Proposta de analise do impacto financeiro de diferentes linguagens de programação em ambiente de nuvem computacional

Guilherme Gomes Caires

Projeto Final de Graduação em Engenharia de Redes de Comunicação

DEPARTAMENTO DE ENGENHARIA ELÉTRICA
FACULDADE DE TECNOLOGIA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

PROJETO FINAL DE GRADUAÇÃO

Proposta de analise do impacto financeiro de diferentes linguagens de programação em ambiente de nuvem computacional

Guilherme Gomes Caires

Projeto Final de Graduação submetida ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Bacharel em Engenharia de Redes de Comunicação

Banca Examinadora

Prof. Fábio Lúcio Lopes de Mendonça, Ph.D, FT/UnB ______________________
Presidente - Orientador

Prof. Georges Daniel Amvame Nze, Ph.D, FT/UnB ______________________
Examinador Interno

Prof. Daniel Alves da Silva, Ph.D, PPEE/UnB, ______________________
Examinador Externo

FICHA CATALOGRÁFICA

CAIRES, GUILHERME GOMES
Proposta de analise do impacto financeiro de diferentes linguagens de programação em ambiente de nuvem computacional [Distrito Federal] 2023.
xvi, 41 p., 210 x 297 mm (ENE/FT/UnB, Bacharel, Engenharia Elétrica, 2023).
Projeto Final de Graduação - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. Linguagens de programação	2. Nuvem computacional
3. Função como serviço	4. Rust
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

CAIRES, G. (2023). *Proposta de analise do impacto financeiro de diferentes linguagens de programação em ambiente de nuvem computacional*. Projeto Final de Graduação, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 41 p.

CESSÃO DE DIREITOS

AUTOR: Guilherme Gomes Caires
TÍTULO: Proposta de analise do impacto financeiro de diferentes linguagens de programação em ambiente de nuvem computacional.
GRAU: Bacharel em Engenharia de Redes de Comunicação ANO: 2023

É concedida à Universidade de Brasília permissão para reproduzir cópias deste Projeto Final de Graduação e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. Do mesmo modo, a Universidade de Brasília tem permissão para divulgar este documento em biblioteca virtual, em formato que permita o acesso via redes de comunicação e a reprodução de cópias, desde que protegida a integridade do conteúdo dessas cópias e proibido o acesso a partes isoladas desse conteúdo. O autor reserva outros direitos de publicação e nenhuma parte deste documento pode ser reproduzida sem a autorização por escrito do autor.

Guilherme Gomes Caires
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

DEDICATÓRIA E AGRADECIMENTOS

Agradeço aos meus pais, Lindauto Caires Ribeiro e Wanessa Gomes Caires, por todo o suporte e amor incondicional, tornando possível toda essa longa jornada.

Ao meu irmão Gustavo Henrique Gomes Caires, meu maior e melhor amigo, que sempre me apoiou em todas as decisões em que eu tive e pelas incontáveis risadas e bons momentos.

Aos meus pássaros, Tony e Caco, que infelizmente se foram durante o percurso da minha graduação, e ao Galileu que sempre tornam meus dias melhores, por mais caótico que tenha sido, e por me darem tanta felicidade e companhia mesmo nos meus piores momentos.

À minha namorada Beatriz Diniz da Conceição, que mesmo quilômetros de distância, nunca nos faltou o amor e carinho, por cada momento de saudade que nos fortaleceu e a cada sonho compartilhado.

Ao meu orientador professor Fábio Lúcio Lopes de Mendonça, por todo o apoio e orientação durante esse trabalho.

A todos os professores que tive contato durante todos os meus anos escolares e acadêmicos.

RESUMO

Este trabalho tem a proposta de analisar o impacto financeiro de diferentes linguagens de programação no ambiente cloud. As simulações do trabalho realizadas em plataformas de Function as a Service para analisar como cada linguagem de programação se comporta com a plataforma Lambda da Amazon. Web Services. Alem do efeito financeiro iremos analisar o tempo de execução de cada uma das linguagens abordadas, com maior aprofundamento na linguagem Rust.

Palavras-chave: Desempenho, Rust, Python, Node, AWS, Computação em nuvem, Otimização.

ABSTRACT

This work proposes to analyze the financial impact of different programming languages in the cloud environment. Work simulations performed on Function as a Service platforms to analyze how each programming language behaves with Amazon Web Services Lambda. In addition to the financial effect, we will analyze the execution time of each of the languages discussed, with greater depth in the Rust language.

Keywords: Performance, Rust, Python, Node, AWS, Cloud Computing, Optimization.

SUMÁRIO

1 INTRODUÇÃO **1**
1.1 OBJETIVO GERAL 2
1.2 OBJETIVOS ESPECÍFICOS 3
1.3 REVISÃO DA LITERATURA 3
1.4 ESTRUTURA DO TRABALHO 3

2 REFERENCIAL TEÓRICO **5**
2.1 COMPUTAÇÃO EM NUVEM 5
2.2 GARBAGE COLLECTION 6
2.3 RUST 7
2.4 PYTHON 9
2.5 NODE.JS 10

3 PROCESSO EXPERIMENTAL **12**
3.1 ARQUITETURA TECNOLÓGICA 12
3.2 ANÁLISE DE *Cold Start* E *Warm Start* 13
3.2.1 ANÁLISE DE *Cold Start* 13
3.2.2 ANÁLISE DE *Warm Start* 13
3.3 FERRAMENTAL UTILIZADO 14
3.3.1 RUST 14
3.3.2 NODE.JS 14
3.3.3 PYTHON 15
3.3.4 SHELL SCRIPT 15
3.3.5 BANCO DE DADOS NÃO RELACIONAL 16
3.4 DESCRIÇÃO DO HARDWARE UTILIZADO 17
3.5 DESCRIÇÃO DO AMBIENTE DE SOFTWARE UTILIZADO 17
3.6 CONSTRUÇÃO DAS FUNÇÕES 18
3.6.1 FUNÇÕES EM RUST 18
3.6.2 FUNÇÕES EM NODE.JS E PYTHON 19
3.7 IMPLEMENTAÇÃO DAS FUNÇÕES 19
3.7.1 RUST 19
3.7.2 PYTHON 23
3.7.3 NODE.JS 25
3.8 COLETA DE DADOS 27
3.8.1 TEMPO DE EXECUÇÃO EM *Cold Start* 27
3.8.2 TEMPO DE EXECUÇÃO EM *Warm Start* 27
3.8.3 VALORES ASSOCIADOS 27
3.8.4 MEMÓRIA USADA 28

4 RESULTADOS EXPERIMENTAIS **29**
4.1 ANÁLISE DE *Cold Start* 29
4.2 ANÁLISE DE WARM START 31
4.3 IMPACTO FINANCEIRO 34

5 CONCLUSÕES **35**
5.1 TRABALHOS FUTUROS 35

REFERÊNCIAS BIBLIOGRÁFICAS **37**

LISTA DE FIGURAS

2.1 Sistema de posse [1] 8
2.2 Sistema de alocação das demais linguagens[1] 8

3.1 Arquitetura da função com integração com banco de dados. Fonte: elaborado pelo autor (2023) 12

4.1 Gráfico do tempo médio de resposta (Cold Start) 31
4.2 Gráfico para $n = 10^3$ 32
4.3 Gráfico para $n = 10^5$ 33
4.4 Gráfico para $n = 10^6$ 33

LISTA DE TABELAS

4.1 Tempo de resposta em *Cold Start* Crivo de Erastótenes 29
4.2 Tempo de resposta em *Cold Start* inserção no banco de dados.................................... 30

LISTA DE SIGLAS

AWS	*Amazon Web Services*
UnB	Universidade de Brasília
IEEE	*Institute of Electrical and Electronic Engineers*
ISO	*Internacional Organization for Standardization*
ABNT	Associação Brasileira de Normas Técnicas
RAM	*Random Access Memory*
TCP/IP	Transmission Control Protocol/Internet Protocol
REST	*Representational State Transfer*
HTTP	*Hypertext Transfer Protocol*
SQL	*Structured Query Language*
IO	*Input/Output*
API	*Application Programming Interface*
CPU	*Central Processing Unit*
EC2	*Elastic Compute Cloud*
RDS	*Relational Database Service*
IAM	*Identity and Access Management*
SSH	*Secure Shell*
JIT	*Just-in-time*
ARM	*Advanced RISC Machines*

1 INTRODUÇÃO

As linguagens de programação permitem que um programador crie programas a partir de um conjunto de ordens, ações consecutivas, dados e algoritmos. Neste contexto a linguagem de programação é um vocabulário e um conjunto de regras gramaticais usadas para escrever programas de computador. Esses programas instruem o computador a realizar determinadas tarefas específicas, sendo uma das principais metas das linguagens de programação é que programadores tenham uma maior produtividade, permitindo expressar suas intenções mais facilmente do que quando comparado com código de máquina. Assim, linguagens de programação são projetadas para adotar uma sintaxe de nível mais alto, que pode ser mais facilmente entendida por programadores humanos. Linguagens de programação são ferramentas importantes para que programadores e engenheiros de software possam escrever programas mais organizados e com maior rapidez. [2]

As Tecnologias Web consistem em conhecer as origens da Internet no mundo e no Brasil, entender os fundamentos da arquitetura da Internet, pois nos dias de hoje é difícil identificar uma área que ainda não tenha investido em tecnologia da informação e se beneficiado com as facilidades trazidas pela informação tratada em tempo real e disponível sob demanda. Para isso é necessário que se utilize de linguagens de programação para cada caso específico.

A computação em nuvem tem sido cada vez mais utilizada ao redor do mundo, como utilizaremos a mesma para realizar teste em ambientes de funções como serviços será necessário explicarmos o que é a nuvem computacional e como o seu uso vem crescendo no mercado. Assim realizamos diversos testes para comparar o impacto financeiro e de performance para algumas linguagens de programação no meio de funções como serviço [3].

Com o avanço da sociedade humana moderna, serviços básicos e essenciais são quase todos entregues de uma forma completamente transparente. Serviços de utilidade pública como água, eletricidade, telefone e gás tornaram-se fundamentais para nossa vida diária e são explorados por meio do modelo de pagamento baseado no uso [4]. As infraestruturas existentes permitem entregar tais serviços em qualquer lugar e a qualquer hora, de forma que possamos simplesmente acender a luz, abrir a torneira ou usar o fogão. O uso desses serviços é, então, cobrado de acordo com as diferentes políticas de tarifação para o usuário final. Recentemente, a mesma ideia de utilidade tem sido aplicada no contexto da informática e uma mudança consistente neste sentido tem sido feita com a disseminação de *Cloud Computing* ou Computação em Nuvem.

Computação em nuvem é uma tendência recente de tecnologia cujo objetivo é proporcionar serviços de Tecnologia da Informação (TI) sob demanda com pagamento baseado no uso. Tendências anteriores à computação em nuvem foram limitadas a uma determinada classe de usuários ou focadas em tornar disponível uma demanda específica de recursos de TI, principalmente de informática [5]. Computação em nuvem pretende ser global e prover serviços para as massas que vão desde o usuário final que hospeda seus documentos pessoais na Internet até empresas que terceirizam toda infraestrutura de TI para outras empresas. Nunca uma abordagem para a utilização real foi tão global e completa: não apenas recursos

de computação e armazenamento são entregues sob demanda, mas toda a pilha de computação pode ser aproveitada na nuvem.

A partir destes contextos, motivou-se a ideia de se usar diversas linguagens de programação que possam garantir cada vez mais performance com menos recurso de processamento, memória e rede. Haja visto que esses recursos são essenciais para o bom desempenho de qualquer sistema, de forma que isso pode impactar no custo de serviços oferecidos em computação em nuvem. Hoje em dia as principais linguagens de programação procuram ser mais rápidas consumindo cada vez menos esses recursos. Assim as principais linguagens web são:

- JavaScript;
- Python;
- Java;
- PHP;
- Linguagens C e C++;
- Rust;

Dessa forma, a proposta deste trabalho é realizar um Estudo entre as principais linguagens de programação, identificando o custo benefício dessas linguagens em funções como serviço. Para isso, pretende realizar a elaboração de um modelo de monitoramento, voltado para o gerenciamento de performance, com um comparativo entre mínimo de 3 linguagens, a partir de uma pequena aplicação identificando os elementos de hardware e software identificando o uso de recursos.

Em primeiro momento, será apresentada a arquitetura e como será estruturada a comunicação de rede entre os micro serviços. Neste momento, será possível observar de maneira global toda a arquitetura do projeto, orientando-o para os micro serviços que serão implementados. Em segundo momento, será possível observar a performance (tempo gasto) e o consumo de processamento e memoria.

Já em terceiro momento, será apresentado como está estruturado o ambiente *cloud*, realizando teste com as linguagens Python, Node.js e Rust. Explicando o processo de configuração e criação do ambiente de teste utilizando componentes de computação em nuvem. Utilização dos dados capturados como estudo comparativo.

Por fim, será apresentado todo o teste realizado e identificando para esse estudo a linguagem que teve melhor comportamento em uma ambiente *cloud*. Portanto, será possível observar os resultados práticos obtidos a partir do monitoramento de tempo de resposta e quantidade de consumo de processamento e memória.

1.1 OBJETIVO GERAL

Este trabalho tem a proposta de analisar o impacto financeiro de diferentes linguagens de programação no ambiente cloud. As simulações do trabalho realizadas em plataformas de Function as a Service para

analisar como cada linguagem de programação se comporta com a plataforma Lambda da Amazon Web Services. Além do efeito financeiro iremos analisar o tempo de execução de cada uma das linguagens abordadas, com maior aprofundamento na linguagem Rust.

1.2 OBJETIVOS ESPECÍFICOS

Para alcançar esse objetivo temos alguns pontos específicos:

- Análise detalhada da tecnologia estudada;
- Propor uma análise do custo benefício do uso de diferentes linguagens;
- Simular situações de monitoramento através de uma prova de conceito;

1.3 REVISÃO DA LITERATURA

A base da bibliografia referenciada neste trabalho levou em conta a busca por artigos, teses, monografias e livro em diversas fontes, especialmente, a Universidade de Brasília (UnB) e *Institute of Electrical and Electronic Engineers* (IEEE). Além disso foram realizadas pesquisas em bases de dados dos organismos de normalização *Internacional Organization for Standardization* (ISO) e Associação Brasileira de Normas Técnicas (ABNT). Foram realizadas pesquisas na base bibliográfica da UnB, que é uma instituição de renome.

A ideia deste projeto tem como base a utilização dos modelos já apresentados através do artigo [6] e, [7], entretanto no trabalho de [6] trata do monitoramento por sistemas de GPS com equipamentos com tecnologias de alto custo e não se preocupa com sistemas de controle de energia, já no artigo [7] ele realiza do controle de gado e não consegue uma localização precisa dos animais.

Apesar das pesquisas dos artigos citados acima apresentarem trabalhos relevantes, nenhum deles aborda a análise proposta neste projeto. No entanto o trabalho proposto pretende-se realizar um sistema de monitoramento através de dispositivos IoT de baixo custo, com precisão e um controle energético dos dispositivo.

1.4 ESTRUTURA DO TRABALHO

Este trabalho será organizado entre os próximos capítulos em quatro partes principais: fundamentação teórica, proposta, análise de resultados e conclusões.

O Capítulo 2 apresenta os principais conceitos e tecnologias que compõem este trabalho, bem como apresenta a arquitetura, softwares e metodologia dos estudos conduzidos com o funcionamento detalhado do sistema.

O Capítulo 3 apresenta uma seção tem como finalidade promover uma visão detalhada da estrutura do

processo experimental do projeto, de forma que a mesma possa ser reproduzida posteriormente em outros trabalhos, a fim de atender a outras propostas de estudo.

O Capítulo 4 mostra de forma detalhada o desenvolvimento do trabalho e uma análise dos resultados obtidos em cada cenário considerado.

O Capítulo 5 contém a discussão final do estudo e conclui o trabalho, apresentando possíveis trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 COMPUTAÇÃO EM NUVEM

O *National Institute of Standards and Technology* (NIST) define a computação em nuvem em cinco características. Atendimento sob demanda: os usuários da nuvem podem provisionar recursos de computação de forma independente, sem exigir interação humana com o provedor de serviços. Amplo acesso à rede: os recursos da nuvem podem ser acessados por meio de uma rede, como a Internet, e por meio de uma variedade de dispositivos. Pool de recursos: os recursos da nuvem são atribuídos e reatribuídos dinamicamente de acordo com a demanda do usuário. Os recursos de computação do provedor de serviços são agrupados para atender a vários consumidores usando um modelo multi-locatário, com diferentes recursos físicos e virtuais atribuídos e reatribuídos dinamicamente de acordo com a demanda do consumidor. Elasticidade rápida: os serviços em nuvem podem aumentar ou diminuir rapidamente para acomodar demandas em constante mudança, permitindo uma resposta rápida a picos no tráfego de usuários. Serviço medido: o uso de recursos de nuvem é monitorado, controlado e relatado, fornecendo transparência tanto para o provedor quanto para o consumidor do serviço utilizado [8].

Em relação aos modelos de implementação, uma nuvem privada é dedicada a uma única organização, enquanto uma nuvem pública é disponibilizada ao público em geral e uma nuvem híbrida é uma combinação de nuvens privadas e públicas, permitindo que dados e aplicativos sejam compartilhados entre elas.

Infraestrutura como serviço (IaaS): IaaS fornece infraestrutura de computação virtualizada, como máquinas virtuais, armazenamento e rede, pela Internet. Isso permite que os usuários aluguem recursos de computação sob demanda, sem precisar comprar e manter sua própria infraestrutura física, assim somente é necessário que o cliente selecione a máquina a qual deseja utilizar executar a aplicação que desejar. Exemplos de provedores de IaaS incluem Amazon Web Services (AWS), Microsoft Azure e Google Cloud Platform (GCP).

Plataforma como serviço (PaaS): A capacidade fornecida ao consumidor é implantar no aplicativo de infraestrutura de nuvem criados ou adquiridos pelo consumidor criados usando programação linguagens e ferramentas suportadas pelo provedor. O consumidor não gerencia ou controla a infraestrutura de nuvem subjacente, incluindo rede, servidores, sistemas operacionais ou armazenamento, mas tem controle sobre os aplicativos implantados e possivelmente o ambiente de hospedagem de aplicativos configurações. Os exemplos mais conhecidos de plataforma como serviço são AWS Elastic Beanstalk, Windows Azure e Google App Engine.

Software como serviço (SaaS): A capacidade fornecida ao consumidor é usar os recursos do provedor aplicativos executados em uma infraestrutura de nuvem. Os aplicativos são acessíveis a partir de vários dispositivos clientes por meio de uma interface, como um navegador da web. O consumidor não gerencia ou controla a infraestrutura de nuvem subjacente, incluindo rede, servidores, sistemas operacionais, armazenamento ou até mesmo recursos de aplicativos individuais, com a possível exceção das configurações limitadas de configuração de aplicativos específicos do usuário. Exemplos de software como serviço

incluem serviços de e-mail na web, Github e Salesforce.

Function as a Service (FaaS): FaaS fornece uma plataforma para executar funções de computação orientadas a eventos, permitindo que os usuários executem código sem ter que gerenciar a infraestrutura de servidores. Esse modelo de serviço permite que os desenvolvedores escrevam e implementem funções individuais, em vez de um código monolítico, e execute-os somente quando acionados por eventos específicos, como redimensionamento de uma imagem ao ser enviada para um servidor. Exemplos de provedores de FaaS incluem AWS Lambda, Google Cloud Functions e Microsoft Azure Functions.

Em resumo, os quatro modelos de serviço em computação em nuvem fornecem uma variedade de opções para as organizações, dependendo de suas necessidades específicas de computação. Desde a infraestrutura de baixo nível de infraestrutura como serviço a aplicações disponibilizadas por alguns provedores, os usuários escolhem o modelo de serviço que melhor atende às suas necessidades e objetivos.

2.2 GARBAGE COLLECTION

A gestão eficiente da memória é uma tarefa crítica no desenvolvimento de sistemas de software. À medida que os programas crescem em complexidade, a alocação e desalocação de memória manual torna-se um desafio cada vez maior. A alocação estática de memória, em que o programador é responsável por gerenciar explicitamente o ciclo de vida dos objetos, pode resultar em vazamentos de memória e erros difíceis de depurar. Para resolver esse problema, surgiram técnicas automáticas de gerenciamento de memória, sendo a mais amplamente utilizada a técnica conhecida como *Garbage Collection* (Coleta de Lixo).

A *Garbage Collection* é uma abordagem automática para a gerência de memória, que visa aliviar os programadores da responsabilidade de alocar e desalocar memória manualmente. Com a Garbage Collection, o sistema toma a responsabilidade de identificar objetos não utilizados e liberar a memória ocupada por eles, permitindo que sejam reutilizados posteriormente. Essa técnica oferece benefícios significativos, como maior produtividade do desenvolvedor, redução de vazamentos de memória e eliminação de erros relacionados à gerência manual de memória. [9]

No contexto da *Garbage Collection*, o termo "lixo"refere-se aos objetos que não são mais referenciados pelo programa e que, portanto, não podem ser acessados futuramente. O processo consiste em identificar e marcar esses objetos não utilizados e, em seguida, liberar a memória ocupada por eles. Para isso, um coletor de lixo realiza uma análise dos objetos em tempo de execução, identificando aqueles que ainda possuem referências válidas e aqueles que estão livres para serem desalocados. [10]

Existem diversas estratégias e algoritmos de Garbage Collection, cada um com características e trade-offs distintos. Alguns algoritmos são baseados em marcação e varredura, enquanto outros utilizam a contagem de referências ou análise estática para determinar a utilização da memória. Além disso, a eficiência da coleta de lixo é um fator crítico, uma vez que a execução de um coletor de lixo ineficiente pode resultar em pausas prolongadas na aplicação, afetando negativamente o desempenho e a experiência do usuário.

Neste trabalho de conclusão de curso, exploraremos os fundamentos teóricos da *Garbage Collection*,

apresentando os principais conceitos e técnicas utilizadas nesse campo. Analisaremos diferentes abordagens de coleta de lixo, discutindo suas vantagens e desvantagens. Além disso, abordaremos aspectos relacionados à implementação de um coletor de lixo eficiente, considerando fatores como coleta concorrente, geração de código otimizado e estratégias de gerenciamento de memória em sistemas de tempo real.

Por fim, destacaremos as tendências e avanços recentes na área de *Garbage Collection*, incluindo técnicas de compilação just-in-time (JIT) e adaptação dinâmica de políticas de coleta de lixo. Compreender e dominar os conceitos e técnicas é fundamental para o desenvolvimento de software robusto e eficiente, especialmente em linguagens de programação modernas que adotam a coleta automática de lixo como parte de sua infraestrutura de execução. [11]

2.3 RUST

O gerenciamento eficiente da memória é um desafio crítico no desenvolvimento de software, impactando diretamente o desempenho, a segurança e a confiabilidade das aplicações. Tradicionalmente, muitas linguagens de programação adotam a técnica de *Garbage Collection* (Coleta de Lixo) como uma abordagem automática para gerenciar a alocação e desalocação de memória. No entanto, essa abordagem pode introduzir *overheads* significativos e imprevisíveis, bem como limitar o controle granular sobre a memória. [12]

Em contraste com a maioria das linguagens, a linguagem de programação Rust adota um modelo de gerenciamento de memória inovador, que não depende de *Garbage Collection*. Esse modelo é baseado em propriedade (*ownership*) e empréstimos (*borrowing*), onde o compilador de Rust realiza análises estáticas para garantir a validade das referências em tempo de compilação. Isso permite que a linguagem evite a necessidade de um coletor de lixo em tempo de execução, fornecendo segurança de memória e alto desempenho.

Se executarmos o código a seguir:

Código 1

```
let s1 = String::from("hello");
let s2 = s1;
```

Em muitas linguagens, isso significaria que s1 e s2 estão apontando para o mesmo local na memória, ou seja, ambos estão apontando para a string "hello". No entanto, em Rust, esse não é o caso.

Quando você atribui s1 a s2, Rust realmente executa uma operação de movimento, não uma cópia superficial. Isso significa que s1 é movido para s2 e s1 não é mais válido. Portanto, eles não compartilham o mesmo local de memória. Após a atribuição, você não pode mais usar s1. Se você tentar usar s1 depois disso, receberá um erro de compilação. Isso ocorre porque o Rust impõe uma regra de "propriedade única", o que significa que apenas uma variável pode "possuir"um dado por vez.

Eis por que o Rust faz isso: quando s2 sai do escopo, o Rust chama automaticamente a função drop para liberar a memória. Se s1 e s2 fossem válidos e apontassem para a mesma memória, você teria um erro

de liberação dupla quando s1 e s2 saíssem do escopo, pois ambos tentariam liberar a mesma memória. Esta é uma fonte comum de bugs e vulnerabilidades de segurança em outros idiomas. Ao garantir propriedade única, o Rust evita esse problema.

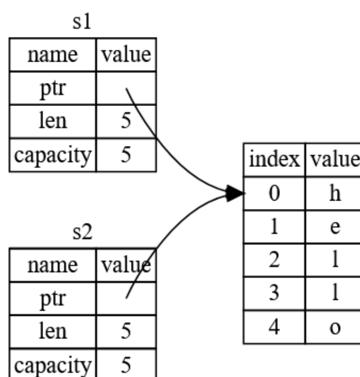

Figura 2.1: Sistema de posse [1]

Se você deseja ter duas variáveis com os mesmos dados, pode criar uma cópia profunda de s1 e atribuí-la a s2 assim:

Código 2

```
let s1 = String::from("hello");
let s2 = s1.clone();
```

Agora s1 e s2 são duas strings diferentes armazenadas em dois locais de memória diferentes, mas ambas contêm os mesmos dados "hello".

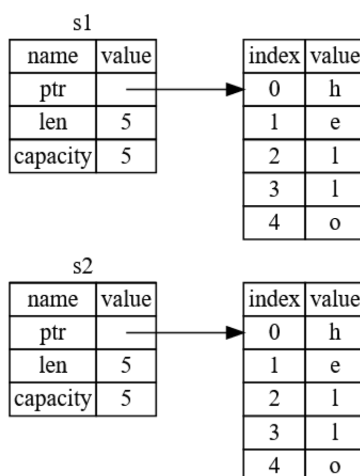

Figura 2.2: Sistema de alocação das demais linguagens[1]

O modelo de propriedade e empréstimos de Rust oferece benefícios distintos em termos de desempenho. Ao eliminar a sobrecarga de execução associada à coleta de lixo, a linguagem Rust é capaz de proporcionar uma alocação de memória mais previsível e um controle refinado sobre o ciclo de vida dos

objetos. Isso resulta em um uso eficiente dos recursos de hardware e reduz a variabilidade no tempo de execução, tornando-a uma escolha atrativa para aplicações com requisitos de desempenho rigorosos.

Outro aspecto importante é a segurança fornecida pelo modelo de propriedade e empréstimos. A análise estática realizada pelo compilador de Rust permite detectar erros de tempo de execução, como referências inválidas, vazamentos de memória e race conditions, antes mesmo da compilação do código. Isso garante que os programas em Rust sejam robustos e confiáveis, eliminando muitas classes comuns de erros de memória.

A abordagem de gerenciamento de memória de Rust também se baseia em outros mecanismos, como referências imutáveis e mutáveis, que permitem o compartilhamento seguro de dados entre diferentes partes do programa. Esses mecanismos são controlados pelo sistema de tipos de Rust, que garante a integridade dos dados e previne problemas como condições de corrida, sem comprometer a eficiência e o desempenho.

Neste trabalho, propomos uma análise mais aprofundada do modelo de gerenciamento de memória de Rust em comparação com a técnica de Garbage Collection. Investigaremos os princípios teóricos subjacentes a ambos os métodos, discutindo suas vantagens e limitações. Além disso, exploraremos os aspectos de segurança e confiabilidade proporcionados pelo modelo de propriedade e empréstimos de Rust.

Ao aprofundar nosso conhecimento sobre o modelo de gerenciamento de memória de Rust, seremos capazes de compreender as nuances teóricas e práticas que o tornam uma alternativa atraente à técnica de Garbage Collection em muitos cenários. Essa análise contribuirá para um melhor entendimento das escolhas de design de linguagens de programação e seus impactos no desenvolvimento de software eficiente, seguro e confiável.

2.4 PYTHON

Python é uma linguagem de programação interpretada de alto nível que ganhou popularidade significativa nos setores acadêmico e industrial devido à sua simplicidade e recursos poderosos. Foi concebido no final dos anos 1980 por Guido van Rossum no Centrum Wiskunde & Informatica (CWI) na Holanda como sucessor da linguagem ABC [13]. O Python foi projetado com ênfase na legibilidade do código e sua sintaxe permite que os programadores expressem conceitos em menos linhas de código do que seria possível em linguagens como C++ ou Java.

A filosofia de design do Python está encapsulada no documento "The Zen of Python", que inclui aforismos como "Contagens de legibilidade"e "Simples é melhor que complexo"[14]. Esse foco na simplicidade e legibilidade fez do Python uma linguagem ideal para iniciantes, enquanto sua extensa biblioteca padrão e um vasto ecossistema de pacotes de terceiros o tornaram uma ferramenta poderosa para profissionais e pesquisadores.

Python tem várias vantagens que contribuíram para seu uso generalizado. Sua sintaxe simples e digitação dinâmica facilitam o aprendizado e o uso, enquanto sua natureza interpretada permite prototipagem rápida e desenvolvimento iterativo. A extensa biblioteca padrão do Python, conhecida como a filosofia "baterias incluídas", fornece uma ampla gama de funcionalidades, desde o desenvolvimento da Web até a

computação científica. Além disso, os fortes recursos de integração e processamento de texto do Python, juntamente com sua comunidade robusta e ecossistema de pacotes de terceiros, o tornam uma linguagem versátil para vários domínios [15].

No entanto, o Python tem suas desvantagens. Sua natureza interpretada pode levar a uma velocidade de execução mais lenta em comparação com linguagens compiladas como C ou Java. Embora isso não seja um problema para muitos aplicativos, pode ser um fator limitante para tarefas de computação intensiva. O Global Interpreter Lock (GIL) do Python, um mecanismo usado no CPython para sincronizar o acesso a objetos Python, também pode limitar o desempenho de aplicativos multi-encadeados).

Apesar dessas limitações, os pontos fortes do Python levaram à sua ampla adoção em vários campos, incluindo desenvolvimento web, análise de dados, aprendizado de máquina, inteligência artificial e computação científica. Seu crescimento não mostra sinais de desaceleração e continua a ser uma ferramenta vital no kit de ferramentas do programador moderno.

2.5 NODE.JS

O Node.js é um ambiente de tempo de execução JavaScript de back-end, de plataforma cruzada e de código aberto que é executado no mecanismo V8 e executa o código JavaScript fora de um navegador da web. Foi criado por Ryan Dahl em 2009, com seu crescimento sendo patrocinado pela Joyent, uma provedora de soluções de hospedagem e computação em nuvem. Dahl foi motivado a criar o Node.js devido à sua insatisfação com as possibilidades limitadas do servidor web mais popular da época, Apache HTTP Server, para lidar com muitas conexões simultâneas e a forma mais comum de criação de código, quando o código bloqueava todo o processo ou implicava múltiplas pilhas de execução no caso de conexões simultâneas. [16]

O Node.js permite o desenvolvimento de aplicativos da Web intensivos em E/S, como sites de streaming de vídeo, aplicativos de página única e outros aplicativos da Web. Ele é construído no mecanismo JavaScript V8 do Chrome, que compila o JavaScript diretamente no código de máquina nativo, melhorando o desempenho.

Uma das principais vantagens do Node.js é sua arquitetura sem bloqueio e orientada a eventos, que o torna adequado para aplicativos em tempo real. Também é leve e eficiente, perfeito para aplicativos em tempo real com uso intensivo de dados executados em dispositivos distribuídos. Além disso, como o Node.js usa JavaScript tanto para o cliente quanto para o servidor, ele torna o processo de desenvolvimento mais consistente e eficiente). [17]

No entanto, o Node.js também tem suas desvantagens. Seu modelo de programação assíncrona e sem bloqueio pode ser mais difícil de entender para desenvolvedores provenientes de programas síncronos. Além disso, embora o Node.js seja excelente para aplicativos vinculados a I/O, não é tão adequado para tarefas com uso intensivo de CPU, pois podem bloquear solicitações recebidas, levando à ineficiência.

No contexto das funções sem servidor, o Node.js mostrou vantagens significativas. Sua natureza sem bloqueio e orientada a eventos se alinha bem com a arquitetura sem servidor, projetada para lidar com

solicitações de forma independente e escalar automaticamente. A natureza leve do Node.js permite inicializações a frio mais rápidas, um fator importante na computação sem servidor, em que as funções geralmente são iniciadas e interrompidas com frequência. Além disso, o extenso ecossistema de módulos Node.js disponíveis via npm pode ser aproveitado em funções sem servidor, fornecendo uma ampla gama de funcionalidades.

Apesar desses desafios, o Node.js ganhou popularidade significativa devido ao seu desempenho, escalabilidade e vantagens de produtividade do desenvolvedor. Ele é usado por muitos sites de grande escala e alto tráfego, demonstrando sua robustez e confiabilidade em ambientes de produção.

3 PROCESSO EXPERIMENTAL

Este capítulo fornece uma visão geral abrangente do projeto experimental e da metodologia empregada para investigar os tempos de inicialização a frio das funções do AWS Lambda escritas em Python, Node.js e Rust. O objetivo desses experimentos foi analisar comparativamente o desempenho dessas linguagens de programação em um ambiente serverless.

3.1 ARQUITETURA TECNOLÓGICA

A estrutura experimental foi estabelecida no AWS Lambda, um importante serviço de computação sem servidor oferecido pela Amazon Web Services (AWS). O AWS Lambda é um serviço de computação orientado a eventos que executa código em resposta a eventos, gerenciando automaticamente os recursos de computação, permitindo assim que os desenvolvedores se concentrem apenas em seu código.

Para cada uma das três linguagens de programação sob investigação - Python, Node.js e Rust - foram criadas duas funções distintas do AWS Lambda. A primeira função foi projetada para executar uma tarefa simples: inserir duas variáveis recebidas de uma solicitação POST em uma tabela do DynamoDB. A segunda função foi projetada para realizar uma tarefa computacionalmente mais intensiva: executar o algoritmo Crivo de Eratóstenes, dado um número de entrada 'N'. Essas tarefas foram escolhidas para fornecer uma visão equilibrada do desempenho de cada linguagem em cenários limitados por E/S e limitados por CPU.

Cada função foi empacotada com todas as dependências necessárias e carregada no AWS Lambda. As funções foram configuradas para usar o máximo de memória disponível (3008 MB no momento do experimento) para minimizar o impacto das restrições de recursos nos tempos de inicialização a frio. Um API Gateway foi configurado para acionar essas funções por meio de uma solicitação POST, com os parâmetros necessários passados no corpo da solicitação.

A tabela do DynamoDB usada pela primeira função foi configurada com um esquema simples, contendo campos para as duas variáveis e o UID. A tabela foi hospedada na mesma região que as funções do Lambda para minimizar a latência da rede.

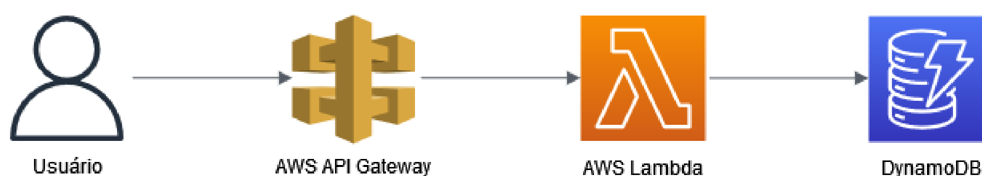

Figura 3.1: Arquitetura da função com integração com banco de dados.
Fonte: elaborado pelo autor (2023)

A função Crivo de Eratóstenes foi projetada para retornar uma lista de todos os números primos até um determinado número 'N'. A função foi implementada de forma a aproveitar os pontos fortes específicos

de cada idioma, ao mesmo tempo em que fornece uma comparação justa. Para avaliar o desempenho de cada linguagem em lidar com diferentes cargas computacionais, o valor de 'N' foi variado em uma faixa de valores. Isso nos permitiu observar como o desempenho de cada linguagem escala com o aumento da complexidade computacional.

O ambiente AWS Lambda fornece um ambiente consistente e controlado para esses experimentos, garantindo que os resultados sejam influenciados exclusivamente pelo desempenho das linguagens de programação e não por quaisquer fatores externos, como latência de rede ou variabilidade de hardware.

3.2 ANÁLISE DE *COLD START* E *WARM START*

A análise de *Cold Start* e *Warm Start* é um aspecto crucial deste estudo, pois essas métricas fornecem informações valiosas sobre o desempenho de funções sem servidor. No contexto do AWS Lambda, uma inicialização a frio ocorre quando uma função é invocada após algum tempo sem ser utilizada, fazendo com que a AWS aloque um novo worker para atender a requisição. Por outro lado, uma inicialização a quente ocorre quando uma função reutiliza um trabalhador de uma chamada anterior.

3.2.1 Análise de *Cold Start*

Para medir com precisão os tempos de inicialização a frio, as funções do Lambda foram invocadas após ficarem ociosas por um período significativo. O AWS Lambda considera uma função como "inicialização a frio"se não for invocada por aproximadamente 5 minutos. Portanto, cada função foi invocada após um período inativo de 5 minutos para garantir uma inicialização a frio.

A hora de início precisa da execução da função foi registrada imediatamente quando a função começou sua execução, e a hora de término foi registrada antes da função retornar sua resposta. O tempo de inicialização a frio foi calculado como a diferença entre esses dois parametros de data/hora, fornecendo uma medida precisa do tempo necessário para uma inicialização a frio.

Este processo foi repetido várias vezes para cada linguagem de programação e cada função, com valores variados de 'N' para a função Crivo de Eratóstenes. Isso resultou em um conjunto de dados robusto para análises subsequentes.

3.2.2 Análise de *Warm Start*

Para a análise de partidas a quente, as funções foram invocadas em rápida sucessão, sem nenhum tempo ocioso entre elas. Isso garantiu que as invocações de função fossem inicializações a quente, reutilizando trabalhadores de invocações anteriores.

Assim como na análise de inicialização a frio, os horários de início e término da execução da função foram registrados e o horário de inicialização a quente foi calculado como a diferença entre data e hora.

Novamente, esse processo foi repetido várias vezes para cada linguagem de programação e cada função,

com valores variados de 'N' para a função Crivo de Eratóstenes.

3.3 FERRAMENTAL UTILIZADO

3.3.1 Rust

Para o desenvolvimento de uma das funções foi utilizado *Rust*. uma linguagem multiparadigma projetada para desempenho e segurança, foi selecionada como uma das linguagens para este estudo devido à sua combinação única de recursos. Originário da *Mozilla Research*, o *Rust* é uma linguagem compilada estaticamente tipada que oferece características de desempenho semelhantes ao C++. No entanto, o modelo de propriedade exclusivo e o foco em abstrações de custo zero fornecem fortes garantias contra erros comuns de programação, como desreferenciação de ponteiro nulo e *buffer overflow*, sem incorrer em custos de tempo de execução.

O design do Rust o torna particularmente adequado para programação em nível de sistema, onde o controle sobre os recursos do sistema e a previsibilidade do desempenho são fundamentais. Essas características também tornam Rust uma escolha atraente para computação sem servidor, onde a utilização eficiente de recursos pode se traduzir diretamente em economia de custos. Além disso, o crescente ecossistema de Rust e a comunidade ativa de código aberto sugerem que ele está bem posicionado para atender às crescentes necessidades da computação sem servidor.

Além desses recursos, o design de linguagem moderna do Rust, que inclui recursos como correspondência de padrões, inferência de tipo e gerenciamento de simultaneidade, o torna uma escolha atraente para desenvolvedores que buscam um equilíbrio entre desempenho, segurança e produtividade do desenvolvedor. O foco da linguagem em explicitação e clareza também ajuda a tornar o código Rust fácil de ler e manter, o que é uma vantagem significativa em grandes bases de código ou ao trabalhar em um ambiente de equipe.

3.3.2 Node.js

O Node.js foi escolhido para este estudo devido à sua ampla adoção no campo de desenvolvimento da Web e seu modelo de I/O sem bloqueio e orientado a eventos. Construído no mecanismo JavaScript V8 do Chrome, o Node.js trouxe o JavaScript, uma linguagem tradicionalmente restrita ao navegador, para o lado do servidor. Isso permitiu o desenvolvimento de aplicativos de rede altamente escaláveis usando uma linguagem familiar para muitos desenvolvedores da web.

A arquitetura orientada a eventos do Node.js o torna particularmente adequado para lidar com tarefas vinculadas a I/O, onde pode gerenciar com eficiência um grande número de conexões simultâneas com uso mínimo de recursos. No entanto, a natureza de encadeamento único do Node.js pode ser uma limitação para tarefas vinculadas à CPU, pois ele não pode aproveitar facilmente os processadores com vários núcleos sem recorrer a processos filhos ou encadeamentos de trabalho.

No contexto da computação sem servidor, essas características tornam o Node.js um estudo de caso in-

teressante. Embora seu modelo orientado a eventos se encaixe bem com a natureza de solicitação-resposta sem estado das funções sem servidor, suas características de desempenho para tarefas vinculadas à CPU podem apresentar desafios.

3.3.3 Python

O Python foi selecionado para este estudo devido à sua simplicidade, legibilidade e amplo uso em uma ampla gama de aplicações. Como uma linguagem interpretada de alto nível, o Python enfatiza a legibilidade do código, permitindo que os desenvolvedores expressem ideias complexas em menos linhas de código do que seria possível em linguagens de nível inferior. A extensa biblioteca padrão do Python e o rico ecossistema de pacotes de terceiros ampliam ainda mais seus recursos, tornando-o uma ferramenta versátil para tudo, desde o desenvolvimento da Web até a análise de dados e o aprendizado de máquina.

No entanto, o desempenho do Python pode ser uma limitação, especialmente em comparação com linguagens compiladas como Rust. O Global Interpreter Lock (GIL) do Python também pode ser um gargalo para aplicativos *multithreaded*, embora isso seja menos preocupante em um contexto sem servidor, no qual as funções são normalmente executadas em instâncias separadas de encadeamento único.

No contexto da computação sem servidor, a simplicidade e a versatilidade do Python o tornam uma escolha popular para desenvolver e implantar funções rapidamente. No entanto, seu desempenho relativo e características de uso de recursos oferecem um contraste interessante com linguagens como Rust e Node.js.

A popularidade do Python nas áreas de ciência de dados e aprendizado de máquina também o torna uma escolha relevante para este estudo. A disponibilidade de bibliotecas poderosas para computação numérica e aprendizado de máquina tornam o Python uma escolha comum para tarefas que exigem computação pesada. Isso torna o desempenho do Python em um ambiente sem servidor uma importante área de estudo.

3.3.4 Shell script

Para gerar uma quantidade significativa de requisições às funções do AWS Lambda e simular um cenário do mundo real, foi utilizado um shell script. Shell scripting é uma ferramenta poderosa que permite a automação de tarefas em um ambiente de sistema operacional semelhante ao Unix. Neste estudo, o shell script foi projetado para enviar uma série de solicitações HTTP para as funções do AWS Lambda.

O shell script foi configurado para enviar uma solicitação POST para o URL específico da função AWS Lambda. O script foi configurado para repetir essa solicitação, enviando efetivamente a solicitação várias vezes. Isso nos permitiu medir o desempenho das funções do Lambda sob uma carga mais representativa de um caso de uso do mundo real.

No entanto, o envio repetido de solicitações de um único endereço IP em um curto período de tempo pode ser confundido com um ataque de negação de serviço (DoS) por medidas de segurança em vigor em muitas plataformas, incluindo a AWS. Para evitar que nossas solicitações legítimas sejam bloqueadas, implementamos uma estratégia de lista de permissões de IP.

A lista de permissões de IP é um recurso de segurança frequentemente usado para controlar o acesso

a uma rede ou serviço. Ao adicionar nosso endereço IP a uma lista de permissões, conseguimos informar à AWS que nosso endereço IP era confiável e não deveria ser bloqueado, mesmo ao fazer solicitações repetidas em um curto período de tempo.

Isso foi obtido definindo as configurações do grupo de segurança da AWS para permitir o tráfego de entrada de nosso endereço IP específico. Isso garantiu que nossas solicitações não fossem confundidas com um ataque DoS e pudessem alcançar as funções do Lambda.

Essa combinação de shell script para solicitações automatizadas e lista de permissões de IP para evitar o bloqueio forneceu um conjunto de ferramentas robusto para testar o desempenho das funções do AWS Lambda de maneira controlada, mas realista.

Código 3

```
#!/bin/bash

URL="endpoint"  # Mudar de acordo com o endpoint

for i in {1..1000}
do
   curl -X POST -d "first_name=Guilherme&last_name=Caires" $URL
done
```

3.3.5 Banco de Dados não relacional

Os bancos de dados NoSQL, também conhecidos como bancos de dados "não relacionais"ou "não SQL", são um tipo de banco de dados que fornece um mecanismo para armazenamento e recuperação de dados modelados em meios diferentes das relações tabulares usadas em bancos de dados relacionais. Eles são particularmente úteis para trabalhar com grandes conjuntos de dados distribuídos. Cada vez mais usados em big data e aplicativos da Web em tempo real. Eles podem armazenar dados estruturados, semi estruturados ou não estruturados e não requerem um esquema fixo. Essa flexibilidade permite o desenvolvimento rápido e a manipulação de modelos de dados que não são adequados para bancos de dados relacionais.

Os bancos não relacionais usam uma variedade de modelos de dados, incluindo documento, gráfico, valor-chave, na memória e pesquisa. Esses tipos de bancos de dados são otimizados para aplicativos que exigem grande volume de dados, baixa latência e modelos de dados flexíveis, que são obtidos relaxando algumas das restrições de consistência de dados dos bancos de dados tradicionais.

O Amazon DynamoDB é um banco de dados de valores-chave e documentos que oferece desempenho de milissegundos de um dígito em qualquer escala. É um banco de dados totalmente gerenciado, multirregional, ativo e durável com segurança integrada, backup e restauração e armazenamento em cache na memória para aplicativos em escala de Internet. Pode lidar com mais de 10 trilhões de solicitações por dia e suportar picos de mais de 20 milhões de solicitações por segundo. Ele fornece escalabilidade perfeita,

permitindo o manuseio de grandes quantidades de dados e tráfego.

No contexto deste estudo, o DynamoDB foi usado para armazenar e recuperar os dados necessários para o teste de desempenho das três linguagens de programação. Suas características de escalabilidade e desempenho o tornaram a escolha ideal para este estudo de alta intensidade e foco no desempenho.

3.4 DESCRIÇÃO DO HARDWARE UTILIZADO

Para este estudo, utilizamos as funções AWS Lambda configuradas com 128 MB de memória. A quantidade de memória alocada para uma função do Lambda tem implicações diretas em seu desempenho. O AWS Lambda aloca potência de CPU, largura de banda de rede e I/O de disco proporcionalmente à quantidade de memória configurada. Portanto, uma função com 128 MB de memória executa com menos poder computacional em comparação com uma função com mais memória.

É importante observar que, embora 128 MB seja a menor configuração de memória disponível para as funções do AWS Lambda, isso não significa necessariamente que seja insuficiente. Dependendo da natureza da tarefa, uma função com 128 MB de memória pode funcionar adequadamente. No entanto, para tarefas de computação mais intensiva, pode ser necessária uma configuração de memória mais alta.

Além da configuração de memória, também utilizamos a arquitetura ARM para nossas funções Lambda. Os processadores ARM são um tipo de microprocessador baseado na arquitetura RISC (computador com conjunto reduzido de instruções) desenvolvido pela Advanced RISC Machines (ARM). Os processadores ARM são amplamente utilizados em dispositivos móveis, eletrodomésticos e outros sistemas embarcados devido ao seu pequeno tamanho, baixo consumo de energia e desempenho razoável.

No contexto do AWS Lambda, as funções baseadas em ARM são executadas nos processadores AWS Graviton2. Eles são personalizados pela Amazon Web Services usando núcleos Arm Neoverse de 64 bits. Os processadores Graviton2 oferecem um grande salto em desempenho e recursos em relação aos processadores AWS Graviton de primeira geração. Eles oferecem desempenho de preço até 40% melhor em comparação com as instâncias baseadas em x86 da geração atual para uma ampla variedade de cargas de trabalho.

Ao utilizar o AWS Lambda com 128 MB de memória e arquitetura ARM, pretendemos investigar as características de desempenho de funções sem servidor sob configurações de recursos restritos e em uma arquitetura de processador cada vez mais comum na nuvem. Essa configuração fornece uma perspectiva exclusiva sobre a eficiência e a economia de diferentes linguagens de programação em um ambiente sem servidor.

3.5 DESCRIÇÃO DO AMBIENTE DE SOFTWARE UTILIZADO

As funções do AWS Lambda para este estudo foram escritas em Rust, Node.js e Python. Essas linguagens foram escolhidas para fornecer uma ampla perspectiva sobre as características de desempenho da

computação sem servidor em uma variedade de linguagens de programação populares e emergentes. Cada função foi projetada para executar uma tarefa específica, como inserir dados em uma tabela do DynamoDB ou executar o algoritmo Crivo de Eratóstenes.

As funções foram implementadas em ambiente AWS configurado com 128MB de memória e rodando em arquitetura ARM. Essa configuração foi escolhida para investigar as características de desempenho de funções sem servidor sob configurações de recursos restritos e em uma arquitetura de processador cada vez mais comum na nuvem.

As funções foram invocadas usando um gatilho HTTP por meio do Amazon API Gateway, que é um serviço totalmente gerenciado para criar, implantar e gerenciar APIs seguras em escala. O API Gateway lida com todas as tarefas envolvidas na aceitação e processamento de chamadas de API simultâneas, incluindo gerenciamento de tráfego, transformação de dados e controle de acesso. Todas as funções foram criadas no servidor sa-east-1, localizado em São Paulo.

O desempenho das funções foi monitorado e registrado usando o AWS CloudWatch, um serviço que fornece insights acionáveis para monitorar aplicativos, entender e responder a mudanças de desempenho em todo o sistema e otimizar a utilização de recursos.

3.6 CONSTRUÇÃO DAS FUNÇÕES

3.6.1 Funções em Rust

Para as funções Rust, o processo de desenvolvimento envolveu não apenas escrever e compilar o código, mas também empacotá-lo para implantação no AWS Lambda. No entanto, o AWS Lambda não oferece suporte nativo ao tempo de execução Rust, o que significa que etapas adicionais são necessárias para preparar as funções Rust para implantação.

Uma dessas etapas é compilar o código Rust em um ambiente que corresponda o mais próximo possível ao ambiente AWS Lambda. AWS Linux 2 é um sistema operacional Linux de código aberto fornecido pela AWS. Ele foi projetado para fornecer um ambiente de execução seguro, estável e de alto desempenho para aplicativos executados na AWS. É o sucessor do Amazon Linux original, que é o sistema operacional subjacente para o ambiente de execução AWS Lambda.

Ao compilar o código Rust no AWS Linux 2, podemos garantir que o binário resultante seja compatível com o ambiente de execução do AWS Lambda. Isso é importante porque diferentes sistemas operacionais e ambientes podem ter diferentes bibliotecas e dependências do sistema, e um binário compilado em um sistema pode não ser executado corretamente em outro.

Depois que o código Rust é compilado em um binário no AWS Linux 2, o binário é compactado em um arquivo .zip junto com quaisquer outros arquivos necessários. Esse arquivo .zip é carregado no AWS Lambda para criar uma nova função. Durante o processo de criação da função, o manipulador é definido com o nome do binário Rust, que é a opção usada para tempos de execução personalizados.

3.6.2 Funções em Node.js e Python

Para as funções Node.js e Python, o processo de desenvolvimento foi semelhante ao Rust, mas com algumas diferenças devido à natureza interpretada dessas linguagens. O código para essas funções foi escrito em um ambiente de desenvolvimento local e depois empacotado em um arquivo .zip, assim como as funções do Rust.

No entanto, como Node.js e Python são linguagens interpretadas, não havia necessidade de compilar o código em um binário. Em vez disso, o arquivo .zip continha os arquivos de código-fonte originais.

O arquivo .zip foi carregado no AWS Lambda para criar uma nova função. Durante o processo de criação da função, o manipulador foi definido com o nome do arquivo e a função dentro desse arquivo que deve ser executada quando a função do Lambda é invocada. O tempo de execução foi definido para a versão apropriada de Node.js ou Python.

3.7 IMPLEMENTAÇÃO DAS FUNÇÕES

3.7.1 Rust

3.7.1.1 Inserção no banco de dados

A implementação da função AWS Lambda foi feita utilizando a linguagem de programação Rust. A função interage com o DynamoDB da Amazon, um serviço de banco de dados NoSQL que fornece desempenho rápido e previsível com escalabilidade perfeita.

As seguintes bibliotecas foram usadas na implementação:

aws_config, aws_sdk_dynamodb e lambda_runtime: essas bibliotecas AWS SDK para Rust fornecem as interfaces necessárias para interagir com os serviços da AWS, como DynamoDB e AWS Lambda.

serde e serde_json: Essas bibliotecas são usadas para serializar e desserializar dados, permitindo uma fácil conversão entre estruturas de dados Rust e JSON.

uuid: Esta biblioteca é usada para gerar e analisar UUIDs, que são usados como identificadores únicos.

tokio: Este é um tempo de execução assíncrono para Rust, permitindo a execução simultânea de tarefas.

A função principal do programa configura o manipulador de funções do Lambda e, em seguida, executa a função do Lambda. A função do manipulador é o núcleo da função do Lambda. Leva em um CustomEvent e um objeto Context. O CustomEvent é uma estrutura que representa a entrada para a função Lambda, que neste caso inclui first_name e last_name. O objeto Context inclui informações sobre o ambiente de tempo de execução.

A função do manipulador gera um novo UUID, configura um cliente DynamoDB e, em seguida, envia uma solicitação put_item ao DynamoDB para inserir um novo item na tabela "users". O item inclui o UUID, nome e sobrenome.

Serde é usado para definir a estrutura CustomEvent, que é automaticamente desserializada do evento

JSON de entrada. Também é usado para criar a resposta JSON. Tokio é usado para executar a função principal e enviar a solicitação do DynamoDB. O atributo [tokio::main] indica que a função deve ser executada no tempo de execução do Tokio. A palavra-chave .await é usada para aguardar de forma assíncrona a conclusão da solicitação do DynamoDB.

Código 4

```
use aws_config::meta::region::RegionProviderChain;
use aws_sdk_dynamodb::model::AttributeValue;
use aws_sdk_dynamodb::Client;
use lambda_runtime::{handler_fn, Context, Error as LambdaError};
use serde::Deserialize;
use serde_json::{json, Value};
use uuid::Uuid;

#[tokio::main]
async fn main() -> Result<(), LambdaError> {
    let func = handler_fn(handler);
    lambda_runtime::run(func).await?;
    Ok(())
}

#[derive(Deserialize)]
struct CustomEvent {
    first_name: String,
    last_name: String,
}

async fn handler(event: CustomEvent, _: Context) -> Result<Value, LambdaError> {
    let uuid = Uuid::new_v4().to_string();

    let region_provider = RegionProviderChain::default_provider()
        .or_else("us-east-1");
    let config = aws_config::from_env().region(region_provider)
        .load().await;
    let client = Client::new(&config);

    let request = client
        .put_item()
        .table_name("users")
        .item("uid", AttributeValue::S(String::from(uuid)))
        .item(
            "first_name",
```

```
            AttributeValue::S(String::from(event.first_name)),
        )
        .item(
            "last_name",
            AttributeValue::S(String::from(event.last_name)),
        );

    request.send().await?;

    Ok(json!({ "message": "Dados inseridos!" }))
}
```

3.7.1.2 Crivo de Eratóstenes

Outra função AWS Lambda foi implementada para calcular números primos usando o algoritmo Crivo de Eratóstenes. Esta função recebe uma entrada n e retorna todos os números primos até n.

A entrada e a saída são representadas pelas estruturas CustomEvent e CustomOutput, respectivamente. A estrutura CustomEvent possui um único campo n, que é o limite superior para os números primos a serem calculados. A estrutura CustomOutput tem um único campo primos, que é um vetor dos números primos.

A função func é o manipulador da função Lambda. Ele primeiro desserializa o evento de entrada em uma estrutura CustomEvent. Em seguida, ele chama a função sieve_of_eratosthenes para calcular os números primos e constrói uma estrutura CustomOutput com os resultados.

A função sieve_of_eratosthenes implementa o algoritmo Crivo de Eratóstenes. Ele primeiro cria um vetor booleano primos de comprimento n + 1 e inicializa todos os elementos como verdadeiros. Em seguida, ele marca iterativamente os múltiplos de cada número começando em 2 como falsos. Os valores verdadeiros restantes no vetor representam os números primos.

A função func então transforma o vetor booleano em um vetor dos índices dos valores verdadeiros, que representam os números primos. Ele faz isso usando a cadeia into_iter().enumerate().filter().map() de métodos iteradores

Código 5

```
use lambda_runtime::{handler_fn, Context, Error};
use serde::{Serialize, Deserialize};
use serde_json::Value;

#[derive(Deserialize)]
struct CustomEvent {
    #[serde(rename = "n")]
    n: usize,
```

```
}

#[derive(Serialize)]
struct CustomOutput {
    #[serde(rename = "primes")]
    primes: Vec<usize>,
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    let func = handler_fn(func);
    lambda_runtime::run(func).await?;
    Ok(())
}

async fn func(event: Value, _: Context) -> Result<CustomOutput, Error> {
    let event: CustomEvent = serde_json::from_value(event)?;
    let primes = sieve_of_eratosthenes(event.n);
    let prime_numbers = primes.into_iter().enumerate()
        .filter(|&(_, is_prime)| is_prime).map(|(i, _)| i).collect();
    Ok(CustomOutput { primes: prime_numbers })
}

fn sieve_of_eratosthenes(n: usize) -> Vec<bool> {
    let mut primes = vec![true; n + 1];
    primes[0] = false;
    if n >= 1 {
        primes[1] = false;
    }

    let mut p = 2;
    while p * p <= n {
        if primes[p] {
            let mut i = p * p;
            while i <= n {
                primes[i] = false;
                i += p;
            }
        }
        p += 1;
    }
    primes
```

```
}
```

3.7.2 Python

3.7.2.1 Inserção no banco de dados

O código fornecido define uma função do AWS Lambda projetada para manipular dados do usuário. Após a chamada, a função é acionada com um evento que contém detalhes do usuário, especificamente seu nome e sobrenome. Para cada usuário, um identificador único (UUID) é gerado usando o método uuid.uuid4(). Esse ID exclusivo, junto com o nome e o sobrenome do usuário, é armazenado como um item em uma tabela do DynamoDB chamada 'users'. Após armazenar os dados com sucesso, a função conclui sua operação retornando um código de status HTTP 200 acompanhado de uma mensagem indicando que os dados foram gravados. Vale a pena notar que há uma resposta de retorno redundante no final da função, que pode ser removida com segurança, pois a resposta já foi fornecida na etapa anterior. Em essência, essa função atua como um componente de back-end direto sem servidor, capturando e registrando dados do usuário em uma tabela do DynamoDB sempre que ela é acionada por um evento apropriado.

Código 6

```
import json
import uuid
import boto3

dynamodb = boto3.resource('dynamodb')
table_name = 'users'

def lambda_handler(event, context):
    first_name = event['first_name']
    last_name = event['last_name']

    uid = str(uuid.uuid4())

    table = dynamodb.Table(table_name)
    table.put_item(
        Item={
            'uid': uid,
            'first_name': first_name,
            'last_name': last_name
        }
    )

    return {
        'statusCode': 200,
```

```
        'body': json.dumps("Data recorded.")
    }


    return response
```

3.7.2.2 Crivo de Eratóstenes

O código fornecido é projetado para calcular números primos até um inteiro especificado n usando o algoritmo. A lógica central é encapsulada na função sieve_of_eratosthenes. Esta função inicializa uma lista de valores booleanos, peneira, representando números de 0 a n, todos definidos inicialmente como True. Os valores nos índices 0 e 1 são então definidos como Falso, pois nem 0 nem 1 são primos.

O algoritmo então itera sobre números de 2 até a raiz quadrada de n. Para cada número, se não tiver sido marcado como falso o que significa que ainda é considerado primo, o algoritmo marca todos os seus múltiplos como falsos, pois não podem ser primos. Esse processo filtra eficientemente os números não primos.

A função conclui retornando uma lista de números que ainda estão marcados como True, que são os números primos até n.

A função lambda_handler atua como o ponto de entrada quando esse código é implantado como uma função do AWS Lambda. Ele extrai o valor de n do evento recebido, calcula os números primos até n usando a função sieve_of_eratosthenes e, em seguida, retorna esses números primos no corpo da resposta. A resposta é estruturada como um objeto JSON com uma chave primos contendo a lista de números primos.

Código 7

```
import json

def sieve_of_eratosthenes(n):
    sieve = [True] * (n + 1)
    sieve[0:2] = [False, False]
    for current in range(2, int(n**0.5) + 1):
        if sieve[current]:
            for multiple in range(current**2, n + 1, current):
                sieve[multiple] = False
    return [prime for prime, checked in enumerate(sieve) if checked]

def lambda_handler(event, context):
    n = int(event['n'])
    primes = sieve_of_eratosthenes(n)
    return {
        'statusCode': 200,
```

```
        'body': json.dumps({'primes': primes})
    }
```

3.7.3 Node.js

As funções do Python são bastante similares as funções em javascript, assim segue a mesma lógica para os seguintes códigos:

3.7.3.1 Inserção no banco de dados

Código 8

```
const AWS = require('aws-sdk');

const {randomUUID} = require('crypto');

const dynamoDB = new AWS.DynamoDB.DocumentClient();

exports.handler = async (event) => {
  const uid = randomUUID()
  console.log(uid)
  const params = {
    TableName: 'users',
    Item: {
      'uid': uid,
      'first_name': event.first_name,
      'last_name': event.last_name,
    },
  };

  try {
    const data = await dynamoDB.put(params).promise();

    const response = {
      statusCode: 200,
      body: JSON.stringify(data),
    };
    return response;
  } catch (error) {
    console.error(error);
    const response = {
```

```
      statusCode: 500,
      body: JSON.stringify(error),
    };
    return response;
  }
};
```

3.7.3.2 Crivo de Eratóstenes

Código 9

```
function sieveOfEratosthenes(n) {
    var array = [], upperLimit = Math.sqrt(n), output = [];

    for (var i = 0; i < n; i++) {
        array.push(true);
    }

    for (var i = 2; i <= upperLimit; i++) {
        if (array[i]) {
            for (var j = i * i; j < n; j += i) {
                array[j] = false;
            }
        }
    }

    for (var i = 2; i < n; i++) {
        if(array[i]) {
            output.push(i);
        }
    }

    return output;
};

exports.handler = async (event) => {
    const n = event.n;
    const primes = sieveOfEratosthenes(n);
    const response = {
        statusCode: 200,
        body: primes,
```

```
    };
    return response;
};
```

3.8 COLETA DE DADOS

O objetivo principal deste estudo é analisar o desempenho e a relação custo-benefício de funções serverless escritas em diferentes linguagens de programação. Para conseguir isso, coletamos dados sobre várias métricas importantes durante a execução das funções. Essas métricas fornecem insights sobre as características de desempenho e uso de recursos das funções, que são fatores críticos na relação custo-benefício da computação sem servidor.

3.8.1 Tempo de execução em *Cold Start*

O tempo de inicialização a frio é o tempo que leva para uma função iniciar a execução em resposta a um evento após ficar ociosa. As partidas a frio ocorrem quando uma nova instância de uma função é iniciada, o que acontece quando a função é invocada pela primeira vez ou após um período de inatividade. O tempo de inicialização a frio inclui o tempo necessário para carregar e inicializar a função e seu ambiente de tempo de execução. Essa métrica é importante porque pode afetar significativamente a latência de uma função, principalmente para funções que não são invocadas com frequência.

3.8.2 Tempo de execução em *Warm Start*

O tempo de execução de inicialização a quente é o tempo que uma função leva para ser executada em resposta a um evento quando a função já está 'quente', ou seja, quando uma instância da função já está em execução. As inicializações a quente ocorrem quando uma função é invocada repetidamente e o AWS Lambda pode reutilizar uma instância existente da função para lidar com a invocação. O tempo de execução de inicialização a quente geralmente exclui a sobrecarga de inicialização de uma inicialização a frio, tornando-a uma medida mais precisa do desempenho de execução da função

3.8.3 Valores Associados

O Valor da execução de uma função no AWS Lambda é baseado na quantidade de tempo de computação usado e na quantidade de memória alocada para a função. O AWS Lambda mede o tempo de computação em milissegundos, desde o momento em que a função começa a ser executada até retornar ou ser encerrada. O preço depende da quantidade de memória configurada para a função, com configurações de memória mais altas resultando em um custo mais alto por milissegundo de tempo de computação. Ao coletar dados sobre o preço de cada chamada de função, podemos analisar o custo-benefício das funções.

3.8.4 Memória Usada

A quantidade de memória usada por uma função durante sua execução é outra métrica importante. O AWS Lambda fornece uma certa quantidade de memória para funções com base em sua configuração, mas a quantidade real de memória usada pode variar dependendo do comportamento da função. As funções que usam mais memória podem ter um desempenho melhor, mas também custam mais. Ao medir a memória usada por cada função, podemos obter informações sobre o uso e a eficiência dos recursos da função.

Em resumo, ao coletar dados sobre o tempo de inicialização a frio, o tempo de execução da inicialização a quente, o preço e a memória usada, podemos analisar o desempenho e o custo-benefício de funções sem servidor escritas em diferentes linguagens de programação. Esses dados fornecerão informação que podem orientar desenvolvedores e organizações na escolha da linguagem mais apropriada para seus aplicativos sem servidor.

4 RESULTADOS EXPERIMENTAIS

Esta seção apresenta os resultados obtidos a partir da série de experimentos realizados para avaliar o desempenho das linguagens de programação Rust, Node e Python em diversos cenários. Os experimentos foram projetados para medir o tempo de resposta e o uso de memória dessas linguagens em diferentes condições, com o objetivo de identificar a linguagem mais performática.

4.1 ANÁLISE DE *COLD START*

Rust	Node	Python
20.07 ms	178.49 ms	109.84 ms
19.95 ms	174.75 ms	110.60 ms
20.09 ms	175.32 ms	118.56 ms
19.76 ms	177.82 ms	109.37 ms
20.53 ms	176.34 ms	120.80 ms
20.35 ms	176.25 ms	110.89 ms
20.16 ms	174.60 ms	119.65 ms
19.77 ms	174.29 ms	123.18 ms
20.00 ms	174.10 ms	115.92 ms
20.21 ms	174.85 ms	117.92 ms
20.19 ms	177.60 ms	116.96 ms
20.54 ms	175.24 ms	122.77 ms
20.38 ms	175.44 ms	119.38 ms
20.27 ms	175.94 ms	117.10 ms
20.06 ms	178.92 ms	105.94 ms
20.08 ms	176.82 ms	113.49 ms
20.24 ms	175.36 ms	107.16 ms
19.76 ms	176.64 ms	114.38 ms
20.26 ms	177.43 ms	120.35 ms
20.74 ms	174.48 ms	113.39 ms
20.24 ms	175.11 ms	110.01 ms

Tabela 4.1: Tempo de resposta em *Cold Start* Crivo de Erastótenes

A execução do algoritmo apresenta os resultados da 4.1 onde é notável a diferença de perfomance de Rust e as outras duas linguagens. Essa diferença é dada pelo fato de Rust ser uma linguagem compilada e sem um coletor de lixo.

Rust	Node	Python
37.73 ms	538.46 ms	366.32 ms
37.47 ms	539.24 ms	341.89 ms
38.69 ms	554.42 ms	349.60 ms
37.37 ms	538.95 ms	359.65 ms
38.23 ms	532.57 ms	361.51 ms
37.25 ms	559.58 ms	347.11 ms
37.63 ms	546.96 ms	334.46 ms
37.91 ms	530.43 ms	360.53 ms
38.00 ms	537.38 ms	332.98 ms
37.28 ms	545.72 ms	364.86 ms
38.25 ms	528.67 ms	351.32 ms
37.99 ms	554.46 ms	351.72 ms
38.73 ms	547.40 ms	351.03 ms
37.22 ms	546.74 ms	355.80 ms
37.39 ms	535.56 ms	327.08 ms
36.48 ms	557.59 ms	360.12 ms
37.84 ms	545.94 ms	354.30 ms
38.49 ms	549.66 ms	345.67 ms
37.39 ms	543.70 ms	365.28 ms
36.83 ms	541.15 ms	337.59 ms
38.27 ms	533.45 ms	349.14 ms

Tabela 4.2: Tempo de resposta em *Cold Start* inserção no banco de dados

Na análise do desempenho de inicialização a frio em diferentes linguagens de programação, apresentamos uma versão condensada dos dados na tabela acima para fins de brevidade e legibilidade no corpo principal deste documento. O conjunto completo de dados, que inclui uma lista mais extensa de medições, pode ser encontrado na seção de anexos deste documento para referência posterior e análise detalhada.

Os dados ilustram claramente uma vantagem significativa para Rust em termos de tempo de inicialização a frio quando comparado com as outras linguagens em consideração. O Rust demonstra consistentemente os tempos de inicialização a frio mais curtos, o que é um fator crítico no desempenho de funções sem servidor, especialmente em cenários em que as funções são invocadas com frequência após períodos de inatividade. O aumento significativo entre a 4.1 e a 4.2 é dado pelo fato de uma conexão com um serviço de banco de dados ser realizado.

Em contraste, o Node.js exibe os tempos de inicialização a frio mais longos entre os três idiomas. Isso pode levar a tempos de resposta iniciais mais lentos em aplicativos sem servidor, o que pode ser perceptível para os usuários finais e pode afetar a experiência geral do usuário.

Essas descobertas se alinham com a filosofia de design da Rust, que enfatiza o desempenho e a utilização eficiente de recursos. A compilação antecipada do Rust, tempo de execução mínimo e gerenciamento de memória eficiente contribuem para seu desempenho superior de inicialização a frio. Por outro lado, Node.js, sendo uma linguagem interpretada com um tempo de execução maior, possui inerentemente mais componentes para carregar e inicializar, o que pode aumentar seus tempos de inicialização a frio.

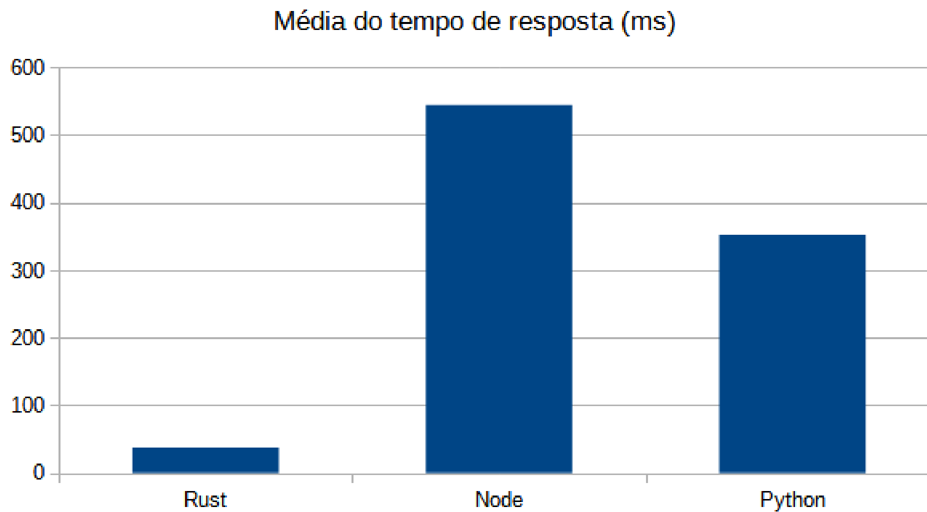

Figura 4.1: Gráfico do tempo médio de resposta (Cold Start)

4.2 ANÁLISE DE WARM START

Na análise de inicialização a quente para o algoritmo Crivo de Erastótenes $n = 10^3$, observamos um padrão diferente em comparação com o cenário de inicialização a frio. Os tempos de execução em todas as três linguagens são relativamente semelhantes, indicando que o desempenho de inicialização a quente dessas linguagens é bastante comparável para essa tarefa computacional específica.

No entanto, uma diferença notável surge quando consideramos o uso de memória. Rust, com foco em abstrações de custo zero e utilização eficiente de recursos, usa a menor quantidade de memória entre as três linguagens. O Python, apesar de sua simplicidade e facilidade de uso, também demonstra um uso de memória eficiente, embora ligeiramente superior ao Rust. Node.js, por outro lado, usa mais memória. Isso pode ser atribuído ao modelo de I/O sem bloqueio e orientado a eventos do Node.js, que pode levar a um maior uso de memória em determinadas circunstâncias.

Essas descobertas ressaltam a importância de considerar o tempo de execução e o consumo de recursos ao avaliar o desempenho das funções sem servidor. Embora o tempo de execução possa ser semelhante em diferentes idiomas, o uso de recursos pode variar significativamente, afetando o custo geral e a eficiência do aplicativo sem servidor.

Também vale a pena observar que o desempenho de inicialização a quente é normalmente mais representativo do desempenho real em um ambiente de produção sem servidor, pois as funções sem servidor geralmente são mantidas aquecidas por invocações frequentes. No entanto, o desempenho de inicialização a frio não deve ser negligenciado, especialmente para aplicativos com padrões de uso imprevisíveis ou esporádicos.

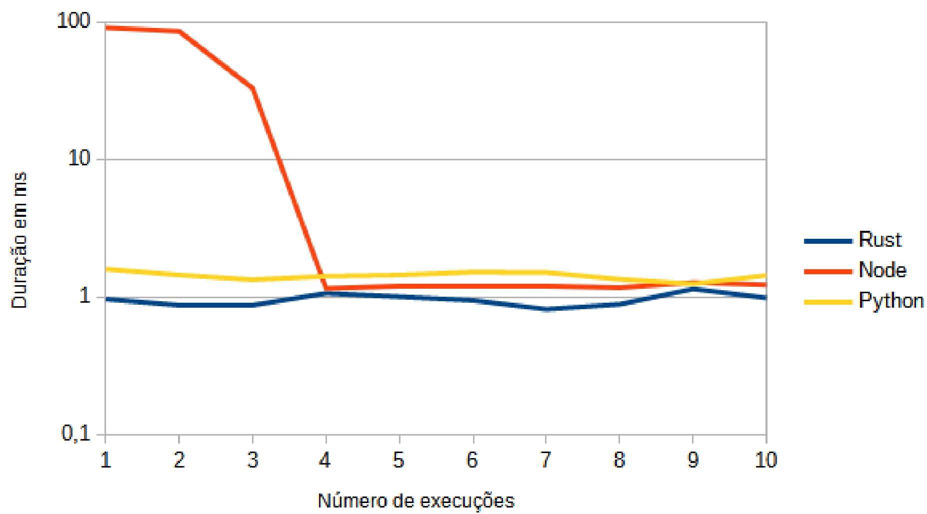

Figura 4.2: Gráfico para $n = 10^3$

Para $n = 10^5$, vemos uma diferença mais pronunciada nos tempos de execução nas três linguagens. Rust continua a demonstrar desempenho superior com os menores tempos de execução, enquanto Node.js e Python exibem tempos de execução significativamente maiores.

Uma das razões para as execuções iniciais mais lentas do Node.js é o processo de compilação Just-In-Time (JIT) usado por seu mecanismo JavaScript V8. A compilação JIT envolve a compilação do código JavaScript para o código de máquina antes de ser executado, o que pode levar a um melhor desempenho a longo prazo. No entanto, esse processo pode introduzir latência adicional durante a execução inicial do código, pois a etapa de compilação precisa ser concluída antes que o código possa ser executado. Isso geralmente é chamado de período de "aquecimento"e é uma característica comum de linguagens que usam compilação JIT(11).

O Python, por outro lado, usa um modelo de execução interpretado, que não envolve uma etapa de compilação separada. Isso significa que o Python não tem um período de "aquecimento"como Node.js, resultando em tempos de execução mais consistentes em diferentes invocações. No entanto, linguagens interpretadas como Python são geralmente mais lentas do que linguagens compiladas como Rust, o que se reflete nos tempos de execução mais altos observados nos dados.

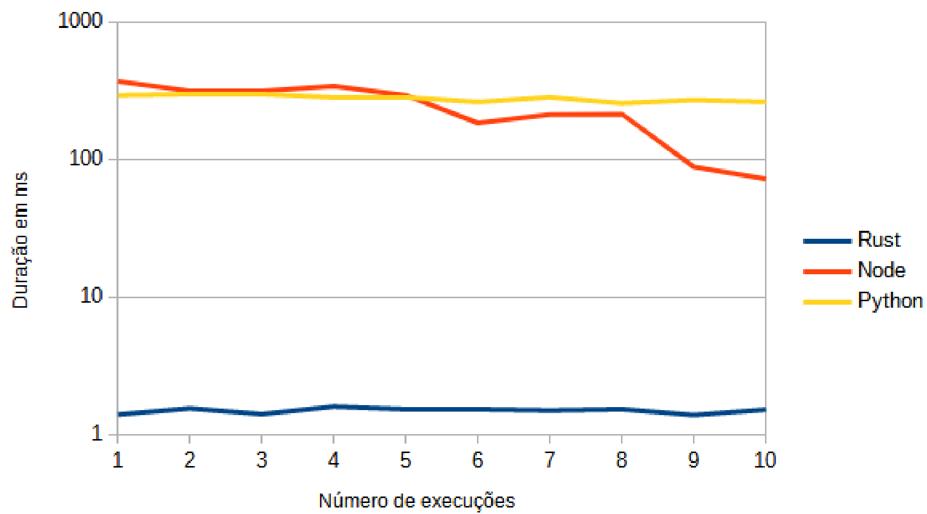

Figura 4.3: Gráfico para $n = 10^5$

Na ultima execução onde $n = 10^6$ é visualizado que a performance de Rust é excepcionalmente mais rápido que as demais linguagens. No caso do Node.js neste experimento, é possível que o número de execuções não tenha sido suficiente para acionar o compilador JIT. Isso significa que o código provavelmente estava sendo interpretado ou compilado usando um método mais lento ao longo do experimento, levando a tempos de execução mais longos que observamos.

Se o experimento fosse executado para um número maior de execuções, poderíamos esperar que os tempos de execução do Node.js diminuíssem à medida que o compilador JIT fosse acionado e começasse a otimizar as partes do código executadas com frequência.

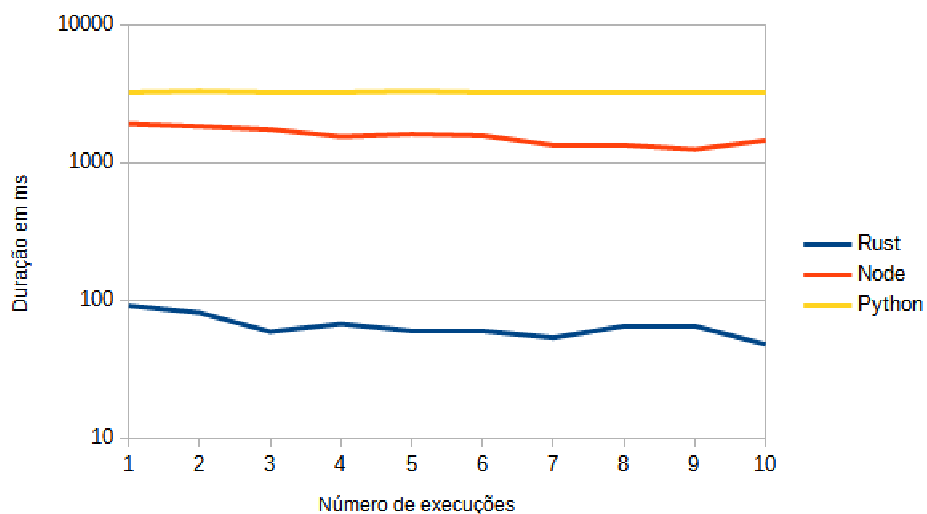

Figura 4.4: Gráfico para $n = 10^6$

4.3 IMPACTO FINANCEIRO

O custo da execução de aplicativos na nuvem é determinado por vários fatores, incluindo a quantidade de tempo de computação usada, a quantidade de memória consumida e a duração da execução. Portanto, uma linguagem que executa mais rápido e usa menos memória pode levar a economias de custos significativas.

Em nosso exemplo, comparamos o custo de execução do algoritmo Sieve com $n = 10^7$ na AWS usando três linguagens diferentes: Node.js, Python e Rust. Os custos foram de 3,44, 6,98 e 0,34, dólares respectivamente.

Ao escolher Rust em vez de Node.js, economizamos aproximadamente 90% do custo. Quando comparado ao Python, a economia é ainda mais substancial, com uma redução de cerca de 95%.

Se você estiver executando um aplicativo pequeno, a diferença pode não parecer grande. No entanto, à medida que a escala de suas operações aumenta, também aumenta o potencial de economia. Para um aplicativo de grande escala que executa milhares ou até milhões de cálculos por dia, uma redução de custo de 90 a 95% pode se traduzir em economias financeiras substanciais.

Além disso, essas economias podem ser ainda mais significativas se você estiver executando tarefas de longa duração e com uso intensivo de memória. Como os provedores de nuvem normalmente cobram pelo tempo de computação e pelo uso de memória, uma linguagem que executa mais rápido e usa menos memória, como Rust, pode levar a uma economia dupla.

Também vale a pena considerar a economia de custos indiretos. Tempos de execução mais rápidos podem levar a uma melhor experiência do usuário, o que pode se traduzir em maior satisfação do usuário e potencialmente maior receita. O menor uso de memória também pode reduzir o risco de erros de falta de memória, levando a aplicativos mais estáveis e confiáveis.

No entanto, é importante observar que essas são economias potenciais. A economia real dependerá de vários fatores, incluindo os detalhes específicos de seu plano de preços de nuvem, a natureza das tarefas que seus aplicativos estão executando e a eficiência de seu código. Também é crucial considerar as compensações. Por exemplo, o Rust pode ser mais eficiente, mas também tem uma curva de aprendizado mais acentuada do que o Node.js ou o Python, o que pode aumentar o tempo e os custos de desenvolvimento.

5 CONCLUSÕES

Ao longo desta pesquisa foi aprofundado nas complexidades do desempenho da linguagem de programação. Os testes empíricos revelaram disparidades significativas em seus tempos de execução, especialmente ao avaliar partidas a frio e a quente. Rust apresentou consistentemente um desempenho superior, com seus tempos de inicialização a frio notavelmente mais baixos do que os outros idiomas. Essa eficiência é atribuída aos recursos de controle e otimização de baixo nível.

Em contraste, Node.js, apesar de sua ampla adoção, exibiu execuções iniciais mais lentas, devido ao processo de compilação JIT do mecanismo V8. Embora a compilação JIT possa melhorar o desempenho em execuções repetidas, nossos testes indicaram que o número limitado de execuções pode não ter acionado totalmente as otimizações JIT do Node. Python, uma linguagem conhecida por sua simplicidade e versatilidade, manteve velocidades relativamente consistentes em diferentes condições de teste.

As implicações financeiras dessas diferenças de desempenho ficaram evidentes ao avaliar os custos da AWS. A eficiência do Rust traduziu-se em uma economia substancial de custos. Essas disparidades de custo ressaltam os benefícios econômicos de escolher uma linguagem de alto desempenho, especialmente para implantações em nuvem em larga escala.

Concluindo, embora cada linguagem de programação tenha seus pontos fortes e aplicativos exclusivos, é crucial considerar o desempenho e as métricas financeiras ao implantar soluções, especialmente em ambientes de nuvem como o AWS. Esta pesquisa destaca a importância de tomar decisões informadas no desenvolvimento de software, garantindo que o desempenho e a relação custo-benefício estejam alinhados com as metas e restrições do projeto.

5.1 TRABALHOS FUTUROS

No decorrer desta pesquisa, foi aprofundado nas complexidades de desempenho de várias linguagens de programação e seus respectivos impactos nos custos de infraestrutura em nuvem. No entanto, o domínio da otimização e execução da linguagem de programação é vasto e existem vários caminhos que ainda precisam ser explorados.

Uma direção promissora para futuras investigações é o exame do código de máquina gerado para cada linguagem de programação. Ao analisar o código da máquina, podemos obter informações sobre as operações e otimizações de baixo nível aplicadas durante o processo de compilação. Isso forneceria uma compreensão mais granular das características de desempenho observadas em nossos testes.

Além disso, uma exploração mais profunda das otimizações do compilador em diferentes linguagens seria inestimável. Os compiladores desempenham um papel crucial na tradução do código de alto nível em instruções eficientes no nível da máquina. Compreender as nuances dessas otimizações pode esclarecer por que certos idiomas funcionam melhor em cenários específicos.

Um estudo mais aprofundado do compilador Just-In-Time (JIT) do mecanismo V8 também está no horizonte. O V8 mostrou características de desempenho intrigantes em nossos testes. Aprofundar-se em seu processo de compilação JIT, entender suas estratégias de otimização e identificar possíveis gargalos ou áreas de melhoria pode abrir caminho para aplicativos Node.js com melhor desempenho no futuro.

REFERÊNCIAS BIBLIOGRÁFICAS

1 FOUNDATION, R. *What is ownership?* 2021. <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html>. Accessed: 2023-07-12.

2 NEWMAN, J. A review of:“the psychology of computer proqramming. by geraldm. weinbeug”.(new york: Van nostrand reinhold, 1971.)[pp. xv+ 288.]£ 4· 75. *ERGONOMICS*, Taylor & Francis, v. 15, n. 6, p. 734–735, 1972.

3 PEDROSA, P. H.; NOGUEIRA, T. Computação em nuvem. *Acesso em*, v. 6, 2011.

4 VECCHIOLA, C.; CHU, X.; BUYYA, R. et al. Aneka: a software platform for .net-based cloud computing. *High speed and large scale scientific computing*, v. 18, n. 3, p. 267–295, 2009.

5 BUYYA, R.; RANJAN, R.; CALHEIROS, R. N. Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In: IEEE. *2009 international conference on high performance computing & simulation*. [S.l.], 2009. p. 1–11.

6 SILVA, C. P.; FEYH, P. G. R.; ROLAND, C. E. de F. Pets: Desenvolvimento de sistema de geolocalização para o monitoramento de animais de estimação. *Revista Eletrônica de Sistemas de Informação e Gestão Tecnológica*, v. 9, n. 3, 2018.

7 NEVES, M.; PEREZ, N. B.; SISTI, R. N. Análise exploratória de dados de monitoramento dos animais em um sistema de integração lavoura-pecuária. In: IN: CONGRESSO BRASILEIRO DE AGRICULTURA DE PRECISÃO, 2014, SÃO PEDRO, SP .... [S.l.], 2014.

8 MELL, P.; GRANCE, T. *The NIST Definition of Cloud Computing*. [S.l.], 2011. Disponível em: <https://csrc.nist.gov/publications/detail/sp/800-145/final>.

9 BLACKBURN, S. M.; GARNER, R.; HOFFMANN, C.; KHAN, A. M.; MCKINLEY, K. S.; BENTZUR, R.; DIWAN, A.; FEINBERG, D.; FRAMPTON, D.; GUYER, S. Z. et al. The dacapo benchmarks: java benchmarking development and analysis. In: ACM. *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. [S.l.], 2006. p. 169–190.

10 CRARY, K.; WALKER, D.; MORRISETT, J. G. Typed memory management in a calculus of capabilities. In: ACM. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. [S.l.], 1999. p. 262–275.

11 BOTCHARNIKO, D. Approaches to optimizing v8 javascript engine. *ISP RAS Proceedings*. Disponível em: <http://ispras.ru/proceedings/docs/2015/27/6/isp_27_2015_6_21.pdf>.

12 ORENDORFF, J.; BLANDY, J. *Programming Rust: Fast, Safe Systems Development*. 2. ed. [S.l.: s.n.], 2018. ISBN 9781492052548.

13 ROSSUM, G. V. Python tutorial. technical report cs-r9526, centrum voor wiskunde en informatica. 1996.

14 PETERS, T. *The Zen of Python*. 2004. <https://peps.python.org/pep-0020/>.

15 FOUNDATION, P. S. *Python 3.9.1 documentation*. 2021. <https://docs.python.org/3/>.

16 DAHL, R. A javascript runtime built on chrome’s v8 javascript engine. 2009.

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

Anexo 1 - Tabelas completas

Tabela 5.1: Tempo de resposta em *Cold Start* (Crivo de Erastótenes) em ms

Rust	Node	Python
20.07	178.49	109.84
19.95	174.75	110.60
20.09	175.32	118.56
19.76	177.82	109.37
20.53	176.34	120.80
20.35	176.25	110.89
20.16	174.60	119.65
19.77	174.29	123.18
20.00	174.10	115.92
20.21	174.85	117.92
20.51	176.59	114.36
19.96	171.12	116.40
20.25	180.49	113.12
20.35	181.40	119.71
20.11	175.67	108.80
20.19	177.60	116.96
20.54	175.24	122.77
20.38	175.44	119.38
20.27	175.94	117.10
20.06	178.92	105.94
20.08	176.82	113.49
20.24	175.36	107.16
19.76	176.64	114.38
20.26	177.43	120.35
20.74	174.48	113.39
20.24	175.11	110.01
20.52	176.83	110.82
20.38	176.28	122.35
20.18	176.14	112.64
19.65	175.81	113.49
20.19	174.94	119.15
20.26	179.97	116.01
20.28	175.17	102.04
20.35	174.65	115.90
20.10	174.98	110.22
19.42	178.35	124.51
20.79	174.12	114.26
20.04	176.66	120.57
20.72	174.12	108.73
19.99	177.06	119.70
19.99	177.28	112.44
19.92	173.99	112.73

Tabela 5.2: Tempo de resposta em *Cold Start* (NoSQL) em ms

Rust (ms)	Node (ms)	Python (ms)
37,73	538,46	366,32
37,47	539,24	341,89
38,69	554,42	349,6
37,37	538,95	359,65
38,23	532,57	361,51
37,25	559,58	347,11
37,63	546,96	334,46
37,91	530,43	360,53
38	537,38	332,98
37,28	545,72	364,86
38,25	528,67	351,32
37,99	554,46	351,72
38,73	547,4	351,03
37,22	546,74	355,8
37,39	535,56	327,08
36,48	557,59	360,12
37,84	545,94	354,3
38,49	549,66	345,67
37,39	543,7	365,28
36,83	541,15	337,59
38,27	533,45	349,14
37,33	547	347,58
38,85	547,11	365,36
38,11	548,51	366,78
37,83	548,72	347,43
37,68	540,18	357,69
37,41	536	346,77
39,53	540,81	363,06
37,27	538,59	360,77
38,11	545,78	355,49
37,27	535,36	362,35
37,46	543,33	345,56
37,66	528,02	340,11
37,64	541,32	344,52
39,09	552,79	360,59
38,3	543,14	353,39
38,92	536,61	353,69
37,38	535,38	358,68
37,43	575,21	353,74
37,41	531,64	363,62
38,32	560,51	359,35
37,74	538,71	358,94
38,26	532,03	353,23
38,45	552,88	350,22
37,54	548,98	353,57
37,56	544,34	353,06