Data Science Platform @ PlayStation & Apache Airflow

Siraj Malik
Hamed Saljooghinejad

Who We Are

Siraj Malik

Engineering Manager - Data Science Platform

Hamed Saljooghinejad

Sr ML Platform Engineer

@hamedhsn

Agenda

- Our Team
- History of Airflow at PlayStation
- Migration to Kubernetes
- Development to Production Lifecycle
  - Custom Tooling
  - CI/CD Pipelines
- Containerization
  - Kubernetes Job (K8sOperator)
  - Spark on Kubernetes (SparK8sOperator)
- Questions

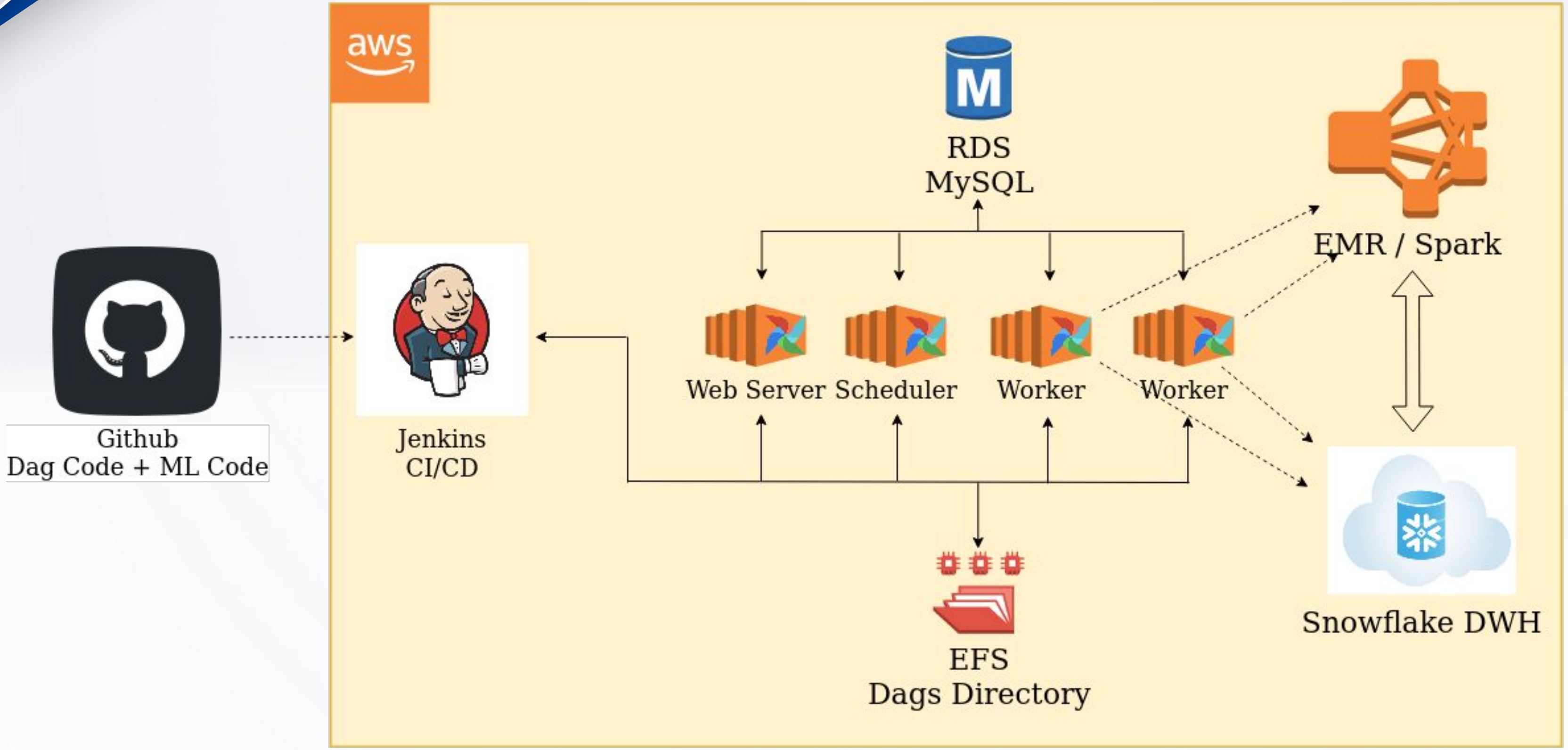
Our Team

- Data Strategy and Operations Organization
    - Centralize Data Warehousing across PlayStation
    - Data Engineering / ETL
    - Analytics Engineering
    - Systems and Infrastructure Engineering
    - Data Governance and Data Literacy
    - Data Science
        - Data Science Platform

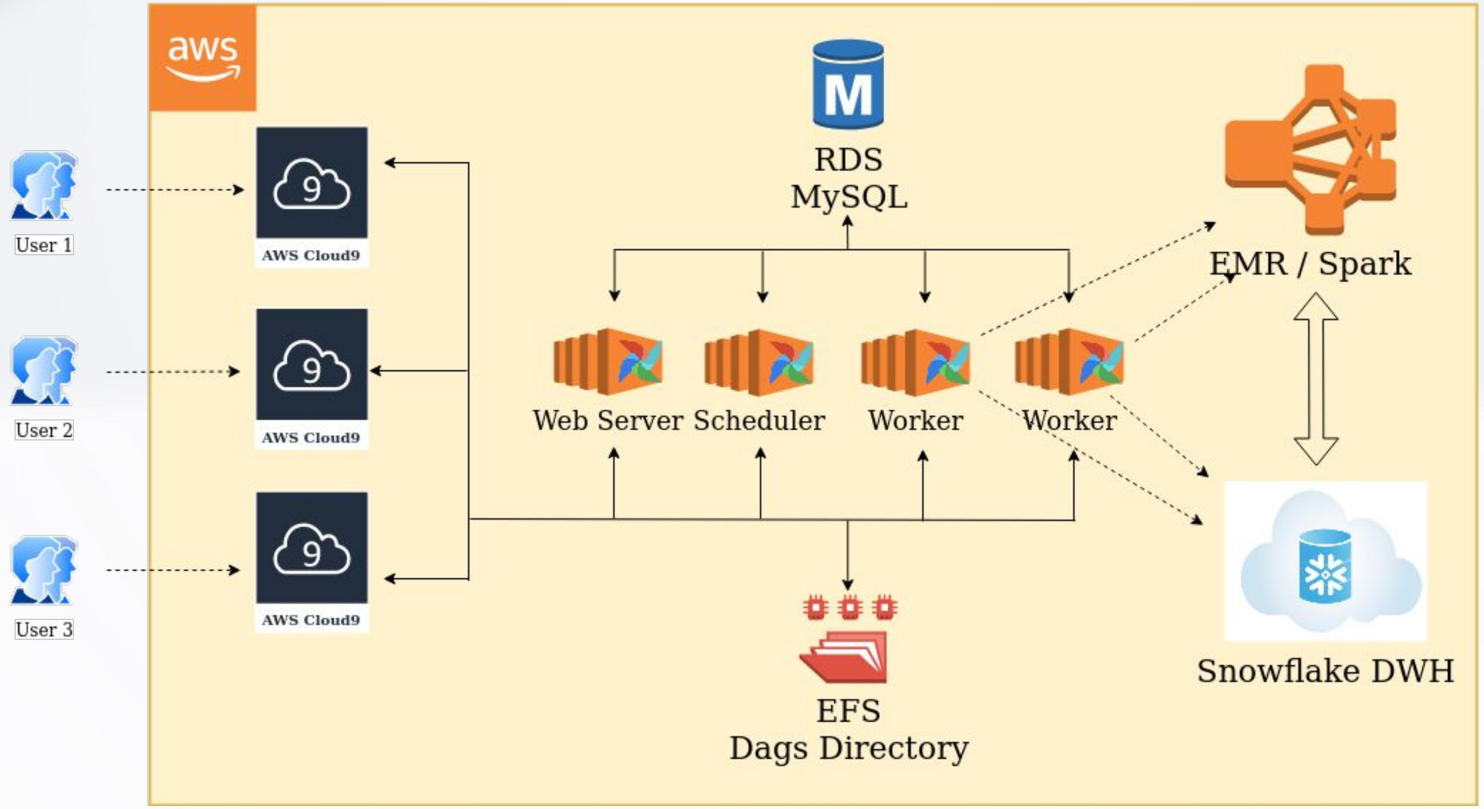
History of Airflow At PlayStation

- Tech refresh program circa 2017
  - Migration from on premise to AWS Cloud
  - Replacement for existing scheduling solution
  - Jenkins + custom internal Python app
- Cross organization Airflow initiative
  - Internal fork of Airflow
  - Internal Docker image
  - Terraform Airflow deployment

Original EC2 Based QA and Prod Setup



Original EC2 Based Dev Setup



Limitations of Original EC2 Based Setup

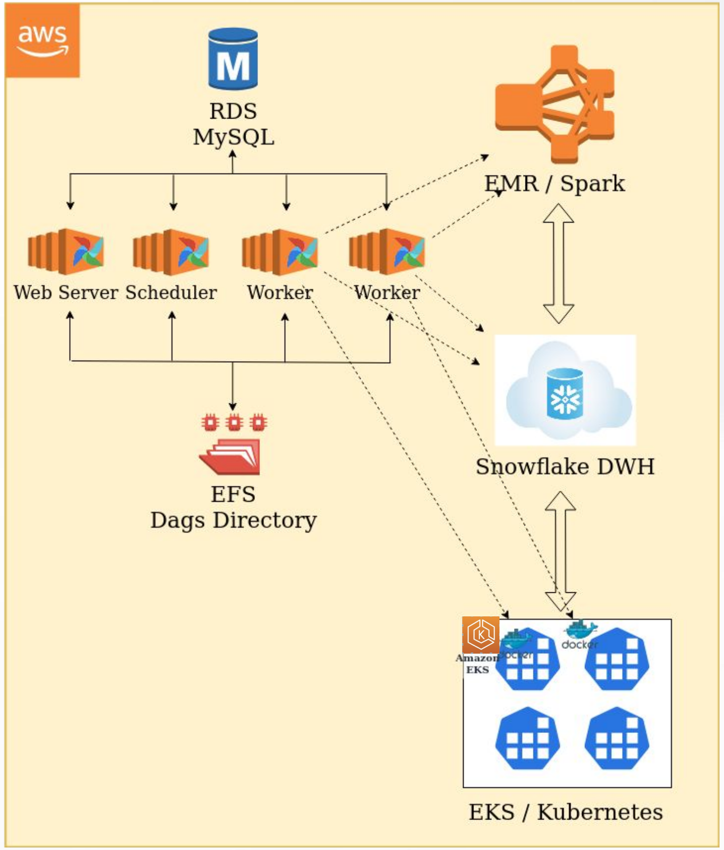
- For Data Scientists
  - All ML model training was done on EMR using Spark
  - Workloads all ran on a single instance type
  - Dev environment Airflow shared by all Data Scientists
  - Not everyone was happy to use Cloud9 for dag development
- For Data Science Platform Engineers
  - Painful to manage (forked Airflow, custom internal Docker image, Terraform, Cloud9, bad CI/CD, etc.)
  - Didn’t scale well, auto scaling groups needed manual intervention
  - Python package/library management
  - Harder to deploy new tools for Data Scientists to use

Migration to Kubernetes

- For Data Scientists
  - Able to select their own libraries (scikit-learn, xgboost, fbprophet, etc.)
  - Can choose underlying hardware to run workloads on
  - Everyone gets their own Airflow dev environment
  - Data Scientists can choose their own IDE
- For Data Science Platform Engineers
  - Easy deployment, separation of application from infrastructure
  - Scales much better, leave scaling up to Kubernetes
  - Package/library dependency management left to Data Scientists in their Dockerfile/requirements/poetry files
  - Easily deploy or update tooling for Data Scientists

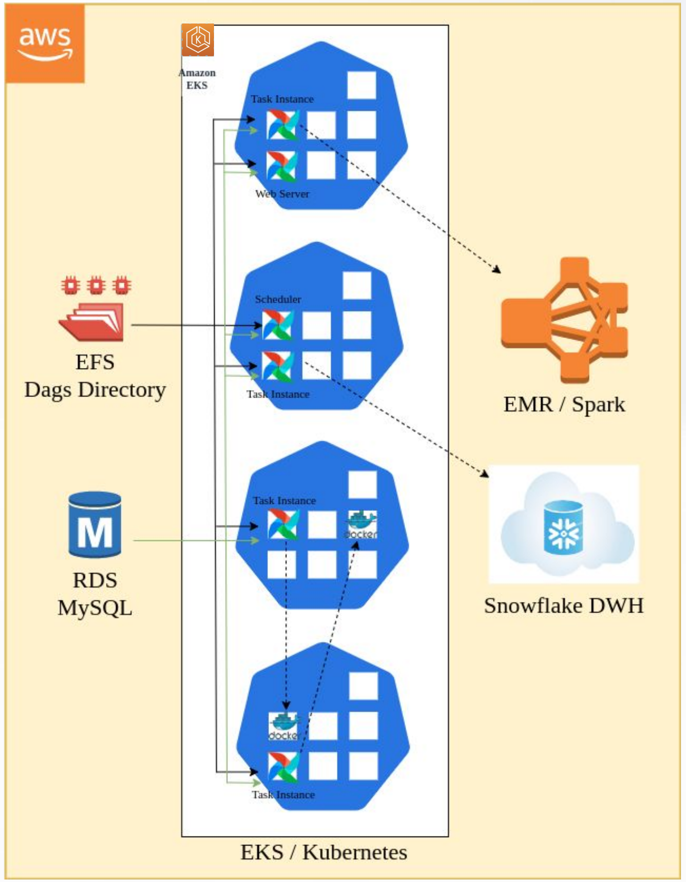
Migration to Kubernetes - Phase 1

- Still using Airflow 1.10.10
- Airflow still running on EC2 instances using CeleryExecutor
- Added a Kubernetes cluster, into which we deploy containerized ML jobs
- Started utilizing Airflow Kubernetes Pod Operator



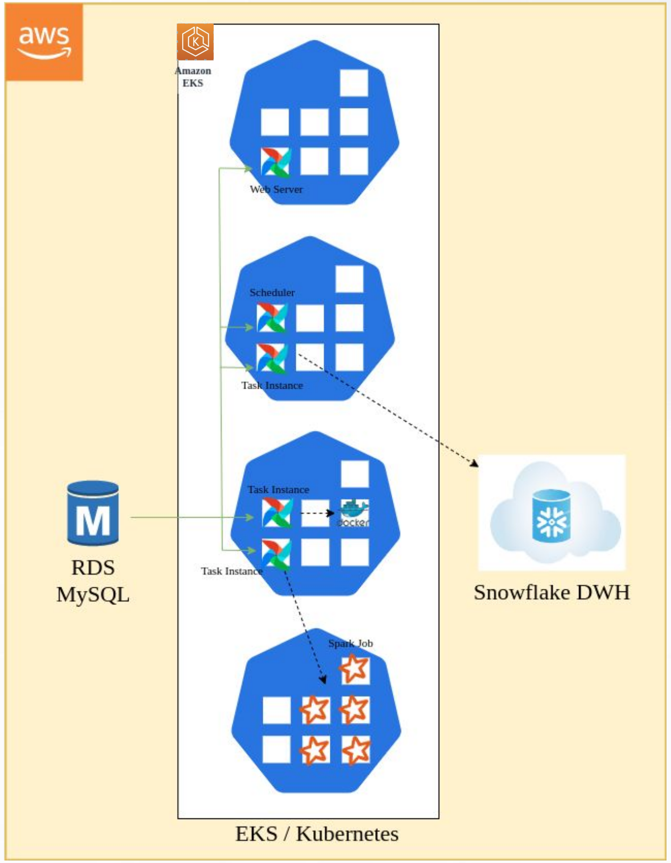
Migration to Kubernetes - Phase 2

- Upgraded Airflow to 2.0
- Airflow running inside of Kubernetes using KubernetesExecutor
- Updated deployment of Airflow to use Helm



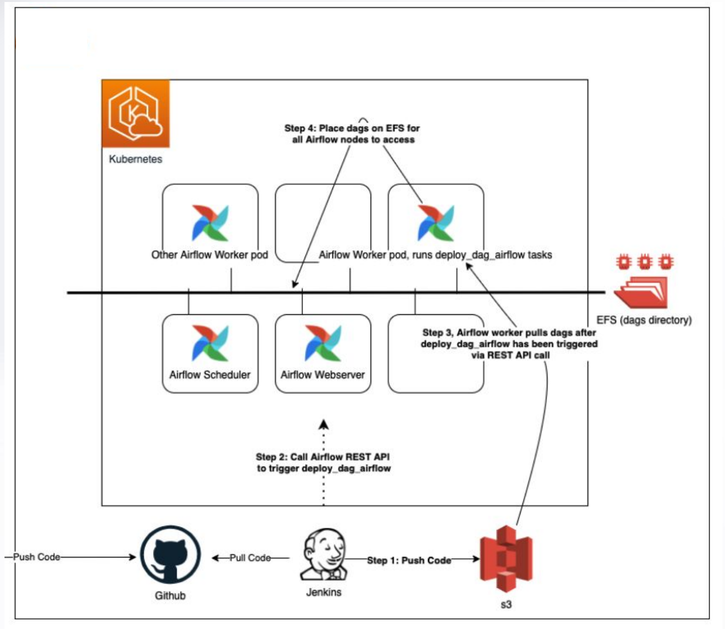
Migration to Kubernetes - Phase 3

- Moved Spark workloads to Spark On Kubernetes
- Deploy Spark jobs with in-house Spark on Kubernetes Airflow Operator
- Removed EFS and opt for git-sync sidecar deployments into Airflow

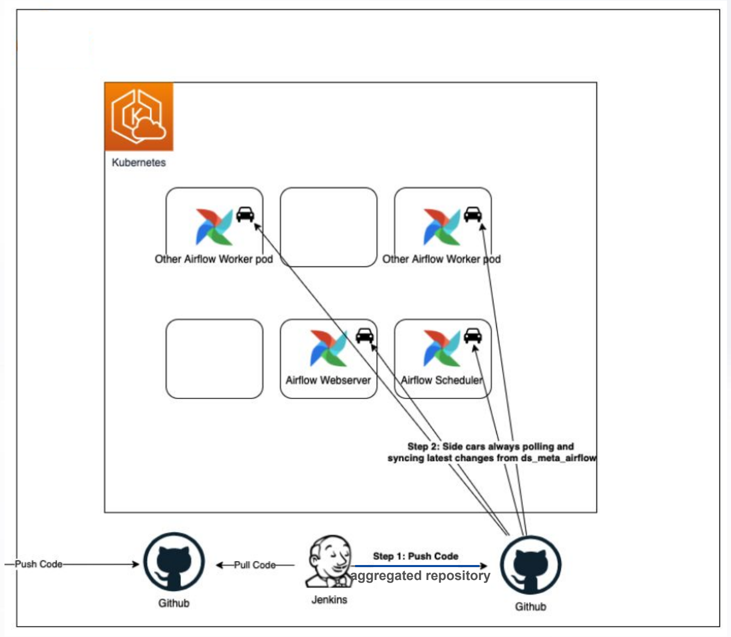


Airflow Dag Deployment in Prod

Old


Kubernetes
Step 4: Place dags on EFS for all Airflow nodes to access
Other Airflow Worker pod
Airflow Worker pod, runs deploy_dag_airflow tasks
EFS (dags directory)
Airflow Scheduler
Airflow Webserver
Step 3, Airflow worker pulls dags after deploy_dag_airflow has been triggered via REST API call
Step 2: Call Airflow REST API to trigger deploy_dag_airflow
Push Code
Github
Pull Code
Jenkins
Step 1: Push Code
s3

VS



New

- Too many apps/processes involved
- Prone to various errors
- Hard to trace back issues
- Slow deployment and code sync

Considerations for a Kubernetes based Platform

- What additional tooling will we need to put in place to enable Data Scientists to do Airflow dag development as well as ML code development and testing within a container-native platform?
- What does our CI/CD need to look like to allow our Data Scientists to go seamlessly from dev to prod?
- What additional Operators do we need to build to align with our custom tools and custom CI/CD processes?

Dev to Prod Lifecycle: Requirements

- Different levels of flexibility for users per environment
  - Very minimal restrictions in dev
  - More control and restrictions into production
- Isolation in dev environment
  - Users provided with sandbox including personal Airflow, personal Snowflake DBs and ability to run isolated workloads using on-demand resources
- Seamless transition from dev to production
  - Same code that runs/tested in dev should work in production without any change (reduces the time to release new changes and delivery time)

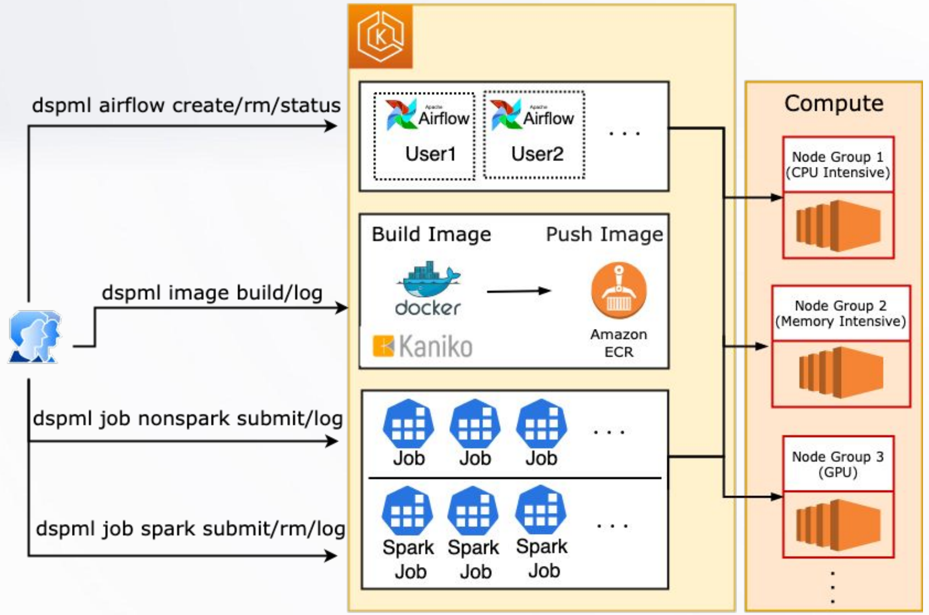
Dev to Prod: Tools & Processes

- Built various custom tools to help users interact with the platform components and monitor workloads
  - Built REST API and a CLI tool for the Data Scientists to interact with the platform
  - Datadog dashboards to track custom metrics and monitor performance

- Built custom CI/CD processes to help promote Airflow dags from dev to production

Dev to Prod: Custom Tools

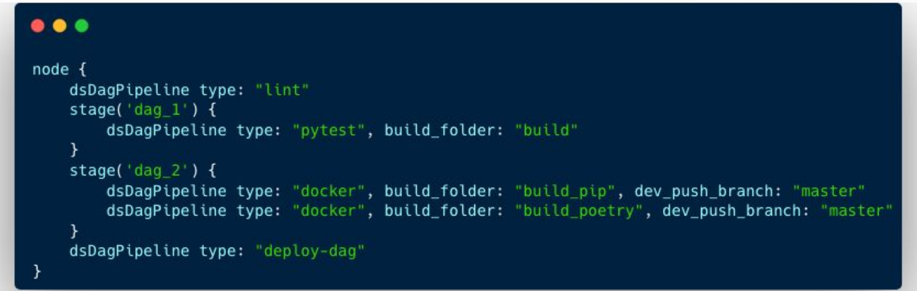
- Built REST API and a CLI(called dspml) for the Data Scientists to
  - Simplify interacting with the platform
  - Don't need to know about Kubernetes



- Users can manage their own Airflow instance
- Users can build Docker images inside the platform
- Users can submit a stand-alone or a Spark job using a command

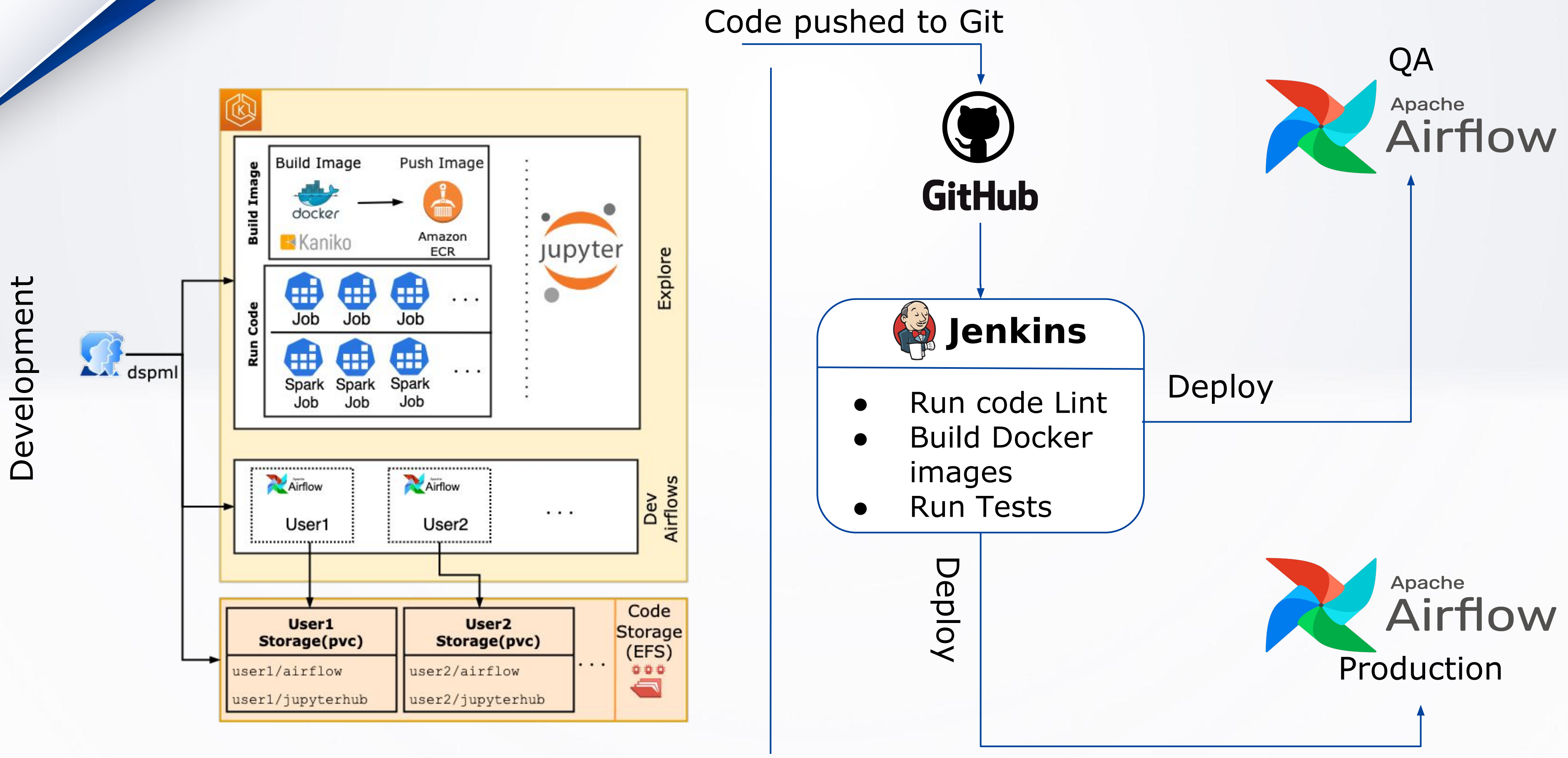
Dev to Prod: Custom CI/CD Processes

- Built custom CI/CD to build Docker images and/or deploy Airflow dags
  - Users can define different dags and in each one they can easily:
    - Build different Docker images for various environments
      - e.g., Main branch for Prod
    - Run tests for each dag
    - Run pre-commit tools e.g., lint code
  - And deploy the dags



```
node {
    dsDagPipeline type: "lint"
    stage('dag_1') {
        dsDagPipeline type: "pytest", build_folder: "build"
    }
    stage('dag_2') {
        dsDagPipeline type: "docker", build_folder: "build_pip", dev_push_branch: "master"
        dsDagPipeline type: "docker", build_folder: "build_poetry", dev_push_branch: "master"
    }
    dsDagPipeline type: "deploy-dag"
}
```

Dev to Prod: Overview



- Run code Lint
- Build Docker images
- Run Tests

User Code Management: Single Repo vs Multi

Single Repository

- Pros
  - Easier to manage and deploy
  - Easy to do bulk changes
- Cons (Impacting Users)
  - Commit history is a nightmare
  - Lack of custom control and rules per team/project
  - Teams slowing down other teams
  - What if a project contains other artifacts than just dags?

Multi Repository

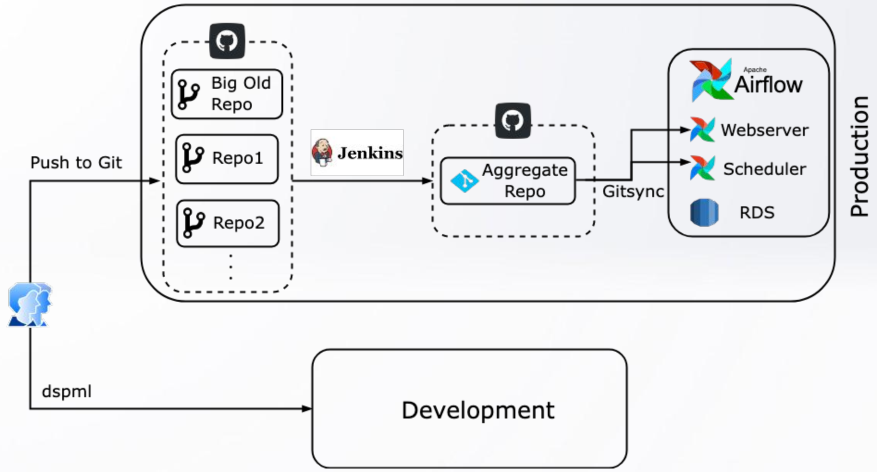
- Pros:
  - Cleaner git history related to project
  - Ability to set custom config per repo
- Cons:
  - Difficult to manage, deploy and bulk changes
- What if we setup the platform to mitigate the cons?
  - Better user experience 🎉

User Code Management: Multi Repo Requirements

- Requirements
  - Old single repo should work alongside the newer one-off repos
  - CI/CD pipeline to deploy all the repos to Airflow
  - Ability to do bulk code updates (https://github.com/mateodelnorte/meta can help here)
  - Easy way to create a new repo from a template and add it to the mix
  - Ability to keep other code than just Airflow dags code
  - No impact on development environment

User Code Management: Transition to Multi Repo

- How did we do it
  - Each repo represents a project that contains multiple dags
  - A CI/CD pipeline aggregates all repos and pushes to one repo to be synced to Airflow
  - Docker images are tagged meaningfully to be traced back to the original repo/code `(e.g., image_tag=sample_repo.dag1.build_folder1)`
  - Dags within each repo are tagged to be filtered easier in UI `(e.g.,dag=DAG(tags=[repo:repo1])`



Containerization: Why?

All our data preprocessing and ML tasks within Data Science used to run on EMR cluster (using in-house custom LivyOperator in Airflow)

But:

- EMR has disadvantages
  - Scaling issues
    - EMR scale up/down is slow
    - Less flexibility to choose the underlying nodes and numbers
  - Shared infrastructure to run all workloads
  - Lacks flexibility to add/remove libraries
    - Certain libraries installed on cluster at the beginning / users can’t add new libraries / users can’t test and use different versions of a library
  - Difficult to integrate into the rest of DS Platform tools e.g., dspml

$\rightarrow$ **Spark Workload**

- No need to run non-Spark code on EMR cluster or Airflow itself when we can run them as a container on Kubernetes

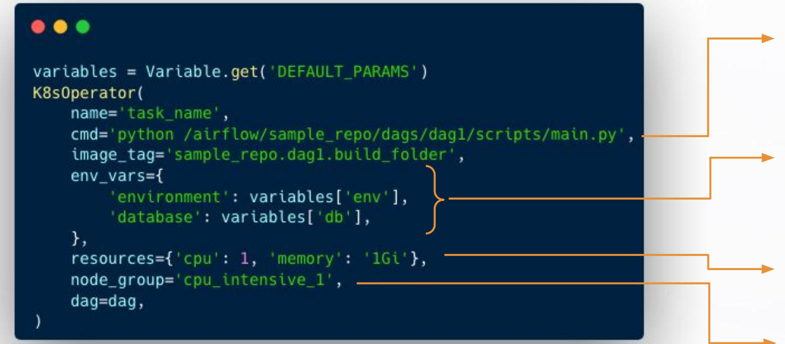
$\rightarrow$ **Non-Spark Workload**

Containerization & Benefits

- Separation of the applications and infrastructure in the platform
- Provide users the flexibility to choose and experiment with any Python libraries
- Allows us to introduce new Python dependency management tools i.e., Poetry
- Provide resource isolation for tasks
- Provide a unified way to run various jobs/tasks (Spark, non-Spark Python code)
- Running different workloads on different types of instances
- Kubernetes perfectly supports that

Containerization(Non-Spark Workload): K8sOperator

- Extended from the official KubernetesPodOperator
- It is being used to run stand-alone ML tasks on various different nodes
- Any Python libraries can be built into a Docker image and run as part of a task in Airflow
- Same code works across different environments without any change



```
variables = Variable.get('DEFAULT_PARAMS')
K8sOperator(
    name='task_name',
    cmd='python /airflow/sample_repo/dags/dag1/scripts/main.py',
    image_tag='sample_repo.dag1.build_folder',
    env_vars={
        'environment': variables['env'],
        'database': variables['db'],
    },
    resources={'cpu': 1, 'memory': '1Gi'},
    node_group='cpu_intensive_1',
    dag=dag,
)
```

Path to code which is similar in different environments

Passing run time parameters or Airflow variables that change w.r.t to different environments

Adjust resources for the task e.g., cpu, memory, gpu

Choose the underlying node type

Containerization(Spark Workload): Spark on Kubernetes

Advantages of running Spark on Kubernetes:

- Scaling is faster and easier
- No more shared resources
- Users can add/remove new Python libraries by creating new Docker images
- Integrates nicely into our dspml cli tool to submit Spark jobs outside Airflow
- Better monitoring of jobs resource utilization using custom Datadog dashboards
- Conceptually similar to K8sOperator i.e., easier for the users to follow
- Cost of infrastructure is lower

Containerization(Spark Workload): SparK8sOperator

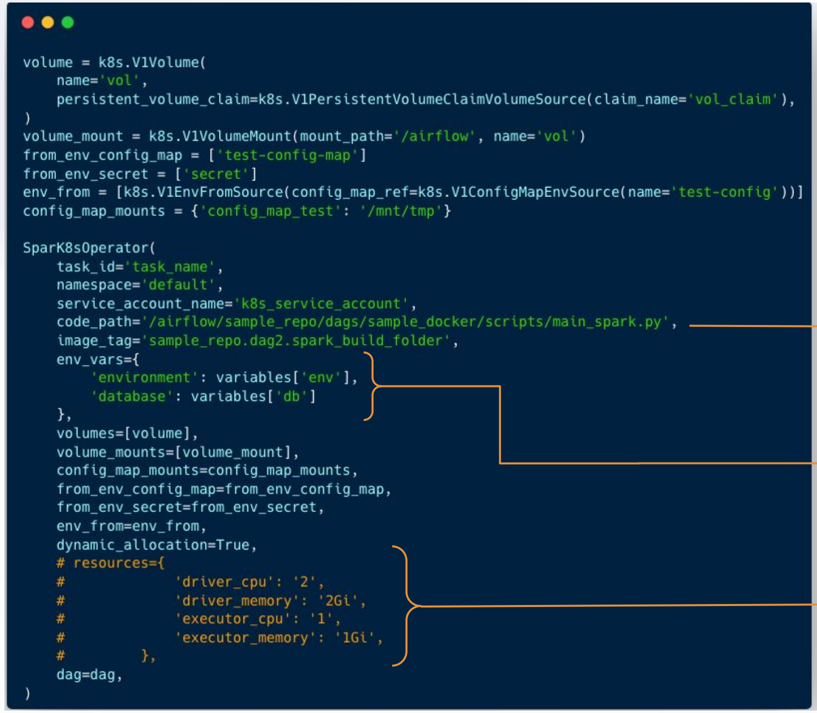
We wrote a new Airflow operator to support running Spark on Kubernetes

- It uses spark-on-k8s-operator behind the scenes to submit a job
  https://github.com/GoogleCloudPlatform/spark-on-k8s-operator
- Simplifies the interface to accept different parameters in **Python**
- Implements the logic to:
  - Wait for a job after submission
  - Manage error handling
  - Retrieve logs from the driver pod
  - Ability to delete a Spark job
  - It also supports out-of-the-box Kubernetes functionalities such as handling volumes, configmaps, secrets, env variables

Soon will be open-sourced and merged into Airflow

Containerization: SparK8sOperator

Example of using the operator



```
volume = k8s.V1Volume(
    name='vol',
    persistent_volume_claim=k8s.V1PersistentVolumeClaimVolumeSource(claim_name='vol_claim'),
)
volume_mount = k8s.V1VolumeMount(mount_path='/airflow', name='vol')
from_env_config_map = ['test-config-map']
from_env_secret = ['secret']
env_from = [k8s.V1EnvFromSource(config_map_ref=k8s.V1ConfigMapEnvSource(name='test-config'))]
config_map_mounts = {'config_map_test': '/mnt/tmp'}

SparK8sOperator(
    task_id='task_name',
    namespace='default',
    service_account_name='k8s_service_account',
    code_path='/airflow/sample_repo/dags/sample_docker/scripts/main_spark.py',
    image_tag='sample_repo.dag2.spark_build_folder',
    env_vars={
        'environment': variables['env'],
        'database': variables['db']
    },
    volumes=[volume],
    volume_mounts=[volume_mount],
    config_map_mounts=config_map_mounts,
    from_env_config_map=from_env_config_map,
    from_env_secret=from_env_secret,
    env_from=env_from,
    dynamic_allocation=True,
    # resources={
    #               'driver_cpu': '2',
    #               'driver_memory': '2Gi',
    #               'executor_cpu': '1',
    #               'executor_memory': '1Gi',
    #          },
    dag=dag,
)
```

Example of configuring Kubernetes resources e.g., mount configmap, mount volume, load configmap to env, load secret to env

Path to Spark entry point which does not change across environments

Passing run time parameters or Airflow variables that change w.r.t to different environments

Configuring resources. Dynamic allocation or manual assignment

Questions?