Avaliação Comparativa do Cold Start em AWS Lambda: Influência da Linguagem de Programação e da Alocação de Memória

Danimar Veriato*, Heitor Scalco Neto*, Alisson Borges Zanetti*, Gustavo Schwitzki Peretti* e Fábio Diniz Rossi†
* Instituto Federal Catarinense, Concórdia, Santa Catarina, Brasil
E-mail: danimar.veriato@ifc.edu.br
† Instituto Federal Farroupilha, Alegrete, Rio Grande do Sul, Brasil
E-mail: fabio.diniz@acad.pucrs.br

***Abstract*—This paper investigates the impact of cold start on the performance of AWS Lambda functions implemented in seven programming languages: Go, Python, Node.js, .NET (C#), Java, Ruby, and PowerShell. A .NET-based orchestrator was developed to invoke the functions under controlled conditions and accurately measure total response time and internal execution time, isolating the initialization overhead. Experiments were conducted with 128MB and 256MB memory allocations using factorial computation as a standardized benchmark. The results show that compiled languages with lightweight runtimes, such as Go and Python, achieve faster initialization, while languages relying on more complex virtual machines, such as Java and PowerShell, exhibit significantly higher overhead. Increasing the memory allocation reduced cold start times by up to 45%. These findings provide valuable insights for selecting the appropriate language and configuration for *serverless* functions, particularly in latency-sensitive applications.**

***Keywords*—Serverless Computing; Cold Start; AWS Lambda.**

***Resumo*—Este trabalho investiga o impacto do *cold start* no desempenho de funções AWS Lambda implementadas em sete linguagens de programação: Go, Python, Node.js, .NET (C#), Java, Ruby e PowerShell. Para isso, foi desenvolvido um orquestrador em .NET capaz de invocar as funções de forma controlada e medir com precisão o tempo de resposta total e o tempo de execução interno, permitindo isolar o overhead de inicialização. Os experimentos foram realizados com alocações de memória de 128MB e 256MB, utilizando o cálculo do fatorial como benchmark padronizado. Os resultados indicam que linguagens compiladas e com runtimes leves, como Go e Python, apresentam menor tempo de inicialização, enquanto linguagens que dependem de máquinas virtuais robustas, como Java e PowerShell, exibem overheads significativamente mais altos. A elevação da memória reduziu o tempo de *cold start* em até 45%. Os achados oferecem subsídios relevantes para a escolha da linguagem e configuração de funções *serverless*, especialmente em aplicações sensíveis à latência.**

***Palavras-chave*—Computação Sem Servidor; Cold Start; AWS Lambda.**

I. Introdução

A computação em nuvem transformou profundamente o desenvolvimento, a implantação e a manutenção de aplicações modernas, ao introduzir novos paradigmas que otimizam o uso de recursos computacionais e reduzem significativamente os custos operacionais. A virtualização da infraestrutura, aliada à elasticidade e ao modelo de pagamento sob demanda, contribuiu para um cenário mais ágil e acessível tanto para grandes corporações quanto para pequenos desenvolvedores.

Entre os diversos modelos de computação em nuvem, destaca-se a computação sem servidor (*serverless*), que representa uma mudança de paradigma ao permitir que os desenvolvedores concentrem-se exclusivamente na lógica de negócio, abstraindo completamente a gestão de servidores, escalonamento e manutenção da infraestrutura subjacente. Nesse modelo, a alocação e liberação de recursos são gerenciadas automaticamente pelo provedor de nuvem, tornando a escalabilidade mais transparente e eficiente. Lançado pela Amazon em 2014, o AWS Lambda tornou-se um dos serviços mais populares dessa categoria, permitindo a execução de funções em resposta a eventos, com escalabilidade automática, alta disponibilidade e modelo de cobrança baseado apenas no tempo de execução efetiva da função [1].

Apesar das inúmeras vantagens, a computação *serverless* ainda enfrenta desafios técnicos relevantes. Um dos principais é o fenômeno conhecido como *cold start*, que ocorre quando uma função é invocada após um período de inatividade e precisa ter seu ambiente de execução inicializado do zero. Esse processo pode introduzir latência adicional na resposta, o que compromete a experiência do usuário e a adequação de sistemas que exigem respostas em tempo quase real, como aplicações

web interativas, sistemas de pagamentos, IoT e microserviços altamente acoplados [2].

O AWS Lambda oferece suporte a diversas linguagens de programação, incluindo Node.js, Python, Java, .NET (C#), Ruby, Go e PowerShell. Cada uma dessas linguagens possui características específicas de compilação, tempo de carregamento e inicialização do ambiente, que influenciam diretamente o desempenho durante o *cold start*. A escolha da linguagem, portanto, não deve ser guiada apenas por preferências pessoais ou exigências da equipe, mas também por critérios técnicos e de desempenho, especialmente em cenários sensíveis à latência.

Este estudo propõe uma análise comparativa do desempenho dessas linguagens no contexto de funções AWS Lambda submetidas ao *cold start*, utilizando o cálculo do fatorial como *benchmark*. Esta operação foi escolhida por sua natureza determinística, implementação simples e por permitir o controle da complexidade computacional em diferentes runtimes, sem introduzir dependências externas ou variações inesperadas no processamento.

A motivação para esta pesquisa surgiu a partir de experiências práticas em ambientes de *e-commerce*, nos quais atrasos introduzidos por *cold starts* afetavam significativamente a performance de integrações entre sistemas e comprometiam a confiabilidade de processos críticos. Dessa forma, o presente trabalho visa oferecer subsídios técnicos que auxiliem desenvolvedores e arquitetos de software na escolha mais adequada da linguagem e configuração de memória, com vistas à otimização do desempenho em plataformas *serverless*. Além disso, os dados obtidos podem embasar estratégias de mitigação, como o uso de *Provisioned Concurrency*, aumento de memória ou técnicas de *keep-warm*, contribuindo para o desenvolvimento de soluções mais eficientes [3].

As próximas seções apresentam o referencial teórico sobre computação sem servidor, a formulação do problema de pesquisa, a descrição da metodologia experimental adotada, a análise dos resultados obtidos e, por fim, as conclusões e recomendações derivadas do estudo.

II. Referencial Teórico

Nesta seção, são apresentados os conceitos fundamentais da computação sem servidor, seus princípios operacionais e as principais diferenças em relação aos modelos tradicionais. Em seguida, discute-se a implementação de destaque no mercado, o AWS Lambda, estabelecendo a base conceitual necessária para compreender o problema do *cold start* e seu impacto no desempenho de diferentes linguagens de programação.

A. Computação sem Servidor

A computação sem servidor representa uma evolução significativa na forma de desenvolver aplicações em nuvem. Apesar do nome sugerir ausência de servidores, o que ocorre, na prática, é a abstração da gestão da infraestrutura pelo provedor de nuvem, isentando o desenvolvedor dessa responsabilidade [4].

A arquitetura *serverless* organiza-se em dois pilares principais, com características e tempos de vida distintos: *Backend as a Service* (BaaS) e *Functions as a Service* (FaaS). O BaaS oferece serviços de longa duração como bancos de dados (Amazon DynamoDB, Firebase Realtime Database), autenticação (Amazon Cognito, Firebase Authentication), armazenamento de arquivos (Amazon S3, Google Cloud Storage), mensageria (Amazon SQS, Google Pub/Sub) e distribuição de conteúdo (Amazon CloudFront, Firebase Hosting). Esses serviços permanecem continuamente disponíveis, sem necessidade de inicialização por evento, o que os torna ideais para manter o estado da aplicação [5].

Por outro lado, o FaaS — representado por serviços como AWS Lambda, Azure Functions e Google Cloud Functions — executa funções de curta duração, sob demanda, acionadas por eventos. Esse modelo se caracteriza pela granularidade fina do código, curta duração e escalabilidade automática [6]. Após a implantação, a função permanece inativa até ser acionada, momento em que o provedor aloca os recursos necessários e, ao término da execução, os libera. Essa característica a torna suscetível ao problema do *cold start*, que não afeta os componentes do tipo BaaS.

Além dos modelos tradicionais, surgiram soluções híbridas como AWS Fargate e Google Cloud Run, classificadas como *Container as a Service* (CaaS), combinando a abstração do modelo *serverless* com a flexibilidade dos contêineres. Essas soluções permitem execuções mais longas e controle refinado sobre o ambiente, sem exigir gerenciamento direto de infraestrutura.

As vantagens da computação *serverless* explicam sua adoção crescente: redução de custos, escalabilidade automática e agilidade no desenvolvimento [7]. No entanto, limitações importantes persistem. O *cold start*, por exemplo, pode introduzir latências críticas em aplicações sensíveis ao tempo de resposta [2]. A cobrança baseada no uso efetivo (tempo de execução, armazenamento ou recursos alocados) difere dos modelos tradicionais baseados em instâncias reservadas [8]. A escalabilidade imediata é viabilizada por técnicas como contêineres leves e microVMs [9].

B. Implementações

O AWS Lambda, lançado em 2014, consolidou-se como referência em soluções de computação *serverless*. Desenvolvido pela Amazon Web Services, o serviço foi pioneiro ao permitir a execução de trechos de código em resposta a eventos, sem que o desenvolvedor precise gerenciar servidores ou provisionar

recursos manualmente. Seu suporte a múltiplas linguagens de programação — incluindo Node.js, Python, Java, .NET (C#), Ruby, Go e PowerShell [10] — torna-o uma opção versátil para diversos tipos de aplicações e perfis de desenvolvedores.

Além disso, o Lambda se integra de forma nativa a mais de 140 serviços da AWS, como S3, DynamoDB, API Gateway, EventBridge e SNS, possibilitando a criação de arquiteturas orientadas a eventos com alta modularidade e escalabilidade. Essa integração nativa é um dos diferenciais da plataforma, permitindo que funções sejam acionadas automaticamente em resposta a mudanças em bancos de dados, uploads de arquivos, mensagens em filas, requisições HTTP, entre outros.

Consciente dos impactos do *cold start*, a AWS introduziu mecanismos específicos para mitigar esse problema. O recurso *Provisioned Concurrency*, por exemplo, mantém instâncias previamente inicializadas da função, prontas para atender imediatamente às requisições, eliminando a latência do primeiro acesso. Já o *Lambda SnapStart*, voltado exclusivamente para a linguagem Java, realiza uma captura do estado da função após a inicialização e permite restaurá-la rapidamente em execuções subsequentes. Embora esses recursos reduzam consideravelmente o tempo de inicialização, eles também acarretam custo adicional, o que exige avaliação criteriosa com base nos requisitos de desempenho e orçamento do projeto.

Outros provedores também evoluíram suas soluções *serverless* em resposta ao sucesso do Lambda. O Azure Functions, da Microsoft, é uma das principais alternativas e destaca-se pela profunda integração ao ecossistema .NET e aos serviços do Azure. Ele oferece diferentes planos de execução — Plano de Consumo, Plano Premium e Plano de Serviço de Aplicativo — permitindo ajustar o comportamento de escalabilidade e disponibilidade conforme a criticidade da aplicação. O Azure Functions também suporta *durable functions*, permitindo a orquestração de fluxos complexos de execução, com manutenção de estado entre chamadas, o que amplia seu espectro de aplicabilidade.

Já o Google Cloud Functions, solução equivalente da Google Cloud Platform, possui integração profunda com ferramentas como Firebase, BigQuery e Cloud Pub/Sub. Essa integração favorece aplicações móveis, web em tempo real e análises baseadas em grandes volumes de dados. Uma extensão natural dessa abordagem é o Google Cloud Run, que combina os benefícios do modelo *serverless* com a flexibilidade dos contêineres. Com ele, é possível empacotar a aplicação completa como um contêiner Docker e executá-la de forma elástica, com escalabilidade automática e sem gerenciamento direto da infraestrutura subjacente.

Embora cada provedor apresente particularidades em sua implementação, todos compartilham desafios comuns — sobretudo em relação ao tempo de inicialização das funções em ambientes efêmeros. A latência introduzida durante o *cold start* é uma limitação recorrente, especialmente em linguagens com ambientes de execução mais complexos, como Java e .NET. Estudos comparativos, como o conduzido por [11], indicam que o AWS Lambda tende a apresentar desempenho superior em cenários de *cold start*, em parte devido à maturidade da plataforma, ao suporte contínuo da AWS para otimizações específicas, e à disponibilidade de mecanismos avançados de mitigação.

Com base nesses aspectos, a escolha entre diferentes plataformas *serverless* deve considerar não apenas os custos e integrações oferecidas, mas também o perfil de uso esperado, a linguagem adotada e a criticidade dos requisitos de latência, que podem ser fortemente influenciados pela forma como o *cold start* é tratado em cada ambiente.

III. Trabalhos Relacionados

A latência introduzida pelo *cold start* em plataformas *p* (FaaS) é um desafio amplamente reconhecido, que tem motivado uma série de investigações na comunidade científica. A literatura recente explora o fenômeno sob diferentes perspectivas, desde a caracterização de cargas de trabalho em produção até a proposição de novas arquiteturas de sistema. Nesta seção, discutimos trabalhos que, embora correlatos, apresentam focos distintos e complementares à nossa análise.

Um estudo conduzido por Shahrad et al. (2020) realizou uma análise empírica sobre o comportamento de funções *serverless* em um provedor de nuvem de grande porte. Ao examinar dados de produção, os autores constataram que a maioria das funções possui execuções de curta duração e são invocadas de maneira esporádica. Essa observação reforça a criticidade do *cold start* como um gargalo de desempenho prevalente em cenários reais. A principal distinção para o nosso trabalho reside na abordagem: enquanto Shahrad et al. (2020) oferecem uma visão macroscópica e quantitativa do problema em um ambiente heterogêneo, nossa pesquisa foca em um benchmark controlado para isolar e comparar o impacto de diferentes runtimes sob condições homogêneas.

Com uma abordagem orientada à infraestrutura, Mohan et al. (2019) investigaram as fases do processo de inicialização e propuseram um mecanismo de "provisionamento ágil" de contêineres para mitigar a latência. Este trabalho decompõe o *cold start* em etapas como alocação de recursos, *download* do código e inicialização do ambiente, focando em otimizações na própria plataforma serverless. Em contrapartida, nossa análise se concentra na perspectiva do desenvolvedor, avaliando como as escolhas de tecnologia dentro da plataforma AWS Lambda existente — especificamente a linguagem de programação e a alocação de memória — influenciam a performance, fornecendo diretrizes práticas para a otimização de aplicações.

Por sua vez, Pereira et al. (2022) exploraram como as estratégias de empacotamento da aplicação afetam o tempo de inicialização no AWS Lambda. Os autores demonstraram que fatores como o tamanho do pacote de implantação e o uso de camadas (*layers*) são determinantes para a latência do *cold start*. Esse estudo é complementar ao nosso, pois, enquanto eles analisam o impacto dos artefatos de *deploy*, nossa investigação se aprofunda nos efeitos intrínsecos ao runtime selecionado. A combinação dos achados de ambos os trabalhos permite uma compreensão mais holística dos fatores que governam o desempenho da inicialização de uma função.

A Figura 1 a seguir posiciona nosso estudo em relação a essas pesquisas, destacando as diferentes abordagens e contribuições de cada um.

Trabalho	Foco Principal	Linguagens Analisadas	Métricas Principais	Principal Contribuição
Shahrad et al. (2020)	Caracterização de cargas de trabalho *serverless* em produção.	C#, F#, JavaScript, Python, PowerShell, Java	Duração, frequência de invocação, uso de memória.	Análise em larga escala que demonstra a prevalência e o impacto dos *cold starts* em ambientes reais.
Mohan et al. (2019)	Proposição de uma nova arquitetura para mitigar o *cold start*.	Não focado em comparação de linguagens específicas.	Tempo de inicialização do contêiner e do *runtime*.	Proposta de um sistema de "provisionamento ágil" para acelerar a inicialização de funções.
Pereira et al. (2022)	Impacto das estratégias de empacotamento e implantação no *cold start*.	Node.js, Python	Latência de *cold start*.	Demonstração de que o tamanho do pacote de código e o uso de *layers* impactam significativamente a latência.
Este Estudo	Análise comparativa do impacto da linguagem e da memória no *cold start*.	Go, Python, Node.js, .NET, Java, Ruby, PowerShell	*Overhead* de inicialização, tempo de execução interno.	*Benchmark* controlado que isola e quantifica o impacto de 7 linguagens e 2 configurações de memória no *cold start* do AWS Lambda.

Fig. 1. Comparativo entre Trabalhos Relacionados

IV. Problema de Pesquisa

A computação *serverless* trouxe avanços relevantes para o desenvolvimento de aplicações em nuvem, ao permitir que equipes foquem na lógica de negócio sem a necessidade de gerenciar infraestrutura. Por outro lado, esse modelo também introduziu desafios específicos, entre esses, o *cold start*, que se destaca como uma das principais limitações enfrentadas por desenvolvedores que utilizam AWS Lambda. Esse fenômeno ocorre quando uma função é invocada após um período de inatividade, exigindo que o ambiente de execução seja inicializado do zero, o que introduz latência adicional perceptível em aplicações com requisitos de resposta imediata [2].

Na prática, o impacto do *cold start* é particularmente relevante em arquiteturas que utilizam múltiplas funções em cadeia, como fluxos de integração de sistemas, pipelines de processamento de dados ou treinamentos de modelos de inteligência artificial. Nesses cenários, cada função depende da anterior, de forma que o tempo de inicialização de uma única etapa pode atrasar todo o fluxo, comprometendo desempenho, custo e experiência do usuário. Assim, a escolha da linguagem de programação utilizada em funções *serverless* pode ter influência direta sobre o tempo total de resposta e a eficiência operacional, especialmente em aplicações que exigem baixa latência e chamadas esporádicas.

O tempo de inicialização (*cold start*) em ambientes serverless varia de acordo com características específicas de cada *runtime*, influenciando diretamente o desempenho e o custo das aplicações. Diante desse cenário, a questão central que este trabalho se propõe a investigar é: **qual das linguagens oficialmente suportadas pelo AWS Lambda apresenta o melhor desempenho em termos de *cold start*?**

Para responder a essa questão, este estudo realiza uma análise comparativa baseada em um benchmark padronizado, utilizando o cálculo do fatorial como função de teste apenas por ser uma operação matemática simples, previsível e suportada em diferentes linguagens. Destaca-se que qualquer outra função controlada poderia ser utilizada, pois o objetivo do trabalho não é analisar o processamento interno, mas sim o comportamento de inicialização da função em si. Com isso, pode-se gerar evidências empíricas que apoiem profissionais de desenvolvimento de software na escolha da linguagem mais eficiente para arquiteturas *serverless*, contribuindo para aplicações mais rápidas, escaláveis e economicamente viáveis.

V. Metodologia

A metodologia deste estudo foi estruturada com o objetivo de realizar uma análise comparativa do desempenho de *cold start* em funções AWS Lambda desenvolvidas em diferentes linguagens de programação. A escolha do AWS Lambda como plataforma de testes se justifica por sua ampla adoção no mercado e por oferecer suporte oficial a múltiplas linguagens, permitindo uma avaliação homogênea em um mesmo ambiente de infraestrutura. Todas as funções foram implantadas na mesma região geográfica (SA East-1, São Paulo) e utilizaram arquitetura *x86_64*, de modo a garantir uniformidade na infraestrutura subjacente. O acesso às funções foi feito por meio da API da AWS, utilizando o SDK oficial para .NET, e as medições foram conduzidas a partir de uma aplicação local desenvolvida em C#.

Para garantir a comparabilidade dos resultados, todas as funções foram configuradas com os mesmos parâmetros operacionais: tempo limite de execução de 30 segundos, ausência de camadas externas (layers) e duas configurações de memória distintas — 128MB e 256MB — para análise do impacto da alocação de recursos sobre o desempenho do *cold start*. Foram selecionadas sete linguagens oficialmente suportadas pela AWS: Java (versão 21), C# (.NET 8), Python (3.13), Node.js

(22), Ruby (3.4), Go e PowerShell. Essas linguagens representam diferentes paradigmas e modelos de execução, como linguagens compiladas, interpretadas, baseadas em máquinas virtuais e orientadas a scripts, o que permite uma avaliação ampla dos efeitos do *cold start* em diferentes contextos.

A arquitetura de testes utilizada neste estudo é ilustrada na Figura 2. Ela mostra a interação entre a aplicação local, responsável por invocar as funções, e os ambientes Lambda remotos. A aplicação envia um payload contendo o número 50 (para cálculo do fatorial), e mede o tempo total de resposta. Cada função também mede seu tempo interno de execução, permitindo a separação entre o tempo de execução do código e o overhead de inicialização.

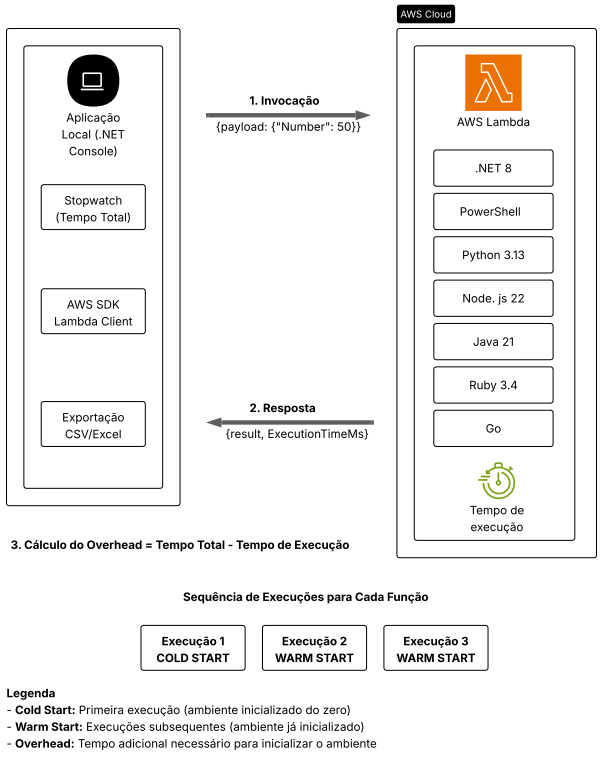

Fig. 2. Arquitetura de testes. A aplicação local invoca cada função Lambda, mede o tempo total e compara com o tempo interno de execução retornado. O overhead (cold start) é obtido pela diferença entre esses tempos. Cada função é executada três vezes: a primeira como cold start, a segunda como warm start (analisada) e a terceira como controle.

A arquitetura de testes consistiu em duas camadas principais: uma aplicação local responsável por invocar as funções Lambda com um payload contendo o número 50 (para cálculo de fatorial), e as próprias funções Lambda, cada uma implementada em uma das linguagens estudadas. A aplicação local media o tempo total da requisição, desde a invocação até o recebimento da resposta, enquanto cada função Lambda media o tempo interno de execução do algoritmo. A diferença entre esses tempos foi utilizada para calcular o overhead de inicialização.

Para garantir que o ambiente fosse realmente reinicializado entre as execuções, adotou-se um intervalo mínimo de 60 segundos entre chamadas consecutivas de funções distintas. Cada função foi executada três vezes em sequência para cada configuração de memória, com o objetivo de reduzir ruídos e validar a consistência dos resultados. A primeira execução foi considerada como *cold start*, representando a inicialização completa do ambiente. A segunda execução, feita imediatamente em seguida, foi tratada como *warm start*, aproveitando o ambiente já alocado. A terceira execução foi utilizada apenas para fins de validação, não sendo incluída nos gráficos comparativos apresentados.

O cálculo do fatorial foi implementado de forma iterativa em todas as linguagens, utilizando bibliotecas nativas para manipulação de grandes números (*BigInteger*). Cada função foi estruturada com um *handler* específico para a AWS Lambda, contendo lógica para o cálculo, medições internas com precisão de milissegundos, e rotinas de serialização e desserialização em formato JSON. Os tempos de execução foram medidos com ferramentas específicas de cada linguagem, como `System.Diagnostics.Stopwatch` para C#, `time.time()` para Python, `performance.now()` para Node.js, `System.nanoTime()` para Java, `Process.clock_gettime()` para Ruby e `time.Since()` para Go.

Todos os resultados foram coletados pela aplicação local e exportados nos formatos CSV e Excel, permitindo análise posterior com ferramentas gráficas e estatísticas. O objetivo principal foi mensurar o impacto do tempo de inicialização sobre a latência total de resposta, isolando o custo do *cold start* em cada ambiente de execução. Dessa forma, os gráficos e análises apresentados no trabalho destacam separadamente o overhead de inicialização *(cold start)* e o tempo real de execução de cada função, permitindo uma comparação completa entre linguagens, configurações de memória e estados de inicialização.

VI. Resultados e Discussões

Os testes de desempenho realizados com funções AWS Lambda revelaram diferenças expressivas no tempo de inicialização entre as linguagens de programação analisadas. A avaliação concentrou-se no tempo de resposta durante o *cold start*, considerando duas configurações de memória: 128MB e 256MB.

A. Análise do Overhead de Cold Start com 128MB

A Figura 3 apresenta os tempos médios de resposta para cada linguagem de programação com 128MB de memória, separando o overhead de inicialização *(cold start)* do tempo real de execução. Observa-se que PowerShell e Java foram os runtimes com overhead mais elevado, com tempos médios de 8.372ms e 7.181ms, respectivamente. Já Go, Python e Node.js apresentaram *overheads* significativamente mais baixos, na ordem de 335ms, 273ms e 407ms, evidenciando menor tempo de provisionamento inicial. O tempo interno de execução foi praticamente desprezível em todas as linguagens, permanecendo abaixo de 1ms na maioria dos casos, exceto PowerShell, que apresentou tempo de execução de aproximadamente 759ms na primeira chamada. Na segunda execução *(warm start)*, o *overhead* foi praticamente eliminado, caindo para valores médios de 50ms ou menos, confirmando que a inicialização só impacta a primeira invocação.

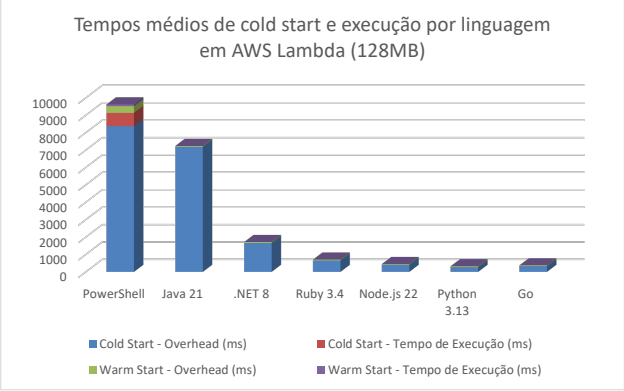

Fig. 3. Tempos médios de inicialização *(cold start)* e execução para funções Lambda em diferentes linguagens, com 128MB de memória. Inclui comparação entre primeira execução *(cold start)* e segunda *(warm start)*.

B. Impacto da Alocação de Memória

A Figura 4 mostra os mesmos tempos de resposta para a configuração com 256MB de memória. O aumento de memória resultou em redução significativa no *overhead* para PowerShell, que caiu de 8.372ms para 4.584ms, uma redução de aproximadamente 45%. Para Java, o *overhead* reduziu de 7.181ms para 6.620ms, uma queda de cerca de 8%, indicando menor sensibilidade ao aumento de memória nesse runtime. Go, Python e Node.js mantiveram overheads baixos mesmo com maior alocação, ficando abaixo de 350ms em média. O tempo de execução interno manteve-se praticamente estável em todas as linguagens, confirmando que o principal ganho está na inicialização do ambiente. Assim como no cenário anterior, o *overhead* foi praticamente eliminado nas execuções subsequentes *warm start*, estabilizando na faixa de 40ms a 76ms.

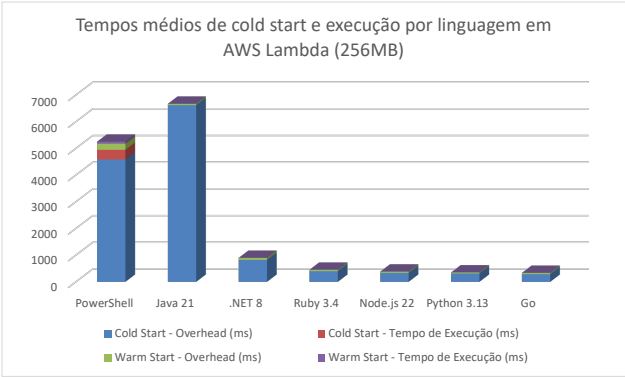

Fig. 4. Tempos médios de inicialização *(cold start)* e execução de funções Lambda para diferentes linguagens, com 256MB de memória. Assim como na configuração de 128MB, o gráfico apresenta a comparação entre *cold start* e *warm start*.

C. Discussão dos Resultados

A análise dos resultados evidencia de forma clara que a linguagem de programação escolhida exerce influência direta e substancial sobre o tempo de inicialização de funções em ambientes *serverless*, sobretudo durante o *cold start*. Essa influência, como demonstrado nos gráficos das Figuras 3 e 4, é ainda mais acentuada quando se utilizam configurações de memória reduzida, como os 128MB iniciais testados. Linguagens como Go, Python e Node.js apresentaram os melhores desempenhos de inicialização. No caso do Go, o excelente resultado provavelmente se deve à sua natureza compilada e à leveza de seu runtime. Como o Go não depende de uma máquina virtual complexa nem de mecanismos de compilação em tempo de execução, seu ambiente de execução é mais enxuto e de carregamento quase imediato. Além disso, a ausência de carregamento dinâmico de bibliotecas em larga escala contribui para um overhead mínimo durante o *cold start*.

O Python, mesmo sendo uma linguagem interpretada, apresentou desempenho competitivo. Um provável motivo é a simplicidade do runtime Python disponibilizado no AWS Lambda, que é altamente otimizado e carrega rapidamente o interpretador necessário. Da mesma forma, o Node.js, baseado no motor V8 da Google, demonstra rapidez por se beneficiar de um runtime altamente performático, com técnicas avançadas de compilação *just-in-time* que, no ambiente da AWS, são otimizadas para execução curta e frequente. Em contrapartida, linguagens como PowerShell e Java apresentaram os maiores tempos de inicialização. Isso pode ser explicado pela complexidade de seus ambientes de execução. No caso do Java,

a necessidade de carregar a JVM (Java Virtual Machine), bem como suas bibliotecas padrão e os processos de otimização do bytecode (*class loading*, verificação e JIT compilation), contribuem significativamente para o tempo elevado de *cold start*. O PowerShell, por sua vez, herda o overhead da plataforma .NET e adiciona camadas de abstração específicas da linguagem, voltadas principalmente para automação de sistemas e não necessariamente otimizadas para execuções efêmeras, como ocorre em ambientes *serverless*.

O aumento da alocação de memória para 256MB reduziu o overhead de forma expressiva em todas as linguagens. Este resultado sugere que o AWS Lambda associa mais poder de CPU proporcionalmente à memória, permitindo uma inicialização mais rápida do runtime e do código do usuário. No caso do Python, por exemplo, o ganho de desempenho foi notável — sua posição relativa no ranking de linguagens melhorou, indicando que seu interpretador é especialmente sensível à disponibilidade de recursos. É plausível afirmar que linguagens com execução baseada em interpretadores se beneficiam mais diretamente de alocações de CPU adicionais, pois a interpretação linha a linha pode ser acelerada significativamente com ciclos de processamento extra.

Outro ponto de interesse é o comportamento mais estável de linguagens como Go e Node.js mesmo com menor alocação de memória. Isso indica que seus ambientes de execução são menos dependentes de ajustes de recursos e já partem de uma base mais enxuta e eficiente. Para sistemas com restrições de orçamento ou arquitetura fortemente orientada à otimização de custo-benefício por invocação, essa característica é altamente desejável.

Por fim, vale destacar que em aplicações com chamadas esporádicas — como APIs de autenticação, processamento de eventos assíncronos e integrações pontuais — o *cold start* tende a ocorrer com frequência, impactando diretamente a experiência do usuário. Nestes cenários, linguagens como Go e Python são mais adequadas. Já em aplicações com chamadas contínuas e cargas constantes — como pipelines de dados ou aplicações com *provisioned concurrency* — linguagens com overhead maior, como Java, podem ser consideradas, desde que acompanhadas de estratégias de mitigação. Soluções como o *Lambda SnapStart*, voltado especificamente para Java, e o *Provisioned Concurrency*, aplicável a todas as linguagens, são alternativas viáveis para mitigar os efeitos negativos do *cold start*, embora impliquem custos adicionais. Estratégias como o uso de invocações periódicas simuladas (*keep-warm*) também podem reduzir a latência percebida, mas com impacto em custo e complexidade de operação.

Em síntese, os resultados destacam que o overhead de inicialização continua sendo o principal fator de latência em funções serverless de execução esporádica. Linguagens como Go, Python e Node.js se mostraram mais consistentes por combinarem overheads baixos com tempos de execução curtos, mantendo o tempo de resposta total dentro de margens aceitáveis mesmo em configurações de menor memória. Por outro lado, linguagens como Java e PowerShell apresentaram overheads significativamente mais altos, que mesmo reduzidos com aumento de memória, ainda impactam o tempo total de resposta. Além disso, o tempo de execução interno, embora pequeno em comparação ao overhead, reforça que a escolha da linguagem e do *runtime* influencia diretamente a eficiência em pipelines e workflows que dependem de múltiplas funções encadeadas. Assim, a análise conjunta do overhead de inicialização e do tempo real de execução oferece uma base mais completa para decisões de projeto em ambientes *serverless*.

VII. Conclusões

Este estudo investigou o impacto do *cold start* em funções AWS Lambda implementadas em sete linguagens de programação distintas: Go, Python, Node.js, .NET (C#), Java, Ruby e PowerShell. A análise foi conduzida sob duas configurações de memória — 128MB e 256MB — com base em um benchmark padronizado de cálculo de fatorial, o que permitiu isolar o comportamento do tempo de inicialização entre os diferentes runtimes.

Os resultados obtidos evidenciaram variações significativas de desempenho entre as linguagens. O Go demonstrou a melhor performance geral em ambos os cenários, seguido por Python e Node.js. Em contrapartida, PowerShell e Java apresentaram os maiores tempos de inicialização, refletindo o impacto de ambientes de execução mais pesados, baseados em máquinas virtuais. O aumento da alocação de memória para 256MB resultou em ganhos substanciais, com reduções de até 45% no tempo de *cold start*, especialmente em linguagens como PowerShell, que são mais sensíveis ao ambiente de inicialização.

Esses achados sugerem que a escolha da linguagem e da configuração de memória não deve ser feita apenas com base em familiaridade ou conveniência, mas também considerando os requisitos de desempenho e latência da aplicação. Para sistemas sensíveis ao tempo de resposta e sujeitos a invocações esporádicas, onde o *cold start* é recorrente, linguagens como Go e Python oferecem vantagens claras. Já em cenários com alta frequência de chamadas, onde predominam os *warm starts*, as diferenças tendem a ser menos expressivas.

Além disso, estratégias complementares como o uso de *Provisioned Concurrency*, o aumento deliberado de alocação de memória, ou mesmo técnicas de *keep-warm* podem ser adotadas para mitigar os efeitos negativos do *cold start* em funções críticas. Dessa forma, a análise conjunta do overhead de inicialização e do tempo real de execução oferece uma

base mais completa para decisões de projeto em ambientes serverless, especialmente em fluxos de trabalho que encadeiam múltiplas funções, como pipelines de processamento de dados e treinamentos de modelos de IA.

Como trabalhos futuros, propõe-se a ampliação do escopo experimental para incluir outros provedores de computação *serverless*, como Azure Functions e Google Cloud Functions, a avaliação do impacto de dependências externas no tempo de inicialização, e a análise da relação entre desempenho, custo operacional e complexidade da função. Tais investigações podem contribuir para um entendimento mais abrangente das variáveis que influenciam a performance em ambientes *serverless*, oferecendo subsídios ainda mais robustos para decisões arquiteturais e de engenharia de software na nuvem.

REFERÊNCIAS

[1] AWS, "Lambda runtimes," 2023, disponível em: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html. Acesso em: 8 jun. 2025.

[2] A. Mohan *et al.*, "Agile cold starts for scalable serverless," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston: USENIX Association, 2019, pp. 477–493.

[3] J. A. Pereira *et al.*, "Application deployment strategies to reduce cold start latency in aws lambda," in *Proceedings of the 15th IEEE International Conference on Cloud Computing (CLOUD)*. Barcelona: IEEE, 2022, pp. 1–10.

[4] P. Castro *et al.*, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[5] I. Baldini *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Singapore: Springer, 2017, pp. 1–20.

[6] E. V. Eyk *et al.*, "The spec cloud group's research vision on faas and serverless architectures," in *Proceedings of the 2nd ACM International Workshop on Serverless Computing*. Zurich: ACM, 2018, pp. 1–4.

[7] J. Schleier-Smith *et al.*, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, vol. 64, no. 5, pp. 76–84, 2021.

[8] L. Wang *et al.*, "Peeking behind the curtains of serverless platforms," in *Proceedings of the USENIX Annual Technical Conference*. Boston: USENIX Association, 2018, pp. 133–146.

[9] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the USENIX Annual Technical Conference*. Boston: USENIX Association, 2020, pp. 205–218.

[10] AWS, "O que é aws lambda?" 2023, disponível em: https://aws.amazon.com/pt/lambda/. Acesso em: 8 jun. 2025.

[11] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software*, vol. 170, p. 110708, 2020.