OperatorFlow

Software Requirements Specification

Yu Chen
SYRACUSE UNIVERSITY, COLLEGE OF ENGINEERING AND COMPUTER SCIENCE

Contents

Revision History

Name	Date	Reason for Changes
Yu Chen	July 24th, 2019	First initial draft
Yu Chen	August 14th, 2019	Iteration on priority objectives
Yu Chen	September 18th, 2019	Final iteration about MVP implementation is completed

Source Code and Github Repositories

There are several source code repositories available for the full project:

- OperatorFlow backend: contains infrastructure and KubeFlow deployment scripts
- OperatforFlow frontend: contains the frontend React code

Video Demonstration

A video demonstration of this project and its various features is available on YouTube.

Introduction

Purpose

The purpose of this software requirements specification document is to facilitate the development and prototyping of a distributed machine learning and data processing platform.

The rise of Big Data and the general availability of high volumes of data has given rise to immense opportunities but unique challenges related to scaling, securing, and deriving analytical insights from large swaths of often unstructured, "messy" raw data. In order to take full advantage of this newly available data, organizations typical set up a data pipeline that flows from the data source down to the final end user (clients or data insight consumers). Different roles within the data pipeline operate at different level of abstraction:

- **Hardware engineers:** responsible for hardware architectures (such as CISC / RISC decisions), pipelining architectures, etc.
- **System administrators**: responsible for the maintenance, deployment, and upkeep of bare-metal servers, infrastructure deployments
- **Backend developers**: responsible for establishing database schemas, ETL pipelines, entity models, etc. for downstream consumers.
- **Data scientists:** Tensorflow, scikit-learn, Apache Spark, etc.

- **Data analysts** → Excel, BI dashboards
- **Account managers** → Excel, PowerPoint, PDFs

A common pain point, however, is that different stakeholders must operate after a level of abstraction that is higher / lower than their role intends or their skillset aligns with. This manifests in a variety of use-cases:

- System administrators designing snowflake schema architectures of a data warehouse (not their area of expertise, goes beyond their core capabilities of provisioning infrastructure and configuring machines and networking
- Account managers having to run SQL queries themselves to generate raw reports for their clients (this is an example of operating at too low a level of abstraction, as most account managers are focused on customer relationship management, interfacing with executives, and preparing higher-level strategic reports, not the syntax of MySQL versus PostgreSQL).
- Data scientists setting up their own infrastructure, networking, and security frameworks to deploy their machine learning models

The last use case is the intended target of OperatorFlow. The framework that is built hopes to abstract away the complexities of deploying models so that data scientists can spend a greater portion of their time operating within their expertise domains.

Document Conventions

Intended Audience And Reading Suggestions

This document is intended for machine learning engineers intending to develop a DevOps pipeline for faster testing, deployment, and execution of machine learning workflows.

Product Scope

The **OperatorFlow** framework is divided into two main components: **SparkFlow** and **KubeFlow**:

- **SparkFlow** is an interface to quickly and efficiently run **Databricks Spark** notebooks. Users will be able to see existing clusters, jobs, and notebooks, and select a specific notebook to run.
- **KubeFlow** is a subplatform for distributed deep learning models, although it will also be able to run any sort of arbitrary container workload.
- **ControlFlow** is a controller and dashboard that exposes a UI where users can navigate into the SparkFlow and KubeFlow subcomponents, as well as monitor the status of their workflows and jobs.

All the various subcomponent configurations of each system is intended to only handle configurations via industry-established protocols, including:

- YAML flat files for Kubernetes and Istio service mesh resource deployments
- JSON for AWS Cloudformation deployment stacks
- Terraform HCL (HashiCorp Language) for cloud infrastructure provisioning

Currently, OperatorFlow is configured only to work within an AWS cloud infrastructure ecosystem. Other cloud vendors and configuration settings (via JSON, in-memory key-value stores, etc.) will be part of future release versions.

References

Since a significant portion of the framework requires software reuse from Google's Kubeflow packages, it is important to list their resources below:

- Kubeflow Documentation
- Kubeflow Open Source codebase (Github repository)

Moreover, the SparkFlow subcomponent of the OperatorFlow framework utilizes a 3rd-party proprietary framework itself called Databricks, which hosts Spark notebooks that run in the cloud. OperatorFlow interacts with the Databricks framework via an API. The resources associated with the API are available below:

- Databricks API documentation
- Databricks Jobs and Clusters resources

Overview of Document

The next section, **Overall Description**, will provide a high-level summary of the core functionality and product perspectives, including the pain points and use cases that inspired the application.

The next section, U**ML Diagrams**, presents different UML diagrams that provide insight into common use cases for the application.

Afterwards, specific **User, System, and Non-Functional Requirements** are provided for the developers who will be in charge of implementing this solution.

Overall Description

Product Perspective

The application is developed for technical users, specifically for data scientists, data engineers, and machine learning engineers. It aims to abstract away the low-level details of running distributed machine learning workflows away from technical professionals, allowing them to focus on tuning and architecting better machine learning models and less time on drudgery (network infrastructure, setting up subnets, generating TLS certificates, etc.).

Product Functions

In general, OperatorFlow aims to provide a consistent container runtime environment for data scientists and data engineers, with as much of the networking, security, and infrastructure details abstracted away as possible.

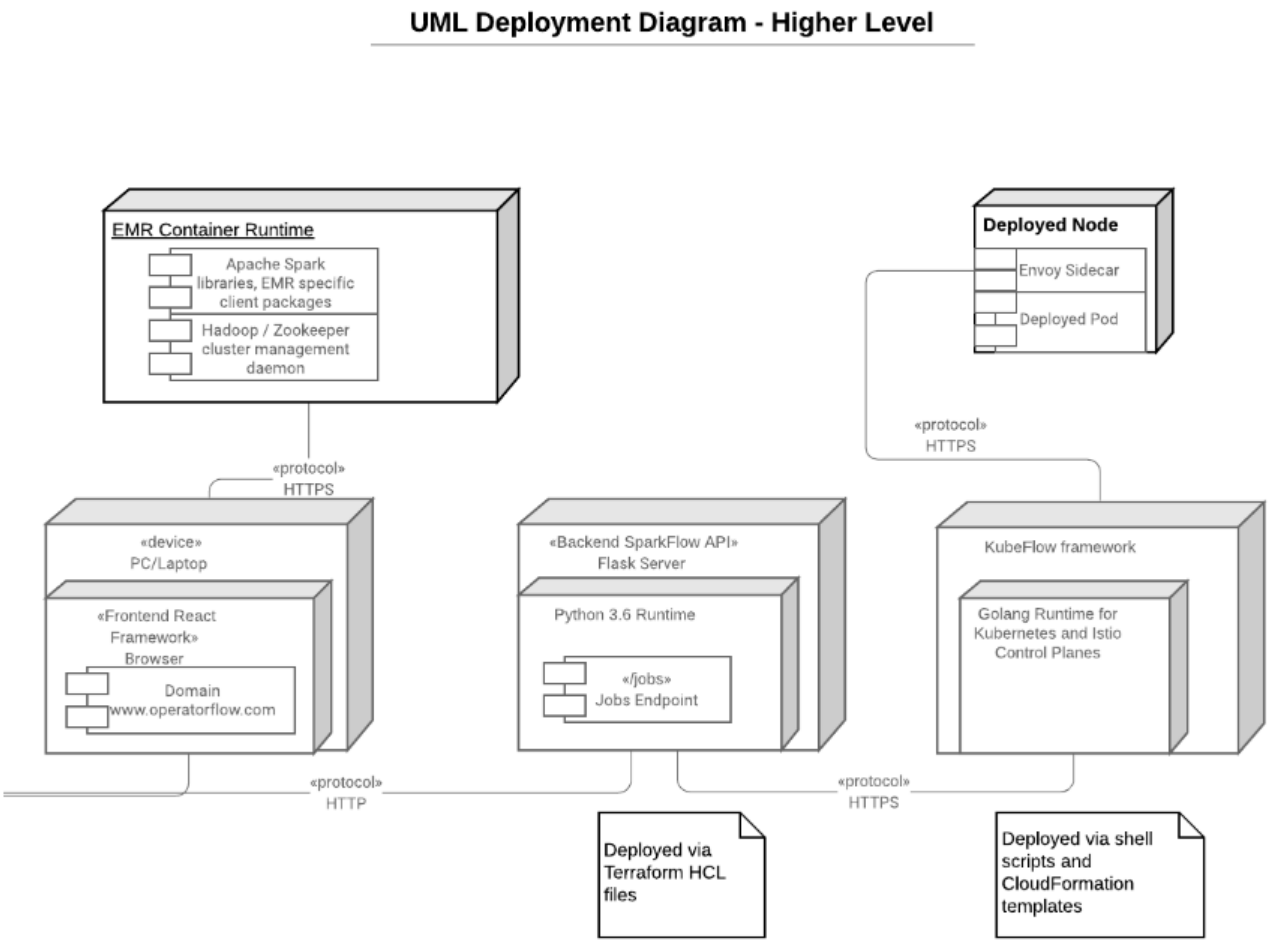
- For distributed ETL workloads, the user will be able to execute a **SparkJob** that will automatically scale up or down depending on CPU consumption and memory usage. A User (data scientist) is responsible for developing his/her model in the form of a **Notebook**.
- For traditional machine learning workflows, the user will be able to execute a KubeJob that is distributed across multiple different nodes within a Kubernetes cluster. A User is responsible for developing his/her model in the form of an **Image**.

UML Diagrams

Deployment Diagram

The infrastructure for this framework shall be deployed via Packer templates, Terraform HCL resources, CloudFormation templates, and deploy shell scripts. The selection of these templates and tools is in order to maintain state as much as possible – this simplifies the deployment process of infrastructure such that it is as painless as possible to provision / de-provision / rollback deployments.

Traditionally, these deployments have been completely manual, where a system administrator logs in to the instances and manually executes bash commands to install packages and configure services. The aim of using these templates and resources is to ensure that these deployments are codified as code (ie. infrastructure as code) to streamline this traditional painful part of the machine learning deployment process.



Use Case Diagram

Below are documented the most common use cases as reported during ethnographic studies and interviews with fellow data scientists.

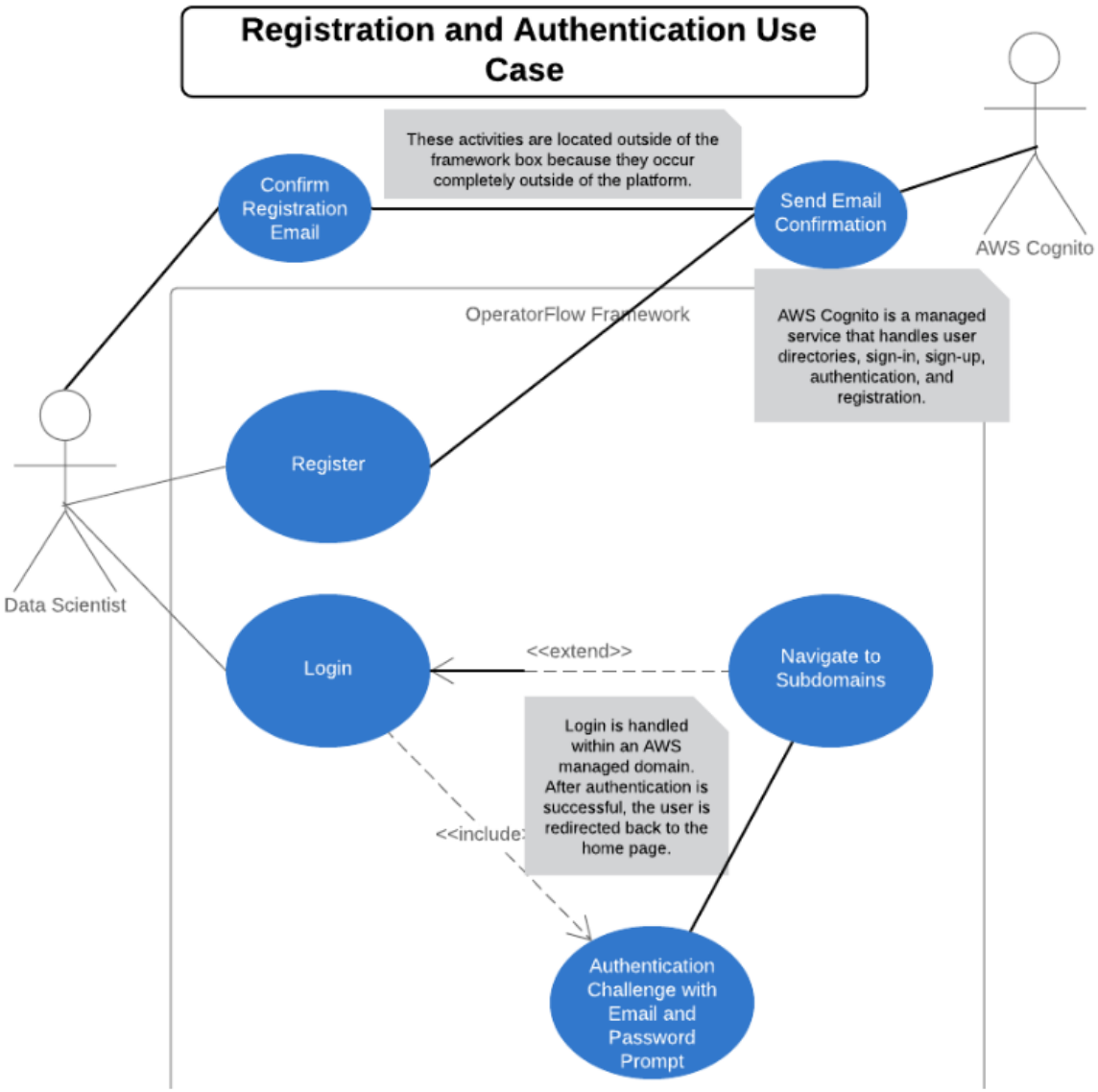
Use Case #0: User Authentication and Registration

Brief Description

The User will access the OperatorFlow domain and be prompted to either register or log in. Registration will involve an email confirmation sent to an email address.

Initial Step-By-Step Description

1. The user navigates to https://www.operatorflow.com.
2. The user select Register, and is taken to a registration page where an email address, password, and password confirmation form are provided.
3. The user can alternatively select to log in by clicking the log in button. This takes the user to a form where he/she will fill in an email address and password. If log in successful, he/she will be redirected to the base domain for OperatorFlow dashboard.
4. If unsuccessful, an error message will be printed.



Use Case Formal Description (from textbook)

System	Control Flow (core authentication and dashboard component of OperatorFlow)
Use Case	User registration and log in
Actors	End user, AWS Cognito JavaScript SDK, AWS Cognito managed service, AWS SES (Simple Email Service)
Data	The user submits his/her username, email, and password via a React component form. This information is packaged up with a request ID and sent to the AWS Cognito endpoint.
Stimulus	End user initiates a registration request by clicking the **Register** form submit.
Response	Response to log in request is an auth object that contains User token, Access token, username, and other metadata related to authentication status.
Comments	Currently, all ID and access tokens for a user's browser session are stored in memory. Often, keys are often kept in local storage, but this has been established as a anti-pattern by Auth0.

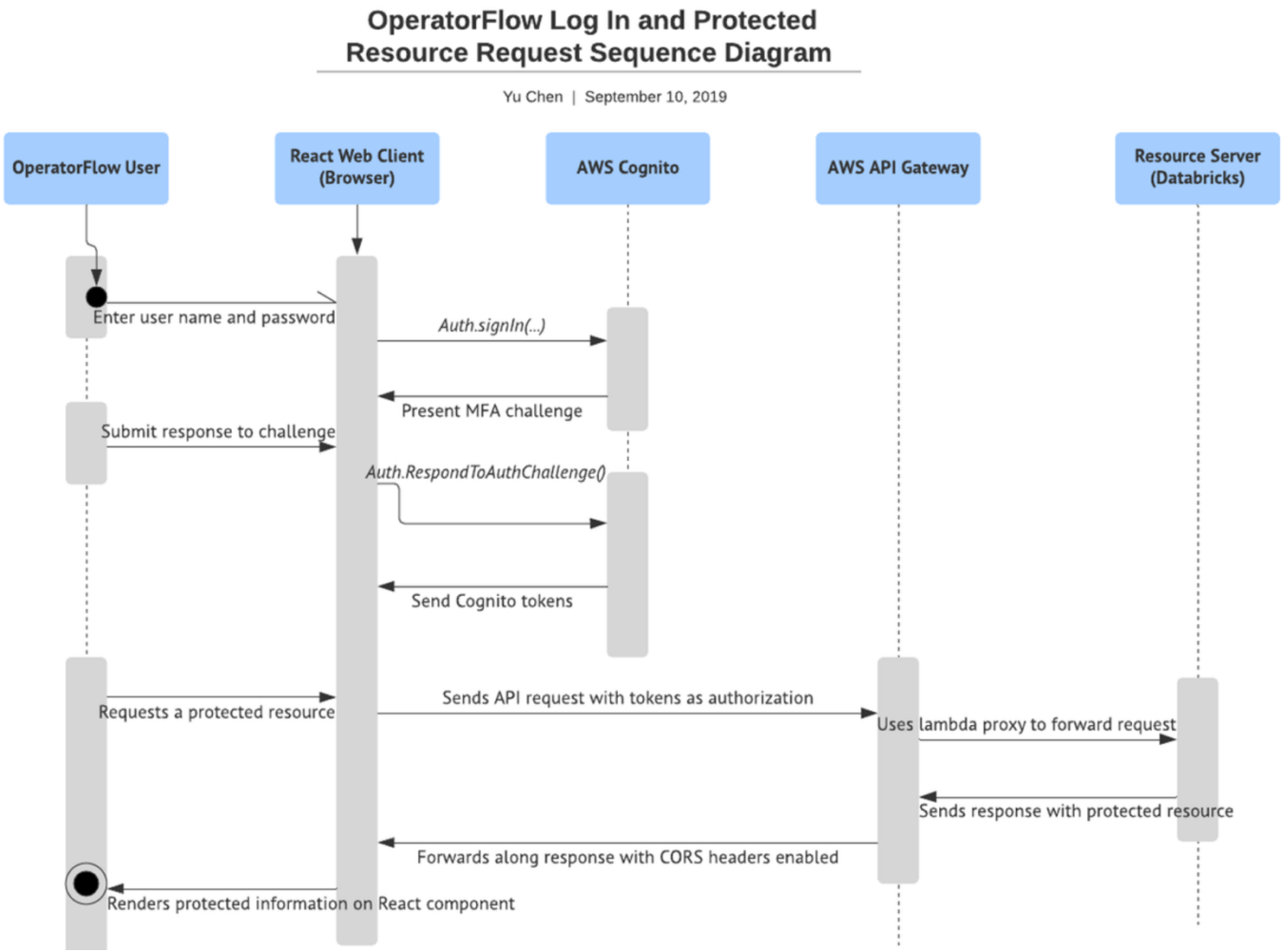
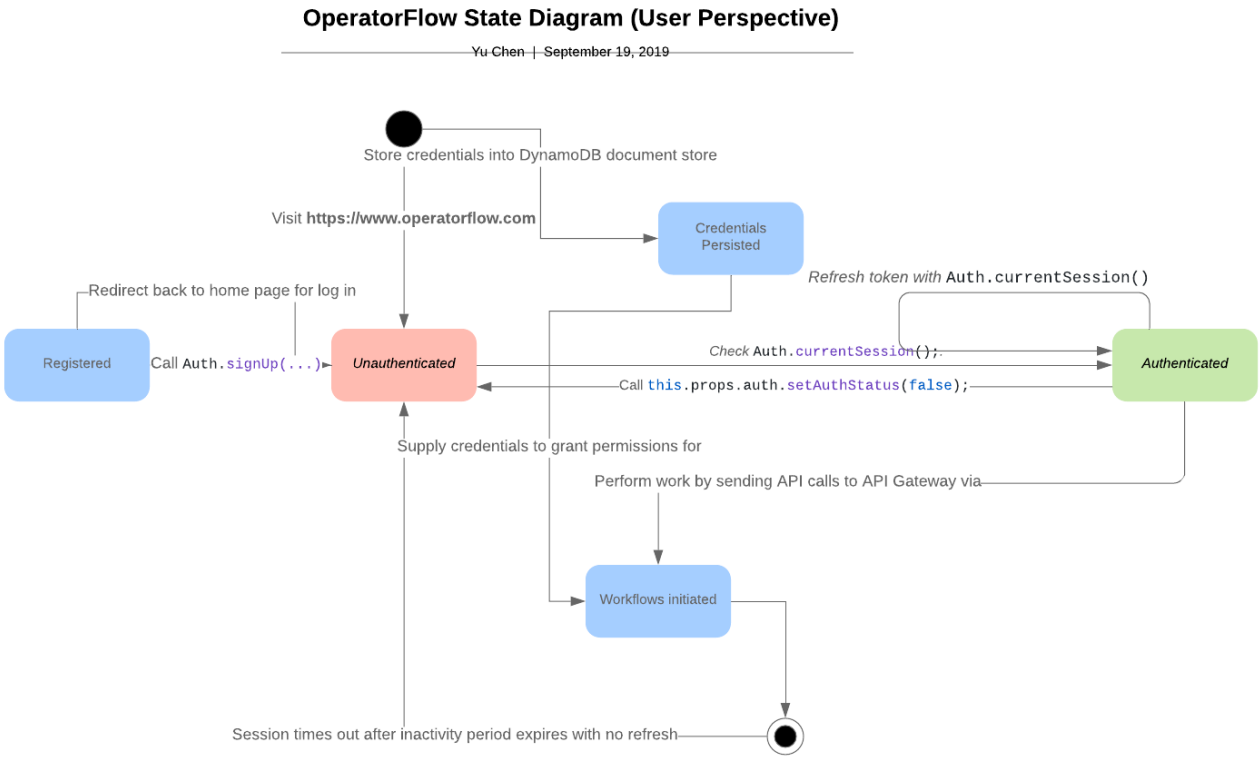


Figure 1 The sequence diagram for user authentication and log in. A preregistered user with AWS Cognito supplies credentials, and the React client calls the AWS Cognito SDK's authentication library to initiate the process.

State Diagram of User/System State During Registration & Authentication



Use Case #1: KubeFlow Deep Learning Model Training

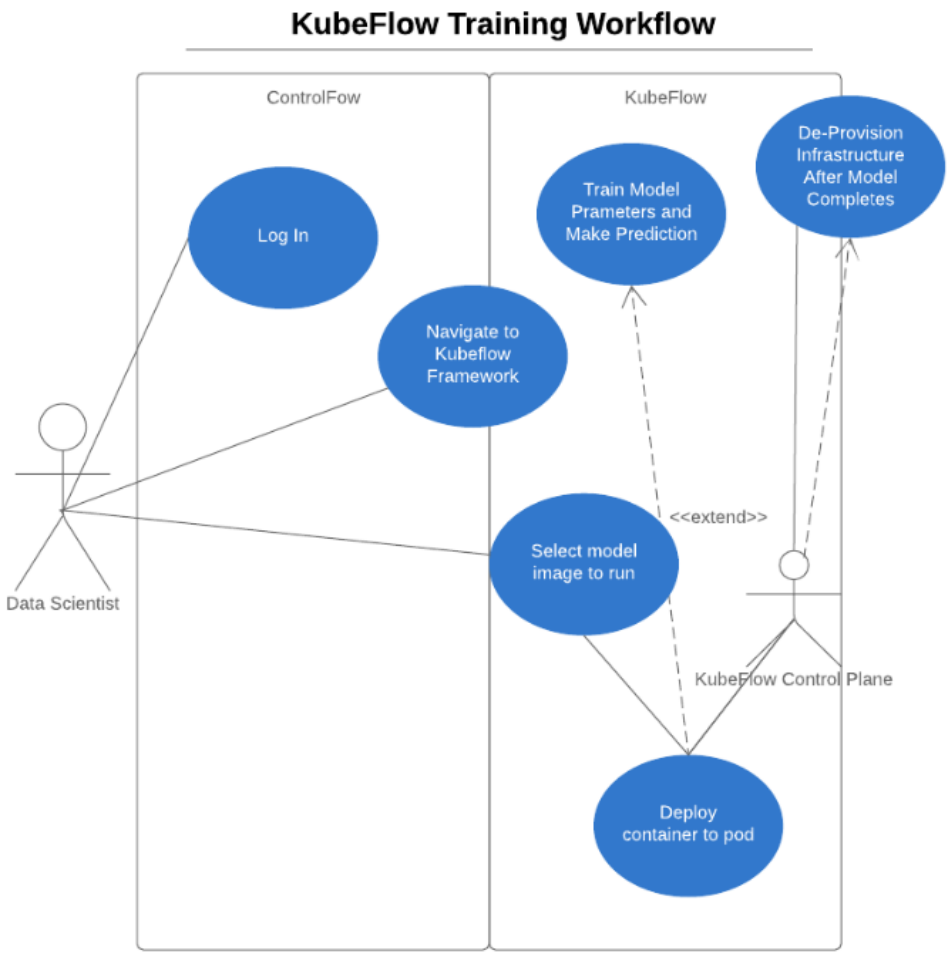
The following steps assume that the user has already authenticated and has been redirected to the OperatorFlow home page.

Brief Description

The user will log into the OperatorFlow framework and then select a known container to run. Each of these containers are batch jobs that have a finite runtime. After the container is run, the model outputs and predictions will be outputted to an object store.

Initial Step-By-Step Description

1. The user navigates to the KubeFlow subdomain, and then the Run Pipeline module. From there, the user can select from a dropdown list of container to run.
2. The user will specify the container to run and object store input field
3. The user can alternatively select to log in by clicking the log in button. This takes the user to a form where he/she will fill in an email address and password. If log in successful, he/she will be redirected to the base domain for OperatorFlow dashboard.
4. If unsuccessful, an error message will be printed.



User Case #2: SparkFlow Notebook

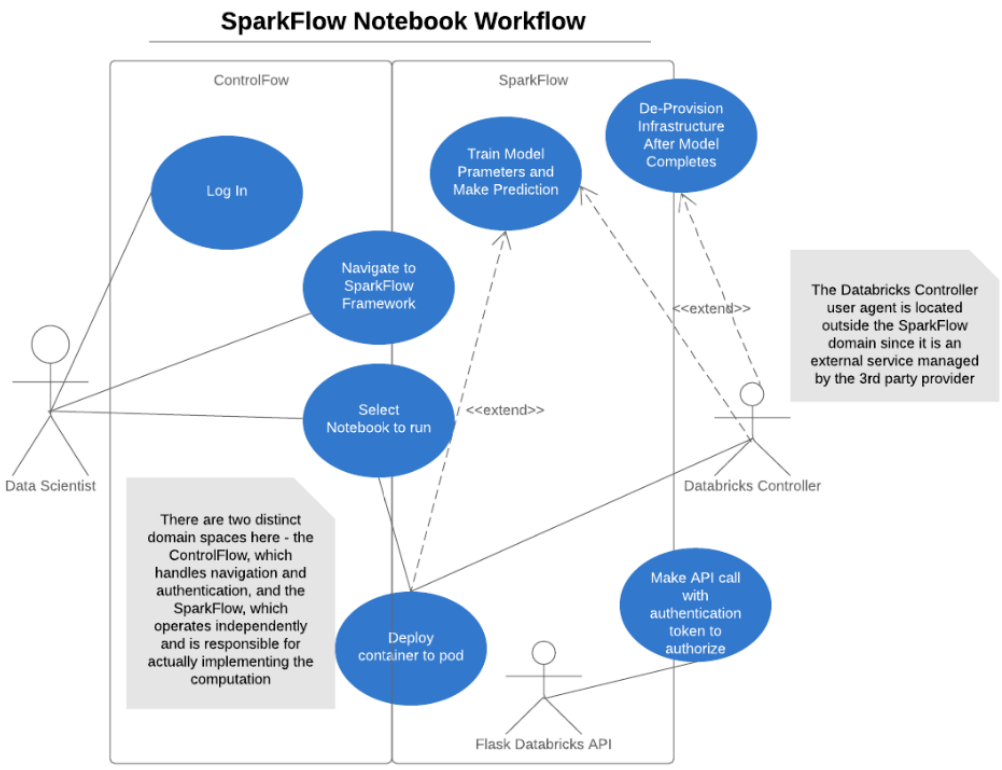
Brief Description

The User will be able to see options for three main activities: visiting the SparkFlow user interface, visiting the KubeFlow framework interface, or uploading and publishing a new model.

Initial Step-By-Step Description

The following steps assume that the user has already authenticated and has been redirected to the OperatorFlow home page.

1. The user navigates to SparkFlow by clicking the SparkFlow tile from the main home page.
2. The user sees a list of available notebooks, and what their current status is (available, currently running).
3. The User can then select a Notebook to run. If they wish to see the status of the job, they can click a refresh button that will refresh the current status of the Job (either **RUNNING** or **AVAILABLE**).



Use Case Formal Description (from textbook)

System	Spark Flow
Use Case	Initialize a scheduled Job that periodically trains a Spark MLLib model
Actors	End user, API Gateway proxy lambda, Databricks Controller
Data	The end user sends an API call to the API Gateway managed service **jobs** resource and endpoint. This API Gateway invokes a lambda function that parses the method and resource, along with any supplemental information carried inside the HTTP payload. The data returned is a status code indicating whether or not the job was correctly scheduled. Any model parameters will be persisted within an S3 bucket or other distributed file storage, but this is implemented internal to Databricks and at a different layer of abstraction than which OperatorFlow works within.
Stimulus	End user initiates request by clicking button on React component, trigger an **axios** RESTful GET or POST call to the API Gateway.
Response	Status code and description indicating success or failure of scheduled machine learning job.
Comments	It's unclear yet the service level agreements (SLAs) provided by Databricks' API. For example, it's not clear yet what level of reliability the API calls will work with – do they succeed 99% of the time? 99.9% of the time?

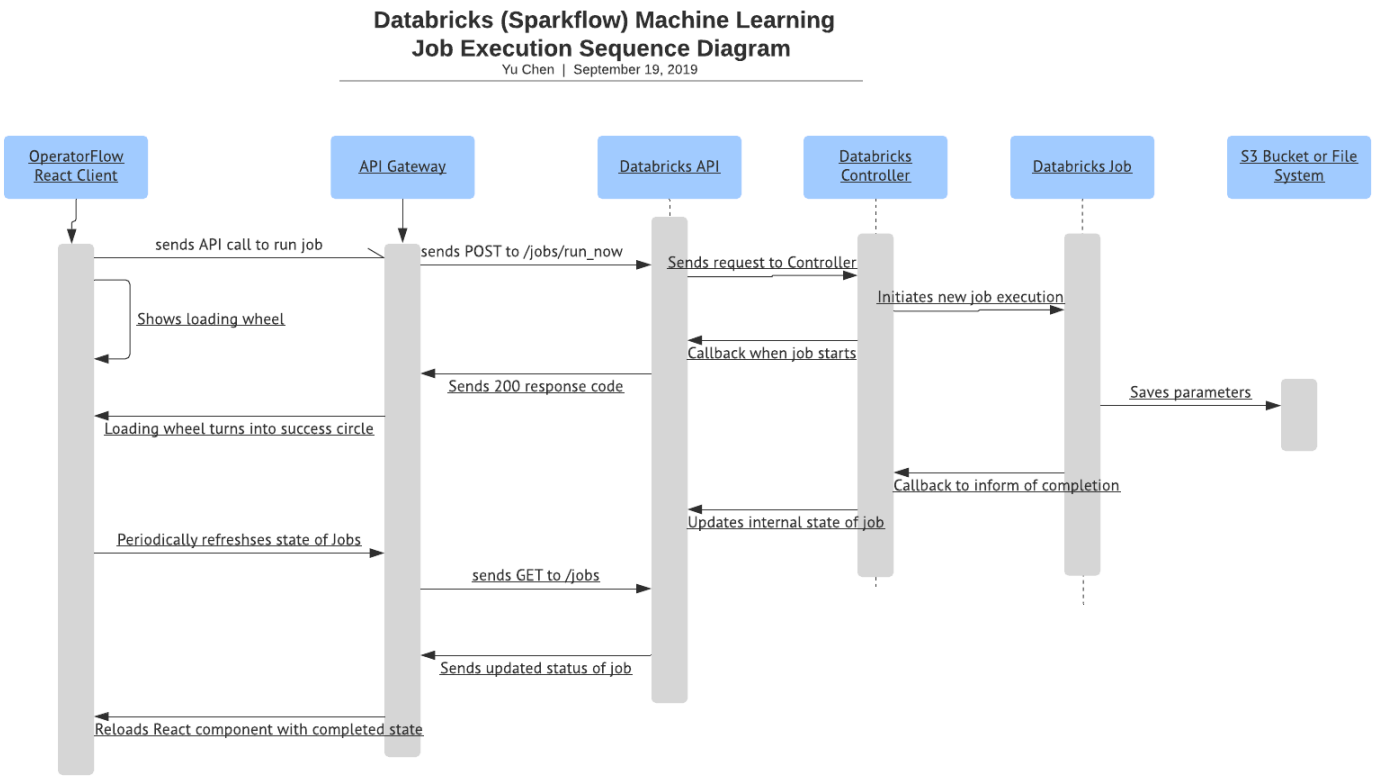


Figure 2Sequence Diagram for SparkFlow training job. Notice that after the job completes, a full callback to the React client is not implemented. The React client must poll for new results periodically to refresh its state.

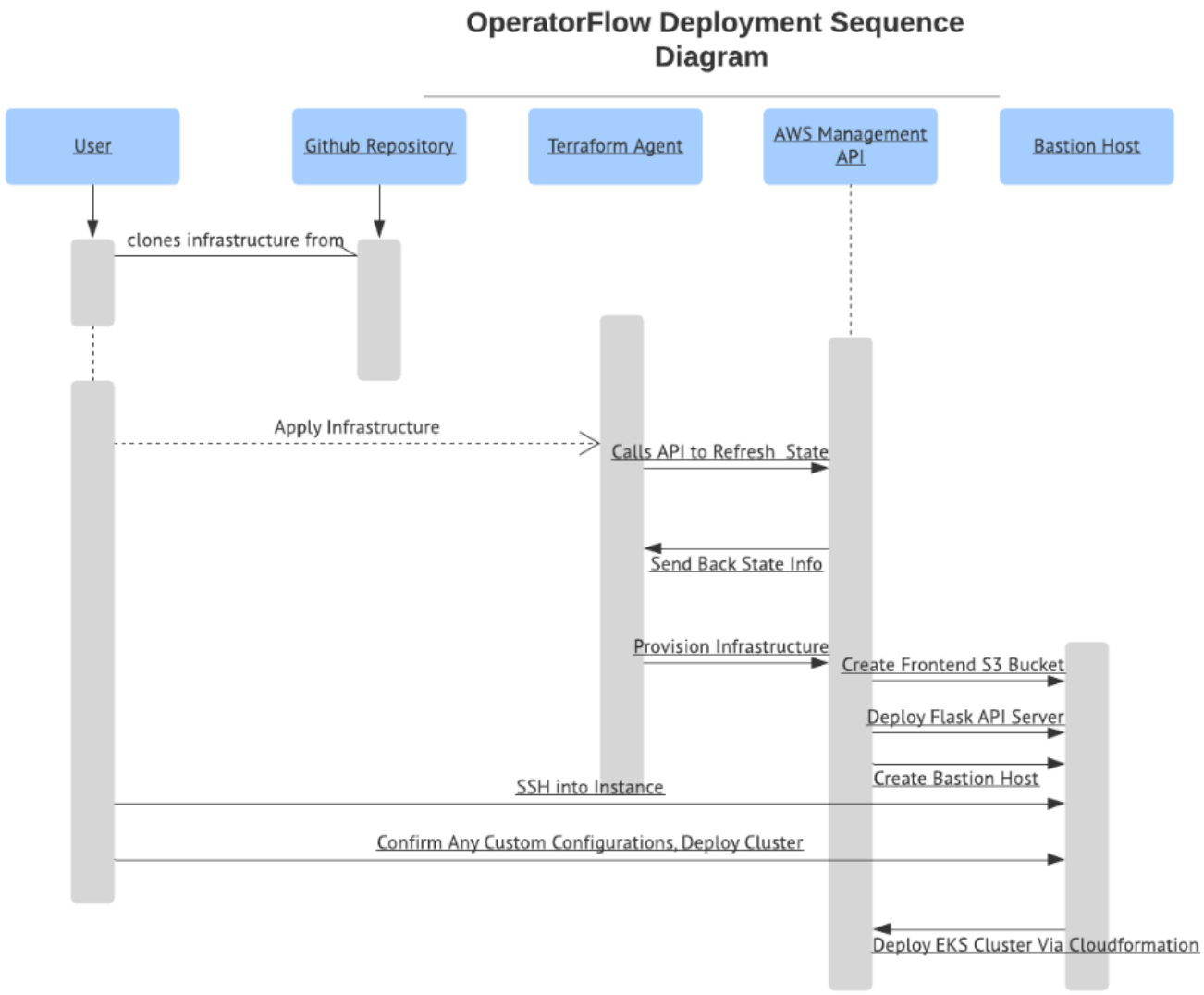
User Case #3: Machine Learning Engineer wishes to deploy a new stack of OperatorFlow

Brief Description

The User (a Data Engineer in charge of deployment) wishes to deploy a new cluster. The use case for this is a result of ethnographic studies where data scientists and data engineers have described a variety of different pain points with very manual deployments.

Initial Step-By-Step Description

1. The User will clone the backend repository to their local desktop.
2. The User will provision the bastion host via Terraform, which will select a customized AMI (Amazon Machine Image) that already has all the tools to deploy KubeFlow and Elastic Kubernetes Service already installed and preconfigured.
3. The Terraform agent will call AWS APIs to deploy the S3 bucket used for hosting the React frontend application, as well as the Flask API server for Databricks Notebook.
4. The User will SSH into the bastion host, make any final configuration changes to the base YAML in the Kubeflow directories he/she desires. Then the user will launch the cluster and infrastructure via a shell script.



Use Case Formal Description (from textbook)

System	KubeFlow
Use Case	Deploy KubeFlow cluster via EKS
Actors	Bastion Host EC2 instance, **eksctl** golang binary, Terraform agent, EKS agent
Data	The end user submits a variety of configuration files that are synthesized by the **eksctl**, **kfctl**, and **kubectl** binaries to declaratively apply and update state: • Terraform files for high-level infrastructure • EKS YAML files for cluster configurations • Kubernetes YAML files for **kfctl** and **kubectl** controllers.
Stimulus	Several stimuli points are used: • Terraform matches state with AWS API • **Eksctl** process, which is used to generate the EKS cluster from scratch first. • **kfctl** generates declarative manifest YAML files to set up Istio service mesh, and then applies these to the Kubernetes API, which posts these as updates via the Kubernetes Controller.

Response	Typical response codes from **kfctl** (Golang binary) and
Comments	It's important to disambiguate that the end user here is not a typical end user of the platform, but rather an OperatorFlow administrator who is in charge of provisioning and administering a Kubernetes cluster.

User Requirements

Note: I have provided a variety of test cases for the highest priority requirements. Each requirement that is highlighted in gray is a **SHALL** mandatory requirement, while the optional requirements are not highlighted and listed as **SHOULD**.

UR 1.1 The framework **shall** execute machine learning training Workflows by receiving input data from an object store such as a S3 bucket and persist learned results and model outputs into another S3 bucket.

- **Test Case:** *User initiates a Taxi Tip Prediction Model Trainer stub model for execution. Assert that execution finishes without any exceptions by checking object store for model results.*

UR 1.2 The framework **shall** automatically provision or de-provision hardware in order to properly execute a workflow. Users should never need to manually increase or decrease compute/memory capacity. This is handled natively via both Databricks and Kubernetes; however, the correct runtime specifications must be passed in at runtime by the Operator. Therefore, users should never receive hardware / performance errors (such as out of memory errors).

- **Test Case:** *In the Databricks Shakespeare stub notebook, switch the amount of data from ~100,000 words to 100,000,000 words, and then execute the Notebook via the SparkFlow API. This assert via the Databricks interface that the notebook executes successfully (in order to do so, it must autoscale its worker instances or it will run into memory management issues).*

UR 1.3 The framework **shall** allow the user to execute both **distributed computation** (parallelization via Apache Spark) and **deep learning** (via Keras) Workflows. A distributed computation workflow request will be handled via API invocation of Databricks notebooks (a third-party service provider). A deep learning workflow request will be handled via the deployment of a Pod on a Kubernetes platform node on **KubeFlow**.

- ***Test Case:*** *Assert that when the user clicks the SparkFlow card link, he/she is redirected to the SparkFlow subdomain.*
- ***Test Case:*** *Assert that when the user clicks the KubeFlow card link, he/she is redirected to the KubeFlow subdomain.*
- ***Test Case:*** *Assert that a control plane cluster node is created within AWS EKS after provisioning of KubeFlow infrastructure.*

UR 1.4 The user **shall** be able to run only workflows once authenticated as a registered user via HTTPS on an internet browser.

- **Test Case:** *Assert that a user who navigates to operatorflow.com via unsecured HTTP receives a connection refused error message.*

UR 1.5 The framework **should** provide the user an interface to package their source code as a Docker image and push it to a remote repository that can be executed within the KubeFlow system.

- **Test Case:** *Assert that the User can select the sample Dockerfile listed in the tests repository of the backend repository.*
- **Test Case:** *Assert that the User can push the container to a repository – after the push event occurs, the image name appears in the Elastic Container Registry on AWS.*

UR 1.6 The framework **should** provide event-based notification via email / text message / Lambda function invocation that is configurable by the user in the form of a YAML file.

- **Test Case:** *Assert that after completion of a Job on either SparkFlow or KubeFlow, an email is sent to the registered email address associated with a particular User.*

UR 1.5 The framework **should** allow the user to specify a series of unit and integration tests in YAML configuration specifically to assert that the shape of learned parameters / matrices conform to expectations, or follow certain defined distributions (such as a standard Gaussian).

System Requirements

User Interfaces

UI 1.1 The app **shall** allow users to list all the currently available jobs / Workflows that can be executed in both SparkFlow and KubeFlow.

- ***Test Case:*** *Assert that the TaxiCab model container image appears in the User dropdown of available models to run on the KubeFlow main component.*
- ***Test Case:*** *Assert that the Shakespeare word count toy notebook appears in the User dropdown of available notebooks to run on SparkFlow's main subdomain React component list.*

UI 1.2 The framework **shall** allow users to execute a machine learning inference workflow by receiving/outputting input directly via a frontend React framework.

UI 1.3 The framework **shall** expose an HTTPS dashboard console that allows users to monitor their queued jobs / event-based workflows.

Hardware Interfaces

Backend Interfaces

BE 1.1 The backend **should** accept model data and configurations via an object store such as AWS S3 or Google Cloud Storage Buckets.

BE 1.2 The framework **should** store user credentials and role assignments within an Encrypted store, either 3rd-party open source such as Vault.

Communication Interfaces

The primary interface for communication between the user and system is in the form of React components and forms. Users will click buttons to select and initiate Jobs within the system.

Nonfunctional Requirements

Performance Requirements

PER 1.1 The frontend component of the framework (the statically hosted React application) **shall** handle 10 users, 1000 users, and 100000 users with the same P95 latency.

PER 1.2 The backend KubeFlow framework **should** be able to run ML jobs within containers that require up to 32 GB of physical storage for data and 16GB of runtime memory.

PER 1.3 The backend SparkFlow framework **should** be able to run ML jobs up to the constraints specified by Databricks and the selected hardware cluster (this is outside the domain of OperatorFlow and configured upon launching of Databricks account).

Safety Requirements

There are no physical safety requirements for this framework. Any security requirements are listed in the following section.

Security Requirements

SER 1.1 All workflows **shall** be initialized by an authenticated user. A user must first register and confirm via email his/her identity prior to usage of the platform.

SER 1.2 If a user does not have an account, he/she **shall** register and confirm via email his/her identity prior to usage of the platform. This includes confirmation by clicking a link in an email.

SER 1.3 All model inputs and outputs (any written from the framework to an S3 bucket) **shall** only be accessed within the **OperatorFlow** framework (ie. public HTTP/S access to objects is forbidden).

SER 1.3 An Operator's default role **should** provides privileges to execute Workflows related only created by that specific user. In order to run or view outputs from other Workflows, these permissions must be explicitly granted.

Vocabulary and Glossary

There are a variety of domain-specific vocabulary terms employed within this implementation. I have divided up each implementation

AWS Deployment

The deployment of networking and compute infrastructure for AWS consists of many DevOps and AWS domain-specific terms:

- **CloudFormation Stack:** a collection of logically coupled AWS resources, such as a VPC, subnets, route tables, and internet gateways, that when organized together represent a meaningful collection of infrastructure.
- **eksctl:** the main provisioning tool used to generate a template EKS cluster configuration. Eksctl is a golang binary supported by Weaveworks, and produced templatized clusters that are easily configurable (by region, autoscaling group sizes, etc.)
- **Node Instance Role:** this is an IAM policy document that specifies a list of resources and actions that the individual worker notes within the EKS cluster are allowed to perform.
- **API Gateway:** a managed service provided by AWS that exposes serverless API endpoints. In this project, I implement this as a RESTful API with the following resources:

- **/jobs:** scheduled executions of workspaces
- **/clusters:** AWS EC2 instance cluster groups available to run compute on
- **/buckets:** S3 bucket file stores available
- **/workspaces**: the individual "applications" to be run on SparkFlow (ie. Databricks notebooks)

Databricks

Databricks' API is a RESTful, well-documented and maintained service that exposes the following resources:

- **Workspace:** this is a REPL-style container of source code that can be executed either as a script or cell by cell. This is the fundamental unit of work in Databricks, and represents an Apache Spark workflow. An example of a Databrick notebook source code is from the screenshot below:

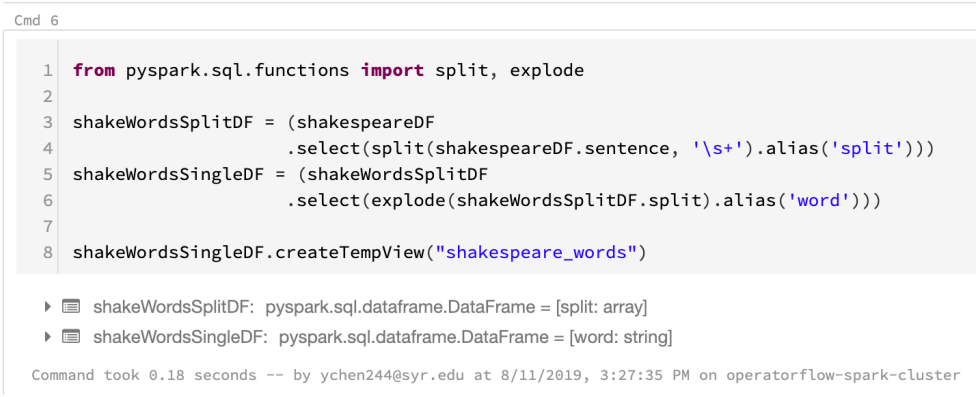


Figure 3 Databricks cell for the operatorflow-spark-cluster notebook. You can see that the command took 0.18 seconds to run and was initiated by user ychen244@syr.edu

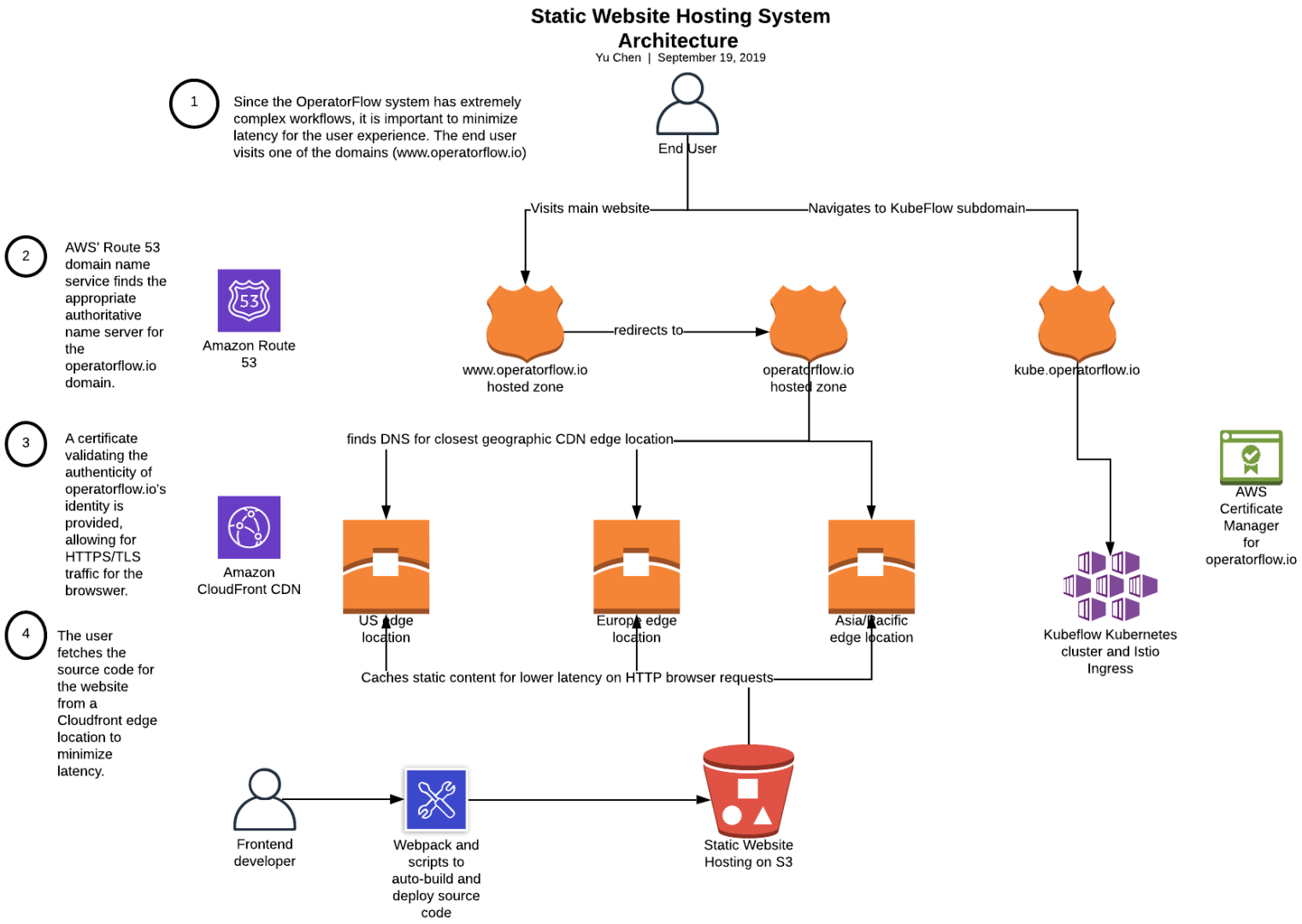
- **Job:** this is a scheduled execution of a **Workspace**
- **Cluster**: the underlying infrastructure (EC2 instances, AWS target groups, auto-scaling, and load balancing) that runs Jobs.

System Architecture

The final system architecture implementation is composed of four distinct pieces:

- Bastion Host
- Frontend Deployment and CDN
- SparkFlow (Databricks integration)
- KubeFlow (Kubernetes cluster runtime)

Frontend Deployment of Static Website with HTTPS Authentication, CDN Edge Locations, and DNS



KubeFlow (Templatized Cloudformation Stack from AWS Console)

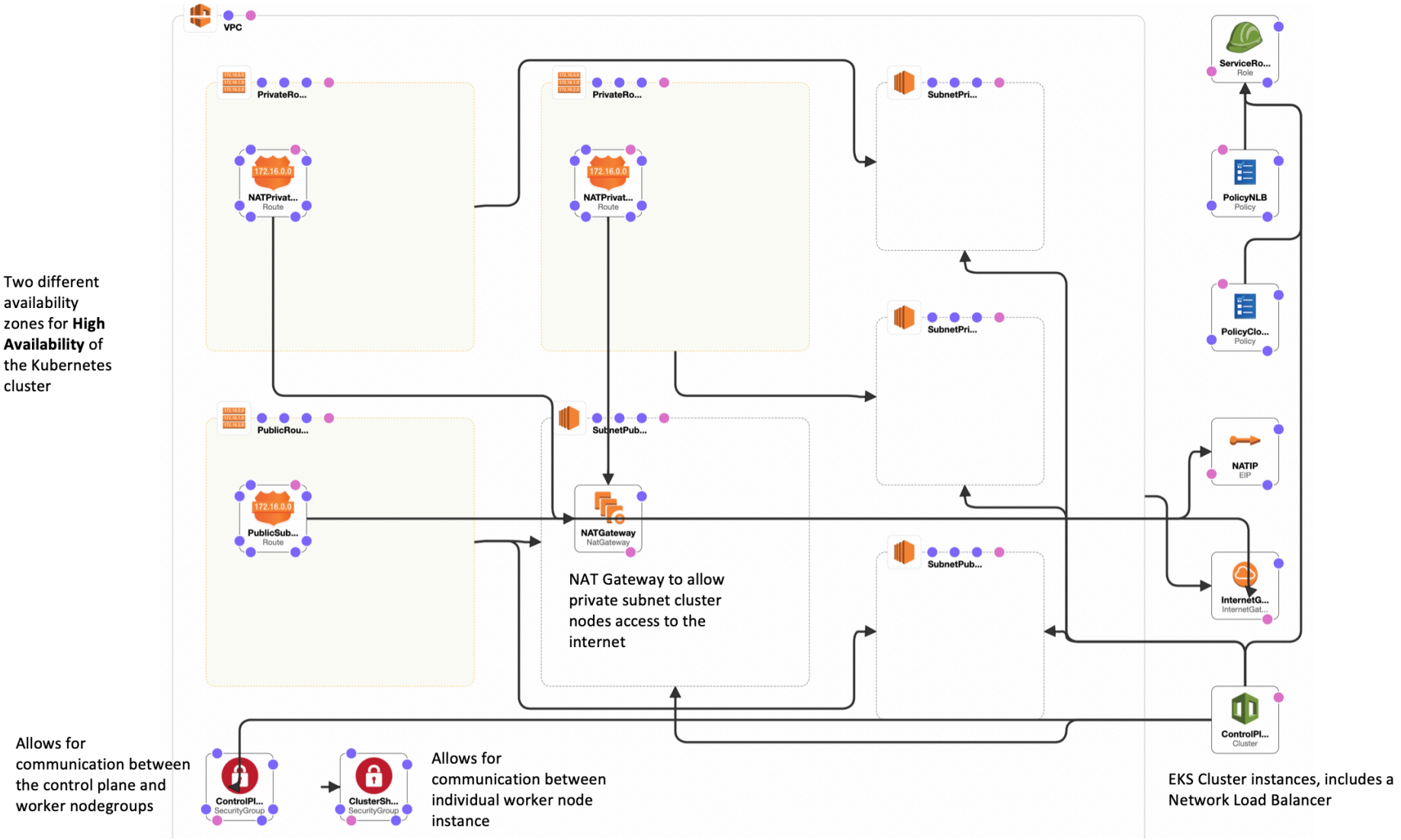
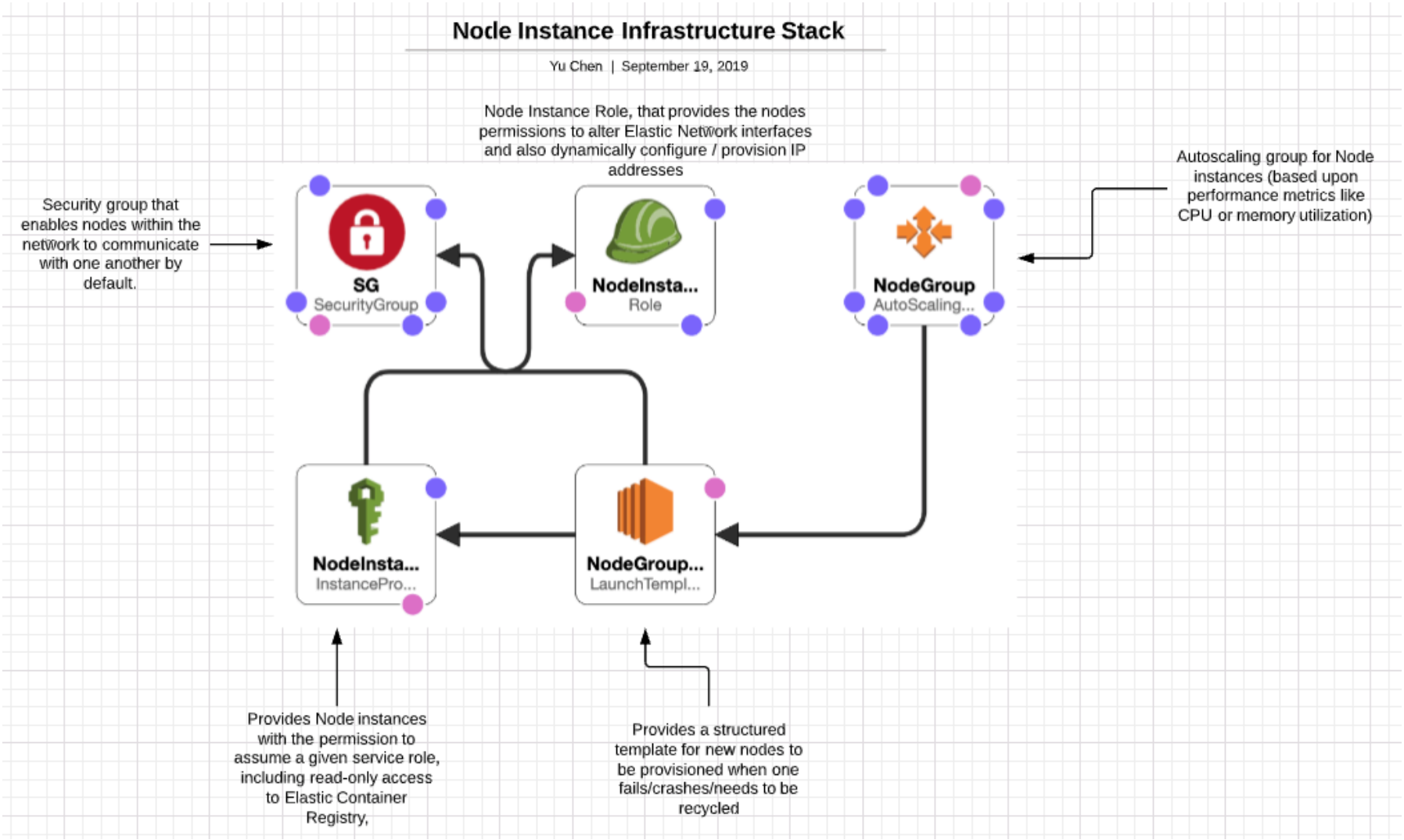


Figure 4 Cloudformation Stack diagram of the EKS cluster infrastructure, with two main availability zones, with public and private subnets and a variety of IAM policies applied for logging and service roles.


Node Instance Infrastructure Stack
Yu Chen | September 19, 2019
Node Instance Role, that provides the nodes permissions to alter Elastic Network interfaces and also dynamically configure / provision IP addresses
Security group that enables nodes within the network to communicate with one another by default.
SG
SecurityGroup
NodeInsta...
Role
NodeGroup
AutoScaling...
Autoscaling group for Node instances (based upon performance metrics like CPU or memory utilization)
NodeInsta...
InstancePro...
NodeGroup...
LaunchTempl...
Provides Node instances with the permission to assume a given service role, including read-only access to Elastic Container Registry,
Provides a structured template for new nodes to be provisioned when one fails/crashes/needs to be recycled

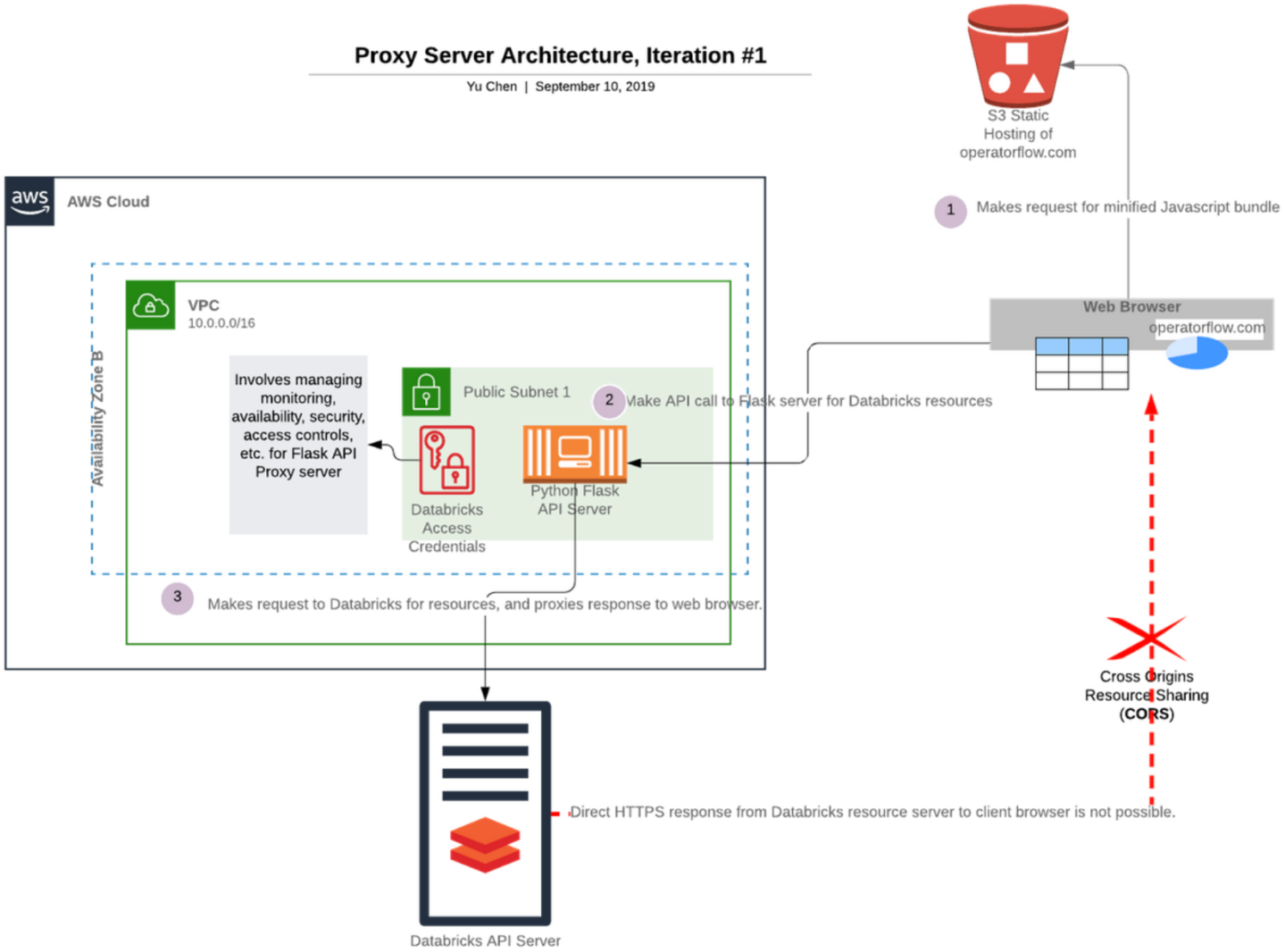


Figure 5 The initial implementation of the API proxy server relied upon a container service instance of a Python Flask API. Databricks credentials were backed into this server instance and it needed to be on continuously to respond to API calls.

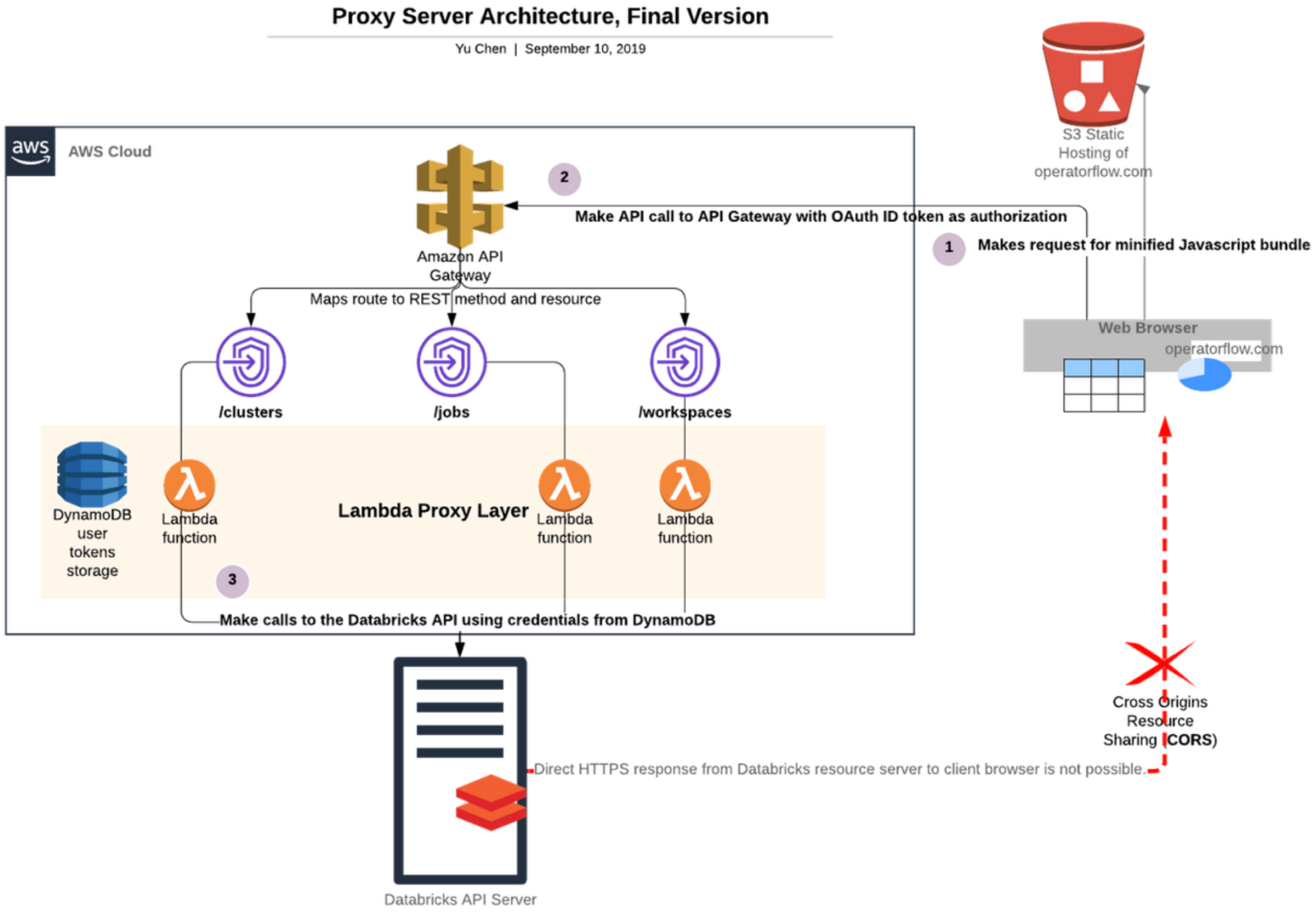
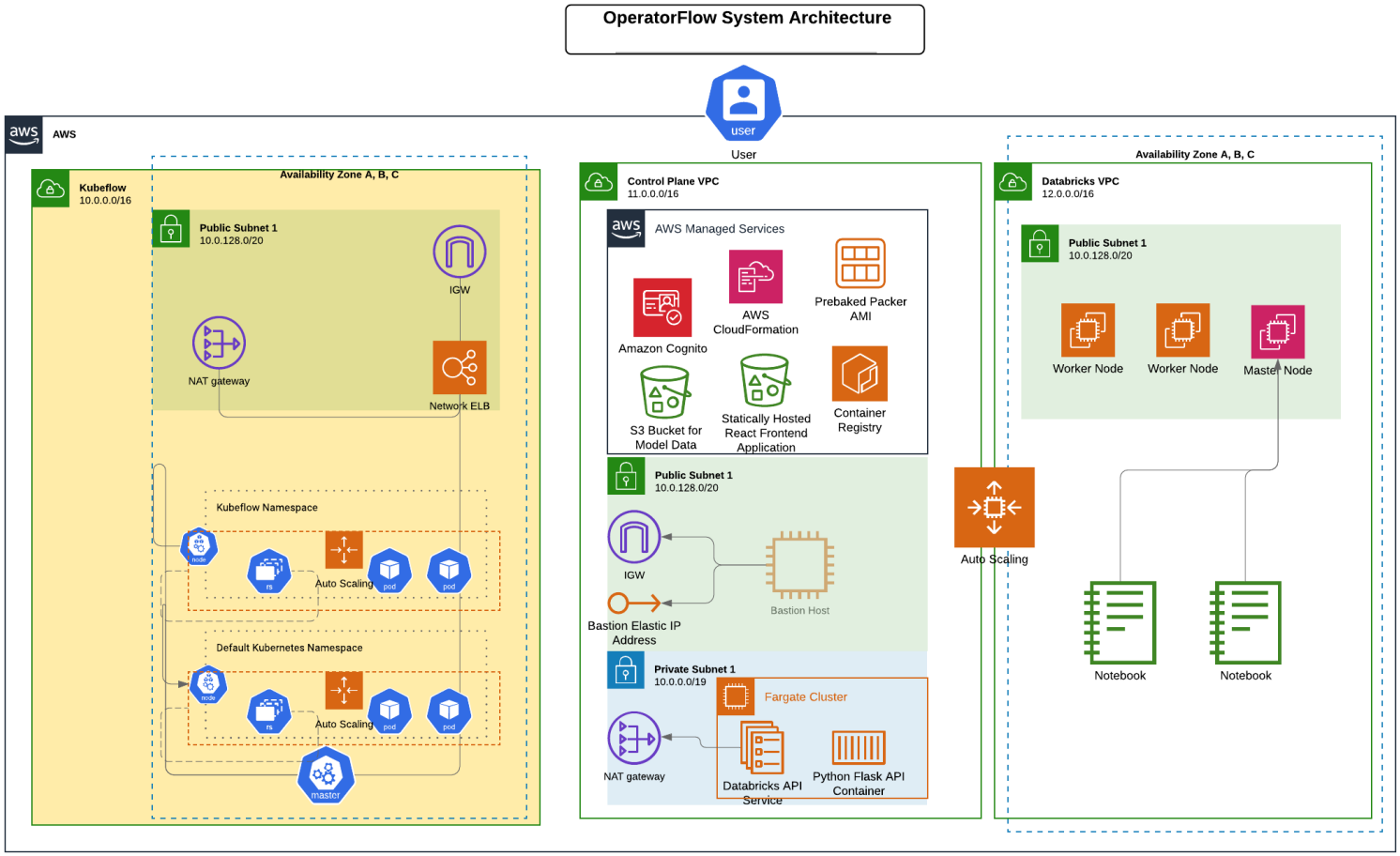


Figure 6 The final API proxy server architecture. This implementation uses serverless API Gateways with Lambda proxies, backed with caching and CORS management.

Appendix A: Network Infrastructure and Architecture



The major components of **OperatorFlow**. On the left, **KubeFlow** - a Kubernetes cluster that is encapsulated within an Istio service mesh and Ingress Controller. On the right, the **SparkFlow** cluster for Databricks Notebook Jobs. In the middle, the ControlFlow architecture, which is currently tied explicitly to AWS resources and managed services. The actual backend serer to initiate calls to the Databricks API is hidden behind a private subnet.