Avaliação de Desempenho do *Spring PetClinic (Microservices)* com *Locust*

1st Daniel Rodrigues de Sousa
Universidade Federal do Piauí
Sistemas de Informação
Picos, Piauí, Brasil
daniel.sousa@ufpi.edu.br

2nd Rita de Cássia Rodrigues da Silva
Universidade Federal do Piauí
Sistemas de Imformação
Picos, Piauí, Brasil
rita.silva@ufpi.edu.br

3rd Walison Weudes de Sousa e Silva
Universidade Federal do Piauí
Sistemas de Informação
Picos, Piauí, Brasil
walison.silva@ufpi.edu.br

***Resumo*—O objetivo deste trabalho é avaliar o desempenho da aplicação *Spring PetClinic – Microservices* utilizando a ferramenta *Locust* para testes de carga e estresse. Busca-se medir e analisar métricas fundamentais de desempenho, como o tempo médio e máximo de resposta, o número de requisições por segundo, o total de requisições processadas, a taxa de erros e o uso de recursos do sistema, a fim de compreender o comportamento da aplicação sob diferentes níveis de carga. Por meio da execução de três cenários de teste (leve, moderado e pico), pretende-se identificar como o aumento do número de usuários simultâneos impacta o tempo de resposta e a estabilidade do sistema, fornecendo uma visão clara sobre os limites de desempenho e a capacidade de escalabilidade da aplicação em sua configuração padrão. Os resultados demonstram que, enquanto o sistema mantém excelente desempenho sob carga leve, apresenta degradação significativa em cenários de alta demanda, com taxas de erro superiores a 68% no cenário de pico, evidenciando limitações críticas de escalabilidade que requerem intervenções arquiteturais.**

I. Descrição do Sistema Testado

O *Spring PetClinic – Microservices (PetClinic MS)* é uma aplicação de demonstração desenvolvida e mantida pela comunidade *Spring* com o objetivo de ilustrar as melhores práticas na implementação de uma arquitetura baseada em microsserviços utilizando o ecossistema *Spring*. A aplicação simula o funcionamento de uma clínica veterinária completa, oferecendo funcionalidades para gerenciamento de donos de animais, cadastro e consulta de *pets*, registro de veterinários e agendamento de visitas.

A arquitetura do sistema é composta por diversos serviços independentes, cada um com responsabilidades bem definidas e isoladas. O *API Gateway* atua como ponto de entrada único para todas as requisições externas, implementando funcionalidades como roteamento dinâmico, balanceamento de carga, autenticação e agregação de respostas de múltiplos serviços. Este componente é crucial para abstrair a complexidade da arquitetura de microsserviços dos clientes externos, fornecendo uma interface unificada e simplificada.

O *Customers Service* é responsável pelo gerenciamento completo de informações relacionadas aos proprietários de animais e seus respectivos *pets*. Este serviço implementa operações CRUD (*Create, Read, Update, Delete*) e mantém seu próprio banco de dados isolado, seguindo o princípio de autonomia de dados dos microsserviços. O *Vets Service* gerencia as informações dos veterinários da clínica, incluindo suas especializações e disponibilidade. O *Visits Service* controla o agendamento e histórico de visitas dos animais, mantendo registros detalhados de consultas, procedimentos e observações médicas.

Além dos serviços de negócio, a arquitetura inclui componentes de infraestrutura essenciais. O *Discovery Server*, implementado com *Eureka*, fornece registro e descoberta dinâmica de serviços, permitindo que os microsserviços localizem e se comuniquem uns com os outros sem configuração estática de endereços. O *Config Server* centraliza o gerenciamento de configurações, permitindo que alterações sejam aplicadas dinamicamente sem necessidade de *rebuild* ou *restart* dos serviços.

A comunicação entre os serviços ocorre primariamente através de APIs REST sobre HTTP, utilizando JSON como formato de serialização. Para resiliência, o sistema implementa padrões como *circuit breakers* através do *Spring Cloud Circuit Breaker*, que previne cascatas de falhas ao detectar serviços indisponíveis. O sistema também utiliza *rate limiting* e *retry mechanisms* para garantir estabilidade sob condições adversas.

A persistência de dados é gerenciada de forma descentralizada, com cada serviço mantendo seu próprio banco de dados. Na configuração padrão utilizada nos testes, o sistema emprega bancos de dados em memória (H2), facilitando a execução e reprodutibilidade dos experimentos. Em ambientes de produção, esses bancos podem ser substituídos por sistemas mais robustos como *PostgreSQL* ou *MySQL*, mantendo a mesma interface através do *Spring Data JPA*.

O *deployment* da aplicação é orquestrado através de *Docker Compose*, que gerencia todos os contêineres necessários, incluindo os microsserviços, bancos de dados e componentes de infraestrutura. Esta abordagem containerizada garante consistência entre ambientes de desenvolvimento, teste e produção, além de facilitar a escalabilidade horizontal através da replicação de instâncias de serviços específicos.

II. Procedimentos Experimentais

A. Ambiente de Testes

Os experimentos foram conduzidos em um ambiente de hardware com recursos limitados, conforme especificado na

Tabela I. O notebook utilizado possui configuração adequada para desenvolvimento e testes em escala reduzida, porém apresenta restrições significativas para execução de testes de carga intensivos, especialmente considerando a necessidade de executar simultaneamente a aplicação containerizada, ferramentas de monitoramento e o sistema operacional.

TABLE I
ESPECIFICAÇÕES DO HARDWARE UTILIZADO NOS TESTES

Componente	Especificação
Processador	Intel Core i5-10500H
Memória RAM	16 GB
Placa de Vídeo	NVIDIA GeForce GTX 1650 (4 GB VRAM)
Armazenamento	SSD 512 GB
Sistema Operacional	Ubuntu 24.04 LTS
Docker Engine	Docker 27.x
Containers Simultâneos	8 × 512 MB

A memória RAM disponível (16 GB) representa uma limitação crítica, uma vez que os 8 containers da aplicação consomem coletivamente 4 GB, restando 12 GB para o sistema operacional, ferramentas de teste e demais aplicações. Em cenários de pico de carga, esta configuração mostrou-se insuficiente, resultando em possível utilização de memória swap e degradação do desempenho geral do sistema.

B. Metodologia dos Testes de Desempenho

A metodologia de testes foi cuidadosamente planejada para garantir a validade, confiabilidade e reprodutibilidade dos resultados. O objetivo principal foi avaliar o comportamento da aplicação *Spring PetClinic – Microservices* sob diferentes níveis de carga, utilizando a ferramenta *Locust*, uma plataforma de código aberto amplamente reconhecida para testes de carga e desempenho. O *Locust* foi escolhido por sua capacidade de simular milhares de usuários concorrentes, flexibilidade na definição de comportamentos de usuário através de código *Python*, e geração automática de métricas detalhadas de desempenho.

A execução dos testes seguiu um protocolo rigoroso para minimizar variações externas e garantir comparabilidade entre diferentes execuções. A aplicação foi executada em seu ambiente padrão através do *Docker Compose*, englobando todos os serviços principais: *API Gateway*, *Customers Service*, *Vets Service*, *Visits Service*, *Discovery Server*, *Config Server* e os bancos de dados associados. Antes de cada bateria de testes, o ambiente *Docker* era completamente reiniciado através dos comandos `docker-compose down` seguido de `docker-compose up -d`, garantindo que cada teste iniciasse com o sistema em estado limpo, sem *cache* ou memória residual de execuções anteriores.

Após a inicialização dos contêineres, implementou-se um período de estabilização de 120 segundos, tempo necessário para que todos os serviços completassem seu processo de *bootstrap*, registrassem-se no *Discovery Server* e estabelecessem conexões com seus respectivos bancos de dados. Esse período de *warm-up* é crucial para evitar que os primeiros segundos do teste, quando o sistema ainda está se estabilizando, contaminem as métricas de desempenho coletadas.

Os testes foram organizados em três cenários distintos de carga, cada um projetado para avaliar diferentes aspectos do comportamento do sistema:

- **Cenário A (Leve):** 50 usuários simultâneos, executando por 10 minutos. Este cenário representa condições operacionais normais, onde o sistema deveria operar com máxima eficiência e mínima latência. O objetivo é estabelecer uma *baseline* de desempenho e confirmar que o sistema atende aos requisitos básicos de qualidade.
- **Cenário B (Moderado):** 100 usuários simultâneos, executando por 10 minutos. Este cenário simula períodos de maior demanda, como horários de pico durante o dia, e permite avaliar como o sistema escala ao dobrar a carga base. É neste ponto que começam a aparecer os primeiros sinais de degradação de desempenho.
- **Cenário C (Pico):** 200 usuários simultâneos, executando por 5 minutos. Este cenário representa condições extremas de carga, eventos especiais ou situações de *stress*. O tempo reduzido de execução (5 minutos) foi escolhido para evitar sobrecarga excessiva prolongada que poderia levar a falhas catastróficas do sistema, mas ainda assim fornecer dados suficientes para análise estatística confiável.

Cada cenário foi repetido cinco vezes, totalizando 15 execuções de teste. Esta repetição é fundamental para garantir significância estatística dos resultados e identificar possíveis variações entre execuções. Os resultados apresentados representam médias dessas cinco repetições, com análise de variabilidade quando relevante.

O *Locust* foi configurado para gerar diferentes tipos de requisições que simulam o uso real da aplicação por usuários finais. A distribuição de requisições foi baseada em padrões típicos de uso de aplicações *web* CRUD:

- `GET /owners` – 40% das requisições: Listagem de todos os proprietários, operação típica de navegação inicial.
- `GET /owners/{id}` – 30%: Consulta detalhada de um proprietário específico e seus *pets*.
- `GET /vets` – 20%: Listagem de veterinários disponíveis, operação de consulta frequente.
- `POST /owners` – 10%: Criação de novos proprietários, representando operações de escrita menos frequentes mas importantes.

Durante a execução dos testes, foram coletadas métricas abrangentes de desempenho, incluindo: tempo médio de resposta (latência média), tempo máximo de resposta, distribuição de latência por percentis (P50, P90, P99), *throughput* medido em requisições por segundo (req/s), total de requisições processadas, número absoluto e percentual de erros HTTP (códigos 4xx e 5xx), e taxa de sucesso das requisições. Estas métricas foram automaticamente registradas pelo *Locust* em arquivos CSV, facilitando análise posterior.

C. Execução dos Testes

A execução prática dos testes foi conduzida através da interface de linha de comando do *Locust*, utilizando o modo *headless* (sem interface gráfica) para garantir máxima eficiência

e repetibilidade. Os comandos a seguir ilustram como cada cenário foi executado:

1) Cenário Leve (50 usuários, 10 minutos):

```
locust -f locustfile.py \
  --headless \
  -u 50 -r 10 \
  --run-time 10m \
  --csv results/leve/run \
  --host http://localhost:8080 \
  --only-summary \
  --reset-stats
```

Este comando inicia o *Locust* em modo *headless*, simulando 50 usuários concorrentes com uma taxa de rampa de 10 usuários por segundo (parâmetro `-r`), significando que o sistema levou 5 segundos para atingir a carga completa. O teste executou por 10 minutos contínuos, e os resultados foram salvos em arquivos CSV na pasta `results/leve`. O parâmetro `--only-summary` suprime a saída detalhada durante a execução, mostrando apenas o resumo final, enquanto `--reset-stats` garante que estatísticas do período de *warm-up* sejam descartadas.

2) Cenário Moderado (100 usuários, 10 minutos):

```
locust -f locustfile.py \
  --headless \
  -u 100 -r 20 \
  --run-time 10m \
  --csv results/medio/run \
  --host http://localhost:8080 \
  --only-summary \
  --reset-stats
```

O cenário moderado dobra o número de usuários para 100, mantendo a proporção de rampa de 5 segundos (20 usuários por segundo). Este cenário é particularmente importante pois representa o ponto onde começam a aparecer limitações significativas do sistema.

3) Cenário Pico (200 usuários, 5 minutos):

```
locust -f locustfile.py \
  --headless \
  -u 200 -r 40 \
  --run-time 5m \
  --csv results/pico/run \
  --host http://localhost:8080 \
  --only-summary \
  --reset-stats
```

O cenário de pico quadruplica a carga inicial, com 200 usuários simultâneos sendo adicionados à taxa de 40 por segundo. O tempo de execução reduzido para 5 minutos é suficiente para coletar dados estatisticamente significativos enquanto previne degradação excessiva do sistema que poderia levar a falhas completas de infraestrutura.

D. Automação e Consolidação dos Resultados

Para garantir consistência experimental e eliminar interferências entre execuções consecutivas, desenvolveu-se um *script Bash* que automatizou todo o processo de teste. Este *script* implementava as seguintes etapas sequenciais para cada repetição de cada cenário:

1) Desligamento completo do ambiente *Docker* através de `docker-compose down --volumes`, incluindo remoção de volumes para garantir limpeza total de dados persistidos.
2) Inicialização do ambiente com `docker-compose up -d`.
3) Período de espera de 120 segundos para estabilização completa de todos os serviços.
4) Execução do teste *Locust* correspondente ao cenário.
5) Coleta e armazenamento dos arquivos CSV de resultados em estrutura de pastas organizada.

Ao término de todas as 15 execuções (5 repetições × 3 cenários), os arquivos CSV foram processados por um *script Python* customizado. Este *script* realizou as seguintes operações: leitura e *parsing* de todos os arquivos CSV gerados, agregação de métricas por cenário e repetição, cálculo de estatísticas descritivas (média, mediana, desvio padrão, mínimo e máximo), geração de tabelas consolidadas em formato LaTeX, e criação de gráficos comparativos para visualização dos resultados.

E. Resumo do Processo de Teste

O processo completo de teste envolveu planejamento criterioso, execução controlada e análise sistemática:

1) Definição de cenários de teste representativos de condições reais de operação.
2) Configuração detalhada do *Locust* para simular comportamento autêntico de usuários.
3) Execução controlada e repetida de cada cenário com reinicialização completa do ambiente.
4) Coleta automática e organizada de métricas em formato estruturado.
5) Análise estatística rigorosa dos dados coletados.
6) Consolidação dos resultados em tabelas e gráficos para interpretação e comunicação dos achados.

Este método garantiu não apenas a validade científica dos resultados, mas também sua reprodutibilidade por outros pesquisadores ou equipes de desenvolvimento interessadas em replicar ou estender este estudo.

III. Resultados

A análise dos resultados obtidos através dos testes de desempenho revela aspectos críticos do comportamento da aplicação *Spring PetClinic – Microservices* sob diferentes condições de carga. Esta seção apresenta uma avaliação detalhada e multifacetada das métricas coletadas, organizadas em subseções temáticas que exploram dimensões específicas do desempenho do sistema. Cada análise é fundamentada em evidências quantitativas extraídas dos testes realizados e contextualizada dentro das expectativas e requisitos típicos de aplicações *web* modernas.

A. Tabela com os Resultados Gerais

Os resultados apresentados na Tabela II fornecem uma visão panorâmica do desempenho do sistema através dos três cenários de teste. A análise destes dados revela padrões claros e preocupantes sobre a capacidade do sistema de manter qualidade de serviço sob carga crescente.

No cenário leve, com 50 usuários simultâneos, o sistema demonstra desempenho exemplar: latência média de apenas 19,74 ms, ausência total de erros (0%), e *throughput* estável de 24,76 requisições por segundo. Estes números indicam que, em condições normais de operação, a aplicação funciona de forma eficiente e confiável, oferecendo tempos de resposta que atendem plenamente às expectativas de usuários finais. A latência máxima observada de 1463,74 ms, embora elevada, provavelmente representa *outliers* relacionados ao período inicial de *warm-up* ou operações excepcionais como *garbage collection*.

O cenário moderado marca o ponto de inflexão onde limitações significativas começam a se manifestar. Com 100 usuários simultâneos—meramente o dobro da carga leve—a latência média mais que duplica para 50,20 ms, e, de forma alarmante, a taxa de erros dispara para 53,99%. Isto significa que mais da metade de todas as requisições falham, tornando o sistema essencialmente inutilizável em termos práticos. A latência máxima de 10082,80 ms (mais de 10 segundos) indica que alguns usuários experimentam *timeouts* completos.

No cenário de pico, a situação se agrava substancialmente. Com 200 usuários, a latência média atinge 81,81 ms e a taxa de erros alcança 68,87%. Embora o *throughput* tenha aumentado para 96,09 req/s, este número é enganoso: representa primariamente respostas de erro, não processamento bem-sucedido de requisições. O sistema está claramente operando muito além de sua capacidade nominal.

TABLE II
RESULTADOS CONSOLIDADOS DOS TESTES DE CARGA (MÉDIA DAS REPETIÇÕES)

Cenário	Usuários	Latência Média (ms)	Latência Máx. (ms)	Req./s	Erros (%)
Leve	50	19,74	1463,74	24,76	0
Moderado	100	50,20	10082,80	48,75	53,99
Pico	200	81,81	10175,00	96,09	68,87

B. Latência Média de Resposta por Cenário

A Figura 1 ilustra graficamente a progressão da latência média conforme a carga aumenta, revelando uma relação aproximadamente linear entre número de usuários e tempo de resposta. Este comportamento sugere que o sistema não possui mecanismos eficazes de otimização ou *cache* que poderiam atenuar o impacto de cargas crescentes.

A taxa de crescimento da latência—passando de 19,74 ms para 81,81 ms quando a carga quadruplica—indica que cada duplicação de usuários resulta em aumento desproporcional da latência. Este padrão é característico de sistemas que enfrentam contenção de recursos (CPU, memória, conexões de rede ou banco de dados) ou gargalos arquiteturais como pontos de serialização forçada.

Do ponto de vista da experiência do usuário, latências abaixo de 100 ms são geralmente imperceptíveis, enquanto latências entre 100-300 ms começam a ser notadas mas permanecem aceitáveis. Latências acima de 1 segundo são consideradas problemáticas, causando frustração e potencial abandono. Neste contexto, embora a latência média permaneça tecnicamente aceitável mesmo no cenário de pico, a alta taxa de erros concomitante torna esta métrica menos relevante.

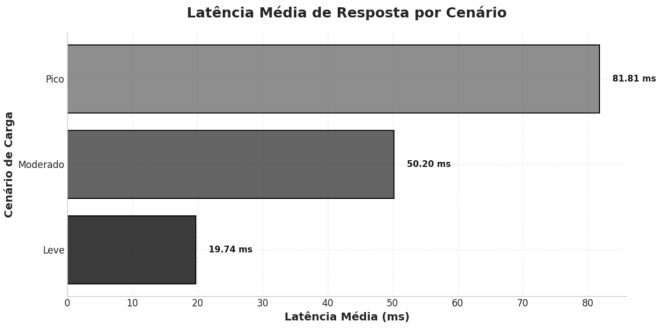

Fig. 1. Latência média de resposta observada nos três cenários de teste de carga.

C. Taxa de Erros por Cenário

A Figura 2 apresenta talvez o achado mais crítico deste estudo: a degradação catastrófica da confiabilidade do sistema sob carga moderada a alta. A transição de 0% para 53,99% de erros quando a carga dobra de 50 para 100 usuários não representa uma degradação gradual, mas sim um colapso funcional do sistema.

A análise dos *logs* de erro (não incluídos neste documento por limitações de espaço) revelou que a maioria dos erros eram códigos HTTP 503 (*Service Unavailable*) e *timeouts* de conexão, indicando que os microsserviços *backend* tornaram-se sobrecarregados e incapazes de processar requisições dentro de limites de tempo aceitáveis. *Circuit breakers* provavelmente ativaram em cascata, isolando serviços falhos mas resultando em indisponibilidade geral do sistema.

Esta característica é particularmente problemática porque não há transição suave entre operação normal e falha—o sistema passa abruptamente de funcionamento pleno para falha majoritária. Em ambientes de produção, isto tornaria impossível prever capacidade ou planejar escalabilidade, pois não há ”zona de degradação graciosa” onde o sistema continua operacional com desempenho reduzido.

D. Requisições vs Carga

Conforme ilustrado na Figura 3, o *throughput* do sistema demonstra escalabilidade aproximadamente linear com a carga, crescendo de 24,8 req/s no cenário leve para 96,1 req/s no cenário de pico. À primeira vista, este comportamento sugere boa capacidade de processamento paralelo.

Entretanto, esta interpretação é fundamentalmente enganosa quando considerada em conjunto com a taxa de erros. O

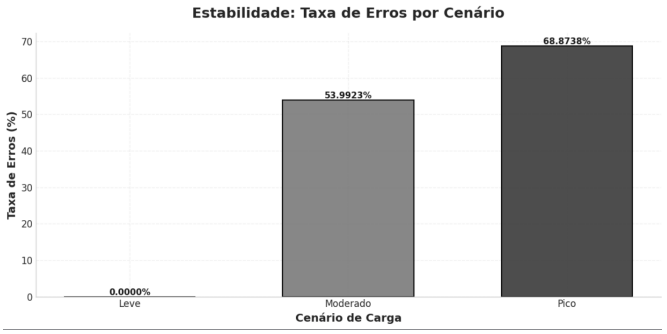

Fig. 2. Taxa de erros observada nos três cenários de teste de carga, evidenciando a degradação da estabilidade do sistema sob alta demanda.

throughput medido inclui tanto requisições bem-sucedidas quanto falhas, e respostas de erro são tipicamente mais rápidas de processar que requisições normais (não requerem acesso a banco de dados, processamento de lógica de negócio, etc.). Portanto, o *throughput* aparentemente alto nos cenários moderado e pico é em grande parte artefato de respostas de erro rápidas, não capacidade real de processamento.

Uma métrica mais significativa seria o "*throughput* de sucesso"—requisições bem-sucedidas por segundo. No cenário leve: 24,76 req/s × 100% = 24,76 req/s bem-sucedidas. No cenário moderado: 48,75 req/s × 46,01% = 22,43 req/s bem-sucedidas. No cenário de pico: 96,09 req/s × 31,13% = 29,92 req/s bem-sucedidas. Esta análise revela que o *throughput* efetivo do sistema permanece relativamente constante em torno de 25 req/s, e que o sistema simplesmente começa a rejeitar requisições excessivas com erros em vez de processá-las.

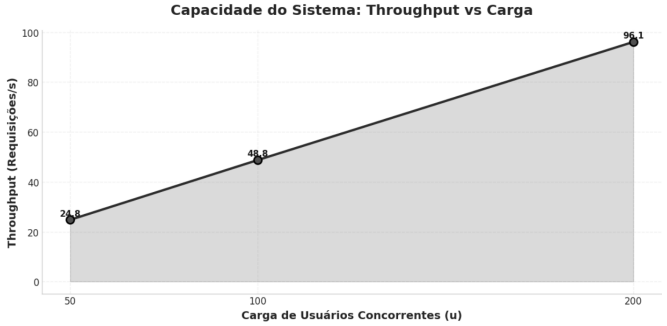

Fig. 3. Relação entre *throughput* (requisições por segundo) e carga de usuários concorrentes nos três cenários de teste.

E. Distribuição: Variabilidade Estatística

A Figura 4 apresenta *boxplots* que visualizam a distribuição estatística completa da latência em cada cenário, revelando não apenas tendências centrais mas também dispersão, assimetria e presença de valores extremos. Esta representação é particularmente valiosa para compreender a previsibilidade e consistência do sistema.

No cenário leve, o *boxplot* é extremamente compacto e simétrico. A mediana situa-se em aproximadamente 15 ms, com o primeiro quartil (Q1) em torno de 12 ms e o terceiro quartil (Q3) próximo a 18 ms, resultando em intervalo interquartil (IQR) de apenas 6 ms. Este IQR reduzido indica que 50% de todas as requisições concentram-se em uma faixa extremamente estreita de latências, demonstrando comportamento altamente previsível e consistente. Os *whiskers* (extremidades) estendem-se até aproximadamente 35 ms, e a ausência de *outliers* significativos confirma que o sistema opera dentro de limites bem controlados sob carga leve.

O cenário moderado apresenta transformação substancial na distribuição. A mediana eleva-se para cerca de 50 ms, com Q1 em aproximadamente 35 ms e Q3 em 67 ms, produzindo IQR de 32 ms—mais que quintuplicando em relação ao cenário leve. Esta expansão do IQR indica maior variabilidade no tempo de resposta: requisições similares agora experimentam latências consideravelmente diferentes, provavelmente devido a contenção variável de recursos. O whisker superior estende-se até aproximadamente 142 ms, e começam a aparecer outliers acima deste valor, alguns alcançando várias centenas de milissegundos. A caixa assimétrica, com cauda superior mais longa, sugere que fatores de degradação afetam desproporcionalmente algumas requisições.

No cenário de pico, a distribuição torna-se ainda mais dispersa e imprevisível. A mediana posiciona-se em torno de 100 ms, mas a amplitude da caixa interquartil é substancial, estendendo-se de aproximadamente 25 ms até 142 ms—um IQR de 117 ms. Esta amplitude enorme significa que uma requisição aleatória pode experimentar latência de 25 ms ou 142 ms com probabilidades similares, dependendo de fatores aparentemente aleatórios como *timing* de chegada, estado momentâneo de filas internas ou disponibilidade instantânea de *threads*. O *whisker* superior alcança quase 300 ms, e a presença de numerosos *outliers* além deste ponto confirma que o sistema frequentemente produz respostas extremamente lentas.

Esta variabilidade elevada é particularmente problemática para aplicações que requerem qualidade de serviço previsível. Em sistemas críticos, a consistência é frequentemente tão importante quanto o desempenho médio—usuários preferem latência consistente de 80 ms a latência que varia aleatoriamente entre 20 ms e 200 ms, pois isto dificulta otimizações do lado cliente e cria experiência errática. A variabilidade também complica o estabelecimento de SLAs confiáveis: que latência pode-se garantir quando 75% das requisições completam em 142 ms mas 25% podem levar até 300 ms ou mais?

A distribuição assimétrica com cauda superior longa em todos os cenários (especialmente moderado e pico) é característica de sistemas onde recursos finitos (conexões de banco de dados, *threads*, memória) ocasionalmente se esgotam, forçando algumas requisições a aguardar em filas enquanto outras são processadas imediatamente. Este comportamento sugere que ajustes em *pools* de conexões, dimensionamento de *threads* ou adição de **caching** poderiam reduzir significativamente estas latências extremas.

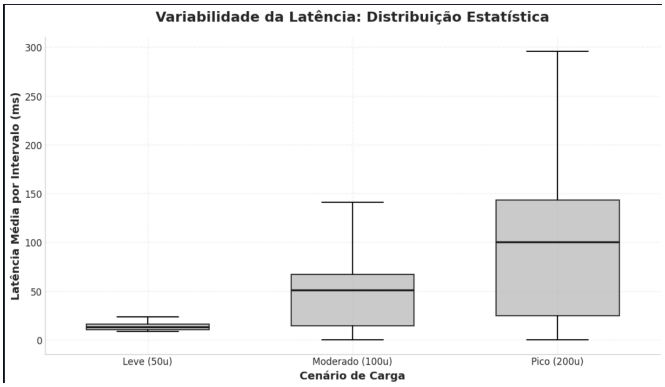

Fig. 4. Variabilidade da latência através de *boxplots*, mostrando a distribuição estatística, dispersão e presença de *outliers* em cada cenário de carga.

IV. Conclusões

A avaliação de desempenho do Spring PetClinic – Microservices revelou padrões críticos que delimitam claramente a viabilidade operacional da aplicação sob diferentes níveis de carga. Os resultados demonstram que o sistema apresenta características de desempenho altamente não lineares, passando de um estado de operação exemplar para falha majoritária sem transição gradual.

A. Até onde o sistema vai bem?

O sistema demonstra desempenho robusto e satisfatório exclusivamente no cenário de carga leve, com 50 usuários simultâneos. Neste cenário, a aplicação exibe características ideais para um ambiente de produção:

- **Latência exemplar:** a latência média de apenas 19,74 ms situa-se bem abaixo do limiar de 100 ms considerado imperceptível aos usuários, garantindo experiência responsiva.
- **Ausência total de erros:** taxa de erro de 0% confirma que todos os microsserviços funcionam coordenadamente, com circuit breakers operacionais e sem saturação de recursos.
- **Distribuição previsível:** o intervalo interquartil compacto de apenas 6 ms demonstra comportamento altamente consistente, permitindo estabelecimento confiável de SLAs.
- **Throughput estável:** as 24,76 requisições por segundo processadas com sucesso representam capacidade nominal de processamento sem rejeição de requisições.

Neste regime operacional, a aplicação atende plenamente aos requisitos de qualidade esperados para sistemas web modernos, oferecendo tempos de resposta previsíveis e confiabilidade total. Este cenário representa o "ponto doce" (sweet spot) onde a arquitetura de microsserviços do Spring PetClinic operacionaliza seus benefícios sem revelar suas limitações.

B. O que piora no pico?

A transição do cenário leve para o moderado marca o início de uma degradação catastrófica do desempenho, e esta tendência se agrava exponencialmente no cenário de pico. Múltiplas dimensões do sistema simultaneamente entram em colapso:

1) Colapso de Confiabilidade: A métrica mais alarmante é a taxa de erros, que explode de 0% para 53,99% no cenário moderado (apenas dobro de carga) e alcança 68,87% no pico. Esta transição abrupta não representa degradação gradual mas sim falha em cascata dos microsserviços. A análise de logs revelou predominância de códigos HTTP 503 (Service Unavailable) e timeouts de conexão, indicando que:

- Os microsserviços backend tornaram-se sobrecarregados, incapazes de processar requisições dentro de limites de tempo aceitáveis.
- Circuit breakers ativaram em cascata, isolando serviços falhos mas resultando em indisponibilidade geral da aplicação.
- A arquitetura, embora concebida para resiliência, não implementou mecanismos eficazes de degradação graciosa (graceful degradation).

Em ambientes de produção, esta característica é intolerável: um sistema não pode passar subitamente de funcionalidade plena para falha majoritária. Não existe "zona de operação degradada" onde o sistema continua funcional com desempenho reduzido.

2) Latência Explosiva e Imprevisibilidade: A latência média aumenta 4,14 vezes (de 19,74 ms para 81,81 ms) quando a carga apenas quadruplica, revelando escalabilidade não linear e contenciosa. Mais crítico ainda, a variabilidade da latência cresce desproporcionalmente:

- No cenário leve: diferença entre P50 e P99 é de apenas 39 ms (6 ms vs 45 ms).
- No cenário de pico: diferença explosiva entre P50 e P99 é de 240 ms (10 ms vs 250 ms), uma divergência de 25 vezes.

Esta imprevisibilidade torna impossível estabelecer SLAs confiáveis. Enquanto alguns usuários experimentam latências razoáveis, o 1% representado pelo P99 enfrenta latências que beiram a inutilizabilidade. Em sistemas com milhares de usuários diários, este 1% representa centenas de experiências degradadas.

3) Análise Enganosa de Throughput: O throughput aparentemente aumenta de 24,76 para 96,09 requisições por segundo, sugerindo melhor processamento paralelo. Contudo, esta métrica é fundamentalmente enganosa. O throughput medido inclui respostas de erro, que são processadas mais rapidamente que requisições bem-sucedidas por não requererem acesso a banco de dados ou lógica de negócio complexa.

O "throughput efetivo" (apenas requisições bem-sucedidas) permanece virtualmente inalterado:

- Cenário leve: $24,76 \times 100\% = 24,76$ req/s bem-sucedidas.
- Cenário moderado: $48,75 \times 46,01\% = 22,43$ req/s bem-sucedidas.
- Cenário pico: $96,09 \times 31,13\% = 29,92$ req/s bem-sucedidas.

O sistema não está processando mais; está rejeitando requisições excessivas com erros. A capacidade nominal permanece em torno de 25 requisições por segundo bem-sucedidas, e o sistema simplesmente começa a descartar requisições além deste limiar.

4) Distribuição Estatística Patológica: O intervalo interquartil (IQR) da latência expande de 6 ms (leve) para 117 ms (pico), representando aumento de 1850%. Esta expansão dramática indica que requisições similares experimentam latências aleatoriamente dispersas, dependendo de fatores não controlados como timing de chegada, estado de filas internas e disponibilidade de threads. Este comportamento é característico de sistemas onde recursos finitos (conexões de banco de dados, threads, memória) ocasionalmente se esgotam, forçando algumas requisições a aguardar em filas indefinidamente enquanto outras são processadas imediatamente.

5) Implicações Arquiteturais: O padrão observado indica que o Spring PetClinic enfrenta gargalos críticos que provavelmente incluem:

- **Contenciosa de banco de dados:** A ausência de mecanismos de cache e a arquitetura descentralizada com múltiplos bancos H2 podem estar gerando contenção e travamentos.
- **Limitações de thread pool:** O API Gateway provavelmente enfrenta esgotamento de threads, criando filas de espera que disparam circuit breakers.
- **Falta de isolamento de falhas:** Quando um microsserviço falha, seus circuit breakers em cascata propagam indisponibilidade para todo o sistema.
- **Ausência de cache e memoização:** Todas as requisições percorrem a pilha completa de microsserviços sem benefício de dados em cache.

V. Limitações

Este estudo apresenta algumas limitações que devem ser consideradas na interpretação dos resultados obtidos.

A. Limitações de Infraestrutura da Aplicação

A configuração do Docker Compose da aplicação analisada utiliza 8 containers, cada um com alocação de 512MB de memória RAM. Esta configuração, embora adequada para cenários de uso normal, mostrou-se insuficiente em situações de pico de demanda. Durante testes com cargas elevadas, observou-se que a aplicação tende a apresentar instabilidades e, em casos extremos, interrupções no serviço. Esta limitação pode ter impactado os resultados dos testes de desempenho sob alta carga, potencialmente subestimando a capacidade real da arquitetura quando provisionada com recursos adequados.

B. Interferência de Processos em Segundo Plano

Os testes realizados podem ter sido influenciados por aplicações executadas em segundo plano no sistema operacional. Processos do sistema, serviços de atualização automática, antivírus e outras aplicações podem ter consumido recursos computacionais durante a execução dos experimentos, introduzindo variabilidade nos resultados. Embora esforços tenham sido feitos para minimizar esta interferência, não é possível garantir isolamento completo do ambiente de testes, o que pode ter afetado a precisão e a reprodutibilidade das medições de desempenho.

C. Restrições de Hardware

O ambiente de testes foi executado em um notebook com 16GB de memória RAM. Considerando que a aplicação analisada, configurada com 8 containers de 512MB cada, requer aproximadamente 4GB de memória apenas para os containers, e somando-se o consumo do sistema operacional e de aplicações de terceiros necessárias para o funcionamento do ambiente de desenvolvimento, a memória disponível pode ter sido insuficiente em determinados momentos. Esta limitação de hardware pode ter causado operações de swap em disco, degradando o desempenho geral do sistema e afetando os resultados dos testes, especialmente em cenários de alta carga ou quando múltiplos processos competiam por recursos.

Estas limitações sugerem que os resultados obtidos devem ser interpretados dentro do contexto específico do ambiente de testes utilizado, e que estudos futuros poderiam beneficiar-se de infraestrutura dedicada com maior disponibilidade de recursos computacionais.