Implementing Cloud Agnostic Machine Learning Workflows With Apache Beam on Kubernetes

By Alexander Lerma & Charles Adetiloye
About The Presenters

Charles Adetiloye is a Cofounder and Lead Machine Learning Platforms Engineer at MavenCode. He has well over 15 years of experience building large-scale distributed applications. He has extensive experience working and consulting with several companies implementing production grade ML platforms.

Twitter: twitter.com/cadetiloye

Alexander Lerma is a Machine Learning Platforms Engineer at MavenCode. He has 10 years of experience working as a Software Engineer and MLOps Engineer. Previously worked with Goldman Sachs, Twitter, and a few other startups.

Twitter: twitter.com/neuralnetes
About MavenCode

MavenCode is an Artificial Intelligence Solutions Company with HQ in Dallas, Texas and remote delivery workforce across multiple time zones. We do training, product development and consulting services with specializations in:

- Provisioning Scalable AI and ML Infrastructure - OnPrem and In the Cloud
- Development & Production Operationalization of ML platforms - OnPrem and In the Cloud
- Streaming Data Analytics and Edge IoT Model Deployment for Federated Learning
- Building out Data lake, Feature Store, and ML Model Management platform

twitter.com/mavencode
Agenda for Today

1. Making the Case for Cloud Agnostic ML Deployments
2. Building it all on Kubernetes
3. Orchestration of Beam Job Deployments with Argo Workflows
4. Agile Team Approach to ML Workflow Deployment
5. Lessons Learned and Summary
Making the Case for Cloud Agnostic Machine Learning Deployments
1. The goal of any Machine Learning application is to build a Statistical Model using curated data and applying Machine Learning algorithms to them.

2. The main artifacts of any Machine Learning Projects are Data, ML Model and the Code.

3. This is how a Simplified ML Workflow looks
Challenges in Building ML Workflow

Reproducibility
- Not Easy to Reproduce ML Model Output on each iterative runs
- Constantly Changing Training Data
- Consistent Environment Configuration Issues

Reusability
- Training Pipelines are not Componentized for Reusability
- No well defined way of doing Model versioning and tagging
- Collaboration and sharing of source code is not well defined

Manageability
- Managing model deployment and serving between environments is difficult
- Versioning and Tracking model artifacts is very difficult and complex
- No defined way to visually track updates and changes

Automation
- A lot of deployment process is still manual
- Steps needed to update model parameters are not not automated
- Most data science teams are not equipped with the right knowledge to take models to production
Building ML Workflows in Reality could be Complex!

- Many Data Sources - Databases, File Systems, Storage Buckets, Streaming Data.
- The Data Sources in most cases are siloed across different locations with various access requirements: In-House datasets, Third Party, Public datasets.
- Different Data Protection Requirements - PI data, GDPR restrictions etc
- Data Availability in some cases are time-bounded! Streaming or Batched delivered Hourly, Daily, Monthly etc
Typical ML workflow in Reality is more Complex!

1. Ingest Data from various sources with different characteristics using Beam IO SDK

2. Write the Ingested data into Delta Lake with Beam IO writers
   - Expose data in a structure that can be queried for wrangling or quick exploratory analysis by Data Scientist or Data Engineers

3. ML Model Training / Testing and Tuning until the best performance is achieved

4. - Model Management and Deployment Rollout
   - Post Deployment Monitoring

Ingest Data from various sources with different characteristics using Beam IO SDK

Write the Ingested data into Delta Lake with Beam IO writers

Expose data in a structure that can be queried for wrangling or quick exploratory analysis by Data Scientist or Data Engineers

ML Model Training / Testing and Tuning until the best performance is achieved

- Model Management and Deployment Rollout
- Post Deployment Monitoring
Our Approach to ML Workflow Deployment

- We have a Polyglot team. Use the best tool to solve the problem, as long as we can containerize it, so we have beam pipeline codes written in Go, Scala, Java and Python.
- Make use of Apache Beam’s Runtime portability makes it easy for us to do local controlled testing of our ML pipelines.
- We build our components to be reusable, Data Source, Data Writer, Feature Store Components, Model Training Components, and Model Serving Components.
- Versioned Containerized Workflow Pipeline with Argo Workflow.
- For team efficiency and consistency, we leverage Containers built on Kubernetes to gain portability across infrastructure.
Embracing the Apache Beam Philosophy

### Beam Advantage

- **Unified Data Pipeline (Stream + Batch)**
- **Multi-Language Support** (with new Portable Runner for Java/Python/Go)
- **Multiple Runner Support** (Direct, Flink, Spark, and Dataflow)
- **Beam ML, Beam Dataframe, and TFX integration for ML workloads**

<table>
<thead>
<tr>
<th>Data Engineer</th>
<th>Data Scientistwriting ML Codes with Beam Dataframes, and Beam SQL</th>
<th>Still very early stage for us</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beam Java</td>
<td>Beam Python</td>
<td>Beam Go</td>
</tr>
<tr>
<td>Pipeline (Runner API)</td>
<td>Direct Runner</td>
<td>Flink Runner</td>
</tr>
</tbody>
</table>

Still very early stage for us
With Kubernetes Added, ML Deployment is Easier

Data Engineer implementing Production Ready Data Pipeline in Java

Data Scientist writing ML Codes with Beam Dataframes, and Beam SQL

Still very experimental not enough proven use-cases

Beam Advantage

- Unified Data Pipeline (Stream + Batch)
- Multi-Language Support (with new Portable Runner for Java/Python/Go)
- Multiple Runner Support (Direct, Flink, Spark and Dataflow)
- Beam ML, Beam Dataframe and TFX integration for ML workloads

Kubernetes Advantage

- Infrastructure Agnostic Setup!

Beam Java  Beam Python  Beam Go

Pipeline (Runner API)

Direct Runner  Flink Runner  Spark Runner

Direct Runner  Flink Runner  Spark Runner

AWS  Azure  GCP

Austin, 2022
Advantages of Running Apache Beam ML Pipeline on Kubernetes

- Leverage Beams Portability, Multi-Language Semantics that now allows support for Python, Java, Scala and Golang. In addition we can use External Transforms to make calls from Python to Java Code.

- Rich beam Library and API available to handle each stage of Workflow process, TFX, Beam SQL, and connector IOs.

- Beam Job a can be Containerized as Unit of work that can be easily maintained on its own and deployed on the Kubernetes Cluster.

- Infrastructure Portability on Kubernetes makes it easy to share or migrate between local Kubernetes environment and production or development environments.

- Consistency between operating environment for Data Scientist, ML Engineers and what is finally deployed.

- Ease of debugging and testing on Local environment before deployment.
Building it All on Kubernetes
Building Apache Beam ML Pipeline Stack on Kubernetes

1. Portability of Coding Semantics (Java, Scala, Python, Go or SQL)
2. Portability of Across Runners (Direct Runner, Flink Runner, Spark Runner, Dataflow Runner)
3. Portability of Across Compute Infrastructure - Local Dev, OnPrem or Cloud
Portability in Apache Beam

Beam Model Pipeline Construction with Runner API (Proto)

The Runner API provides SDK and Runner Independent definition of the Beam Pipeline

Fn API allows the Runner to invoke SDK specific environment
Portability in Apache Beam

Beam Model Pipeline Construction with Runner API (Proto)

The Runner API provides SDK and Runner Independent definition of the Beam Pipeline

Beam Model Execution: (Fn API)

Fn API allows the Runner to invoke SDK specific environment

- Beam Java
- Beam Python
- Beam Golang
- Future SDK Implementations
- Spark Runner
- Flink Runner
- Samza Runner
- Dataflow Runner
- Other Runner Implementations
- Java Execution Environment
- Python Execution Environment
- Golang Execution Environment
- Future Execution Env Implementations

Austin, 2022
How Apache Beam Portability Works

### Beam Job Example

```python
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

options = PipelineOptions([  
    "--runner=PortableRunner",
    "--job_endpoint=localhost:8099",
    "--environment_type=DOCKER"
])

with beam.Pipeline(options) as p:
    ...
```

### Job Service Endpoint

- **1:**
  - `docker run apache/beam_spark_job_server:latest`  
  - `--spark-master-url=spark://<SPARK_MASTER_URL>:7077`
  - OR

- **2:**
  - `docker run apache/beam_flink1.14_job_server:latest`  
  - `--flink-master=<FLINK_MASTER_URL>:8081`

### Deployment Clusters

- **3:**
  - JobService Runner Deploys Job to remote or local Spark or Flink Cluster

---

Developer Implements Beam Job in Local Environment in this Case Python

- Starts up a Job Service Runner that targets the specific ENV they want
How Apache Beam Portability Works

Beam Job Example

```python
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

options = PipelineOptions([ "--runner=PortableRunner",
                            "--job_endpoint=localhost:8099",
                            "--environment_type=LOOPBACK"
                        ])
with beam.Pipeline(options) as p:
    ...
```

Job Service Endpoint

- `docker run apache/beam_spark_job_server:latest
  --spark-master-url=spark://<SPARK_MASTER_URL>:7077`
- `OR`
- `docker run apache/beam_flink1.14_job_server:latest
  --flink-master=<FLINK_MASTER_URL>:8081`

Deployment Clusters

1. Developer Implements Beam Job in Local Environment in this Case Python
2. Starts up a Job Service Runner that targets the specific ENV they want
3. JobService Runner Deploys Job to remote or local Spark or Flink Cluster
Beam Portability Advantages

- Rich set of IOs already implemented in Beam Java can be invoked in Python, GO etc
  
  ```python
  from apache_beam.options.pipeline_options import PipelineOptions
  from apache_beam.io.kafka import ReadFromKafka

  p = beam.Pipeline(options=pipeline_options)
  res = (p | 'ReadFromKafka' >> ReadFromKafka(consumer_config={"bootstrap.servers": "localhost:9092"},topics=['<TOPICS>']))
  ```

- Using Expansion Service with External Transforms to call Java APIs
  
  ```python
  from apache_beam.options.pipeline_options import PipelineOptions
  from apache_beam.io.kafka import ReadFromKafka

  p = beam.Pipeline(options=pipeline_options)
  res = (p | 'ReadFromKafka' >> ReadFromKafka(consumer_config={"bootstrap.servers": "localhost:9092"},topics=['<TOPICS>'])
          | 'ReadFromKafka' >> beam.JavaExternalTransform("org.apache.beam.sdk.io.TextIO").write().to("<STORAGE_PATH>"))
  ```
Infrastructure Portability on Kubernetes with Apache Spark

Data Scientist and Engineers can iteratively quickly test out their Beam Code on their local environment that mirrors the prod clusters before deployment to the prod environments.

Minikube (Local Dev)

OnPrem or Managed Cloud Deployment of Kubernetes

Minikube (Local Dev)
Pipeline Package Management and Deployment on Kubernetes with Kustomize

- The Spark/Flink Runner Cluster deployment process is managed with Kustomize, making it easy to version control and manage the deployment via GitOps process.

- We can target different deployment environment with Kustomize overlay that overrides the base configuration, allowing us to deploy across multiple environments.

- We can use Kustomize configuration to target various Kubernetes Infrastructure - OnPrem, AWS, GCP, Azure.

- It is easy to progressively extend and manage various version of packages that is trackable via git release.
Infrastructure Portability with Kubernetes (Apache Spark)

Data Scientists and Engineers can iteratively quickly test out their Beam Code on their local environment that mirrors the prod clusters before deployment to the prod environments.

Minikube (Local Dev)

- Minikube
- Data Service
- Beam Job
- Spark
- worker
- worker
- worker

OnPrem or Manage Cloud Deployment of Kubernetes

- Kubernetes Namespace
- Git Managed Spark Manifest

Git Managed Spark Manifest gets deployed on the Kubernetes Clusters

Spark Manifest
Overview and Quick Demo of How it All fits together
So with Apache Beam and Kubernetes ...

- We gain Portability across our development environments
- Easily leverage the functionalities of all the extensive Apache Beam Libraries especially Java
- We use Kustomize Manifest to Deploy the Spark or Flink Clusters on Kubernetes
Orchestration of Beam Job Deployment with Argo Workflows
Implementing ML Workflow Pipelines on Kubernetes

We have achieved Portability of Code and Portability of Infrastructure!

But beyond that we need to create Workflows that can chain “Tasks” that we need to execute together and while also enforcing the dependencies between them.
Introducing Argo Workflow for Orchestrating Workflows

Argo Workflow is an open source container-native workflow engine for orchestrating parallel jobs on Kubernetes.
Why Use Argo Workflow with Beam?

- Runs natively on Kubernetes
- We can define each stage of Beam Job as a task that runs in it’s own container
- The Argo Workflow abstraction makes it easy for us to create multi-step tasks with varying availability and latencies
- Easy to compose complex tasks as a series of steps and because it’s running on kubernetes, it’s easily portable across infrastructure
Implementing Argo Workflow for Beam Jobs

- ML Engineers / Data Scientists are responsible for different components
- The Components are containerized and Tagged as a Beam Job (or Any other Job type)
- Argo DSL will be used to compose the Pipeline DAG and Graph
Implementing Argo Workflow for Beam Jobs

Argo Workflow DAG

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: beam-dag
spec:
  entrypoint: main
templates:
- name: main
dag:
tasks:
  - name: kafka-reader-io
    template: kafka-reader-io
  - name: enrichment-feature-transform
    depends: kafka-reader-io
    template: enrichment-feature-transform
  - name: dataset-split
    depends: enrichment-feature-transform
    template: dataset-split
  - name: model-training
    depends: dataset-split
    template: model-training
  - name: model-testing
    depends: dataset-split
    template: model-testing
  - name: kafka-reader-io
    container:
  - name: enrichment-feature-transform
    container:
      image: gcr.io/beam-summit-mlops/enrichment-feature-transform:latest
  - name: dataset-split
    container:
      image: gcr.io/beam-summit-mlops/dataset-split:latest
  - name: model-training
    container:
      image: gcr.io/beam-summit-mlops/model-training:latest
  - name: model-testing
    container:
      image: gcr.io/beam-summit-mlops/model-testing:latest
```

Austin, 2022
Implementing Argo Workflow for Beam Jobs
Using Argo Events to Trigger Beam Pipeline Workflows

- Allows for dynamic creation of Argo Workflows to run our beam jobs
- Allows for various event trigger sources such as kafka or calendar events (cron jobs)
- You can combine workflows based on conditional triggers
- Cloud agnostic, not tied to any managed service
- Kubernetes native
- Portable across infrastructures
Agile Team Development Approach to ML Workflow Deployment
Collaborative ML Component Pipeline Development

1. **ML Engineer**
   - Step 1
     - Create Component
     - Validate Input/Output
     - Create a TestRun
     - Inspect Output Artifact

2. **ML Engineer**
   - Step 1
     - Create Component
     - Validate Input/Output
     - Create a TestRun
     - Inspect Output Artifact

3. **Data Scientist**
   - Step 2
     - Create Transform Component
     - Validate Input/Output
     - Create a TestRun
     - Feature Vector ML

- **TFX Beam SQL**
- **TFX Beam SQL**

**Source**
- read-pubsub
- dataflow-eti
- rollup-storage
- train-ml

Austin, 2022
Collaborative ML Component Pipeline Development

- Each Beam Job runs in Container as manageable Unit
- The Container is versioned / Tracked with other artifacts needed for the deployment
- Pipeline for the Workflow is implemented in Argo and also tracked and versioned
- The output artifact from the Pipeline run is saved with the Pipeline Run Version ID
Argo Workflow ML Pipeline Components (Versioning)

ML Engineer working on a particular component can branch out and create a new version of the component without breaking the existing implementations.
Argo Workflow ML Pipeline Workflow (Versioning)
Lessons Learned and Summary
Challenges in Building ML Workflow

Reproducibility
- Not Easy to Reproduce ML Model Output on each Iterative Runs
- Constantly Changing Training Data
- Consistent Environment Configuration Issues

- Using Argo Workflow
- Kubernetes

Reuseability
- Training Pipelines are not Componentized for Reusability
- No well defined way of doing Model versioning and tagging
- Collaboration and sharing of source code is not well defined

- Containerization with Docker
- Argo Workflow Pipeline
- Leveraging Beam APIs

Manageability
- Managing model deployment and serving between environments is difficult
- Versioning and tracking model artifacts is very difficult and complex
- No defined way to visually track updates and changes

- Beam Portability
- Infrastructure Portability with Kubernetes
- Argo Workflows
- Configurable Runners

Automation
- A lot of deployment process is still manual
- Steps needed to update model parameters are not automated
- Most data science teams are not equipped with the right knowledge to take models to production

- GitOps
- Kubernetes
Lessons Learned + Summary

- Apache Portable Runner Implementation blueprint is very solid even though it’s evolving, it makes easy for us to quickly test our implementations on a small scale before production deployment.

- We are able to leverage Apache Beam / Kubernetes development environment setup to make it easy for Data Scientist, ML engineers, Data Engineers to easily collaborate on our team.

- Version of SDKs, JobService Containers etc, could easily get mismatched, It’s always advisable to have a CI environment for testing releases.

- Aside from the initial overhead of getting the environment setup, our productivity and team efficiency increased significantly.

- Cloud is not Cheap, we can easily manage our compute resource utilization.
Q & A!

Thanks for Coming :-) !

Connect with Us on
Twitter @mavencode
Github @mavencode
Email: hello@mavencode.com