Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação de um Framework FaaS: Funções como serviço

Rogério D. Moreira

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Orientador
Prof.a Dr.a Priscila América Solis

Brasília
2019

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

Dias Moreira, Rogério
Di Implementação de um Framework FaaS: Funções como serviço / Rogério Dias Moreira; orientador Priscila América Solis. -- Brasília, 2019.
100 p.

Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) -- Universidade de Brasília, 2019.

1. Funções como Serviço (FaaS). 2. Computação em Nuvem. 3. Serverless. 4. Arquitetura de Software. I. América Solis, Priscila, orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação de um Framework FaaS: Funções como serviço

Rogério D. Moreira

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof.a Dr.a Priscila América Solis (Orientador)
CIC/UnB

Prof.a Dr.a Mariza Ferro
LNCC

Prof. Dr. Eduardo A. P. Alchieri
CIC/UnB

Prof.a Dr.a Aletéia Patrícia Favacho de Araújo
Coordenadora do Programa de Pós-graduação em Computação Aplicada

Brasília, 05 de julho de 2019

Dedicatória

Dedico este trabalho aos meus amados pais Altamiro e Vera, aos meus queridos irmãos Jadher e Josy e ao meu amor, esposa e companheira Tânia Borges.

Agradecimentos

Agradeço à minha orientadora, a Prof.a Dr.a Priscila Solis pelo apoio e ensinamentos que permitiram a conclusão deste trabalho. Ao SERPRO e a todos os colegas desta empresa que apoiaram este projeto. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Este trabalho descreve uma proposta de *framework* para o problema de partida a frio em Funções como Serviço (*FaaS*). O *framework* proposto tem como principal objetivo reduzir o tempo de execução. Um protótipo foi implementado e avaliado com cinco cenários experimentais diferentes e comparado com uma plataforma comercial, o *FaaS* AWS Lambda da Amazon. Os resultados mostram que o *framework* proposto pode ser considerado uma solução para o problema de partida a frio tendo sua implementação obtido tempos de partida a frio menores que 10 milissegundos. Outro benefício desta proposta foi o aumento no nível de compartilhamento de recursos, sendo que em uma única máquina foi possível executar 30.000 funções de *FaaS*.

Palavras-chave: FaaS, cloud computing, framework, desempenho.

Abstract

This work describes a framework proposal for the cold start problem in Function-as-a-service (FaaS). The proposed framework has the main objective of reducing execution time. A prototype was implemented and evaluated with five different experimental scenarios and compared to a commercial platform, Amazon's AWS Lambda. The results show that the proposed framework can be considered a solution to the cold start problem and its implementation obtained cold start times of less than 10 milliseconds. Another benefit of this proposal was the increase in the level of resource sharing, and in a single machine it was possible to execute 30,000 FaaS functions.

Keywords: FaaS, cloud computing, framework, performance.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API Application Programming Interface.

AWS Amazon Web Services.

FaaS Function as a Service.

IaaS Infraestrutura como serviço.

NIST National Institute of Standards and Technology.

PaaS Plataforma como serviço.

SaaS Software como serviço.

VM Virtual Machine.

VMs Virtual Machines.

Web World Wide Web.

Capítulo 1

Introdução

Devido à necessidade em se ter um ambiente de nuvem governamental, o SERPRO, empresa de tecnologia da informação do Ministério da Economia, implantou no ano de 2017 o Estaleiro que é uma Infraestrutura e Plataforma como Serviços (PaaS) que fornece ao desenvolvedor uma maior liberdade e flexibilidade para desenhar e implementar seus serviços [14].

Esta plataforma é baseada em *container* e após sua implantação, cerca de 1.200 desenvolvedores passaram a promover mais rapidamente e com um menor tempo uma nova versão [14] [15].

Com a consolidação desta plataforma, torna-se essencial explorar novas técnicas focadas em computação em nuvem para a diminuição cada vez maior dos custos operacionais e de implantação de produtos e serviços de tecnologia da informação, sendo que, a proposta apresentada neste trabalho está alinhada com estas necessidades empresariais.

Dentro do contexto de Computação em Nuvem e Borda, o *FaaS* está emergindo como uma nova abordagem para redução de custo operacional e de implantação de sistemas em plataformas comerciais como a *AWS Amazon* e *Microsoft Azure* através de seus serviços chamados respectivamente de: *AWS Lambda* e *Azure Functions* [16]. Esta abordagem implementa uma arquitetura inspirada em micro serviços que é oferecida através de um ambiente de PaaS e sendo uma tecnologia recente, com sua primeira implementação feita pela empresa Amazon em novembro de 2014, ainda possuem diversos desafios a serem superados.

1.1 Definição do Problema

Com o surgimento de um novo paradigma para construção de aplicações chamado de *FaaS* que tem como um dos principais objetivos uma grande redução nos custos financeiros de

infraestrutura, este trabalho propõe a criação de um *framework* de *FaaS* que possa ser utilizado em cenários que necessitam de um baixo tempo de partida a frio.

1.2 Justificativa

Um dos principais desafios na utilização de *FaaS* é o problema da partida a frio que acontece quando uma função a ser executada não possui recursos alocados para sua execução, necessitando assim de um tempo maior já que o tempo gasto para que ela seja instanciada causa impacto no tempo total. Aplicações que exigem baixo tempo de resposta como as de telemedicinas toleram latência máxima de 10 milissegundos [17].

A criação de um *framework* além de ser uma solução para o problema da partida a frio, irá facilitar o desenvolvimento deste tipo de aplicação que pode ser executada tanto na nuvem quanto na borda de forma transparente, escalável e com a mínima intervenção do usuário, características estas definidas como *serverless*.

Um último desafio é a perda de desempenho quando existe uma quantidade grande de comunicação entre as funções. Como a lógica de negócio está bastante fragmentada entre diversas funções ou micro serviços o uso de tráfego de rede pode se tornar intenso trazendo assim uma maior latência para a resposta final ao cliente.

1.3 Objetivos

1.3.1 Objetivo Geral

O principal objetivo deste trabalho é a elaboração de uma proposta de *framework* para solução do problema de partida a frio do *FaaS*. O framework será implementado em um protótipo o qual será avaliado e comparado com outras soluções.

1.3.2 Objetivos Específicos

Os objetivos específicos deste trabalho para atingir o objetivo geral são:

- Implementar uma solução para a partida a frio com tempo de partida médio menor do que 10 milissegundos para atender cenários que possuem baixa tolerância ao tempo de resposta como em telemedicina [17];
- Maximizar o compartilhamento de recursos para minimizar custos financeiros de infraestrutura;
- Facilitar o desenvolvimento de *FaaS*;

- Minimizar o tráfego de rede na transferência de dados quando as funções comunicam-se entre si.

1.4 Estrutura do trabalho

Os próximos capítulos estão estruturados da seguinte forma:

- Capítulo 2 contém uma revisão da literatura onde serão descritos os conceitos e trabalhos relacionados a esta pesquisa;
- Capítulo 3 contém a proposta onde será descrito a arquitetura da solução em maiores detalhes;
- Capítulo 4 descreve os resultados;
- Capítulo 5 apresenta as conclusões e trabalhos futuros.

Capítulo 2

Revisão da literatura

Este capítulo apresenta uma revisão bibliográfica relacionada aos conceitos de computação em nuvem e sua evolução. A Figura 2.1 apresenta uma taxonomia proposta, organizada em camadas dos termos utilizados em referência à evolução e a relação com cada camada. As Seções 2.1, 2.2 e 2.3 abordarão respectivamente cada camada. As Seções 2.4 e 2.5 abordarão os termos *Serverless* e *FaaS* que são transversais a todas as camadas e a Seção 2.6 apresenta os trabalhos de pesquisa mais recentes relacionados ao foco desta pesquisa.

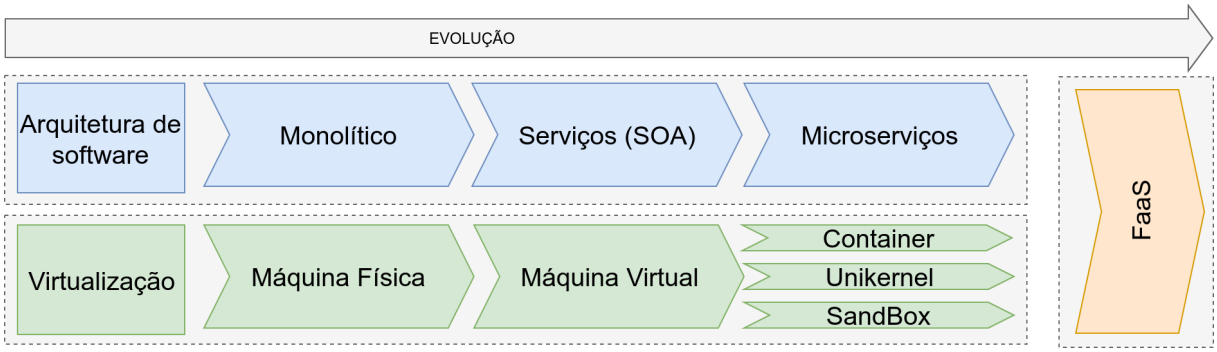

Figura 2.1: Evolução das arquiteturas de software e Virtualização.

2.1 Localização

A localização se refere ao usuário e sua distância em relação à capacidade computacional do ambiente de interesse. A localização pode ser: local, em borda, em nuvem conforme representado na Figura 2.2.

Segundo Chao Li [1] quanto mais próximo do usuário menor é o poder computacional dos dispositivos com menor consumo de energia e quanto mais distante do usuário e

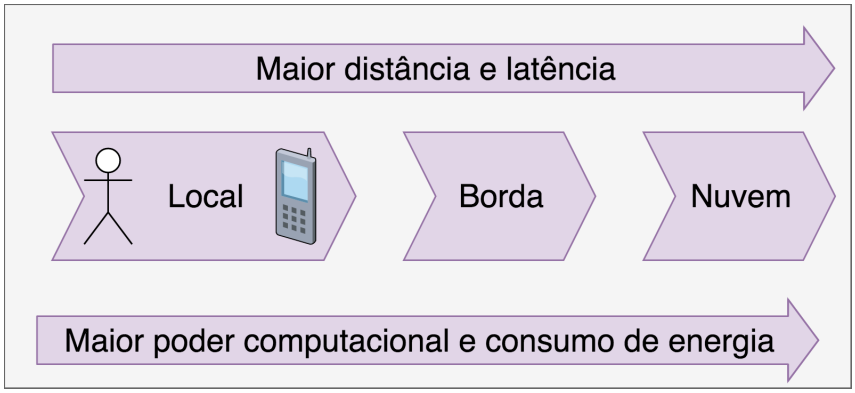

Figura 2.2: Localização (Adaptada [1]).

próximo da nuvem, maior o poder computacional dos dispositivos com maior consumo de energia.

2.1.1 Computação local

A computação local é um ambiente de computação muito próximo do usuário ou ao próprio dispositivo autônomo e pode ser classificada em:

- Computação estacionária: Ambiente de computação pessoal com pouca mobilidade como, por exemplo: estações de trabalho;
- Computação móvel: Ambiente de computação pessoal com muita mobilidade como, por exemplo: celulares e *notebooks*;
- Computação de dispositivos: Ambiente de computação para dispositivos autônomos como, por exemplo: sensores e veículos.

2.1.2 Computação em nuvem

A computação em nuvem é um modelo de computação para acessar, por meio de rede, um *pool* de recursos computacionais compartilhados como, por exemplo: servidores, aplicações, serviços, armazenamento, entre outros, que são rapidamente provisionados e liberados com um pequeno esforço de gerenciamento [18]. Segundo o NIST, este modelo possui as seguintes características [18]:

- Autoatendimento sob demanda: O cliente pode configurar sozinho capacidades computacionais;
- Amplo acesso à rede: Os recursos estão disponíveis por meio de rede e acessados por mecanismos padronizados independentes da plataforma do cliente;

- Agrupamento de recursos: Recursos computacionais como, por exemplo: área de armazenamento, processador, memória e largura de banda da rede são agrupados para servir múltiplos clientes utilizando modelos multilocatário que são providos dinamicamente de acordo com a demanda do cliente;
- Rápida elasticidade: Os recursos podem ser provisionados e liberados rapidamente de forma elástica e em alguns casos de forma automática;
- Medição de serviço: Recursos usados podem ser monitorados, controlados e reportados tanto para o provedor quanto para o cliente.

O NIST [18] divide os tipos de serviços em três modelos:

- *SaaS - Software como serviço*: O recurso provido para o cliente são aplicações que executam na infraestrutura da *cloud* cujo controle e gerenciamento como, por exemplo: servidores, rede, sistema operacional e armazenamento são feitos sem a intervenção do cliente;
- *PaaS - Plataforma como serviço*: O recurso provido para o cliente é a publicação de aplicativos na infraestrutura da *cloud* cujo controle e gerenciamento como, por exemplo: servidores, rede, sistema operacional e armazenamento são feitos sem a intervenção do cliente;
- *IaaS - Infraestrutura como serviço*: O recurso provido para o cliente é o provisionamento de recursos computacionais fundamentais que permite a este publicar qualquer sistema operacional ou aplicação.

Estas categorias de serviços podem ser melhor visualizadas através da arquitetura proposta por Salas [2] ilustrada na Figura 2.3:

- Equipamento: Gerencia recursos físicos da *Cloud* incluindo servidores físicos, roteadores, *switches*, energia e sistemas de resfriamento;
- Infraestrutura: Cria um conjunto de recursos computacionais que podem ser particionados através de tecnologias de virtualização como VMware, Xen e KVM;
- Plataforma: É composta de sistema operacional e *frameworks* de aplicações;
- Aplicação: Aplicações reais que ao contrário das tradicionais podem se beneficiar do escalonamento automático para obter um melhor desempenho, disponibilidade e um menor custo operacional.

Uma das grandes vantagens da computação em nuvem é a redução de custo, escalonamento dinâmico dos recursos computacionais e acesso das informações de qualquer lugar [2].

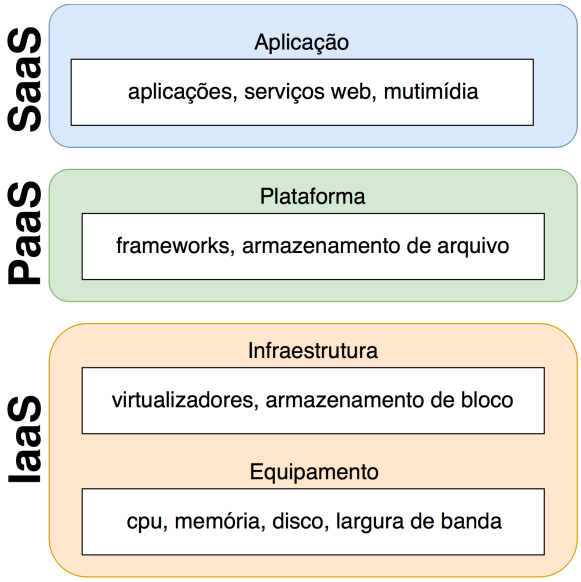

Figura 2.3: Computação em nuvem: Modelo de serviços (Adaptado [2]).

2.1.3 Computação em borda

A computação de borda permite que o processamento seja executado perto da fonte do dado ao invés de ser processado em lugares remotos viabilizando a execução de aplicações que precisam de baixíssimo tempo de resposta ou que necessitem trafegar alto volume de dados como, por exemplo:

- Loja Web Online: O *cache* das páginas Web podem ser feito em servidores da borda reduzindo assim a latência [19];
- Localizar criança desparecida: Devido a grande quantidade de dados que pode ser trafegado por cada dispositivo de câmera em áreas públicas de uma cidade os servidores de borda podem fazer o processamento destes dados diminuindo o custo no uso das redes de dados [19];
- Gerenciamento de tráfego: As tarefas sensíveis ao atraso são executadas em servidores de borda implantados em unidades ao lado da estrada [20];
- Monitoramento de oceano: Milhares de dispositivos de monitoração geram uma grande quantidade de dados que podem ser processados nos servidores de borda para melhorar o tempo de resposta [19].

Como o processamento fica próximo à fonte de dados e o mesmo não precisa ser enviado a um sistema centralizado, a velocidade e o desempenho do transporte de dados podem ser melhorados [21].

Existe um tipo específico de *computação de borda* chamado de *computação de borda móvel* que utiliza equipamentos de borda de redes móveis podendo fazer uso de contexto do usuário como a localização [20].

Os desenvolvedores têm grandes desafios nesta abordagem pela necessidade de particionar as funções da aplicação entre os servidores de borda e de nuvem sendo que os esforços iniciais estão sendo feitos de forma manual como ilustrado na Figura 2.4. Nesta ilustração a aplicação chamada de *App1* pode ser executada nas três camadas: local, em borda e em nuvem, mas a aplicação chamada de *App2* só pode ser executada na camada de computação em borda e em nuvem por possuir requisitos como, por exemplo: alto poder de processamento. Já a aplicação chamada de *App6* só pode ser executada na camada local e em borda por possuir requisitos como, por exemplo: informação de contexto sobre posicionamento de latitude e longitude do dispositivo ou do usuário. As demais aplicações como a *App3*, *App4*, entre outras destacadas pela cor branca, cada uma tem requisitos específicos para ser executada numa única camada.

É necessário o surgimento de *frameworks* e ferramentas para facilitar este tipo de desenvolvimento [19];

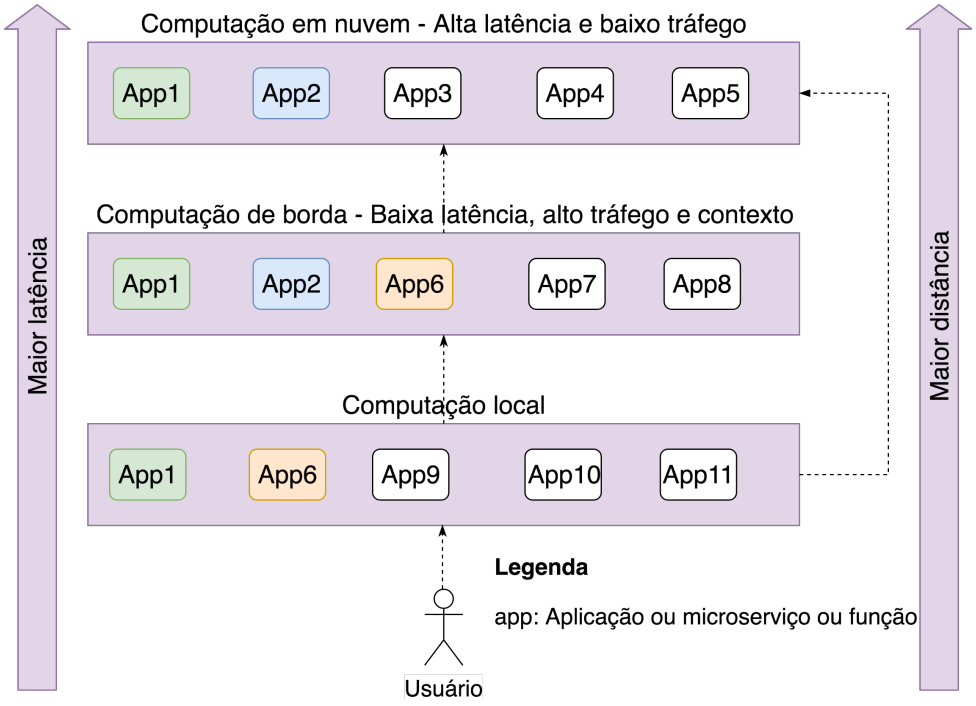

Figura 2.4: Computação de borda - local de execução das aplicações.

Fog Computing (Computação em nevoeiro)

A computação de nevoeiro, um termo criado pela empresa *Cisco* em 2014, é um padrão que implementa *computação de borda* [21]. Segundo o NIST [22], esta arquitetura é implementada em camadas e possui as seguintes características:

- Reconhecimento de localização contextual e baixa latência;
- Distribuição geográfica cuja implantação é amplamente distribuída, mas geograficamente identificável;
- Heterogeneidade de formas de comunicação por meio de rede;
- Interoperabilidade e federação. Os componentes de computação podem interoperar e os serviços devem ser federados entre domínios;
- Interações em tempo real;
- Escalabilidade e agilidade de federar suportando computação elástica.

Fog Node é o principal componente da arquitetura consistindo de máquinas físicas ou virtuais cientes de sua localização que fornecem recursos de computação para dispositivos da borda. Estas máquinas podem ser autônomas comunicando entre si para fornecer serviços ou podem ser federadas para fornecer escalabilidade horizontal e possuem os seguintes atributos:

- Autonomia: Podem operar independentemente, tomando decisões locais, no nível do nó ou do *cluster*;
- Heterogeneidade: Podem ser implantados em uma ampla variedade de ambientes;
- Agrupamento hierárquico: Suportam estruturas hierárquicas com diferentes camadas;
- Gerenciabilidade: São gerenciados e orquestrados por sistemas que realizam a maioria das operações de forma automática;
- Programabilidade: São programáveis em vários níveis como: operadores de rede, especialistas em domínio, fornecedores de equipamentos ou usuários finais.

O NIST [22], assim como na computação em nuvem, divide os tipos de serviços nos seguintes modelos: *SaaS*, *PaaS* e *IaaS*.

2.2 Arquitetura de Software

A arquitetura de *software* ou o modo como se constrói a aplicação pode ser divida em: aplicações monolíticas, aplicações baseadas em serviços ou SOA e aplicações baseadas em microsserviços conforme ilustrado na Figura 2.5.

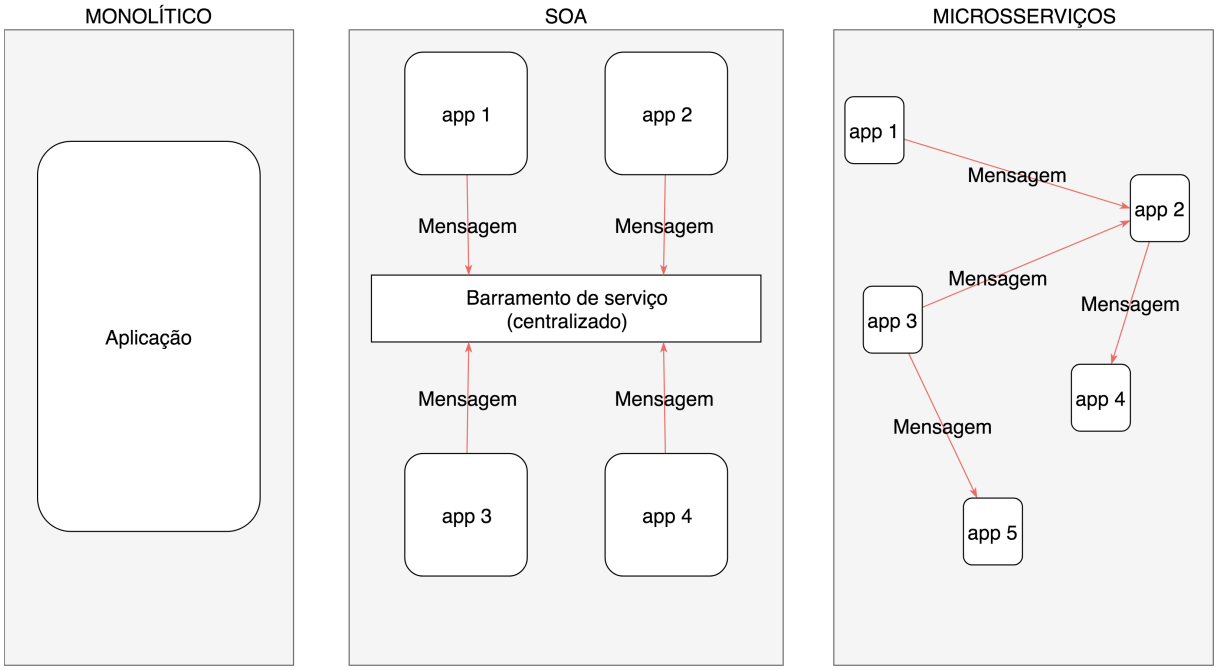

Figura 2.5: Arquiteturas - Comparação (Adaptado [3] [4] [5]).

2.2.1 Aplicações Monolíticas

Aplicações monolíticas são construídas através de uma arquitetura que mantém juntas todas as funcionalidades em uma única aplicação tornando o seu desenvolvimento, teste e publicação simples, porém, o seu tempo de inicialização pode ser lento devido à necessidade em se alocar mais recursos do equipamento como, por exemplo: memória [23].

As aplicações monolíticas podem ser baseadas em componentes cuja construção se dá através da junção de pacotes de *software* independentes que provêm funcionalidades via *interfaces* bem definidas. A grande vantagem deste enfoque é a redução do custo de desenvolvimento através do reuso de funcionalidades já implementadas [4]. Uma grande desvantagem desta abordagem é o aumento no tempo de desenvolvimento de três a cinco vezes o tempo requerido para desenvolver uma unidade de propósito específico já que o foco de um componente é o seu reuso [24].

2.2.2 Aplicações baseadas em serviços - SOA

As aplicações baseadas em serviços devem ter as seguintes características [3]:

- Reutilizáveis: Um mesmo serviço pode ser utilizado em várias partes da aplicação;
- Contrato formal: O cliente que chama o serviço deve obedecer ao contrato ou interface de acesso;
- Baixo acoplamento: Significa uma baixa dependência em relação à implementação;
- Abstração: Os detalhes da implementação não são visíveis para o cliente que chama o serviço;
- Composição: Um serviço pode chamar outro serviço;
- Autônomo: Um serviço independe de um elemento externo para executar sua lógica;
- Sem estado: Um serviço não mantém informações de estado;
- Descoberta: Um serviço deve ser visível para que clientes encontrem-no e possa chamá-los.

A arquitetura SOA pode ser implementada orientada a eventos (*event-driven*) que consistem de componentes independentes e altamente desacoplados que executam tarefas e se comunicam através da emissão e recebimento de eventos, facilitando sua escalabilidade e concorrência. Um desafio encontrado neste tipo de arquitetura é a dificuldade de identificar relações entre os componentes para análise, manutenção e evolução da arquitetura devido à ausência de informações explícitas sobre as dependências de seus componentes [25].

Tilkov [26] descreve que na programação orientada a eventos a aplicação registra interesse em algum evento e quando ele ocorre o sistema de notificação notifica esta aplicação para que ela possa manipular o evento, sendo que uma dificuldade neste tipo de abordagem é que alguns tipos de comunicações não podem ser tratadas pelo sistema de eventos como, por exemplo, o compartilhamento de memória.

2.2.3 Aplicações baseadas em microsserviços

As aplicações baseadas em microsserviços (*microservices*) consistem em pequenas aplicações independentes executadas em processos isolados, distribuídas que são publicadas como serviços comunicando-se por meio de mecanismos leves como o protocolo HTTP [5]; podendo ser escritos em diferentes linguagens de programação e tecnologias de armazenamento e podendo ser publicados de maneira orquestrada e automatizada com um mínimo de gerenciamento centralizado através de práticas *DevOps* [7].

Sam Newman [6] define *Microservices* como serviços autônomos pequenos que trabalham juntos comunicando-se via rede, podendo ser publicados de forma independente sem afetar os serviços dependentes.

Conforme percebido por Fowler [5], Newman [6] e Aderaldo [7] a arquitetura deste tipo de aplicações pode ser ilustrada conforme a figura Figura 2.6. Ao contrário das aplicações monolíticas que mantém junta numa mesma aplicação todas as funcionalidades [23], as aplicações baseadas em micro serviços são decompostas em pequenos serviços independentes.

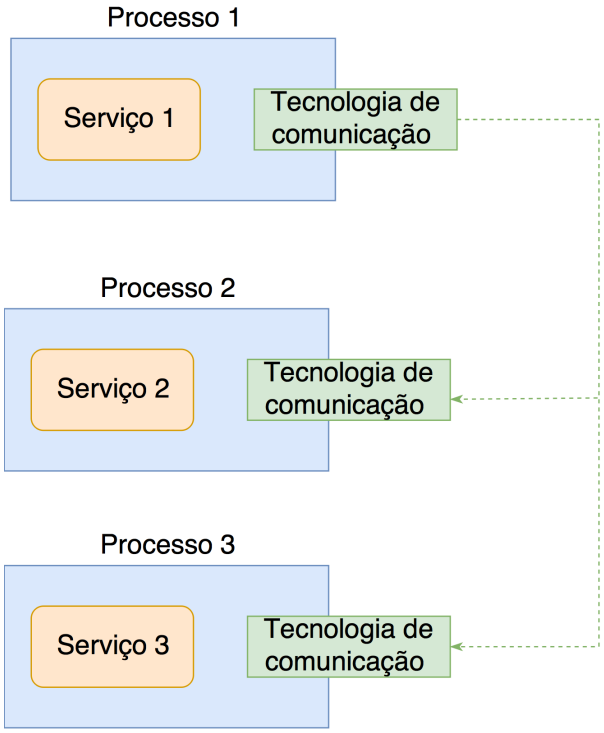

Figura 2.6: Arquitetura de microsserviços (Adaptado [5], [6] [7]).

2.3 Virtualização

A virtualização é o processo de criar uma representação em *software* de alguma coisa física como, por exemplo: servidores, armazenamento, entre outros, sendo uma técnica bastante eficaz para reduzir custos [27].

Esta técnica sofreu várias mudanças nos últimos anos que podem ser divididas em 4 fases [8], conforme ilustrado na Figura 2.7:

- Fase 1: Nenhum compartilhamento de recursos: Cada aplicação executa numa máquina física própria com um Sistema Operacional e *Runtime*;

- Fase 2 - VM: Utilização de Virtual Machines (VMs) que compartilham uma mesma máquina física. Cada Virtual Machine (VM) possui seu próprio Sistema Operacional com um *Runtime* que executa uma aplicação;
- Fase 3 - *Container*: Utilização de *containers* que compartilham uma mesma máquina física e Sistema Operacional. Cada *container* possui seu próprio *Runtime* que executa uma aplicação e oferece uma maior velocidade no provisionamento em relação às VMs;
- Fase 4 - FaaS: Funções como serviço, que compartilham uma mesma máquina física, Sistema Operacional e *Runtime*. Cada FaaS executa sua própria aplicação e oferece uma maior abstração da infraestrutura subjacente.

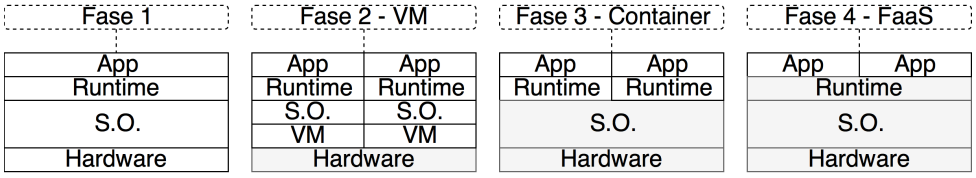

Figura 2.7: Fases da virtualização, compartilhamento em cinza (adaptada [8]).

Abordagens mais leves do que *container* estão sendo exploradas como *unikernel* e *sandbox*, que serão detalhados a seguir.

2.3.1 Unikernel

Unikernel é uma tecnologia de virtualização otimizada que ocupa menos espaço de armazenamento e possui um tempo de inicialização mais baixo do que as tecnologias de *container* e máquinas virtuais. Isto acontece porque nesta abordagem a aplicação e o sistema operacional são únicos e utilizam o mesmo espaço de endereço na memória, sendo que as bibliotecas ao nível do sistema operacional que não são necessárias para que a aplicação funcione não fazem parte da imagem final [9].

Pode-se perceber que o *Unikernel* permite virtualizar uma aplicação para que ela seja um pequeno sistema operacional especializado numa determinada tarefa que a aplicação se propõe a fazer conforme ilustrado na Figura 2.8.

2.3.2 Sandbox

Sandbox é uma técnica leve para isolar a execução de um código de uma aplicação sendo que não existe uma dependência do sistema operacional para sua execução sendo que uma das tecnologias que pode ser utilizada é o *NodeJS* [28].

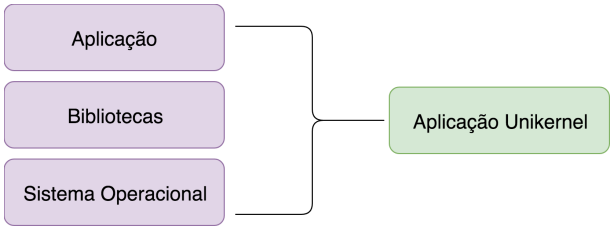

Figura 2.8: Unikernels - Comparação com aplicações tradicionais (adaptada [9]).

Node.js é um ambiente de servidor focado em performance e baixo consumo de memória para a execução de *JavaScript* por meio do *Runtime V8* que suporta um modelo de execução concorrente e não bloqueante, através de mecanismos de *callback* fornecido pela própria linguagem, baseado em eventos de *I/O* assíncronos ao invés do modelo de *multithreading* presente na maioria dos ambientes modernos [26].

Um processo de servidor *Node* possui uma única *thread* que pode servir muitos clientes concorrentemente por meio de um mecanismo de ciclo de eventos que gerencia quando um *callback* que manipula um evento de *I/O* é acionado [26].

O ciclo de eventos é dividido em seis fases [10] como ilustrado na Figura 2.9 onde cada fase possui uma fila *FIFO* contendo *callbacks* a serem executados e quando o *loop* de eventos entra numa fase ele executa todos os *callbacks*, com uma quantidade máxima limitadora, para depois passar para a próxima fase:

- *timers*: Executa *callbacks* agendados pelo *setTimeout()* e *setInterval()*;
- *I/O callbacks*: Executa a maioria dos *callbacks* e faz agendamento de temporizadores e *setImmediate()*;
- *idle, prepare*: Somente usado internamente;
- *poll*: Procura por novos eventos de *I/O*;
- *check*: Invoca *callback* de *setImmediate()*;
- *close callback*: Invoca *callback* de fechamento como, por exemplo recursos de rede.

O *Node.js* possui um gerenciador de pacotes, escritos no padrão *CommonJS*, chamado de *npm* que permite a instalação de bibliotecas e suas dependências [26].

Ao contrário do modelo de uma única *thread*, em aplicações *multithreading* o escalonador do sistema operacional pode alternar o contexto de execução de uma *thread* para outra enquanto elas executam operações de *I/O* como, por exemplo a escrita de dados em *TCP socket*. Este chaveamento entre as *threads* consomem recursos da máquina física como ciclos do processador e memória [26].

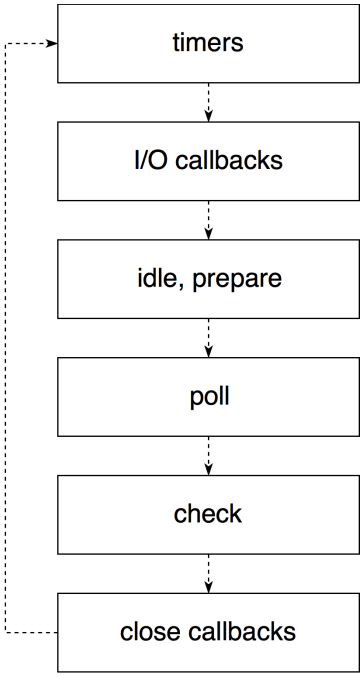

Figura 2.9: Node.js - loop de eventos (Adaptado de [10])).

2.4 Serverless computing

O *Serverless (tradução livre, computação sem servidor)* é um modelo de computação que tem como objetivo abstrair a infraestrutura e serviços da computação em nuvem do usuário para que o mesmo não necessite conhecer e configurar a infraestrutura levando a uma diminuição dos custos operacionais [11] [29].

O modelo de cobrança é por tempo de uso ao invés de recursos reservados [11] e algumas definições como em [30] [16] associam o conceito de *serverless* com FaaS, mas *serverless* é um conceito independente.

Serverless pode ser entendido como uma implementação pura e verdadeira de um modelo de *PaaS* focado numa alta abstração da infraestrutura para o usuário, podendo assim ser utilizado tanto em cenário de *FaaS* quanto em outros cenários envolvendo *PaaS*.

2.5 FaaS

O *FaaS* são pequenos blocos de códigos sem estados com interface de entrada e saída que são executados por reação a eventos de forma assíncrona dentro de uma infraestrutura de *serverless* que provisiona e gerencia a execução das funções de forma transparente ao usuário, sendo que a cobrança é feita somente pelo tempo de execução de cada função não sendo contabilizado o tempo ocioso [8] [11] [29] [30] [16] [31].

No mercado empresarial a plataforma tida como referência na implementação desta arquitetura é a *Amazon AWS Lambda* lançada em dezembro de 2014 que hoje conta com fortes concorrentes como *Microsoft Azure Function*, *Google Cloud Function* e *IBM Bluemix OpenWhisk* [8] [32]. Utilizando o *Amazon AWS Lambda* aplicações que possuem períodos de ociosidade como aplicações típicas Web podem ter seu custo reduzido em até 77,08% em relação ao uso de máquinas virtuais [16] mas, por outro lado aplicações de uso intenso que não ficam ociosas tem seu custo elevado em até três vezes [31]. Nestes provedores quando uma nova instância da função é ativada, o sistema cria um *container* para executar a função em menos de dois segundos, mas apesar da grande velocidade este tempo de espera pode ser inapropriado para aplicações de tempo real. Um grande fator negativo no uso destes provedores de mercado é que a aplicação fica com uma dependência alta em relação ao *Runtime* e serviços oferecidos por cada provedor dificultando a sua portabilidade, sendo que as seguintes características estão presentes em várias destas plataformas [11]:

- Custo: O usuário paga somente pelo tempo e recurso utilizado quando as funções estiverem em execução. A capacidade de escalar para zero instâncias é uma das diferenças chaves nesta plataforma;
- Desempenho e limites: Existe uma variedade de limites, que podem ser configurados, como o número de requisições concorrentes e o máximo de memória e CPU disponíveis para execução da função;
- Linguagem de programação: Suporte a uma variedade de linguagens como Javascript, Java, Python;
- Modelo de programação: Executa uma função principal que possui uma estrutura de dados para a entrada e uma estrutura de dados para a saída;
- Composição: Algumas plataformas oferecem um modo fácil de construir aplicações complexas fazendo composição de funções;
- Implantação: As plataformas buscam facilitar a implantação sendo necessário, na maioria das vezes, fornecer somente o código fonte da função ou um arquivo único que empacota múltiplos outros arquivos necessários para a execução da função;
- Segurança e bilhetagem: As plataformas são *multi-tenant (Tradução livre, multilocatários)* e fornece a bilhetagem individual para cada cliente;
- Monitoramento e depuração: As plataformas suportam mecanismos para gravação de logs e rastreamento;

Uma implementação de arquitetura FaaS típica possui os seguintes componentes como ilustrada na Figura 2.10:

- Sistema de processamento de eventos: Recebe e gerencia eventos através de uma fila;
- Expedição: Recebe um evento de uma fonte, determina que função deve ser invocada, procura por instâncias da função ou cria uma nova caso ela não exista, invoca a função, aguarda a resposta da função, executa tratamento de *logs*, disponibiliza a resposta ao usuário e destrói a função quando ela não é mais necessária;

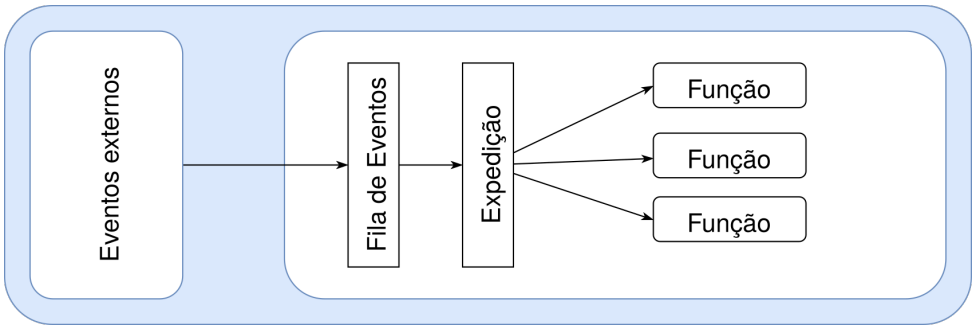

Figura 2.10: Componentes comuns da arquitetura de FaaS (adaptada) [11]).

Este tipo de arquitetura ainda possuem desafios a serem superados como [11]:

- Custo: Minimizar o uso de recursos quando a função estiver em execução ou ociosa;
- Partida a frio: Quando a função não está instanciada, ou seja, não está preparada para execução. Sendo assim, é necessário um tempo específico para instanciar a função com o objetivo de deixá-la disponível ao uso;
- Limite de recursos: Limites de recursos como memória, CPU, tempo de execução são necessários para garantir que a plataforma possa lidar com picos de carga e gerenciar ataques;
- Segurança: O forte isolamento das funções é uma questão crítica devido ao ambiente compartilhado que pode ser utilizado por múltiplos usuários;
- Escalabilidade: A plataforma deve assegurar a escalabilidade das funções provisionando recursos em resposta da carga;

As *FaaS* podem ser inicializadas por vários eventos, por exemplo:

- Eventos gerados pela infraestrutura como: mudança em um banco de dados, submissão de um novo arquivo, mensagens de sistemas [12];
- Diretamente através de chamadas por meio de API [12];

2.6 Trabalhos relacionados

Conforme apresentado na seção anterior, *FaaS* apresenta alguns desafios na sua implementação. Nesta seção serão discutidos vários trabalhos recentes que abordam esta problemática.

Com o objetivo de facilitar a codificação de *FaaS*, a utilização de *framework* foi explorada por McGrath e Brenner [30]. Os autores propuseram um *framwork* dividido em quatro camadas: Serviços *Web*, repositório, mensageria e trabalhadores. A arquitetura adotada como referência foi a de *SOA*, sendo o barramento de serviços implementado pela mensageria que age como uma camada intermediária entre as requisições dos clientes que são recebidas pela camada de serviços *Web* por meio do protocolo *HTTP*. A invocação das funções *FaaS* são feitas por meio da camada de trabalhadores que contém um escalonador construído com a tecnologia *Microsoft .NET* que é executado pelo sistema operacional *Windows Nano Server*. Este escalonador recebe uma mensagem para executar uma função, implementada com *NodeJS*, que está armazenada como uma imagem de volume *Docker* na camada de repositório implementada pelo serviço de nuvem *Azure Blob Storage*. Quando o escalonador cria um *container Docker* para executar uma determinada função pela primeira vez, ele utiliza uma imagem base contendo o *NodeJS* e monta um volume contendo a função a ser executada.

Os autores efetuaram testes de desempenho comparando esta proposta com os serviços de *FaaS* da *Azure*, *Google*, *OpenWhisk* e *AWS* utilizando a quantidade de execuções por segundo como métrica. A proposta teve um melhor desempenho em relação ao demais até o número de treze requisições concorrentes quando começa a degradar seu desempenho.

Uma vantagem desta proposta é a não utilização de uma imagem *Docker* para cada função com o objetivo de evitar a necessidade de uma grande área de armazenamento, sendo que foi utilizada uma técnica para montagem de volume. Uma desvantagem desta proposta é a forte dependência com os serviços de nuvem da *Microsoft Azure* dificultando sua implementação em outros locais.

Uma abordagem utilizando o orquestrador *HyperFlow* foi explorada por Malawski [12] para gerenciar a invocação das funções hospedadas nos provedores *AWS Amazon* e *Google Cloud* num cenário de computação científica que consiste em um alto número de tarefas dependentes. Os autores descrevem dois tipos de arquitetura que pode ser utilizado: *API Gateway* e barramento de serviço. Com o barramento de serviço, é utilizado um servidor de mensagens para receber e gerenciar eventos através de filas que são monitoradas pelo invocador de funções, sendo que neste tipo de arquitetura facilita a execução de eventos assíncronos, aumenta o desacoplamento, mas cria uma complexidade maior na arquitetura. Com o uso de *Gateway* as funções são acionadas de forma síncrona através do protocolo

HTTP, sendo que este tipo de arquitetura é mais simples de ser implementada, mas deixa para o cliente a tarefa de fazer a orquestração das funções.

Como vantagem em relação aos outros trabalhos, é a capacidade de compor várias funções para realizar uma tarefa específica e de ter uma maior independência já que existe um meta invocador de funções que possui adaptadores configurados para cada plataforma utilizada como ilustrado na Figura 2.11.

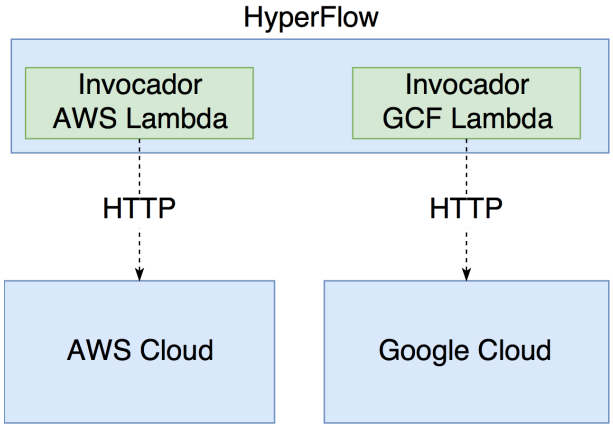

Figura 2.11: Arquitetura HyperFlow (Adaptado de [12]).

Abad [33] propõe um algoritmo de escalonamento centralizado para programas feitos em linguagem *Python* que tenta colocar num mesmo servidor, funções que dependem de um mesmo pacote através do uso de *cache* inteligente. O autor relata que *frameworks* como *OpenWhisk*, *Fission* e *OpenLambda* não trabalham com estratégias de *cache* inteligente, sendo que uma grande vantagem desta abordagem é que ela tenta evitar o tráfego de rede entre as funções já que existe uma tendência delas ficarem num mesmo servidor.

Yin [34] propõe um *framework*, chamado de *Tentacle*, de suporte à decisão para provisionar servidores de borda para provedores de serviços online (OSPs) baseado em proximidade e capacidade com o menor custo. A proximidade é calculada baseado no endereço *IP* do cliente. Uma vantagem deste trabalho é a tentativa de rotear as mensagens considerando a latência de rede que o torna muito relevante para cenários de computação em borda.

Weissman [35] propõe um *framework*, chamado de *Nebula*, contendo componentes de segurança, persistência de dados, coleta de métricas e execução de tarefas. Uma grande vantagem desta abordagem é a tentativa de executar as tarefas o mais próximo da fonte de dados que o torna muito relevante para cenários de computação em borda.

A partida a frio, como visto na seção anterior, ocorre quando uma função tem que ser instanciada na memória o que ocasiona uma perda significativa de desempenho sendo este um dos maiores problemas do *FaaS*. O trabalho de Savolainen [28] mostra-se uma

boa abordagem para endereçar este problema cujo trabalho propõe o uso de *sandbox* de JavaScript para execução de aplicações, chamadas de *Spacelets*, em cenários de espaços inteligentes, sendo que cada *sandbox* é executada dentro de uma única máquina virtual. Apesar do seu foco em cenários de espaços inteligentes esta abordagem é interessante para cenários de *FaaS* já que ela faz uso de *sandbox* ao invés de *container* o que pode ser uma opção mais adequada para problemas de partida a frio.

A utilização de arquitetura de *FaaS* na computação em borda é algo que está começando a ser explorado sendo Krol [36] um dos pioneiros cujo trabalho propõe um *framework* chamado de *NFaaS* que implementa uma arquitetura de *FaaS* em computação de borda utilizando tecnologia *unikernel* ao invés do uso de *container* como o *Docker*. Neste modelo, foi proposto a construção de um repositório, chamado de *Kernel Store*, de *unikernel* a semelhança do *registry* no *Docker* que além de armazenar as funções, consegue tomar decisões sobre roteamento, ou seja, a função pode ser executada nesta máquina ou a mensagem pode ser enviada para ser executada num outro *Kernel Store*. Uma grande vantagem deste trabalho é o uso de *unikernel* ao invés de *container* que permite com que as funções sejam levantadas com mais velocidade ao mesmo tempo, em que elas necessitam de menos espaço de armazenamento no repositório de funções.

2.7 Resumo do capítulo

Neste capítulo foi apresentada uma taxonomia dos conceitos e da evolução da computação em nuvem que ajudam a entender a problemática que será abordada nesta dissertação. Foram apresentados os conceitos sobre computação em nuvem e computação em borda e as arquiteturas comumente utilizadas como SOA e microsserviços além de ter sido descrito os principais conceitos sobre virtualização. O capítulo foi finalizado com a descrição de *FaaS* seguida de uma revisão bibliográfica de trabalhos relacionados a esta pesquisa.

Capítulo 3

Proposta de um Framework para FaaS

Este capítulo apresenta a proposta de um *framework* para *FaaS* em cenários de computação em nuvem e borda.

3.1 Justificativas da arquitetura proposta

O *sandbox* é a técnica utilizada como base nesta arquitetura por ser um método mais leve de virtualização sendo que, no trabalho de Savolainen[28] mostrou-se como uma boa abordagem para aplicações em cenários de espaços inteligentes.

A tecnologia *NodeJS* foi escolhida para implementação desta arquitetura devido a sua maturidade e características tais como: velocidade, economia de recursos e foco em *sandbox* [10] [37].

A organização macro da arquitetura proposta foi inspirada nos trabalhos correlatos e nos principais componentes contidos na arquitetura *Docker* [13] conforme visto na Figura 3.1. Nesta arquitetura o *Registry* é responsável por manter as imagens da aplicação que em tempo de execução são chamadas de *containers*. O responsável pela execução da aplicação é chamado de *Docker deamon*.

O componente de *Gateway*, como uma forma do *Runtime* receber eventos externos, foi inspirado na arquitetura do HyperFlow [12].

Foi selecionada a técnica de compilação e execução sob demanda em memória *RAM* com o objetivo de dar mais flexibilidade e dinamismo ao *Runtime*. Outro benefício é a possibilidade de sua execução em dispositivos físicos com pouco espaço de armazenamento como os utilizados em cenários de *IOT*, sendo assim, é uma técnica interessante para execução em ambiente diversos.

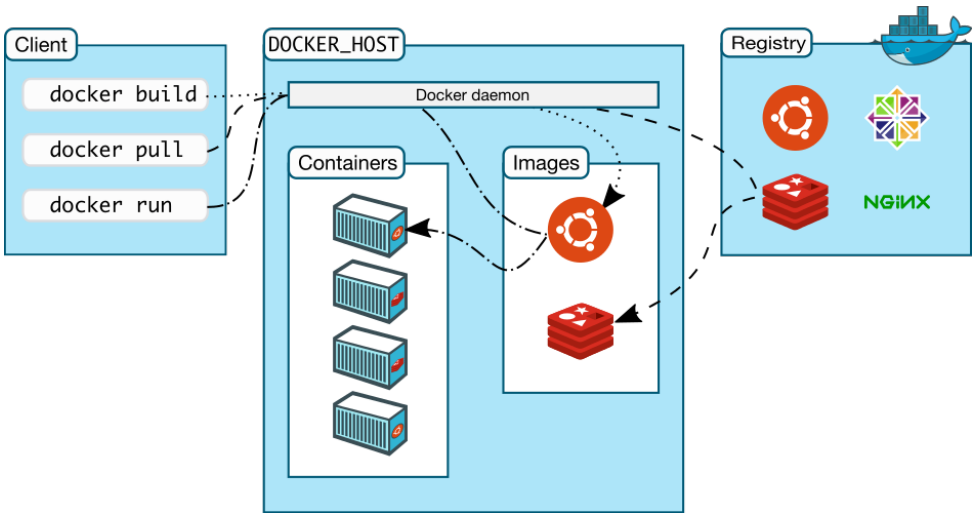

Figura 3.1: Arquitetura Docker [13].

3.2 Visão da arquitetura proposta

A arquitetura proposta possui quatro características:

- Oferecer facilidade no desenvolvimento e implantação de aplicações de *FaaS* para ambientes em nuvem e borda;
- Oferecer baixo tempo de resposta para execução de funções em cenário de partida a frio;
- Oferecer *cache* inteligente para reuso de instâncias das funções;
- Oferecer catálogo automático de funções;

A arquitetura é composta de cinco componentes distribuídos em duas camadas independentes como ilustrado na Figura 3.2. O cliente é responsável por acessar o serviço de *FaaS* disponibilizado pela camada do orquestrador e o Administrador é responsável por gerenciar as funções que serão disponibilizados na camada de repositório.

3.2.1 Repositório - Funções

Este componente contido na camada de repositório será responsável por armazenar as funções e será compatível com a interface *npm* [38], devido ao fato dela ser um padrão de mercado já consolidado. Cada função será empacotada através do formato *tar.gz* e armazenada neste repositório.

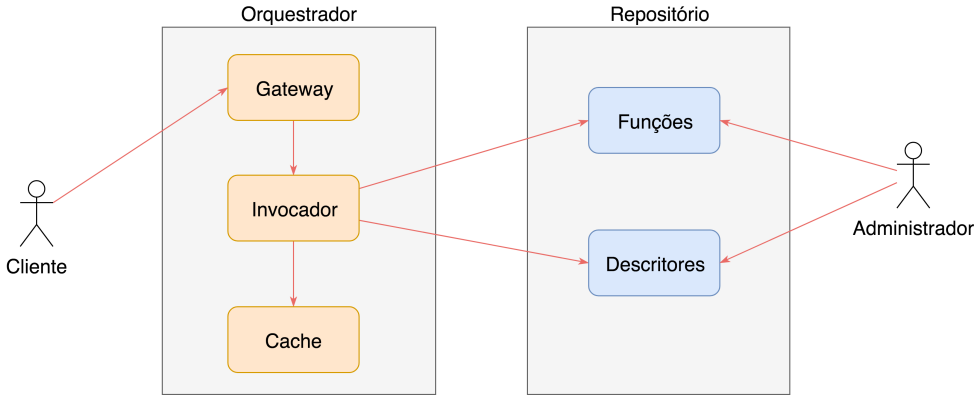

Figura 3.2: Arquitetura em alto nível.

3.2.2 Repositório - Descritores

Este componente contido na camada de repositório será responsável por armazenar os arquivos de informações sobre cada função armazenada seguindo o formato *package.json* que conterá as seguintes informações:

- Nome: Nome que identifica a função;
- Grupo: Grupo lógico a qual a função pertence. Será utilizado para uma melhor organização como, por exemplo: identificação da empresa detentora da função;
- Versão: As funções serão versionadas seguindo o padrão *semver* [39] que utiliza números para identificar: grandes mudanças, pequenas mudança e correções;
- Dependências: Lista de dependência de outras funções necessárias para sua execução;
- Permissão: Informações sobre permissões para um controle mínimo de segurança.

Cada função será identificada unicamente pelo grupo, nome e versão.

3.2.3 Orquestrador - Gateway

Este componente contido na camada do orquestrador será responsável por receber os eventos externos, compatíveis como o protocolo *HTTP* e *HTTPS*, para que seja executado uma função. Após a recepção do evento, este componente irá se comunicar com o componente Invocador e aguardará uma resposta para que seja possível a montagem da mensagem de retorno ao cliente.

3.2.4 Orquestrador - Cache

Este componente contido na camada do orquestrador será o responsável por gerenciar o *cache* em memória *RAM* de funções que já foram executadas. O objetivo deste componente é aumentar o desempenho por minimizar o tráfego de rede como exemplificado na Figura 3.3. Este exemplo foi dividido em três cenários temporais identificados na figura como:

- Orquestrador - Cenário 1: Neste cenário o cliente necessita chamar a Função B e como a mesma não se encontra no *cache* o invocador faz uma chamada via rede para obter o código desta função, aumentando assim seu tempo de resposta;
- Orquestrador - Cenário 2: Neste cenário o cliente necessita chamar a Função B novamente e como a mesma já se encontra no *cache* o invocador não necessita fazer uma chamada via rede, diminuindo assim seu tempo de resposta;
- Orquestrador - Cenário 3: Neste cenário o cliente necessita chamar a Função C que tem como dependência a Função B e como somente a Função B está no *cache* é necessário que o invocador faça uma chamada via rede para obter a Função C, diminuindo o tempo de resposta já que ao invés de fazer duas chamadas via rede, foi feita apenas uma.

É responsabilidade deste componente inferir quando não será mais necessário manter em *cache* determinadas funções para economizar recursos de memória. A política de expiração do *cache* é de arbitragem livre, pois, conforme resultados experimentais a estratégia de *cache* tem pouco impacto no tempo de partida a frio.

3.2.5 Orquestrador - Invocador

Este é o principal componente do orquestrador sendo responsável por:

- Carregar dinamicamente uma função em memória e suas dependências: O componente obtém o arquivo em formato *tar.gz* contendo o código da função a ser processada e executa um processo de descompactação em memória *RAM* para não fazer uso do disco com o objetivo de melhorar o desempenho evitando latência de acesso ao disco. Outra vantagem em se trabalhar somente com a memória *RAM* é uma melhor portabilidade para dispositivos pequenos como sensores que possuem poucos recursos para persistência de dados local;
- Analisar dependências: Analisar todas as dependências necessárias para executar a função e para cada dependência encontrada ela será obtida via rede para ser carregada na memória *RAM* caso a mesma não esteja presente no *cache*. Esta

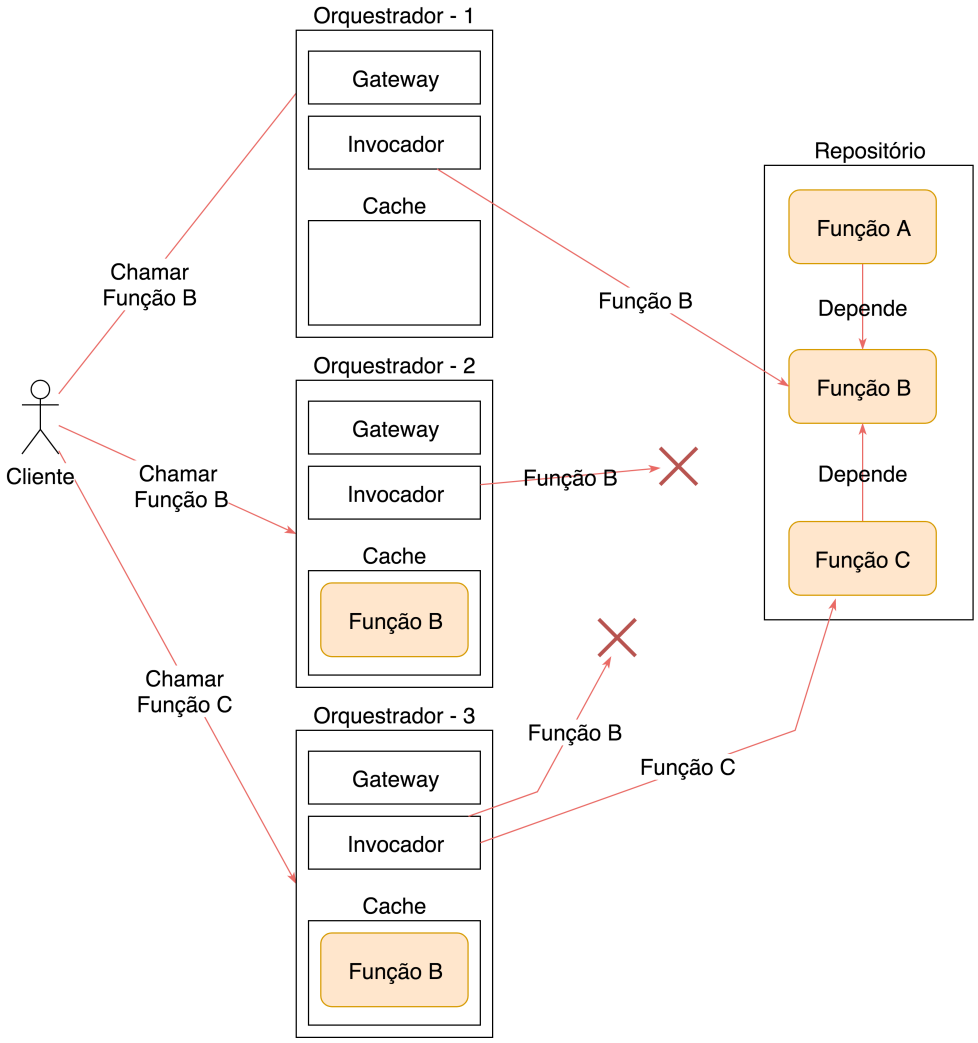

Figura 3.3: Cenários de uso do cache.

etapa será feita de forma recursiva para cada nova dependência encontrada já que elas também podem ter suas próprias dependências;

- Compilação: Compilar o código da função que foi carregado para a memória *RAM* em tempo de execução, técnica conhecida também como compilação sob demanda (*on the fly*);
- *Sandbox*: Executar a função num ambiente isolado e controlado sendo o responsável pela virtualização da função. Este componente se assemelha ao isolamento de processo feito por *container Docker* e foi escolhido devido ao seu bom desempenho em cenários de espaços inteligentes como proposto por Savolainen [28].

3.3 Gerador de carga

Foi implementado um gerador de carga e monitor para o *AWS Lambda* e para o *framework*, conforme o anexo 1 e 2, na tecnologia *NodeJS* que faz chamada as funções *FaaS* via protocolo de transporte *HTTPS* de forma paralela ou serial que é quando a requisição subsequente só é feita após a resposta da requisição anterior.

3.4 Resumo do capítulo

Neste capítulo foi apresentada a proposta de um *framework* para *FaaS* e as justificativas das principais decisões arquiteturais. A arquitetura foi inspirada na arquitetura do *Docker* [13] e no trabalho de Savolainen[28] em cenários de espaços inteligentes.

Capítulo 4

Resultados

Este capítulo tem como objetivo avaliar a proposta descrita através da apresentação de experimentos que serão comparados com os resultados da plataforma de *FaaS Amazon AWS Lambda*. O serviço de FaaS da Amazon foi escolhido por ser um dos principais provedores e o pioneiro neste tipo de plataforma [16]. Além disso, esta plataforma possui os seguintes recursos [40] conforme visto na Tabela 4.1:

4.1 Metodologia

Os experimentos foram executados dentro do ambiente de nuvem da *Amazon AWS* localizado nos Estados Unidos no estado da Virgínia [41], sendo que para o módulo de *Registro* foram também utilizado ambiente de computação externo como ilustrado na Figura 4.1.

Para cada evento o monitor grava as métricas definidas nas tabelas: Tabela 4.2, Tabela 4.3 e Tabela 4.4.

Todos os valores numéricos decimais medidos foram arredondados para números inteiros e para todos os testes foram configurados intervalos de confiança em 99% utilizando distribuição *t de Student* em cenários de testes que possuem de 50.000 amostras coletadas até 150.000.

Tabela 4.1: AWS Lambda - Recursos.

Recurso	**Descrição**
Linguagens de programação	Java, Go, PowerShell, Node.js, C#, Python e Ruby
Escalabilidade	Automática. Máximo de 1.000 funções simultâneas
Orquestração	Possui orquestrador através do produto AWS Step Functions
Segurança	Possui recursos de segurança através do produto AWS Identity and Access Management (IAM)

Tabela 4.2: Parâmetros Simples - Variáveis Quantitativas.

Código	**Nome**	**Unidade**	**Descrição**
S1	Tempo monitor início	Milissegundos	Valor numérico contendo o tempo atual em milissegundos da máquina local onde o monitor está sendo executado no momento anterior a requisição *HTTPS*
S2	Tempo monitor fim	Milissegundos	Valor numérico contendo o tempo atual em milissegundos da máquina local onde o monitor está sendo executado no momento posterior a requisição *HTTPS*
S3	Tempo função	Milissegundos	Valor numérico contendo o tempo atual em milissegundos da máquina local onde a função *FaaS* está sendo executada
S4	Tempo de vida	Segundos	Valor numérico contendo o tempo em segundos desde que a máquina virtual que executa a função *FaaS* foi iniciada.
S5	Memória alocada	Bytes	Valor numérico representando em *bytes* a memória total alocada para o processo do *NodeJS* que executa a função *FaaS*.
S6	Memória total	Bytes	Valor numérico representando em *bytes* o máximo de memória disponível dentro da máquina virtual que executa a função *FaaS*.
S7	Memória livre	Bytes	Valor numérico representando em *bytes* a memória livre disponível dentro da máquina virtual que executa a função *FaaS*
S8	Quantidade CPU	Quantidade	Valor numérico representando a quantidade de *cpu* da máquina virtual que executa a função *FaaS*.

Tabela 4.3: Parâmetros Simples - Variáveis Categóricas.

Código	**Nome**	**Descrição**
S9	ID	Identificador único gerado pela função *FaaS*.
S10	Modelo CPU	Identificador representando o modelo de *cpu* da máquina virtual que executa a função *FaaS*.
S11	Nome da máquina	Identificador representando o nome da máquina virtual que executa a função *FaaS*.
S12	Plataforma	Identificador representando o nome do sistema operacional da máquina virtual que executa a função *FaaS*.
S13	Versao SO	Identificador representando a versão do sistema operacional da máquina virtual que executa a função *FaaS*.

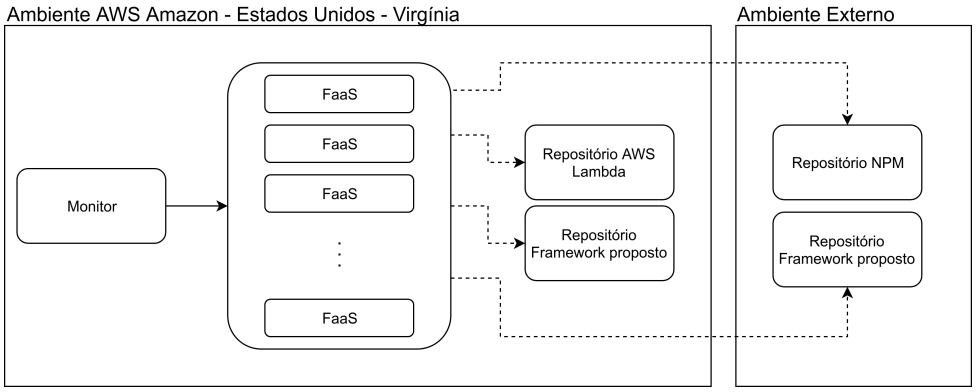

Figura 4.1: Arquitetura geral dos experimentos.

Tabela 4.4: Parâmetros Compostas - Variáveis Quantitativas.

Código	**Nome**	**Unidade**	**Descrição**
C1	Tempo de partida	Milissegundos	Valor numérico contendo a diferença entre as métricas S3 e S1, representando o tempo de partida a frio ou a quente. Fórmula: C1 = S3 - S1.
C2	Tempo de resposta	Milissegundos	Valor numérico contendo a diferença entre as métricas S2 e S1, representando o tempo de resposta de uma partida a frio ou a quente. Fórmula: C2 = S2 - S1.
C3	Latência do Monitor	Milissegundos	Valor numérico contendo a diferença entre as métricas C2 e C1, representando a latência do Monitor em relação ao *Runtime*. Fórmula: C3 = C2 - C1.

Em todos os cenários o monitor e o ambiente principal de execução do *FaaS* estão em um mesmo segmento de rede ou em uma mesma região de computação em nuvem com os relógios das máquinas virtuais sincronizados através do serviço *Amazon Time Sync Service* [42]. A sincronização dos relógios é importante para as medições que foram feitas pois, através deste tempo é obtido a principal métrica que é o tempo da partida a frio.

O fluxo de execução dos testes está exemplificado na Figura 4.2 e descrito na Tabela 4.5.

Cada função *FaaS* é independente possuindo o mesmo código fonte conforme contido no *Apêndice C* desta dissertação.

Tabela 4.5: Fluxo de execução.

Código	**Nome**	**Descrição**
1	Sequência 1	Monitor obtém a métrica (Tempo monitor).
2	Sequência 2	Gerador de carga invoca uma função *FaaS* através do protocolo de transporte *HTTPS*.
3	Sequência 3	Plataforma de *FaaS* recebe a requisição *HTTPS* e instancia um ambiente de execução para a função *FaaS*.
4	Sequência 4	Plataforma de *FaaS* executa a nova função instanciada repassando os dados recebidos pelo protocolo *HTTPS*.
5	Sequência 5	Função *FaaS* que foi executada obtém primeiramente a métrica (Tempo função) e após este processamento obtém as demais métricas.
6	Sequência 6	Função *FaaS* que foi executada obtém as métricas: ID, Tempo de vida, Memória alocada, Memória total, Memória livre, Quantidade CPU, Modelo CPU, Nome da máquina, Plataforma, Versão SO.
7	Sequência 7	Função *FaaS* termina sua execução.
8	Sequência 8	Plataforma de *FaaS* responde a solicitação *HTTPS* proveniente do monitor.
9	Sequência 9	O monitor obtém a métrica (Tempo de resposta).
10	Sequência 10	O monitor grava em um arquivo no disco local as informações de todas as métricas obtidas com o identificador do cenário de teste.

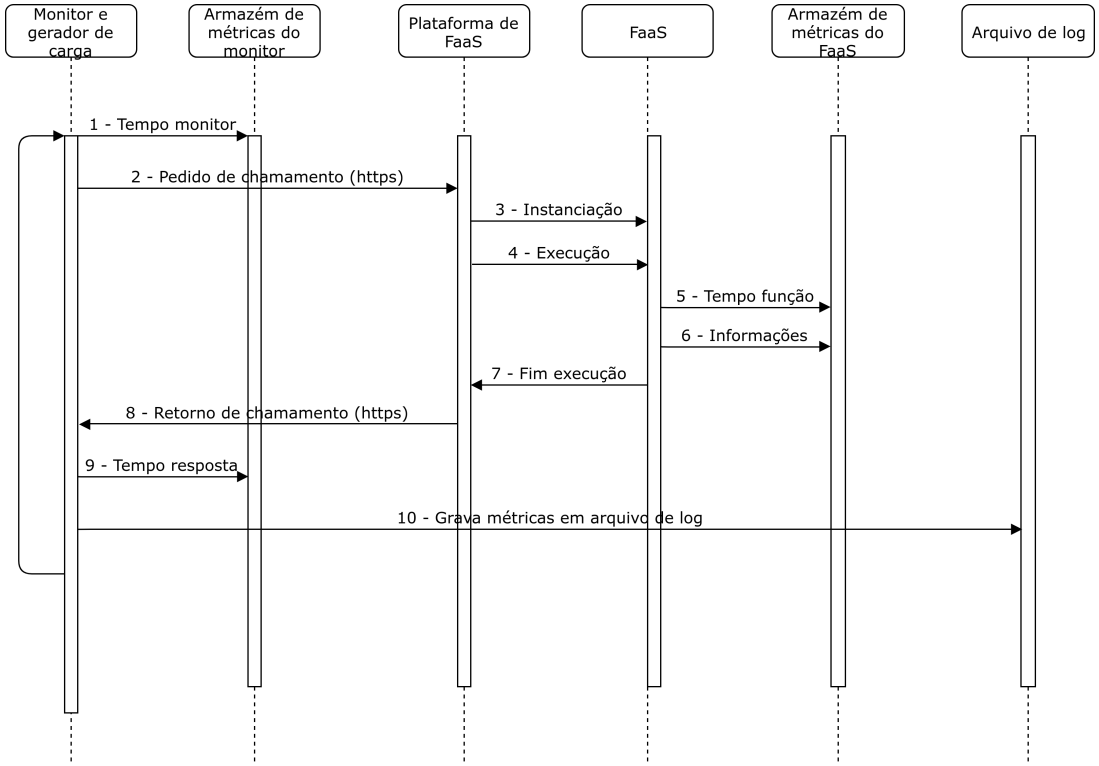

Figura 4.2: Fluxo de sequência dos testes.

4.2 Cenários de avaliação

A avaliação foi dividida em 6 cenários de teste: Cenário Base, Cenário 1, Cenário 2, Cenário 3, Cenário 4 e Cenário 5. As diferenças entre cada um destes cenários está na tecnologia, protocolo de comunicação e localização do módulo de Registro como diagramado na Figura 4.3, sendo que o Cenário Base é utilizado como referência para comparação dos resultados.

Cada cenário de avaliação foi segmentado em baterias que são subconjuntos menores de testes. Esta segmentação é importante para introduzir um pequeno tempo aleatório entre estes subgrupos de testes, além de evitar a alocação de um grande número de máquinas virtuais no ambiente de nuvem da *Amazon AWS*, o que poderia inviabilizar a execução dos experimentos. Após cada bateria de teste é feita uma limpeza no ambiente, após um pequeno tempo aleatório entre 5 segundos e 20 minutos, para garantir os requisitos de partida a frio que as próximas baterias irão testar seguindo as etapas:

- Cenário base;

  É solicitado a plataforma *AWS Lambda* que exclua todas as funções *FaaS* que foram catalogadas e instanciadas para posterior recriação das mesmas.

- Demais cenários: Cenário 1, 2, 3, 4 e 5;

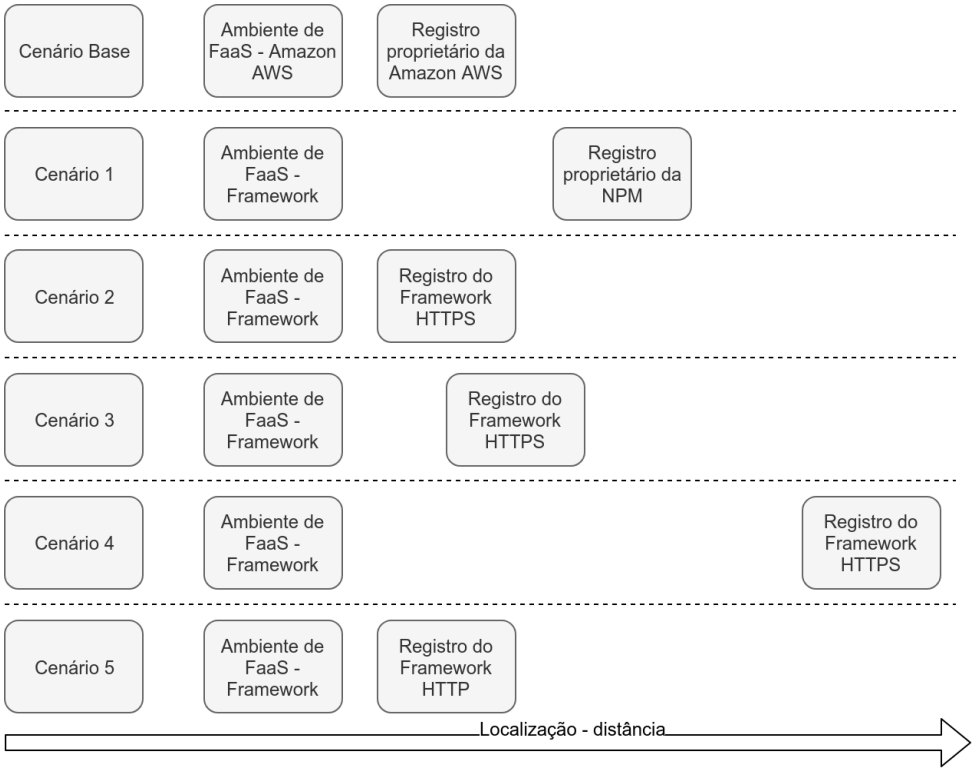

Figura 4.3: Visão dos cenários de teste.

O processo do sistema operacional que executa o *Runtime* do *framework* proposto é reiniciado.

Cada bateria de teste é subdivida em dois grupos:

- Partida a frio: É a primeira sequência de execuções de funções *FaaS* que não estão instanciadas;
- Partida a quente: É a segunda sequência de execuções de funções *FaaS* que foram instanciadas pela partida a frio;

Através da informação de (ID) foi constatado em todos os cenários testados que nenhuma função *FaaS* foi reutilizada de forma indevida, ou seja, para uma única função independente ela só foi utilizada duas vezes: para a partida a frio e para a partida a quente.

O tempo de resposta em todos os cenários foram maiores que o tempo de partida, sendo um bom indício que o serviço de sincronização de relógios provido pela *Amazon AWS* é efetivo. Caso tivesse algum resultado com tempo de resposta menor que o tempo de partida, isto poderia significar que os relógios das *VMs* poderiam ter algum problema de sincronização.

Tabela 4.6: Cenário base - Configuração do ambiente.

Ativo	**Descrição**
Localização do módulo Registro	Este módulo é proprietário da própria *Amazon AWS* sendo executado na mesma região de nuvem do monitor
VM do gerador de carga e monitor	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
VM do *FaaS*	Xeon(R) CPU E5-2666, E5-2676, E5-2680, E5-2686 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-72.76.amzn1.x86_64
Node.js do *FaaS*	Versão 8.10
VM do módulo de Registro	Desconhecido
Protocolo do módulo de Registro	Desconhecido
Limite memória do *FaaS*	128 *megabytes*
Partida a frio - quantidade de *FaaS*	1.000
Partida a frio - total de baterias	50
Partida a frio - total de execuções	50.000
Partida a quente - quantidade de *FaaS*	1.000
Partida a quente - total de baterias	50
Partida a quente - total de execuções	50.000

4.2.1 Cenário Base: AWS Lambda

Este cenário avalia a partida a frio da plataforma de *FaaS* da *Amazon AWS* chamada de *AWS Lambda*. Esta escolha se deve por ela ser uma plataforma de referência para serviços de *FaaS* [8] [32]. O ambiente de teste foi configurado conforme descrição da tabela Tabela 4.6.

Após as 50 baterias de 1.000 funções únicas, foram executadas um total de 50.000 funções de *FaaS* independentes sendo obtidos os resultados mostrados na Tabela 4.7.

Tabela 4.7: Cenário base - Resultado - 1.000 funções únicas

	Mínimo	Máximo	Média	Desvio	Intervalo (99%)
Partida a frio					
Tempo de partida (ms)	33	4918	242	119	240 - 244
Tempo de resposta (ms)	143	5017	284	132	282 - 286
Memória livre (MB)	2703	3528	3089	194	3086 - 3092
Memória alocada (MB)	29	30	30	0	30 - 30
Tempo de vida (s)	28	28570	1421	2988	1386 - 1456
Partida a quente					
Tempo de partida (ms)	1	7319	31	65	30 - 32
Tempo de resposta (ms)	14	7322	34	65	33 - 35
Memória livre (MB)	2698	3525	2796	90	2794 - 2798
Memória alocada (MB)	29	31	30	0	30 - 30
Tempo de vida (s)	66	28740	1554	2974	1519 - 1589
Latência					
Tempo médio (ms)			3		
Sobrecarga					
Tempo de partida			780%		
Tempo de resposta			835%		

Através da informação do nome da máquina, foi constatado que a infraestrutura do serviço *AWS Lambda* criou 1918 instâncias de *VMs* sendo que destas, somente 36 foram reutilizadas. Através da informação de identificação de cada instância *FaaS* juntamente com o nome da máquina foi observado que o número máximo de instâncias *FaaS* alocada para cada *VM* foi de 26, valor este próximo do limite teórico de instâncias *FaaS* por *VM* que é de 30 (3.75 *gigabytes* de uma *VM* / 128 *megabytes* de cada *FaaS*). A diferença de 512 *megabytes* (30 - 26 = 4; 4 * 128 *megabytes* = 512 *megabytes*) é uma memória que pode estar sendo reservada dentro de cada máquina virtual para uso dos sistemas bases da plataforma *AWS Lambda*. Wang [43] observou que cada função *AWS Lambda*, em média, tem um tempo de vida de 6.2 horas mesmo que ela não esteja sendo utilizada sendo que esta informação ajuda a entender o motivo da plataforma ter instanciado 1918 instâncias de máquinas virtuais.

Conforme evidenciado na Tabela 4.7, o consumo médio de memória alocada por cada função foi de 30 *megabytes* enquanto a média de memória livre de cada máquina virtual foi de 3089 *megabytes*, valor este próximo ao esperado já que em média cada *VM* tem 26

instancias de funções *FaaS* alocada (30 megabytes * 26 instâncias = 780 megabytes utilizados, então 3.75 *gigabytes* disponível para cada *VM* - 780 *megabytes* = 3060 *megabytes*).

O total de memória alocada para a execução das 1.000 funções *FaaS* foi de 30.000 *megabytes* pois, em média, foi alocado 30 *megabytes* para cada função e o total de memória reservada foi de 7 *terabytes*, pois, foram instanciadas 1918 máquinas virtuais de 3.75 *gigabytes* cada.

O menor tempo de vida de uma *VM* que executou uma função de *FaaS* foi de 28 segundos o que pode significar que a plataforma de *FaaS* da *Amazon AWS* consegue manter máquinas virtuais prontas para o uso antes delas serem realmente necessárias. Este mesmo comportamento também foi observado por Wang [43] que obteve o valor de 132 segundos em tempo de vida para este mesmo cenário.

A partida a frio demorou em média 242 milissegundos enquanto a partida a quente demorou 31 milissegundos. Para este mesmo cenário, Wang [43] observou o valor de 265 milissegundos para a partida a frio, valor este que é 9.5% maior do que o encontrado neste experimento. Esta diferença pode ser explicada pela diferença das versões do *NodeJS*: nos resultados de Wang [43], foi utilizado a versão 6.10.3 e para este experimento foi utilizada a versão 8.10 que segundo testes feitos pelo fornecedor da tecnologia [37], tem uma performance melhor do que a versão anterior.

A latência média entre o Monitor e o *Runtime* que executa o *FaaS* foi de 3 milissegundos, representando uma baixa sobrecarga em relação ao tempo de resposta.

A sobrecarga entre a partida a frio em relação à partida a quente foi de 780%, ou seja, a partida a frio foi 780% mais lenta do que a partida a quente o que demonstra que a plataforma de *FaaS* da *Amazon AWS* tem problemas com o cenário de partida a frio. Já a sobrecarga do tempo de resposta foi de 835% representando um valor próximo ao do tempo de partida.

4.2.2 Cenário 1: Framework - Módulo Registro - NPM

Este cenário avalia a partida a frio do *framework* de *FaaS* proposto com o módulo de Registro externo *NPM*. Esta escolha se deve ao fato que ele atualmente é a maior plataforma de Registro de *software* em uso [38] tornando-o relevante para que seja avaliado. O protocolo de transporte entre o *Runtime* e o módulo de Registro foi o *HTTPS*, sendo que o ambiente de teste foi configurado conforme descrição da tabela Tabela 4.8.

Após as 50 baterias de 1.000 funções únicas, foram executadas um total de 50.000 funções de *FaaS* independentes numa única máquina virtual sendo obtidos os resultados mostrados na Tabela 4.9.

Tabela 4.8: Cenário 1 - Configuração do ambiente.

Ativo	**Descrição**
Localização do módulo Registro	Este módulo é proprietário da empresa *NPM* sendo acessível de forma pública na internet
VM do gerador de carga e monitor	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
VM do *FaaS*	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
Node.js do *FaaS*	Versão 10.15.1
VM do módulo de Registro	Desconhecido
Protocolo do módulo de Registro	HTTPS
Quantidade máquinas virtuais do *FaaS*	1
Limite memória do *FaaS*	1.7 *gigabytes*
Partida a frio - quantidade de *FaaS*	1.000
Partida a frio - total de baterias	50
Partida a frio - total de execuções	50.000
Partida a quente - quantidade de *FaaS*	1.000
Partida a quente - total de baterias	50
Partida a quente - total de execuções	50.000

Tabela 4.9: Cenário 1 - Resultado - 1.000 funções únicas

	Mínimo	Máximo	Média	Desvio	Intervalo (99%)
Partida a frio					
Tempo de partida (ms)	276	2137	395	99	393 - 397
Tempo de resposta (ms)	278	2139	396	99	394 - 398
Memória livre (MB)	3448	3491	3469	10	3468 - 3470
Memória alocada (MB)	41	69	56	8	55 - 57
Partida a quente					
Tempo de partida (ms)	1	21	2	1	1 - 3
Tempo de resposta (ms)	2	25	3	1	2 - 4
Memória livre (MB)	3437	3461	3448	6	3447 - 3449
Memória alocada (MB)	67	79	75	4	74 - 76
Latência					
Tempo médio (ms)			1		
Sobrecarga					
Tempo de partida			20%		
Tempo de resposta			14%		

Conforme evidenciado na Tabela 4.9 e os gráficos ilustrados na Figura 4.4 e Figura 4.5, a partida a frio demorou em média 395 milissegundos enquanto a partida a quente demorou 2 milissegundos. Comparando com o Cenário Base, a partida a frio foi 39% mais lenta enquanto a partida a quente foi 15 vezes mais rápida. A memória máxima alocada para a execução de 1.000 funções únicas foi de 79 *megabytes* enquanto no Cenário Base foi de 30.000 *megabytes* para as mesmas 1.000 funções.

Devido ao fato da partida a frio ser mais lenta que o Cenário Base, o uso do Registro *npm* não é uma escolha apropriada para atingir os objetivos propostos, mas o fato da partida a quente ter sido mais rápida do que o Cenário Base mostra que o *framework* proposto tem condições de ser uma solução viável desde que o gargalo do módulo de Repositório seja mitigado.

A latência média entre o Monitor e o *Runtime* que executa o *FaaS* foi de 1 milissegundo, representando uma baixa sobrecarga em relação ao tempo de resposta.

A sobrecarga entre a partida a frio em relação à partida a quente foi de 20%, ou seja, a partida a frio foi 20% mais lenta do que a partida a quente o que demonstra que o *framework* utilizando o módulo de Registro *NPM* tem problemas com o cenário de partida a frio. Já a sobrecarga do tempo de resposta foi de 14% representando um valor um pouco distante em relação ao tempo de partida, sendo que esta discrepância se deve ao fato do módulo de Registro estar num ambiente fechado com inexistência de controle ou informação a respeito.

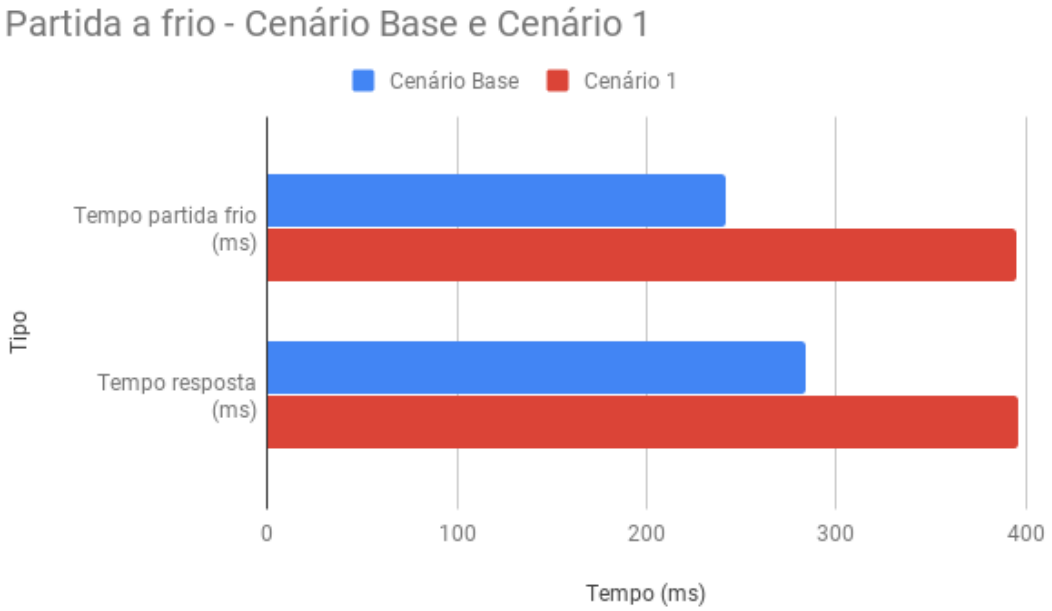

Figura 4.4: Cenário 1 - Partida a frio.

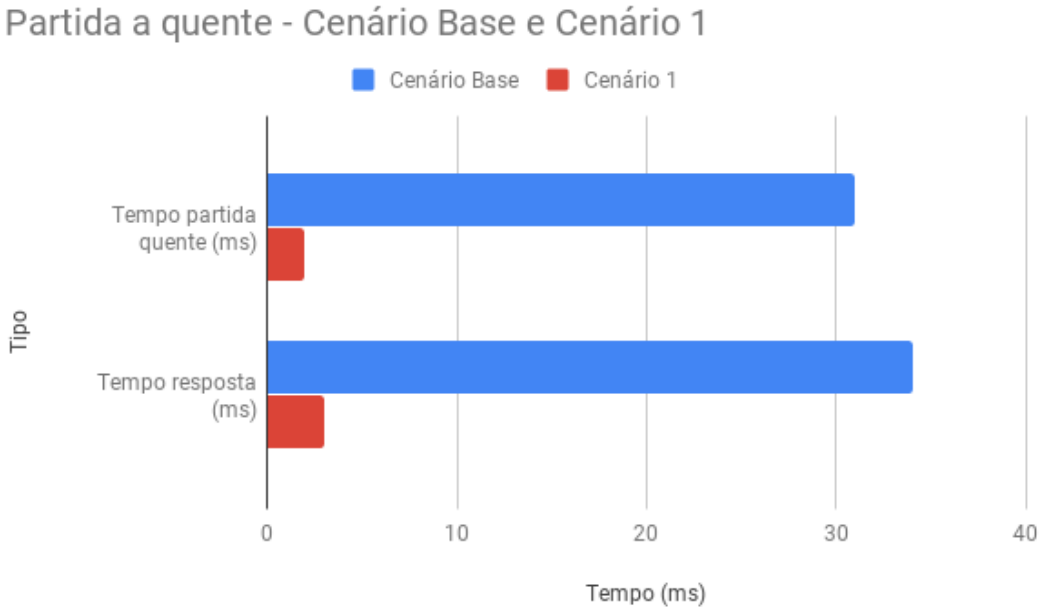

Figura 4.5: Cenário 1 - Partida a quente.

4.2.3 Cenário 2: Framework - Módulo Registro - Mesmo local

Este cenário avalia a partida a frio do *framework* de *FaaS* proposto com o módulo de Registro implementado pelo próprio *framework* que é executado no mesmo centro de dados do módulo de *sandbox*. Uma única *VM* foi utilizada para instanciar as funções de *FaaS*, sendo que foi feito variações no parâmetro de quantidade de funções únicas com o objetivo de verificar a capacidade de compartilhamento de máquinas virtuais da solução proposta. O protocolo de transporte entre o *Runtime* e o módulo de Registro foi o *HTTPS*, sendo que todos os ambientes de teste deste cenário foi configurado conforme descrição da tabela Tabela 4.10.

Os testes foram divididos em seis sub cenários:

- 2A) Executa 1.000 funções *FaaS* únicas;
- 2A-100) Executa 1.000 funções *FaaS* únicas com 100 execuções em paralelo;
- 2A-200) Executa 1.000 funções *FaaS* únicas com 200 execuções em paralelo;
- 2B) Executa 10.000 funções *FaaS* únicas;
- 2C) Executa 20.000 funções *FaaS* únicas;
- 2D) Executa 30.000 funções *FaaS* únicas.

Tabela 4.10: Cenário 2 - Configuração do ambiente de todos sub cenários.

Ativo	**Descrição**
Localização do módulo Registro	Este módulo faz parte do *framework* proposto sendo que o módulo de *sandbox* e o módulo de Registro estão num mesmo centro de dados
VM do gerador de carga e monitor	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
VM do *FaaS*	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
Node.js do *FaaS*	Versão 10.15.1
VM do módulo de Registro	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
Protocolo do módulo de Registro	HTTPS
Quantidade máquinas virtuais do *FaaS*	1
Limite memória do *FaaS*	1.7 *gigabytes*

Tabela 4.11: Cenário 2A - Configuração do ambiente.

Ativo	**Descrição**
Partida a frio - quantidade de *FaaS*	1.000
Partida a frio - total de baterias	50
Partida a frio - total de execuções	50.000
Partida a quente - quantidade de *FaaS*	1.000
Partida a quente - total de baterias	50
Partida a quente - total de execuções	50.000

Cenário 2A

Este cenário executa 1.000 funções *FaaS* únicas cujo ambiente de teste foi configurado conforme tabela Tabela 4.11

Após as 50 baterias de 1.000 funções únicas, foram executadas um total de 50.000 funções de *FaaS* independentes numa única máquina virtual sendo obtidos os resultados mostrados na Tabela 4.12.

Tabela 4.12: Cenário 2A - Resultado - 1.000 funções únicas

	Mínimo	Máximo	Média	Desvio	Intervalo (99%)
Partida a frio					
Tempo de partida (ms)	2	213	3	3	2 - 4
Tempo de resposta (ms)	2	214	4	3	3 - 5
Memória livre (MB)	3446	3460	3456	11	3455 - 3457
Memória alocada (MB)	39	100	69	15	55 - 57
Partida a quente					
Tempo de partida (ms)	1	1004	1	5	0 - 2
Tempo de resposta (ms)	1	1005	2	5	1 - 3
Memória livre (MB)	3383	3402	3395	43	3394 - 3396
Memória alocada (MB)	83	102	98	3	97 - 99
Latência					
Tempo médio (ms)			1		
Sobrecarga					
Tempo de partida			300%		
Tempo de resposta			200%		

Conforme evidenciado na Tabela 4.12 e os gráficos ilustrados na Figura 4.6 e Figura 4.7, a partida a frio demorou em média 3 milissegundos enquanto a partida a quente demorou 1 milissegundo. Comparando com o Cenário Base, a partida a frio foi 80 vezes mais rápida enquanto a partida a quente foi 31 vezes mais rápida evidenciando assim que a solução proposta é viável. Comparando com o tempo de partida a frio do Cenário 1 que teve um tempo de partida 131 vezes maior utilizando o Registro *NPM*, fica evidente que o módulo de Registro é um componente crítico para que se obtenha um baixo tempo na partida a frio já que a única diferença entre estes dois cenários foi a variação deste módulo. A memória máxima alocada para a execução de 1.000 funções únicas foi de 102 *megabytes* enquanto no Cenário Base foi de 30.000 *megabytes* para as mesmas 1.000 funções. Isto significa que este cenário teve um consumo 294 vezes menor de memória alocada do que o Cenário Base.

A latência média entre o Monitor e o *Runtime* que executa o *FaaS* foi de 1 milissegundo, representando uma baixa sobrecarga em relação ao tempo de resposta.

A sobrecarga entre a partida a frio em relação à partida a quente foi de 300%, ou seja, a partida a frio foi 300% mais lenta do que a partida a quente e apesar desta sobrecarga ser somente 260% menor do que o Cenário Base, esta diferença em valor absoluto foi de somente 2 milissegundos enquanto no Cenário Base a diferença foi de 211 milissegundos.

A sobrecarga do tempo de resposta foi de 200% representando um valor próximo ao do tempo de partida.

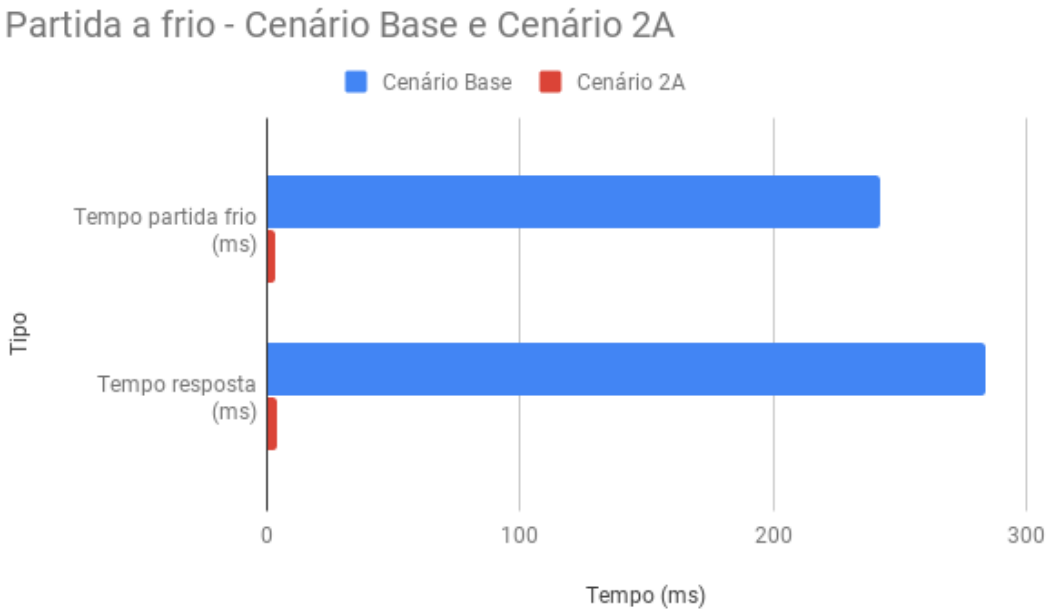

Figura 4.6: Cenário 2A - Partida a frio.

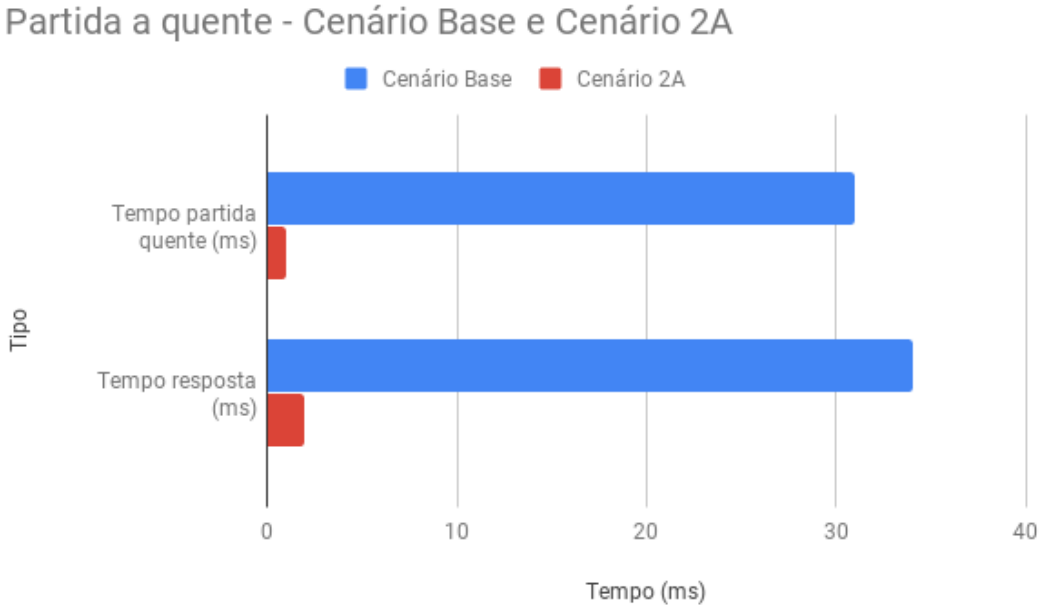

Figura 4.7: Cenário 2A - Partida a quente.

Cenário 2A-100

Este cenário executa os testes do Cenário 2A paralelizando as requisições em grupos de 100, ou seja, através da criação de 100 requisições simultâneas ao invés de execução serial. Após as 50 baterias de 1.000 funções únicas, foram executadas um total de 50.000 funções de *FaaS* independentes numa única máquina virtual sendo obtidos os resultados mostrados na Tabela 4.13.

Tabela 4.13: Cenário 2A-100 - Resultado - 1.000 funções únicas

	Mínimo	Máximo	Média	Desvio	Intervalo (99%)
Partida a frio					
Tempo de partida (ms)	59	1931	340	487	334 - 346
Tempo de resposta (ms)	78	1931	341	487	335 - 347
Memória livre (MB)	3244	3365	3295	25	3294 - 3296
Memória alocada (MB)	51	113	89	17	88 - 90
Partida a quente					
Tempo de partida (ms)	33	1078	84	22	83 - 85
Tempo de resposta (ms)	34	1078	86	22	85 - 87
Memória livre (MB)	3240	3304	3269	18	3268 - 3270
Memória alocada (MB)	106	119	114	2	113 - 115
Latência					
Tempo médio (ms)			2		
Sobrecarga					
Tempo de partida			404%		
Tempo de resposta			396%		

Conforme evidenciado na Tabela 4.13 e os gráficos ilustrados na Figura 4.8 e Figura 4.9, a partida a frio demorou em média 340 milissegundos enquanto a partida a quente demorou 84 milissegundos. Comparando com o Cenário 2A, a partida a frio foi 113 vezes mais lenta enquanto a partida a quente foi 84 vezes, sendo esta proporção de lentidão um resultado esperado já que houve somente uma única instância do *NodeJS* executando as funções *FaaS* que foram gerenciadas internamente através da criação de uma fila de execução. Em relação ao Cenário Base, a partida a frio foi 42% mais lenta e a partida a quente foi somente 3% mais lenta.

A memória máxima alocada para a execução de 1.000 funções únicas foi de 119 *megabytes* enquanto no Cenário 2A foi de 102 *megabytes* para as mesmas 1.000 funções.

Isto significa que este cenário teve um consumo 17% maior de memória alocada do que o Cenário 2A.

A latência média entre o Monitor e o *Runtime* que executa o *FaaS* foi de 2 milissegundos, representando uma baixa sobrecarga em relação ao tempo de resposta.

A sobrecarga entre a partida a frio em relação à partida a quente foi de 404%, ou seja, a partida a frio foi 404% mais lenta do que a partida a quente enquanto a sobrecarga do tempo de resposta foi de 396% representando um valor próximo ao do tempo de partida.

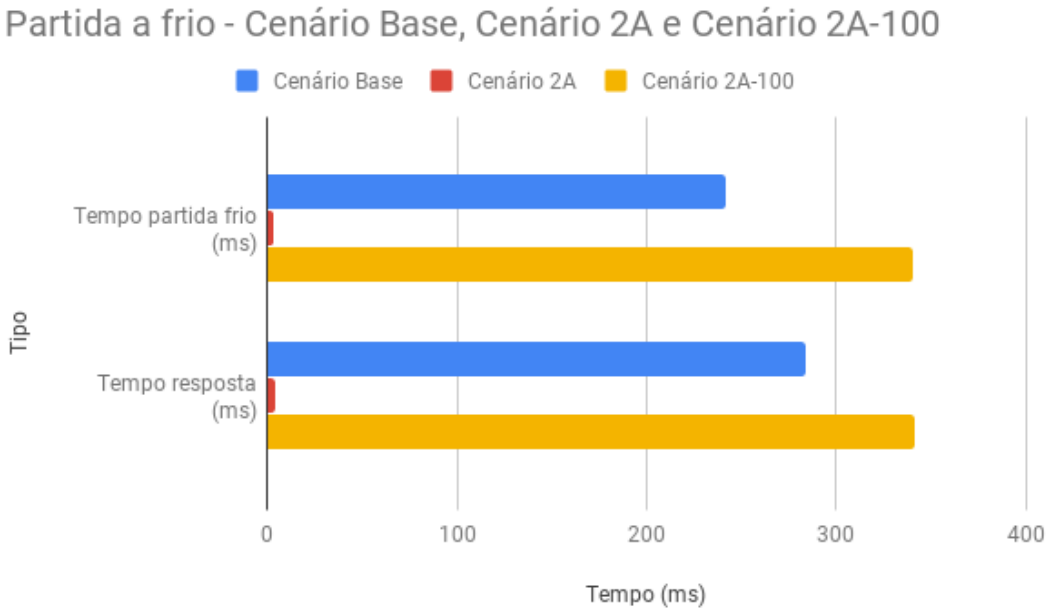

Figura 4.8: Cenário 2A-100 - Partida a frio.

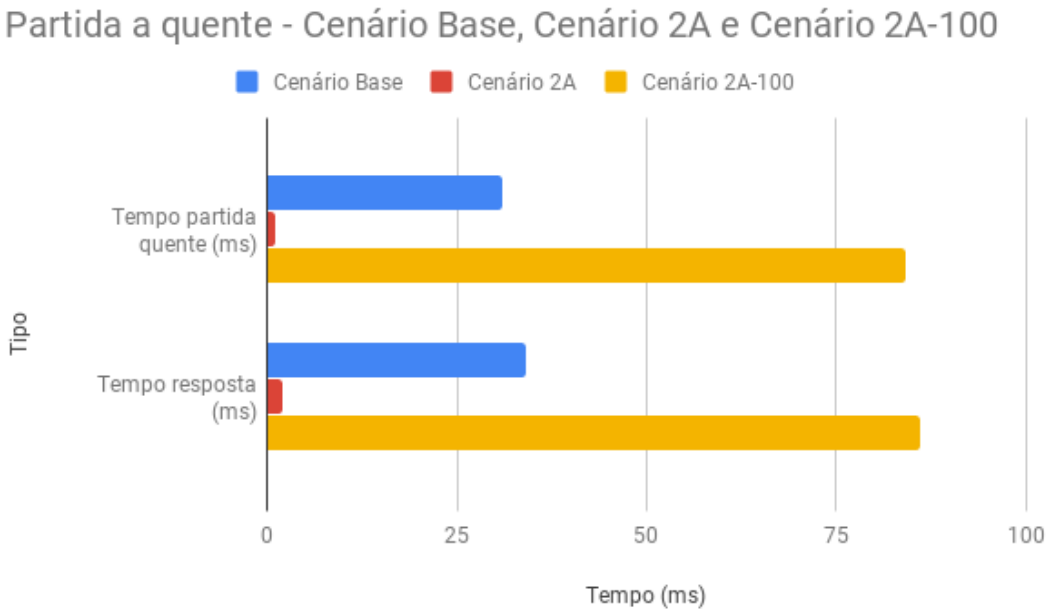

Figura 4.9: Cenário 2A-100 - Partida a quente.

Cenário 2A-200

Este cenário executa os testes do Cenário 2A paralelizando as requisições em grupos de 200, ou seja, através da criação de 200 requisições simultâneas ao invés de execução serial. Após as 50 baterias de 1.000 funções únicas, foram executadas um total de 50.000 funções de *FaaS* independentes numa única máquina virtual sendo obtidos os resultados mostrados na Tabela 4.14.

Tabela 4.14: Cenário 2A-200 - Resultado - 1.000 funções únicas

	Mínimo	Máximo	Média	Desvio	Intervalo (99%)
Partida a frio					
Tempo de partida (ms)	130	3752	1007	1273	992 - 1022
Tempo de resposta (ms)	164	3895	1009	1273	994 - 1024
Memória livre (MB)	3127	3290	3201	38	3200 - 3202
Memória alocada (MB)	65	127	104	17	103 - 105
Partida a quente					
Tempo de partida (ms)	73	1087	193	144	191 - 195
Tempo de resposta (ms)	82	1087	195	143	193 - 197
Memória livre (MB)	3120	3225	3176	34	3175 - 3177
Memória alocada (MB)	122	135	128	3	127 - 129
Latência					
Tempo médio (ms)			2		
Sobrecarga					
Tempo de partida			521%		
Tempo de resposta			517%		

Conforme evidenciado na Tabela 4.14 e os gráficos ilustrados na Figura 4.10 e Figura 4.11, a partida a frio demorou em média 1007 milissegundos enquanto a partida a quente demorou 193 milissegundos. Comparando com o Cenário 2A, a partida a frio foi 335 vezes mais lenta enquanto a partida a quente foi 193 vezes, sendo esta proporção de lentidão um resultado esperado já que houve somente uma única instância do *NodeJS* executando as funções *FaaS* que foram gerenciadas internamente através da criação de uma fila de execução. Em relação ao Cenário Base, a partida a frio e quente foram 3 vezes mais lenta.

A memória máxima alocada para a execução de 1.000 funções únicas foi de 135 *megabytes* enquanto no Cenário 2A foi de 102 *megabytes* para as mesmas 1.000 funções. Isto significa que este cenário teve um consumo 28% maior de memória alocada do que o Cenário 2A.

A latência média entre o Monitor e o *Runtime* que executa o *FaaS* foi de 2 milissegundos, representando uma baixa sobrecarga em relação ao tempo de resposta.

A sobrecarga entre a partida a frio em relação à partida a quente foi de 521%, ou seja, a partida a frio foi 521% mais lenta do que a partida a quente enquanto a sobrecarga do tempo de resposta foi de 517% representando um valor próximo ao do tempo de partida.

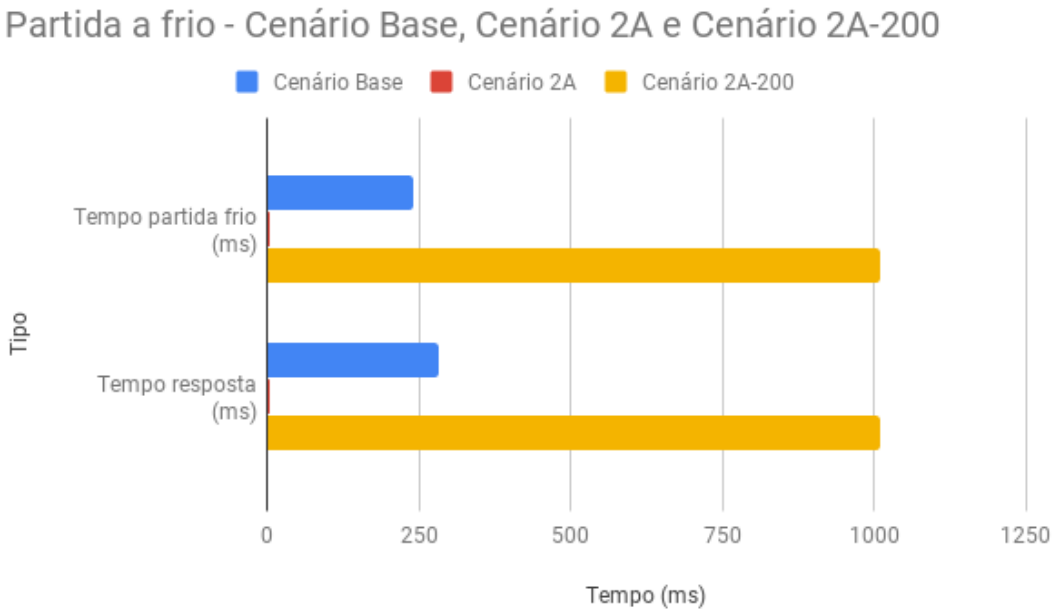

Figura 4.10: Cenário 2A-200 - Partida a frio.

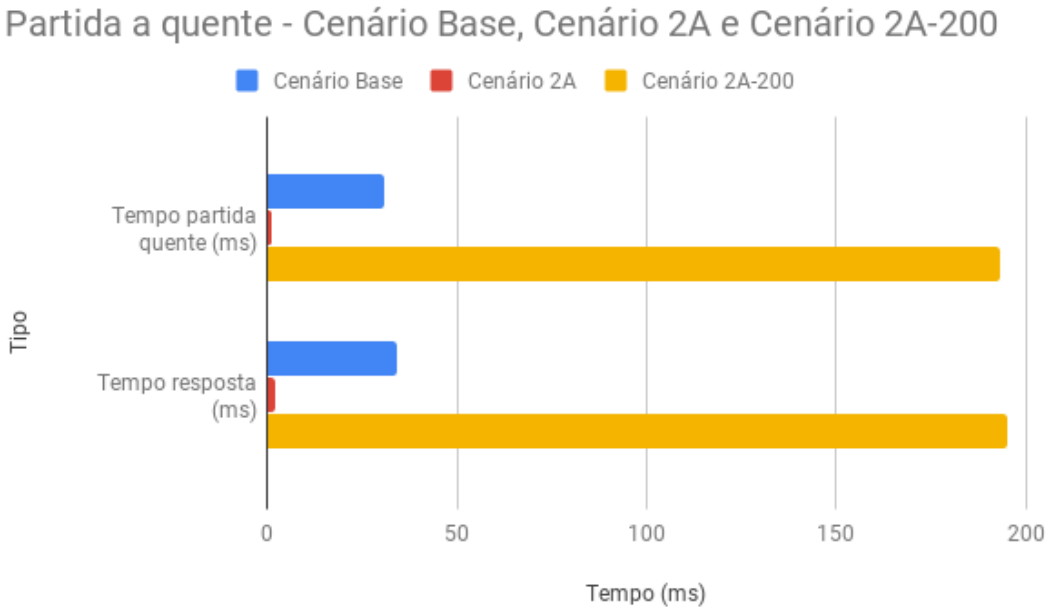

Figura 4.11: Cenário 2A-200 - Partida a quente.

Tabela 4.15: Cenário 2B - Configuração do ambiente.

Ativo	**Descrição**
Partida a frio - quantidade de *FaaS*	10.000
Partida a frio - total de baterias	5
Partida a frio - total de execuções	50.000
Partida a quente - quantidade de *FaaS*	10.000
Partida a quente - total de baterias	5
Partida a quente - total de execuções	50.000

Cenário 2B

Este cenário executa 10.000 funções *FaaS* únicas cujo ambiente de teste foi configurado conforme tabela Tabela 4.15

Após as 5 baterias de 10.000 funções únicas, foram executadas um total de 50.000 funções de *FaaS* independentes numa única máquina virtual sendo obtidos os resultados mostrados na Tabela 4.16.

Tabela 4.16: Cenário 2B - Resultado - 10.000 funções únicas

	Mínimo	Máximo	Média	Desvio	Intervalo (99%)
Partida a frio					
Tempo de partida (ms)	2	1015	4	10	3 - 5
Tempo de resposta (ms)	2	1016	5	11	4 - 6
Memória livre (MB)	3198	3489	3333	72	3332 - 3334
Memória alocada (MB)	39	305	182	69	181 - 183
Partida a quente					
Tempo de partida (ms)	1	1032	2	7	1 - 3
Tempo de resposta (ms)	1	1033	2	7	1 - 3
Memória livre (MB)	3193	3222	3209	8	3208 - 3210
Memória alocada (MB)	294	310	302	4	301 - 303
Latência					
Tempo médio (ms)			1		
Sobrecarga					
Tempo de partida			200%		
Tempo de resposta			250%		

Conforme evidenciado na Tabela 4.16 e os gráficos ilustrados na Figura 4.12 e Figura 4.13, a partida a frio demorou em média 4 milissegundos enquanto a partida a quente demorou 2 milissegundos. Comparando com o Cenário Base, a partida a frio foi

60 vezes mais rápida enquanto a partida a quente foi 15 vezes mais rápida evidenciando assim que a solução proposta, nesta configuração, ainda é viável.

A memória máxima alocada para a execução de 10.000 funções únicas foi de 310 *megabytes* enquanto no Cenário Base seria de aproximadamente 300.000 *megabytes* já que para cada função única no Cenário Base foi alocado em média 30 *megabytes*. Isto significa que este cenário teve um consumo 967 vezes menor de memória alocada do que teria no Cenário Base com a mesma quantidade de funções únicas.

A latência média entre o Monitor e o *Runtime* que executa o *FaaS* foi de 1 milissegundo, representando uma baixa sobrecarga em relação ao tempo de resposta.

A sobrecarga entre a partida a frio em relação à partida a quente foi de 200%, ou seja, a partida a frio foi 200% mais lenta do que a partida a quente e apesar desta sobrecarga ser somente 390% menor do que o Cenário Base, esta diferença em valor absoluto foi de somente 2 milissegundos enquanto no Cenário Base a diferença foi de 211 milissegundos. A sobrecarga do tempo de resposta foi de 250% representando um valor próximo ao do tempo de partida.

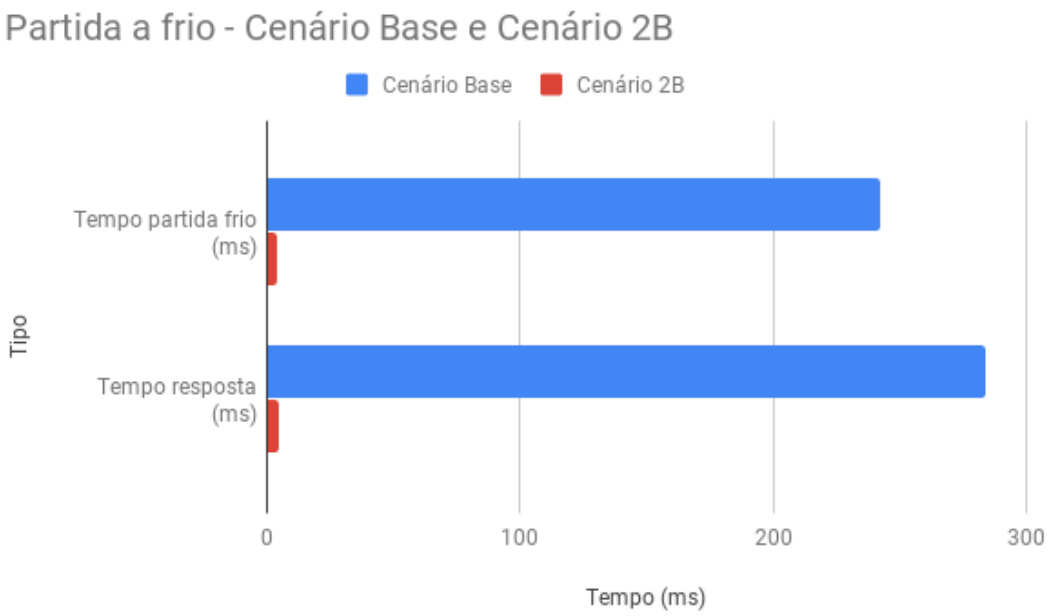

Figura 4.12: Cenário 2B - Partida a frio.

Tabela 4.17: Cenário 2C - Configuração do ambiente.

Ativo	**Descrição**
Partida a frio - quantidade de *FaaS*	20.000
Partida a frio - total de baterias	5
Partida a frio - total de execuções	100.000
Partida a quente - quantidade de *FaaS*	20.000
Partida a quente - total de baterias	5
Partida a quente - total de execuções	100.000

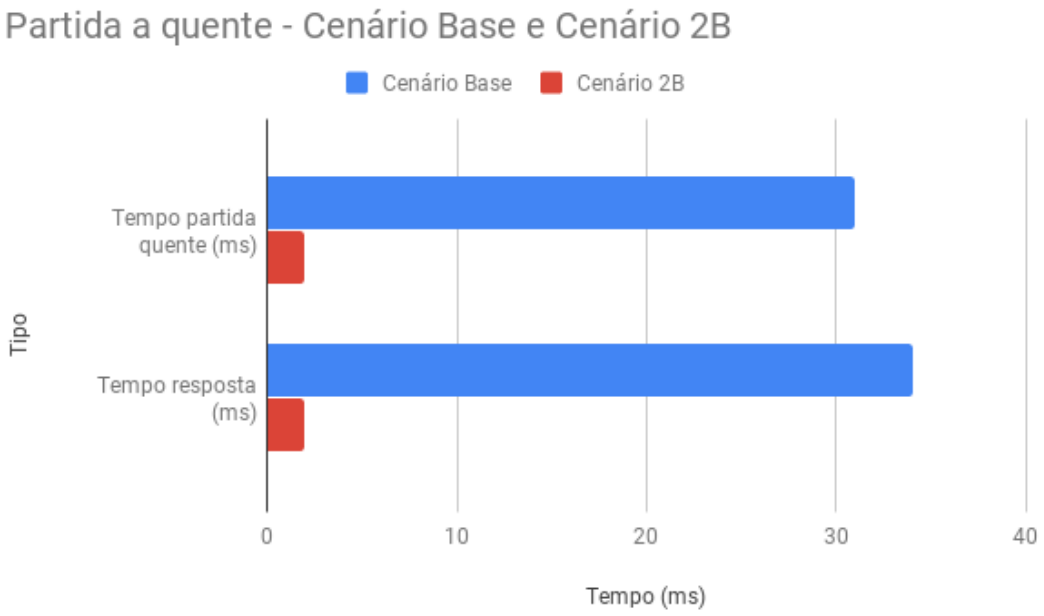

Figura 4.13: Cenário 2B - Partida a quente.

Cenário 2C

Este cenário executa 20.000 funções *FaaS* únicas cujo ambiente de teste foi configurado conforme tabela Tabela 4.17.

Após as 5 baterias de 20.000 funções únicas, foram executadas um total de 100.000 funções de *FaaS* independentes numa única máquina virtual sendo obtidos os resultados mostrados na Tabela 4.18.

Tabela 4.18: Cenário 2C - Resultado - 20.000 funções únicas

	Mínimo	Máximo	Média	Desvio	Intervalo (99%)
Partida a frio					
Tempo de partida (ms)	2	1029	3	4	2 - 4
Tempo de resposta (ms)	3	1030	4	4	3 - 5
Memória livre (MB)	2879	3488	3168	143	3166 - 3170
Memória alocada (MB)	39	524	298	129	296 - 300
Partida a quente					
Tempo de partida (ms)	1	1027	1	6	0 - 2
Tempo de resposta (ms)	2	1028	2	6	1 - 3
Memória livre (MB)	2869	3005	2938	51	2937 - 2939
Memória alocada (MB)	506	534	522	8	521 - 523
Latência					
Tempo médio (ms)			1		
Sobrecarga					
Tempo de partida			300%		
Tempo de resposta			200%		

Conforme evidenciado na Tabela 4.18 e os gráficos ilustrados na Figura 4.14 e Figura 4.15, a partida a frio demorou em média 4 milissegundos enquanto a partida a quente demorou 4 milissegundos. Comparando com o Cenário Base, a partida a frio foi 60 vezes mais rápida enquanto a partida a quente foi 7 vezes mais rápida evidenciando assim que a solução proposta, nesta configuração, ainda é viável.

A memória máxima alocada para a execução de 20.000 funções únicas foi de 534 *megabytes* enquanto no Cenário Base seria de aproximadamente 600.000 *megabytes* já que para cada função única no Cenário Base foi alocado em média 30 *megabytes*. Isto significa que este cenário teve um consumo 1123 vezes menor de memória alocada do que teria no Cenário Base com a mesma quantidade de funções únicas.

A latência média entre o Monitor e o *Runtime* que executa o *FaaS* foi de 1 milissegundo, representando uma baixa sobrecarga em relação ao tempo de resposta.

A sobrecarga entre a partida a frio em relação à partida a quente foi de 300%, ou seja, a partida a frio foi 300% mais lenta do que a partida a quente e apesar desta sobrecarga ser somente 260% menor do que o Cenário Base, esta diferença em valor absoluto foi de somente 2 milissegundos enquanto no Cenário Base a diferença foi de 211 milissegundos. A sobrecarga do tempo de resposta foi de 200% representando um valor próximo ao do tempo de partida.

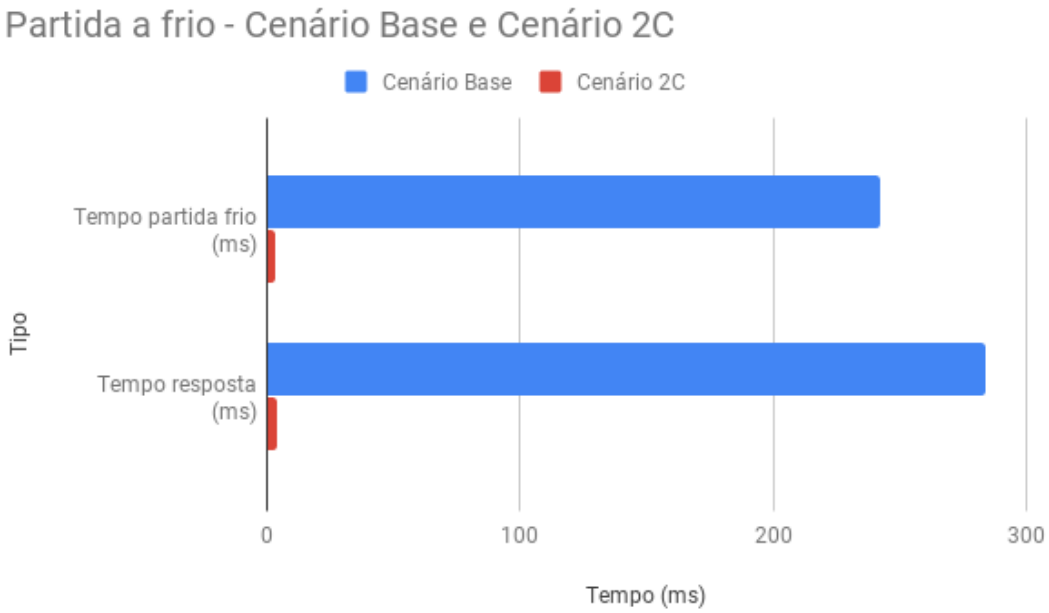

Figura 4.14: Cenário 2C - Partida a frio.

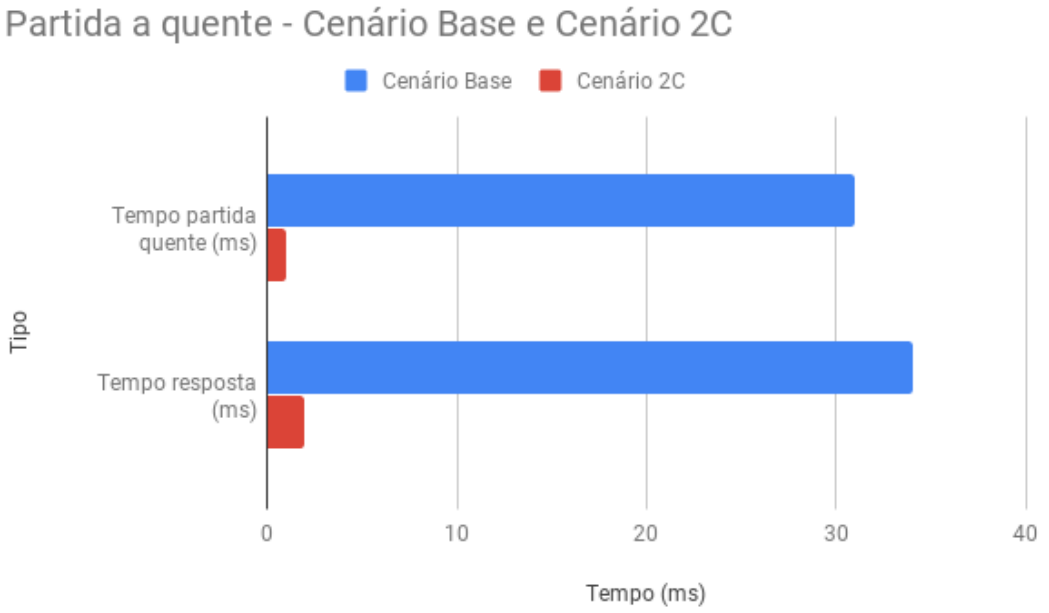

Figura 4.15: Cenário 2C - Partida a quente.

Tabela 4.19: Cenário 2D - Configuração do ambiente.

Ativo	**Descrição**
Partida a frio - quantidade de *FaaS*	30.000
Partida a frio - total de baterias	5
Partida a frio - total de execuções	150.000
Partida a quente - quantidade de *FaaS*	30.000
Partida a quente - total de baterias	5
Partida a quente - total de execuções	150.000

Cenário 2D

Este cenário executa 30.000 funções *FaaS* únicas cujo ambiente de teste foi configurado conforme tabela Tabela 4.19.

Após as 5 baterias de 30.000 funções únicas, foram executadas um total de 150.000 funções de *FaaS* independentes numa única máquina virtual sendo obtidos os resultados mostrados na Tabela 4.20.

Tabela 4.20: Cenário 2D - 30.000 funções únicas

	Mínimo	Máximo	Média	Desvio	Intervalo (99%)
Partida a frio					
Tempo de partida (ms)	2	1034	3	12	2 - 4
Tempo de resposta (ms)	2	1035	4	12	3 - 5
Memória livre (MB)	2674	3463	3042	199	3040 - 3044
Memória alocada (MB)	39	732	400	191	398 - 402
Partida a quente					
Tempo de partida (ms)	1	1031	2	7	1 - 3
Tempo de resposta (ms)	1	1033	3	7	2 - 4
Memória livre (MB)	2662	2764	2701	31	2700 - 2702
Memória alocada (MB)	718	744	733	6	732 - 734
Latência					
Tempo médio (ms)			1		
Sobrecarga					
Tempo de partida			150%		
Tempo de resposta			133%		

Conforme evidenciado na Tabela 4.20 e os gráficos ilustrados na Figura 4.16 e Figura 4.17, a partida a frio demorou em média 3 milissegundos enquanto a partida a quente demorou 2 milissegundos. Comparando com o Cenário Base, a partida a frio foi

80 vezes mais rápida enquanto a partida a quente foi 15 vezes mais rápida evidenciando assim que a solução proposta, nesta configuração, ainda é viável.

A memória máxima alocada para a execução de 30.000 funções únicas foi de 744 *megabytes* enquanto no Cenário Base seria de aproximadamente 900.000 *megabytes* já que para cada função única no Cenário Base foi alocado em média 30 *megabytes*. Isto significa que este cenário teve um consumo 1209 vezes menor de memória alocada do que teria no Cenário Base com a mesma quantidade de funções únicas.

A latência média entre o Monitor e o *Runtime* que executa o *FaaS* foi de 1 milissegundo, representando uma baixa sobrecarga em relação ao tempo de resposta.

A sobrecarga entre a partida a frio em relação à partida a quente foi de 150%, ou seja, a partida a frio foi 150% mais lenta do que a partida a quente e apesar desta sobrecarga ser somente 520% menor do que o Cenário Base, esta diferença em valor absoluto foi de somente 1 milissegundo enquanto no Cenário Base a diferença foi de 211 milissegundos. A sobrecarga do tempo de resposta foi de 133% representando um valor próximo ao do tempo de partida.

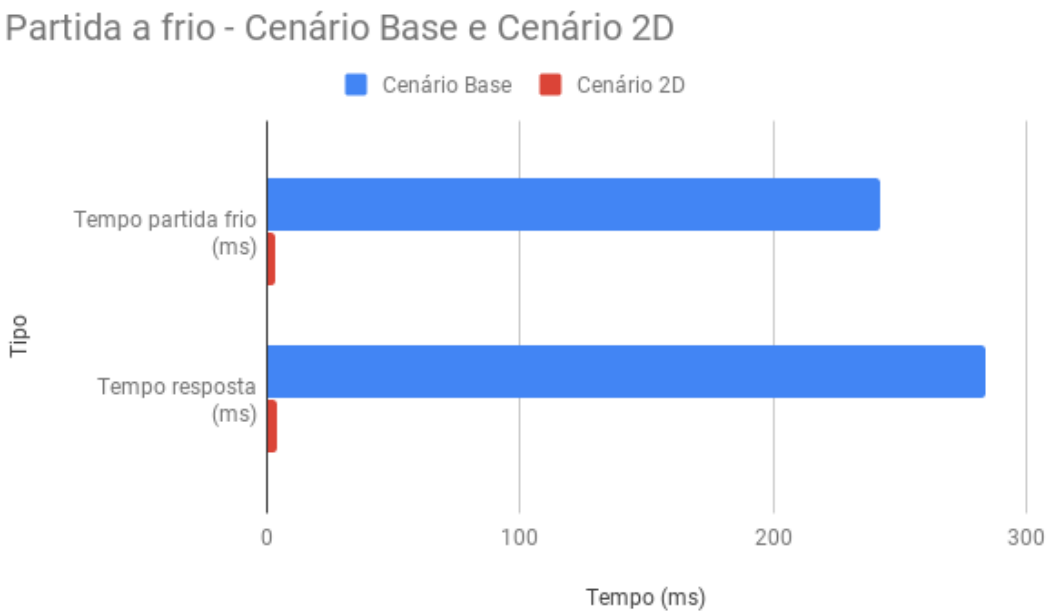

Figura 4.16: Cenário 2D - Partida a frio.

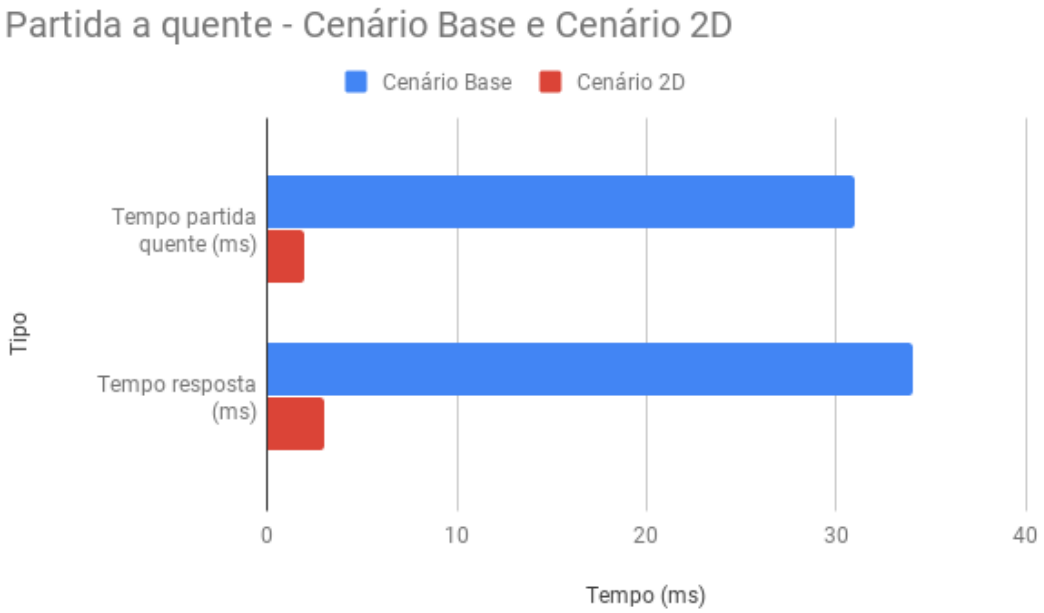

Figura 4.17: Cenário 2D - Partida a quente.

4.2.4 Cenário 3: Framework - Módulo Registro - Local próximo

Este cenário avalia a partida a frio do *framework* de *FaaS* proposto juntamente com o módulo de Registro implementado pelo próprio *framework* que é executado em um centro de dados diferente do módulo de *sandbox*, mas fisicamente próximos, localizados numa mesma cidade. Uma única *VM* foi utilizada para instanciar as funções de *FaaS*. O protocolo de transporte entre o *Runtime* e o módulo de Registro foi o *HTTPS*, sendo que o ambiente de teste foi configurado conforme descrição da tabela Tabela 4.21.

Após as 5 baterias de 10.000 funções únicas, foram executadas um total de 50.000 funções de *FaaS* independentes numa única máquina virtual sendo obtidos os resultados mostrados na Tabela 4.22.

Tabela 4.21: Cenário 3 - Configuração do ambiente.

Ativo	**Descrição**
Localização do módulo Registro	Este módulo faz parte do *framework* proposto sendo que o módulo de *sandbox* e o módulo de Registro estão em centro de dados diferentes em localizações físicas próximas numa mesma cidade
VM do gerador de carga e monitor	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
VM do *FaaS*	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
Node.js do *FaaS*	Versão 10.15.1
VM do módulo de Registro	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
Protocolo do módulo de Registro	HTTPS
Quantidade máquinas virtuais do *FaaS*	1
Limite memória do *FaaS*	1.7 *gigabytes*
Partida a frio - quantidade de *FaaS*	10.000
Partida a frio - total de baterias	5
Partida a frio - total de execuções	50.000
Partida a quente - quantidade de *FaaS*	10.000
Partida a quente - total de baterias	5
Partida a quente - total de execuções	50.000

Tabela 4.22: Cenário 3 - Resultado - 10.000 funções únicas

	Mínimo	Máximo	Média	Desvio	Intervalo (99%)
Partida a frio					
Tempo de partida (ms)	2	249	4	8	3 - 5
Tempo de resposta (ms)	3	253	4	9	3 - 5
Memória livre (MB)	3197	3487	3324	71	3323 - 3325
Memória alocada (MB)	39	301	186	68	185 - 187
Partida a quente					
Tempo de partida (ms)	1	64	2	2	1 - 3
Tempo de resposta (ms)	2	65	2	2	1 - 3
Memória livre (MB)	3192	3224	3207	9	3206 - 3208
Memória alocada (MB)	293	306	300	3	299 - 301
Latência					
Tempo médio (ms)			1		
Sobrecarga					
Tempo de partida			200%		
Tempo de resposta			200%		

Conforme evidenciado na Tabela 4.22 e os gráficos ilustrados na Figura 4.18 e Figura 4.19, a partida a frio demorou em média 4 milissegundos enquanto a partida a quente demorou 2 milissegundos. Comparando com o Cenário Base, a partida a frio foi 60 vezes mais rápida enquanto a partida a quente foi 15 vezes mais rápida evidenciando assim que a solução proposta, nesta configuração, ainda é viável. A quantidade máxima de memória alocada foi de 306 *MB* enquanto no Cenário Base foi de 31 *MB*. Comparando com os resultados do Cenário 2B, para 10.000 funções independentes, o tempo médio de partida a frio foi igual sendo que um tempo maior era esperado devido a maior latência de rede. Uma hipótese é que a latência de rede entre os dois centro de dados é muito baixa.

A latência média entre o Monitor e o *Runtime* que executa o *FaaS* foi de 1 milissegundo, representando uma baixa sobrecarga em relação ao tempo de resposta.

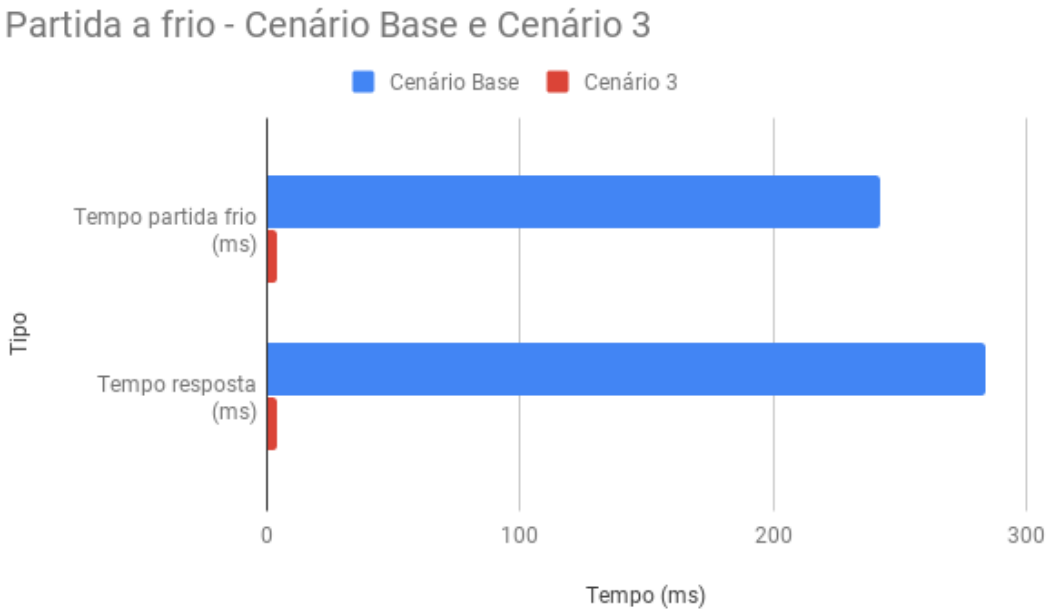

Figura 4.18: Cenário 3 - Partida a frio.

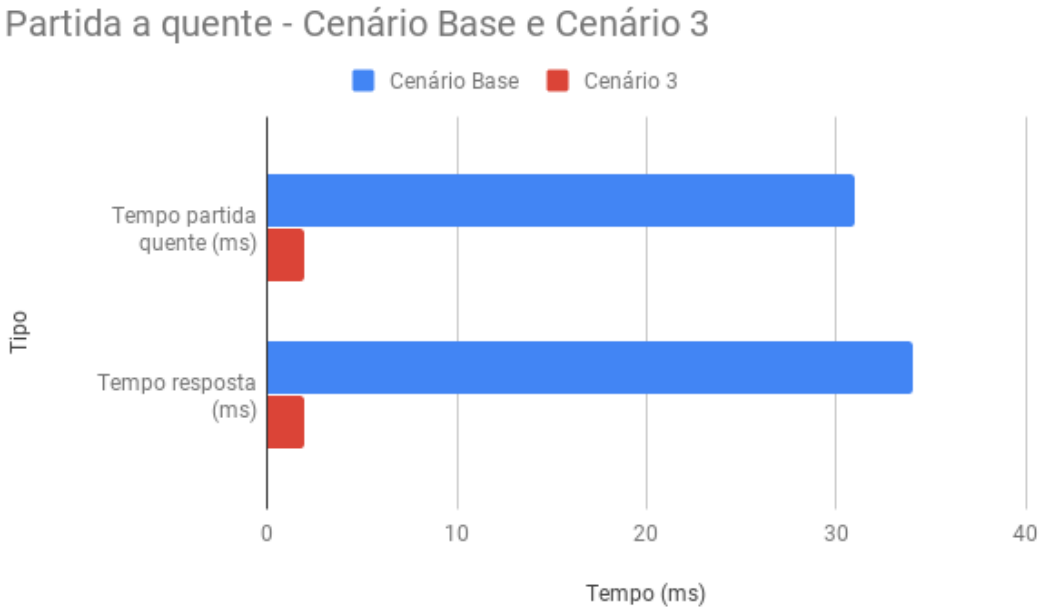

Figura 4.19: Cenário 3 - Partida a quente.

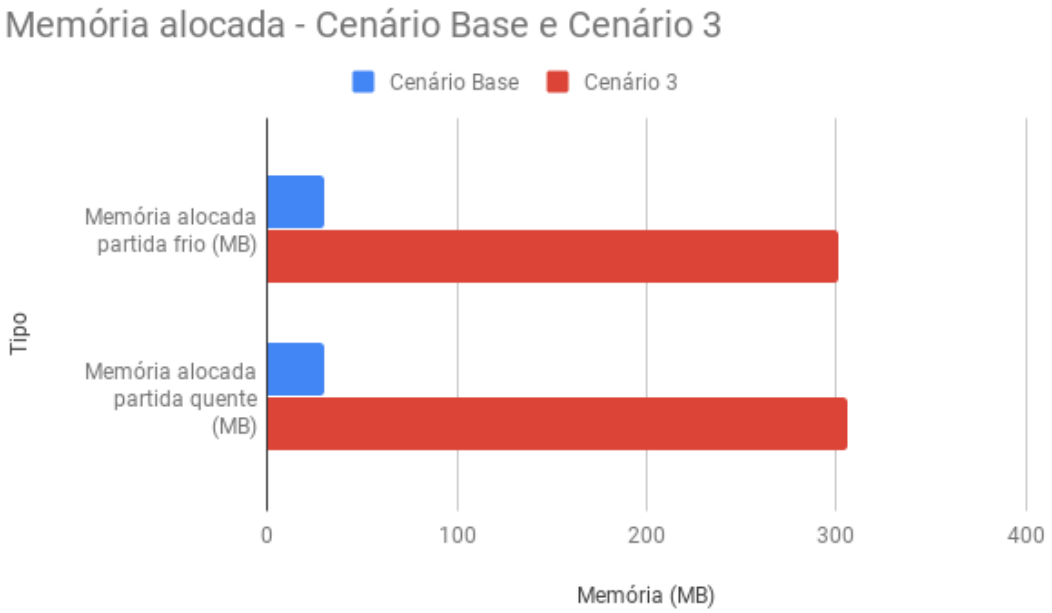

Figura 4.20: Cenário 3 - Memória alocada.

4.2.5 Cenário 4: Framework - Módulo Registro - Local distante

Este cenário avalia a partida a frio do *framework* de *FaaS* proposto com o módulo de Registro implementado pelo próprio *framework* que é executado em um centro de dados diferente do módulo de *sandbox* e fisicamente distante, localizados em países diferentes. Uma única *VM* foi utilizada para instanciar as funções de *FaaS*. O protocolo de transporte entre o *Runtime* e o módulo de Registro foi o *HTTPS*, sendo que o ambiente de teste foi configurado conforme descrição da tabela Tabela 4.21.

Após as 5 baterias de 10.000 funções únicas, foram executadas um total de 50.000 funções de *FaaS* independentes numa única máquina virtual sendo obtidos os resultados mostrados na Tabela 4.24.

Tabela 4.23: Cenário 4 - Configuração do ambiente.

Ativo	**Descrição**
Localização do módulo Registro	Este módulo faz parte do *framework* proposto sendo que o módulo de *sandbox* e o módulo de Registro estão em centro de dados diferentes em localizações físicas distantes em países diferentes
VM do gerador de carga e monitor	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
VM do *FaaS*	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
Node.js do *FaaS*	Versão 10.15.1
VM do módulo de Registro	Core(TM) i7-4500U com 8 GB de RAM e 4 vCPU. Sistema Operacional *Linux* versão 4.15.0-20-generic
Protocolo do módulo de Registro	HTTPS
Quantidade máquinas virtuais do *FaaS*	1
Limite memória do *FaaS*	1.7 *gigabytes*
Partida a frio - quantidade de *FaaS*	10.000
Partida a frio - total de baterias	5
Partida a frio - total de execuções	50.000
Partida a quente - quantidade de *FaaS*	10.000
Partida a quente - total de baterias	5
Partida a quente - total de execuções	50.000

Tabela 4.24: Cenário 4 - Resultado - 10.000 funções únicas

	Mínimo	Máximo	Média	Desvio	Intervalo (99%)
Partida a frio					
Tempo de partida (ms)	133	689	147	12	146 - 148
Tempo de resposta (ms)	134	715	150	12	149 - 151
Memória livre (MB)	118	1057	626	226	623 - 629
Memória alocada (MB)	40	260	153	62	152 - 154
Partida a quente					
Tempo de partida (ms)	0	62	2	2	1 - 3
Tempo de resposta (ms)	1	64	4	2	3 - 5
Memória livre (MB)	288	848	556	181	553 - 559
Memória alocada (MB)	258	292	285	8	284 - 286
Latência					
Tempo médio (ms)			3		
Sobrecarga					
Tempo de partida			7.350%		
Tempo de resposta			3.750%		

Conforme evidenciado na Tabela 4.24 e os gráficos ilustrados na Figura 4.21 e Figura 4.22, a partida a frio demorou em média 147 milissegundos enquanto a partida a quente demorou 2 milissegundos. Comparando com o Cenário Base, a partida a frio foi 1.6 vezes mais rápida enquanto a partida a quente foi 15 vezes mais rápida evidenciando assim que a solução proposta, nesta configuração, ainda é viável. A quantidade máxima de memória alocada foi de 292 *MB* enquanto no Cenário Base foi de 31 *MB*. Comparando com os resultados do Cenário 2B, para 10.000 funções independentes, o tempo médio de partida a frio foi 36 vezes maior conforme esperado devido a maior latência de rede.

A latência média entre o Monitor e o *Runtime* que executa o *FaaS* foi de 3 milissegundos, representando uma baixa sobrecarga em relação ao tempo de resposta.

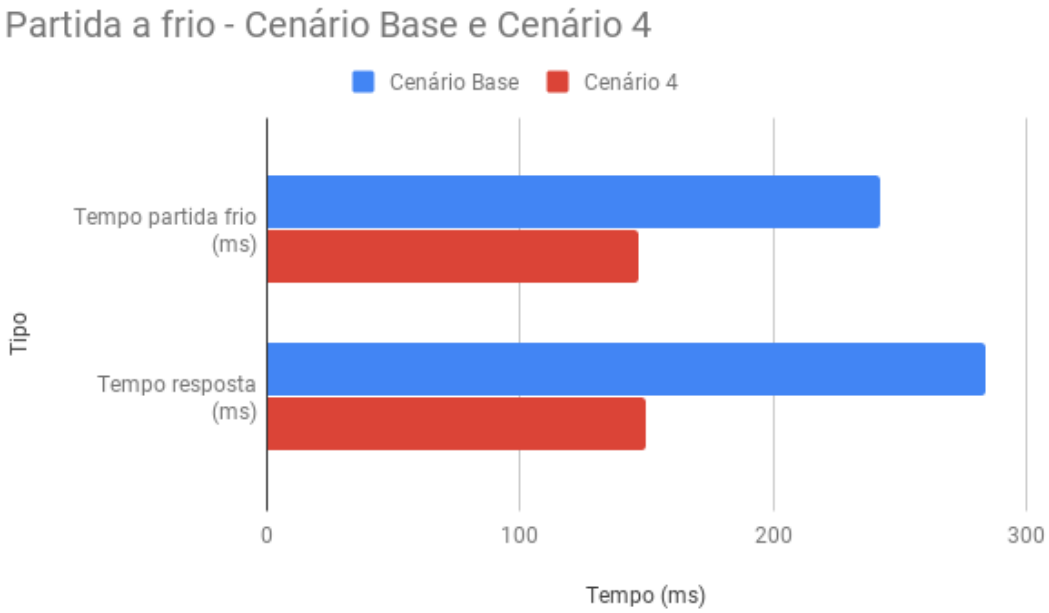

Figura 4.21: Cenário 4 - Partida a frio.

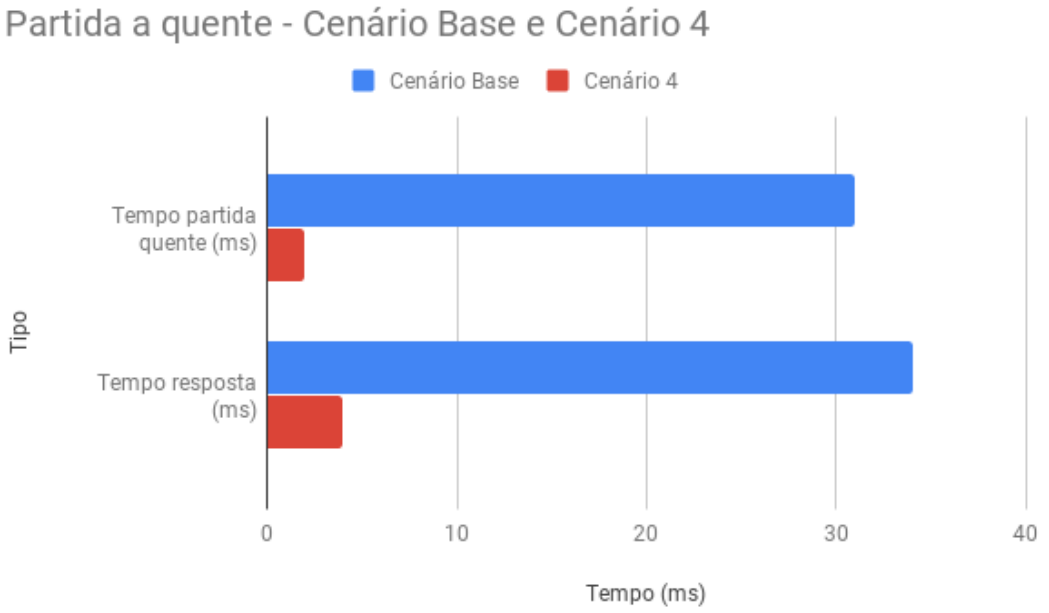

Figura 4.22: Cenário 4 - Partida a quente.

Tabela 4.25: Cenário 5 - Configuração do ambiente.

Ativo	**Descrição**
Localização do módulo Registro	Este módulo faz parte do *framework* proposto sendo que o módulo de *sandbox* e o módulo de Registro estão em uma mesma máquina virtual
VM do gerador de carga e monitor	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
VM do *FaaS*	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
Node.js do *FaaS*	Versão 10.15.1
VM do módulo de Registro	Xeon(R) CPU Xeon(R) E5-2666 com 3.75 GB de RAM e 2 vCPU. Sistema Operacional *Linux* versão 4.14.88-88.76.amzn2.x86_64
Protocolo do módulo de Registro	HTTP
Quantidade máquinas virtuais do *FaaS*	1
Limite memória do *FaaS*	1.7 *gigabytes*
Partida a frio - quantidade de *FaaS*	1.000
Partida a frio - total de baterias	50
Partida a frio - total de execuções	50.000
Partida a quente - quantidade de *FaaS*	1.000
Partida a quente - total de baterias	50
Partida a quente - total de execuções	50.000

4.2.6 Cenário 5: Framework - Módulo Registro - Mesma máquina

Este cenário avalia a partida a frio do *framework* de *FaaS* proposto com o módulo de Registro implementado pelo próprio *framework* que é executado na mesma máquina virtual, sendo que uma única *VM* foi utilizada para instanciar as funções de *FaaS*. O protocolo de transporte entre o *Runtime* e o módulo de Registro foi o *HTTP*, sendo que o ambiente de teste foi configurado conforme descrição da tabela Tabela 4.25.

Após as 50 baterias de 1.000 funções únicas, foram executadas um total de 50.000 funções de *FaaS* independentes numa única máquina virtual sendo obtidos os resultados mostrados na Tabela 4.26.

Tabela 4.26: Cenário 5 - Resultado - 1.000 funções únicas

	Mínimo	Máximo	Média	Desvio	Intervalo (99%)
Partida a frio					
Tempo de partida (ms)	1	64	2	3	1 - 3
Tempo de resposta (ms)	2	69	3	3	2 - 4
Memória livre (MB)	3030	3082	3056	12	3055 - 3057
Memória alocada (MB)	40	88	65	11	64 - 66
Partida a quente					
Tempo de partida (ms)	0	19	1	1	1 - 2
Tempo de resposta (ms)	1	22	2	1	1 - 3
Memória livre (MB)	3019	3043	3029	8	3028 - 3030
Memória alocada (MB)	77	99	91	7	90 - 92
Latência					
Tempo médio (ms)			1		
Sobrecarga					
Tempo de partida			200%		
Tempo de resposta			150%		

Conforme evidenciado na Tabela 4.26 e os gráficos ilustrados na Figura 4.23 e Figura 4.24, a partida a frio demorou em média 2 milissegundos enquanto a partida a quente demorou 1 milissegundo. Comparando com o Cenário Base, a partida a frio foi 121 vezes mais rápida enquanto a partida a quente foi 31 vezes mais rápida evidenciando assim que a solução proposta, nesta configuração, ainda é viável. A quantidade máxima de memória alocada foi de 99 *MB* enquanto no Cenário Base foi de 31 *MB*.

Além da localização, outra grande diferença em relação ao Cenário 2A foi a utilização do protocolo de comunicação *HTTP* entre o *Runtime* e o módulo de Registro ao invés do protocolo *HTTPS*, sendo que o tempo médio de partida a frio foi 66% menor e o tempo de partida a quente foi igual.

A latência média entre o Monitor e o *Runtime* que executa o *FaaS* foi de 1 milissegundo, representando uma baixa sobrecarga em relação ao tempo de resposta.

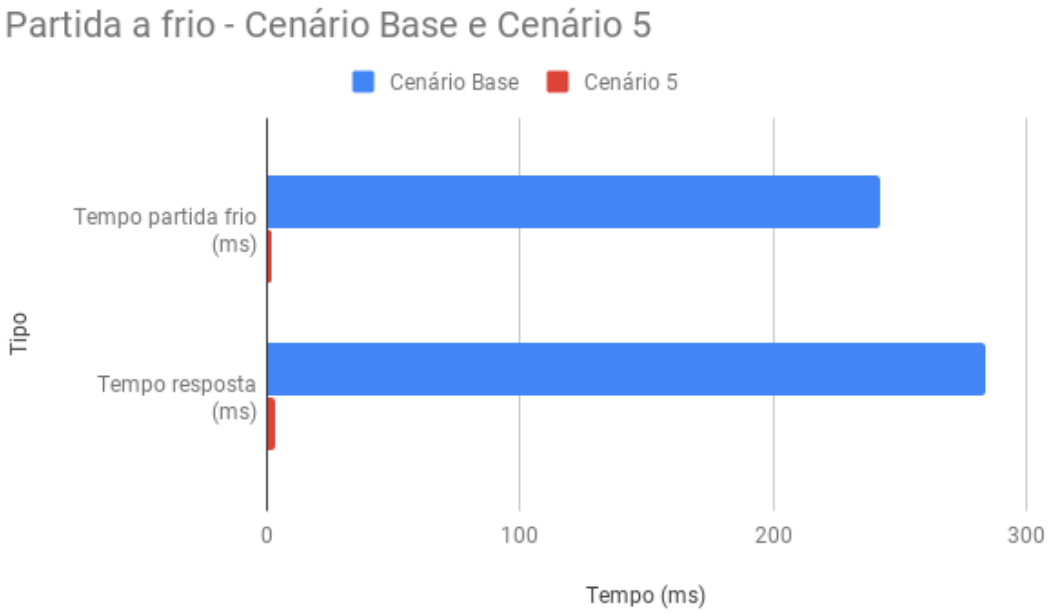

Figura 4.23: Cenário 5 - Partida a frio.

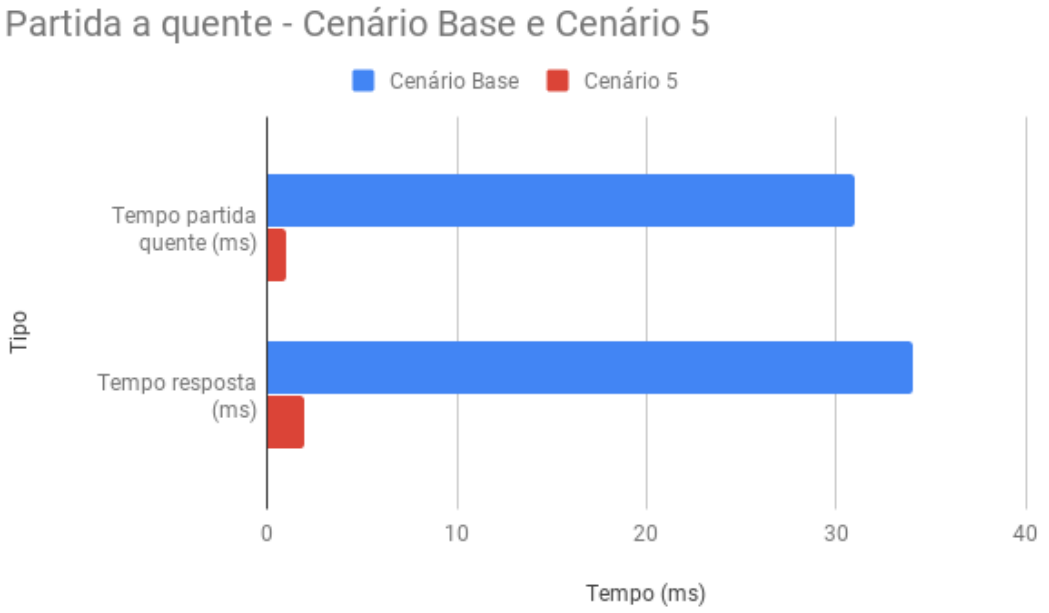

Figura 4.24: Cenário 5 - Partida a quente.

Tabela 4.27: Resultado - Cenários por distância e tecnologia do Registro.

Cenário	**Tempo (ms)**	**Distância**	**Tecnologia**
Cenário Base	242	Muito perto	AWS Amazon Lambda
Cenário 1	395	Longe	NPM
Cenário 2A	3	Muito perto	Framework
Cenário 2B	4	Muito perto	Framework
Cenário 2C	4	Muito perto	Framework
Cenário 2D	3	Muito perto	Framework
Cenário 3	4	Perto	Framework
Cenário 4	147	Muito longe	Framework
Cenário 5	2	Muitíssimo perto	Framework

4.3 Análise dos resultados

As Seções anteriores mostraram os resultados dos testes experimentais com um protótipo do *framework* proposto para avaliar se o mesmo é uma solução adequada para o problema de partida a frio em uma plataforma de *FaaS*.

Em todos os cenários os tempos de partida a frio foram maiores do que o tempo de partida a quente e o tempo de resposta foi maior que o tempo de partida sendo que estes resultados estão em conformidade com o comportamento esperado. Já a latência entre o Monitor e o *Runtime* ficou sempre abaixo de 3 milissegundos o que significa que ela pode ser considerada desprezível para o tempo de resposta.

Nas subseções seguintes serão analisados os resultados dos testes agrupados por cenário.

4.3.1 Análise dos resultados - Tempo de partida

Os resultados experimentais mostraram que o *framework* proposto é uma solução para o problema de partida a frio em um ambiente de computação em nuvem único, alcançando tempos médios máximos de 4 milissegundos que proporcionalmente é de 60 vezes mais rápido do que a plataforma *AWS Amazon Lambda*.

Os cenários de testes mostram que a tecnologia e a localização do módulo de Registro causam grandes impactos nos tempos de partida a frio como evidenciado na Tabela 4.27. Os cenários 2, 3 e 4 que utilizam o módulo de Registro do *framework* tiveram o tempo de partida a frio menor do que os que usam as tecnologias *NPM* e *AWS Amazon Lambda*, sendo que no cenário de maior distância, o Cenário 4, o tempo de partia a frio foi 39% menor do que o *AWS Amazon Lambda*.

No *framework* proposto, o tempo de partida a frio e quente são sensíveis à quantidade de funções únicas instanciadas conforme exibido nos gráficos da Figura 4.25 e Figura 4.26.

Não existe uma tendência de quanto maior o número de funções únicas alocadas maior será o tempo de partida a frio e quente portando, na existência de requisitos de baixíssimos tempos de resposta pode ser recomendado limitar a quantidade máxima de funções únicas que podem ser alocadas por máquina virtual.

Quando ocorrem requisições em paralelo, conforme exibido no gráfico da Figura 4.27 existe uma tendência de crescimento linear do tempo de resposta da partida a quente em relação à quantidade de requisições simultâneas, mas para a partida a frio esta tendência está mais próxima de um crescimento logarítmico o que implica que o *framework* proposto não consegue escalar requisições simultâneas em cenários de partida a frio com uma única instância. Este comportamento pode ser explicado devido ao fato da tecnologia *NodeJS* trabalhar somente com uma única *thread* do Sistema Operacional, sendo que, para utilizar as demais unidades de processamento disponíveis na máquina virtual é necessário a criação de novas instâncias na mesma máquina ou em outras externas.

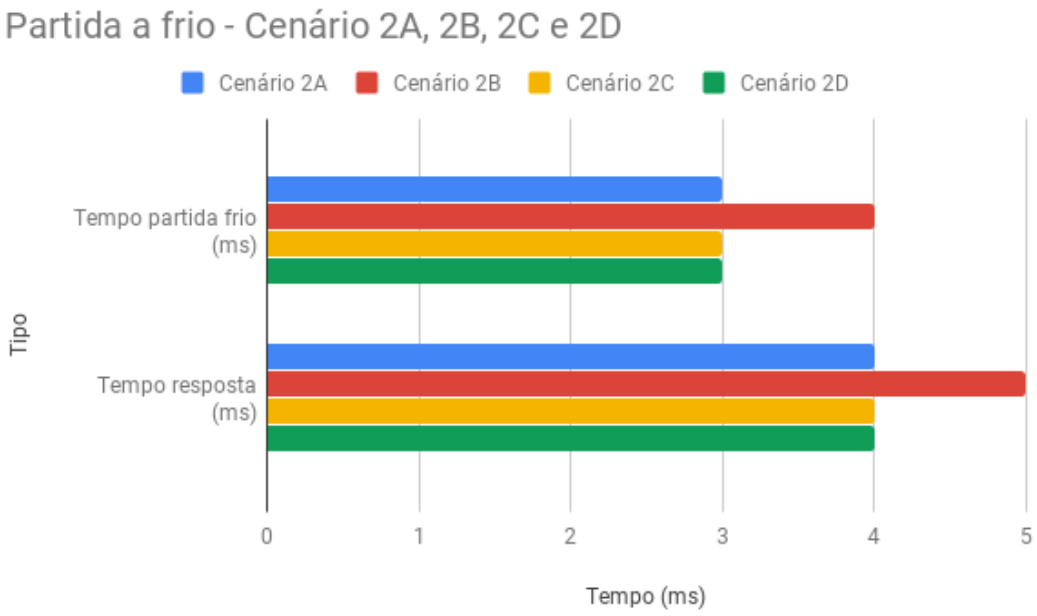

Figura 4.25: Cenário 2A, 2B, 2C e 2D - Partida a frio.

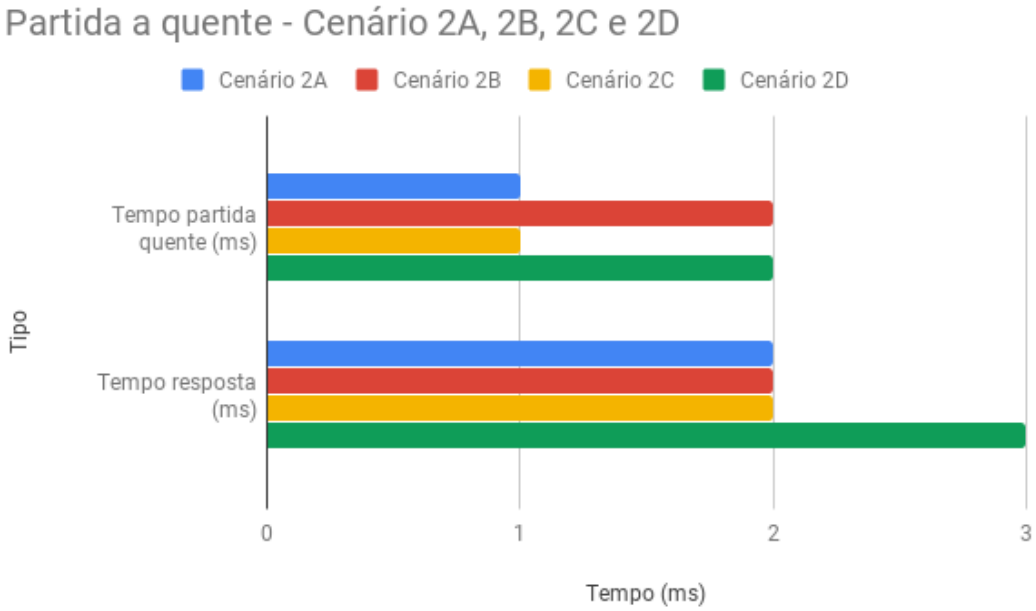

Figura 4.26: Cenário 2A, 2B, 2C e 2D - Partida a quente.

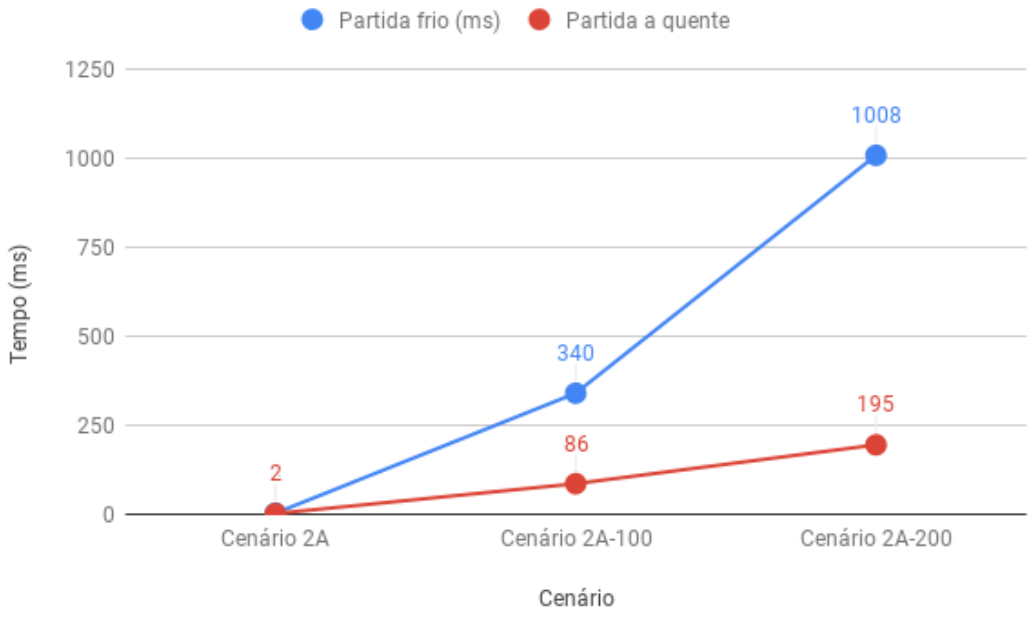

Figura 4.27: Cenário 2A, 2A-100, 2A-200 - Partida a frio e quente.

Para melhor evidenciar o impacto da distância e do protocolo de comunicação entre o *Runtime* e o módulo de Registro, foi executado Cenário 5 que hospedou ambos os

módulos numa mesma máquina e utilizou o protocolo de comunicação *HTTP* ao invés do *HTTPS* por ser um protocolo com uma performance melhor apesar de ser menos seguro já que não implementa a criptografia do canal. Os resultados, como evidenciados na Figura 4.28, mostraram que o tempo de partida a frio foi 66% menor do que o Cenário 2A onde o *Runtime* e o módulo de Registro estão num mesmo segmento de rede num mesmo centro de dados. Haja vista que foi alcançado o tempo de 2 milissegundos para a partida a frio com o *framework* proposto e devido à importância da redução de custo quando se utiliza *FaaS* concluímos que é mais relevante a distância e o protocolo de comunicação com o Registro do que utilizar estratégias de *cache* para instanciação de funções de *FaaS* já que quanto maior o número de funções em *cache* maior o consumo de memória como evidenciado na Figura 4.29.

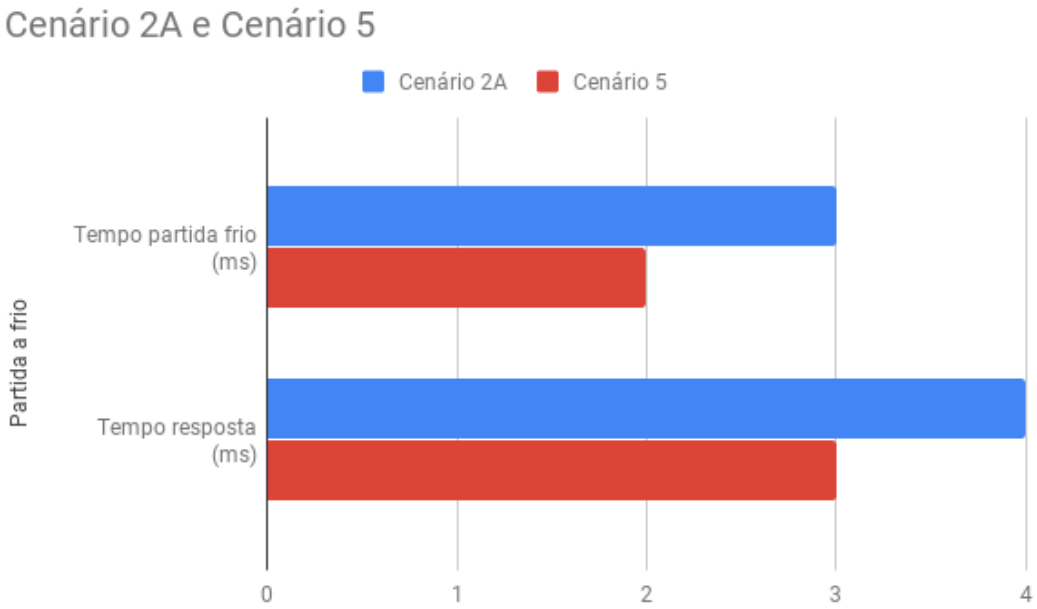

Figura 4.28: Cenário 5 e Cenário 2A - Partida a frio.

4.3.2 Análise dos resultados - Memória alocada

No *framework* proposto, a memória alocada aumenta linearmente com a quantidade de funções únicas instanciadas conforme resultado do Cenário 2 exibido no gráfico da Figura 4.29. Este comportamento é esperado já que todas as funções que foram executadas continuaram alocadas em memória dentro de uma mesma bateria de teste.

Conforme evidenciado na Tabela 4.28 e o gráfico exibido na Figura 4.30 a proporção entre a memória alocada pelo *Amazon AWS Lambda* e o *framework* tende a ficar pró-

Tabela 4.28: Resultado - Memória alocada.

	Amazon AWS Lambda (MB)	**Framework (MB)**	**Proporção**
1.000 FaaS	30.000	102	294
10.000 FaaS	300.000	310	967
20.000 FaaS	600.000	534	1123
30.000 FaaS	900.000	744	1209

ximo de 1250 quando instancia-se mais de 30.000 funções independentes. Este resultado demonstra que foi necessário somente uma única instância de máquina virtual para processar 30.000 funções de *FaaS* distintas com um consumo mínimo de 744 *megabytes* de memória.

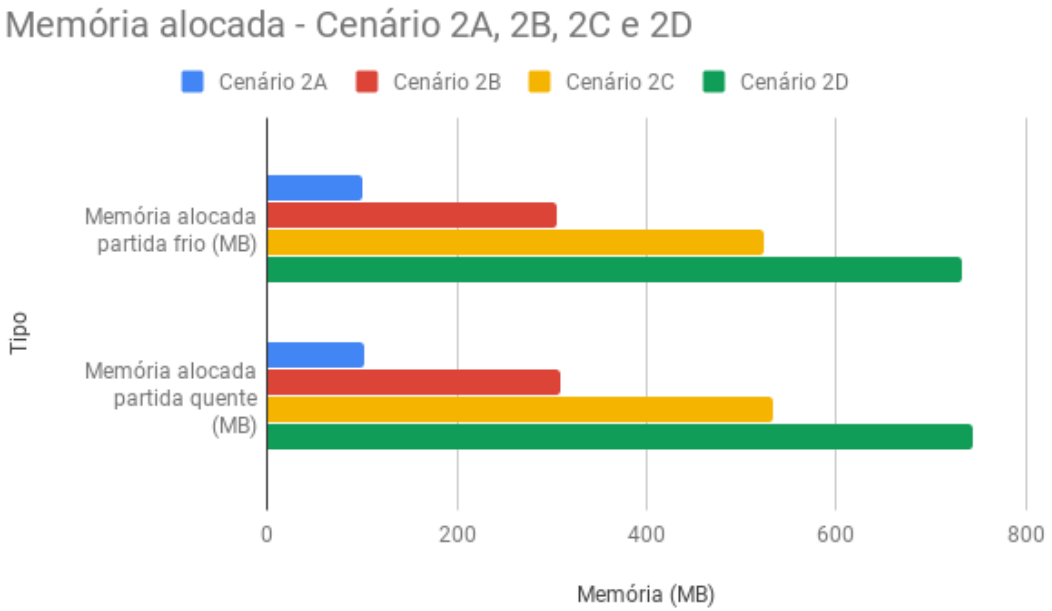

Figura 4.29: Cenário 2A, 2B, 2C e 2D - Memória alocada.

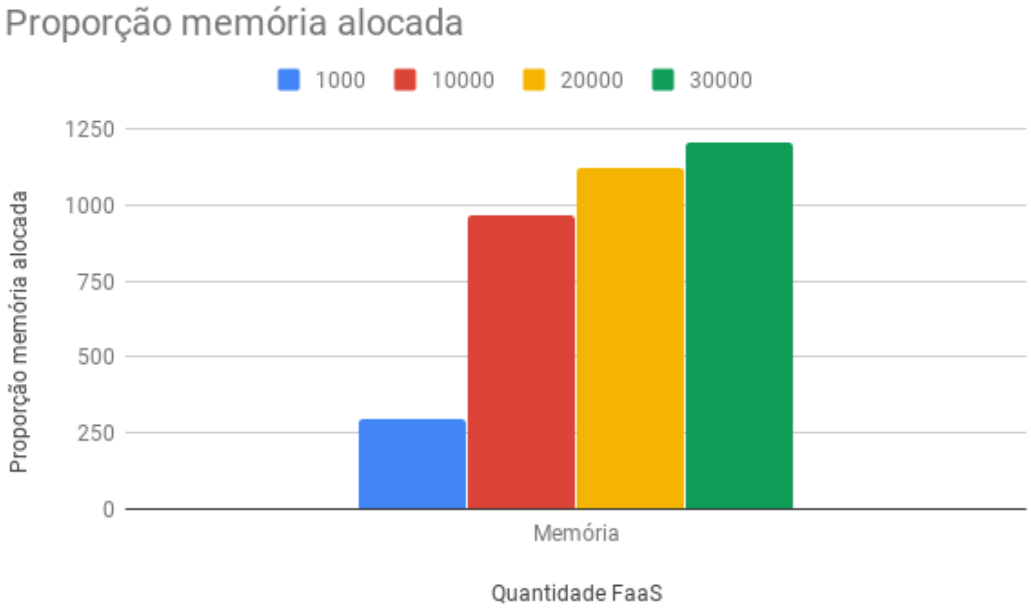

Figura 4.30: Proporção memória alocada Framework x Amazon AWS Lambda.

4.3.3 Análise dos resultados - Custo - Framework x Amazon AWS Lambda

Os resultados observados no Cenário Base e cenários 1, 2, 3 e 4 demostram que a plataforma de *FaaS* da *Amazon AWS* tem um baixo nível de compartilhamento de máquinas virtuais, já que para a execução de 50.000 funções *FaaS* independentes a plataforma instanciou 1918 instancias de máquinas virtuais, sendo que cada uma executa no máximo 26 funções *FaaS*. Já o *framework* proposto executou 30.000 funções *FaaS* numa única máquina virtual com o uso máximo de 744 *megabytes* da máquina virtual, o que significa que ele conseguiu executar um milhar de vezes (1.153) mais funções por *VM* do que a plataforma *AWS Lambda* e por consequência, para quem fornece a plataforma de *FaaS*, o *framework* proposto tem um custo um milhar de vezes mais barato do que o *AWS Lambda*.

4.3.4 Lições Aprendidas

Ficou evidenciado que a técnica de *sandbox* possibilita um alto nível de compartilhamento de recursos em relação a *AWS Lambda*, sendo assim, uma técnica muito eficaz para redução de custo de infraestrutura. O resultados demonstram que a localização do registro em relação ao *runtime* é mais importante do que a utilização de alguma estratégia de *cache* em ambiente de *FaaS*.

4.3.5 Considerações Finais

Este capítulo apresentou cinco cenários de avaliação com diferentes configurações e carga de trabalho com um cenário de referência utilizando a plataforma líder de mercado *Amazon AWS Lambda* [8].

A configuração e carga de trabalho do cenário de referência, Cenário Base, foram configuradas de acordo com o trabalho de Wang [43] e a divergência do resultado em relação a este trabalho foi de 9.5%. O resultado obtido neste cenário pode ser utilizado para validar o trabalho de Wang.

Os resultados dos cenários 2, 3 e 5 demonstraram que o *framework* proposto é uma solução viável para o problema de partida a frio, pois, tiveram tempos de partida menores que 10 milissegundos. Já o Cenário 4 demostrou que se o módulo de Registro estiver longe do *Runtime* torna-se inviável atingir um baixíssimo tempo de partida, mas foi obtido um resultado melhor do que a *Amazon AWS Lambda*. Por fim, o Cenário 1 demonstrou que não é viável utilizar o módulo de Registro *NPM* que teve como resultado tempo mais altos do que o Cenário Base.

As avaliações dos resultados mostram que o *framework* proposto tem um índice de compartilhamento de recursos bem melhor do que o *Amazon AWS Lambda* o que significa que ele possibilita ter um menor custo financeiro.

4.4 Resumo do capítulo

Neste capítulo foram apresentados os resultados da avaliação de uma implementação da proposta do *framework* para *FaaS*. Foram avaliados cinco cenários de testes e os resultados foram comparados com o serviço de *FaaS Amazon AWS Lambda* e por fim, considerações finais foram feitas. O resultados mostram que a proposta é uma solução viável para o problema de partida a frio, podendo o *framework* ser utilizado em ambientes computacionais diversos.

Capítulo 5

Conclusão e Trabalhos Futuros

Este trabalho foi instigado pela necessidade de alternativas para redução de custo de infraestrutura que pudessem ser utilizadas pela nova plataforma de nuvem do SERPRO chamada de Estaleiro. A utilização de *FaaS* é uma abordagem disponível que tem como sua principal premissa a redução de custo, mas possui um grande desafio que é a problemática da partida a frio, sendo que o *framework* proposto neste trabalho de dissertação teve como principal objetivo oferecer uma solução para este problema.

Um protótipo do *framework* proposto foi avaliado e os resultados mostraram que ele é uma solução viável para o problema da partia a frio alcançando tempos de partida menores que 10 milissegundos. A utilização de tecnologia *NodeJS* mostrou-se bastante promissora para diminuição do consumo de memória nas máquinas virtuais possibilitando um aumento no nível de compartilhamento de recursos e por consequência numa diminuição de custos financeiros já que uma quantidade menor de máquinas virtuais serão necessárias para execução do *FaaS*.

As contribuições desta dissertação são:

- Uma solução para o problema de partia a frio do *FaaS* além de uma solução para a melhoria no tempo de partia a quente;
- Validação de que a técnica de *sandbox* além de ser decisiva para a resolução do problema da partida a frio possibilita um aumento do nível de compartilhamento de recursos, o que possibilita uma diminuição de custos financeiros na infraestrutura de uma ambiente de *FaaS* e sendo assim, foi mostrado que a virtualização com *sandbox* é uma técnica mais interessante para *FaaS* do que a virtualização utilizando máquinas virtuais ou *container*;
- Foi demonstrado que a localização do módulo de Registro é essencial para a obtenção de baixos tempos de partida para uma plataforma de *FaaS*;

- Foi demonstrado que a utilização de *FaaS* é adequada para cenários que exigem baixo tempo de resposta como os ambientes de telemedicina, que exigem tempo de resposta fim a fim menores do que 10 milissegundos;
- Facilitar o desenvolvimento de *FaaS* através da implementação de um protótipo de *framework*;
- Validação experimental que é possível minimizar tráfego de rede na transferência de dados quando as funções comunicam-se entre si ao executar as funções dependentes num mesmo *Runtime*. Isto é possível pela análise do arquivo package.json que contém a lista de funções dependentes e por meio desta informação é possível obter remotamente o código da função FaaS de cada dependência;
- Validação parcial do trabalho de Wang [43] através da execução do Cenário Base que montou e analisou o mesmo experimento com resultados parecidos;
- Criação de um gerador de carga para teste de *FaaS*.

Para a execução das baterias de testes, uma das dificuldades foi ter que automatizar o gerenciamento do ambiente de nuvem da *Amazon* para instanciação e configuração de máquinas virtuais para execução das aplicações que compõe os componentes arquiteturais do *framework* devido a grande quantidade de testes que foram executados além da necessidade de excluir estas máquinas virtuais durante o tempo em que não estavam sendo utilizadas para diminuir o custo financeiro necessário para execução destes experimentos. Para esta tarefa foi utilizado uma biblioteca de programação disponibilizada pela própria plataforma e sendo assim, as máquinas virtuais só eram instanciadas quando os experimentos estavam em execução. Os *scripts* de automação estão contidos nos Anexos IV, V, VI, VII, VIII e IX.

Outra dificuldade foi a criação de um gerador de carga próprio para este experimento conforme visto nos Anexos I e II. Isto foi necessário devido as especifidades necessárias para execução de uma função *FaaS* da plataforma *AWS Amazon Lambda* assim como, a necessidade de recolher as métricas específicas dos servidores e coordenar o fluxo de execução de cada requisição remota através do protocolo HTTP.

Este trabalho focou em um pequeno recorte do assunto de *FaaS* e por consequência, as principais limitações encontradas nesta dissertação foram:

- Não foram tratados aspectos de segurança e bilhetagem em ambiente de *FaaS*;
- Não foram avaliados outras plataformas de *FaaS* disponíveis pelo mercado;
- Este trabalho concentrou-se no problema de partida a frio e não avaliou orquestradores e sendo assim, a implementação do *framework* proposto necessita de orquestradores externos para que seja escalável.

5.1 Trabalhos Futuros

As oportunidades de pesquisas identificadas para trabalhos futuros são:

- Avaliação do uso de tecnologias de transporte *peer-to-peer* como o *IPFS* para o módulo de Registro. Uma hipótese é que este tipo de tecnologia pode contribuir para a utilização de *FaaS* em computação de borda tanto em problemáticas de partida a frio quanto em aspectos de segurança;
- Avaliação da proposta deste trabalho em computação de borda e cenários de IoT;
- Avaliação de outras plataformas de *FaaS*;

Referências

[1] Li, Chao, Yushu Xue, Jing Wang, Weigong Zhang e Tao Li: *Edge-oriented computing paradigms: A survey on architecture design and system management.* ACM Computing Surveys (CSUR), 51(2):39, 2018. xi, 4, 5

[2] Salas Zarate, Maria e Luis Colombo Mendoza: *Cloud computing: a review of paas, iaas, saas services and providers.* Lampsakos, (7):47–57, 2012, ISSN 2145-4086. `http://dialnet.unirioja.es/servlet/oaiart?codigo=4490150`. xi, 6, 7

[3] Erl, Thomas: *Soa: Princípios de design de serviços.* Tradução Edson Furmankiewicz e Carlos, 2009. xi, 10, 11

[4] Mahmood, Sajjad, Richard Lai e Yong Soo Kim: *Survey of component-based software development.* IET software, 1(2):57–66, 2007. xi, 10

[5] Fowler, Martin e James Lewis: *Microservices, 2014.* `http://martinfowler.com/articles/microservices.html`, 2014. Online; acessado em 12/03/2018. xi, 10, 11, 12

[6] Newman, Sam: *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015. xi, 12

[7] Aderaldo, Carlos M, Nabor C Mendonça, Claus Pahl e Pooyan Jamshidi: *Benchmark requirements for microservices architecture research.* Em *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*, páginas 8–13. IEEE Press, 2017. xi, 11, 12

[8] Lynn, Theo, Pierangelo Rosati, Arnaud Lejeune e Vincent Emeakaroha: *A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms.* Em *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, páginas 162–169. IEEE, 2017. xi, 12, 13, 15, 16, 33, 71

[9] unikernel.org: *Unikernel.* `http://unikernel.org/`, 2018. Online; acessado em 03/07/2018. xi, 13, 14

[10] Foundation, Node.js: *The node.js event loop, timers, and process.nexttick().* `https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/`, 2018. Online; acessado em 01/04/2018. xi, 14, 15, 21

[11] Baldini, Ioana, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen J. Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric M. Rabbah, Aleksander Slominski e Philippe Suter: *Serverless computing: Current trends and open*

problems. CoRR, abs/1706.03178, 2017. http://arxiv.org/abs/1706.03178. xi, 15, 16, 17

[12] Malawski, Maciej, Adam Gajek, Adam Zima, Bartosz Balis e Kamil Figiela: *Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions*. Future Generation Computer Systems, 2017, ISSN 0167-739X. http://www.sciencedirect.com/science/article/pii/S0167739X1730047X. xi, 17, 18, 19, 21

[13] Docker: *Docker overview*. https://docs.docker.com/engine/docker-overview/, 2019. Online; acessado em 17/06/2019. xi, 21, 22, 26

[14] SERPRO: *Estaleiro: a nuvem do serpro*. http://www.serpro.gov.br/tema/edicao-238/estaleiro-a-nuvem-do-serpro, 2017. Online; acessado em 27/06/2018. 1

[15] Convergenciadigital: *Serpro quer ser o fornecedor de paas do governo federal*. http://convergenciadigital.uol.com.br/, 2018. Online; acessado em 16/06/2019. 1

[16] Villamizar, Mario, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano *et al.*: *Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures*. Service Oriented Computing and Applications, 11(2):233–247, 2017. 1, 15, 16, 27

[17] Kaul, Siddarth, Anuj Jain e Rajesh Saini: *Understanding fifth generation communication and internet of things (iot)*. Indian Journal of Computer Science, 3(4):32–36, 2018. 2

[18] Mell, Peter, Tim Grance *et al.*: *The nist definition of cloud computing*. 2011. 5, 6

[19] Shi, Weisong e Schahram Dustdar: *The promise of edge computing*. Computer, 49(5):78–81, 2016. 7, 8

[20] Ahmed, Ejaz e Mubashir Husain Rehmani: *Mobile edge computing: opportunities, solutions, and challenges*, 2017. 7, 8

[21] CISCO: *Edge computing vs. fog computing: Definitions and enterprise uses*. https://www.cisco.com/c/en/us/solutions/enterprise-networks/edge-computing.html, 2018. Online; acessado em 26/06/2018. 7, 9

[22] NIST: *Fog computing conceptual model*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-325.pdf, 2018. Online; acessado em 26/06/2018. 9

[23] Li, Shanshan: *Understanding quality attributes in microservice architecture*. Em *Software Engineering Conference Workshops (APSECW), 2017 24th Asia-Pacific*, páginas 9–10. IEEE, 2017. 10, 12

[24] Jyotsna, Mrs: *Component-based development technologies and limitations*. International Journal Of Engineering And Computer Science, 3(10), 2014. 10

[25] Tragatschnig, Simon e Uwe Zdun: *Modeling change patterns for impact and conflict analysis in event-driven architectures*. Em *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2015 IEEE 24th International Conference on*, páginas 44–46. IEEE, 2015. 11

[26] Tilkov, Stefan e Steve Vinoski: *Node.js: Using javascript to build high-performance network programs*. IEEE Internet Computing, 14(6):80–83, 2010. 11, 14

[27] vmware: *Virtualização*. `https://www.vmware.com/br/solutions/virtualization.html`, 2018. Online; acessado em 11/07/2018. 12

[28] Savolainen, Petri, Sumi Helal, Jukka Reitmaa, Kai Kuikkaniemi, Giulio Jacucci, Mikko Rinne, Marko Turpeinen e Sasu Tarkoma: *Spaceify: A client-edge-server ecosystem for mobile computing in smart spaces*. Em *Proceedings of the 19th annual international conference on Mobile computing & networking*, páginas 211–214. ACM, 2013. 13, 19, 21, 25, 26

[29] Savage, Neil: *Going serverless*. Commun. ACM, 61(2):15–16, janeiro 2018, ISSN 0001-0782. `http://doi-acm-org.ez54.periodicos.capes.gov.br/10.1145/3171583`. 15

[30] McGrath, Garrett e Paul R Brenner: *Serverless computing: Design, implementation, and performance*. Em *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*, páginas 405–410. IEEE, 2017. 15, 18

[31] Eivy, Adam: *Be wary of the economics of" serverless" cloud computing*. IEEE Cloud Computing, 4(2):6–12, 2017. 15, 16

[32] Anderson, Charles: *Docker [software engineering]*. IEEE Software, 32(3):102–c3, May 2015, ISSN 0740-7459. 16, 33

[33] Abad, Cristina L, Edwin F Boza e Erwin van Eyk: *Package-aware scheduling of faas functions*. Em *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, páginas 101–106. ACM, 2018. 19

[34] Yin, Hao, Xu Zhang, Hongqiang H Liu, Yan Luo, Chen Tian, Shuoyao Zhao e Feng Li: *Edge provisioning with flexible server placement*. IEEE Transactions on Parallel and Distributed Systems, 28(4):1031–1045, 2017. 19

[35] Weissman, Jon B, Pradeep Sundarrajan, Abhishek Gupta, Matthew Ryden, Rohit Nair e Abhishek Chandra: *Early experience with the distributed nebula cloud*. Em *Proceedings of the fourth international workshop on Data-intensive distributed computing*, páginas 17–26. ACM, 2011. 19

[36] Król, Michał e Ioannis Psaras: *Nfaas: named function as a service*. Em *Proceedings of the 4th ACM Conference on Information-Centric Networking*, páginas 134–144. ACM, 2017. 20

[37] Nodejs: *Nodejs benchmarking*. `https://benchmarking.nodejs.org/`, 2019. Online; acessado em 11/02/2019. 21, 35

[38] NPM: *Npm*. https://docs.npmjs.com/about-npm/, 2019. Online; acessado em 02/02/2019. 22, 35

[39] semver: *semver*. https://semver.org, 2019. Online; acessado em 21/07/2019. 23

[40] aws: *aws lambda*. https://aws.amazon.com/pt/lambda, 2019. Online; acessado em 21/07/2019. 27

[41] amazon: *Regiões aws*. https://docs.aws.amazon.com/pt_br/AWSEC2/latest/UserGuide/using-regions-availability-zones.html, 2019. Online; acessado em 11/02/2019. 27

[42] Amazon: *Aws time sync*. https://aws.amazon.com/pt/about-aws/whats-new/2017/11/introducing-the-amazon-time-sync-service/, 2019. Online; acessado em 30/01/2019. 29

[43] Wang, Liang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart e Michael Swift: *Peeking behind the curtains of serverless platforms*. Em *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, páginas 133–146, 2018. 34, 35, 71, 73

Anexo I

Código fonte do gerador de carga e monitor Framework

```
"use strict";

const https = require("https");
const path = require("path");
const fs = require("fs");
const arquivoResultado1 = path.join(__dirname, "dados/resultado3.json");
const arquivoResultado2 = path.join(__dirname, "dados/resultado4.json");
const quantidade = 1000;

var resultadoRegistros1 = null;
var resultadoRegistros2 = null;

try {
    resultadoRegistros1 = JSON.parse(fs.readFileSync(arquivoResultado1));
}
catch (error) {
    resultadoRegistros1 = [];
}
try {
    resultadoRegistros2 = JSON.parse(fs.readFileSync(arquivoResultado2));
}
catch (error) {
    resultadoRegistros2 = [];
}

function post(index){
```

```
    return new Promise(function(resolve, reject){
        let body = [];

        var post_options = {
            host: "18.232.199.146",
            port: "8443",
            path: "/",
            method: "POST",
            rejectUnauthorized:false,
            headers: {"Content-Type": "application/json"}
        };

        var dados = '{"scope":"performance1","method":"test' + index + '"}';

        var post_req = https.request(post_options, function(res) {
            res.on("data", function (chunk) {
                body.push(chunk);
            }).on("end", function(){
                try {
                    body = Buffer.concat(body);
                    body = JSON.parse(body);
                    resolve(body.result);
                }
                catch (errTry) {
                    reject(errTry)
                }
            });
        });

        post_req.write(dados);
        post_req.end();
    });
}


async function executarBateriaTeste(){
    var somaTotalTipo1 = 0;
    var somaTotalTipo2 = 0;

    console.log("Executando... tipo 1");
```

```
    for (var i = 1; i < quantidade + 1; i++){
        var timeInvoke = new Date().getTime();
        var dados = await post(i);
        var responseTime = new Date().getTime() - timeInvoke;
        dados.timeInvoke = timeInvoke;
        dados.delay = dados.timeInit - dados.timeInvoke;
        dados.responseTime = responseTime;
        resultadoRegistros1.push(dados);
        somaTotalTipo1 += dados.delay;
        console.log("Executando ", i);
    }
    fs.writeFileSync(arquivoResultado1, JSON.stringify(resultadoRegistros1));
    console.log("Fim tipo 1, qtd:", resultadoRegistros1.length);

    console.log("Executando... tipo 2");
    for (var i = 1; i < quantidade + 1; i++){
        var timeInvoke = new Date().getTime();
        var dados = await post(i);
        var responseTime = new Date().getTime() - timeInvoke;
        dados.timeInvoke = timeInvoke;
        dados.delay = dados.timeInit - dados.timeInvoke;
        dados.responseTime = responseTime;
        resultadoRegistros2.push(dados);
        somaTotalTipo2 += dados.delay;
        console.log("Executando ", i);
    }
    fs.writeFileSync(arquivoResultado2, JSON.stringify(resultadoRegistros2));
    console.log("Fim tipo 2, qtd:", resultadoRegistros2.length);

    console.log("Media tipo 1: ", somaTotalTipo1 / quantidade);
    console.log("Media tipo 2: ", somaTotalTipo2 / quantidade);
};

executarBateriaTeste();
```

Anexo II

Código fonte do gerador de carga e monitor AWS Lambda

```
"use strict";

const AWSManager = require("../lib/AWS/AWSManager");
const awsManager = new AWSManager();
const region = "us-east-1";
const path = require("path");
const fs = require("fs");
const arquivoResultado1 = path.join(__dirname, "dados/resultado1.json");
const arquivoResultado2 = path.join(__dirname, "dados/resultado2.json");
const quantidade = 1000;

var codigoLambdaBuffer = fs.readFileSync(path.join(__dirname, "lambdaCodigo.zip"));

var resultadoRegistros1 = null;
var resultadoRegistros2 = null;

try {
    resultadoRegistros1 = JSON.parse(fs.readFileSync(arquivoResultado1));
}
catch (error) {
    resultadoRegistros1 = [];
}
try {
    resultadoRegistros2 = JSON.parse(fs.readFileSync(arquivoResultado2));
}
catch (error) {
```

```
    resultadoRegistros2 = [];
}

async function executarBateriaTeste(){
    var somaTotalTipo1 = 0;
    var somaTotalTipo2 = 0;

    //criar funções
    try {
        for (var i = 1; i < quantidade + 1; i++){
            var respostaCriar = await awsManager.lambda.createFunction(
                region, "performance_" + i, codigoLambdaBuffer);
            console.log("Criado", i);
        }
    }
    catch (errTry) {
        console.log(errTry.message);
    }

    console.log("Executando... tipo 1");
    for (var i = 1; i < quantidade + 1; i++){
        var timeInvoke = new Date().getTime();
        var respostaExecutar = await awsManager.lambda.invokeFunction(
            region, "performance_" + i);
        var responseTime = new Date().getTime() - timeInvoke;
        var dados = JSON.parse(respostaExecutar.Payload);

        dados.timeInvoke = timeInvoke;
        dados.delay = dados.timeInit - dados.timeInvoke;
        dados.responseTime = responseTime;
        resultadoRegistros1.push(dados);

        somaTotalTipo1 += dados.delay;

        console.log("Executando ", i);
    }
    fs.writeFileSync(arquivoResultado1, JSON.stringify(resultadoRegistros1));
    console.log("Fim tipo 1, qtd:", resultadoRegistros1.length);

    console.log("Executando... tipo 2");
```

```
    for (var i = 1; i < quantidade + 1; i++){
        var timeInvoke = new Date().getTime();
        var respostaExecutar = await awsManager.lambda.invokeFunction(
            region, "performance_" + i);
        var responseTime = new Date().getTime() - timeInvoke;
        var dados = JSON.parse(respostaExecutar.Payload);
        dados.timeInvoke = timeInvoke;
        dados.delay = dados.timeInit - dados.timeInvoke;
        dados.responseTime = responseTime;
        resultadoRegistros2.push(dados);

        somaTotalTipo2 += dados.delay;

        console.log("Executando ", i);
    }
    fs.writeFileSync(arquivoResultado2, JSON.stringify(resultadoRegistros2));
    console.log("Fim tipo 1, qtd:", resultadoRegistros2.length);

    console.log("Media tipo 1: ", somaTotalTipo1 / quantidade);
    console.log("Media tipo 2: ", somaTotalTipo2 / quantidade);

    //excluir funções
    try {
        for (var i = 1; i < quantidade + 1; i++){
            var respostaDeletar = await awsManager.lambda.deleteFunction(
                region, "performance_" + i);
            console.log("Excluido", i);
        }
    }
    catch (errTry) {
        console.log(errTry.message);
    }
};


(async function (){
    for (var i = 1; i < 50; i++){
        console.log("**** Executando bateria ", i);
        await executarBateriaTeste();
    }
```

```
})();
```

Anexo III

Código fonte FaaS

```
"use strict";

const os = require("os");

var id = parseInt(Math.random() * 100000000).toString() + new Date()
    .getTime().toString();

module.exports.handler = function(message){
    return new Promise(function(resolve, reject){
        try {
            var timeInit = new Date().getTime();
            var response = {};
            var delay = parseInt(message.delay) || null;

            response.id = id;
            response.timeInit = timeInit;
            response.delay = delay;
            response.uptime = os.uptime();
            response.totalmem = os.totalmem();
            response.freemem = os.freemem();
            response.rss = process.memoryUsage().rss;
            response.cpu_qtd = os.cpus().length;
            response.hostname = os.hostname();
            var network_eth0 = os.networkInterfaces()["eth0"];
            if (network_eth0 && network_eth0[0]){
                response.newtwork = network_eth0[0].address;
            }
            else{
```

```
                response.newtwork = "";
            }
            var infoCPU = os.cpus()[0];
            response.cpu_model = infoCPU.model;
            response.cpu_speed = infoCPU.speed;
            response.platform = os.platform();
            response.release = os.release();

            if (delay){
                setTimeout(function(){
                    resolve(response);
                }, delay);
            }
            else{
                resolve(response);
            }
        }
        catch (error) {
            reject(error);
        }
    });
};
```

Anexo IV

Código fonte Automação AWS - AWS Manager

```
"use strict";

const AWS = require("aws-sdk");
const AWS_ElasticIP = require("./modules/ElasticIP");
const AWS_Instances = require("./modules/Instances");
const AWS_Lambda = require("./modules/Lambda");

var AWSManager = function(){
    var self = this;
    var cache_ec2 = {};
    var cache_lambda = {};

    this.config = require("../../config");

    this.elasticIP = new AWS_ElasticIP(self);
    this.instances = new AWS_Instances(self);
    this.lambda = new AWS_Lambda(self);

    this.getEC2 = function(p_region){
        var ec2 = null;

        if (!p_region){
            p_region = this.config.aws.defaultRegion;
        }

        ec2 = cache_ec2[p_region];
```

```
    if (ec2){
        return ec2;
    }
    else{
        ec2 = new AWS.EC2({accessKeyId: this.config.aws.accessKeyId,
            secretAccessKey: this.config.aws.secretAccessKey,
            region: p_region});
        cache_ec2[p_region] = ec2;
        return ec2;
    }
};

this.getLambda = function(p_region){
    var lambda = null;

    if (!p_region){
        p_region = this.config.aws.defaultRegion;
    }

    lambda = cache_lambda[p_region];
    if (lambda){
        return lambda;
    }
    else{
        lambda = new AWS.Lambda({accessKeyId: this.config.aws.accessKeyId,
            secretAccessKey: this.config.aws.secretAccessKey,
            region: p_region});
        cache_lambda[p_region] = lambda;
        return lambda;
    }
};

this.createImage = async function(p_region, p_configEC2, p_imageName){
    var instanceID = null;
    var responseAllocate = null;
    try {
        console.log("create ec2...");
        var responseCreateEC2 = await this.instances.create(
            p_region, p_configEC2);
        instanceID = responseCreateEC2.Instances[0].InstanceId;
```

```
            console.log("wait runnning... " + instanceID);
            await this.instances.waitRunning(p_region, instanceID);

            console.log("allocateAndAssociate public ip... " + instanceID);
            responseAllocate = await this.elasticIP.allocateAndAssociate(
                p_region, instanceID);

            console.log("wait stopped");
            await this.instances.waitStopped(p_region, instanceID);

            console.log("createImage...");
            var responseCreateImage = await this.instances.createImage(
                p_region, instanceID, p_imageName);
            console.log("ImageId: " + responseCreateImage.ImageId);

            return responseCreateImage.ImageId;
        }
        catch (errTry) {
            console.log("Err in create image: " + errTry);
            throw errTry;
        }
        finally {
            if (responseAllocate){
                console.log("disassociateAndReleaseByPublicIp");
                await this.elasticIP.disassociateAndReleaseByPublicIp(
                    p_region, responseAllocate.PublicIp);
            }
            if (instanceID){
                console.log("terminate ec2");
                await this.instances.terminate(p_region, [instanceID]);
            }
        }
    };
};

module.exports = AWSManager;
```

Anexo V

Código fonte Automação AWS - configurar Registry

```
"use strict";

const config = {};

config.ImageId = "ami-01e3b8c3a51e88954"; //linux v2
config.InstanceType = "t3.nano";
config.KeyName = "entreposto";
config.SubnetId = "subnet-0e0894a734069c3bb";
config.MinCount = 1;
config.MaxCount = 1;
config.SecurityGroupIds = ["sg-0cbf956ef953db5d5"]; //entreposto
config.BlockDeviceMappings = [
    {DeviceName: "/dev/xvda",
        Ebs: {
            VolumeType: "gp2",
            VolumeSize: 8,
            DeleteOnTermination: true
        }
    }
];
config.Monitoring = {Enabled:false};
config.TagSpecifications = [{ResourceType: "instance", Tags:
    [{Key: "Name", Value: "vm-build-registry"},
    {Key: "ambiente", Value: "build"}]}];

var commands = [];
```

```
commands.push("#!/bin/bash\n");
commands.push("sudo yum update");
commands.push("curl --silent --location https://rpm.nodesource.com/setup_10.x
    | sudo bash -");
commands.push("sudo yum -y install nodejs");
commands.push("cd /home/ec2-user");
commands.push("wget https://github.com/functions-io/functions-io-registry/
    archive/master.zip");
commands.push("unzip master.zip");
commands.push("rm master.zip");
commands.push("cd functions-io-registry-master");
commands.push("npm install");
commands.push("cd ..");
commands.push("chown -R ec2-user:ec2-user functions-io-registry-master");

var fileServie = "/lib/systemd/system/registry.service";
commands.push("echo '[Unit]' >> " + fileServie);
commands.push("echo 'Description=registry' >> " + fileServie);
commands.push("echo 'After=network.target' >> " + fileServie);
commands.push("echo '[Service]' >> " + fileServie);
commands.push("echo 'Environment=NODE_PORT=8080' >> " + fileServie);
commands.push("echo 'Type=simple' >> " + fileServie);
commands.push("echo 'User=ec2-user' >> " + fileServie);
commands.push("echo 'WorkingDirectory=/home/ec2-user/
    functions-io-registry-master' >> " + fileServie);
commands.push("echo 'ExecStart=/usr/bin/node /home/ec2-user/
    functions-io-registry-master/server/server' >> " + fileServie);
commands.push("echo 'Restart=on-failure' >> " + fileServie);
commands.push("echo '[Install]' >> " + fileServie);
commands.push("echo 'WantedBy=multi-user.target' >> " + fileServie);

commands.push("systemctl daemon-reload");
commands.push("systemctl enable registry");
commands.push("shutdown now");

config.UserData = Buffer.from(commands.join("\n")).toString("base64");

module.exports = config;
```

Anexo VI

Código fonte Automação AWS - configurar Runtime

```
"use strict";

const config = {};

config.ImageId = "ami-01e3b8c3a51e88954"; //linux v2
config.InstanceType = "t3.nano";
config.KeyName = "entreposto";
config.SubnetId = "subnet-0e0894a734069c3bb";
config.MinCount = 1;
config.MaxCount = 1;
config.SecurityGroupIds = ["sg-0cbf956ef953db5d5"]; //entreposto
config.BlockDeviceMappings = [
    {DeviceName: "/dev/xvda",
        Ebs: {
            VolumeType: "gp2",
            VolumeSize: 8,
            DeleteOnTermination: true
        }
    }
];
config.Monitoring = {Enabled:false};
config.TagSpecifications = [{ResourceType: "instance", Tags:
    [{Key: "Name", Value: "vm-build-runtime"},
    {Key: "ambiente", Value: "build"}]}];

var commands = [];
```

```
commands.push("#!/bin/bash\n");
commands.push("sudo yum update");
commands.push("curl --silent --location https://rpm.nodesource.com/setup_10.x
    | sudo bash -");
commands.push("sudo yum -y install nodejs");
commands.push("cd /home/ec2-user");
commands.push("wget https://github.com/functions-io/functions-io/
    archive/master.zip");
commands.push("unzip master.zip");
commands.push("rm master.zip");
commands.push("cd functions-io-master");
commands.push("npm install");
commands.push("cd ..");
commands.push("chown -R ec2-user:ec2-user functions-io-master");

var fileServie = "/lib/systemd/system/runtime.service";
commands.push("echo '[Unit]' >> " + fileServie);
commands.push("echo 'Description=registry' >> " + fileServie);
commands.push("echo 'After=network.target' >> " + fileServie);
commands.push("echo '[Service]' >> " + fileServie);
commands.push("echo 'Environment=NODE_PORT=8080' >> " + fileServie);
commands.push("echo 'Type=simple' >> " + fileServie);
commands.push("echo 'User=ec2-user' >> " + fileServie);
commands.push("echo 'WorkingDirectory=/home/ec2-user/
    functions-io-master' >> " + fileServie);
commands.push("echo 'ExecStart=/usr/bin/node /home/ec2-user/
    functions-io-master/server/server' >> " + fileServie);
commands.push("echo 'Restart=on-failure' >> " + fileServie);
commands.push("echo '[Install]' >> " + fileServie);
commands.push("echo 'WantedBy=multi-user.target' >> " + fileServie);

commands.push("systemctl daemon-reload");
commands.push("systemctl enable runtime");
commands.push("shutdown now");

config.UserData = Buffer.from(commands.join("\n")).toString("base64");

module.exports = config;
```

Anexo VII

Código fonte Automação AWS - criar Entreposto

```
"use strict";

const config = {};

config.ImageId = "ami-0922553b7b0369273";
config.InstanceType = "t3.nano";
config.KeyName = "entreposto";
config.SubnetId = "subnet-0e0894a734069c3bb";
config.MinCount = 1;
config.MaxCount = 1;
config.SecurityGroupIds = ["sg-0fb7fdee63c6c2eaf"]; //producao-publica
//config.BlockDeviceMappings = [{DeviceName: "/dev/sdh", Ebs: {VolumeSize: 12}}];
config.BlockDeviceMappings = [
    {DeviceName: "/dev/xvda",
        Ebs: {
            VolumeType: "gp2", //io1 | gp2 | sc1 | st1
            VolumeSize: 10,
            DeleteOnTermination: true
            //Encrypted: false,
            //Iops: 100
            //SnapshotId
            //KmsKeyId
        }
    }
];
config.Monitoring = {Enabled:true};
```

```
config.TagSpecifications = [{ResourceType: "instance", Tags:
    [{Key: "Name", Value: "vm-dinamica"},
    {Key: "ambiente", Value: "producao"}]}];

module.exports = config;
```

Anexo VIII

Código fonte Automação AWS - criar Registry

```
"use strict";

const config = {};

config.ImageId = "ami-0b1ad171881e8e43c";
//config.InstanceType = "t3.nano";
config.InstanceType = "c4.large";
config.KeyName = "entreposto";
config.SubnetId = "subnet-0e0894a734069c3bb";
config.MinCount = 1;
config.MaxCount = 1;
config.SecurityGroupIds = ["sg-0fb7fdee63c6c2eaf"]; //producao-publica
config.BlockDeviceMappings = [
    {DeviceName: "/dev/xvda",
        Ebs: {
            VolumeType: "gp2",
            VolumeSize: 8,
            DeleteOnTermination: true
        }
    }
];
config.Monitoring = {Enabled:true};
config.TagSpecifications = [{ResourceType: "instance", Tags:
    [{Key: "Name", Value: "vm-dinamica-registry"},
    {Key: "ambiente", Value: "producao"}]}];
```

```
module.exports = config;
```

Anexo IX

Código fonte Automação AWS - criar Runtime

```
"use strict";

const config = {};

config.ImageId = "ami-073846a509c26dee8";
//config.InstanceType = "t3.nano";
config.InstanceType = "c4.large";
config.KeyName = "entreposto";
config.SubnetId = "subnet-0e0894a734069c3bb";
//config.SubnetId = "subnet-0a7427140c2c2441b"; //zona diferente
config.MinCount = 1;
config.MaxCount = 1;
config.SecurityGroupIds = ["sg-0fb7fdee63c6c2eaf"]; //producao-publica
config.BlockDeviceMappings = [
    {DeviceName: "/dev/xvda",
        Ebs: {
            VolumeType: "gp2",
            VolumeSize: 8,
            DeleteOnTermination: true
        }
    }
];
config.Monitoring = {Enabled:true};
config.TagSpecifications = [{ResourceType: "instance", Tags:
    [{Key: "Name", Value: "vm-dinamica-runtime"},
    {Key: "ambiente", Value: "producao"}]}];
```

```
module.exports = config;
```