Universidade de Passo Fundo - UPF
Instituto de Tecnologia - ITEC
Ubiquitous Computing: Clouds, Iot e Smart Environments

Testes de Desempenho com APIs e Gateway

Bernardo José Zaparoli[1]

[1]Instituto de Tecnologia – Universidade de Passo Fundo (UPF)
Passo Fundo – RS – Brazil

189797@upf.br

Abstract. *This report compares the performance of Flask (Python) and Laravel (PHP) APIs in a Docker environment under load testing (JMeter), aiming to identify the performance bottleneck. Flask demonstrated superiority, achieving significantly better throughput and latency (e.g., 15-23 ms vs. 120-148 ms). Resource analysis revealed that Laravel saturated the CPU ($\approx 100\%$), confirming PHP as the limiting factor. In contrast, Flask maintained low CPU consumption ($\approx 12\%$). It is concluded that Flask/Python is the most efficient and robust option for backend services requiring high concurrency and scalability.*

Resumo. *Este relatório compara o desempenho de APIs Flask (Python) e Laravel (PHP) em ambiente Docker sob teste de carga (JMeter), visando identificar o gargalo de performance. O Flask demonstrou superioridade, alcançando throughput e latência significativamente melhores (ex: 15-23 ms vs. 120-148 ms). A análise de recursos revelou que o Laravel saturou a CPU ($\approx 100\%$), confirmando o PHP como o fator limitante. Em contraste, o Flask manteve o consumo de CPU baixo ($\approx 12\%$). Conclui-se que o Flask/Python é a opção mais eficiente e robusta para serviços de backend com alta concorrência e escalabilidade.*

1. Introdução

Este relatório foca na **análise comparativa de desempenho** entre *APIs* desenvolvidas em *Flask (Python)* e *Laravel (PHP)*, integradas a um banco de dados **PostgreSQL** e orquestradas em um ambiente containerizado por meio do **Docker**. O estudo utiliza um ambiente de testes que simula uma arquitetura de produção com um *gateway* **Nginx** (configurado com *Round Robin* para roteamento) e a ferramenta **Apache JMeter** para simulação de carga. A pesquisa tem como objetivo **monitorar métricas críticas** como consumo de **CPU** (*Central Processing Unit*), *Random Access Memory (RAM)*, latência e vazão (*throughput*) sob estresse. O resultado visa fornecer dados quantitativos para evidenciar as vantagens e gargalos de performance de cada *stack*.

2. Implementação

O ambiente de testes foi construído utilizando contêineres *Docker*, garantindo isolamento e reprodutibilidade. Foram desenvolvidas duas *APIs RESTful*, uma em *Flask*

(*Python*) e outra em *Laravel* (*PHP*), com dependências gerenciadas respectivamente por `requirements.txt` e `composer.json`. A orquestração dos serviços foi feita via `docker-compose`, integrando um *gateway Nginx* configurado com o algoritmo *Round Robin* no arquivo (`nginx.conf`) para distribuir as requisições entre as instâncias.

Para a persistência de dados, utilizou-se um contêiner de banco de dados *PostGresSQL* dedicado, inicializado pelo *script* `banco.sql`, que definiu *schemas* isolados para cada aplicação. As *APIs* implementaram *endpoints* para operações *CRUD* (criação, leitura, atualização e exclusão), interagindo diretamente com o banco. Essa abordagem assegurou que as métricas de desempenho incluíssem tanto o tempo de processamento interno quanto a latência das operações de banco de dados.

A validação do roteamento e do algoritmo *Round Robin* no *gateway* foi realizada por meio de *scripts Bash*. Já os testes de carga foram executados com a ferramenta *JMeter*, mensurando o desempenho de cada *API* sob diferentes volumes de usuários e requisições. Os dados coletados, como consumo de *CPU* e tempos de resposta, foram organizados para análise comparativa, estando todo o procedimento de configuração e execução documentado no arquivo `README.md`.

3. Testes e Resultados

Para a realização dos testes de desempenho, foi utilizado um conjunto de cenários que variaram tanto o volume de requisições quanto o número de usuários simultâneos, simulando diferentes níveis de carga sobre as *APIs*. Os testes foram automatizados por meio da ferramenta *JMeter*, que permitiram a execução controlada e repetitiva das requisições aos *endpoints* implementados. As configurações dos testes realizados pelo *Jmeter* são as seguintes:

- 10 / 10 / 10 (Duração: 0.15 segundos)
- 50 / 10 / 10 (Duração: 5min.43segundos)
- 100 / 10 / 100 (Duração: 8min.27segundos)
- 500 / 10 / 500 (Duração: 22min.34segundos)
- 1000 / 10 / 1000 (Duração: 40min.32segundos)
- Número de usuários / Tempo de inicialização / Contador de interação

3.1. Consumo de CPU, memória e tempo de execução dos testes

Table 1. Médias de CPU e Memória capturadas pelos contâiners Docker

Container	CPU (10/10/10)	MEM (10/10/10)	CPU (50/10/50)	MEM (50/10/50)	CPU (100/10/100)	MEM (100/10/100)	CPU (500/10/500)	MEM (500/10/500)	CPU (1000/10/1000)	MEM (1000/10/1000)
api_gateway	0.00%	2.54 MiB	0.00%	2.54 MiB	0,00%	2,49 MiB	0,00%	2,54 MiB	0,00%	2,54 MiB
flask_api	8.17%	81.86 MiB	11,65%	82,05 MiB	11,60%	81,50 MiB	11,65%	82,10 MiB	13,40%	82,10 MiB
flask_api2	7.94%	81.87 MiB	11,32%	82,10 MiB	11,25%	81,45 MiB	11,30%	82,15 MiB	12,87%	82,15 MiB
laravel_api	65.92%	58.79 MiB	93,50%	60,01 MiB	98,50%	63,20 MiB	93,50%	60,10 MiB	95,62%	64,78 MiB
laravel_api2	68.25%	59.14 MiB	97,20%	60,80 MiB	101,20%	63,15 MiB	97,20%	60,80 MiB	99,52%	51,28 MiB
pgadmin_web	23.40%	186.17 MiB	28,00%	205,00 MiB	0,04%	212,50 MiB	0,05%	205,00 MiB	0,05%	205,00 MiB
postgres_db	35.14%	32.73 MiB	51,50%	35,50 MiB	52,40%	31,50 MiB	51,50%	35,50 MiB	51,50%	35,50 MiB

Conforme demonstrado na **Tabela 1**, as instâncias da *API Laravel* atingiram saturação de **CPU (próximo a 100%)** sob carga, estabelecendo o processamento *PHP* como o **gargalo computacional** primário do sistema e o fator limitante para o *throughput* máximo. Em contraste, as *APIs Flask* mantiveram uma utilização média de apenas $**\approx 12\%$ da *CPU***, evidenciando alta eficiência e uma folga operacional significativa para maior concorrência. A distribuição simétrica da carga entre as instâncias *(api1*

e *api2)* de ambas as tecnologias valida a uniformidade do teste. A análise revela um **trade-off** no consumo de recursos: o *Flask* apresentou um *overhead* de memória maior ($\approx 80 - 90$ *MiB*), mas demonstrou superioridade em eficiência de *CPU*. O *Laravel*, por sua vez, foi mais econômico em *RAM* ($\approx 60 - 70$ *MiB*), à custa da saturação do processamento. Por fim, o banco de dados *PostgreSQL* operou com *CPU* moderada ($\approx 50 - 60\%$), o que confirma que a **persistência de dados não foi o gargalo** primário, reforçando a limitação na camada de aplicação *PHP*.

3.2. Percentual de erros, *Throughput* e Tempo médio de resposta

Table 2. Médias das requisições capturadas pelo Jmeter

Rótulo	% de Erros (10/10/10)	Throughput (10/10/10)	Tempo Resposta (ms) (10/10/10)
Flask (Python)	0,00%	~ 148,8 Req/s	~ 17
Laravel (PHP)	0,00%	~17,5 Req/s	~ 155
Rótulo	**% de Erros (50/10/50)**	**Throughput (50/10/50)**	**Tempo Resposta (ms) (50/10/50)**
Flask (Python)	0,00%	~ 147,7 Req/s	~ 155
Laravel (PHP)	0,00%	~ 17,6 Req/s	~ 155
Rótulo	**% de Erros (100/10/100)**	**Throughput (100/10/100)**	**Tempo Resposta (ms) (100/10/100)**
Flask (Python)	0,00%	~ 148,8 Req/s	~ 18
Laravel (PHP)	0,00%	~ 17,3 Req/s	~ 156
Rótulo	**% de Erros (500/10/500)**	**Throughput (500/10/500)**	**Tempo Resposta (ms) (500/10/500)**
Flask (Python)	0,00%	~ 147,7 Req/s	~ 25
Laravel (PHP)	0,00%	~ 17,6 Req/s	~ 156
Rótulo	**% de Erros (1000/10/1000)**	**Throughput (1000/10/1000)**	**Tempo Resposta (ms) (1000/10/1000)**
Flask (Python)	0,00%	~ 145,5 Req/s	~ 24
Laravel (PHP)	0,00%	~ 17,4 Req/s	~ 160

Na **Tabela 2**, as rotas *Laravel* demonstram latências na faixa superior do gráfico, indicando que o tempo de resposta é consistentemente 6 a 8 vezes superior (mais lento) em comparação com as rotas *Flask*. Por sua vez, as rotas *Flask* exibem o menor consumo de tempo de resposta, com latências agrupadas na faixa inferior, confirmando a alta velocidade de processamento da *stack Python*. No que tange à estabilidade do serviço, ambas as tecnologias demonstraram um desempenho notavelmente previsível sob estresse: a proximidade entre o Tempo Médio e o Tempo Máximo em todos os rótulos de requisição indica uma baixa variação *(jitter)*. Esta estabilidade é crucial, pois no caso do *Laravel*, ela corrobora a análise de *CPU*, confirmando que a saturação do processamento impõe um teto consistente, embora elevado, ao tempo de resposta. A *stack Flask* mantém essa mesma estabilidade, mas em um patamar de latência extremamente baixo, validando sua superior eficiência computacional para o cenário de concorrência testado.

4. Conclusão

Os resultados demonstram que a *stack* **Flask** alcançou desempenho superior, apresentando *throughput* até 8 vezes maior e latência média ≈ 6 a 8 vezes menor que o *Laravel* (Média de $15 - 23$ ms versus $120 - 148$ ms). O principal achado é o **gargalo de *CPU***: as instâncias *Laravel* operaram em saturação (próximo a 100%), limitando o desempenho máximo, enquanto as instâncias *Flask* mantiveram uma folga de processamento significativa ($\approx 12\%$ de *CPU*). Embora o *Laravel* tenha sido mais econômico em memória *RAM*, a alta eficiência de *CPU* do *Flask* o torna drasticamente mais escalável. Conclui-se que, para projetos que demandam alta performance e capacidade de resposta sob estresse de concorrência, a *stack Flask/Python* demonstrou ser a opção tecnologicamente mais eficiente e robusta.

5. References

O desenvolvimento dos contâiners *Docker* para este projeto, foram realizados seguindo os padrões de criação e orquestração disponibilizados pela própria ferramenta. [doc 2025].

Da mesma forma que as duas *APIs* foram construídas seguindo a própria documentação fornecida pelos *frameworks* [fla 2025] [lar 2025].

References

(2013-2025). Docker manuals. Disponível em: https://docs.docker.com/manuals/. Acesso em: 21 nov. 2025.

(2025). Flask documentation. Disponível em: https://flask.palletsprojects.com/en/stable/. Acesso em: 21 nov. 2025.

(2025). Laravel documentation. Disponível em: https://laravel.com/docs/12.x. Acesso em: 21 nov. 2025.