UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA

CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

BRUNO AUGUSTO PEREIRA PINTO

ANÁLISE COMPARATIVA DE LINGUAGENS E FRAMEWORKS PARA IMPLEMENTAÇÕES DE BACKEND

FORTALEZA

2023

BRUNO AUGUSTO PEREIRA PINTO

ANÁLISE COMPARATIVA DE LINGUAGENS E FRAMEWORKS PARA IMPLEMENTAÇÕES DE BACKEND

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Flávio Rubens de Carvalho Sousa.

FORTALEZA

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

P726a Pinto, Bruno Augusto Pereira.
Análise comparativa de linguagens e frameworks para implementações de backend / Bruno Augusto Pereira Pinto. – 2023.
75 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Tecnologia, Curso de Engenharia de Computação, Fortaleza, 2023.
Orientação: Prof. Dr. Flávio Rubens de Carvalho Sousa.

1. Node.js. 2. Backend. 3. Flask. 4. Análise comparativa. I. Título.

CDD 621.39

BRUNO AUGUSTO PEREIRA PINTO

ANÁLISE COMPARATIVA DE LINGUAGENS E FRAMEWORKS PARA IMPLEMENTAÇÕES DE BACKEND

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Aprovada em: 11/12/2023.

BANCA EXAMINADORA

Prof. Dr. Flávio Rubens de Carvalho Sousa (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Victor Aguiar Evangelista de Farias
Universidade Federal do Ceará (UFC)

Prof. Dr. João Marcelo Uchôa de Alencar
Universidade Federal do Ceará (UFC)

Aos amigos que me acompanharam nessa jornada, aos meus pais que me mantiveram nela e a minha esposa que me deu o encorajamento final, meu muito obrigado.

AGRADECIMENTOS

Ao Prof. Dr. Flávio Rubens de Carvalho Sousa, pela excelente orientação.

Aos professores participantes da banca examinadora Prof. Dr. Victor Aguiar Evangelista de Farias e Prof. Dr. João Marcelo Uchôa de Alencar pelo tempo, pelas valiosas colaborações e sugestões.

"A força bruta pode derrubar uma porta trancada, mas o conhecimento é uma chave mestra." (Jace Beleren)

RESUMO

Tendo em vista o uso de métricas computacionais para precificar o uso infraestrutura de computação em nuvem, Um bom entendimento sobre como linguagens e frameworks se mostra crucial. Ao fazer isso, visamos com esse trabalho entender os pontos fortes e fracos das duas principais tecnologias de desenvolvimento de backend no mercado.

Este trabalho realiza uma análise comparativa entre implementações de backend em Python e Node.js, duas linguagens de programação mais utilizadas segundo a pesquisa do Stack overflow do ano corrente. O estudo aborda diferentes aspectos de desempenho, sendo eles consumo de CPU e memória, em cenários de carga de trabalho variável. Para tal, uma mesma implementação de um CRUD foi realizada em ambas as linguagens e foi submetida a diferentes volumes de carga.

Os resultados revelam nuances interessantes no comportamento das implementações, destacando a eficiência relativa em diferentes contextos.

No que diz respeito ao consumo de CPU, a implementação em Node.js mostrou-se consistentemente mais eficiente em cenários de carga leve, indicando um desempenho superior em operações rápidas e menos intensivas. Por outro lado, a análise do consumo de memória revelou que ambas as linguagens conseguem manter níveis estáveis de eficiência, adaptando-se de maneira notável a cargas variáveis. A implementação em Python exibiu uma leve redução no consumo, sugerindo uma adaptação eficaz a cargas mais intensas, enquanto a implementação em Node.js manteve níveis eficientes, embora com uma variação mais pronunciada.

Este trabalho contribui para o entendimento das nuances de desempenho entre Python e Node.js, promovendo o desenvolvimento de soluções mais eficazes e eficientes em um cenário em constante evolução.

Palavras-chave: Backend; Consumo de CPU; Consumo de memória; Node.js; Javascript; Python; Flask.

ABSTRACT

Given the use of computational metrics to price the infrastructure usage in cloud computing, a thorough understanding of how languages and frameworks function proves to be crucial. In doing so, this work aims to comprehend the strengths and weaknesses of the two primary backend development technologies in the market.

This study conducts a comparative analysis between Python and Node.js backend implementations, which are the two most widely used programming languages according to the Stack Overflow survey of the current year. The research encompasses various aspects of performance, including CPU and memory consumption, under varying workload scenarios. To achieve this, an identical CRUD implementation was carried out in both languages and subjected to different load volumes.

The results reveal interesting nuances in the behavior of the implementations, highlighting relative efficiency in different contexts.

Regarding CPU consumption, the Node.js implementation consistently proved to be more efficient in light workload scenarios, indicating superior performance in quick and less intensive operations. On the other hand, the analysis of memory consumption showed that both languages can maintain stable levels of efficiency, adapting remarkably to variable workloads. The Python implementation exhibited a slight reduction in consumption, suggesting effective adaptation to more intense loads, while the Node.js implementation maintained efficient levels, albeit with a more pronounced variation.

This work contributes to understanding the performance nuances between Python and Node.js, fostering the development of more effective and efficient solutions in an ever-evolving scenario.

Keywords: Backend; CPU consumption; Memory consumption; Node.js; Javascript; Python; Flask.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

API	Interface de Programação de Aplicações
CSS	Folha de Estilo em Cascatas
DOM	Modelo de Documento por Objetos
HTML	Linguagem de Marcação de Hipertexto
JS	JavaScript
REST	Transferência Representacional de Estado

SUMÁRIO

1 INTRODUÇÃO

A escolha da linguagem de programação e do framework para o desenvolvimento de sistemas backend desempenha um papel crucial no desempenho, eficiência e escalabilidade de uma aplicação. Este trabalho propõe uma análise comparativa entre implementações em Python e Node.js, duas linguagens amplamente utilizadas no cenário de desenvolvimento web, cada uma apoiada por ecossistemas robustos e comunidades ativas. Além disso, exploraremos o impacto das diferentes cargas de trabalho nos recursos computacionais, focando em operações fundamentais, como criar, deletar, recuperar um usuário e recuperar todos os usuários.

O objetivo principal desta análise é proporcionar insights valiosos para desenvolvedores, equipes de engenharia e arquitetos de software na tomada de decisões fundamentadas ao escolher a tecnologia mais adequada para seus projetos. Comparamos não apenas o desempenho bruto, mas também a eficiência no consumo de recursos, considerando aspectos cruciais como consumo de CPU e memória. Através dessa análise visamos auxiliar na resolução do problema da decisão da tecnologia a ser utiliza em projeto de software hoje. Entender como as respectivas tecnologias performam quanto as métricas em questão nos auxiliam a economizar dinheiro, uma vez que o faturamento de instâncias de plataformas de computação em nuvem levam em consideração esses valores.

A análise abrange cenários comuns de desenvolvimento, avaliando o desempenho em operações essenciais em sistemas web modernos. Ao finalizar essa investigação, espera-se fornecer uma visão abrangente que não apenas destaque as diferenças quantitativas, mas também forneça uma compreensão qualitativa das implicações práticas para o desenvolvimento de software. O presente estudo visa contribuir para a discussão contínua sobre a escolha de tecnologias no desenvolvimento backend, um tópico em constante evolução na indústria de tecnologia da informação.

A metodologia adotada para a avaliação de desempenho é explicada na seção correspondente, esclarecendo as métricas consideradas e o cenário experimental estabelecido. Os resultados obtidos são apresentados e discutidos em seções separadas, proporcionando uma análise aprofundada do consumo de CPU e memória em diferentes cargas de trabalho e operações de backend. Já na seção de resultados é onde temos os dados coletados seguidos de interpretações dos mesmos, permitindo uma compreensão clara das implicações práticas.

2 TRABALHOS RELACIONADOS

Diversos estudos têm explorado o desempenho de aplicações e as características de tecnologias de backend, fornecendo insights valiosos para desenvolvedores e tomadores de decisão. Esta seção revisa trabalhos relevantes que se aprofundaram em análises comparativas e considerações semelhantes às abordadas neste estudo.

2.1 Estudo de mecanismos e fatores que impactam no desempenho de aplicações Web

Temos o trabalho de (JúNIOR, 2017) que se dedica a analisar a performance de aplicações web com foco em aspectos como consumo de dados e velocidade, pode ser considerado um trabalho relacionado ao presente estudo. Ambos compartilham o interesse comum na eficiência e desempenho de aplicações web, embora com abordagens e métodos diferentes. Enquanto o nosso estudo se concentra na análise comparativa de linguagens e frameworks para implementações de backend, o trabalho citado explora a eficiência de páginas web por meio de experimentos práticos, destacando métricas como a localização do código JavaScript e minificação do código. Ambos os trabalhos contribuem para a compreensão global dos fatores que influenciam a eficiência de aplicações web, oferecendo perspectivas valiosas para desenvolvedores e pesquisadores interessados no aprimoramento contínuo desses sistemas. Dele temos também o uso da ferramenta JMeter para a realização da experimentação.

2.2 Avaliação do impacto de balanceadores de carga sobre o gRPC

Em (MOURA, 2022) temos um estudo sobre o impacto de latência em comunicação entre serviços através do gRPC, considerando balanceadores de carga do tipo Proxy e Client Side, oferece uma perspectiva valiosa que complementa os objetivos do presente trabalho. Enquanto nossa análise se concentra na eficiência e consumo de recursos em implementações de backend, esse estudo aborda especificamente o componente de comunicação em ambientes de microsserviços. A análise comparativa da latência e o impacto no desempenho, realizados em um contexto de cluster Kubernetes, fornecem insights práticos sobre as escolhas de design e otimização em sistemas distribuídos. A constatação de uma adição média de 1 milissegundo de latência com o uso de balanceadores do tipo Proxy em comparação com o Client Side, juntamente com considerações sobre o consumo de recursos, é uma contribuição valiosa para a compreensão do trade-off entre funcionalidades adicionais e desempenho em ambientes de microsserviços

3 FUNDAMENTAÇÃO TEÓRICA

Ao longo desse capítulo vamos introduzir e elucidar diversos conceitos que servirão de base para a leitura e entendimento do trabalho. Na sessão 3.1 vamos definir e descrever backend, bem como apresentar o modelo de cliente servidor.

Nas sessões 3.2 e 3.3 vamos discorrer um pouco mais sobre os conceitos de API e REST, respectivamente.

Por fim na sessão 3.4 vamos apresentar as principais linguagens de backend e alguns de seus frameworks, de modo que de posse dessas informações podemos modelar nosso experimento no capitulo 4

3.1 Backend

No início, a web consistia basicamente de páginas estáticas escritas em uma linguagem de marcação desenvolvida especificamente para esse fim, o HTML, de modo que somente existia a camada de apresentação (BERNERS-LEE *et al.*, 1992). Nesse ponto temos a introdução do Backend, inicialmente como a aplicação de servidor responsável por prover o conteúdo estático.

Mais a frente, com o advento da web 2.0 (SYKORA, 2017) tivemos a introdução de funcionalidades e a necessidade de execução de processos, transformação e persistência de dados. Com isso o servidor não mais era responsável apenas por servir o conteúdo estático e sim processar informações de acordo com as regras de negócio aplicáveis (SOMMERVILLE, 2011).

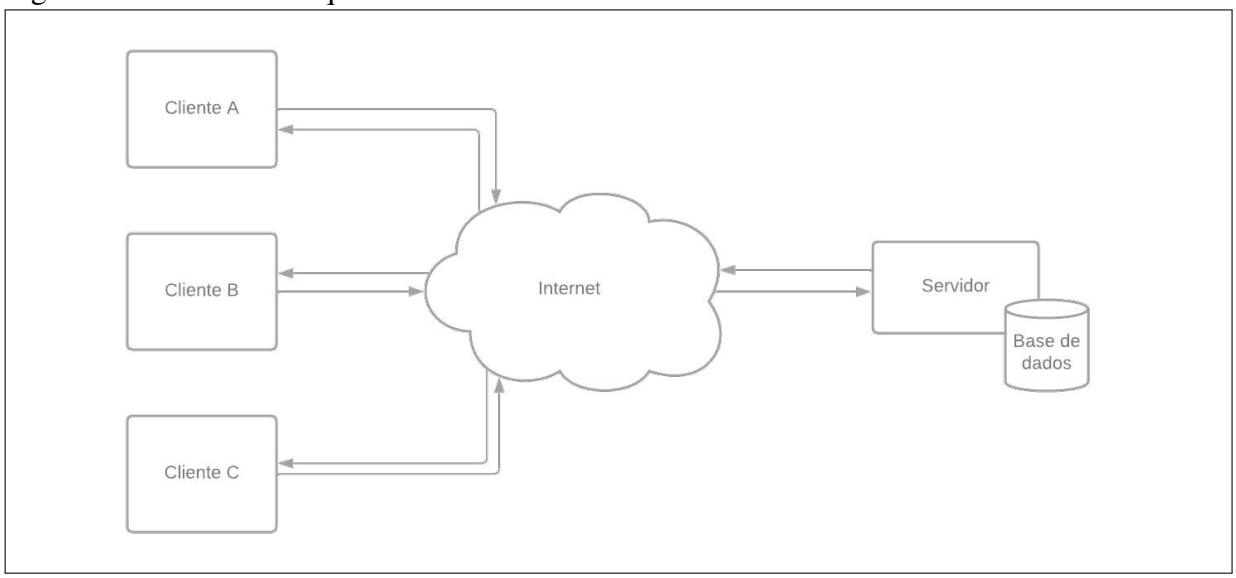
Tendo isso em mente podemos entender como backend a camada de negócios de um software, cuja a execução irá ocorrer no "lado"servidor da arquitetura cliente-servidor. Por esse paradigma, e na conjuntura atual, vemos cada vez mais que o papel do cliente pode ser desempenhado por diversos tipos de software.

3.2 API

Uma vez estabelecido que diferentes aplicações desejam se comunicar para realizar a execução das suas atividades, surge a necessidade de estabelecer uma interface de comunicação entre elas. Disso as Interface de Programação de Aplicações (API) (REDDY, 2011).

O termo API no contexto atual é usado para se referir as Web APIs, isto é, ferramentas para permitir a comunicação entre aplicações através da internet. Entretanto, vale ressaltar que

Figura 1 – Modelo da arquitetura cliente servidor



Fonte: Baseado no MENDES (2002).

o termo API precede esse uso, podendo ser usado no contexto de linguagens de programação, bibliotecas de software, sistemas operacionais e outros. Ao longo desse trabalho vamos nos ater as WEB APIs. (POSTMAN, 2023)

Diferente de uma interface de usuário, uma API não tem o propósito de ser consumida por um usuário final diretamente. Usualmente é composta por diversas funcionalidades que podem ser acionadas de forma independente. Esses acionamentos ou chamadas realizadas são conhecidas como endpoints.

3.3 REST

Transferência Representacional de Estado (REST), consiste em um estilo de arquitetura de software que descreve uma interface entre componentes separados. Sendo bem comum seu uso na internet, em arquiteturas de modelo cliente-servidor, vide imagem 1.

Na arquitetura REST são definidos 6 limitações, que são:

a) Interface Uniforme:
   - Cada recurso dentro do sistema deve conter apenas um único identificador e deve prover um modo de se consumir informações relacionadas.
   - Caso o recuso seja grande, sempre que relevante e necessário, ele deve ser particionado e conter relações para os identificadores dos recursos extraídos, de forma que se possa consumir essas informações.
   - Todas as representações dos recursos empregadas ao longo da aplicação devem

seguir convenções e formatos definidos. Por exemplo, devem ser empregados sempre que possível convenções para nomenclatura, formatos específicos para links ou formatos pre-definidos de dados (XML ou JSON por exemplo).

b) Cliente-Servidor:
   - Uma vez definida e garantida a uniformidade da interface, a aplicação do modelo cliente-servidor garante a separação das responsabilidades. De modo que ambas as aplicações podem evoluir, se seguirem obedecendo o contrato da interface, sem causar problemas entre si.
   - Esse modelo de separação também gera uma simplificação dos componentes servidores, aumentando assim a escalabilidade dos mesmos,

c) Sem estado:
   - A implementação também deve garantir que estados não sejam necessários ou guardados pelo receptor, de forma que todas as informações para processar aquela requisição devem estar contidas nela mesma.
   - Usualmente as informações de controle de sessão são enviadas ao cliente, que encapsula essas informações e as envia junto de todas as requisições que requererem.

d) Capaz de ser salvo em Cache:
   - Sempre que possível, os recursos devem ser salvos em cache. Isso deve ser feito de forma que se tenha uma garantia que apenas recursos aptos a serem cacheados o estão sendo. Evitando o consumo de recursos desatualizados.

e) Sistema em camadas:
   - Ao usar um sistema em camadas, o cliente não consegue distinguir se está conectado no servidor final ou em algum intermediário, o que é um ponto crucial quando atrelado a cacheabilidade dos recursos.

f) Código sob demanda (opcional):
   - Em algumas aplicações o retorno da requisição pode ser um algum tipo de código executável. De forma que a resposta do servidor passaria a "transformar"o cliente.

3.3.1 RESTfull API

A arquitetura REST, que significa Transferência de Estado Representacional, é amplamente utilizada no desenvolvimento de APIs (Interfaces de Programação de Aplicações) devido à sua simplicidade e eficiência. Ao adotar o modelo REST em uma API, busca-se alcançar o que é conhecido como uma API RESTful. Isso implica seguir princípios específicos, como a definição de uma URL base que representa o ponto de entrada para os recursos da API. Essa URL base fornece uma estrutura hierárquica para a organização dos endpoints, facilitando a navegação e compreensão da API. (RICHARDSON; RUBY, 2007)

Além disso, uma API RESTful deve oferecer suporte aos métodos padrões do HTTP, como GET, POST, PUT e DELETE, para realizar operações CRUD (Create, Read, Update, Delete) nos recursos da aplicação. Esses métodos são fundamentais para interações uniformes e previsíveis com a API, proporcionando uma experiência consistente aos desenvolvedores que a utilizam.

Outro aspecto crucial na construção de uma API RESTful é a definição de um "media type". O "media type"especifica o formato dos dados utilizados para o envio e recebimento de informações durante uma requisição. O formato mais comum é o JSON (JavaScript Object Notation), devido a sua simplicidade, leveza e facilidade de leitura por máquinas e humanos. No entanto, outros formatos, como XML (eXtensible Markup Language), também podem ser empregados, dependendo das necessidades do sistema e das preferências dos desenvolvedores.

A escolha de um "media type"adequado é essencial para garantir a interoperabilidade entre sistemas e facilitar a integração de diferentes plataformas. Uma API RESTful bem projetada, com uma URL base coerente, suporte aos métodos HTTP e um "media type"bem definido, contribui para a construção de sistemas robustos e escaláveis.(FIELDING, 2000)

3.4 Linguagens de Backend

3.4.1 JavaScript - Node.js

JavaScript (JS), é uma linguagem de programação que, junto com Linguagem de Marcação de Hipertexto (HTML) e Folha de Estilo em Cascatas (CSS), é uma das principais tecnologias da World Wide Web. Atualmente uma grande parcela dos sites usam JavaScript do lado do cliente, geralmente usando bibliotecas de terceiros. Todos os principais navegadores

da Web possuem um mecanismo JavaScript dedicado para executar o código nas máquinas dos usuários (FLANAGAN, 2011).

JavaScript é uma linguagem de alto nível e geralmente é compilada durante a execução que funciona de acordo com o padrão ECMAScript. Tem tipagem dinâmica e orientação a objetos baseada em protótipos. É um multi paradigma que oferece suporte a métodos de programação imperativos, funcionais e orientados a eventos(CROCKFORD, 2008). Ele inclui APIs para trabalhar com texto, datas, expressões regulares, estruturas de dados padrão e o Modelo de Documento por Objetos (DOM).

O padrão ECMAScript não abrange funções de entrada/saída (E/S), como rede, armazenamento ou funções gráficas. Na prática, o navegador da Web ou outro sistema operacional fornece uma API JavaScript para E/S (INTERNATIONAL, ongoing).

Originalmente usados apenas em navegadores da web, os mecanismos JavaScript agora são componentes essenciais de alguns servidores e vários aplicativos. O mais comum para esse uso é o Node.js.

Node.js é um software de código aberto para servidor, multiplataforma, rodando em Windows, Linux, Unix e macOS. Ele consiste em uma runtime JavaScript de back-end. Ele é executado no V8 JavaScript Engine e executa o código JavaScript fora de um navegador da web.

O Node.js permite que os desenvolvedores usem JavaScript para escrever ferramentas de linha de comando e para scripts do lado do servidor, portanto representa um paradigma de "JavaScript em todos os lugares", unificando o desenvolvimento de aplicativos da Web em torno de uma única linguagem de programação, em vez de linguagens diferentes para scripts do lado do servidor e do lado do cliente (MEAD, 2019).

O Node.js tem uma arquitetura orientada a eventos capaz de E/S assíncrona. Essas escolhas de design visam otimizar o rendimento e a escalabilidade em aplicativos da Web com muitas operações de entrada/saída, bem como para aplicativos da Web em tempo real (por exemplo, programas de comunicação em tempo real e jogos de navegador).

A arquitetura do Node.js é centrada no conceito de "single-threaded"(único segmento de execução), mas altamente escalável. O segredo por trás dessa eficiência reside na utilização de um modelo de I/O não bloqueante e assíncrono. Enquanto as plataformas tradicionais bloqueiam a execução do código enquanto aguardam operações de entrada/saída, o Node.js permite que o servidor continue a lidar com outras tarefas enquanto aguarda operações de I/O completarem.

O coração da arquitetura do Node.js é o Event Loop (Laço de Eventos). O Event

Loop permite que operações assíncronas sejam delegadas e, uma vez concluídas, seus callbacks correspondentes são acionados. Isso significa que o Node.js pode gerenciar milhares de conexões simultâneas sem consumir recursos excessivos, tornando-o especialmente adequado para aplicações em tempo real, como chats e transmissões ao vivo.

3.4.1.1 Express.js

O Express.js, muitas vezes referenciado apenas como Express, é um poderoso e minimalista framework para o desenvolvimento de aplicativos web com Node.js. Criado para facilitar a construção de APIs robustas e aplicações web escaláveis, o Express oferece uma arquitetura flexível baseada em middleware, permitindo que os desenvolvedores gerenciem facilmente solicitações e respostas HTTP (BROWN, 2014). Com uma curva de aprendizado suave e uma abordagem descomplicada, o Express se destaca por sua capacidade de acelerar o desenvolvimento de servidores web, proporcionando uma estrutura organizada para lidar com rotas, manipulação de solicitações e respostas, além de integração eficiente com outros módulos e ferramentas(CONTRIBUTORS, ongoing).

A simplicidade do Express não sacrifica a flexibilidade. Os desenvolvedores têm a liberdade de escolher entre uma variedade de bibliotecas e plugins para aprimorar as funcionalidades do framework de acordo com as necessidades específicas de seus projetos. A comunidade ativa em torno do Express contribui para uma extensa documentação e uma ampla gama de recursos adicionais disponíveis, tornando-o uma escolha popular para desenvolvedores que buscam agilidade e desempenho em suas aplicações Node.js(OSMANI, 2018).

Além disso, a abordagem "unopinionated"do Express significa que ele oferece liberdade aos desenvolvedores para decidirem a estrutura e as ferramentas que melhor se adéquam ao seu contexto, ao mesmo tempo em que fornece um conjunto essencial de funcionalidades. Ao longo dos anos, o Express estabeleceu-se como o principal framework para Node.js, impulsionando o desenvolvimento de inúmeras aplicações web de sucesso, desde startups até grandes empresas, solidificando sua posição como uma ferramenta indispensável no ecossistema Node.js.

3.4.2 Python

Python é uma linguagem de programação de alto nível e de uso geral. A filosofia de design enfatiza a legibilidade do código por meio de identação de código. Python possui tipagem e "coleta de lixo"de forma dinâmica. Ele oferece suporte a vários paradigmas de programação,

incluindo programação estruturada (principalmente processual), orientada a objetos e funcional. Por causa de sua extensa biblioteca de padrões, muitas vezes é vista como uma linguagem de "pilhas inclusas".

Guido van Rossum começou a trabalhar no Python como o sucessor da linguagem de programação ABC no final dos anos 1980, lançando-o pela primeira vez como Python 0.9.0 em 1991. O Python 2.0 foi lançado em 2000 e introduziu novos recursos, como compreensão de lista, detecção de coleta de lixo circular e suporte a Unicode. Lançado em 2008, o Python 3.0 foi uma grande revisão que não era exatamente compatível com as versões anteriores. Python 2 foi descontinuado e declarado obsoleto em 2020 com a versão 2.7.18

3.4.2.1 Flask

Flask é um microframework da web escrito em Python. É classificado como um microframework porque não requer ferramentas ou bibliotecas especiais. Ele não possui camada de abstração de banco de dados, validação de formulário ou outros componentes que fornecem funcionalidades comuns providas por bibliotecas de terceiros. No entanto, o Flask oferece suporte a extensões que podem adicionar funcionalidade aos aplicativos como se fossem implementados nativamente no Flask.

O Flask se tornou popular entre os entusiastas do Python. Em novembro de 2022, ele tinha o segundo maior número de estrelas no GitHub entre os frameworks de desenvolvimento da web em Python, apenas um pouco atrás do Django, e foi eleito o framework da web mais popular no Python Developers Survey 2018, 2019, 2020 e 2021.

O Flask adota um modelo de I/O bloqueante como padrão, simplificando a lógica de programação ao tratar operações de entrada/saída de maneira linear. Este modelo é particularmente eficaz em situações onde a clareza e a simplicidade do código são cruciais. Operações como leitura de arquivos, consultas a bancos de dados e interações com APIs externas são tratadas de maneira direta e compreensível.

Este modelo de I/O bloqueante está alinhado com a prática comum em desenvolvimento Python, integrando-se perfeitamente ao ecossistema da linguagem e aproveitando as convenções e bibliotecas disponíveis para operações de I/O bloqueantes.

Embora o Flask adote um modelo de I/O bloqueante, ele é capaz de suportar threads para a execução concorrente de tarefas. No entanto, é crucial considerar o Global Interpreter Lock (GIL) do Python, que restringe a execução simultânea de instruções Python em múltiplas

threads em um processo.

3.4.2.2 Django

Django é um framework web gratuito e de código aberto baseado em Python que segue o padrão arquitetônico model–template–views (MTV). Ele é mantido pela Django Software Foundation (DSF), uma organização independente estabelecida nos EUA sem fins lucrativos.

O principal objetivo do Django é facilitar a criação de sites complexos e baseados em banco de dados. A estrutura enfatiza a reutilização e a "capacidade de conexão"dos componentes, menos código, baixo acoplamento, desenvolvimento rápido e o princípio de não se repetir. O Django também fornece uma interface administrativa opcional de criação, leitura, atualização e exclusão que é gerada dinamicamente por meio de introspecção e configurada por meio de modelos administrativos.

3.5 JMeter

O Apache JMeter é uma aplicação Java, construída para avaliar o desempenho funcional e de carga de diferentes tipos de serviços e aplicações, incluindo servidores web. Sua arquitetura modular permite a extensão de funcionalidades, tornando-o uma escolha popular para uma variedade de cenários de teste. Sua principal finalidade é realizar testes de desempenho e estresse em aplicações, simulando o comportamento de múltiplos usuários e avaliando como o sistema responde a diferentes cargas de trabalho. Além disso, o JMeter é eficaz na execução de testes de carga para medir a capacidade de um sistema e identificar possíveis gargalos ou pontos de falha. O processo de utilização do JMeter envolve a criação de planos de teste, nos quais são definidos os cenários de teste, configurações de carga e parâmetros de execução. Os usuários podem criar casos de teste detalhados que incluem solicitações HTTP, consultas a bancos de dados, manipulação de cookies, entre outras operações. O JMeter oferece uma interface gráfica intuitiva para a criação desses planos de teste, permitindo a configuração precisa de cada etapa. O JMeter oferece uma variedade de recursos, incluindo a capacidade de monitorar e analisar resultados em tempo real, gerar relatórios detalhados e suportar testes de carga distribuídos. Sua extensibilidade é evidenciada por meio de plugins que adicionam funcionalidades específicas e integrações com outras ferramentas. Com uma comunidade ativa e uma extensa documentação, o JMeter continua a evoluir, atendendo às demandas do desenvolvimento de software moderno.

4 METODOLOGIA

Nessa sessão vamos discutir sobre o modelo de experimentação elaborado, discorrendo sobre as métricas selecionadas, o processo de execução do experimento e obtenção dos valores das métricas. Na sessão 4.1 vamos discorrer sobre a estrutura básica do projeto e sobre o modelo das implementações realizadas. Na sessão 4.2 vamos abordar as métricas selecionadas para o trabalho. Já na sessão 4.3 vamos abordar o processo de escolha das linguagens a serem utilizadas no trabalho. nas sessões 4.4, 4.5 e 4.6 vamos adentrar nas implementações realizadas, seguindo os requerimentos já abordados. Finalmente vamos entrar nas sessões 4.8 e 4.9 onde vamos adentrar no processo de execução dos testes e na coleta dos resultados.

4.1 Descrição do experimento

O experimento consiste de duas simples implementações de APIs REST que foram submetidos a testes de carga. As implementações contém 4 rotas cada, sendo:

a) Recuperar todos
   - Método HTTP: GET .
   - Descrição: Recupera todos os registros criados no banco de dados .

b) Recuperar um
   - Método HTTP: GET.
   - Descrição: Recupera um registro existente no banco de dados de um ID especifico.

c) Criar Novo
   - Método HTTP: POST.
   - Descrição: Cria um novo registro no banco usando as informações de Usuário e senha recebidas.

d) Deletar um
   - Método HTTP: DELETE.
   - Descrição: Remove um registro no banco de dados de um ID especifico.

Cada uma das rotas foi submetida a 5 séries de testes, sendo os cenários:

a) 128 requisições

b) 256 requisições

c) 512 requisições

d) 1024 requisições

e) 2048 requisições

As implementações consistem de um simples sistema de gerenciamento de usuários. Cada uma delas esta integrada com um mesmo banco de dados cujos detalhes da implementação e modelagem serão explicitados na sessão 4.4

4.2 Métricas

Duas métricas foram definidas para comparar as medidas de desempenho das APIs desenvolvidas. As métricas são: consumo de CPU e consumo de memória.

4.2.1 Consumo de CPU

Essa métrica visa medir quanto de processamento precisamos alocar para executar a aplicação. Tendo em vista que é um ponto chave quando vamos decidir a infraestrutura que será alocada para a execução da aplicação. Em modelos de hospedagem de aplicações em nuvem, similar ao adotado pela AWS, o consumo de CPU é levado em consideração para a precificação do serviço.

4.2.2 Consumo de memória

Similar a métrica anterior, o consumo de memória é medido nesse caso por ser um ponto chave para decidir a infraestrutura que será alocada. Outro ponto que vale ressaltar é que o consumo de memória é uma das métricas mais percebidas pelo autor durante a pesquisa para a escrita desse trabalho, seja em trabalhos acadêmicos quanto em artigos publicados pela comunidade. Ressaltando assim a percepção de importância dessa métrica para a comunidade no geral.

Nesse cenário teremos essa unidade medida em porcentagem da memória consumida pela memória total disponível e pode ser descrita através da equação 4.1

$$\frac{m}{M} * 100 \tag{4.1}$$

onde "m"é a memória consumida pelo processo e "M"é a memória total disponível.

4.3 Seleção das linguagens e frameworks

Para a seleção das linguagens e frameworks a serem analisados, tomou-se como parâmetro a pesquisa anual do Stack Overflow. A pesquisa contou com 89184 respostas de desenvolvedores de 185 países. Temos por tanto, ao olhar para pesquisa, uma imagem do atual cenário de desenvolvimento web no mundo.

Um ponto de atenção sobre a pesquisa é que em sua sessão de tecnologias, não há distinção entre backend e frontend, de modo que uma análise teve de ser feita em cima dos resultados.

Segundo a pesquisa, as 7 linguagens mais utilizadas são:

1. JavaScript
2. HTML/CSS
3. Python
4. SQL
5. TypeScript
6. Bash/Shell (all shells)
7. Java

Dessas, as que podemos considerar para o desenvolvimento de APIs são: JavaScript, Python, Typescript e Java. No entanto como os números de JavaScript e Typescript podem estar inflados dado que nessa porcentagem também estão contidos a utilização da linguagem para o frontend, se fez necessário a análise da sessão de frameworks. Segundo a pesquisa, os frameworks mais utilizadas são:

1. Node.js
2. React
3. jQuery
4. Express
5. Angular
6. Next.js
7. ASP.NET CORE
8. Vue.js
9. WordPress
10. ASP.NET
11. Flask

12. Spring Boot
13. Django
14. Laravel
15. FastAPI

Ao fazermos a intercessão das duas listas obtemos JavaScript com Node.js/Express e Python com Flask como combinação de linguagens e frameworks escolhidos para a análise.

4.3.1 Javascript, Node.js e Express

Na pesquisa, foi a linguagem dominante. No entanto vale relembrar que como a pesquisa não faz distinção entre frontend e backend os números estão inflados. Isso é ainda mais perceptível quando olhamos a lista de frameworks onde os 6 primeiros são relacionados a Javascript. Mesmo com essa observação, a excelente colocação do Node.js e do Express ajuda a solidificar a escolha dessa combinação.

4.3.2 Python e Flask

Quando se analisa a lista de Linguagens Python aparece em terceiro lugar, isso antes de descartamos HTML/CSS que escapa do escopo do trabalho. Após essa análise o Python fica em segundo lugar em termos de linguagens. Uma vez que a decisão de linguagem foi feita, a decisão pelo framework foi estritamente baseada na pesquisa, onde temos Flask melhor colocado que a outra opção que tínhamos para Python, o Django

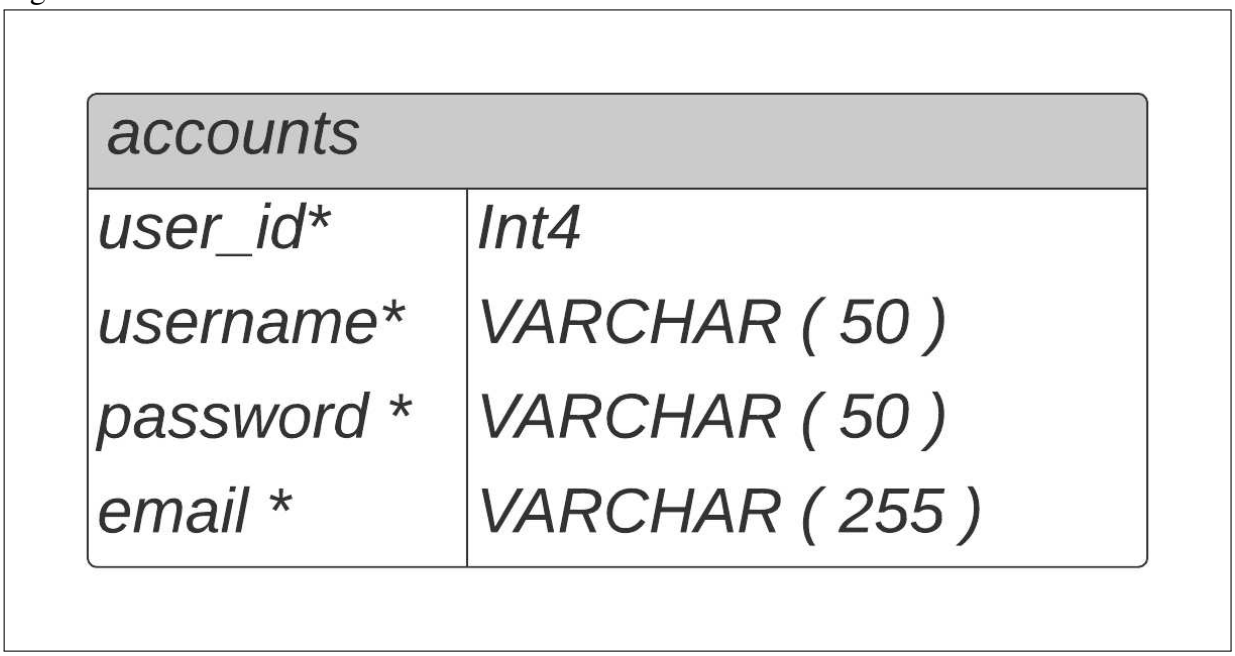
4.4 Banco de dados

O Banco de dados escolhido para a implementação foi o Postgres, sendo esse o banco mais utilizado atualmente segundo a pesquisa do Stack Overflow com mais de 45% de utilização. Outro ponto levado em consideração foi a facilidade de implementação e de integração com as linguagens selecionadas para a implementação.

Não foi investido muito esforço nesse ponto, uma vez que como a mesma implementação de banco de dados foi utilizada para ambas implementações e ambas as tecnologias contavam com bibliotecas padrões para realizar a conexão com o banco.

Para essa aplicação apenas um banco de dados postgres foi instalado e um script que foi utilizado para gerar uma tabela, vide 2.

Figura 2 – Modelo ER do banco de dados



A tabela em questão consiste de quatro campos obrigatórios, sendo o User_id gerado automaticamente pelo banco e os outros três campos, username, password e email, sendo fornecidos no momento da inserção.

Vale ressaltar que a simples implementação do banco é apenas para chegar mais perto do processamento feito por uma requisição a uma API, tendo em vista que é frequente o acesso ao banco atrelado como processamento em algumas rotas. Não optamos por uma implementação mais robusta do banco pois apenas operações simples no banco já supriam a necessidade.

4.5 Implementação em Node.js

A implementação em Node, seguindo a modelagem de rotas já proposta e discutida anteriormente está disponível no Github no link (https://github.com/BrunoAugustoPereira/TCCNodeAPI)

4.6 Implementação em Python

A implementação em Python, seguindo a modelagem de rotas já proposta e discutida anteriormente está disponível no Github no link (https://github.com/BrunoAugustoPereira/TCCPythonAPI)

4.7 Configurações das máquinas

Para executar o experimento, as APIs desenvolvidas foram executadas usando a infraestrutura da AWS. Para isso, optou-se por usar as máquinas t2.micro que fazem parte da camada de instâncias de baixo custo na AWS.

A instância T2.micro é parte da família de instâncias T2, projetada para cargas de trabalho de uso geral que exigem equilíbrio entre desempenho e custo.

a) Recursos de Computação:
   - Virtual CPUs (vCPUs): 1 núcleo virtual.
   - Memória RAM: Geralmente, cerca de 1 GB de memória RAM.

b) Desempenho Burstable:
   - As instâncias T2 são conhecidas por seu modelo de "desempenho burstable". Isso significa que a instância pode acumular créditos de CPU durante os períodos em que a carga de trabalho é leve e usar esses créditos para obter um desempenho mais elevado durante picos de atividade.

c) Armazenamento:
   - Armazenamento EBS (Elastic Block Store): A instância T2.micro geralmente vem com armazenamento EBS. O tamanho padrão pode variar, mas frequentemente é de 8 GB.

d) Conectividade:
   - Rede: Conectividade de rede de alto desempenho.
   - Elastic IP: Pode ser associado a instâncias para facilitar a manutenção do endereço IP.

e) Sistema Operacional:
   - Linux

4.8 Automação do experimento

De forma a padronizar a execução do experimento, optou-se pelo uso da ferramenta JMeter, comumente utilizada no mercado para testes de carga em aplicações. Além de prover um modo estruturado para realizar as requisições, ele permitia de forma fácil coletar informações sobre o teste que estava sendo executado. Por conta disso, ao final de cada rodada de requisições, também fizemos a análise da porcentagem de erros e usamos o valor de Vazão como indicativo

do volume de carga aplicado.

Conforme discutido na 3.5, o teste via JMeter se baseia na configuração do plano de testes. Para tal dois valores são cruciais, o número de usuários virtuais e o tempo de inicialização.

O número de usuários virtuais para o JMeter é o mesmo que o numero de threads a serem criadas. Cada thread criada pode executar qualquer numero de requisições HTTP, conforme configurado no plano de testes. Afim de facilitar a execução dos testes, em cada execução apenas uma requisição HTTP era realizada por vez de modo que o numero de requisições e threads era o mesmo ao longos do experimento.

O segundo parâmetro é o tempo de inicialização, ou ramp-up. Refere-se ao período durante o qual o número de usuários virtuais ou threads é gradualmente aumentado para atingir a carga máxima especificada em um teste de desempenho. Em outras palavras, é o intervalo de tempo que o JMeter leva para passar de zero usuários ativos para o número total de usuários definido para o teste.

Ao configurar um teste de carga no JMeter, você define a carga máxima simultânea que deseja simular. O ramp-up determina como essa carga será atingida ao longo do tempo. Durante o período de ramp-up, o JMeter adiciona gradualmente novos usuários até alcançar a carga máxima desejada.

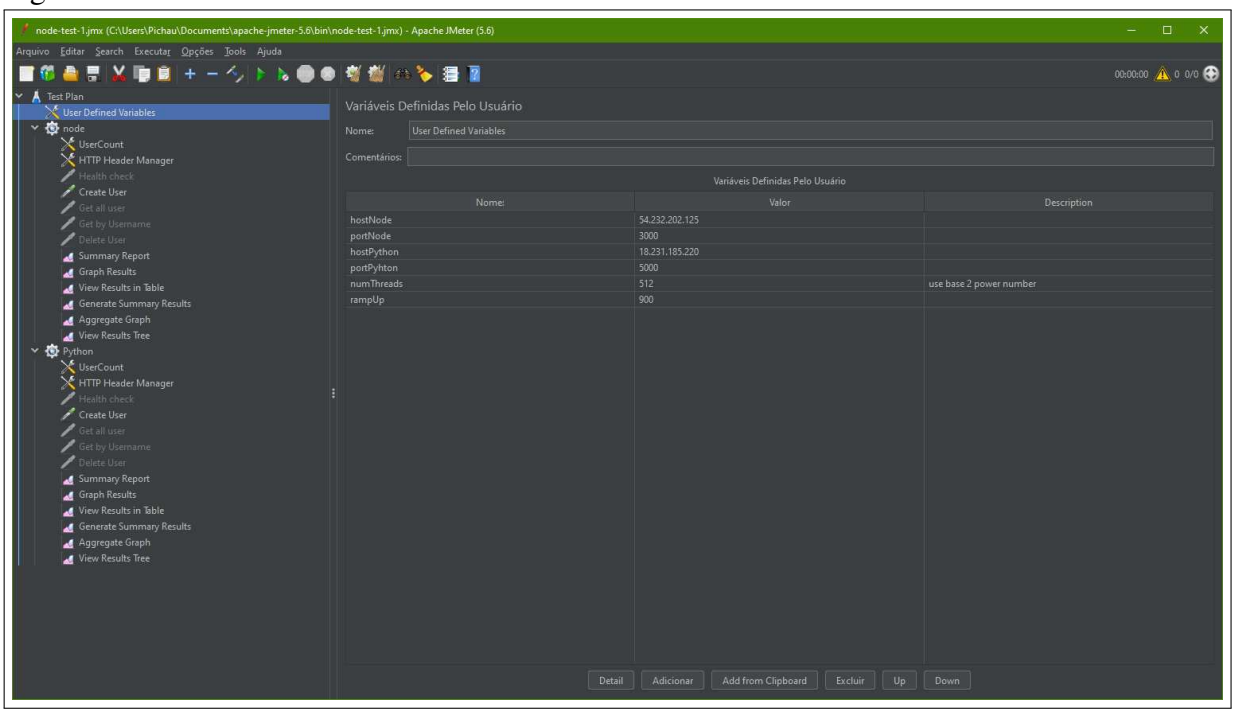
Munidos dessas informações podemos entender melhor ao plano de testes utilizado no experimento. Os valores de ramp-up foram fixados em 128 segundos em todas as iterações do teste e o numero de threads acompanhava as potencias de base 2 iniciando em 128 e indo ate 2048.

4.9 Coleta das métricas

Para a coleta das métricas foi criado um script em Python que realiza leituras das métricas desejadas a cada segundo e as escreve-as em um arquivo .csv, junto de um timestamp. Para tal fez-se uso da biblioteca "psutil". O script está disponivel em (https://github.com/BrunoAugustoPereira/scrip

A biblioteca psutil é uma ferramenta útil para acessar informações detalhadas sobre processos em execução e a utilização de recursos do sistema. Seu nome é uma abreviação de "process and system utilities". Essa biblioteca proporciona uma interface fácil de usar para interagir com informações do sistema operacional, como CPU, memória, discos, rede e sensores de temperatura. O psutil é compatível com diversos sistemas operacionais, incluindo Windows, Linux, macOS, e BSD.

Figura 3 – JMeter



Entre os recursos oferecidos pela biblioteca, destacam-se a capacidade de obter informações sobre processos em execução, como PID (identificador do processo), CPU e consumo de memória, além de possibilitar o monitoramento em tempo real dessas métricas. Também é possível obter dados sobre a utilização do sistema, como o status da CPU, informações sobre a memória disponível e a utilização da rede.

A psutil é valiosa para desenvolvedores que precisam criar ferramentas de monitoramento de sistema, automação de tarefas relacionadas a processos ou mesmo para diagnóstico de desempenho. Sua simplicidade de uso e a ampla variedade de informações que disponibiliza fazem dela uma escolha popular na comunidade Python para tarefas relacionadas à administração e análise de sistemas.

5 RESULTADOS

Neste capítulo vamos apresentar os resultados obtidos após a execução do experimento detalhado no capítulo anterior. Na sessão 5.1 vamos abordar os resultados da primeira métrica analisada, o Consumo de CPU. Já na sessão 5.2 vamos nos aprofundar nos resultados obtidos ao analisar o consumo de memória das máquinas ao executar as implementações. Para facilitar a comparação, 4 tipos de visualizações foram criadas a partir dos dados coletados, sendo eles:

a) Média
b) Mediana
c) Máximo
d) $\frac{Min+Max}{2}$

5.1 Consumo de CPU

5.1.1 Endpoint Criar um usuário

Na tabela 1 temos a comparação entre a média do consumo de CPU em porcentagem para cada cenário de testes na criação de um usuário. Através dela conseguimos perceber que a implementação em Python desde a carga mais baixa já tinha valores mais altos. Além disso podemos perceber que o fator de crescimento no consumo na implementação em Python é bem maior que da em Node, levantando a suposição que em maiores volumes teríamos um gargalo primeiro nessa implementação. Tal suposição será discutida um pouco mais a frente.

Tabela 1 – Create média

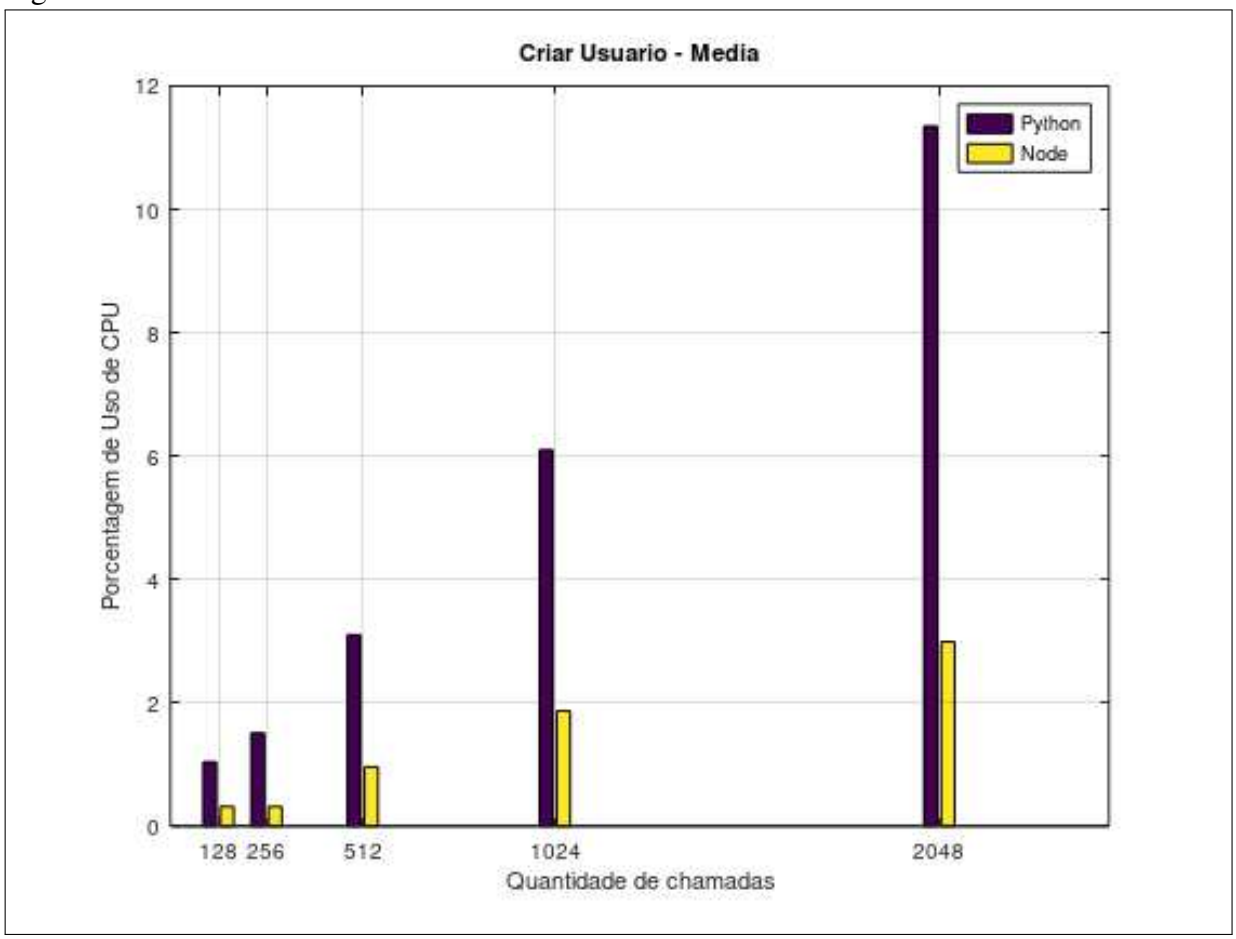
	128	**256**	**512**	**1024**	**2048**
Python	1,04%	1,51%	3,1%	6,1%	11,35%
Node	0,316%	0,316%	0,96%	1,87%	2,99%

A tabela 2 apenas reforça as análises já feitas, ao trazer um fator de crescimento maior na implementação em Python e uma maior entre os valores obtidos.

Tabela 2 – Create Mediana

	128	**256**	**512**	**1024**	**2048**
Python	1%	2%	3,05%	6,9%	13%
Node	0%	0%	1%	2%	3%

Figura 4 – Criar Usuário - Média de uso da CPU



No entanto ao analisarmos as tabelas 3 e 4 e compararmos os picos de consumo, percebemos que os valores são mais próximos do que as métricas anteriores indicavam, de modo que a suposição levantada na análise da tabela 1 é colocada em cheque. Para confirmar tal análise seria necessário expandir o escopo do trabalho, tal possibilidade será debatida no capitulo 6.

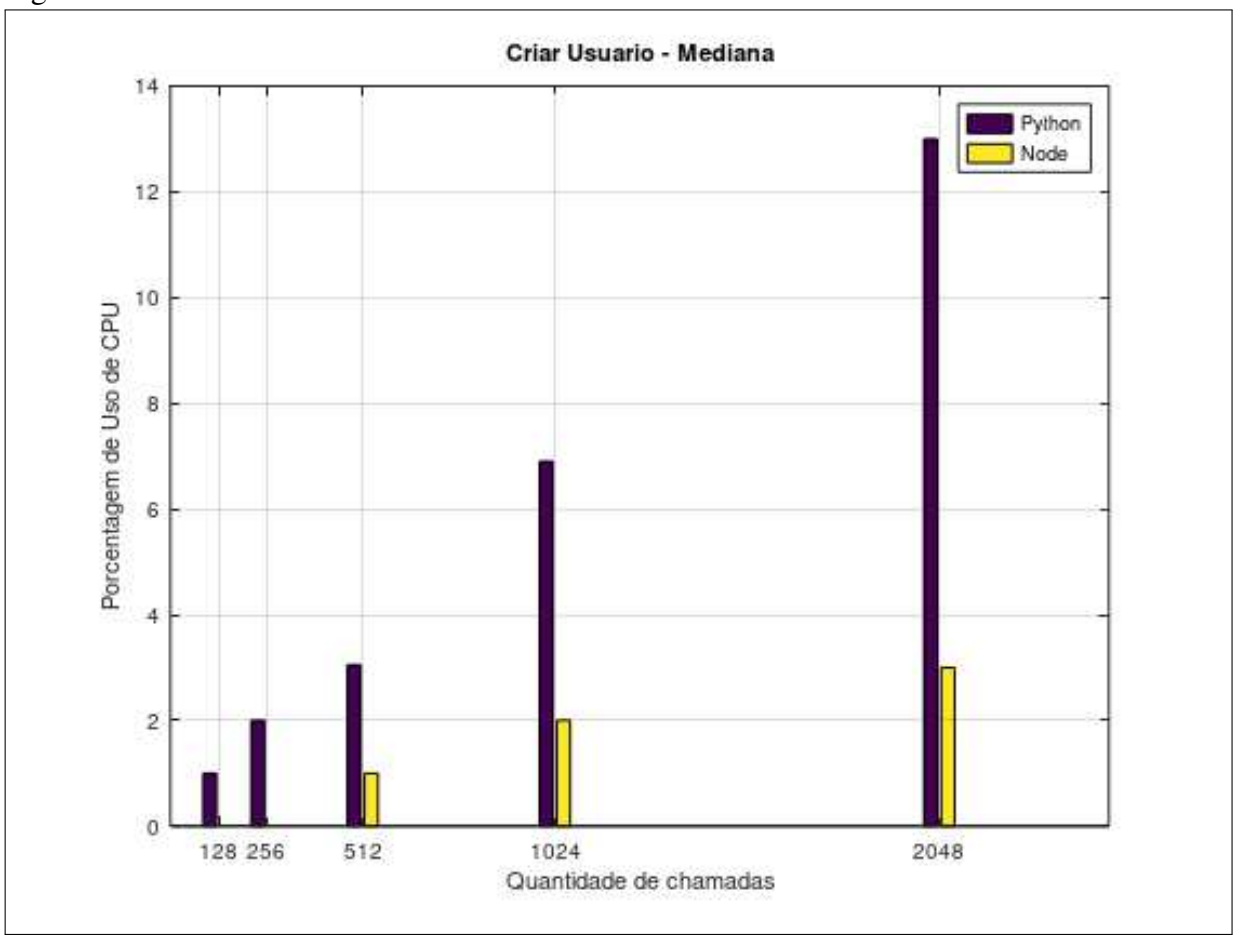
Tabela 3 – Create Máximo

	128	**256**	**512**	**1024**	**2048**
Python	4%	3,9%	5,9%	8,9%	15,5%
Node	3%	3%	3%	8%	8,5%

Tabela 4 – Create (Min+Max)/2

	128	**256**	**512**	**1024**	**2048**
Python	2%	1,95%	2,95%	4,45%	7,75%
Node	1,5%	1,5%	1,5%	4%	4,25%

Figura 5 – Criar Usuário - Mediana de uso da CPU



5.1.2 Endpoint deletar um usuário

Nessa sessão temos a análise do desempenho perante o endpoint de remoção de um usuário. Na tabela 5 salta aos olhos o alto valor do Node na primeira iteração do experimento (128 requisições). Esse valor se destaca como um possível outlier ainda mais por conta do fato das iterações seguintes não acompanharem tal tendência, voltando a resultados próximos aos obtidos na sub sessão 5.1.1

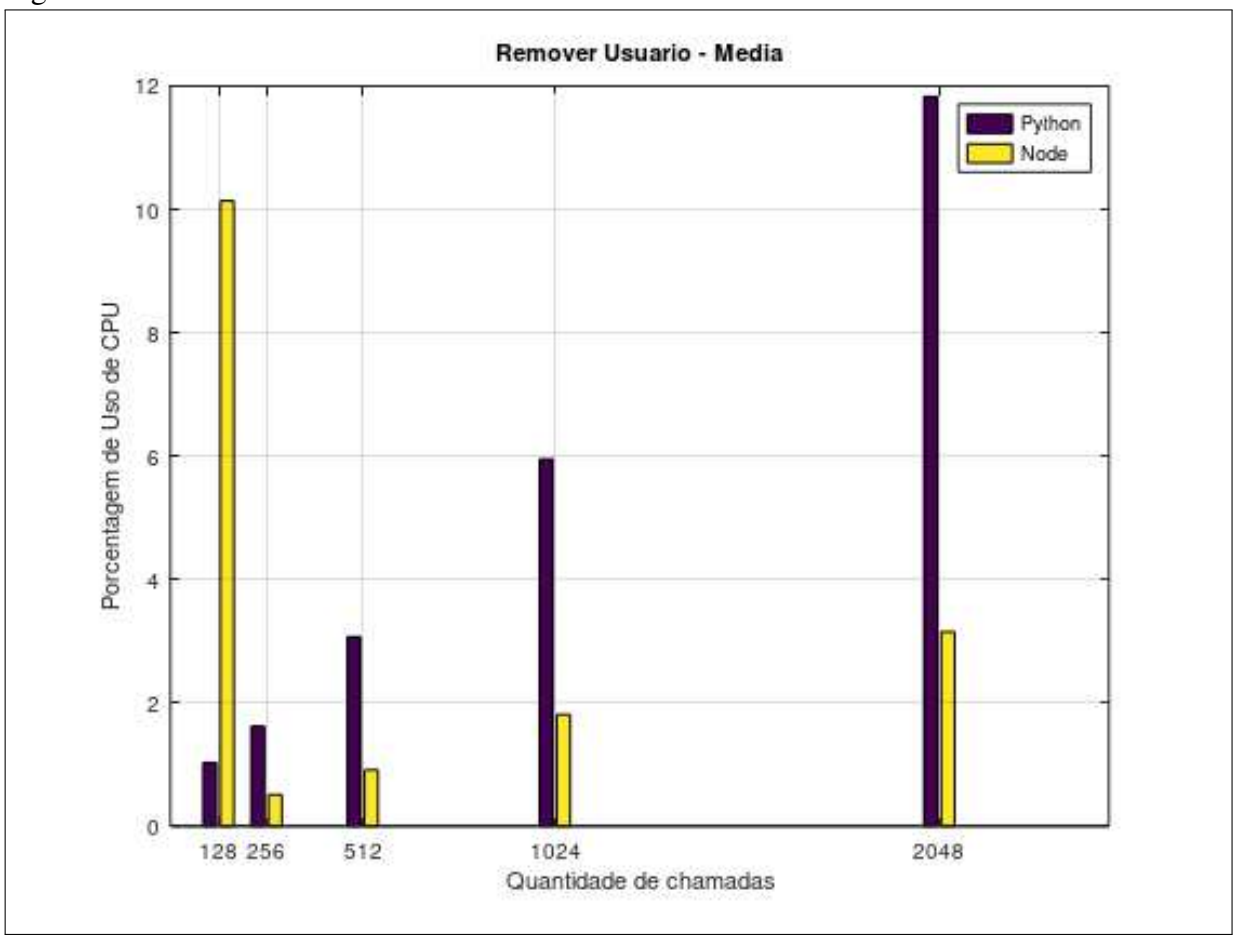
Tabela 5 – Delete média

	128	**256**	**512**	**1024**	**2048**
Python	1,03%	1,62%	3,07%	5,95%	11,83%
Node	10,14%	0,51%	0,91%	1,81%	3,15%

Tabela 6 – Delete mediana

	128	**256**	**512**	**1024**	**2048**
Python	1%	2%	3%	7%	13,9%
Node	0%	0%	1%	2%	3%

Figura 6 – Remover Usuário - Média de uso da CPU



Já nas tabelas 7 e 8 fica ainda mais evidente o motivo do desvio no primeiro cenário, tendo a implementação chegado a um pico de 100% de consumo da CPU. Esse pico causou a discrepância nessa rodada de experimentação mas não identificamos como uma tendência, uma vez que aconteceu apenas em uma das rodadas de experimentação e apenas em um dos endpoints. Outro ponto que as tabelas 7 e 8 também trazem a tona foram os valores elevados no Node na 5 rodada de experimentação (2048 requisições), que diferente do que havíamos visto na sub sessão 5.1.1 teve um consumo de CPU bem mais elevado que o da implementação em Python.

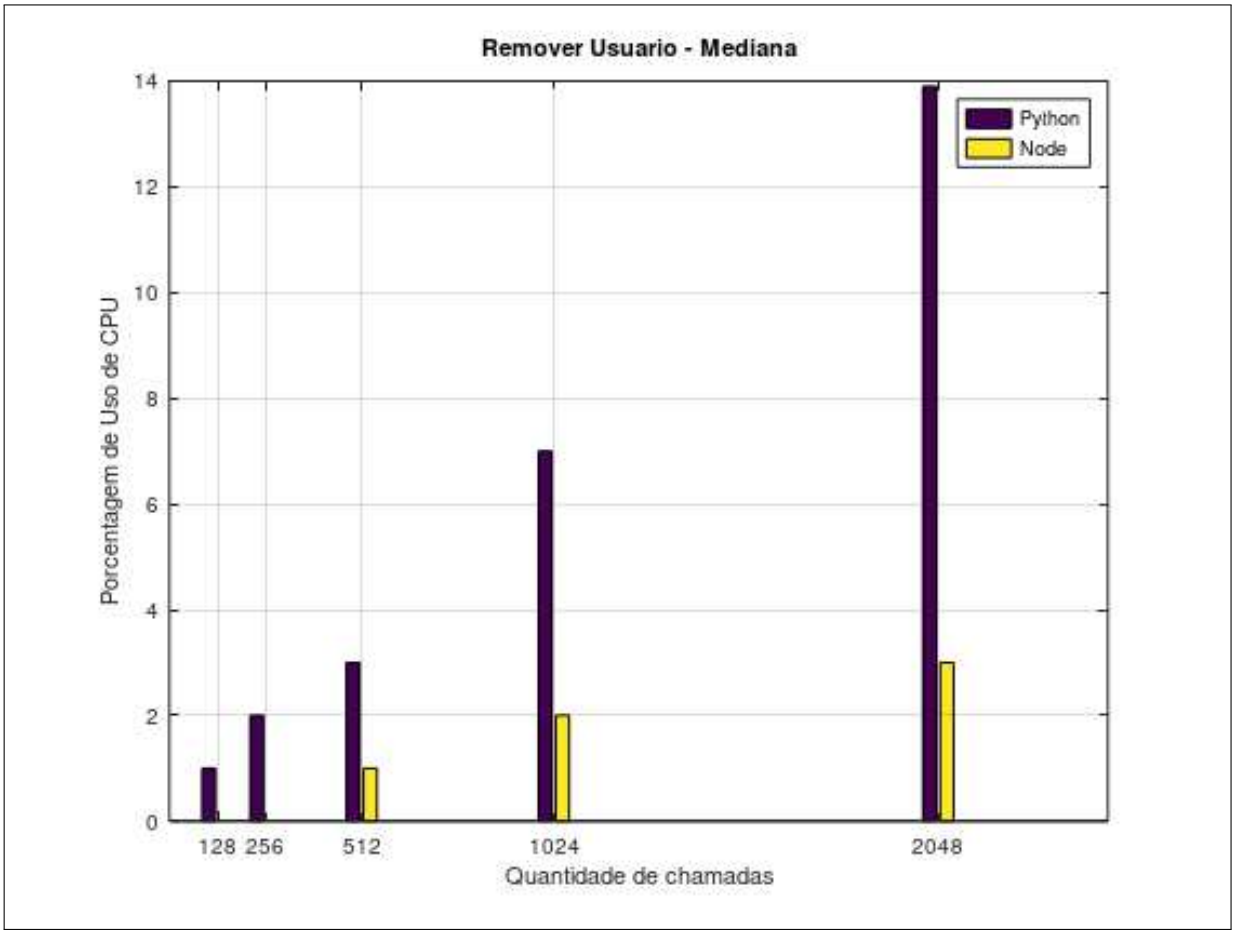
Tabela 7 – Delete máximo

	128	**256**	**512**	**1024**	**2048**
Python	3%	4%	6%	8,9%	17,6%
Node	100%	2%	4%	10%	28,7%

Tabela 8 – Delete (Min+Max)/2

	128	**256**	**512**	**1024**	**2048**
Python	1,5%	2%	3%	4,45%	8,8%
Node	50%	1%	2%	5%	14,35%

Figura 7 – Remover Usuário - Mediana de uso da CPU



5.1.3 Endpoint recuperar um usuário

Nessa sub sessão temos os resultados das análises do consumo de CPU quando olhamos a implementação do endpint de consulta de 1 usuário. No geral, para essa rota tivemos um comportamento similar ao observado na sub sessão 5.1.1 com a implementação em Node consumindo menos recursos em todas as iterações da experimentação e com a discrepância sendo cada vez mais evidente quanto mais requisições são realizadas. Nas tabelas 9 e 10 também destacamos a diferença dos valores de consumo quando olhamos para as amostras de 2048 requisições, onde a diferença estava na faixa dos 8% entre as duas implementações.

Tabela 9 – Recuperar um usuário - Média

	128	**256**	**512**	**1024**	**2048**
Python	0,91%	1,6%	3,07%	6,29%	10,86%
Node	0,34%	0,49%	0,7%	1,5%	2,52%

Outro ponto curioso que percebemos nas tabelas 11 e 12 foi que na primeira iteração o pico foi exatamente igual.

Figura 8 – Recuperar um Usuário - Média de uso da CPU

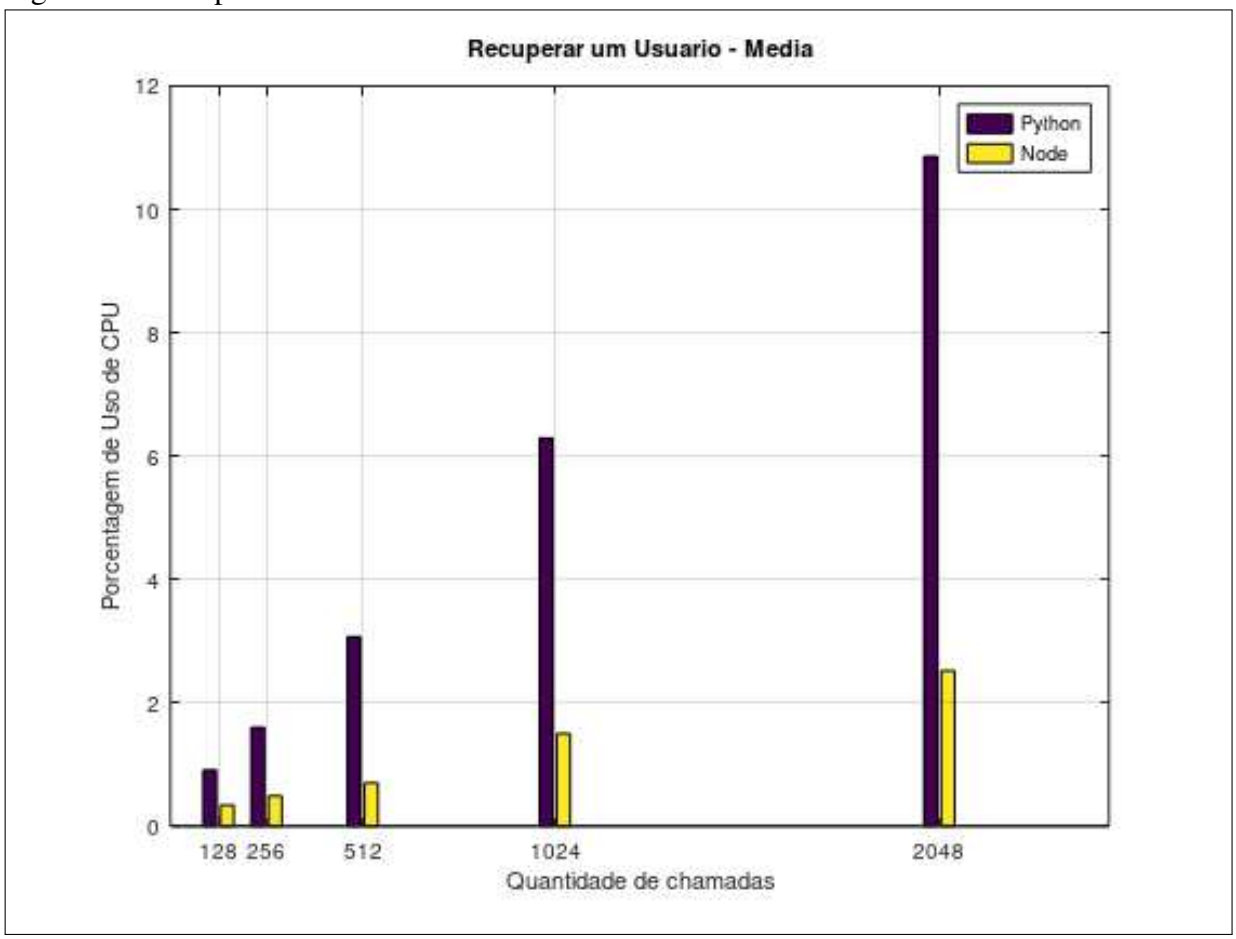


Tabela 10 – Recuperar um usuário - Mediana

	128	**256**	**512**	**1024**	**2048**
Python	1%	2%	3%	7%	13,1%
Node	0%	0%	1%	1%	3%

Tabela 11 – Recuperar um usuário - Máximo

	128	**256**	**512**	**1024**	**2048**
Python	2,9%	4,9%	5,9%	9%	17%
Node	2,9%	3%	2%	5,9%	7,8%

Tabela 12 – Recuperar um usuário - (Min+Max)/2

	128	**256**	**512**	**1024**	**2048**
Python	1,45%	2,45%	2,95%	4,5%	8,5%
Node	1,45%	1,5%	1%	2,95%	3,9%

5.1.4 Endpoint recuperar todos os usuários

As Tabelas 13, 14 e 15 fornecem uma visão abrangente do comportamento do consumo de CPU para as implementações em Python e Node.js no endpoint que recupera todos os usuários, à medida que a carga de trabalho é incrementada. Primeiramente, ao observar a

Figura 9 – Recuperar um Usuário - Mediana de uso da CPU

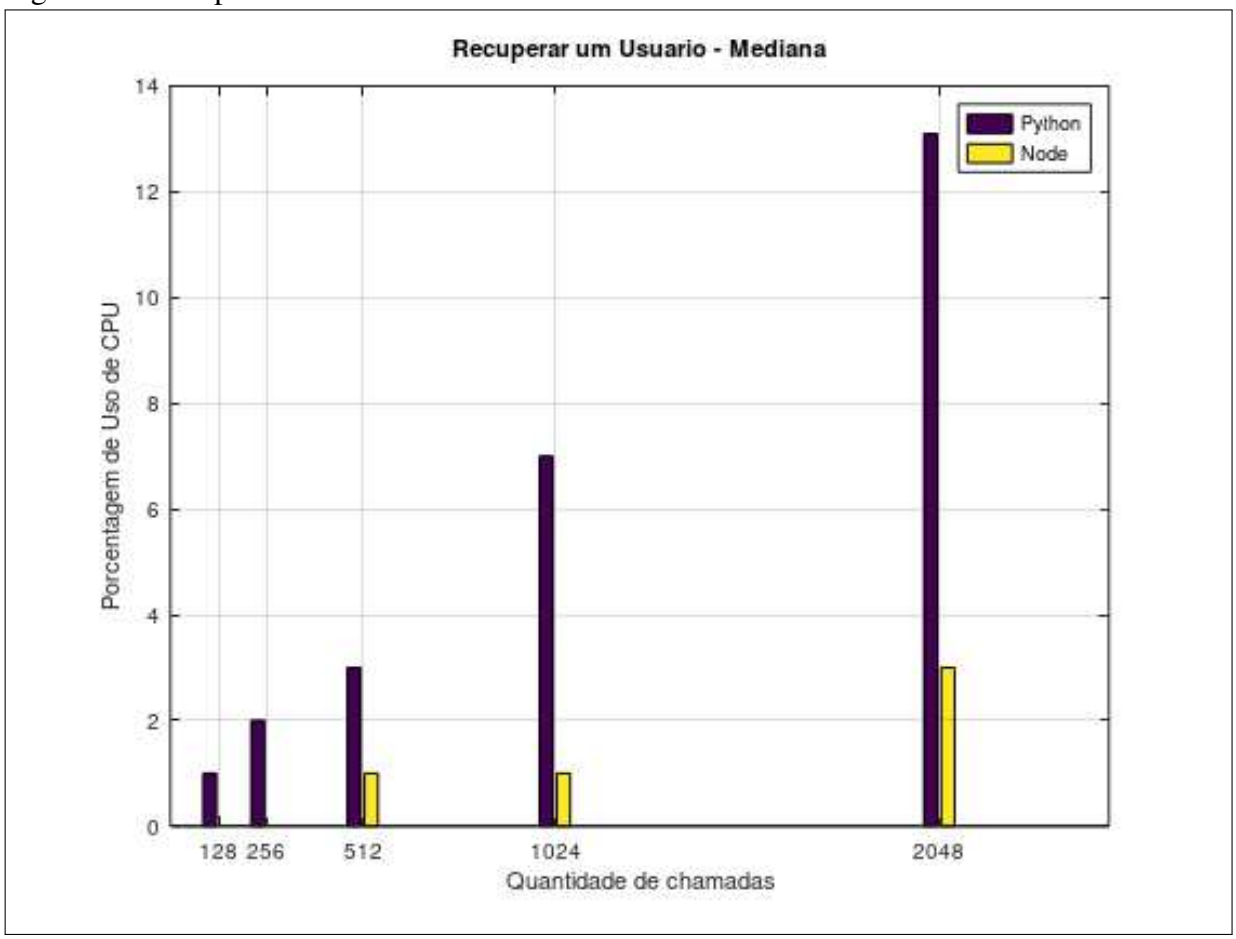


Tabela 13, que apresenta a média do consumo de CPU, percebemos que a implementação em Python tende a exigir mais recursos em comparação com o Node.js em todas as cargas avaliadas. Esse padrão é consistente com a ideia de que Python, sendo uma linguagem de alto nível, pode resultar em um consumo mais elevado.

Tabela 13 – Recuperar todos os usuários - Média

	128	**256**	**512**	**1024**	**2048**
Python	1,2%	2,31%	4,68%	11,02%	25,05%
Node	0,36%	0,67%	1,38%	4,27%	15,14%

Ao analisar a Tabela 14, que expõe a mediana do consumo de CPU, notamos uma tendência semelhante, com a implementação em Python mantendo níveis mais altos de consumo em comparação com o Node.js. A mediana, ao contrário da média, é menos sensível a valores extremos, fornecendo uma perspectiva mais estável do comportamento do sistema. O aumento progressivo nos valores destaca a influência direta da carga na alocação de processamento, sugerindo que ambas as implementações são afetadas de maneira proporcional ao aumento da carga.

Figura 10 – Recuperar todos Usuários - Média de uso da CPU

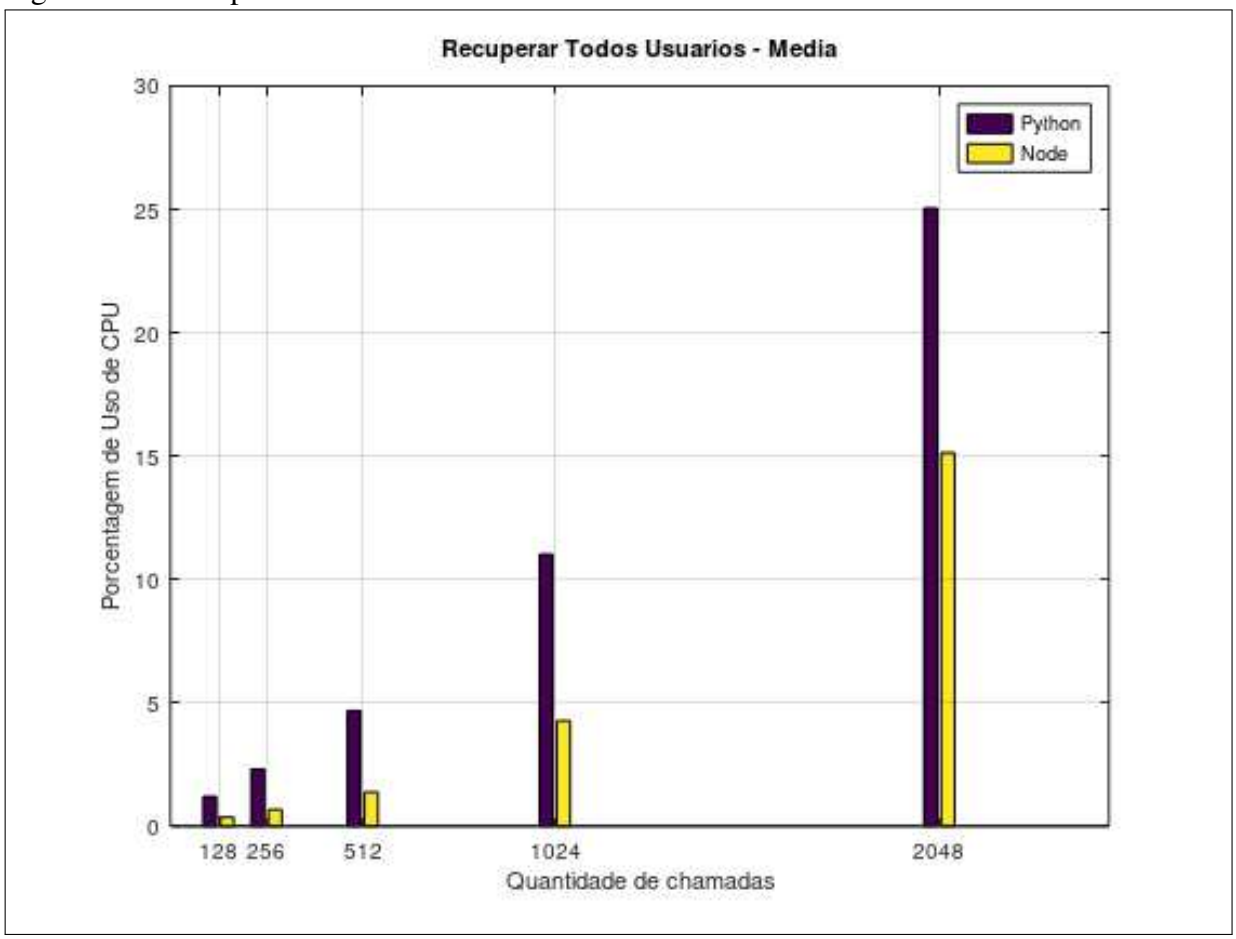


Tabela 14 – Recuperar todos os usuários - Mediana

	128	**256**	**512**	**1024**	**2048**
Python	1%	2%	5,1%	12,95%	34%
Node	0%	1%	1%	5%	17,2%

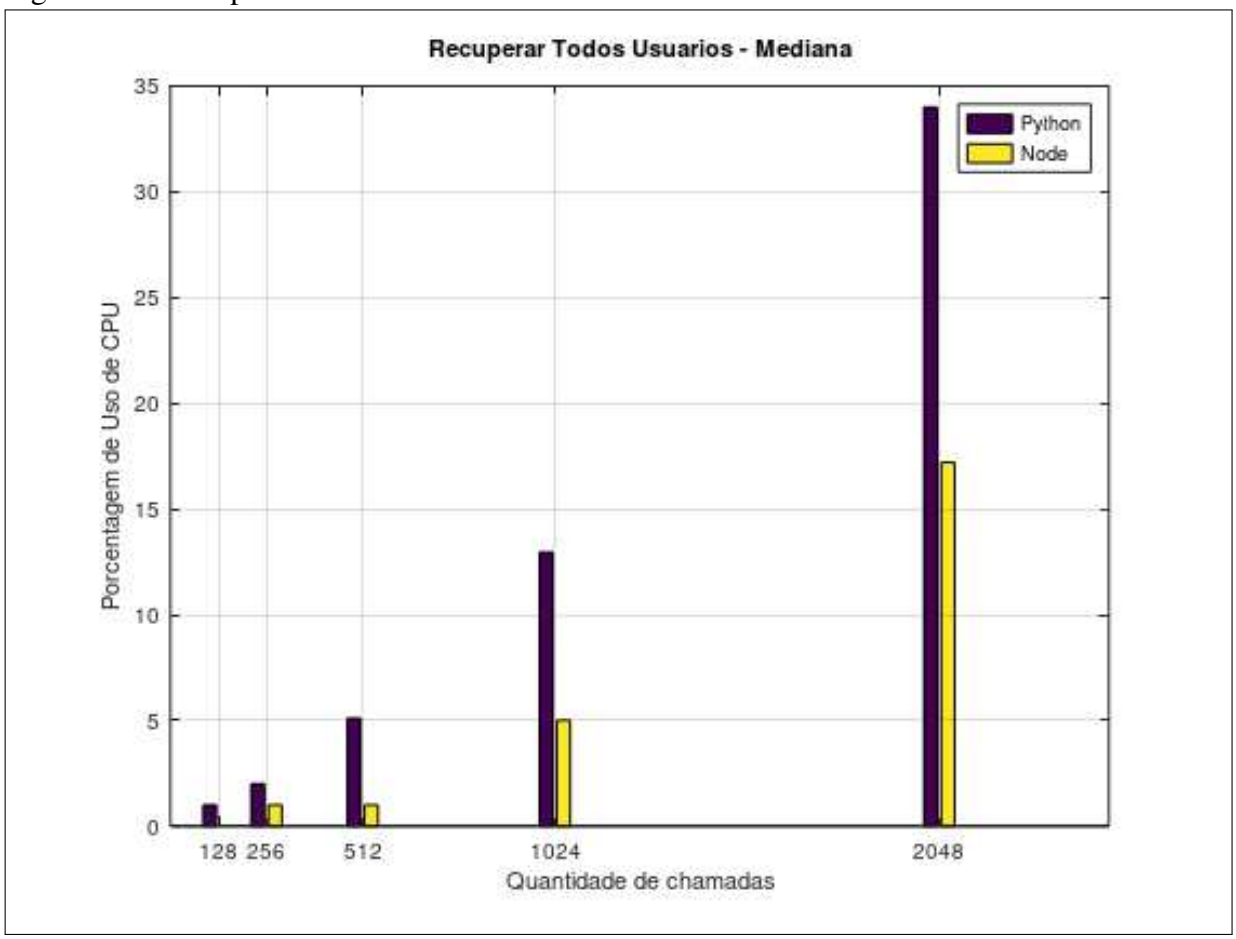
Por fim, a Tabela 15, que retrata o valor máximo do consumo de CPU, confirma a tendência geral de que a implementação em Python atinge valores mais elevados em comparação com o Node.js. Os dados indicam que, em situações de carga mais intensa, a implementação em Python pode experimentar picos significativos no consumo de CPU, o que pode ser crucial para aplicações sensíveis a limitações de recursos.

Tabela 15 – Recuperar todos os usuários - Máximo

	128	**256**	**512**	**1024**	**2048**
Python	9,9%	5,9%	8,7%	17,3%	41,2%
Node	2,9%	3%	4%	8,9%	21,6%

Em conjunto, essas análises das três tabelas sugerem que, embora ambas as implementações mostrem um aumento no consumo de CPU com a carga crescente, a implementação

Figura 11 – Recuperar todos Usuários - Mediana de uso da CPU



em Node.js tende a ser mais eficiente em termos de utilização de recursos, especialmente em comparação com a implementação em Python

Tabela 16 – Recuperar todos os usuários - (Min+Max)/2

	128	**256**	**512**	**1024**	**2048**
Python	4,95%	2,95%	4,35%	8,65%	20,06%
Node	1,45%	1,5%	2%	4,45%	10,8%

5.2 Consumo de Memória

5.2.1 Endpoint Criar um usuário

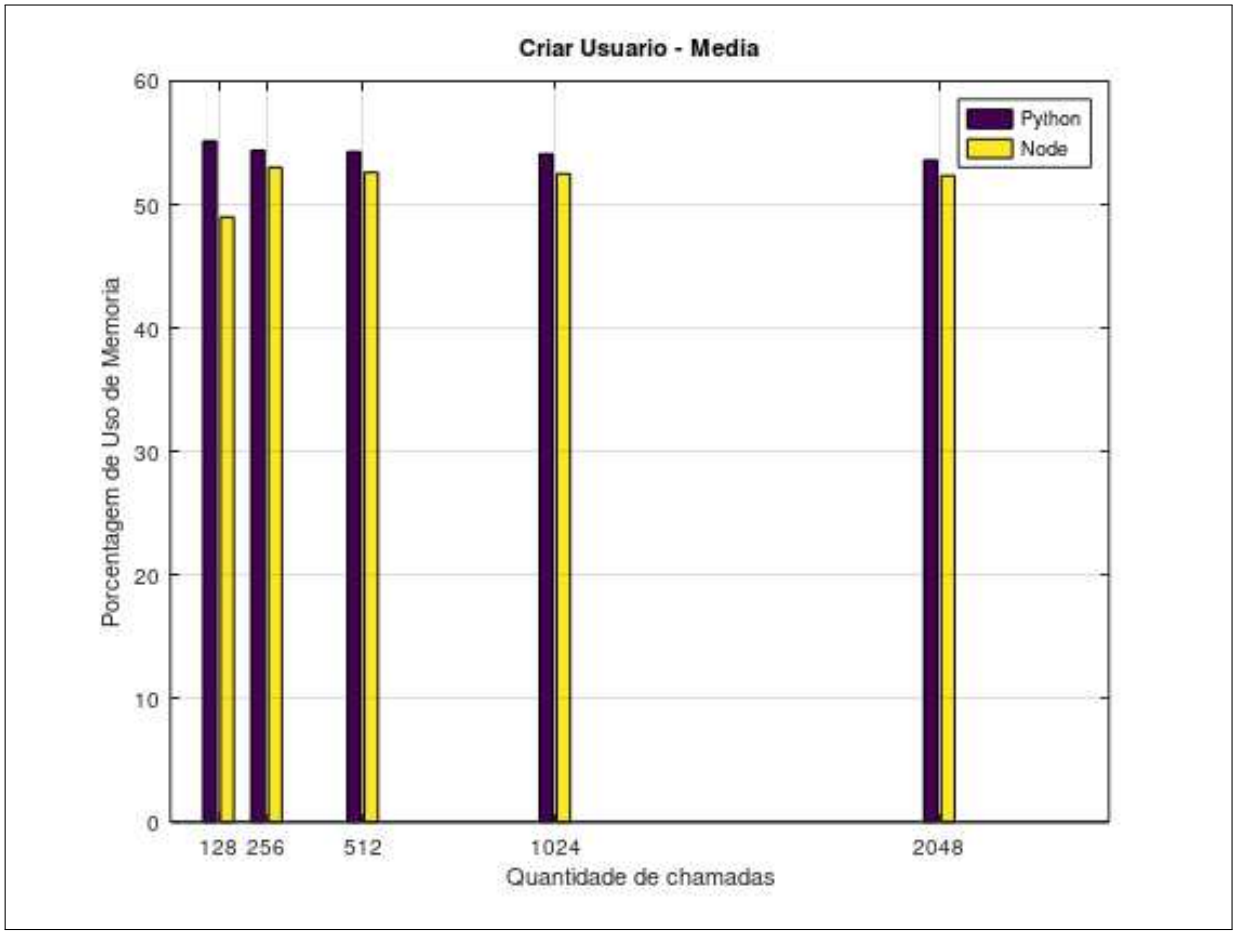
As Tabelas 17, 18 e 19 oferecem uma análise detalhada do consumo de memória para as implementações em Python e Node.js no contexto do endpoint de criação de usuário, à medida que a carga de trabalho é incrementada. Primeiramente, ao observar a Tabela 17, que apresenta a média do consumo de memória, nota-se que ambas as implementações mantêm

níveis relativamente estáveis de consumo à medida que a carga aumenta. A implementação em Python apresenta uma ligeira redução no consumo, indicando uma eficiência consistente mesmo com cargas mais intensas, enquanto a implementação em Node.js exibe uma variação um pouco mais pronunciada, embora permaneça comparativamente eficiente.

Tabela 17 – Create Média

	128	**256**	**512**	**1024**	**2048**
Python	55,15%	54,4%	54,28%	54,1%	53,6%
Node	49%	53,02%	52,61%	52,5%	52,35%

Figura 12 – Criar Usuário - Média de uso da Memória



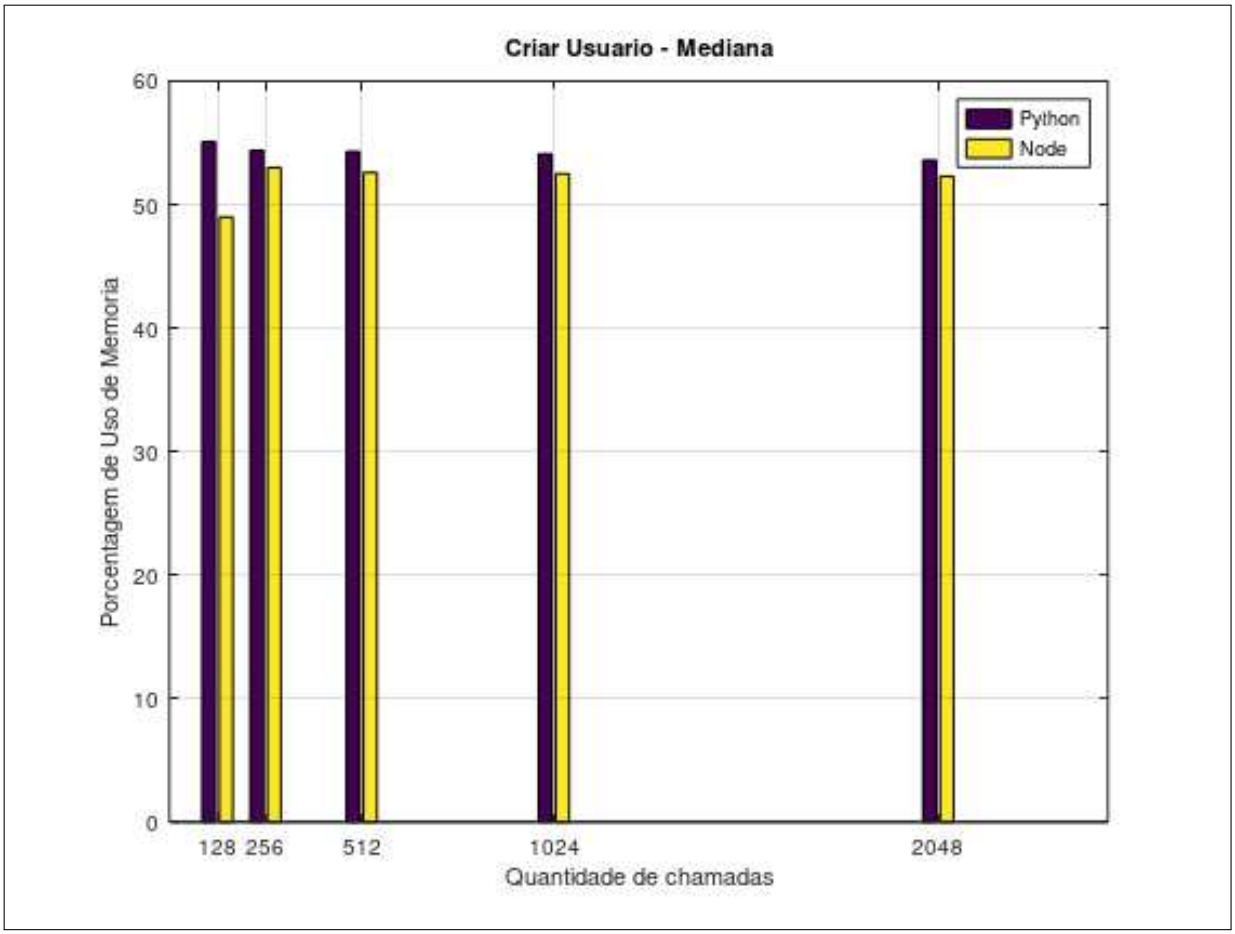
Ao analisar a Tabela 18, que mostra a mediana do consumo de memória, observa-se um comportamento semelhante. Ambas as implementações mantêm níveis estáveis de consumo, com a implementação em Python demonstrando uma leve redução à medida que a carga aumenta. A mediana, sendo menos sensível a valores extremos, reforça a estabilidade nas alocações de memória para ambos os casos, independentemente da carga.

Por fim, a Tabela 19, que destaca o valor máximo do consumo de memória, mostra

Tabela 18 – Create Mediana

	128	**256**	**512**	**1024**	**2048**
Python	55,1%	54,4%	54,3%	54,1%	53,6%
Node	49%	53%	52,6%	52,5%	52,3%

Figura 13 – Criar Usuário - Mediana de uso da Memória



uma consistência notável. Ambas as implementações mantêm valores máximos relativamente estáveis à medida que a carga cresce, indicando uma robustez no gerenciamento de recursos mesmo em situações de alta demanda. A implementação em Python, embora possua valores máximos um pouco mais elevados, mantém uma tendência geral de estabilidade.

Tabela 19 – Create Máximo

	128	**256**	**512**	**1024**	**2048**
Python	55,8%	54,4%	54,3%	54,1%	53,6%
Node	49%	53,1%	52,7%	52,5%	52,4%

Em resumo, as análises das Tabelas 17, 18 e 19 sugerem que ambas as implementações, tanto em Python quanto em Node.js, apresentam um desempenho consistente e eficiente no consumo de memória no contexto do endpoint de criação de usuário, mesmo diante de cargas

de trabalho variáveis. A escolha entre as linguagens pode depender de outros fatores, como a facilidade de desenvolvimento, a manutenibilidade do código e os requisitos específicos do projeto, mas, em termos de consumo de memória, ambas as opções demonstram estabilidade e eficiência.

Tabela 20 – Create (Min+Max)/2

	128	**256**	**512**	**1024**	**2048**
Python	55,45%	54,4%	54,25%	54,05%	53,6%
Node	49%	53,05%	52,65%	52,5%	52,35%

5.2.2 Endpoint deletar um usuário

As Tabelas 21, 22 e 23 oferecem uma análise detalhada do consumo de memória para as implementações em Python e Node.js no contexto do endpoint de remoção de usuário, à medida que a carga de trabalho é incrementada. Ao observar a Tabela 21, que apresenta a média do consumo de memória, percebe-se que ambas as implementações mantêm níveis relativamente estáveis de consumo à medida que a carga aumenta. A implementação em Python mostra uma tendência geral de redução no consumo, indicando uma eficiência consistente mesmo sob cargas mais intensas. Por outro lado, a implementação em Node.js também mantém níveis eficientes, embora com uma variação um pouco mais acentuada.

Tabela 21 – Delete Média

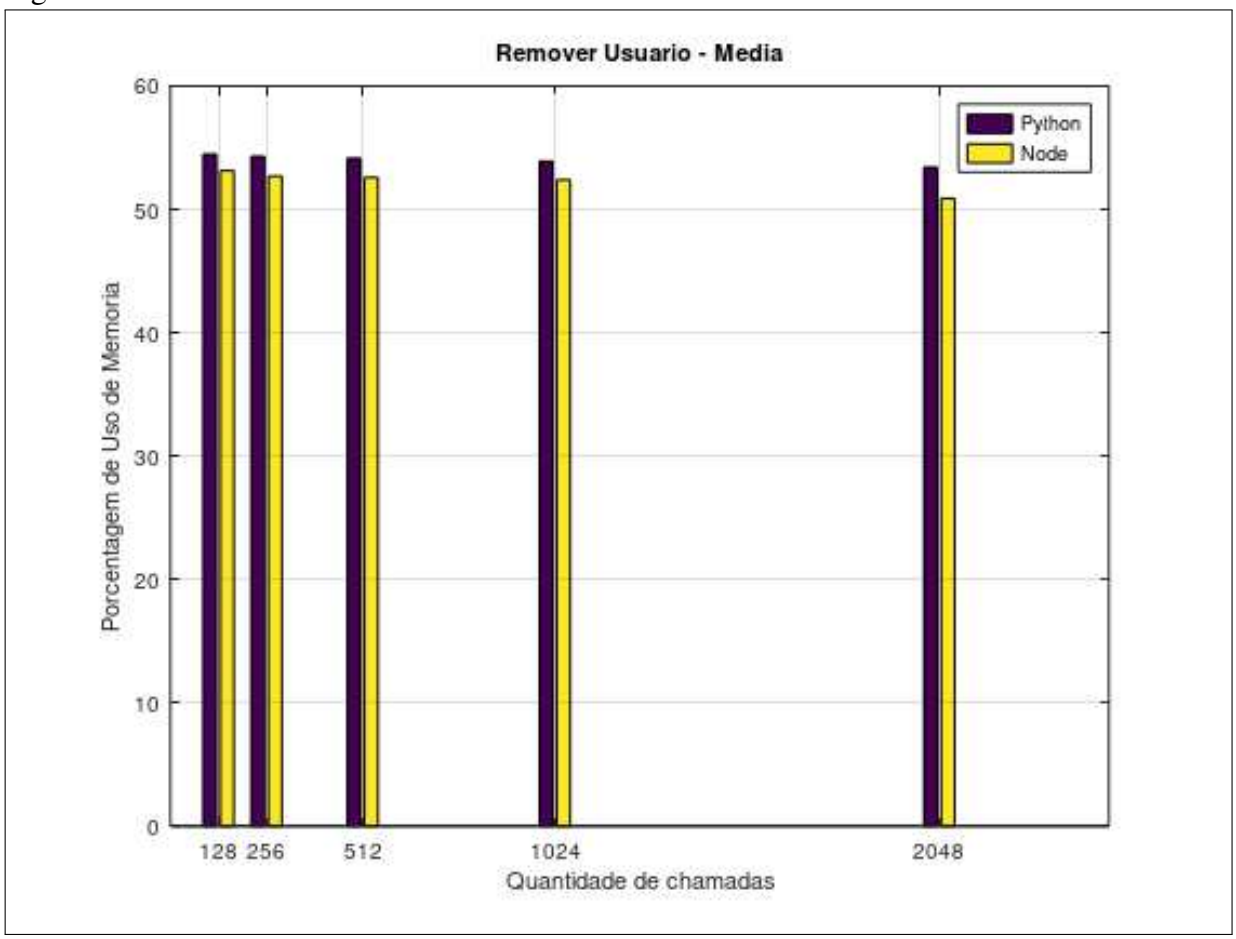
	128	**256**	**512**	**1024**	**2048**
Python	54,48%	54,3%	54,15%	53,9%	53,42%
Node	53,14%	52,7%	52,59%	52,4%	50,9%

Analisando a Tabela 22, que mostra a mediana do consumo de memória, observa-se um comportamento semelhante. Ambas as implementações mantêm níveis estáveis de consumo, com a implementação em Python demonstrando uma ligeira redução à medida que a carga aumenta. A mediana, sendo menos sensível a valores extremos, sugere uma estabilidade nas alocações de memória para ambos os casos, independentemente da carga de trabalho.

Tabela 22 – Delete mediana

	128	**256**	**512**	**1024**	**2048**
Python	54,4%	54,3%	54,2%	53,9%	53,4%
Node	53,3%	52,7%	52,6%	52,4%	50,4%

Figura 14 – Remover Usuário - Média de uso da Memória



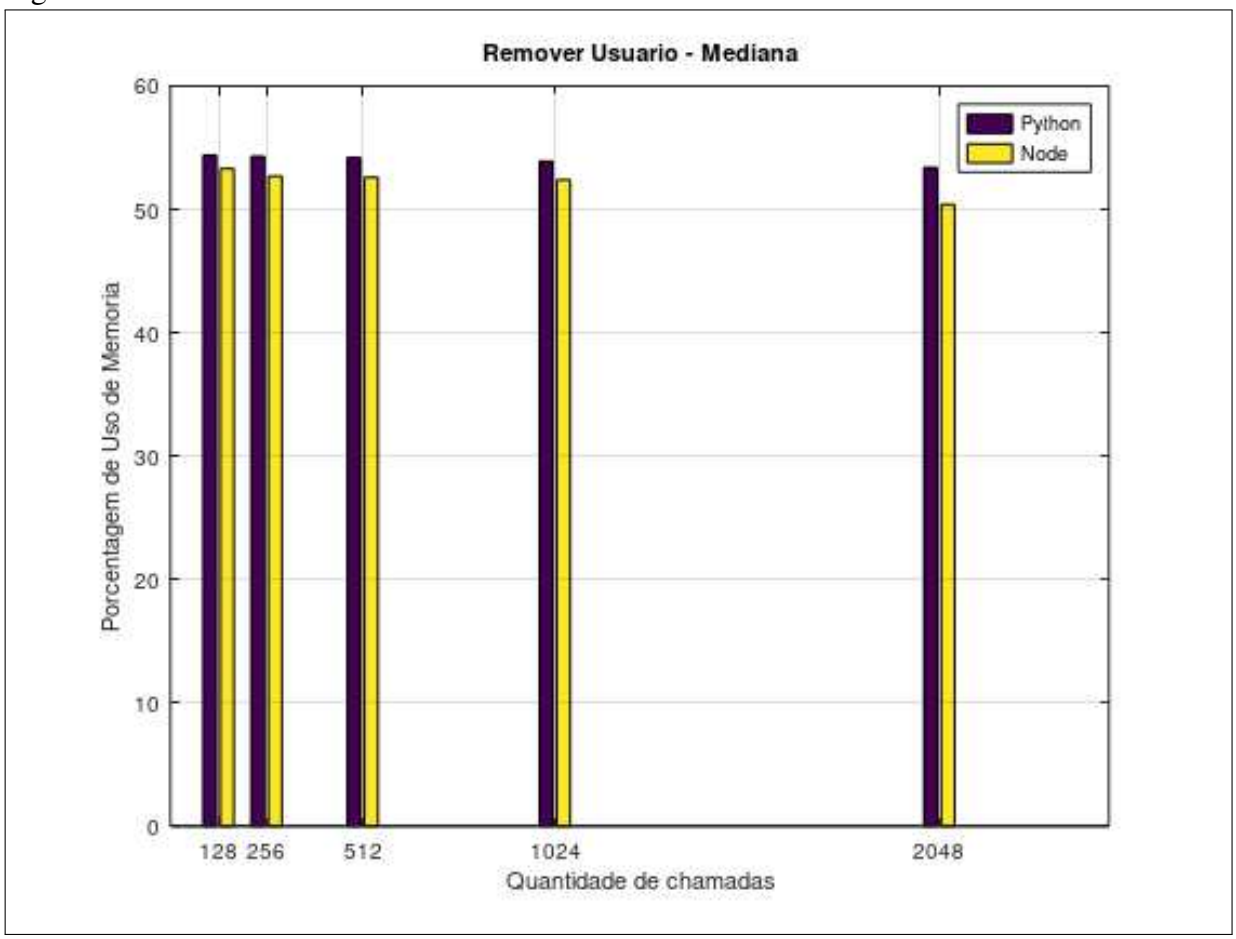
Finalmente, ao considerar a Tabela 23, que destaca o valor máximo do consumo de memória, observa-se consistência nas implementações. Ambas mantêm valores máximos relativamente estáveis à medida que a carga cresce, indicando robustez no gerenciamento de recursos mesmo em situações de alta demanda. A implementação em Python, embora com valores máximos ligeiramente mais elevados, mantém uma tendência geral de estabilidade, enquanto a implementação em Node.js mostra uma variação menor nos valores máximos.

Tabela 23 – Delete máximo

	128	**256**	**512**	**1024**	**2048**
Python	54,6%	54,3%	54,2%	53,9%	53,5%
Node	54%	52,7%	52,6%	52,4%	52,4%

Em resumo, as análises das Tabelas 21, 22 e 23 sugerem que ambas as implementações, em Python e Node.js, demonstram um desempenho eficiente e consistente no consumo de memória no contexto do endpoint de remoção de usuário, mesmo diante de cargas de trabalho variáveis. A escolha entre as linguagens pode depender de outros fatores, como preferências de desenvolvimento, manutenibilidade do código e requisitos específicos do projeto, mas, em

Figura 15 – Remover Usuário - Mediana de uso da Memória



termos de consumo de memória, ambas as opções exibem estabilidade e eficiência.

Tabela 24 – Delete (Min+Max)/2

	128	**256**	**512**	**1024**	**2048**
Python	54,5%	54,3%	54,15%	53,9%	53,45%
Node	52,35%	52,7%	52,55%	52,4%	51,35%

5.2.3 Endpoint recuperar um usuário

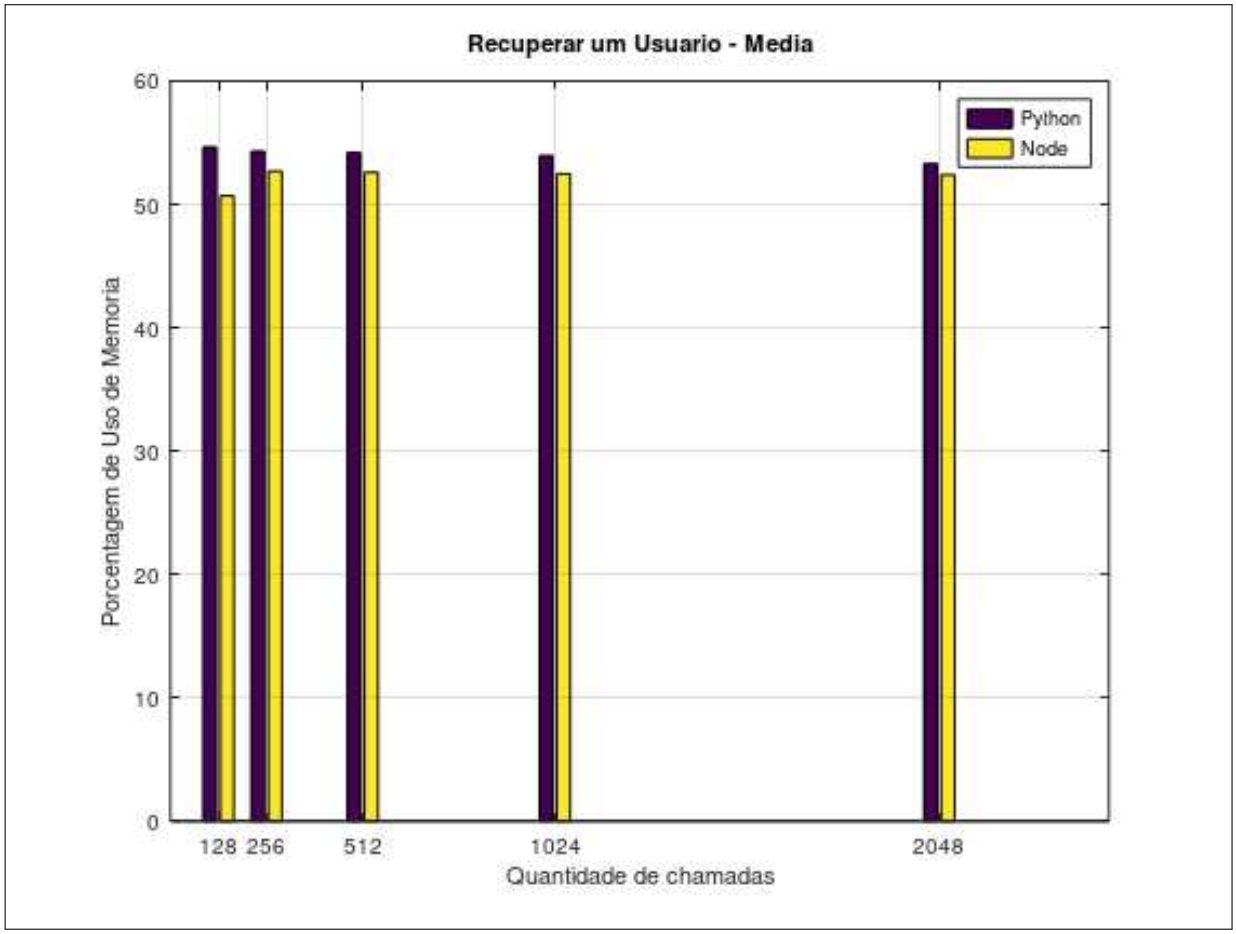
As Tabelas 25, 26 e 27 proporcionam uma visão detalhada do consumo de memória para as implementações em Python e Node.js no contexto do endpoint de consulta de um usuário, à medida que a carga de trabalho é incrementada. Ao examinar a Tabela 25, que apresenta a média do consumo de memória, observamos que ambas as implementações mantêm níveis relativamente estáveis de consumo à medida que a carga aumenta. A implementação em Python exibe uma tendência de leve redução no consumo, indicando uma eficiência consistente mesmo em cargas mais intensas, enquanto a implementação em Node.js mantém níveis eficientes, com

uma variação um pouco mais acentuada.

Tabela 25 – Recuperar um usuário - Média

	128	**256**	**512**	**1024**	**2048**
Python	54,65%	54,31%	54,2%	53,95%	53,3%
Node	50,7%	52,7%	52,6%	52,48%	52,4%

Figura 16 – Recuperar um Usuário - Média de uso da Memória

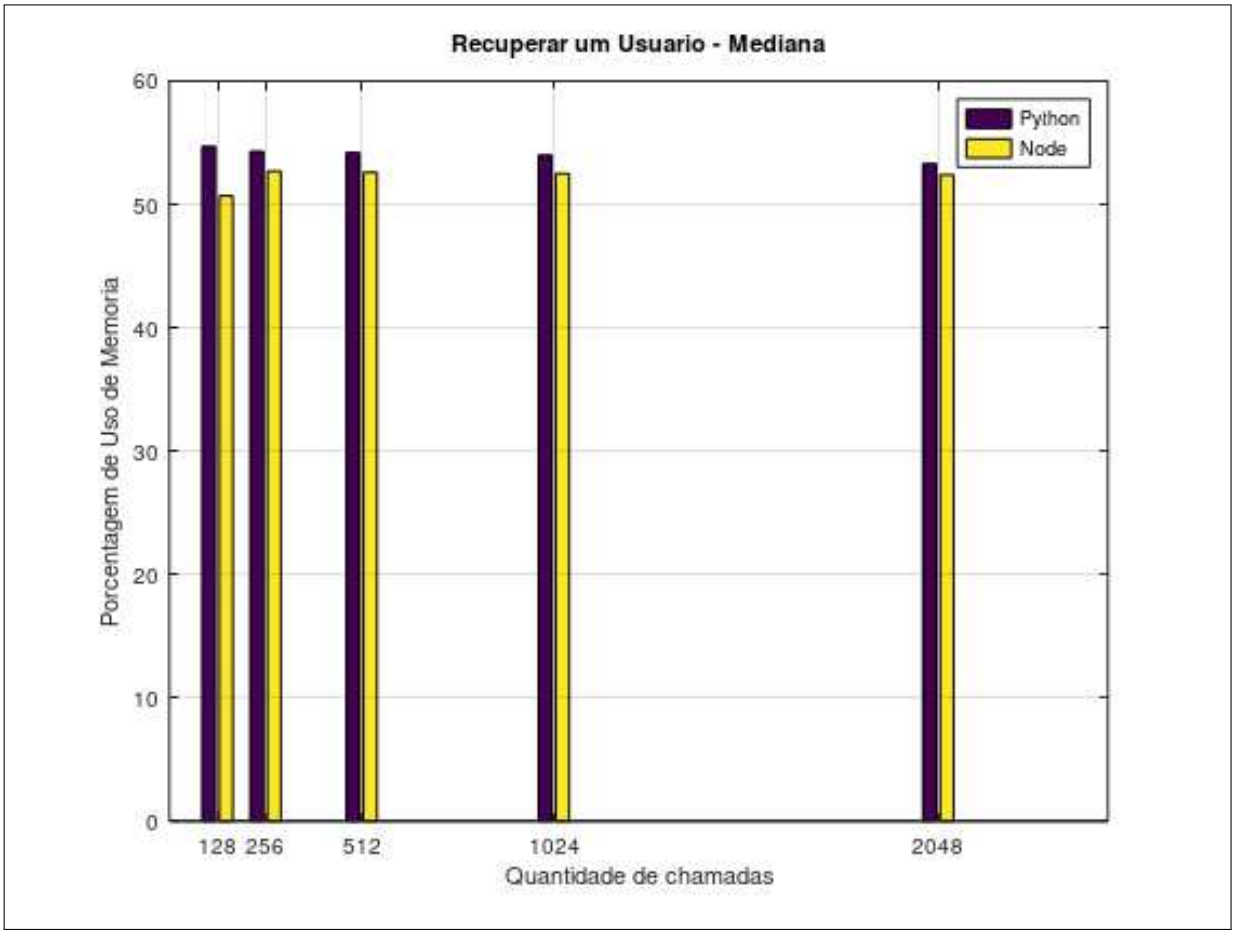


Ao analisar a Tabela 26, que mostra a mediana do consumo de memória, nota-se um comportamento semelhante. Ambas as implementações mantêm níveis estáveis de consumo, com a implementação em Python apresentando uma ligeira redução à medida que a carga aumenta. A estabilidade na mediana sugere consistência nas alocações de memória para ambos os casos, independentemente da carga de trabalho.

Tabela 26 – Recuperar um usuário - Mediana

	128	**256**	**512**	**1024**	**2048**
Python	54,7%	54,3%	54,2%	54%	53,3%
Node	50,7%	52,7%	52,6%	52,5%	52,4%

Figura 17 – Recuperar um Usuário - Mediana de uso da Memória



Considerando a Tabela 27, que destaca o valor máximo do consumo de memória, percebemos uma consistência notável nas implementações. Ambas mantêm valores máximos relativamente estáveis à medida que a carga cresce, indicando robustez no gerenciamento de recursos mesmo em situações de alta demanda. A implementação em Python, embora com valores máximos ligeiramente mais elevados, mantém uma tendência geral de estabilidade, enquanto a implementação em Node.js mostra uma variação menor nos valores máximos.

Tabela 27 – Recuperar um usuário - Máximo

	128	**256**	**512**	**1024**	**2048**
Python	54,7%	54,4%	54,2%	54%	53,4%
Node	50,7%	52,7%	52,6%	52,5%	52,4%

Em resumo, as análises das Tabelas 25, 26 e 27 sugerem que ambas as implementações, em Python e Node.js, apresentam um desempenho eficiente e consistente no consumo de memória no contexto do endpoint de consulta de um usuário, mesmo diante de cargas de trabalho variáveis. A escolha entre as linguagens pode depender de outros fatores, como preferências de desenvolvimento, manutenibilidade do código e requisitos específicos do projeto, mas, em

termos de consumo de memória, ambas as opções exibem estabilidade e eficiência.

Tabela 28 – Recuperar um usuário - (Min+Max)/2

	128	**256**	**512**	**1024**	**2048**
Python	54,65%	54,35%	54,2%	53,95%	53,3%
Node	50,7%	52,7%	52,6%	52,45%	52,4%

5.2.4 Endpoint recuperar todos os usuários

As Tabelas 29, 30 e 31 oferecem uma análise abrangente do consumo de memória para as implementações em Python e Node.js no contexto do endpoint de consulta de todos os usuários, à medida que a carga de trabalho é incrementada. Observando a Tabela 29, que apresenta a média do consumo de memória, notamos que ambas as implementações mantêm níveis relativamente estáveis à medida que a carga aumenta. A implementação em Python exibe uma tendência de leve redução no consumo, indicando uma eficiência consistente mesmo em cargas mais intensas, enquanto a implementação em Node.js também mantém níveis eficientes, embora com uma variação um pouco mais acentuada.

Tabela 29 – Recuperar todos os usuários - Média

	128	**256**	**512**	**1024**	**2048**
Python	54,76%	54,4%	54,2%	53,96%	53,59%
Node	50,7%	52,7%	52,6%	52,4%	52,38%

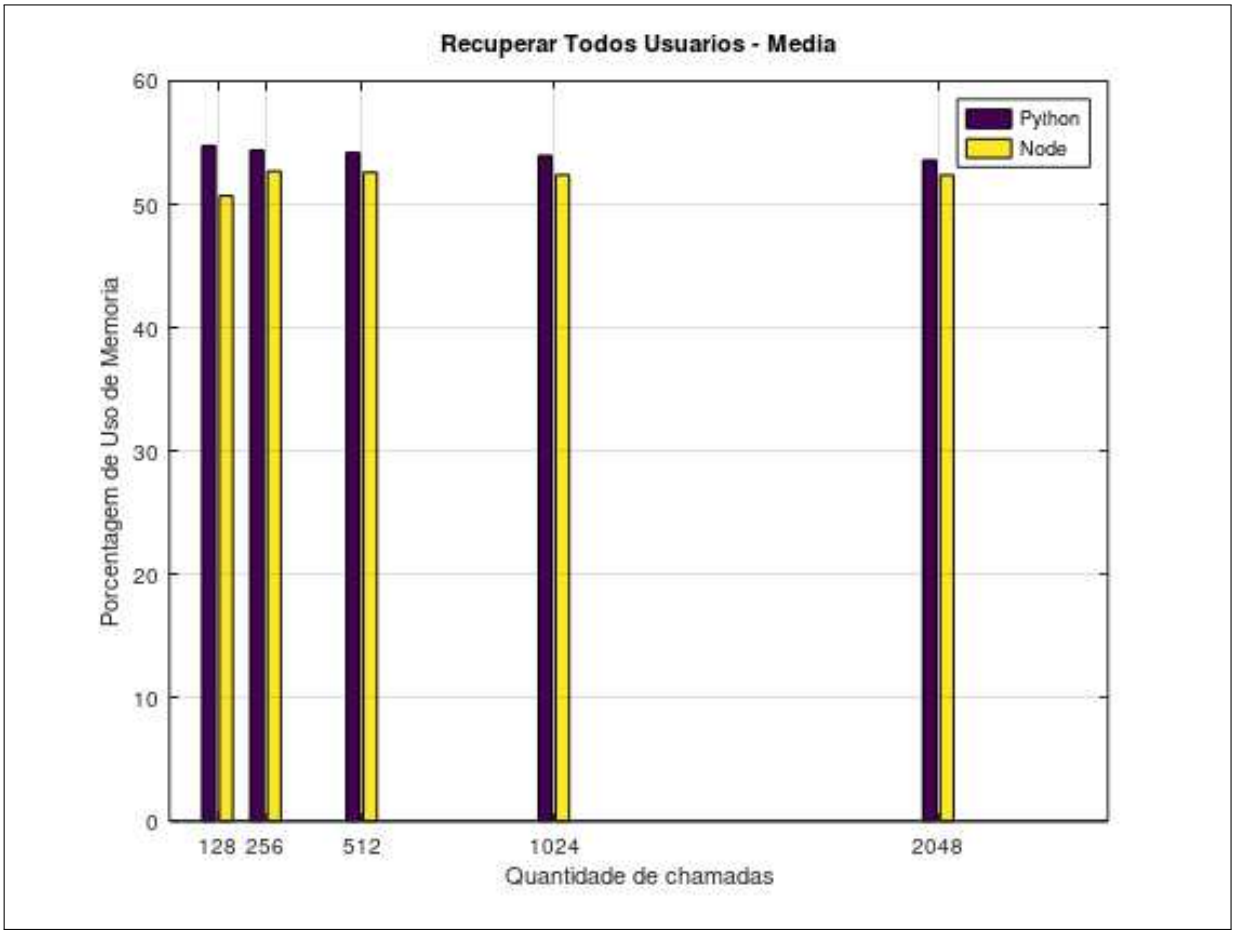
Ao analisar a Tabela 30, que mostra a mediana do consumo de memória, percebe-se um comportamento semelhante. Ambas as implementações mantêm níveis estáveis de consumo, com a implementação em Python apresentando uma ligeira redução à medida que a carga aumenta. A estabilidade na mediana sugere consistência nas alocações de memória para ambos os casos, independentemente da carga de trabalho.

Tabela 30 – Recuperar todos os usuários - Mediana

	128	**256**	**512**	**1024**	**2048**
Python	54,7%	54,4%	54,2%	54%	53,6%
Node	50,7%	52,7%	52,6%	52,4%	52,4%

Considerando a Tabela 31, que destaca o valor máximo do consumo de memória, observa-se uma consistência notável nas implementações. Ambas mantêm valores máximos relativamente estáveis à medida que a carga cresce, indicando robustez no gerenciamento de

Figura 18 – Recuperar Todos Usuários - Média de uso da Memória



recursos mesmo em situações de alta demanda. A implementação em Python, embora com valores máximos ligeiramente mais elevados, mantém uma tendência geral de estabilidade, enquanto a implementação em Node.js mostra uma variação menor nos valores máximos.

Tabela 31 – Recuperar todos os usuários - Máximo

	128	**256**	**512**	**1024**	**2048**
Python	55%	54,4%	54,2%	54%	53,6%
Node	50,7%	52,7%	52,6%	52,4%	52,4%

Em resumo, as análises das Tabelas 29, 30 e 31 sugerem que ambas as implementações, em Python e Node.js, apresentam um desempenho eficiente e consistente no consumo de memória no contexto do endpoint de consulta de todos os usuários, mesmo diante de cargas de trabalho variáveis. A escolha entre as linguagens pode depender de outros fatores, como preferências de desenvolvimento, manutenibilidade do código e requisitos específicos do projeto, mas, em termos de consumo de memória, ambas as opções exibem estabilidade e eficiência.

Figura 19 – Recuperar Todos Usuários - Mediana de uso da Memória

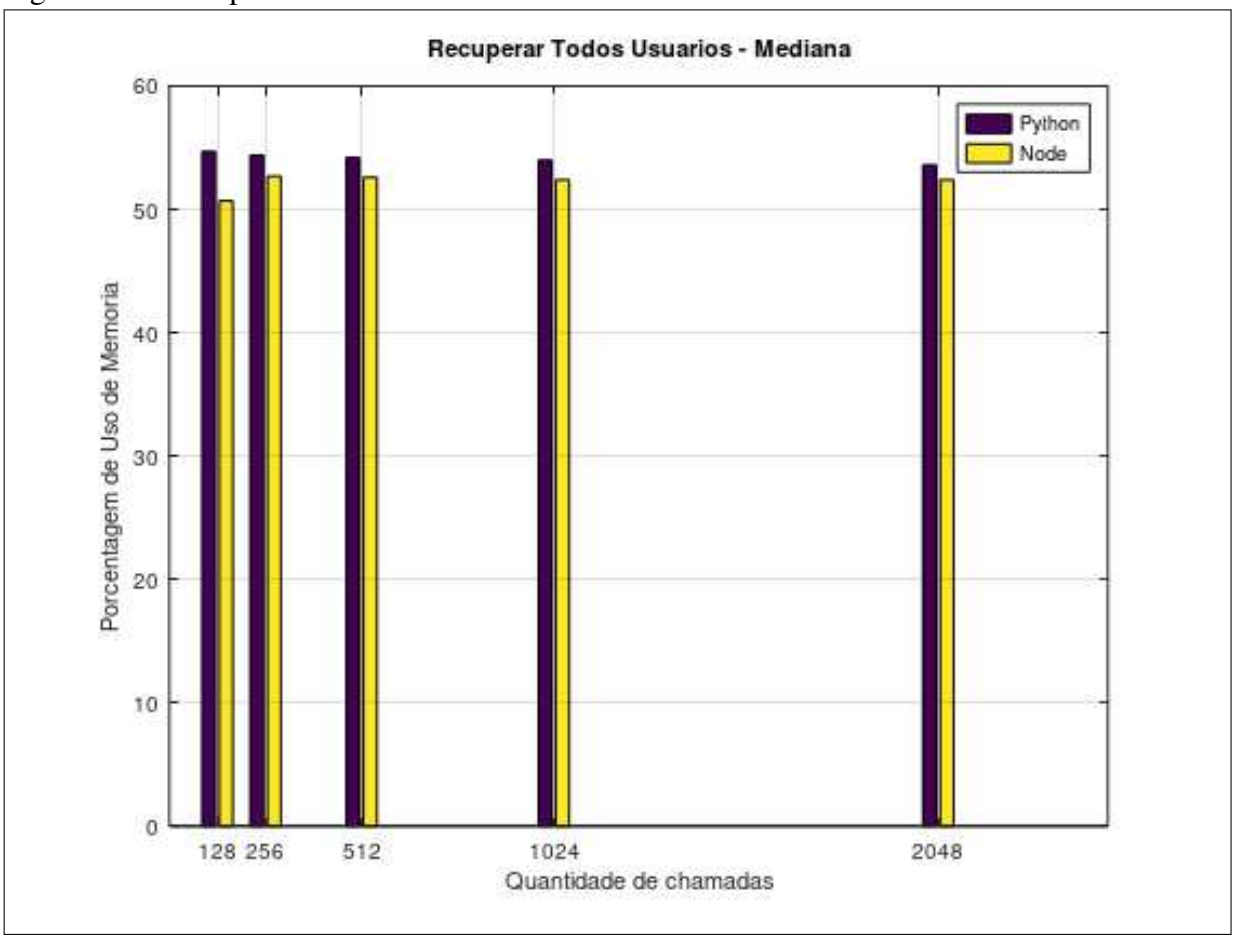


Tabela 32 – Recuperar todos os usuários - (Min+Max)/2

	128	**256**	**512**	**1024**	**2048**
Python	54,85%	54,4%	54,2%	53,95%	53,55%
Node	50,7%	52,7%	52,6%	52,4%	52,35%

6 CONCLUSÕES E TRABALHOS FUTUROS

No decorrer deste estudo, conduzimos uma análise abrangente das implementações de backend em Python e Node.js, explorando diversos aspectos de desempenho, desde o consumo de CPU até a alocação de memória, em diferentes cenários de carga de trabalho. Os resultados obtidos fornecem insights valiosos para desenvolvedores e equipes de TI, orientando a escolha entre essas duas linguagens amplamente utilizadas.

Em relação ao consumo de CPU, observamos que ambas as implementações apresentam um aumento gradual conforme a carga de trabalho cresce. No entanto, a implementação em Node.js demonstrou, de maneira consistente, uma eficiência superior em cenários de carga mais leve, indicando uma performance notável em operações mais rápidas e menos intensivas.

A análise do consumo de memória revelou um padrão interessante. Ambas as linguagens mantiveram níveis estáveis de consumo, mostrando uma eficiência notável mesmo sob cargas variáveis. A implementação em Python exibiu uma leve redução no consumo, indicando uma adaptação eficaz a cargas mais intensas, enquanto a implementação em Node.js, embora com uma variação mais pronunciada, manteve níveis eficientes.

Além dos aspectos técnicos, é crucial considerar outros fatores ao escolher a linguagem de backend, como a familiaridade da equipe e a facilidade de manutenção do código. Embora os resultados indiquem diferenças de desempenho, a escolha ideal dependerá das necessidades específicas do projeto e das preferências da equipe de desenvolvimento.

Em suma, este estudo oferece uma visão abrangente do desempenho de implementações de backend em Python e Node.js, fornecendo informações valiosas para orientar decisões de arquitetura de software. À medida que as tecnologias continuam a evoluir, é fundamental que desenvolvedores estejam cientes das nuances de desempenho para tomar decisões informadas e garantir soluções eficazes e eficientes.

REFERÊNCIAS

BERNERS-LEE, T.; CAILLIAU, R.; GROFF, J.-F. The world-wid web. Geneva, I, p. 454–459, 1992.

BROWN, E. **Web Development with Node and Express: Leveraging the JavaScript Stack**. [*S. l.*]: O'Reilly Media, 2014. ISBN 978-1491949306.

CONTRIBUTORS, E. **Express.js Documentation**. ongoing. Disponível em: https://expressjs.com/.

CROCKFORD, D. **JavaScript: The Good Parts**. [*S. l.*]: O'Reilly Media, 2008. ISBN 978-0596517748.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. Tese (Doutorado) – University of California, Irvine, 2000.

FLANAGAN, D. **JavaScript: The Definitive Guide**. [*S. l.*]: O'Reilly Media, 2011. ISBN 978-0596805524.

INTERNATIONAL, E. **ECMAScript Language Specification**. ongoing. Disponível em: https://www.ecma-international.org/ecma-262/.

JúNIOR, M. G. C. **Estudo de mecanismos e fatores que impactam no desempenho de aplicações Web**. Trabalho de Conclusão de Curso – Universidade Federal de Ouro Preto, João Monlevade, MG, setembro 2017.

MEAD, M. C. **Node.js Design Patterns**. [*S. l.*]: Packt Publishing, 2019. ISBN 978-1789951216.

MENDES, A. **Arquitetura de Software: desenvolvimento orientado para arquitetura**. [*S. l.*]: Editora Campus. Rio de Janeiro - RJ, 2002. ISBN 9788535210132.

MOURA, J. F. d. M. R. **Avaliação do impacto de balanceadores de carga sobre o gRPC**. 35 p. Trabalho de Conclusão de Curso – Universidade Federal de Pernambuco, Recife, 2022. Orientador: Nelson Souto Rosa.

OSMANI, A. Building robust apis with express.js. **Medium**, 2018. Disponível em: https://medium.com/@addyosmani/building-robust-web-apis-with-express-js-24be0eecc386.

POSTMAN, B. **Introduction to APIs: A Brief History of APIs**. 2023. Accessed on November 30, 2023. Disponível em: https://blog.postman.com/intro-to-apis-history-of-apis/.

REDDY, M. **API Design for C++**. [*S. l.*]: Elsevier Science, 2011. ISBN 9780123850041.

RICHARDSON, L.; RUBY, S. **RESTful Web Services**. [*S. l.*]: O'Reilly Media, 2007.

SOMMERVILLE, I. **Engenharia de software**. [*S. l.*]: Pearson Prentice Hall, 2011. ISBN 978-8-579-36108-1.

SYKORA, M. D. Web 1.0 to web 2.0: an observational study and empirical evidence for the historical r(evolution) of the social web. **Int. J. Web Eng. Technol.**, v. 12, p. 70–94, 2017.

APÊNDICE A – CÓDIGOS-FONTES UTILIZADOS PARA

Código-fonte 1 – Código para criação do banco de dados

```
CREATE TABLE accounts (
  user_id serial PRIMARY KEY,
  username VARCHAR ( 50 ) UNIQUE NOT NULL,
  password VARCHAR ( 50 ) NOT NULL,
  email VARCHAR ( 255 ) UNIQUE NOT NULL
);
```

ANEXO A – RELATÓRIO DE SUMÁRIO JMETER

Figura 20 – Relatório de sumário - Node.js - 128 requisições - Criar usuário

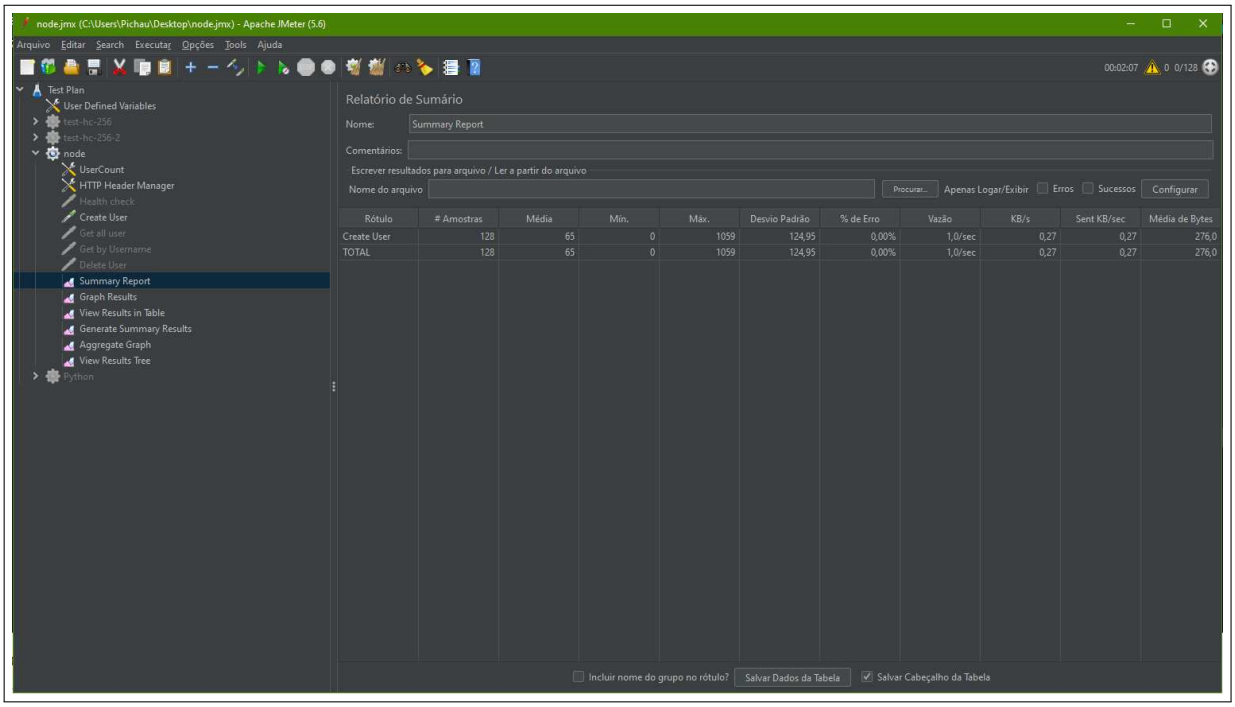


Figura 21 – Relatório de sumário - Node.js - 256 requisições - Criar usuário

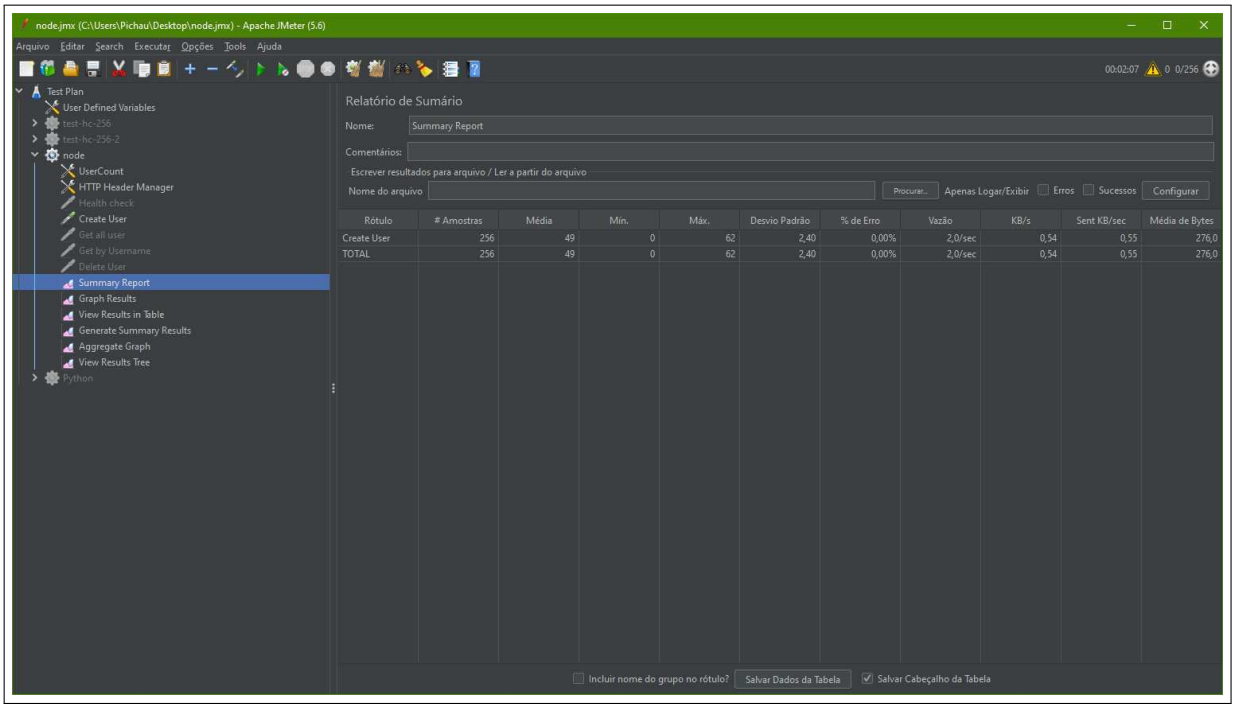


Figura 22 – Relatório de sumário - Node.js - 512 requisições - Criar usuário

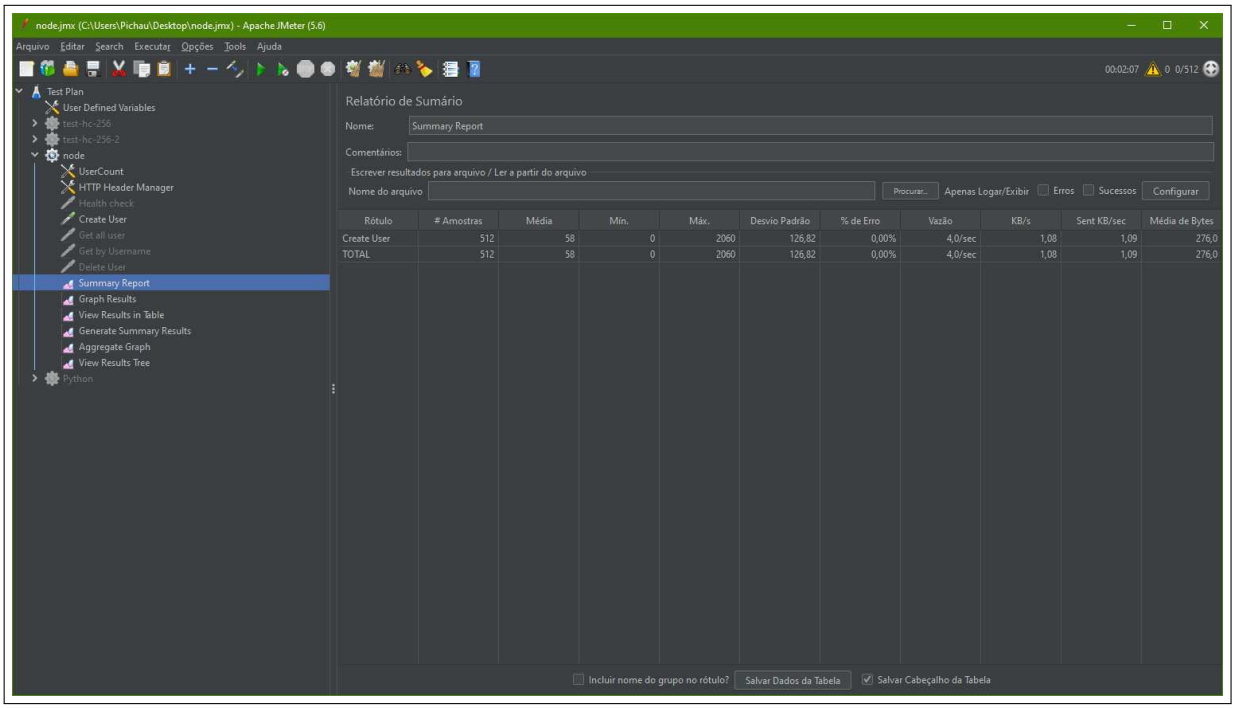


Figura 23 – Relatório de sumário - Node.js - 1024 requisições - Criar usuário

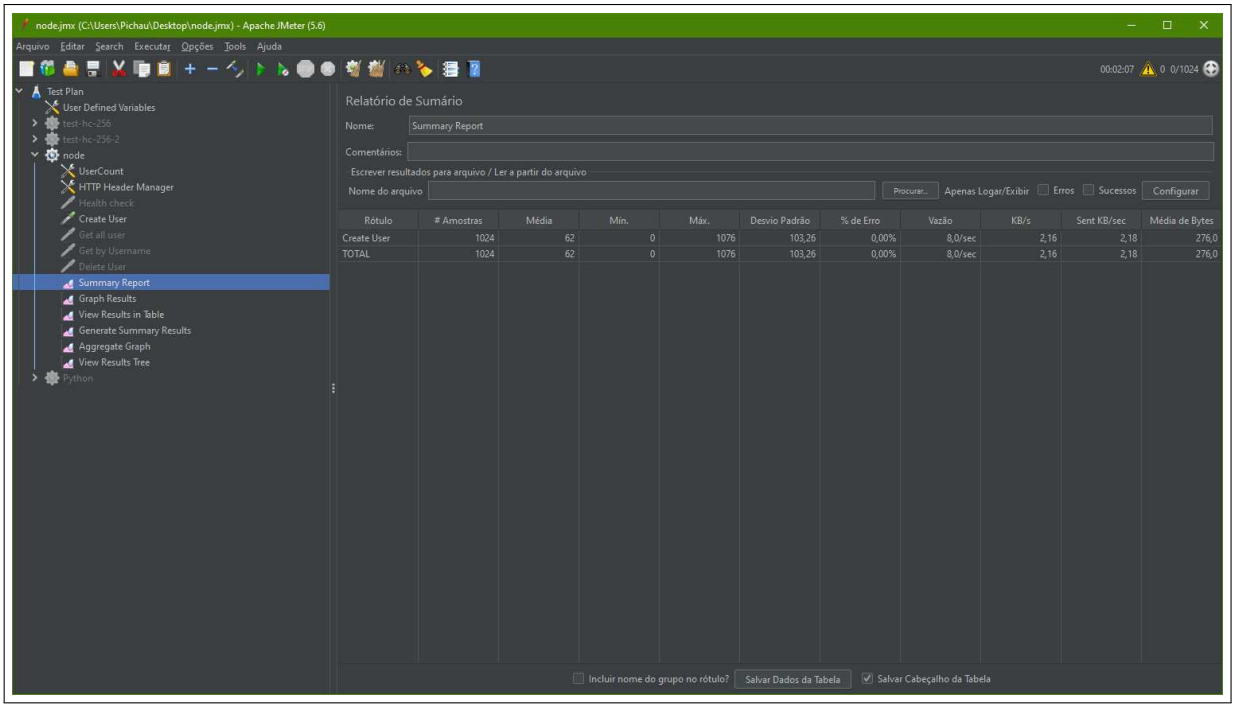


Figura 24 – Relatório de sumário - Node.js - 2048 requisições - Criar usuário

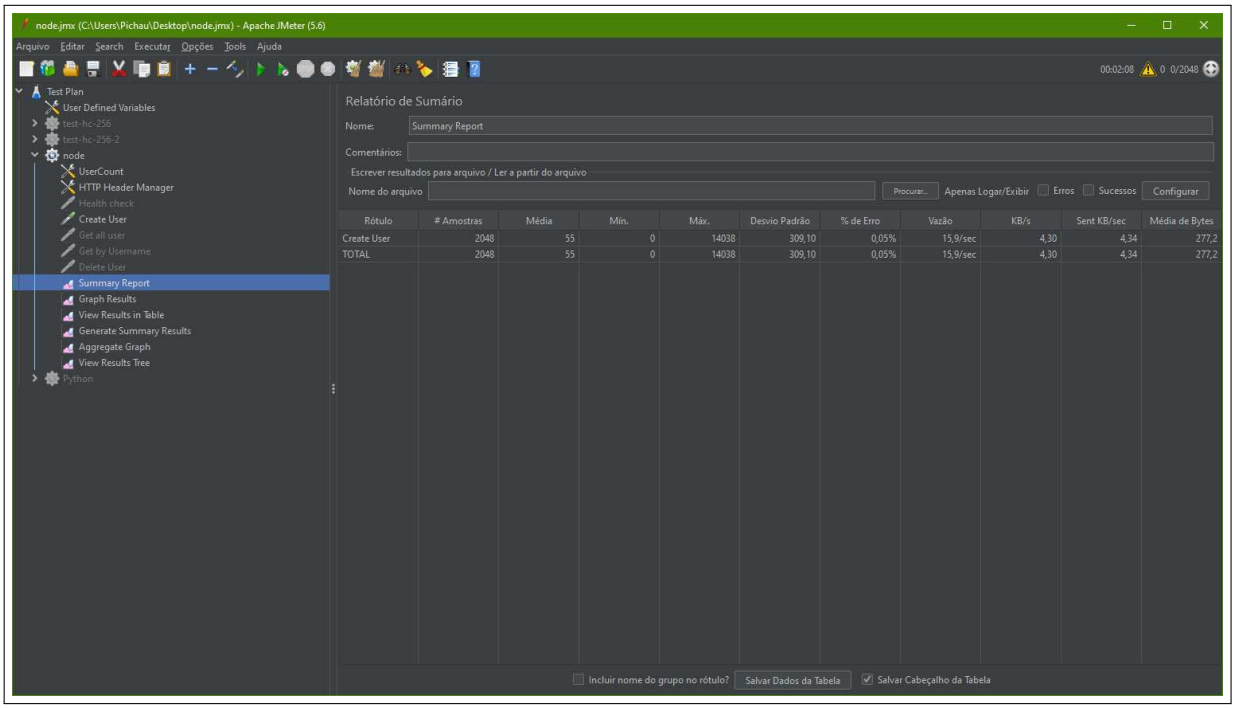


Figura 25 – Relatório de sumário - Node.js - 128 requisições - Remover usuário

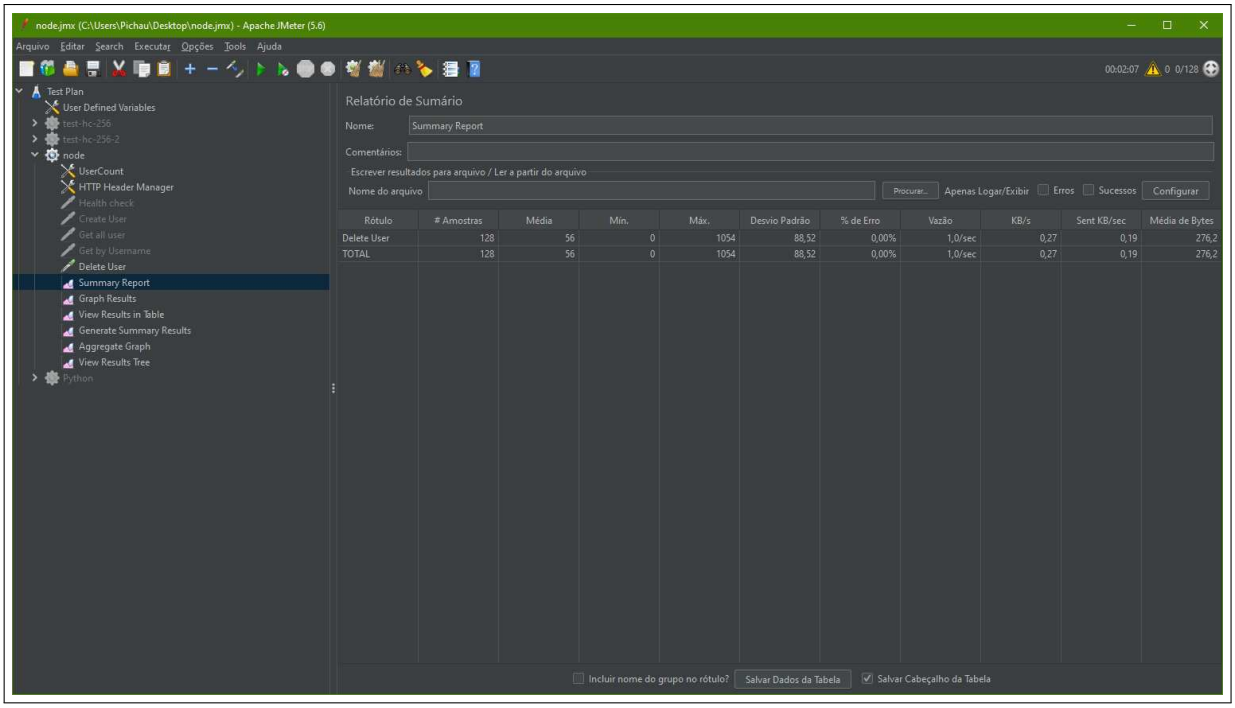


Figura 26 – Relatório de sumário - Node.js - 256 requisições - Remover usuário

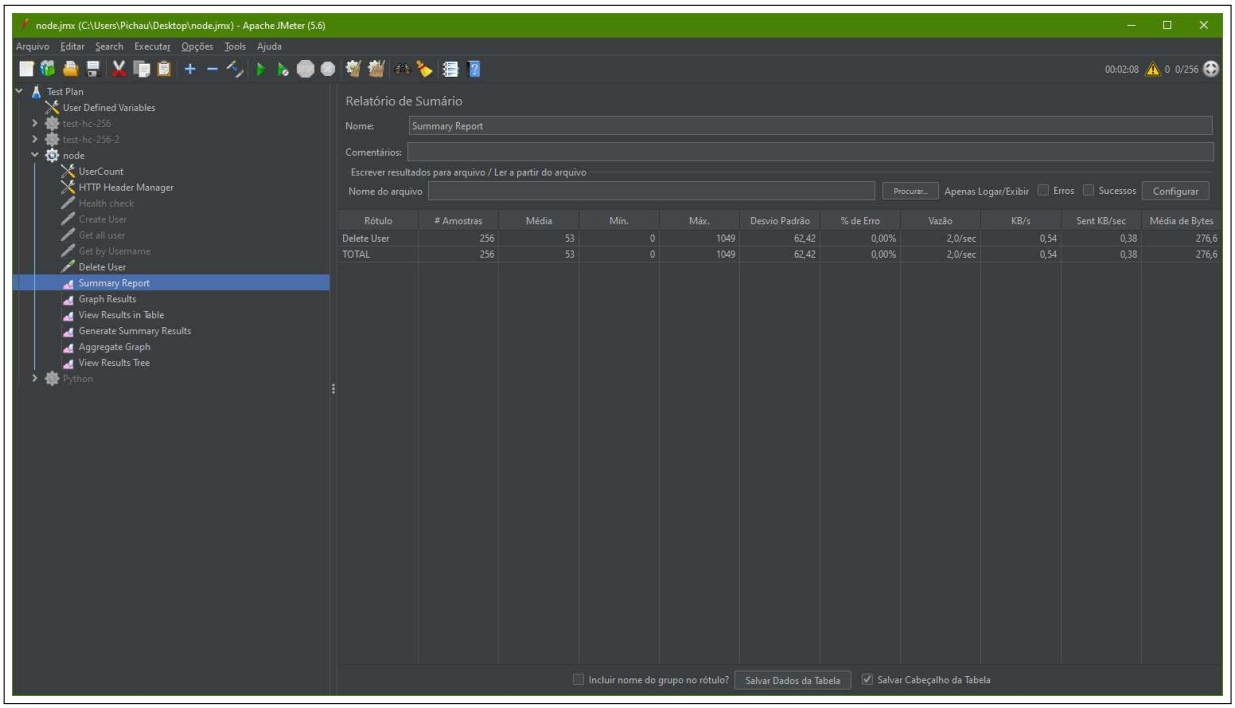


Figura 27 – Relatório de sumário - Node.js - 512 requisições - remover usuário

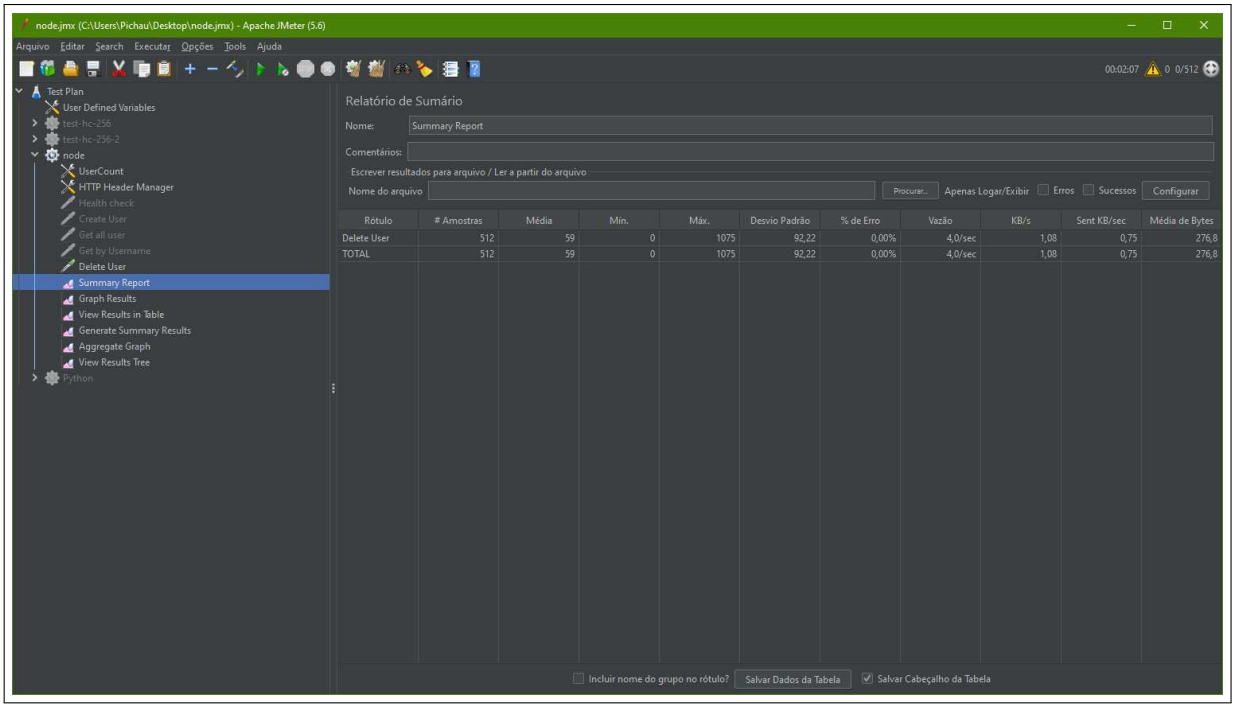


Figura 28 – Relatório de sumário - Node.js - 1024 requisições - Remover usuário

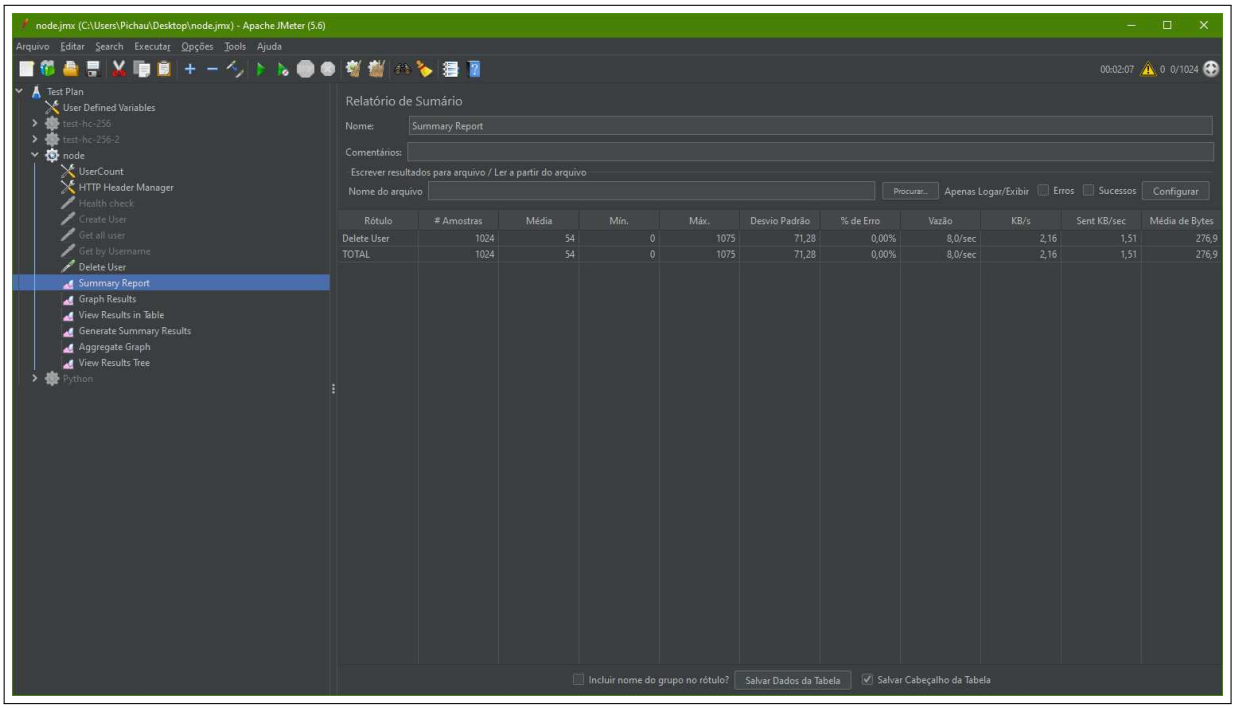


Figura 29 – Relatório de sumário - Node.js - 2048 requisições - Remover usuário

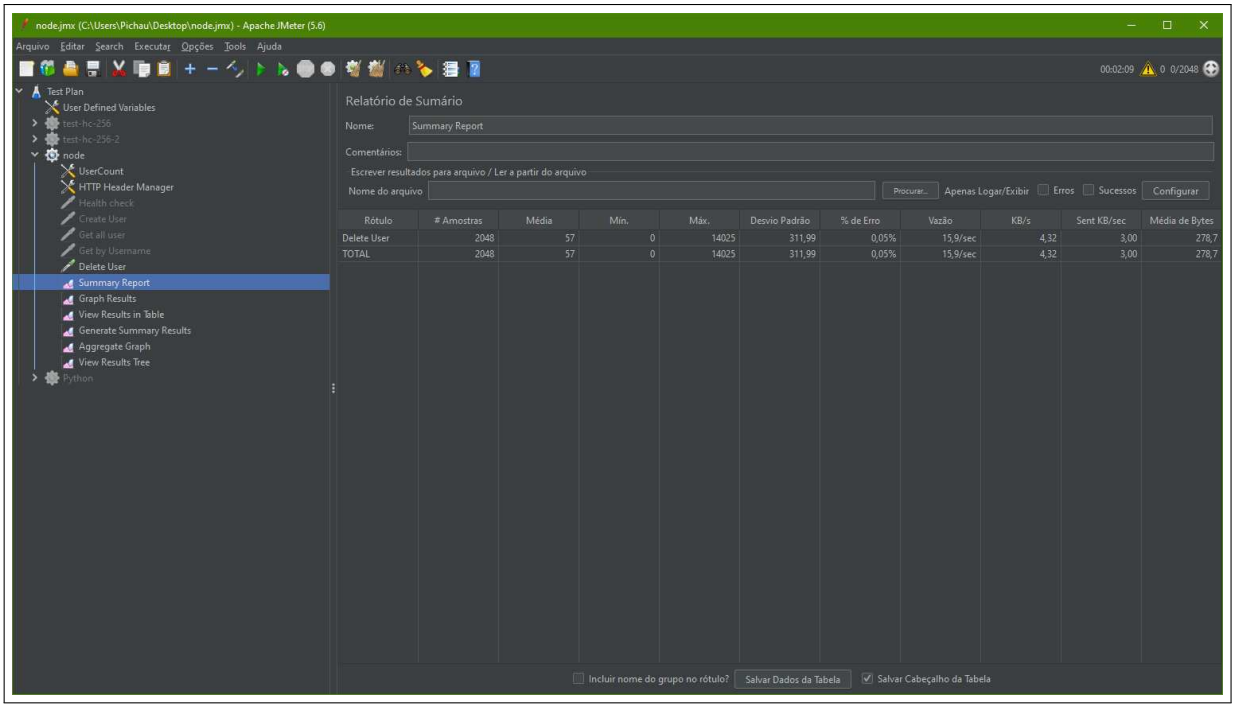


Figura 30 – Relatório de sumário - Node.js - 128 requisições - Recuperar um usuário

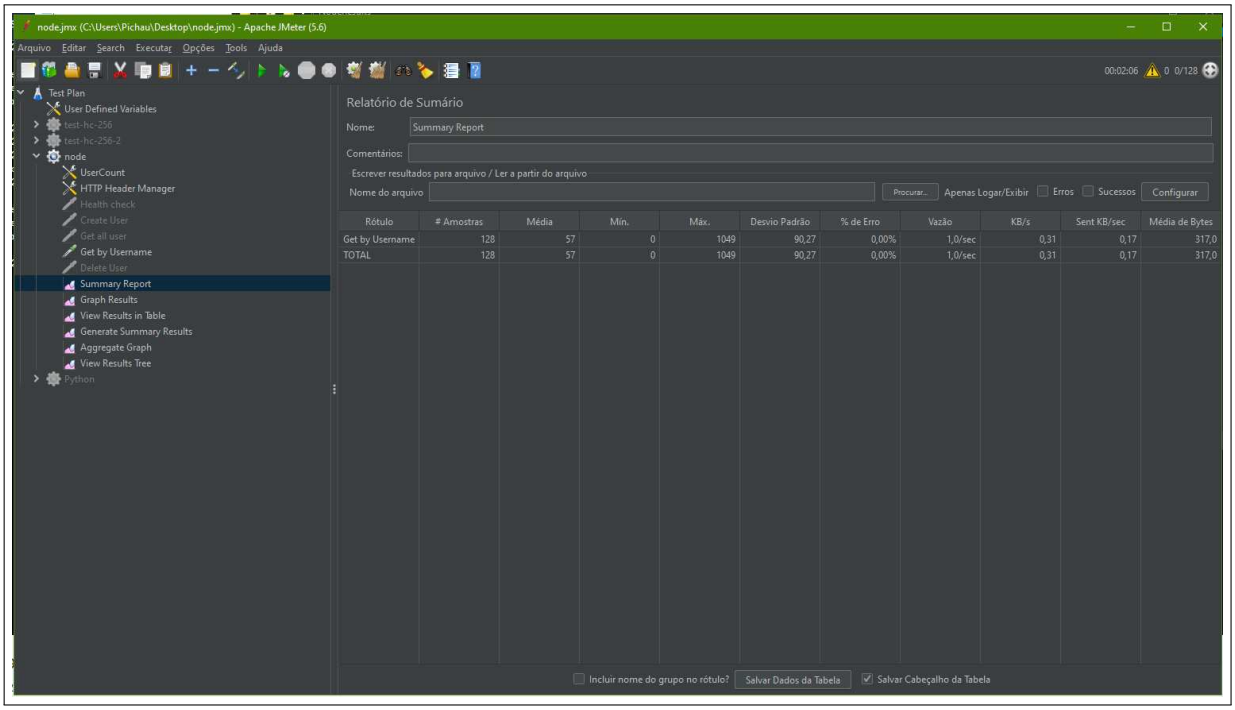


Figura 31 – Relatório de sumário - Node.js - 256 requisições - Recuperar um usuário

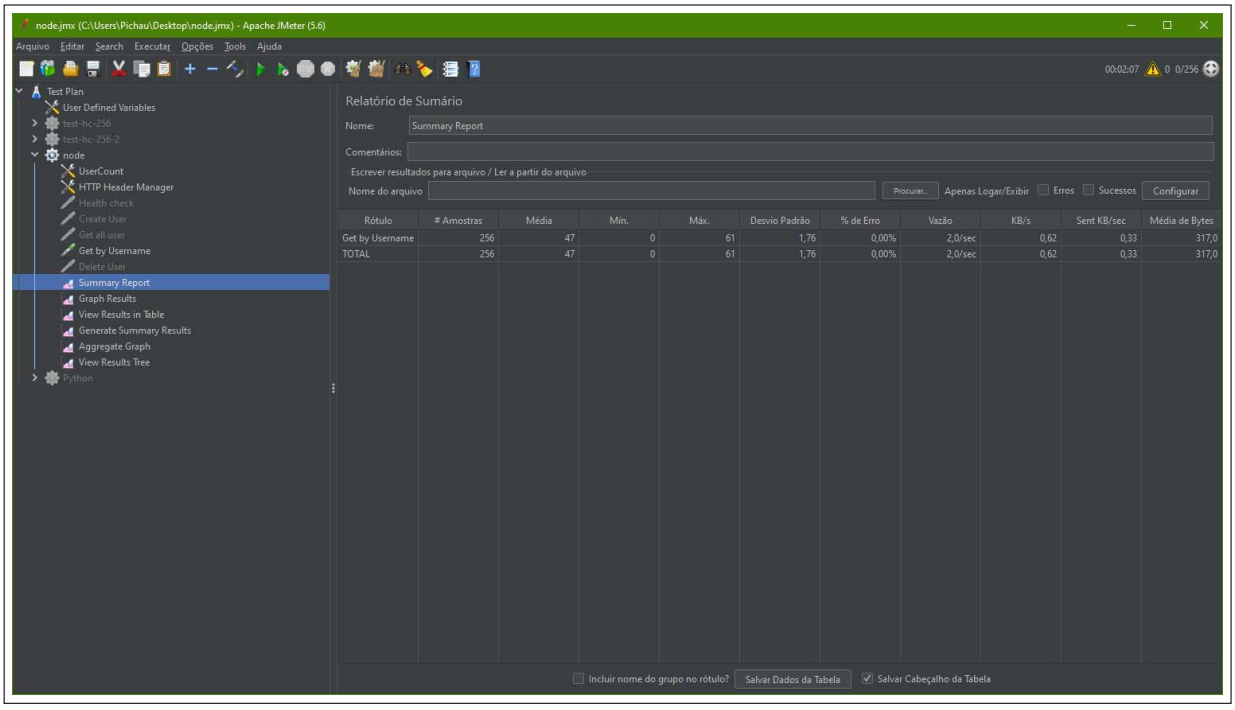


Figura 32 – Relatório de sumário - Node.js - 512 requisições - Recuperar um usuário

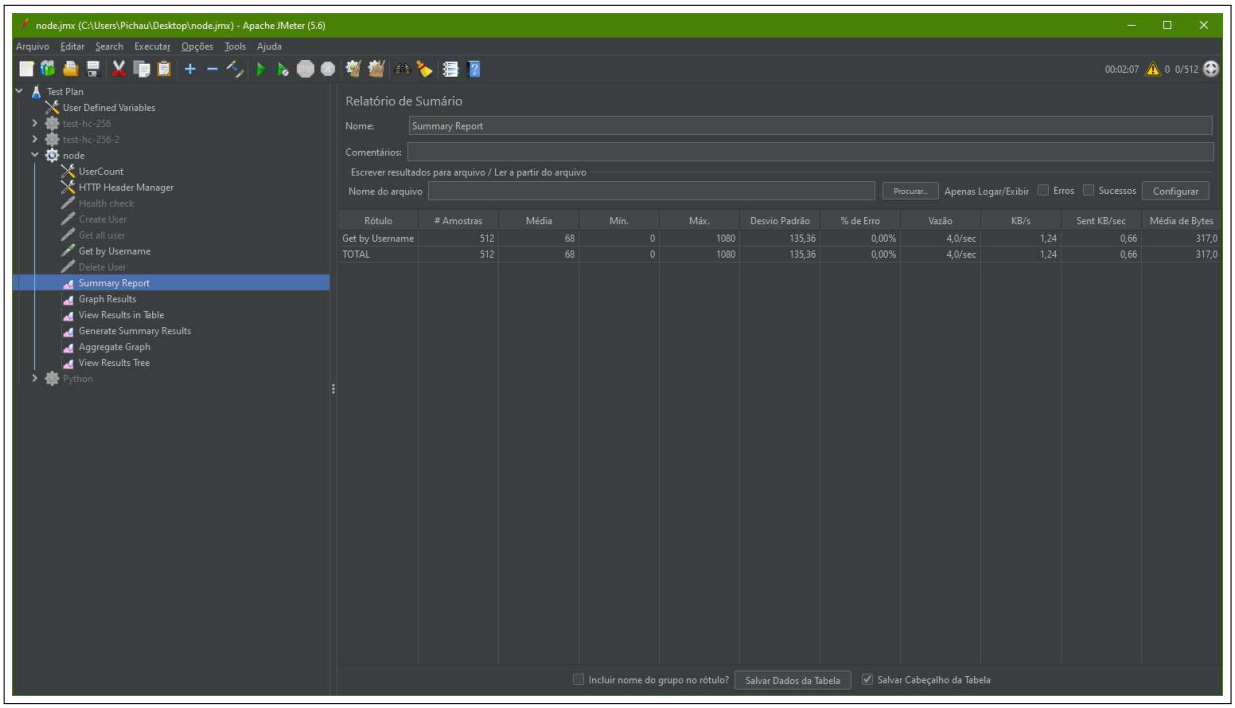


Figura 33 – Relatório de sumário - Node.js - 1024 requisições - Recuperar um usuário

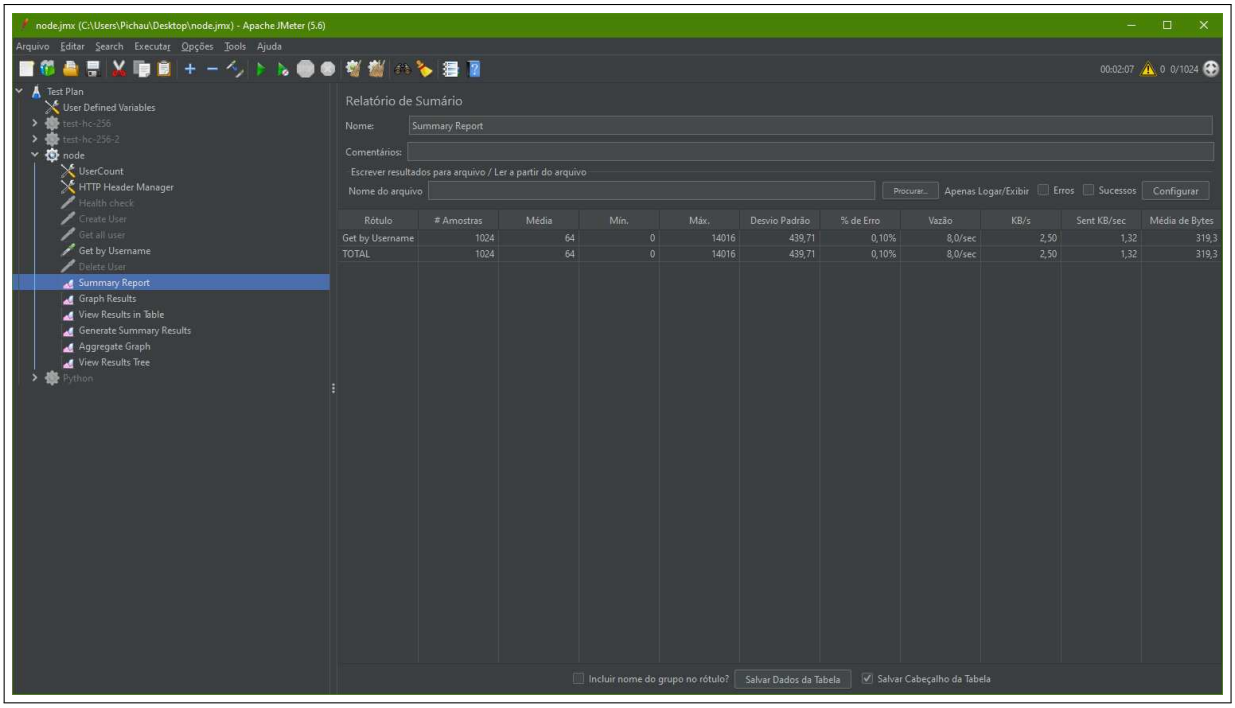


Figura 34 – Relatório de sumário - Node.js - 2048 requisições - Recuperar um usuário

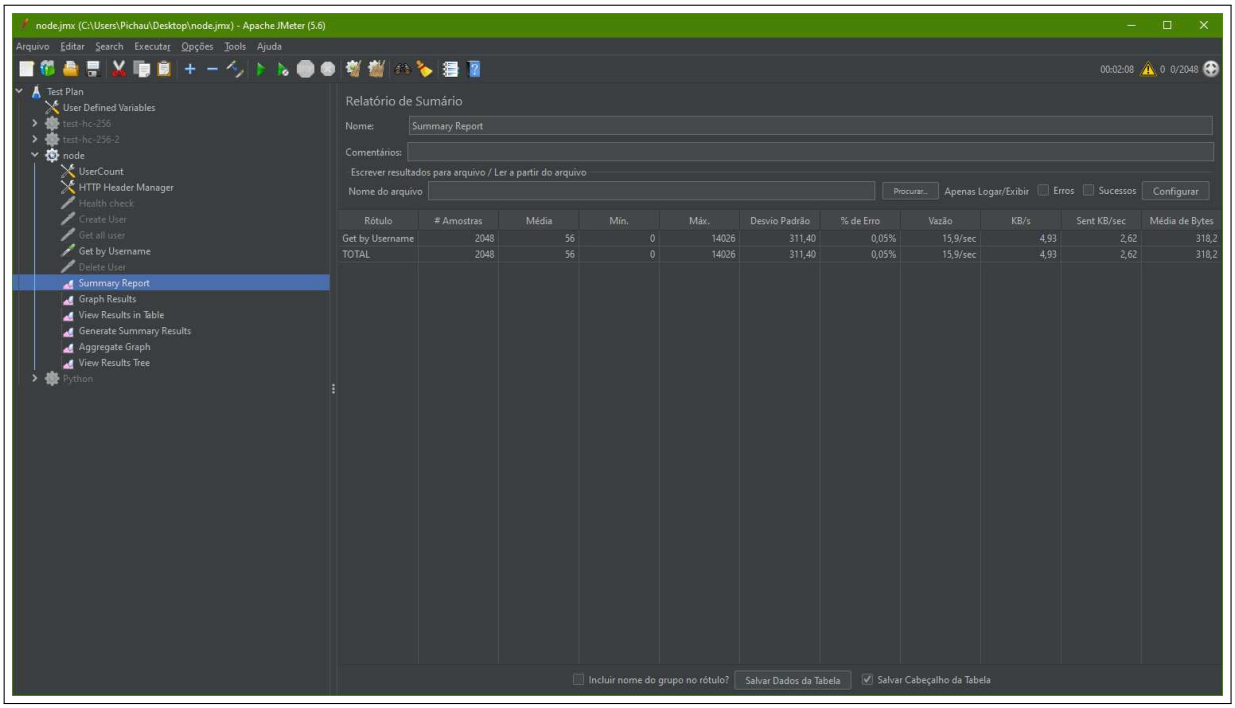


Figura 35 – Relatório de sumário - Node.js - 128 requisições - Recuperar todo os usuários

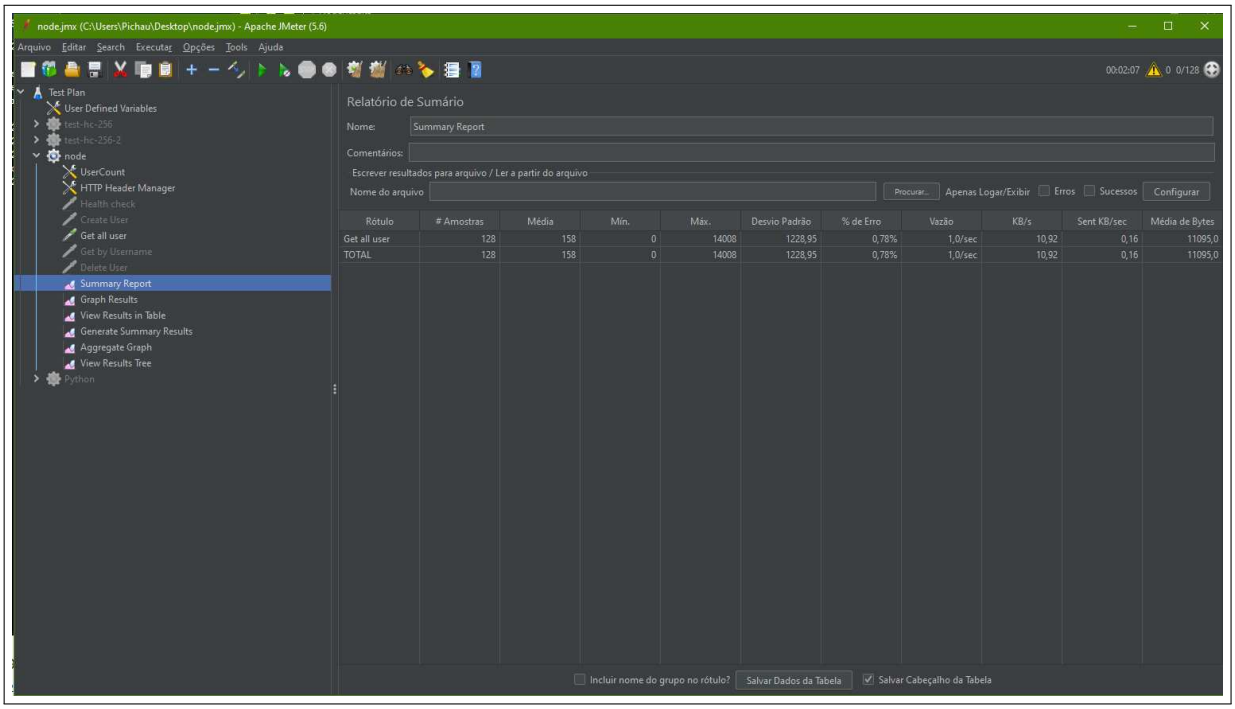


Figura 36 – Relatório de sumário - Node.js - 256 requisições - Recuperar todo os usuários

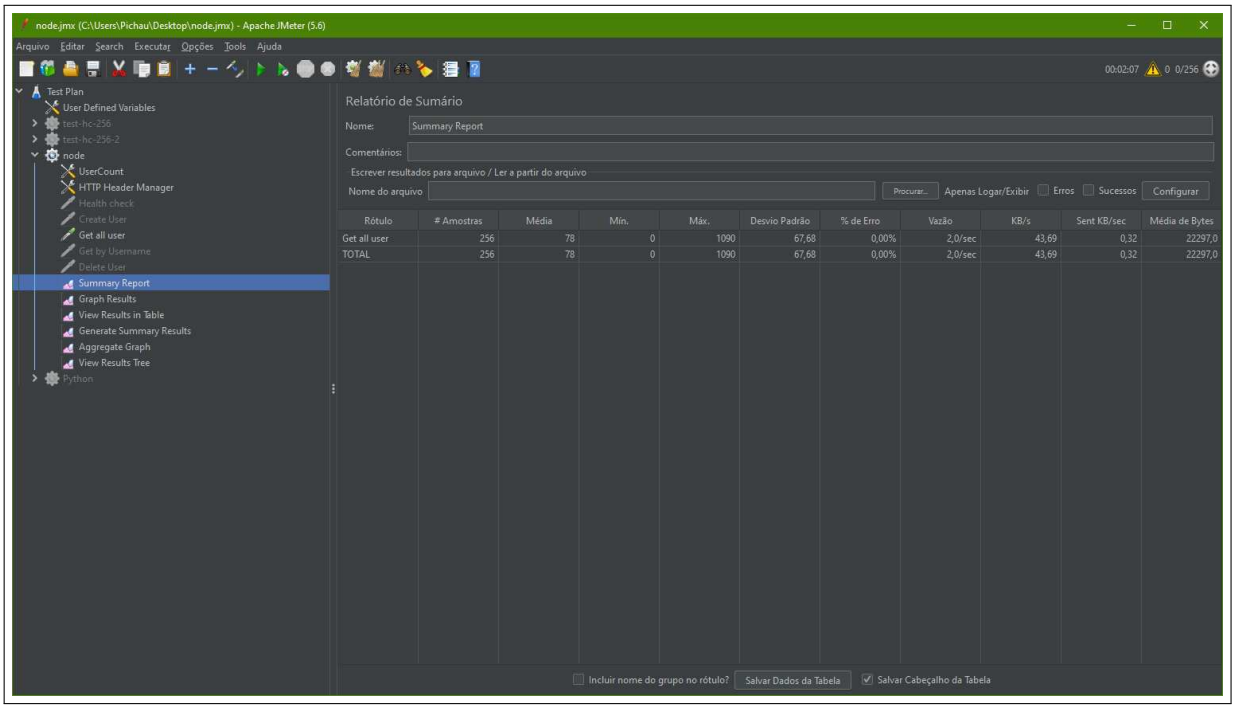


Figura 37 – Relatório de sumário - Node.js - 512 requisições - Recuperar todo os usuários

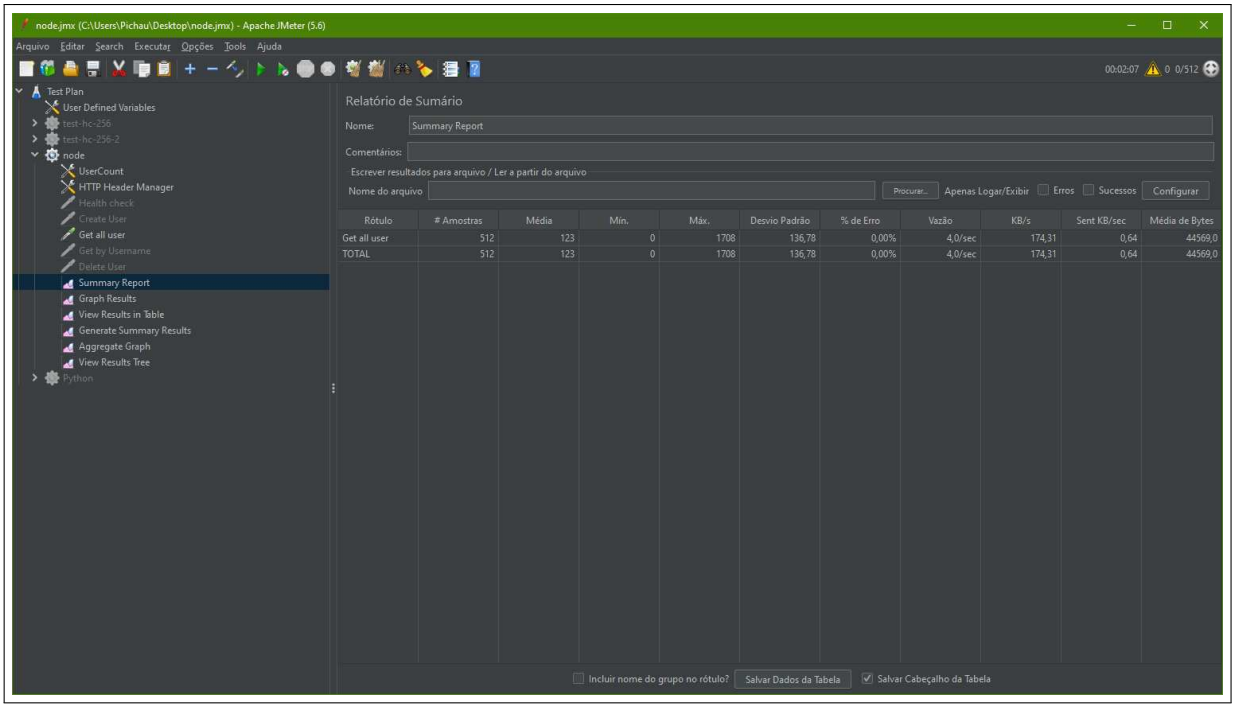


Figura 38 – Relatório de sumário - Node.js - 1024 requisições - Recuperar todo os usuários

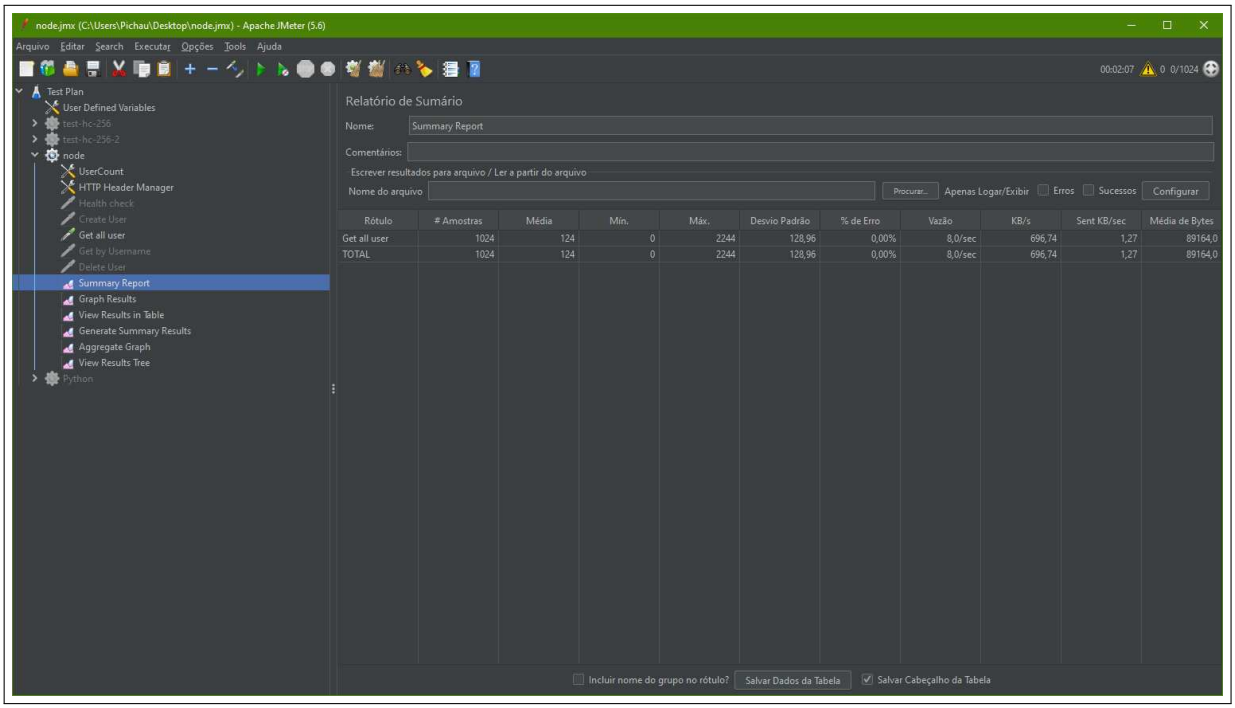


Figura 39 – Relatório de sumário - Node.js - 2048 requisições - Recuperar todo os usuários

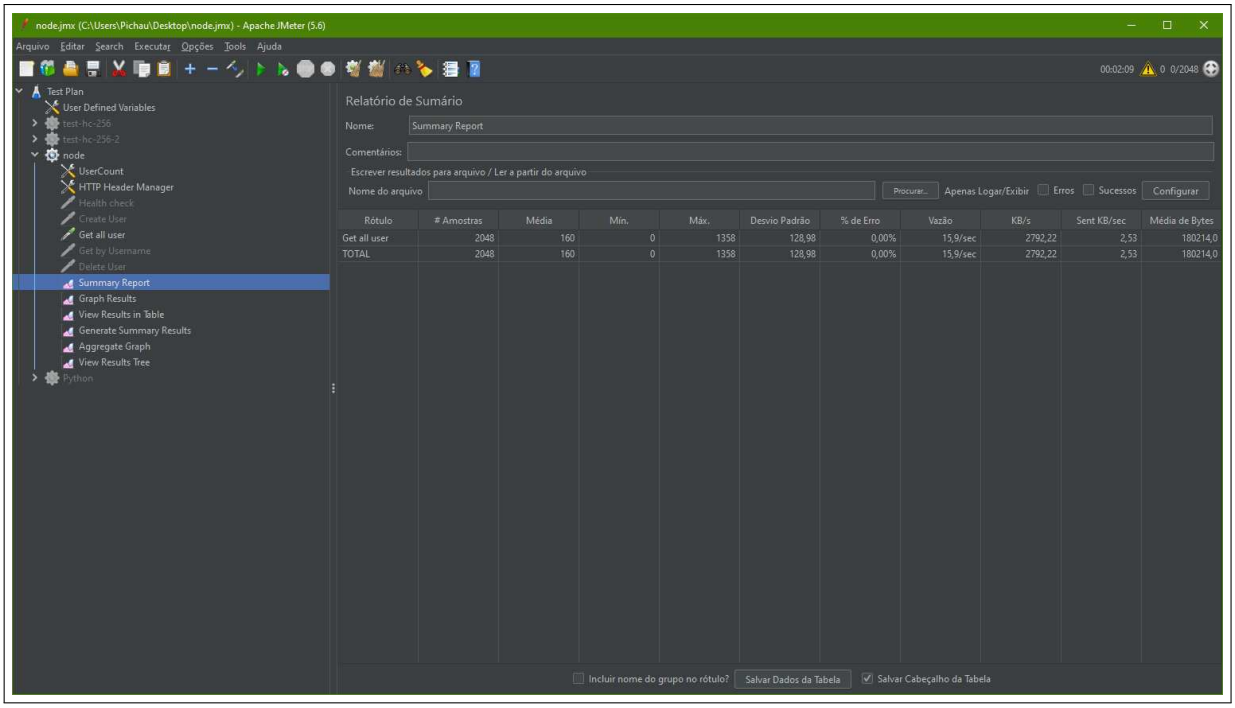


Figura 40 – Relatório de sumário - Python - 128 requisições - Criar usuário

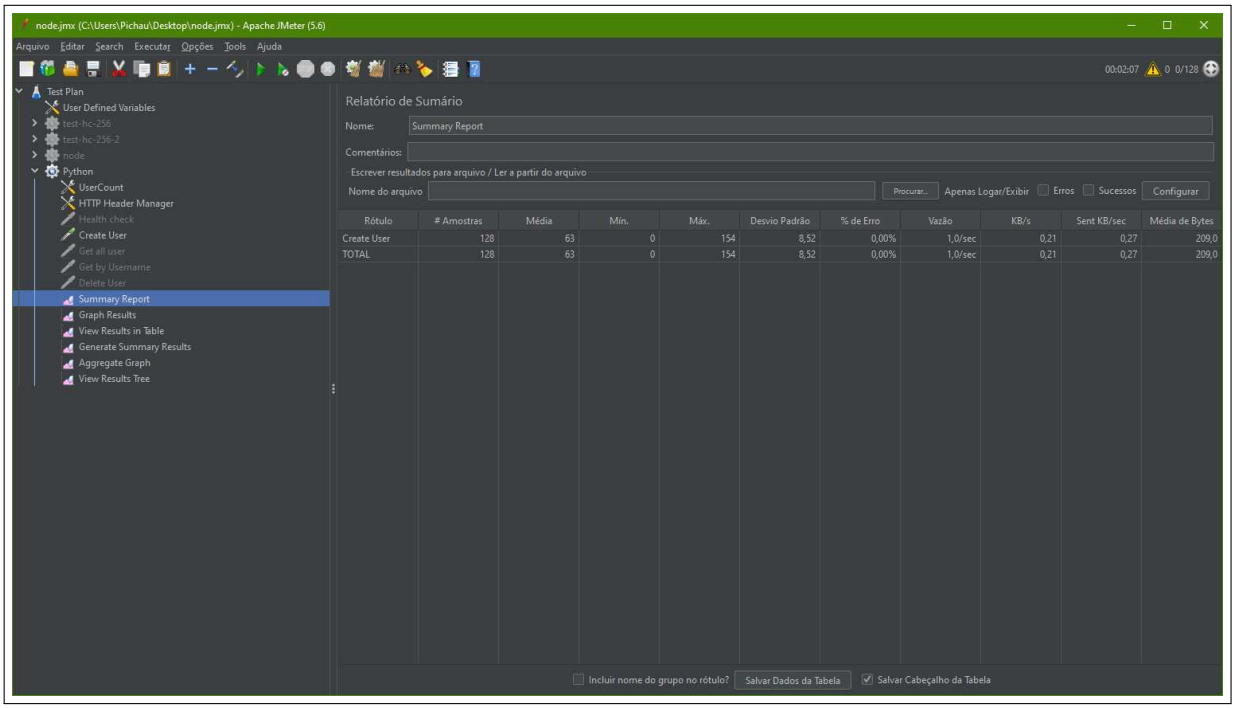


Figura 41 – Relatório de sumário - Python - 256 requisições - Criar usuário

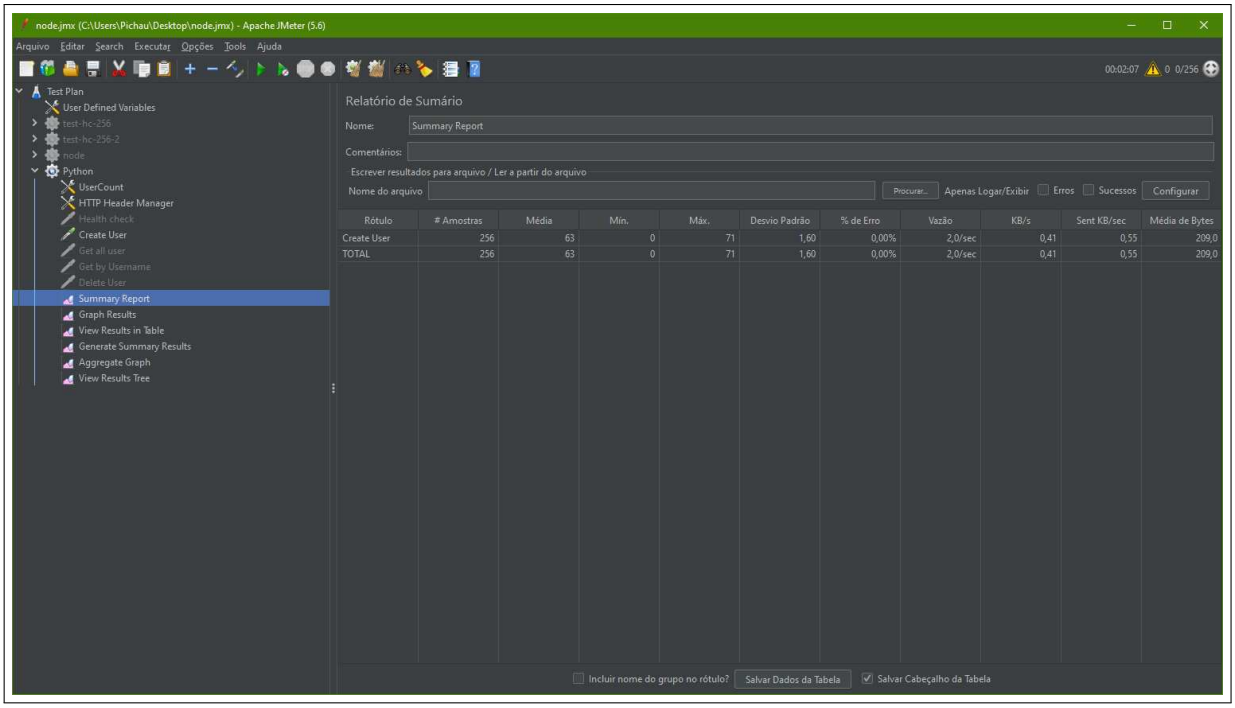


Figura 42 – Relatório de sumário - Python - 512 requisições - Criar usuário

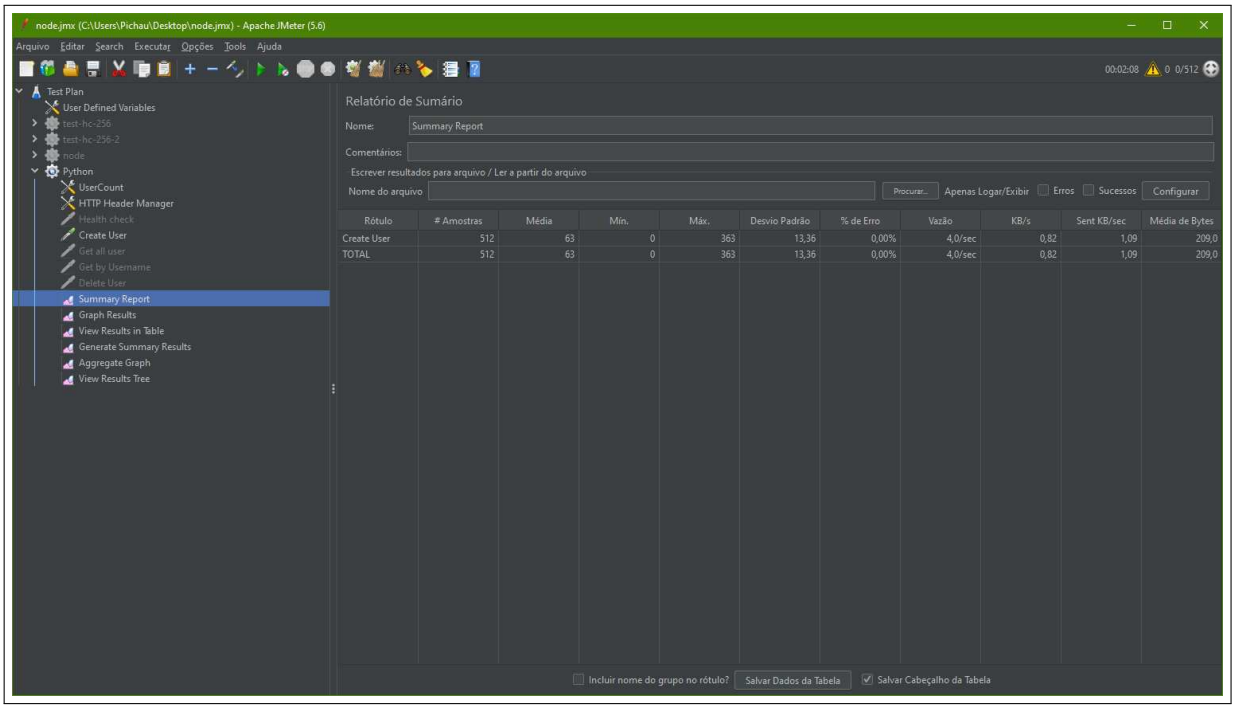


Figura 43 – Relatório de sumário - Python - 1024 requisições - Criar usuário

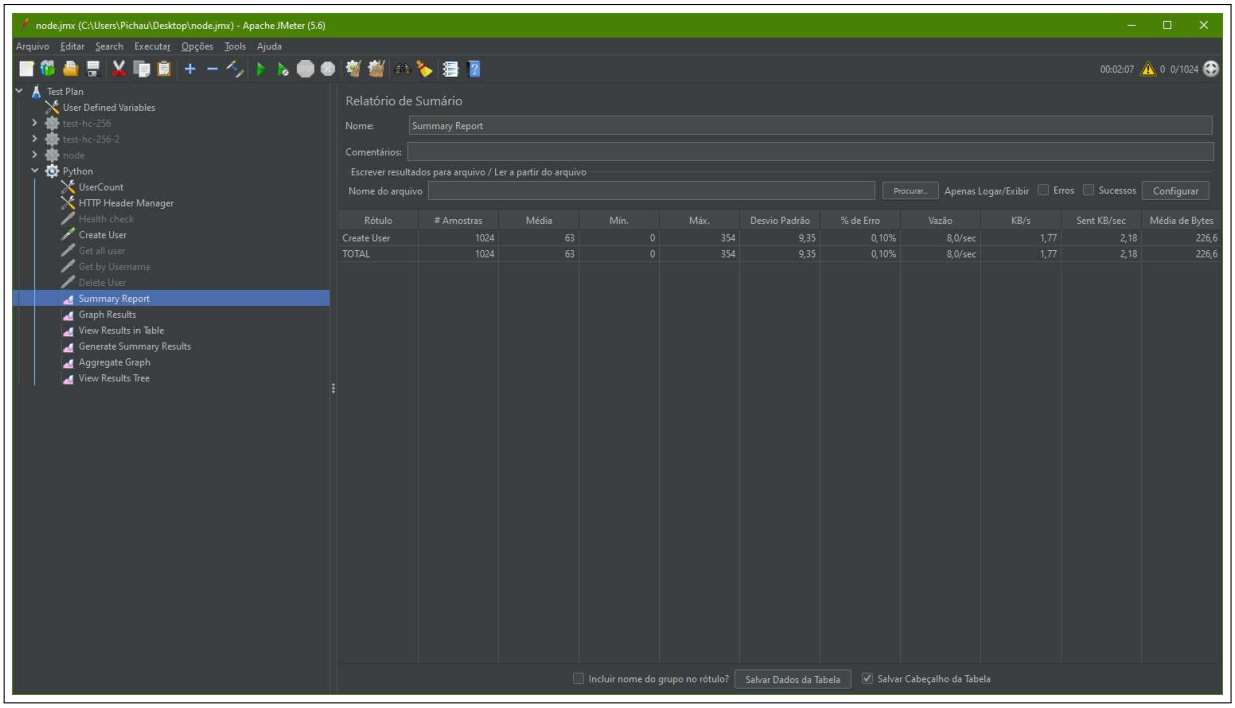


Figura 44 – Relatório de sumário - Python - 2048 requisições - Criar usuário

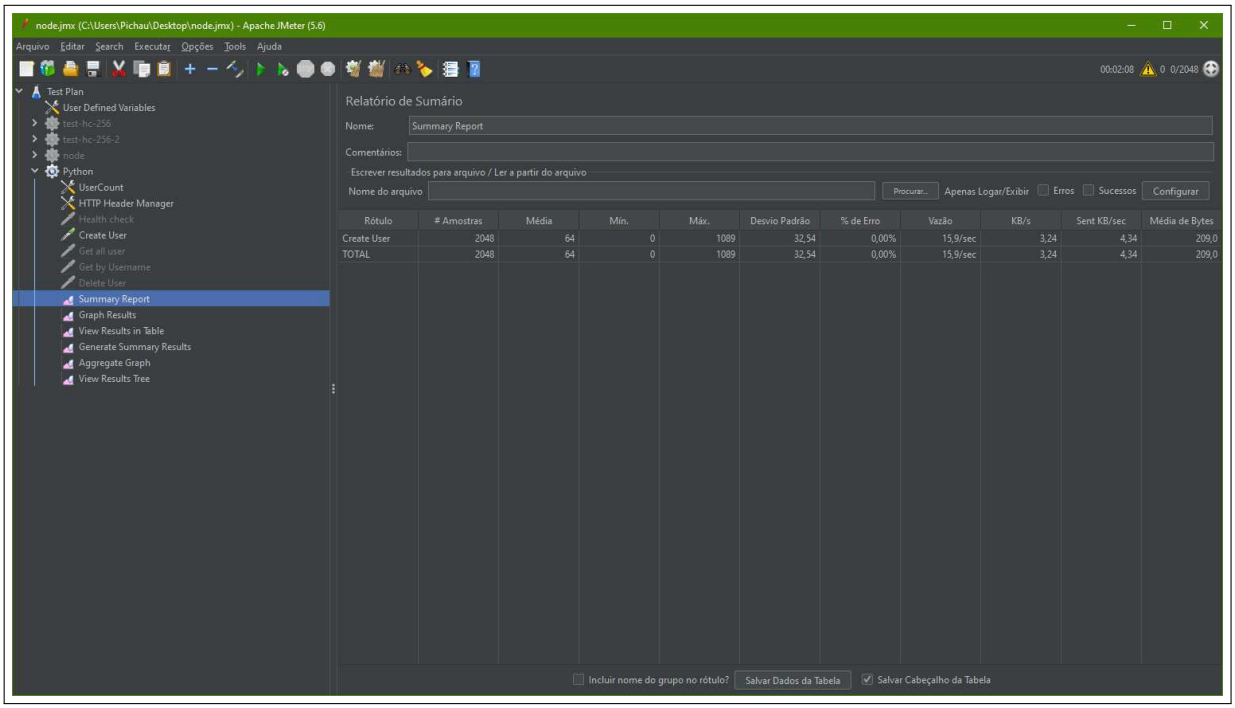


Figura 45 – Relatório de sumário - Python - 128 requisições - Remover usuário

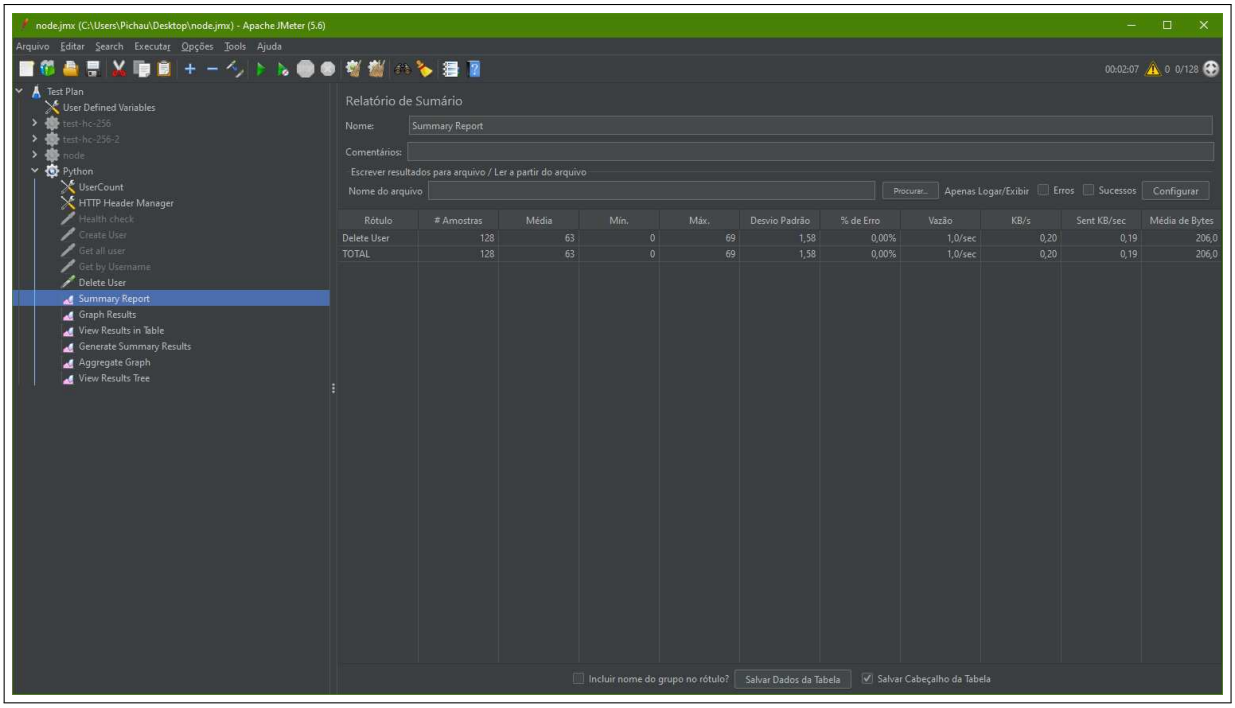


Figura 46 – Relatório de sumário - Python - 256 requisições - Remover usuário

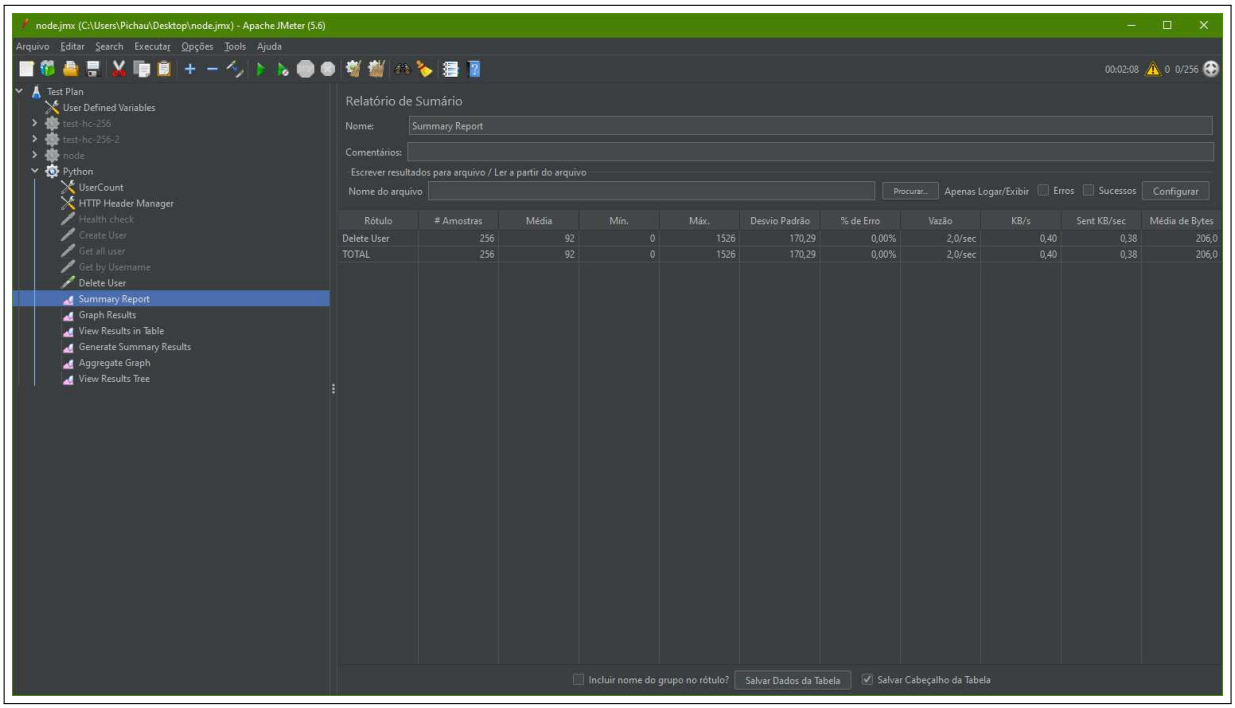


Figura 47 – Relatório de sumário - Python - 512 requisições - remover usuário

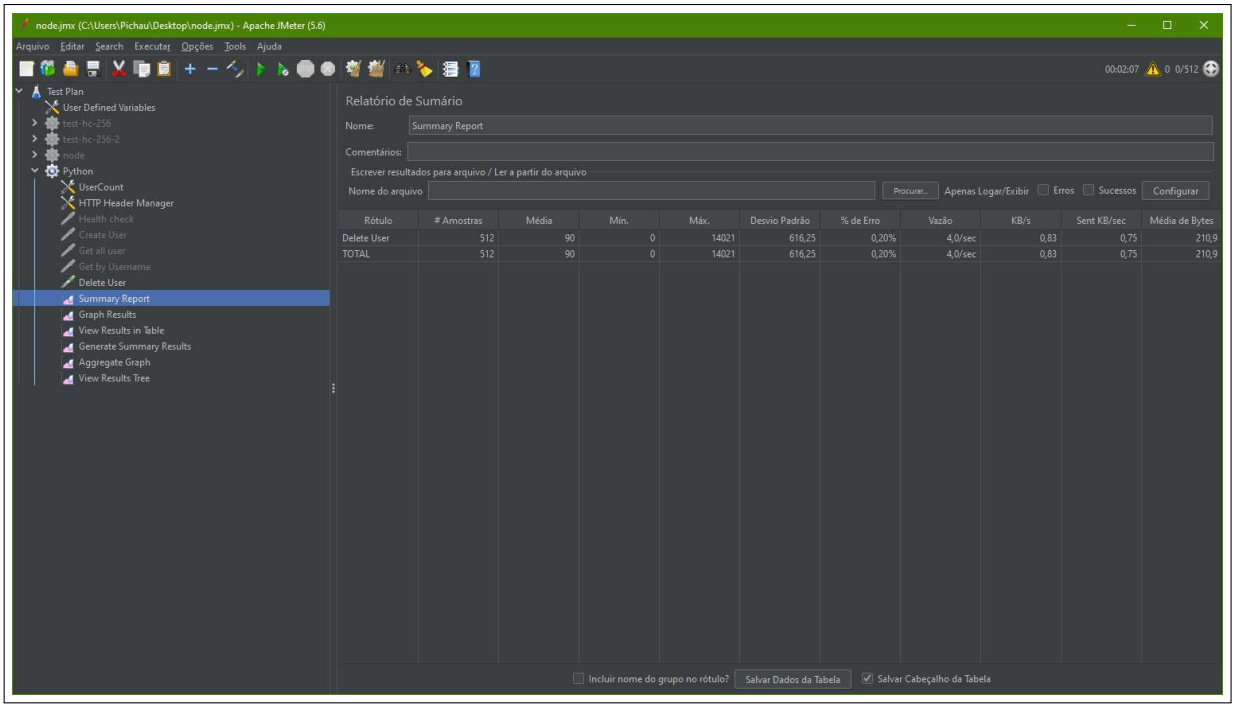


Figura 48 – Relatório de sumário - Python - 1024 requisições - Remover usuário

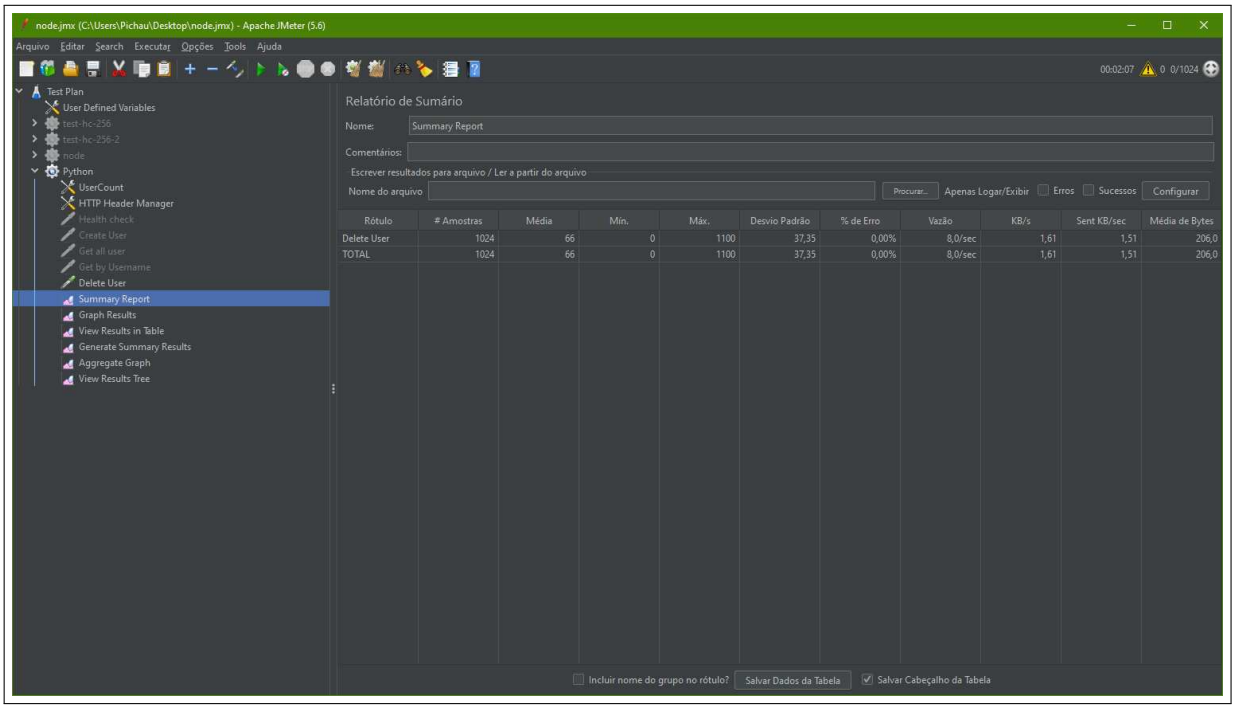


Figura 49 – Relatório de sumário - Python - 2048 requisições - Remover usuário

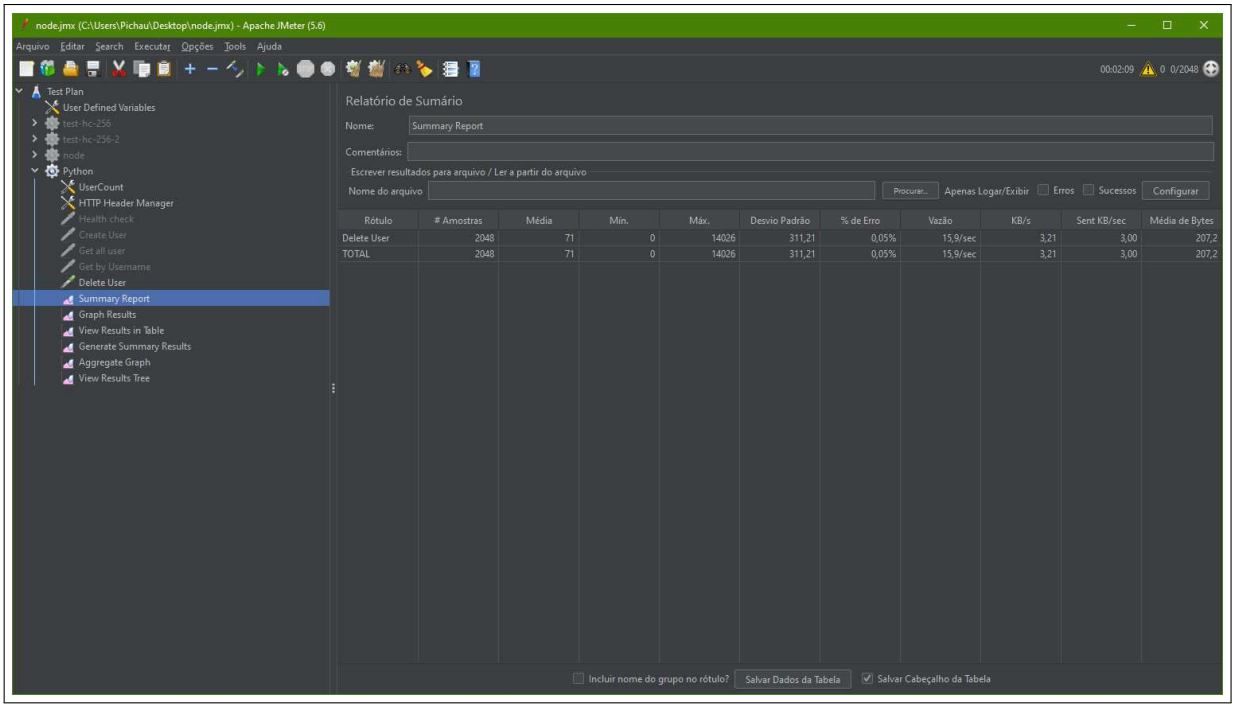


Figura 50 – Relatório de sumário - Python - 128 requisições - Recuperar um usuário

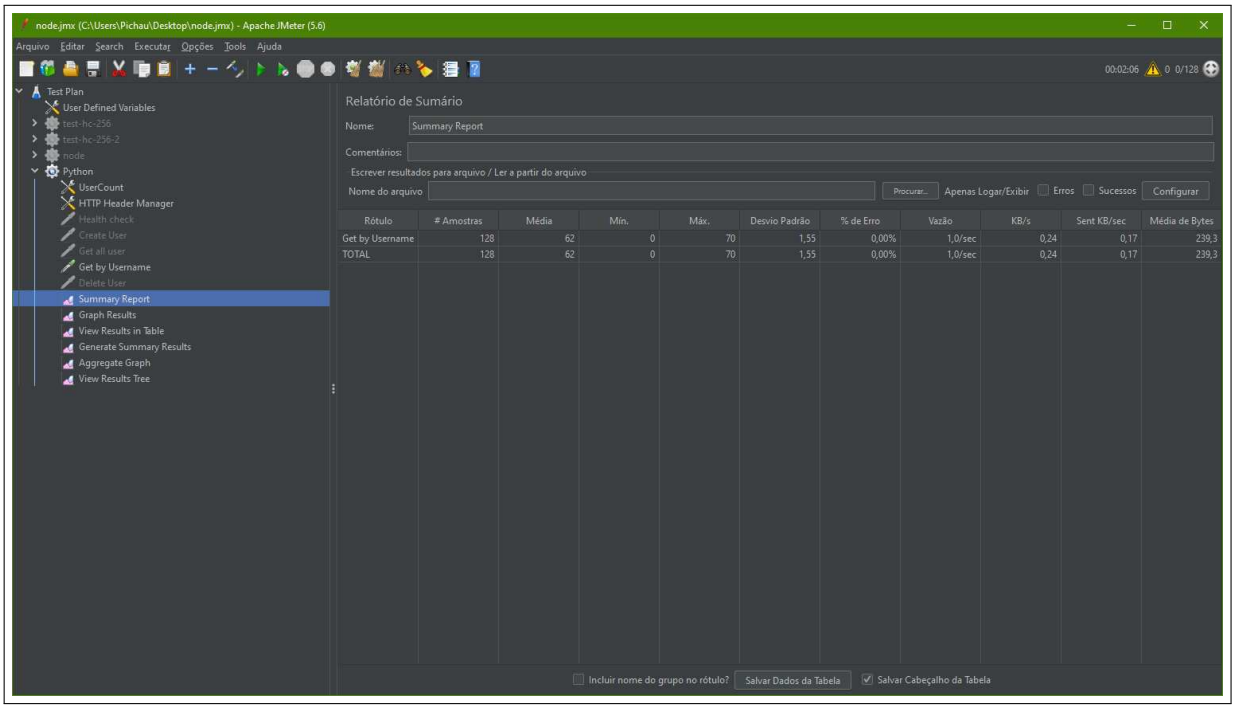


Figura 51 – Relatório de sumário - Python - 256 requisições - Recuperar um usuário

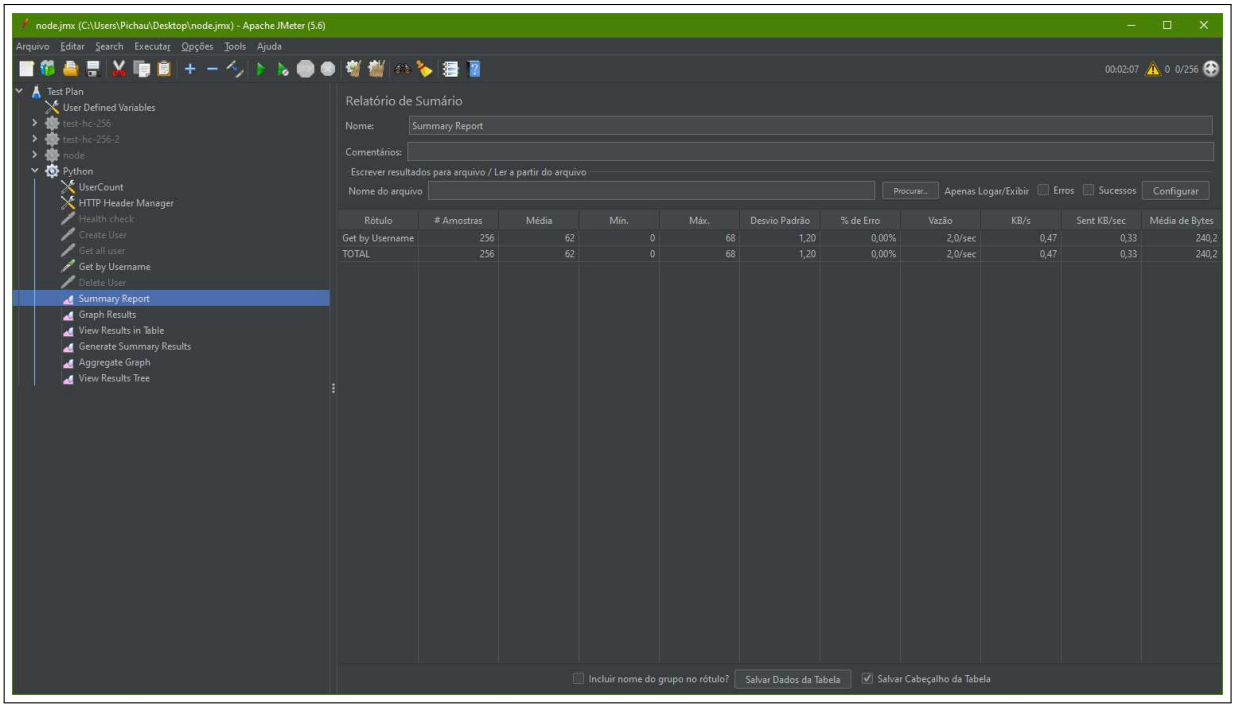


Figura 52 – Relatório de sumário - Python - 512 requisições - Recuperar um usuário

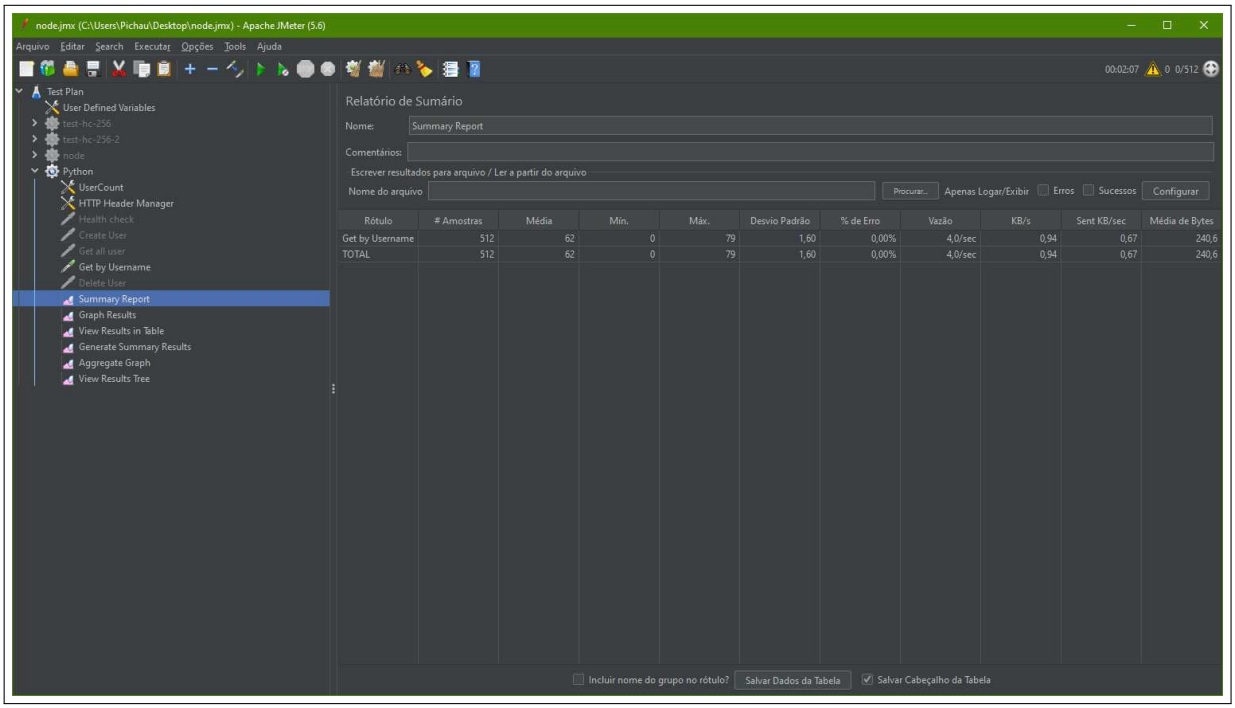


Figura 53 – Relatório de sumário - Python - 1024 requisições - Recuperar um usuário

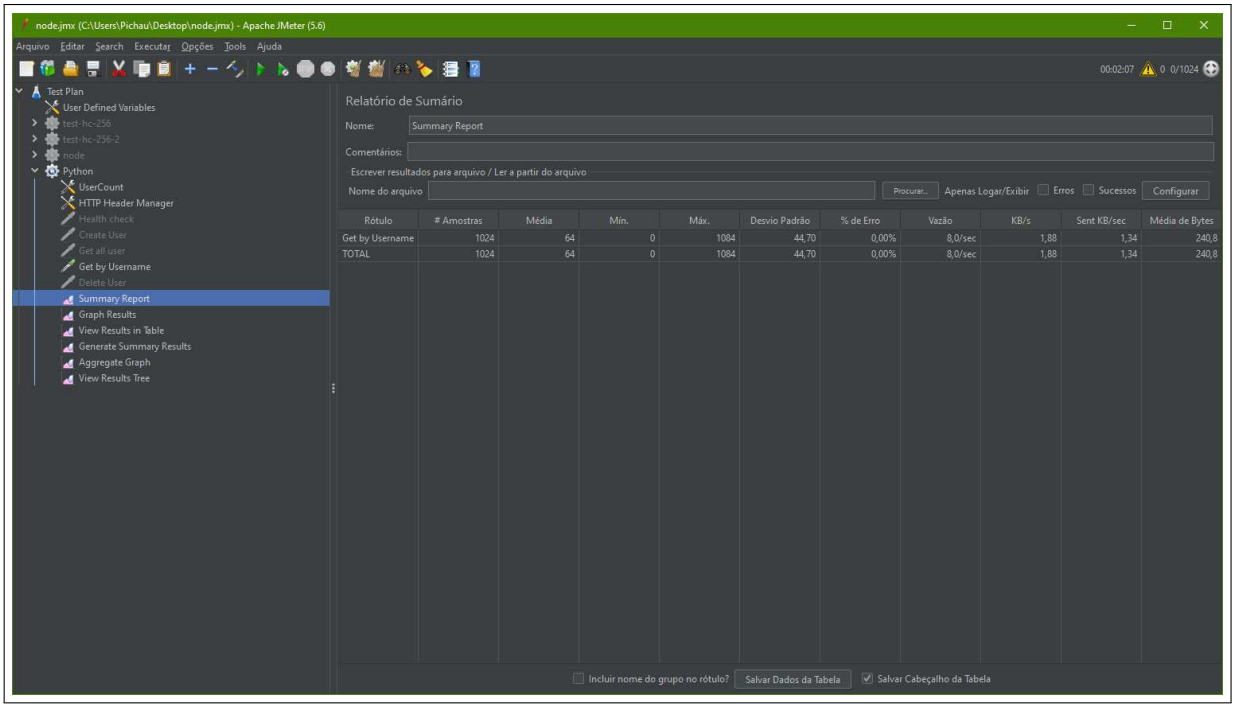


Figura 54 – Relatório de sumário - Python - 2048 requisições - Recuperar um usuário

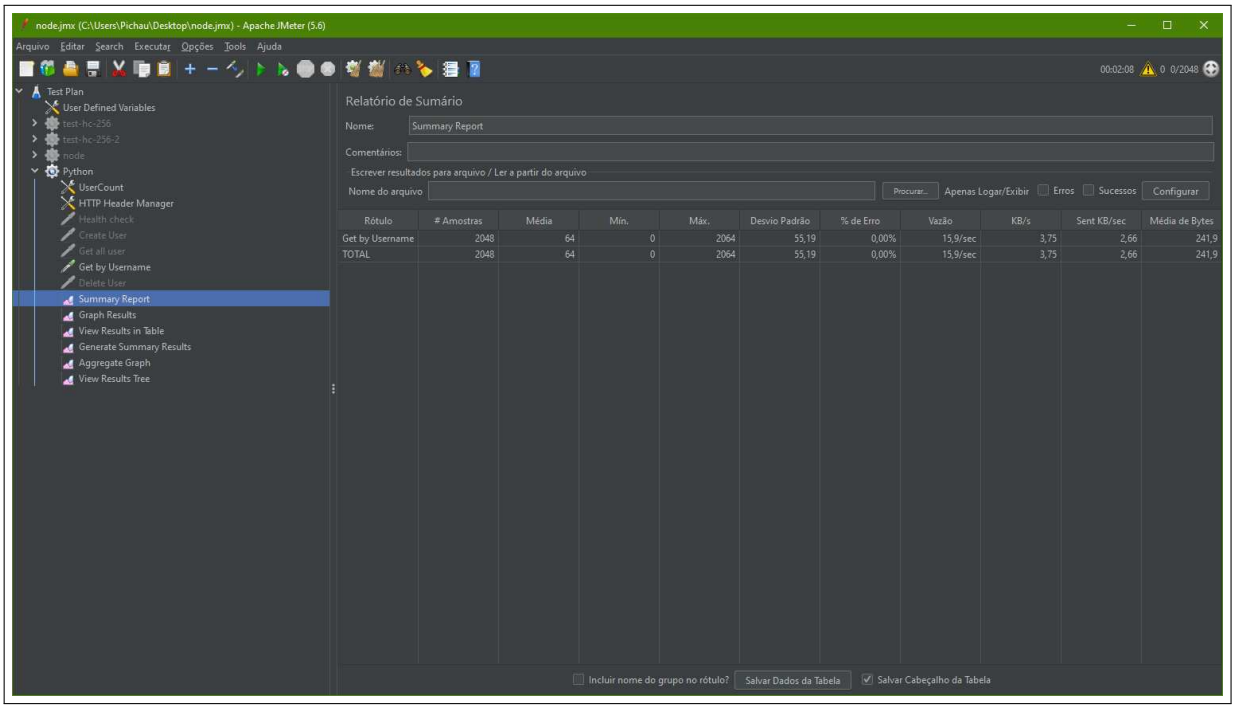


Figura 55 – Relatório de sumário - Python - 128 requisições - Recuperar todo os usuários

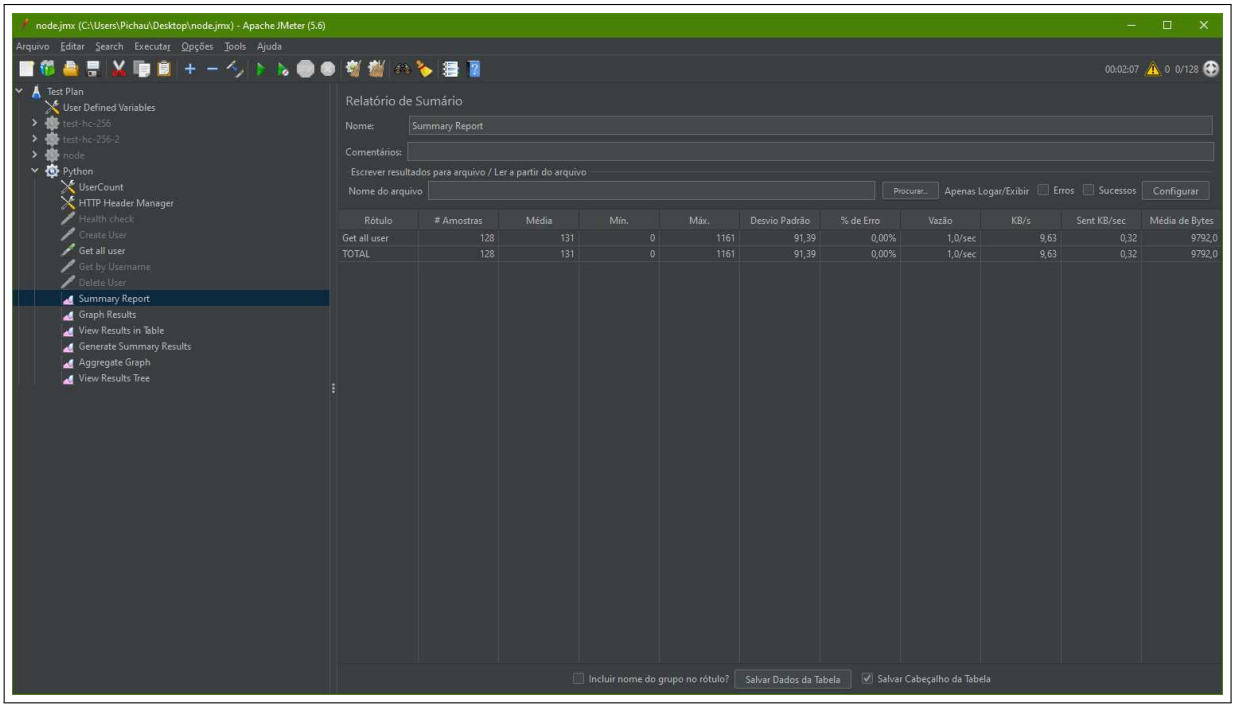


Figura 56 – Relatório de sumário - Python - 256 requisições - Recuperar todo os usuários

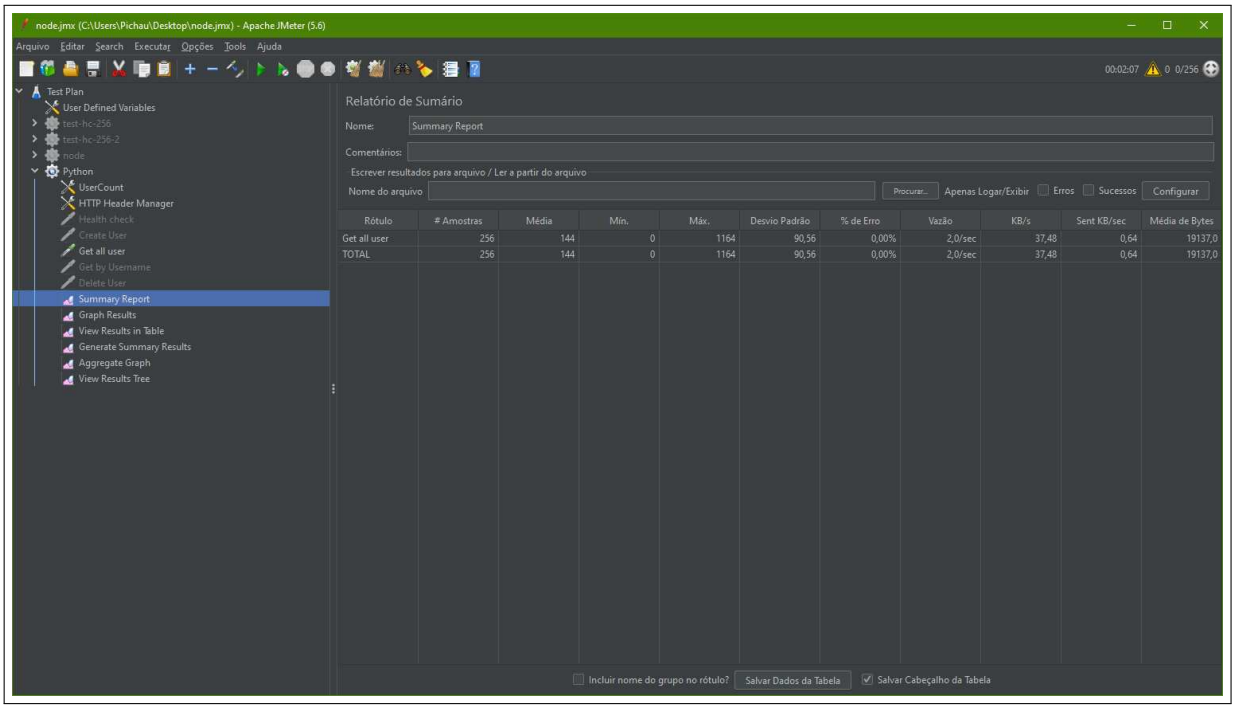


Figura 57 – Relatório de sumário - Python - 512 requisições - Recuperar todo os usuários

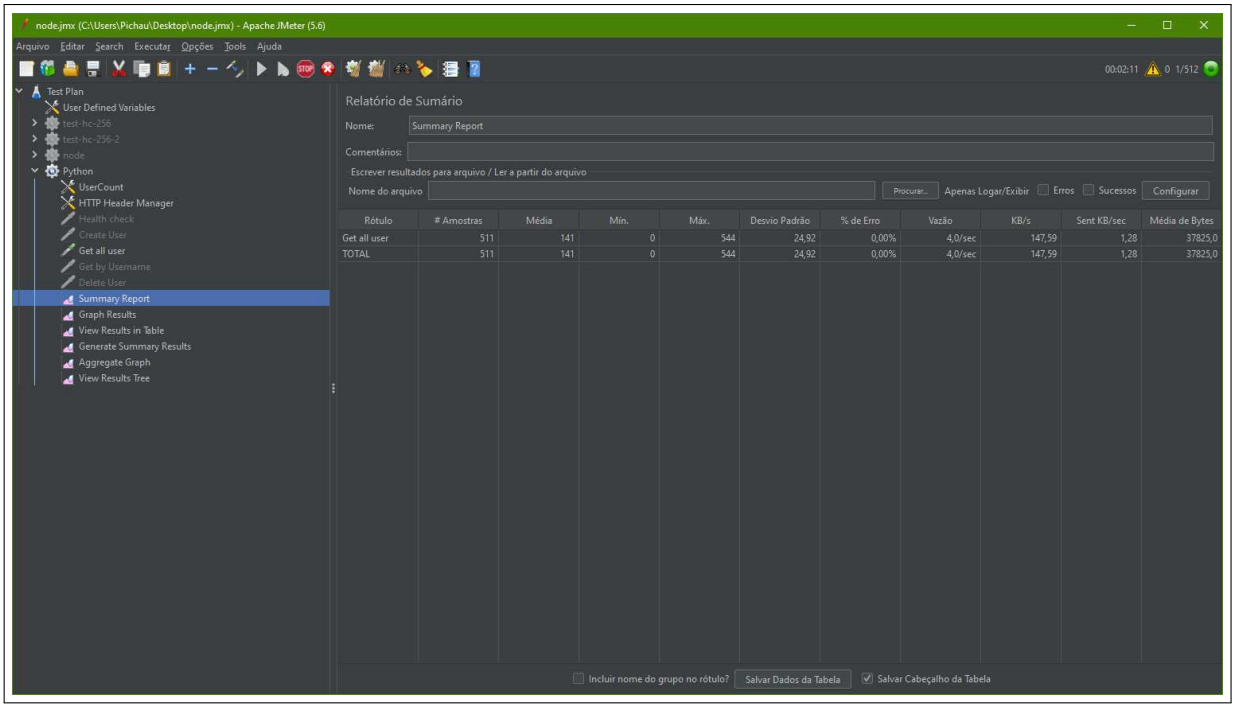


Figura 58 – Relatório de sumário - Python - 1024 requisições - Recuperar todo os usuários

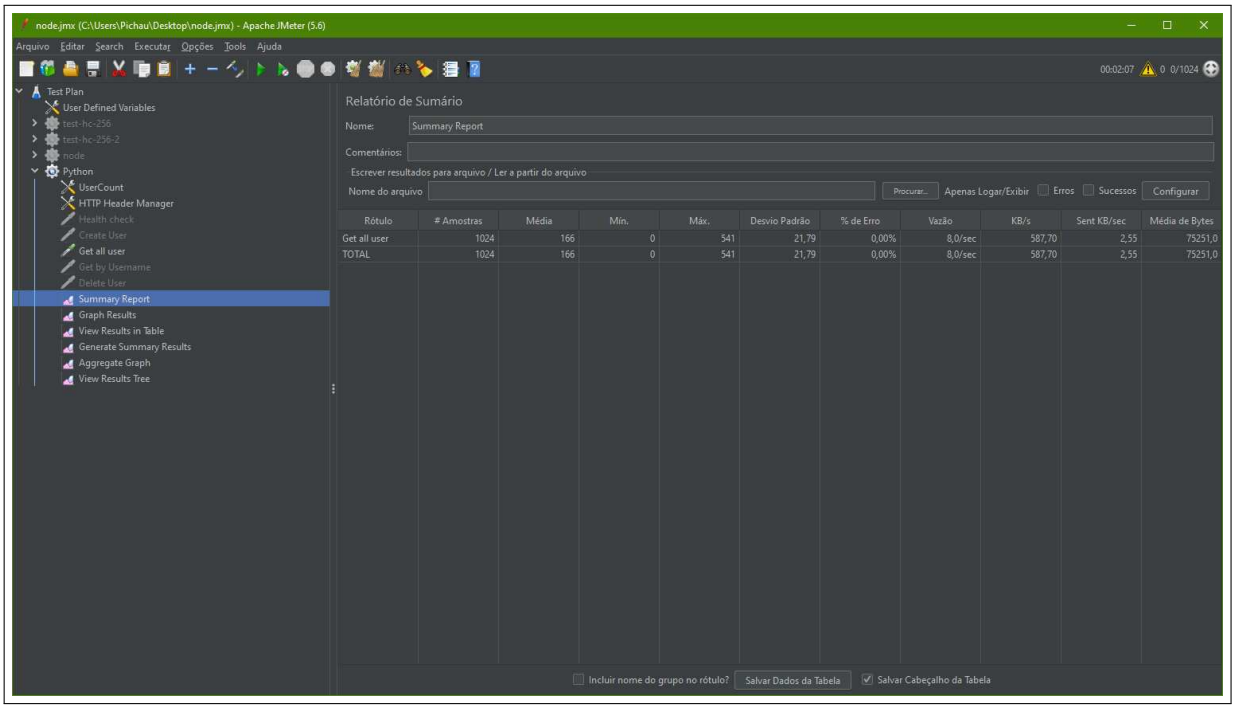


Figura 59 – Relatório de sumário - Python - 2048 requisições - Recuperar todo os usuários