Avaliação de Desempenho do Spring PetClinic (Microservices) com Locust

1st Robson de Moura Santos Júnior
Sistemas Distribuídos0
Universidade Federal do Piauí - UFPI
Santa Cruz do Piauí, Brasil
robson.junior@ufpi.edu.br

2nd Simão de Carvalho Morais
Sistemas Distribuídos ()
Universidade Federal do Piauí - UFPI
Picos, Brasil
simao.morais@ufpi.edu.br

3rd Pedro Tercio do Nascimento Vieira
Sistemas Distribuídos ()
Universidade Federal do Piauí - UFPI
Picos, Brasil
pedrotercio@ufpi.edu.br

***Abstract*—Este artigo tem como objetivo principal avaliar o desempenho e a escalabilidade da aplicação Spring PetClinic, especificamente em sua arquitetura baseada em microsserviços, utilizando a ferramenta de teste de carga Locust. A decomposição de sistemas monolíticos em serviços independentes, embora ofereça benefícios de manutenção e implantação contínua, também introduz desafios em relação à latência de comunicação e consumo de recursos.**

A metodologia empregada consistiu na execução de testes de carga em três cenários distintos, mantendo a configuração padrão do sistema: Carga Leve (50 usuários), Carga Moderada (100 usuários) e Carga Pico (200 usuários). Foi utilizado um mix de requisições que exercita os principais endpoints para simular o comportamento real do usuário. As métricas coletadas abrangem tempo de resposta (médio e máximo), taxa de requisições por segundo (RPS), taxa de erros e uso de recursos computacionais, como CPU e memória. Os resultados obtidos permitiram identificar o ponto em que o aumento da concorrência compromete a qualidade de serviço, revelando que a aplicação, em sua configuração original, demonstra uma rápida degradação da latência e perda de confiabilidade ao atingir níveis moderados e picos de carga.

***Index Terms*—Microsserviços, Avaliação de Desempenho, Teste de Carga, Spring PetClinic, Locust.**

I. Introdução

O crescimento de aplicações baseadas em microsserviços tem impulsionado a necessidade de técnicas eficientes para avaliação de desempenho. A decomposição de sistemas monolíticos em múltiplos serviços independentes oferece benefícios de escalabilidade, manutenção e implantação contínua. Entretanto, essa arquitetura também introduz desafios, como aumento da latência de comunicação e maior consumo de recursos computacionais.

Este trabalho tem como objetivo avaliar o desempenho da aplicação Spring PetClinic (versão baseada em microsserviços) sob diferentes níveis de carga, utilizando a ferramenta Locust para geração de requisições simultâneas. A partir dos testes, pretende-se observar como a aplicação se comporta em termos de tempo de resposta, taxa de requisições atendidas, erros e uso de recursos (CPU e memória).

Os experimentos foram realizados em três cenários de carga — leve, moderado e pico —, mantendo a configuração original do sistema. Os resultados servirão de base para identificar gargalos e discutir a escalabilidade da aplicação.

II. Descrição do sistema

O crescimento de aplicações baseadas em microsserviços tem impulsionado a necessidade de técnicas eficientes para avaliação de desempenho. A decomposição de sistemas monolíticos em múltiplos serviços independentes oferece benefícios de escalabilidade, manutenção e implantação contínua. Entretanto, essa arquitetura também introduz desafios, como aumento da latência de comunicação e maior consumo de recursos computacionais.

Este trabalho tem como objetivo avaliar o desempenho da aplicação Spring PetClinic (versão baseada em microsserviços) sob diferentes níveis de carga, utilizando a ferramenta Locust para geração de requisições simultâneas. A partir dos testes, pretende-se observar como a aplicação se comporta em termos de tempo de resposta, taxa de requisições atendidas, erros e uso de recursos (CPU e memória).

Os experimentos foram realizados em três cenários de carga — leve, moderado e pico —, mantendo a configuração original do sistema. Os resultados servirão de base para identificar gargalos e discutir a escalabilidade da aplicação.

III. Testes

Os testes de desempenho foram conduzidos com a ferramenta Apache Locust, utilizando um conjunto de tarefas que simulam o comportamento de usuários interagindo com os principais endpoints do sistema Spring PetClinic Microservices. As requisições foram distribuídas conforme ilustrado na Figura 1.

Foram realizadas duas etapas de testes: uma primeira etapa exploratória com uma única repetição por cenário para identificar o comportamento geral do sistema, e uma segunda etapa com 30 repetições por cenário para obter resultados estatisticamente robustos e avaliar a estabilidade do sistema ao longo de múltiplas execuções.

A Tabela I de especificações técnicas da máquina utilizada nos testes detalha componentes essenciais para contextualizar os resultados da avaliação de desempenho da Spring PetClinic em microsserviços com Locust. O modelo Acer Aspire 5 A515-45 R760, com processador AMD Ryzen 7 5700U (8

núcleos, 16 threads), 20 GB de RAM DDR4-3200 MHz e Windows 11 Home Versão 24H2, representa um setup de laptop consumidor, adequado para simulações leves, mas limitante para cargas altas devido a potenciais gargalos como sobrecarga térmica e overheads do SO.

TABLE I
ESPECIFICAÇÕES TÉCNICAS DA MÁQUINA UTILIZADA NOS TESTES

Parâmetro	**Descrição**
Modelo	Acer Aspire 5 A515-45 R760
Processador	AMD Ryzen 7 5700U (8 núcleos, 16 threads)
Memória RAM	20 GB DDR4 – 3200 MHz
Sistema Operacional	Windows 11 Home Single Language Versão 24H2

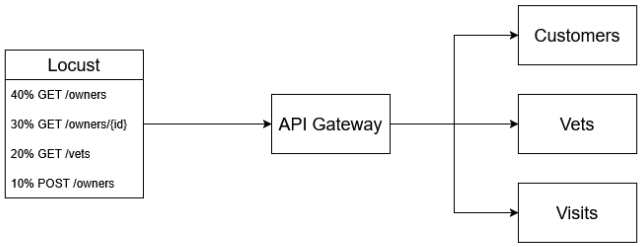

Fig. 1. Fluxo lógico de requisições de teste geradas pela ferramenta Locust para a aplicação Spring PetClinic Microservices.

A. Primeira Etapa: Testes Exploratórios

Na primeira etapa, foram realizados testes com uma única repetição para cada cenário (leve, moderado e pico), com duração de 11 minutos para os cenários leve e moderado, e 6 minutos para o cenário de pico. Estes testes iniciais permitiram identificar o comportamento geral do sistema e estabelecer uma linha de base para comparação.

A Figura 2 ilustra a evolução dos tempos de resposta médio e máximo nos três cenários testados nesta primeira etapa. Observa-se uma degradação significativa no desempenho à medida que a carga aumenta. No cenário leve, o tempo médio de resposta foi de apenas 50,41 ms, demonstrando excelente desempenho. No entanto, no cenário moderado, este valor aumentou para 827,94 ms (incremento de 1.543%), e no cenário de pico atingiu 2.352,88 ms (incremento de 4.567% em relação ao cenário leve). O tempo máximo de resposta também cresceu de forma expressiva, atingindo 10.045,92 ms no cenário de pico.

A Figura 3 apresenta o throughput alcançado em cada cenário na primeira etapa, medido em requisições por segundo. O sistema conseguiu processar mais requisições por segundo conforme aumentou o número de usuários, porém essa escalabilidade não foi linear. O throughput aumentou de 24,25 req/s no cenário leve para 35,03 req/s no cenário

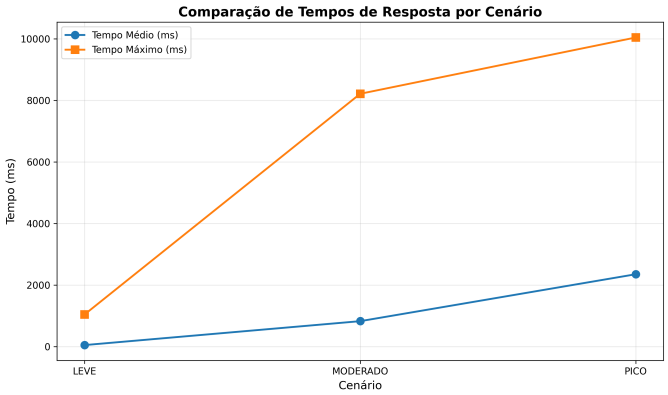

Fig. 2. Comparação de tempos de resposta entre cenários - Testes exploratórios (1 repetição)

moderado, representando um ganho de 44,5%. No cenário de pico, o throughput atingiu 45,24 req/s, um aumento adicional de apenas 29,1% em relação ao cenário moderado, indicando que o sistema se aproximava do seu limite de capacidade.

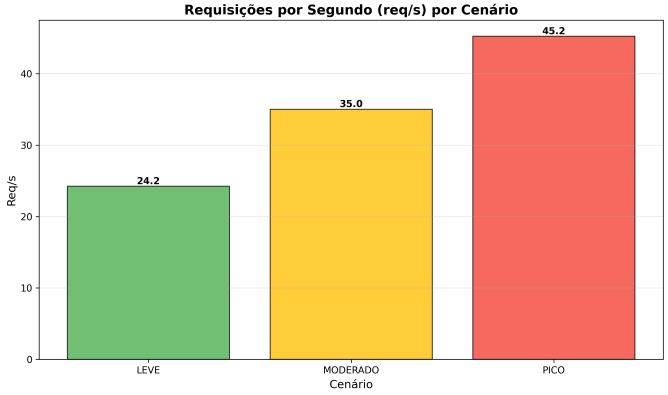

Fig. 3. Throughput (requisições por segundo) por cenário - Testes exploratórios (1 repetição)

A Figura 4 mostra a taxa de sucesso das requisições em cada cenário da primeira etapa. Os cenários leve e moderado apresentaram taxa de sucesso de 100%, indicando que o sistema conseguiu processar adequadamente todas as requisições sob essas condições. Entretanto, no cenário de pico, a taxa de sucesso caiu para 88,35%, com 1.897 falhas registradas de um total de 16.286 requisições. Esta degradação sugeriu que o sistema atingia seu limite de capacidade próximo a 200 usuários simultâneos.

A Figura 5 apresenta uma comparação entre o volume total de requisições processadas e o número de falhas em cada cenário da primeira etapa. É notável que, enquanto os cenários leve e moderado não apresentaram falhas, o cenário de pico registrou 1.897 falhas. As falhas foram concentradas principalmente no endpoint `GET /vets`, que apresentou 1.863 falhas (98,2% do total) com erro 405 (Method Not Allowed), indicando um gargalo específico neste serviço. Outras operações afetadas incluíram `GET /owners` com 17 falhas,

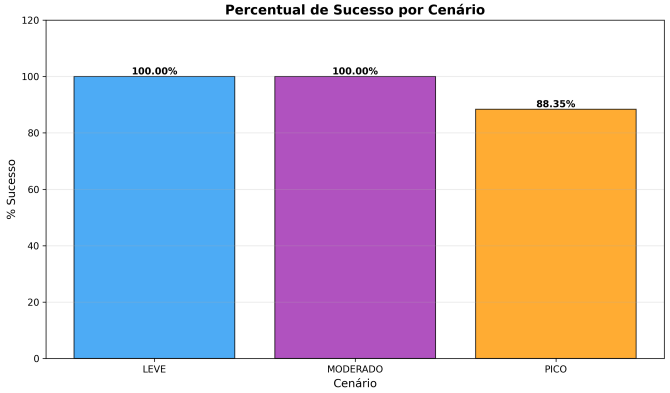

Fig. 4. Percentual de sucesso das requisições - Testes exploratórios (1 repetição)

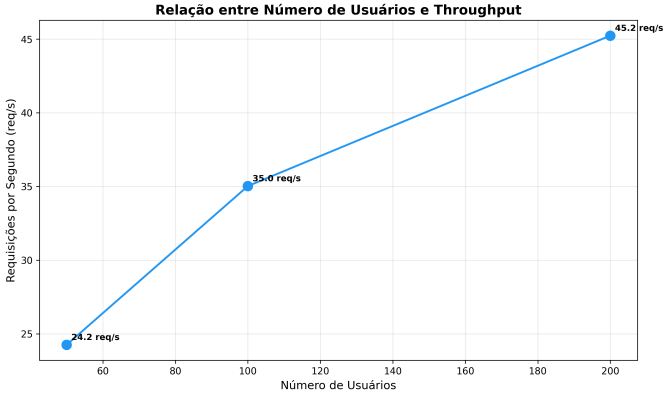

Fig. 6. Relação entre número de usuários e throughput - Testes exploratórios (1 repetição)

`GET /owners/{id}` com 11 falhas, e `POST /owners` com 6 falhas (erro 503).

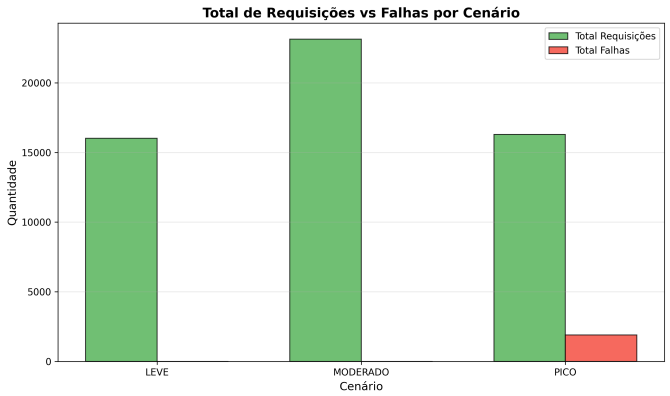

Fig. 5. Total de requisições versus falhas por cenário - Testes exploratórios (1 repetição)

A Figura 6 demonstra a relação entre o número de usuários simultâneos e o throughput do sistema na primeira etapa. A análise revelou que, embora o throughput aumentasse com mais usuários, o sistema não escalava linearmente. A eficiência por usuário diminuiu significativamente sob alta carga, passando de 0,485 req/s/usuário no cenário leve para 0,226 req/s/usuário no cenário de pico, uma redução de 53,4%. Isto evidenciou contenção de recursos e sugeriu a necessidade de otimizações arquiteturais.

B. Segunda Etapa: Testes com 30 Repetições

Após a análise dos testes exploratórios, foi realizada uma segunda etapa mais abrangente, aplicando 30 repetições para cada cenário. Este teste iniciou às 21:10 horas e se estendeu até as 11:10 do dia seguinte, totalizando 14 horas de execução contínua. O objetivo foi avaliar a estabilidade do sistema ao longo de múltiplas execuções e obter dados estatisticamente mais robustos.

Durante esta etapa extensiva, o consumo excessivo de memória foi identificado como o principal causador de uma alta taxa de erros que não havia sido aparente nos testes exploratórios iniciais. Os resultados desta segunda etapa contrastam drasticamente com a primeira, revelando problemas de estabilidade que apenas se manifestam sob uso prolongado.

1) Tempos de Resposta nas 30 Repetições: A Figura 7 apresenta os tempos de resposta médio e máximo calculados como média das 30 repetições. Diferentemente dos testes exploratórios, onde os tempos aumentavam com a carga (comportamento esperado), observa-se aqui uma tendência inversa: o tempo médio diminui de 788,56 ms no cenário leve para 337,41 ms no moderado, e para 256,13 ms no cenário de pico. Esta inversão contra-intuitiva é um indicador de que o sistema está falhando rapidamente (*fail-fast*) ao invés de processar requisições completas. Os tempos máximos, por outro lado, mantêm-se consistentemente elevados (aproximadamente 10 segundos em todos os cenários), sugerindo que o sistema atinge limites de timeout ou esgotamento de recursos de forma sistemática.

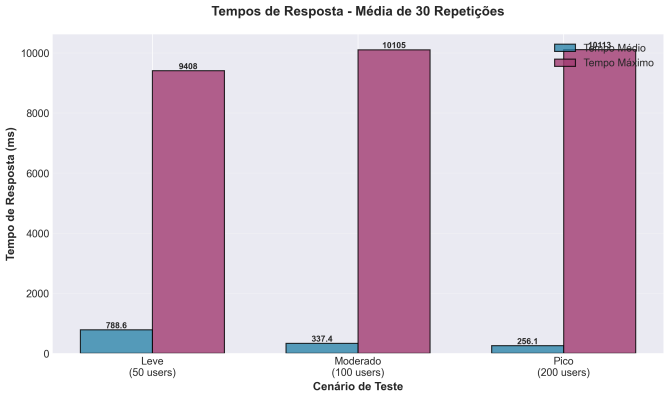

Fig. 7. Tempos de resposta médio e máximo - Média de 30 repetições

2) Throughput nas 30 Repetições: A Figura 8 mostra o throughput do sistema nas 30 repetições. Observa-se um aumento significativo: 17,90 req/s no cenário leve, 41,14 req/s no moderado (aumento de 129,9%), e 81,18 req/s no pico (dobrando em relação ao moderado). Este aumento expressivo

no throughput, que poderia ser interpretado como melhoria de desempenho, é na verdade um artefato das falhas sistemáticas: o sistema processa mais requisições por segundo porque está retornando erros rapidamente, sem executar o trabalho completo necessário. Este comportamento é característico de sistemas sob severa pressão de memória, onde a maioria das requisições falha antes de consumir recursos significativos.

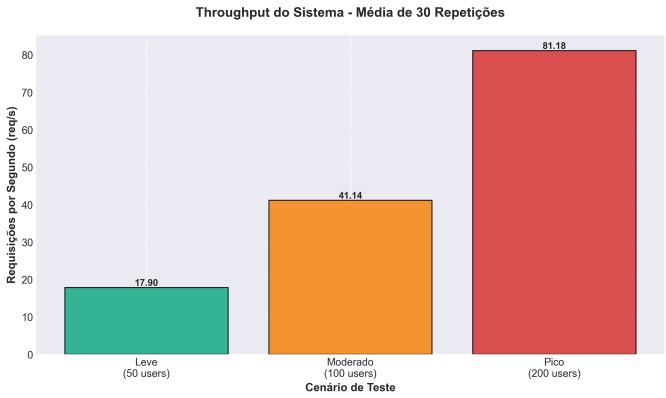

Fig. 8. Throughput do sistema - Média de 30 repetições

3) Taxa de Sucesso nas 30 Repetições: A Figura 9 revela o problema mais crítico identificado nesta segunda etapa: taxas de sucesso dramaticamente baixas em todos os cenários. No cenário leve, apenas 18,03% das requisições foram bem-sucedidas, indicando que mesmo sob carga reduzida, o sistema já apresenta falhas em mais de 80% das operações. Nos cenários moderado e pico, a situação é ainda mais grave, com taxas de sucesso de 1,47% e 0,86%, respectivamente. Estes valores indicam que o sistema é essencialmente não funcional em todos os níveis de carga testados, contrastando fortemente com os 100% de sucesso observados nos cenários leve e moderado dos testes exploratórios. Esta discrepância evidencia que problemas de memória se acumulam ao longo de múltiplas execuções, manifestando-se como falhas sistemáticas.

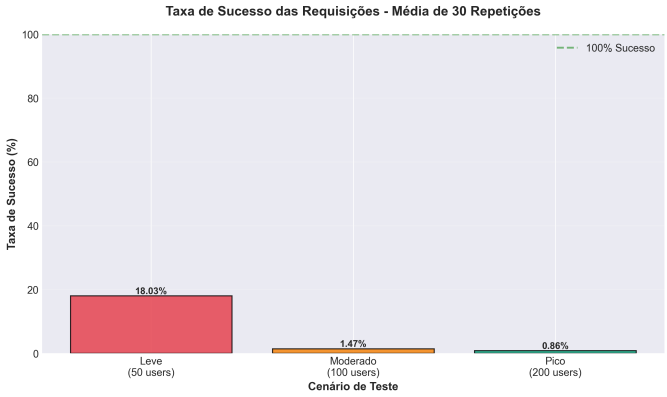

Fig. 9. Taxa de sucesso das requisições - Média de 30 repetições

4) Volume de Requisições e Falhas nas 30 Repetições: A Figura 10 compara o volume total de requisições processadas com o número de falhas em cada cenário. A visualização evidencia a magnitude do problema: no cenário leve, de 10.384 requisições em média, 8.746 resultaram em falha. Nos cenários moderado e pico, praticamente todo o volume de requisições resulta em falha, com 24.258 falhas de 24.619 requisições (moderado) e 24.061 falhas de 24.269 requisições (pico). A proximidade entre o volume total e o volume de falhas nestes cenários demonstra graficamente que menos de 2% das requisições são processadas com sucesso. Este padrão de falha generalizada, combinado com o diagnóstico de consumo excessivo de memória, sugere vazamento de memória (*memory leak*) ou dimensionamento inadequado dos recursos do sistema.

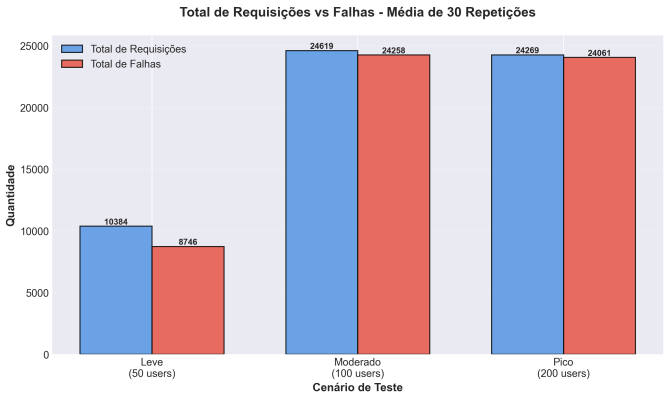

Fig. 10. Comparação entre requisições totais e falhas - Média de 30 repetições

5) Eficiência por Usuário nas 30 Repetições: A Figura 11 apresenta a eficiência do sistema medida como requisições processadas por segundo por usuário. Diferentemente do esperado em sistemas sobrecarregados, onde a eficiência diminui com o aumento de usuários, observa-se aqui uma eficiência relativamente estável: 0,358 req/s/usuário no cenário leve, aumentando ligeiramente para 0,411 req/s/usuário no moderado e mantendo-se em 0,406 req/s/usuário no pico. Esta estabilidade anômala reforça a interpretação de que o sistema não está realizando trabalho útil, mas sim processando falhas de forma consistente e rápida. Em um sistema saudável sob crescente carga, esperaríamos ver diminuição da eficiência devido à contenção de recursos compartilhados; a estabilidade observada indica que o sistema atingiu um estado de falha sistemática onde cada usuário adicional contribui para o volume de erros de forma proporcional.

6) Escalabilidade Real versus Ideal nas 30 Repetições: A Figura 12 compara o throughput real obtido nas 30 repetições com um throughput ideal calculado assumindo escalabilidade linear perfeita (linha tracejada verde). Em um cenário ideal, ao dobrar o número de usuários, o throughput deveria dobrar proporcionalmente. Os resultados mostram que o throughput real (linha vermelha) supera o ideal: 41,14 req/s no moderado versus 35,80 req/s esperado, e 81,18 req/s no pico versus 71,60 req/s esperado. Este aparente ”super-desempenho” é, paradoxalmente, um indicador de falha crítica. O sistema consegue processar mais requisições por segundo do que seria fisicamente possível em operação normal porque está

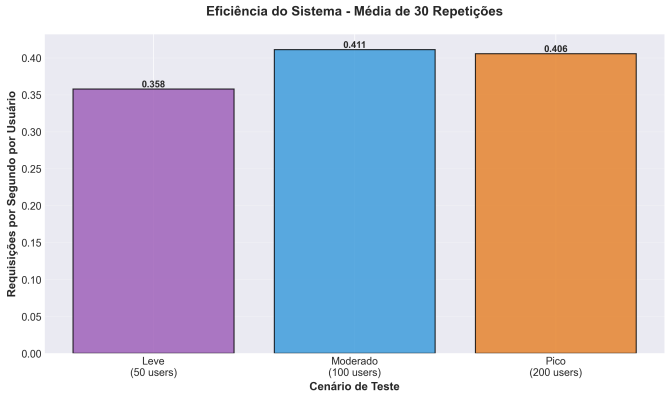

Fig. 11. Eficiência do sistema por usuário - Média de 30 repetições

retornando erros imediatamente, sem executar a lógica de negócio completa. Um sistema saudável tenderia a apresentar throughput inferior ao ideal linear sob alta carga devido à contenção de recursos; o fato de superar o ideal linear confirma que o sistema está em estado de falha sistemática.

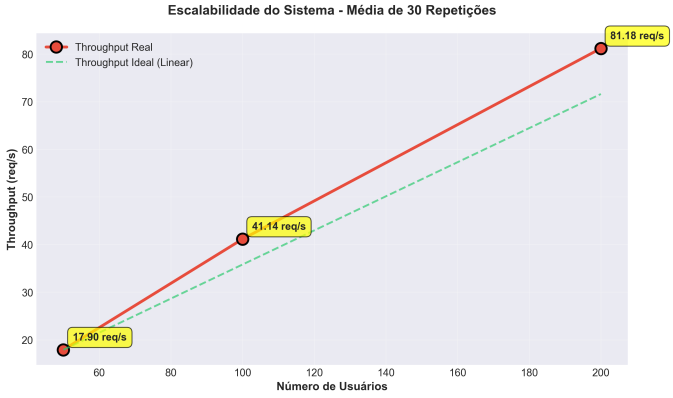

Fig. 12. Escalabilidade do sistema: real vs. ideal - Média de 30 repetições

IV. Resultados

Esta seção apresenta a análise comparativa dos resultados obtidos nas duas etapas de testes: os testes exploratórios iniciais (1 repetição por cenário) e os testes extensivos (30 repetições por cenário). A comparação revela diferenças significativas no comportamento do sistema, evidenciando problemas que apenas se manifestam sob uso prolongado e repetido.

A. Síntese dos Resultados - Testes Exploratórios

A Tabela II apresenta um resumo consolidado das principais métricas coletadas durante a primeira etapa de testes exploratórios. Observa-se uma progressão no número de usuários simultâneos (50, 100 e 200), com variações significativas nas métricas de desempenho.

Os resultados da primeira etapa evidenciaram que os cenários leve e moderado mantiveram taxa de sucesso de 100%, enquanto o cenário de pico apresentou degradação significativa, com 11,65% de falhas. O tempo médio de resposta

TABLE II
Síntese das Métricas dos Testes Exploratórios (1 Repetição)

Métrica	Leve	Moderado	Pico
Usuários Simultâneos	50	100	200
Duração do Teste	11 min	11 min	6 min
Total de Requisições	16.010	23.126	16.286
Total de Falhas	0	0	1.897
Taxa de Sucesso (%)	100,00	100,00	88,35
Tempo Médio (ms)	50,41	827,94	2.352,88
Tempo Máximo (ms)	1.046,05	8.216,05	10.045,92
Throughput (req/s)	24,25	35,03	45,24

aumentou em 46,7 vezes do cenário leve para o cenário de pico, indicando saturação dos recursos do sistema.

B. Síntese dos Resultados - 30 Repetições

A Tabela III apresenta os resultados da segunda etapa, calculados como média de 30 repetições. Os resultados contrastam drasticamente com os testes exploratórios, revelando problemas críticos de estabilidade que não foram aparentes na primeira etapa.

TABLE III
Resultados Médios dos Testes de Carga (30 Repetições)

Métrica	Leve	Moderado	Pico
Usuários Simultâneos	50	100	200
Duração Média do Teste	10 min	10 min	5 min
Total de Requisições	10.384	24.619	24.269
Total de Falhas	8.746	24.258	24.061
Taxa de Sucesso (%)	18,03	1,47	0,86
Tempo Médio (ms)	788,56	337,41	256,13
Tempo Máximo (ms)	9.407,83	10.105,49	10.113,20
Throughput (req/s)	17,90	41,14	81,18

Os resultados da segunda etapa evidenciam um comportamento crítico do sistema: taxas de sucesso extremamente baixas em todos os cenários testados. Enquanto o cenário leve apresentou taxa de sucesso de apenas 18,03%, os cenários moderado e pico demonstraram taxas ainda mais críticas de 1,47% e 0,86%, respectivamente. O consumo excessivo de memória foi identificado como o principal causador da alta taxa de erros observada.

C. Análise Comparativa entre as Duas Etapas

A comparação entre os resultados das duas etapas de testes revela discrepâncias significativas que evidenciam problemas estruturais do sistema:

1) Taxa de Sucesso: Enquanto os testes exploratórios mostraram taxas de sucesso de 100% nos cenários leve e moderado, os testes com 30 repetições revelaram taxas de sucesso de apenas 18,03% e 1,47%, respectivamente. Esta diferença dramática sugere que:

- Os testes exploratórios capturaram o sistema em condições favoráveis (recém-iniciado, caches aquecidos)
- A execução contínua de 30 repetições expõe problemas de acúmulo de memória que não aparecem em testes isolados

- O vazamento de memória é cumulativo, manifestando-se apenas após múltiplas execuções sequenciais

2) Tempos de Resposta: Os testes exploratórios apresentaram tempos médios de resposta crescentes com a carga (50,41 ms → 827,94 ms → 2.352,88 ms), comportamento esperado em sistemas sob crescente pressão. Em contraste, os testes com 30 repetições mostraram tempos decrescentes (788,56 ms → 337,41 ms → 256,13 ms), um padrão contra-intuitivo característico de sistemas que falham rapidamente (*fail-fast*), onde requisições que não conseguem alocar recursos necessários falham imediatamente, contribuindo para tempos de resposta menores.

3) Throughput: Nos testes exploratórios, o throughput aumentou de forma moderada (24,25 → 35,03 → 45,24 req/s). Nos testes com 30 repetições, observou-se aumento mais acentuado (17,90 → 41,14 → 81,18 req/s), mas este aparente ganho de desempenho é enganoso: o sistema está processando mais requisições por segundo porque a maioria delas falha prematuramente, sem executar o processamento completo necessário.

D. Análise Detalhada dos Testes com 30 Repetições

1) Problema Crítico de Memória: O consumo excessivo de memória identificado como causa principal das falhas indica que o sistema não está adequadamente dimensionado para processar as cargas propostas. Possíveis causas incluem:

- Vazamento de memória (*memory leaks*) nos serviços
- Configuração inadequada do heap da JVM
- Acúmulo de objetos não coletados pelo *Garbage Collector*
- Pool de conexões de banco de dados mal dimensionado
- Falta de limites de memória adequados nos contêineres Docker

2) Inversão de Comportamento das Métricas: Os resultados da segunda etapa apresentam padrões contra-intuitivos que são característicos de sistemas em falha:

- **Tempos de resposta diminuem com aumento da carga**: Requisições falham rapidamente antes de processar completamente
- **Throughput aumenta além do esperado linearmente**: Sistema processa erros mais rapidamente do que requisições bem-sucedidas
- **Eficiência por usuário mantém-se estável**: Taxa de processamento de falhas é relativamente constante

A Figura 11 demonstra que a eficiência por usuário mantém-se relativamente estável (0,358 → 0,411 → 0,406 req/s/usuário) nos testes com 30 repetições, em contraste com a queda acentuada observada nos testes exploratórios. Este comportamento confirma que o sistema está processando falhas rapidamente, mantendo uma taxa de processamento aparentemente eficiente, mas sem executar o trabalho útil requerido.

3) Sistema Não Funcional em Condições Reais: Com taxas de sucesso de 18,03% (leve), 1,47% (moderado) e 0,86% (pico), o sistema é efetivamente não funcional em todos os cenários testados na segunda etapa. Mesmo sob a menor carga testada (50 usuários), mais de 80% das requisições falham, indicando que o sistema não está apto para uso em produção sem intervenções significativas.

E. Escalabilidade Real vs. Ideal

A Figura 12 compara o throughput real obtido nos testes com 30 repetições com um throughput ideal calculado assumindo escalabilidade linear perfeita. Os resultados mostram throughput de 41,14 req/s (moderado) e 81,18 req/s (pico), valores superiores ao ideal linear (35,80 e 71,60 req/s, respectivamente). Este resultado aparentemente positivo é, na verdade, um indicador de falha sistemática: o sistema está processando requisições mais rapidamente porque a maioria delas falha prematuramente.

F. Recomendações

Com base nos resultados observados, especialmente na segunda etapa de testes, são necessárias as seguintes intervenções para tornar o sistema funcional:

1) **Análise de memória**: Realizar *profiling* de memória usando ferramentas como VisualVM ou JProfiler para identificar vazamentos e objetos não liberados
2) **Configuração da JVM**: Ajustar parâmetros de heap (-Xmx, -Xms), configurar *Garbage Collector* adequado, e definir limites de memória nos contêineres
3) **Otimização de banco de dados**: Revisar pool de conexões, implementar cache de consultas frequentes, e otimizar queries problemáticas
4) **Escalabilidade horizontal**: Implementar múltiplas instâncias dos microsserviços com balanceamento de carga
5) **Limites de taxa**: Implementar *rate limiting* e *circuit breakers* para prevenir cascata de falhas
6) **Monitoramento**: Estabelecer monitoramento contínuo de métricas de memória, CPU, e latência em produção

V. Limitações

Uma limitação chave dos testes reside na população limitada do banco de dados, com apenas 10 owners, o que não simula cenários reais de produção e pode subestimar impactos em endpoints de leitura como GET /owners. Além disso, o ambiente local em uma única máquina Windows, sem distribuição como Kubernetes em cluster, mascara problemas de rede e escalabilidade horizontal típicos de deployments reais. A configuração padrão sem otimizações, como caching ou ajustes na JVM, restringe a análise a um baseline não representativo, alinhando-se às especificações do TRABALHO 07, mas limitando insights sobre setups produtivos.

Outra restrição é a duração curta dos testes (5-10 minutos por cenário), que não captura degradação cumulativa como memory leaks observados nas 30 repetições, sugerindo necessidade de execuções mais longas. A ausência de métricas avançadas, como CPU por serviço ou latência de banco, impede identificação precisa de gargalos nos microsserviços. O mix de requisições simplificado exclui operações como PUT

ou DELETE, reduzindo a fidelidade à simulação de usuários reais, enquanto o hardware consumidor introduz overheads não presentes em servidores Linux dedicados.

Ademais, a falta de análise estatística robusta, como desvios padrão ou intervalos de confiança nas médias das repetições, diminui a confiabilidade dos resultados. Fatores externos, como rede ou concorrência de processos, não foram considerados, afetando a reprodutibilidade. Finalmente, sem comparação com a versão monolítica, os trade-offs da arquitetura microsserviços permanecem subexplorados, destacando que os testes servem como base inicial, mas demandam expansões para validade em contextos amplos.

VI. Conclusão

Os resultados obtidos nesta avaliacao de desempenho da aplicacao Spring PetClinic em arquitetura de microsservicos, utilizando a ferramenta Locust, indicam limitacoes expressivas na escalabilidade e estabilidade do sistema em sua configuracao padrao. Nos testes exploratorios com uma unica repeticao por cenario, observou-se desempenho satisfatorio sob cargas leves de cinquenta usuarios e moderadas de cem usuarios, com taxas de sucesso de cem por cento, tempos medios de resposta de cinquenta ponto quatro um milissegundos e oitocentos e vinte e sete ponto nove quatro milissegundos, respectivamente, e throughput crescente de vinte e quatro ponto dois cinco requisicoes por segundo para trinta e cinco ponto zero tres requisicoes por segundo.

Contudo, no cenario de pico com duzentos usuarios, registrou-se degradacao significativa, com taxa de sucesso reduzida a oitenta e oito ponto tres cinco por cento, tempo medio de resposta atingindo dois mil trezentos e cinquenta e dois ponto oito oito milissegundos e throughput limitado a quarenta e cinco ponto dois quatro requisicoes por segundo, evidenciando saturacao de recursos e falhas concentradas no endpoint GET barra vets. Em contraste, a segunda etapa com trinta repeticoes por cenario revelou problemas cronicos de consumo excessivo de memoria, resultando em taxas de sucesso drasticamente baixas de dezoito ponto zero tres por cento no cenario leve, um ponto quatro sete por cento no moderado e zero ponto oito seis por cento no pico.

Curiosamente, os tempos medios de resposta diminuíram com o aumento da carga, de setecentos e oitenta e oito ponto cinco seis milissegundos para trezentos e trinta e sete ponto quatro um milissegundos e duzentos e cinquenta e seis ponto um tres milissegundos, enquanto o throughput elevou-se de dezessete ponto nove zero requisicoes por segundo para oitenta e um ponto um oito, refletindo um comportamento de falha rapida onde a maioria das requisicoes falha sem completar a logica de negocio. Essa discrepancia entre as etapas sugere que falhas cumulativas, como vazamentos de memoria, manifestam-se apenas sob execucao prolongada e repetida, tornando o sistema inoperante para cargas realistas sem otimizacoes.

Os achados apontam para a necessidade de intervencoes como ajustes na configuracao da JVM, limites de memoria em conteineres Docker, otimizacao de pools de conexao com o banco de dados e escalabilidade horizontal para mitigar gargalos. Em sintese, embora a arquitetura de microsservicos ofereca vantagens teoricas de independencia e manutencao, a PetClinic em sua configuracao original exibe vulnerabilidades que comprometem sua viabilidade em ambientes de producao, servindo como base para futuras melhorias em gerenciamento de recursos e resiliencia sob estresse.

References

[1] Pivotal Software, Inc., “Spring PetClinic Microservices,” GitHub repository. Available: https://github.com/spring-petclinic/spring-petclinic-microservices. Accessed: Oct. 22, 2025.

[2] Locust.io, “Locust - A modern load testing framework,” Available: https://locust.io. Accessed: Oct. 22, 2025.

[3] VMware, Inc., “Spring Boot Actuator Reference Documentation,” Available: https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html. Accessed: Oct. 22, 2025.

[4] Docker, Inc., “Docker Documentation,” Available: https://docs.docker.com. Accessed: Oct. 22, 2025.

[5] Prometheus Authors, “Prometheus Monitoring System,” Available: https://prometheus.io. Accessed: Oct. 22, 2025.