Relatório de Benchmark: Limites de Performance por Stack

Escopo do Teste

O teste consiste em avaliar o comportamento das stacks em um cenário real de negócio através de dois endpoints principais:

1. `POST /bonus` **(Escrita/Processamento)**:
    - Valida se o cliente existe e está ativo no banco de dados.
    - Aplica uma regra de negócio: se o valor for > 100, aplica um bônus de 10%.
    - Registra o bônus com data de expiração (30 dias) e descrição formatada.
    - Persiste os dados no PostgreSQL.
2. `GET /bonus/recents` **(Leitura/Estresse de Memória)**:
    - Busca os últimos 100 registros do banco de dados.
    - **Ordenação em Memória**: A ordenação por data de criação é feita propositalmente na camada de aplicação (e não no banco) para estressar CPU e Memória.
    - Retorna apenas os 10 resultados mais recentes após o processamento.

Resumo Executivo

Este documento detalha o teste de performance realizado em diversas tecnologias (stacks) para identificar a **taxa máxima de requisições por segundo (RPS)** que cada uma suporta, mantendo a latência **P95 abaixo de 200ms**.

O teste focou em encontrar o "teto" de cada stack sob condições de hardware idênticas, utilizando uma estratégia de carga progressiva (ramping).

Metodologia do Teste

Infraestrutura e Hardware

Para garantir a isonomia, cada aplicação foi executada com exatamente os mesmos recursos:

- **CPU:** 1 Core com clock de 4.3GHz
- **Memória RAM:** 1 GB
- **Instâncias:** 2 instâncias por stack, rodando em um orquestrador **Docker Swarm**.
- **Banco de dados:** foi utilizado banco de dados postgres com pgbouncer na frente garantindo que todas as stacks fossem limitadas igualmente a 240 conexões máximas.

Pontos importantes

- Calibragem: foram realizados diversos testes calibrando parametros importantes nas linguagens como por exemplo workers no octane e python , max children no fpm, por exemplo foram testados valores como 1,2,3,4,5,6 e foi definido nessas stacks o melhor cenário.

Coleta de Métricas

As métricas foram coletadas e centralizadas em um servidor **Prometheus**, recebendo dados de duas fontes principais:

1. **Cadvisor:** Métricas de consumo de recursos do container (CPU/Memória).
2. **Traefik:** O ponto de entrada (Edge Router/Load Balancer) que forneceu as métricas de latência e throughput do tráfego real.

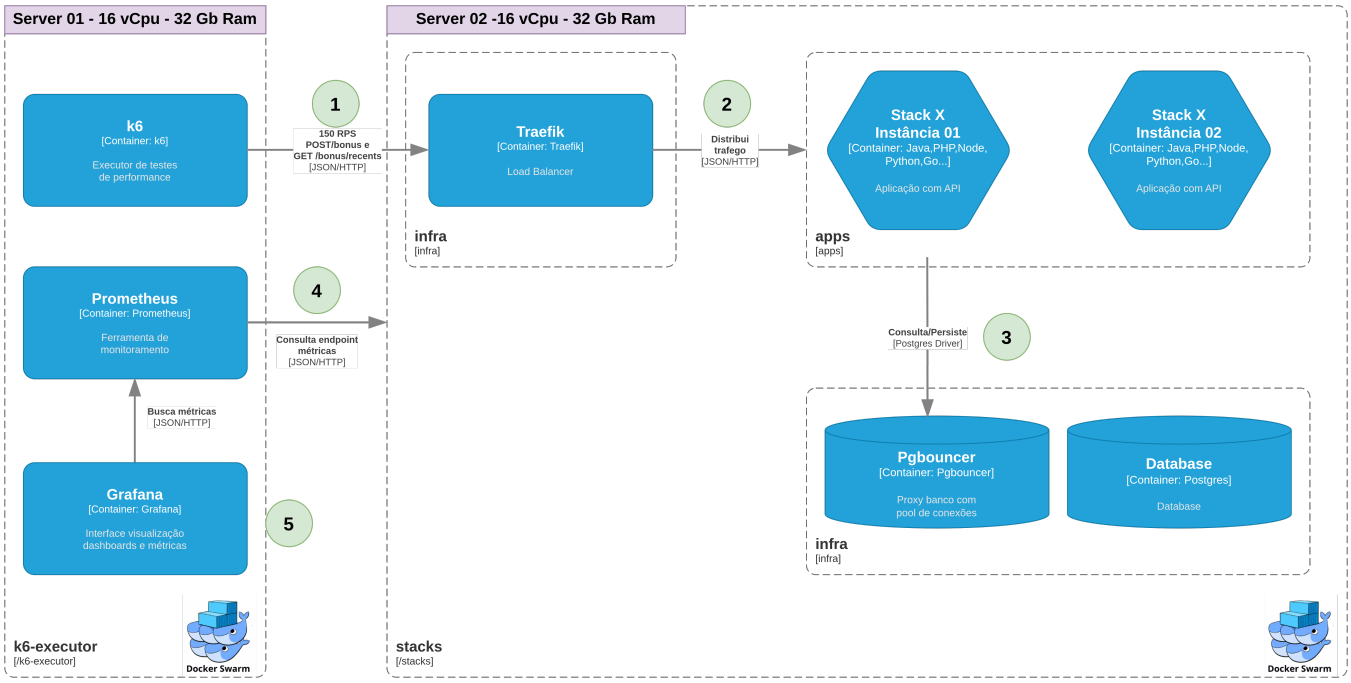
Estratégia de Carga (k6)

O script de teste k6 utilizou o executor `ramping-arrival-rate` do k6:

- **Aquecimento (Warm-up):** O script escala a carga gradualmente. Foram ignorados os primeiros **8 minutos** de cada execução (utilizando `delayAbortEval`) para permitir o aquecimento da JVM, JIT e caches internos.
- **Escalonamento:** A carga inicia em 20 RPS e sobe em degraus (25,50,75, 100, 125, 150, 175, 200, 300, 400, 500, 600, 700, 800, 900 e 1000), com cada patamar durando 2 minutos.
- **Abort automático:** O teste é interrompido automaticamente se o threshold de latência for violado por um período sustentado, garantindo que o limite reportado seja o último patamar estável.

Infraestrutura e Coleta

As aplicações rodaram no **Docker swarm** e foram coletadas via **Prometheus**, consolidando dados do Docker (container) e do **Traefik** (edge router). Segue abaixo diagrama explicando a infraestrutura envolvida no teste:



- 1 - Na máquina 01 execução do teste pelo K6 enviando chamadas
- 2 - Na máquina 02 as aplicações de cada stack rodando de forma separada atendendo as requisições, tendo como ponto de entrega Traefik
- 3 - Na máquina 02 persiste e busca bônus no postgres

- 4 - Na máquina 01 ambiente de monitoramento com prometheus consulta os endpoints cadvisor e traefik para coletar métricas
- 5 - Na máquina 01 Grafana expoe dashboards para visualizar as métricas durante o teste

---

Resultados Obtidos

A tabela abaixo lista o **RPS máximo** alcançado por cada stack mantendo o **P95 < 200ms**:

Posição	Stack	RPS Máximo
1º	Rust	1200
1º	Java Quarkus	1200
2º	Java MVC (Virtual Threads)	1000
2º	Node + NestJS + Fastify	1000
3º	.NET	800
3º	Golang	800
3º	Node + NestJS + Express	800
3º	Java WebFlux	800
4º	Java MVC (Sem Virtual Threads)	600
5º	Python	400
5º	PHP + Laravel + Octane	400
6º	PHP + Laravel + FPM	200

Materiais/Documentos

O código fonte das aplicações e teste escrito está disponível em https://github.com/crmbonus-oficial/benchmark-stacks/benchmark/benchmark-limites

- `load-all-swarm-limites.js`: script com cenário de teste k6.
- `graficos-grafana.md`: Links dos dashboards grafana com as métricas de performance obtidas ao longo do teste.
- `reports`: Relatórios gerados pelo k6 resumindo o teste executado.

Conclusões

- **Rust e Java Quarkus** lideraram o benchmark (1200 RPS), confirmando alta eficiência de runtime e excelente aproveitamento de 1 core sob alta concorrência.
- **Java MVC (Virtual Threads)** e **Node + NestJS + Fastify** ficaram logo atrás (1000 RPS), demonstrando que otimizações no modelo de concorrência e no servidor HTTP reduzem significativamente o gap para os líderes.

- **.NET, Golang, Node + NestJS + Express e Java WebFlux** formaram o grupo intermediário (800 RPS), com desempenho consistente, porém inferior aos frameworks mais otimizados para baixa sobrecarga de CPU.
- **Java MVC (sem Virtual Threads)** apresentou queda relevante (600 RPS), evidenciando o impacto do modelo tradicional baseado em threads de plataforma.
- **Python** e **PHP + Laravel + Octane** atingiram 400 RPS, mostrando melhora no caso do Octane em relação ao modelo tradicional do PHP, mas ainda com limitações sob CPU restrita.
- **PHP + Laravel + FPM** obteve o menor throughput (200 RPS), refletindo o custo do modelo multiprocessos em ambiente limitado a 1 core.

Resumo: sob forte restrição de CPU, eficiência de runtime e modelo de concorrência foram decisivos, gerando até 6x de diferença de throughput entre as stacks.