Comparação de Custo e Performance entre PostgreSQL com Railway, MongoDB Atlas e MySQL com PlanetScale em Arquiteturas *Serverless* em *Cold Start*

João Victor Galdino Ferreira Silva[1], Robson Wagner A. de Medeiros[2]
[1]Universidade Federal Rural de Pernambuco, joao.vgfsilva@ufrpe.br
[2]Universidade Federal Rural de Pernambuco, robson.medeiros@ufrpe.br

Resumo

Este estudo explora a utilização de tecnologias de banco de dados em computação *serverless*, focando em como diferentes estratégias de banco de dados afetam a performance e o custo de funções serverless no AWS Lambda durante cold starts. Este trabalho utiliza PostgreSQL via Railway, MongoDB Atlas, e MySQL com PlanetScale como estudos de caso para investigar essa dinâmica, empregando testes em cenários de *cold start* para quantificar o impacto do tempo de inicialização das diferentes estratégias. Os resultados revelam diferenças significativas na latência de cold start, com o PlanetScale demonstrando uma redução de tempo próxima de 85%. Além disso, analisamos a precificação das soluções, destacando que, embora o PlanetScale se mostre tecnicamente superior, considerações de custo podem levar à seleção de alternativas dependendo do contexto específico de uso. Este trabalho explora essas influências e orienta desenvolvedores na escolha de estratégias que balanceiam desempenho e custo em arquiteturas *serverless*.

Palavras-chave: *Serverless*, *Cold Start*, Performance, Custo, Banco de Dados, PlanetScale, PostgreSQL, MongoDB, AWS, Lambda

Title: A comparison of cost and performance of databases in *serverless* architectures during *cold starts*

Abstract: This study explores the use of database technologies in serverless computing, focusing on how different database strategies affect the performance and cost of serverless functions in AWS Lambda during cold starts. This work uses PostgreSQL via Railway, MongoDB Atlas, and MySQL with PlanetScale as case studies to investigate these dynamics, employing tests in cold start scenarios to quantify the impact of the startup time of different strategies. The results reveal significant differences in cold start latency, with PlanetScale demonstrating a notable advantage in terms of time reduction. Furthermore, we analyzed the pricing of the solutions, highlighting that, although PlanetScale proves to be technically superior, cost considerations may lead to the selection of alternatives depending on the specific context of use. This work explores these influences and guides developers in choosing strategies that balance performance and cost in serverless architectures.

Keywords: Serverless, Cold Start, Performance, Cost, Databases, PlanetScale, PostgreSQL, MongoDB, AWS, Lambda

1. Introdução

Nos últimos anos, as arquiteturas *serverless* têm se destacado como uma inovação significativa no campo do desenvolvimento de software. Essas arquiteturas promovem uma abordagem disruptiva à implementação e gestão de aplicações, oferecendo aos desenvolvedores a liberdade de se concentrarem exclusivamente na lógica e funcionalidades das aplicações, sem a necessidade de gerenciar diretamente a infraestrutura de servidores (Hassan, 2021, 1-2). Este modelo, frequentemente associado à execução de "Funções *Serverless*", permite uma separação clara entre o desenvolvimento de software e a operacionalização de infraestrutura, potencializando uma maior eficiência e agilidade no ciclo de vida do desenvolvimento.

Uma das vantagens mais expressivas das arquiteturas *serverless* é a capacidade de escalabilidade automática. Neste modelo, os recursos computacionais, como containers, são dinamicamente alocados e escalados para atender às demandas em tempo real, podendo expandir-se rapidamente durante picos de uso e reduzir-se a quase nenhum recurso (conhecido como "*scale-to-zero*") em períodos de baixa demanda. Tal flexibilidade não só assegura uma alocação eficiente de recursos, como também estabelece uma estrutura de custo variável, onde os usuários pagam estritamente pelos recursos que efetivamente utilizam. Contudo, apesar desses benefícios, as arquiteturas *serverless* enfrentam o desafio dos *cold starts* — a latência introduzida quando funções inativas são reativadas. Esses atrasos, decorrentes do tempo necessário para inicializar e executar uma função, podem comprometer a experiência do usuário, especialmente em aplicações críticas para o negócio (Hassan, 2021, 9).

Reconhecendo a importância de abordar essa questão, este estudo se aprofunda em um aspecto crítico frequentemente subestimado dos *cold starts*: o desempenho dos bancos de dados nesse contexto. Bancos de dados relacionais tradicionais, como o PostgreSQL (The PostgreSQL Global Development Group, n.d.), dependem de conexões persistentes que, quando interrompidas, necessitam ser restabelecidas, possivelmente exacerbando a latência durante *cold starts*. Em contrapartida, soluções modernas, como Planetscale, que adotam requisições HTTP para a interação com o banco de dados, sugerem uma susceptibilidade menor a essas demoras. Além disso, bancos de dados não relacionais, como MongoDB, apresentam um paradigma alternativo que pode influenciar de maneira distinta o desempenho em cenários de *cold start*.

Com o objetivo de fornecer uma análise comparativa aprofundada, este trabalho visa mensurar e comparar o desempenho e os custos associados a diferentes estratégias de bancos de dados frente aos desafios impostos pelos *cold starts*. Através dessa análise, esperamos oferecer percepções para desenvolvedores e arquitetos de sistemas, facilitando a escolha da tecnologia de banco de dados mais adequada para maximizar a eficiência operacional e a experiência do usuário em arquiteturas *serverless*.

2. Contexto

2.1 *Serverless*

O paradigma *serverless* representa uma evolução significativa na maneira como os desenvolvedores constroem e implantam aplicações, enfatizando a eficiência, escalabilidade e economia de custos. Em essência, "*serverless*" significa que os desenvolvedores podem criar e executar aplicações e serviços sem a necessidade de gerenciar a infraestrutura subjacente de servidores (Baldini, 2017, 3).

Numa arquitetura *serverless*, a unidade de execução é a função, um bloco de código isolado que é executado em resposta a eventos específicos. Estes eventos podem variar desde um pedido HTTP até uma alteração em um banco de dados ou uma nova mensagem em uma fila de processamento. Quando um evento ocorre, o provedor de serviços *serverless* (como AWS Lambda, Google Cloud Functions ou Azure Functions) aloca dinamicamente recursos para executar a função correspondente. O código é executado dentro de um container ou ambiente de execução temporário, que é inicializado com as dependências necessárias do código. Após a execução, os recursos são liberados, tornando-se disponíveis para outras solicitações. Esse paradigma difere fundamentalmente de abordagens tradicionais de infraestrutura que implementam servidores com recursos finitos e definidos que são alocados de antemão e permanecem em utilização a todo momento (Hassan, 2021, 1-2).

Em um ambiente *serverless* quando uma função *serverless* é invocada múltiplas vezes em um intervalo de tempo curto o provedor de serviços cloud pode reutilizar um ambiente de execução previamente inicializado para essa função. Isso significa que o ambiente de execução, incluindo o runtime da linguagem e quaisquer conexões de banco de dados ou recursos externos já estabelecidos, está pronto para executar a função imediatamente, sem a sobrecarga do processo de inicialização, esse processo de reutilização recebe o nome de "*warm start*". *Warm starts* reduzem significativamente a latência de execução da função, proporcionando uma experiência mais rápida e responsiva para o usuário final (Hassan, 2021, 18-19).

Em contraponto ao *warm start*, o *cold start* ocorre quando uma função *serverless* é invocada após um período de inatividade, resultando na necessidade de inicializar um novo ambiente de execução para a função. Esse processo inclui a alocação de recursos necessários, como memória e CPU, carregamento do código da função do armazenamento, inicialização do runtime da linguagem de programação, e execução de inicializações específicas da aplicação. Devido a esses passos adicionais, as invocações de *cold start* experimentam latências significativamente maiores em comparação com *warm starts*. É importante destacar que a existência de *cold starts* é uma consequência direta do conceito de "*scale-to-zero*" do paradigma *serverless* e contribui para sua eficiência de custo, de maneira que, quando uma função *serverless* não está sendo usada, ou seja, não há invocações por um determinado período, o provedor de serviços pode desalocar completamente os recursos associados a essa função, reduzindo a utilização de recursos e consequentemente dos custos para zero

(Beswick, 2021).

2.4 Gerenciamento de Conexões Tradicionais de Bancos de Dados em Arquiteturas *Serverless*

Nas arquiteturas de aplicativos tradicionais, a conexão com bancos de dados é gerenciada de forma persistente: uma vez estabelecida, a conexão pode ser reutilizada para múltiplas consultas e transações. Esta abordagem minimiza o overhead associado ao estabelecimento de novas conexões, mas presume a disponibilidade contínua tanto do aplicativo quanto do banco de dados. No contexto das funções *serverless*, contudo, o gerenciamento de conexões de banco de dados enfrenta desafios únicos devido ao ciclo de vida efêmero das funções e ao modelo de execução orientado a eventos (Beswick, 2021):

- **Inicialização de Conexão**: Cada *cold start* de uma função *serverless* implica a necessidade de estabelecer uma nova conexão com o banco de dados, pois o ambiente de execução anterior, incluindo suas conexões, é desalocado após a função ser inativa. Este processo aumenta a latência, afetando a performance da aplicação, especialmente em bancos de dados relacionais tradicionais que não são otimizados para conexões frequentemente estabelecidas e encerradas.

- **Limites de Conexão**: Bancos de dados tradicionais muitas vezes têm um número máximo de conexões simultâneas. Em um ambiente *serverless*, onde centenas ou milhares de instâncias da função podem ser iniciadas simultaneamente, o risco de exceder esse limite é significativo, podendo resultar em erros de conexão e degradação da disponibilidade do serviço.

- **Gerenciamento de Estado**: Funções *serverless* são stateless, ou seja, não mantêm estado entre invocações. Isso significa que informações sobre conexões de banco de dados não são preservadas, dificultando a reutilização de conexões sem estratégias de gerenciamento de estado específicas.

2.5 O desafio

Bancos de dados tradicionais, sejam eles relacionais como o PostgreSQL ou não relacionais como o MongoDB, foram projetados em uma época em que as aplicações eram hospedadas em servidores que mantinham conexões de longa duração com o banco de dados. Essas conexões TCP/IP persistentes são ideais para ambientes estáveis, onde a latência na inicialização da conexão pode ser amortizada ao longo de muitas operações de leitura e escrita.

No entanto, o modelo *serverless* introduz um paradigma diferente. Funções *serverless* são efêmeras e stateless por natureza, escalando para zero quando não estão em uso e sendo re-instanciadas em resposta a eventos específicos. Este comportamento leva a frequentes *cold starts*, onde cada nova instância da função requer uma nova conexão TCP/IP com o banco de dados. Este processo de estabelecimento de conexão, que inclui o handshake TCP e a autenticação no banco de dados, introduz uma latência adicional significativa, que pode prejudicar a performance da aplicação, especialmente para operações que exigem resposta em

tempo real (Manner, 2018, 2).

Além disso, o modelo de conexão persistente tradicional entra em conflito com a escalabilidade automática das funções *serverless*. À medida que o número de instâncias de função aumenta para atender a picos de demanda, o mesmo acontece com o número de conexões ao banco de dados. Isso pode rapidamente esgotar os limites de conexões simultâneas suportadas pelo banco de dados, levando a erros de conexão e potencialmente a uma falha total do serviço (Baldini, 2017, 14-16).

Neste artigo, propomos uma abordagem ao problema do *cold start*: ao invés de evitar o *cold start*, busca-se compreender e reduzir o próprio tempo de ativação das funções *serverless*. Particularmente, este estudo visa investigar como a escolha de estratégias que não dependem de conexões persistentes com bancos de dados pode impactar o desempenho das funções serverless em cenários de cold start. A hipótese sendo de que, ao adotar abordagens que minimizem a dependência de conexões persistentes, pode-se reduzir o tempo necessário para a ativação das funções, resultando em uma melhoria significativa na latência e, crucialmente, sem os custos adicionais associados à infraestrutura.

3. Revisão bibliográfica

A emergência e rápida adoção de arquiteturas *serverless* no desenvolvimento de software trouxe consigo desafios únicos, dentre os quais os *cold starts* se destacam como um dos mais significativos (Hassan, 2021, 16). Esta revisão bibliográfica visa explorar o espectro de pesquisas existentes relacionadas a arquiteturas *serverless*, com um foco particular nas estratégias para mitigar os efeitos dos *cold starts*.

3.1 Composição de Funções

Uma abordagem inovadora para mitigar o impacto dos *cold starts* em arquiteturas *serverless* é a composição de funções (Tirkey, 2023, 1-5). Esta estratégia envolve a organização e orquestração de múltiplas funções *serverless* menores, que juntas executam as operações necessárias de uma aplicação. Ao invés de implementar uma única função *serverless* grande, responsável por múltiplas tarefas, a composição de funções permite que cada tarefa seja encapsulada em sua própria função, otimizando o tempo de inicialização e melhorando a eficiência geral.

A composição de funções refere-se ao processo de dividir a lógica de aplicativos em várias funções menores e independentes, que podem ser executadas sequencialmente ou em paralelo, dependendo da natureza da tarefa. Cada função é responsável por uma única operação ou um conjunto pequeno de operações, facilitando a manutenção, atualização e escalabilidade da aplicação. A implementação eficaz da composição de funções requer ferramentas e serviços de orquestração, como *AWS Step Functions* ou *Azure Durable Functions*, que gerenciam o fluxo de execução entre funções *serverless*. Estas ferramentas permitem definir explicitamente as dependências entre funções, bem como controlar o fluxo de dados e a execução paralela ou condicional.

Sendo assim a maneira com que a composição de funções mitiga o problema de *cold starts* é reduzindo sua quantidade total enquanto mantém uma escalabilidade granular, onde apenas as funções com demanda aumentada são escaladas, otimizando o uso de recursos e potencialmente reduzindo custos.

3.2 Pré aquecimento

Outra abordagem inclui o uso de redes neurais profundas e LSTM (*Long Short-Term Memory*) para capturar padrões de invocação de funções passadas, visando mitigar a latência de *cold start* e a frequência desses eventos (Kumari, 2022). A rede neural profunda aprende o padrão de invocação para determinar a duração ideal do período de inatividade do container, enquanto o modelo LSTM captura as dependências entre os padrões de requisição de funções e a quantidade de latência de *cold start*, ajudando a determinar o número necessário de containers "quentes" diante de múltiplas solicitações de invocação simultâneas.

Um estudo desenvolveu um modelo adaptativo que opera em duas fases distintas, aplicando dois algoritmos de aprendizado de máquina diferentes (Kumari, 2022). Na primeira fase, utiliza-se uma rede neural profunda para aprender os padrões de invocação de funções anteriores e prever o período ideal de inatividade do container. Na segunda fase, o modelo LSTM é empregado para prever o número possível de solicitações futuras, baseando-se nisso para preparar antecipadamente um número adequado de containers pré-aquecidos, visando reduzir a latência e a frequência de *cold starts*.

Os resultados experimentais validam a eficácia do modelo adaptativo, demonstrando que é possível reduzir tanto a latência quanto a frequência de *cold starts* ajustando dinamicamente a política de gerenciamento de *cold starts* de acordo com o padrão das solicitações. Essa abordagem não apenas diminui o número de *cold starts* mas também melhora significativamente o tempo de resposta das funções *serverless*. Este avanço representa um passo significativo na otimização de ambientes *serverless*, oferecendo uma solução mais eficiente e responsiva para aplicações sensíveis a atrasos.

3.3 Implementação de Retentativa

Uma estratégia explorada foi a de retentativas de funções (Bardsley, 2018, 6). Seu funcionamento mitiga os efeitos de *cold starts* ao realizar múltiplas tentativas de utilizar handlers já alocados enquanto espera por *cold starts*, dessa maneira se no tempo de inicialização desse *cold start* algum recurso de um container já quente for liberado a requisição pode utilizar seus recursos e abandonar a chamada ao container frio.
Essa abordagem requer ajustes cuidadosos nos parâmetros de retentativas para evitar a ativação desnecessária de um grande volume de funções, o que poderia elevar os custos operacionais sem necessidade.

3.4 Outros trabalhos

Os trabalhos anteriormente citados foram os mais relevantes no tema, no entanto há também

outras literaturas relacionadas cujo detalhamento foge do escopo deste trabalho por apresentar resultados e/ou metodologias semelhantes aos detalhados nas seções anteriores mas que serão citados a seguir.

McGrath utilizou uma fila de containers quentes para cada função como forma de pré-aquecimento de funções (McGrath, 2017).

Agarwal utilizou o algoritmo de Q-learning para prever o número necessário de instâncias das funções como forma de pré-aquecimento de funções (Agarwal et al., 2021).

Ferramentas como AWS CloudWatch (*Amazon CloudWatch*, n.d) e Lambda Warmer (*Lambda Warmer*, n.d.) foram utilizadas para manter funções pré-aquecidas em uma frequência predeterminada.

A metodologia de "*Keep-alive"* que mantém as funções sempre quentes também foi utilizada (Lloyd, 2018)

3.5 Conclusão da revisão

A revisão bibliográfica revelou uma diversidade de estratégias desenvolvidas para mitigar o impacto dos *cold starts* em ambientes *serverless* (Bardsley, 2018). Essas abordagens, padrões de design até a adoção de estratégias de aquecimento de funções *serverless*, compartilham um objetivo comum: evitar a ocorrência do estado de container frio e, consequentemente, reduzir a latência inicial percebida pelos usuários finais. No entanto, a maioria dessas estratégias concentra-se em contornar os sintomas associados aos *cold starts*, sem abordar diretamente a raiz do problema – o tempo intrínseco necessário para ativar e configurar novos containers para atender a solicitações de funções.

4. Trabalho Original

O presente trabalho propõe uma investigação de uma abordagem a ser utilizada em conjunto com estratégias de mitigação anteriormente citadas, que não se limita a evitar o estado frio dos containers, mas sim busca reduzir diretamente o tempo necessário para superar os *cold starts*. Ao contrário das estratégias convencionais que procuram manter os containers em um estado quente ou minimizar a necessidade de sua inicialização.

4.2 Estratégias de Banco de Dados Escolhidas

Neste estudo, examinamos três estratégias de banco de dados distintas em termos de arquitetura, desempenho em ambientes *serverless* e comportamento durante *cold starts*. São eles: PostgreSQL com Railway, MongoDB Atlas e MySQL com PlanetScale. Cada um desses sistemas de gerenciamento de banco de dados oferece características únicas que podem influenciar a performance de aplicações *serverless*, especialmente durante o processo de inicialização ou *cold starts*.

PostgreSQL é um sistema de gerenciamento de banco de dados relacional (RDBMS) de código aberto, conhecido por sua robustez, recursos avançados e conformidade com SQL. Ele é amplamente utilizado para uma variedade de aplicações, desde pequenos sistemas a grandes soluções empresariais. No contexto de arquiteturas *serverless*, a principal consideração com PostgreSQL é o gerenciamento de conexões persistentes. *Cold starts* podem ser particularmente desafiadores, pois cada função *serverless* inativa requer a restauração de conexões, potencialmente aumentando a latência (The PostgreSQL Global Development Group, n.d.).

Railway é uma plataforma de infraestrutura que se inspira no Heroku, projetada para tornar as ferramentas de desenvolvimento de aplicativos mais acessíveis para os desenvolvedores e permite que o desenvolvedor escolha e faça a implantação rápida de um banco de dados. Por ter surgido como uma alternativa ao Heroku, Railway tem se tornado uma escolha de baixo custo de infraestrutura de banco de dados (Railway Corporation, n.d.).
MongoDB é um banco de dados NoSQL orientado a documentos, que oferece flexibilidade na modelagem de dados e uma escalabilidade horizontal eficiente. É particularmente adequado para aplicações que requerem armazenamento de grandes volumes de dados não estruturados ou semiestruturados. MongoDB pode ser mais tolerante a ambientes *serverless* devido à sua natureza stateless e suporte para conexões efêmeras. Isso pode reduzir o tempo de *cold start*, já que o estabelecimento de novas conexões é geralmente mais rápido e menos custoso do que em RDBMS tradicionais (MongoDB, Inc., n.d.).

PlanetScale é uma plataforma de banco de dados como serviço (DBaaS) construída sobre Vitess, oferecendo recursos de escalabilidade horizontal sem esforço, com base em MySQL. Ele é projetado para lidar com grandes volumes de tráfego e oferecer alta disponibilidade, sem sacrificar a consistência dos dados. O PlanetScale foi projetado para operar eficientemente em ambientes *serverless*, graças ao seu modelo de conexão stateless e suporte para requisições HTTP. Isso o torna particularmente adequado para cenários com frequentes *cold starts*, pois a sobrecarga de conexão é minimizada (PlanetScale, 2023).

A expectativa com esse estudo é prover uma visão ampla de como a escolha de estratégia de banco de dados influencia na performance e custo de funções *serverless* em situações de *cold start*.

4.3 Metodologia

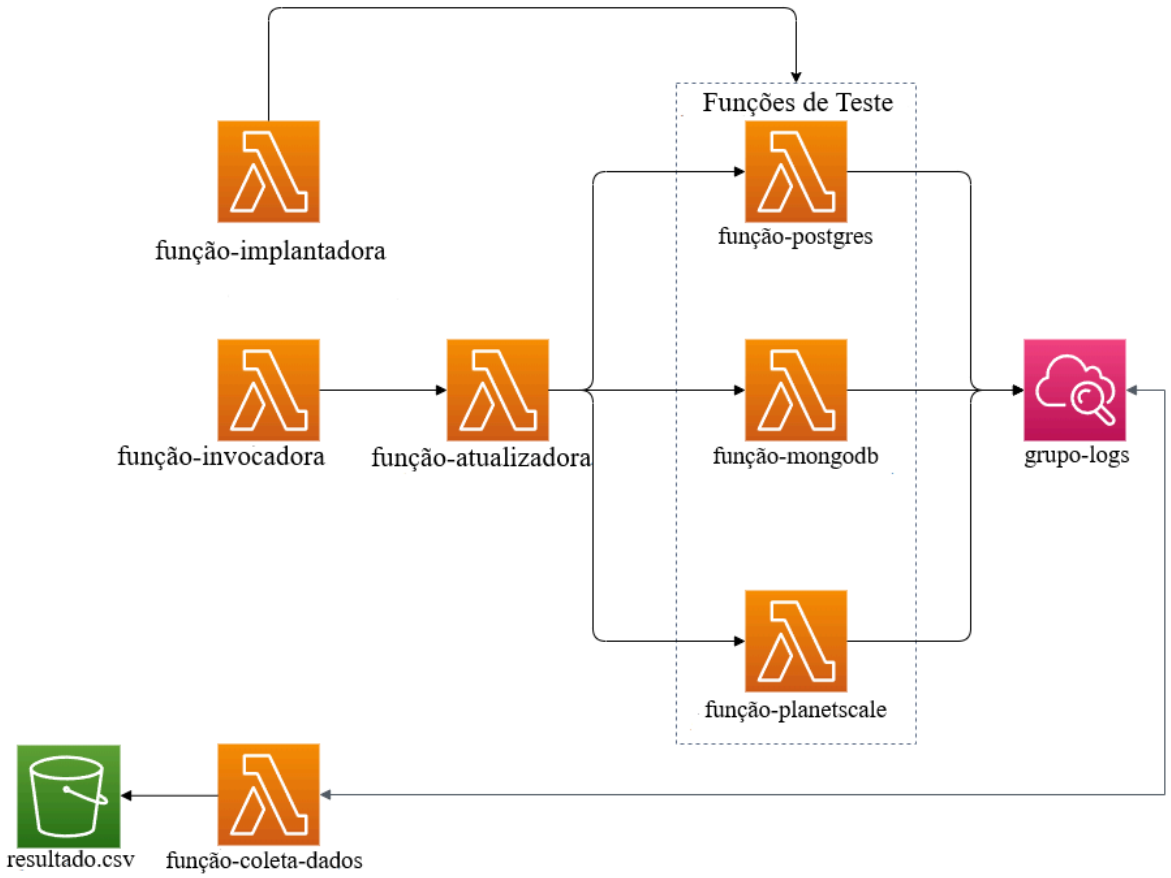

Figura 1. Fluxo da aplicação

Toda a implementação do trabalho foi feita com a linguagem TypeScript (*What Is TypeScript?*, n.d.) e Node (*Node.js — About Node.js®*, n.d.) na versão 18.17.1. Como o objetivo do trabalho é comparar a performance de cada estratégia de maneira isolada inicialmente foram implementadas as 3 funções de teste com as diferentes estratégias de conexão de banco de dados. Cada uma dessas funções requer um conector para que possa se comunicar com o banco escolhido. Para a função de teste com Postgres foi utilizado o conector da biblioteca "pg" (*Pg - Npm*, 2023) por ser a mais utilizada para conexão com bancos PostgreSQL. A instância do banco utilizada foi hospedada separadamente da infraestrutura do projeto na AWS em uma instância do serviço Railway a fim de evitar possíveis otimizações por parte da AWS que poderiam ter uma influência nos resultados. A função do banco MongoDB foi implementada utilizando a biblioteca "mongodb" (*Mongodb - Npm*, 2023), semelhante ao caso do banco Postgres, por ser a mais utilizada. A hospedagem do banco foi feita no serviço MongoDB Atlas. Por fim, a função do Planetscale foi feita com a biblioteca "@planetscale/database" (*@Planetscale/database - Npm*, n.d.) por ser a recomendada pelo próprio serviço e permitir a utilização do protocolo HTTP para as requisições ao banco. Para todas as funções de teste a chamada de conexão com o serviço de banco de dados foi implementada fora do handler da função, esse padrão é importante pois é o que permite que esse recurso seja compartilhado entre múltiplas chamadas da função Lambda (Amazon Web Services, n.d.), simulando um ambiente de produção. Uma vez

implementadas as funções de teste foram desenvolvidas as funções do fluxo de testes. O fluxo se divide em Implantação, Invocação e Coleta de Dados.

Uma vez que o código de cada uma das funções de teste foi implementado é necessário que haja o envio do código para a infraestrutura da AWS no formato de uma função Lambda, para a execução dessa funcionalidade foi desenvolvida a "Função de Implantação". A função de Implantação primeiramente transpila o código de cada uma das funções de teste de TypeScript para JavaScript utilizando a biblioteca "esbuild" (*Esbuild - Npm*, 2024), e realiza a compressão do código em um arquivo em formato ZIP para que possa ser hospedado como uma função Lambda na AWS, e então utiliza o *Software Development Kit* (SDK) da Amazon através da biblioteca "aws-sdk" (*Aws-Sdk - Npm*, 2024) para realizar o envio desse arquivo ZIP de maneira automatizada. As funções Lambda também são conectadas à *Log Groups* da Amazon CloudWatch para que seja feito o registro dos dados das invocações que serão utilizados para comparação.

Uma vez implantadas as funções de teste a função de Invocação é chamada, como o objetivo é testar a performance em *cold start* é necessário que seja possível garantir que durante a invocação das funções testes as Lambdas estejam em estado "frio", para isso a função de Invocação primeiramente utiliza o aws-sdk para realizar uma chamada de atualização das funções de teste modificando uma variável ambiente para um valor aleatório gerado com a biblioteca "uuid", essa atualização força as funções a voltarem ao estado "frio", por fim a função realiza uma requisição para cada uma das funções Lambda de testes.

Por fim a função de Coleta de Dados é chamada, essa função utiliza novamente o aws-sdk para fazer uma chamada para o *Log Group* de cada uma das funções de teste. A função então busca o parâmetro de tempo de inicialização das chamadas, que é o tempo do *cold start*. Os dados são então dispostos em formato de *Comma Separated Value* (CSV) e salvos em um arquivo utilizando o serviço Amazon S3.

4.4 Resultados

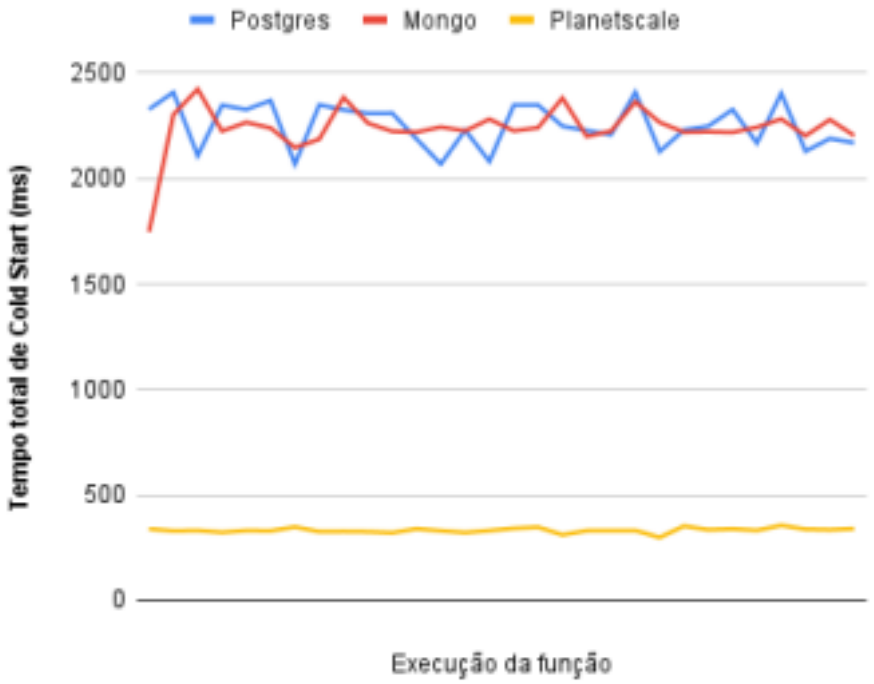

Figura 2. Resultados

Os resultados finais revelaram um tempo médio de *cold start* de 2253 milissegundos para as execuções conectadas ao banco PostgreSQL, 2238 milissegundos para o banco MongoDB, e notavelmente menor, 333 milissegundos para as execuções utilizando o PlanetScale. Esses resultados estão em conformidade com as expectativas iniciais, especialmente considerando que o PlanetScale opera sem a necessidade de estabelecer uma conexão via protocolo TCP, característica que contribui significativamente para a redução do tempo de *cold start* observado.

A análise dos resultados demonstra uma variação considerável nos tempos de *cold start* entre os bancos de dados que dependem de conexões TCP tradicionais (PostgreSQL e MongoDB) e o PlanetScale, que utiliza um modelo baseado em requisições HTTP. Este achado sublinha a influência significativa do método de conexão ao banco de dados no tempo de inicialização de funções *serverless*, destacando a eficiência do PlanetScale em ambientes *serverless*, especialmente em aplicações onde a latência de inicialização é um fator crítico.

Já do ponto de vista de custos, a implementação de funções Lambda não contabiliza o tempo de *cold start* em sua cobrança, portanto a influência do custo se dá aos aspectos de implantação de cada banco de dados. Notavelmente PlanetScale se destaca como a opção mais custosa mesmo em suas opções mais simples, custando cerca de $39 e sem opções disponíveis de tiers grátis na data de escrita deste trabalho. Em contraponto, serviços de bancos de dados relacionais como *Relational Database Service* (RDS) da Amazon e Railway disponibilizam máquinas equivalentes custando cerca de $5 por mês e com tiers grátis. Na vertente do MongoDB o serviço Atlas oferece cobrança por leitura a partir de $0.10 por 1 milhão de leituras ideal para aplicações com leituras intensas.

4.5 Conclusão

Embora os resultados do estudo indiquem claramente o PlanetScale como uma solução eficaz para a redução de tempos de *cold start* em ambientes *serverless* devido à sua implementação com o protocolo HTTP ao invés do tradicional TCP, é crucial considerar que não existe uma "solução única" ideal para todas as aplicações. A escolha da tecnologia de banco de dados adequada depende de uma variedade de fatores, incluindo não apenas a performance, mas também a estrutura de custos associada a cada solução. Neste contexto, a precificação desempenha um papel significativo na determinação da estratégia mais vantajosa.

O PlanetScale, apesar de oferecer vantagens notáveis em termos de redução da latência, é frequentemente associado a um custo mais elevado em comparação com soluções de bancos de dados SQL tradicionais, que geralmente apresentam uma precificação mais acessível. Por outro lado, o MongoDB pode representar uma opção de custo-benefício atrativa, especialmente para aplicações que demandam um volume elevado de operações de leitura, graças às suas opções de precificação favoráveis para tais cenários.

Portanto, a decisão sobre qual tecnologia de banco de dados adotar em uma arquitetura *serverless* deve levar em conta um balanceamento cuidadoso entre desempenho e custo. Em alguns casos, a minimização dos tempos de *cold start* pode ser a prioridade máxima, justificando o investimento em uma solução mais cara como o PlanetScale. Em outras situações, a eficiência de custo pode ser mais crítica, tornando soluções de bancos de dados SQL tradicionais ou o MongoDB opções mais atraentes.

Essa análise multidimensional reforça que a escolha da melhor solução depende das necessidades específicas de cada aplicação, incluindo requisitos de performance, padrões de acesso ao banco de dados e limitações orçamentárias. Assim, é fundamental para desenvolvedores e arquitetos de sistemas realizar uma avaliação criteriosa que considere tanto os benefícios em termos de redução de latência quanto os impactos financeiros das diferentes opções de tecnologia de banco de dados.

Em conclusão, enquanto este estudo fornece percepções valiosas sobre como diferentes tecnologias de banco de dados podem influenciar os tempos de *cold start* em ambientes *serverless*, a escolha da solução ideal requer uma análise cuidadosa que equilibre desempenho e custo. Para trabalhos futuros, seria importante expandir a investigação para incluir análises comparativas mais amplas entre diferentes configurações de bancos de dados e suas implicações em variados cenários de uso *serverless*, contribuindo assim para uma compreensão ainda mais profunda e diretrizes mais refinadas para a comunidade de desenvolvimento.

Referências

Amazon Web Services. (n.d.). *Amazon Lambda*. AWS. Retrieved February 26, 2024, from

https://aws.amazon.com/pt/lambda/

aws-sdk - npm. (2024, February 23). NPM. Retrieved February 26, 2024, from https://www.npmjs.com/package/aws-sdk

Baldini, I. (2017, Junho 12). Serverless Computing: Current Trends and Open Problems.

Bardsley, D. (2018). Serverless performance and optimization strategies. *2018 IEEE International Conference on Smart Cloud*. 10.1109/SmartCloud.2018.00012 Beswick, J. (2021, April 26). *Operating Lambda: Performance optimization – Part 1*. AWS. Retrieved February 23, 2024, from https://aws.amazon.com/pt/blogs/compute/operating-lambda-performance-optimization-part-1/

esbuild - npm. (2024, February 19). NPM. Retrieved February 26, 2024, from https://www.npmjs.com/package/esbuild

Hassan, H. B. (2021). Survey on serverless computing. *Journal of Cloud Computing*.

Kumari, A. (2022). Mitigating Cold-Start Delay using Warm-Start Containers in Serverless Platform. *INDICON 2022 - 2022 IEEE 19th India Council International Conference*. 10.1109/INDICON56171.2022.10040220

Manner, J. (2018). *Cold start influencing factors in function as a service*. MongoDB, Inc. (n.d.). *What is MongoDB Atlas? — MongoDB Atlas*. MongoDB. Retrieved February 26, 2024, from https://www.mongodb.com/docs/atlas/

mongodb - npm. (2023, November 16). NPM. Retrieved February 26, 2024, from https://www.npmjs.com/package/mongodb

Node.js — About Node.js®. (n.d.). Node.js. Retrieved February 26, 2024, from https://nodejs.org/en/about

pg - npm. (2023, August 16). NPM. Retrieved February 26, 2024, from https://www.npmjs.com/package/pg

PlanetScale. (2023, April 5). *What is PlanetScale*. PlanetScale. Retrieved February 23, 2024, from https://planetscale.com/docs/concepts/what-is-planetscale

@planetscale/database - Npm. (n.d.). YouTube: Home. Retrieved February 26, 2024, from https://www.npmjs.com/package/@planetscale/database

The PostgreSQL Global Development Group. (n.d.). *About*. PostgreSQL. Retrieved February 26, 2024, from https://www.postgresql.org/about/

Railway Corporation. (n.d.). *About Railway*. Railway Docs. Retrieved February 26, 2024, from https://docs.railway.app/overview/about-railway

Tirkey, K. (2023). A Novel Function Fusion Approach for Serverless Cold Start. 1-5. 10.1109/IC3S57698.2023.10169477

What is TypeScript? (n.d.). TypeScript: JavaScript With Syntax For Types. Retrieved February 26, 2024, from https://www.typescriptlang.org/

Agarwal, S., Rodriguez, A. r. l. a. t. r. s. f. c. s. f. A., & Buyya, R. (2021). A reinforcement learning approach to reduce serverless function cold start frequency.

McGrath, G. (2017). Serverless Computing: Design, Implementation, and Performance.

Amazon CloudWatch. (n.d.). AWS. Retrieved March 22, 2024, from https://aws.amazon.com/pt/cloudwatch/

Lambda Warmer. (n.d.). GitHub. Retrieved March 22, 2024, from https://github.com/jeremydaly/lambda-warmer

Lloyd, W., Vu, M., Zhang, B., David, O. Leavesley, G. (2018). Improving application migration to serverless computing platforms: Latency mitigation with keep-Alive workloads. Proceedings - 11th IEEE/ACM International

Conference on Utility and Cloud Computing Companion, UCC Companion 2018, 195–200. https://doi.org/10.1109/UCC-Companion.2018.00056