Desempenho e Eficiência de Recursos em Sistemas Web: Uma Comparação entre Rust e Node.js

Henrique Lago Luhm dos Santos (UEPG) E-mail: 20009923@uepg.br
Luciano Jose Senger (DEINFO-UEPG) E-mail: ljsenger@uepg.br

Resumo: A demanda por aplicações web com capacidade de processamento elevada e baixa latência tem intensificado a busca por tecnologias mais eficientes. Nesse contexto, a avaliação de linguagens que ofereçam desempenho adequado e controle de recursos torna-se relevante para a construção de sistemas modernos. Este trabalho apresenta uma análise comparativa entre Rust e Node.js na construção de sistemas web de alto desempenho, com foco em desempenho e uso de recursos. Para esse fim, foram implementadas duas aplicações equivalentes, utilizando apenas bibliotecas nativas, e avaliadas em cenários CPU-bound, I/O-bound, de carga combinada e de alta concorrência, com múltiplos acessos simultâneos. Os experimentos foram conduzidos em ambiente controlado com contêineres Docker, utilizando ferramentas para geração de carga e para monitoramento. Os resultados indicam que Rust apresenta melhor desempenho em cenários com processamento intensivo e alta concorrência, com maior *throughput*, menor tempo de resposta, menor variabilidade e menor consumo de memória. Por outro lado, o Node.js demonstrou melhor desempenho em cenários I/O-bound, em função do modelo assíncrono baseado em *event loop*, em comparação com a implementação síncrona adotada em Rust. A análise qualitativa indica que, embora Rust exija investimento inicial em aprendizado, a linguagem apresenta melhor previsibilidade e segurança de memória, enquanto Node.js favorece maior rapidez no desenvolvimento. Os resultados evidenciam que a escolha da tecnologia depende do perfil da aplicação. O estudo contribui com evidências empíricas para apoiar decisões no desenvolvimento de sistemas web que demandam desempenho e eficiência na utilização de recursos.

Palavras-chave: Rust; Node.js; avaliação de desempenho; concorrência; sistemas web

1. Introdução

Com a adoção de arquiteturas baseadas em microsserviços e aplicações *serverless*, aliada à necessidade contínua de otimização de custos operacionais e redução de latência, a escolha da tecnologia e linguagem de programação para o desenvolvimento *backend* tem se consolidado como um fator estratégico no projeto de sistemas modernos. O *backend* corresponde à camada responsável pelo processamento lógico da aplicação, pela comunicação com bancos de dados e serviços externos, bem como pelo atendimento e resposta às requisições provenientes da camada cliente (*frontend*).

A linguagem Rust, originalmente desenvolvida pela Mozilla, tem ganhado destaque por combinar desempenho próximo ao de linguagens de baixo nível com garantias de segurança de memória, sem a necessidade de um coletor de lixo (*garbage collector*) (BUGDEN; ALAHMAR, 2022). Essa característica permite maior previsibilidade no consumo de recursos e reduz a ocorrência de erros comuns relacionados à gestão de memória, como *buffer overflows* e *data races*.

Embora tradicionalmente classificada como uma linguagem de sistemas, Rust vem sendo progressivamente adotada no desenvolvimento de aplicações web, posicionando-se como uma alternativa a tecnologias consolidadas, como JavaScript/TypeScript no ambiente Node.js. Ainda assim, observa-se uma carência de estudos experimentais que investiguem, de forma prática, o impacto da utilização de Rust no desenvolvimento de sistemas web, especialmente no que se refere ao desempenho e à confiabilidade.

Essa lacuna na literatura técnica motiva a seguinte questão de pesquisa: quais são os benefícios, desafios e limitações associados ao uso de Rust na construção de sistemas web, quando comparado a tecnologias amplamente utilizadas, como o Node.js? Tal questionamento é particularmente relevante para equipes de desenvolvimento que buscam maior controle sobre a alocação de recursos, garantias de segurança em tempo de compilação e redução no consumo de CPU e memória.

Este trabalho se justifica pela necessidade de fornecer uma avaliação prática, objetiva e atualizada acerca do uso de Rust em aplicações web. A comparação direta com o Node.js possibilita identificar se os ganhos em desempenho e segurança de memória são suficientemente relevantes para justificar sua adoção em cenários reais de desenvolvimento. Assim, o objetivo geral deste estudo consiste em analisar o uso da linguagem Rust na construção de sistemas web, comparando-a com o Node.js (TypeScript) a partir de medidas de desempenho e evidências relacionadas à segurança de memória. Para isso, são implementadas duas aplicações com funcionalidades equivalentes, executadas sob condições controladas, de modo a permitir a coleta e análise de dados comparativos. Adicionalmente, realiza-se uma análise qualitativa da experiência de desenvolvimento em ambas as tecnologias, considerando aspectos como curva de aprendizado, produtividade e complexidade de implementação.

2. Revisão da literatura

Rust é uma linguagem de propósito geral, com suporte a múltiplos paradigmas, incluindo programação concorrente, funcional e imperativa. Projetada com foco em desempenho e segurança, busca eliminar classes comuns de erros presentes em linguagens como C e C++, especialmente aqueles relacionados ao gerenciamento de memória. Seu principal diferencial é o modelo de *ownership*, no qual a alocação e liberação de memória são determinadas em tempo de compilação, dispensando o uso de *garbage collector*. Nesse modelo, variáveis são automaticamente desalocadas ao sair de escopo, evitando erros como *use-after-free* e *double free* (BUGDEN; ALAHMAR, 2022). Outro aspecto relevante é que Rust adota o conceito de *borrowing*, que impõe regras rígidas sobre o compartilhamento de referências: múltiplas referências imutáveis são permitidas, ou uma única referência mutável, mas nunca ambas simultaneamente. Essa restrição elimina condições de corrida em tempo de execução, contribuindo para a segurança em sistemas concorrentes (WESSEL, 2019).

Rust também não permite aritmética de ponteiros fora de blocos *unsafe* e realiza verificação automática de limites, o que previne acessos inválidos à memória, como *buffer overflow*. Em casos de acesso fora dos limites, o programa é interrompido de forma controlada, evitando comportamento indefinido (BUGDEN; ALAHMAR, 2022). Por fim, o modelo de *ownership* também pode ser interpretado como um paradigma de programação, no qual o controle explícito do ciclo de vida dos dados impõe maior rigor

ao desenvolvimento. Embora isso implique uma curva de aprendizado mais acentuada, resulta em maior previsibilidade e redução de falhas relacionadas à memória (ALHAZMI, 2018).

O Node.js, por sua vez, adota um modelo de execução baseado em *event loop* de thread única, combinado com operações de I/O não bloqueantes. Essa abordagem permite gerenciar muitas conexões simultâneas com baixo consumo de memória, em contraste com arquiteturas baseadas em múltiplas threads, como servidores Java ou plataformas ASP.NET (TILKOV; VINOSKI, 2010). Nesse modelo, cada requisição é tratada por funções de *callback*, enquanto operações de I/O são executadas de forma assíncrona. Tarefas bloqueantes podem ser delegadas para *worker threads*, permitindo melhor aproveitamento dos recursos computacionais. Essa arquitetura torna o Node.js especialmente eficiente em aplicações orientadas a I/O e com alta concorrência. O ambiente de execução é baseado no motor V8 e permite o uso de linguagens como TypeScript, que introduz tipagem estática, contribuindo para maior robustez e manutenção do código (BIERMAN et al., 2014).

Outro fator relevante na análise de desempenho do Node.js é o mecanismo de gerenciamento de memória. O V8, motor sobre o qual o Node.js é executado, utiliza *garbage collection* (GC), processo de coleta automático e transparente que pode gerar pausas e variações nos tempos de resposta, especialmente em aplicações sensíveis à latência (HAVELUND; SHANKAR, 2024). Estudos mostram que diferentes estratégias de GC impactam não apenas o tempo de execução, mas também a localidade de memória e o *throughput* (BLACKBURN; CHENG; MCKINLEY, 2004). Adicionalmente, o impacto do GC pode variar conforme o padrão de carga da aplicação, sendo mais perceptível em cenários intensivos (KRYLOV et al., 2020). Embora esse mecanismo simplifique o gerenciamento de memória e reduza vazamentos, ele introduz sobrecargas que podem comprometer a previsibilidade e a escalabilidade. Em contraste, o Rust delega o gerenciamento de memória ao compilador, eliminando a necessidade de coleta automática e favorecendo a previsibilidade do desempenho, sem abrir mão das garantias de segurança de memória (BUGDEN; ALAHMAR, 2022).

Ivanov (2022) comparou Rust e C++ em algoritmos de ordenação e estruturas de dados, observando que o desempenho varia conforme o tipo de problema. Rust superou o C++ em algoritmos como Merge Sort, enquanto o C++ apresentou melhor desempenho no algoritmo *Insertion Sort*. Em algoritmos híbridos, os resultados mostraram dependência das estratégias de implementação e otimização adotadas. Em operações com estruturas de dados, como *hashmaps* e árvores balanceadas, ambas as linguagens apresentaram comportamentos semelhantes, com vantagem pontual para Rust em deleções, embora o C++ tenha mantido melhor desempenho geral.

Zomer, Hoffmann e Griebler (2023) avaliaram o desempenho do Rust em aplicações paralelas utilizando o *kernel* EP (*Embarrassingly Parallel*) do conjunto NAS Parallel Benchmarks (NASA, 2024). Os resultados indicaram que Rust supera o C++ em execuções sequenciais e apresenta desempenho próximo ao de Fortran em cenários numéricos. Em aplicações paralelas, demonstrou capacidade de crescimento em escala, especialmente com o uso de bibliotecas específicas, indicando seu potencial para aplicações de alto desempenho. Esse esforço foi posteriormente estendido com a NPB-Rust, uma implementação dos NAS Parallel Benchmarks em Rust (MARTINS et al., 2025).

Crestani (2024) comparou Rust com C++, Go, Python e Node.js em diferentes tipos de tarefas. Os resultados mostraram que Rust frequentemente supera o C++ em operações simples e mantém desempenho elevado em tarefas intensivas. Linguagens como Python apresentaram limitações em alguns cenários, enquanto Go e Node.js mostraram desempenho intermediário.

Demashov e Gosudarev (2019), por sua vez, avaliaram a eficiência de diferentes *frameworks* de servidores web em Node.js, mostrando que a escolha do *framework* influencia o desempenho obtido, o que reforça a importância de controlar essa variável em estudos comparativos.

Em conjunto, os estudos analisados apresentam limitações em relação ao problema investigado neste trabalho, conforme sintetizado no Quadro 1. Primeiro, a maior parte compara Rust a C++ ou Fortran em *benchmarks* algorítmicos ou numéricos (IVANOV, 2022; ZOMER; HOFFMANN; GRIEBLER, 2023), contexto em que predomina o tempo de processamento e não há concorrência entre requisições nem espera por I/O. Segundo, quando o Node.js é incluído na comparação (CRESTANI, 2024), avalia-se a velocidade de execução de tarefas isoladas, e não o comportamento de um servidor sob carga concorrente. Terceiro, o trabalho voltado a servidores web restringe-se a um único ecossistema (DEMASHOV; GOSUDAREV, 2019). Assim, entre os trabalhos analisados, nenhum compara Rust e Node.js na construção de servidores web considerando, de forma conjunta, tempo de resposta, *throughput* e consumo de CPU e memória sob diferentes perfis de carga. É essa lacuna que o presente trabalho busca preencher.

Trabalho	Linguagens comparadas	Tipo de avaliação	Servidor web?
Ivanov (2022)	Rust, C++	Algoritmos de ordenação e estruturas de dados	Não
Zomer, Hoffmann e Griebler (2023)	Rust, C++, Fortran	*Benchmark* paralelo NAS (*kernel* EP)	Não
Crestani (2024)	Rust, C++, Go, Python, JavaScript	Velocidade de processamento em tarefas isoladas	Não
Demashov e Gosudarev (2019)	Node.js (*frameworks*)	Eficiência de servidores web	Sim
Este trabalho	Rust, Node.js (TypeScript)	CPU-*bound*, I/O-*bound*, carga combinada e alta concorrência	Sim

Quadro 1 – Síntese dos trabalhos relacionados

3. Material e Métodos

Este estudo adota uma abordagem quantitativa, com implementação experimental e análise comparativa. Embora o foco principal seja quantitativo, com ênfase na mensuração de desempenho e uso de recursos em diferentes cenários de carga, foi realizada também uma análise qualitativa da experiência de desenvolvimento e uma análise dos aspectos referentes à segurança de memória.

A etapa experimental consiste na implementação de duas versões equivalentes de uma API REST, uma em Node.js com TypeScript e outra em Rust. Foram utilizadas bibliotecas nativas de cada linguagem, com o objetivo de reduzir interferências externas e evitar o impacto de frameworks na análise de desempenho. As aplicações foram projetadas para contemplar dois tipos principais de carga: operações CPU-bound,

representadas pelo cálculo recursivo da sequência de Fibonacci, e operações I/O-bound, simuladas por meio de atrasos controlados que reproduzem o comportamento de chamadas a serviços externos.

Cada aplicação disponibiliza dois *endpoints* principais. O *endpoint* `/fibonacci` executa o cálculo da sequência de Fibonacci para um valor fornecido, representando um cenário de processamento intensivo. Já o endpoint `/async-io` executa um conjunto de operações com atraso controlado, simulando múltiplas requisições a serviços externos e caracterizando um cenário de entrada e saída.

Quanto ao modelo de concorrência, as implementações seguem a abordagem idiomática de cada ambiente quando restrito às bibliotecas nativas. Em Rust, adotou-se a abordagem *thread-per-request*, na qual cada requisição é tratada por uma *thread* independente e criada via *thread*::*spawn*(), sem limitação do número de *threads* simultâneas, o que permite paralelismo em múltiplos núcleos. Em Node.js, as requisições são atendidas pelo *event loop* de *thread* única, com operações de I/O tratadas de forma assíncrona. No *endpoint* /async-io, a implementação em Rust utiliza chamadas bloqueantes (thread::sleep), enquanto a implementação em Node.js emprega *setTimeout* com *Promises*, permitindo a espera concorrente sem bloqueio da *thread* principal. Essa assimetria decorre do fato de que a biblioteca padrão do Rust, embora ofereça a sintaxe async/await, não inclui um *runtime* assíncrono, o qual depende de bibliotecas externas, como o Tokio. Suas implicações são discutidas na Seção 4.3.

Os experimentos foram organizados em quatro cenários distintos: execução isolada de operações CPU-*bound*, execução isolada de operações I/O-*bound*, execução combinada dos dois tipos de operação e um cenário de alta concorrência com aumento progressivo do número de usuários virtuais. Os parâmetros de execução de cada cenário estão sintetizados no Quadro 2.

Cenário	*Endpoint*(s)	Usuários virtuais	Duração	Intervalo entre requisições
CPU-*bound*	/fibonacci	10	30 s	0,1 s
I/O-*bound*	/async-io	20	30 s	0,1 s
Carga combinada	/fibonacci e /async-io	10	30 s	1 s
Alta concorrência	/fibonacci e /async-io	Rampa de 0 a 50	30 s	0,1 s

Quadro 2 – Parâmetros de execução dos experimentos

Em todos os cenários, o cálculo de Fibonacci foi executado com n = 35. Nos cenários de carga combinada e de alta concorrência, cada iteração de usuário virtual chama /fibonacci e, em seguida, /async-io. Cada chamada ao *endpoint* /async-io dispara cinco operações com atraso de 50 ms cada: no Node.js, elas são aguardadas em conjunto por meio de *Promises*; no Rust, cada uma é realizada por uma chamada bloqueante a thread::sleep. Os contêineres foram executados sem limites explícitos de CPU e memória, com as configurações padrão do Docker, no mesmo hospedeiro e hardware. As aplicações foram expostas nas portas 3000 (Node.js) e 3001 (Rust).

Foram coletadas medidas de desempenho e de uso de recursos durante a execução dos experimentos: tempo médio e máximo de resposta, *throughput* (requisições por segundo) e uso de CPU e memória ao longo do tempo. A coleta foi realizada por meio da ferramenta k6, para geração de carga e medição de desempenho, e

do comando *docker stats*, para monitoramento dos contêineres em execução. Cada cenário foi executado uma vez, em ambiente dedicado, opção discutida na Seção 4.3. Nas tabelas de resultados, a coluna "Dispersão estimada" corresponde a uma estimativa aproximada da dispersão dos tempos de resposta, obtida a partir da faixa de variação (mínimo e máximo) observada em cada teste. Esse valor indica apenas a ordem de grandeza da dispersão e não deve ser interpretado como desvio padrão medido nem como intervalo em torno da média.

Os experimentos foram conduzidos em ambiente controlado, utilizando contêineres Docker para isolar as implementações e garantir condições equivalentes de execução. O ambiente experimental foi composto por um sistema Linux Mint 22, baseado em Ubuntu 24.04, com kernel 6.8, executado em um processador AMD Ryzen 5 7600 (6 núcleos e 12 threads), 32 GB de memória RAM DDR5 e armazenamento em SSD. Foram utilizadas as ferramentas Docker (v28.2.2) e Docker Compose (v2.36.2) para orquestração dos contêineres, além das linguagens e ambientes Rust (rustc 1.85.1, cargo 1.85.1) e Node.js (v22.12.0) com TypeScript (5.1.6), além do k6 (v1.0.0) para geração de carga. As execuções foram realizadas em contêineres sobre o mesmo hospedeiro e hardware, de modo a reduzir interferências externas e garantir a comparabilidade entre as implementações. A utilização de Docker e k6 também possibilita a reprodutibilidade dos experimentos em outros ambientes, a partir dos arquivos de configuração fornecidos.

De forma complementar, foi realizada uma análise qualitativa da experiência de desenvolvimento, considerando aspectos como curva de aprendizado, complexidade, produtividade e robustez das soluções. Embora secundária, essa análise contribui para contextualizar os resultados quantitativos. Por fim, os resultados foram interpretados de forma integrada, combinando os dados experimentais com os fundamentos teóricos apresentados na revisão da literatura. Essa abordagem permite analisar o desempenho observado à luz das características arquiteturais de cada tecnologia, considerando tanto aspectos de execução quanto implicações práticas no desenvolvimento de sistemas web.

4. Resultados

4.1 Resultados por tipo de carga

Considerando o cenário CPU-*bound*, Rust superou o Node.js em desempenho (Tabela 1). O *throughput* foi 5,02 vezes maior (168,79 req/s contra 33,61 req/s), enquanto o tempo médio de resposta foi 27,8 vezes menor (17,60 ms contra 489,09 ms). O uso de CPU em Rust variou entre 141% e 183%, indicando paralelismo efetivo, enquanto o Node.js permaneceu próximo a 100%. O consumo de memória também foi inferior em Rust, com aproximadamente 1,2 MiB, ante 15 MiB no Node.js. Esses resultados indicam que, em cargas computacionalmente intensivas, o modelo *multithread* de Rust permite melhor aproveitamento dos recursos disponíveis, enquanto o Node.js permanece limitado pelo modelo de execução em thread única. Como o cálculo recursivo de Fibonacci ocupa a *thread* principal durante toda a sua execução, as requisições simultâneas ao Node.js aguardam em fila até que o *event loop* seja liberado; o tempo de resposta passa, assim, a refletir sobretudo o tempo de espera na fila, e não apenas o de processamento. Isso explica tanto a média elevada quanto a alta variabilidade: o tempo máximo (2.260 ms) chegou a 4,6 vezes a média no Node.js, contra 1,7 vez em Rust, já que cada requisição encontra uma fila de tamanho diferente.

Tecnologia	Resposta média (ms)	Dispersão estimada (ms)	Resposta máxima (ms)	*Throughput* (req/s)
Node.js	489,09	549,2	2260	33,61
Rust	17,6	3,6	29,67	168,79

Tabela 1 – Desempenho no cenário CPU-bound

Nota: dispersão estimada a partir da faixa de variação (mínimo e máximo) observada em cada teste.

No cenário I/O-*bound*, o Node.js apresentou desempenho superior ao Rust (Tabela 2). O *throughput* foi 2,32 vezes maior (264,46 req/s contra 113,76 req/s), enquanto o tempo médio de resposta foi 4,97 vezes menor (50,48 ms contra 250,90 ms). O uso de CPU permaneceu baixo em ambas as implementações (inferior a 2,12%), indicando que o desempenho foi limitado por operações de I/O, e não por capacidade de processamento. O consumo de memória foi maior no Node.js (aproximadamente 13,5 MiB), em comparação a 1,4 MiB no Rust, porém com impacto reduzido nesse tipo de carga. Esses resultados indicam que o modelo assíncrono baseado em *event loop* do Node.js é mais eficiente em cenários dominados por operações de entrada e saída, nos quais períodos de espera podem ser utilizados para o atendimento de outras requisições. Contudo, a diferença deve ser interpretada à luz das implementações adotadas (Seção 3). Como cada chamada ao /async-io realiza cinco operações de 50 ms, o tempo médio no Node.js (50,48 ms) corresponde a uma única espera, pois as operações são aguardadas concorrentemente. Em Rust, o tempo médio (250,90 ms) corresponde à soma das cinco esperas, o que indica que as chamadas bloqueantes a *thread*::*sleep* são executadas em sequência. Assim, o resultado compara o modelo assíncrono do Node.js com um modelo síncrono em Rust e não deve ser tomado como limitação intrínseca da linguagem, que dispõe de *runtimes* assíncronos não avaliados neste estudo. A proximidade entre os tempos máximo e médio nas duas tecnologias é coerente com o caráter determinístico dos atrasos simulados.

Tecnologia	Resposta média (ms)	Dispersão estimada (ms)	Resposta máxima (ms)	*Throughput* (req/s)
Node.js	50,48	0,76	51,73	264,46
Rust	250,9	1,79	257,51	113,76

Tabela 2 – Desempenho no cenário I/O-bound

Nota: dispersão estimada a partir da faixa de variação (mínimo e máximo) observada em cada teste.

No cenário combinado, que integra operações CPU-*bound* e I/O-*bound*, o Rust apresentou desempenho ligeiramente superior (Tabela 3). O *throughput* foi maior (31,47 req/s contra 28,07 req/s), e o tempo médio de resposta foi menor (134,77 ms contra 211,72 ms). O uso de CPU em Rust variou entre 16% e 24%, enquanto no Node.js oscilou entre 6% e 60%, indicando maior estabilidade na execução em Rust. A variabilidade dos tempos de resposta também foi menor em Rust (tempo máximo de 1,9 vez a média, contra 2,9 vezes no Node.js), o que é coerente com o fato de que, no Node.js, as requisições de I/O precisam aguardar a conclusão dos cálculos que ocupam o *event loop*. O consumo de memória manteve o padrão observado nos demais cenários, com aproximadamente 1,57 MiB em Rust e 14,7 MiB em Node.js. Os resultados indicam que, em cargas mistas, o desempenho depende da proporção entre operações computacionais e de I/O, com leve vantagem para o modelo adotado em Rust neste experimento.

Tecnologia	Resposta média (ms)	Dispersão estimada (ms)	Resposta máxima (ms)	*Throughput* (req/s)
Node.js	211,72	138,0	605,14	28,07
Rust	134,77	59,85	254,94	31,47

Tabela 3 – Desempenho no cenário de carga combinada (CPU-bound e I/O-bound)

Nota: dispersão estimada a partir da faixa de variação (mínimo e máximo) observada em cada teste.

No cenário com condições de alta concorrência, o Rust apresentou desempenho superior ao Node.js (Tabela 4). O *throughput* foi 5,92 vezes maior (196,70 req/s contra 33,22 req/s), e o tempo médio de resposta foi 6,09 vezes menor (133,76 ms contra 815,15 ms). A diferença nos tempos máximos de resposta também foi expressiva, com 255,86 ms em Rust e 5,06 s em Node.js. Como cada iteração combina uma requisição a /fibonacci e outra a /async-io, o tempo médio em Rust (133,76 ms) aproxima-se da média entre os tempos observados nos cenários CPU-bound (17,60 ms) e I/O-bound (250,90 ms), enquanto no Node.js as requisições a /async-io também passam a aguardar na fila do event loop ocupado pelos cálculos. No Node.js, o tempo máximo chegou a 6,2 vezes a média, indicando que, com o aumento progressivo do número de usuários, o enfileiramento de requisições no *event loop* elevou acentuadamente a latência de parte delas, enquanto em Rust essa razão permaneceu em 1,9.

O uso de CPU em Rust aumentou progressivamente, atingindo até 326%, o que indica utilização efetiva de múltiplos núcleos. Em contraste, o Node.js permaneceu limitado a aproximadamente 100%, refletindo o uso de um único núcleo. O consumo de memória em Rust manteve-se baixo (cerca de 2,3 MiB), enquanto o Node.js apresentou uso consistente em torno de 14,8 MiB. Esses resultados indicam maior capacidade de escalabilidade do Rust em cenários com alta concorrência.

Tecnologia	Resposta média (ms)	Dispersão estimada (ms)	Resposta máxima (ms)	*Throughput* (req/s)
Node.js	815,15	1252,45	5060	33,22
Rust	133,76	60,13	255,86	196,70

Tabela 4 – Desempenho no cenário de alta concorrência

Nota: dispersão estimada a partir da faixa de variação (mínimo e máximo) observada em cada teste.

4.2 Análise dos resultados

Os resultados indicam padrões consistentes associados às características de cada tecnologia. O Rust apresentou menor consumo de memória em todos os cenários avaliados, com diferenças entre 6,4 vezes (alta concorrência) e 12,5 vezes (CPU-bound), resultado do modelo de gerenciamento de memória em tempo de compilação, sem uso de *garbage collector*. Em relação ao uso de CPU, o Rust demonstrou maior capacidade de utilização de múltiplos núcleos, especialmente nos cenários CPU-bound e de alta concorrência, com valores superiores a 100%. Em contraste, o Node.js permaneceu limitado a um único núcleo, em função do modelo baseado em *event loop* de thread única.

Quanto ao tipo de carga, o Node.js apresentou melhor desempenho no cenário I/O-bound, explorando seu modelo assíncrono, ao passo que a implementação em Rust, baseada em operações síncronas, apresentou desempenho inferior nesse contexto. Nos

demais cenários, incluindo cargas mistas e de alta concorrência, o Rust apresentou melhor desempenho geral. Adicionalmente, o Rust apresentou menor variabilidade nos tempos de resposta nos cenários CPU-bound, combinado e de alta concorrência (tempo máximo entre 1,7 e 1,9 vez a média, contra 2,9 a 6,2 vezes no Node.js), bem como no uso de CPU, indicando maior previsibilidade sob diferentes condições de carga.

4.3 Limitações

Os resultados devem ser interpretados considerando algumas limitações do desenho experimental. A primeira refere-se ao número de execuções. Cada cenário foi executado uma única vez, sob condições controladas, o que não permite construir intervalos de confiança entre execuções nem aplicar testes de significância. Essa opção apoia-se em três características do desenho experimental. Primeiro, os testes foram conduzidos em máquina dedicada, com as aplicações isoladas em contêineres e cargas determinísticas (parâmetro fixo no cálculo de Fibonacci e atrasos fixos nas operações de I/O), o que reduz as fontes de variação entre execuções. Segundo, cada execução agrega muitas requisições, entre aproximadamente 840 e 7.900 conforme o cenário e a tecnologia, de modo que as médias reportadas não dependem de poucas observações. Terceiro, nos cenários CPU-bound, I/O-bound e de alta concorrência, as diferenças observadas variam de 2,3 a 27,8 vezes, magnitude muito superior à flutuação esperada entre execuções em ambiente controlado. A exceção é o cenário combinado, cuja diferença de *throughput* (cerca de 12%) é menor e deve ser interpretada com mais cautela.

A segunda limitação diz respeito à equivalência das implementações. A comparação no cenário I/O-*bound* opõe um modelo assíncrono (Node.js) a um modelo síncrono baseado em *threads* bloqueantes (Rust). Além disso, a abordagem *thread-per-request* sem limitação do número de *threads* pode não se sustentar sob níveis de concorrência superiores aos avaliados, em razão do custo de criação e escalonamento de *threads*. O uso exclusivo de bibliotecas nativas, embora reduza interferências, também distancia as implementações das aplicações reais, que em geral empregam *frameworks* como Express ou Fastify, no Node.js, e Axum ou Actix Web, no Rust.

Por fim, há fatores não controlados relacionados à geração de carga e ao ambiente. O k6 opera com um número fixo de usuários virtuais, cada um aguardando a resposta antes de emitir nova requisição (modelo fechado); nesse modelo, o *throughput* é limitado pelo tempo de resposta somado ao intervalo entre requisições, o que explica por que a razão entre os *throughputs* no cenário CPU-bound (5,02) é bem menor que a razão entre os tempos médios (27,8). A curta duração dos testes (30 s) pode não ser suficiente para manifestar ciclos completos de coleta de lixo no V8, e a execução do gerador de carga e dos contêineres em uma mesma máquina física pode introduzir competição por CPU. Além disso, os contêineres foram executados sem limites de CPU e memória. Nessa condição, a implementação em Rust pôde utilizar livremente os múltiplos núcleos do processador, enquanto o Node.js, por sua arquitetura, permaneceu restrito a um núcleo. Parte da vantagem do Rust nos cenários CPU-*bound* e de alta concorrência decorre, portanto, da disponibilidade de múltiplos núcleos e poderia ser menor em ambientes com cotas de CPU restritivas, comuns em implantações de produção. A carga CPU-*bound*, representada exclusivamente pelo cálculo recursivo de Fibonacci, também não esgota a diversidade de processamento encontrada em aplicações reais.

4.4 Análise qualitativa

A análise qualitativa complementa os resultados quantitativos, considerando aspectos relacionados à experiência de desenvolvimento, produtividade e complexidade de implementação. Por basear-se na experiência dos autores durante a implementação das aplicações, essa análise tem caráter exploratório e não deve ser generalizada. A implementação em Node.js com TypeScript apresentou menor barreira inicial, com desenvolvimento mais direto, ampla documentação e ecossistema consolidado. A prototipagem foi facilitada pela natureza dinâmica da linguagem e pelo uso de ferramentas maduras. Em contraste, Rust exigiu maior investimento inicial em aprendizado, especialmente devido aos conceitos de *ownership* e *borrowing*. A rigidez do compilador, embora reduza a velocidade inicial de desenvolvimento, permite a detecção antecipada de erros que, em outras linguagens, ocorreriam em tempo de execução (CRICHTON, 2021).

Em termos de produtividade, Node.js mostrou vantagem em implementações simples e ciclos rápidos de desenvolvimento. No entanto, após a fase inicial, Rust apresentou produtividade comparável, com redução no tempo de depuração, especialmente em problemas relacionados à concorrência e gerenciamento de memória. Quanto à qualidade do código, Rust oferece garantias mais rigorosas em tempo de compilação, resultando em maior previsibilidade e segurança. Por outro lado, Node.js se beneficia de um ecossistema mais maduro e ampla disponibilidade de bibliotecas, o que facilita integração e adoção. O ferramental do Rust, incluindo compilador, gerenciador de pacotes e ferramentas de análise, contribui para a manutenção do código, com mensagens de erro informativas e suporte integrado a testes.

No aspecto de segurança, Rust elimina classes inteiras de vulnerabilidades relacionadas à memória, como *buffer overflows* e *data races*, enquanto o Node.js, apesar das melhorias do motor V8, permanece sujeito a limitações decorrentes da natureza dinâmica da linguagem e do modelo de execução. De forma geral, a escolha entre as tecnologias depende do contexto de aplicação. Node.js tende a ser mais adequado em cenários que exigem desenvolvimento rápido e menor curva de aprendizado, enquanto Rust é mais indicado para sistemas que demandam maior controle de recursos, confiabilidade e desempenho consistente.

5. CONCLUSÃO

Este estudo avaliou o desempenho de Rust e Node.js na construção de sistemas web de alto desempenho em diferentes cenários de carga, evidenciando diferenças diretamente relacionadas aos modelos de execução de cada tecnologia. Os resultados mostram que o Rust apresentou ganhos expressivos, em *throughput* e tempo de resposta, em cenários CPU-*bound*, com *throughput* cerca de 5 vezes maior e tempo médio de resposta 27,8 vezes menor em comparação ao Node.js. Em cenários de alta concorrência, a vantagem em *throughput* foi ainda maior (5,92 vezes), com redução significativa nos tempos máximos de resposta (255,86 ms em Rust contra 5,06 s em Node.js), além de melhor aproveitamento de múltiplos núcleos de CPU.

Em contrapartida, no cenário I/O-*bound*, o Node.js apresentou desempenho superior, com *throughput* 2,32 vezes maior e tempo médio de resposta 4,97 vezes menor, evidenciando a eficiência do modelo assíncrono baseado em *event loop* para

operações dominadas por espera de I/O. No cenário combinado, o Rust manteve vantagem moderada, com menor tempo de resposta e maior estabilidade no uso de CPU. Além do desempenho, o Rust demonstrou menor consumo de memória em todos os cenários avaliados, com reduções que chegaram a mais de uma ordem de grandeza em comparação ao Node.js. Esse comportamento, aliado à menor variabilidade nos tempos de resposta, indica maior previsibilidade e controle de recursos.

Os resultados evidenciam que a escolha entre Rust e Node.js depende do perfil da aplicação. O Node.js é mais adequado para cargas de trabalho I/O-*bound* e desenvolvimento rápido, enquanto o Rust se destaca em aplicações que demandam alta concorrência, processamento intensivo e eficiência de recursos. Dessa forma, este trabalho fornece evidências quantitativas que contribuem para uma tomada de decisão mais fundamentada no desenvolvimento de sistemas web de alto desempenho.

Esses resultados, contudo, estão sujeitos a limitações. Cada cenário foi executado uma única vez e por curta duração; a vantagem do Node.js no cenário I/O-*bound* reflete a comparação com uma implementação síncrona em Rust, e não com um *runtime* assíncrono; e as implementações utilizaram apenas bibliotecas nativas, sem os *frameworks* amplamente empregados em cada ecossistema. Os resultados devem, portanto, ser entendidos como evidências sobre os modelos de execução avaliados, e não como medida definitiva do desempenho das tecnologias em aplicações de produção.

Como trabalhos futuros, recomenda-se a realização de experimentos com maior duração e múltiplas execuções independentes, de modo a aumentar a robustez estatística e capturar efeitos de longo prazo, como impactos do *garbage collection* e variações de desempenho ao longo do tempo. Também se sugere a inclusão de diferentes frameworks e *runtimes* amplamente utilizados em cada ecossistema, especialmente alternativas assíncronas em Rust, como o Tokio, que não foram exploradas nesta análise. Adicionalmente, recomenda-se a condução de experimentos em ambientes mais próximos de produção, com cargas de trabalho mais heterogêneas e padrões de acesso realistas, de forma a avaliar o comportamento das tecnologias em cenários mais representativos. Essas extensões permitem ampliar a validade dos resultados e aprofundar a compreensão sobre o desempenho de Rust e Node.js em aplicações web reais.

REFERÊNCIAS

ALHAZMI, E. *The concept of ownership in Rust and Swift*. 2018. Tese (Doutorado) – University of Sheffield.

BIERMAN, Gavin; ABADI, Martín; TORGERSEN, Mads. Understanding TypeScript. In: **EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP)**, 2014. Proceedings […]. 2014.

BLACKBURN, S. M.; CHENG, P.; MCKINLEY, K. S. Myths and realities: the performance impact of garbage collection. **ACM SIGMETRICS Performance Evaluation Review**, New York, v. 32, n. 1, p. 25–36, 2004.

BUGDEN, W.; ALAHMAR, A. Rust: the programming language for safety and performance. *arXiv preprint*, arXiv:2206.05503, 2022.

CRESTANI, Alexandre de Rosso. *Uma comparação empírica em velocidade de processamento entre C++, Go, Rust, Python e JavaScript*. 2024. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Universidade Federal do Rio Grande do Sul, Porto Alegre, 2024.

CRICHTON, W. The usability of ownership. In: **WORKSHOP ON HUMAN ASPECTS OF TYPES AND REASONING ASSISTANTS (HATRA)**, 1., 2021, New York. Proceedings […]. New York: ACM, 2021. p. 1–8.

DEMASHOV, D.; GOSUDAREV, I. Efficiency evaluation of Node.js web-server frameworks. In: **MICSECS**, 2019. Proceedings […]. 2019.

HAVELUND, K.; SHANKAR, N. What is a garbage collector? An exercise in compositional refinement. In: **THE PRACTICE OF FORMAL METHODS: ESSAYS IN HONOUR OF CLIFF JONES**. Cham: Springer, 2024. p. 195–215.

IVANOV, N. *Is Rust C++-fast? Benchmarking system languages on everyday routines*. *arXiv preprint*, arXiv:2209.09127, 2022.

KRYLOV, Georgiy; PATROU, Maria; DUECK, Gerhard; SIU, Joran. The evolution of garbage collection in V8: Google's JavaScript engine. In: **MEDITERRANEAN CONFERENCE ON EMBEDDED COMPUTING (MECO)**, 2020.

MARTINS, E. M.; FAÉ, L. G.; HOFFMANN, R. B.; BIANCHESSI, L. S.; GRIEBLER, D. *NPB-Rust: NAS Parallel Benchmarks in Rust*. *arXiv preprint*, arXiv:2502.15536, 2025.

NASA. *NAS Parallel Benchmarks (NPB)*. Moffett Field: NASA Advanced Supercomputing Division, 2024.

TILKOV, Stefan; VINOSKI, Steve. Node.js: Using JavaScript to build high-performance network programs. **IEEE Internet Computing**, v. 14, n. 6, p. 80–83, 2010.

WESSEL, N. *The semantics of ownership and borrowing in the Rust programming language*. 2019. Trabalho de Conclusão de Curso (Bachelor of Science in Computing Science) – Faculty of Science, Radboud University, Nijmegen, 2019.

ZOMER, B. B.; HOFFMANN, R. B.; GRIEBLER, D. Implementação e avaliação de desempenho da linguagem Rust no NAS Embarrassingly Parallel Benchmark. In: **ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD-RS)**, 2023. Anais […]. 2023. p. 53–56.