Pedro Felipe Santos Magalhães

Desenvolvimento de aplicações que usam o modelo Serverless

RELATÓRIO DE PROJETO FINAL

DEPARTAMENTO DE INFORMÁTICA
Programa de graduação em Ciência da Computação

Rio de Janeiro
Dezembro de 2020

Pontifícia Universidade Católica
do Rio de Janeiro

Pedro Felipe Santos Magalhães

Desenvolvimento de aplicações que usam o modelo Serverless

Relatório de Projeto Final

Relatório de Projeto Final, apresentado ao programa Ciência da Computaçãoda PUC-Rio como requisito parcial para a obtenção do titulo de Engenheiro de Computação.

Orientador : Noemi de La Rocque Rodriguez
Co-orientador: Maria Julia de Lima

Rio de Janeiro
Dezembro de 2020

Sumário

1
Introdução

Tradicionalmente, o desenvolvimento de uma aplicação para a web envolve o desenvolvimento de dois componentes principais, a aplicação web(*frontend*) e os serviços web(*backend*) (1). O *frontend* executa no cliente enquanto o *backend* executa em um ou mais servidores gerenciados pelo desenvolvedor da aplicação.

Um dos principais desafios desta abordagem é dimensionar os servidores para atender todas as requisições dos clientes. Um superdimensionamento dos servidores gera um gasto muito alto, enquanto o subdimensionamento limita o número de clientes que podem ser atendidos simultaneamente, aumentando assim a latência das respostas (2).

Além disso, a instalação, configuração e manutenção do ambiente físico onde os servidores vão rodar não é uma tarefa trivial, exigindo grande esforço de configuração e manutenção do desenvolvedor ou equipe de devolvimento para que esse ambiente esteja disponível. Muitas empresas tem equipes dedicadas à essa função.

O modelo de programação *Serverless* utiliza uma arquitetura em que o *backend* de uma aplicação não é administrado diretamente pelo desenvolvedor. Esse modelo vem ganhando força (3) com o investimento pesado nos últimos anos das principais empresas de tecnologia de nuvem como IBM, Amazon, Google e Microsoft. A principal vantagem dessa abordagem de programação é que o desenvolvedor passa a não se preocupar com a infraestrutura de um servidor local, podendo implementar apenas a lógica da aplicação que vai rodar em um servidor externo.

Existem diferentes modelos de programação que se encaixam na categoria *serverless* (4). Estudaremos aqui o modelo chamado de *Function as a Service* ou *FaaS*, onde o programador desenvolve pequenas funções independentes que executam sob demanda na plataforma *serverless* utilizada.

Neste projeto estudamos como o desenvolvimento de aplicações com o modelo *serverless* impacta o desenvolvimento da aplicação sob o ponto de vista do desenvolvedor. Desenvolvemos aplicações com características distintas utilizando a plataforma do Google e uma plataforma que roda no *Kubernetes*[1].

[1]Kubernetes é um produto *open source* utilizado para automatizar a implantação, o

Contrastamos o desenvolvimento dessas aplicações no modelo *serverless* com a nossa experiência com o modelo tradicional, buscando também saber em quais cenários o modelo traz vantagens e em quais apresenta desvantagens ao desenvolvedor.

dimensionamento e o gerenciamento de aplicativos em contêiner.

2
O modelo Serverless

O modelo *Serverless* é bem recente. O termo surgiu por volta de 2010. Em artigo de 2012 (5), Ken Fromm esclarece que apesar do termo dar a ideia de que o servidor deixaria de existir, ele na verdade apenas indica que o servidor não seria mais gerenciado pelo desenvolvedor da aplicação. Ou seja, os servidores continuam existindo, mas são gerenciados por terceiros.

Hoje empresas como Amazon, Google, Microsoft e IBM oferecem algum tipo de serviço para implementação de ambientes *serverless*. A primeira delas a oferecer esse serviço foi a Amazon, com o "AWS Lambda" lançado em 2014, e as outras três lançaram suas implementações em 2016. Em geral, o pagamento para o uso dessas plataformas varia de acordo com o número de requisições recebidas e o tempo de computação utilizado. Atualmente, todas elas oferecem uma versão grátis do serviço.

Além dos serviços em nuvem, existem implementações *open source* que são destinadas a rodar em um ambiente local, geralmente em *clusters on-premise*. Como exemplos de implementação *open source* temos o Apache OpenWhisk[1], IronFunctions[2], *Kubeless*[3], OpenFaas[4], OpenLambda[5].

Em termos gerais, todas essas plataformas permitem que o desenvolvedor de aplicações não precise se preocupar com o provisionamento e a configuração dos servidores dedicando-se exclusivamente ao desenvolvimento da lógica da aplicação. Para desenvolver os serviços que atendem as requisições da aplicação, o desenvolvedor constrói funções *stateless* cadastrando-as na plataforma que fica responsável por ativá-las quando ocorre algum evento.

Inicialmente o termo *serverless* era usado para descrever aplicações que utilizavam grande parte ou todo o seu *backend* hospedado em uma plataforma de nuvem. Usualmente essas plataformas oferecem diversos serviços(4) que são de uso comum a muitas aplicações, como autenticação de usuário, administração de banco de dados, armazenamento de dados em nuvem ou *push notification*, dentre outros serviços específicos de cada empresa que provê a

[1] https://openwhisk.apache.org/
[2] https://open.iron.io/
[3] https://kubeless.io/
[4] https://www.openfaas.com/
[5] https://github.com/open-lambda/open-lambda

plataforma de nuvem. Esse modelo ficou conhecido como *backend as a service* (BaaS).

Mais recentemente o termo *Serverless* passou a se referir a aplicações em que a lógica do servidor é escrita pelo desenvolvedor em pequenas funções sem estado, que, em geral, seguem uma arquitetura de microsserviços(6), e ficam hospedadas em uma plataforma na nuvem ou *on-premise*. Como cada função não guarda estado e tipicamente roda em um *container* (7), a tarefa de tratar a elasticidade dessas funções fica mais simples e as plataformas conseguem instanciar zero ou mais instâncias da função de acordo com a demanda. Assim, as funções somente consomem recursos e são instanciadas quando são ativadas por algum evento, permitindo que as plataformas de nuvem cobrem do desenvolvedor apenas o tempo em que as funções ficam efetivamente executando. Esse modelo também é conhecido como *function as a service* (FaaS). Em geral, as plataformas de nuvem oferecem tanto o *BaaS* quanto *FaaS* em seu portfólio *serverless*, permitindo que os modelos sejam usados em conjunto.

Vale destacar que, apesar de em geral as aplicações desenvolvidas no modelo *FaaS* utilizarem a arquitetura de microsserviços, um não é sinônimo do outro. Em um microsserviço temos uma fronteira bem definida que executa de forma independente dos outros componentes da aplicação. Já uma função no modelo *FaaS* é um pedaço de código relativamente pequeno que executa apenas uma ação em resposta a um evento. Aplicações baseadas em microsserviços podem utilizar *function as a service* como parte de sua implementação mas, em geral, dependem de outros funcionalidades e requisitos que o modelo *function as a service* não oferece. Isso porque existem limitações impostas por cada plataforma, como por exemplo, na *AWS lambda* o tempo máximo de execução de uma função é de 15 minutos e o tamanho máximo do código, incluindo dependências, é de 50mb.

Cada função deve fornecer o código com um ponto de entrada, suas dependências e o tipo de evento que vai iniciar sua execução. Cada plataforma oferece tipos diferentes de eventos que podem iniciar a execução de uma função, mas geralmente o evento é uma chamada http, uma mensagem em um serviço de mensageria ou um evento baseado em tempo, quando queremos, por exemplo, que uma função rode uma vez por dia.

As plataformas *serverless* ficam responsáveis por questões como a escalabilidade e elasticidade, disparando novas instâncias das funções de acordo com a demanda. O uso mais eficiente dos recursos é um dos principais benefícios deste modelo de programação, uma vez que os serviços do *backend* são efêmeros, ou seja, só ficam ativos durante uma execução, após serem disparados por

um evento.

O modelo *serverless* oferecido pelas plataformas de nuvem oferece alguns benefícios. Um deles ocorre pelo fato do desenvolvedor, ou equipe, não precisar gerenciar diretamente o servidor, deixando para a plataforma o trabalho, por exemplo, de garantir a disponibilidade, gerar backups ou garantir redundância. Outro benefício, é a redução na latência da comunicação entre a aplicação cliente e o servidor para clientes afastados do local físico da empresa, uma vez que, em geral, as plataformas de nuvem possuem servidores espalhados pelo mundo. Em geral cada uma dessas plataformas proprietárias possuem serviços extras que estão disponíveis apenas na própria plataforma. Isso facilita o desenvolvimento da aplicação, mas tem o efeito de *Lock-in* (8) em que, com o passar do tempo, fica cada vez mais custoso migrar a aplicação para uma outra plataforma *serverless*. Esse é um ponto fundamental na hora de avaliar qual plataforma utilizar.

Nas plataformas *open source* temos um maior controle sobre o funcionamento interno do modelo. Podemos adaptar e incrementar as funcionalidades de acordo com demandas específicas. Por exemplo, podemos criar eventos customizados para o acionamento de funções e adicionar ambientes de execução para outras linguagens. No entanto, passamos a ter um trabalho grande com infraestrutura para configurar e manter a plataforma.

Apesar de ser um modelo de programação recente, alguns estudos tem sido feitos sobre o tema. Para a realização deste trabalho, por exemplo, estudamos alguns artigos que tratam especificamente sobre *serverless*.

Inicialmente fizemos a leitura do artigo “Serverless: What it is, what to do and what not to do"(9) que traz um pequena introdução sobre o modelo, e explora os pontos que funcionam bem para esse ambiente e o que não é ideal, citando também alguns problemas em aberto.

Em seguida, lemos o artigo de pesquisadores da Berkeley intitulado “Cloud programming simplified: A Berkeley view on serverless computing"(10), que faz uma introdução sobre o modelo, avalia suas limitações e faz previsões sobre o que o *serverless* deve se tornar.

Outro artigo também da Berkeley, mas que mostra uma visão mais crítica dos pesquisadores sobre o modelo *serverless* e tenta indicar quais os principiais problemas que deveriam ser corrigidos é o “Serverless computing:One step forward, two steps back."(11).

```
kafkaProducer.send([{
        topic: 'new-article-topic',
        messages: JSON.stringify(article),
        partition: 0
}], (err, data) => {
    if (err) {
        reject(err);
    } else {
        resolve(article);
    }
});
```

```
const dataBuffer = Buffer.from(JSON.stringify(article));
topic.publish(dataBuffer)
    .then(messageId => {
        console.log(`message ${messageId} published`);
        res.status(201).send(article);
    })
    .catch(err => {
        console.log('ERROR', err);
        res.status(400).send(err);
    });
```

Figura 2.1: Função para publicar mensagem no serviço de mensageria,a esquerda o código do *Kubeless* e a direita o do Google

2.1 Lock-in

O termo *lock-in*, no desenvolvimento de software, se refere a situação em que um desenvolvedor fica dependente de uma ferramenta ou ambiente porque a troca dessa ferramenta ou ambiente teria um custo muito alto. Esse efeito de *lock-in*, pode ser intencional por parte dos responsáveis pelo desenvolvimento das ferramentas ou simplesmente o resultado de uma ausência de padronização entre as ferramentas. No caso do *serverless*, esse *lock-in* parece ocorrer por ser uma tecnologia recente e em rápida evolução, onde cada plataforma tenta oferecer serviços diferenciados para atrair mais desenvolvedores.

Durante o desenvolvimento das aplicações verificamos que para cada plataforma temos que fazer diversas adaptações ao código das funções. Por exemplo, a interface de publicação de mensagens no serviço de mensageria é diferente em cada uma das plataformas, como podemos ver na figura 2.1.

Apesar desses problemas serem mitigados se criarmos uma boa arquitetura de software, ainda pode ser muito custoso portar uma aplicação de um ambiente para o outro. Isso porque, em geral, uma aplicação nesse modelo é composta por várias funções *serverless* e, além de recriar e configurar cada uma das funções, temos que no mínimo alterar os parâmetros recebidos como entrada das funções, pois em cada plataforma recebemos objetos distintos com diferentes atributos, como vemos na figura 2.2.

Além do *lock-in* causado por diferença no código das funções, muitas vezes os desenvolvedores sofrem esse efeito por dependerem de outros serviços integrados apenas em determinados ambientes. Por exemplo, o serviço de armazenamento pode servir de gatilho para acionar as funções do desenvolvedor na AWS e no Google, mas no *Kubeless* para obter a mesma funcionalidade o desenvolvedor teria que implementar manualmente esse gatilho.

Figura 2.2: Hello World no *Google Functions*(esquerda) e na AWS Lambda(direita) .

3
Proposta e objetivos do trabalho

Nesse trabalho, buscamos entender melhor as vantagens e limitações do modelo *serverless*. Para essa finalidade, avaliamos duas plataformas *serverless*: (i) o *Google Functions* oferecido pela plataforma de nuvem *Google Cloud*, (ii) O *Kubeless* como uma solução *open source* rodando em um *cluster Kubernetes* local.

Definimos e desenvolvemos três aplicações que apresentam requisitos diferentes. Um dos cenários que nos interessa é o de serviços em pipeline. Esse cenário é caracterizado pelo encadeamento de serviços em que um evento inicial gera eventos para outros serviços desenvolvidos por nós. Um outro cenário é o de desenvolvimento de um serviço que utiliza outros serviços internamente.

Com o desenvolvimento dessas aplicações foi possível analisar a flexibilidade oferecida por cada plataforma, incluindo suas facilidade e limitações, tanto durante o desenvolvimento quanto durante seu funcionamento.

Para escolhermos as aplicações, tentamos buscar requisitos diferentes que pudessem explorar possíveis cenários um pouco mais complexos para esse modelo de desenvolvimento.

A primeira aplicação é um serviço de postagem de mensagens, similar a um fórum. Nessa aplicação o *backend* vai utilizar o modelo *serverless*.

A segunda aplicação é um sistema de log para chamadas feitas a um *endpoint* de busca, que permite ao usuário fazer buscas em um histórico de execuções de simuladores rodados no servidor. Nossa aplicação vai servir para extrair e armazenar quais termos de busca foram usados nas consultas feitas pelos usuários para que seja possível, por exemplo, verificar que tipo de consultas são mais comuns.

A outra aplicação é uma implementação de um *MapReduce* (12) que busca testar o modelo no desenvolvimento de uma aplicação mais complexa.

4
Ambientes de desenvolvimento

Nesta seção descrevemos os dois ambientes principais que utilizamos para o desenvolvimento das aplicações no modelo *serverless*: (i) Google Cloud. (ii)Kubernetes. Escolhemos estes ambientes pelo fato de os termos disponíveis para a realização dos nossos testes e, além disso, permitir exercitarmos o desenvolvimento das aplicações em um ambiente de nuvem e outro baseado em uma instalação *on-premise*.

4.1
Google Cloud

A Google Cloud é uma plataforma de nuvem que oferece um serviço de *software as a service (FaaS)* e disponibiliza também outros diversos tipos diferentes de produtos. O desenvolvedor paga somente pelo que usa de cada produto, por exemplo, pelo tempo de execução de uma função, ou pela quantidade de dados armazenados no serviço de armazenagem da plataforma.

O principal serviço que vamos utilizar aqui é o *Google Functions*, que é a implementação *FaaS* do Google. Nesse serviço, podemos desenvolver cada função em uma das quatro linguagens oferecidas: Nodejs, Python, Java e Go.

Nesse ambiente a execução das funções ocorre nos servidores do Google e toda a configuração do ambiente de execução das funções são feitas por eles. Com isso, temos algumas limitações impostas pelo ambiente na execução das funções. Por exemplo, uma função não pode usar mais de 2048MB de memória principal, existe também um tempo máximo de 540 segundos(9 minutos) de execução para cada instância das funções. Esses limites são descritos na documentação[1].

Podemos criar as funções de três maneiras principais, através do site do Google Cloud que oferece um editor de texto online e podemos escrever todo o código lá, submetendo a função ao terminarmos. Podemos também, desenvolver o código em nosso ambiente local e quando queremos criar ou atualizar uma função com o código desenvolvido, basta usar o *gcloud*[2], uma interface de linha de comando, ou fazer uma chamada para sua API[3].

[1] https://cloud.google.com/functions/quotas
[2] https://cloud.google.com/sdk/gcloud/reference/functions
[3] https://cloud.google.com/functions/docs/reference/rest

No momento de criação das funções, podemos passar por parâmetro algumas configurações como, por exemplo, variáveis de ambiente, quantidade de memória alocada para a função, tipo de evento que vai invocar a função, numero máximo de instancias, dentre outros. O limite de memória para cada instância de uma função é de 2GB.

Os principais eventos que podem ser usados para invocar funções são: chamadas HTTP, mensagem no serviço de mensageria do Google e eventos no *Cloud Storage*, um dos serviços de armazenamento do Google, dentre outros que não serão usados aqui.

4.2 Kubernetes

O *Kubernetes*[4] é uma plataforma *open source* de orquestração e gerenciamento de *clusters* de *containers* Docker[5], que automatiza, por exemplo, o *deploy* e a escalabilidade de uma aplicação.

Figura 4.1: Exemplo de arquivo de deployment

controllers/nginx-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Através de um arquivo de configuração como o da figura 4.1, que define as propriedades de uma aplicação, o *Kubernetes* pode executar essa aplicação em qualquer nó do cluster, usando uma imagem Docker. Isso simplifica bastante o processo de *deploy*.

O *Kubernetes* oferece uma ferramenta chamada *Kubeless* que, uma vez instalada no cluster, dá suporte ao modelo *serverless*. Além disso, usaremos

[4] https://kubernetes.io/
[5] https://www.docker.com/

o *Apache Kafka* como serviço de mensageira, para possibilitar que as funções utilizem comunicação *Pub/Sub*.

Para o desenvolvimento das funções no *Kubeless*, de acordo com a documentação[6], podemos usar as seguintes linguagens: Nodejs, Python, Ruby, Java, Php, Go, Ballerina.

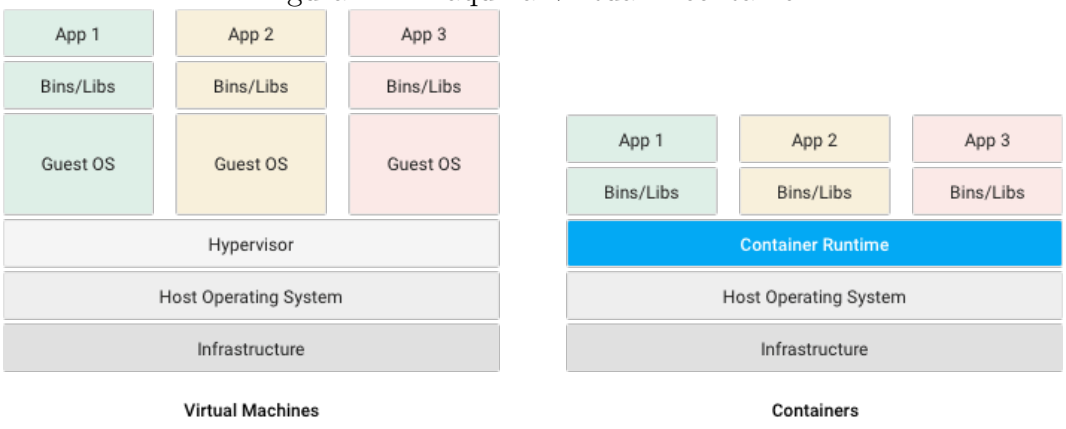
Para invocar as funções, o *Kubeless* aceita como eventos: chamadas HTTP, menssagens enviadas ao Kafka e disparadas por passagem de tempo, um *cron job*. Para criar as funções no *Kubeless*, podemos usar sua interface de linha de comando ou podemos usar um arquivo de *deploy* do *Kubernetes*. No momento da criação, definimos o tipo de trigger que a função vai usar. Também podemos definir algumas configurações como, por exemplo, limitar o número de CPUs, a quantidade de memória disponível para a função e o tempo de *timeout* dela.

Para o desenvolvimento das aplicações no nosso trabalho, utilizamos o Minikube[7] que simula um ambiente do *Kubernetes*. Também utilizamos, em alguns testes, um ambiente Kubernetes com 10 nós disponíveis em uma infraestrutura local.

4.2.1 Containers

Os *containers*, orquestrados pelo *Kubernetes*, oferecem um mecanismo de empacotamento de aplicativos que permite abstrair o ambiente em que o aplicativo vai ser executado, desacoplando assim, a plataforma de execução do desenvolvimento do software. São muito similares às máquinas virtuais, mas são mais otimizadas para ter um menor impacto de desempenho na máquina física que está rodando os *containers*.

Figura 4.2: Máquina Virtual x container



[6] https://kubeless.io/docs/runtimes/
[7] https://kubernetes.io/docs/setup/learning-environment/minikube/

Dessa forma, basta o desenvolvedor definir o *container* de sua aplicação para que ela possa executar em qualquer outra máquina que utilize essa tecnologia. Na hora de executar a aplicação, a máquina executa o *container* e não a aplicação diretamente. O *container* de uma aplicação inclui todas as suas dependências, como pacotes e bibliotecas, representadas na figura 4.2 como "Bins/Libs".

Essa tecnologia foi essencial para o avanço da computação na nuvem; com ela a execução de aplicações de usuários é muito mais simples e independente da configuração física das máquinas que compõem a infra-estrutura da nuvem. Qualquer máquina pode executar qualquer aplicação usando o *container* definido por ela.

5
Atividades realizadas

Inicialmente fizemos um estudo abrangente sobre o tema, buscando informações gerais como possíveis benefícios e dificuldades do modelo *serverless*. Em seguida, passamos a verificar quais plataformas ofereciam o modelo, quais as diferenças entre os serviços oferecidos, optando então por utilizar o *Kubeless* para a plataforma *Kubernetes* e *Google Cloud Functions* por termos fácil acesso a esses dois ambientes. Passamos então a realizar estudos mais específicos sobre cada uma dessas plataformas. Durante o período de desenvolvimento do projeto, semanalmente, discutimos artigos diferentes sobre o tema para nos ajudar a entender melhor esse modelo em cada plataforma. Isso nos ajudou bastante porque tivemos alguma dificuldade em avaliar os pontos negativos do modelo através da documentação das plataformas uma vez que, aparentemente, existe muito marketing tentando vender o modelo e os pontos negativos do modelo são por vezes omitidos.

Em seguida, a partir do estudo inicial, definimos as aplicações que desenvolvemos. Cada aplicação teve requisitos distintos para que fosse possível identificar vantagens e desvantagens do modelo. De início, desenvolvemos uma aplicação de fórum, para adquirir familiaridade com o modelo de programação e com os ambientes. Em seguida, desenvolvemos uma aplicação com uso real que consolidava o log de uma aplicação de busca. Por fim, tentamos implementar uma aplicação de *MapReduce* que, devido a dificuldades de implementação, não conseguimos concluir. O código das aplicações desenvolvidas está no github do projeto[1].

[1] https://github.com/Pedro-Magalhaes/ProjetoFinal

6
Análise e desenvolvimento

Para escolhermos as aplicações, buscamos escolher requisitos diferentes para explorarmos possíveis cenários variados para esse modelo de desenvolvimento.

Na primeira aplicação,desenvolvemos um serviço de postagem de mensagens, similar a um fórum. Nessa aplicação o *backend* utilizou o modelo *serverless*.

A segunda aplicação é um sistema de log para chamadas feitas a um *endpoint* que permite ao usuário fazer buscas em um histórico de execuções de simuladores. Nossa aplicação extrai o termo buscado e o armazena em um arquivo com as consultas feitas pelos usuários, permitindo, por exemplo, que uma outra aplicação possa usar esse log para verificar que tipo de consultas são mais comuns.

Por último, desenvolvemos uma implementação de um *MapReduce* (12), para exercitar um cenário de aplicação mais complexa mas que tem um potencial grande de se beneficiar de um ambiente distribuído.

6.1
Fórum

O objetivo da aplicação de Fórum foi ganhar familiaridade com o modelo e com os ambientes de desenvolvimento. Ela foi desenvolvida seguindo um tutorial online(14) que mostra o desenvolvimento utilizando o *Kubeless* como plataforma *serverless*.

Nessa aplicação, desenvolvemos o *frontend* e o *backend* de um fórum. Através de uma interface web, o usuário pode criar uma postagem com um título e um texto associado, ler as postagens já criadas e visualizar novas postagens no fórum criadas por outros usuários. No *backend* vamos utilizar o modelo *serverless*.

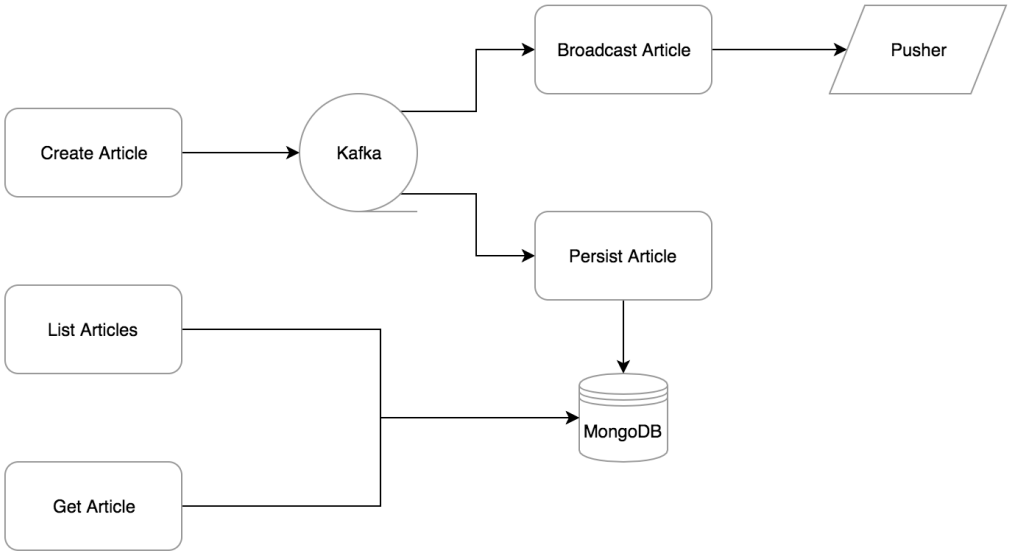
Para o funcionamento do *backend* da aplicação temos três módulos principais: i) o módulo que persiste uma postagem do usuário. ii) o módulo que lista as postagens existentes. iii) módulo que notifica um usuário de novas postagens.

Para a persistência dos dados de postagem, temos a dependência de um banco de dados, que nesse caso foi o mongoDB. Além disso, para que o usuário seja notificado de novas postagens o *frontend* se conecta via *websocket* com o *backend*.

O *backend* seguiu a arquitetura da figura 6.1, onde cada retângulo representa uma função independente. Dessa forma, por exemplo, ao criar um novo tópico, o cliente web tem que fazer uma requisição http para a função *Create Article* que, por sua vez, produz uma mensagem e a envia para o serviço de mensageria. Em seguida, duas outras funções são invocadas pois estão subscritas ao serviço no canal onde a função *Create Article* publica mensagens. A *Persist Article* salva no mongoDB o tópico e a *Broadcast Article* comunica ao serviço *Pusher* que um novo tópico foi criado.

O serviço *Pusher* implementa uma comunicação via *websocket* para que o *frontend* possa estabelecer uma conexão e receber atualizações em tempo real sobre a criação de novos tópicos. Como o *websocket* é uma conexão persistente e não é suportada pelas funções do modelo *serverless* no *Kubeless* e no *Google Functions*, o *Pusher* foi usado como intermediário.

Figura 6.1: *Backend* da aplicação do fórum



Durante o desenvolvimento do Fórum, identificamos algumas diferenças interessantes entre as duas plataformas. No *Kubeless* temos que lidar com tarefas de instalação e configuração dos serviços, o que causa um pouco mais de dificuldade, mas permite também um maior grau de liberdade nas configurações. Já o *Google Cloud* traz um ambiente mais simples para o desenvolvimento, com diversas facilidades para criar e executar as funções. No entanto, não temos muita liberdade de configuração. Um exemplo é que,

enquanto no *Kubernetes* tivemos que configurar a regra de ingresso no *cluster* para que uma função ficasse acessível para um cliente fora dele, no ambiente do Google isso vem de graça mas não temos, por exemplo, a opção de configurar a URL gerada para o acesso.

Além disso, verificamos diferenças na hora de fazer *deploy* do nosso código. Na plataforma de desenvolvimento do *Google Cloud*, podemos cadastrar as funções de três formas principais: (i) através de uma aplicação do terminal que auxilia no envio do código, (ii) diretamente no site do *Google Cloud* através do editor online, ou (iii) usando sua API. Já no *Kubeless*, temos a aplicação do terminal como interface para cadastrar as funções, ou podemos criar a função através do arquivo de *deploy* do *Kubernetes*.

Outro ponto importante é que a arquitetura da biblioteca que usamos para desenvolver as funções em cada plataforma é bem diferente em cada caso. Dessa forma, parte do código teve que ser reescrito quando terminamos a aplicação de teste em uma plataforma e fomos implementá-la na outra. Esse trabalho, portanto, mostra que a escolha da plataforma é uma decisão muito importante, uma vez que se for necessário trocar de plataforma, boa parte do código desenvolvido vai precisar ser refeito.

Também verificamos que não há suporte para conexões via *websocket* através das funções no modelo *serverless* nas duas plataformas. Com isso, foi necessário utilizar um serviço de terceiros (Pusher[1]) para a conexão com o cliente web. A plataforma do Google já possui um serviço chamado *App Engine*[2] que dá suporte a essa conexão e poderia ser usado no lugar do *pusher* na versão do aplicativo implementado na plataforma do Google.

6.2 Log

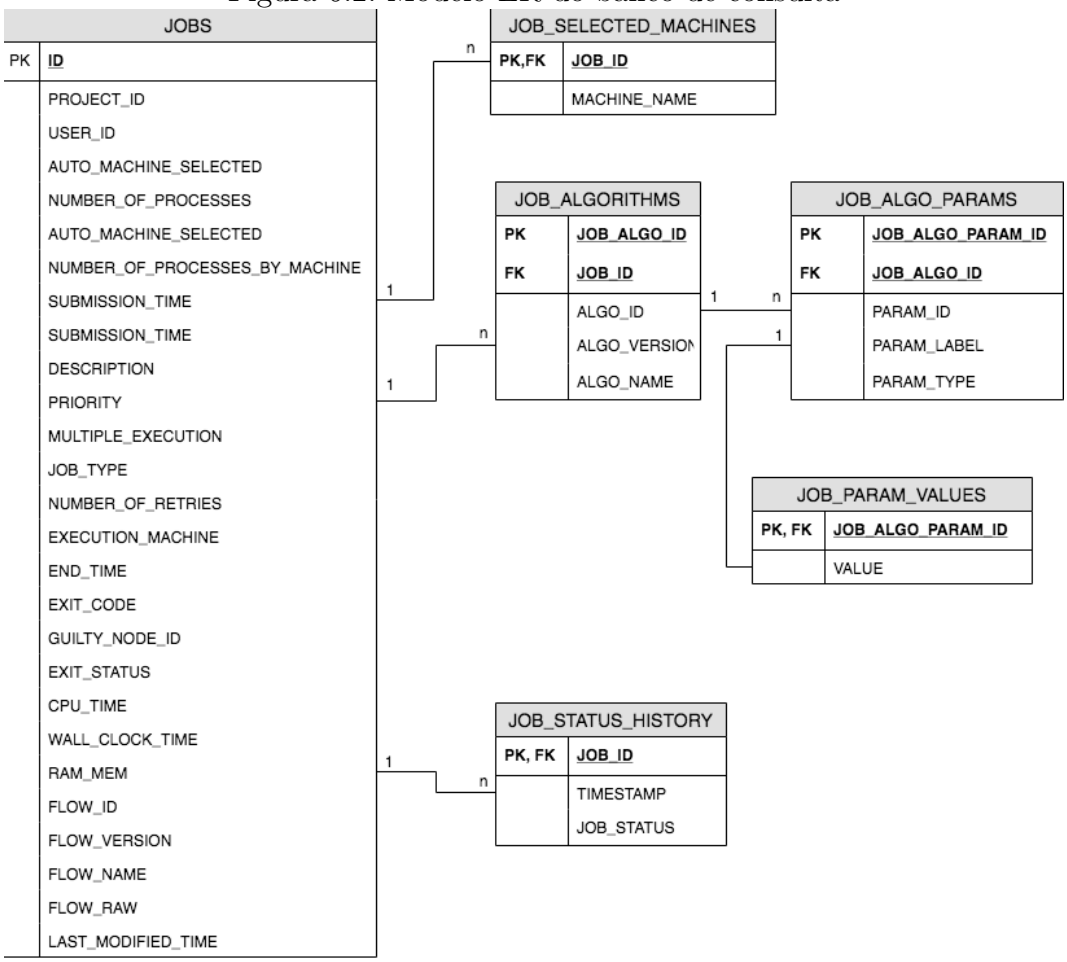
O objetivo do desenvolvimento da aplicação de log foi o de testar um cenário que parece favorável ao uso do modelo *serverless*, onde já temos um serviço em funcionamento e queremos adicionar uma funcionalidade de forma rápida e sem alterar o código do serviço.

Para essa aplicação utilizamos um *endpoint* que permite a busca em um histórico de execução de simulações submetidas por usuários de uma plataforma web. Os usuários são pesquisadores, em sua maioria, e executam simulações complexas que, muitas vezes, são fluxos compostos por diferentes simuladores. Esse serviço está em produção e permite ao usuário enviar

[1] https://pusher.com/
[2] https://cloud.google.com/appengine

consultas no formato RSQL (13), um formato similar ao SQL, para criar filtros de busca.

Figura 6.2: Modelo ER do banco de consulta

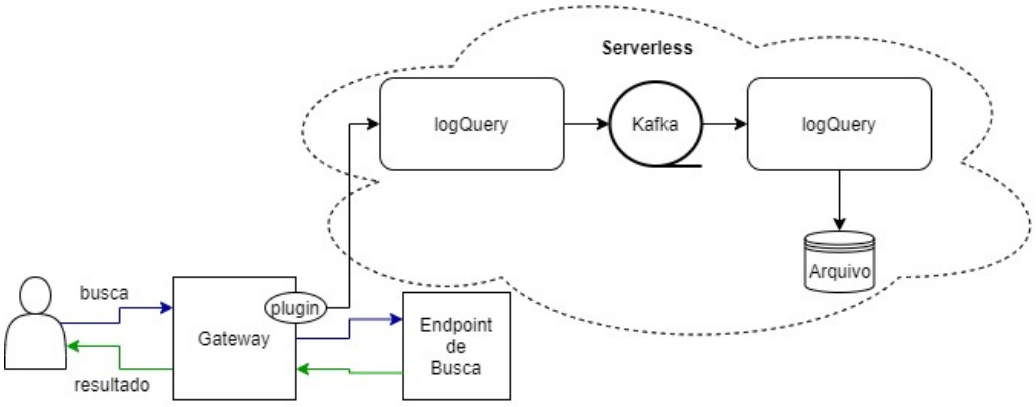


O objetivo da aplicação *serverless* que desenvolvemos foi gerar um log do uso do*endpoint* de consulta de execuções, guardando as consultas feitas pelos usuários (figura 6.2), e permitindo assim que, futuramente, uma outra aplicação consulte esse log para, por exemplo, verificar as consultas mais comuns.

Nesta aplicação, cada consulta feita pelos usuários ao *endpoint* analisado aciona uma função no modelo *serverless* através de uma chamada HTTP. Essa função, por sua vez, cria uma mensagem no serviço de mensageria com um Id único e a consulta feita. A mensagem ativa a função que persiste esse dado em um arquivo com o log completo (figura: 6.3).

Como o serviço de busca já estava implementado e em funcionamento, implementamos um *plugin* no *gateway* do sistema. O *plugin* é acionado quado uma consulta é feita ao serviço de busca e faz a chamada HTTP que ativa a função *logQuery* passando como parâmetro a busca feita pelo usuário.

Figura 6.3: Modelo da aplicação de Log



O modelo de programação *serverless* se encaixou bem com essa aplicação e foi possível desenvolver a integração com um sistema em funcionamento sem ter que mexer em seu código. A única alteração feita foi a de adicionar o plugin ao *gateway*.

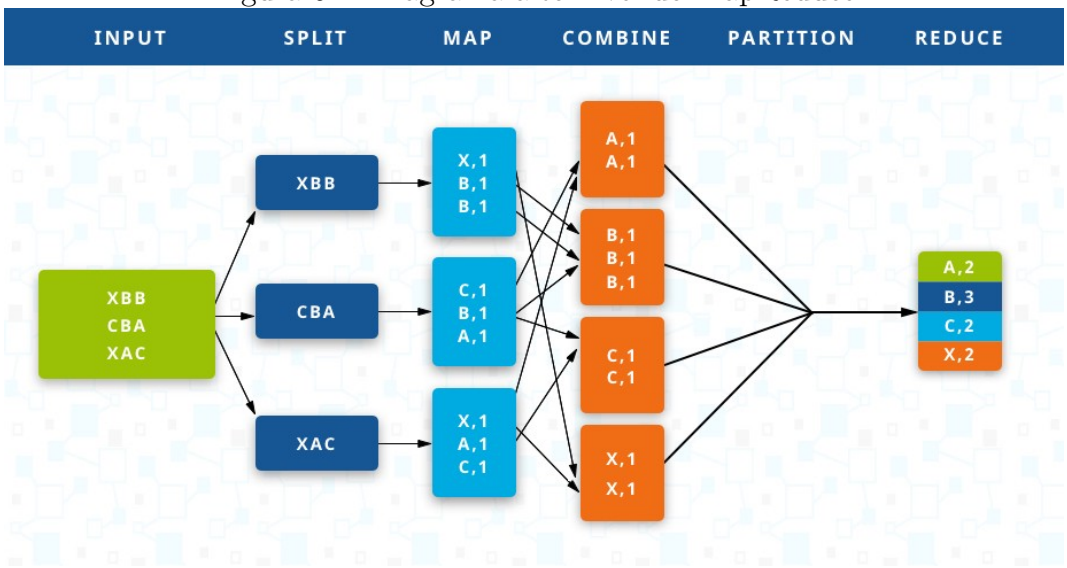
Percebemos durante o desenvolvimento dessa aplicação que testar aplicações nesse modelo não é uma tarefa simples. O teste que conseguimos fazer de forma local é muito limitado e em geral não era suficiente para auxiliar no desenvolvimento. Na maioria das vezes o teste foi feito instalando a função com mensagens de *debug* no ambiente de execução das plataformas e verificando o log da execução. Muitas vezes também, algum erro na função abortava sua execução e não havia uma mensagem clara sobre o que tinha ocorrido.

Verificamos aqui que novamente foi mais complexo implementar a aplicação no *Kubeless*. Isso porque, tivemos que criar uma área no *Kubernetes* para persistir os dados do Log e vincular essa área à função que vai acessá-la, o que não é uma tarefa muito simples. No Google, basta configurar o *Google Storage* e usar sua API dentro da função para conseguir lidar com o acesso a arquivos.

6.3 Aplicação de *MapReduce*

O objetivo de desenvolver a aplicação de *MapReduce* foi verificar as dificuldades que encontraríamos para construir uma aplicação mais complexa que, por exemplo, depende de sincronização entre as funções para passar da etapa de *Map* para a de *Reduce*. Nesse caso, temos que saber se todas as instâncias da função de *Map* já terminaram seu trabalho para acionar as as instâncias do *Reduce*. Além disso a aplicação precisa utilizar algum modelo de persistência para guardar o estado e transmitir a informação produzida na etapa *Map* para o *Reduce* 6.4.

Figura 6.4: Diagrama alto nível do MapReduce



Usualmente uma implementação de *MapReduce* funciona da seguinte forma: um processo coordenador inicia a execução e particiona os dados recebidos para os N processos de Map. Enquanto esses processos executam, o coordenador se comunica com os processos de Map para verificar se eles estão ativos ou se abortaram sem finalizar sua tarefa. Toda vez que um processo de *Map* finaliza seu trabalho ele avisa ao coordenador que verifica se ainda existem outros processos de *Map* executando. Caso contrário, o coordenador passa para a etapa de *Reduce* que tem uma interação parecida com o processo coordenador. Em cada uma das etapas, se um processo para de se comunicar com o coordenador sem que tenha terminado seu trabalho, um novo processo é instanciado para refazer seu trabalho, criando um mecanismo de tolerância a falhas.

Logo no inicio do desenvolvimento da aplicação verificamos que teríamos dificuldades nesse cenário. Isso porque, a limitação de tempo máximo de execução de uma função e a impossibilidade de comunicação direta entre instâncias de funções ativas tornam muito complexa a criação de uma função *serverless* com o papel de coordenação. Uma vez que as funções não guardam estado e cada plataforma implementa persistência de forma diferente, tivemos grande dificuldade mesmo na fase de planejamento. Outros pontos interessantes que precisam ser observados nessa aplicação é a possibilidade de comunicação entre as funções, o acesso concorrente a arquivos e a facilidade de escalar o número de *workers* em cada etapa.

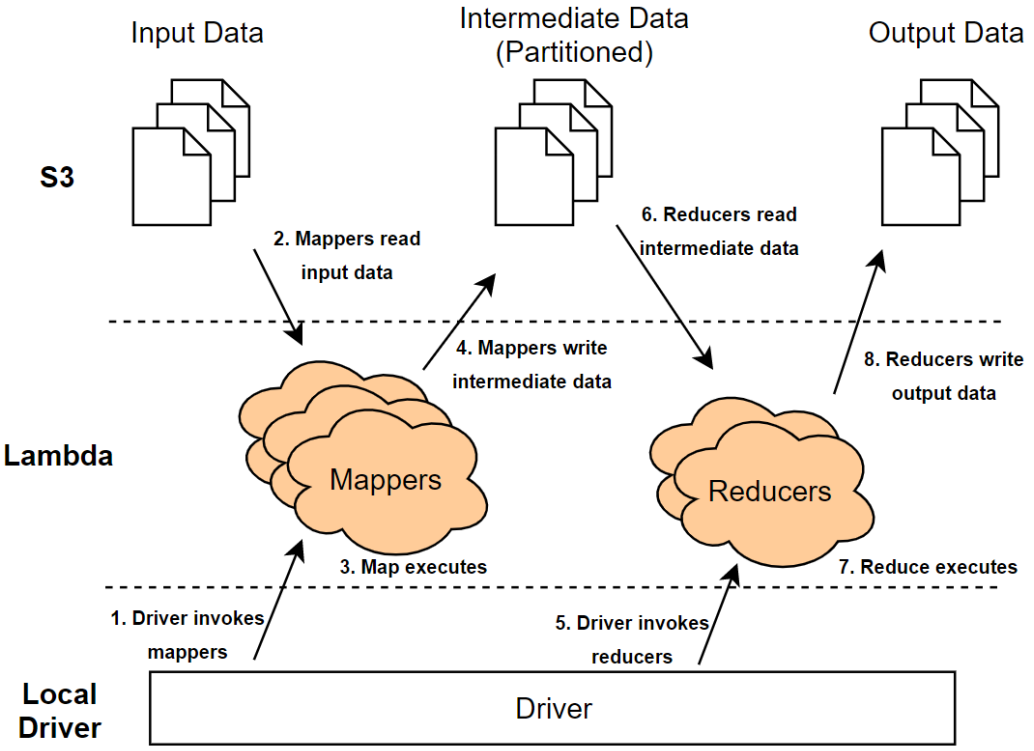
Estudando a literatura para verificar soluções para esse problema de coordenação, verificamos que esse realmente é um problema em aberto do modelo. Por exemplo, no artigo "Serverless Computing: Current Trends and

Open Problems"(4), os autores citam o fato de não conseguirmos rodar uma função por mais tempo e a falta de comunicação entre instâncias de funções como problemas em aberto que limitam o desenvolvimento de funções com papel de coordenação. Já no artigo "A mixed-method empirical study of Function-as-a-Service software development in industrial practice"(15), em que os autores entrevistaram desenvolvedores que utilizavam o *FaaS* para o desenvolvimento de aplicações e sistemas, os entrevistados relataram que o modelo *serverless* não seria indicado para programas que necessitam de processos com longa duração, como o processo de coordenação do *MapReduce*.

Em seguida, passamos a buscar por exemplos de implementação do *MapReduce* utilizando o modelo *serverless* para basear nossa implementação. Estudamos duas implementações que descreviam como evitaram o problema com o processo de coordenação.

Na primeira implementação(16), o autor utilizou um modelo híbrido em que o processo de coordenação executa na máquina local do usuário enquanto os processos de *Map* e *Reduce* executam como funções *serverless* no ambiente da AWS, como na figura 6.5. Optamos por não replicar essa implementação para tentar uma abordagem usando somente o modelo *serverless*.

Figura 6.5: Diagrama alto nível Corral

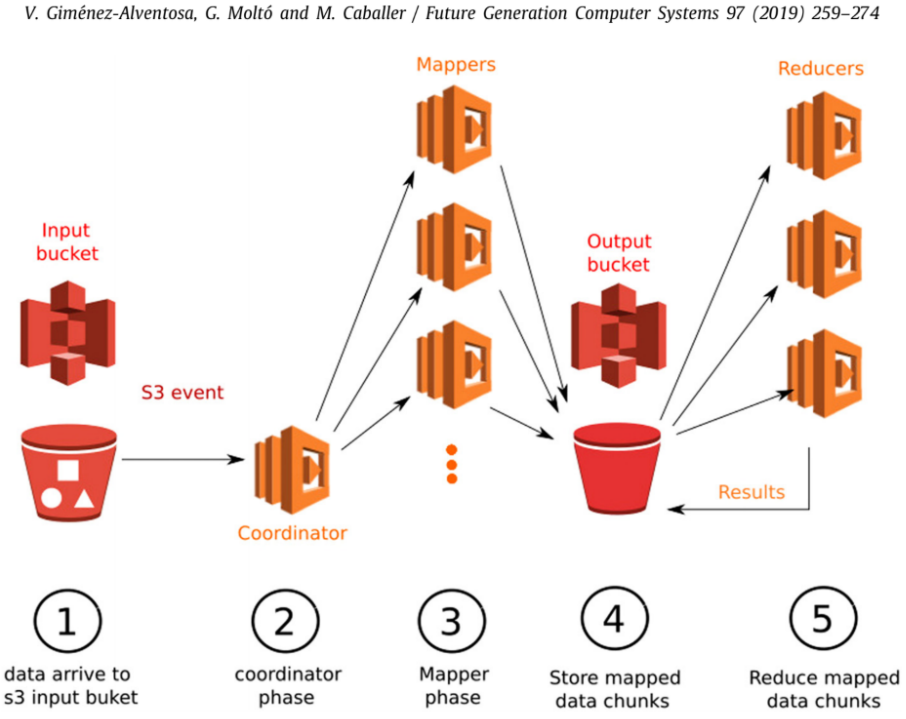


A segunda implementação(17) que analisamos, desenvolveu um *framework* chamado Marla[3] para a execução de um *MapReduce* utilizando o

[3] https://github.com/grycap/marla

modelo *serverless* que também utiliza o ambiente da AWS. Nessa versão, o autor tentou solucionar o problema de coordenação delegando as tarefas de coordenação para as fases de *Map* e *Reduce*, ou seja, cada uma das funções herdou também parte da lógica da coordenação. Por exemplo, na fase do *Map*, existe uma lógica que é executada pela instância da função de *Map* que recebe a penúltima partição do dado de entrada para ativar a primeira instância da função de *Reduce* que, por sua vez, fica responsável por verificar se todas as instâncias de *Map* escreveram os dados de entrada para que a etapa de *Reduce* possa ser iniciada. Para contornar o problema do tempo limite de execução no teste de termino da etapa de *Map*, o autor implementou uma chamada "recursiva", em que uma instância da função *Reduce* só testa algumas vezes para verificar o fim da fase de *Map*. Se o fim não for identificado nessas iterações a função invoca outra para continuar esse processo até que as funções de *Map* finalizem. Quando o final do *Map* é identificado, a função de *Reduce* que está ativa, com papel de coordenação, ativa as instâncias da função *Reduce* que vão efetivamente continuar a execução da aplicação.

Figura 6.6: Diagrama alto nível Marla



Passamos então a tentar implementar o *MapReduce* seguindo o exemplo do *Marla*, no entanto tivemos muitas dificuldades.

Inicialmente, tivemos problemas com o *Kubeless* porque, para iniciar o *MapReduce*, a função que inicia o processo teria que receber um evento indicando que um arquivo foi adicionado, passando o local e o nome do

arquivo. No entanto, no *Kubeless* não existe esse *trigger*. Devido a dificuldade em implementar um novo *trigger* no *Kubeless*, resolvemos passar a iniciar a execução através de uma chamada HTTP que indicaria o local do arquivo para a função inicial.

Em seguida, passamos a ter problemas ao portar o código do Marla para as plataformas que estamos utilizando, porque o código foi feito para executar no ambiente da AWS e, além da diferença natural do código para cada plataforma, dependia de algumas variáveis de ambiente. Para cada plataforma, a forma de acessar as informações necessárias é diferente e, no caso do *Kubeless*, algumas variáveis não existem.

Por fim, o que mais dificultou o desenvolvimento foi a dificuldade de testar o código que íamos produzindo. Como essa aplicação era mais complexa e a única forma de testar as funções era submetendo uma execução, acabávamos perdendo muito tempo para pegar erros simples no código.

Com a data limite de entrega do projeto se aproximando, acabamos desistindo de implementar a aplicação, decidindo relatar aqui os problemas de uma implementação de *MapReduce* nesse ambiente avaliando os problemas que nós tivemos durante o desenvolvimento e o que foi observado nas duas implementações que vimos para o *MapReduce* no modelo *serverless*.

7
Conclusão

Nosso trabalho buscou avaliar o impacto para o desenvolvedor ao utilizar o modelo de programação *serverless* no desenvolvimento de diferentes tipos de aplicações. Verificamos que esse modelo vêm crescendo bastante, com diversas plataformas oferecendo esse tipo de ambiente para os desenvolvedores. Mesmo as que já oferecem o serviço de *serverless* há mais tempo, estão sempre atualizando o ambiente para atrair mais usuários. Por exemplo, verificamos que na plataforma do Google novas versões de Python e NodeJs foram disponibilizadas para uso.

Através do estudo de artigos relacionados e do desenvolvimento das aplicações nesse trabalho, concluímos que, por enquanto, esse modelo se aplica melhor a cenários específicos. Componentes de aplicações que dependem de muito I/O, de comunicação entre as funções ou de preservação de estado dificultam sua utilização pelo desenvolvedor.

Um exemplo dessas dificuldades foi possível de experimentar na aplicação de *MapReduce* que requer uma coordenação entre as partes da solução. O ciclo de vida curto e impossibilidade de comunicação direta entre as partes dificulta a coordenação e a construção de tolerância a falhas, necessários na aplicação *MapReduce*. Outro fator que dificulta o desenvolvimento de aplicações mais complexas é a dificuldade de implementar testes que consigam abstrair o ambiente das plataformas.

Além disso, ficamos com a impressão que o uso de uma plataforma *serverless* no modelo *on-premise* não faz muito sentido. Tipicamente, é um modelo que se aplica bem a plataformas de nuvem, onde auto-escalabilidade e "pague-o-que-use" é uma das premissas dessa infraestrutura.

Por outro lado, funcionalidades que tem um ciclo de vida curto, são acionadas por eventos, não dependem de estado, não tem um volume alto de I/O e não dependem de comunicação direta, podem se beneficiar bastante da auto-escalabilidade e da simplicidade do modelo de desenvolvimento em *serverless*, principalmente aqueles oferecidos em plataformas de nuvem. No caso da aplicação de log, por exemplo, não tivemos muita dificuldade no desenvolvimento. O único ponto mais complicado foi a maneira de acessar o arquivo que armazena o log pois em cada plataforma a forma de acesso e

escrita era diferente. Apesar disso acreditamos que o modelo se encaixou bem com a aplicação.

Um outro ponto que verificamos é que o efeito de *lock-in* esteve presente em todas as aplicações que desenvolvemos. Grande parte do código tinha que ser alterada quando mudávamos de plataforma e acreditamos que para aplicações maiores compostas por diversas funções o esforço de migrar de uma plataforma para a outro pode acabar sendo muito alto.

Por fim, observamos que a maioria dos estudos se baseiam na plataforma AWS da Amazon e nos parece que para um primeiro contato com o modelo pode ser interessante começar por esta plataforma simplesmente pela maior disponibilidade de material online.

Referências bibliográficas

[1] PAULSON, L. D.. **Building rich web applications with Ajax**. Computer, 38(10):14–17, 2005.

[2] IYENGAR, A.; MACNAIR, E. ; NGUYEN, T.. **An analysis of web server performance**. In: GLOBECOM 97. IEEE GLOBAL TELECOMMUNICATIONS CONFERENCE. CONFERENCE RECORD, volumen 3, p. 1943–1947. IEEE, 1997.

[3] JAMSHIDI, P.; PAHL, C.; MENDONÇA, N. C.; LEWIS, J. ; TILKOV, S.. **Microservices: The journey so far and challenges ahead**. IEEE Software, 35(3):24–35, 2018.

[4] BALDINI, I.; CASTRO, P.; CHANG, K.; CHENG, P.; FINK, S.; ISHAKIAN, V.; MITCHELL, N.; MUTHUSAMY, V.; RABBAH, R.; SLOMINSKI, A. ; OTHERS. **Serverless computing: Current trends and open problems**. In: RESEARCH ADVANCES IN CLOUD COMPUTING, p. 1–20. Springer, 2017.

[5] FROMM, K.. **Why the future of software and apps is serverless (2012)**. URL http://readwrite. com/2012/10/15/why-the-future-of-software-and-apps-is-serverless, 2018.

[6] NAMIOT, D.; SNEPS-SNEPPE, M.. **On micro-services architecture**. International Journal of Open Information Technologies, 2(9):24–27, 2014.

[7] BERNSTEIN, D.. **Containers and cloud: From LXC to Docker to Kubernetes**. IEEE Cloud Computing, 1(3):81–84, 2014.

[8] ZHU, K. X.; ZHOU, Z. Z.. **Research note—lock-in strategy in software competition: Open-source software vs. proprietary software**. Information Systems Research, 23(2):536–545, 2012.

[9] NUPPONEN, J.; TAIBI, D.. **Serverless: What it is, what to do and what not to do**. In: 2020 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE COMPANION (ICSA-C), p. 49–50. IEEE, 2020.

[10] JONAS, E.; SCHLEIER-SMITH, J.; SREEKANTI, V.; TSAI, C.-C.; KHANDELWAL, A.; PU, Q.; SHANKAR, V.; CARREIRA, J.; KRAUTH, K.; YADWADKAR, N. ; OTHERS. **Cloud programming simplified: A berkeley view on serverless computing**. arXiv preprint arXiv:1902.03383, 2019.

[11] HELLERSTEIN, J. M.; FALEIRO, J.; GONZALEZ, J. E.; SCHLEIER-SMITH, J.; SREEKANTI, V.; TUMANOV, A. ; WU, C.. **Serverless computing: One step forward, two steps back**. arXiv preprint arXiv:1812.03651, 2018.

[12] DEAN, J.; GHEMAWAT, S.. **Mapreduce: Simplified data processing on large clusters**. Communications of ACM, 51(1):107–113, Jan. 2008.

[13] JÄKEL, T.; KÜHN, T.; VOIGT, H. ; LEHNER, W.. **Rsql – a query language for dynamic data types**. In: PROCEEDINGS OF THE 18TH INTERNATIONAL DATABASE ENGINEERING & APPLICATIONS SYMPOSIUM, p. 185–194. ACM, 2014.

[14] COX, G.. **Adding realtime functionality to a blog using Kubeless**, 2018. Acesso em: 11/11/2019.

[15] LEITNER, P.; WITTERN, E.; SPILLNER, J. ; HUMMER, W.. **A mixed-method empirical study of function-as-a-service software development in industrial practice**. Journal of Systems and Software, 149:340–359, 2019.

[16] CONGDON, B.. **Introducing corral: A serverless mapreduce framework**, 2018. Acesso em: 24/07/2020.

[17] GIMÉNEZ-ALVENTOSA, V.; MOLTÓ, G. ; CABALLER, M.. **A framework and a performance assessment for serverless mapreduce on aws lambda**. Future Generation Computer Systems, 97:259–274, 2019.