CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS TIMÓTEO

Abner Souza Kaizer

TESTES DE COMPILADORES NA NUVEM PÚBLICA: UMA AVALIAÇÃO DE VIABILIDADE USANDO PYTHON NA PLATAFORMA AWS LAMBDA

Timóteo

2024

Abner Souza Kaizer

TESTES DE COMPILADORES NA NUVEM PÚBLICA: UMA AVALIAÇÃO DE VIABILIDADE USANDO PYTHON NA PLATAFORMA AWS LAMBDA

Monografia apresentada à Coordenação de Engenharia de Computação do Campus Timóteo do Centro Federal de Educação Tecnológica de Minas Gerais para obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Leonardo Lacerda Alves

Timóteo

2024

Abner Souza Kaizer

TESTES DE COMPILADORES NA NUVEM PÚBLICA: UMA AVALIAÇÃO DE VIABILIDADE USANDO PYTHON NA PLATAFORMA AWS LAMBDA

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais, campus Timóteo, como requisito parcial para obtenção do título de Engenheiro de Computação.

Trabalho aprovado. Timóteo, 14 de Fevereiro de 2025.

Leonardo Lacerda Alves
Orientador

Aléssio Miranda Júnior
Professor Convidado

Lucas Pantuza Amorim
Professor Convidado

Timóteo
2025

FOLHA DE APROVAÇÃO DE TCC Nº 1/2025 - DECOMTM (11.63.11)

(Nº do Protocolo: NÃO PROTOCOLADO)

(Assinado digitalmente em 19/02/2025 11:28)
ALESSIO MIRANDA JUNIOR
PROFESSOR ENS BASICO TECN TECNOLOGICO
DECOMTM (11.63.11)
Matrícula: ###134#0

(Assinado digitalmente em 20/02/2025 11:33)
LEONARDO LACERDA ALVES
PROFESSOR ENS BASICO TECN TECNOLOGICO
DECOMTM (11.63.11)
Matrícula: ###653#3

(Assinado digitalmente em 18/02/2025 23:50)
LUCAS PANTUZA AMORIM
PROFESSOR ENS BASICO TECN TECNOLOGICO
DECOMTM (11.63.11)
Matrícula: ###974#1

Visualize o documento original em https://sig.cefetmg.br/documentos/ informando seu número: ***1****, ano:* ***2025****, tipo:* ***FOLHA DE APROVAÇÃO DE TCC****, data de emissão:* ***17/02/2025*** *e o código de verificação:* ***8839186494***

.

Dedico aos
meus pais.

Agradecimentos

Agradeço primeiramente a Deus e aos meus pais, Abmael e Lina, por terem me criado e sustentado até aqui.

Agradeço ao meu orientador, Leonardo Lacerda Alves, por aceitar me auxiliar neste trabalho.

“Todo o que ama a disciplina
ama o conhecimento,
mas aquele que odeia a repreensão é tolo. ”.
Provérbios 12:1

Resumo

O cenário escolhido envolve a crescente adoção da computação em nuvem e a transição para arquiteturas *serverless*, que oferecem escalabilidade e eficiência, mas também impõem desafios na avaliação de compiladores. Diante desse contexto, o problema de pesquisa proposto consiste em verificar a viabilidade de avaliar compiladores em ambientes de computação em nuvem pública. Para responder a essa questão, o estudo visa testar e avaliar os compiladores Python (CPython e PyPy JIT) em aplicações *serverless* utilizando o AWS Lambda. Os objetivos específicos incluem analisar as características de uma arquitetura *serverless*, propor uma lista de programas com as características típicas desse ambiente e verificar a viabilidade de avaliar os compiladores escolhidos. Os resultados demonstraram que a comparação entre CPython e PyPy JIT na AWS Lambda é inviável, uma vez que o CPython possui suporte nativo e tratamento especial na plataforma, enquanto o PyPy JIT enfrenta restrições operacionais, impossibilitando uma avaliação justa entre os dois compiladores.

Palavras-chave: *serverless*, compiladores, *benchmarking*, desempenho, CPython, PyPy JIT, Python, AWS Lambda.

Abstract

The chosen scenario involves the increasing adoption of cloud computing and the transition to serverless architectures, which offer scalability and efficiency but also pose challenges for compiler evaluation. In this context, the proposed research problem is to verify the feasibility of evaluating compilers in public cloud computing environments. To address this question, the study aims to test and evaluate Python compilers (CPython and PyPy JIT) in serverless applications using AWS Lambda. The specific objectives include analyzing the characteristics of a serverless architecture, proposing a list of programs with the typical features of this environment, and verifying the feasibility of evaluating the selected compilers. The results demonstrated that comparing CPython and PyPy JIT on AWS Lambda is unfeasible, as CPython has native support and special handling on the platform, while PyPy JIT faces operational restrictions, preventing a fair evaluation between the two compilers.

Keywords: serverless, compilers, benchmarking, performance, CPython, PyPy JIT, Python, AWS Lambda.

Lista de ilustrações

Lista de abreviaturas e siglas

AWS	Amazon Web Services
IAM	*Identity and Access Management*
IEEE	Institute of Electrical and Eletronics Engineers
ACM	Association for Computing Machinery
GCC	GNU Compiler Collection
LLVM	Low-Level Virtual Machine
OpenJDK	Open Java Development Kit
WSGI	*Web Server Gateway Interface*
JIT	*Just-In-Time*
SDK	*Software Development Kit*
URL	*Uniform Resource Locator*
API	*Application Programming Interface*

Sumário

1 Introdução

“A resposta certa não importa nada:
o essencial é que as perguntas estejam certas”.
Mário Quintana

Buscando, primeiramente, a eficiência econômica e, posteriormente, o desenvolvimento de aplicações web, cada vez mais empresas e indivíduos procuram soluções de computação em nuvem. Dessa forma percebem que esse modelo de desenvolvimento é mais ágil (UEDA; NAKAIKE; OHARA, 2016).

Assim, a recente mudança das arquiteturas de sistemas empresariais locais e monolíticos para uma baseada em contêineres e microsserviços teve como objetivo alcançar maior eficiência. Mesmo que ao custo da perda de parte do controle sobre seções de um sistema computacional, delegando essas seções a terceiros. Para obter maior eficiência e controle, a arquitetura de microsserviços *serverless* (sem servidor) emergiu como um novo paradigma para a implantação de aplicações na nuvem (STRATEGIES, 2016).

Segundo a *Amazon Web Services* (AWS):

> Uma arquitetura *serverless* é uma maneira de criar e executar aplicações e serviços sem precisar gerenciar nenhuma infraestrutura. Sua aplicação continua sendo executada em servidores, mas esses servidores são totalmente gerenciados por um provedor na nuvem (AWS, 2023, 1º parágrafo).

Naturalmente, uma arquitetura *serverless* também depende de compiladores. Pode-se dizer que todo programa executado em um computador, desde sistemas operacionais até pequenos *scripts* criados por usuários finais, foi traduzido por um compilador(CHEN et al., 2020).

1.1 Justificativa

Esta pesquisa justificou-se pela necessidade de avaliar o desempenho dos compiladores Python no contexto da arquitetura de microsserviços *serverless*. Python é atualmente a linguagem mais popular, segundo o índice TIOBE de janeiro de 2025 (TIOBE, 2025).

Toda avaliação de desempenho de sistemas computacionais é denominada *benchmarking*. Segundo Crapé e Eeckhout (2020): "Benchmarking e análise de desempenho estão na base da arquitetura de computadores, bem como na pesquisa e desenvolvimento de sistemas computacionais."

Entretanto, nem todo benchmarking é igual. Segundo Li et al. (2022), os testes de desempenho de aplicações *serverless* ainda são escassos. No entanto, existem diversas iniciativas voltadas para a medição e análise de desempenho de aplicações *serverless*, como

as de Brenner e Kapitza (2019), Carreira et al. (2021), Zhang et al. (2021) e Seth e Chintale (2024).

1.2 Problema

A seguinte questão constitui o problema desta pesquisa: É viável avaliar compiladores em ambiente de computação em nuvem pública?

1.3 Objetivos

Para responder ao problema proposto, o objetivo geral deste trabalho é testar e avaliar compiladores Python em aplicações *serverless* no ambiente de computação em nuvem pública AWS Lambda. Também, objetivam-se mais especificamente:

1. Descrever as características típicas de uma arquitetura *serverless*;
2. Propor uma lista de programas com as características típicas *serverless* ;
3. Verificar a viabilidade de avaliar os desempenhos dos dois compiladores escolhidos (CPython e PyPy JIT) usando a lista proposta.

1.4 Estrutura da monografia

Introdução: capítulo atual.

Procedimentos metodológicos: consiste em explicar o que e como foi feito cada parte do trabalho de pesquisa e desenvolvimento.

Fundamentos históricos, teóricos e metodológicos: consiste em uma revisão da literatura sobre compiladores e aplicações *serverless*.

Análise do desempenho da compilação de aplicações *serverless*: consiste nos testes dos exemplos escolhidos e verificação se são suficientes para a avaliação de desempenho.

Conclusão: consiste na caracterização dos resultados, limitações e possíveis trabalhos futuros.

2 Procedimentos metodológicos

“Um bom começo é a metade”.
Aristóteles

A presente pesquisa é descritiva e exploratória do ponto de vista dos objetivos; aplicada do ponto de vista de sua natureza; qualitativa e quantitativa quanto à abordagem ao problema; e pode ser classificada como pesquisa experimental na perspectiva dos procedimentos técnicos, embora tenha mobilizado diferentes procedimentos técnicos em diferentes partes, que merecem classificação diferenciada (RODRIGUES et al., 2007).

Os procedimentos metodológicos são organizados nas seguintes etapas:

1. Reunir e estudar trabalhos referentes à caracterização de cargas de trabalho *serverless*, para reconhecer as características das aplicações *serverless* que já foram elucidadas por outros autores, de diversas áreas do conhecimento. Ao mesmo tempo, reunir e estudar trabalhos referentes a testes de desempenho de compiladores, em especial da linguagem Python;
2. Propor uma lista de programas de teste *serverless* que possam ser usadas para testar os compiladores escolhidos, se baseando no estudo dos trabalhos reunidos no item anterior;
3. Averiguar a viabilidade da avaliação do desempenho, por meio da lista de programas, dos dois compiladores escolhidos na etapa 2, utilizando um procedimento metodológico de teste que também será escolhido por meio da revisão bibliográfica.

As diferentes etapas são descritas mais detalhadamente nas próximas seções do presente capítulo.

2.1 Revisão da literatura

A revisão de literatura empregada neste trabalho resulta em quatro conjuntos distintos de trabalhos relacionados: i) a revisão do estado da arte em análise de aplicações *serverless*; ii) o levantamento bibliográfico dos principais resultados de trabalhos anteriores que explicitaram características do domínio *serverless*, com origem em diversas áreas do conhecimento; iii) a revisão do estado da arte em análise de desempenho de compiladores; e iv) o levantamento bibliográfico dos principais resultados de trabalhos anteriores que explicitaram características do domínio compiladores, com origem em diversas áreas do conhecimento.

O primeiro produto da revisão é o estado da arte e da técnica em análise de aplicações *serverless* é apresentado no capítulo 3. O contexto *serverless* foi adotado para limitar o escopo deste trabalho e torná-lo viável para estudos no tempo dado para conclusão deste

trabalho. Com isso, trabalhos relacionados mais especificamente às aplicações *serverless* são de especial interesse.

O segundo produto da revisão engloba um conjunto das principais características e métodos de avaliação de aplicações *serverless*, bem como as métricas usadas para avaliação experimental do desempenho dos métodos.

O terceiro produto da revisão é um apanhado geral do cenário dos compiladores atualmente, suas principais características com o foco na análise do desempenho destes.

O quarto produto da revisão se refere às técnicas e métodos de avaliação de desempenho de compiladores, bem como as métricas usadas para avaliação experimental do desempenho dos métodos.

Neste trabalho, foi utilizado a ferramenta de busca Google Acadêmico (GOOGLE. . . , 2023). Foram feitas buscas entre os dias 29 de junho de 2023 e 20 de setembro de 2023 e posteriormente em 3 de novembro de 2024. Se utilizou das palavras-chave: "cargas de trabalho benchmark", "workload benchmark programming languages", "serverless application benchmark", "compiler benchmarking", "python benchmarking", "compiler server benchmarking", "python compiler benchmarking"e "serverless applications benchmark". Dentre os principais publicadores encontrados se destacam o *Institute of Electrical and Electronics Engineers* (IEEE) e a *Association for Computing Machinery* (ACM).

2.2 Componentes e plataformas usados

Para este trabalho foi escolhida a plataforma *serverless* AWS Lambda, sendo ela a primeira plataforma deste tipo e a AWS é a plataforma de nuvem mais usada segundo a pesquisa do Stack Overflow(OVERFLOW, 2024). Outro fator foi a dificuldade de encontrar como usar o PyPy JIT nas outras plataformas mais usadas.

A AWS possui um número gratuito de um milhão de requisições por mês no AWS Lambda, sendo esta uma das limitações dos testes há serem executados neste trabalho. Outra limitação é o tempo de 6 meses dado para a conclusão do trabalho.

As aplicações de teste usadas são de código aberto e estão disponíveis em Serverless. Este site possui diversos exemplos de aplicações *serverless* para diversas linguagens, além de Python. Sendo um *framework* para construção de aplicações *serverless* usando AWS Lambda. Permite o envio e monitoramento do desempenho e de erros das funções em execução.

Além dos exemplos fornecidos pelos desenvolvedores do Serverless Framework, também foi utilizado o exemplo criado e analisado inicialmente por Scheller (2018) em seu artigo não acadêmico.

Os programas de teste que serão utilizados são os seguintes:

1. HTTP API
2. REST API

3. REST API com DynamoDB.
4. Flask API
5. Flask API com DynamoDB
6. Scheduled Cron
7. PynamoDB S3 SigURL
8. Python
9. HTTP API com PynamoDB
10. SQS Produtor-Consumidor
11. Endpoint;
12. Números primos

O baixo número de programas de teste reflete duas questões principais: a primeira é a falta de manutenção dos exemplos disponíveis nos repositórios e nas documentações — não apenas do framework escolhido, mas também de outros, como o AWS Chalice e recursos disponíveis em sites como o Serverless Land; a segunda é o fato de se tratar de uma área ainda pouco explorada.

2.3 Avaliação de desempenho

No trabalho, serão empregados os procedimentos metodológicos de teste descritos por Crapé e Eeckhout (2020), desde que se constate ser possível avaliar o desempenho dos dois interpretadores. Com base nos programas de teste e na ferramenta escolhida, o trabalho será estruturado da seguinte forma:

1. Testar a ferramenta e configurar a plataforma para os testes, será tratado em um capítulo posterior;
2. Realizar quaisquer adaptações para medir o desempenho necessário para o estudo e executar os testes, será tratado em um capítulo posterior;
3. Elucidar os resultados obtidos e realizar o estudo destes com base nos procedimentos metodológicos escolhidos, será tratado em um capítulo posterior.

Testar a ferramenta consiste em aprender sobre sua utilização, descrevendo todo o processo. Configurar a plataforma AWS Lambda consiste em documentar o processo de criação e uso da plataforma. Está etapa serve de base para as etapas posteriores, servindo para documentar os sistemas em sua forma mais natural.

Realizar as alterações e testes consiste em adequar os componentes e procedimento metodológico de testes ao cenário *serverless.*A etapa anterior trata-se das ferramentas em suas formas brutas, o que pode não ser útil para os testes no cenário serverless.

Elucidar os resultados consiste em descrevê-los e ponderar sobre eles a luz dos conhecimentos relacionados a testes de compiladores e aplicações *serverless*. Não se trata meramente de analisar o desempenho do compilador, mas da análise sobre o desempenho no cenário.

Os procedimentos metodológicos de teste usados neste trabalho consistem na separação de dois conceitos descritos por Crapé e Eeckhout (2020): estado de inicialização; e Estado de prontidão.

Por um lado, para medir o desempenho do estado de inicialização se fez as seguintes etapas (CRAPÉ; EECKHOUT, 2020, p.4, tradução nossa):

> Para cada programa de teste serão feitas múltiplas invocações dos compiladores usados. Onde cada invocação roda uma única vez cada programa de teste, medindo o tempo de execução;
> Usando 95% de grau de confiança calcula-se dinamicamente o intervalo de confiança das últimas i invocações, sendo i um número arbitrário pré-definido. Se o intervalo de confiança for menor que 5% ou se forem realizadas 30 invocações as medições são paradas. Será usado a tabela t para o cálculo dos intervalos de confiança.

Por outro lado, para medir o desempenho do estado de prontidão se fez as seguintes etapas (CRAPÉ; EECKHOUT, 2020, p.5, tradução nossa):

> Serão feitas no mínimo 30 execuções usando uma mesma invocação de compilador;
> É determinado a iteração s em que o estado de prontidão é atingido. Quando o coeficiente de variação é calculado, das últimas 4 iterações, e ele é menor que 2% as 30 iterações são realizadas e o tempo é medido;
> É calculado a média, em estado de prontidão das iterações medidas.

3 Fundamentos históricos, teóricos e metodológicos

“O período de maior ganho em conhecimento e experiência é o período mais difícil da vida”.
Dalai Lama

O principal objetivo deste capítulo é apresentar os fundamentos históricos, teóricos e metodológicos que embasam esta pesquisa, além de mapear os principais e mais recentes trabalhos relacionados à análise de compiladores e sistemas *serverless*. As seções seguintes destacam os estudos mais relevantes e atuais sobre esses temas, com foco nas implicações diretas ou indiretas nos processos de avaliação de desempenho de aplicações *serverless*.

Este trabalho de conclusão de curso visa caracterizar o domínio *serverless*, com o intuito de facilitar a atividade de teste de desempenho de compiladores. Em outras palavras, busca-se identificar as características internas e externas desse domínio, compreendendo as implicações que ele apresenta à sua comunidade, e explorar essas características para aprimorar sistemas automáticos voltados ao teste de desempenho de compiladores.

3.1 Compiladores

Compiladores são ferramentas onipresentes no mundo moderno, essenciais para o funcionamento de praticamente todos os dispositivos computacionais. Ao utilizar um navegador, por exemplo, os sites são criados e executados com o auxílio de compiladores. Da mesma forma, sistemas operacionais, aplicativos de escritório, sistemas de comunicação e muitos outros softwares dependem de um ou mais compiladores para seu desenvolvimento e execução. Apesar de sua importância indiscutível, muitas vezes seu papel passa despercebido (CHEN et al., 2020, p.2).

Reconhecendo essa relevância, este trabalho adota a linguagem Python como objeto de estudo. Embora seja dinamicamente tipada e interpretada, características que geralmente resultam em menor desempenho em comparação com outras linguagens populares, Python continua amplamente utilizada. Para mitigar suas limitações de desempenho, iniciativas como o PyPy JIT foram criadas. Esse interpretador oferece compatibilidade semântica com o CPython, o interpretador padrão, permitindo melhorias no desempenho de execução (BARANY, 2014, p.1).

O objetivo deste trabalho é avaliar a viabilidade de se testar o desempenho de dois compiladores Python, CPython e PyPy, utilizando os mesmos programas de teste e suas configurações padrão. Serão usados programas específicos para avaliar chamadas, APIs internas e externas, além do uso de memória, com base em medições do tempo de execução.

A complexidade e a importância dos compiladores demandam grande cuidado em seu desenvolvimento. As entradas e saídas de um compilador abrangem um espectro amplo, variando em tamanho e escopo (CHEN et al., 2020, p.2). O desafio fundamental reside em traduzir uma linguagem de alto nível, como Python, para uma linguagem de baixo nível, como assembly, preservando a semântica. Contudo, a ausência de uma especificação formal para esse processo exige que os desenvolvedores dos compiladores estabeleçam a correspondência entre essas linguagens (CHEN et al., 2020, p.2).

Com uma especificação semântica clara, é possível prever o comportamento da maioria dos programas durante a execução. Assim, a partir de uma mesma especificação semântica, torna-se viável testar diferentes implementações de uma mesma linguagem (CHEN et al., 2020, p.3). Dessa forma, os esforços podem ser concentrados nos testes de desempenho, utilizando um conjunto consistente de programas de teste escritos na mesma versão da linguagem.

3.1.1 Construindo programas de teste

Ao testar softwares, é comum a criação de casos de teste com entradas variadas e de diferentes tipos. No caso de compiladores, esses casos de teste assumem a forma de programas de teste, desenvolvidos com base na mesma especificação da linguagem, de modo a garantir a validade dos testes. Programas inválidos, por sua vez, são tratados pelos próprios compiladores, sendo rejeitados durante a análise (CHEN et al., 2020, p.6).

Aumentar a diversidade sintática de um programa pode ampliar a cobertura de teste, porém também eleva a probabilidade de criar programas inválidos. Por esse motivo, opta-se por utilizar apenas um subconjunto das estruturas da linguagem ao desenvolver os programas de teste. Essa escolha, no entanto, impõe uma restrição ao escopo dos testes, o que pode influenciar os resultados obtidos. Para mitigar possíveis comportamentos indefinidos introduzidos pelos programadores dos programas de teste, é vantajoso utilizar múltiplas implementações de um mesmo compilador (CHEN et al., 2020, p.8).

É fundamental destacar que o foco está em avaliar a viabilidade de se testar o desempenho do compilador, e não a qualidade ou validade dos programas de teste.

3.1.2 Construindo manualmente os programas de teste

A criação manual de programas de teste permite que sejam moldados conforme as necessidades específicas do cenário em questão. Compiladores como GCC (2023), LLVM (2023) e OpenJDK (2023) oferecem conjuntos de testes desenvolvidos manualmente (CHEN et al., 2020, p.8).

3.1.3 Geração de programas teste

Foram desenvolvidas técnicas para gerar automaticamente programas de teste para compiladores, uma vez que a criação manual exige grande esforço. De acordo com Chen et

al. (2020, p.9, tradução nossa), a criação automática pode ser dividida em três categorias: direcionada pela gramática, guiada pela gramática e outras abordagens.

1. Direcionada pela gramática: "Tem como entrada a gramática da linguagem e gera os programas baseados nela (CHEN et al., 2020, p.9, tradução nossa)";
2. Guiada pela gramática: "Tem como entrada a gramática, porém utiliza-se de heurísticas para adicionar sensibilidade ao contexto (CHEN et al., 2020, p.11, tradução nossa)";
3. Outras abordagens: não utilizam a gramática como entrada, mas, por exemplo, empregam códigos pré-definidos como base para gerar os programas de teste (CHEN et al., 2020, p.13, tradução nossa).

3.1.4 Mutação de programas

A mutação de programas de teste existentes é utilizada para testar partes desses programas, ampliando a área de teste ao adicionar variações. Avanços significativos ocorreram na década de 2010. Segundo Chen et al. (2020), a mutação pode ser classificada em duas categorias:

1. Mutação com preservação semântica, cujo objetivo é manter o comportamento do programa (CHEN et al., 2020, p.16, tradução nossa);
2. Mutações sem preservação semântica têm como principais objetivos aumentar a diversidade e evitar comportamentos não definidos (CHEN et al., 2020, p.16, tradução nossa).

3.1.5 Oráculos

A criação de programas de teste do tipo oráculo visa validar se um dado de teste é adequado ou não para um teste de compilador. Chen et al. (2020, p.17, tradução nossa) classificou tais testes em duas categorias: “Teste diferencial” e ‘Teste metamórfico”.

Os testes diferenciais, como o nome sugere, visam testar as diferenças entre dois ou mais compiladores. Para isso, os compiladores testados devem obedecer à mesma especificação (CHEN et al., 2020, p.17,18, tradução nossa).

Há três estratégias de testes diferenciais: os que comparam as saídas de dois ou mais compiladores, os que comparam níveis de otimização de uma mesma versão de compilador e os que testam a saída de diferentes versões de um compilador (CHEN et al., 2020, p.18).

A primeira estratégia, inter-compiladores, possui três pontos a serem observados: primeiro, compiladores diferentes, com a mesma entrada, devem gerar a mesma saída; segundo, se houver erro na entrada, raramente ambos mostrarão o mesmo erro, pois são implementações distintas; e terceiro, se possuírem saídas diferentes, um deles está errado (CHEN et al., 2020).

Os testes metamórficos visam, ao alterar a entrada, verificar os impactos na saída. Um exemplo é estabelecer as relações metamórficas entre dois programas cuja semântica é a mesma (CHEN et al., 2020).

3.1.6 Otimizando o processo de teste

Como os compiladores estão no centro do desenvolvimento de sistemas de programas de computadores, encontrar erros é uma tarefa difícil. O processo de teste de compiladores é demorado. Para acelerar esse processo, foram desenvolvidas técnicas que priorizam o teste dos programas que sabemos conter erros e excluem programas de teste redundantes (CHEN et al., 2020, p.22 e 23).

3.1.7 Importância de *benchmarking* no teste de compiladores

Criar testes de desempenho para compiladores tem como motivação auxiliar no desenvolvimento de pesquisas sobre o teste de compiladores e permitir avaliar quão efetivos os novos testes de compiladores são (CHEN et al., 2020, p.30).

Os testes de desempenho servem para guiar o desenvolvimento de softwares, melhorando a percepção do usuário sobre a fluidez do programa, podendo também melhorar a eficiência energética ao realizar mais funções em menos tempo, além de orientar os desenvolvedores em suas tomadas de decisões (CRAPÉ; EECKHOUT, 2020, p.1).

Assim, podemos concluir que a área de testes de desempenho de compiladores é de suma importância para o desenvolvimento de sistemas computacionais, sendo mais do que uma mera curiosidade, pois pode gerar ganhos econômicos e temporais.

3.2 Serverless

A computação em nuvem, de modo geral, tem como foco o pagamento por uso dos recursos computacionais, ou seja, paga-se apenas pelo que se consumiu. A modalidade de computação em nuvem *serverless* ou *Function as a Service* (FaaS) vai além, pois discrimina o uso ao nível de chamadas de funções, permitindo um ajuste de custo ainda mais preciso. Isso proporciona maior escalabilidade e aumenta a integração entre a aplicação e a nuvem (VIEIRA et al., 2020, p.2).

O nome *serverless* se deve à ausência da necessidade do cliente em configurar o ambiente para executar suas funções; cabe a ele apenas escolher o ambiente e enviar ou utilizar as funções existentes que necessita usar. Assim, sua responsabilidade fica restrita aos códigos que ele desenvolveu (VIEIRA et al., 2020, p.7).

Ao distribuir poder computacional, maximiza-se o uso de *hardware*, e a separação entre os sistemas operacionais, proporcionada pelas máquinas virtuais, permitiu um maior acesso a essas tecnologias. A computação em nuvem do tipo *serverless* vai além, pois permite que os desenvolvedores criem funções em linguagens suportadas pelo provedor e as disponibilizem por meio de chamadas HTTP ou disparo de mensagens (VIEIRA et al., 2020, p.2).

A computação *serverless* abstrai toda a plataforma de nuvem abaixo dela para a aplicação que a estiver utilizando. O custo é baseado no tempo de execução, pagando-se apenas pelo que se consumiu, sem custos adicionais. Como é de uso simples, fácil configuração e

custo acessível, está aumentando sua capilaridade no desenvolvimento de sistemas em nuvem (VIEIRA et al., 2020, p.3).

Importante falar das outras modalidades de nuvem que são mais antigas:

1. *Infrastructure as a Service - IaaS (Infraestrutura como serviço)*: Consiste no aluguel de máquinas virtuais e redes privadas virtuais (VPS) (VIEIRA et al., 2020, p. 5).
2. *Platform as a Service - PaaS (Plataforma como serviço)*: É o aluguel de uma plataforma onde se abstrai para quem a utiliza a infraestrutura que está sendo usada (VIEIRA et al., 2020, p. 5).
3. *Software as a Service - SaaS (Software como Serviço)*: Uma aplicação é disponibilizada mediante uma interface que pode ser uma API (*Application Programming Interface*) ou outro tipo de interface, que pode ser usada por outra aplicação ou por um usuário final (VIEIRA et al., 2020, p. 5).

FaaS não é nada mais do que um passo além, fornecendo uma base em que se executa um código específico sem que o cliente saiba em qual plataforma ou infraestrutura tal código está sendo executado. O desenvolvedor precisa apenas garantir o tratamento de dados, além de configurar dependências de código e a política de escalonamento usada pela aplicação em sua execução (VIEIRA et al., 2020, p.5).

3.2.1 Características de uma aplicação Serverless

O desenvolvimento de aplicações *serverless* consiste em duas etapas:

1. Criação da função: os desenvolvedores criam suas funções seguindo as regras fornecidas pelo provedor de nuvem e, após a criação, as enviam utilizando uma ferramenta própria do provedor. É gerado um *Uniform Resource Locator* (URL) para que a função seja utilizada (LI et al., 2022, p.1524 e 1525).
2. Uso da função: utilizando a URL ou um evento configurado, deve-se fornecer a função por meio de uma API fornecida pelo provedor (LI et al., 2022, p.1525).

O desenvolvedor delega ao provedor do serviço a execução da função, desviando sua preocupação para o código (VIEIRA et al., 2020, p.3).

Existem alguns usos documentados de computação *serverless*, como: execução temporizada de tarefas; resposta a eventos de uma fila de tarefas; participação em tempo real; ferramentas de análise para grandes volumes de dados que precisam ser processados individualmente; uso sob demanda de serviços urbanos; uso massivo de funções, como no aprendizado de máquina; funções reativas em sistemas de segurança; e Internet das Coisas (VIEIRA et al., 2020).

Tarefas podem ser executadas quando necessário e de forma recorrente. Serviços periódicos podem ser configurados, e sua execução fica a cargo do provedor, sendo a cobrança

realizada apenas quando utilizados. Caso haja eventos a serem disparados em uma lista, eles serão executados à medida que surgem. Assim, sua escalabilidade é sob demanda. Como os recursos são solicitados apenas quando requisitados, não há um custo, computacional ou financeiro, muito elevado quando comparado a máquinas virtuais e contêineres (VIEIRA et al., 2020, p.4).

Como se trata de uma área muito nova, com cerca de 10 anos, ainda há muito a ser desenvolvido. Como mencionado por Vieira et al. (2020, p.4), ainda precisam ser desenvolvidas as seguintes áreas: "depuração orientada a custo financeiro, decomposição facilitada de sistemas legados e compartilhamento de ambientes de execução."No que diz respeito ao custo financeiro, a medição de desempenho é crucial para a melhor tomada de decisão, pois o custo por tempo de execução é uma medida relevante, sendo este o pilar deste trabalho.

Ainda há problemas a serem tratados, como, por exemplo: latência de inicialização, que afeta o tempo entre a invocação e a execução de uma função; a política de escalonamento definida pelo provedor de nuvem ou pela plataforma *serverless* local, que também é um obstáculo; infraestrutura, que é um desafio; a escolha do banco de dados que armazenará os dados e a rede a ser usada, além de como configurá-la; e tolerância a falhas, já que alguns serviços *serverless* solicitam que o usuário tente executar novamente a função em caso de falha. Uma abordagem mais sofisticada é garantir a atomicidade, consistência, isolamento e durabilidade (ACID) das execuções de funções (LI et al., 2022, p.1526 a 1531).

3.2.2 Frameworks de código aberto típicos

Existem diversos *frameworks* que implementam a arquitetura *serverless*, como, por exemplo, Apache OpenWhisk, IronFunctions, Oracle Fn, entre outros. Muitos implementam usando serviços de nuvem já estabelecidos, enquanto outros utilizam servidores ou máquinas virtuais para servir as aplicações (LI et al., 2022, p. 1531).

Há graus de dependência entre a aplicação *serverless* e a plataforma. De acordo com Li et al. (2022, p. 1531, tradução nossa), essas dependências são separadas em três categorias: "dependência fraca, semi-dependência e dependência forte".

Ainda possui os seguintes significados dados por Li et al. (2022, p. 1531):

1. dependência fraca é quando a plataforma não se utiliza ou pouco se utiliza da infraestrutura da nuvem e de seus serviços;
2. semi-dependência ocorre quando se utiliza parte dos componentes vindos da infraestrutura da nuvem;
3. dependência forte consiste na plataforma criada usar de diversos componentes da infraestrutura da nuvem e de seus serviços em sua criação.

3.2.3 Métodos de teste das aplicações Serverless

A estrutura de um sistema *serverless*, apesar de simples, possui uma implementação com muitos componentes, o que pode prejudicar o seu desempenho (LI et al., 2022, p. 1532).

Assim, uma das áreas que ainda carecem de mais atenção, apesar do crescimento da demanda por serviços *serverless*, é o teste de desempenho.

Li et al. (2022, p. 1535, tradução nossa) diz que: "Desenvolvedores usam essas métricas antes de enviar suas funções para a nuvem ou para decidir quais ferramentas são melhores para suas cargas de trabalho."Com isso, ainda há a necessidade de criar mais testes de desempenho, pois muitos casos de uso não são cobertos.

3.3 Trabalhos correlatos

Os trabalhos analisados abrangem diversas abordagens para a avaliação de desempenho de aplicações *serverless* em plataformas de computação em nuvem. Entre eles, destacam-se estudos de caso com testes de desempenho conduzidos pelos autores, artigos que apresentam metodologias para avaliação de desempenho, além de pesquisas que discutem desafios e oportunidades no campo da computação *serverless*. Ademais, algumas abordagens exploram questões relacionadas à segurança, ao impacto do ambiente de execução e ao uso de linguagens específicas nesse paradigma.

O primeiro trabalho é Maissen et al. (2020), cujo objetivo era criar uma aplicação multiplataforma para testar o desempenho de aplicações *serverless*. A ferramenta automatiza o envio, execução e limpeza dos testes, além de fornecer os dados observados e apresentar um modelo de custo para as plataformas suportadas.

O segundo trabalho é Copik et al. (2021), que foca na especificação de cargas de trabalho representativas e na sua reprodutibilidade, sendo também multiplataforma.

O terceiro é Somu et al. (2020), que automatiza os processos de envio, execução e medição de desempenho para cada plataforma suportada.

O quarto é Eismann et al. (2022), um estudo de caso sobre aplicações *serverless*. O trabalho explora os recursos do ambiente e atribui a responsabilidade pelos requisitos de desempenho aos desenvolvedores, mesmo quando o provedor de nuvem gerencia o hardware e os sistemas-base.

O quinto é Martins, Araujo e Cunha (2020), que, além de automatizar o envio, execução e a coleta de dados de desempenho, visa ajudar a identificar pontos de melhoria nas plataformas suportadas.

O sexto trabalho correlato é Crapé e Eeckhout (2020), que apresenta os procedimentos metodológicos a serem usados neste trabalho, com as adaptações necessárias. Esse estudo usa esses procedimentos para medir o desempenho em dois compiladores Python, utilizando programas de teste locais.

O sétimo trabalho é o de Li et al. (2022), que realizou uma análise sobre o estado da arte, desafios e oportunidades no setor de computação *serverless*.

O oitavo trabalho é Brenner e Kapitza (2019), que trata da segurança no processamento de dados de aplicações *serverless*. Implementaram um sistema usando JavaScript que, segundo os autores, obteve bons níveis de desempenho, mesmo com medidas de segurança

adicionais.

O nono trabalho é Carreira et al. (2021), que demonstrou que os ambientes de execução, particularmente a AWS Lambda, não utilizam as otimizações fornecidas pelas linguagens.

O décimo trabalho é Zhang et al. (2021), que realizou um estudo sobre o uso de Java no cenário *serverless*.

O décimo primeiro trabalho é Seth e Chintale (2024), que examina o desempenho de aplicações *serverless* usando AWS Lambda e a ferramenta ServerlessBench.

Os trabalhos que merecem mais destaque são os de Li et al. (2022) e Seth e Chintale (2024).

Por um lado, o trabalho de Li et al. (2022) oferece uma visão geral sobre o estado da arte e oportunidades de pesquisa no setor de computação *serverless*, mas não apresentou informações especificas sobre testes de desempenho em nuvem pública no cenário *serverless*.

Por outro lado, o trabalho de Seth e Chintale (2024) apresentou uma avaliação de desempenho usando a nuvem pública AWS Lambda em conjunto com a ferramenta ServelessBench, mas seu foco foi a ferramenta de testes em si e não uma linguagem específica.

4 Análise do desempenho da compilação de aplicações *serverless*

“Há apenas um bem, o saber;
e apenas um mal, a ignorância”.
Sócrates

As medições foram realizadas utilizando o *dashboard* da AWS Lambda e a ferramenta Serverless. O AWS Lambda fornece informações detalhadas sobre os tempos de execução e inicialização das funções, sem considerar a latência de rede. No entanto, os tempos de latência para os servidores da AWS podem ser obtidos separadamente por meio da ferramenta Amazon Workspace Health.

4.1 Preparando o ambiente e escolhendo os programas de teste

Primeiramente, é necessário configurar as credenciais na AWS para o uso do Serverless *framework*. Para isso, é preciso ativar o serviço *Identity and Access Management* (IAM) na conta AWS. Vale notar que o IAM não está disponível para servidores localizados no Brasil, sendo o servidor mais próximo o us-east-1, localizado no estado da Virgínia, Estados Unidos.

Após configurar a conta, é preciso configurar o *framework* Serverless. Assim, o tutorial disponível foi executado como descrito na documentação da AWS. O tutorial consiste em criar um runtime customizado para Bash e, em seguida, explica como publicar uma *layer* (camada) contendo apenas o runtime, permitindo sua reutilização em outras funções.

Em seguida, para criar uma *layer* com o interpretador PyPy JIT, o tutorial disponível no site foi seguido. Substituí a instalação da biblioteca pelo interpretador baixado neste link. A versão escolhida foi a de Linux x86_64 do PyPy JIT v7.3.17, compatível com o Python 3.10.

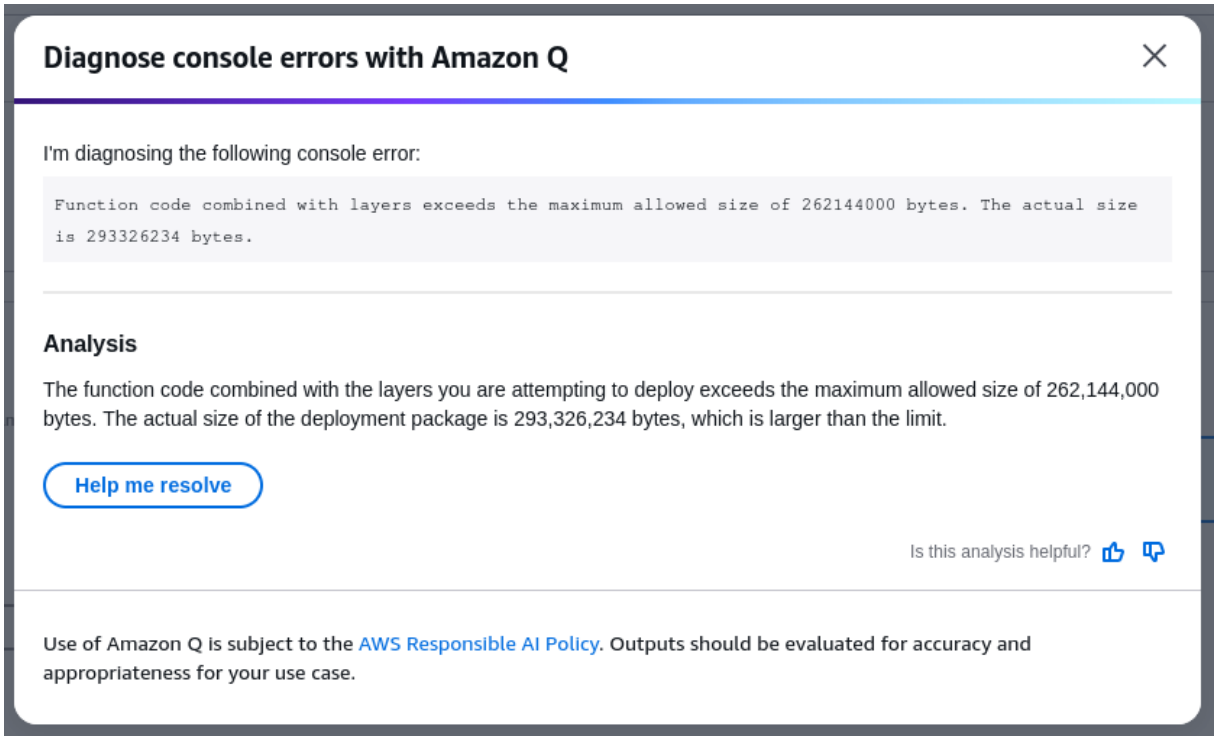

Figura 1 – Limite de tamanho das camadas do AWS Lambda

A criação de uma *layer* com o interpretador PyPy JIT não foi possível, pois há um limite de dados para inclusão em uma *layer*, como mostrado na Figura 1. Dessa forma, não é viável utilizar essa funcionalidade da AWS Lambda para melhorar o desempenho das funções que serão executadas com o PyPy JIT.

Todos os exemplos utilizados estarão disponíveis no seguinte repositório no GitHub.

4.2 Exemplos Serverless framework

Foi realizada a clonagem do repositório de exemplos do framework Serverless.

Os exemplos contidos no repositório serverless/examples que não foram testados, não o foram porque necessitavam de tokens de autorização para o uso de ferramentas fora da AWS, estavam utilizando a versão Python 2 ou eram redundantes. Assim, decidiu-se testar apenas aqueles que possuíam características diferentes e que não dependessem de ferramentas ou plataformas externas à AWS.

4.2.1 Exemplo 1 - AWS Python HTTP API

O primeiro exemplo utilizado foi o "AWS Python HTTP API", que consiste em um endpoint HTTP GET que retorna uma mensagem.

A função foi testada como está no repositório, com exceção da versão do Python, que foi alterada de 3.12 para 3.10, a última versão da linguagem compatível com o PyPy JIT. Após

essa alteração, a execução com o CPython ocorreu sem intercorrências como visto na Figura 2.

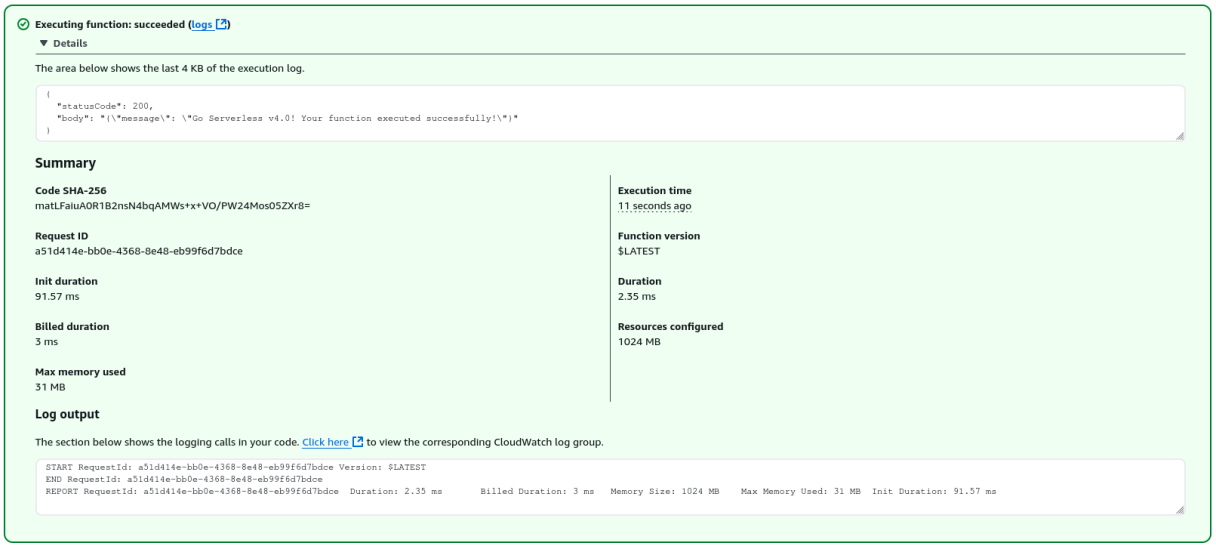

Figura 2 – Serverless Config CPython

Baseando-se no trabalho descrito por Scheller (2018), foi alterado o arquivo *runtime_interface.py*, sendo adicionada a biblioteca *requests* na raiz da pasta com a função e atualizando o PyPy JIT para a última versão. Dessa forma, o exemplo fornecido pelos desenvolvedores do framework Serverless utilizando o PyPy JIT foi executado, como mostrado na Figura 3.

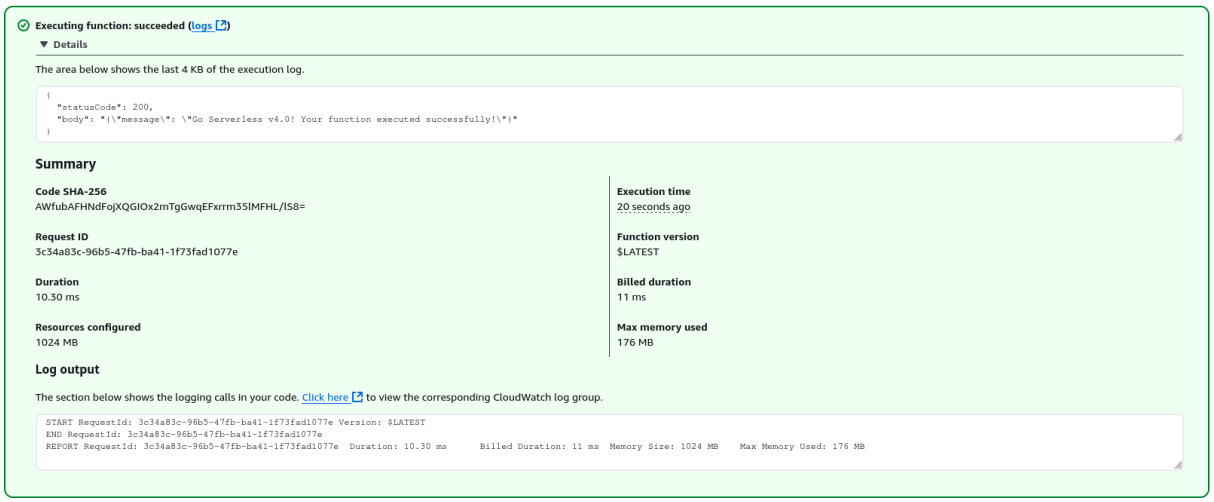

Figura 3 – PyPy JIT - Exemplo 1

A execução do PyPy foi realizada enviando o interpretador com a função, mas na execução do CPython isso não ocorre, o que impacta a avaliação de desempenho.

4.2.2 Exemplo 2 - AWS Python REST API

A execução de mais um exemplo foi realizada, o aws-python-rest-api. Foi possível a execução usando tanto o CPython quanto o PyPy JIT, ambos compatíveis com a versão 3.10 da linguagem.

A execução utilizando o CPython pode ser observada na Figura 4.

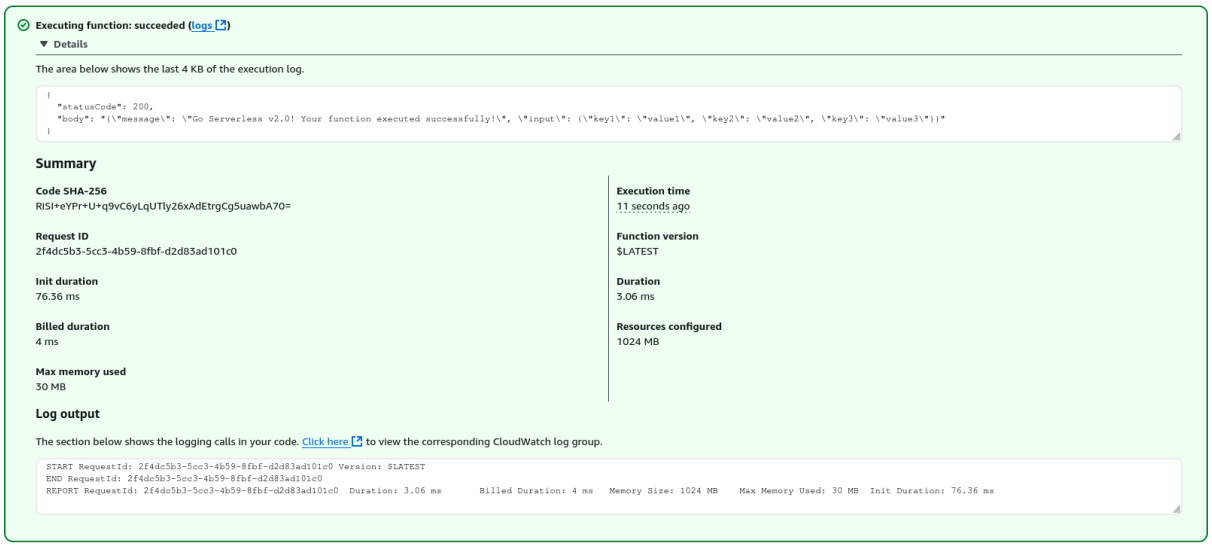

Figura 4 – CPython - Exemplo 2

Já a execução utilizando o PyPy JIT pode ser vista na Figura 5.

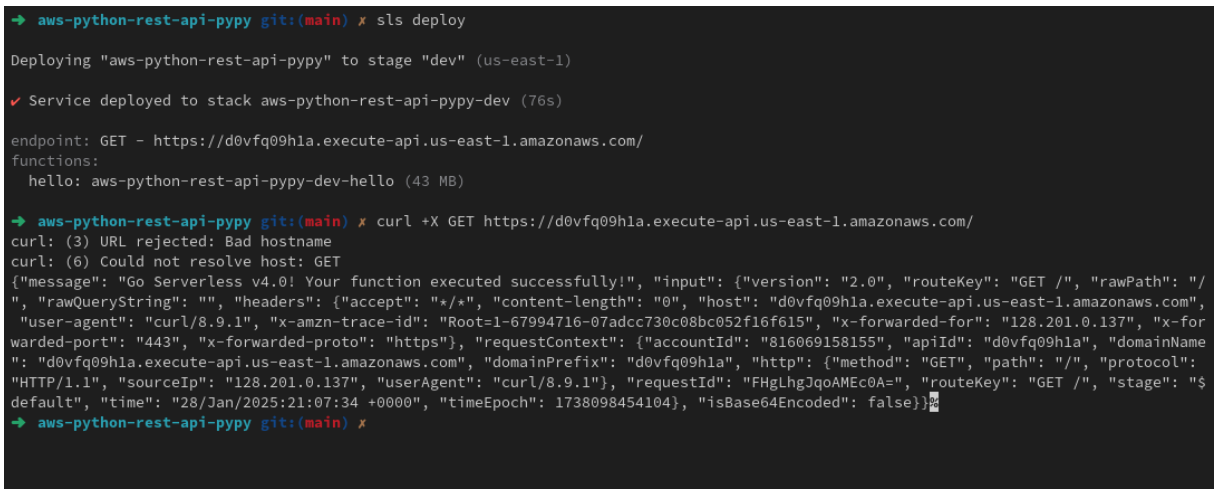

Figura 5 – PyPy JIT - Exemplo 2

4.2.3 Exemplo 3 - AWS Python REST API com DynamoDB

Foi feita uma tentativa de executar o exemplo aws-python-rest-api-with-dynamodb usando CPython, mas ocorreram os seguintes erros nas funções get, mostrados na Figura 6, e create,

mostrados na Figura 7.

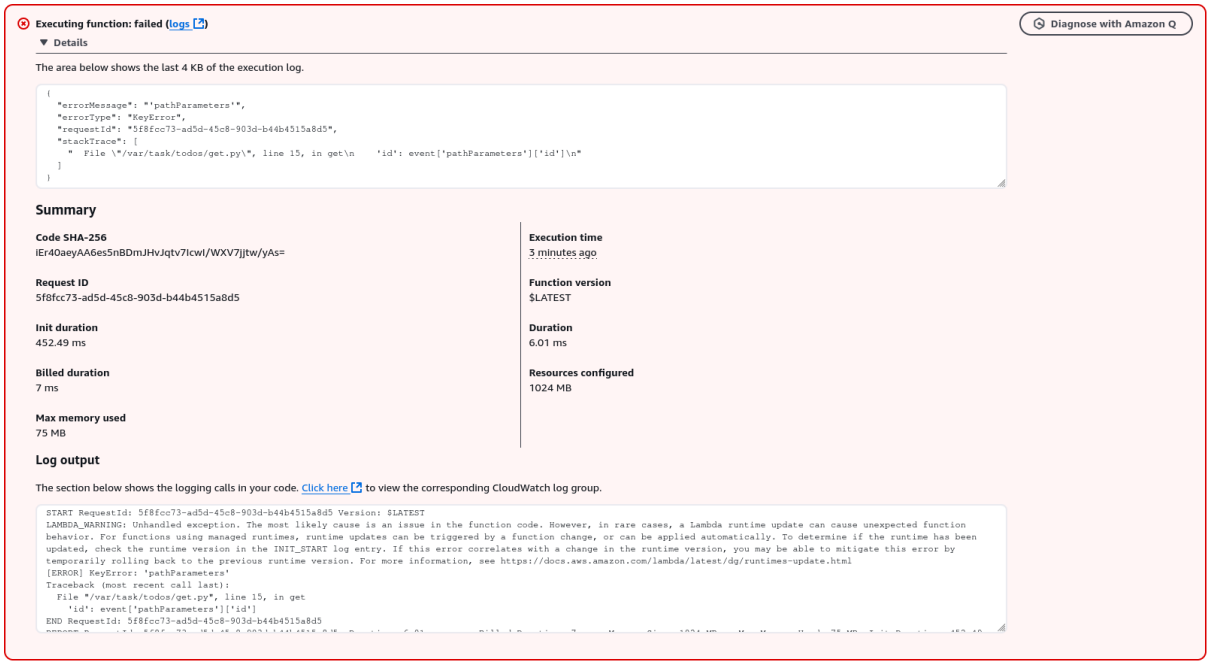

Figura 6 – CPython - Exemplo 3 método GET

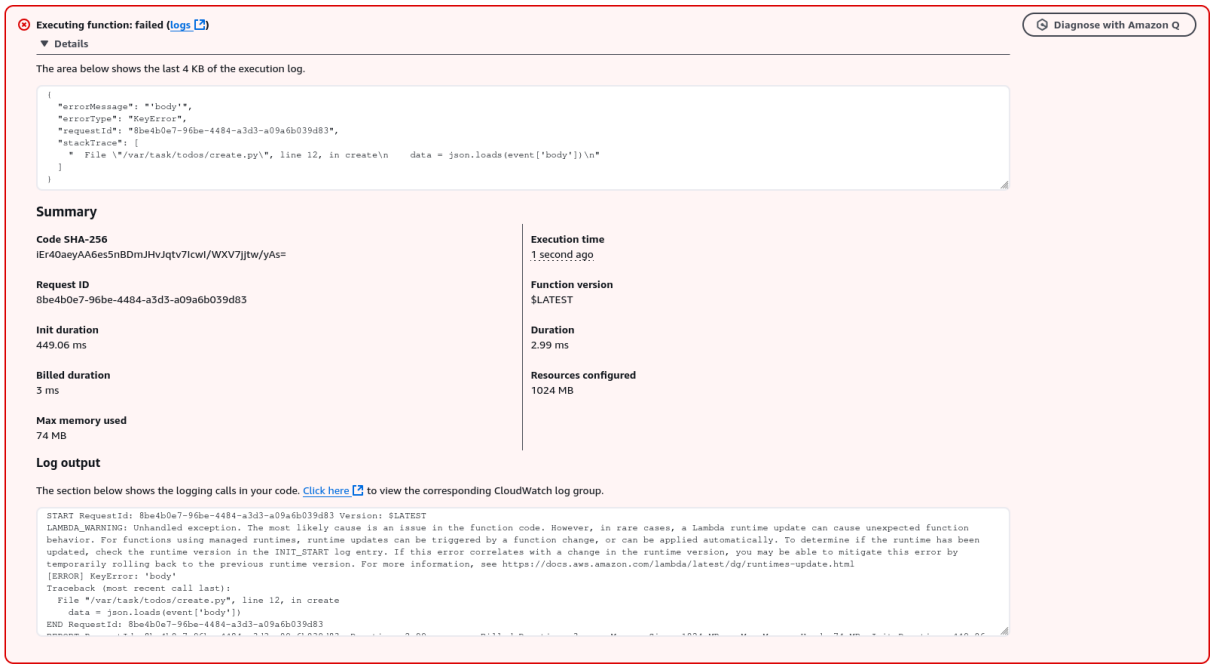

Figura 7 – CPython - Exemplo 3 método POST

Se realizou uma tentativa de executar o exemplo utilizando o método POST e o PyPy, porém não houve êxito. A função testada foi a "create". Utilizou-se o programa "curl"com a seguinte linha de comando:

```
$ curl -X POST https://xxxxxxxxx.execute-api.us-east-1.amazonaws.com/todos
--data '{"text":"Learn Serverless"}' -H "Content-Type:application/json"
```

Ao realizar a tentativa de execução usando o PyPy JIT, como mostrado na Figura 8, o resultado não foi bem-sucedido.

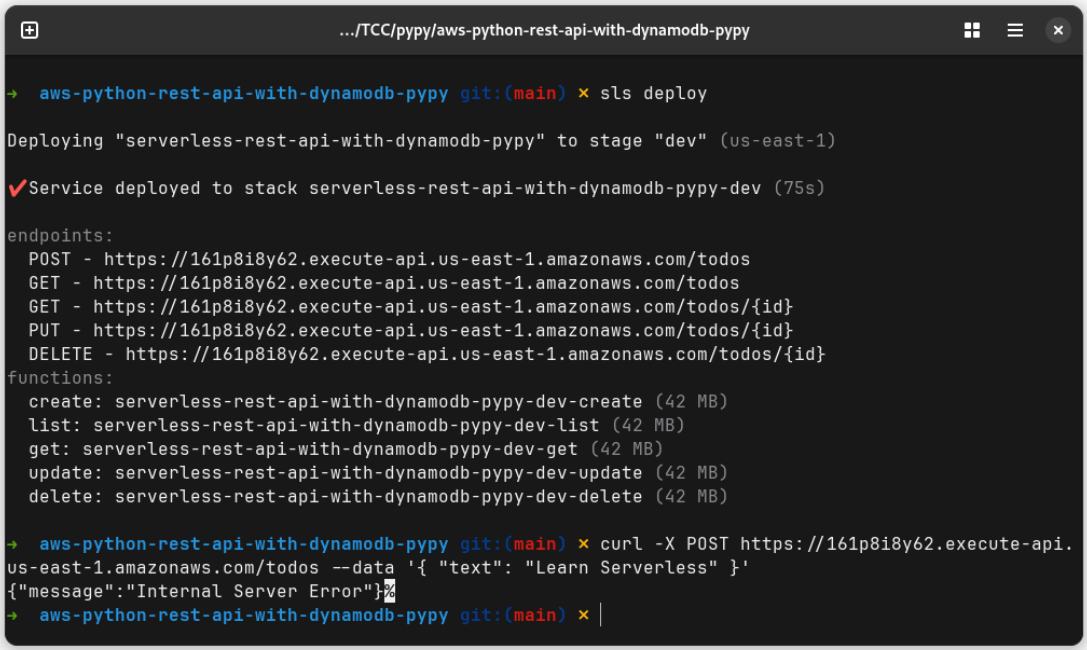

Figura 8 – PyPy JIT - Exemplo 3 método POST

4.2.4 Exemplo 4 - AWS Python Flask API

Este exemplo apresenta um comportamento inesperado, como pode ser visto nas Figuras 9 e 10. É informado que o Python não foi encontrado no *runtime* python3.10, o oficial CPython da AWS Lambda, nem no *runtime* customizado que usa o provided.al2023 da AWS Lambda como base.

A execução deste exemplo é realizada utilizando um *plugin* chamado "serverless-wsgi", que, como mostrado nas Figuras citadas, está contornando os *runtimes* usados para utilizar o seu próprio.

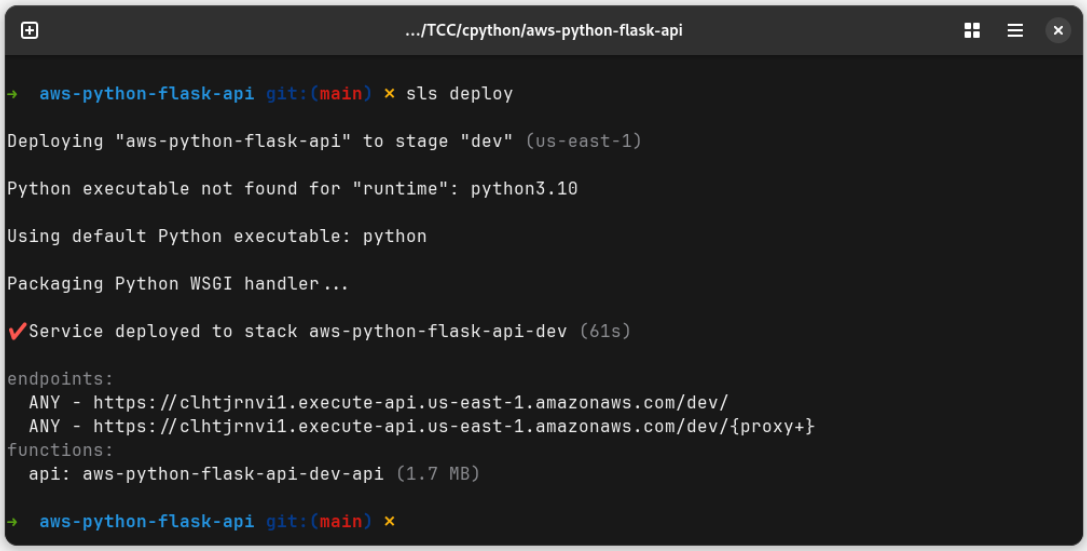

Figura 9 – CPython - Exemplo 4

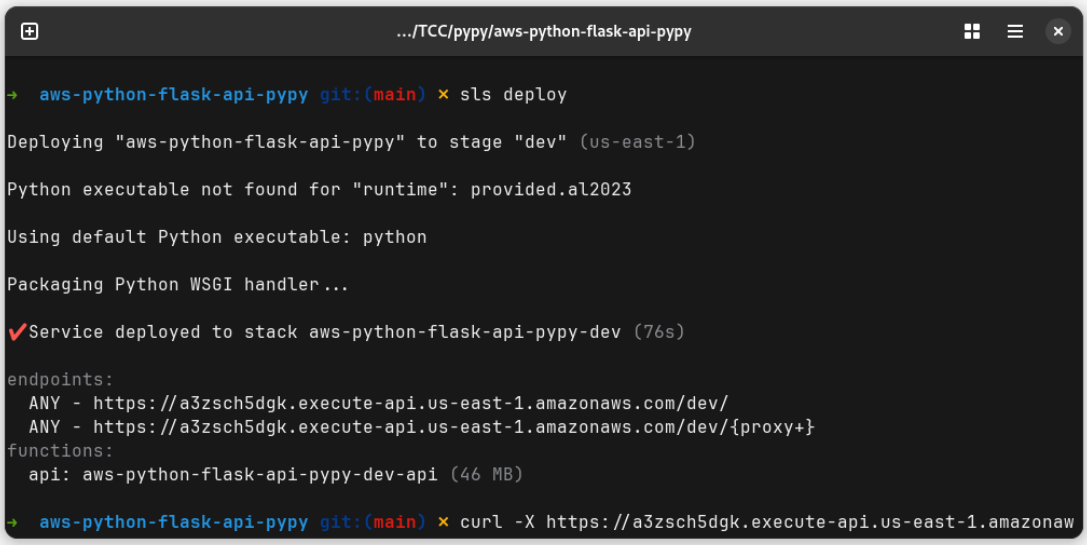

Figura 10 – PyPy JIT - Exemplo 4

4.2.5 Exemplo 5 - AWS Lambda Flask API com DynamoDB

O exemplo AWS Lambda Flask API com DynamoDB, disponível neste link, funcionou corretamente com o interpretador CPython. Foi seguida a documentação do exemplo, mas se alterou a versão do Python para a 3.10 a fim de manter a compatibilidade com o PyPy JIT.

Como mostrado na Figura 11, a mesma mensagem do exemplo 4.2.4 aparece ao enviar esta função usando CPython.

Ao realizar o envio com o interpretador PyPy, a seguinte mensagem aparece na Figura 12, a mesma mensagem exibida na Figura 10.

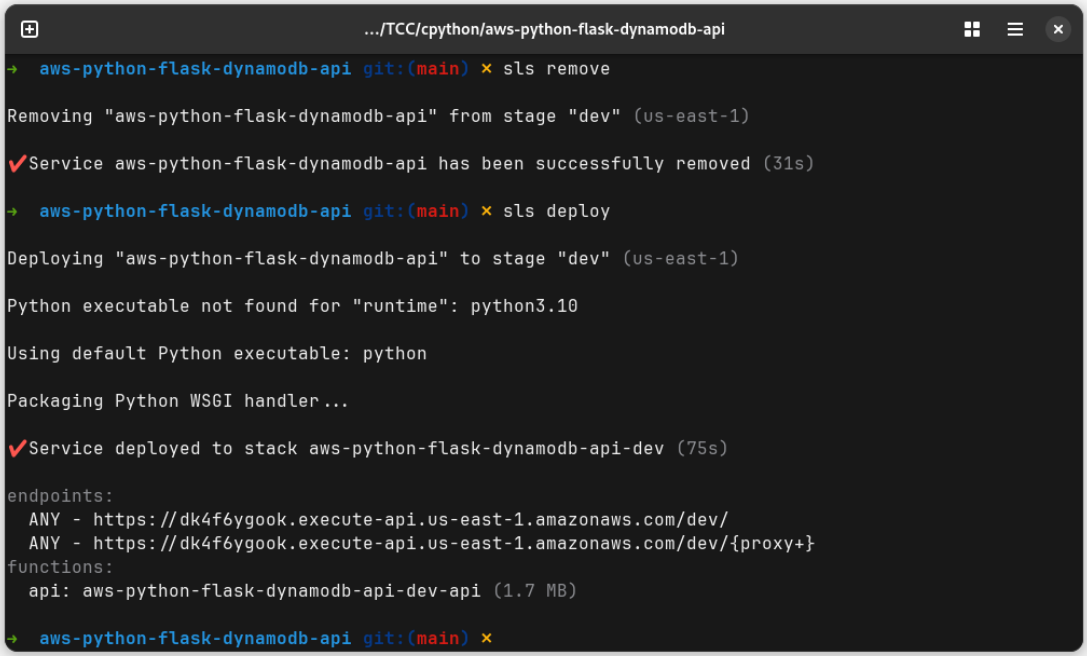

Figura 11 – CPython - Exemplo 5

Figura 12 – PyPy JIT - Exemplo 5

A utilização do plugin "serverless_wsgi"para o Serverless Framework contorna os *runtimes* que não são compatíveis com ele.

O *Web Server Gateway Interface* (WSGI) é uma camada de abstração entre servidores web e aplicações web ou *frameworks* da linguagem Python, funcionando como uma interface entre eles.

O uso do WSGI é recomendado para a implantação de aplicações desenvolvidas com

Flask, conforme indicado na documentação oficial do *framework* (FLASK, 2025).

4.2.6 Exemplo 6 - AWS Python Scheduled Cron

Como mostrado na Figura 13, o exemplo AWS Python Scheduled Cron foi executado normalmente com CPython.

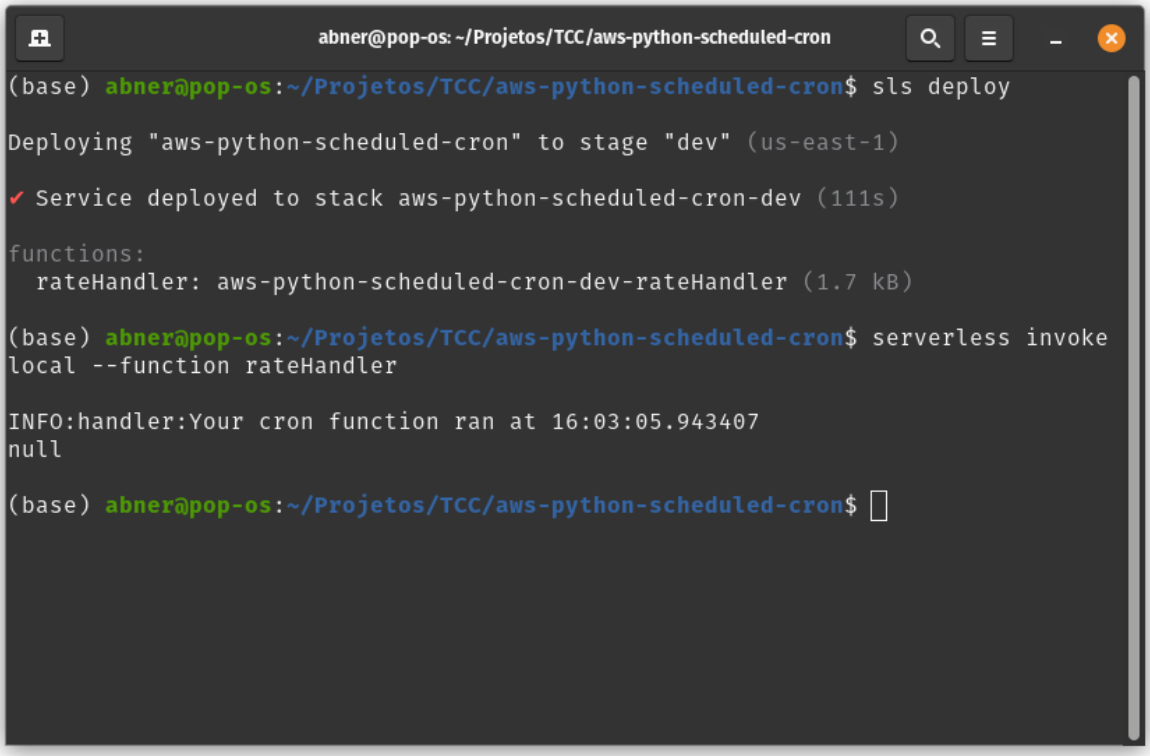

Figura 13 – CPython Cron

O mesmo exemplo, quando executado com PyPy, foi enviado, mas não foi possível executar localmente, como visto na Figura 14. Ao testar usando a ferramenta de teste do AWS Lambda, o código foi executado corretamente, como mostrado na Figura 15.

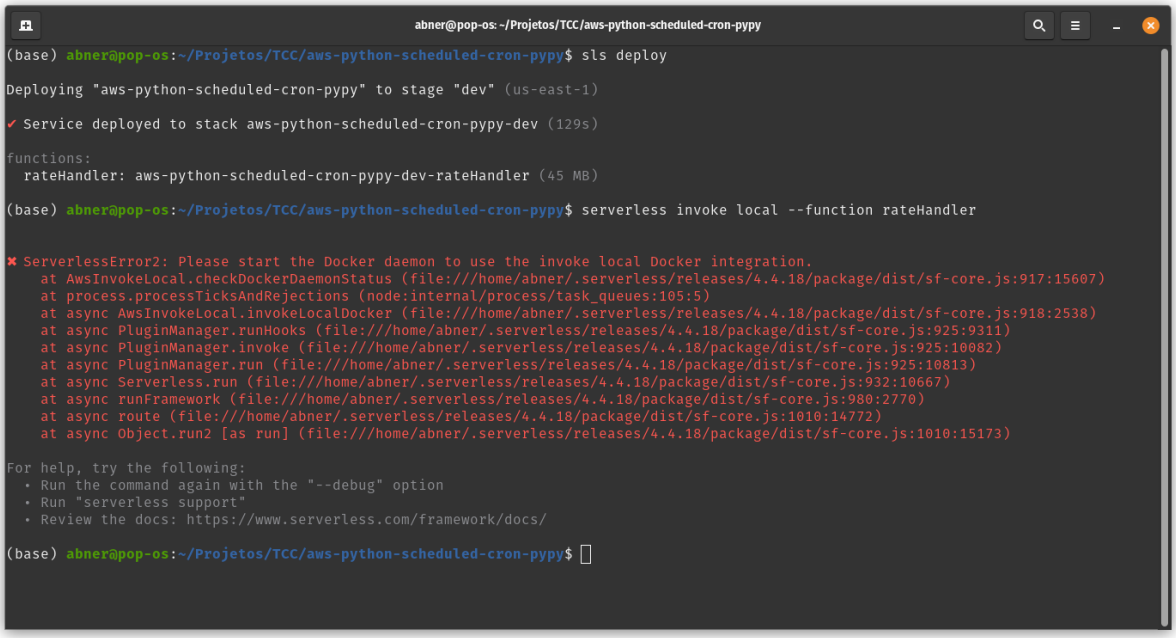

Figura 14 – PyPy Cron Local

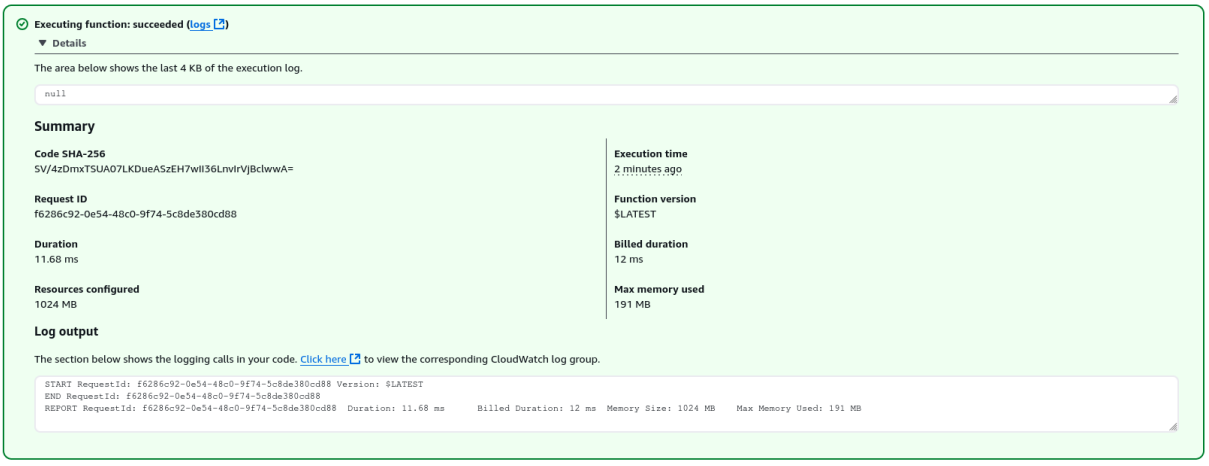

Figura 15 – PyPy Cron AWS console

4.2.7 Exemplo 7 - AWS Python PynamoDB S3 SigURL

O exemplo está disponível no repositório serverless/examples. Como mostrado na Figura 16, o exemplo foi enviado para a AWS Lambda após algumas atualizações de sintaxe e versão do Python no arquivo "serverless.yml". Na Figura 17 é mostrado que o exemplo foi enviado com o PyPy JIT.

```
(base) abner@pop-os:~/Projetos/TCC/aws-python-pynamodb-s3-sigurl$ sls deploy

Deploying "aws-python-pynamodb-s3-sigurl" to stage "dev" (us-east-1)

✔ Service deployed to stack aws-python-pynamodb-s3-sigurl-dev (80s)

endpoints:
  POST - https://br6wa1rbha.execute-api.us-east-1.amazonaws.com/dev/asset
  GET - https://br6wa1rbha.execute-api.us-east-1.amazonaws.com/dev/asset
  GET - https://br6wa1rbha.execute-api.us-east-1.amazonaws.com/dev/asset/{asset_id}
  PUT - https://br6wa1rbha.execute-api.us-east-1.amazonaws.com/dev/asset/{asset_id}
  DELETE - https://br6wa1rbha.execute-api.us-east-1.amazonaws.com/dev/asset/{asset_id}
functions:
  create: sig-s3-uploader2-test-create (15 MB)
  bucket: sig-s3-uploader2-test-bucket (15 MB)
  list: sig-s3-uploader2-test-list (15 MB)
  get: sig-s3-uploader2-test-get (15 MB)
  update: sig-s3-uploader2-test-update (15 MB)
  delete: sig-s3-uploader2-test-delete (15 MB)
```

Figura 16 – CPython Exemplo 7

```
(base) abner@pop-os:~/Projetos/TCC/aws-python-pynamodb-s3-sigurl (1)$ sls deploy

Deploying "aws-python-pynamodb-s3-sigurl-pypy" to stage "dev" (us-east-1)

✔ Service deployed to stack aws-python-pynamodb-s3-sigurl-pypy-dev (102s)

endpoints:
  POST - https://0jlxjxzu62.execute-api.us-east-1.amazonaws.com/dev/asset
  GET - https://0jlxjxzu62.execute-api.us-east-1.amazonaws.com/dev/asset
  GET - https://0jlxjxzu62.execute-api.us-east-1.amazonaws.com/dev/asset/{asset_id}
  PUT - https://0jlxjxzu62.execute-api.us-east-1.amazonaws.com/dev/asset/{asset_id}
  DELETE - https://0jlxjxzu62.execute-api.us-east-1.amazonaws.com/dev/asset/{asset_id}
functions:
  create: sig-s3-uploader-pypy-test-create (60 MB)
  bucket: sig-s3-uploader-pypy-test-bucket (60 MB)
  list: sig-s3-uploader-pypy-test-list (60 MB)
  get: sig-s3-uploader-pypy-test-get (60 MB)
  update: sig-s3-uploader-pypy-test-update (60 MB)
  delete: sig-s3-uploader-pypy-test-delete (60 MB)
```

Figura 17 – PyPy JIT Exemplo 7

Como visto na documentação deste exemplo, é necessário pré-assinar uma URL do AWS S3 para enviar os arquivos. A assinatura é realizada por meio de uma biblioteca chamada "boto3", que está presente no AWS SDK for Python.

Devido à complexidade adicional imposta pelo uso do serviço S3, mesmo sendo um recurso nativo da AWS, essa abordagem não se mostrou vantajosa para este estudo. Além disso, a execução com o PyPy JIT exigiria o envio de uma biblioteca adicional, a "boto3", o que torna o processo ainda mais complexo.

4.2.8 Exemplo 8 - AWS Python

O exemplo AWS Python demonstra apenas como enviar uma função simples para a AWS Lambda em Python, sendo muito similar ao exemplo 1.

```
➜  aws-python sls deploy

Deploying "aws-python" to stage "dev" (us-east-1)

✔ Service deployed to stack aws-python-dev (46s)

functions:
  hello: aws-python-dev-hello (1.7 kB)

➜  aws-python serverless invoke --function hello

{
    "statusCode": 200,
    "body": "{\"message\": \"Go Serverless v4.0! Your function executed successfully!\"}"
}
```

Figura 18 – CPython Exemplo 8

Na Figura 18, é possível observar que a função foi enviada e executada com o CPython. As Figuras 19 e 20 mostram que o exemplo foi enviado e executado usando o PyPy JIT no AWS Lambda.

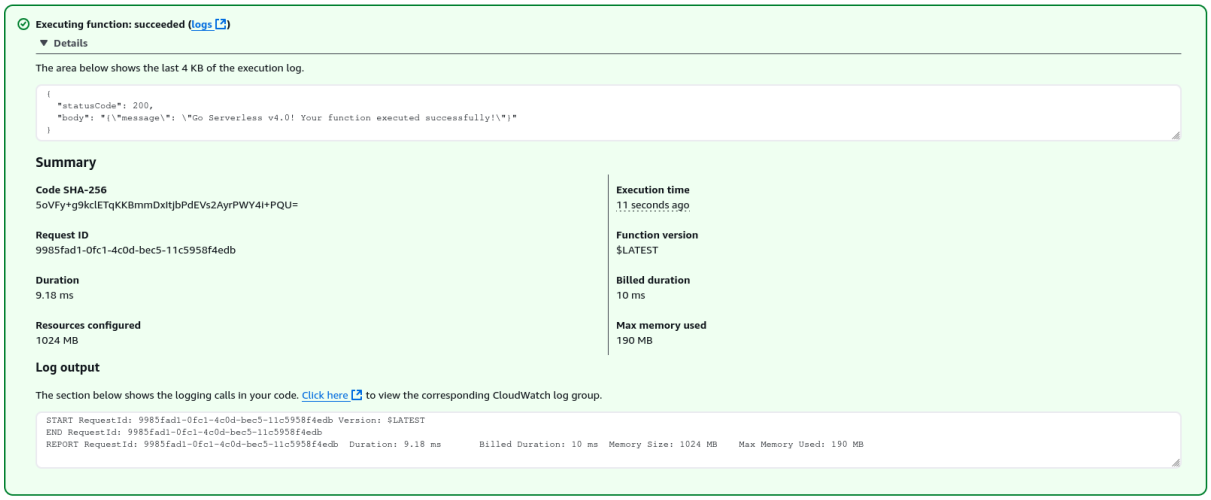

Figura 19 – PyPy Exemplo 8 - Console AWS Lambda

```
➜  aws-python-pypy sls deploy

Deploying "aws-python-pypy" to stage "dev" (us-east-1)

✔ Service deployed to stack aws-python-pypy-dev (67s)

functions:
  hello: aws-python-pypy-dev-hello (45 MB)

➜  aws-python-pypy serverless invoke --function hello

{
    "statusCode": 200,
    "body": "{\"message\": \"Go Serverless v4.0! Your function executed successfully!\"}"
}
```

Figura 20 – PyPy Exemplo 8 - Terminal

4.2.9 Exemplo 9 - AWS Python HTTP API com PynamoDB

Nas Figuras 21 e 22, é possível ver que o exemplo foi enviado tanto para o CPython quanto para o PyPy JIT, mas não foi possível executá-los sem erro. Ao executar usando o CPython, o erro apresentado está na Figura 23. Já executando com PyPy JIT o erro é exibido como mostrado na Figura 24.

```
➜  aws-python-http-api-with-pynamodb sls deploy

Deploying "serverless-http-api-pynamodb" to stage "dev" (us-east-1)

✔ Service deployed to stack serverless-http-api-pynamodb-dev (78s)

endpoints:
  POST - https://wuspmlxga5.execute-api.us-east-1.amazonaws.com/todos
  GET - https://wuspmlxga5.execute-api.us-east-1.amazonaws.com/todos
  GET - https://wuspmlxga5.execute-api.us-east-1.amazonaws.com/todos/{id}
  PUT - https://wuspmlxga5.execute-api.us-east-1.amazonaws.com/todos/{todo_id}
  DELETE - https://wuspmlxga5.execute-api.us-east-1.amazonaws.com/todos/{todo_id
}
functions:
  create: serverless-http-api-pynamodb-dev-create (15 MB)
  list: serverless-http-api-pynamodb-dev-list (15 MB)
  get: serverless-http-api-pynamodb-dev-get (15 MB)
  update: serverless-http-api-pynamodb-dev-update (15 MB)
  delete: serverless-http-api-pynamodb-dev-delete (15 MB)
```

Figura 21 – CPython - Exemplo 9

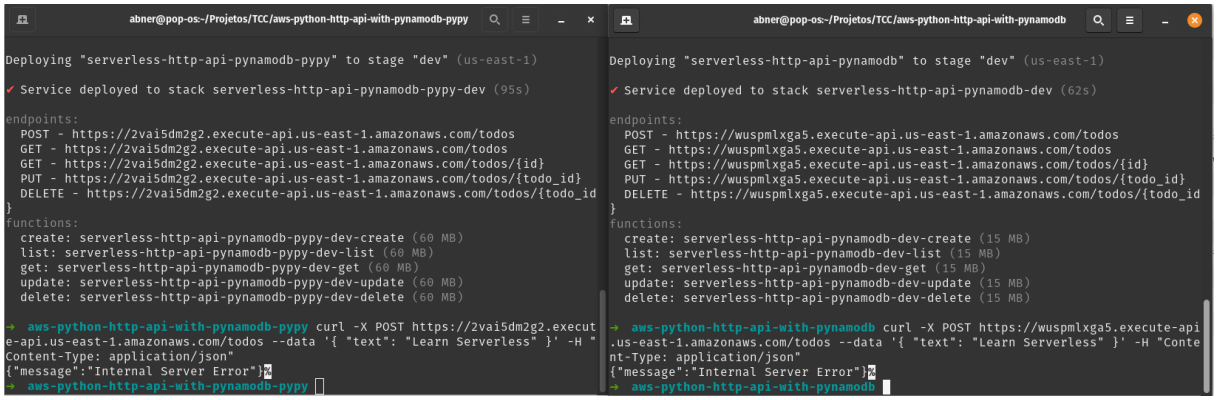

Figura 22 – Execução PyPy JIT e CPython - Exemplo 9

Figura 23 – CPython - Log do erro do exemplo 9

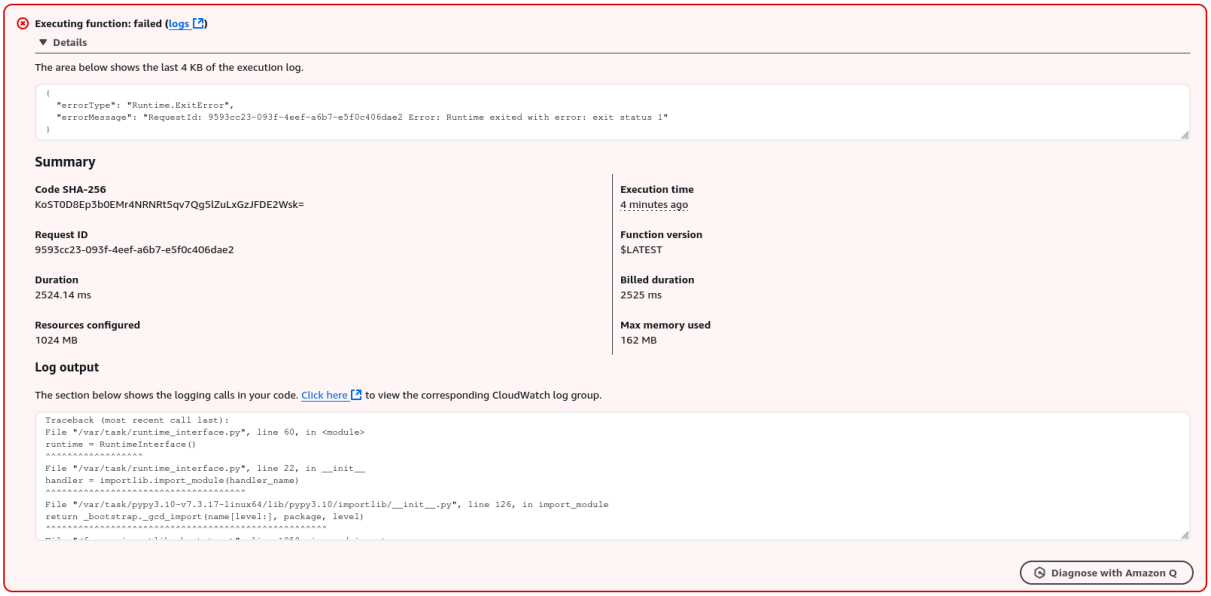

Figura 24 – PyPy JIT - Exemplo 9

O erro apresentado na Figura 23, indica que o problema está intrinsecamente relacionado ao exemplo utilizado, e não necessariamente ao interpretador empregado.

4.2.10 Exemplo 10 - AWS Python SQS Producer-Consumer

Na Figura 25, é possível ver que o exemplo foi enviado e executado com o CPython. Já a Figura 26 mostra que, apesar de ter sido enviado com o PyPy JIT, ele não foi executado. Na Figura 27, é mostrado o erro, o qual é a falta da biblioteca 'boto3' para o PyPy JIT.

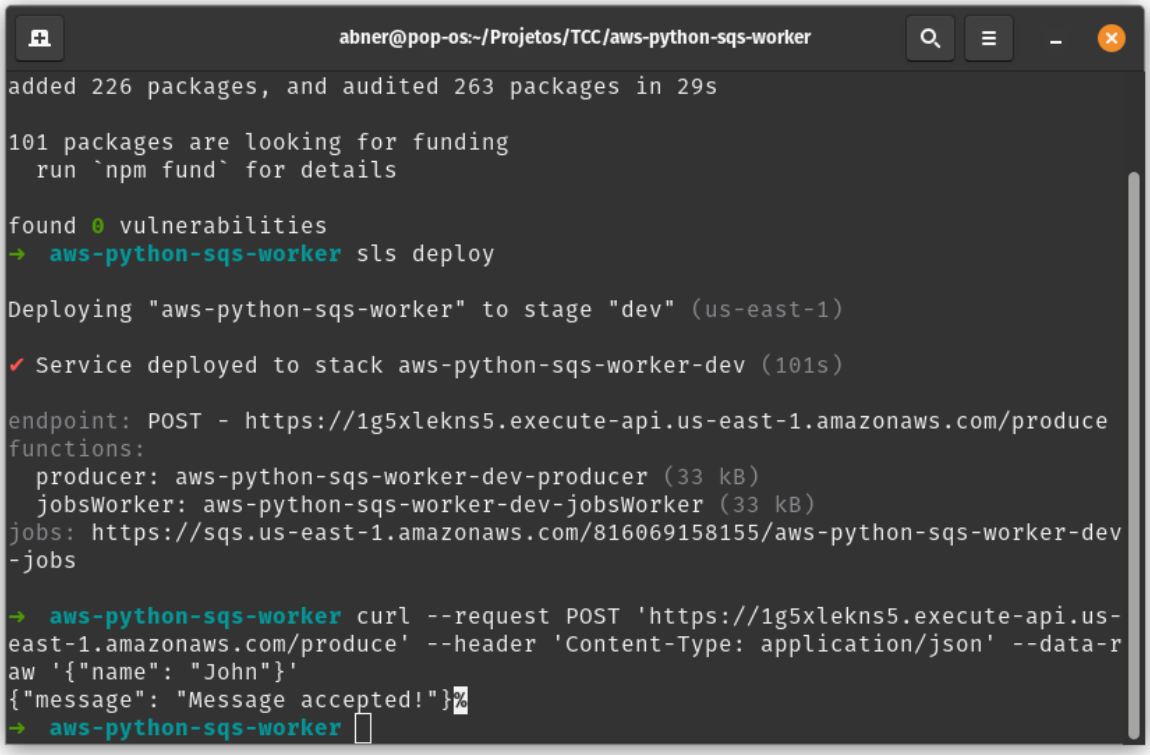

Figura 25 – CPython exemplo 10

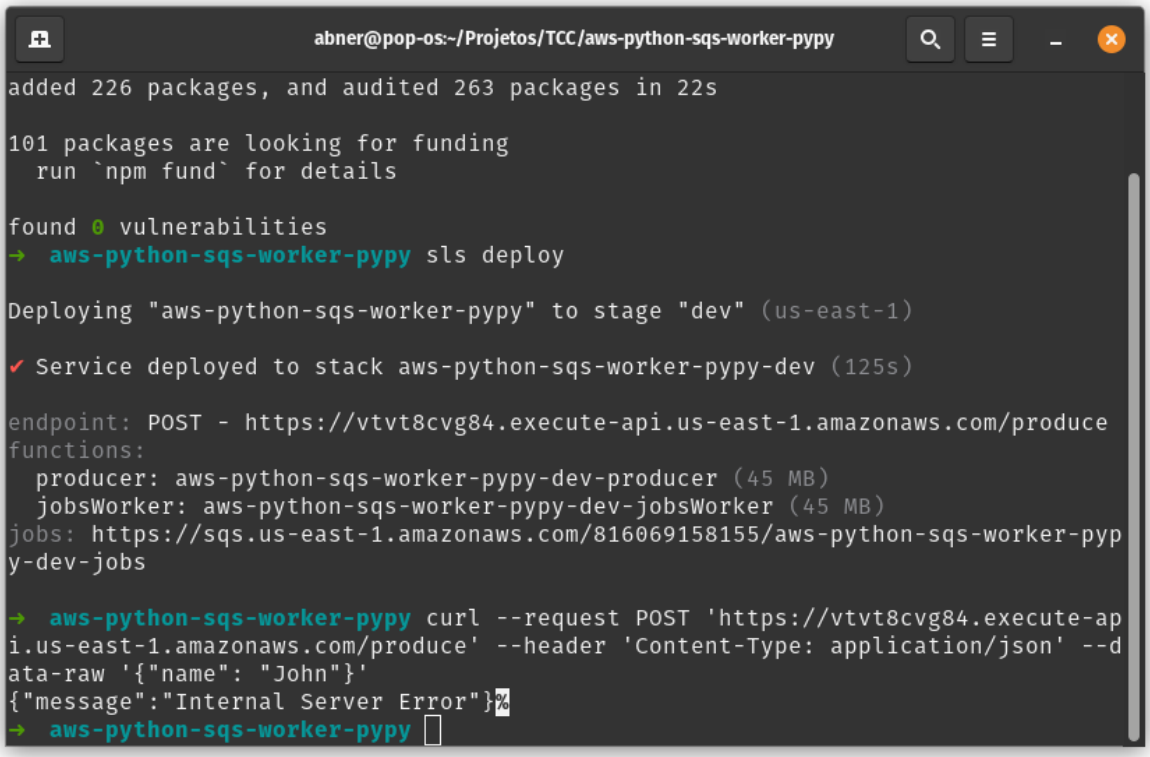

Figura 26 – PyPy Exemplo 10 Terminal

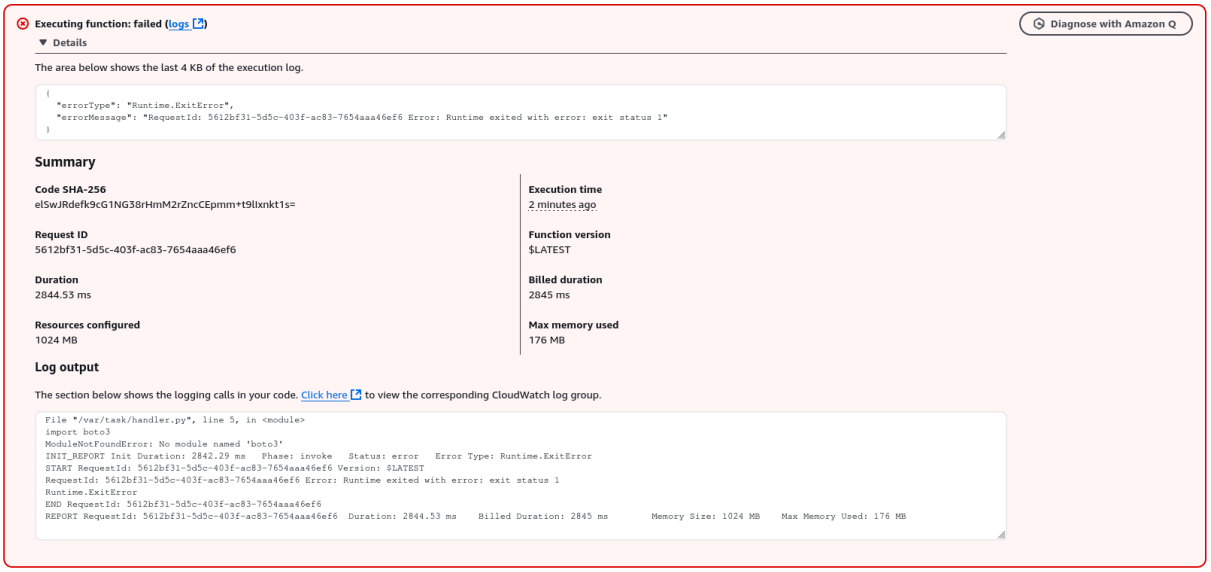

Figura 27 – PyPy Exemplo 10 Console AWS

4.2.11 Exemplo 11 - AWS Python Simple HTTP Endpoint

O exemplo a seguir consiste em um endpoint GET que retorna o horário atual do servidor e está disponível no seguinte link. As Figuras 28 e 30 ilustram o envio das funções para os

interpretadores CPython e PyPy JIT, respectivamente. A execução foi bem-sucedida utilizando o CPython, conforme mostrado na Figura 29. No entanto, ao tentar executá-lo com o PyPy JIT, a operação não foi concluída com sucesso, como evidenciado na Figura 31.

```
$ sls deploy

Deploying "aws-python-simple-http-endpoint" to stage "dev" (us-east-1)

✓ Service deployed to stack aws-python-simple-http-endpoint-dev (54s)

endpoint: GET - https://p9k8r67yea.execute-api.us-east-1.amazonaws.com/time
functions:
  currentTime: aws-python-simple-http-endpoint-dev-currentTime (2 kB)
```

Figura 28 – CPython HTTP Endpoint

```
abner@AbnerPC MINGW64 ~/OneDrive/Documentos/TCC/cpython/aws-python-simple-http-endpoint (main)
$ serverless invoke --function currentTime --log

{
    "statusCode": 200,
    "body": "{\"message\": \"Hello, the current time is 21:19:52.971051\"}"
}

--------------------

START - b1d1046a-9a8a-4987-b2e5-051a62231de8 - Version: $LATEST

REPORT - b1d1046a-9a8a-4987-b2e5-051a62231de8
Duration: 1.34 ms       Billed Duration: 2 ms   Memory Size: 1024 MB    Max Memory Used: 31 MB  Init Duration: 79.31 ms
```

Figura 29 – CPython Execução HTTP Endpoint

```
abner@AbnerPC MINGW64 ~/OneDrive/Documentos/TCC/pypy/aws-python-simple-http-endpoint-pypy (main)
$ sls deploy

Deploying "aws-python-simple-http-endpoint-pypy" to stage "dev" (us-east-1)

✓ Service deployed to stack aws-python-simple-http-endpoint-pypy-dev (54s)

endpoint: GET - https://egq49y4n13.execute-api.us-east-1.amazonaws.com/time
functions:
  currentTime: aws-python-simple-http-endpoint-pypy-dev-currentTime (43 MB)
```

Figura 30 – PyPy JIT HTTP Endpoint

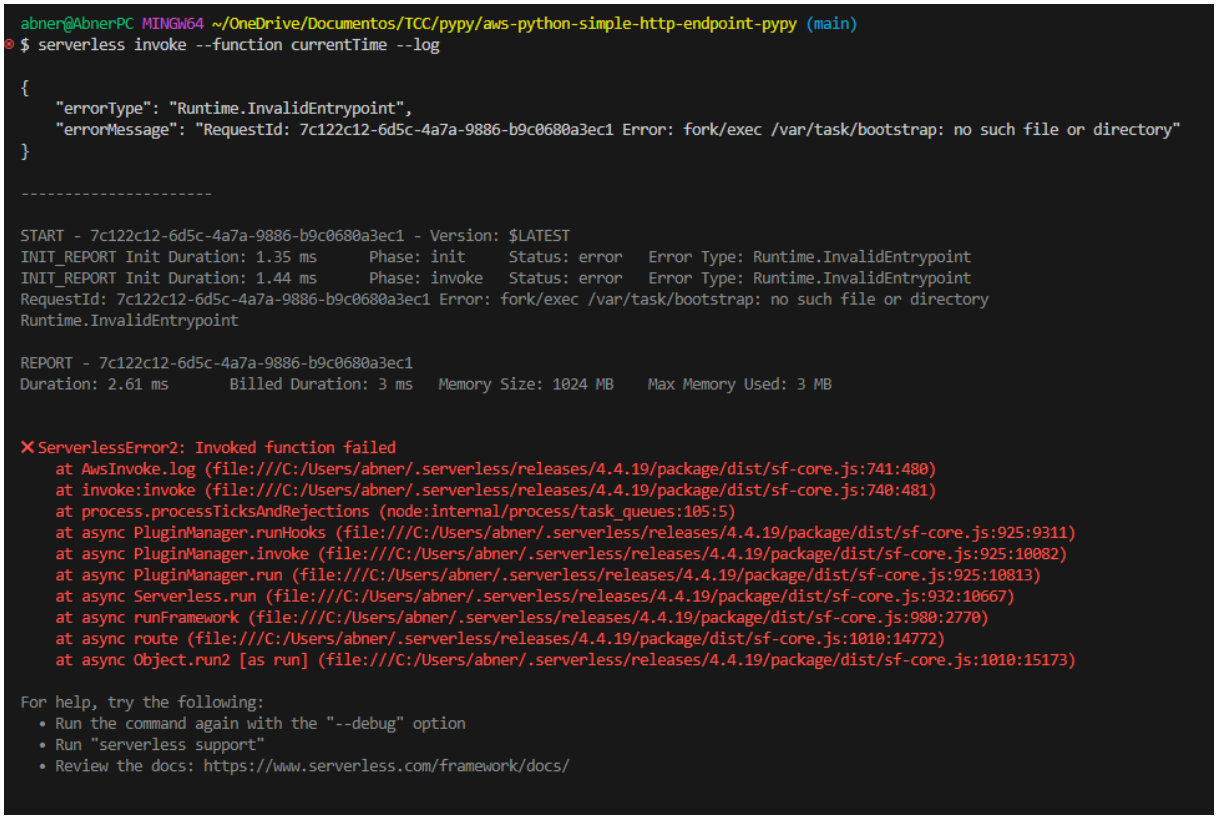

Figura 31 – PyPy JIT Execução HTTP Endpoint

4.2.12 Exemplo 12 - Números primos

Scheller (2018) criou um exemplo de *benchmark* usando o cálculo de números primos, tanto com CPython quanto com PyPy JIT. Por isso, foi executado novamente seu exemplo usando ambos os interpretadores.

Na Figura 32, é possível observar que o exemplo de Scheller (2018) foi enviado e executado normalmente com o PyPy JIT. Já na Figura 33, foi registrado um erro ao executar a função com o CPython. Para corrigir esse erro, foi necessário aumentar o tempo de execução da função no AWS Lambda, ajustando o tempo padrão de 6 segundos para 30 segundos. Após essa alteração, a função foi executada normalmente, sem intercorrências.

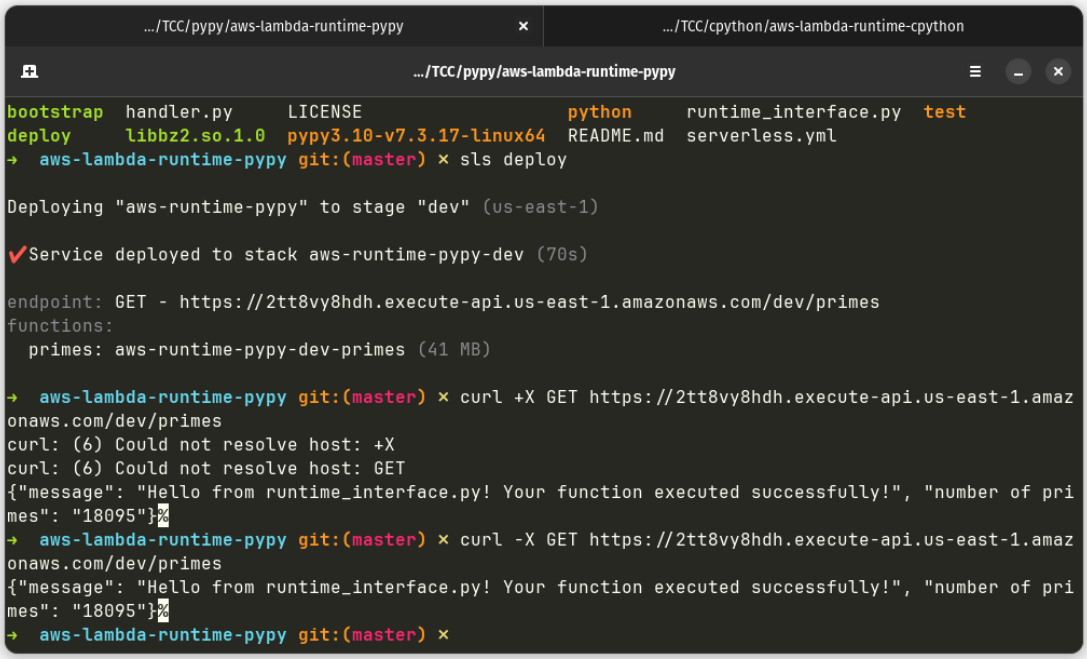

Figura 32 – PyPy Primes

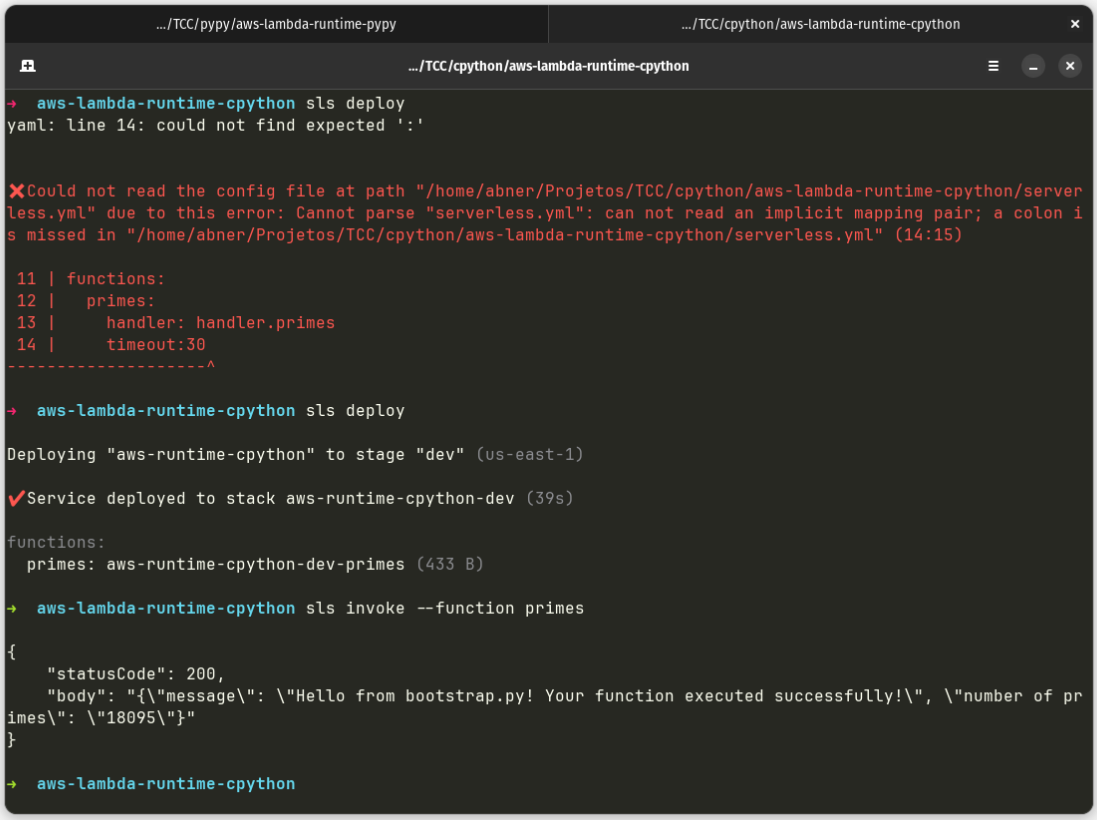

Figura 33 – CPython Primes

4.3 Análise dos resultados

Os testes revelaram a falta de garantias tanto na execução do PyPy JIT na AWS Lambda quanto na confiabilidade dos exemplos fornecidos pelos desenvolvedores do Serverless Framework. Dos doze exemplos testados, apenas quatro foram executados com o PyPy JIT, enquanto outros quatro não foram executados com o CPython, indicando que os exemplos não foram atualizados para acompanhar as mudanças na AWS Lambda.

Como o PyPy JIT é executado representa uma limitação significativa, pois o interpretador deve ser enviado juntamente com a função e as bibliotecas utilizadas. Mesmo com essa adaptação, alguns exemplos não foram executados com sucesso, comprometendo a garantia de que funcionarão corretamente, tanto no presente quanto no futuro.

Além disso, há pouca informação disponível sobre como os *runtimes* customizados são executados na AWS Lambda, e não se sabe exatamente como a AWS os configura ou se, de fato, utiliza o CPython – podendo, inclusive, tratar-se apenas de uma emulação do seu funcionamento.

A falta de conhecimento sobre a configuração do ambiente de execução do CPython,

aliada à necessidade de enviar o interpretador PyPy JIT e as bibliotecas necessárias para a execução das funções, torna inviável um teste comparativo de desempenho entre os dois compiladores.

5 Conclusão

“As palavras fogem quando precisamos delas e sobram quando não pretendemos usá-las.”
Carlos Drummond de Andrade

Esta pesquisa justifica-se pela necessidade de avaliar a viabilidade de testar compiladores distintos em aplicações *serverless* para uma mesma linguagem. Embora o desempenho não tenha sido avaliado neste trabalho, estudos desse tipo continuam sendo necessários. Para tanto, buscou-se compreender a estrutura típica de uma aplicação e de uma plataforma *serverless*. Dessa forma, optou-se pela AWS como plataforma e pelo Python como linguagem popular para realizar a verificação do processo de avaliação de desempenho.

O estudo empreendeu uma análise preliminar do domínio por meio de estudos de caso envolvendo exemplos de funções *serverless*. Durante os testes, foram identificadas diversas dificuldades na avaliação dessas aplicações.

Concluiu-se, a partir dos testes realizados, que é inviável avaliar o desempenho de compiladores de uma mesma linguagem quando um deles utiliza um ambiente de execução customizado, pois não há igualdade de condições entre os compiladores. Por um lado, o CPython recebe tratamento especial na plataforma e sua configuração interna permanece desconhecida. Por outro lado, o PyPy JIT precisa ser enviado juntamente com a função e suas dependências, o que impacta sua avaliação de desempenho ao adicionar tempo à inicialização da função.

Assim, justifica-se a ausência de métricas – mesmo que preliminares – acerca do desempenho desses compiladores no cenário *serverless*.

A seção 5.1 apresenta os principais resultados alcançados, enquanto a seção 5.2 discute as limitações da pesquisa. Por fim, a seção 5.3 aponta algumas direções para trabalhos futuros.

5.1 Resultados

Três principais resultados foram obtidos neste trabalho.

O primeiro resultado refere-se à seleção de um conjunto de 20 exemplos como uma potencial representação de programas *serverless* típicos. Para isso, foram selecionados exemplos fornecidos pelos desenvolvedores da ferramenta, além do exemplo criado por Scheller (2018). Embora o repositório serverless/examples contenha mais exemplos, alguns não foram utilizados por exigirem tokens de autorização, dependerem de ferramentas externas à AWS ou utilizarem Python 2.7.

O segundo resultado refere-se à execução de parte desses exemplos utilizando o compilador PyPy JIT. Esse resultado pode ser dividido em duas etapas. A primeira consistiu na

execução dos exemplos utilizando apenas o CPython 3.10. Já a segunda envolveu a execução desses mesmos exemplos com o PyPy JIT. A versão 3.10 do Python foi escolhida por ser a mais recente compatível com o PyPy JIT no início dos testes. Constatou-se que a maioria dos exemplos não havia sido atualizada para a versão mais recente do framework, a 4. Assim, quando possível, foram realizadas atualizações e correções relacionadas a problemas de versionamento.

O terceiro resultado refere-se à impossibilidade de testar o desempenho dos dois compiladores escolhidos em igualdade de condições.

Por um lado, o CPython, por ser oficialmente suportado, recebe tratamento especial na plataforma. Seu real funcionamento interno é um segredo comercial e, portanto, oculto para os usuários. Dessa forma, não é possível determinar com precisão como a AWS executa códigos compilados em CPython.

Por outro lado, o PyPy JIT apresenta grandes limitações na AWS. O interpretador precisa ser enviado com a função a ser executada, assim sabemos seu funcionamento. Foi tentada a inclusão do PyPy JIT em uma camada, porém, devido às restrições de tamanho impostas pela AWS, essa abordagem não foi viável.

Como consequência, esta pesquisa concluiu que não é possível realizar uma comparação justa entre os dois compiladores dentro do ambiente AWS Lambda.

5.2 Limitações

Ao longo do trabalho, foram identificadas algumas limitações em funções dos procedimentos metodológicos adotados.

A primeira limitação foi a escolha de apenas 20 exemplos. Essa decisão se justificou pela dificuldade em selecionar códigos *serverless* representativos e pela falta de atualizações nos exemplos disponíveis. Ampliar a quantidade de programas de teste seria fundamental para abranger um espectro mais amplo de casos de uso no cenário *serverless*, que inclui diversas funcionalidades não exploradas neste estudo, como processamento de vídeo e aprendizado de máquina.

A segunda limitação está na escolha da plataforma. Optou-se pela AWS por ser a maior provedora de computação em nuvem e por oferecer o AWS Lambda, o serviço *serverless* mais antigo do mercado. No entanto, a ausência de testes em outras plataformas representa uma limitação. A inclusão de outras provedores permitiria comparar diferentes abordagens na implementação de ambientes customizados de execução.

A terceira limitação diz respeito à escolha da linguagem. O uso exclusivo de Python restringiu a análise, uma vez que outras linguagens poderiam ter sido consideradas, como linguagens compiladas (por exemplo, C e Rust) ou linguagens baseadas em máquinas virtuais (como Java e Elixir).

A quarta limitação foi a escolha de apenas dois interpretadores Python. Além do CPython e do PyPy JIT, existem outros compiladores Python, como o RustPython, que poderia permitir

comparações em condições mais equitativas dentro do AWS Lambda, pois também exigiria um ambiente customizado de execução.

A quinta limitação está na escolha de uma nuvem pública. Caso uma nuvem privada tivesse sido utilizada, os resultados poderiam ser diferentes. Em um ambiente controlado, com conhecimento detalhado sobre o hardware e os processos em execução, seria possível isolar variáveis e obter uma análise mais precisa do desempenho dos compiladores.

5.3 Trabalhos futuros

O trabalho realizado abriu novas possibilidades para pesquisas futuras. A seguir, são discutidas algumas direções para estudos baseados nos resultados encontrados e nas limitações identificadas.

A primeira oportunidade é a ampliação do conjunto de exemplos analisados. Embora mais exemplos pudessem ter sido selecionados, a limitação de tempo para a execução dos testes restringiu a quantidade utilizada neste estudo. Considerando a diversidade do cenário *serverless*, aumentar o número de programas avaliados será essencial para uma análise mais abrangente de desempenho.

A segunda oportunidade é a investigação de outras plataformas públicas de computação em nuvem, como Microsoft Azure e Google Cloud. Compreender como essas plataformas lidam com a criação de ambientes customizados e comparar as vantagens e desvantagens em relação ao AWS Lambda pode fornecer visões valiosas sobre diferentes abordagens na execução de funções *serverless*.

A terceira oportunidade envolve a inclusão de outras linguagens de programação e/ou compiladores de uma mesma linguagem. A ampliação dos testes para contemplar diferentes linguagens permitiria uma visão mais completa sobre o desempenho de aplicações *serverless* em ambientes customizados, bem como possíveis otimizações para cada caso.

A quarta oportunidade é a criação de um ambiente de nuvem privada. Em um ambiente privado, onde há total controle sobre a infraestrutura e a execução das funções, seria possível reduzir variáveis desconhecidas, garantindo maior precisão na avaliação de desempenho ao testar diferentes linguagens e compiladores.

Referências

AWS. *O que é uma arquitetura sem servidor ?* 2023. Disponível em: <https://aws.amazon.com/pt/lambda/serverless-architectures-learn-more/>. Citado na página 12.

BARANY, G. Python interpreter performance deconstructed. In: *Proceedings of the Workshop on Dynamic Languages and Applications*. [S.l.: s.n.], 2014. p. 1–9. Citado na página 18.

BRENNER, S.; KAPITZA, R. Trust more, serverless. In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. [S.l.: s.n.], 2019. p. 33–43. Citado nas páginas 13 e 24.

CARREIRA, J. et al. From warm to hot starts: Leveraging runtimes for the serverless era. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. [S.l.: s.n.], 2021. p. 58–64. Citado nas páginas 13 e 25.

CHEN, J. et al. A survey of compiler testing. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 53, n. 1, p. 1–36, 2020. Citado nas páginas 12, 18, 19, 20 e 21.

COPIK, M. et al. Sebs: A serverless benchmark suite for function-as-a-service computing. In: *Proceedings of the 22nd International Middleware Conference*. [S.l.: s.n.], 2021. p. 64–78. Citado na página 24.

CRAPÉ, A.; EECKHOUT, L. A rigorous benchmarking and performance analysis methodology for python workloads. In: IEEE. *2020 IEEE International Symposium on Workload Characterization (IISWC)*. [S.l.], 2020. p. 83–93. Citado nas páginas 12, 16, 17, 21 e 24.

EISMANN, S. et al. A case study on the stability of performance tests for serverless applications. *Journal of Systems and Software*, Elsevier, v. 189, p. 111294, 2022. Citado na página 24.

FLASK. *Deployin to production*. 2025. Disponível em: <https://flask.palletsprojects.com/en/stable/deploying/>. Citado na página 34.

GCC. *GCC developers. GCC Testsuites*. 2023. Disponível em: <https://gcc.gnu.org/onlinedocs/gccint/Testsuites.html#Testsuites>. Citado na página 19.

GOOGLE Acadêmico. 2023. Disponível em: <https://scholar.google.com>. Citado na página 15.

LI, Y. et al. Serverless computing: state-of-the-art, challenges and opportunities. *IEEE Transactions on Services Computing*, IEEE, v. 16, n. 2, p. 1522–1539, 2022. Citado nas páginas 12, 22, 23, 24 e 25.

LLVM. *LLVM developers. LLVM Testing Infrastructure Guide.* 2023. Disponível em: <https://llvm.org/docs/TestingGuide.html>. Citado na página 19.

MAISSEN, P. et al. Faasdom: A benchmark suite for serverless computing. In: *Proceedings of the 14th ACM international conference on distributed and event-based systems*. [S.l.: s.n.], 2020. p. 73–84. Citado na página 24.

MARTINS, H.; ARAUJO, F.; CUNHA, P. R. da. Benchmarking serverless computing platforms. *Journal of Grid Computing*, Springer, v. 18, p. 691–709, 2020. Citado na página 24.

OPENJDK. *OpenJDK developers. OpenJDK Testsuite.* 2023. Disponível em: <http://openjdk.java.net/jtreg>. Citado na página 19.

OVERFLOW, S. *StackOverflow Survey 2024*. 2024. Disponível em: <https://survey.stackoverflow.co/2024/technology#most-popular-technologies-platform>. Citado na página 15.

RODRIGUES, W. C. et al. Metodologia científica. *Faetec/IST. Paracambi*, p. 2–20, 2007. Citado na página 14.

SCHELLER, U. *A Pypy Runtime for AWS Lambda*. 2018. Disponível em: <https://www.ulrich-scheller.de/a-pypy-runtime-for-aws-lambda/>. Citado nas páginas 15, 28, 43 e 47.

SETH, D.; CHINTALE, P. Performance benchmarking of serverless computing platforms. 2024. Citado nas páginas 13 e 25.

SOMU, N. et al. Panopticon: A comprehensive benchmarking tool for serverless applications. In: IEEE. *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. [S.l.], 2020. p. 144–151. Citado na página 24.

STRATEGIES, S. *Nginx announces results of 2016 future of application development and delivery survey*. Speakeasy Strategies, 2016. Disponível em: <https://www.globenewswire.com/news-release/2016/03/29/1060819/0/en/NGINX-Announces-Results-of-2016-Future-of-Application-Development-and-Delivery-Survey.html>. Citado na página 12.

TIOBE. 2025. Disponível em: <https://www.tiobe.com/tiobe-index/>. Citado na página 12.

UEDA, T.; NAKAIKE, T.; OHARA, M. Workload characterization for microservices. In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*. [S.l.: s.n.], 2016. p. 1–10. Citado na página 12.

VIEIRA, A. G. et al. Computação serverless: Conceito, aplicações e desafios. *Sociedade Brasileira de Computação*, 2020. Citado nas páginas 21, 22 e 23.

ZHANG, Y. et al. Towards a serverless java runtime. In: IEEE. *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2021. p. 1156–1160. Citado nas páginas 13 e 25.