Otimização de Aplicações Web com Programação Assíncrona em PHP e Swoole

Jailson Maciel da Luz[1], **Roger Sá da Silva**[1], **Cassandro Davi Emer**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Campus Avançado Veranópolis – Veranópolis – RS – Brasil

eumanito@gmail.com, [roger.silva, cassandro.emer]@veranopolis.ifrs.edu.br

Abstract. *The growing demand for faster and more scalable web applications has necessitated strategic decisions in system development to ensure both performance and efficiency. This study provides a comparative analysis of PHP web application development using the traditional synchronous model and the asynchronous model implemented with Swoole. The methodology involved conducting load tests in a controlled experimental environment, assessing the throughput of both models under varying load levels. Results were analyzed using ANOVA and T-Test statistical methods, confirming that the operating mode was a critical factor in optimizing performance, with the asynchronous model outperforming the synchronous model in handling multiple requests.*

Resumo. *O aumento constante na demanda por aplicações web mais rápidas e escaláveis tem exigido decisões estratégicas no desenvolvimento de sistemas a fim de garantir desempenho e eficiência. Este artigo apresenta uma análise comparativa entre o desenvolvimento de aplicações web em PHP utilizando o modelo síncrono tradicional e o modelo assíncrono, implementado com o Swoole. A metodologia envolveu testes de carga em um ambiente experimental controlado, avaliando-se a taxa de resposta (throughput) de ambos os modelos sob diferentes níveis de carga. Os resultados foram analisados por meio dos métodos estatísticos ANOVA e Teste T, e confirmou-se que o modo de operação foi o fator determinante para otimizar o desempenho, com o modelo assíncrono superando o modelo síncrono na capacidade de lidar com múltiplas requisições.*

1. Introdução

A programação assíncrona tem ganhado cada vez mais destaque no desenvolvimento de sistemas modernos, especialmente em aplicações *web* que demandam alta performance e escalabilidade. Diferente do modelo síncrono, onde as operações são executadas de forma sequencial, a programação assíncrona permite que múltiplas tarefas sejam realizadas simultaneamente [Rimma Zaripova e Perukhin 2023]. Isso otimiza o uso dos recursos do sistema e reduz o tempo de resposta, algo essencial para serviços que exigem baixa latência e alta capacidade de resposta.

Historicamente a linguagem PHP não oferecia a possibilidade de programação assíncrona devido a limitações no gerenciamento de E/S (entrada e saída) e execução paralela. Entretanto, com o surgimento de extensões como o Swoole, o PHP passou a oferecer ferramentas robustas para implementação de operações assíncronas de forma eficiente. O Swoole adiciona ao PHP a capacidade de trabalhar com corrotinas e gestão

de conexões com o banco de dados, por exemplo. Assim ampliando significativamente as possibilidades da linguagem no desenvolvimento de aplicações de alta performance.

O Swoole em projetos PHP representa uma grande evolução na linguagem, permitindo que os desenvolvedores se beneficiem das vantagens da programação assíncrona sem abandonar a familiaridade da linguagem. Com o Swoole é possível criar sistemas capazes de receber milhares de requisições simultâneas, melhorando significativamente o tempo de resposta, oferecendo escalabilidade e desempenho.

Este trabalho tem como objetivo comparar o desempenho dos modelos de programação síncrona e assíncrona em PHP, explorando os conceitos da programação assíncrona e demonstrando que a extensão Swoole pode ser aplicada para otimizar aplicações *web* desenvolvidas nessa linguagem. O estudo visa analisar o impacto da execução assíncrona em operações de entrada e saída como o principal ponto de melhoria no desempenho de sistemas, com foco na otimização do tempo de resposta das requisições (*throughput*), em relação ao modelo síncrono (PHP tradicional).

O trabalho está estruturado da seguinte forma: na seção 2 apresenta-se os conceitos teóricos e os fundamentos necessários para contextualizar a programação assíncrona em PHP. Na seção 3, estão descritos os materiais e métodos aplicados para o experimento comparativo construído para este estudo. A seção 4 detalha os resultados obtidos com o experimento e sua análise, enquanto a seção 5 discute a comparação entre os modos síncrono e assíncrono, abordando aspectos como desempenho, escalabilidade e desafios. Por fim, a seção 6 traz as considerações finais, incluindo as limitações deste estudo e sugestões para futuros trabalhos.

2. Embasamento teórico

Para compreensão do objetivo deste trabalho e o experimento realizado, serão apresentados os fundamentos da programação síncrona e assíncrona, abordando-se as principais diferenças entre esses dois modelos. Nesta seção explora-se as vantagens e desafios de cada abordagem, com foco na aplicação em sistemas *web* que necessitam de alto desempenho.

2.1. Funcionamento básico do PHP

De acordo com a sua documentação oficial [PHP 2024], PHP é um acrônimo recursivo para *Hypertext Preprocessor* (Pré-Processador de Hipertexto), uma linguagem de *script* de código aberto amplamente utilizada para desenvolvimento *web*, operando principalmente no modelo de requisição e resposta. Pode-se detalhar um ciclo de vida para este processo. Quando um cliente (geralmente um navegador *web*) faz uma requisição a um servidor que executa *scripts* PHP, essa requisição é recepcionada por um serviço *web* como Apache ou Nginx, sendo então encaminhada para o interpretador PHP. A partir desse ponto, inicia-se o processo de interpretação e execução do *script*.

A maneira tradicional de executar o PHP em um servidor *web* é via *mod_php*, o que significa que a execução do PHP *engine* ocorre dentro do próprio servidor *web*, de forma a requerer a menor configuração e basicamente “funciona imediatamente” devido ao PHP possuir o seu próprio gerenciador de processos (*Process Manager*) [Heng 2015].

2.2. Arquitetura PHP-FPM

Devido à necessidade de escalabilidade surgiu o PHP-FPM (*FastCGI Process Manager*). [Heng 2015] explica que, diferente do CGI tradicional, onde cada requisição gera um novo processo, o FastCGI mantém processos persistentes, reduzindo a carga de CPU. No fluxo de funcionamento do PHP-FPM, o Apache atua como intermediário, encaminhando as requisições para o servidor PHP-FPM, que executa de forma independente do Apache.

[Liu et al. 2023] explica que o fluxo de requisições recepcionadas pelo Apache entram em uma fila e ficam aguardando por um *worker*. Cada *worker* é representado por um processo, sendo este responsável por processar uma única requisição de forma independente e do início ao fim, antes de aceitar uma nova requisição.

Este modelo permite distribuir a carga entre vários servidores PHP-FPM, de forma que cada requisição seja atendida de maneira ordenada e eficiente. Assim melhorando a gestão de memória e a segurança, pois cada componente pode ser gerenciado e atualizado de forma independente, além de garantir que o servidor não sobrecarrege os recursos disponíveis.

Figura 1. Bloqueio sequencial de operações de entrada e saída no modelo síncrono, onde o processo aguarda a conclusão de cada operação de E/S antes de iniciar a próxima, tornando-se bloqueante [Dou 2020].

De acordo com [Dou 2020], o PHP-FPM não possui estado (*stateless*), de forma que os recursos não são compartilhados entre processos distintos. Dessa forma, todos os recursos utilizados são criados e destruídos junto ao ciclo de vida das solicitações. Na Figura 1 observa-se quatro tarefas executadas em sequência, onde a área cinza claro são o tempo de espera de requisições E/S e a área cinza escuro o tempo de CPU. Portanto, por natureza, o PHP-FPM executa os processos de forma bloqueante.

2.3. *Multithreading* Assíncrono

O conceito de *multithreading* assíncrono aborda a capacidade de um *software* realizar múltiplas operações simultaneamente. Mesmo utilizando-se de um único processador, pode-se executar diferentes tarefas simultâneas por meio de comutação rápida, assim otimizando o uso de *threads* do sistema para melhorar o seu desempenho [Yang et al. 2018].

O autor traz o conceito de *multithreading* e faz analogia entre *threads* de baixo nível – utilizadas pelo sistema operacional ao comunicar-se com o *hardware* – e *threads* de alto nível que são emuladas pelo *software*, também conhecidas como *threads* virtuais. O autor explica: “as *threads* virtuais são uma abstração que permite a criação

de vários *threads* sem exigir *threads* dedicados do sistema operacional para cada um" [Chigurov et al. 2024].

[Yang et al. 2018] A tecnologia *multithreading* assíncrona é utilizada em casos de alta simultaneidade como por exemplo o sistema de buscas do Google e Yahoo, plataformas de comércio eletrônico, redes sociais, entre outros.

> O cenário de uso específico deve atender a diversas condições: nenhum recurso compartilhado ou somente leitura para recursos compartilhados; nenhuma relação estrita quanto ao momento; sem operações atômicas; nenhuma influência na lógica do *thread* principal, etc., frequentemente usada para operações demoradas, como operações de E/S [Yang et al. 2018].

Em um cenário onde 100 requisições chegam simultaneamente, na arquitetura utilizando o PHP-FPM, seria necessário iniciar 100 processos independentes, cada um estabelecendo uma conexão própria com o banco de dados, devido à impossibilidade de reutilização de conexões entre essas solicitações.

De acordo com [Dou 2020], os processos são concluídos rapidamente em situações de baixa complexidade, como na simples exibição de um texto estático. Porém o autor também contrapõe o cenário com aplicações modernas que requerem frequentemente operações externas de E/S, como consultas e gravações em bancos de dados, manipulação de arquivos, e interações com API's externas. Essas operações, se não gerenciadas de forma eficiente, podem resultar em processos ociosos, apenas aguardando a conclusão dessas operações externas, sem realizar nenhuma tarefa útil. Para tanto, [Dou 2020] explica: "Tornar o processo assíncrono é a ideia geral para resolver esse problema de simultaneidade".

2.4. Arquitetura com Swoole

O Swoole é uma extensão PHP implementada em linguagem C que oferece funcionalidades avançadas de programação de rede e E/S assíncrona, superando as limitações tradicionais do PHP. Diferente das extensões PHP convencionais que apenas fornecem funções de biblioteca, o Swoole assume o controle do PHP após a execução, entrando em um *loop* de eventos. Nesse *loop*, ao ocorrer um evento E/S, o Swoole controla a requisição externa e, posteriormente, retorna a função PHP específica. O Swoole facilita a construção de servidores assíncronos e *multithread*, fornecendo suporte para diversos serviços como *WebSockets*, HTTP/2.0, e clientes de banco de dados assíncronos, entre outros [Yang et al. 2018].

Figura 2. Multiplexação de operações de E/S no modelo assíncrono, destacando a utilização simultânea do tempo de CPU e E/S [Dou 2020].

Na Figura 2, observa-se o assincronismo permitindo que quatro tarefas sejam executadas simultaneamente, de forma que o tempo de espera de E/S de uma tarefa é utilizado por outras tarefas. As áreas cinza claro são o tempo de espera de E/S e a área cinza escuro o tempo de CPU.

O sistema de E/S sem bloqueio aumenta a utilização da CPU e pode executar mais tarefas e usar menos tempo em comparação com os sistemas de E/S com bloqueio. E/S sem bloqueio é uma das razões pelas quais o Swoole pode lidar com mais conexões simultâneas em comparação com o PHP-FPM [Dou 2020].

2.5. Corrotinas

As corrotinas são consideradas uma solução eficiente para a programação concorrente, oferecendo uma alternativa leve em comparação ao modelo tradicional de *multithreading* com memória compartilhada. Corrotinas, diferentemente das *threads*, operam inteiramente no modo do usuário (*user mode*), o que significa que suas operações não envolvem diretamente o *kernel* do sistema, resultando na redução de custo de criação e consumo [Yang et al. 2018].

Quando falamos de corrotinas no contexto do Swoole, elas são frequentemente chamadas de "*threads* leves" ou "*green threads*". Isso porque o Swoole agenda as corrotinas "com base na espera de E/S ou no *status* de conclusão" e fornece um agendador preemptivo que alterna entre corrotinas quando uma delas ocupa muito tempo da CPU [Dou 2020].

A seguir, será apresentado um exemplo de rotina em PHP que utiliza a função *sleep()* para simular o tempo de espera de uma operação de entrada/saída (E/S). A execução total da rotina será de 3 segundos, uma vez que o PHP síncrono aguarda a conclusão da função *sleep(1)* para então imprimir o texto "a", espera mais 2 segundos na função *sleep(2)*, e em seguida imprime o texto "b".

```
<?php
    sleep(1); // Espera 1 segundos
    echo("a");
    sleep(2); // Espera 2 segundos
    echo("b");
```

Agora, observe a seguir um exemplo utilizando corrotinas com o Swoole que irá imprimir "a" após 1 segundo do início da execução do *script*, e imprimir "b" após 2 segundos do início da execução do *script*. A diferença está em que o tempo de "b" não precisa aguardar o tempo de "a", pois são processos independentes.

```
<?php
    use Swoole\Coroutine\System;
    go(static function() {
        sleep(1); // Espera 1 segundos
        echo("a");
    });
    go(static function() {
        sleep(2); // Espera 2 segundos
        echo("b");
    });
```

3. Materiais e métodos

Esta seção descreve o experimento realizado neste trabalho, com o comparativo dos modelos síncrono e assíncrono de execução em PHP, ambos conectados a um banco de dados MySQL com *schemas* iguais. O experimento foi planejado com base em dados fornecidos pela empresa comercial Bling[1], incluindo dados sobre a arquitetura do sistema, volumes de operações em condições normais e de pico, além dos principais processos de entrada e saída envolvidos na geração de notas fiscais.

O objetivo de usar um cenário baseado em operações reais é garantir que os resultados fiquem mais próximos da realidade de produção. No caso, a geração de uma nota fiscal envolve diversas etapas, como operações no banco de dados (inserções, consultas, atualizações e exclusões) e a geração de arquivos. Essa abordagem permite avaliar melhor o desempenho dos modelos em situações de alta carga e concorrência, especialmente em sistemas que exigem muitas operações de entrada e saída e grande volume de transações.

3.1. Ambiente de teste

Para realizar a análise comparativa de desempenho, foram configurados dois ambientes distintos utilizando *containers* em Docker (Figura 3), de modo a permitir alternar entre os ambientes PHP síncrono e assíncrono. Optou-se por ativar cada *container* de forma independente para garantir o isolamento total durante os testes. Os ambientes foram hospedados nos servidores da Amazon utilizando a ferramenta AWS EC2, enquanto o banco de dados MySQL foi hospedado separadamente na ferramenta AWS RDS, a fim de evitar interferência no processamento dos ambientes durante os testes. O experimento consistiu em executar *scripts* PHP com duas operações principais: interagir com o banco de dados e gravar arquivos, de forma a simular o cenário real de geração de notas fiscais.

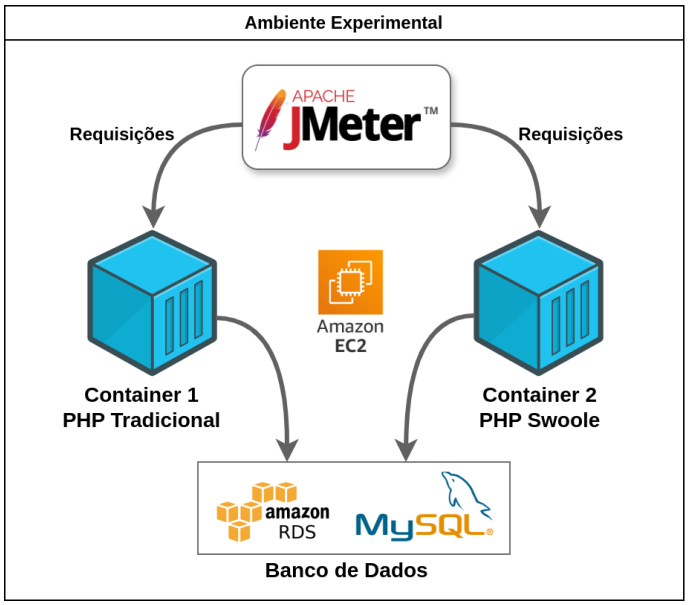

Figura 3. Diagrama do ambiente isolado do experimento, mostrando a configuração de *containers* Docker para simular a interação entre os diferentes componentes do sistema, com ênfase na separação de recursos entre os ambientes síncrono e assíncrono [autor].

[1]Dados fornecidos sob autorização para fins de pesquisa

3.1.1. Ambiente síncrono

Configurado com PHP 8, Apache como servidor *web*, e MySQL 5.7 como banco de dados. Neste ambiente, as operações de E/S e banco de dados são executadas de forma sequencial e bloqueante, onde cada operação precisa ser concluída antes que a próxima seja iniciada.

3.1.2. Ambiente assíncrono

Configurado com PHP 8 e a extensão Swoole, juntamente com MySQL 5.7 como banco de dados. Este ambiente permite a execução de operações de E/S de forma não bloqueante fornecido pelo Swoole, o que possibilita a execução simultânea de múltiplas operações.

3.1.3. Operações em banco de dados

Para cada execução do *script* programou-se as seguintes operações no banco da dados: 14 inserções (*INSERT's* com *payload* de aproximadamente 12,7 KB cada), 175 consultas (*SELECT's* nos dados inseridos), 4 atualizações (*UPDATE's* nos dados já inseridos) e 1 exclusão (*DELETE*). Todas as operações utilizando-se de índices no banco de dados de forma que as operações reflitam a carga típica de processamento e operações de E/S, semelhante ao cenário real.

3.1.4. Modelagem de dados

O experimento utiliza-se de uma tabela denominada "notas_fiscais", modelada para simular as características típicas de uma aplicação de geração de notas fiscais. A tabela possui as seguintes características:

- **Colunas:** 331 colunas, sendo 110 do tipo *string* e 110 do tipo *integer* e 111 do tipo *double*. As colunas de *string* foram configuradas para armazenar dados de tamanho variado, enquanto as colunas de *integer* e *double* armazenam dados numéricos. Essa configuração simula uma carga média de 12,7 KB de dados por operação de inserção (*INSERT*).
- **Chave Primária:** A tabela possui uma chave primária do tipo *integer*, garantindo a unicidade de cada registro.
- **Índices:** além da chave primária ser um índice, acrescentou-se a coluna "numero" a qual é utilizada para as operações de atualização (*UPDATE's*) e obtenção (*SELECT's*).

3.1.5. Persistência de arquivos

Durante o processo de geração de cada nota fiscal, são gerados e gravados dois arquivos, simulando a persistência da DANF-e (Documento Auxiliar da Nota Fiscal Eletrônica) em formato PDF, e do arquivo XML da NF-e. Os arquivos possuem um tamanho de 140 KB e 50 KB, respectivamente. Esses arquivos são gravados no sistema de arquivos do *container* do Docker, reproduzindo a segunda operação de E/S.

3.1.6. Configuração do *hardware* e ferramentas de teste

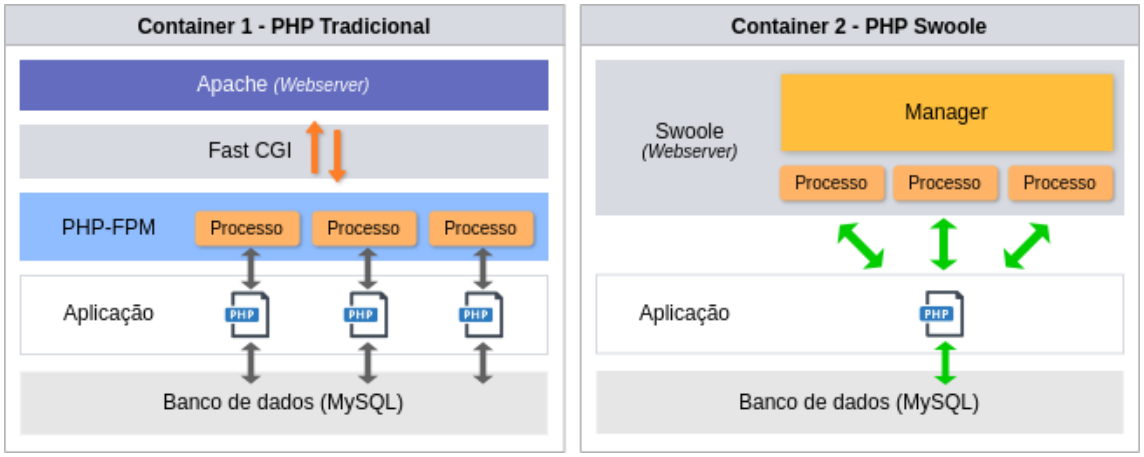

Figura 4. Diagrama da estrutura interna dos containers, comparando o fluxo de dados entre o PHP-FPM no modo síncrono e o Swoole no modo assíncrono, ilustrando a interação dos componentes do sistema durante o processamento das requisições [autor].

O experimento foi realizado em uma instância da AWS EC2, com sistema operacional Linux Ubuntu 24.04, que possui um processador AMD 64 bits com 1 núcleo de 3,3 GHz e 1 GB de memória RAM. Na Figura 4 tem-se o diagrama ilustrando os *containers* do Docker estruturados. As requisições foram disparadas a partir de um computador local com processador Intel i5 8265 Quad-core de 3,90 GHz, GPU integrada (Intel UHD *Graphics* 620, com até 1.100 MHz), 8 GB de memória RAM, armazenamento SSD e Sistema operacional Linux Mint 21.2. A conexão com a internet do computador local com banda larga de 200 Mbps. Para a execução dos testes de carga, foi utilizado o *software* Apache JMeter, configurado para gerar requisições controladas e permitindo a coleta de métricas para cada requisição. Essas métricas foram posteriormente analisadas para avaliar a performance comparativa entre os dois modelos.

O banco de dados foi hospedado em uma instância AWS RDS, configurada com 8 vCPUs, 64 GiB de memória e largura de banda de 4750 Mbps, proporcionando um ambiente isolado e de alta performance para as operações de banco de dados durante os testes.

3.1.7. Análise estatística

Para avaliar o desempenho e executar uma análise comparativa dos modelos síncrono e assíncrono foram aplicados métodos estatísticos utilizando o R versão 4.4.2 [R Core Team 2024], que permitiram verificar as diferenças no *throughput* e a significância dos resultados obtidos nos testes de carga. Para isso, utilizou-se da análise de variância (ANOVA), que possibilitou a observação dos efeitos do modo de operação e do nível de carga sobre o desempenho do sistema. Além disso, foi verificada a interação entre os fatores para compreender como a carga influencia os resultados em cada cenário. O Teste T foi utilizado para comparar as médias de *throughput* entre os dois modelos de operação em cenários específicos, complementando a análise realizada

pela ANOVA. Os dados foram organizados em tabelas e gráficos – gerados com o *ggpubr* [Kassambara 2023] – para facilitar a interpretação dos resultados.

4. Resultados

O experimento iniciou-se com a determinação do melhor *throughput* para cada combinação de modo de operação (síncrono e assíncrono) e carga aplicada (média e alta). Assim, identificou-se os cenários de maior valor de *throughput* possível quando não houvesse falhas em nenhuma das requisições. Essa abordagem permitiu determinar as condições de desempenho máximo para cada cenário, a fim de garantir-se a estabilidade e eficiência dos resultados. O Gráfico 1 apresenta os valores de *throughput* alcançados em cada cenário – carga alta à esquerda do gráfico e carga média à direita – onde os pontos azuis representam o modo assíncrono e os pontos amarelos o modo síncrono.

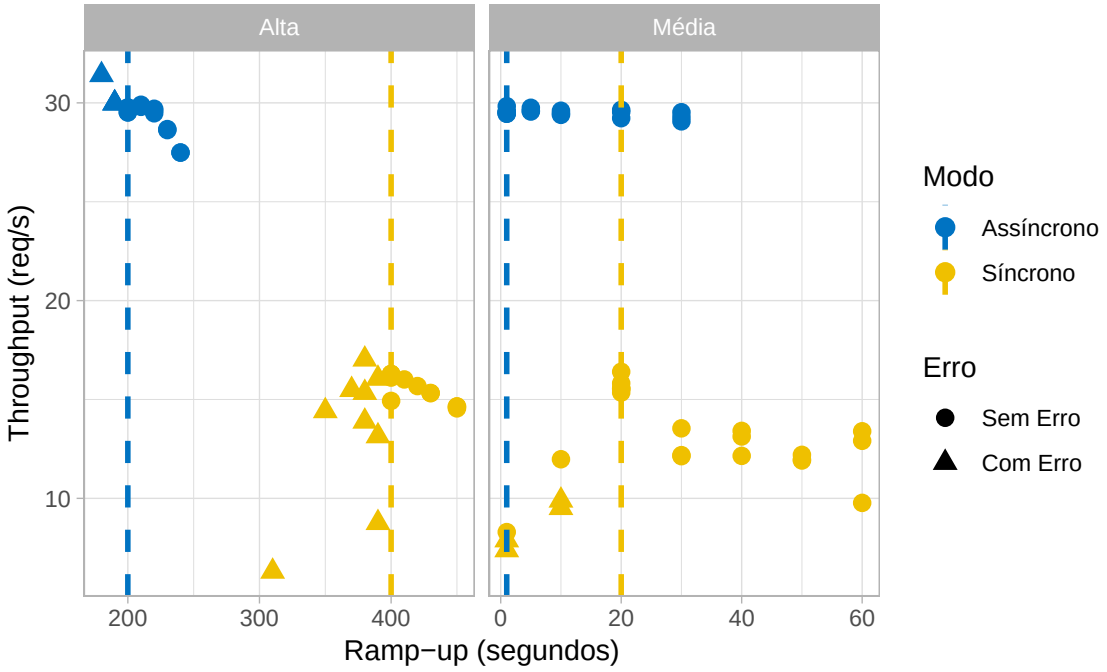

Gráfico 1. *Throughput* vs. *Ramp-up* para os cenários de carga (média e alta) e modo de operação (síncrono e assíncrono). As linhas verticais indicam o *Ramp-up* de desempenho máximo observado sem erros [autor].

As requisições que retornaram erro estão representadas pelos triângulos em suas respectivas cores e cenário de carga. As linhas verticais tracejadas identificam as condições que proporcionaram o melhor desempenho para cada modo e nível de carga. Essa etapa inicial do experimento foi fundamental para estabelecer um ponto de referência claro sobre o desempenho, direcionando a análise subsequente dos resultados e permitindo uma avaliação comparativa mais precisa entre os modos de operação.

4.1. Aplicação de métodos estatísticos

A fim de aprofundar a análise, aplicaram-se métodos estatísticos para efetuar uma análise comparativa entre os cenários de melhor desempenho de carga e modo.

4.1.1. ANOVA

Na Tabela 1 estão os dados resultantes do método ANOVA (Análise de Variância) executado com o objetivo de avaliar como o modo de operação (síncrono e assíncrono) e o nível de carga (média e alta) afetam o desempenho do sistema, medido pelo *throughput*.

Tabela 1. Análise de variância (ANOVA) para os efeitos de Modo, Carga e interação Modo:Carga.

Fonte de variação	Soma dos quadrados	Graus de liberdade	Quadrado médio	Valor de F	Valor de p (Pr(>F))
Modo	1135	1	1135	9765,464	<2e-16
Carga	0,3	1	0,3	2,363	0,140
Modo:Carga	0,1	1	0,1	0,555	0,465
Resíduo	2,3	20	0,1		

O fator modo demonstra uma diferença significativa no *throughput*, com um valor de F muito elevado (9765,464) e um valor de P inferior a 2e-16, indicando que as variações no modo de operação geraram diferenças substanciais no desempenho. Esse resultado evidencia que o modo de operação é a principal variável explicativa para as mudanças no desempenho observadas, com o modo assíncrono mostrando um desempenho superior em relação ao síncrono.

Por outro lado, o fator carga apresentou um valor P de 0,140, não atingindo um critério significativo para determinação de efeito sobre o resultado. Desta forma, pode-se afirmar que o modo de operação não foi influenciado pela carga aplicada, o que reforça que o modo de operação é o principal fator para a otimização de desempenho, podendo-se desconsiderar o fator carga.

Por fim, o resíduo, que refere-se ao erro não explicado pelo método ANOVA, são representados pela soma de quadrados residual de 2,3, com um quadrado médio de 0,1. Esses valores indicam que uma parte da variação do *throughput* não tem relação com fatores analisados, o que é esperado em qualquer experimento devido à variabilidade natural dos sistemas.

4.1.2. Teste T

Aplicou-se o Teste T para comparar as médias de *throughput* entre dois cenários específicos, com o objetivo de verificar se as diferenças observadas são estatisticamente significativas. O Gráfico 2 forneceu a base visual para essa comparação, e os resultados mostraram que a diferença entre os cenários foi de fato significativa, confirmando as conclusões obtidas pela ANOVA.

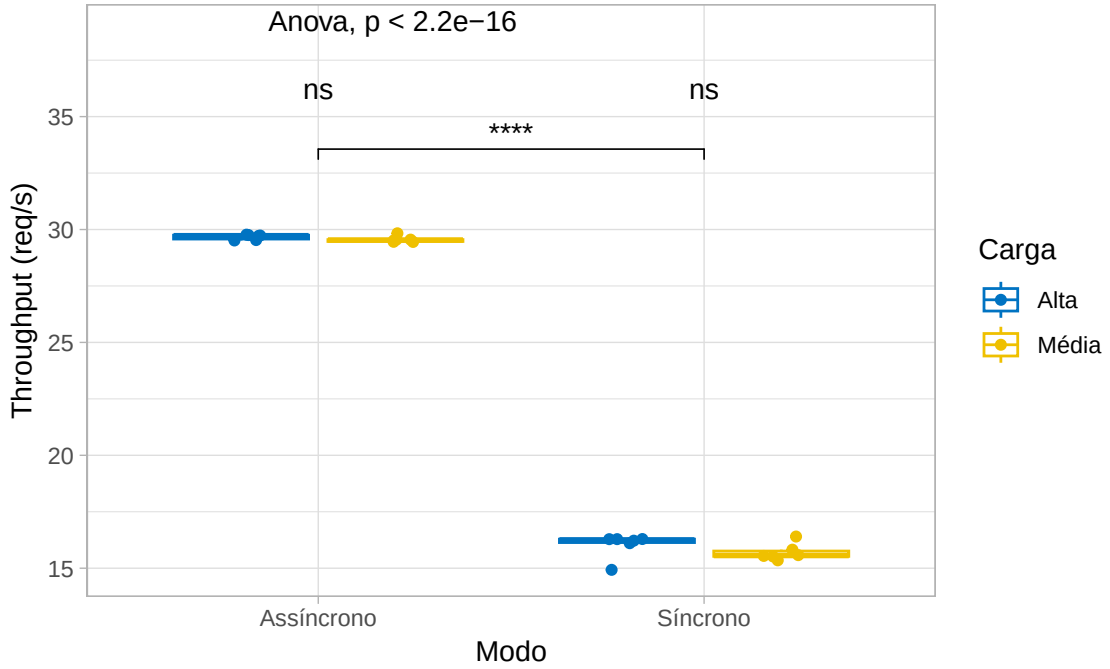

Gráfico 2. Comparação do *throughput* entre os modos de operação (síncrono e assíncrono) em diferentes cenários de carga [autor].

Com base nos resultados obtidos através dos métodos estatísticos (ANOVA e Teste T) no experimento realizado, foi possível confirmar que o modo de operação é o principal fator determinante para o desempenho do sistema em termos de *throughput*. Os efeitos do nível de carga e da interação entre os fatores não apresentaram significância a nível estatístico, o que reforça que o modo de operação por si só já garante um melhor desempenho. Sendo assim, o modo assíncrono se mostrou significativamente mais eficiente que o modo síncrono no contexto deste estudo, tornando-se a escolha ideal para sistemas que lidam com grandes volumes de dados e requisições simultâneas, como no caso da geração de notas fiscais.

Os resultados do experimento confirmaram que a implementação de programação assíncrona em processos que lidam com entrada e saída podem otimizar o desempenho de sistemas, proporcionando maiores taxas de *throughput* mesmo em cenários de alta carga e múltiplas requisições simultâneas, o que favorece a escalabilidade e a eficiência de sistemas *web*.

5. Discussão

A análise comparativa dos modos assíncrono e síncrono em PHP envolve diversos fatores técnicos que podem justificar a superioridade do modo assíncrono, especialmente no *design* de sistemas de alto desempenho. A programação assíncrona permite operações não bloqueantes, possibilitando ao sistema lidar com múltiplas requisições simultaneamente sem a necessidade de aguardar a conclusão de cada operação antes de iniciar a próxima. Essa característica é particularmente vantajosa em cenários como API's *REST* e processamento de eventos, onde o alto *throughput* é essencial. Por exemplo, estudos demonstram que o Node.js, que utiliza operações de E/S assíncronas, consistente-

mente supera o PHP tradicional no tratamento de requisições concorrentes, alcançando até 100% de *throughput* em comparação com 48,70% do PHP sob condições semelhantes [Prayogi et al. 2020, Agus Pratama 2023]. Essa eficiência é ainda mais acentuada quando os tempos de *Ramp-up* são otimizados, reduzindo a probabilidade de erros em cenários de alta carga [Prayogi et al. 2020].

Além disso, o *design* arquitetural de sistemas de alto desempenho pode se beneficiar significativamente da adoção de estratégias assíncronas. API's assíncronas permitem maior escalabilidade e responsividade, aspectos críticos em aplicações *web* modernas que demandam processamento de dados em tempo real e interação contínua. Por exemplo, em implementações de API's *RESTful*, a capacidade de processar múltiplas requisições de forma concorrente, sem bloquear a *thread* principal, pode resultar em experiências de usuário aprimoradas e redução de latência [Gokhale et al. 2021]. Isso é particularmente relevante em aplicações que envolvem tarefas em lote e arquiteturas orientadas a eventos, onde a eficiência na utilização de recursos é primordial [Yatini 2023, Marii e Zholubak 2022].

Entretanto, a transição para um modelo assíncrono em larga escala apresenta desafios arquiteturais e de depuração. Um dos principais problemas está na complexidade de gerenciar fluxos de trabalho assíncronos, que podem dificultar o processo de depuração e o rastreamento do fluxo de dados no sistema [Gokhale et al. 2021]. A programação assíncrona frequentemente resulta no fenômeno conhecido como "*callback hell*", onde *callbacks* aninhados tornam o código difícil de ler e manter. Assim, o tratamento de erros se torna mais complexo, já que erros podem não se propagar da mesma maneira que em códigos síncronos, potencialmente gerando exceções não tratadas que podem degradar o desempenho do sistema [Gokhale et al. 2021]. Por fim, garantir que o sistema permaneça performático sob alta carga requer considerações cuidadosas de gerenciamento de recursos e técnicas de otimização [Prayogi et al. 2020, Agus Pratama 2023].

6. Considerações finais

O estudo teve como objetivo principal a comparação dos modos síncrono e assíncrono em PHP, para isso utilizou-se como base um cenário real de geração de notas fiscais, visto que este recebe um alto volume requisições. Foi possível validar que o modo assíncrono (PHP com Swoole) apresentou uma performance significativamente melhor ao analisar-se o *throughput*.

Os resultados reforçam que o modelo assíncrono é uma alternativa viável e eficiente para sistemas onde a demanda por alto desempenho e operações de E/S intensivas são requisitos críticos. A análise estatística, como o Teste T e ANOVA, comprovou que o modo de operação (síncrono ou assíncrono) foi o fator mais relevante na diferença de desempenho observada. O experimento também apontou que o nível de carga não influenciou de forma significativa os resultados neste experimento.

Algumas limitações deste estudo ficarão como sugestões para trabalhos futuros, como por exemplo considerar o consumo de CPU e memória como uma métrica relevante para a análise, a fim de oferecer uma visão mais ampla sobre os impactos das abordagens testadas. Além disso, sugere-se a utilização de ambientes mais realistas para realização do experimento, como sistemas distribuídos com multiplos nós. Outra possibilidade seria explorar demais linguagens que suportam processamento assíncrono, como Node.js e

GoLang, analisando não só o desempenho, mas também aspectos como escalabilidade e facilidade de manutenção.

De forma geral, o trabalho confirmou que o modelo assíncrono traz benefícios importantes, consolidando o PHP com Swoole como uma opção competitiva para sistemas de alta demanda.

Referências

Agus Pratama, I. P. (2023). Node.js performance benchmarking and analysis at virtualbox, docker, and podman environment using node-bench method. *Joiv International Journal on Informatics Visualization*.

Chigurov, M., Kulikova, V., e Krylova, E. (2024). Virtual threads in java 19 and spring boot 3: Enhancing performance and efficiency of applications. *Vestnik of M. Kozybayev North Kazakhstan University*.

Dou, B. (2020). *Mastering Swoole PHP: Build High Performance Concurrent System with Async and Coroutines*. Transfon Limited.

Gokhale, S., Turcotte, A., e Tip, F. (2021). Automatic migration from synchronous to asynchronous javascript apis. *Proceedings of the Acm on Programming Languages*.

Heng, A. L. K. (2015). *Apache, PHP-FPM & Nginx*. Practical Guide Series Book, 1ª edition.

Kassambara, A. (2023). *ggpubr: 'ggplot2' Based Publication Ready Plots*. R package version 0.6.0.

Liu, X., Sha, L., Diao, Y., Froehlich, S., e nand Sujay Parekn, J. L. H. (2023). Online response time optimization of apache web server. *teste*.

Marii, B. e Zholubak, I. (2022). Features of development and analysis of rest systems. *Advances in Cyber-Physical Systems*.

PHP (2024). Documentação oficial. `https://www.php.net/docs.php`. [Online: acesso em 19-Julho-2024].

Prayogi, A. A., Niswar, M., Indrabayu, e Rijal, M. R. (2020). Design and implementation of rest api for academic information system. *Iop Conference Series Materials Science and Engineering*.

R Core Team (2024). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rimma Zaripova, A. M. e Perukhin, M. (2023). Integrating parallelism and asynchrony for high-performance software development. *International Scientific Conference on Biotechnology and Food Technology*.

Yang, J., Shan, Z., e Chen, Z. (2018). Research and practice of swoole asynchronous multithreading design method. *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*.

Yatini, I. (2023). Performa microframework php pada rest api menggunakan metode load testing. *Fahma*.