UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Câmpus de São José do Rio Preto

Igor Andrade Brito

Serverless-simulator - Simulador de computação sem servidores

São José do Rio Preto
2024

Igor Andrade Brito

Serverless-simulator - Simulador de computação sem servidores

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação (PPGCC), do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Câmpus de São José do Rio Preto.

Orientador: Profª. Profª. Drª. Renata Spolon Lobato

São José do Rio Preto
2024

B862s

Brito, Igor Andrade
Serverless-simulator - Simulador de computação sem servidores, / Igor Andrade Brito. -- São José do Rio Preto, 2024
64 p.

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp), Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto
Orientadora: Renata Spolon Lobato

1. Ciência da computação. 2. Computação em nuvem. 3. Simulação. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca do Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

Igor Andrade Brito

Serverless-simulator - Simulador de computação sem servidores

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação (PPGCC), do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Câmpus de São José do Rio Preto.

Comissão Examinadora

Profª. Drª. Renata Spolon Lobato
UNESP – Câmpus de São José do Rio Preto
Orientadora

Prof. Dr. Rodrigo Capobianco Guido
UNESP – Câmpus de São José do Rio Preto

Prof. Dr. Henrique Dezani
FATEC – Câmpus de São José do Rio Preto

São José do Rio Preto
25 de março de 2024

AGRADECIMENTOS

Conseguir cursar minha pós graduação em ciência da computação foi uma grande realização pessoal. Foram 3 anos de muitos desafios e aprendizado.

Primeiramente, gostaria de agradecer aos meus pais por sempre me apoiarem e acreditarem em mim, agradeço em especial a minha mãe que é minha maior incentivadora, me ajuda diariamente e é a grande responsável por essa conquista.

Agradeço ao IBILCE, seus funcionários e corpo acadêmico pelo conhecimento compartilhado. Gratidão especial à minha orientadora, Profª. Drª Renata Spolon pela compreensão e ensinamentos.

Por fim, gostaria de estender meus agradecimentos a todos os colegas, amigos e familiares que me acompanharam nesta jornada. Suas palavras de incentivo, apoio moral e compreensão foram inestimáveis e me deram forças nos momentos mais difíceis.

Este trabalho não teria sido possível sem o apoio e contribuição dessas pessoas, e por isso, expresso minha mais profunda gratidão a todos que tornaram este sonho uma realidade.

RESUMO

A computação sem servidores (serverless) é uma arquitetura de computação em nuvem que ganhou crescente interesse devido à sua capacidade de escalabilidade automática, pagamento baseado no uso e execução de aplicativos na nuvem, eliminando a necessidade de gerenciar infraestrutura de servidores. Neste trabalho foi realizado um estudo sobre computação em nuvem, computação sem servidores e seus simuladores juntamente com uma revisão sistemática da literatura. Após isso é apresentado o desenvolvimento de um simulador para computação sem servidores, o serverless-simulator foi projetado para reproduzir com precisão o comportamento de plataformas serverless em uma variedade de cenários operacionais, permitindo análises detalhadas do desempenho. Foram realizados diversos testes para avaliar o desempenho do simulador, abrangendo uma variedade de cenários e cargas de trabalho, seguido da comparação dos resultados obtidos com as expectativas estabelecidas. Os resultados dos experimentos atenderam os resultados esperados, bem como validou a utilização do sistema para replicar plataformas serverless.

Palavras–chave: Computação em nuvem. Computação sem servidores. Simulador. Simulador de computação sem servidores.

ABSTRACT

Serverless computing is a cloud computing architecture that has gained increasing interest due to its ability for automatic scalability, pay-per-use pricing model, and running of applications in the cloud without the need to manage server infrastructure. This work presents a study on cloud computing, serverless computing, and their simulators along with a systematic literature review. Subsequently, the development of a serverless computing simulator is presented. The serverless-simulator was designed to accurately replicate the behavior of serverless platforms across various operational scenarios, enabling detailed performance analyses. Several tests were carried out to evaluate the simulator's performance across various scenarios and workloads, followed by a comparison of the results obtained with the established expectations. The experimental results met the expected outcomes and validated the system's utility in replicating serverless platforms.

Keywords: Cloud computing. Serverless computing. Simulator. Serverless simulator.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BaaS	Backend como Serviço
CPU	Unidade Central de Processamento
FaaS	Funções como Serviço
FIFO	Primeiro inserido é o primeiro a ser retirado
HTTP	Hypertext Transfer Protocol
IaaS	Infraestrutura como Serviço
iSPD	Iconic Simulator of Parallel and Distributed systems
NIST	Instituto Nacional de Padrões e Tecnologia dos EUA
PaaS	Plataforma como Serviço
QoS	Qualidade de Serviço
SaaS	Software como Serviço
TI	Tecnologia da Informação
URI	Identificador único de recurso
VM	Máquina virtual
VMM	Monitor de máquina virtual

SUMÁRIO

1 INTRODUÇÃO

A sociedade atualmente está completamente conectada, o uso da Internet e seus diversos serviços diferentes só cresce, e por conta disso existe o interesse constante na melhoria e aperfeiçoamento desses serviços, seja em relação a facilidade de desenvolvimento, gerenciamento e implantação, redução de custos, aprimoramento do desempenho e maior segurança.

Por conta disso, surgiu um modelo de computação baseado na computação em nuvem, chamado de *serverless* (sem servidores) (RAJAN, 2020). Este método está disponível nas principais provedoras de computação em nuvem e é utilizado por diversas empresas em seus negócios e também para fins de pesquisa ou industriais.

Dessa maneira, este trabalho teve como foco principal o estudo sobre a computação em nuvem e a computação sem servidores, junto com seus principais conceitos com a finalidade de desenvolver um modelo que seja capaz de fazer uma simulação de um sistema *serverless* de acordo com os parâmetros informados pelos usuários de forma fácil e intuitiva.

1.1 Motivação e objetivos

A análise do desempenho e comportamento de sistemas computacionais é importante pois pode ajudar na sua melhoria de diversos modos diferentes, como reduzir custos, melhorar performance e eficiência, além de permitir um exame detalhado sem a necessidade de efetivamente possuir o sistema. Portanto, a motivação para a realização deste estudo foi aprofundar sobre sistemas *serverless* e para isso precisamos de ferramentas capazes de simular o comportamento desse tipo de sistema. A intenção é trazer contribuições para os usuários de forma que seja uma ferramenta com uma interface gráfica intuitiva para que seja fácil para qualquer usuário utilizar o sistema sem problemas.

Deste modo, este trabalho teve como objetivo propor um modelo para simulação de computação sem servidores.

1.2 Organização do texto

A estrutura do trabalho é organizada em 6 capítulos, sendo este primeiro para introdução.

No **Capítulo 2** são apresentados os conceitos da computação em nuvem, assim como suas características, definições e modelos.

No **Capítulo 3** é apresentada a computação sem servidores e seus modelos de serviço, com uma aprofundamento sobre as Funções como Serviço

No **Capítulo 4** são apresentados simuladores de nuvem e simuladores das funções *serverless*.

No **Capítulo 5** é explicado como foi o desenvolvimento do trabalho, os testes usados para validar o sistema e seus resultados

No **Capítulo 6** está descrito a conclusão e trabalhos futuros.

2 COMPUTAÇÃO EM NUVEM

Computação em nuvem (*cloud computing*) são os diferentes tipos de serviço oferecidos e gerenciados pela Internet e se tornou o modelo de negócio dominante no mercado de TI. Esses serviços podem ser de processamento, armazenamento, infraestrutura, aplicações. Com a computação em nuvem o modelo que antes era focado no produto se tornou algo global focado no serviço passando de TI como produtos para TI como serviços, alterando o jeito de desenvolvimento, implantação, manutenção, atualização e pagamento (SUNYAEV, 2020). Ela permite que os recursos sejam acessados de acordo com a demanda, de qualquer lugar e em qualquer momento, tanto por pessoas quanto empresas.

Atualmente, por exemplo, nós contamos regularmente com muitos serviços em nuvem como: armazenamento de arquivos (Dropbox, OneDrive, Google Drive), escrita de documentos (Office 365, Google Docs). As empresas também contam com sistemas de computação em nuvem, como plataformas de desenvolvimento e implantação de aplicações (Docker, Kubernetes).

2.1 Computação em Nuvem: Definição e conceitos

Segundo (SUNYAEV, 2020) a definição de computação em nuvem mais aceita é a do Instituto Nacional de Padrões e Tecnologia dos EUA, o NIST (*National Institute of Standards and Technology*) que diz que a computação em nuvem é um modelo que permite acesso onipresente de acordo com a demanda do usuário a um conjunto de recursos computacionais compartilhados que podem ser ajustados rapidamente a qualquer momento e de qualquer lugar do mundo pela Internet com um esforço mínimo de gestão por parte do solicitante (MELL; GRANCE, 2011).

2.1.1 Características

Segundo a NIST, o modelo de computação em nuvem possui cinco características, que são: autoatendimento sob demanda, acesso remoto pela rede, agrupamento de recursos, rápido escalonamento e serviço medido, explicados abaixo:

- **Autoatendimento sob demanda:** Os sistemas permitem a escolha e alteração dos serviços computacionais sem necessidade de interação humana. Essa alteração pode ser qualquer coisa, desde aumentar ou diminuir capacidade computacional e armazenamento até pagamentos.
- **Acesso remoto pela rede:** os sistemas utilizam a Internet e podem ser acessados por qualquer aparelho (*smartphone*, *tablet*, computadores) em qualquer horário e lugar.
- **Agrupamento de recursos:** os recursos computacionais da provedora são agrupados para atender a múltiplos clientes sendo que esses recursos podem ser físicos ou virtuais e são alocados dinamicamente de acordo com a demanda de cada cliente. Não existe uma dependência de localização, sendo que normalmente os clientes não têm controle ou conhecimento de onde seus recursos estão alocados fisicamente. Entretanto, em um alto grau de abstração, (país, estado ou *datacenter*) é possível especificar a localização, inclusive as empresas com grande volume de dados normalmente tem profissionais cuja função é saber onde estão os dados e controlar a organização e proteção dos dados (SUNYAEV, 2020).
- **Rápido escalonamento:** a quantidade de recursos é flexível, podendo ser alterada para ajustar a demanda e possibilitando o aumento ou redução dos mesmos de uma maneira ágil. Dessa forma, o cliente tem a impressão que os recursos são quase ilimitados e estão disponíveis em qualquer momento.
- **Serviço medido:** os sistemas de computação em nuvem automaticamente controlam e otimizam os recursos sendo que é possível monitorar, controlar e reportar o seu uso, fornecendo transparência tanto para o cliente quanto para o provedor.

Com a computação em nuvem não existe gasto com a compra de *hardware* e *software* e com a instalação e execução de *datacenters* locais. O custo com eletricidade também é reduzido, pois os servidores locais demandam de energia e resfriamento constante, assim como o custo com especialistas para o gerenciamento da infraestrutura.

A maior parte dos serviços de computação em nuvem é fornecida sob demanda ou por auto-serviço, inclusive até grandes quantidades de recursos computacionais podem ser provisionados em minutos, de forma muito fácil,

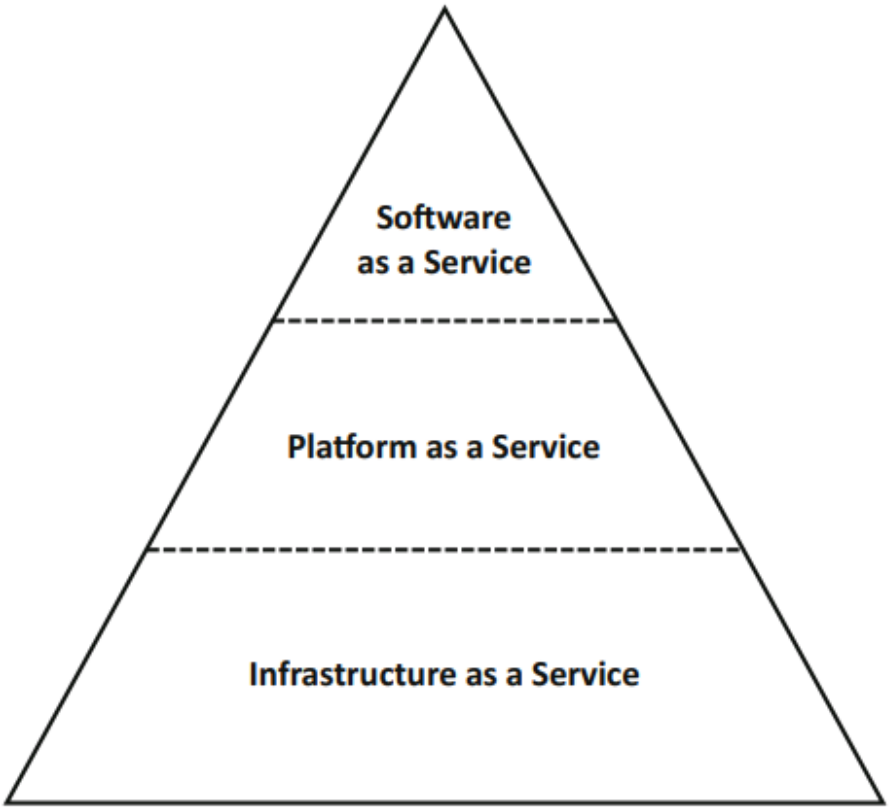
fornecendo às empresas uma flexibilidade muito grande. Por conta dessa elasticidade, a quantidade correta de recursos podem ser fornecidos.

Um ponto importante é que quando se utiliza a computação em nuvem está trocando um provável grande investimento em um *datacenter* ou servidor físico sem ter o conhecimento sobre a utilização por um modelo em que o pagamento é feito de acordo com seu consumo e somente quando os recursos computacionais forem consumidos, podemos então ter um custo variável mais baixo do que o normal.

2.1.2 Modelos de serviço

Os modelos de serviço apresentados pela NIST são divididos em três: Infraestrutura como Serviço (*Infrastructure as a Service* - IaaS), Plataforma como Serviço (*Platform as a Service* - PaaS) e Software como Serviço (*Software as a Service* - SaaS) (Figura 1).

Figura 1: Modelos de serviço da computação em nuvem



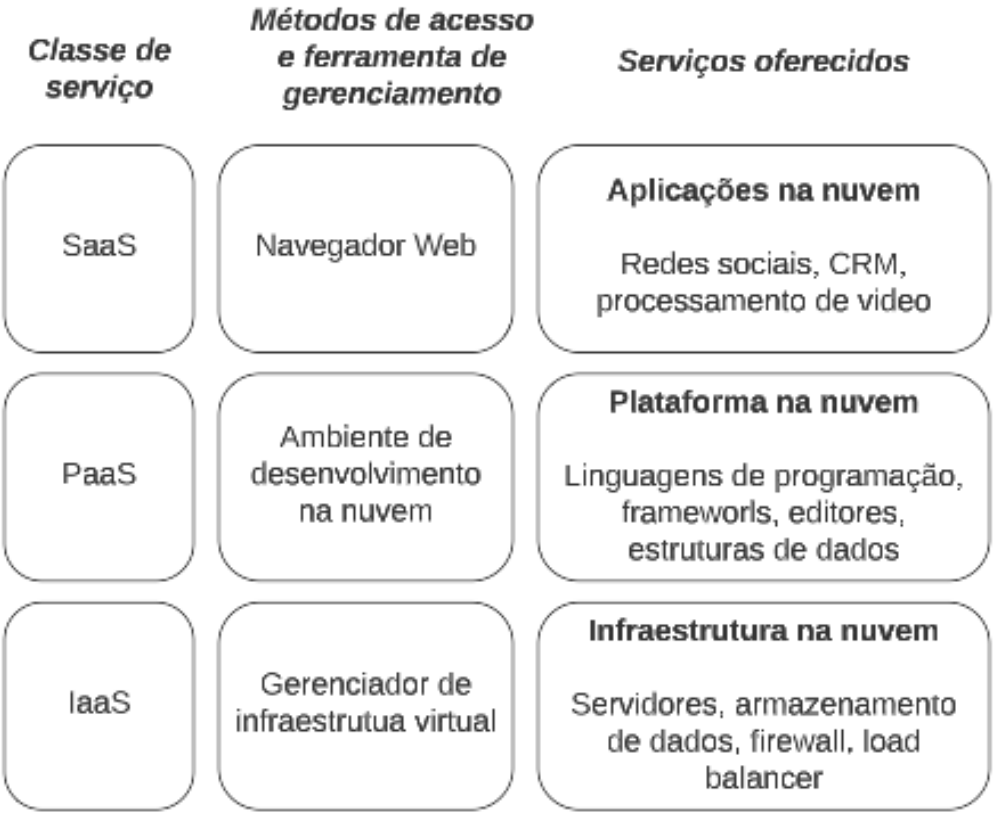
Fonte: Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies. SUNYAEV, 2020

- ***Infrastructure as a Service*:** Nesta classe de serviço o provedor oferece ao cliente recursos como processamento, armazenamento, redes e outros serviços computacionais fundamentais com os quais ele consegue executar e implantar seu software para qualquer finalidade arbitrária, sendo a este delegada a responsabilidade de gerenciamento dos recursos. Os clientes não têm autonomia para controlar ou gerenciar as camadas inferiores de infraestruturas mas têm controle sobre os sistemas operacionais, armazenamento e as aplicações implantadas. Tipicamente o provedor desse serviço possui uma grande infraestrutura física e disponibiliza para o cliente pequenas partes de recursos virtuais. Alguns exemplos de IaaS são: servidores na nuvem que podem ser configurados dinamicamente como necessário; sistemas de *backups* e recuperação de arquivos e dados; serviços de armazenamento massivo de dados que podem ser usados por aplicações, armazenamento de arquivos, *backups*; gerenciamento das plataformas de infraestrutura em nuvem.
- ***Platform as a Service*:** Neste modelo de serviço o cliente consegue implantar aplicações em uma infraestrutura usando linguagens de programação, bibliotecas, serviços e ferramentas suportadas pela provedora. Além disso, o cliente não consegue gerenciar as camadas inferiores de infraestrutura, mas tem controle sobre as aplicações implantadas e configurações sobre o ambiente de hospedagem delas. Alguns exemplos de PaaS são: plataformas para desenvolvimento e testes; serviços de bancos de dados relacionais e NoSQL; ambientes para rodar aplicações específicas em tempo de execução.
- ***Software as a Service*:** Este modelo de serviço consiste na oferta de *softwares* por parte do provedor. Essas aplicações, geralmente são acessíveis para vários clientes e dispositivos através de uma interface web ou programa e estão sendo executadas em uma infraestrutura na nuvem. O cliente não tem autonomia para controlar ou gerenciar as camadas inferiores de infraestruturas da nuvem. A ideia deste tipo de serviço é oferecer uma alternativa ao modelo tradicional de aquisição de softwares, em que o usuário adquire o *software* e o instala em seu computador pessoal. Esta é a classe de serviço voltada para usuários comuns de computação

(RITTINGHOUSE;RANSOME, 2009), e alguns de seus exemplos são editores de documentos, planilhas e apresentações; plataformas de email; ferramentas para colaboração em grupo dentro ou fora do ambiente empresarial.

Na Figura 2 são apresentadas as classes de serviço com seus métodos de acesso, ferramentas de gerenciamento e serviços oferecidos.

Figura 2: Modelos de serviço com seus métodos de acesso, ferramentas de gerenciamento e serviços oferecidos



Fonte: Cloud Computing: Principles and Paradigms. BUYYA; BROBERG; GOSCINSKI, 2011. Traduzido pelo autor

Quanto ao modelo de implantação, um sistema de computação em nuvem pode ser classificado como público, privado, comunitário e híbrido.

- **Nuvem pública:** são sistemas que oferecem serviços a qualquer pessoa que queira utilizá-los (não necessariamente gratuitos). Podem ser possuídos, gerenciados e operados por uma empresa, universidade, organização governamental ou alguma combinação deles.
- **Nuvem privada:** são sistemas que têm seus serviços disponíveis a uma pessoa ou um grupo restrito de pessoas e normalmente são utilizados em ambientes corporativos. Podem ser possuídos, gerenciados e operados por uma ou mais organizações, um terceiro externo ou uma combinação deles.
- **Nuvem comunitária:** são sistemas que têm seu uso exclusivo para uma comunidade específica de clientes que possuem objetivos em comum. Podem ser possuídos, gerenciados e operados por uma ou mais organizações dentro da comunidade, um terceiro externo ou uma combinação deles.
- **Nuvem híbrida:** é a composição de dois ou mais dos modelos citados acima. Suas infraestruturas individuais continuam porém estão conectadas por alguma tecnologia padronizada ou própria e que permite a troca de dados e aplicações entre as infraestruturas conectadas.

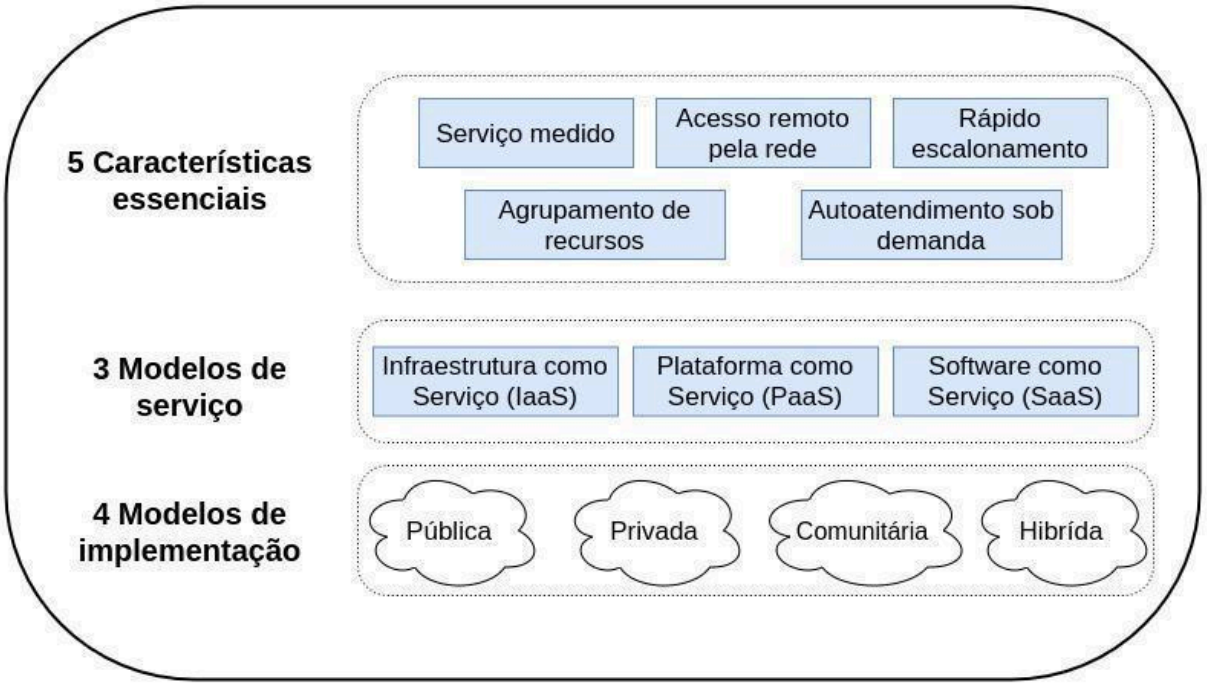
Tem se observado um aumento das pesquisas em computação em nuvem nos últimos anos e um dos fatores que influenciaram nesse aumento foi a integração entre as classes de serviço. Essa integração ocorre com a organização, virtualização e automação dos recursos computacionais, como: servidores, armazenamento e rede. A virtualização permite a integração e o gerenciamento de diferentes classes de serviços, seja em nuvens públicas, privadas, híbridas ou comunitárias, permitindo que se tenha a disponibilização desses recursos para os diferentes tipos de nuvem.

Na Figura 3 é apresentada uma síntese do modelo NIST de computação em nuvem, com suas características essenciais, modelos de serviços e modelos de implantação.

2.2 Diferenças da computação em nuvem para computação em grade

As grades computacionais foram desenvolvidas com o intuito de melhorar o desempenho de sistemas interligando vários computadores e seu conceito, segundo Wilkinson (2010): “[...] uso distribuído e interconectado de computadores e coleção de recursos para alcançar alta performance computacional na distribuição de recursos”.

Figura 3: Modelo NIST de computação em nuvem



Fonte: Elaborado pelo autor

Sendo assim, a computação em grade proporciona o compartilhamento de recursos e resolução de problemas coordenados em uma organização virtual dinâmica em larga escala e geograficamente distribuída. As grades permitem uma arquitetura distribuída e ponto a ponto sendo que o objetivo de tal paradigma é permitir o compartilhamento de recursos em ambientes dinâmicos e distribuídos.

Nuvens e Grades têm muitos pontos em comum como a ideia, arquitetura e a tecnologia. A evolução da computação em nuvem dá-se muito por conta do conceito de entregar recursos computacionais e armazenamentos vindo das grades com o objetivo de tentar ser mais econômico e oferecer outros serviços e recursos mais abstratos. A ideia principal, tanto da grade quanto da nuvem, é a mesma, de reduzir o custo de computação, aumentar a confiabilidade e flexibilidade. Os dois modelos compartilham algumas funções porém também diferem em muitos aspectos como a segurança, modelo de programação, aplicações, modelo de negócio e de computação.

Em um modelo de negócio baseado na computação em nuvem, normalmente o cliente paga o provedor de acordo com o seu consumo baseado em uma escala econômica para definir preços a serem pagos e os lucros da provedora. Em

contraste disso, o modelo de negócio das grades é orientado em projetos onde os usuários ou a comunidade tem um certo número de unidades de serviço (como tempo de CPU) que podem usar para atingir os objetivos do projeto. Além disso, a maioria dos sistemas em nuvem possuem um *data center* dedicado pertencente a mesma organização, e dentro desse data center estão o *hardware*, as configurações de *software* e as plataformas suportadas sendo mais homogêneo. Já as grades são construídas assumindo que os recursos são heterogêneos onde cada grade pode ter a sua administração e autonomia de operação.

2.3 Virtualização

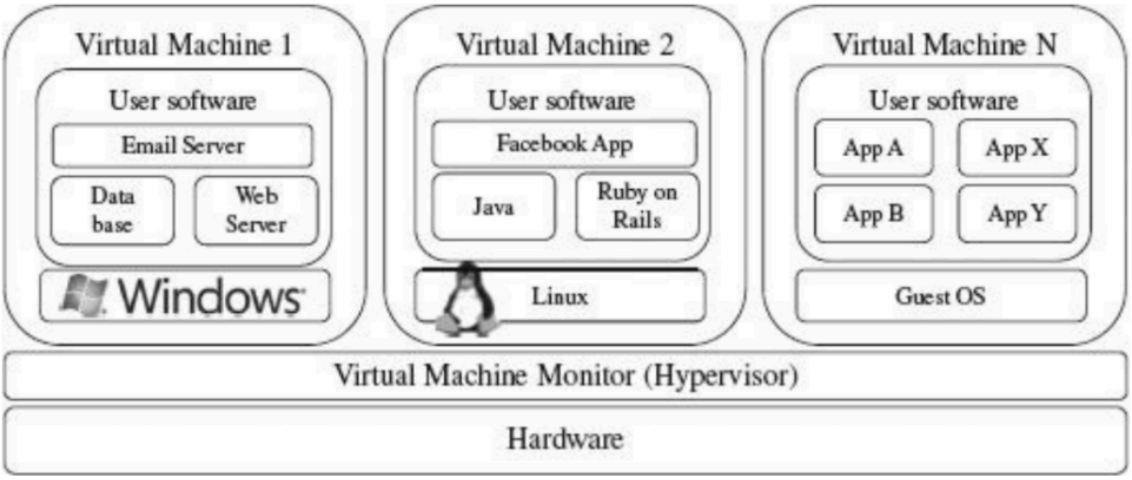
A virtualização é uma parte essencial dos sistemas de computação em nuvem, pois sua infraestrutura física é composta por um grande número de máquinas conectadas que formam uma grade computacional e com isso temos o problema das máquinas ficarem boa parte do tempo ociosas. Portanto a virtualização nada mais é do que possibilitar que uma única máquina física execute mais de uma máquina virtual. Esse agrupamento de máquinas virtuais permite estabelecer um novo modelo de negócio onde os recursos da máquina física são locados e por isso o gerenciamento otimizado desses recursos faz-se necessário, para reduzir os custos de hardware e processamento e aproveitar melhor os recursos disponíveis.

O responsável por fazer a instanciação e gerenciamento das máquinas virtuais e a comunicação entre elas e o *hardware* é chamado de Virtual Machine Monitor (VMM) ou *hypervisor*. Na Figura 4 é demonstrado um caso de virtualização com três máquinas virtuais com diferentes sistemas operacionais em uma mesma máquina física.

2.3.1 Hypervisors

Um *hypervisor* requer uma pequena quantidade de especificações para inicializar um sistema operacional convidado: uma imagem do sistema operacional para inicializar e uma configuração básica, como por exemplo, disco, memória, processadores e dispositivo de rede. O disco e o dispositivo de rede normalmente são mapeados para o disco físico da máquina e para o dispositivo de rede físico.

Figura 4: Hospedagem de máquinas virtuais com diferentes sistemas operacionais



Fonte: Cloud Computing: Principles and Paradigms. BUYYA; BROBERG; GOSCINSKI, 2011

O *hypervisor* é responsável por fazer o monitoramento da virtualização, ele garante que cada máquina virtual seja desvinculada uma da outra de forma que uma não necessite da liberação de recursos ou de interação com a outra para realizar qualquer uma de suas funcionalidades. Dessa forma, ele mantém o isolamento das tarefas das máquinas virtuais sendo possível isolar alguma máquina que apresentar falhas para que somente ela seja prejudicada com o problema e as outras máquinas continuem funcionando normalmente.

Cada máquina virtual é tratada de forma independente, cada uma com seu sistema operacional, memória, discos e processadores, e para o usuário final isto se torna transparente pois ele não tem e nem precisa ter conhecimento que a máquina que está utilizando é física ou virtual. Desta forma também é possível que vários sistemas operacionais diferentes sejam executados em apenas uma máquina física. Portanto, a virtualização é a principal tecnologia da computação em nuvem e permite que processadores, memória, disco e rede sejam tratados como um agrupamento de recursos para que então possam ser utilizados sob demanda (Chieu et al., 2009).

Por conta dessa importância da virtualização para a computação em nuvem existem diversos virtualizadores disponíveis para criar e gerenciar máquinas virtuais. A seguir são apresentados os *hypervisors* mais conhecidos comercial e cientificamente.

- **VMWare:** Foi desenvolvido e é mantido pela empresa de mesmo nome VMWare Inc., sendo ela uma pioneira no mercado de virtualização. A ferramenta possui algumas distribuições diferentes, desde ambientes para virtualização de desktop, como o VMWare Workstation, até para a implantação de servidores de grande porte, como o VMWare ESX Server, que consiste em um sistema operacional especializado em virtualização (VMWARE, 2021).
- **KVM:** O KVM (*Kernel-based virtual machine*) é um subsistema de virtualização dos sistemas operacionais Linux. Faz parte do kernel Linux desde a versão 2.6.20, portanto têm suporte native para diversas distribuições. Por ser uma ferramenta diretamente acoplada ao núcleo do sistema operacional, permite a virtualização assistida por *hardware*, que implica em ganhos de performance em relação à paravirtualização (KVM, 2021).
- **Xen:** O Xen foi inicialmente desenvolvido como um projeto *open-source* pela Universidade de Cambridge e depois foi comprado pela Citrix System Inc, que suporta o desenvolvimento em código aberto e distribui em paralelo versões de código fechado do produto. O Xen tem uma boa base de produtos e serviços que a utilizam comercialmente, entre eles estão o Citrix XenServer e Oracle VM (XEN. 2021).
- **Microsoft Hyper-V:** É uma ferramenta desenvolvida pela Microsoft e encontra-se integrada em distribuições do sistema operacional Windows e Windows Server. Também permite virtualização assistida por *hardware*. (HYPER-V, 2021)

2.3.2 Sistemas gestores

Os sistemas gestores são responsáveis pela automação do processo de virtualização, ele trabalha de forma distribuída realizando o escalonamento das instâncias de máquinas virtuais nas máquinas físicas que fazem parte do sistema. A seguir são apresentados algumas soluções prontas para gestão de sistemas de computação em nuvem:

- **Apache VCL:** O Apache VLC (*Virtual Computing Lab*) é um laboratório de computação virtual que foi inicialmente criado por pesquisadores da *North Carolina State University* para oferecer ambientes customizados para os usuários do laboratório. O software usado foi disponibilizado como *open-source* e depois incorporado pela Apache. Os ambientes oferecidos pelo *software* podem variar de uma simples máquina virtual a um cluster com muitas máquinas para simulações complexas de alto desempenho. Possui interface web e suporte para *hypervisor* VMware (ESX, ESXi, e servidor) (VLC, 2021).
- **Citrix Essentials:** O Citrix Essentials tem um foco maior no gerenciamento e automação dos *datacenters*. Tem suporte para os *hypervisors* XenServer e Hyper-V, providência três níveis de alta disponibilidade (recuperação pelo reinício da VM, recuperação ativando uma VM duplicada pausada e VM duplicada rodando continuamente) e proteção de dados com o Citrix Consolidated Backup.
- **Eucalyptus:** O Eucalyptus foi um dos primeiros projetos *open-source* com foco na construção de nuvens IaaS. Foi desenvolvido com o intuito de disponibilizar quase idêntica funcionalmente ao Amazon Web Services APIs, portanto os usuários podem interagir com o software usando as mesmas ferramentas usadas para acessar o Amazon EC2.
- **Nimbus:** O nimbus foi criado com base no Framework Globus, e seu diferencial é que ele fornece uma interface de acesso para o framework Globus Web Services Resource (WSRF). Possui suporte para *hypervisors* Xen e KVM.
- **OpenNebula:** O OpenNebula foi inicialmente criado para gerenciar infraestruturas locais, mas também incluia interfaces públicas que viabilizaram a construção de nuvens públicas. Ele é um dos gerenciadores *open-source* com maior número de recursos. Possui uma arquitetura modular e conectável, permite alocação de recursos dinâmicos e suporte para *hypervisors* Xen, KVM e VMware (OPENNEBULA, 2021);
- **VMWare vSphere:** O VMWare vSphere é uma ferramenta que transforma a infraestrutura em uma nuvem privada e seu diferencial é ser muito rico em recursos por conta da organização oferecer serviços em diversos níveis de arquitetura. A alocação de recursos possui regras pré-definidas e é

gerenciada pelo *Distributed Resource Scheduler* (DRS). Ele possui compatibilidade com os *hypervisors* VMWare ESX e ESXi.

- **OpenStack:** O OpenStack é um gestor de código aberto para criar nuvens públicas e privadas. É uma plataforma com diferentes projetos *open-source* que são mantidos pela comunidade e que juntos formam um dos gestores de nuvem mais importantes e que é mantido por mais de 200 empresas. Ele possui suporte aos *hypervisors* VMWare, KVM, QEMU, Xen, Microsoft Hyper-V e Docker. Suporta a integração com serviços de computação em nuvem da Amazon (OPENSTACK, 2021).

  Na Tabela 1 são mostrados os principais gestores de nuvens, *hypervisors* suportados e algumas de suas características.

2.4 Considerações finais

Neste capítulo foram apresentados os principais conceitos e definições da computação em nuvem, além de suas características, modelos de serviço e implantação, e sua virtualização e ferramentas necessárias. No próximo capítulo serão apresentados os conceitos de computação sem servidores e de funções como serviço.

Tabela 1 - Gestores de nuvem

	LICENÇA	PLATAFORMA DE INSTALAÇÃO	HYPERVISOR(S)	VIRTUALIZAÇÃO DO ARMAZENAMENTO	INTERFACE PARA NUVEM PUBLICA	REDE VIRTUAL	ALOCAÇÃO DINAMICA DE RECUSROS	RESERVA ANTECIPADA DE CAPACIDADE	ALTA DISPONIBILIDADE	PROTEÇÃO DE DADOS
Apache VCL	Apache v2	Multiplataforma (Apache/Php)	VMware ESX, ESXi, Server	Não	Não	Sim	Não	Sim	Não	Não
AppLogic	Proprietário	Linux	Xen	Global Volume Store (GVS)	Não	Sim	Sim	Não	Sim	Sim
Citrix Essentials	Proprietário	Windows	XenServer, Hyper-V	Citrix Storage Link	Não	Sim	Sim	Não	Sim	Sim
Enomaly ECP	GPL v2	Linux	Xen	Não	Amazon EC2	Sim	Não	Não	Não	Não
Eucalyptus	BSD	Linux	Xen, KVM	Não	EC2	Sim	Não	Não	Não	Não
Nimbus	Apache v2	Linux	Xen, KVM	Não	EC2	Sim	Via integração com OpenNebula	Sim (Via integração com OpenNebula)	Não	Não
OpenNEbula	Apache v2	Linux	Xen, KVM	Não	Amazon EC2, Elastic Hosts	Sim	Sim	Sim (via Haizea)	Não	Não
OpenPEX	GPL v2	Multiplataforma (Java)	XenServer	Não	Não	Não	Não	Sim	Não	Não
oVirt	GPL v2	Fedora Linux	KVM	Não	Não	Não	Não	Não	Não	Não
Platform ISF	Proprietário	Linux	Hyper-V, XenServer, VMWare ESX	Não	EC2, IBM, CoD, HP Enterprise Services	Sim	Sim	Sim	Indefinido	Indefinido
Plataform VMO	Proprietário	Linux, Windows	XenServer	Não	Não	Sim	Sim	Não	Sim	Não
VM Ware vSphere	Proprietário	Linux, Windows	VMware ESX, ESXi	VMware vStorage VMFS	VMWare vCloud partners	Sim	VMware DRM	Não	Sim	Sim

Fonte: Cloud Computing: Principles and Paradigms. BUYYA; BROBERG; GOSCINSKI, 2011.
Traduzido pelo autor

3 FUNCTION AS SERVICE

3.1 Serverless

Serverless computing ou computação sem servidores é uma plataforma que fornece desenvolvimento e implantação eficiente de aplicações sem a necessidade de gerenciar as camadas de infraestrutura (NUPPONEN; TAIBI, 2020), facilitando e permitindo que os desenvolvedores tenham um foco maior na lógica do negócio sem se preocupar com escalonamento e providenciamento de recursos e infraestrutura pois tecnicamente a aplicação está executando em um servidor externo garantido pela provedora. Nesse modelo, a provedora aloca e gerencia os servidores e os códigos são executados em *containers* gerados a partir de gatilhos definidos. Esse modelo de computação foi introduzido inicialmente pela Amazon Lambda em 2014 e após isso outras empresas como Google e Microsoft a adoram em 2016.

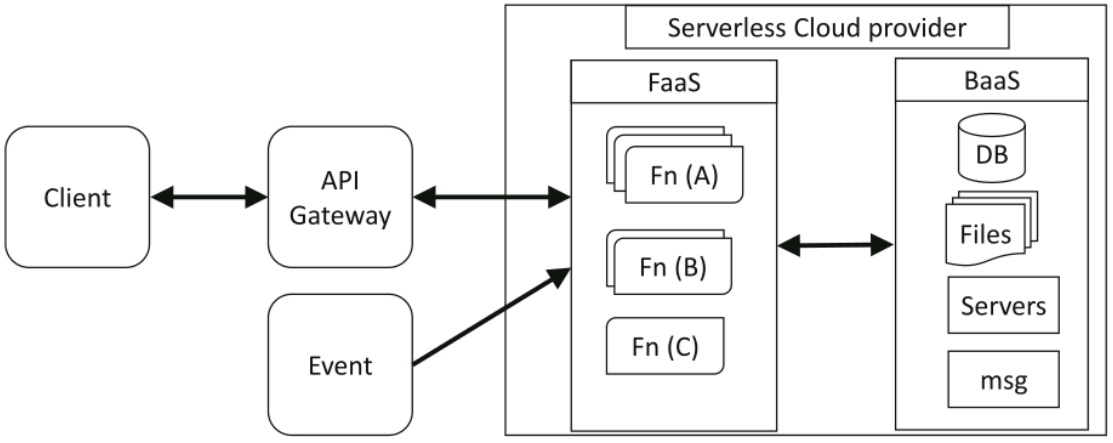
Na visão de (JONAS, et al, 2019), para um serviço ser considerado *serverless* ele deve escalar automaticamente sem a necessidade de provisionamento explícito e ter o pagamento baseado em seu uso. Em qualquer plataforma de computação sem servidores o usuário deve somente escrever uma função em alguma linguagem de alto nível, selecionar o evento que a inicializa, como carregar uma imagem no storage e deixar o sistema *serverless* lidar com todo o resto: seleção da instância, escalonamento, implantação, tolerância a falhas, monitoramento, configurações de segurança, etc (JONAS, et al, 2019). O modelo permite aos desenvolvedores dissolver grandes aplicações em pequenas funções possibilitando os componentes escalar individualmente (MCGRATH; BRENNER, 2017), e pode ser divido em 2 categorias: *Backend-as-Service* (BaaS) e *Function-as-Service* (FaaS) (NUPPONEN; TAIBI, 2020):

- ***Backend-as-Service*:** é um serviço para substituir a parte do servidor. Ele permite que todos elementos de um serviço de *backend* seja terceirizado e que os desenvolvedores possam manter toda a lógica do aplicativo e de regras de negócio no *frontend*. São exemplos desse tipo de serviço sistemas de autenticação, gerenciamento de bases de dados, armazenamento de arquivos e envio de mensagens, e um exemplo na parte empresarial é o Google Firebase, que é um banco de dados totalmente gerenciado com vários serviços diferentes que podem ser usados diretamente nas aplicações.

- ***Function-as-Service*:** é a implementação mais conhecida e importante da computação sem servidores, também conhecido por *serverless functions*. Com esse modelo os desenvolvedores precisam somente implementar o código de uma função de pequena duração e definir os gatilhos (*trigger*) para executá-la (NUPPONEN; TAIBI, 2020). Essas funções tipicamente descrevem pequenas partes de uma aplicação completa e seu tempo de execução normalmente é limitado. Elas não ficam constantemente ativas e as plataformas de FaaS permitem que seja definido um evento para que as funções sejam iniciadas, esse evento pode ser uma requisição do cliente, serviços externos, algum fluxo de dados ou outros. Portanto a provedora é responsável por precificar, executar as funções de acordo com a demanda em instâncias isoladas e escalar suas execuções horizontalmente de acordo com o número de eventos recebidos.

Na Figura 5 é apresentada a arquitetura da computação sem servidores com suas duas categorias.

Figura 5 - Arquitetura *Serverless*



Fonte: Survey on serverless computing. HASSAN;BARAKAT;SARHAN, 2021

Apesar do nome, a computação sem servidores pode ser considerada um oxímoro - porque servidores ainda são usados para fazer a computação. O nome sugere que o usuário do serviço simplesmente escreve o código e deixa o servidor responsável por alocar e administrar as tarefas em nuvem (JONAS, et al, 2019).

3.2 Diferença entre computação sem servidores e o modelo tradicional

A computação sem servidores é geralmente defendida como uma ferramenta para economia de custos e existem múltiplos trabalhos que reportam melhores oportunidades financeiras implantando uma solução usando arquitetura de micro serviços em plataformas *serverless* do que usando modelos tradicionais de aplicação com servidor (*serverful*) (MCGRATH; BRENNER, 2017). No modelo tradicional o servidor atua como um sistema monolítico que contém toda a lógica de negócio, enquanto com a arquitetura *serverless* o sistema é modelado em ações menores que são inicializadas por algum evento, essa distribuição em funções menores aumenta a eficiência no desenvolvimento e diminui a chance de um ponto com falha (HASSAN;BARAKAT;SARHAN, 2021).

Existem três principais diferenças entre a computação *serverless* e *serverful* (JONAS, et al, 2019):

- **Computação e armazenamento desacoplado:** o armazenamento e a computação escalam, são providenciadas e precificadas isoladamente. Normalmente o armazenamento é oferecido por meio de um serviço na nuvem separado.
- **Executar código sem gerenciamento de recursos:** o usuário fornece uma parte de um código e o serviço automaticamente providencia os recursos necessários para a execução solicitada. Dessa forma, os programadores são liberados do gerenciamento dos recursos do servidor.
- **Pagamento sob demanda:** a cobrança é proporcional aos recursos usados ao invés de recursos alocados e está associada com a execução e seu tempo.

Outro ponto é que as ferramentas de monitoração e depuração presentes nas aplicações monolíticas não estão disponíveis no modelo *serverless*, obrigando os desenvolvedores a usar ferramentas inerentes para essa finalidade. O poder computacional não é uma preocupação para os programadores que utilizam as plataformas *serverless*. Entretanto, no modelo tradicional, normalmente é necessário possuir duas instâncias do servidor, uma para uso primário e outra para *backup* em caso de falha, causando um aumento de custo. A arquitetura *serverless* pode ser

bem mais econômica quando se tem um grande número de requisições de forma inconstante (HASSAN;BARAKAT;SARHAN, 2021).

Na Tabela 2 são retratadas as diferenças entre o modelo de computação tradicional e o modelo de computação sem servidores. As especificações e preços correspondem ao AWS Lambda e instâncias AWS EC2 sob demanda (JONAS, et al, 2019).

Tabela 2: Diferença das características de um modelo de servidor tradicional na nuvem (*serverful*) e sem servidores (*serverless*)

	Características	**AWS *Serverless Cloud***	**AWS *Serverful Cloud***
Programador	Quando o programa é executado	Em evento selecionado pelo usuário	Continuamente até parado explicitamente
	Linguagem de programação	JavaScript, Python, Java, Go, C#	Qualquer
	Estado do programa	Mantido em armazenamento (*stateless*)	Qualquer (*stateful* ou *stateless*)
	Tamanho máximo da memória	0,125 - 3 GB (Usuário seleciona)	0,5 - 1952 GB (Usuário seleciona)
	Armazenamento máximo local	0,5 GB	0 - 3600 GB (Usuário seleciona)
	Tempo máximo de execução	900 segundos	Nenhum
	Unidade mínima para cobrança	0,1 segundos	60 segundos
	Preço por unidade mínima de cobrança	$0,0000002 (assumindo que seja 0,125 GB)	$0,0000867 - $0,4080000
	Sistemas operacionais e bibliotecas	Provedora da nuvem seleciona	Usuário seleciona
Administrador do sistema	Instanciar servidores	Provedora da nuvem seleciona	Usuário seleciona
	Escalonamento	Responsabilidade da provedora da nuvem	Responsabilidade do usuário
	Implantação	Responsabilidade da provedora da nuvem	Responsabilidade do usuário
	Tolerância a falhas	Responsabilidade da provedora da nuvem	Responsabilidade do usuário
	Monitoramento	Responsabilidade da provedora da nuvem	Responsabilidade do usuário

	Logs	Responsabilidade da provedora da nuvem	Responsabilidade do usuário

Fonte: Cloud Programming Simplified: A Berkeley View on Serverless Computing. JONAS, et al, 2019. Traduzido pelo autor

3.3 Function-as-Service

Nesta seção serão apresentados as características e limitações das *Function-as-Service*. Ele é uma nova abordagem para a computação em nuvem que permite que os desenvolvedores executem funções orientadas a eventos na nuvem sem a necessidade de gerenciar e alocar os recursos ou configurar o ambiente de execução. Desse modo os programadores podem focar inteiramente no código da aplicação, e a provedora é responsável por gerenciar as dependências, compilar o código, configurar os sistemas operacionais e gerenciar a alocação dos recursos, além de carregar novas instâncias se a demanda aumentar muito e finalizar as instâncias terminadas (JANGDA; et al, 2019).

3.3.1 Características técnicas

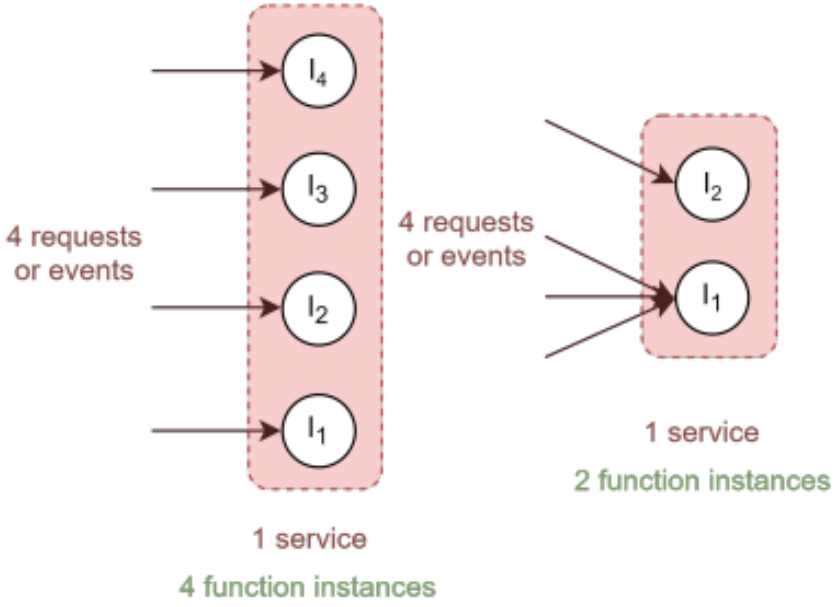
- **Estados das instâncias das funções:** são 3 estados definidos para as instâncias (MAHMOUDI;KHAZAEI, 2021): inicializando, executando, em espera (*initializing, running,* e *idle*). O estado inicializando acontece quando a infraestrutura está começando novas instâncias, o que pode incluir a configuração de novas máquinas virtuais ou *containers* para lidar com a carga de trabalho excessiva e também a inicialização da aplicação que são as tarefas iniciais realizadas pela instância como importar bibliotecas ou criar uma conexão com o banco de dados. A instância vai permanecer em estado de inicialização até ser capaz de responder as chamadas que chegam, e enquanto esse estado não é finalizado a instância não é capaz de aceitar nenhuma requisição. É importante ressaltar que a parte de inicialização da aplicação é tarifada pela maioria das provedoras deste tipo de serviço, enquanto o resto da parte de inicialização não é. Quando uma requisição é recebida pela instância ela passa para o estado executando, é nesse estado

que ela é analisada e processada. O tempo nesse estado também é tarifado pelas provedoras. Depois do processamento terminar, a plataforma mantém as instâncias ativas (*warms*) e ociosas por um determinado período de tempo para que ela consiga lidar com a carga de trabalho que possam chegar. Esse é o chamado estado de espera, e ele não é tarifado.

- **Início quente/frio:** em (MAHMOUDI; KHAZAEI, 2021), quando uma requisição tem um início frio (*cold start*) significa que ela passou por todo o processo de lançamento e inicialização de uma nova instância. No caso de a plataforma possuir alguma instância em espera quando a requisição chegar, ela vai ser direcionada para que a instância seja reutilizada. Esse direcionamento para que uma nova instância não seja iniciada se chama início quente (*warm start*). Os inícios frios podem ser muito mais longos que os inícios quentes para algumas aplicações, portanto um número alto de inícios frios podem impactar na capacidade de resposta da aplicação e a experiência do usuário (Wang et al., 2018). Muitas das pesquisas feitas na área de computação *serverless* focam em aperfeiçoar o início frio.
- **Autoescalonamento:** foram identificados três padrões de autoescalonamento utilizados nas plataformas mais populares de computação *serverless*: 1) escalonamento por requisição; 2) escalonamento por nível de concorrência; 3) escalonamento baseado em métricas. No escalonamento por requisição as plataformas de FaaS quando recebem as requisições tentam direcionar para uma instância quente em espera (*warm start*) ou vai ser inicializado uma nova instância (*cold start*) que funciona como um pequeno servidor para as próximas requisições. Quando a carga de trabalho diminuir, a plataforma também escala o número de funções para baixo. Portanto, nesse modelo de escalonamento não existe nenhuma fila envolvida e desde que a requisição chegue até o limite de expiração da instância (em que ela está quente) ela vai ser atendida, esse tempo limite chegar a instância é finalizada e seus recursos dispensados. Por simplificar o processo de cobrança esse é a técnica de escalonamento dominante no mercado, e a maioria das plataformas a utilizam, como AWS Lambda, Google Cloud Functions, IBM Cloud Functions, Apache OpenWhisk, and Azure Functions (WANG et al., 2018; VAN EYK et al., 2018).

No modelo de escalonamento por nível de concorrência (GOOGLE, 2021) as instâncias podem receber múltiplas requisições ao mesmo tempo. O número de requisições que podem ser recebidas simultaneamente é o nível de concorrência. Na Figura 6 é apresentado o efeito do valor do nível de concorrência e as instâncias das funções.

Figura 6: O efeito do nível de concorrência das instâncias das funções. O serviço da esquerda tem nível 1 e o da direita tem nível 3.



Fonte: SimFaaS: A Performance Simulator for Serverless Computing Platforms. MAHMOUDI;KHAZAEI, 2021

O modelo de escalonamento baseado em métricas tenta deixar as métricas como utilização de CPU e memória em um nível pré definido. Algumas plataformas que o utilizam são: WS Fargate, Azure Container Instances, OpenFaaS, Kubeless, and Fission.

- **Nível máximo de concorrência:** toda plataforma *serverless* tem suas limitações em relação ao número de instâncias que podem ser inicializadas e executadas. Para garantir que o sistema esteja disponível sempre o número de instâncias que um usuário pode executar simultaneamente é limitado, e esse número é chamado de nível máximo de concorrência. Por exemplo, no

AWS Lambda o nível máximo de concorrência em 2020 é de 1000 instâncias. Quando o sistema chega nesse número de funções, qualquer requisição que chegue vai receber como resposta do servidor que ele não está disponível para completar a requisição naquele momento.

- **Roteamento de requisições:** com o intuito de minimizar o número de *containers* quentes e liberar mais recursos, a plataforma roteia as requisições para as funções mais recentes, a não ser que todas estejam ocupadas (McGRATH; BRENNER, 2017). Ou seja, a prioridade são sempre as instâncias com data de criação mais recente e dessa forma o sistema tenta maximizar as chances das funções mais antigas serem expiradas e finalizadas.

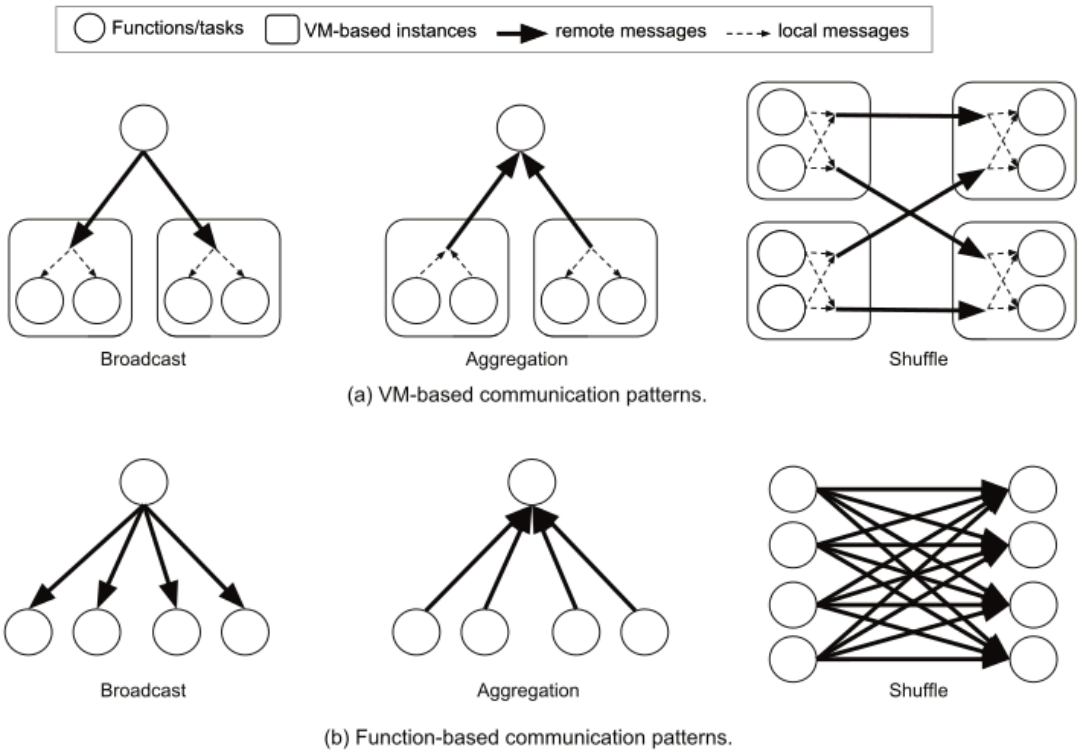
3.3.2 Limitações e desafios

No trabalho de (JONAS, et al, 2019) os autores tentaram implementar cinco aplicações diferentes usando *serverless* para analisar quais obstáculos que apareceram e suas soluções. Os aplicativos usados são: um para *upload* de vídeos em tempo real (estilo Youtube), um MapReduce para processamento de dados, um para cálculos mais complexos de álgebra linear, um para treinamento de Aprendizado de Máquina e outro para utilização de um banco de dados SQLite. Como limitações os autores encontraram:

- **Armazenamento inadequado para operações refinadas:** por serem *stateless*, as plataformas *serverless* têm dificuldade com aplicações que seja necessário manter um estado refinado e complexo sempre atualizado e compartilhado com as outras instâncias. Isso se dá por conta das limitações presentes nos serviços de armazenamento oferecidos pelas provedoras.
- **Falta de coordenação:** para conseguir oferecer aplicações *stateful*, as plataformas *serverless* devem garantir um método para que as tarefas trabalhem coordenadamente. Muitos protocolos que têm a intenção de garantir a consistência dos dados também precisam de uma coordenação parecida. Por exemplo, se a tarefa A precisa do resultado da tarefa B, é necessário que a tarefa A tenha conhecimento quando o valor estiver disponível, mesmo que eles estejam em diferentes instâncias.

- **Baixa performance para os modelos de comunicação padrão:** os três padrões de comunicação nos sistemas distribuídos são o *broadcast*, *aggregation* e *shuffle*. Na Figura 7 são mostrados os padrões de comunicação para soluções baseadas em funções e baseadas em máquinas virtuais.

Figura 7: Três padrões de comunicação para sistemas distribuídos: *broadcast*, *aggregation* e *shuffle*. (a) Mostra esses padrões para instâncias de máquina virtual. (b) Mostra esses padrões para instâncias funções na nuvem. É importante notar o pequeno número de mensagens remotas nas VMs por conta de seu compartilhamento de dados entre as tarefas antes de enviá-las ou recebê-las.



Fonte: Cloud Programming Simplified: A Berkeley View on Serverless Computing. JONAS, et al 2019

Nas soluções baseadas em máquinas virtuais, todas as tarefas que estão executando na mesma instância podem compartilhar uma cópia do seu dado em *broadcast* ou fazer uma agregação antes de enviar resultados parciais a outras instâncias. Deste modo, a complexidade das operações para comunicação *broadcast* e *aggregation* é O(N), onde N é o número de instâncias de VMs no sistema. No entanto, para soluções que utilizam as

funções *serverless*, essa complexidade vai para O(N × K), onde K é o número de funções por VM. A diferença é ainda maior para a operação *shuffle*, com as soluções baseadas em VM, todas as tarefas locais combinam e compartilham seus dados de forma que tenha somente uma mensagem entre duas instâncias de VM. Assumindo o mesmo número de remetentes e recebedores, temos N² mensagens. Agora para as funções, temos que enviar (N × K)² mensagens. Tipicamente o valor de K varia entre 10 e 100, portanto uma aplicação usando *serverless* pode ter que enviar a mais de dados cerca de um número com dois a quatro graus de magnitude a mais que o equivalente usando VMs.

- **Performance previsível:** apesar das funções terem um tempo menor de inicialização que o modelo tradicional baseado em instâncias de máquinas virtuais, os *delays* que ocorrem ao inicializar podem ser grandes para algumas aplicações. Existem três fatores que influenciam a latência de um *cold start*: 1) o tempo para iniciar essa função na nuvem; 2) o tempo para inicializar o ambiente de *software* da função (ex: carregar bibliotecas da linguagem); e 3) inicializações específicas no código do usuário (ex: conectar um banco de dados). Os últimos dois são os que causam maior impacto, enquanto pode levar menos que um segundo para iniciar uma função na nuvem, pode levar mais de dez segundos para carregar todas as bibliotecas da aplicação.

Além disso, em (HASSAN;BARAKAT;SARHAN, 2021) foi feito um levantamento na literatura sobre computação sem servidores e foram identificados outros desafios e pontos problemáticos, sendo alguns deles:

- ***Cold start*:** a computação sem servidores pode escalar até zero quando não há requisições para funções ou serviços. Quando isso acontece nós temos o problema dos inícios frios. Métodos e técnicas são estudados e utilizados para reduzir os inícios frios.
- **Segurança:** é o maior desafio das plataformas *serverless*. Um dos problemas é a isolação, que precisa ser boa o suficiente por conta que muitas funções estão sendo executadas por vários usuários em uma plataforma compartilhada.

- **Duração das funções:** as funções a serem executadas têm um tempo de execução limite que é curto. Não é possível executar funções com duração grande porque elas são *stateless*, ou seja, não é possível pausar e retomar depois.
- **Desenvolvimento e depuração:** há uma falta de ferramentas para depuração, então as ferramentas de monitoração são necessárias para observar como as funções estão funcionando. Portanto, exibir os logs dos eventos das funções é um bom método para descobrir e tratar erros. Nas plataformas *serverless* não existe algo equivalente ao *stack trace* atualmente.

3.4 Considerações finais

Neste capítulo foram apresentados conceitos sobre computação sem servidores e suas categorias, diferenças para a computação tradicional e os conceitos de funções como serviço, suas características e limitações. No próximo capítulo serão apresentados alguns simuladores, tanto de nuvem quanto de *serverless*.

4 SIMULADORES

Os simuladores podem ser usados para avaliar o desempenho de sistemas sem a necessidade de realizar experimentações com o sistema real.

O uso de simuladores se faz necessário pois o *benchmarking* em sistemas de computação em nuvem é inviável por conta do seu alto custo para ser realizado. Além disso, seu uso fornece insumos para tomadas de decisão nas empresas e ainda resulta em menores custos de implementação.

4.1 Simuladores de nuvem

O iSPD (*Iconic Simulator of Parallel and Distributed Systems*) foi criado e é mantido pelo Grupo de Sistemas Paralelos e Distribuídos (GSPD) da Universidade Estadual Paulista “Júlio de Mesquita Filho”. É um simulador desenvolvido em Java, orientado a eventos discretos para modelagem e simulação de grades computacionais (MANACERO et al. 2012). Ele possui uma interface gráfica que tem como um dos pontos fortes a facilidade de uso por ser uma interface icônica.

O CloudSim (BUYYA; RANJAN; CALHEIROS, 2009) foi desenvolvido em Java e conta com um conjunto de bibliotecas e classes que realizam a simulação de computação em nuvem. Não possui uma interface gráfica, portanto para utilizá-lo é preciso ter um conhecimento prévio de programação e de Java.

O iCanCloud (CASTANE; NUNEZ; CARRETERO, 2012) foi desenvolvido com o os frameworks de simulação de eventos OMNet++ e INET e têm como propósito de realizar as simulações para serviços da classe IaaS. Ele também possui uma interface gráfica.

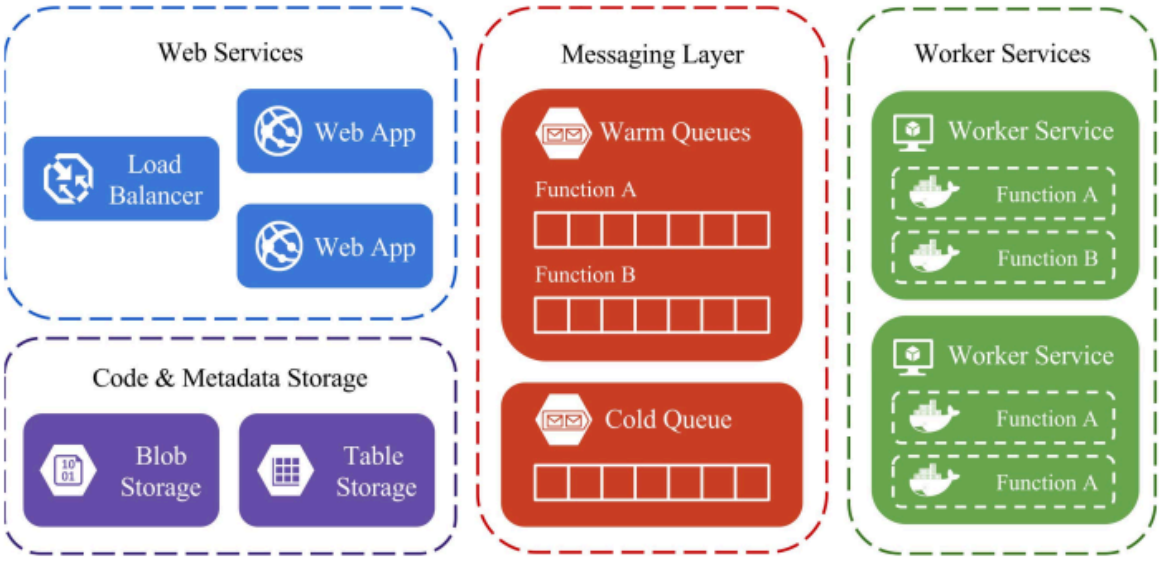
4.2 Simuladores de FaaS

Em (MCGRATH; BRENNER, 2017) foi feita uma plataforma para computação sem servidores com foco em performance. Foi implementado usando .NET e implantado com Microsoft Azure usando o Azure Storage como ferramenta para persistir os dados. O protótipo desenvolvido consiste de dois componentes: um serviço web que disponibiliza uma REST API e um serviço de trabalho que gerencia e executa os contêineres das funções. A camada web descobre os trabalhadores

através de uma camada de mensagem que consiste de várias filas de Azure Storage.

Para executar as funções é preciso chamar “/invoke” na REST API, o corpo da chamada são os *inputs* para a função e a resposta contém os *outputs*. A execução começa quando o serviço web recebe a chamada, então um objeto com os dados e inputs é criado e ocorre uma tentativa de localizar algum container disponível com o *worker service* para processar a requisição. A interação entre os serviços web e *worker* é feita por uma camada compartilhada de troca de mensagens, mais especificamente, uma fila quente (*warm queue*) e fila fria (*cold queue*). Essas filas guardam as mensagens dos *containers*, que consiste de uma URL com o endereço da instância trabalhadora e o nome do container disponível. Nas filas frias as mensagens indicam que um trabalhador liberou memória e pode iniciar um novo *container*, já nas filas quentes elas indicam que existem *containers* que não estão realizando nenhuma execução no momento. A arquitetura e os componentes do protótipo desenvolvido são apresentados na Figura 8.

Figura 8: Arquitetura e os componentes do protótipo desenvolvido



Fonte: Serverless Computing: Design, Implementation, and Performance. MCGRATH; BRENNER, 2017

O serviço *web* tenta retirar uma mensagem da fila quente, se nenhuma mensagem for encontrada uma mensagem da fila fria é retirada, que então vai

associar um novo container para aquela função. Se todos os trabalhadores estiverem alocados e executando a fila fria estará vazia, portanto se não for encontrado um container disponível, o serviço irá retornar HTTP 503 *Service Unavailable* porque não tem recurso para completar a requisição. Assim que a mensagem é encontrada e retirada da fila, o serviço *web* envia uma requisição HTTP para um serviço trabalhador usando a URI que está no corpo da mensagem, e então esse trabalhador executa a função e retorna os outputs.

Existem dois jeitos dos *containers* serem removidos: quando uma função é deletada, então ele é removido junto com os *containers* e a memória reservada e o serviço web deleta todas sua fila quente; ou quando o container fica em espera por 15 minutos, após isso ele é removido e sua memória recuperada. Sempre que a memória é recuperada o serviço trabalhador envia novos containers para a fila fria se a memória não utilizada for maior que o tamanho máximo de memória da função.

Para testar a performance, os autores realizaram dois testes em diversas plataformas como AWS Lambda, Azure Functions, Google Cloud Functions, and Apache OpenWhisk, implementado usando Node.js que é um linguagem aceita por todas. O primeiro teste foi o de concorrência usado para medir a capacidade da plataforma de responder de forma performática às requisições das funções. A ferramenta reenvia a requisição sempre que chega uma nova resposta da requisição anterior. O teste começa com uma chamada, e a cada 10 segundos uma nova chamada é adicionada, até o máximo de 15 chamadas paralelas. Esse teste foi feito 10 vezes em cada uma das plataformas. O segundo teste foi o de *Backoff*, que foi feito para analisar os inícios frios e o comportamento dos limites de tempo das funções. Esse teste envia uma única requisição a intervalos crescentes, variando de 1 a 30 minutos.

Para concluir, foram identificadas algumas limitações do trabalho pelos autores, como:

- a implementação das filas quentes, que nesse caso foi usado uma fila FIFO e pode ser problemático quando se tem uma carga muito pesada porque um container com execução demorada pode “segurar” outros atrás que já finalizaram;
- execuções assíncronas, que o modelo implementado atualmente só aceita invocações síncronas, e a dificuldade em execuções assíncronas é garantir

pelo menos uma resposta portanto é necessário adicionar uma lógica para ter certeza que a execução completou com sucesso;

- utilização dos trabalhadores, pois de forma realista seria necessário fazer uma super-alocação dos recursos trabalhadores tendo em mente que nem todas as funções em um trabalhador estão em execução ou utilizando toda a memória reservada para ela. A falta de *datasets* com dados referentes a essa utilização por parte das plataformas dificulta um melhor entendimento;
- segurança também é uma questão que está em aberto, executar código de usuários nos *containers* de sistemas multilocatários pode ser um perigoso e é preciso que se tenha cuidado ao construir e nas execuções das funções para prevenir vulnerabilidades;
- medição de performance, ainda não se tem muitas definições a respeito, porém a qualidade dessas medidas pode ser aumentada pelo melhor tratamento das latências de rede e estabelecimento de testes. Além disso, as variações da latência entre as linguagens, tamanho do código da função, performance entre os diferentes tipos de eventos, e escalonamento da utilização de CPU também justificam estudos.

Já no trabalho de (KRITIKOS;SKRZYPEK, 2019) foi feito um simulador com os conceitos de *Simulation-as-Service* (SimaaS) que tem como premissa a utilização de filas de mensagens para uma organização assíncrona das execuções das funções e que consiga lidar com altas cargas de mensagens.

Ele é totalmente compatível com múltiplas plataformas de computação em nuvem e pode ser executado em uma combinação de provedoras e com suporte para microserviços e componentes *serverless*. Para esse suporte de múltiplas plataformas é utilizado a plataforma MELODIC com a extensão Functionizer (MELODIC, 2021), que é capaz de provisionar dinamicamente e continuamente uma aplicação entre vários ambientes *cloud* diferentes de forma otimizada através do uso de funções utilitárias, monitoramento multi-level e métodos de configuração global. Além disso, também possui grande diversidade em componentes de implantação e provisionamento de recursos, incluindo suporte para provisionamento de máquinas virtuais, *containers* Docker e ainda aplicações com grande volume de dados baseados em Spark. As provedoras que atualmente têm suporte são: Amazon AWS,

Microsoft Azure, Google Cloud Platform, Oktawave, ProfitBricks e qualquer provedora baseada em OpenStack.

O sistema é dividido entre os seguintes elementos:

- Interface usuário: módulo para operar a aplicação, contém a tela que coleta informações do usuário e faz a requisição para simulação. Foi desenvolvido em Java e Angular e implantado como *container* do Docker;
- *Simulation Orchestrator*: é responsável por preparar e iniciar a instância da simulação, além de receber as respostas e devolver para o usuário. Cada instância de simulação tem seus itens de trabalho de acordo com sua simulação e os envia para o *Multi-Cloud Message Queue Broker*. Foi desenvolvido usando Java Spring e implantado como *container* do Docker;
- *Simulation Launcher*: componente que inicializa o *Simulation Executor* de acordo com sua carga de trabalho. Foi implementado em Java Spring e implantado como *container* do Docker;
- *Simulation Executor*: tem como função executar a simulação e retornar o seu resultado. Esse componente pode ser implantado em qualquer plataforma *serverless* determinada pelo usuário. Foi implementado em Java e implantado como um componente *serverless*.
- *Multi-Cloud Message Queue Broker*: é a fila de mensagens multi-cloud, possui duas filas, uma para enviar requisições de execução ou itens de trabalho, e outra para reunir os resultados das simulações. Foi implementado usando framework RabbitMQ com extensão JMS.

Para avaliação, os autores utilizaram dados reais de uma empresa de investimentos e fizeram uma comparação de custos com 3 diferentes máquinas e configurações: uma utilizando a plataforma *serverless* desenvolvida, e as outras duas usando o modelo de computação em nuvem tradicional (VMs), uma utilizando um servidor *single core* e outro utilizando um servidor mais potente com 128 *cores*. A plataforma *serverless* utilizada foi o AWS. Na Figura 9 são apresentados os resultados alcançados pelos autores

Figura 9: Resultados alcançados no trabalho de (KRITIKOS;SKRZYPEK, 2019)

Scenario	Execution Cost (USD)	Execution Time (Hours)	Num of Parallel Executions
1-core server	1866	19440	1
128-core server	2500	151	128
Serverless solution with 1000 instances	450	19.4	1000

Fonte: Simulation-as-a-Service with Serverless Computing. KRITIKOS;SKRZYPEK, 2019

No trabalho de (MAHMOUDI; KHAZAEI, 2021) foi desenvolvido um simulador de performance público para computação *serverless* com alto grau de flexibilidade, fidelidade e acurácia chamado de SimFaaS. Feito em Python, está totalmente disponível em um projeto do Github (https://github.com/pacslab/simfaas), juntamente com sua documentação.

O SimFaaS foi criado com o objetivo de validar a performance do modelo desenvolvido e permitir uma previsão precisa da performance e conta com ferramentas integradas para visualização, análise e verificação de um modelo feito. O simulador proposto pelos autores (MAHMOUDI; KHAZAEI) pode prever algumas questões relacionadas a QoS como a probabilidade de um início frio, tempo médio de resposta e probabilidade de requisições rejeitadas sob diferentes cargas de trabalho. Além disso, também é possível prever a quantidade média de servidores em execução e a quantidade total de servidores que ajudam a calcular a previsão do custo do serviço do desenvolvedor e do custo da infraestrutura necessário pela provedora.

Para utilizar a plataforma, o usuário deve escrever uma aplicação Python ou um Jupyter *notebook*, inicializando as classes e passando os parâmetros necessários. Com ela é possível simular os mais novos provedores de computação *serverless* como AWS Lambda, Google Cloud Functions, IBM Cloud Functions, Apache OpenWhisk e Azure Functions.

No SimFaaS a carga de trabalho das instâncias é caracterizada pela taxa de chegada, tempo de resposta do serviço (*warm start*) e tempo de provisionamento (*cold start* até ficar pronto). A única informação para caracterizar a plataforma FaaS é o limite de expiração, que é o tempo que uma função é finalizada e seus recursos liberados depois de responder sua última requisição. Na Tabela 3 são retratados os parâmetros de *input* e *output* de uma simulação.

Tabela 3: Parâmetros de *input* e *output* de uma simulação usando o SimFaas. Os parametros marcados com * são os outputs

Parâmetro	**Valor**
Taxa de chegada	0,9 req/s
Tempo do serviço quente	1,991 seg
Tempo do serviço frio	2,244 seg
Limite de expiração	10 min
Tempo de simulação	10^6 seg
Pular tempo inicial	100 seg
*Probabilidade de início frio	0,14%
*Probabilidade de rejeição	0%
*Tempo médio de vida útil da instância	6207,7389 seg
*Contagem média de servidores	7,6795
*Média de servidores em execução	1,7902
*Média de servidores em espera	5,8893

Fonte: SimFaaS: A Performance Simulator for Serverless Computing Platforms. MAHMOUDI; KHAZAEI, 2021. Traduzido pelo autor

Para realizar os experimentos avaliativos do SimFaaS, os autores usaram o AWS Lambda e todos os cenários e dados/carga de trabalho estão disponíveis no repositório do Github.

4.3 Considerações finais

Neste capítulo foram apresentados simuladores de computação em nuvem e de computação sem servidores, além de suas especificações e características. No próximo capítulo será apresentado o simulador *serverless-simulator* desenvolvido.

5 SERVERLESS-SIMULATOR

Neste capítulo é apresentado o trabalho desenvolvido para criação do *serverless-simulator*, sua arquitetura, funcionamento, testes e resultados.

5.1 Desenvolvimento

Foi feito um estudo das mais diversas aplicações que têm como objetivo simular uma execução de computação sem servidores e analisar o seu comportamento e desempenho. Na Tabela 4 podemos ver um comparativo entre os simuladores citados no Capítulo 4 e o desenvolvido (*serverless-simulator*).

Tabela 4: Comparativo entre os simuladores FaaS estudados e o desenvolvido

	SimFaas	***Simulation-as-Service***	**Protótipo *Serverless***	**Serverless-simulator**
Interface gráfica	Não	Sim	Não	Sim
Ling. Prog.	Python	Microsserviços (Angular, Java, Spring)	.NET	Simulador: Java - Spring framework Interface: Angular
Código aberto	Sim	Não	Não	Sim
Implementação	Sem filas, escalonamento por requisição	Sem filas, escalonamento por requisição. Utilização de componentes *serverless*	Filas pelo azure storage	Sem filas, escalonamento por requisição
Previsão de custo	Não	Sim	Não	Não

Fonte: Elaborado pelo autor

Cada um dos trabalhos estudados tem a sua particularidade, entre eles o *Simulation-as-Service* é o único que possui uma interface gráfica e uma previsão de custos de acordo com uma provedora real. Já o SimFaas é o único que tem o código aberto e permite que seja possível analisar o código fonte do simulador. O protótipo *Serverless* é o diferente em relação à implementação pois faz uso de filas pelo azure

storage, enquanto os outros dois usam o escalonamento por requisição que não possui filas e é o modelo mais usado pelas provedoras de nuvem. A única coisa que os três estudos diferem totalmente é na linguagem de programação utilizada, o SimFaas utiliza Python, o protótipo *Serverless* usa .NET e o *Simulation-as-Service*, como são vários componentes diferentes, utilizam microsserviços com partes em Angular e Java Spring.

Após isso, o primeiro passo para o desenvolvimento foi determinar os requisitos que nosso sistema deveria ter. Ficou definido que o nome do sistema é *serverless-simulator* e seus requisitos necessários são:

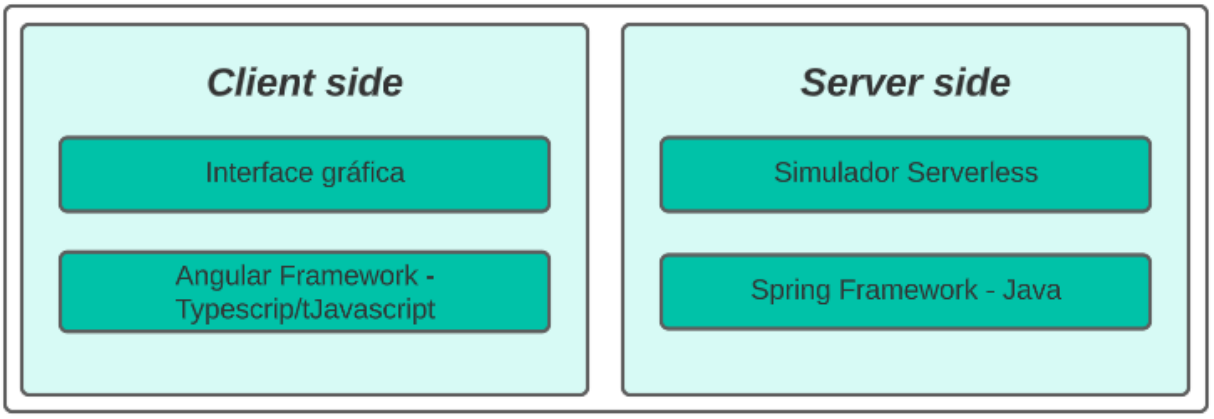
- possuir interface gráfica para facilitar a interação do usuário com o sistema e permitir que os usuários leigos, que não conhecem linguagens de programação ou possuem dificuldades ao trabalhar com essas, por não serem especialistas da área de computação, possam desenvolver simuladores para avaliar o desempenho de seus trabalhos;
- utilizar o escalonamento por requisição como técnica de escalonamento e simular um processo que não tenha nenhum tipo de fila;
- possuir uma análise de resultados com informações sobre a execução do simulador como a quantidade total de máquinas que rodaram, quantas eram quente e quantas eram fria e seus respectivos tempos de execução, quantidade de vezes que as máquinas ficaram paradas e sua duração nesse estado;
- ter o código aberto e disponível para que qualquer pessoa interessada possa examinar e entender melhor a aplicação.

Inicialmente a ideia era acoplar esse simulador ao ISPD, mas optamos por desacoplar pois não seria possível aproveitar muito do que foi desenvolvido lá por conta dos conceitos entre os sistemas serem diferentes e algo completamente novo teria que ser feito de qualquer forma. Foi decidido fazer uma aplicação *web* pela facilidade de desenvolvimento e de acesso, fazendo com que seu uso seja possível em qualquer plataforma que permita usar um navegador com conexão à internet. Essa facilidade de utilização em qualquer dispositivo, aliado ao fato de possuir uma interface gráfica simples e prática são os diferenciais do nosso simulador. Além disso, outro ponto é que o código-fonte da aplicação está disponível no github com o objetivo de fornecer um acesso mais amplo para todos. (Links: Front =

https://github.com/igorbrito97/serverless-simulator-front. Servidor = https://github.com/igorbrito97/serverless-simulator-back)

Na Figura 10 é mostrado o diagrama contendo a separação do projeto e as tecnologias escolhidas. Para a parte de cliente, que é tela que o usuário irá acessar, foi escolhido o *framework* Angular (versão 14) que é baseado em Typescript/Javascript. Já o simulador *Serverless*, que é a nossa aplicação principal da parte do servidor, foi escolhido o *framework* Spring (versão 2.7.1) baseado em Java (versão 8).

Figura 10: Modelo cliente-servidor utilizado no *serverless-simulator* e seus componentes



Fonte: Elaborado pelo autor

5.2 Interface Gráfica

A tela principal do sistema pode ser vista na Figura 11.

No canto superior direito é possível selecionar o idioma, entre português e inglês. O formulário do simulador possui todos os campos e informações necessárias para realizar uma simulação. Os campos usados no *serverless-simulator* são:

- **Taxa de chegada**: é a quantidade de requisições que o sistema vai receber. Medido em requisições por segundo;
- **Tempo da resposta quente**: representa o tempo médio que o sistema leva para responder uma chegada quente (*warm start*). Medido em segundos;

- **Tempo da resposta fria**: representa o tempo médio que o sistema leva para responder uma chegada fria (*cold start*), considerando o tempo de inicialização que uma instância leva para ficar pronta para executar a requisição. Tem que ser maior que o tempo da resposta quente. Medido em segundos;
- **Tempo limite de expiração**: é o tempo que uma instância fica em espera (*idle*) aguardando a chegada de uma nova requisição. Se esse tempo se esgota sem que uma requisição chegue a instância é finalizada. Medido em segundos;
- **Tempo da simulação**: é o tempo total da simulação. Medido em segundos;

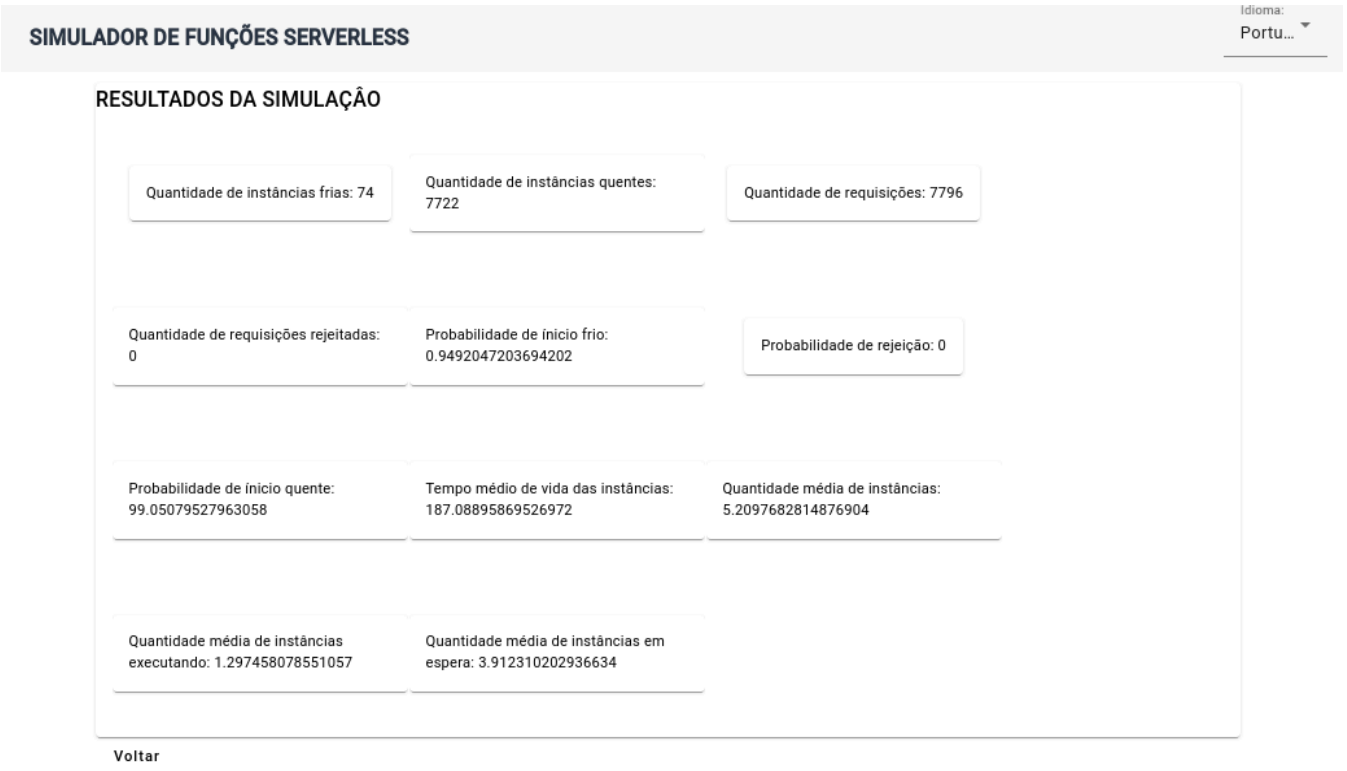
Figura 11: Tela principal.

Fonte: Elaborado pelo autor

Para iniciar a simulação deve-se preencher todos os campos e clicar no botão para avançar. Após o processo de simulação finalizar, o formulário será substituído pelo relatório contendo os resultados da simulação. Esse relatório gerado inclui a quantidade total de requisições feitas, quantidade de requisições frias e sua porcentagem, quantidade de requisições quentes e sua porcentagem, quantidade de requisições rejeitadas e sua porcentagem, o tempo médio de um vida das instâncias, a quantidade média de instâncias totais e a média de instância executando e em espera. Com essas informações, o usuário do sistema consegue prever e analisar

custos baseado em diferentes configurações do sistema. A tela de relatório pode ser vista na Figura 12.

Figura 12: Tela de relatório.

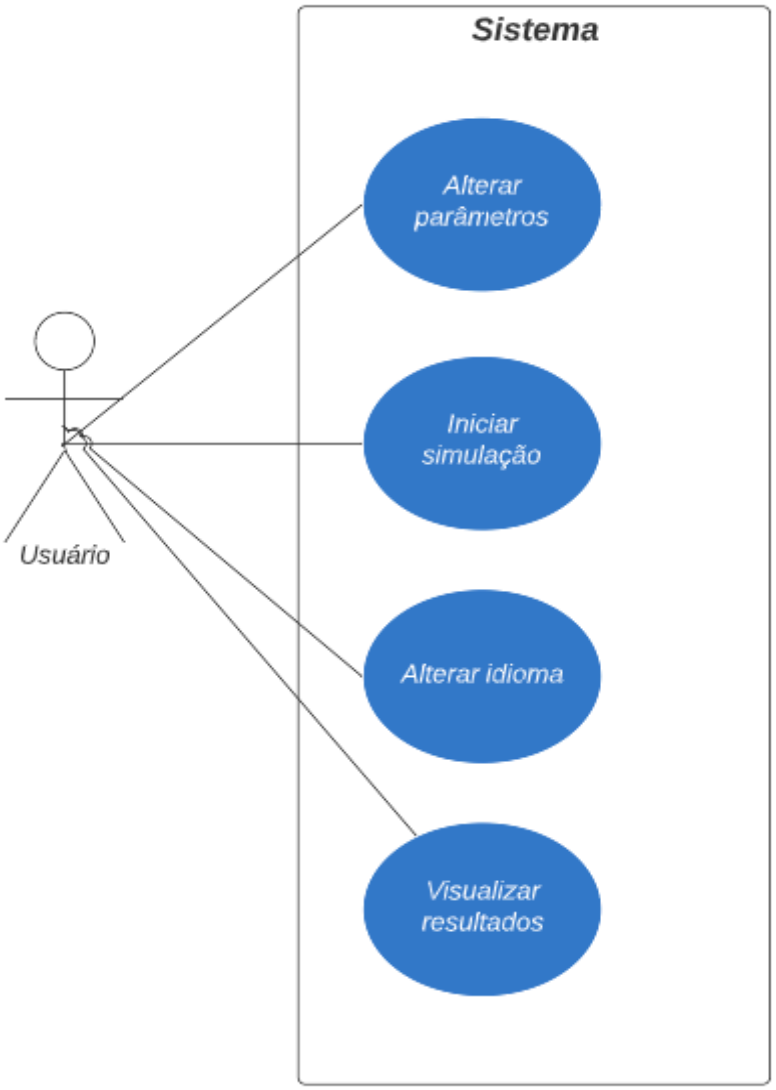


Fonte: Elaborado pelo autor

Na Figura 13 é retratado o diagrama de caso de uso da tela e descreve as principais funcionalidades do sistema.

A principal ação que o usuário do sistema pode fazer é iniciar a simulação, para isso ele também pode alterar o valor de qualquer parâmetro, todos os campos são numéricos. Após realizar a simulação o usuário pode visualizar os resultados da simulação que será exibido através de texto. Além disso, também é possível alterar o idioma do sistema, entre Português e Inglês. O idioma padrão é o Português.

Figura 13: Diagrama de caso de uso.



Fonte: Elaborado pelo autor

5.3 Simulador

Foram definidas as classes a serem utilizadas durante o desenvolvimento e o diagrama de classe pode ser visto na Figura 14.

Figura 14: Diagrama de classe.

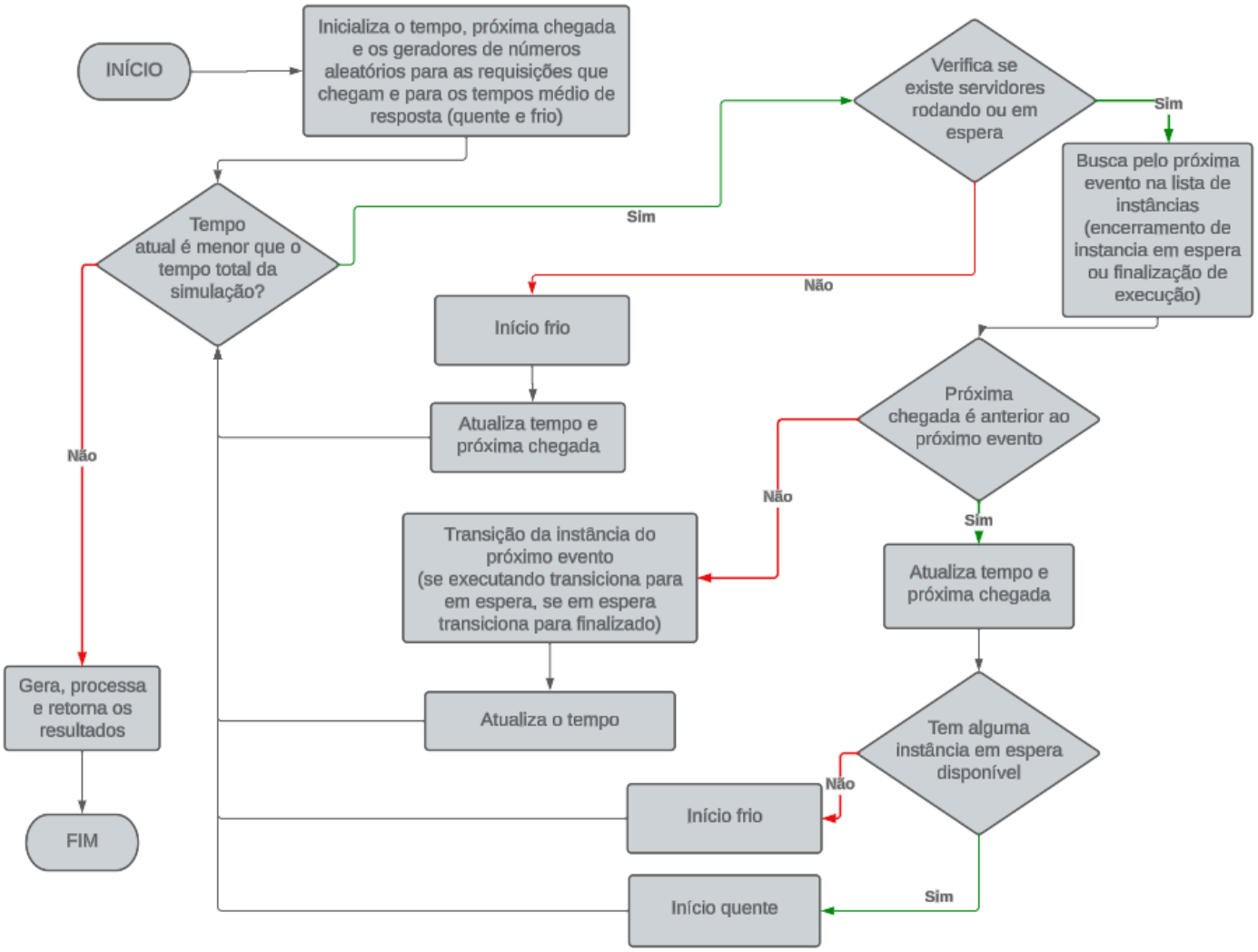
Fonte: Elaborado pelo autor

As classes do sistema são:

- ServicoSimulacao: Responsável por manter e controlar o objeto da simulação. É o ponto de entrada do processo.
- Simulação: Classe que representa a simulação e possui seus atributos.
- FunçãoServerless: Classe que representa a função *Serverless* e a sua execução.
- InputSimulacao: É a classe com os dados de entrada do sistema que o usuário vai preencher para iniciar a simulação.
- ResultadoSimulacao: É o que é gerado pela simulação e retrata como foi a sua execução.
- AnaliseResultados: Responsável por pegar os resultados e gerar uma análise mais detalhada, com informações mais relevantes e dados para visualização gráfica.

Na Figura 15 é apresentado o fluxograma da simulação, e mostra o passo a passo do sistema ao iniciar uma simulação

Figura 15: Fluxograma da execução de uma simulação.



Fonte: Elaborado pelo autor

O fluxo começa com a inicialização dos geradores de números aleatórios que representam o tempo de chegada das requisições e os tempos de execução para os inícios frios e quentes, esses valores são calculados pela taxa de chegada, o tempo médio de resposta fria e o tempo médio de resposta quente, todos informados pelo usuário. Essa geração utiliza a distribuição de probabilidade exponencial para simular a chegada de clientes no sistema. As distribuições de probabilidade contém várias informações sobre um certo evento e podem ser utilizadas para analisar desempenho de simulações. A distribuição exponencial é utilizada para modelar diversos eventos que acontecem na natureza, ela não tem uma memória sobre todos os eventos, portanto a ocorrência de um evento não está relacionado com os

eventos passados. Além da inicialização dos geradores, no início da simulação também são inicializadas as variáveis que controlam o tempo e a próxima chegada.

Após essa etapa, é iniciada a repetição principal da simulação que só vai acabar quando o tempo de simulação chegar no tempo total (informado pelo usuário).

O primeiro passo dentro dessa repetição é verificar se existe algum servidor executando ou em espera. Se não existir nenhum, um novo início frio é executado e o tempo e próxima chegada são atualizados. Se algum servidor estiver executando ou em espera é feito uma busca pelo próximo evento na lista de instâncias, esse próximo evento pode ser um encerramento de um instância em espera ou a finalização da execução. Em seguida é verificado o menor valor entre o próximo evento e a próxima chegada de requisição, se o menor valor for o do próximo evento é realizado a transição da instância para em espera se a instância estiver executando ou para finalizado se ela estiver em espera, em seguida o tempo é atualizado e a execução volta para o início da repetição. Mas se o menor valor for a chegada de requisição a primeira coisa a ser feita é atualizar o tempo e a próxima chegada e depois é verificado se existe alguma instância em espera disponível, se sim um início quente é executado e se não o executado é o início frio. Após isso a execução volta para o início da repetição.

Quando o tempo da execução atingir o tempo total da simulação informado pelo cliente, a repetição e simulação são encerradas e é realizada uma análise e processamento da simulação e por fim é gerado o relatório com os resultados finais.

5.4 Cenários de teste

Uma série de testes foi conduzida para avaliar o desempenho do simulador, abrangendo diversos cenários e cargas de trabalho. Os resultados obtidos foram então comparados com as expectativas estabelecidas.

5.4.1 Cenário 1

O primeiro cenário de teste consiste em realizar uma mesma simulação 10 vezes e analisar as diferenças entre seus resultados. Os valores utilizados são fixos e foram os mesmo utilizados no SimFass e estão descritos na Tabela 5:

Tabela 5: Valores utilizados para testes do cenário 1

Parâmetro	Valor
Taxa de chegada	0,9 req/s
Tempo do serviço quente	1,991 seg
Tempo do serviço frio	2,244 seg
Limite de expiração	10 min
Tempo de simulação	10^6 seg

Fonte: Elaborado pelo autor

Ao rodar a simulação com os mesmos valores várias vezes, a expectativa era de que os valores obtidos não seriam excessivamente discrepantes, embora não necessariamente idênticos. Essa expectativa se fundamenta no fato de que o algoritmo subjacente ao nosso teste emprega uma distribuição exponencial para gerar valores aleatórios. A distribuição exponencial é caracterizada por sua tendência a produzir valores que se concentram em torno de uma média, mas com uma probabilidade razoável de desvio. Portanto, esperávamos observar uma certa variação nos resultados, refletindo a natureza estocástica do processo, enquanto ainda mantínhamos uma coerência geral. Na Tabela 6 temos alguns atributos do resultado, sua média, desvio padrão e intervalo de confiança, sendo α (alpha) = 0,5 e a confiança de 95%. Os valores usados para cálculo foram arredondados na quarta casa decimal.

Tabela 6: Média, desvio padrão e intervalo de confiança de alguns atributos da resposta da simulação

Atributo	Média	Desvio padrão	Intervalo de confiança
Probabilidade início frio (%)	0.07705	0.00163049072	[0.076, 0.0781]
Probabilidade rejeitar (%)	0	0	[0, 0]
Média de vida da instância (seg)	2084.53714	40.3617304	[2060, 2110]

Média de instâncias (quantidade)	4.44768	0.0100116732	[4.44, 4.45]
Média de instâncias executando (quantidade)	0.45136	0.000711617875	[0.451, 0.452]
Média de instâncias em espera (quantidade)	3.99631	0.00954530775	[3.99, 4]

Fonte: Elaborado pelo autor

Além disso, é importante ressaltar que o desvio padrão associado aos valores gerados pelo algoritmo é relativamente pequeno. O desvio padrão é uma medida estatística que indica a dispersão dos valores em torno da média. Neste contexto, um desvio padrão reduzido sugere que a variação entre os resultados esperados é limitada, o que aumenta nossa confiança na consistência do modelo. Assim, embora possamos esperar certa variabilidade nos resultados devido à natureza da distribuição exponencial, a magnitude dessa variação é mitigada pela estreita dispersão dos valores em relação à média, fornecendo uma base sólida para nossa análise e interpretação dos dados resultantes.

5.4.2 Cenário 2

Projetamos para o segundo cenário de teste observar a relação entre a taxa de chegada de requisições e a probabilidade de ocorrência de inícios frios. Nesse teste foram realizadas simulações com o valor da taxa de chegada variando entre 0.001 e 10 requisições por segundo e com o tempo da simulação variando de 1000 até 1000000 segundos. A Tabela 7 abaixo contém os valores dos parâmetros utilizados

Tabela 7: Valores utilizados para testes do cenário 2

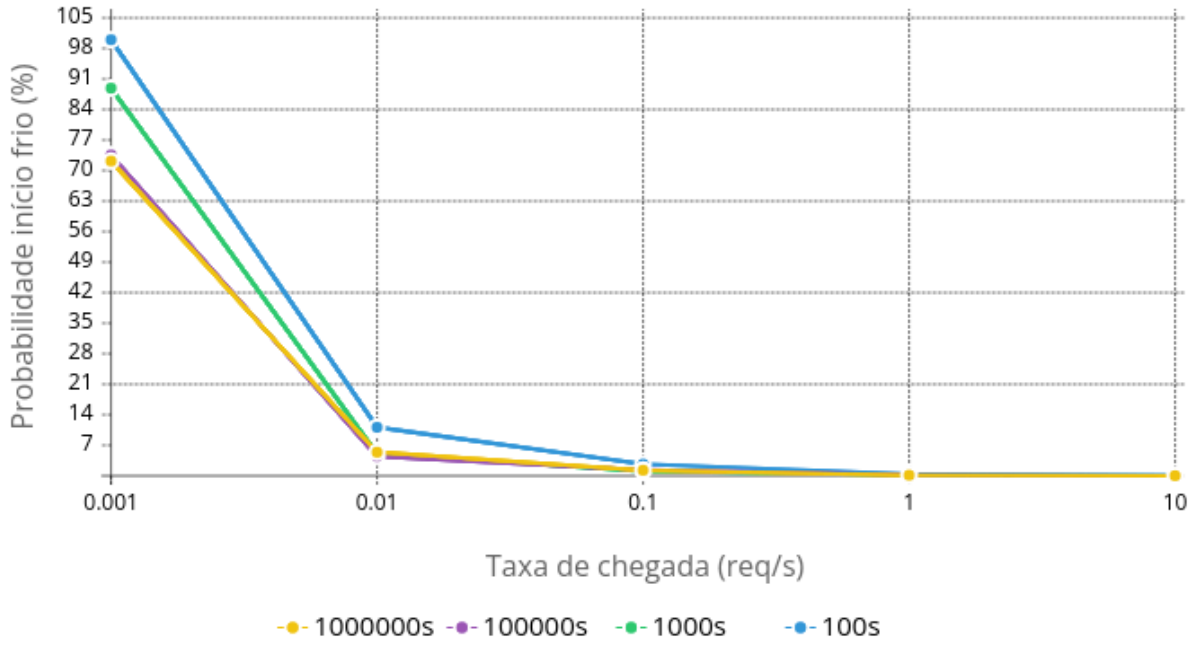
Parâmetro	**Valor**
Taxa de chegada	0.001 req/s - 10 req/s
Tempo do serviço quente	2,016 seg
Tempo do serviço frio	3,114 seg

Limite de expiração	5 min
Tempo de simulação	10^3 seg - 10^6 seg

Fonte: Elaborado pelo autor

Esperávamos que, conforme a taxa de chegada aumentasse, a probabilidade de início frio diminuísse. Isso se deve ao fato de que uma maior taxa de chegada implica em uma maior atividade do sistema, reduzindo assim o tempo de inatividade entre as requisições e diminuindo as chances de um início frio. Além disso, consideramos também o tempo de simulação como um fator influente. Era esperado que, à medida que o tempo de simulação diminuísse, a probabilidade de início frio aumentasse. Isso se deve à natureza do início frio, onde o sistema tem menos tempo para acumular recursos e preparar-se para o processamento de requisições. Na Figura 16 abaixo podemos ver um gráfico onde as linhas representam o tempo de simulação, o eixo x é a taxa de chegada (em req/s) e o eixo y é a probabilidade de início frio (em porcentagem).

Figura 16: Gráfico com a taxa de chegada, sua probabilidade de início frio em diferentes tempos de simulação



Fonte: Elaborado pelo autor

No entanto, é importante destacar que o usuário deve analisar cuidadosamente o impacto da taxa de chegada de requisições. Embora seja

esperado que uma taxa de chegada mais alta diminua a probabilidade de início frio, há um ponto em que esse efeito pode não ser tão significativo, especialmente se a taxa de chegada for suficientemente alta para manter o sistema ativo na maior parte do tempo, mesmo em períodos de simulação mais curtos.

5.4.3 Cenário 3

O próximo cenário de teste proposto envolve alterar o tempo limite de expiração entre 10 segundos e 30 minutos e verificar como o sistema se comporta. A Tabela 8 contém os valores utilizados nos parâmetros para o caso de teste.

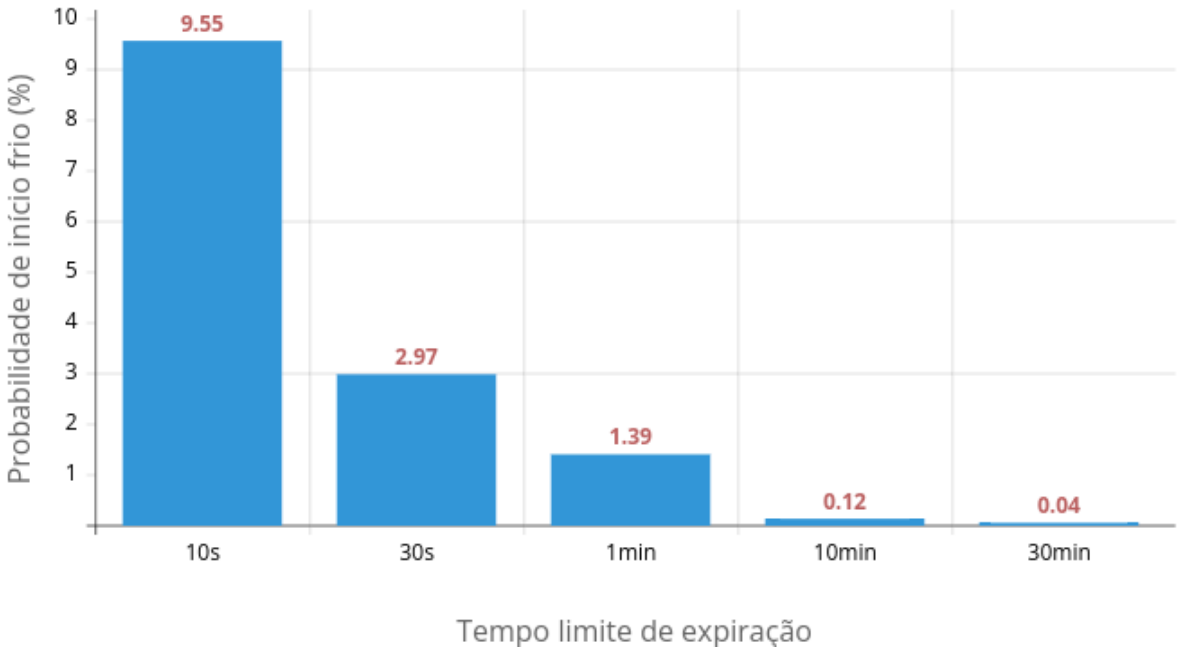
Tabela 8: Valores utilizados para testes do cenário 3

Parâmetro	**Valor**
Taxa de chegada	0.5 req/s
Tempo do serviço quente	1,915 seg
Tempo do serviço frio	2,621 seg
Limite de expiração	10 seg - 30 min
Tempo de simulação	10^6 seg

Fonte: Elaborado pelo autor

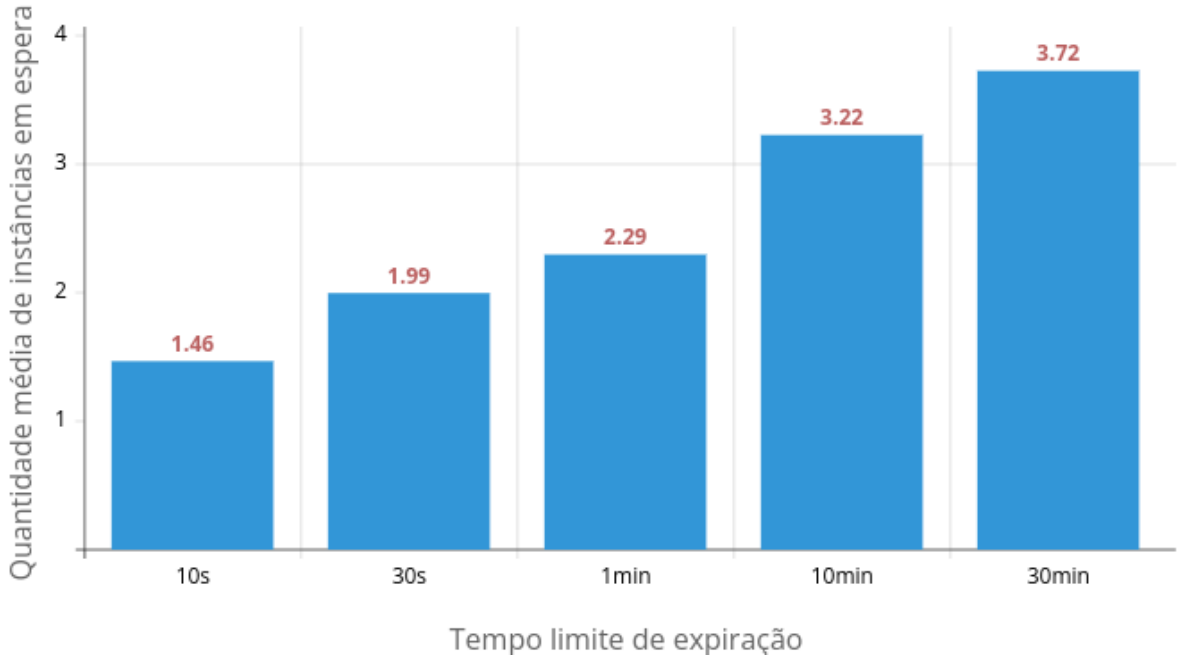
Nossa expectativa é que, à medida que o tempo limite de expiração diminua, aumente a probabilidade de ocorrência de inícios frios. Isso se deve ao fato de que um tempo limite mais curto resulta em uma janela de oportunidade mais estreita para reutilizar instâncias previamente inicializadas, levando potencialmente a um aumento na inicialização de novas instâncias. Além disso, também é esperado que a redução do tempo limite também esteja correlacionada com uma diminuição na quantidade média de instâncias em espera, já que a alta rotatividade de instâncias resultante de tempos de expiração mais curtos pode reduzir o congestionamento do sistema e melhorar a eficiência global de utilização de recursos. Na Figura 17 é exibido a probabilidade de início frio de acordo com o tempo de simulação, e na Figura 18 é apresentado a quantidade média de instâncias em espera de acordo com o tempo da simulação.

Figura 17: Gráfico com a probabilidade de início frio de acordo com o tempo de simulação



Fonte: Elaborado pelo autor

Figura 18: Gráfico com a quantidade média de instâncias em espera de acordo com o tempo de simulação



Fonte: Elaborado pelo autor

5.5 Análise de resultados

Nesta seção apresentamos a análise dos resultados obtidos a partir dos três cenários de teste implementados. Nosso objetivo principal foi avaliar o desempenho do simulador e validar se ele está produzindo resultados que se enquadram com o esperado.

Os resultados dos três cenários de teste foram consistentemente satisfatórios e estiveram dentro das expectativas estabelecidas. As ideias iniciais foram cumpridas e as métricas de desempenho, como tempo de resposta, probabilidades dos inícios frio e quantidade média de instâncias, mostraram-se dentro de valores aceitáveis em todos os cenários testados.

Em resumo, os resultados obtidos nesta fase da pesquisa fornecem uma validação do sistema proposto. A consistência e a eficácia demonstradas nos cenários de teste confirmam a viabilidade e eficiência do sistema em lidar com simulações de plataformas *serverless*, portanto a aplicação deste simulador revelou-se uma ferramenta valiosa para avaliar o desempenho e comportamento de plataformas *serverless* em uma variedade de condições operacionais. No entanto, reconhecemos que ainda há espaço para refinamentos e otimizações adicionais, e pretendemos explorar essas oportunidades em futuras iterações do projeto.

5.6 Considerações finais

Em suma, os resultados dos cenários de teste demonstraram que o simulador de plataforma *serverless* desenvolvido é uma ferramenta eficaz e precisa para análise e simulação de sistemas *serverless*. Os resultados obtidos fornecem uma base sólida para futuras investigações e desenvolvimentos nesta área, e confirmam a utilidade e a confiabilidade do simulador como uma ferramenta de pesquisa e desenvolvimento, podendo contribuir com avanços significativos no campo da computação em nuvem e da computação sem servidores.

6 CONCLUSÕES

Neste trabalho foi apresentado uma revisão sobre computação em nuvem e uma ramificação dela que é a computação sem servidores, além do processo de simulação desse tipo de serviço. Para isso, foi realizada uma revisão da literatura existente, um estudo e análise sobre plataformas que façam essa simulação.

Então foi introduzido o *serverless-simulator,* um simulador para computação *serverless* que tem como objetivo prever o comportamento de um sistema de acordo com os valores informados. Também foi explicado as particularidades do sistema, sua arquitetura e funcionamento. Com o *serverless-simulator* é possível prever a infraestrutura necessária para aplicação, a qualidade do serviço e seu comportamento, permite também analisar o desempenho do seu sistema com diferentes parâmetros e possibilita que o usuário otimize melhor seus recursos. Tudo isso com uma interface gráfica simples e fácil de ser utilizada que agrega valor ao simulador.

Além disso, os cenários de teste bem-sucedidos destacam a eficácia do simulador como uma ferramenta para análise de desempenho, permitindo aos pesquisadores e desenvolvedores explorar diferentes configurações e otimizações em um ambiente controlado. Isso é essencial para impulsionar o avanço contínuo da computação sem servidores e para aprimorar sua aplicação em uma variedade de cenários e setores.

Para trabalhos futuros pensamos na possibilidade de implementar uma previsão de custos de plataformas *serverless,* como por exemplo a AWS Lambda, Google Cloud Functions e Azure Cloud Functions. Também temos como objetivo manter o *serverless-simulator* atualizado e adicionar novos recursos oferecidos pelas plataformas.

REFERÊNCIAS

SUNYAEV, ALI. Cloud Computing. In: Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies. 1. ed.Springer, Cham, 2020.

MELL, P. M.; GRANCE, T. The NIST definition of cloud computing. 2011.

RITTINGHOUSE, JOHN. W.; RANSOME, JAMES F.. Cloud Computing: Implementation, Management, and Security. CRC Press, 2009.

RAJAN, A. P. A review on serverless architectures - function as a service (FaaS) in cloud computing. TELKOMNIKA (Telecommunication Computing Electronics and Control), 2020.

BUYYA, R.; BROBERG, J.; GOSCISNSKI, A. M. Cloud Computing: Principles and Paradigms. John Wiley and Sons: San Francisco, 2011.

VMWARE, VMWare, Inc. Disponível em: https://www.vmware.com. Acesso em 01 dez 2021.

KVM, KVM contributors. Disponível em: https://www.linux-kvm.org. Acesso em 01 dez 2021.

XEN, The Linux Foundation. Disponível em: https://xenproject.org. Acesso em 01 dez 2021.

HYPER-V, Microsoft Disponível em: https://docs.microsoft.com/pt-br/windows-server/virtualization/hyper-v/hyper-v-technology-overview. Acesso em 01 dez 2021.

VCL, Apache Foundation. Disponível em: https://vcl.apache.org. Acesso em: 01 dez 2021.

OPENNEBULA, OpenNebula Systems. Disponível em: https://opennebula.io. Acesso em: 01 dez 2021.

OPENSTACK, OpenInfra Foundation. Disponível em: https://www.openstack.org. Acesso em: 01 dez 2021.

NUPPONEN, JUSSI; TAIBI, DAVIDE. Serverless: What it Is, What to Do and What Not to Do. 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), 2020.

GOOGLE. Google Cloud Platform - Simultaneidade. Disponível em: https://cloud.google.com/run/docs/about-concurrency. Acesso em 18 dez 2021.

JANGDA, ABHINAV, et al. Formal foundations of serverless computing. Proceedings of the ACM on Programming Languages, v. 3, n. OOPSLA, p. 1-26, 2019.

JONAS, Eric et al. Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383, 2019.

MCGRATH, GARRETT; BRENNER, PAUL R. Serverless Computing: Design, Implementation, and Performance. 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), 2017.

VAN EYK, E., et al. A spec rg cloud group's vision on the performance challenges of faas cloud architectures. Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, pages 21–24, ACM, 2018.

WANG, L.,et al. Peeking behind the curtains of serverless platforms. USENIX Annual Technical Conference (USENIX ATC 18), pages 133–146, 2018.

HASSAN, HASSAN B.; BARAKAT, SAMAN A.; SARHAN, QUSAY I. Survey on serverless computing. Journal of Cloud Computing, v. 10, n. 1, 2021.

MANACERO, A. et al. iSPD: an iconic-based modeling simulator for distributed grids. Annals of 45th Annual Simulation Symposium (ANSS12, CDROM), 2012.

BUYYA, R.; RANJAN, R.; CALHEIROS, R. N. Modeling and simultion of scalable cloud environments and the Cloudsim toolkit: Challenges and opportunities, 2009

CASTANE, G. G.; NUNEZ, A.; CARRETERO, J. iCanCloud: A brief architecture overview. ISPA. IEEE, 2012

MAHMOUDI, NIMA; KHAZAEI, HAMZEH. SimFaaS: A Performance Simulator for Serverless Computing Platforms. Proceedings of the 11th International Conference on Cloud Computing and Services Science, 2021.

KRITIKOS, KYRIAKOS; SKRZYPEK, PAWEL. Simulation-as-a-Service with Serverless Computing. 2019 IEEE World Congress on Services (SERVICES), 2019.

MELODIC, Melodic Cloud. Disponível em: https://melodic.cloud. Acesso em: 26 dez 2021.

MELODIC, Functionizer. Disponível em: https://melodic.cloud/functionizer. Acesso em: 26 dez 2021.