Capítulo

5

Computação Serverless: Conceitos, Aplicações e Desafios

André G. Vieira[1], Gustavo Pantuza[1], Jean H. F. Freire[1], Lucas F. S. Duarte[2], Racyus D. G. Pacífico[1], Marcos A. M. Vieira[1], Luiz F. M. Vieira[1], José A. M. Nacif[2]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
CEP: 31270-901 – Belo Horizonte – MG – Brasil
{andregarcia,pantuza,jean,racyus,mmvieira,lfvieira}@dcc.ufmg.br

[2]Instituto de Ciências Exatas e Tecnológicas
Universidade Federal de Viçosa (UFV)
CEP: 35690-000 – Florestal – MG – Brasil
{lucas.f.duarte, jnacif}@ufv.br

Abstract

Serverless computing is a novel application deployment model that provides scaling, fine-grained billing, and platform decoupling for developers. This model has the potential for changing the landscape of technology development for users and cloud providers. In this short course, we present the Serverless Computing state of the art: concepts, limitations, use cases, research opportunities, caveats, and a view on some current platforms.

Resumo

Computação Serverless é um novo modelo de disponibilização de aplicações que provê escalabilidade, precisão de cobrança e desacoplamento de plataforma para desenvolvedores. Este modelo possui potencial para mudar o cenário de desenvolvimento de tecnologias tanto para usuários quanto para provedores de nuvem. Neste minicurso apresentamos o estado da arte em Computação Serverless: seus conceitos, limitações, casos de uso, oportunidades de pesquisa, ressalvas e uma visão sobre algumas plataformas atuais.

5.1. Introdução

Segundo Castro et al. [Castro et al. 2019], é esperado em 2020 que 67% dos gastos com *software* e infraestrutura pelas corporações sejam baseadas em tecnologias em nuvem. Eles também afirmam que essas tecnologias tinham como promessas o modelo de pagar apenas pelo que for utilizado e de abstrair os detalhes de como os servidores eram utilizados e mantidos. A computação *Serverless* surge como um novo modelo de tecnologia em nuvem que permite escalabilidade, precisão de cobrança e desacoplamento de plataforma para desenvolvedores.

5.1.1. Histórico

Historicamente o interesse no uso de plataformas virtualizadas cresceu ao longo da popularização da internet e aplicações web. O uso de máquinas virtuais permitiu que um mesmo *datacenter* pudesse ser compartilhado entre diversas aplicações ou mesmo clientes, devido à possibilidade de reaproveitamento de *hardware* e isolamento entre sistemas operacionais. Serviços de hospedagem de máquinas virtuais como AWS EC2 (2006), Google Cloud Compute Engine (2012) ou Azure Virtual Machines (2010) surgiram como negócios neste paradigma. Estas plataformas são comumente encaixadas sob o termo guarda-chuva IaaS *Infrastructure as a Service* – Infraestrutura como um Serviço).

Uma abordagem diferente é utilizada por serviços PaaS (*Platform as a Service* – Plataforma como um Serviço). Neste modelo, não se especifica a infraestrutura na qual o serviço é executado, mas sim a plataforma (bibliotecas, ambientes de execução) na qual se executa. Neste modelo se destacam Heroku (2006) e Google App Engine (2008). Estas plataformas permitem um nível de abstração acima de máquinas virtuais, mas ainda exigem configurações específicas e carecem de padrões de código aberto que facilitem migração de uma plataforma para outra.

A popularização do uso de contêiners levou ao surgimento de ofertas comerciais que os suportassem. Largamente baseados no sucesso do Docker (2013) como principal ferramenta de conteinerização, desenvolvedores e administradores de sistema se tornaram aptos a poder implantar soluções de maneira reproduzível em diferentes fornecedores ou localmente, se necessário - com uma necessidade de configuração automatizável e menor do que um Sistema Operacional completo.

O AWS Lambda (2014) é considerado como o primeiro serviço *Serverless* assim nomeado. Nele, o responsável pela implantação do serviço desenvolve uma função em alguma das linguagens de programação aceitas e esta fica disponível numa URL, sendo ativada de acordo com critérios pré-especificados, tais como uma chamada HTTP ou um evento disparado por uma fila de mensagens.

5.1.2. Computação em nuvem moderna

Nos últimos 10 anos as tecnologias em nuvem conseguiram maximizar a utilização por meio de virtualização e reduzir o custo de escalabilidade para grandes *datacenters*. No entanto, os usuários da nuvem continuam a suportar uma carga de operações complexas e nem sempre se beneficiam do compartilhamento de recursos com outros usuários como discutido por Jonas [Jonas et al. 2019].

A computação *Serverless* é a plataforma que abstrai dos desenvolvedores o servidor e roda código sob-demanda que é escalado e contabilizado automaticamente somente pelo tempo em que é executado, assim definido por Castro [Castro et al. 2019]. Além disso, Jonas [Jonas et al. 2019] pontuam que a camada *Serverless* situa-se entre as camadas de aplicação e da plataforma de nuvem subjacentes; simplificando a programação em nuvem.

Em função de sua simplicidade e vantagens econômicas, a computação *Serverless* vem ganhando popularidade como reportado pela taxa de crescimento do termo de busca *Serverless* no Google Trends. O tamanho do mercado é estimado em 7.72 bilhões de dólares até 2021, afirmado por Castro [Castro et al. 2019].

A fundação *Linux Foundation* possui um grupo de trabalho chamado *Cloud Native Computing Foundation* (CNCF), destinada exclusivamente às tecnologias nativamente desenhadas para computação em nuvem. Esse grupo de trabalho possui uma seção dedicada às tecnologias *Serverless* e disponibilizam uma fotografia atualizada com as principais ferramentas em uso desse mercado [CNCF landscape 2020]. Essa fotografia é chamada de *CNCF Serverless Landscape*.

5.1.3. Usos da Computação Serverless

Shafiei [Shafiei et al. 2020] afirmam que a computação *Serverless* se difere da computação em nuvem tradicional no sentido em que a infraestrutura e a plataforma às quais os serviços estão executando são de fato transparentes para o usuário da nuvem. Nessa abordagem o desenvolvedor se preocupa exclusivamente com as funcionalidades demandadas por suas aplicações e todo o resto é delegado ao provedor de serviço em nuvem.

As plataformas *Serverless* permitem uma variedade de usos possíveis. Dentre eles, se destacam:

- Invocação periódica de tarefas de rotina, tais como atualizações de bases de dados;
- Responder a eventos disparados numa fila de tarefas;
- Colaboração em tempo real como chats e sistemas de notificação;
- Ferramentas analíticas que recebem grandes volumes de mensagens para processamento individual que podem escalar facilmente em função do volume de mensagens;
- Uso e cobrança sob demanda em aplicações de serviços urbanos planejados para cidades inteligentes;
- Aprendizado de máquina no uso massivo e paralelo das funções;
- Segurança na forma de funções reativas a detecções de intrusão;
- Internet das coisas.

Essas formas de utilização aproveitam das características de tais plataformas. Tarefas de rotina agendadas precisam ser executadas apenas nos momentos necessários, de

modo que pode ser desnecessário manter um servidor executando apenas em espera deste momento. Uma maneira tradicional de se lidar com agendamentos periódicos em sistemas UNIX, por exemplo, é o utilitário *cron*, que depende da execução contínua do sistema para que a tarefa seja executada nos momentos especificados. Com uma plataforma *Serverless*, fica a cargo do provedor do serviço a ativação periódica, enquanto que a função implantada deve apenas ser executada nestes momentos, sendo cobrada apenas por cada ativação e consequente uso de recursos computacionais.

De maneira similar, funções que respondam a um evento disparado numa fila de tarefas apenas serão ativadas quando houverem eventos para serem processados. Outra característica das plataformas *Serverless* que pode ser aproveitada neste caso é a rápida escalabilidade: em momentos nos quais houver muitos eventos para serem processados mais funções podem ser invocadas mais rápido e a um custo tanto financeiro quanto computacional potencialmente menor do que ativar mais máquinas virtuais ou contêineres.

5.1.4. Potenciais da computação *Serverless*

Por se tratar de uma tecnologia recente em relação a máquinas virtuais e contêineres há potencial para ser explorado com a compreensão e desenvolvimento na tecnologia. Sob a perspectiva de trabalhos de pesquisa, Hendrickson et al. [Hendrickson et al. 2016] falam sobre depuração orientada a custo financeiro, decomposição facilitada de sistemas legados e compartilhamento de ambientes de execução.

Além disso, a computação *Serverless* é uma camada de serviço que realmente abstrai a infraestrutura de nuvem subjacente, minimizando a curva de aprendizado para que desenvolvedores e projetos utilizem programação em nuvem. Potencialmente, por conseguir contabilizar o uso de forma mais granular, reduz os custos com infraestrutura de nuvem para corporações além de discriminar melhor onde os custos foram alocados.

5.1.5. O minicurso

Este minicurso destina-se a apresentar o estado da arte nesta área de conhecimento: Computação *Serverless* (*Serverless Computing* – Computação sem Servidor). Neste modelo de computação, a unidade básica e objeto de trabalho é uma única função escrita numa linguagem de programação de alto nível. Nenhuma das camadas como infraestrutura, máquina virtual, sistema operacional, processo ou *contêiner* são de responsabilidade do usuário da plataforma de computação *Serverless*. Esse modelo requer uma decomposição maior na escrita de aplicações e total dependência sobre a rede e serviços externos. Entretanto, esse modelo fornece capacidades consideráveis de escala conforme discutido por Hendrickson [Hendrickson et al. 2016], granularidade de cobrança e desacoplamento da infraestrutura.

O minicurso está organizado da seguinte forma: A seção 5.2 detalha o paradigma de computação "Função como Serviço". A seção 5.3 explica e detalha o modelo de computação *Serverless*. Na seção 5.4 são apresentadas as principais plataformas. A seção 5.5 mostra projetos de pesquisas utilizando *Serverless*, enquanto na seção 5.6 são mostrados os desafios e limitações em utilizar tal modelo. A seção 5.7 introduz conceitos básicos sobra a plataforma OpenFaas, apresenta também um tutorial de instalação e de como escrever funções, além de descrever alguns exemplos. Finalmente, a seção 5.8 traz a

conclusão e as discussões finais deste minicurso.

5.2. Funções como Serviço

Nesta seção se apresenta o paradigma de funções como serviço a partir do exame do conceito "como serviço" em si e em comparação a modelos precursores.

5.2.1. Paradigma "como serviço"

O paradigma de disponibilização de aplicações denominado "como serviço" tem essa terminologia devido à ocultação de aspectos do usuário [Wang et al. 2018]. Exemplos do paradigma são:

- Plataforma como serviço (PaaS – *Platform as a Service*). Estas ofertas de computação na nuvem fornecem uma plataforma de execução para aplicações, ocultando do desenvolvedor a infraestrutura subjacente. Exemplos de plataforma como serviço incluem o Google App Engine e o Heroku.

- Infraestrutura como serviço (IaaS – *Infrastructure as a Service*). Esta modalidade consiste no fornecimento de componentes de infraestrutura tais como máquinas virtuais ou redes privadas virtuais (VPS – *Virtual Private Server*). Google Cloud Compute Engine, Amazon Elastic Compute Cloud e Azure VMs são exemplos de infraestrutura como serviço.

- Software como serviço (SaaS – *Software as a Service*). Este modelo compreende o software em si sendo oferecido para um cliente por uma interface (web, por exemplo) mas executando remotamente em servidores do provedor. Exemplos de tal tipo de software são o cliente de e-mail Gmail ou o Microsoft Office Online.

Nesta tendência, o paradigma Funções como serviço (FaaS – *Functions as a Service*) fornece uma perspectiva similar. Ao invés de se provisionar infraestrutura ou a própria plataforma na qual o código é executado, se fornece uma estrutura que irá executar uma unidade de código em si. Abstraído para o desenvolvedor se encontram a plataforma subjacente, sendo necessário apenas fornecer uma listagem de dependências do código; a maneira exata pela qual as execuções de função devem escalar, sendo relacionadas às limitações de escala; e a parte de conectividade da função, sendo que o provedor fornece um identificador de recurso uniforme (*Uniform Resource Identifier* – URI) para a função e o desenvolvedor apenas lida com os dados da requisição e o que deve ser devolvido como resposta.

Entretanto, os detalhes deixados fora do controle do desenvolvedor da aplicação continuam sendo importantes. Uma organização que se utilize de funções como serviço pode integrá-las a uma estrutura maior em seu provedor de computação na nuvem. Um exemplo se encontra nas redes privadas virtuais, para as quais pode-se criar regras sobre quais clientes podem acessar quais funções. O fato de se remover certas preocupações sobre em qual estrutura serão executadas as funções não significa que elas devam ser removidas de toda a cadeia de desenvolvimento.

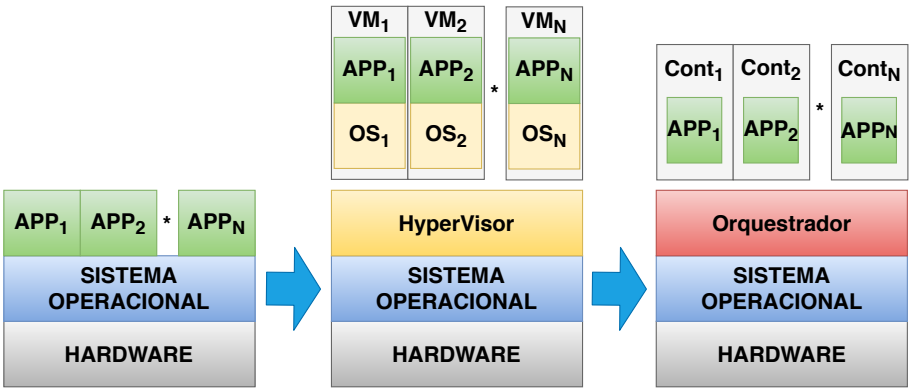

Figura 5.1. Evolução dos serviços de virtualização.

5.2.2. Evolução: *bare-metal*, virtualização, contêineres

Jonas et al. [Jonas et al. 2019] e Hendrickson et al. [Hendrickson et al. 2016] descrevem um caminho evolutivo na maneira em que aplicações e serviços foram disponibilizados.

Inicialmente, no modelo *bare-metal*, cada aplicação roda diretamente sob uma maquina física, contando apenas com o sistema operacional como intermediário. Apesar das vantagens apresentadas no desempenho da aplicação, visto que todos os recursos da máquina estavam disponíveis para ela, o alto custo e a dificuldade de prover e gerenciar esse tipo de sistema levaram ao advento da virtualização.

Em sistemas virtualizados é adicionada uma camada extra sobre o sistema operacional, chamada de *hypervisor* que permite executar vários sistemas operacionais de maneira isolada sobre o mesmo hardware. Isto resulta em maior flexibilidade e torna a administração destes sistemas mais simples. O gerente da aplicação não precisa mais se preocupar com questões relativas à máquina física e os custos de implantação são reduzidos pois não é mais necessário adquirir um novo servidor para cada aplicação bastando apenas instanciar uma nova máquina virtual.

Mais recentemente surgiram os contêineres, que nada mais são do que um sistema de virtualização em nível de sistema operacional, ou seja, permitem que aplicações sejam executadas de forma isoladas em espaço de usuário. Nestes sistemas as funções básicas do sistema operacional são compartilhadas por todos os contêineres permitindo encapsular as aplicações de modo muito mais enxuto e eficiente que as máquinas virtuais.

A figura 5.1 apresenta um diagrama dessa evolução dos serviços de virtualização. A primeira parte da figura ilustra as aplicações que executam diretamente em cima do sistema operacional. A seguir se observa o sistema virtualizado com a presença do *hypervisor* e as aplicações sendo executadas em máquinas virtuais. Finalmente, na terceira parte da figura pode-se observar as aplicações sendo executadas em contêineres com a presença do orquestrador.

5.2.3. Comunicação em contêineres

O modelo de redes para contêineres [Church et al. 2020] fornece uma série de *drivers* de redes nativos que podem ser escolhidos com base nos requisitos da aplicação, são eles

Host, *Bridge*, *Overlay*, *MACVLAN* e *None*. Contêineres que utilizam o driver *Host* usam a pilha de rede do hospedeiro sendo que não existe separação de *namespace* e todas as interfaces do hospedeiro podem ser usadas diretamente pelo contêiner.

O *driver Bridge* provê uma rede *bridge* entre o hospedeiro e o contêiner a ser gerenciada pelo orquestrador. Os contêineres conectados à mesma rede *bridge* podem se comunicar entre si provendo isolamento dos que não estão conectados. É o tipo padrão de rede, que o orquestrador usa quando o *driver* não é especificado.

O *driver Overlay* possibilita a comunicação entre mais de um *host* que rodam um orquestrador de contêineres, possibilitando assim a criação de *clusters*. Assim, como em redes do tipo *bridge*, os contêineres conectados a mesma rede *overlay* podem se comunicar, porém neste tipo de rede isso é possível para contêineres rodando em diferentes hospedeiros.

No *driver* do tipo *MACVLAN* cada contêiner tem um endereço MAC atribuído permitindo que ele seja usado com aplicações legadas que precisam de acesso direto à rede local e aplicações que realizam monitoramento de rede. Neste tipo rede é necessário associar cada interface de rede criada a uma interface "pai", geralmente a interface física do hospedeiro, como o acesso é feito diretamente a rede física é necessário um *gateway* para acesso externo, isto permite também o isolamento do trafego através da associação de cada sub-interface criada a uma *Vlan* diferente.

Por fim, o *driver* do tipo *None* é usado para contêineres isolados que não precisam se comunicar nem com outros contêineres nem com redes externas.

5.3. Modelo Serverless

O modelo de computação *Serverless* ("sem servidor") é assim chamado não por dispensar por completo o uso de servidores (máquinas, físicas ou virtuais, que disponibilizam aplicações *online*) mas por retirar do desenvolvedor a responsabilidade de configurar o recurso computacional, deixando apenas a escrita das funções que compõem o artefato desenvolvido [Hendrickson et al. 2016].

É possível argumentar que, configuraria um uso do modelo *Serverless*, o caso de aplicações nas quais um grupo de pessoas configura os servidores e outro grupo desenvolve os programas. Desta maneira, entretanto, tem-se uma divisão na qual ainda se tem a necessidade e o conhecimento sobre a plataforma computacional na qual são executados os programas por algum componente do projeto; o modelo *Serverless* deixa a cargo do provedor de serviço da aplicação (por exemplo, AWS [AWS Lambda 2014], Azure [Azure Functions 2016] ou Google Cloud [Google Cloud Functions 2016]) a infraestrutura computacional – desde a disponibilização básica até a escalabilidade do serviço. Nenhum membro da equipe possuiu acesso a uma máquina virtual ou contêiner individual, apenas à aplicação em si.

5.3.1. Sem servidor mesmo?

Conforme mencionado na seção 5.3, membros da equipe de desenvolvimento não possuem acesso aos elementos da infraestrutura, sejam eles máquinas físicas, virtuais ou contêineres. Comercialmente se trata de um argumento de vendas que reduz a necessidade

de se gastar recursos com a administração de servidores: a equipe de desenvolvimento consegue colocar em produção um sistema sem interagir com a infraestrutura.

Essa abordagem, no entanto, possui limitações. Funções no modelo *Serverless* são inerentemente sem estado, seguindo o princípio de que elas são horizontalmente escaláveis e independem da infraestrutura subjacente. Assim sendo, sistemas que necessitem de dados persistentes devem acessá-los de outra fonte - bancos de dados, por exemplo. O banco de dados em si deve ser configurado. Fornecedores de serviço na nuvem, como os já citados, fornecem também armazenamento-como-serviço, de maneira que o desenvolvedor pode ligar sua aplicação a algum destes serviços. Esta estrutura limita de fato o contato com a infraestrutura tradicional, mas de certa forma apenas desloca a responsabilidade de um administrador de sistemas convencional para um administrador de sistemas na nuvem. Há vantagens nesta percepção: as tarefas de gerenciamento das máquinas, manutenção e segurança são deslocadas para a plataforma, que em tese possui mais recursos e pessoal para lidar com a infraestrutura do que uma empresa menor. Desvantagens incluem o preço de tais comodidades e o risco de se ficar preso a um fornecedor (*vendor lock-in*).

Com relação aos recursos computacionais, serviços como AWS Lambda ou ferramentas como OpenFaaS [Ellis 2017g] ou OpenLambda [Hendrickson et al. 2016] se utilizam de contêineres para prover isolamento entre as funções e modelo rápido de escala – quanto mais instâncias de execução da função se desejarem simultaneamente, mais contêineres devem ser criados. Estes contêineres, internamente, devem ser controlados via orquestradores como Docker Swarm ou Kubernetes, e toda essa estrutura deve ser executada sobre máquinas tradicionais - virtuais ou físicas. Mais uma vez, todos estes componentes são ocultos do desenvolvedor da aplicação. Assim sendo, no modelo *Serverless* não há servidores visíveis para o usuário final.

5.3.2. Responsabilidades da plataforma

Uma plataforma *Serverless* fornece ao usuário uma localidade na qual ele disponibiliza o serviço para utilização, geralmente por meio de uma URL. Esta disponibilidade pode ser combinada e assegurada por meio de um SLA (*Service Level Agreement*). Escalabilidade do serviço, sob os parâmetros definidos pelo cliente, também deve ser fornecida. Maneiras de se inserir, modificar e monitorar o uso da função devem ser providos. Segurança básica contra ataques DDoS ou vírus na plataforma subjacente são considerações importantes.

5.3.3. Responsabilidades do desenvolvedor

O desenvolvedor da aplicação deve estar atento ao ecossistema que sua aplicação necessita: bancos de dados, acessos a outras funções e recursos. Conhecimento da plataforma *Serverless* utilizada também é importante: tempo máximo no qual uma função pode ser executada, limites de memória e custo dos recursos utilizados. Uma aplicação pouco otimizada possui um custo diretamente mensurável, então além de se identificar possíveis pontos de melhora há também o monitoramento, utilizando ferramentas da plataforma ou auxiliares. Preocupações com segurança também existem do lado do desenvolvedor: informações expostas, níveis de acesso e configuração de permissões.

5.3.4. Precificação

Funções são cobradas por invocação, ou seja, sem custos por recursos não utilizados ou ociosos [Wang et al. 2018]. Nesse modelo, além de fornecer um nível de granularidade que permite medições mais acuradas, o preço também inclui as capacidades de administração de sistemas como redundância, disponibilidade, monitoramento, entre outros [Jonas et al. 2019].

5.3.5. Segurança

Mesmo com a mudança do foco da administração para o provedor da plataforma *Serverless* o desenvolvedor ainda possui responsabilidades com a segurança da aplicação. A mudança de foco em si não exime que se tome este cuidado; de fato, apenas modifica como deve ser realizado ou mesmo insere novos desafios. Em [Jonas et al. 2019] os autores citam como exemplos:

- Randomização de escalonamento e isolamento físico;
- Contextos de segurança de fina granularidade;
- Computação *Serverless* "esquecível".

5.3.6. Modelo Serverful

Serverful é o nome usado por Jonas et al. [Jonas et al. 2019] para descrever a abordagem tradicional dos sistemas em nuvem. Nesta abordagem, o desenvolvedor aluga a infraestrutura de Tecnologia da Informação (TI), como servidores, máquinas virtuais, armazenamento, redes e sistemas operacionais, de um provedor de nuvem, e executa e disponibiliza suas aplicações através dessa infraestrutura.

Porém, diferente do modelo descrito na seção 5.3, ainda é de responsabilidade do desenvolvedor a configuração dos pacotes e serviços necessários à aplicação. O desenvolvedor tem acesso direto a máquina virtual ou contêiner e é o responsável por gerí-lo. No que tange a precificação, o pagamento é realizado por *slots* de tempo, mesmo quando os recursos não estão sendo utilizados. Em casos em que a demanda aumenta, uma nova máquina virtual e/ou contêiner deve ser instanciado, aumentado os custos e, potencialmente, desperdiçando recursos.

Emfim, apesar do modelo *Serverful* ter retirado do desenvolvedor as preocupações com a compra e gerência de hardware, ele ainda não conseguiu abstrair completamente as questões de infraestrutura, objetivo melhor alcançado pelo modelo *Serverless*.

5.3.7. Modelo Serverful x Modelo Serverless

Segundo Jonas et al. [Jonas et al. 2019], o ganho do modelo *Serverless* em relação ao modelo *Serverful* pode ser comparado a transição entre linguagens de alto e baixo nível. Enquanto as linguagens de alto nível libertaram o desenvolvedor do gerenciamento de memória, o modelo *Serverless* libertou-o da gerência do servidor de aplicação.

Existem três principais diferenças do modelo *Serveless* em relação ao *Serverful*, são elas:

Tabela 5.1. Comparação entre os modelos *Serverful* e *Serveless*.

	Serverful	*Serverless*
Escalabilidade	É necessário subir novas máquinas e/ou contêineres caso a demanda aumente. Em caso de baixa demanda, recursos são desperdiçados.	Escala automaticamente, não é necessário nenhuma ação por parte do desenvolvedor da aplicação
Manutenção	Requer manutenção, é preciso instalar, monitorar e manter atualizados os softwares necessários a aplicação	Não requer manutenção, isso fica a cargo do provedor do serviço. Basta ao desenvolvedor escrever e implantar código usando as ferramentas fornecidas por esse provedor.
Custo	É pago por tempo para manter o servidor disponível mesmo que não esteja sendo usado.	É pago por invocação, caso não ocorra o uso não existe cobrança.
Implantação	Permite implantar serviços diretamente. É necessário logar na maquina e rodar o serviço e suas dependências.	É baseado em eventos, o desenvolvedor implanta uma função e ela é executada com resposta a um evento que ocorre no sistema (*triggers* de banco de dados, requisições HTTP).

- O armazenamento e a computação ocorrem em separado, geralmente são ofertados por diferentes serviços de nuvem e a computação é sem estado.
- O desenvolvedor fornece apenas um pedaço de código a ser executado. Os recursos para executá-lo são totalmente fornecidos pelo provedor do serviço.
- O pagamento é realizado pelo recurso usado e não pelo recurso alocado.

A tabela 5.1 apresenta a sumarização das principais características de ambos.

5.4. Plataformas

Atualmente existem diversas plataformas de computação *Serverless* disponíveis. As plataformas podem ser pagas ou de código livre. Esta seção apresenta algumas dessas plataformas, e discute alguns aspectos sobre cada uma dessas das plataformas. É apresentada uma descrição mais detalhada da plataforma OpenFaaS, devido à escolha dela para a parte prática desse minicurso.

5.4.1. AWS Lambda

A AWS Lambda [AWS Lambda 2014] é a plataforma sem servidor da Amazon e faz parte do pacote de serviços Amazon Web Services (AWS). O serviço de computação em nuvem da Amazon foi lançado em 2006 e a AWS Lambda foi lançado em 2014. Segundo

pesquisa realizada pela FLEXERA [Flexera 2019] a AWS é a plataforma em nuvem mais utilizada pelas empresas consultadas. Em consequência desse domínio do mercado, a plataforma de computação *Serverless* AWS Lambda também domina parte significativa do mercado de funções como serviço. A AWS Lambda foi responsável pela popularização do serviço e por moldar, em partes, o ecossistema de aplicações sem servidor.

A AWS Lambda é, como o modelo de computação *Serverless*, orientada a eventos, o que significa dizer que as funções Lambdas são executas como resposta a algum evento. Os eventos podem ser desde uma requisição HTTP, alguma solicitação de alguma aplicação específica ou algum gatilho periódico contido nas configurações. A AWS não fornece detalhes sobre a arquitetura ou funcionamento da AWS Lambda. Contudo, as funções usam um serviço personalizado, chamado Firecracker, para criar máquinas virtuais leves.

A AWS já conta com um extenso escopo de serviços e plataformas de computação em nuvem. Sendo assim, uma das vantagens em utilizar sua plataforma de computação sem servidor reside no fato de ser facilmente integrado com as outras aplicações e serviços oferecidos. Além disso, o AWS Lambda conta com vasta documentação, tanto da própria Amazon quanto de usuários de seus serviços.

Apesar de a AWS Lambda ser um serviço pago, a Amazon oferece 12 meses de acesso gratuito. O acesso ao nível gratuito para os serviços AWS, até o momento da escrita desse minicurso, fornece até 1 milhão de chamada de funções por mês. A AWS fornece uma granularidade de 100 milissegundos para o tempo de execução das funções, o que resultaria em 100 mil segundos de computação. Entretanto, o nível gratuito oferece mais de 3 milhões de segundos de computação.

As funções *Serverless* executadas na plataforma AWS Lambda podem ser escritas utilizando diversas linguagens. Até o momento da escrita desse minicurso as linguagens suportadas pela AWS Lambda são, Java, Go, PowerShell, JavaScript[1], C#, Python e Ruby.

5.4.2. Azure Functions

A Azure Functions [Azure Functions 2016] é a plataforma sem servidor da Azure. A Azure é a oferta de computação em nuvem oferecida pela Microsoft, sendo lançada em 2010 inicialmente sob nome de Windows Azure, sendo renomeada para Microsoft Azure. Segundo a pesquisa da FLEXERA [Flexera 2019] a Microsoft Azure é a segunda oferta *cloud* mais utilizada no mercado, ficando atrás somente da AWS.

Assim como a AWS, a Microsoft Azure não disponibiliza maiores detalhes sobre a arquitetura da Azure Functions. Entretanto, o Azure Functions oferece um subproduto chamado *Durable Functions* cujo serviço permite escrever funções com estado dentro de um ambiente *Serveless*. Contudo, esse serviço só consegue manter estado entre duas chamadas de funções do modelo sem servidor, funcionando mais como um *checkpoint* entre duas funções e possui certas restrições como de linguagem ou de serviços suportados.

A Microsoft Azure conta com vários serviços de computação em nuvem e podem ser facilmente integrados com o Azure Functions. Além disso pode ser usado com segurança em requisições HTTP de outros provedores como Facebook, Google e Twitter. No momento de escrita deste artigo, o Azure Functions suporta as linguagens C#, JavaS-

[1] Através do interpretador Node.js

cript[2], F#, Java, PowerShell, Python e TypeScript. Já o ambiente Durable Functions dá suporte somente as linguagens C#, JavaScript e F#.

O serviço Microsoft Azure também é um serviço pago. Entretanto, também oferece 12 meses gratuitos. Ele conta com vasta documentação da Microsoft.

5.4.3. Cloud Functions

A Cloud Functions [Google Cloud Functions 2016] é a plataforma *Serverless* da Google Cloud. A Google começou a oferecer serviços de computação em nuvem em 2008, passando a integrar computação sem servidor em 2016. O Cloud Functions promete execução local ou em nuvem. Assim como descrito nas seções 5.4.1 e 5.4.2 a Cloud Functions não disponibiliza maiores detalhes de sua arquitetura.

Assim como as demais plataformas a Cloud Functions não necessita de provisionamento de recursos e se integra facilmente aos demais serviços oferecidos pela plataforma Cloud. A Google também oferece 12 meses de acesso gratuito a plataforma.

A Cloud Functions, dentre as 3 plataformas citadas até aqui, é a que dispõe de menos documentação sobre seus serviços. As linguagens suportadas até o momento da escrita desse minicurso são JavaScript, Go e Python.

5.4.4. OpenFaaS

Começou como um projeto pessoal de Alex Ellis em Outubro de 2016. Alex queria executar funcionalidades da Alexa e funções Lambdas como serviços sobre Docker Swarm. Em Dezembro de 2016, com o sucesso inicial de seus resultados, Alex lançou a primeira versão que denominou OpenFaaS, escrita na linguagem Golang e publicada no GitHub [Ellis 2016]. Após a primeira publicação, o projeto ganhou mais de 4.000 estrelas no GitHub impulsionando o conhecimento do projeto na comunidade e indústria da área. Em Abril de 2017, em Austin, Alex ganhou a oportunidade de participar da sessão de palestras da Moby's Cool Hacks em Dockercon. A missão de Alex era usar o OpenFaaS para ultrapassar os limites de desempenho do Docker, onde obteve sucesso. Após estes acontecimentos o projeto do OpenFaaS tornou-se uma plataforma *Serverless* com suporte de 136 colaboradores ativos em seu repositório oficial no GitHub [Ellis 2017f].

A plataforma OpenFaaS é de código aberto e baseado na licença MIT [MIT 1980]. Atualmente, OpenFaaS presta serviços comerciais, suporte a usuários, e gerenciamento de patrocínios na página oficial através do OpenFaaS Ltd responsável pela marca. No OpenFaaS, funções são executadas sobre contêineres de forma escalável baseado no aumento do número de funções. Além disso, todo o gerenciamento de operações com funções pode ser realizada via interface gráfica ou interface de comando atendendo níveis de usuários diferentes, do iniciante ao avançado. A plataforma do OpenFaaS também pode rodar sobre hardwares existentes no mercado, ou em servidores de nuvem privados ou públicos sendo totalmente portátil entre ambos ambientes.

Funções *Serverless* sobre OpenFaaS podem ser escritas nas linguagens de programação Go, C#, JavaScript, Java, Ruby e PHP. Estas linguagens são oficialmente suportadas pela plataforma, no entanto, usuários podem adicionar outras linguagens. Na

[2]Através do interpretador Node.js

seção 5.7.3 descrevemos como adicionar uma nova linguagem na plataforma.

5.4.4.1. Arquitetura

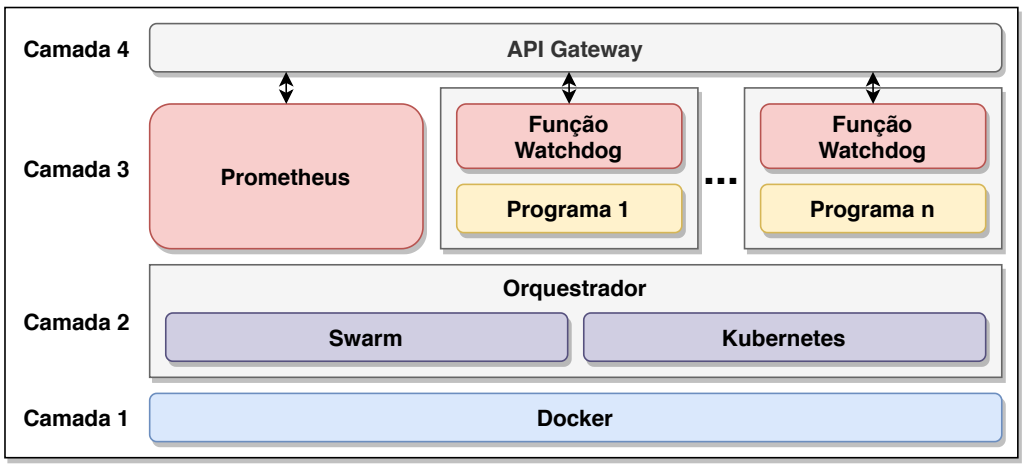

Figura 5.2. Pilha da arquitetura OpenFaaS.

A arquitetura do OpenFaaS é baseada no modelo de pilha de componentes. Neste modelo qualquer processo pode ser transformado em função *Serverless* sobre um contêiner Docker. A figura 5.2 apresenta a pilha da arquitetura OpenFaaS composta por quatro camadas. Nesta seção descrevemos cada camada da pilha OpenFaaS na ordem *top-down*, ou seja, da camada quatro (mais abstrata) até a camada um (menos abstrata).

Camada 4: Essa camada tem como objetivo gerenciar o processo de comunicação entre usuários e funções *Serverless* como serviços utilizando o protocolo HTTP. O componente API Gateway é o responsável por gerenciar esse processo. Ele recebe e redireciona requisições HTTP enviadas entre usuários e serviços de acordo com a operação e status do serviço. Além disso, ele cria e exclui réplicas de serviços usando o orquestrador (Camada 2) de acordo com o aumento no número de requisições. O API Gateway ainda utiliza uma interface Web intuitiva, denominada Portal UI para facilitar a iteração do usuário com serviços no OpenFaaS. Como medida de segurança e desempenho o API Gateway aloca um contêiner próprio sobre OpenFaaS, evitando que uma função seja requisitada mais de uma vez enquanto estiver sendo utilizada por um usuário específico.

Camada 3: Ocorre o processo de comunicação entre o usuário e funções *Serverless*, e o monitoramento de estatísticas do OpenFaaS. Uma função criada pelo usuário será executada em um contêiner específico. Por padrão, o contêiner da função é composto por uma instância da função chamado *Watchdog*, e um programa com a função a ser processada no contêiner. O programa com a função pode ser escrito em qualquer linguagem de programação desde que adicionado o suporte no OpenFaaS. A função *Watchdog* funciona como um mecanismo de comunicação intermediário entre API Gateway e o programa criado pelo usuário. Na função *Watchdog* existe um processador HTTP interno para realizar o procedimento de comunicação.

A figura 5.3 apresenta a visão geral da comunicação entre API *Gateway*, *Watchdog*

e programa da função. Por exemplo, o fluxo de execução de uma função que soma dois números ocorre da seguinte forma: uma requisição com dois números inteiros é enviada pelo usuário para o processador HTTP que processa a requisição, e envia os parâmetros para o programa da função através da entrada padrão. O programa da função realiza a operação soma de dois números no contêiner da função. Em seguida, o resultado da soma retorna para o processador HTTP que transforma esse resultado em uma resposta. Por fim, a resposta é enviada e recebida na interface do usuário.

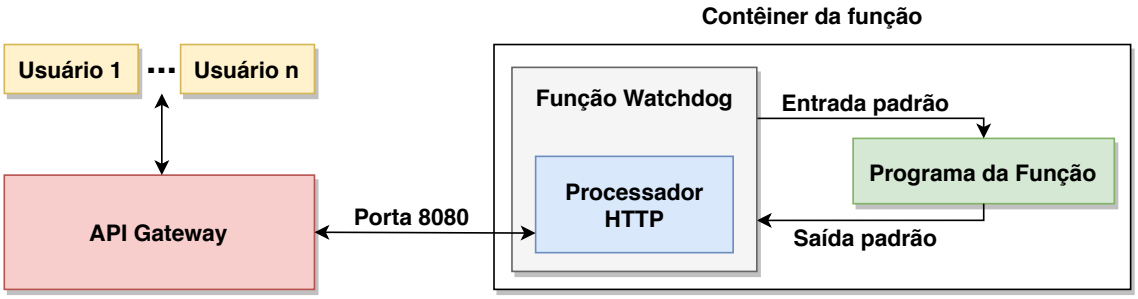

Figura 5.3. Comunicação entre API *Gateway* e contêiner da função.

O monitoramento de estatísticas no OpenFaaS ocorre através da ferramenta Prometheus, que é um programa mantido pela Linux Foundation [Linux Foundation 2016]. Prometheus foi adicionado ao OpenFaaS para monitorar serviços de acordo com a requisição das funções. Ele permite analisar funções do OpenFaaS produzindo informações relevantes, por exemplo, tempo médio de execução da função, número de requisições recebidas, e níveis de escalabilidade. A informação medida pelo componente é coletada toda vez que a requisição para uma função é detectada pelo API *Gateway*. OpenFaaS também suporta o uso de ferramentas de *dashboards* dinâmicos (por exemplo, Grafana [Grafana Labs 2014]) integradas com o Prometheus para visualização iterativa de informações em tempo de execução.

Camada 2: É onde ocorre a gerência dos contêineres através do componente denominado orquestrador. O orquestrador é responsável por configurar e coordenar contêineres na rede de forma automatizada. Contêineres podem ser alocados em servidores distintos pelo orquestrador de modo horizontal na rede, comunicando entre si como se estivessem lado a lado contidos no mesmo hardware. O OpenFaaS suporta dois tipos de orquestradores: Docker Swarm e Kubernetes.

Docker Swarm (ou Swarm) foi o primeiro orquestrador usado no OpenFaaS desde a primeira versão lançada em 2016. Swarm é um orquestrador simples, porém poderoso e nativo do Docker [Docker Inc. 2014]. Ele é recomendado para projetos simples e para usuários iniciantes no OpenFaaS. Swarm atua como uma ferramenta de clusterização que orquestra contêineres de acordo com a demanda provendo escalabilidade. Além disso, cria um cluster de contêineres utilizando apenas o comando `$ docker swarm init`. Através deste comando, a máquina hospedeira torna-se o nó principal do cluster. Novos nós podem ser adicionados com a alocação de um novo contêiner. Swarm é quem decide qual contêiner será alocado, simplificando o processo operacional de criação de novos nós do cluster de acordo com o recurso disponível no hospedeiro.

Kubernetes ou K8s foi criado pela Google [Google Inc. 2015] para gerenciar contêineres da rede interna da empresa. Em 2015, a empresa transformou o K8s em código aberto. A partir disso, o K8s passou a ser mantido pela Linux Foundation. Kubernetes cria ambientes *Serverless* com maior eficiência sobre contêineres. Além disso, apresenta algumas vantagens em relação ao Swarm, por exemplo, consumo menor de recursos do hospedeiro e da rede. Ele também permite implementar contêineres para uso local e na nuvem realizando pequenas alterações. Kubernetes é recomendado em projetos que demandam estabilidade, eficiência e baixo tempo de resposta. Entretanto, não existem restrições ao usar o Kubernetes em projetos mais simples. Em 2017, o OpenFaaS passou a ter suporte ao Kubernetes através da ferramenta faas-netes desenvolvida pelo criador da plataforma e disponibilizada em um repositório exclusivo [Ellis 2017a].

Camada 1: É o coração da plataforma OpenFaaS. O OpenFaaS utiliza contêineres para encapsular funções no formato de microsserviços. Nesta camada contêineres são criados através do Docker também denominado motor Docker. Docker é um sistema de virtualização que permite encapsular funções como serviços em contêineres. Contêineres são estruturas que possuem um sistema de arquivo próprio, executam processos, e contêm interfaces de rede. Eles compartilham recursos do *kernel* do sistema operacional hospedeiro com outros processos. Além disso, são rápidos e leves em relação à máquinas virtuais. Por exemplo, um contêiner construído a partir da imagem oficial do Linux Alpine [Alpine Linux 2018] gasta aproximadamente 3 segundos para inicializar e consome apenas 8 MB de espaço em disco.

5.4.4.2. Funcionalidades

Nesta seção descrevemos as principais funcionalidades do OpenFaaS em relação às demais plataformas descritas neste minicurso. O OpenFaaS é uma plataforma de código aberto que possibilita a implementação de funções como serviço sobre contêineres totalmente escalável. O OpenFaaS fornece uma interface simples e amigável, onde funções são executadas usando poucos comandos ou via interface gráfica com apenas um clique do mouse. No OpenFaaS todo processo de instalação de serviços básicos, por exemplo, API Gateway e Prometheus ocorre executando o *script* `deploy_stack.sh`.

Funções *Serverless* criadas sobre OpenFaaS são totalmente independentes, sem um tipo de linguagem de programação estabelecida. No OpenFaaS o usuário pode definir qual linguagem de programação quer escrever suas funções, não estando preso a uma linguagem específica. Funções *Serverless* sobre o OpenFaaS também podem ser executadas em *clusters* gerenciados pelos orquestradores Swarm e K8s, como se estivessem no mesmo hardware. O OpenFaaS oferece suporte a plataforma OpenFaaS Cloud [Ellis 2017b], uma versão melhorada do OpenFaaS para gerência de projetos que contêm equipes de desenvolvimento grande. O OpenFaaS Cloud suporta Git e protocolo HTTPS para atender a demanda de datacenters, e está disponível em um repositório exclusivo desenvolvido pelo criador da plataforma junto à comunidade.

Funções assíncronas são funções que podem levar vários segundos para executar ou inicializar, e o usuário não precisa do resultado da função. Elas são interessantes em cenários como aprendizagem de máquina usando TensorFlow, requisições de tra-

balho em lote e limitação na taxa de funções Lambdas. OpenFaaS suporta processamento de funções assíncronas através de uma fila distribuída. A implementação desta fila é baseado no projeto NATS Streaming [Ellis 2017c], mas pode ser estendida para ser usado com Kafka [Ellis 2017d, Kafka 2011] ou qualquer outra estrutura similar a uma fila [Ellis 2017f].

5.4.5. OpenWhisk

A plataforma OpenWhisk [OpenWhisk 2016] é uma plataforma de computação sem servidor desenvolvida pela IBM e conta com suporte Apache. Ela é uma plataforma de código aberto e pode ser implantada sobre diferentes sistemas. OpenWhisk utiliza-se de outras plataformas que dão suporte à sua arquitetura, Docker [Docker Inc. 2008], Nginx [Igor Sysoev 2005], Kafka [Kafka 2011] e CouchDB [CouchDB 2005].

A plataforma pode ser invocada em uma instância local ou em algum provedor em nuvem. OpenWhisk possui uma interface cliente via linha de comando chamada “wsk"ela pode ser usada para criar, executar e gerenciar alguma instância OpenWhisk.

As linguagens suportadas pela plataforma, até o momento de escrita desse minicurso são JavaScript[3], Go, Python, Java, PHP, Ruby, Swift e .NET. O modelo de programação OpenWhisk é baseado em três pontos: Ações, Gatilhos e Regras. As Ações são as funções *Serverless*, também conhecidas e citadas nesse minicurso como Lambdas. Os Gatilhos são os disparados sempre que um evento ocorra, um evento pode ter várias origens, por exemplo uma mensagem chegando em uma fila de mensagens ou a chegada de dados de um dispositivo IoT. As Regras associam um Gatilho a uma Ação.

5.4.5.1. Arquitetura

A figura 5.4 apresenta o fluxo de trabalho da plataforma OpenWhisk. Primeiro há uma solicitação HTTP, a aplicação *wsk* traduz os comandos para requisições HTTP para o sistema OpenWhisk.

O ponto de entrada da plataforma é a plataforma Nginx. O Nginx é um servidor web orientado a eventos. Lançado em 2004, ele é responsável por uma grande parcela do tráfego global, considerando apenas sites ativos. Após processar a solicitação e, não havendo mais nada as ser feito pelo Nginx, a solicitação é repassada ao controlador. O controlador é uma aplicação que serve como interface para as solicitações do usuário. O controlador traduz uma solicitação do usuário para uma ação existente no sistema. Após a tradução o controlador autentica as permissões do usuário que está fazendo a solicitação. A autenticação e autorização é feita através de uma instância do banco de dados CouchDB. Após a validação de que o usuário tem os privilégios necessários o controlador consulta novamente o banco de dados carregando a ação. O registro da ação contém os parâmetros que deseja-se transmitir a ação, o código a ser executado na ação, além de conter as restrições de recursos do sistema que serão detalhados na seção 5.4.5.2.

O controlador possui uma visão geral do sistema e fica constantemente verificando o estado do sistema, possui também um balanceador de carga e através dele escolhe um

[3]Através do interpretador Node.js

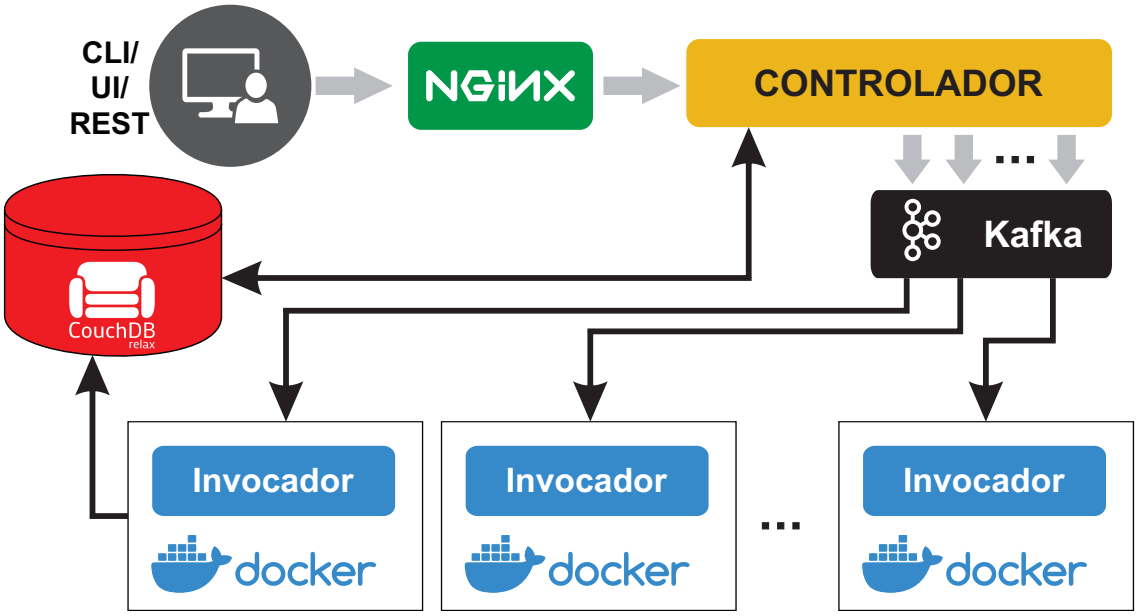

Figura 5.4. Arquitetura do OpenWhisk.

dos invocadores disponíveis para invocar a ação solicitada. Exitem dois problemas, mais graves, que podem ocorrer nesse ponto. O primeiro deles é o sistema falhar perdendo a invocação, e o segundo, o sistema está com uma demanda tão alta que precisa esperar que outras chamadas sejam concluídas. A solução usada para solucionar ambos é um serviço de mensagens chamado Kafka. Ele é um sistema de mensagens de alto rendimento, basicamente ele desacopla os produtores de fluxo dos consumidores de fluxo garantindo através de persistência a entrega das mensagens, no caso do OpenWhisk ele desacopla o controlador (Produtor) dos invocadores (Consumidor) e toda a comunicação entre o controlador e os invocadores é feita utilizando o Kafka.

O invocador tem como objetivo invocar uma ação. Para executar ações o invocador utiliza o Docker para configurar e gerar um contêiner. Para cada ação chamada é criado um contêiner para ela, o código da ação é inserido dentro do contêiner. O contêiner, então, é executado usando os parâmetros passados, o resultado da computação é obtido pelo invocador e o contêiner é destruído. Os resultados obtidos pelo invocador são salvos no banco de dados de ativações, eles residem no CouchDB. O registro das invocações e dos resultados podem ser acessados.

5.4.5.2. Detalhes do Sistema

Nesta sessão serão descritos alguns detalhes de sistema da plataforma OpenWhisk. As Ações, Gatilhos e Regras pertencem, sempre, a um *namespace*[4] e, opcionalmente a um *package*. Um *package* pode conter ações. Entretanto não se pode fazer aninhamento de *packages*, ou seja, um *packages* não pode conter outro.

As Ações possuem alguns limites, como por exemplo, uso de memória e tempo

[4]Optou-se por manter os mesmos nomes usados na plataforma.

Tabela 5.2. Parâmetros de sistema da plataforma OpenWhisk.

Limite	Descrição	Configurável	Unidade	Padrão
timeout	Um contêiner não está autorizado a executar mais do que N milisegundos	por Ação	ms	60000
memory	Um contêiner não está autorizado a alocar mais do que N Megabytes	por Ação	MB	256
logs	Um contêiner não está autorizado a escrever mais do que N Megabytes na saída padrão	por Ação	MB	10
concurrent	Não pode ser enviado mais do que N ativações por *namespace* em execução ou na fila para execução	*namespace*	número	100
minuteRate	Não pode ser enviado mais do que N ativações por *namespace* por minuto	*namespace*	número	120
codeSize	Tamanho máximo que o código da Ação pode ter	por ação	MB	48
parameters	Tamanho máximo dos parâmetros que podem ser inseridos	não configurável limite por Ação/*package*/Gatilho	MB	1
result	Tamanho máximo do resultado de uma Ação	não configurável limite por Ação	MB	1

máximo de computação. Esses limites podem ser configurados e cada limite possui uma granularidade diferente. A tabela 5.2 mostra os limites que podem ser configurados e o nível que pode ser configurado se é por Ação, por *namespace* ou por *package*. Quevedo et al. [Quevedo et al. 2019] avaliaram o desempenho da plataforma para as linguagens Java e JavaScript. Eles avaliaram a plataforma utilizando os limites padrão e o que eles consideraram a melhor configuração para a plataforma.

A Plataforma OpenWhisk conta com vasta documentação e, como já foi dito, com suporte Apache. Além disso, conta com diversos trabalhos acadêmicos que avaliaram ou usaram a plataforma em suas pesquisas.

5.4.6. OpenLambda

A plataforma de computação sem servidor OpenLambda [Hendrickson et al. 2016], desenvolvida em 2016, teve sua implementação norteada pela plataforma *Serverless* da Amazon. Ela foi escrita em linguagem Go e conta com licença da Apache [Apache 2004].

A documentação para esta plataforma é escassa. O foco do projeto é criar uma plataforma com a abordagem em computação sem servidor. A plataforma não faz apresentação de sua arquitetura, nem todas as linguagens suportadas pela plataforma. Entretanto, pode-se constatar que a plataforma aceita linguagens de programação interpretativas de compilação *Just-in-time*. As linguagens, até o momento de escrita desse minicurso são Java, JavaScript e Python.

5.4.7. Comparativo

As plataformas de computação *Serverless* apresentadas neste minicurso podem ser diferenciadas entre plataformas comerciais (AWS Lambda, Azure Functions e Cloud Functions) e de código aberto (OpenFaaS, OpenWhisk e OpenLambda). Entretanto, algumas das plataformas de código aberto oferecem funções como serviço em seus servidores. São elas OpenFaaS e OpenWhisk.

A AWS Lambda, pioneira no paradigma FaaS, tem como forte atrativo a estreita integração com outros serviços *cloud* oferecidos pela Amazon. A plataforma Azure Functions, também conta com fácil integração com outros serviços em nuvem da Microsoft Azure. Além disso, Azure permite que as funções sejam testadas localmente antes de serem carregadas para a plataforma. O Cloud Functions se diferencia dos demais por configuração personalizada para IoT através do serviço de mensagens da Google globalmente distribuído. O Cloud Functions pode ser facilmente integrado a serviços de terceiros.

As plataformas de código aberto apresentadas neste minicurso representam soluções para pesquisas e utilização de computação sem servidor. A plataforma OpenFaaS rapidamente ganhou destaque. Pode-se destacar como um grande atrativo para a adoção da plataforma é a grande quantidade de linguagens de programação oficialmente aceitas pela plataforma. Além disso, ela conta com suporte para que o usuário acrescente alguma linguagem que queira usar. Graças ao sucesso que a plataforma obteve a documentação é ampla e bastante completa.

A plataforma OpenWhisk tem como atrativo o suporte da Apache e faz parte da plataforma *cloud* da IBM. Além disso, conta com ampla gama de linguagens de programação. A OpenWhisk, assim como a OpenFaaS, dá suporte a inserir novas linguagens de programação. Entretanto, para isso é necessário uma ferramenta diferente do “wsk", o que pode tornar a inserção um pouco mais complexa.

A tabela 5.3 apresenta um resumo das plataformas mostrando as linguagens suportadas e outras informações necessárias.

5.5. Projetos de pesquisa

Nesta seção descrevemos tópicos de pesquisa *Serverless* de seis áreas da computação. Os tópicos abordados nesta seção foram baseados em artigos publicados no período de 2017 a 2019.

5.5.1. Processamento de vídeo

Processamento de vídeo na Internet aumentou de forma exponencial após a popularidade de provedores de serviço *streaming* como YouTube e Netflix. Ainda assim, alguns desafios em processar vídeo de forma escalável com tempo de resposta em poucos segundos estão em aberto. Processamento de vídeo consome várias CPUs, por exemplo, um vídeo no formato 4K com uma hora de vídeo leva mais de 30 horas de uso de CPUs para processar. Para atender o tempo de resposta em poucos segundos sem atrasos, múltiplos *threads* devem ser invocados funcionando em paralelo com múltiplas CPUs. Além disso, alguns tipos de vídeos contêm codificadores que não possibilitam processar o vídeo de forma paralela em milhões de partes. *Serverless* apresenta características que enquadram

Tabela 5.3. Tabela comparativa de plataformas de computação sem servidor.

Plataforma	Linguagens Suportadas	Infraestrutura	Virtualização	Gatilhos	Tempo máximo de computação	Cobrança
Amazon Lambda	C#, Go, Java, Powershell, Ruby, Python, Node.js	Nuvem	Firecracker (KVM)	HTTP, serviços da AWS	900	Requisição, tempo de execução memória
Azure Functions	C#, F#, Java, Python, JavaScript	Nuvem, Local	Imagens de SO	HTTP, serviços da Azure	600	Requisição, tempo de execução
Cloud Functions	BASH, Go, Node.js, Python	Nuvem, Local	Não definido	HTTP, Pub/Sub, Google storage	450	Requisição, tempo de execução memória
OpenFaaS	Go, C#, JavaScript, Java, Ruby e PHP	Nuvem, Local	Docker	HTTP, FaaS-CLI	Indefinido	Cloud: Não definido Local: Não se aplica
OpenWhisk	JavaScript, Go, Python, Java, PHP, Ruby, Swift e .NET	Nuvem, Local	Docker	HTTP, IBM Cloud, wsk	300	IBM Cloud: Requisição, tempo de execução Local: Não se aplica
OpenLambda	Java, JavaScript e Python	Nuvem, Local	Docker	HTTP	Indefinido	Não se aplica

no contexto de processamento de vídeo como escalabilidade, paralelismo, e processamento em poucos segundos. No entanto, segundo Fouladi et al. [Fouladi et al. 2017], alguns pontos como desempenho e custos no processamento de vídeo usando *Serverless* não estão claros.

Zhang et al. [Zhang et al. 2019] apresentam um estudo sobre os desafios do processamento de vídeo utilizando computação *Serverless*. Neste trabalho esquemas de implementação e configuração de funções padrão utilizadas em processamento de vídeo foram implementados e avaliadas em duas plataformas *Serverless*, AWS Lambda e GCF (*Google Cloud function*). Para realizar esta análise foram utilizadas as métricas de medição de impacto da alocação de recursos de funções referente ao consumo de memória, esquemas de implementação de funções locais ou APIs externas, comportamento do vídeo em relação à duração, e o custo monetário.

Como resultado do artigo, os autores destacam que implementar funções de processamento de vídeo não é uma tarefa trivial porque as funções estão limitadas a quantidade de recurso disponível na plataforma, por exemplo, tamanho da memória. Aumentar o tamanho da memória não é uma estratégia viável porque não melhora o desempenho de processamento, mesmo podendo configurar dinamicamente o tamanho da memória de acordo com a demanda da aplicação. Em relação ao custo, tarefas de processamento de vídeo complexas executadas através de API externas são mais caras do que funções implementadas localmente. Por exemplo, realizar detecção de face em um vídeo com o modelo parcialmente treinado localmente tem custo menor em relação à APIs externas com o mesmo resultado. Como último resultado, os autores destacam que o AWS Lambda tem mais vantagens que o GCF em termos de duração de execução e custo monetário utilizando o mesmo esquema de configuração. No entanto, AWS Lambda tem o desempenho afetado devido à restrições de recurso, e não é estável como o GCF.

Ao et al. [Ao et al. 2018] projetaram o Sprocket, um framework *Serverless* para processamento de vídeo. No Sprocket, usuários podem executar um conjunto de operações em paralelo no conteúdo do vídeo de modo modular, criando *pipelines* de processamento. Sprocket processa o vídeo de acordo com a função carregada pelo usuário. O usuário implementa as funções usando linguagem de domínio específico, como um grafo de fluxo de dados acíclico direcionado (GAD). O GAD é composto por vértices responsáveis por executar funções individuais, definidas pelo usuário. As funções são executadas nos dados (quadros individuais do vídeo, grupos de quadros ou segmentos compactados de quadros consecutivos) que chegam no grafo. Após ser processado, o dado é encaminhado para o próximo vértice até chegar nos vértices borda do grafo denominados vértices de saída. No GAD, arestas são denominadas fluxos, e tem a funcionalidade de transmitir dados entre vértices. Um programa GAD é representado como um *pipeline*, tal que, os vértices no DAG representam os estágios do *pipeline* de processamento.

Com Sprocket funções complexas de processamento de vídeo podem ser implementadas facilmente devido ao projeto do framework. Sprocket foi implementado nas plataformas AWS Lamda, Microsoft Azure Function e Google Cloud Function. Como resultado, os autores avaliaram uma variedade de condições para mostrar que o sistema consegue ter paralelismo, baixa latência e baixo custo. No artigo, os autores mencionam que um vídeo de 3.600 segundos custa menos de três dólares por hora de vídeo processado

usando o Sprocket sobre as plataformas *Serverless* avaliadas.

5.5.2. Aprendizagem de máquina

Modelos de aprendizagem de máquina são treinados em *clusters* compostos por máquinas virtuais utilizando em seu fluxo de trabalho processos como pré-processamento, modelo de treinamento e ajuste de hiperparâmetros. Cada processo de treinamento requer uma quantidade diferente de recurso gerando sobrecarga ou subutilização do *cluster*. *Serverless* pode contornar estes desafios escalonando cada processo de treinamento independente da demanda de recurso [Jonas et al. 2019].

Ishakian et al. [Ishakian et al. 2018] avaliaram o desempenho de modelos de rede neural sobre a plataforma AWS Lambda usando o framework de aprendizagem profundo Amazon MXNet. Na avaliação realizada, três modelos de reconhecimento de imagem (SqueezeNet, ResNet e ResNeXt-50) foram testados. Para avaliar estes modelos dois cenários foram monitorados: (i) se o contêiner precisa ser inicializado e se a função Lambda está no estado de execução e termina de ser processada. Isso implica em uma sobrecarga adicional no sistema e maior atraso; (ii) se o contêiner é inicializado e a função está no estado de execução. A sobrecarga de execução é a mesma da função Lambda. As métricas consideradas nestas situações foram o tempo de resposta, tempo de predição e custo da função. Os resultados apresentados demonstraram que, no primeiro cenário, a latência é aceitável, enquanto no segundo cenário ocorre uma sobrecarga significativa.

Feng et al. [Feng et al. 2018] investigaram o treinamento de redes neurais utilizando paralelismo de dados sobre a plataforma AWS Lambda. Neste trabalho foram implementados técnicas de otimização para reduzir latência, custo de desempenho e monetário do processamento de redes neurais utilizando *Serverless*. Os experimentos foram realizados baseado em dois conjuntos de dados CIFAR-10 e MNIST. CIFAR-10 foi treinado por uma rede neural de convolução composta por duas camadas de convolução, duas camadas de *pooling*, duas camadas de normalização, duas camadas totalmente conectadas e uma camada de saída *softmax*. MNIST foi treinado por uma rede neural totalmente conectada, cuja estrutura é investigada através do ajuste do hiperparâmetro. Ambos os conjuntos de dados foram gerados aleatoriamente com um milhão de amostras sendo executadas na plataforma AWS Lambda. Fatores como latência, uso de memória e custo monetário foram monitorados. Os resultados obtidos demonstraram vantagens no desempenho e custo do treinamento de redes neurais sobre a plataforma AWS Lambda.

5.5.3. Computação Científica

Operações com álgebra linear são utilizadas em problemas de computação científica como simulação climática, montagem de genomas e dinâmicas dos fluídos. Esses problemas requerem ser processados em supercomputadores ou *clusters* de alto desempenho. Infelizmente, executar operações com álgebra linear de modo distribuído em grande escala é um desafio para cientistas e analistas de dados devido à restrições de acessibilidade e gerenciamento do *cluster*. *Serverless* fornece acesso à plataformas com grande capacidade de processamento, sem o usuário se preocupar com gerenciamento do *cluster*. Ainda assim, plataformas *Serverless* têm recurso limitado ou subutilizado com paralelismo variando drasticamente ao processar problemas científicos, como apontado por Jo-

nas et al. [Jonas et al. 2019].

Shankar et al. [Shankar et al. 2018] criaram Numpywren, um sistema distribuído *Serverless* que permite executar algoritmos de álgebra linear em larga escala utilizando funções *Serverless*. Numpywren contém uma linguagem de domínio específico denominada LAmbdaPACK. LAmbdaPACK permite usuários desenvolverem algoritmos de álgebra linear como multiplicação de matrizes, decomposição de valor único e decomposição de Cholesky sobre a plataforma AWS Lambda. Todos esses algoritmos tem complexidade cúbica e podem gastar horas para serem processados. Sobre o processamento, Numpywren consegue se adaptar a quantidade de paralelismo disponível e decompor programas em várias CPUs, provendo usabilidade e tolerância à falhas. Numpywren foi comparado ao ScalaPACK (biblioteca FORTRAN industrial) e ao DASK (biblioteca de tolerância à falhas escrita em Python). Os resultados demonstraram que o Numpywren comparado ao ScaLAPACK consome 20 a 30% menos horas de CPU. Em relação a eficiência, para problemas grandes Numpywren é 320% mais rápido que DASK.

Werner et al. [Werner et al. 2018] desenvolveram um protótipo de sistema *Serverless* para multiplicação de matrizes. Para realizar multiplicação de matrizes o sistema precisa de três funções sem estado para obter o resultado geral. Nenhuma dessas funções se comunica entre si, sendo os dados transferidos para nuvem. O sistema é composto por três elementos denominados cálculo, armazenamento, e orquestração. O elemento cálculo processa essas funções. O elemento armazenamento fica responsável por armazenar os dados processados na nuvem. Por fim, o elemento orquestração garante que o processamento ocorra de modo distribuído e compartilhado entre as três funções. O projeto do sistema foi implementado sobre a plataforma AWS Lambda. As funções Lambda foram escritas na linguagem Python porque a plataforma AWS fornece bibliotecas para manipular dados de matrizes e operações com multiplicação. Para validar o sistema três fatores foram avaliados: escalabilidade, capacidade de ajuste e custo de desempenho comparados aos *clusters* Apache Spark11 e Hadoop MapReduce12 gerenciados pelo serviço Amazon EMR13. Os resultados obtidos demonstraram que o sistema tem custos de infraestrutura e operacional baixos se comparados à outras plataformas.

5.5.4. Computação nas bordas

É um paradigma emergente que ganhou atenção da indústria e academia devido à popularidade de tecnologias como 5G, Internet das coisas (IoT) e redes veiculares. Nesse paradigma serviços de computação em nuvem foram estendidos para borda da rede para prover baixa latência, tempo de resposta rápido e mobilidade Khan et al. [Khan et al. 2019]. *Serverless* originalmente foi criado para computação em nuvem, mas tornou-se primordial para computação na borda. Computação na borda apresenta limitações como gerenciamento, produção de grande volume de dados, e alto custo. *Serverless* pode solucionar essas limitações, no entanto, as arquiteturas de plataformas *Serverless* precisam ser adaptadas para suportar computação de dispositivos na borda. Além disso, questões de pesquisa como desempenho, elasticidade, gerenciamento e segurança entre *Serverless* e computação na borda precisam ser reavaliados [Glikson et al. 2017].

Xiong et al. [Xiong et al. 2018] implementaram KubeEdge, uma plataforma construída sobre Kubernetes que permite funções *Serverless* serem executadas sobre nós borda

da rede e servidores de nuvem, em tempo de execução, de modo unificado. KubeEdge utiliza contêineres orquestrados pelo Kubernetes fornecendo um canal de comunicação entre nós borda e servidores de nuvem via protocolo RPC (*Remote Procedure Call*). KubeEdge suporta mecanismos de sincronização e armazenamento de metadados com gerenciamento automatizado de funções sobre contêineres. KubeEdge é composto por quatro componentes: barramento Kube, controlador nó borda, serviço de sincronização de metadados, e núcleo nó borda. Barramento Kube é uma camada de rede virtual que conecta nós borda e máquinas virtuais alocadas na nuvem. Controlador nó borda é um *plugin* do controlador Kubernetes responsável por gerenciar remotamente nós borda, permitindo que serviços sejam implantados via Kubernetes. Serviço de sincronização de metadados ocorre de modo bidirecional entre a borda da rede e nuvem. Por fim, o núcleo nó borda é composto por um agente que roda sobre nós borda. O agente é responsável por inicializar e gerenciar contêineres e funções. No artigo os autores não apresentaram nenhum resultado de desempenho da plataforma.

Vilalta et al. [Vilalta et al. 2017] propuseram TelcoFog, uma arquitetura de computação distribuída que suporta serviços como 5G, NFV e IoT na extremidade da rede integrado com servidores de nuvem. TelcoFog é composto por nós na extremidade da rede, um controlador centralizado e serviços. Nós são integrados à infraestrutura de telecomunicações e podem trabalhar em uma rede sobreposta à rede de operadoras permitindo a integração de serviços. O controlador TelcoFog é responsável por garantir a qualidade dos serviços baseado na modelagem de dados de serviço usando a linguagem YANG (*Yet Another Next Generation*). YANG é uma linguagem de modelagem de dados integrada a arquitetura de gerenciamento e orquestração do operador usada para modelar operações e configuração de dados dos dispositivos da rede. Os serviços rodam sobre a infraestrutura de telecomunicações e o TelcoFog. TelcoFog permite que vários serviços sejam implantados dinamicamente de modo distribuído com baixa latência para operadores. O protótipo foi validado por meio de uma prova de conceito para serviços de IoT. Os resultados apresentados demonstram o tempo gasto em cada componente da arquitetura.

5.5.5. Segurança

Funções *Serverless* são encapsuladas em contêineres. Contêineres são construídos via imagens Docker composta por softwares de desenvolvedores oficiais e da comunidade. Softwares adicionados a imagem Docker podem conter vulnerabilidades de segurança, e essas vulnerabilidades podem comprometer a segurança de dados ou a integridade do sistema. *Hackers* podem utilizar o processamento distribuído *Serverless* para executar sistemas maliciosos para realizarem ataques. Esses sistemas são difíceis de serem detectados devido à estrutura da arquitetura *Serverless* [Wu et al. 2018].

Bila et al. [Bila et al. 2017] propuseram uma arquitetura automatizada para detecção de vulnerabilidades em contêineres utilizando OpenWhisk e Kubernetes. Neste artigo os autores estenderam a API do Kubernetes para criar um mecanismo de quarentena capaz de lidar com vulnerabilidades em imagens Docker e contêineres isolando ameaças do restante da rede em tempo de execução. Além disso, criaram um gerenciador de políticas baseado no OpenWhisk que permite o usuário adicionar regras para melhorar o sistema de isolamento de contêineres. Essas regras recebem informações do mecanismo de detecção de vulnerabilidades e acionam a API do Kubernetes para realizar ações de segurança.

Outra contribuição do trabalho refere-se a estender um serviço de verificação de vulnerabilidades para gerar eventos que ativam as regras do gerenciador de políticas para ajudar na detecção de ameaças. A abordagem proposta fornece varredura contínua de contêineres. Após uma ameaça ser detecta, ela é encaminhada para uma estrutura de isolamento que analisa relatórios de vulnerabilidade, e toma decisões com base em políticas definidas pelo usuário. Nenhum resultado foi apresentado para validar a arquitetura. Como trabalhos futuros, os autores pretendem utilizar inteligência artificial para gerar políticas de segurança de forma automatizada de acordo com o tipo de ameaça detectada.

Wu et al. [Wu et al. 2018] propuseram SLBot, um sistema *Serverless* baseado no *Service Flux*, um protocolo de comunicação com três canais de transmissão de dados que garante maiores níveis de disfarce da aplicação na rede. Neste trabalho os autores também discutem a viabilidade e eficácia do uso de serviços públicos na Web para a construir *botnets Serverless*. Testes com SLbot foram realizados comparando o SLbot a outras tecnologias similares. Os resultados obtidos demonstraram que o SLBot é mais eficiente que *botnets* tradicionais, sendo mais seguro. Como conclusão do artigo, os autores apresentaram técnicas para detectar e combater *botnets* construídas de forma semelhante ao SLBot.

5.5.6. Processsamento em SmartNICs

Todas as plataformas de computação em nuvem realizam processamento *Serverless* sobre servidores com CPU x86_x64 utilizando virtualização de funções Lambda de modo paralelo. CPUs x86_x64 foram projetadas para processar uma sequência de instruções de modo rápido. No entanto, são inadequadas para executar milhões de funções em paralelo. Cada função interrompe a CPU para armazenar estados em registradores ou memória, gerando atrasos de processamento e sobrecarga da memória devido à troca de contexto. Para contornar essas limitações, provedores de computação em nuvem com suporte *Serverless* estão adotando o uso de SmartNics na tentativa de melhorar a eficiência energética, redução de custos operacionais, e latência de *datacenters*.

Liu et al. [Liu et al. 2019] propuseram uma plataforma distribuída que permite executar microsserviços em *datacenters* usando SmartNics denominado E3. E3 processa microsserviços como processos com múltiplos threads sobre SmartNics. O projeto do E3 estende os componentes chave da plataforma *Azure Service Fabric* sobre SmartNics instaladas em servidores agrupados em *racks*. Cada *rack* contém um comutador ToR (*Top-of-Rack*), onde as SmartNics são conectadas. Para detectar a sobrecarga de processamento, o E3 usa técnicas como balanceamento de carga com roteamento de caminhos múltiplos com custo igual entre placa para o hospedeiro, posicionamento da localização onde o microsserviço será processado utilizando o reconhecimento da topologia da rede, e um orquestrador no plano de dados.

Sobre os componentes do E3, ele contém um controlador responsável por gerenciar o recurso do *cluster*, e aplicação *runtime* do microsserviço em cada hospedeiro e SmartNic. Cada *runtime* inclui um motor de execução, um agente orquestrador e um subsistema de comunicação. O E3 segue o modelo de programação baseado no fluxo de dados representado por um grafo acíclico direcionado (GAD). GAD é representado com microsserviços sendo os nós do grafo, e as arestas os canais de comunicação na direção

do fluxo RPC (*Remote procedure call*). Vários GADs podem ser criados e executados simultaneamente no E3. GAD descreve todos os caminhos do RPC e execução de um único microsserviço. O E3 foi implementado dentro de um *cluster* composto por servidores Xeon com até 4 SmartNics de 10 Gbps por servidor. Os resultados obtidos do E3 demonstram que *offloading* de microsserviços sobre SmartNics melhoram a eficiência energética em até 3x, e reduzem a latência em até 4x comparado com outras plataformas.

Choi et al. [Choi et al. 2019] apresentaram λ-NIC, um *framework Serverless* que processa milhões de funções Lambdas de modo iterativo em uma SmartNic composta por várias NPUs (*Network Processing Unit*). λ-NIC introduz uma nova abstração denominado casamento-Lambda que oculta a complexidade existente em SmartNics. Além disso, estende o modelo de máquina casamento-ação do P4 para melhorar a eficiência de códigos e executar funções Lambdas sobre SmartNics.

No λ-NIC, funções Lambdas são compiladas e executadas em tempo de execução com a chegada de pacotes na placa. Quando um pacote chega, o cabeçalho do pacote é analisado. Se ocorre o casamento do pacote com a função carregada no λ-NIC, uma ação é realizada. λ-NIC infere quais cabeçalhos do pacote são usados por cada função e gera automaticamente um analisador correspondente para os cabeçalhos, eliminando a necessidade de especificar lógica de processamento de pacotes manualmente para cada função. Usuários programam funções sobre no λ-NIC sem importar com detalhes em nível de hardware, por exemplo, λ-NIC analisa padrões de acesso à memória e endereços de acesso válido. O λ-NIC mapeia funções de modo otimizado nas memórias da SmartNic, garantindo que os acessos à memória sejam isolados. Funções são alocadas para processamento em uma única NPU sendo executada até ser concluída.

Por fim, λ-NIC emprega uma semântica de entrega pouco consistente com RDMA (*Remote Direct Memory Access*) processando solicitações diretamente nos núcleos da placa sem utilizar a CPU do hospedeiro. Os resultados obtidos mostram que λ-NIC alcança uma melhora de 736x a 880x na latência e vazão, reduzindo o uso de CPU e memória em relação a outros *frameworks Serverless* existentes.

Pacífico et al. [Pacífico et al. 2020] projetaram um sistema de processamento de pacotes *Serverless* com OpenFaaS orquestrando funções via contêineres e realizando *offloading* na plataforma NetFPGA SUME 10 G. Os principais componentes que compõem o sistema são a plataforma OpenFaaS, agente e processador de funções *Serverless*. Antes de executar uma função, o usuário precisa adicionar o código da função no OpenFaaS para enviar requisições de execução da função via interface gráfica ou linha de comando. Quando uma requisição chega no OpenFaaS, um contêiner é alocado para função. Após a função ter sido criada, uma requisição de execução com o tempo de processamento e função eBPF são enviados para o componente agente. O agente é responsável por escalonar a execução de funções de acordo com disponibilidade do processador. A computação da função termina com o usuário recebendo o resultado de processamento, liberando processador e contêiner.

Para melhorar o desempenho do sistema otimizações de hardware foram realizadas, como processador eBPF com *pipeline*, sistema de memória com *buffer* duplo, estrutura FIFO_MD para cópia zero de pacotes, e uso de mapas. Os autores avaliaram fatores do sistema como processamento *Serverless*, tempo de inatividade, vazão, latência

e energia. Quatro funções de rede foram implementadas: Wire, learning switch L2, roteador IPv4 e mitigação DDoS. Os resultados do sistema mostram que o sistema opera em taxa de linha de modo programável via espaço de usuário consumindo menos energia em relação a outros sistemas existentes.

5.6. Desafios e limitações

Essa seção é dedicada a apresentar e discutir os desafios das tecnologias *serverless* assim como suas limitações. Exemplos de aplicações, tópicos de pesquisa e aprendizados serão apresentados.

A computação *Serverless* obteve muito sucesso em sua aplicabilidade em diversas classes de cargas de trabalho como desenvolvimento de APIs e tratamento de eventos em tempo real como descrito por Jonas [Jonas et al. 2019].

No entanto, existem alguns tópicos em que as ofertas de computação *Serverless* são limitadas. Uma delas seria ser eficiente no processamento de dados. Outra seria que ela dificulta o progresso no desenvolvimento de sistemas distribuídos conforme apresentado por Hellerstein [Hellerstein et al. 2018].

Alguns exemplos de limitações ofertadas pela computação *Serverless* são apresentadas por Hellerstein [Hellerstein et al. 2018] e podem ser vistas a seguir:

- **Ciclo de vida limitado** No caso da AWS por exemplo, após 15 minutos, funções invocadas são desligadas pela infraestrutura *Lambda*.
- **Gargalos de Entrada/Saída (I/O)** A infraestrutura *AWS Lambda* conecta-se e depende de outros serviços da nuvem como dispositivos de armazenamento conectados com acesso via rede. Na prática, isso significa transferir dados entre equipamentos dentro dos centros de dados.
- **Comunicação entre dispositivos** Enquanto a *AWS Lambda* conecta-se via rede, elas não podem ser diretamente endereçáveis na rede enquanto estão em execução.
- **Ausência de computadores especializados** Atualmente as ofertas de computação *Serverless* são baseadas em seções de tempo de uso de uma CPU e alguma quantidade de memória primária. Não se utiliza computadores especializados no formato de computação *Serverless*

Jonas [Jonas et al. 2019] apresenta alguns exemplos práticos de aplicações limitadas pela computação *Serverless*:

- **Codificação de vídeo em tempo real** Serviços de armazenamento muito lentos para suportar comunicação com granularidade muito fina entre funções. Soluções paralelas porém diretas tem melhor desempenho neste tipo de tarefa.
- **MapReduce** O embaralhamento necessário na implementação de MapReduce não escala em função da latência na busca de objetos e por limites de velocidade nas operações de leitura/escrita por segundo.

- **Computar modelos de Álgebra Linear** Difícil de implementar comunicação difusa (*broadcast*) devido a limitação na latência de dispositivos de armazenamento. Além disso, provisionar um conjunto fixo de funções pode deixar recursos com sub-utilização.

- **Aprendizado de máquina** Ao construir encadeamento (*pipeline*) a falta de armazenamento de acesso veloz para armazenar parâmetros impacta no desempenho das tarefas de agregação de dados nesses tipos de algoritmos.

- **Bancos de dados** A falta de memória compartilhada impede o desenvolvimento de bancos de dados, pois o desempenho é baixo em função da latência para acessar serviços de armazenamento em blocos remotos.

Castro [Castro et al. 2019] afirma que: uma vez que a computação *Serverless* é uma área nova, existem diversas oportunidades de pesquisa a serem endereçadas e que representam desafios em Ciência da Computação.

O trabalho de Jonas [Jonas et al. 2019] descreve pesquisas desenvolvidos pela Universidade de *Berkeley* enfatizam: centros de dados, sistemas distribuídos, aprendizado de máquina e modelos de programação.

Como uma visão geral, são citados por Jonas [Jonas et al. 2019] cinco categorias dessas áreas que representam desafios:

- Abstrações;

- Sistemas;

- Redes;

- Segurança;

- Arquitetura.

Abaixo são listadas alguns exemplos de áreas de pesquisa que se enquadram dentro das categorias citadas:

- **Abstrações** Hoje as ofertas em computação *Serverless* permitem especificar a quantidade de memória e tempo de execução aos quais as funções irão fazer uso. No entanto, outros recursos não estão sob controle do desenvolvedor.

- **Dependência de dados** Atualmente as funções não tem conhecimento sobre a dependência de dados entre funções. Essa ignorância gera comunicações ineficientes por transações feitas com desempenho sub-ótimo.

- **Persistência** Existe uma carência em soluções para computação *Serverless* que implementem armazenamento persistente e efêmero por funções.

- **Tempo de inicialização** Existem três etapas na inicialização de funções: (1) agendamento e inicialização de ambiente da linguagem utilizada na função (*runtime*). (2) Descarregar (*Download*) o código, as bibliotecas e recursos necessário para executar a função. (3) Carregar bibliotecas e estruturas de dados em memória para que estejam disponível quando a função for executada.
- **Redes e sistemas distribuídos** Permitir que funções sejam colocadas em servidores específicos como por exemplo escolher a localidade por proximidade. Colocar funções em uma mesma máquina virtual para poder comunicar com mais eficiência.
- **Segurança** Agendamento aleatório para isolamento físico de funções. Além disso as funções deveriam ter acesso granular de configurações, chaves privadas, objetos de armazenamento e até dados temporários localmente.
- **Arquitetura de computadores** Equipamentos (*hardware*) heterogêneos, precificação e fácil administração para que usuários da computação *Serverless* tenham total controle do desempenho das funções.

Uma predição feita por Jonas [Jonas et al. 2019] é que a computação *Serverless* irá decolar e que a nuvem será híbrida. As aplicações irão diminuir em tamanho mesmo havendo alguns casos em que, por motivos de regulação e governança de dados, as aplicações precisarem usar legados.

Como mostrado por Ao [Ao et al. 2018], processamento de vídeo é um tópico emergente na utilização de computação *Serverless*. Aprendizado de máquina também caracteriza-se como fonte para pesquisa e inovação nesta área conforme apresentado discutidos por Ishakia [Ishakian et al. 2018] e Feng [Feng et al. 2018].

A computação *Serverless*, apesar de suas limitações, está cheia de desafios em Ciência da Computação para que pesquisadores possam investigar os problemas e propor soluções eficientes que possam contribuir com a ascensão de tecnologias *Serverless*; afirmação reforçada por Castro [Castro et al. 2019].

5.7. Prática

Nesta seção descrevemos os passos de instalação, criação de funções *Serverless*, suporte de uma nova linguagem, e executar exemplos de funções sobre a plataforma OpenFaaS. Nós escolhemos o OpenFaaS para a prática do minicurso devido às funcionalidades e características presentes nesta plataforma não serem encontradas em conjunto nas demais plataformas *Serverless* existentes e descritas neste minicurso.

5.7.1. Instalação

Para utilizar a plataforma pode-se escolher possuir uma instância da plataforma OpenFaaS instalada ou usar alguma plataforma disponível na nuvem.

5.7.1.1. Instalação de uma instância da plataforma OpenFaaS

Para a instalação da plataforma OpenFaaS, é necessário ter instalado as aplicações:

- Git: `$ sudo apt-get install git`
- Docker: `$ sudo apt-get install docker.io`

Para instalar O OpenFaaS é necessário ter instalado o Git. Também clonar o repositório para o computador que irá hospedar a plataforma. No terminal, execute o comando:

```
$ git clone https://github.com/openfaas/faas.git
```

Serão baixados cerca de 14,5 MB de arquivos para o computador. Após o *download* execute os comandos:

```
$ sudo su
$ cd faas
$ make
```

Com isso a plataforma será instalada, diversas imagens de contêineres serão baixadas e construídas. Esse processo pode levar vários minutos. Ao final desse processo a plataforma ainda não estará ativa. Ainda é necessário construir e publicar. Para construir use o *script* build via o comando:

```
$ ./build.sh
```

Nesse ponto serão construídos todas as dependências que a plataforma necessita para seu completo funcionamento. Ao final do processo a plataforma está apta a ser publicada através do comando:

```
$ docker swarm init --advertise-addr [interface de rede]
$ ./deploy_stack.sh
```

Após executar os comandos anteriores serão criadas as credenciais de usuário e senha para acessar o serviço OpenFaaS. Guarde as credenciais, elas serão necessárias mais tarde. Se tudo estiver correto a plataforma já está apta a ser usada para o envio de funções. É possível, nesse momento, acessar a plataforma por meio de um navegador, digite na barra de endereços 127.0.0.1 : 8080.

No primeiro acesso será necessário se autenticar fornecendo as credenciais geradas quando foi feito a publicação do OpenFaas. A figura 5.5 mostra a janela de autenticação. Preencha os campos de usuário e senha clique em *OK*.

5.7.1.2. Instalação da aplicação Cliente

Para utilizar os serviços de desenvolvedor da plataforma, é necessário ter instalado uma aplicação cliente do OpenFaaS, o *faas-cli*. Exitem três modos básicos de instalação via *script*, via *brew* e uma terceira forma para usuários do sistema operacional Windows. Aqui descreveremos a instalação via *script*. No terminal execute o comando:

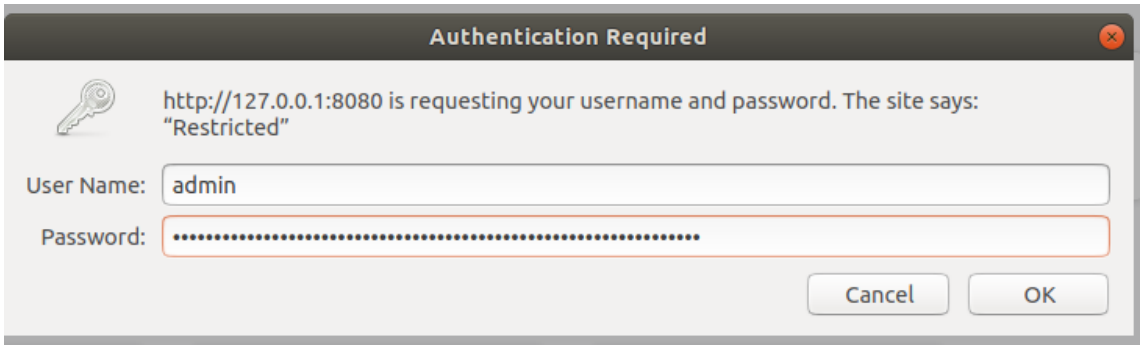

Figura 5.5. Exemplo da janela de autenticação da plataforma via navegador.

```
$ curl -sSL https://cli.openfaas.com | sudo sh
```

Caso não ocorra nenhum erro no processo de instalação a aplicação cliente do OpenFaas está pronta para divulgar funções em alguma plataforma OpenFaaS. o *gateway* por padrão é 127.0.0.1 : 8080. É realizar o login no *gateway* usando as credenciais obtidas no passo anterior:

```
faas-cli login [--username USERNAME] [--password PASSWORD]
```

Caso deseje fazer o *download* de todos os *templates* oficiais basta executar:

```
$ faas-cli template pull
$ ls template/
```

5.7.2. Implementação de funções

Com o *faas-cli* instalado para começar a escrever as funções basta, em um diretório arbitrário[5], digitar o seguinte comando:

```
$ faas-cli new --lang [linguagem] [nome da função]
```

Após esse comando serão criados alguns arquivos no diretório. Sem perda de generalidade é apresentado agora um exemplo de criação de função utilizando a linguagem Python. Execute o comando:

```
$ faas-cli new --lang python hello-minicurso
```

Serão criados 3 arquivos. Um no diretório raiz e dois em uma pasta com o nome que foi dado para a função

```
hello-minicurso/handler.py
hello-minicurso/requirements.txt
hello-minicurso.yml
```

[5]É recomendável um diretório exclusivo para isso.

A função de cada um dos arquivos será explicada posteriormente, ainda nessa seção. É necessário editar cada um dos três arquivos.

Primeiro edita-se o arquivo **handler.py**

Código 1 Conteúdo do arquivo handler.py.

```
1  def handle(nome):
2      print("Ola: " + nome)
```

O arquivo **handler.py** com o código 1 é a função propriamente dita e, neste caso, simplesmente imprime na tela "Olá xxx", sendo xxx o que é fornecido à função como parâmetro.

No Arquivo **hello-minicurso.yml** são definidas as opções tal qual a plataforma a ser usada, qual o *gateway*, e quais as funções serão executadas. O código 2 mostra o conteúdo do arquivo.

Código 2 Conteúdo do arquivo hello-minicurso.yml

```
1  provider:
2    name: faas
3    gateway: http://localhost:8080
4
5  functions:
6    hello-minicurso:
7      lang: python
8      handler: ./hello-minicurso
9      image: hello-minicurso
```

- *gateway* - Especifica o servidor no qual sua função irá rodar.
- *functions* - Este bloco define as funções. Há a possibilidade de mais de uma função por arquivo.
- *lang:* especifica em qual linguagem de programação a sua função foi escrita.
- *handler* especifica qual o caminho até sua função.
- *image* especifica qual o nome da imagem Docker.

No arquivo **requirements.txt**, específica qualquer dependência que seja necessária para executar a função. Neste exemple específico não será editado. Então pode-se construir a função com o comando:

```
$ faas-cli build -f ./hello-minicurso.yml
...

Successfully tagged hello-minicurso:latest
Image: hello-minicurso built
```

Com a imagem *Serverless* gerada pode-se publicar função:

```
$ faas-cli deploy -f ./hello-minicurso.yml
```

Caso a função seja publicada sem nenhum erro, há um retorno da plataforma com código 200 e o endereço para invocar a função:

```
200 OK
URL: http://localhost:8080/function/hello-minicurso
```

Neste ponto a função está publicada e pronta pra ser invocada.

5.7.2.1. Invocar funções

O OpenFaaS permite que usuários se conectem à plataforma invocando funções *Serverless* a partir de softwares cliente. Cliente é o componente do software responsável por interagir com o usuário através de interfaces ou linhas de comando. No OpenFaaS, clientes são utilizados para manipular funções. Para que a comunicação entre o software cliente e a plataforma ocorra, é necessário que o cliente tenha suporte ao protocolo HTTP.

Ao invocar uma função, o usuário pode fornecer parâmetros, por exemplo, uma sequência finita de números inteiros para uma função que calcule a média entre os valores fornecidos. Os parâmetros e o endereço da função solicitada são armazenados em uma requisição HTTP e enviados para o *API Gateway*, que redireciona o conteúdo para o componente *watchdog* da função correspondente. Os dados são processados e o resultado é enviado de volta para o software cliente. A seguir, apresentamos três clientes compatíveis com o OpenFaaS.

cURL (*client URL* ou URL): é uma ferramenta de linha de comando para transferência e recebimento de dados usando sintaxe URL. De acordo com o manual oficial [Stenberg 1997], cURL usa a biblioteca libcurl e suporta todos os protocolos que o libcurl suporta, como HTTP, FTP e LDAP. cURL pode ser utilizado em linha de comando ou integrado a *scripts* executados por outros programas como *Makefiles* ou *scripts* escritos em Python. cURL está disponível para a maioria das versões do Linux e pode ser instalado via gerenciador de pacotes da distribuição. No Ubuntu, o comando de instalação é:

```
$ sudo apt-get install curl
```

Para invocar uma função via cURL o usuário deve fornecer quatro parâmetros: (1) endereço IP do OpenFaaS, sendo o endereço padrão localhost ou 127.0.0.1; (2) porta do componente API Gateway, por padrão a porta 8080; (3) nome da função *Serverless* e (4) os parâmetros da função, se existirem. No último caso, a opção -d deve ser adicionada. A seguir, apresentamos a estrutura do comando cURL e invocamos a função *hello-minicurso*. O resultado desta função é a mensagem enviada pelo usuário como parâmetro via cURL.

```
$ curl <ip>:<porta>/function/<nome> -d <parâmetros>
$ curl localhost:8080/function/hello-minicurso -d "oi"
```

FaaS CLI: é a interface de linha de comando oficial do OpenFaaS. FaaS CLI possui vários comandos para gerenciar funções *Serverless*. A lista completa de opções pode ser obtida através do comando `$ faas-cli help`.

A opção que permite invocar funções é a `invoke`. Em comparação com cURL, `invoke` requer menos parâmetros, sendo necessário informar somente o nome da função desejada. A seguir, apresentamos um exemplo que invoca a função *hello-minicurso* com o comando `faas-cli`:

```
$ faas-cli invoke hello-minicurso
```

Ao executar o comando acima, uma mensagem de início de leitura será exibida no terminal. Após a mensagem, o FaaS CLI aguarda parâmetros de entrada que podem ser inseridos a qualquer momento pelo usuário. A função *Serverless* só será efetivamente invocada após o encerramento do processo de leitura dos parâmetros de entrada. A seguir, apresentamos um exemplo completo do processo de invocação de uma função com FaaS CLI. O parâmetro fornecido foi a frase bom dia"e o resultado obtido através da função *hello-minicurso* foi a mensagem "Ola: bom dia".

```
$ faas-cli invoke hello-minicurso
Reading from STDIN - hit (Control + D) to stop.
bom dia
Ola: bom dia
```

UI Portal ***(User Interface Portal)***: é o cliente web oficial do OpenFaaS. Ele apresenta uma lista de funções ativas e possui uma interface que permite inserir parâmetros e invocar funções. UI Portal pode ser acessado em qualquer navegador nos endereços `http://localhost:8080/` ou `http://127.0.0.1:8080/`. Quando uma função é selecionada, dois formulários (figuras 5.6 e 5.7) são exibidos simultaneamente no navegador. O primeiro formulário (figura 5.6) apresenta informações da função como status, total de chamadas realizadas, número de réplicas, imagem Docker e URL. Existe ainda um botão de exclusão que finaliza o processo de execução do contêiner e remove a função da lista de funções disponíveis.

O segundo formulário permite invocar a função selecionada. Os parâmetros de entrada são fornecidos através do campo de texto *request body*, ou corpo da requisição. O resultado é apresentado no campo *response body*, ou corpo da resposta. O usuário pode escolher vê-lo no formato de texto sem nenhuma formatação ou como JSON (*JavaScript Object Notation*, ou Notação de Objetos do JavaScript), um formato compacto de estruturação de dados como objetos. Existe ainda a opção de *download*, em que o conteúdo da resposta é armazenado em um arquivo que é baixado em seguida.

Além disso, informações referentes à execução da função, como o código de status da resposta e o tempo total de execução da função, são apresentados em campos próprios.

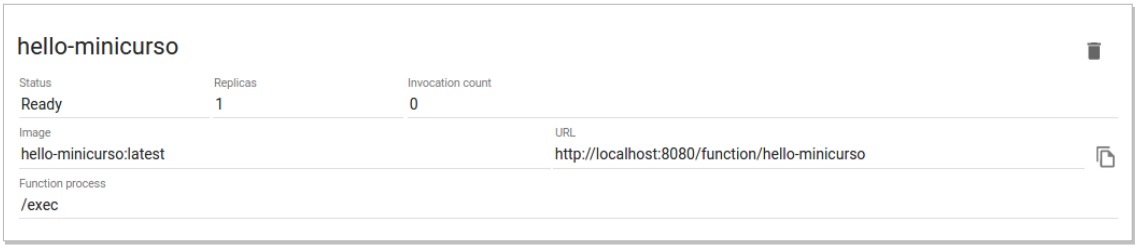

Figura 5.6. Dados da função apresentados pelo UI Portal.

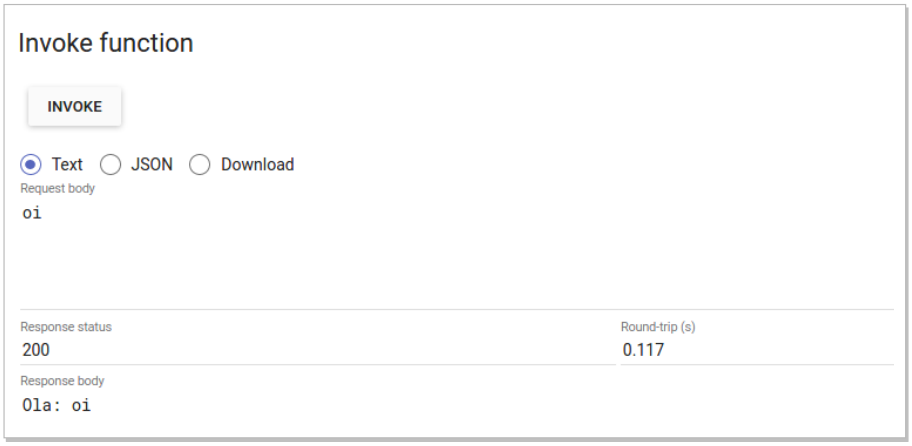

Figura 5.7. Ferramenta de execução de testes do UI Portal.

5.7.3. *Templates*

Templates são conjuntos de arquivos usados como modelo para criar novas funções compatíveis com OpenFaaS. *Templates* podem ser criados para suportar qualquer linguagem. OpenFaaS suporta onze *templates* oficiais que podem ser customizados de acordo com a necessidade do usuário. Um *template* é composto pelos seguintes arquivos:

- **Dockerfile:** Contém instruções específicas utilizadas pelo motor Docker para criar uma nova imagem;
- ***Index*:** Arquivo responsável por manipular a entrada e saída padrão. Este arquivo funciona como interface entre o componente *watchdog* e arquivo *handler* recebendo parâmetros de entrada, e retornando resultados de processamento;
- ***Handler*:** Arquivo que recebe parâmetros de entrada do *index*, inicializa o processamento, e retorna o resultado para interface;
- ***Template.yml***: Contém comandos de configuração de serviços disponíveis para função.

Os arquivos *index* e *handler* podem ser escritos na linguagem do *template*, e podem interagir com outros elementos do OpenFaaS, por exemplo, *Makefiles* e *scripts*. Algumas linguagens como Java, PHP e Python podem exigir um número maior de arquivos,

por exemplo, classes, *scripts* de configuração, e listas de dependências. Mas, de modo geral, todos os *templates* possuem a mesma estrutura básica apresentada nesta seção.

5.7.3.1. Inserindo um *template* no OpenFaaS

OpenFaaS possui um repositório denominado *Classic Templates* [Ellis 2017e] exclusivo para *templates* oficiais. A ferramenta FaaS CLI descrita na seção 5.7.2.1 acessa o repositório durante a criação de uma nova função em busca de um *template* que corresponde a linguagem solicitada. O repositório é clonado automaticamente se não for encontrado no diretório em que o comando `faas-cli new` for executado. No minicurso, criamos um novo *template* para a linguagem C. No entanto, esse procedimento pode ser realizado para qualquer linguagem de programação desde que dependências como compiladores e arquivos do programa da função sejam alterados devidamente.

O primeiro passo para inserir um *template* é criar um novo diretório de trabalho ou acessar um diretório existente. Se o conjunto de *templates* oficiais ainda não estiver disponível no diretório, o comando `faas-cli template pull` pode ser utilizado para obtê-lo. A seguir, apresentamos a sequência inicial de comandos.

```
$ mkdir -p ~/functions && cd ~/functions
$ faas-cli template pull
$ cd template && ls
$ mkdir c-language && cd c-language
```

O último comando cria e acessa a pasta para o *template* a ser customizado. Os arquivos *header.h* e *handler.h* são criados logo após a execução dos comandos apresentados no primeiro passo. Esses arquivos são responsáveis por processar os parâmetros de entrada e produzir os resultados. É interessante mantê-los em um diretório exclusivo dentro da pasta do *template*. Nós criamos uma pasta chamada *function* para armazenar esses arquivos.

```
$ mkdir function/ && cd function/
$ touch header.h handler.h
```

O arquivo *header* inclui bibliotecas básicas e define o cabeçalho da função *handler*. No código 3 apresentamos o escopo da função *handler*. Ela recebe como entrada uma sequência de caracteres e retorna um valor inteiro.

Código 3 Escopo da função *handler*.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int handler(char *entrada);
```

O código 4 representa o conteúdo da função *handler*. Dentro da função *handler* o usuário pode definir o comportamento da função *Serverless*, e também distribuir o processamento através de outras funções. Como exemplo ilustrativo, exibimos apenas uma mensagem concatenada com a *string* fornecida como parâmetro de entrada.

Código 4 Função *handler*.

```c
#include "header.h"

int handler(char *entrada) {
    printf("Oi! A mensagem recebida foi '%s'\n", entrada);
    return 0;
}
```

O próximo arquivo a ser criado é o *index.c*. Ele deve ser criado ao lado da pasta *function/*, isto é, em *~/functions/template/language-c/*.

```
$ cd ..
$ touch index.c
```

O arquivo *index* é responsável por receber parâmetros a partir da entrada padrão e enviá-los ao *handler*. No codigo 5 o contéudo do arquivo *index* é apresentado. A variável `TAM_MAX_DADOS` define o tamanho máximo para a sequência de caracteres de entrada. O arquivo pode ser alterado para converter a entrada em outros tipos de dados.

Código 5 Contéudo arquivo *index*.

```c
#include "header.h"

#define TAM_MAX_DADOS 16

int main() {
    char *dados;
    int tamanho;

    // Alocando a memória para o vetor de dados;
    dados = malloc(TAM_MAX_DADOS * sizeof(char));

    // Armazenando dados da entrada padrão;
    fgets(dados, TAM_MAX_DADOS, stdin);
    tamanho = strlen(dados);

    // Removendo '\n' do final do vetor;
    if (dados[tamanho - 1] == '\n')
       dados[--tamanho] = '\0';

    // Enviando dados recebidos para o handler;
    handler(dados);
    return 0;
}
```

Após todos os arquivos do *template* serem criados, o próximo passo é escrever os arquivos *Dockerfile* e *template.yml*, utilizados para configurar a imagem Docker e os serviços necessários para converter a função criada em um contêiner compatível com OpenFaaS. Os arquivos devem ser criados em *~/functions/template/language-c/*, ao lado da pasta *functions/* e do arquivo *index*.

```
$ touch Dockerfile template.yml
```

Dockerfile é um arquivo com sintaxe própria que descreve as etapas a serem executadas pelo Docker para criar uma nova imagem. Ele inclui comandos para instalar pacotes, criar arquivos e diretórios, executar arquivos, criar variáveis de ambiente, entre outros. A imagem gerada após a execução do *Dockerfile* é utilizada como modelo para a criação dos contêineres da função que serão implantados no OpenFaaS. No código 6 o conteúdo do arquivo é apresentado.

Código 6 Contéudo arquivo *Dockerfile*.

```
1  FROM openfaas/classic-watchdog:0.18.0 as watchdog
2
3  FROM alpine:latest
4
5  RUN apk add build-base
6
7  COPY --from=watchdog /fwatchdog /usr/bin/fwatchdog
8  RUN chmod +x /usr/bin/fwatchdog
9
10 WORKDIR /app/
11
12 COPY function/header.h      .
13 COPY function/handler.c     .
14 COPY index.c                .
15
16 RUN gcc header.h handler.c index.c -static -o /exec \
17 && chmod +x /exec
18
19 ENV fprocess="/exec"
20
21 HEALTHCHECK --interval=3s CMD [ -e /tmp/.lock ] || exit 1
22
23 CMD ["fwatchdog"]
```

De modo geral, as tarefas realizadas por cada comando no *Dockerfile* são descritos na ordem em que aparecem no código 6. Os comandos presentes nas linhas 1, 3, 7, 8, 10, 19, 21 e 23 são comuns à maioria dos *templates*. Os comandos das linhas 5, 12, 13, 14, 16 e 17 podem variar de acordo com a linguagem. Além disso, é possível adicionar instruções para realizar *download* e instalação de dependências, execução de *scripts* e outros, conforme a necessidade do usuário. O repositório oficial de *templates* do OpenFaaS [Ellis 2017e] possui vários exemplos que podem ser utilizados como exemplo e material de estudo.

1. **Linha 1:** Importa a imagem do componente *watchdog*;

2. **Linha 3:** Importa a imagem do *Linux Alpine*. Alpine é utilizado como ambiente de execução devido ao tamanho mínimo de sua estrutura base (5 MB excluindo o *kernel* e outras dependências);

3. **Linha 5:** Instala o pacote *build-base* que contém o compilador GCC 5.3.0 e outros pacotes essenciais como *binutils* e *libc-dev*;

4. **Linhas 7 e 8:** Copiam o conteúdo do componente *watchdog* da imagem oficial para nova imagem e concedem permissão para execução;

5. **Linha 10:** Define o diretório */app/* como ambiente de trabalho do contêiner. Comandos como `COPY`, `RUN` e `CMD` passam a ser executados nesse diretório;

6. **Linhas 12, 13 e 14** Copiam os arquivos do *template* para o diretório de trabalho definido na linha 10;

7. **Linhas 16 e 17:** Compilam os arquivos obtidos com o compilador GCC e geram o executável do arquivo *index.c*;

8. **Linha 19:** Define o executável criado como processo principal do contêiner. Este comando é obrigatório. Sem ele, dados enviados ao contêiner não serão recebidos pelo arquivo *index*;

9. **Linha 21:** Define em 3 segundos o intervalo de atuação do mecanismo que impede que o contêiner em uso seja alocado por outro usuário;

10. **Linha 23:** Inicializa o *watchdog* que passará a receber solicitações da aplicação cliente via API *Gateway*.

O último arquivo a ser configurado é o *template.yml*. Este arquivo é responsável por informar aos serviços do OpenFaaS qual é o principal arquivo executável da função. Ele serve também para descrever rotinas de serviços específicos implementados pelo usuário. O conteúdo do arquivo *template.yml* é apresentado no código 7.

Código 7 Contéudo arquivo *template.yml*.

```
language: c
fprocess: /exec

welcome_message:
    Parabéns! Você criou uma função serverless em C!
    Não se esqueça de atualizar o Dockerfile ao adicionar
    novos arquivos.
```

Após a sequência de passos realizados, todos os arquivos necessários para suportar uma nova linguagem foram configurados no OpenFaaS.

5.7.4. Criando funções eBPF no OpenFaaS

Nesta seção discutimos como adicionar suporte eBPF no OpenFaaS e apresentamos exemplos práticos. eBPF é uma tecnologia utilizada no processamento de pacotes que provê programabilidade no plano de dados. OpenFaaS pode ser utilizado como um sistema de processamento de pacotes *Serverless* totalmente programável e escalável usando funções *Serverless* eBPF.

5.7.4.1. eBPF (*extended Berkeley Packet Filter*)

É uma máquina virtual com arquitetura RISC de 64 bits de propósito geral. Foi inserida no *kernel* do Linux a partir da versão 3.15. eBPF realiza processamento rápido de pacotes de forma independente de protocolos e em tempo de execução dentro do *kernel*, provendo programabilidade na computação de pacotes. Programas escritos em eBPF são compilados em *bytecode* eBPF antes de serem executados no *kernel*. Algumas linguagens como P4 e C já suportam essa tecnologia. O minicurso intitulado *Processamento Rápido de Pacotes com eBPF e XDP* [Vieira et al. 2019] e o tutorial *Fast packet processing with eBPF and XDP* [Vieira et al. 2020] apresentam mais informações sobre essa tecnologia.

5.7.4.2. Adicionando suporte uBPF no *template* C

Nessa seção vamos utilizar o uBPF [Iovisor 2020] que nada mais é do que uma implementação da maquina virtual eBPF em espaço de usuário.

A configuração apresentada na seção 5.7.3.1 não é suficiente para compilar e executar códigos em C uBPF. Para isso, é necessário instalar pacotes de dependências e compilador Clang, além de gerar a maquina virtual. As instalação desses pacotes pode ser feito alterando os arquivos do *template* da linguagem C apresentado na seção 5.7.3.1. Outra alternativa pode ser copiar o conteúdo da pasta *c-language* para criar um *template* exclusivo para C uBPF, mantendo o *template* da linguagem C intacto como apresentado abaixo.

```
$ cd ~/functions/template
$ cp -r c-language c-ebpf && cd c-ebpf
```

Precisamos alterar nosso *Dockerfile* conforme pode ser visto no código 8. As linhas 5 a 8 instalam as dependências, as linhas 10 a 12 baixam o código do uBPF do repositório oficial e compilam a maquina virtual ubpf de teste, a linha 16 compila o código uBPF escrito pelo usuário no arquivo handle.c e, por fim a linha 26 executa o código gerado na maquina virtual. Diferente do *template c-language* do exemplo anterior, neste *template* usaremos apenas o arquivo handle.c que deve ser alterado conforme o código 9.

5.7.4.3. Exemplo usando uma função no contêiner uBPF

Nesta seção apresentamos um exemplo trivial, usando o *template* gerado na seção 5.7.4.2.

Código 8 Contéudo do arquivo *Dockerfile* com suporte uBPF.

```
FROM openfaas/classic-watchdog:0.18.0 as watchdog

FROM alpine:latest

RUN apk add build-base \
      clang \
          git \
          llvm
WORKDIR /home/app
RUN git clone https://github.com/iovisor/ubpf.git
WORKDIR ubpf
RUN make -C vm

COPY function/handler.c .

RUN  clang -O2 -emit-llvm -c handler.c -o - | \
llc -march=bpf -filetype=obj -o handler.o

# Add non root user
# Create a non-root user
RUN addgroup --system app \
    && adduser --system --ingroup app app

RUN chown app:app -R /home/app
USER app

ENV fprocess="/home/app/ubpf/vm/test /home/app/ubpf/handler.o"

# Set to true to see request in function logs
ENV write_debug="false"

EXPOSE 8080

HEALTHCHECK --interval=3s CMD [ -e /tmp/.lock ] || exit 1

CMD ["fwatchdog"]
```

Código 9 Contéudo do arquivo *handler.c* com suporte uBPF.

```
#include "vm/inc/ubpf.h"

int  main() {

}
```

Primeiro instanciamos uma função baseada no *template*.

```
$ faas-cli new --lang c-ebpf hello_ebpf
$ cd hello_ebpf/function/
```

Adicionamos o código que queremos executar. Neste exemplo vamos apenas elevar uma constante ao quadrado, bastando para isso editar o arquivo handler.c conforme o código 10.

Código 10 Conteúdo do arquivo *handler.c*.

```
#include "vm/inc/ubpf.h"

int  main() {
    int a;
    a=2;
    return a*a;
}
```

Podemos gerar a imagem com o comando:

```
$ faas-cli build -f ./hello-ebpf.yml
```

Com a imagem *Serverless* gerada, pode-se publicar função:

```
$ faas-cli deploy -f ./hello-ebpf.yml
```

Em seguida, evocar a função e receber o resultado esperado.

```
$ curl http://127.0.0.1:8080/function/hello-ebpf
$ 4
```

Apesar de simples, o exemplo demonstra como, com essa ferramenta em mãos, se pode rapidamente gerar e disponibilizar funções ebpf.

5.8. Conclusões e Discussões Finais

Este minicurso apresentou o estado da arte em Computação *Serverless*. Foram discutidos os conceitos, as limitações, os casos de uso, as áreas de pesquisa em aberto e as plataformas disponíveis no mercado.

Exemplos práticos de implementação de funções *Serverless* utilizando as principais ferramentas de código aberto e algumas plataformas de provedores de serviços em nuvem foram demonstradas de modo a aprofundar o estudo do ecossistema da computação *Serverless*.

Discutiu-se também, o futuro desse emergente modelo de computação, que apesar de permitir implementações distribuídas, tem limitações claras relacionadas à latência na

troca de mensagens, persistência e compartilhamento de dados e a falta de conhecimento global de modo a permitir algoritmos mais eficientes.

Tudo isso abre oportunidades de pesquisa que remetem a tópicos e problemas clássicos em Ciência da Computação como otimizações em Sistemas Operacionais, Sistemas Distribuídos, Localidade de dados em Redes, Carregamento eficiente de programas e segurança.

Este minicurso contribui trazendo uma visão abrangente sobre a computação *Serverless* de modo que o leitor possa compreender o estado da arte e para onde caminha essa tecnologia. Além disso, contribui com um arcabouço rico em exemplos de implementação nas principais ferramentas de código aberto e do mercado facilitando assim o aprendizado e ambientação com essa emergente área de conhecimento e pesquisa.

Agradecimentos

Agradecemos as agências de pesquisa CNPq, FAPESP projeto 2018/23085-5 e FAPEMIG pelo apoio financeiro. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

Referências

[Alpine Linux 2018] Alpine Linux (2018). `https://alpinelinux.org/about/`. Acessado em: 01/03/2020.

[Ao et al. 2018] Ao, L., Izhikevich, L., Voelker, G. M., and Porter, G. (2018). Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, New York, NY, USA. Association for Computing Machinery.

[Apache 2004] Apache (2004). Apache license. `https://www.apache.org/licenses/LICENSE-2.0`. Acessado em: 20/03/2020.

[AWS Lambda 2014] AWS Lambda (2014). Aws lambda. `https://aws.amazon.com/lambda/`. Accessado em: 29/09/2019.

[Azure Functions 2016] Azure Functions (2016). Azure functions. `https://azure.microsoft.com/en-us/services/functions/`. Accessado em: 29/09/2019.

[Bila et al. 2017] Bila, N., Dettori, P., Kanso, A., Watanabe, Y., and Youssef, A. (2017). Leveraging the serverless architecture for securing linux containers. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 401–404. IEEE.

[Castro et al. 2019] Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. (2019). The rise of serverless computing. *Commun. ACM*, 62(12):44–54.

[Choi et al. 2019] Choi, S., Shahbaz, M., Prabhakar, B., and Rosenblum, M. (2019). λ-nic: Interactive serverless compute on programmable smartnics. *arXiv preprint arXiv:1909.11958*.

[Church et al. 2020] Church, M., Ruiz, M., Seifert, A., and Marshall, T. (2020). Docker - docker swarm reference architecture: Exploring scalable, portable docker container networks.

[CNCF landscape 2020] CNCF landscape (2020). Cncf serverless landscape. `https://landscape.cncf.io/format=serverless`. Accessado em: 08/03/2020.

[CouchDB 2005] CouchDB (2005). Couchdb. `https://couchdb.apache.org/`. Acessado em: 07/03/2020.

[Docker Inc. 2008] Docker Inc. (2008). Docker. `https://www.docker.com/`. Acessado em: 07/03/2020.

[Docker Inc. 2014] Docker Inc. (2014). Swarm mode overview. `https://docs.docker.com/engine/swarm/`. Acessado em: 07/02/2020.

[Ellis 2016] Ellis, A. (2016). `https://github.com/openfaas/faas/`. Acessado em: 29/02/2020.

[Ellis 2017a] Ellis, A. (2017a). `https://github.com/openfaas/faas-netes/`. Acessado em: 01/03/2020.

[Ellis 2017b] Ellis, A. (2017b). `https://github.com/openfaas/faas-cloud/`. Acessado em: 01/03/2020.

[Ellis 2017c] Ellis, A. (2017c). `https://github.com/openfaas/nats-queue-worker`. Acessado em: 03/03/2020.

[Ellis 2017d] Ellis, A. (2017d). `https://github.com/openfaas-incubator/kafka-connector`. Acessado em: 03/03/2020.

[Ellis 2017e] Ellis, A. (2017e). `https://github.com/openfaas/templates/`. Acessado em: 17/02/2020.

[Ellis 2017f] Ellis, A. (2017f). Introducing functions as a service (openfaas). `https://blog.alexellis.io/introducing-functions-as-a-service/`. Acessado em: 01/03/2020.

[Ellis 2017g] Ellis, A. (2017g). Openfaas: Serverless functions made simple for docker and kubernates. `https://www.openfaas.com/`. Acessado em: 29/09/2019.

[Feng et al. 2018] Feng, L., Kudva, P., Da Silva, D., and Hu, J. (2018). Exploring serverless computing for neural network training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 334–341. IEEE.

[Flexera 2019] Flexera (2019). Cloud computing trends: 2019 state of the cloud survey. https://www.flexera.com/blog/cloud/2019/02/cloud-computing-trends-2019-state-of-the-cloud-survey/. Acessado em: 19/11/2019.

[Fouladi et al. 2017] Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., and Winstein, K. (2017). Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 363–376.

[Glikson et al. 2017] Glikson, A., Nastic, S., and Dustdar, S. (2017). Deviceless edge computing: Extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, New York, NY, USA. Association for Computing Machinery.

[Google Cloud Functions 2016] Google Cloud Functions (2016). Google cloud functions. `https://cloud.google.com/functions/`. Accessado em: 29/09/2019.

[Google Inc. 2015] Google Inc. (2015). Production-grade container orchestration. `https://kubernetes.io/`. Acessado em: 07/02/2020.

[Grafana Labs 2014] Grafana Labs (2014). The analytics platform for all your metrics. `https://grafana.com/grafana/`. Acessado em: 13/02/2020.

[Hellerstein et al. 2018] Hellerstein, J. M., Faleiro, J., Gonzalez, J. E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., and Wu, C. (2018). Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*.

[Hendrickson et al. 2016] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2016). Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 33–39, Berkeley, CA, USA. USENIX Association.

[Igor Sysoev 2005] Igor Sysoev (2005). Nginx. `https://www.nginx.com/`. Acessado em: 07/03/2020.

[Iovisor 2020] Iovisor (2020). iovisor/ubpf.

[Ishakian et al. 2018] Ishakian, V., Muthusamy, V., and Slominski, A. (2018). Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262. IEEE.

[Jonas et al. 2019] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Menezes Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J., Popa, R. A., Stoica, I., and Patterson, D. A. (2019). Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley.

[Kafka 2011] Kafka (2011). Kafka. `https://kafka.apache.org/`. Acessado em: 07/03/2020.

[Khan et al. 2019] Khan, W. Z., Ahmed, E., Hakak, S., Yaqoob, I., and Ahmed, A. (2019). Edge computing: A survey. *Future Generation Computer Systems*, 97:219 – 235.

[Linux Foundation 2016] Linux Foundation (2016). What is prometheus. https://prometheus.io/docs/introduction/overview/. Acessado em: 01/03/2020.

[Liu et al. 2019] Liu, M., Peter, S., Krishnamurthy, A., and Phothilimthana, P. M. (2019). E3: energy-efficient microservices on smartnic-accelerated servers. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 363–378.

[MIT 1980] MIT (1980). Mit license. https://en.wikipedia.org/wiki/MIT_License. Acessado em: 02/03/2020.

[OpenWhisk 2016] OpenWhisk, I. (2016). IBM OpenWhisk Project. http://developer.ibm.com/openwhisk/. Acessado em: 08/03/2020".

[Pacífico et al. 2020] Pacífico, R. D. G., Duarte, L. F. S., Nacif, J. A. M., and Vieira, M. A. M. (2020). Sistema de processamento de pacotes *Serverless*. In *XXXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos (SBRC)*, Rio de Janeiro, RJ, Brasil. SBC.

[Quevedo et al. 2019] Quevedo, S., Merchán, F., Rivadeneira, R., and Dominguez, F. X. (2019). Evaluating apache openwhisk-faas. In *2019 IEEE Fourth Ecuador Technical Chapters Meeting (ETCM)*, pages 1–5. IEEE.

[Shafiei et al. 2020] Shafiei, H., Khonsari, A., and Mousavi, P. (2020). Serverless computing: A survey of opportunities, challenges and applications.

[Shankar et al. 2018] Shankar, V., Krauth, K., Pu, Q., Jonas, E., Venkataraman, S., Stoica, I., Recht, B., and Ragan-Kelley, J. (2018). Numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679*.

[Stenberg 1997] Stenberg, D. (1997). Curl: The man page. https://curl.haxx.se/docs/manpage.html. Acessado em: 06/03/2020.

[Vieira et al. 2019] Vieira, M. A. M., Castanho, M. S., Pacífico, R. D. G., Santos, E. R. S., Câmara Júnior, E. P. M., and Vieira, L. F. M. (2019). Processamento Rápido de Pacotes com eBPF e XDP. In *Minicurso do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos (SBRC)*, Porto Alegre, RS, Brasil. SBC.

[Vieira et al. 2020] Vieira, M. A. M., Castanho, M. S., Pacífico, R. D. G., Santos, E. R. S., Júnior, E. P. M. C., and Vieira, L. F. M. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1).

[Vilalta et al. 2017] Vilalta, R., López, V., Giorgetti, A., Peng, S., Orsini, V., Velasco, L., Serral-Gracia, R., Morris, D., De Fina, S., Cugini, F., et al. (2017). Telcofog: A unified flexible fog and cloud computing architecture for 5g networks. *IEEE Communications Magazine*, 55(8):36–43.

[Wang et al. 2018] Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. (2018). Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA. USENIX Association.

[Werner et al. 2018] Werner, S., Kuhlenkamp, J., Klems, M., Müller, J., and Tai, S. (2018). Serverless big data processing using matrix multiplication as example. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 358–365. IEEE.

[Wu et al. 2018] Wu, D., Fang, B., Yin, J., Zhang, F., and Cui, X. (2018). Slbot: A serverless botnet based on service flux. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, pages 181–188. IEEE.

[Xiong et al. 2018] Xiong, Y., Sun, Y., Xing, L., and Huang, Y. (2018). Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377. IEEE.

[Zhang et al. 2019] Zhang, M., Zhu, Y., Zhang, C., and Liu, J. (2019). Video processing with serverless computing: A measurement study. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '19, page 61–66, New York, NY, USA. Association for Computing Machinery.