Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Benchmark de runtimes Javascript

Pedro de Melo Milet

Trabalho de Graduação

Recife - PE
Julho/2024

Universidade Federal de Pernambuco

Centro de Informática

Pedro de Melo Milet

Benchmark de runtimes Javascript

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de bacharel em Ciência da Computação.

Orientador: *Kiev Santos Gama*

Recife - PE
Julho/2024

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Milet, Pedro de Melo.
Benchmark de runtimes javascript / Pedro de Melo Milet. - Recife, 2024.
27 p. : il., tab.

Orientador(a): Kiev Santos da Gama
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2024.

1. Benchmark. 2. JavaScript. 3. Nodejs. 4. Bun. 5. Deno. I. Gama, Kiev Santos da. (Orientação). II. Título.

000 CDD (22.ed.)

Pedro de Melo Milet

Benchmark de runtimes Javascript

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Ciência da Computação.

Aprovado em: 18/07/2024

BANCA EXAMINADORA

_____

Prof. Kiev Santos Gama (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Nelson Souto Rosa (Examinador Interno)

Universidade Federal de Pernambuco

Agradecimentos

Gostaria de expressar minha sincera gratidão à minha família, amigos e à minha companheira, cujo apoio inabalável e encorajamento foram fundamentais nesta jornada. Eles estiveram sempre ao meu lado, fornecendo a força necessária para superar os desafios enfrentados e constituem a razão primordial pela qual mantive minha determinação e não desisti.

"Practice any art, music, singing, dancing, acting, drawing, painting, sculpting, poetry, fiction, essays, reportage, no matter how well or badly, not to get money and fame, but to experience becoming, to find out what is inside you, to make your soul grow"

—KURT VONNEGUT

Resumo

Este estudo apresenta uma comparação de desempenho entre três runtimes JavaScript: Node.js, Deno e Bun. Por meio de testes de carga que simulam cenários do mundo real, foram analisadas as características de desempenho desses runtimes. O estudo focou em *throughput* , latência, número de requisições, uso de CPU e uso de memória sob diferentes cenários de carga de trabalho. Os resultados mostraram diferenças significativas de desempenho entre os runtimes, com Deno e Bun superando Node.js na maioria dos casos. No entanto, à medida que a carga de trabalho aumentava, a diferença de desempenho entre os runtimes diminuía. Deno demonstrou desempenho especialmente forte em todos os cenários, apesar de Bun mostrar maior uso de CPU e memória, particularmente durante operações de inserção no banco de dados.

Palavras-chave: JavaScript, Node.js, Deno, Bun, benchmarking, performance testing

Abstract

This study presents a benchmarking comparison of three JavaScript runtimes: Node.js, Deno, and Bun. Through load tests simulating real-world scenarios, the performance characteristics of these runtimes were analyzed. The study focused on *throughput* , latency, number of requests, CPU usage, and memory usage under different workload scenarios. Results showed significant performance differences among the runtimes, with Deno and Bun outperforming Node.js in most cases. However, as workload increased, the performance gap between the runtimes diminished. Deno demonstrated particularly strong performance across all scenarios, despite Bun showing higher CPU and memory usage, especially during database insertion operations.

Keywords: JavaScript, Node.js, Deno, Bun, benchmarking, performance testing

Sumário

Lista de Figuras

Capítulo 1

Introdução

Em 2009, foi lançado o Node.js [1], criado por Ryan Dahl e baseado na V8 engine, um motor de JavaScript de código aberto desenvolvido pelo Google. A V8 *engine* compila código JavaScript diretamente para código de máquina nativo antes de executá-lo, o que melhora significativamente o desempenho. Isso tornou o Node.js o runtime JavaScript mais amplamente utilizado. Nove anos depois, Ryan Dahl retorna à cena com o Deno [2], um novo runtime JavaScript projetado para corrigir várias falhas que ele identificou na criação do Node, como o problema com o *node_modules* e a utilização da ferramenta GYP [3]. O Deno, também baseado na V8, introduz a capacidade de utilizar módulos escritos em outra linguagem que vem ganhando popularidade nos últimos anos por sua segurança e eficiência: Rust.

Em 2023, surgiu o Bun [4], um novo runtime JavaScript que começou a ganhar popularidade. Desenvolvido na linguagem emergente Zig, o Bun promete ser até quatro vezes mais rápido do que projetos escritos em Node.js [4]. Além disso, ele incorpora um gerenciador de pacotes integrado, visando solucionar os problemas de velocidade na instalação de dependências. Anteriormente, esse problema era abordado por ferramentas como pnpm [5] e yarn [6], que conseguiram melhorias em velocidade. No entanto, o gerenciador de pacotes do Bun consegue superá-los, sendo até 17 vezes mais rápido que o pnpm e 33 vezes mais rápido que o yarn [4].

Dado o cenário atual, com a presença de vários runtimes, este trabalho tem os seguintes:

Objetivos:

- Objetivo geral: Realizar uma análise comparativa de desempenho entre os runtimes JavaScript Node.js, Deno e Bun, através de testes de carga simulando cenários de uso reais, visando identificar suas características de desempenho em diferentes situações.

- Objetivo específico: Configurar ambientes de teste para cada um dos runtimes Node.js, Deno e Bun.

- Objetivo específico: Desenvolver aplicações de teste idênticas em JavaScript para cada runtime, contemplando diferentes tipos de operações, como resposta estática, cálculo intensivo e interação com banco de dados.

- Objetivo específico: Realizar testes de carga utilizando a ferramenta Autocannon, simulando diferentes níveis de carga em cada aplicação.

- Objetivo específico: Coletar e analisar métricas de desempenho, incluindo *throughput*, latência, número de requisições, uso de CPU e memória, para cada runtime e tipo de operação.
- Objetivo específico: Comparar os resultados obtidos para identificar diferenças significativas de desempenho entre os runtimes em diferentes cenários de uso.

Capítulo 2

Fundamentação

2.1 Trabalhos Relacionados

Na pesquisa realizada em 2021, Hardeep Kaur Dhalla [7] realiza um benchmarking para comparar duas APIs RESTful: uma escrita em Java com o framework Spring Boot e outra em C# com o framework .NET Core .Ele detalha três tipos de testes usados para avaliar as APIs: teste de carga, teste de estresse e teste de capacidade. Por fim, opta por seguir com o teste de carga, destacando métricas a serem coletadas, como tempo de resposta, uso de CPU e memória, e número de requisições realizadas em um determinado período.

No entanto, há algumas limitações que devem ser consideradas. Primeiramente, o ambiente de teste utilizado foi restrito, com um conjunto específico de máquinas com configurações particulares de hardware e software, o que pode não refletir todas as condições reais de produção, especialmente em ambientes com configurações variadas. Além disso, as implementações utilizaram configurações padrão para as plataformas Spring Boot e MS.NET Core, sem explorar ajustes específicos de configuração e otimizações que poderiam melhorar o desempenho, como o uso de *connections pools* as quais permitem o Spring Boot reutilizar conexões. Ademais, a pesquisa se concentrou apenas nas operações básicas de CRUD (Create, Read, Update, Delete), o que pode não representar completamente o desempenho das aplicações em situações reais, onde operações mais complexas e diferentes padrões de uso são comuns.

Um estudo anterior, conduzido em 2014 por Kai Lei, Yining Ma e Zhi Tan [8], também menciona o tempo de resposta, número de requisições, além do tempo médio por requisição e throughput. Os autores descrevem cenários comuns ao desenvolvimento web, que auxiliam no projeto. Neste estudo, são utilizados três cenários: um simples "Hello World", que simula um cenário comum de servir uma resposta estática; o cálculo de Fibonacci, que simula um cálculo intensivo; e a realização de uma query em um banco de dados, caso comum em aplicações web. Além disso, o experimento varia apenas o número de usuários simulados utilizando o sistema.

No entanto, o número de usuários utilizados, com um pico de 500 usuários, é muito pouco expressivo, considerando que sistemas podem ter centenas de milhares ou até milhões de usuários se conectando simultaneamente. Além disso, o estudo testou apenas uma configuração de hardware, o que pode não refletir a diversidade de ambientes de produção. O estudo também não testou um banco de dados NoSQL, limitando a análise a uma única abordagem de gerenciamento de dados. Ademais, não foram consideradas diferentes arquiteturas para distribuição de carga, que são cruciais para entender como

as aplicações se comportariam em cenários de alta demanda e escalabilidade horizontal.

D. Demashov e I. Gosudarev [9] realizaram um trabalho com configuração muito similar ao proposto aqui. Nele, uma aplicação template é reimplementada em diversos frameworks JavaScript. Cada implementação é subsequentemente testada com testes de carga utilizando a ferramenta Autocannon. Dessa forma, dados de latência, número de requests e throughput foram colhidos diretamente dos logs da ferramenta. Com esses dados, os autores puderam tirar diversas conclusões sobre o comportamento de cada um dos frameworks.

Entretanto, este trabalho possui limitações comuns a diversos outros trabalhos de mesma natureza, como o limite de configurações do ambiente de teste. O hardware utilizado não varia entre os testes, limitando-se a um único conjunto de especificações. Além disso, o estudo se restringe a dois cenários comuns em ambientes de servidor web: operações CRUD e servir arquivos estáticos. Essas limitações podem não refletir todas as condições reais de produção, especialmente em ambientes com configurações variadas e cargas mais complexas.

2.2 Problemática e investigação

O runtime Node.js foi criado em 2009 por Ryan Dahl [1] e ao longo dos anos tem evoluído, enfrentando importantes decisões para serem adicionadas em cima dessa antiga fundação. Tais decisões levaram a problemas que hoje são abordados pelo Deno e Bun. Ao analisarmos o que esses dois runtimes visam corrigir do Node.js, podemos apontar melhor as problemáticas desse runtime.

Principais problemas de *design* que o Node.js apresenta e como o Deno visa corrigi-los:

- Promises vs Callbacks: em junho de 2009, as promises foram adicionadas ao Node.js, mas foram removidas em 2010. Posteriormente, foram reintroduzidas em 2015. Essa inconsistência fez com que diversas APIs que se desenvolveram utilizando versões antigas do Node ficassem presas aos callbacks, enquanto o mercado evolui para promises. Deno foi construído desde o início apenas com promises.

- Segurança: O Node oferece acesso completo à máquina para o programa, enquanto o Deno trabalha com um sistema de permissões.

- Build System (GYP): GYP é uma ferramenta de automação de compilação criada em 2011 e introduzida no engine V8 da Google. O Node.js então adotou o GYP, que foi rapidamente depreciado pela Google em 2016 e substituído pelo GN. No entanto, o Node.js ficou preso ao GYP.

- Modules: Node.js opera com um sistema de módulos, no qual é necessário definir um ‘package.json‘ que especifica quais módulos fazem parte do software. Além disso, é preciso declarar os módulos em uso através do método ‘require()‘. Por outro lado, Deno considera essa abordagem uma complexidade desnecessária e simplifica

a gestão de módulos. No Deno, basta usar a instrução ‘import‘ com a URL do módulo nos arquivos que precisam desse módulo. Esses módulos são automaticamente baixados e armazenados em cache durante a execução do programa.

- Typescript: TypeScript é uma linguagem de programação que atua como um superset de JavaScript, introduzindo tipagem estática, o que permite aos desenvolvedores escreverem códigos mais seguros e mais fáceis de manter. Diferentemente do Node.js, que não possui suporte nativo para TypeScript, exigindo que o código seja compilado para JavaScript antes de ser executado, Deno incorpora um compilador de TypeScript.

Além das questões já discutidas, um aspecto particularmente interessante abordado neste trabalho é a performance. Enquanto o Node.js é majoritariamente implementado em JavaScript, o Deno é escrito em Rust, uma linguagem segura e com ótimo desempenho. Adicionalmente, o Node.js utiliza uma versão mais antiga da engine V8 do Google, ao passo que o Deno aproveita a versão mais recente dessa engine, beneficiando-se das últimas melhorias em termos de performance.

Assim como o Deno, Bun ataca diversos dos mesmos problemas:

- Typescript: Bun possui um transpilador embutido para TypeScript, permitindo que o TypeScript seja executado diretamente na engine sem a necessidade de ser previamente transpilado para JavaScript.

- Modules: Ao contrário do Deno, que visa eliminar o uso de ‘package.json‘ e o diretório ‘node_modules‘, o Bun apresenta seu próprio gerenciador de pacotes, com o objetivo de substituir o npm, o gerenciador padrão do Node.js. O Bun opera de forma semelhante ao npm, porém com melhorias em velocidade e eficiência.

Uma diferença fundamental entre Deno, Bun e Node.js, especialmente relevante para a questão de performance, é que, ao contrário do Node.js e do Deno, que utilizam a engine V8 do Google, o Bun opta pela engine JavaScriptCore, parte do Apple WebKit do navegador Safari. Além disso, assim como o Deno substitui implementações em JavaScript por Rust para ganhar em segurança e performance, o Bun é implementado em Zig, outra linguagem conhecida por sua eficiência e segurança.

Capítulo 3

Metodologia

Esta seção detalhará a análise de desempenho realizada, seguindo as etapas sugeridas por Jain [10]. O estudo tem como objetivo comparar o desempenho dos runtimes JavaScript Node.js, Deno e Bun, através de testes de carga que simulam cenários de uso reais. Serão avaliadas métricas como throughput, latência, número de requisições, uso de CPU e uso de memória, para identificar as características de performance de cada runtime em diferentes situações.

3.1 Métricas Utilizadas

Para avaliar os runtimes, foram selecionadas as seguintes métricas: throughput, latência (neste trabalho, a latência considerada é relativa ao RTT - round trip time, do início da requisição à resposta do servidor) e número de requests por segundo. A ferramenta Autocannon foi escolhida tanto para realizar as requisições quanto para coletar automaticamente os dados após cada teste. A Autocannon é uma das ferramentas mais utilizadas para testes dessa natureza [9], sendo utilizada em diversos trabalhos acadêmicos [11] [12], realizando testes de carga. Além dessas métricas, o uso de CPU e memória também servirá para a avaliação dos runtimes, e esses dados serão coletados por meio da observação do contêiner Docker no qual os servidores de testes serão executados.

3.2 Toolchain

Como auxílio neste projeto, foram utilizadas duas ferramentas principais. A primeira é a Autocannon [13], uma biblioteca JavaScript de benchmarking HTTP amplamente utilizada. A Autocannon foi escolhida por sua facilidade de configuração, permitindo a variação no número de conexões/usuários, número de threads e tipos de request, além de ser escrita em JavaScript. Essa biblioteca não só realiza as requisições HTTP, mas também metrifica cada uma delas, fornecendo logs detalhados sobre latência, número de requisições e throughput.

A segunda ferramenta utilizada é o Docker [14], uma tecnologia de containerização que facilita a replicação do projeto com um único comando, configurando automaticamente o ambiente dentro do contêiner. O Docker também simplifica o monitoramento dos recursos consumidos pelo processo, já que cada contêiner mapeia diretamente para o processo executado. Com o comando docker stats, é possível coletar facilmente dados relevantes sobre o uso de CPU e memória, garantindo um monitoramento eficiente dos

recursos durante a execução dos testes.

3.3 Carga de Trabalho

Existem três tipos principais de testes que podem ser realizados no cenário proposto neste trabalho: teste de carga, teste de estresse e teste de capacidade. O teste escolhido foi o teste de carga. D. Demashov e I. Gosudarev [9] argumentam que o teste de carga é a maneira mais reprodutível de se testar e determinar a performance de uma aplicação, enquanto os outros dois tipos, teste de estresse e teste de capacidade, se aplicam em casos onde estamos testando tolerância a falhas ao invés de performance.

3.4 Design do Experimento

Foram desenvolvidas três aplicações idênticas em JavaScript, destinadas a serem executadas em cada um dos três runtimes observados, todas contendo os mesmos três endpoints. O primeiro endpoint, "/", entrega um JSON com a mensagem "Hello world", simulando um cenário de pagina estática. O segundo, "/fibonacci", calcula os primeiros 100 números da sequência de Fibonacci, demonstrando um caso de uso que exige alto processamento pelo servidor. O terceiro e último endpoint, "/user", executa uma operação de inserção em um banco de dados SQLite3, instanciado no momento da execução, representando uma interação típica com banco de dados.

Foram estabelecidos dois cenários distintos, caracterizados pelo número de conexões simultâneas:10 e 100, visando simular a presença de diversos usuários. Em cada cenário, vários requests do tipo GET foram enviados para um endpoint específico por runtime, seguindo a sequência "/", "/fibonacci"e "/user". Cada teste foi conduzido por um período determinado de 2 minutos, permitindo uma análise detalhada do comportamento do sistema sob diferentes níveis de carga.

Neste trabalho, os parâmetros incluem a CPU "Intel© Core™ i5-9300H CPU @ 2.40GHz × 4", 32 GB de memória RAM com frequência de 3200MHz, e os runtimes Deno V1.41.3, Bun V1 e Node V21.6.1. Já os fatores são o throughput, latência, número de requisições, uso da CPU e uso de memória RAM. Além disso, a ferramenta Autocannon será utilizada juntamente com as rotas previamente descritas, sendo o único parâmetro a ser manipulado o número de conexões concorrentes.

Docker [14] foi empregado como suporte no projeto, facilitando a gestão do ambiente de execução. Os dados referentes ao hardware, incluindo uso de memória e CPU, foram coletados utilizando a ferramenta integrada do Docker, docker stats. Por outro lado, métricas como throughput, número de requests e tempo de resposta foram automaticamente fornecidas ao término dos testes pela ferramenta Autocannon, oferecendo uma visão abrangente do desempenho das aplicações sob teste.

3.5 Limitações

As configurações e o número de testes de carga utilizados neste trabalho representam uma limitação significativa. Cenários de computação mais complexa e intensa, como a resolução do problema do caixeiro-viajante, poderiam fornecer insights valiosos sobre o comportamento de cada runtime sob cargas de trabalho complexas. Além disso, a diversidade dos bancos de dados poderia ser aumentada, testando não apenas diferentes bancos SQL, mas também bancos NoSQL de variadas arquiteturas. O tempo disponível para a realização deste trabalho também é uma limitação importante, pois cada variável adicionada aumentaria o número de cenários possíveis a serem testados dobrando a cada variável adicionada.

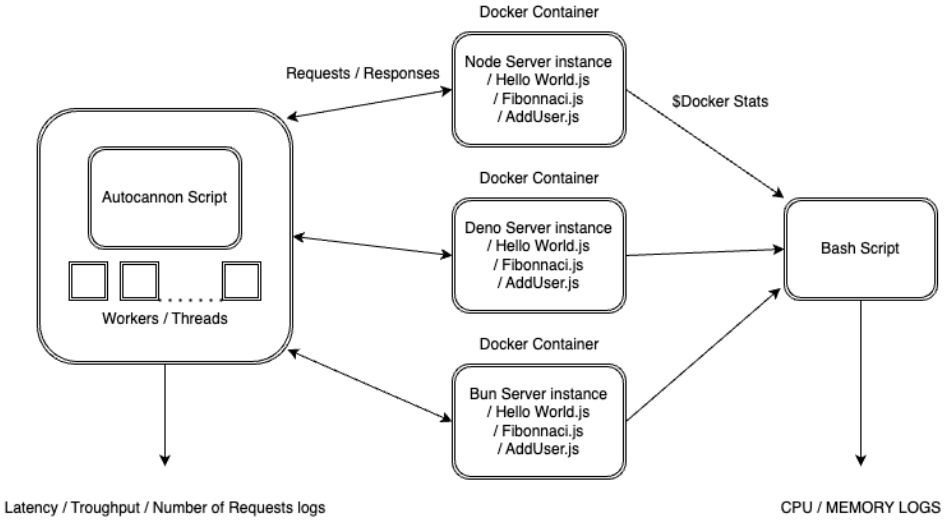

Figura 3.1 Arquitetura do experimento.

Capítulo 4

Análise dos resultados

4.1 Hello World

A rota "hello world"é a mais simples de ser testada. Cada servidor responde apenas com um JSON contendo "hello world"após um request GET. Mesmo sendo simples, pode-se observar na Figura 4.1 uma diferença significativa no número de requests por segundo realizados pelos runtimes Deno e Bun em relação ao Node. O Deno realiza, em média, 3.2 vezes mais requests, enquanto o Bun alcança 5.3 vezes mais requests, em média. Essa diferença de desempenho também é evidente ao analisarmos a latência no pior caso. Para o Node, a latência ficou em 39 ms, o que representa um aumento de 70% em relação ao pior caso do Bun e 50% em relação ao pior caso do Deno. Podemos observar um desvio padrão significativamente maior para o Deno e o Bun. O desvio padrão do Deno é 3.9 vezes maior que o do Node, enquanto o desvio padrão do Bun é 5 vezes maior. Isso mostra que, apesar de ser mais lento em geral, o Node é mais estável em suas métricas.

```
1 app.get("/", (req, res) => {
2   res.json({ message: "Hello World!" });
3 });
```

Listing 4.1 Hello world Route code

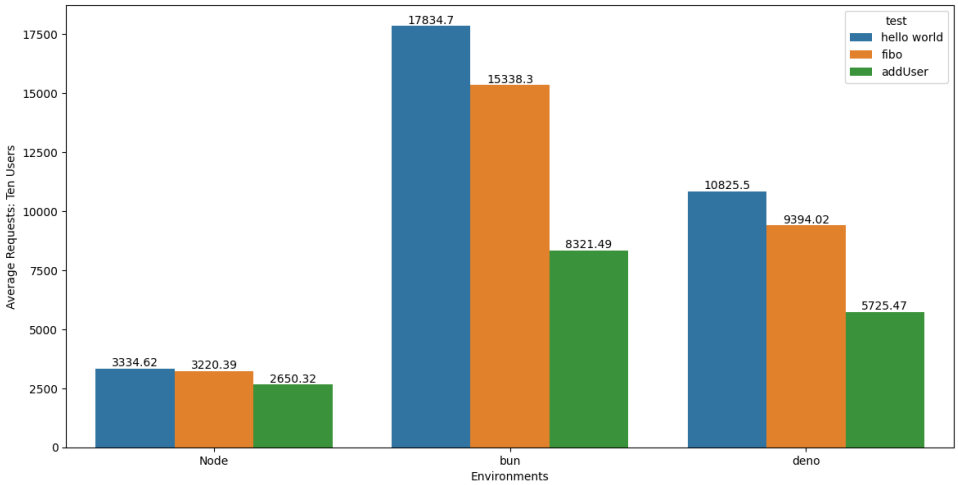

Figura 4.1 Número médio de requests por segundo para 10 usuários

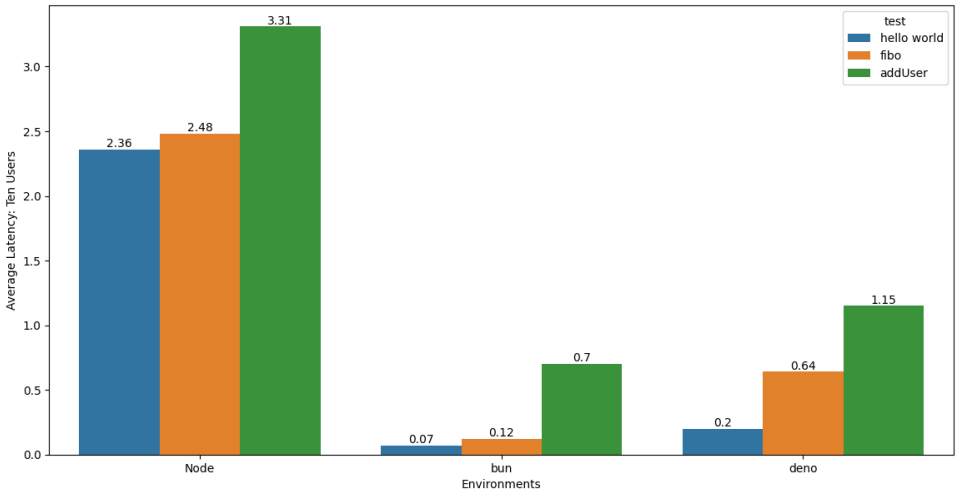

Figura 4.2 Latência media para 10 usuários

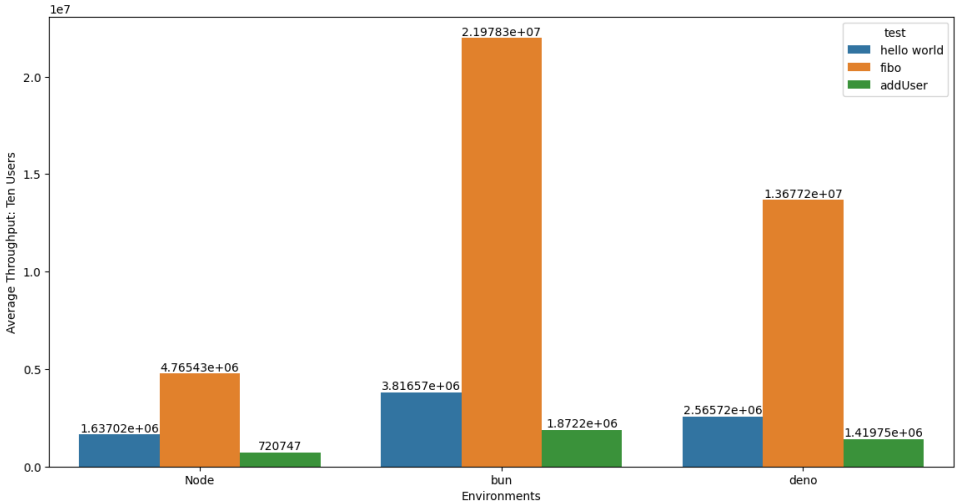

Figura 4.3 Troughput médio para 10 usuários

4.2 Fibonacci

Ainda na Figura 4.1, podemos analisar a rota ‘/fibo‘, que recebe um request GET, calcula os 100 primeiros números de Fibonacci e retorna o resultado. Os resultados são praticamente idênticos em proporção, mas é interessante notar que, enquanto as rotas "hello world"processaram 1.16 vezes mais requests por segundo em média no Bun e 1.5 vezes mais requests por segundo em média no Deno, indicando um processo bem mais custoso, o Node teve uma variação muito menor em sua rota "hello world", tendo em média apenas 1.03 vezes mais requests por segundo.

```
app.get("/fibo", (req, res) => {
  function fibonacci(n) {
    let fibSequence = [0, 1];
    for (let i = 2; i < n; i++) {
      fibSequence.push(fibSequence[i - 1] + fibSequence[i - 2]);
    }
    return fibSequence;
  }
  const fibNumbers = fibonacci(100);

  res.send(fibNumbers.join(', '));
});
```

Listing 4.2 fibonnaci route code

4.3 Adicionar Usuário

Por último, podemos analisar a rota mais custosa para processamento, ‘/addUser‘. Ao receber um request GET, essa rota realiza a inserção de um usuário com ID aleatório em um banco de dados em memória SQLite3. O custo pode ser melhor observado quando comparado com a rota mais rápida, "hello world". No caso do Node, por exemplo, há uma média de 1.25 vezes mais requests por segundo em comparação consigo mesmo, e de forma semelhante, o Deno tem uma média de 1.89 vezes mais requests por segundo, enquanto o Bun tem 2.14 vezes mais requests por segundo em média. Essa disparidade de tempo era esperada e as proporções se mantêm parecidas, com o Bun apresentando o maior número de requests por segundo, seguido pelo Deno e, por último, o Node. As proporções também continuam similares, com o Deno tendo 2.16 vezes mais requests por segundo em média que o Node, e o Bun tendo em média 3.13 vezes mais requests por segundo que o Node. No pior caso, diferente das rotas anteriores, eles estão quase iguais, sendo o pior caso do Deno 33ms, o pior caso do Node 36 ms e o Bun teve seu pior caso em 36 ms.

```
app.get("/addUser", (req, res) => {
  const username = "John";
  const email = 'John@example.com';

  const sql = 'INSERT INTO users (username, email) VALUES (?, ?)';
```

```
6     db.run(sql, [username, email], (err) => {
7       if (err) {
8         console.error('Error adding user:', err);
9         res.status(500).json({ error: 'Failed to add user' });
10      } else {
11        console.log('User added successfully');
12        res.status(200).json({ message: 'User added successfully' });
13      }
14    });
15  });
```

Listing 4.3 addUser route code

4.4 Utilização de CPU

Na Figura 4.4, observamos o uso da CPU com 10 usuários simultâneos. Como os testes foram realizados sequencialmente em cada um dos servidores, podemos observar que ao final de cada ciclo de testes, o uso da CPU cai para 0. Além disso, notamos que todos os servidores utilizaram mais de 100% da CPU, indicando que foram capazes de atingir o máximo de um núcleo e solicitar outro núcleo. O maior custo de CPU foi o custo de inicialização do servidor Node, que necessitou de mais de 150% da CPU, além de ter um custo médio de processamento mais elevado, em torno de 125%, mesmo tendo o *throughput* mais baixo entre os três servidores.

O Bun apresentou um custo médio de 110% para as operações nas rotas de "hello world"e Fibonacci, porém, para a rota de inserção no banco de dados, o uso de CPU foi elevado, com uma média de 135%. Já o Deno manteve constante o seu uso de CPU em 110%.

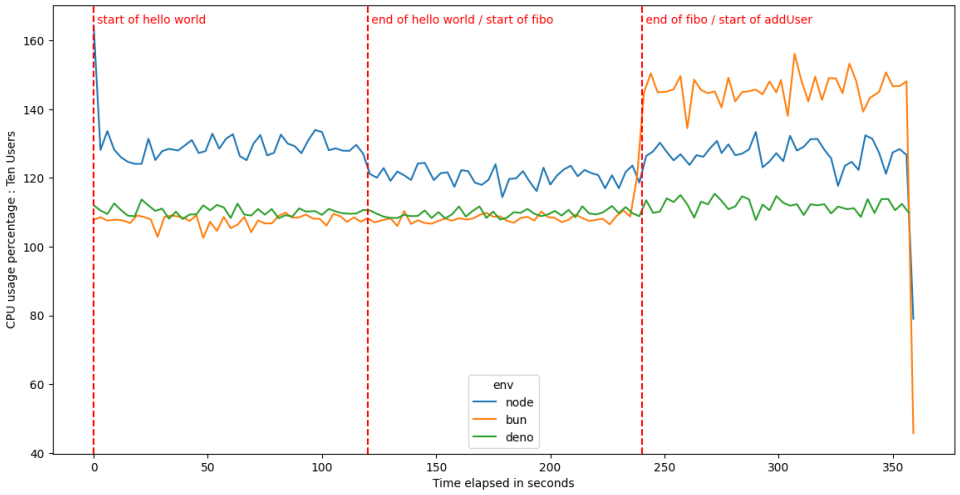

Figura 4.4 Uso de CPU para 10 usuários

4.5 Utilização de memória

Na Figura 4.5, pode-se analisar o custo de memória dos três servidores ao longo dos testes com 10 usuários. Observa-se que o custo de memória é baixo, e todos os servidores demonstram o mesmo padrão de aumento de uso de memória no último terço do experimento, causado pelos *requests* que realizam a inserção no banco de dados. No entanto, é evidente que o custo mais significativo é o do Bun, que utiliza apenas 1% de memória no último terço, e essa memória não é realocada ao sistema após o término da carga de trabalho.

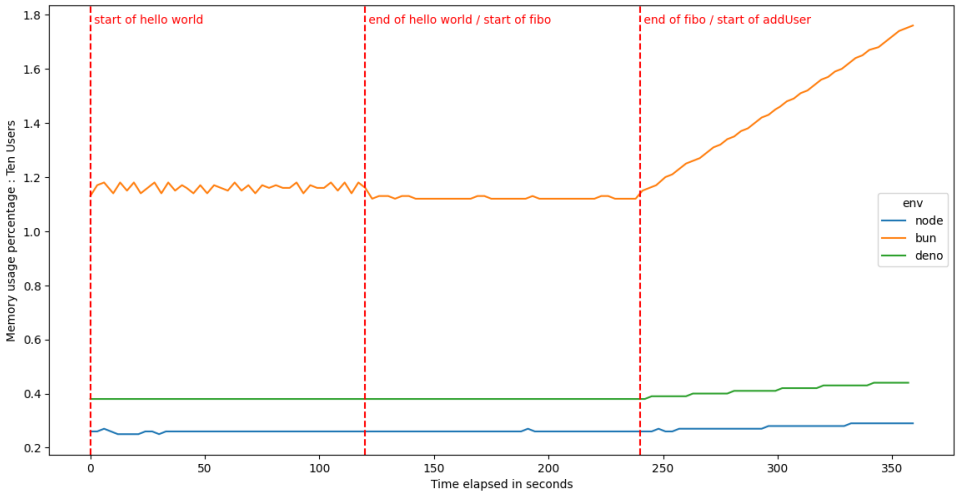

Figura 4.5 Uso de memória para 10 usuários

4.6 Aumentando o numero de usuários

A variação do número de usuários de 10 para 100 resulta nas Figuras 4.6, 4.7, 4.8, 4.9, 4.10, nas quais é possível observar a maior mudança na latência, que, no pior cenário, teve um aumento de 424 vezes na rota "hello world"do Deno. Mesmo com o aumento da latência, o *throughput* e o número de requisições aumentaram.

O Node experimentou um aumento médio na latência de suas rotas em 9 vezes, enquanto o *throughput* médio e o número de requisições aumentaram em 30%.

O Bun registrou um aumento médio de 83 vezes na latência, porém, suas rotas "hello world"e Fibonacci mantiveram aproximadamente o mesmo número de requisições e *throughput*. A rota "addUser"foi a única a se beneficiar com o aumento do número de usuários, apresentando um aumento de 27% no número de requisições e *throughput*.

O, o Deno teve um aumento médio de 170 vezes na sua latência, sem aumento significativo nas rotas "hello world"e Fibonacci, e um aumento de 17% na rota "addUser".

Ao comparar a Figura 4.9 com a Figura 4.4 e a Figura 4.10 com a Figura 4.5, é evidente que a tendência geral se mantém, apresentando um comportamento e utilização dos

recursos de hardware semelhantes. No entanto, é possível destacar uma maior oscilação no uso de memória e CPU no cenário com 10 usuários, observada pela presença de picos mais altos e vales mais profundos.

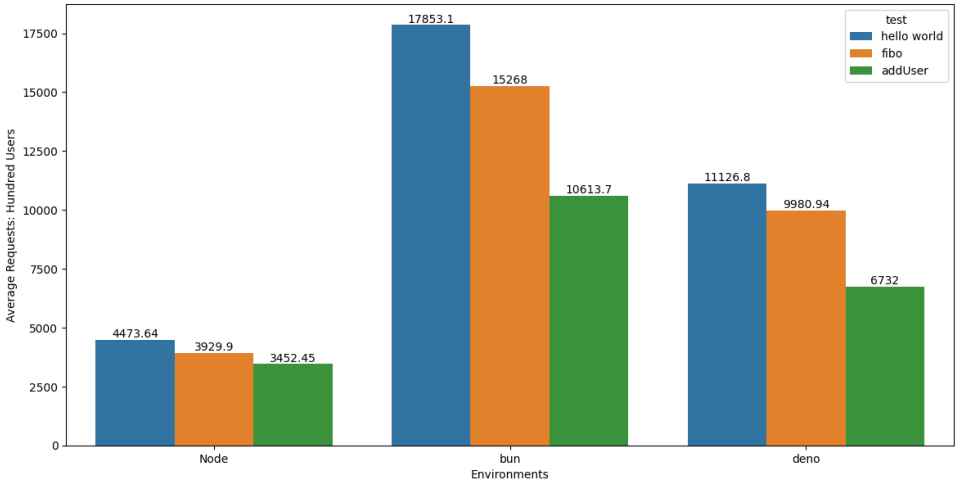

Figura 4.6 Número médio de requests por segundo para cem usuários

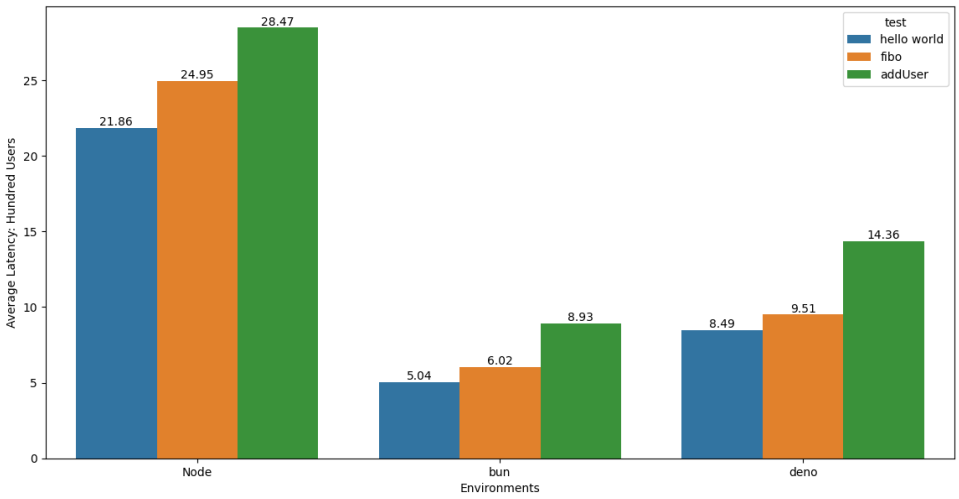

Figura 4.7 Latência media para cem usuários

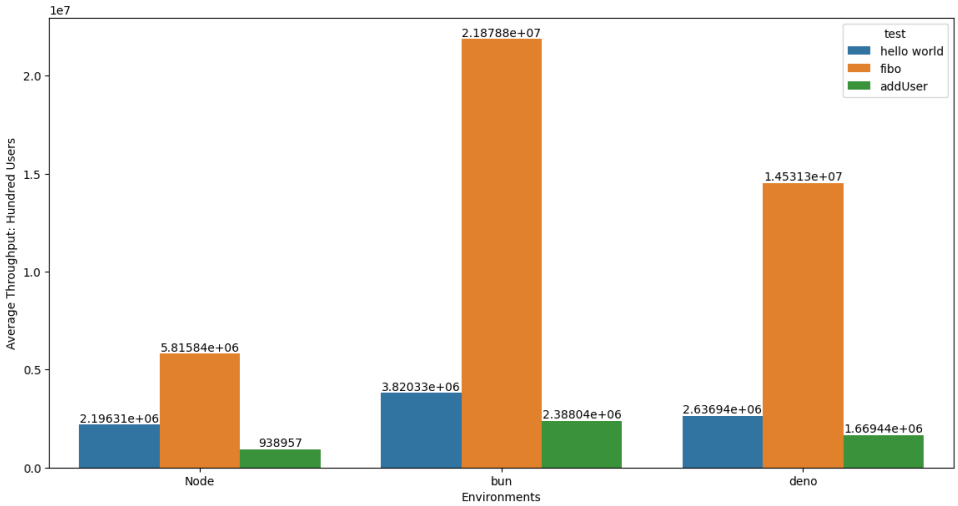

Figura 4.8 *throughput* médio para cem usuários

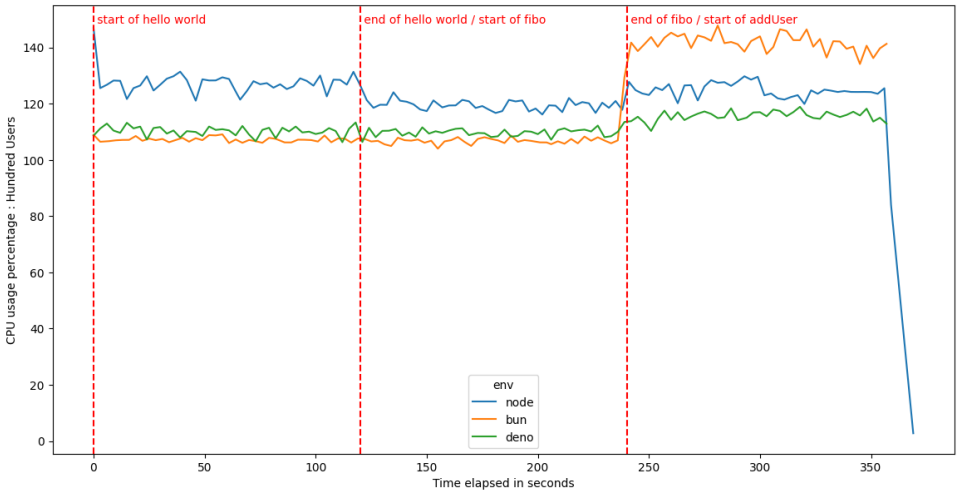

Figura 4.9 Uso de CPU para cem usuários

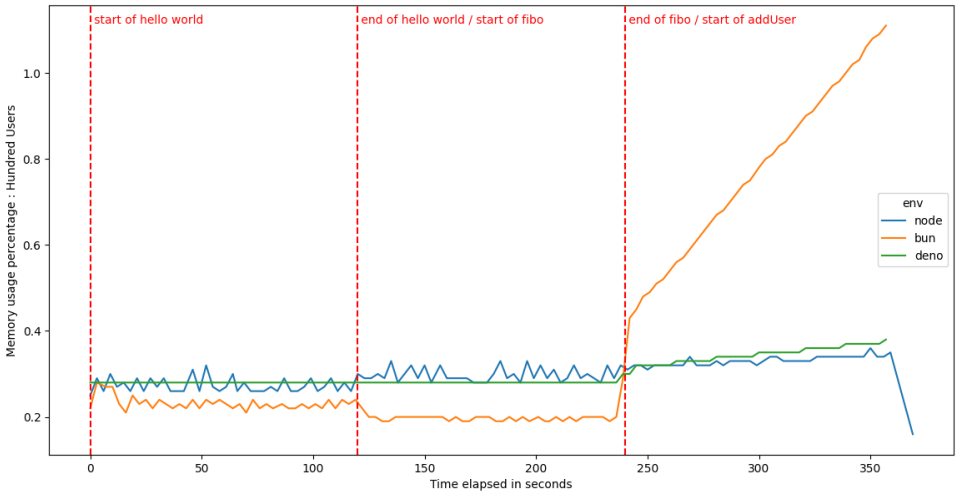

Figura 4.10 Uso de memória para cem usuários

Capítulo 5

Conclusão

Com base nos resultados obtidos nesta pesquisa, é possível concluir que foram alcançados o objetivo geral e os objetivos específicos propostos no trabalho.

Primeiramente, os ambientes de teste foram configurados com sucesso para cada um dos runtimes Node.js, Deno e Bun. Isso permitiu que as aplicações de teste fossem desenvolvidas em JavaScript para cada runtime, contemplando diferentes tipos de operações, como resposta estática, cálculo intensivo e interação com banco de dados. Além disso, os testes de carga foram realizados utilizando a ferramenta Autocannon, simulando diferentes níveis de carga em cada aplicação.

Em relação à coleta e análise das métricas de desempenho, os resultados foram considerados satisfatórios. Foram examinadas métricas essenciais, como *throughput*, latência, número de requisições, utilização de CPU e memória, para cada runtime e tipo de operação. Essas métricas ofereceram insights valiosos sobre o desempenho relativo de cada runtime em diferentes cenários de uso.

Ao comparar os resultados obtidos, pode-se identificar diferenças significativas de desempenho entre os runtimes em diversas situações. Tanto o Bun quanto o Deno demonstraram uma superioridade notável em termos de latência e *throughput*. Entretanto, é importante notar que o Bun apresentou um uso mais intensivo de memória e CPU, especialmente no cenário de inserção de dados no banco de dados.

Ao sobrecarregar o sistema com 10 vezes o número de usuários, observamos um aumento expressivo na latência do Bun e Deno, chegando a aumentos de até 424 vezes no pior caso. Apesar disso, o *throughput* para ambos os casos permaneceu praticamente o mesmo. Por outro lado, o Node teve um aumento proporcional na latência e no *throughput* ao aumentar a carga em 10 vezes.

Os resultados alcançados por este trabalho, ao analisar a performance do Bun vs Node.js vs Deno, são consistentes com outro estudo realizado por Md Feroj Ahmod [15], que também propôs fazer um benchmarking do Bun comparado ao Node.js. Esse estudo encontrou que o Bun foi superior em latência, número de requests e *throughput* em relação ao Node.js. Além disso, nossos resultados estão alinhados com diversas fontes de literatura cinza que apontam que o Deno é superior ao Node.js nas métricas mencionadas, e que o Bun supera o Deno [16].

Pode-se concluir, portanto, que o Bun e o Deno demonstraram melhor desempenho em geral, porém, conforme a carga aumenta, a diferença de desempenho entre os runtimes diminui. O Deno, em particular, se destacou em todos os casos, principalmente considerando o uso mais intensivo de CPU e memória pelo Bun nos cenários de inserção de dados no banco de dados.

Referências Bibliográficas

[1] Node.js. `https://nodejs.org/`. Accessed: 2024-06-17.

[2] Deno: A modern runtime for javascript and typescript. `https://deno.com/`, 2024. Accessed: 2024-07-18.

[3] Ryan Dahl. 10 things i regret about node.js - ryan dahl - jsconf eu 2018. JSConf, June 6 2018. retrieved 2019-05-17.

[4] Bun. `https://bun.sh/`. retrieved 2024-03-19.

[5] pnpm. pnpm: Fast, disk space efficient package manager. `https://pnpm.io/pt/`, 2024. Accessed: 2024-07-18.

[6] Yarn. Yarn: Installation guide. `https://classic.yarnpkg.com/lang/en/docs/install/#mac-stable`, 2024. Accessed: 2024-07-18.

[7] Hardeep Kaur Dhalla. A performance comparison of restful applications implemented in spring boot java and ms.net core. *Journal of Physics: Conference Series*, 1933:012041, 2021. Virtual Conference on Engineering, Science and Technology (ViCEST) 2020, 12-13 August 2020, Kuala Lumpur, Malaysia.

[8] Kai Lei, Yining Ma, and Zhi Tan. Performance comparison and evaluation of web development technologies in php, python, and node.js. In *2014 IEEE 17th International Conference on Computational Science and Engineering*, pages 661–668, 2014.

[9] Danil Demashov and Ilya Gosudarev. Efficiency evaluation of node.js web-server frameworks. In *Proceedings of the 2019 International Conference on Information Technology and Management Science (ITMS)*, Saint Petersburg, Russia, 2019. ITMO University.

[10] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley Professional Computing. Wiley, 1991.

[11] Yustus Eko Oktian and Sang-Gon Lee. Borderchain: Blockchain-based access control framework for the internet of things endpoint. *IEEE Access*, 9:3592–3615, 2021.

[12] Ivan Teixeira, Diogo Branco, and Sergi Bermúdez I Badia. Reh@store: An open-source framework for enhancing ict-based health interventions with secure distribution, maintenance, and data collection. In *CENTERIS – International Conference on ENTERprise Information Systems / ProjMAN – International Conference on Project MANagement / HCist – International Conference on Health and Social Care Information Systems and Technologies 2023*, volume 00, pages 000–000. Elsevier B.V., 2023. Available online at www.sciencedirect.com. This is an open access article under the CC BY-NC-ND license. Peer-review under responsibility of the scientific committee of the CENTERIS – International Conference on ENTERprise Information Systems / ProjMAN - International Conference on Project MANagement / HCist - International Conference on Health and Social Care Information Systems and Technologies 2023.

[13] Matteo Collina. Autocannon, 2024. Accessed: 2024-06-17.

[14] Docker. `https://www.docker.com/`. Accessed: 2024-06-17.

[15] Md Feroj Ahmod. Javascript runtime performance analysis: Node and bun. Master's thesis, Tampere University, Faculty of Information Technology and Communication Sciences (ITC), June 2023. The originality of this thesis has been checked using the Turnitin Originality Check service.

[16] Mayank Choubey. Node.js vs deno vs bun: Benchmark for a real-world case — jwt, postgres, pdf gen. *Tech Tonic*, 2024. Published on Medium, February 18, 2024.