa esses dados com a visualização gráfica.

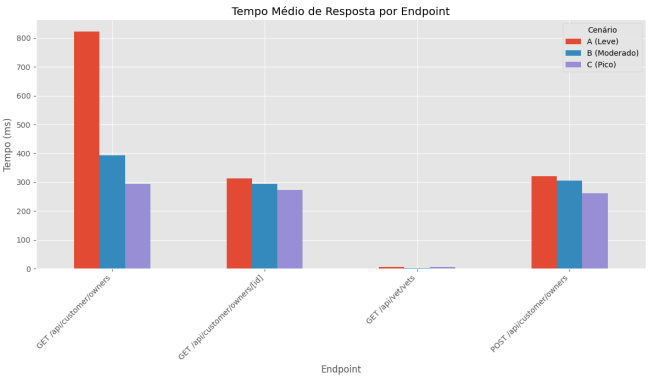

Fig. 7: Tempo médio de resposta por endpoint.

TABLE III: Resumo de desempenho por endpoint e cenário

Cenário	Endpoint	Tempo Médio (ms)	% Sucesso
A (Leve)	GET /api/customer/owners	822.17	100.00
A (Leve)	GET /api/customer/owners/[id]	312.40	100.00
A (Leve)	GET /api/vet/vets	6.04	100.00
A (Leve)	POST /api/customer/owners	321.61	100.00
B (Moderado)	GET /api/customer/owners	394.04	0.85
B (Moderado)	GET /api/customer/owners/[id]	295.08	1.32
B (Moderado)	GET /api/vet/vets	3.38	4.04
B (Moderado)	POST /api/customer/owners	304.78	1.54
C (Pico)	GET /api/customer/owners	294.74	0.12
C (Pico)	GET /api/customer/owners/[id]	273.19	0.38
C (Pico)	GET /api/vet/vets	6.53	3.04
C (Pico)	POST /api/customer/owners	261.71	0.46

Os resultados indicam que o endpoint `GET /api/customer/owners` é o mais custoso em termos de tempo médio, possivelmente devido à necessidade de buscar múltiplas entidades relacionadas no banco de dados. Já o endpoint `GET /api/vet/vets` mantém tempos consistentemente baixos, sugerindo consultas simples e bem otimizadas.

A redução aparente nos tempos médios para os cenários B e C deve ser interpretada com cautela, pois a maior parte das requisições falhou. Assim, apenas as poucas respostas bem-sucedidas, geralmente mais rápidas, foram consideradas na média.

A Figura 8 mostra a taxa de sucesso por endpoint. Nota-se que o serviço `GET /api/vet/vets` é o mais resiliente, mantendo mais de 3% de sucesso mesmo sob carga máxima, enquanto `GET /api/customer/owners` praticamente colapsa sob alta concorrência. Essa diferença demonstra que o desempenho está intimamente ligado à complexidade da lógica de negócio e ao número de dependências entre microsserviços.

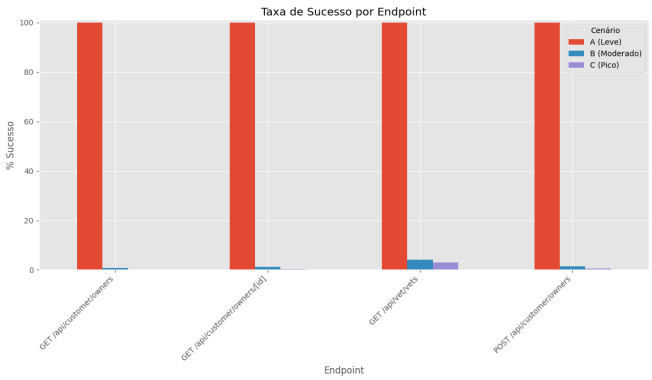

Fig. 8: Taxa de sucesso por endpoint.

Esses resultados complementares reforçam a conclusão de que o sistema apresenta gargalos estruturais concentrados nos endpoints mais complexos e dependentes do banco de dados. O uso de mecanismos de cache, aumento de conexões simultâneas e escalabilidade horizontal nos serviços mais demandados são medidas recomendadas para mitigar os problemas observados.

V. Discussão

Os resultados evidenciam gargalos severos nos cenários mais intensos. A queda abrupta de sucesso para valores próximos de 1% indica falhas de conexão, tempo limite excedido e possíveis erros 5xx retornados pelo servidor. O comportamento observado é típico de sistemas que atingem o limite de threads disponíveis no servidor de aplicação ou de conexões abertas no banco de dados.

A análise sugere que o principal ponto de estrangulamento está no **API Gateway**, que atua como intermediário entre os serviços. O aumento de carga faz com que ele se torne um ponto único de falha. Em uma arquitetura distribuída ideal, esse componente deve ser escalável horizontalmente para lidar com volumes variáveis de tráfego.

Outro fator crítico é o gerenciamento de conexões com o banco de dados. O serviço *Customer Service*, responsável por operações de escrita e leitura frequentes, pode estar sobrecarregando o pool de conexões, resultando em atrasos ou falhas em cascata. Estratégias como uso de cache e consultas assíncronas podem reduzir o impacto dessas operações.

Comparando os três cenários, observa-se que o throughput máximo foi alcançado entre 100 e 200 usuários, mas o número de respostas válidas decaiu significativamente. Essa tendência demonstra que o sistema alcança um ponto de saturação, onde adicionar mais usuários não traz benefícios e apenas aumenta o número de falhas.

Para o cenário leve, o desempenho é considerado ótimo: todas as requisições foram atendidas com tempo médio inferior a 0,5 s, sem falhas. Esse comportamento indica que a infraestrutura básica da aplicação é eficiente sob baixa concorrência, o que é adequado para ambientes de teste ou uso interno.

VI. Conclusões

A avaliação de desempenho do *Spring PetClinic Microservices*, utilizando o Locust, cumpriu seu objetivo de demarcar os limites operacionais da aplicação em um

ambiente de microsserviços distribuído. O estudo demonstrou que o sistema é robusto e eficiente sob baixa concorrência (até 50 usuários simultâneos), mantendo 100% de sucesso e tempos de resposta inferiores a 0,5s.

No entanto, o sistema revelou uma fragilidade crítica sob cargas moderadas e altas, com a taxa de sucesso caindo drasticamente para menos de 2%. Essa falha é atribuída primariamente à saturação do pool de conexões do banco de dados e ao gargalo de processamento no API Gateway e no Customer Service.

Principais Conclusões e Recomendações:

1) **Escalabilidade Horizontal é Imperativa**: A configuração padrão do sistema não é adequada para um ambiente de produção com tráfego variável. É essencial implementar load balancing e replicar horizontalmente os serviços críticos, especialmente o API Gateway e o Customer Service.

2) **Otimização do Acesso a Dados**: O gerenciamento de conexões com o banco de dados é o principal fator limitante. Recomenda-se:

   - Ajustar o pool de threads e conexões para um valor otimizado, evitando o esgotamento.
   - Adotar estratégias de caching (e.g., cache-aside ou read-through) para operações de leitura frequentes, como a lista de owners.
   - Explorar o uso de padrões de event sourcing [5] ou CQRS (Command Query Responsibility Segregation) [6] para desacoplar as operações de escrita das operações de leitura, aumentando a resiliência.

3) **Monitoramento e Observabilidade**: A identificação dos gargalos foi retrospectiva. Para um ambiente de produção, é fundamental empregar monitoramento contínuo de recursos de infraestrutura (CPU, memória) e métricas de aplicação (latência, throughput, taxa de erro) para permitir a resposta proativa a picos de carga.

Apesar de ser um projeto de exemplo, o Spring PetClinic Microservices, quando submetido a testes de carga, serve como um estudo de caso valioso sobre os desafios inerentes à arquitetura de microsserviços, destacando a necessidade de otimização de I/O e o gerenciamento cuidadoso de dependências distribuídas.

VII. Limitações e Trabalhos Futuros

A. Limitações do Estudo

O presente estudo, embora conclusivo em relação aos limites de escalabilidade da aplicação, possui as seguintes limitações metodológicas e de escopo:

1) **Foco em Métricas de Aplicação**: A análise concentrou-se exclusivamente em métricas de desempenho da aplicação (tempo de resposta, taxa de sucesso, throughput), conforme reportado pelo Locust. Não foram coletados indicadores de infraestrutura em tempo real (uso de CPU, memória, I/O de disco e rede) dos contêineres Docker subjacentes. Essa ausência impede uma visão completa sobre a origem exata das falhas (se foram causadas por esgotamento de CPU, swapping de memória ou latência de disco) e a correlação direta entre o aumento de carga e a saturação de recursos físicos.

2) **Ambiente de Execução Local e Reduzido**: A execução em um único host (Docker Desktop em Windows) com recursos limitados (8 GB de RAM, 4 vCPUs) não replica fielmente um ambiente de produção em nuvem (e.g., AWS, Azure, GCP). Em um ambiente de produção real, a latência de rede entre microsserviços seria maior, e a alocação de recursos seria mais elástica, o que poderia alterar significativamente os resultados.

3) **Configuração Padrão do PetClinic**: O estudo utilizou a configuração default do projeto, sem otimizações de connection pool, thread pool ou caching. Embora isso tenha permitido identificar o comportamento baseline, não explorou o potencial máximo de desempenho que a aplicação poderia atingir com ajustes finos.

B. Trabalhos Futuros

Para mitigar as limitações identificadas e aprofundar a compreensão sobre o desempenho de microsserviços, propõem-se as seguintes direções para trabalhos futuros:

1) **Observabilidade Integrada (Locust + Prometheus/Grafana)**: Integrar o Locust com ferramentas de monitoramento como Prometheus [7] (para coleta de métricas de sistema e aplicação) e Grafana [8] (para visualização em dashboards). Isso permitirá a análise de correlação entre a queda na taxa de sucesso e o esgotamento de recursos de infraestrutura (CPU, Memória, I/O) em tempo real, fornecendo uma causa raiz mais precisa para os gargalos.

2) **Testes em Ambiente de Orquestração (Kubernetes)**: Realizar os mesmos testes de carga em um cluster Kubernetes [9], explorando as capacidades de escalonamento automático (Horizontal Pod Autoscaler

- HPA) e load balancing nativas. Isso permitiria avaliar a elasticidade da arquitetura e determinar se o sistema consegue se recuperar de picos de carga através da adição dinâmica de novas instâncias de microsserviços.

3) **Comparação com Outras Ferramentas de Carga**: Replicar a metodologia de teste utilizando outras ferramentas de carga, como JMeter [10] e K6 [11], para validar a consistência dos resultados e comparar a eficiência e a facilidade de uso do Locust no contexto de testes de microsserviços.

REFERENCES

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

[2] M. Fowler and J. Lewis, "Microservices," https://martinfowler.com/articles/microservices.html, Mar 2014, acessado em: 24 de out. de 2025.

[3] Spring Team, "Spring petclinic microservices," https://github.com/spring-petclinic/spring-petclinic-microservices, 2024, acessado em: 24 de out. de 2025.

[4] Locust.io, "Locust - an open source load testing tool," https://locust.io, 2024, acessado em: 24 de out. de 2025.

[5] M. Fowler, "Event sourcing," https://martinfowler.com/eaaDev/EventSourcing.html, Dez 2005, acessado em: 24 de out. de 2025.

[6] ——, "Cqrs," https://martinfowler.com/bliki/CQRS.html, Jul 2011, acessado em: 24 de out. de 2025.

[7] Prometheus.io, "Prometheus: From metrics to insight," https://prometheus.io, 2024, acessado em: 24 de out. de 2025.

[8] Grafana Labs, "Grafana: The open and composable observability platform," https://grafana.com, 2024, acessado em: 24 de out. de 2025.

[9] Kubernetes.io, "Kubernetes documentation," https://kubernetes.io/docs/home/, 2024, acessado em: 24 de out. de 2025.

[10] Apache JMeter, "Apache jmeter," https://jmeter.apache.org/, 2024, acessado em: 24 de out. de 2025.

[11] K6.io, "K6: Open source load testing for developers," https://k6.io, 2024, acessado em: 24 de out. de 2025.