ORBITER: um Arcabouço para Implantação Automática de Aplicações *Big Data em Arquiteturas Serverless*[*]

João Loureiro[1], **Daniel de Oliveira**[1]

[1]Instituto de Computação - Universidade Federal Fluminense (UFF)

joao_loureiro@id.uff.br, danielcmo@ic.uff.br

Resumo. *Tem se tornado cada vez mais comum a oferta de mecanismos de computação serverless em nuvens públicas. Embora o desenvolvimento de uma aplicação seguindo uma arquitetura serverless apresente vantagens, o mesmo ainda traz consigo alguns desafios como a baixa portabilidade. Além disso, a arquitetura serverless também acaba adicionando um certo nível de morosidade na construção da arquitetura de aplicações distribuídas, que já é algo complexo. Este artigo propõe um arcabouço para implantação de aplicações em uma arquitetura serverless, de forma fácil para o desenvolvedor. O arcabouço proposto foi desenvolvido utilizando ferramentas de código aberto, e foi avaliado com uma aplicação de análise de dados de acidentes de trânsito da ANTT.*

Abstract. *It has become increasingly common to offer serverless computing mechanisms in public clouds. Although the development of an application following a serverless architecture presents advantages, it still brings challenges such as low portability. In addition, the serverless architecture also ends up adding a certain level of delay in building the architecture of distributed applications, which is already complex. This paper proposes a framework for deploying big data applications in a serverless architecture. The proposed framework was developed using open source tools and was evaluated with an ANTT traffic accident data analysis application.*

1. Introdução

Na última década, temos observado um aumento na produção e disponibilização de dados, sejam eles estruturados, semiestruturados ou não estruturados. De fato, o dados têm sido produzidos e disponibilizados por uma gama de aplicações que variam desde redes sociais até dispositivos de IoT (*e.g.*, sensores pluviométricos e *wearables*). Esses dados podem ser encontrados em uma granularidade muito fina e em escalas espaço-temporais sem precedentes [Nandury and Begum 2016]. Apesar de tal volume de dados possibilitar diversas oportunidades de pesquisa tanto na academia quanto na indústria, ele acaba trazendo também um grande desafio, que é como processá-lo de forma eficiente.

Diversas soluções têm sido propostas para apoiar o desenvolvimento de aplicações capazes de processar e extrair informação útil desses dados. Essas soluções variam de arcabouços para processamento distribuído em memória (*e.g.*, Apache Spark) até plataformas de processamento de *streams* (*e.g.*, Kafka). Tais soluções representaram um avanço

[*]O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. A pesquisa foi também apoiada parcialmente por CNPq e FAPERJ.

no que tange o apoio ao desenvolvimento de aplicações que precisam processar um grande volume de dados. Entretanto, essa miríade de soluções acabou por criar um ecossistema de *software* complexo que o desenvolvedor de aplicações deve dominar. Apesar dos provedores de nuvem existentes disponibilizarem serviços e imagens de máquinas virtuais com instalações prévias dessas soluções, a configuração e a manutenção de todo o ecossistema comumente não é trivial, e requer muito esforço por parte do desenvolvedor.

De forma a reduzir a complexidade do desenvolvimento e da implantação de aplicações distribuídas, surgiu o modelo *Serverless* [Hellerstein et al. 2019]. O modelo *serverless* é uma alternativa ao modelo tradicional de implantação de aplicações (especialmente em nuvem), no qual os desenvolvedores não são mais os responsáveis por gerenciar os servidores que executam as aplicações. Diferentemente da computação em nuvem tradicional, no modelo *serveless*, os desenvolvedores focam apenas na aplicação, deixando para o provedor de nuvem questões como escalabilidade, tolerância a falhas, *etc* [Hassan et al. 2021]. O modelo *serveless* segue a ideia do *pay-as-you-go*, *i.e.*, o desenvolvedor paga apenas pelo tempo em que sua aplicação executa. Apesar de apresentar as vantagens citadas anteriormente, o modelo *serverless* ainda apresenta alguns desafios associados [Sousa 2020], como por exemplo, a configuração do *cluster* de contêineres que contém a aplicação (*e.g.*, Kubernetes) e a portabilidade de aplicações entre mecanismos *serverless* (*e.g.*, Lambda da AWS e *Azure Functions* da MS Azure).

Com o objetivo de auxiliar o desenvolvedor na criação desse *cluster* de contêineres e na consequente portabilidade da aplicação entre diferentes mecanismos *serverless*, esse artigo apresenta o `ORBITER` (*framewOrk foR BIg daTa dEployment on seRveless mechanisms*). O `ORBITER` é um arcabouço que faz a implantação automática de aplicações *Big Data* seguindo uma arquitetura *serverless*. O `ORBITER` é capaz de implantar uma aplicação em mecanismos *serverless* de diferentes provedores de nuvem, sem que o desenvolvedor tenha que se preocupar com características específicas de tais provedores. O `ORBITER` foi avaliado com um estudo de caso de implantação de uma aplicação para processamento de dados de acidentes de trânsito da Agência Nacional de Transportes Terrestres (ANTT), e mostrou sua viabilidade. Esse artigo se encontra organizado em quatro seções além da Introdução. Na Seção 2 discutimos o modelo *serverless* e alguns trabalhos relacionados. Na Seção 3 apresentamos a arquitetura do `ORBITER`. Na Seção 4, avaliamos o `ORBITER` com um estudo de viabilidade, e, finalmente, na Seção 5 concluímos o presente artigo.

2. Computação *Serverless*

O modelo de computação *serverless* pode ser organizado em duas principais categorias [Mampage et al. 2021]: (i) *Backend* como serviço (BaaS, do inglês *Backend as a Service*) e (ii) Função como serviço (FaaS, do inglês *Function as a Service*). No BaaS, além da aplicação que está sendo desenvolvida, os desenvolvedores conseguem acessar serviços de terceiros, *e.g.*, autenticação e serviços de banco de dados. O acesso a esses serviços é realizado por meio de APIs específicas. Já no FaaS (foco do presente artigo), os desenvolvedores são responsáveis por definir a lógica da aplicação do lado do servidor, e a aplicação é executada inteiramente em contêineres (*e.g.*, Docker, Singularity, *etc*) que são gerenciados pelo provedor de nuvem. Esse contêineres não possuem estado, o que significa que aplicações que precisam de persistência de dados podem ter certa dificuldade de usar esse modelo (precisam usar SGBDs disponibilizados como funções). Além disso, são efêmeros, *i.e.*, projetados para serem executados em uma pequena janela de tempo. A grande maioria dos

provedores de nuvem já oferece mecanismos FaaS, *e.g.*, *Azure Functions* (MS Azure) e o Lambda (AWS). Assim, diferentemente do que ocorre no IaaS (*Infrastruture as a Service*), no FaaS o desenvolvedor não necessita criar uma máquina virtual, configurá-la e mantê-la (*e.g.*, aplicando *patches* de segurança, *etc*). Os servidores de nuvem são abstraídos durante o processo de desenvolvimento da aplicação, de forma a facilitar a operação.

Essa diferença no funcionamento faz com que o custo financeiro do uso de FaaS possa ser reduzido se comparado com IaaS em alguns cenários de uso, *e.g.*, com poucas requisições esparsas no tempo (essa vantagem já não acontece com aplicações que são intensivas em computação, uma vez que o custo de mecanismos FaaS usados por longas janelas de tempo pode superar o custo do IaaS). Outra vantagem da computação *serverless* é o tempo de implantação. Tanto máquinas virtuais quanto contêineres levam mais tempo para realizar a configuração inicial do que as funções no modelo *serverless*, pois é necessário configurar a infraestrutura *a priori*. Como as funções não precisam realizar essa configuração inicial para serem usadas, em geral são necessários somente alguns milissegundos para serem implantadas. Uma desvantagem existente no modelo *serverless* é sua portabilidade, uma vez que ainda é complicado transferir recursos de um mecanismo *serverless* para outro. Em geral, esses mecanismos são muito específicos em termos de tecnologia aplicada, o que dificulta a portabilidade em caso de necessidade de troca de provedor/serviço.

Algumas abordagens na literatura já foram propostas para auxiliar o desenvolvedor a realizar a implantação de suas aplicações seguindo uma arquitetura *serverless*. [Wang et al. 2020] propõem um serviço de armazenamento efêmero de baixa latência baseado em *cache* para apoiar a persistência de dados. Já [Perron et al. 2020] propõem mecanismos para execução de consultas como funções de forma a reduzir latência e o *overhead* do modelo *serverless*. Apesar de representarem um avanço, essas abordagens acabam focando somente no apoio ao armazenamento de dados e não todo o desenvolvimento da aplicação.

3. Abordagem Proposta: `ORBITER`

Conforme mencionado anteriormente, o `ORBITER` é um arcabouço que visa facilitar a implantação de aplicações seguindo o modelo *serverless* e sua arquitetura é apresentada na Figura 1. A arquitetura proposta é organizada em três camadas principais: (i) Inicialização, (ii) Ingestão/Armazenamento e (iii) Processamento.

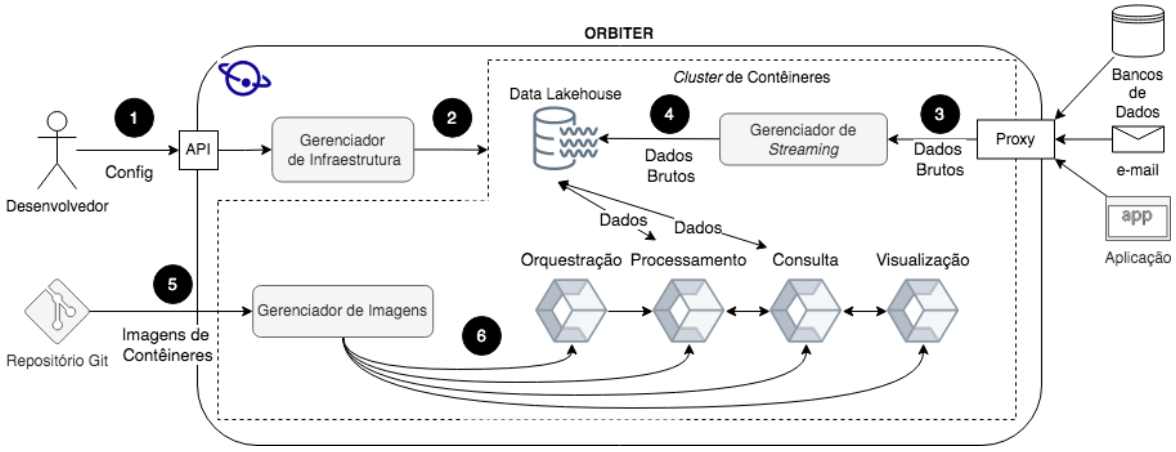

Figura 1. Arquitetura do `ORBITER`

A Camada de Inicialização é a responsável por criar o *cluster* de contêineres onde a aplicação final executa. O principal componente dessa camada é o *Gerenciador de Infraestrutura*. Em sua versão atual, o `ORBITER` utiliza o Terraform [de Carvalho and de Araújo 2020] para essa função. O Terraform é capaz de implantar (Passo ❷ na Figura 1) o *cluster* de Kubernetes (sistema de orquestração de contêineres) nos três principais provedores de nuvem, *i.e.*, *Amazon Web Service* (AWS), *Google Cloud Platform* (GCP) e Microsoft Azure, a partir de configurações definidas pelo desenvolvedor (Passo ❶ na Figura 1). Além disso, o *Gerenciador de Infraestrutura* permite a configuração da versão do Kubernetes que será utilizada, e a configuração dos nós que compõem o *cluster*. Por fim, o *Gerenciador de Infraestrutura* também provisiona o ArgoCD (`https://argoproj.github.io/cd/`) que é usado para provisionar os demais componentes nas camadas de ingestão/armazenamento e processamento. Na versão atual do `ORBITER`, o Kubernetes é utilizado junto com o gerenciador de pacotes Helm (`https://helm.sh/`). Com o Helm é possível empacotar a aplicação desenvolvida e determinar a ordem em que os componentes são executados no *cluster* de Kubernetes. Cada componente da aplicação é um *Chart* do Helm. Por meio do *Chart* de uma aplicação é que o arcabouço permite que sejam configuradas a quantidade de CPU, memória e disco utilizados.

A Camada de Ingestão/Armazenamento é a responsável por receber dados de fontes externas (Passo ❸ na Figura 1) e armazenar no *cluster* de contêineres criado (Passo ❹ na Figura 1). Além de ser escalável e tolerante à falhas, a camada de ingestão deve carregar os dados no menor tempo possível, possibilitando o processamento em tempo real ou em *batch*. O principal componente dessa cama é o *Gerenciador de Streaming*. Na versão atual do `ORBITER`, o *Gerenciador de Streaming* é implementado por meio do o Apache Kafka, que é uma plataforma distribuída de eventos em tempo real onde os eventos são produzidos nos tópicos por um produtor e consumidos por um ou mais consumidores por meio de APIs. Além disso, usamos o Kafka *Connect*, que cria produtores e consumidores customizados para uma determinada ferramenta ou ambiente, *e.g.*, PostgreSQL, MySQL, S3, *etc*. Para armazenar os dados obtidos, foi criado um *Data Lakehouse* [Behm et al. 2022], que é a união dos conceitos de *Data Lake* e *Data Warehouse*, e sua proposta é prover um repositório de dados em grande volume e em formatos diferentes, ao mesmo tempo que proporciona garantias de transações ACID. Para a implementação do *Data Lakehouse*, a ferramenta escolhida foi o *Delta Lake* (`https://github.com/delta-io`), pois ele se integra com as ferramentas de processamento e exploração de dados. Como sistema de arquivos, escolhemos o Minio (`https://min.io/`), que é um *object storage* multi-nuvem que se comunica via protocolo S3. Ele é compatível com a maioria das soluções de *Big Data* disponíveis.

A Camada de Processamento é responsável por sanitizar e manipular os dados disponíveis no *Data Lakehouse*. Nessa camada, contêineres são configurados pelo componente *Gerenciador de Imagens* para atender múltiplas funções da aplicação que está sendo desenvolvida (Passo ❻ na Figura 1): (i) orquestração de tarefas (*e.g.*, Apache Airflow), processamento distribuído (*e.g.*, Apache Spark), consultas a dados (*e.g.*, Apache Hive) e visualização (*e.g.*, Apache Superset). O *Gerenciador de Imagens* segue o conceito de GitOps [Beetz and Harrer 2022] para acessar o catálogo de imagens (Passo ❺ na Figura 1). A ideia do GitOps é que tenhamos um repositório *Git* contendo as descrições declarativas da infraestrutura e um processo automatizado para fazer o ambiente de produção corresponder ao estado descrito no repositório. Na versão atual do `ORBITER`, o *Geren-*

ciador de Imagens foi implementado sobre o ArgoCD, que é uma ferramenta para GitOps no Kubernetes. A partir do *Gerenciador de Imagens*, os contêineres são instanciados para a execução da aplicação. O código-fonte do ORBITER se encontra disponível em https://github.com/UFFeScience/orbiter.

4. Estudo de Viabilidade

A fim de avaliar a viabilidade do ORBITER, foi elaborado um estudo baseado na execução de uma aplicação que processa um conjunto de dados de acidentes rodoviários disponibilizado pela ANTT[1]. O conjunto de dados em questão apresenta o quantitativo de acidentes por via (*e.g.*, BR 101), por mês e ano. A aplicação carrega esses dados no ORBITER via *proxy* e usa o *Kafka Connect* para realizar a ingestão desses dados no *Data Lakehouse* em formato de arquivo *Parquet* com compressão *Snappy*, e utilizando o conector para o S3. Uma vez que os dados se encontram carregados no *Data Lakehouse*, o serviço do Airflow executa uma tarefa agendada que invoca o Spark. Por sua vez, o Spark transforma os dados em tabelas no formato do *Delta Lake*. Assim, os dados passam a ficar disponíveis para consulta por meio do Trino (um mecanismo de apoio a consultas analíticas distribuídas) e passíveis de serem apresentados em um *dashboard* no *Superset*. Para que a aplicação escolhida para o caso de uso pudesse ser executada via ORBITER, algumas imagens customizadas de contêineres precisaram ser construídas: (i) Uma imagem contendo o Kafka *Connect* e os arquivos JAR dos conectores para o sistema de arquivos e para o S3; (ii) Uma imagem contendo o Airflow e o *provider* do Kubernetes, assim o Airflow pode se comunicar com o *cluster* de Kubernetes e com o Spark; (iii) Uma imagem contendo a aplicação Spark (com o código *PySpark* da aplicação) e a biblioteca *Delta-Spark* (para que a aplicação possa escrever dados no formato do *Delta Lake*); e (iv) Uma imagem contendo o *Superset* para visualização de dados e a biblioteca *sqlalchemy-trino* para que o *Superset* consulte os dados usando o Trino.

Em termos de ambiente de execução, foi escolhido o ambiente de nuvem da Microsoft Azure. Foi configurado um *cluster* de Kubernetes com uma instância *Master* com 2vCPUs e 4GB RAM e uma instância *Worker* (nó comum) com 2vCPUs e 16GB RAM. Cada instância foi configurada com um disco de 30GB, e foi necessária a configuração de um IP público e URLs para acessar os serviços que serão implantados. Uma vez que as configurações e imagens de contêineres foram definidas (é importante ressaltar que tais imagens precisam ser construídas apenas uma vez, mas podem ser utilizadas diversas vezes), o ORBITER pode ser executado. Foram analisados tanto o tempo de execução da aplicação no ORBITER quanto o custo financeiro envolvido na execução. O tempo de execução total da aplicação no ORBITER depende de um *pipeline* composto de três etapas: (i) Implantação do *cluster* de Kubernetes pelo provedor de nuvem, (ii) Implantação de todos os serviços no *cluster* de Kubernetes (*e.g.*, Kafka, Airflow, *etc*) e (iii) Execução da aplicação. A primeira etapa (implantação do *cluster*) apresentou tempo médio de execução de $\bar{x} = 10$ minutos. A segunda etapa (implantação dos serviços no *cluster*) apresentou tempo médio de execução de $\bar{x} = 5$ minutos. E, finalmente, a terceira etapa (execução da aplicação) também apresentou tempo médio de execução de $\bar{x} = 5$ minutos. No total, a execução das três etapas do *pipeline* totalizou aproximadamente 20 minutos. É importante ressaltar que a etapa de implantação do *cluster* de Kubernetes e dos serviços dentro do *cluster* é realizada somente uma vez (a menos que ocorram atualizações nos serviços), *i.e.*, em futuras execuções, o tempo de execução total tende a ser de aproximadamente 5 minutos, o que é aceitável, uma

[1] https://www.gov.br/antt/pt-br

vez que é o tempo de processamento efetivo da aplicação. Em relação ao custo financeiro, a execução do ORBITER com as configurações supracitadas custou aproximadamente R$ 2,00, o que mostra a viabilidade financeira do arcabouço proposto. Apesar dos resultados iniciais se mostrarem promissores, novos experimentos se fazem necessários tanto utilizando aplicações mais complexas, quanto explorando outros provedores de nuvem, como a Amazon AWS e o Google *Cloud Platform*.

5. Conclusões

O modelo de computação *serverless* tem ganho muita relevância nos últimos anos. Apesar de oferecer uma gama de vantagens (*e.g.*, o desenvolvedor não precisa se preocupar com questões de infraestrutura), realizar a implantação de aplicações que processam grandes volumes de dados em uma arquitetura *serverless* pode ainda não ser uma tarefa trivial de ser desempenhada. A necessidade de se gerar imagens de contêineres para realizar a implantação e a falta de portabilidade entre os mecanismos *serverless* dos provedores de nuvem pode ser um desafio. Nesse artigo, apresentamos o ORBITER, que é um arcabouço com o objetivo de implantar de forma automática aplicações que seguem o modelo *serverless* em múltiplos provedores de nuvem. O ORBITER foi avaliado por meio de um estudo de caso que implantou uma aplicação que processa dados de acidentes de trânsito disponibilizados pela ANTT na nuvem da Microsoft Azure. Os resultados se mostraram promissores, mas experimentos com aplicações mais complexas e outros provedores de nuvem ainda se fazem necessários.

Referências

Beetz, F. and Harrer, S. (2022). Gitops: The evolution of devops? *IEEE Softw.*, 39(4):70–75.

Behm, A., Palkar, S., et al. (2022). Photon: A fast query engine for lakehouse systems. SIGMOD '22, page 2326–2339, New York, NY, USA. ACM.

de Carvalho, L. R. and de Araújo, A. P. F. (2020). Performance comparison of terraform and cloudify as multicloud orchestrators. In *CCGRID*, pages 380–389. IEEE.

Hassan, H. B., Barakat, S. A., and Sarhan, Q. I. (2021). Survey on serverless computing. *J. Cloud Comput.*, 10(1):39.

Hellerstein, J. M., Faleiro, J. M., et al. (2019). Serverless computing: One step forward, two steps back. In *CIDR*. www.cidrdb.org.

Mampage, A., Karunasekera, S., and Buyya, R. (2021). A holistic view on resource management in serverless computing environments: Taxonomy, and future directions. *CoRR*, abs/2105.11592.

Nandury, S. V. and Begum, B. A. (2016). Strategies to handle big data for traffic management in smart cities. In *ICACCI 2016, India*, pages 356–364. IEEE.

Perron, M., Fernandez, R. C., DeWitt, D. J., and Madden, S. (2020). Starling: A scalable query engine on cloud functions. In *SIGMOD], June 14-19, 2020*, pages 131–141. ACM.

Sousa, F. (2020). Computação serverless e gerenciamento de dados. In *Anais do XXXV Simpósio Brasileiro de Bancos de Dados*, pages 199–204, Porto Alegre, RS, Brasil. SBC.

Wang, A., Zhang, J., et al. (2020). Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In Noh, S. H. and Welch, B., editors, *USENIX FAST*, pages 267–281. USENIX Association.