INSTITUTO DE GESTÃO E TECNOLOGIA
DA INFORMAÇÃO

Desenho de Arquiteturas de Dados Escaláveis

Neylson Crepalde

2022

Desenho de Arquiteturas de Dados Escaláveis

Bootcamp Engenheiro(a) de Dados Cloud

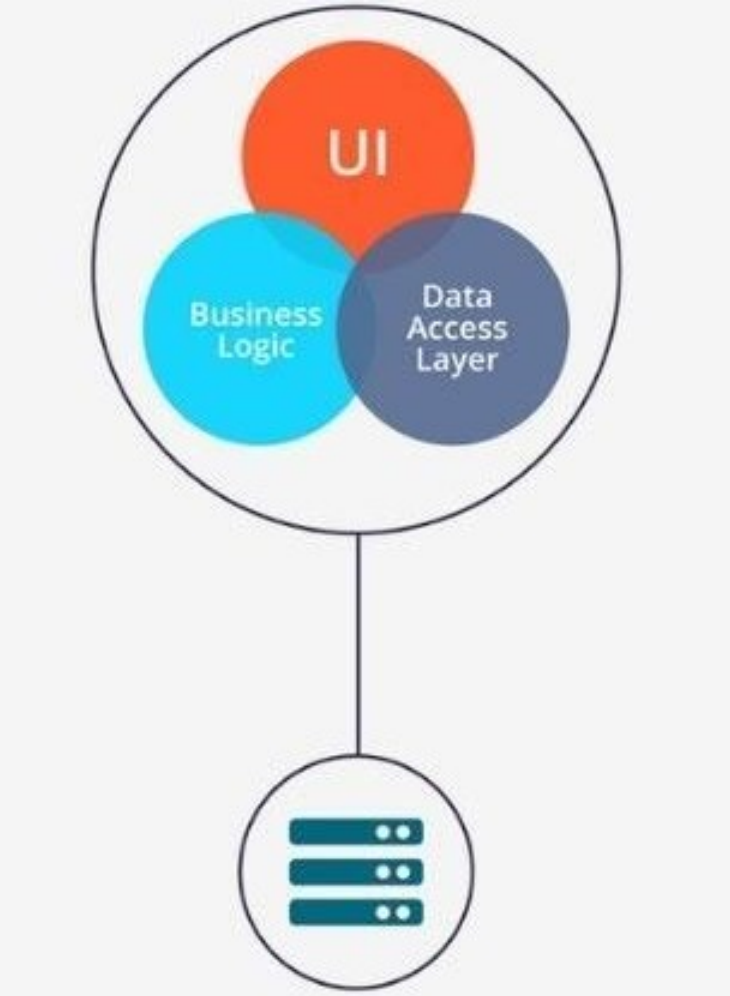
Neylson Crepalde

Sumário

Capítulo 1. Docker

Há alguns anos, as aplicações desenvolvidas pelos times de software seguiam um padrão de arquitetura conhecido como *monolito*. Nesse desenho, todos os componentes da aplicação ficavam juntos em apenas um único artefato conforme a figura 1 abaixo.

Figura 1 – Monolito



Fonte: https://www.everit.com.br/arquitetura-de-micro-servicos/.

Esse tipo de arquitetura de software apresentou, ao longo dos anos, algumas dificuldades aos times de desenvolvimento:

- **Ciclos de releases lentos** – Aplicações monolíticas normalmente tinham ciclos de releases mais lentos devido ao grande volume de componentes a serem verificados.

- Aplicações monolíticas precisam ser testadas nas integrações de todos os seus componentes e esse trabalho exige muito mais tempo quando algum componente recebe atualização. Por esse motivo, elas têm **updates menos frequentes.**

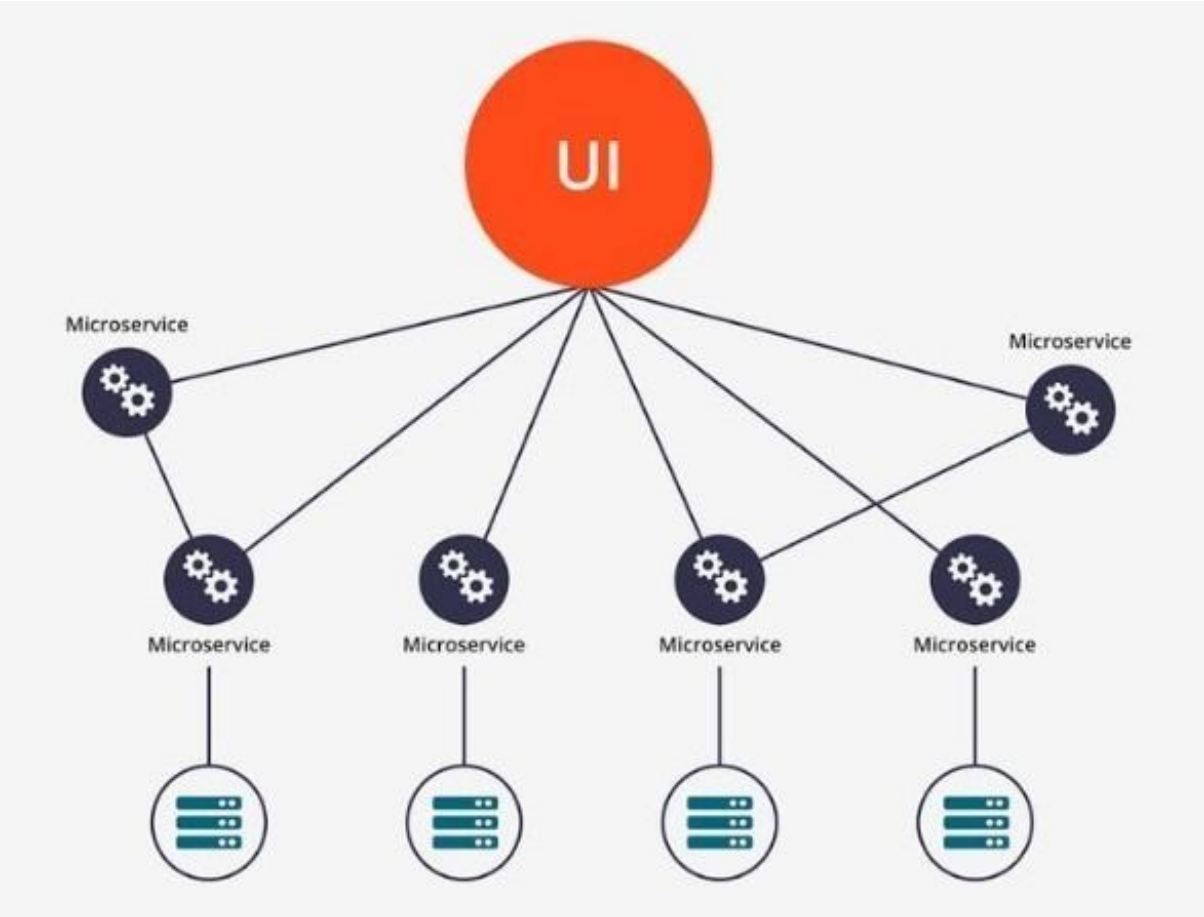
- É necessário realizar o **build do sistema inteiro após o fim de um ciclo atualização**, o que impacta diretamente os dois últimos pontos mencionados.

- Uma aplicação monolítica exige que **os servidores tenham poder computacional suficiente** para suportar todos os workloads da aplicação.

- Aplicações monolíticas normalmente trabalham com **escalonamento vertical** quando há a necessidade de aumento de recursos, isto é, aumentar os recursos disponíveis (RAM, CPU etc.) para um servidor no qual a aplicação está implantada. É bastante difícil implantar escalonamento horizontal (adição de novas máquinas) em aplicações monolíticas.

Gradativamente, o desenho de arquitetura monolítico foi dando lugar a um desenho de **arquitetura de microsserviços**. A arquitetura de microsserviços separa os componentes da aplicação de modo que cada um funciona de maneira independente e autônoma. Os microsserviços, por sua vez, se comunicam através de chamadas de API ou por algum sistema de filas. A figura 2 apresenta uma representação de uma arquitetura de microsserviços.

Figura 2 – Arquitetura de microsserviços.



Fonte: https://www.everit.com.br/arquitetura-de-micro-servicos/.

A arquitetura de microsserviços possui algumas vantagens sobre a arquitetura monolítica, a saber:

- **Ciclos de releases rápidos** – esse desenho de arquitetura permite que partes menores possuam atualização e tenham suas próprias esteiras de implantação gerando mais velocidade no ciclo de releases.

- **Updates pequenos e frequentes** – aplicações em microsserviços podem ter updates menores em seus diferentes componentes e, por consequência, o ciclo de atualização fica mais frequente.

- A arquitetura de microsserviços permite que **cada parte do sistema possa ser atualizada separadamente**, o que tem impacto direto nos dois pontos citados acima, isto é, aumenta a velocidade do ciclo de realeases e a frequência de atualizações.

- Como a arquitetura de microsserviços possui cada componente funcionando de maneira autônoma e separada, precisamos de **recursos computacionais nos servidores apenas para suprir o necessário para cada componente**. Nesse sentido, podemos distribuir a implantação de cada componente em servidores com tamanhos diferentes otimizando a alocação de recursos.

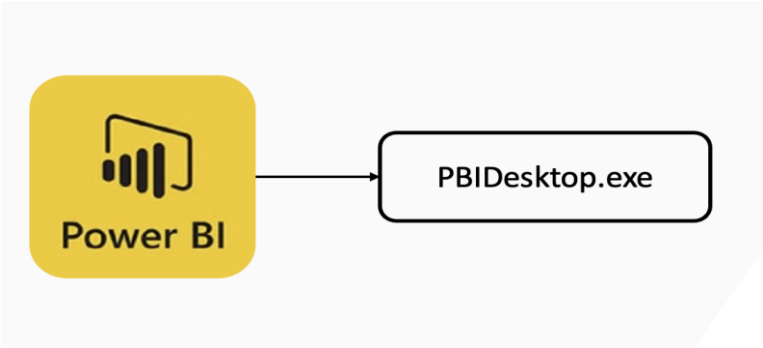
- A arquitetura de microsserviços facilita o **escalonamento horizontal**, isto é, o aumento de recursos computacionais disponíveis para a aplicação por meio da adição de máquinas.

Alguns exemplos

Para citar um exemplo de uma aplicação monolítica, podemos utilizar o **Power BI**. O Power BI é uma das ferramentas mais conhecidas e mais utilizadas no mundo de dados para a visualização e análise de dados estratégica. Sem dúvida, é um líder de mercado.

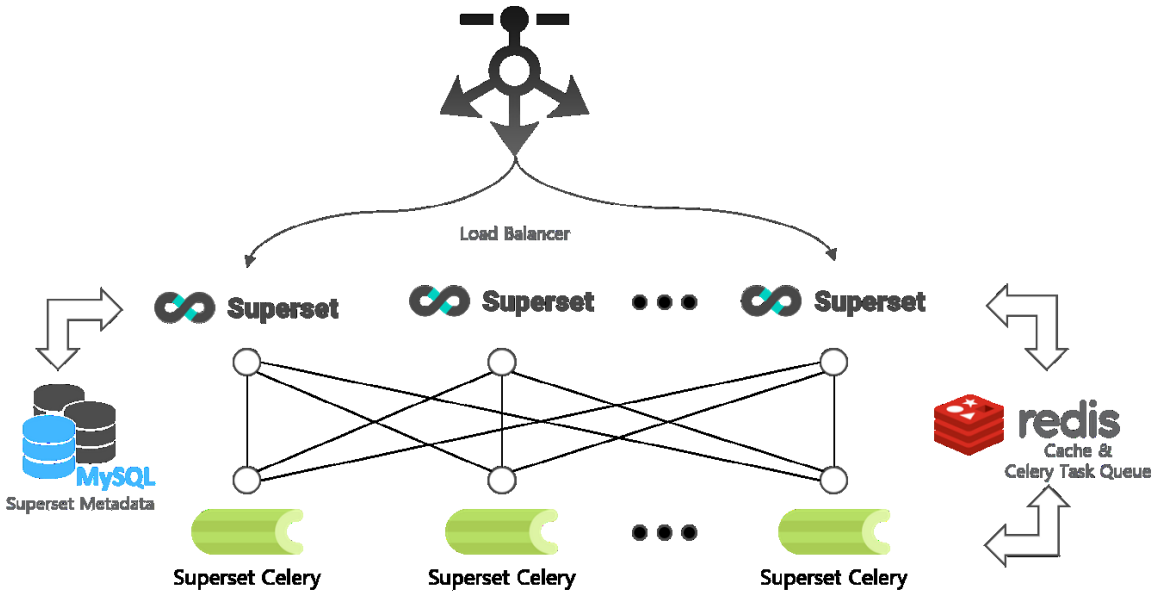
A arquitetura da aplicação possui um desenho monolítico. Todos os componentes da aplicação estão contidos em um único artefato nomeado *PBIDesktop.exe* o qual fica disponível em nossa máquina de trabalho (Fig. 3).

Figura 3 – Power BI.



Já o **Superset**, ferramenta open source de visualização e análise de dados, possui um desenho arquitetural de microsserviços conforme a representação na Figura 4 abaixo.

Figura 4 – Arquitetura do Superset.



Fonte: https://medium.com/@blcksrx/apache-superset-in-detail-b3744033384f.

O Superset possui quatro componentes principais:

- A **Database de metadados** que é responsável pelo armazenamento dos dados da aplicação.

- A **Interface web** com a qual os usuários vão interagir.

- O serviço de mensageria **Celery**, por sua vez, composto por:

  – O **Celery Broker**, normalmente um banco de dados Redis e

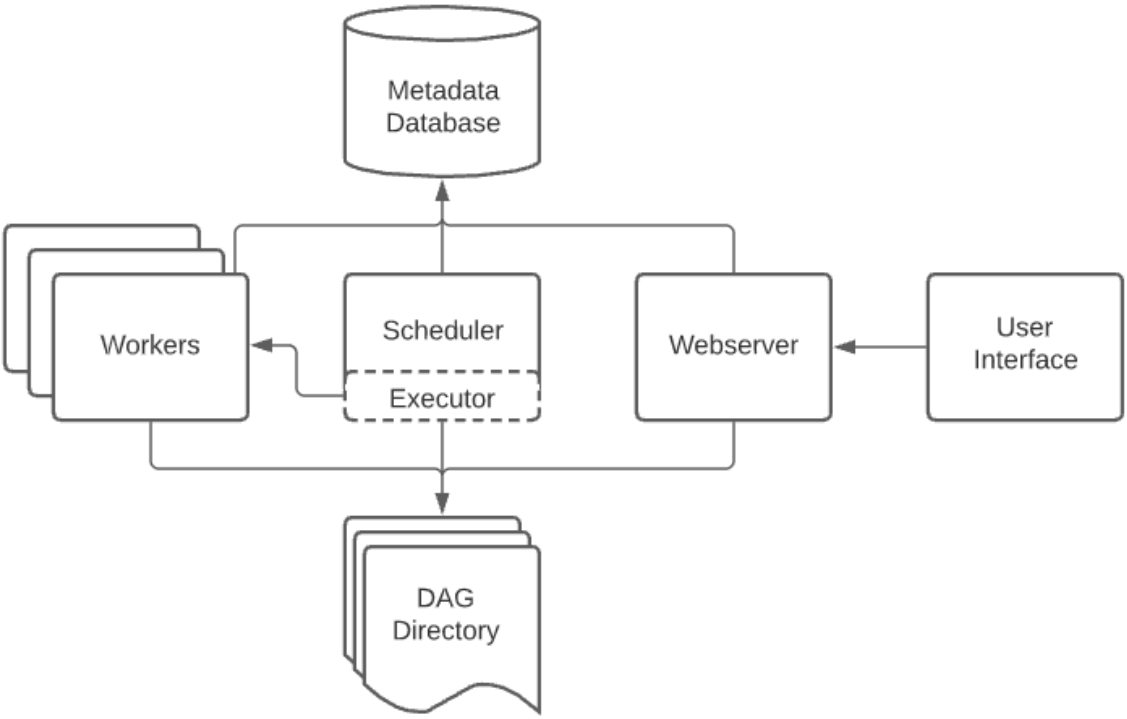
  – Os **Celery Workers**, instâncias onde a computação das tarefas será de fato executada.

Além das funções citadas acima, o Redis, no caso específico do Superset, possui uma função extra, o de Query Caching, ou seja, ele faz o armazenamento *in memoria* de algumas consultas recorrentes realizadas pelos usuários.

Perceba que para atualizar um único componente do Power BI, todo o executável *PBIDestktop.exe* deve ser atualizado e novamente instalado na máquina. Por outro lado, é possível atualizar qualquer componente do Superset em qualquer momento e todo o sistema continuará funcionando normalmente. Se algum componente do PowerBI apresentar falha, todo o sistema deve ser atualizado e reinstalado no servidor. Se algum componente do Superset apresentar falha, ele pode facilmente ser substituído (às vezes de maneira automática, dependendo do tipo de implantação) garantindo a confiabilidade do sistema.

Outro exemplo de aplicação baseada numa arquitetura de microsserviços é o – também bastante conhecido – Apache Airflow (Fig. 5).

Figura 5 – Arquitetura do Apache Airflow



Fonte: https://airflow.apache.org/docs/apache-airflow/stable/concepts/overview.html.

O Apache Airflow, assim como o Superset, possui cinco componentes, a saber:

- Database de metadados, responsável pelo armazenamento dos dados da aplicação.
- O Webeserver com o qual o usuário vai interagir para o controle de DAGs (pipelines do Airflow).
- O Scheduler, responsável pela organização e distribuição das execuções das tarefas no Airflow.
- Os Workers, instâncias onde a computação das tarefas acontece de fato e

- O Diretório de DAGs, uma pasta de sistema, volume ou repositório git onde os códigos das DAGs serão armazenados.

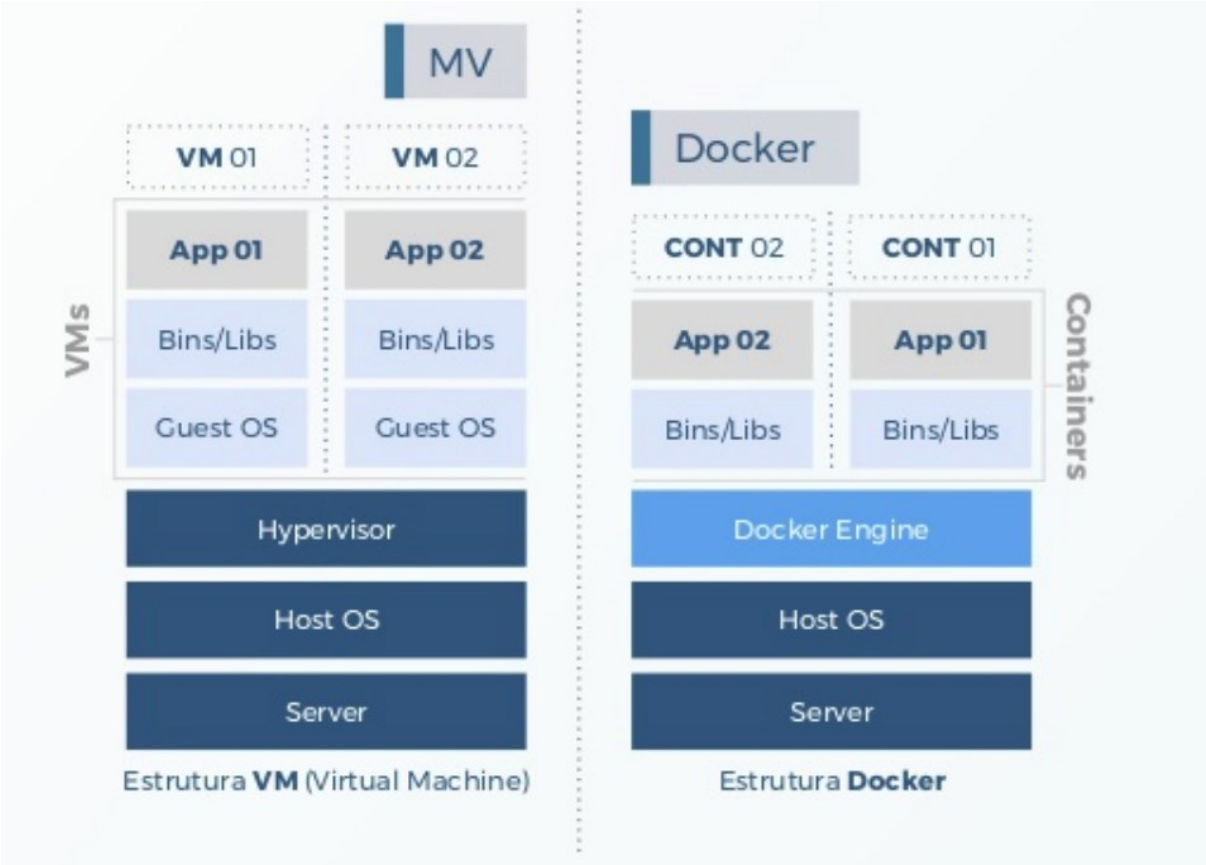
No gerenciamento de pipelines, há alguns cenários possíveis relacionados ao escalonamento de recursos. No caso de haver poucos usuários do *webserver* e uma carga grande processamento relacionada às tasks, é possível escalar apenas os workers de execução para dar a cada tarefa os recursos necessários. No caso de haver tasks com pouca necessidade de poder de processamento e muitos usuários simultâneos, podemos escalar apenas os *webservers*. No caso de haver DAGs com um alto grau de complexidade e paralelização, por exemplo, é possível escalar o scheduler. A alocação de recursos para cada um desses componentes é totalmente modificável de maneira dinâmica de acordo com o tipo de consumo e funcionamento dos pipelines.

Virtualização clássica e containers

Comparados às clássicas máquinas virtuais, os containers são mais leves permitindo que vários componentes da aplicação possam ser executados no mesmo hardware.

Nas máquinas virtuais, tudo o que é necessário para a execução de uma aplicação precisa estar contido lá, isto é, sistema operacional, bibliotecas de sistema, linguagens de programação etc. Já os containers possuem uma estrutura em camadas contendo as diversas etapas de configuração do ambiente. A figura 6 abaixo ilustra essa diferença.

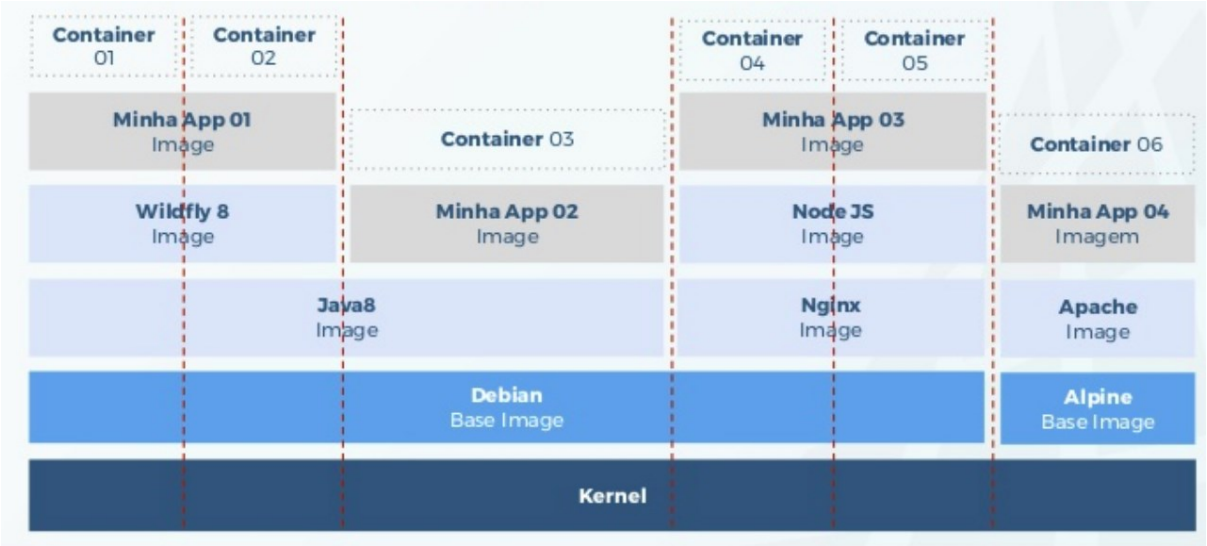
Figura 6 – VMs vs Containers.



Fonte: https://pt.slideshare.net/AndrJusti/treinamento-docker-bsico.

Na definição de um container, cada etapa do processo é registrada como uma camada que pode ser reaproveitada por outros containers no mesmo ambiente. A figura 7 abaixo ilustra o esquema de camadas de containers.

Figura 7 – Camadas de Containers.



Fonte: https://pt.slideshare.net/AndrJusti/treinamento-docker-bsico.

Dockerfile

O Dockerfile é o código que define as camadas de composição de uma imagem de container. A figura 8 abaixo possui uma ilustração de um Dockerfile simples para build de uma imagem com um processo Python.

Figura 8 – Exemplo de um Dockerfile.

```
1   FROM alpine
2
3   RUN apk add --no-cache python3 py3-pip
4
5   RUN pip3 install fastapi uvicorn
6
7   EXPOSE 80
8
9   COPY ./app /app
10
11  CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

Na Figura 8, a primeira linha faz a importação de uma imagem do sistema operacional alpine, uma distribuição Linux bastante enxuta para aplicação com configurações mínimas. A linha 3 faz a instalação do python e do package manager pip. A linha 5 faz a instalação de duas bibliotecas python, *fastapi* e *uvicorn*. A linha 7 abre porta 80 do container para receber requisições. A linha 9 copia todo o conteúdo da pasta local ./app para a pasta /app dentro do container e a linha 11 declara o comando que será executado sempre que o container for iniciado.

Capítulo 2. Kubernetes

À medida que o número de componentes das aplicações a serem implantadas aumenta, seu gerenciamento fica cada vez mais difícil. Aplicações modernas podem ter dezenas de componentes os quais, por sua vez, podem ter dezenas de réplicas dependendo da demanda pelo seu funcionamento

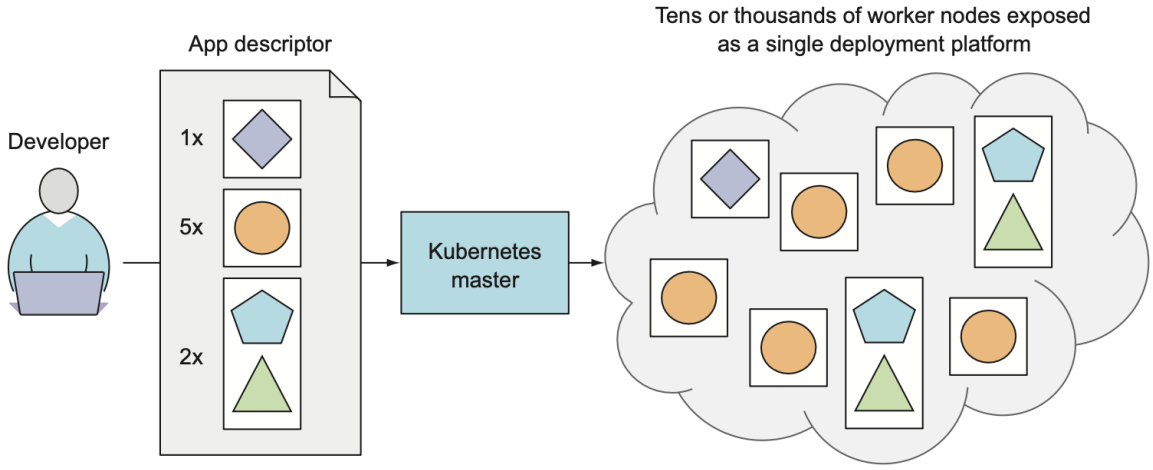
É fácil atingir um ponto em que a gestão dos componentes fica inviável. Gerenciar 100 containers, 200 containers, já é bastante trabalho. Na hipótese de algum sistema sofrer atualização, imagine o trabalho de atualizar manualmente 200 containers...

O Google talvez tenha sido a primeira organização a atentar para a necessidade de um melhor modo de implantação e gerenciamento desses componentes para uma escalabilidade global.

Na década de 1990, o Google trabalhava em um sistema privado chamado *Borg,* que tinha o objetivo de realizar o gerenciamento de componentes de aplicação de maneira mais automatizada e eficiente. O projeto Borg, alguns anos mais tarde, seria substituído pelo projeto Omega, até que em 2014 o Google anuncia a tecnologia do Kubernetes *open source* para toda a comunidade. Em 2015, a versão 1.0 do Kubernetes foi publicada (Luksa, 2018).

A Figura 9 ilustra de maneira simplificada o processo de atuação de um desenvolvedor para a implantação de suas aplicações utilizando Kubernetes.

Figura 9 – Atuação com Kubernetes



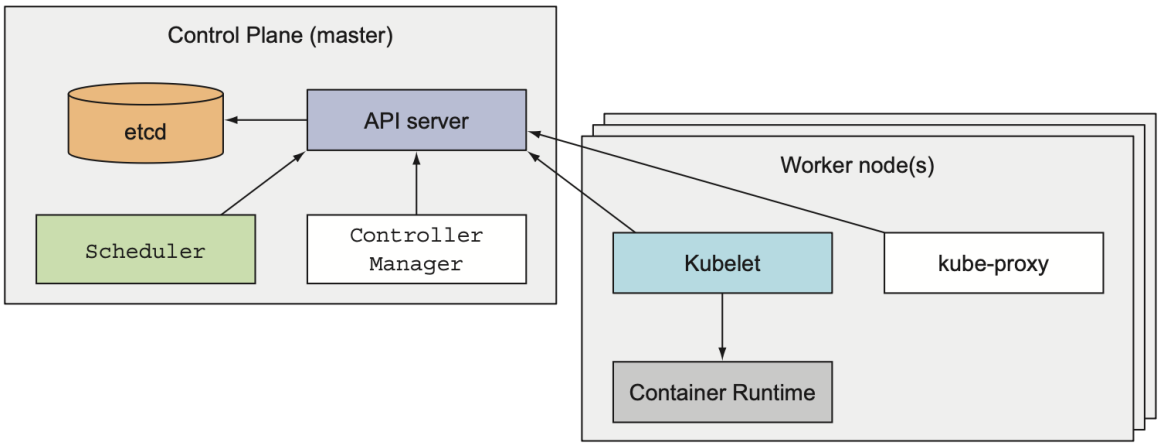
Fonte: Luksa, 2018

A figura mostra que, a partir de um código contendo a descrição geral do funcionamento de uma aplicação e do comportamento esperado de cada um de seus componentes, inclusive quantas réplicas cada componente deve ter, o nó *máster* do Kubernetes faz a implantação de maneira automática e otimizada dos componentes no cluster disponível.

Arquitetura do Kubernetes

A Figura 10 abaixo mostra a arquitetura de hardware e software base do Kubernetes.

Figura 10 – Arquitetura Base do Kubernetes.



Fonte: Luksa, 2018.

A arquitetura base do Kubernetes é composta por 1 nó *máster* e nós *workers.* O nó *master* é responsável pelo gerenciamento do cluster enquanto os nós *workers* são o local onde os componentes das aplicações são de fato implantados e onde toda a carga de computação é realizada no cluster.

Dentro do nó *master*, alguns elementos são necessários:

- API server – O servidor da API do Kubernetes. É com este componente que vamos nos comunicar para fazer requisições ou definir “manifestos” de deploy e ele se comunicará com os demais nós e componentes do Kubernetes.
- Controller Manager – O controlador responsável pelas operações a nível de cluster.
- Scheduler – o componente que toma as decisões sobre em que nó serão alocados cada componente das aplicações.

- *etcd* – um storage do tipo chave-valor para armazenamento de metadados relacionados ao cluster. Nenhum dado de aplicação é armazenado aqui, apenas metadados do cluster.

Nos nós *workers* estes elementos são necessários:

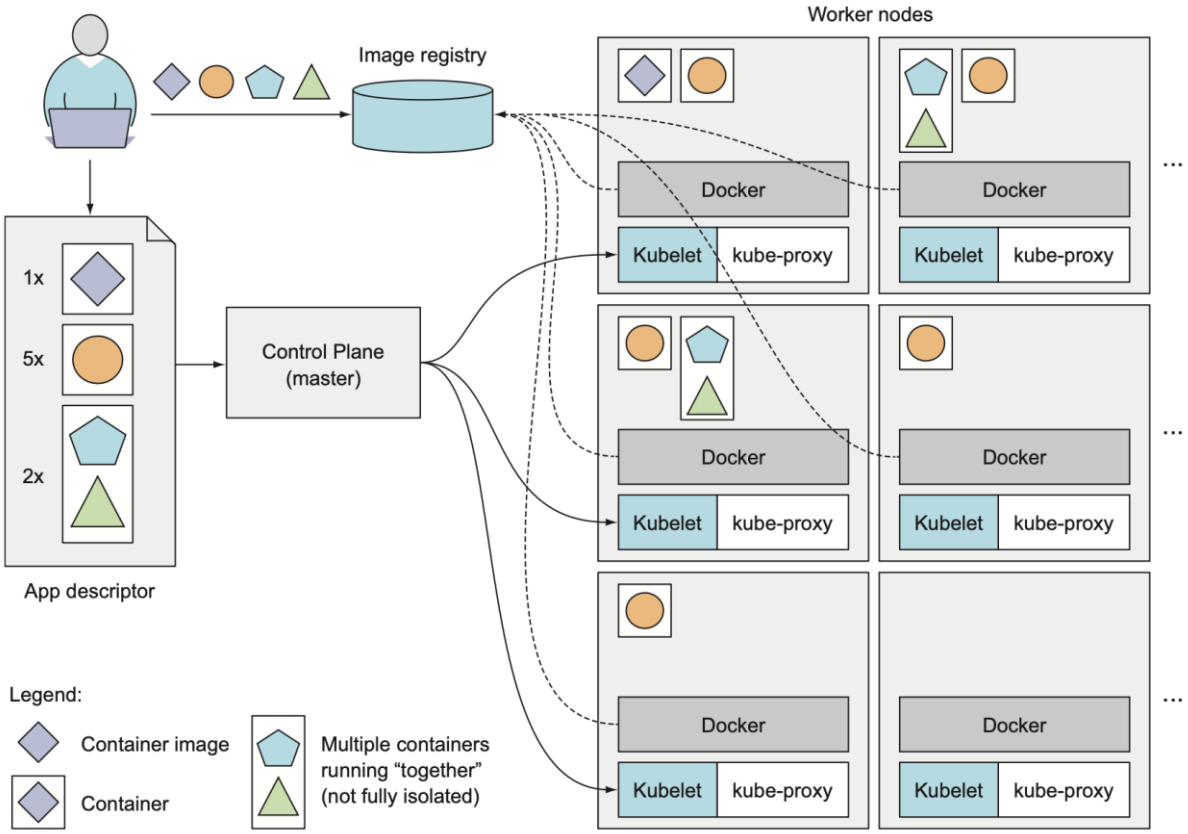
- Kubelet – componente do Kubernetes responsável pela comunicação do nó *worker* com o *master*. Deve ser instalado em todas as instâncias de nós *worker* no cluster.

- Container runtime – a engine de execução dos containers no cluster Kubernetes.

- Kube-proxy – componente responsável pela rede interna do Kubernetes.

A Figura 11 abaixo ilustra, de maneira mais fidedigna, o funcionamento da implantação de uma aplicação utilizando a estrutura do Kubernetes.

Figura 11 – Funcionamento do Kubernetes.



Fonte: Luksa, 2018

Ao elaborar um código contendo os parâmetros e definições de funcionamento de uma determinada aplicação, o modo de deploy de todos os seus componentes, número de réplicas a serem implantadas para cada componente, *claims* de volumes persistentes, serviços etc. (o *manifesto*), o nó *master* toma as decisões sobre a alocação de recursos para os componentes mais otimizada quanto possível e inicia o processo de implantação. Nesse momento, o Scheduler toma a decisão de quais componentes serão implantados em quais nós levando em conta recursos necessários, dependências entre componentes, número de réplicas etc. A API server se comunica com o kubelet instalado em cada um dos workers e, em seguida, as imagens necessárias para os respectivos

componentes são baixadas nas máquinas. Então inicia-se o processo de deploy de cada componente e disponibilização da aplicação para consumo.

Por que utilizar Kubernetes

O uso do Kubernetes traz algumas vantagens.

- Um processo de deploy com alto grau de automatização e simplificado em comparação com o deploy tradicional feito manualmente;
- Configuração do ambiente da aplicação totalmente automatizado;
- Melhor utilização dos recursos de hardware disponíveis através da otimização realizada pelo próprio Kubernetes;
- *Health-checking*, isto é, verificação da saúde das aplicações automatizada;
- Recuperação automatizada de componentes em caso de falha;
- Escalonamento automatizado;
- Desenvolvimento e implantação de aplicações simplificado com foco mais voltado para a aplicação em si e seu código e menos voltado para a problemas de gerenciamento de infraestrutura.

Para um breve tutorial dos principais comandos de kubernetes utilizando a CLI *kubectl*, consulte a documentação oficial do Kubernetes em https://kubernetes.io/pt-br/docs/home/.

Capítulo 3. Tecnologias de Engenharia de Dados em Kubernetes

Alguns dos softwares mais utilizados hoje pelos times de dados já estão preparados para serem utilizados numa arquitetura baseada em Kubernetes. Em sua maioria, eles já possuem um deploy encapsulado para facilitar a sua implantação no Kubernetes utilizando o package manager *helm* (https://helm.sh/).

O *helm* trabalha com o conceito de *Chart*. Um *chart* consiste em um conjunto de manifestos desenvolvidos como templates e um arquivos normalmente chamado *values.yaml* definindo alguns parâmetros para o deploy da aplicação.

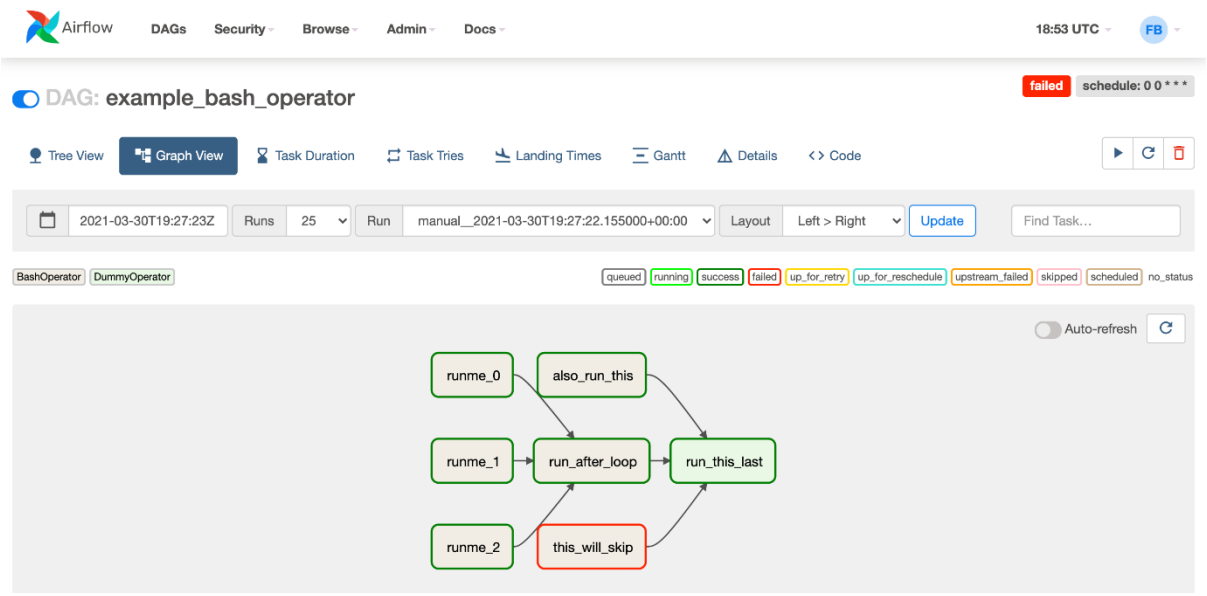
Para realizar a instalação de um software utilizando seu helm Chart, são necessários apenas alguns passos, a saber:

- A instalação do chart localmente com o comando helm repo add <nome_do_chart> <url_do_chart>.
- A customização do arquivo values.yaml.
- A instalação da aplicação com o comando helm install <nome_do_release> <nome_do_repo_do_helm_chart> -f <custom_values.yaml> -n <namespace>.

Esses passos são demonstrados nos vídeos para a implantação de ferramentas como o Airflow, Argo CD e o Spark Operator.

Para ter acesso aos códigos de implantação referenciados nas aulas, visite o repositório https://github.com/neylsoncrepalde/edc_mod4_exercise_igti/tree/dev.

Figura 12 – Webserver do Airflow.



Fonte: Documentação oficial do Airflow.

O deploy do Airflow no Kubernetes pode ser realizado utilizando seu *helm chart* oficial (https://airflow.apache.org/docs/helm-chart/stable/index.html). Para a customização do arquivo de *values*, alguns pontos devem ser levados em conta:

- O repositório default do Airflow a ser utilizado (normalmente *apache/airflow* quando não utilizamos alguma imagem customizada);
- A tag da imagem a ser utilizada;
- A versão do Airflow a ser utilizada. Observe que esta informação não se confunde com a informação anterior. Esta se reserva a indicar algumas "decisões" que serão tomadas pelo Chart de acordo com a versão do Airflow a ser implantada;

- O tipo de *Executor* a ser utilizado (os mais comuns são o CeleryExecutor e o KubernetesExecutor. Algumas diferenças entre eles e possibilidades de configuração são tratadas nesta live do Meetup de Engenharia de Dados – https://www.youtube.com/watch?v=1SdtlobcWOs&t=52s);

- Algumas variáveis de ambiente para uma fácil alteração do arquivo de configuração do Airflow;

- Metadados de conexão do postgres para que o Airflow consiga acessá-lo (válido tanto para PODs de postgres no Kubernetes quanto para databases externas);

- Os dados do usuário default da interface do Airflow;

- O tipo de serviço da interface do usuário (ClusterIP, LoadBalancer etc.);

- A utilização (ou não, dependendo do Executor escolhido) de um POD contendo uma instância Redis para o Celery Message Queue;

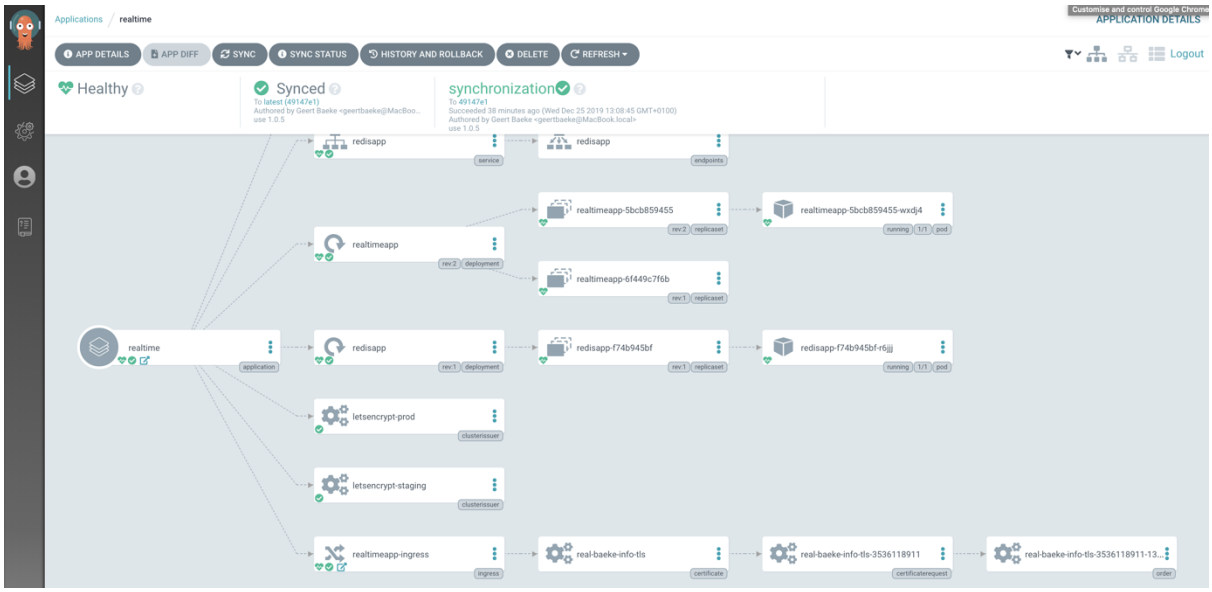
- A utilização de uma instância postgres como database de metadados dentro do Kubernetes ou externa;

- O método de sincronização de Dags utilizando um volume persistente ou (preferencialmente) um processo GitSync (responsável por sincronizar os códigos de dados de um repositório Git com a instância do Airflow;

- Credenciais de acesso ao repositório Git para sincronização de DAGs através de um Kubernetes Secret;

- Armazenamento de logs de execução.

Figura 13 – Interface do Argo CD.

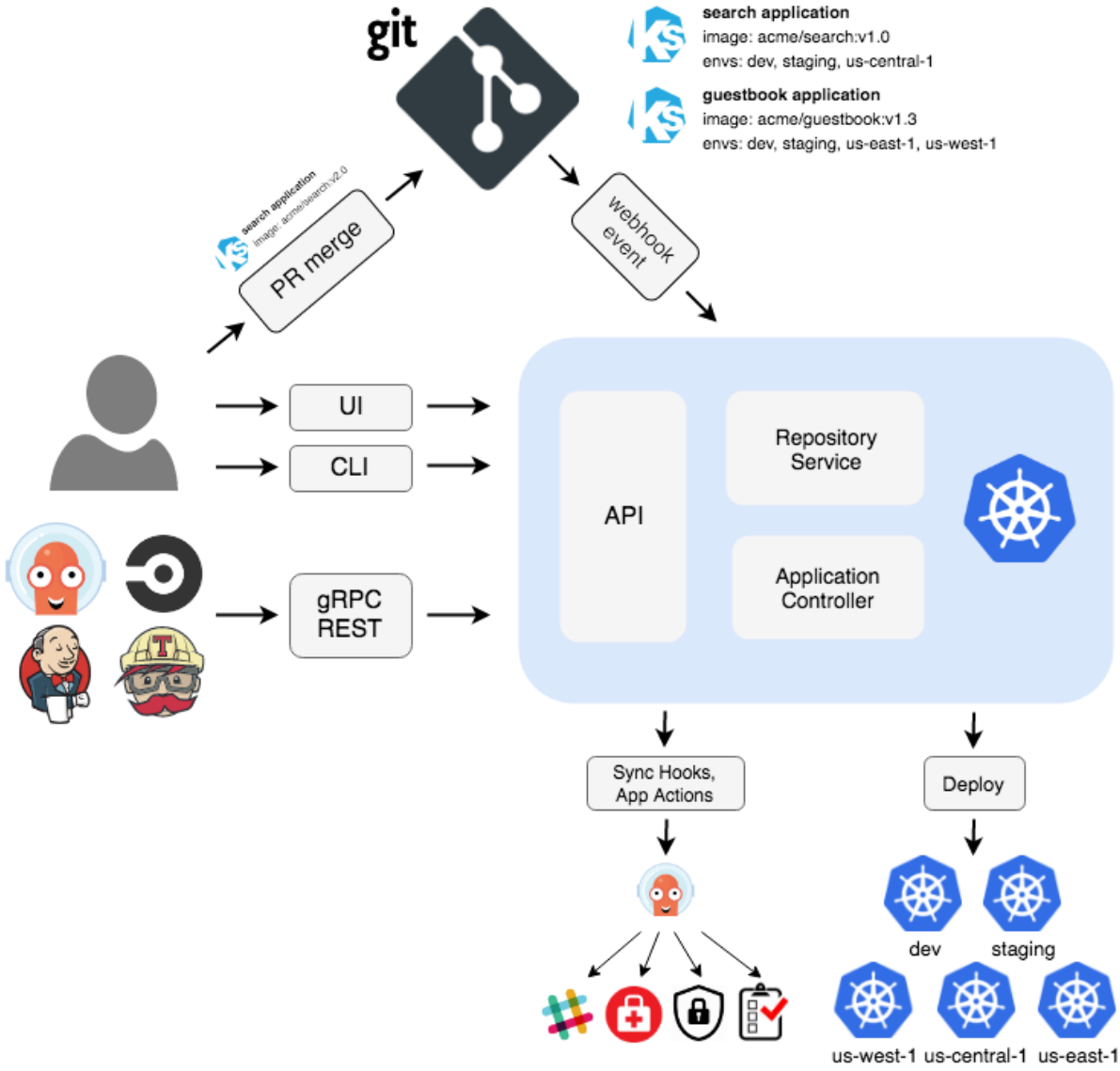


Fonte: https://blog.baeke.info/2019/12/25/giving-argo-cd-a-spin/.

O Argo CD é uma ferramenta para controle de implantação e versionamento de aplicações em geral no Kubernetes. É uma ferramenta ideal para um ambiente mais controlado ou implantações em ambiente de produção.

Sua arquitetura faz a integração de repositórios git ou repositórios helm com sua API para o controle de versão das aplicações a serem implantadas. A integração pode ser configurada tanto de uma CLI quanto da interface do usuário da aplicação.

Figura 14 – Arquitetura do Argo CD.



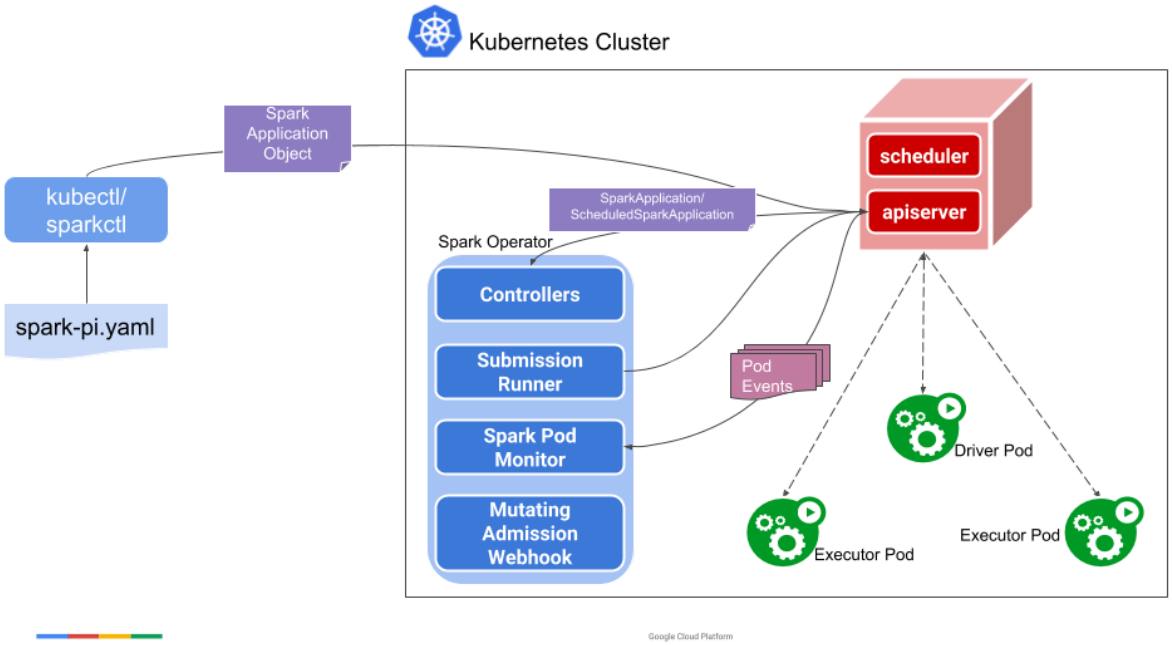
Fonte: https://blog.baeke.info/2019/12/25/giving-argo-cd-a-spin/.

Instruções de implantação do Argo CD e configuração de integrações podem ser acessadas em https://argoproj.github.io/argo-cd/.

Sobre o Spark no Kubernetes

É possível utilizar a tecnologia do Spark no Kubernetes com a ajuda do Spark Operator. Um *operator* é uma estrutura que usamos para criar tipos de recursos customizados no Kubernetes. Com o auxílio do Spark Operator, podemos subir um recurso do tipo *SparkApplication* que subirá no ambiente do Kubernetes um cluster efêmero para a execução de um Spark Job.

Figura 15 – Arquitetura do Spark Operator



Fonte: https://dzlab.github.io/ml/2020/07/14/spark-kubernetes/.

Para isso, é necessário garantir que:

- O Spark Operator esteja devidamente instalado no cluster;

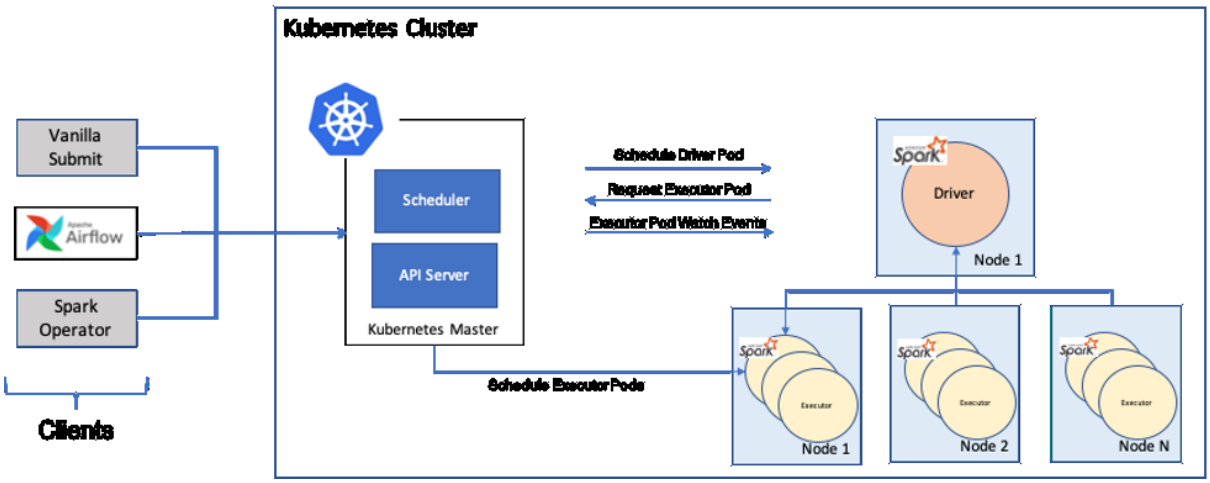
- Uma imagem com o conector do Spark Operator esteja disponível para a execução dos Spark Jobs (uma imagem padrão pode ser encontrada em <gcr.io/spark-operator/spark:v3.0.0>);
- Um serviceaccount do Kubernetes;
- Permissões adequadas outorgadas através de um clusterrolebinding;
- Um código Spark a ser executado em um repositório externo;
- Um manifesto de definição de como o Job Spark será implantado no cluster.

Para mais informações e um guia completo de utilização do Spark Operator, acesse https://github.com/GoogleCloudPlatform/spark-on-k8s-operator.

Capítulo 4. Pipelines de Dados Kubernetes Based

Para a implementação de um exemplo de pipeline de dados baseado em infraestrutura Kubernetes, vamos utilizar o Apache Airflow e o Spark Operator. Na figura 16 abaixo, uma representação do funcionamento da solução em questão.

Figura 16 – Arquitetura do Airflow + SparkKubernetesOperator



Fonte: https://aws.amazon.com/pt/blogs/compute/running-cost-optimized-spark-workloads-on-kubernetes-using-ec2-spot-instances/.

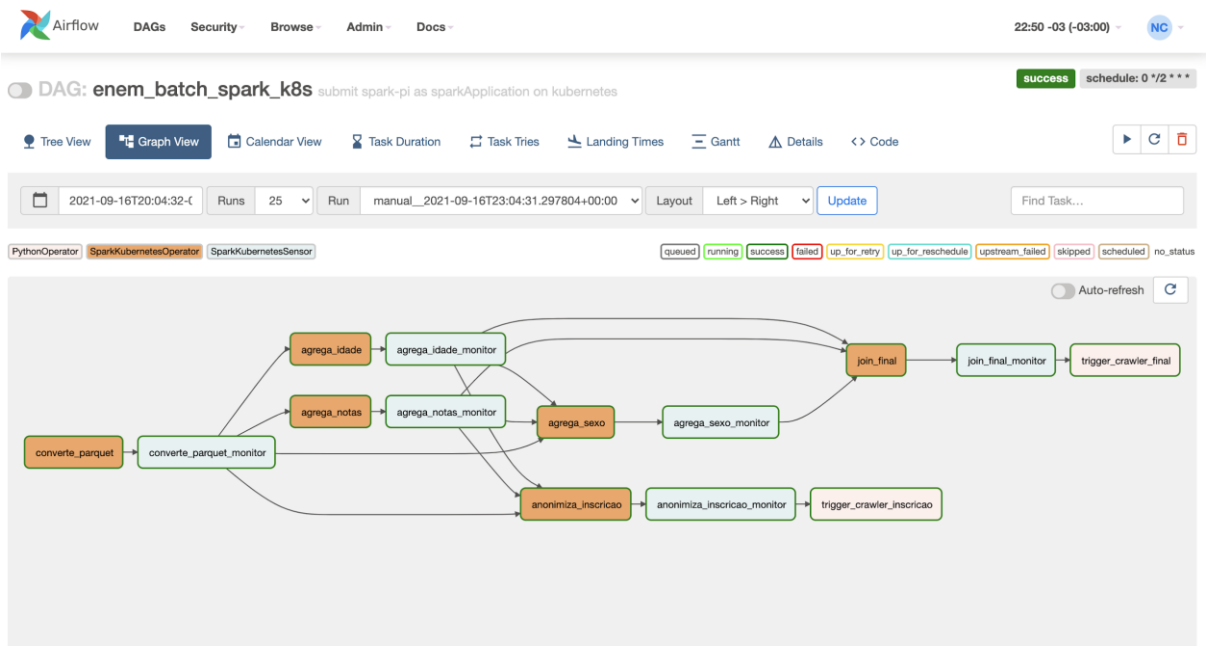
No exemplo da figura 16, as tasks do Airflow que utilizam o SparkKubernetesOperator criam um job do tipo *SparkApplication*. Esse job, uma vez criado, iniciará a criação de um cluster Spark efêmero dentro da infraestrutura do Kubernetes. Um POD driver (equivalente ao nó master do cluster Spark) vai ser alocado e PODs *executors* (similares aos workers nodes de um cluster Spark) serão posteriormente implantados na estrutura.

Podemos acompanhar a execução do job utilizando comandos *kubectl* conhecidos bem como o monitoramento de pods no namespace designado.

No caso de falha ou havendo a necessidade de algum tipo de *debbuging*, podemos acessar os logs de execução do job spark utilizando o comando *kubectl logs* no POD driver do cluster Spark. Por padrão, o POD driver permanecerá no cluster Kubernetes, embora inativo, e com uma marcação de job *completo*. Caso não desejemos, por algum motivo, que o POD não permaneça na infraestrutura do Kubernetes, será necessário realizar a deleção do job Spark criado.

A figura 17 abaixo ilustra o exemplo de pipeline implementado para a aula prática.

Figura 17 – DAG com SparkKubernetesOperator.



O Airflow é responsável pelo *scheduling* de tasks utilizando o *SparkKubernetesOperator* (operador do Airflow). Este operador dispara a execução de uma *sparkapplication* a partir de um yaml de configuração e um código Spark disponibilizado em algum storage externo (em nosso caso, o S3). Além disso, o SparkKubernetesSensor será utilizado para monitorar a execução do job e avanço nas

tasks da pipeline. PythonOperators serão utilizados para disparar a execução de Glue Crawlers para disponibilização dos dados para consulta no AWS Athena.

Referências

ARMBRUST, Michael *et al*. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. **11th Annual Conference on innovative Data Systems Research - CIDR**, Jan. 2021

LUKSA, Marko. **Kubernetes in Action.** Manning, 2018.