Machine Learning Operations

Chuks Okoli

Last Updated: July 6, 2024

Introduction

Hello there, and welcome to Machine Learning Operations. This work is a culmination of hours of effort to create my reference for machine learning operations. All of the explanations are in my own words but majority of the content are based on Alexey Grigorev's DataTalksClub MLOps Zoomcamp course.

Explaining MLOps to a 5 year old

Imagine you have a magical garden. In this garden, you have a special plant that can grow different kinds of fruits, but you need to take good care of it.

MLOps is like having a magical gardener to help you. This gardener does three important things:

1. ***Teaching the Plant**: The gardener shows the plant pictures of different fruits (like apples, oranges, and bananas) so it can learn to grow them. This is like the gardener helping the plant understand what to do. In MLOps, this is called training a machine learning model.*
2. ***Taking Care of the Garden**: The gardener makes sure the soil is rich, the water is just right, and there are no weeds. The gardener also provides the plant with the best tools and instructions to grow healthy fruits. In MLOps, this is called managing the infrastructure for the machine learning model.*
3. ***Sharing the Fruits**: Once the plant grows the fruits, the gardener picks them and puts them in baskets for everyone to enjoy. The gardener makes sure the fruits are easy to find and delicious. In MLOps, this is called deploying the model so it can be used for something useful.*

Here's the magical part: The gardener watches over the garden to make sure everything is running smoothly. If a bug tries to eat the plant or if the plant stops growing fruits, the gardener fixes the problem right away.

So, MLOps is like having a magical gardener who helps your special plant (machine learning model) stay happy, healthy, and productive, making sure it grows the best fruits for everyone to enjoy!

1.1 What is MLOps?

> *MLOps, also known as DevOps for machine learning, is an umbrella term that encompasses philosophies, practices, and technologies that are related to implementing machine learning lifecycles in a production environment.*
>
> — Microsoft Blog

Machine Learning Operations (MLOps) is a set of best practices for putting machine learning models into production. The process for a machine learning project involves:

- Design - define if machine learning is the right tool for solving the problem
- Train - train the model to find the best possible model
- Operate - deploy the model, and monitor degradation or quality of the model

MLOps is a set of practices for automating everything and working together as a team on a machine learning project.

Fun Fact: MLOps Principles

As machine learning and AI become more common in software, it's important to create guidelines and tools for testing, deploying, managing, and monitoring ML models in real-world use. This is where MLOps comes in. It helps prevent "technical debt" in machine learning projects by ensuring smooth operation and maintenance of models throughout their lifecycle.

MLOps helps to manage and orchestrate the end-to-end machine learning lifecycle by ensuring models are consistently accessible, reproducible, and scalable. It focuses on automating deployment and monitoring of ML pipelines while optimizing the model development process.

The three-phase approach to implementing machine learning (ML) solutions are:

- **Business Understanding and Design**: This phase involves identifying user needs, designing ML solutions to address them, and assessing project develop-

ment. Prioritizing ML use cases and defining data requirements are key steps. The architecture of the ML application, serving strategy, and testing suite are designed based on functional and non-functional requirements.

- **ML Experimentation and Development**: This phase focuses on verifying the applicability of ML by implementing Proof-of-Concept models. It involves iteratively refining ML algorithms, data engineering, and model engineering to deliver a stable, high-quality ML model for production.
- **ML Operations**: Here, the emphasis is on deploying the developed ML model into production using established DevOps practices. Testing, versioning, continuous delivery, and monitoring are essential aspects of this phase.

These phases are interconnected, with design decisions influencing experimentation and deployment options.

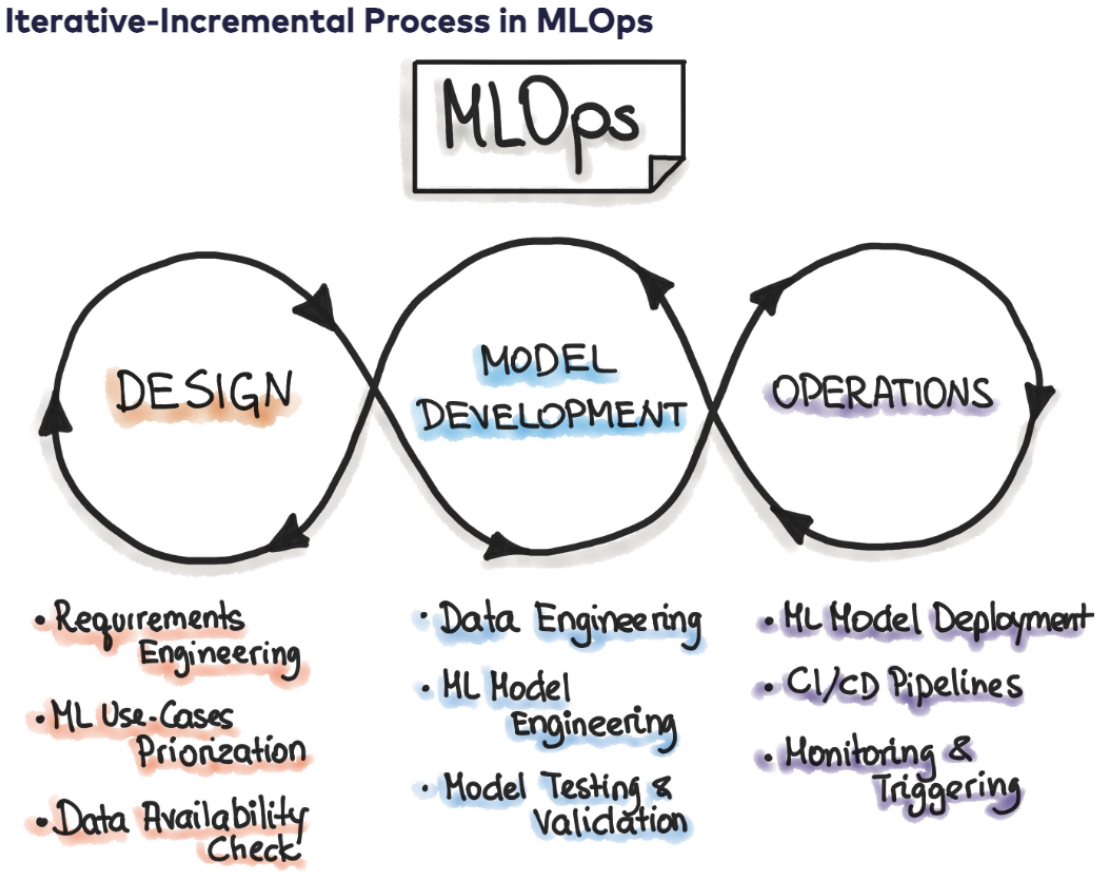

Figure 1.1: The complete MLOps process includes three broad phases of "Designing the ML-powered application", "ML Experimentation and Development", and "ML Operations".

MLOps leverages various tools to simplify the machine learning lifecycle.

- **Machine learning frameworks** like Kubernetes, TensorFlow and PyTorch for model development and training.
- **Version control systems** like Git for code and model version tracking.
- **CI/CD tools** such as Jenkins or GitLab CI/CD for automating model building, testing and deployment.
- **MLOps platforms** like Kubeflow and MLflow manages model lifecycles, deployment and monitoring.
- **Cloud computing platforms** like AWS, Azure and IBM Cloud provide scalable infrastructure for running and managing ML workloads.

1.2 Environment Preparation

1.2.1 Configuring Environment with GitHub Codespaces

To configure the environment using GitHub codespaces, first create a repository on GitHub, give the repository a name, add a "README" file and a `.gitignore` template, choose a license and create the repo. In the repo main page, click on `Create codespace on main` as shown in **Fig. 1.2**.

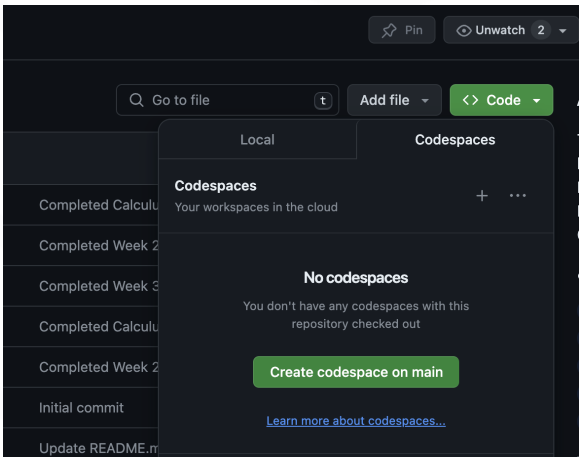

Figure 1.2: GitHub Codespaces setup

A connection to Visual Studio Code can be made through Codespaces. Start a new terminal, change directory to the workspaces and download and install Anaconda distribution of python in it using the step below.

- **Step 1**: Download and install the Anaconda distribution of Python

```
wget https://repo.anaconda.com/archive/Anaconda3-2022.05-Linux-
    x86_64.sh
```

- **Step 2**: Run this command:

```
bash Anaconda3-2022.05-Linux-x86_64.sh
```

After installing Anaconda, initialize it. In a new terminal, confirm that Anaconda is running in the workspaces

```
(base) @your_username -> /workspaces/mlops-zoomcamp-2024 (main) $
   which python
/home/codespace/anaconda3/bin/python
```

- **Step 3**: Make sure to install `pyarrow` in order to download parquet file

```
!pip install pyarrow
```

- **Step 4**: Run jupyter notebook

```
jupyter notebook
```

1.2.2 Configuring Environment with Amazon Web Service (AWS)

To install using AWS, create an account in AWS, go to EC2 and create an instance. Select the OS to use e.g. Ubuntu with 64-bit(x86) architecture. Select the instance type that will be sufficient for your project. Configure the cloud resources and launch. Recommended development environment: Linux

- **Step 1**: Download and install the Anaconda distribution of Python

```
wget https://repo.anaconda.com/archive/Anaconda3-2022.05-Linux-
   x86_64.sh
bash Anaconda3-2022.05-Linux-x86_64.sh
```

- **Step 2**: Update existing packages

```
sudo apt update
```

- **Step 3**: Install Docker

```
sudo apt install docker.io
```

To run docker without sudo:

```
sudo groupadd docker
sudo usermod -aG docker \$USER
```

- **Step 4**: Install Docker Compose
Install docker-compose in a separate directory

```
mkdir soft
cd soft
```

To get the latest release of Docker Compose, go to https://github.com/docker/compose and download the release for your OS.

```
wget https://github.com/docker/compose/releases/download/v2.5.0/
   docker-compose-linux-x86_64 -O docker-compose
```

Make it executable

```
chmod +x docker-compose
```

Add the soft directory to PATH. Open the .bashrc file with nano:

```
nano ~/.bashrc
```

In .bashrc, add the following line:

```
export PATH="${HOME}/soft:${PATH}"
```

Save it and run the following to make sure the changes are applied:

```
source ~/.bashrc
```

- **Step 5**: Run Docker

```
docker run hello-world
```

1.3 Model Development

1.3.1 Taxi trip prediction with machine learning

EXAMPLE 1.1: **TLC Trip Prediction**

In this section, we use machine learning to predict the trip duration for trips in New York. The data used were collected from the NYC Taxi and Limousine Commission (TLC). This dataset contains yellow and green taxi trip records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. The data is stored in the PARQUET format. We built a simple linear regression model and save the model as a pickle file for subsequent use later one. See the duration prediction notebook.

We can download parquet file using:

```
# download parquet data
!wget https://d37ci6vzurychx.cloudfront.net/trip-data/
   green_tripdata_2021-01.parquet
```

1.4 MLOps Maturity Model

The MLOps maturity model helps clarify the Development Operations (DevOps) principles and practices necessary to run a successful MLOps environment. The maturity is the extent to which MLOps is implemented in a team. The MLOps maturity model encompasses five levels of technical capability.

Level	Description	Highlights	Technology
0	No MLOps	• Hard to manage lifecycle • Disparate teams • Systems as "black boxes"	• Manual builds/deployments • Manual testing • No centralized tracking
1	DevOps but no MLOps	• Less painful releases • Limited feedback • Hard to reproduce results	• Automated builds • Automated tests for application code
2	Automated Training	• Managed, traceable training • Easy to reproduce model • Low friction releases	• Automated model training • Centralized tracking of model training performance • Model management
3	Automated Deployment	• Releases are low friction and automatic • Full traceability from deployment back to original data • Entire environment managed: train > test > production	• Integrated A/B testing • Automated tests for all code • Centralized tracking
4	Full MLOps Automated Operations	• Fully automated, monitored • Systems provide improvement information • Zero-downtime	• Automated training/testing • Centralized metrics

Table 1.1: Machine Learning operations maturity model (culled from Microsoft Blog

Experiment Tracking and Model Management

> *MLOps is critical for the success of AI projects because it allows teams to iterate quickly and deploy machine learning models reliably and at scale.*
>
> — Andrew Ng

2.1 Introduction to Experiment Tracking

Fun Fact: MLflow

MLflow is an open-source platform that helps manage the end-to-end machine learning lifecycle. It provides a set of tools to streamline and automate various stages of the machine learning process, from experimentation to deployment.

Machine Learning experiment is the process of building an ML model. Experiment run represents each trial in an ML experiment. A run artifact is any file associated with an ML run. The experiment metadata is all the information related to the experiment. *Experiment tracking* is the process of keeping track of all the **relevant information** from an **ML experiment**, which includes:

- Source code
- Environment
- Data
- Models
- Hyperparameters
- Metrics

Why is Experiment Tracking so important

Experiment tracking is important because of **Reproducibility**, **Organization**, and **Optimization**

Experiment tracking is done using MLflow. MLflow is "an open source platform for the machine learning lifecycle". In practice, it's just a Python package that can be installed with pip, and it contains four main modules:

- Tracking
- Models
- Model Registry
- Projects

The MLflow Tracking module allows you to organize your experiments into runs, and to keep track of:

- Parameters
- Metrics
- Metadata
- Artifacts
- Models

Along with this information, MLflow automatically logs extra information about the run:

- Source code i.e., the name of the file that was used to run the experiment
- Version of the code (git commit)
- Start and end time
- Author

2.1.1 How MLflow Helps Manage Machine Learning Experiments

Here are some key ways in which MLflow can help with machine learning experiments:

Experiment Tracking

MLflow Tracking allows you to log and query experiments using APIs or a web-based interface. Key features include:

- **Log Parameters and Metrics:** Easily log hyperparameters, metrics, and artifacts (such as model files) for each run.
- **Organize and Compare Runs:** Organize runs by experiments and compare results visually.

- **Search and Query:** Search and filter experiments using a web UI or API, allowing you to quickly find runs with specific attributes.

Reproducibility

MLflow promotes reproducibility of experiments by providing:

- **Code Versioning:** Integrates with version control systems (like Git) to log the version of the code that produced a particular run.
- **Environment Management:** Capture and reproduce the software environment using Conda environments or Docker images.
- **Artifacts:** Store and retrieve artifacts such as datasets, models, and images, ensuring all elements of an experiment are preserved.

Model Management

MLflow Models facilitate model packaging, sharing, and deployment:

- **Standardized Format:** Save models in a standardized format that includes the model itself along with its dependencies and environment.
- **Multi-Platform Support:** Export models to various formats (e.g., TensorFlow, PyTorch, scikit-learn) and deploy them to different environments (e.g., cloud services, Docker).
- **Model Registry:** Register and version models, track model lineage, and transition models through stages (e.g., "staging", "production").

Deployment

MLflow provides tools for deploying models to various platforms:

- **Built-in Deployment Options:** Deploy models to cloud platforms like AWS SageMaker, Azure ML, and Google Cloud ML Engine.
- **Custom Deployments:** Create custom deployment logic using the MLflow REST API or the command-line interface (CLI).
- **Batch and Real-Time Serving:** Support for both batch and real-time serving of models, enabling various deployment scenarios.

Collaboration

MLflow facilitates collaboration among team members by providing:

- **Centralized Tracking Server:** A centralized tracking server where team members can log and view experiment runs.

- **Sharing and Collaboration:** Share results, models, and insights easily within the team, fostering better collaboration and knowledge sharing.
- **Integration with CI/CD:** Integrate MLflow with continuous integration and continuous deployment (CI/CD) pipelines for automated testing and deployment of models.

Scalability

MLflow is designed to scale with your needs:

- **Distributed Tracking:** Support for tracking experiments across distributed environments, making it suitable for large-scale machine learning projects.
- **Flexible Storage Options:** Use various backend storage systems for logging data, including file systems, databases, and cloud storage.

By integrating MLflow into your machine learning workflow, you can enhance the management, reproducibility, and deployment of your experiments. It provides a comprehensive set of tools that streamline the entire lifecycle of machine learning models, making it easier to track, reproduce, and deploy models at scale.

2.2 Experiment tracking with MLflow

To start experiment tracking, we need to create a conda environment for tracking experiment and activate it.

```
conda create -n exp-tracking-env python=3.9
conda activate exp-tracking-env
```

Once the environment is activated, install the "requirements.txt" file using:

```
pip install -r requirements.txt
```

Check the installed packages using:

```
pip list
```

To launch mlflow, and store all the artifacts in an sqlite database, we run:

```
mlflow ui --backend-store-uri sqlite:///mlflow.db --port 5001
```

If you encounter error when launching mlflow, use:

```
ps -A | grep gunicorn
```

to view the processes using the port and kill them with:

```
kill <process number>
```

before re-launching mlflow.

To access mlflow ui, open in your browser. The mlflow interface is seen in **Fig. 2.1**.

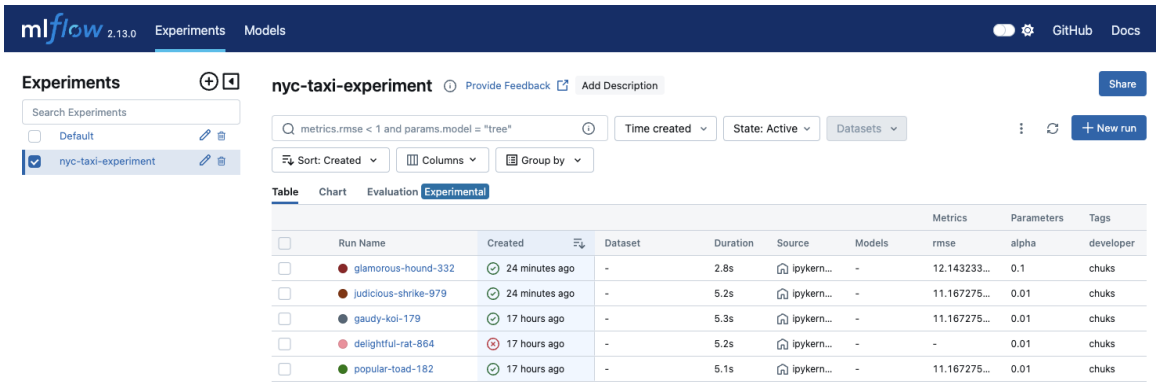

Figure 2.1: MLflow Experiment Interface

The updated notebook with experiment tracking using MLflow and logged predictions in MLflow UI is shown in this notebook for experiment tracking with mlflow.

2.3 Machine Learning Lifecycle

> FUN FACT: **MLOps cycle**
>
> The Machine Learning lifecycle refers to the multiple steps that are needed to build and maintain a machine learning model.

In Machine Learning Lifecycle, we train some model, we tuned the hyperparameters, we evaluated the model and then logged some metrics, hyperparameters and other information needed to mlflow. Once we finish with this experiment tracking stage, it means that we are happy with the model. Then we can start thinking of saving this model and have some kind of versioning. After that, we would like to deploy the model. We may realize that the model needs to be updated in order to scale. Finally, once we deploy the model, the prediction monitoring stage starts.

2.3.1 Logging models in MLflow

Two options :

- Log model as an artifact

```
mlflow.log_artifact("<mymodel>", artifact_path="models/")
```

- Log model using the method “log_model”

```
mlflow.<framework>.log_model(model, artifact_path="models/")
```

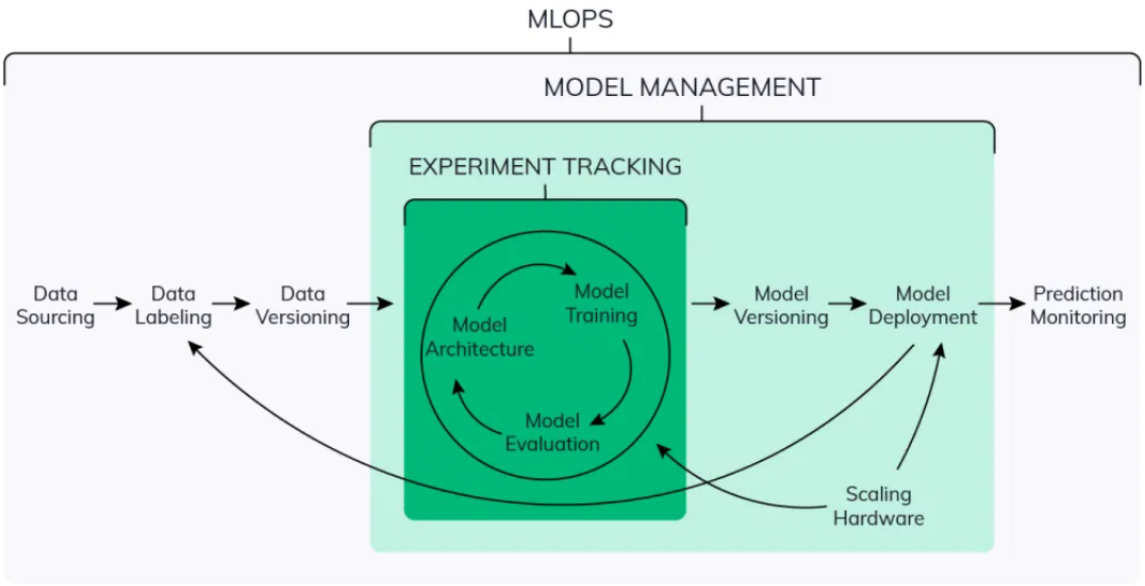

Figure 2.2: MLOps cycle and machine learning experiment tracking

2.3.2 Model Management

Model management is a critical aspect of the machine learning lifecycle, encompassing the processes and tools used to effectively organize, version, and track machine learning models. Effective model management ensures that models can be deployed, monitored, and updated seamlessly, maintaining their performance and relevance over time. The main components of model management:

1. Versioning

- **Importance:** Versioning allows data scientists to keep track of different iterations of a model, ensuring that they can reproduce results and compare performance across versions.
- **Tools:** Tools like MLflow, DVC, and Git provide functionalities to version models, track changes, and manage model metadata.

2. Deployment

- **Purpose:** Deployment involves taking a trained model and making it available for inference in a production environment.
- **Methods:** Models can be deployed as a python function, in a docker container, as an APIs, embedded in applications, as a batch job in Apache Spark or integrated into data pipelines. Platforms like Kubernetes, AWS SageMaker, and Azure ML facilitate seamless deployment.

3. **Monitoring**

- **Need:** Continuous monitoring of models in production is essential to ensure they perform as expected and to detect any performance degradation or data drift.
- **Metrics:** Key metrics to monitor include accuracy, latency, throughput, and error rates. Tools like Prometheus, Grafana, and custom logging solutions can be used for monitoring.

4. **Retraining**

- **Process:** As new data becomes available, models may need to be retrained to maintain their performance.
- **Automation:** Automated retraining pipelines can be set up to periodically retrain models, incorporating the latest data and ensuring that the model remains up-to-date.

5. **Governance and Compliance**

- **Compliance:** Ensuring that models comply with regulatory standards and organizational policies is crucial, especially in industries like finance and healthcare.
- **Documentation:** Proper documentation and audit trails are necessary for compliance and to provide transparency into how models are developed and used.

6. **Collaboration**

- **Teamwork:** Effective model management facilitates collaboration among data scientists, engineers, and business stakeholders.
- **Tools:** Collaborative tools like Jupyter Notebooks, GitHub, and MLflow allow teams to share code, experiments, and insights.

Benefits of Effective Model Management

- **Reproducibility:** Ensures that models can be consistently reproduced, which is crucial for validation and debugging.
- **Scalability:** Facilitates the scaling of machine learning efforts across an organization.
- **Efficiency:** Streamlines the deployment and monitoring processes, reducing time to market for new models.
- **Compliance:** Helps maintain compliance with legal and regulatory requirements.

In summary, model management is essential for maintaining the lifecycle of machine learning models, from development through deployment and monitoring, ensuring they continue to deliver value and perform optimally in production environments.

2.3.3 Model Registry in Machine Learning

Why Model registry

Model registry is an essential tool for managing the lifecycle of machine learning models, ensuring that they are well-organized, reproducible, and efficiently deployable, while enhancing collaboration and compliance within an organization.

A model registry is a centralized repository that stores and manages machine learning models. It plays a crucial role in the machine learning lifecycle by providing a systematic way to organize, version, and track models. The key components and benefits of a model registry:

1. **Versioning**
   - **Purpose:** Keeps track of different iterations and versions of a model.
   - **Functionality:** Allows comparison of model performance over time and ensures reproducibility.
2. **Metadata Management**
   - **Purpose:** Stores important information about models, such as hyperparameters, training data, and evaluation metrics.
   - **Functionality:** Facilitates model auditability and governance.
3. **Lifecycle Management**
   - **Purpose:** Manages the lifecycle of models from development to deployment and monitoring.
   - **Functionality:** Ensures smooth transitions between different stages of the model lifecycle.
4. **Access Control**
   - **Purpose:** Defines who can access and modify models in the registry.
   - **Functionality:** Enhances security and ensures that only authorized users can make changes.
5. **Deployment Integration**
   - **Purpose:** Facilitates the deployment of models to production environments.
   - **Functionality:** Provides integration with deployment tools and platforms for seamless model serving.

As we continue to generate more models, we need a model registry consisting of a tracking server to keep track of the models. Once you've decided that some of the model are ready for production, you then register the model into the mlflow model registry. Within the model registry, we have a staging , production and archive area for the models. In the model registry, all the models that are ready for production are stored here.

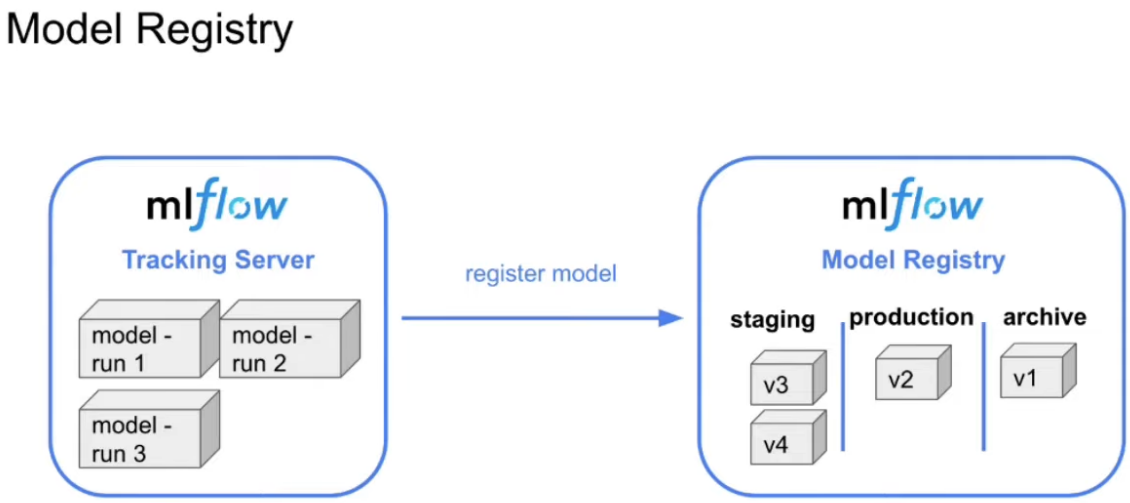

Figure 2.3: Model Registry

The MLOps Engineer can look at the models in the registry, inspect their hyperparameters, the size of the model and so on. He can then decide to move the models between the different stages. The model that is ready for production can be assigned to "staging", the current model that is used in production can be assigned to the "production" stage, and some of the models can be archived in the "archived" stage. The archived models can be retrieved from the archived stage and move them back to production if we want to roll back some deployments. The model registry is not deploying any model, it just lists the models that are production ready and the stages are just labels assigned to the model. The model registry will need to be complemented with some CI/CD code in order to do the actual deployment of the models.

Using MlflowClient

`mlflow.client` is a module in MLflow that provides a way to interact programmatically with an MLflow tracking server. The primary class in this module is MlflowClient, which provides a comprehensive API for managing experiments, managing runs, models, logging artifacts, and interacting with model registry.

2.3.4 Interacting Programmatically with MLflowClient

The `mlflow.client` module is useful for advanced use cases where you need programmatic control over MLflow entities, especially in production environments where automated workflows and integrations are required.

Here's a basic example demonstrating how to use `MlflowClient`:

```python
import mlflow
from mlflow.tracking import MlflowClient

# Initialize the client
client = MlflowClient()

# Create an experiment
experiment_id = client.create_experiment("My New Experiment")

# Start a new run in the experiment
run = client.create_run(experiment_id)
run_id = run.info.run_id

# Log parameters, metrics, and tags
client.log_param(run_id, "param1", 5)
client.log_metric(run_id, "metric1", 0.89)
client.set_tag(run_id, "tag1", "value1")

# Log an artifact (a file)
with open("output.txt", "w") as f:
    f.write("Hello, world!")
client.log_artifact(run_id, "output.txt")

# End the run
client.set_terminated(run_id)

# Get information about the run
run_info = client.get_run(run_id)
print(run_info)
```

Listing 2.1: Example Usage of `MlflowClient`

Key Methods of `MlflowClient`

1. Experiment Management

- `create_experiment(name, artifact_location=None, tags=None)`: Creates a new experiment.
- `get_experiment(experiment_id)`: Retrieves an experiment by ID.
- `delete_experiment(experiment_id)`: Deletes an experiment.

2. Run Management

- `create_run(experiment_id, start_time=None, tags=None)`: Starts a new run.
- `log_param(run_id, key, value)`: Logs a parameter.
- `log_metric(run_id, key, value, timestamp=None, step=None)`: Logs a metric.
- `set_tag(run_id, key, value)`: Sets a tag.
- `log_artifact(run_id, local_path, artifact_path=None)`: Logs an artifact.

3. Model Registry Management

- `create_registered_model(name)`: Creates a new registered model.
- `create_model_version(name, source, run_id, tags=None, run_link=None, description=None)`: Creates a new model version.
- `transition_model_version_stage(name, version, stage)`: Transitions a model version to a new stage.

2.4 MLflow in Practice

2.4.1 Different scenarios for running MLflow

Let's consider these three scenarios:

- **`A single data scientist participating in an ML competition`**: In this scenario, having remote tracking server will be an overkill. Saving this information locally will be enough. Also, using model registry is useless since the data scientist is not deploying this model to production.
- **`A cross-functional team with one data scientist working on an ML model`**:Here, sharing the experiment information is important. Also, using model registry will be a good idea but it can be run remotely or local host.
- **`Multiple data scientists working on multiple ML models`**: Since multiple data scientist are working on multiple model, collaboration and sharing experiment information is very important. One data scientist can build the model, another data scientist can tune different hyperparameters to add to the model. They need a way to keep track of the models using a remote tracking server. Also, it is important to manage the lifecycle of the model since multiple people build and deploy the model hence model registry is important.

2.4.2 Things to Consider before Configuring MLflow

There are different things to consider before configuring MLflow.

- **Backend Store**: Where MLflow saves information about your experiement such as metadata, models etc
  - local filesystem
  - SQLAlchemy compatible DB (e.g. SQLite)
- **Artifacts Store**: Decide where to store the artifact i.e., locally or remote
  - local filesystem
  - remote (e.g. s3 bucket)
- **Tracking Server**: Decide how to run tracking server
  - no tracking server
  - localhost
  - remote

2.4.3 Setting Up MLflow Tracking Server in AWS

Assuming we have multiple data scientist working on multiple ML models. We can setup a remote tracking server on Amazon Elastic Compute Cloud (Amazon EC2) with PostgreSQL backend using Amazon Relational Databse Service (RDS) and Amazon Simple Storage Service (S3) to store the model and artifacts. MLflow uses the backend store and artifact store for persistent runs. The PostgreSQL backend stores metadata such as parameters, metrics, and tags while the artifact store holds large files such as serialized models and config files. MLflow experiments can be explored by accessing the remote server. To run experiments using AWS, we have to configure an AWS account.

Basic AWS Setup

- **Step 1**: Launch a new EC2 instance
  For this, you can select one of the instance types that are free tier eligible. For example, we will select an Amazon Linux OS (`Amazon Linux 2 AMI (HVM) - Kernel 5.10, SSD Volume Type`) and a `t2.micro` instance type, which are free tier eligible.

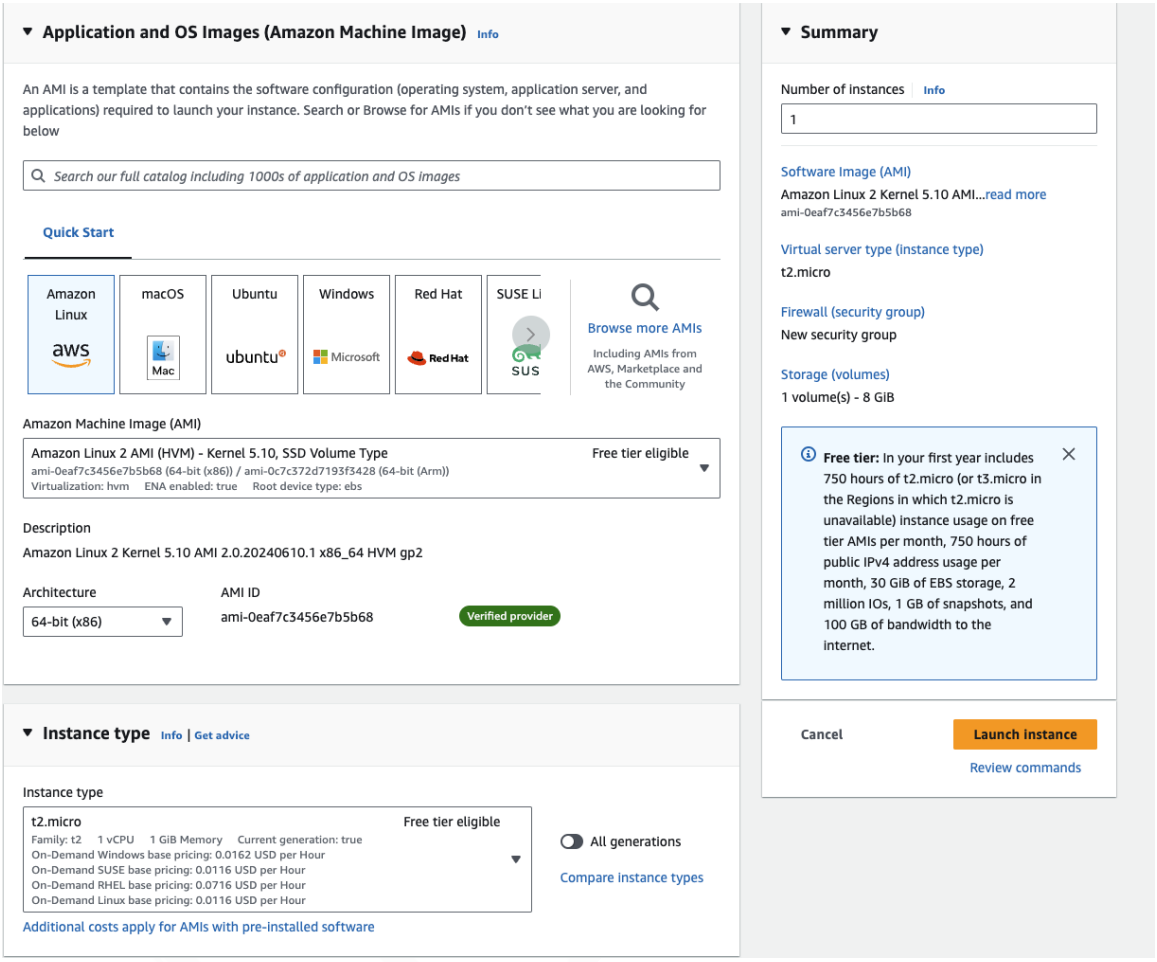

Figure 2.4: Application and OS Setup

You'll also need to create a new key pair so later you can connect to the new instance using SSH. Click on "Create new key pair" and complete the details like in the image below:

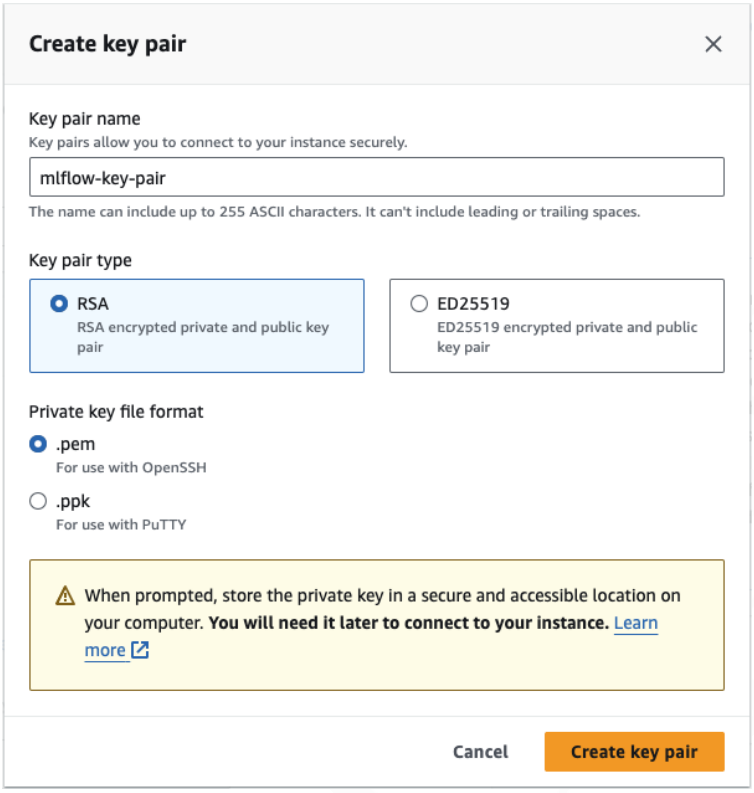

Figure 2.5: Create Key Pair

Select the new key pair and then click on "Launch Instance".

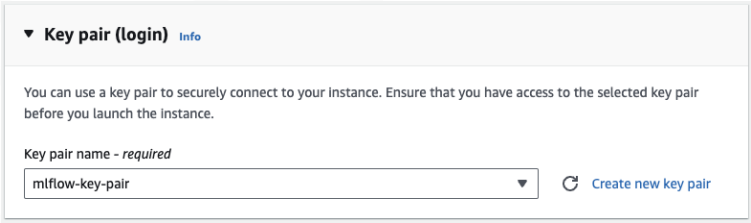

Figure 2.6: Select Key Pair

Leave the rest of the configuration the way it is an launch the instance.

- **Step 2**: Configure the security group
  Open up the new instance and select the newly created "Security groups" in "Security". Edit security group so the EC2 instance accepts SSH (port 22) and HTTP connections (port 5000). Do this by "Edit Inbound rules".

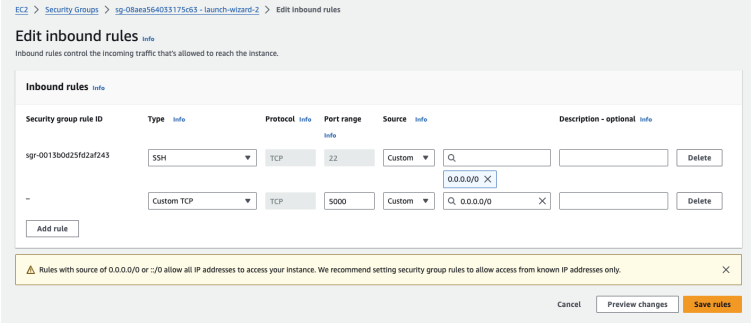

Figure 2.7: Security group setup

- **Step 3**: Create an S3 bucket
  The Amazon Simple Storage Service (S3) will be used as the artifact store. Go to "S3" and click on "Create bucket". Fill in the bucket name as in the image below and leave all the other configurations with their default values.

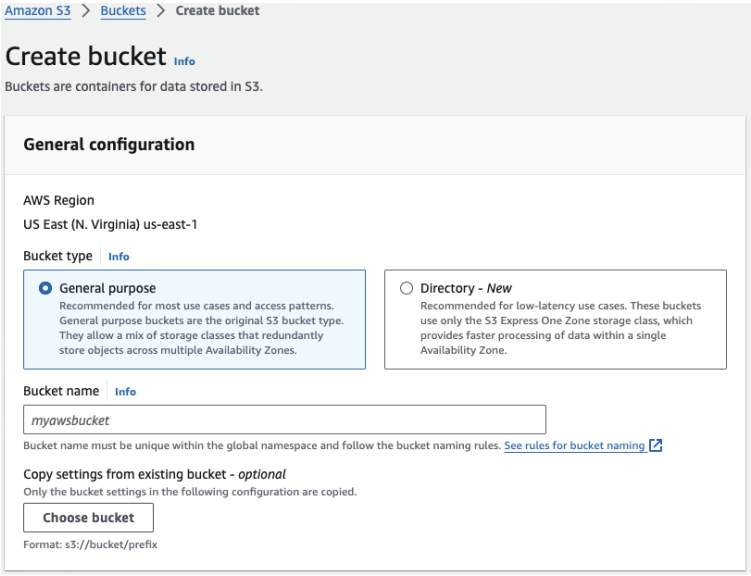

Figure 2.8: S3 bucket setup

- **Step 4**: Create a new PostgreSQL database in RDS
  The Amazon Relational Databse Service (RDS) uses PostgreSQL engine as the backend store. Go to the RDS Console and click on "Create database". Make sure to select "PostgreSQL" engine type and the "Free tier" template.

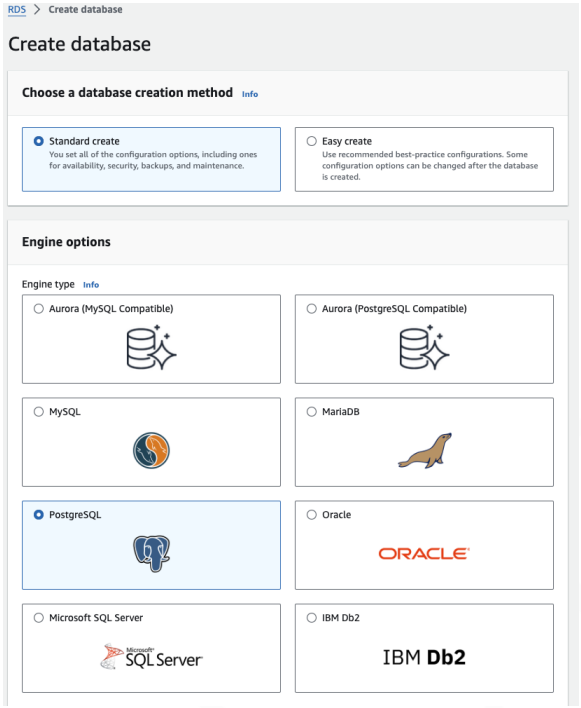

Figure 2.9: Create RDS Database

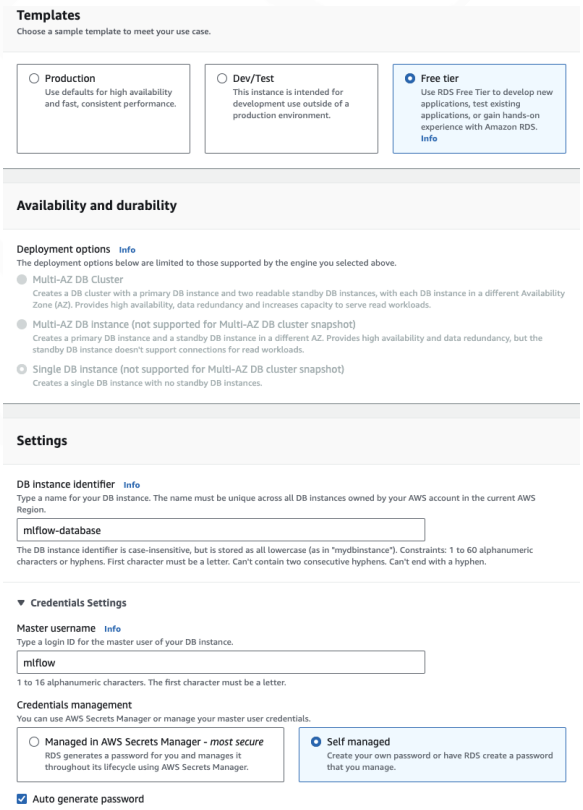

Figure 2.10: Setup RDS username and password

Select a name for your DB instance, set the master username as "mlflow" and tick the option "Auto generate a password" so Amazon RDS generate a password automatically. Setup instance connection, use the default VPC and connectivity to the compute resource. Use default values for all the other configurations.

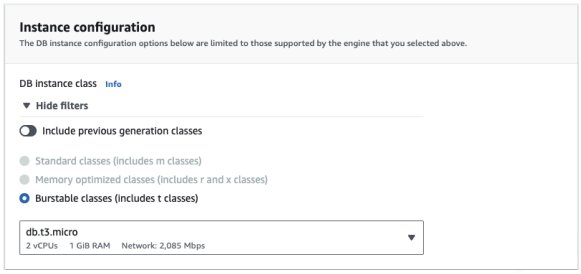

Figure 2.11: Instance Configuration

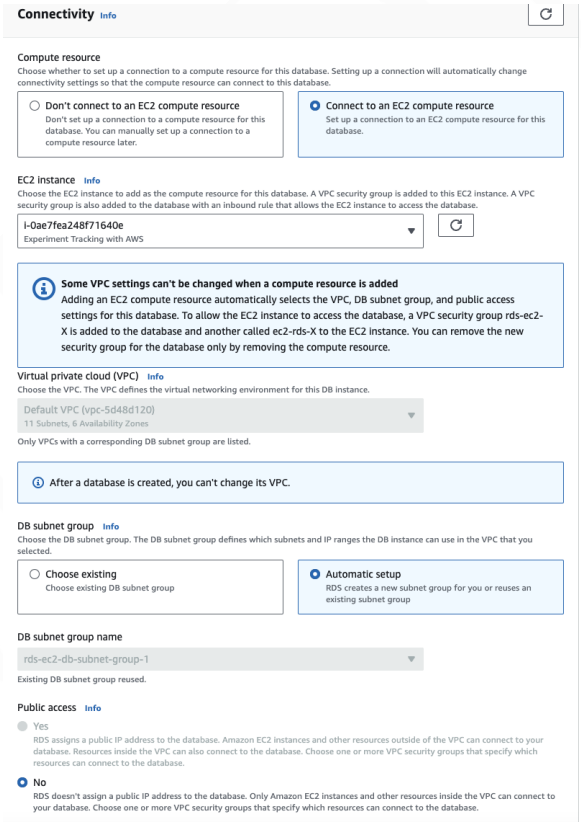

Figure 2.12: EC2 connectivity to DB

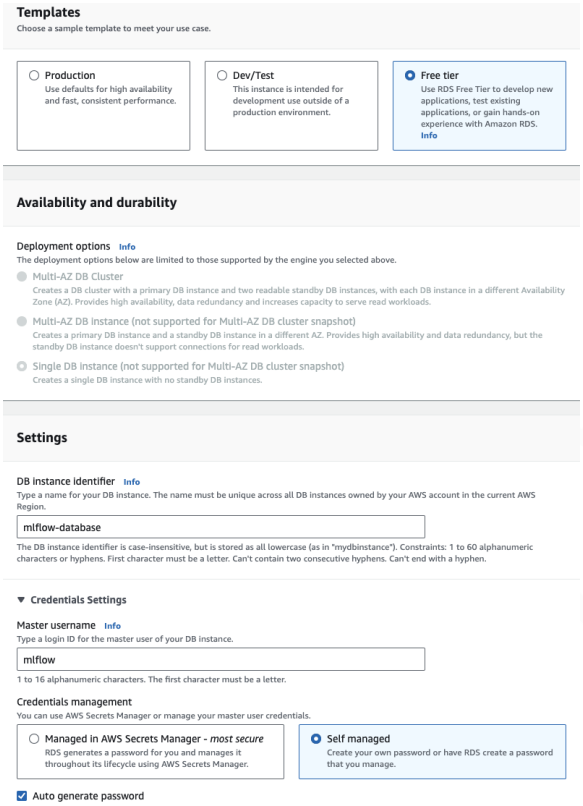

Figure 2.13: RDS Setup

Specify a database name so that RDS will create the database for you.

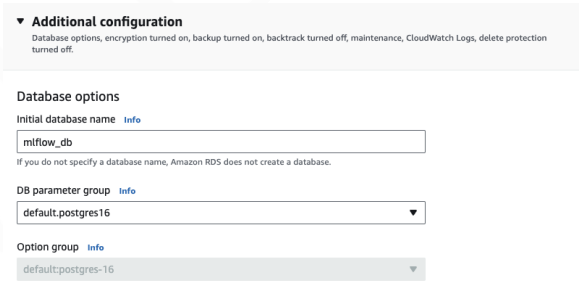

Figure 2.14: Additional configuration to set database name

After clicking on "launch database" you will be able to check the newly generated password, but take into account that the automatically generated password will be shown only once!. Take note of the master username, password, initial database name, and endpoint.

Once the DB instance is created, go to the RDS console, select the new db and under "Connectivity & security" select the VPC security group. Modify the security group by adding a new inbound rule that allows postgreSQL

connections on the port 5432 from the security group of the EC2 instance. This way, the server will be able to connect to the postgres database.

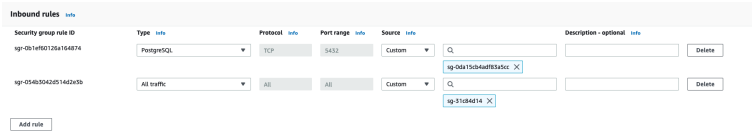

Figure 2.15: Edit Security Groups

- **Step 5**: Connect to the EC2 instance and launch the tracking server
  Go to the EC2 Console and find the instance launched and click on "Connect" and then follow the steps described in the tab "SSH".

  Run the following commands to install the dependencies, configure the environment and launch the server:

```
sudo yum update
pip3 install mlflow boto3 psycopg2-binary
aws configure # you'll need to input your AWS credentials here
mlflow server -h 0.0.0.0 -p 5000 --backend-store-uri postgresql://
    DB_USER:DB_PASSWORD@DB_ENDPOINT:5432/DB_NAME --default-artifact-
    root s3://S3_BUCKET_NAME
```

  Note: before launching the server, check that the instance can access the s3 bucket created. To do that, just run this command from the EC2 instance: `aws s3 ls`. You should see the bucket listed in the result.

- **Step 6**: Access the remote tracking server from your local machine
  Open a new tab on your web browser and go to this address: `http://<EC2_PUBLIC_DNS>:5000` (you can find the instance's public DNS by checking the details of your instance in the EC2 Console).

2.5 MLflow: Benefits, limitations and alternatives

2.5.1 Remote tracking server

The tracking server can be easily deployed to the cloud. Some of the benefits are:

- Shared experiments with other data scientist
- Collaborate with others to build and deploy models
- Give more visibility of the data science efforts

2.5.2 Issues with running a remote (shared) MLflow server

Some issues can arise when running a remote MLflow server. They include:

- Security

- Restrict access to the server (e.g. access through VPN)
- Scalability
  - Check Deploy MLflow on AWS Fargate
  - Check MLflow at Company Scale
- Isolation
  - Define standard for naming experiments, modles, and a set of default tags
  - Restrict access to artifacts (e.g. use s3 buckets living in different AWS accounts)

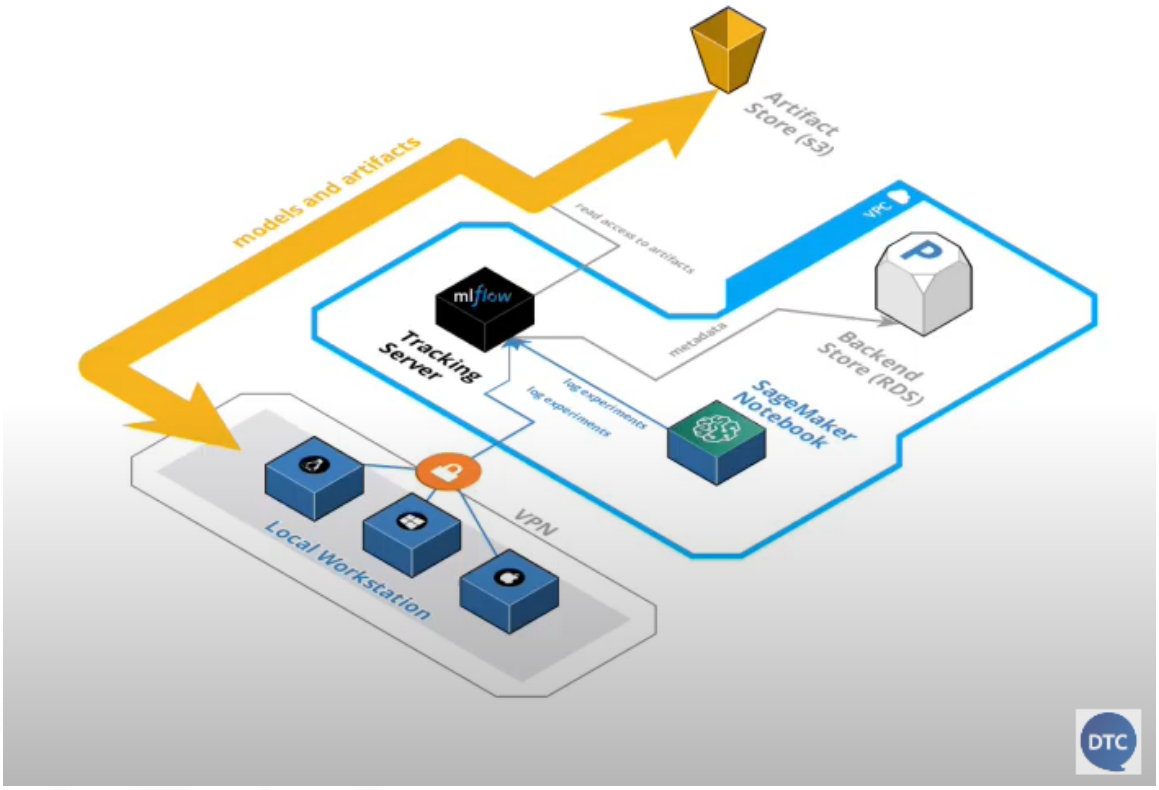

Figure 2.16: Remote Tracking Server with MLflow

2.5.3 MLflow limitations (and when not to use it)

There are some limitations for MLflow. For example:

- **Authetication & Users:** The open source version of MLflow doesn't provide any sort of authetication.
- **Data versioning:** To ensure full reproducibility, we need to version the data used to train the model. MLflow doesn't provide a built-in solution for that but there are a few ways to deal with this limitation.
- **Model/Data Monitoring & Alerting:** This is outside the scope of MLflow and currently there are more suitable tools for doing this.

There are some paid alternatives to MLflow including Neptune.ai, Comet, Weight & Biases and many more.

MLOps

Orchestration and ML Pipelines

3.1 Introduction: ML pipelines and Mage

3.1.1 Operationalizing ML models

Fun Fact: Operationalizing ML models

Operationalizing ML models involves moving them from development to production to drive business value

To operationalize ML models, we take the following steps:

- **`Preparing the model for deployment`** involves optimizing performance, ensuring it handles real-world data, and packaging it for integration into existing systems.
- **`Deploying the model`** involves moving it from development to production, making it accessible to users and applications.
- Once deployed, models must be **`continuously monitored for accuracy and reliability,`** and may need retraining on new data and updates to maintain effectiveness.
- **`Integrating operationalized model into existing workflows, applications, and decision-making processes`** to drive business impact.

Effective operationalization enables organizations to move beyond experimentation and drive tangible value from ML at scale, powering intelligent applications that personalize the customer experience and creates real business value.

3.1.2 Why we need to operationalize ML

- **Productivity** - MLOps fosters collaboration between data scientists, ML engineers, and DevOps teams by providing a unified environment for experiment tracking, feature engineering, model management, and deployment. This breaks down silos and accelerates the entire machine learning lifecycle.

- **Reliability** - MLOps ensures high-quality, reliable models in production through clean datasets, proper testing, validation, CI/CD practices, monitoring, and governance.
- **Reproducibility** - MLOps enables reproducibility and compliance by versioning datasets, code, and models, providing transparency and auditability to ensure adherence to policies and regulations.
- **Time-to-value** - MLOps streamlines the ML lifecycle, enabling organizations to successfully deploy more projects to production and derive tangible business value and ROI from AI/ML investments at scale.

3.1.3 How Mage helps MLOps

- **Data preparation** - Mage offers features to build, run, and manage data pipelines for data transformation and integration, including pipeline orchestration, notebook environments, data integrations, and streaming pipelines for real-time data.
- **Training and deployment** - Mage helps prepare data, train machine learning models, and deploy them with accessible API endpoints.
- **Standardize complex processes** - Mage simplifies MLOps by providing a unified platform for data pipelining, model development, deployment, versioning, CI/CD, and maintenance, allowing developers to focus on model creation while improving efficiency and collaboration.

3.1.4 Project setup for Mage

Clone the following respository containing the complete code for this module:

```
git clone https://github.com/mage-ai/mlops.git
```

Change directory into the cloned repo:

```
cd mlops
```

Launch Mage and the database service (PostgreSQL):

```
./scripts/start.sh
```

If you don't have bash in your enviroment, modify the following command and run it:

```
PROJECT_NAME=mlops \
    MAGE_CODE_PATH=/home/src \
    SMTP_EMAIL=$SMTP_EMAIL \
    SMTP_PASSWORD=$SMTP_PASSWORD \
    docker compose up
```

The subproject that contains all the pipelines and code is named `unit_3_observability`

MLOps

Deployment

4.1 Model Deployment

4.1.1 Three ways of deploying models

Fun Fact: Deploying ML models

Deploying machine learning model models is an iterative process that often involves multiple rounds of training, testing, and validating before the model is ready for production.

Deploying a model means that other application can get predictions from our model. There are three modes of deployment namely **online** deployment, **offline** or batch deployment, and **streaming**.

First, we need to ask ourselves if we need to have predictions immediately or it can wait a little bit for an hour, a day or a week. If it can wait for a little bit, then we go for batch deployment. i.e., the model is not running all the time and we just apply our model to new data regularly. In online deployment, the model is up and running all the time and is always available. Two variant of online deployment is deployment as a web service and deployment via streaming. In web service deployment, we send HTTP requests and the service sends out prediction. In streaming, there is an "events model service" listening for events on the stream and reacting to this event.

4.1.2 Web services: Introduction to Flask

Web Service

A `web service` is a method for communicating between two devices over a network using some protocols.

Assuming we want to use our model inside a `churn service` in order to make some predictions. The `marketing service` will communicate with our churn service by

sending some request and getting a response. This can be done using a web service. In **web service**, a user sends a request in the form of a query, then the web service sends back a response to the user with the result. In **Fig.** 4.1, we have a notebook that was used to train the churn model. The model is saved to file. We can load this model from a different process or web service called the `churn service`. If the `marketing service` wants to identify if the user will churn, they send a request to the `churn service` with information about the user then they get back the predictions and based on these predictions, the marketing service can decide whether they want to send a promotional email with say 25% discount to prevent churn.

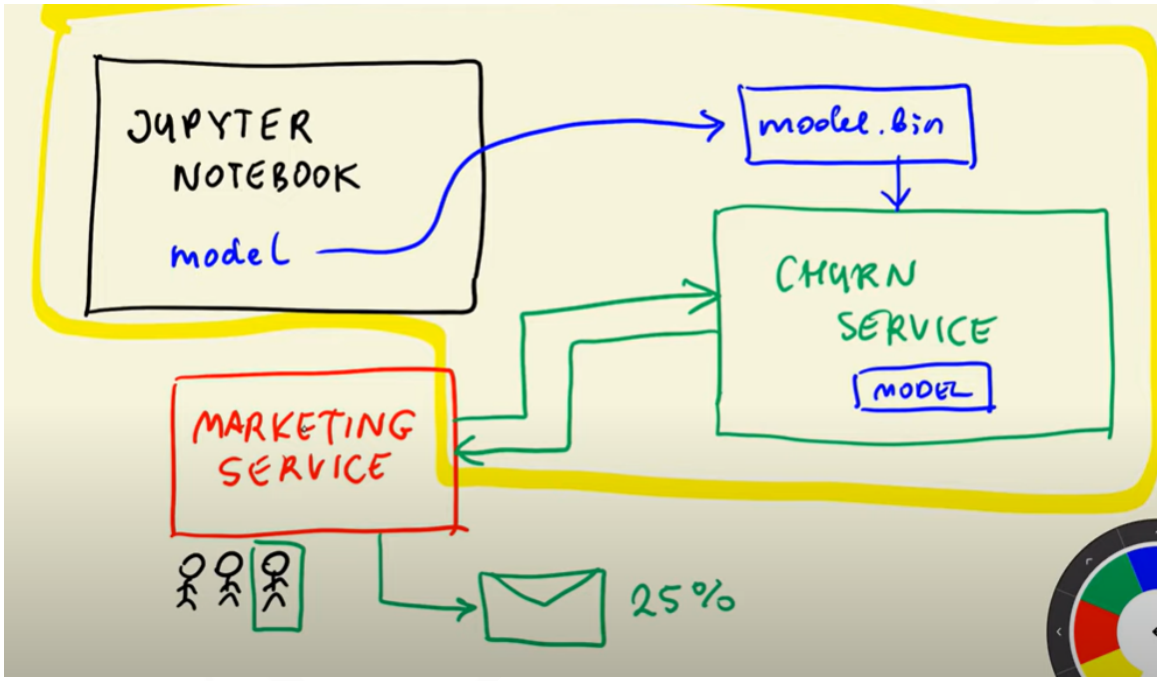

Figure 4.1: Serving Churn Model

To do this, we put the model inside a web service using **flask**, which is a framework for creating web services in python, then we isolate dependencies for this service so they don't interfere with other services on ur machine by creating a special environment for python dependencies using **Pipenv**. Then we add another layer with system dependencies using **Docker**, and then finally we deploy the container containing this model to the cloud using **AWS Elastic Beanstalk**.

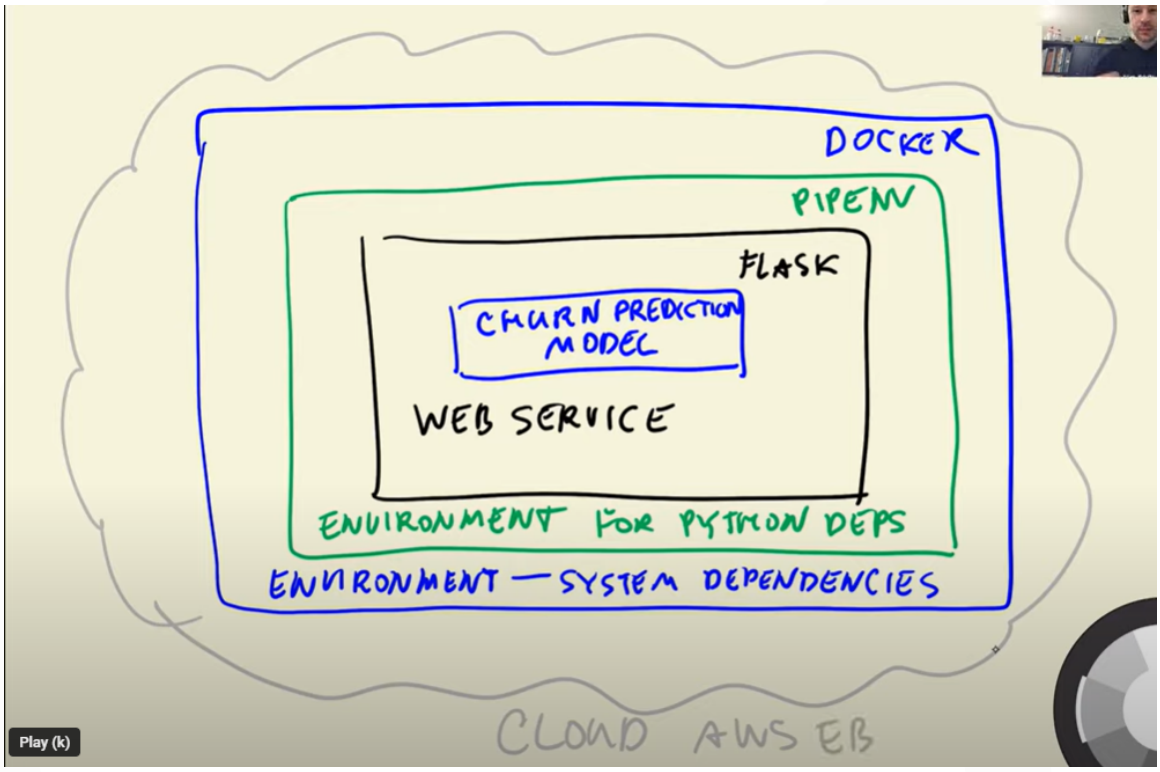

Figure 4.2: Layers to deployment

To deploy a model, we turn our notebook to Python script called `train.py` where we save the model as a pickle file. Then using our `predict.py`, we can load the model and make prediction.

There are some methods in web services we can use it to satisfy our problems. Here below we would list some.

- `GET`: GET is a method used to retrieve files, For example when we are searching for a cat image in google we are actually requesting cat images with GET method.
- `POST`: POST is the second common method used in web services. For example in a sign up process, when we are submiting our name, username, passwords, etc we are posting our data to a server that is using the web service. (Note that there is no specification where the data goes)
- `PUT`: PUT is same as POST but we are specifying where the data is going to.
- `DELETE`: DELETE is a method that is used to request to delete some data from the server.

We can create a simple service using flask that pings and send a response back. To do that, we create a `ping.py` file containing:

```
from flask import Flask
```

```
app = Flask('ping')

@app.route('/ping', methods=['GET'])
def ping():
    return "PONG"

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=9696)
```

We start by installing and importing flask

```
from flask import Flask
```

We create a flask app

```
app = Flask('ping')
```

and then add a decorator, which is a way to add some extra functionality to our functions. This extra functionality will allow us turn the function into a web service. We specify the address the 'ping' will live in, and the method to access this route.

```
@app.route('/ping', methods=['GET'])
```

We run the app in debug mode and specify the host to run on.

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=9696)
```

We get the following output when it runs:

```
$ python ping.py
 * Serving Flask app 'ping'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:9696
 * Running on http://172.20.10.2:9696
Press CTRL+C to quit
 * Restarting with watchdog (fsevents)
 * Debugger is active!
 * Debugger PIN: 976-978-559
```

Figure 4.3: Ping result

To test it, open your browser and search `localhost:9696/ping`, You'll see that the 'PONG' string is received. Congrats You've made a simple web server.

4.1.3 Serving the Churn Model with Flask

In serving the churn model with flask, we want the model to be available at `/predict` in the `churn service`. The `marketing service` will send the `churn service` with information about the customers, and then we reply them with the probability of churning. The `churn service` also sends a promotional email to the customer with 25% discount. To make the web service predict the

churn value for each customer, we need to first load the previous saved model and use a prediction function in a special route.

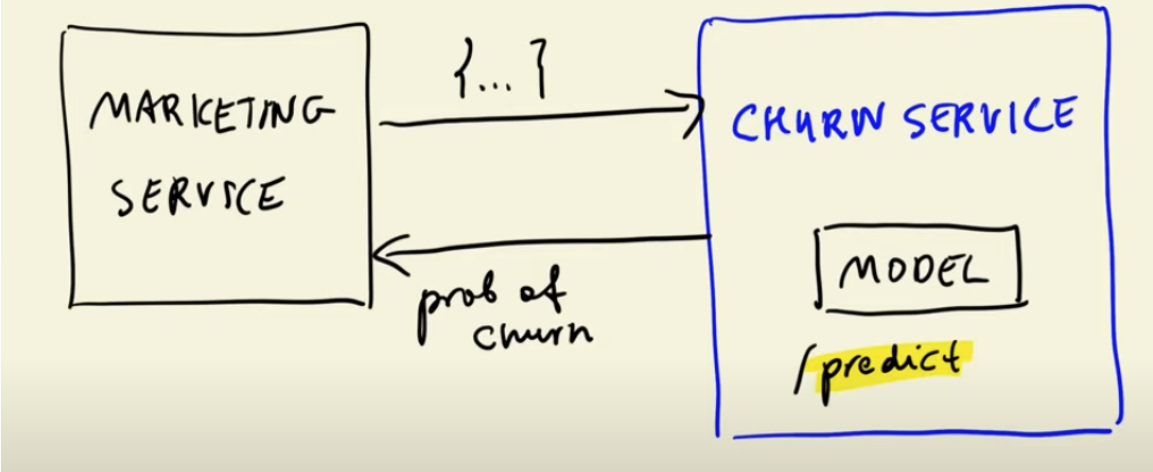

Figure 4.4: Ping result

- To load the previous saved model, we create a `predict.py` file that loads the model

```
# Load the model
import pickle
from flask import Flask
from flask import request
from flask import jsonify

model_file = 'model_C=1.0.bin'

# load the model
with open(model_file, 'rb') as f_in: # read binary file
    dv, model = pickle.load(f_in)     # load() loads the file
```

- A function to predict a value for a customer

```
def predict_single(customer, dv, model):
    X = dv.transform([customer])  # one-hot encoding to the data
    y_pred = model.predict_proba(X)[:, 1]
    return y_pred[0]
```

- Then the function used to create the web service that makes prediction and sends an email to the customer.

```
app = Flask('churn') # name of the app
@app.route('/predict', methods=['POST'])  # in order to send the
   customer information we need to post its data.
def predict():
    customer = request.get_json()  # web services work best with
   json frame
    prediction = predict_single(customer, dv, model)
```

```
    churn = prediction >= 0.5

    result = {
        'churn_probability': float(prediction),  # cast numpy float
    type to python native float type
        'churn': bool(churn),  # casting the value using bool
    method
    }
    return jsonify(result)  # send back the data in json format to
    the user
```

To deploy, the customer information in the `deploy.py` file is sent as JSON. The predict script is turned into a web service and sends a response back to the `marketing service` with predictions as JSON and an email to the customer likely to churn.

```
customer_id = "asdx-123d"
customer_email = "asdx-123d@yahoo.com"
customer = {
    "gender": "female",
    "seniorcitizen": 0,
    "partner": "yes",
    "dependents": "no",
    "phoneservice": "no",
    "multiplelines": "no_phone_service",
    "internetservice": "dsl",
    "onlinesecurity": "no",
    "onlinebackup": "yes",
    "deviceprotection": "no",
    "techsupport": "no",
    "streamingtv": "no",
    "streamingmovies": "no",
    "contract": "two_year",
    "paperlessbilling": "yes",
    "paymentmethod": "electronic_check",
    "tenure": 10,
    "monthlycharges": 29.85,
    "totalcharges": (2 * 29.85)
}
# Making requests
import requests
url = "http://localhost:9696/predict"
response = requests.post(url, json=customer).json()
print(response)

if response["churn"]:
    print(f"Sending email to {customer_id} with email:", {customer_email
    })
else:
    print(f"Customer {customer_id} will not churn")
```

You can run the prediction app using:

```
python predict.py
```

When you run the app, you will see a warning that it is not a WGSI server and not suitable for production environments. To fix this issue and run this as a production server there are plenty of ways available. One way to create a WSGI server is to use gunicorn. To install it use the command

```
pip install gunicorn
```

And to run the WGSI server you can simply run it with the command

```
gunicorn --bind 0.0.0.0:9696 predict:app
```

Note that in `predict:app` the name "predict" is the name we set for the file containing the code `app = Flask('churn')`(for example: predict.py), You may need to change it to whatever you named your Flask app file. So far, we have been able to make a production server that predict the churn value for new customers.

4.1.4 Dependencies and Environment Management: Pipenv

Every time we're running a file from a directory we're using the executive files from a global directory. For example when we install python on our machine the executable files that are able to run our codes will go to somewhere like **/home/username/python/bin/** for example the pip command may go to **/home/username/python/bin/pip**. Sometimes the versions of libraries conflict (the project may not run or get into massive errors). For example we have an old project that uses sklearn library with the version of 0.24.1 and now we want to run it using sklearn version 1.0.0. We may get into errors because of the version conflict. To solve the conflict we can make virtual environments. Virtual environment is something that can seperate the libraries installed in our system and the libraries with specified version we want our project to run with. There are a lot of ways to create a virtual environments.

One way we are going to use is using a library named `pipenv`. "pipenv" is a library that can create a virutal environment. To install this library just use the classic method `pip install pipenv`. After installing `pipenv`, we must install the libraries we want for our project in the new virtual environment. It's really easy, Just use the command `pipenv` instead of `pip`. i.e., **`pipenv install numpy scikit-learn==0.24.2 flask gunicorn`**. With this command we installed the libraries we want for our project. Note that using the pipenv command we made two files named "Pipfile" and "Pipfile.lock". If we look at this files closely we can see that in Pipfile the libraries we installed are named. If we specified the library name, it's also specified in Pipfile.

The Pipfile looks like:

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
numpy = "*"
scikit-learn = "*"
flask = "*"
gunicorn = "*"

[dev-packages]

[requires]
python_version = "3.9"
```

In "Pipfile.lock" we can see that each library with it's installed version is named and a hash file is there to reproduce if we move the environment to another machine. If we want to run the project in another machine, we can easily installed the libraries we want with the command **`pipenv install.`** This command will look into Pipfile and Pipfile.lock to install the libraries with specified version. After installing the required libraries we can run the project in the virtual environment with "pipenv shell" command.

```
pipenv shell
```

This will go to the virtual environment's shell and then any command we execute will use the virtual environment's libraries. Installing and using the libraries such as gunicorn is the same as the last session. To run the `predict` app, we can call gunicorn in the "pipenv shell" using:

```
gunicorn --bind 0.0.0.0:9696 predict:app
```

If you can't access the port, first kill any running process by checking the running processes with:

```
sudo lsof -t -i:9696
```

Then kill the port using:

```
kill <PID number>
```

After killing the port, you can try to run the "predict" app with gunicorn again. Until here we made a virtual environment for our libraries with a required specified version. To seperate this environment more, such as making gunicorn be able to run in windows machines we need another way. The other way is using **`Docker`**. Docker allows us to seperate everything more than before and make any project able to run on any machine that support Docker smoothly.

4.1.5 Dependencies and Environment Management: Docker

Let's say we have our host machine which is a laptop with Ubuntu and then on this laptop, we have different virtual environment. For instance, we may have a virtual environment for `churn service` and another one for `lead scoring service` with different python versions and dependencies. With Docker, we can get even more isolation. So instead of creating virtual environment for each services, we can put each service in a separate container and these services will not know anything about each other.

To perform more isolation .i.e, separate our project file from our system machine, there is an option named Docker. With Docker you are able to pack all your project in a system that you want and run it in any system machine. For example if you want Ubuntu 20.4 you can have it in a mac or windows machine or other operating systems.

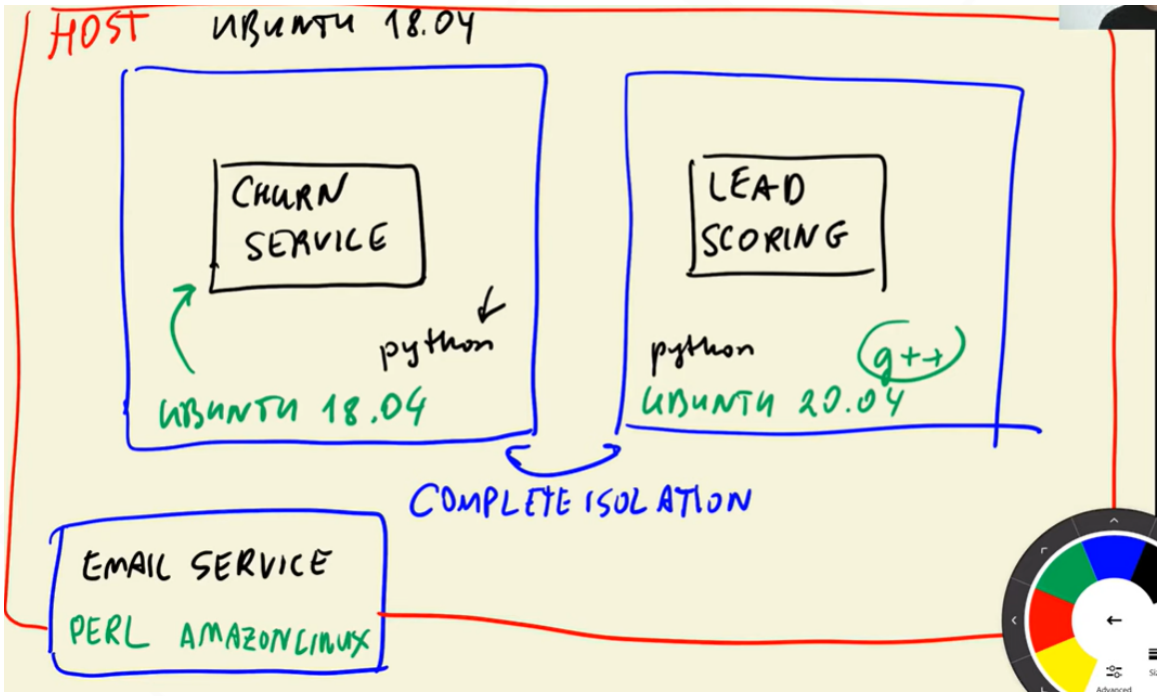

Figure 4.5: Docker Isolation

To get started with Docker for the churn prediction project you can follow the instructions below.

- On Ubuntu

  ```
  sudo apt-get install docker.io
  ```

  To run docker without sudo, follow this instruction.

- On MacOS

- Follow the steps in the Docker docs.

To run docker using the default command in python:

```
docker run -it --rm python:3.8.12-slim
```

We can change the entrypoint to be bash using:

```
docker run -it --rm --entrypoint=bash python:3.8.12-slim
```

Once our project was packed in a Docker container, we're able to run our project on any machine. First we have to make a Docker image. In Docker image file, there are settings and dependencies we have in our project. To find Docker images that you need you can simply search the Docker website.

We can create a Dockerfile where we define everything that we want to do in a docker image. Here is a Dockerfile (There should be no comments in Dockerfile, so remove the comments when you copy)

```
# First install the python 3.8, the slim version uses less space
FROM python:3.8.12-slim

# Install pipenv library in Docker
RUN pip install pipenv

# create a directory in Docker named app and we're using it as work
   directory
WORKDIR /app

# Copy the Pip files into our working derectory
COPY ["Pipfile", "Pipfile.lock", "./"]

# install the pipenv dependencies for the project and deploy them.
RUN pipenv install --deploy --system

# Copy any python files and the model we had to the working directory of
    Docker
COPY ["*.py", "model_C=1.0.bin", "./"]

# We need to expose the 9696 port because we're not able to communicate
   with Docker outside it
EXPOSE 9696

# If we run the Docker image, we want our churn app to be running
ENTRYPOINT ["gunicorn", "--bind", "0.0.0.0:9696", "predict:app"]
```

The flags `-deploy` and `-system` makes sure that we install the dependencies directly inside the Docker container without creating an additional virtual environment (which pipenv does by default).

If we don't put the last line ENTRYPOINT, we will be in a python shell. Note that for the entrypoint, we put our commands in double quotes.

After creating the Dockerfile, we need to build it:

```
docker build -t churn-prediction .
```

To run it, execute the command below:

```
docker run -it -p 9696:9696 churn-prediction:latest
```

Flag explanations:

- `-t`: is used for specifying the tag name "churn-prediction".
- `-it`: in order for Docker to allow us access to the terminal.
- `-rm`: allows us to remove the image from the system after we're done.
- `-p`: to map the 9696 port of the Docker to 9696 port of our machine. (first 9696 is the port number of our machine and the last one is Docker container port.)
- `-entrypoint=bash`: After running Docker, we will now be able to communicate with the container using bash (as you would normally do with the Terminal). Default is python.

At last you've deployed your prediction app inside a Docker continer.

4.2 Online Deployment

4.2.1 Web services: Deploying models with Flask and Docker

Creating a Virtual Environment with Pipenv

To deploy the model, we have to create a virtual environment using `Pipenv`. In your project folder, create a web-service folder. This "web-service" will house the files required to deploy the application. We should make sure that the version we used to create the pickle file is the exact same version we install in the virtual environment.

```
pip freeze | grep scikit-learn
```

Using the same exact version of scikit-learn, we install that in the virtual environment with "pipenv". In the web-service folder, run the command to install the necessary packages using "pipenv".

```
pipenv install scikit-learn==1.0.2 flask --python=3.9
```

This creates a `Pipfile` and `Pipfile.lock` containing the packages we just installed and their exact version. Once the virtual environment have been created, launch the subshell in virtual environment. To enter the virtual environment, run the command below:

```
pipenv shell
```

The prompt is usually long. We can reduce it using:

```
PS1="> "
```

Creating a script for predicting

To predict, we create a script for prediction called `predict.py`:

```
import pickle

with open('lin_reg.bin', 'rb') as f_in:
    (dv, model) = pickle.load(f_in)

def prepare_features(ride):
    features = {}
    features['PU_DO'] = '%s_%s' % (ride['PULocationID'], ride['
   DOLocationID'])
    features['trip_distance'] = ride['trip_distance']
    return features

def predict(features):
    X = dv.transform(features)
    preds = model.predict(X)
    return preds[0]
```

We can test our prediction with a `test.py` file

```
import predict

ride = {
    "PULocationID": 10,
    "DOLocationID": 50,
    "trip_distance": 40
}

features = predict.prepare_features(ride)
pred = predict.predict(ride)
print(pred)
```

To run prediction, we simply run **`python test.py`** in terminal.

Putting the script into a Flask app

To turn the scripts above to a flask application, we create a function that will be a wrapper around the `predict.py` and `test.py` file. This function will take web requests, HTTP request and the score it gets in the request and return the prediction. Let's call the function `predict_endpoint()`. Update the `predict.py` file with the function `predict_endpoint()`.

```
import pickle
from flask import Flask, request, jsonify

with open('lin_reg.bin', 'rb') as f_in:
    (dv, model) = pickle.load(f_in)

def prepare_features(ride):
```

```
    features = {}
    features['PU_DO'] = '%s_%s' % (ride['PULocationID'], ride['
   DOLocationID'])
    features['trip_distance'] = ride['trip_distance']
    return features

def predict(features):
    X = dv.transform(features)
    preds = model.predict(X)
    return float(preds[0])

app = Flask('duration-prediction')

@app.route('/predict', methods=['POST']) # turn predict function into an
    endpoint
def predict_endpoint():
    ride = request.get_json()

    features = prepare_features(ride)
    pred = predict(features)

    result = {
        'duration': pred
    }
    return jsonify(result)


if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=9696)
```

In the above code, we added an import for Flask, requests and jsonify. Requests gives us a way get the data, the result of our prediction will be a dictionary with duration and jsonify returns this as a `json` object. We also added a decorator which adds some extra functionality to the predict function by turning the function into HTTP endpoint and we can start sending requests to this endpoint and get responses. This library has a method called "POST" that we use for sending post requests and here we need to specify the url end point and port to which we want to send the request.

We update the `test.py` file with the **`url`** variable and **`response`** variable.

```
import requests
import predict

ride = {
    "PULocationID": 10,
    "DOLocationID": 50,
    "trip_distance": 40
}

url = 'http://localhost:9696/predict'
response = requests.post(url, json=ride)
print(response.json())
```

The `requests` library is used for making HTTP requests in Python. The **`url`** variable contains the address of the local server endpoint **`/predict`** running on port 9696. The **`requests.post`** method is used to send a POST request to the **`url`**. The **`json`** parameter is used to send the **`ride`** dictionary as a JSON payload in the request body. The response from the server is stored in the **`response`** variable. The **`response.json()`** method is called to parse the JSON response from the server into a Python dictionary. The parsed response is then printed to the console.

The code in the `test.py` file sends a POST request to a local server to get a prediction based on the provided ride details. In the prompt in the virtual environment, we run **`python predict.py`** and open another terminal to run **`python test.py`**. Now we have sucessfully put our model in a flask and deployed that model in a development environment. Alternatively, we can install gunicorn if we want to put this into production. The application we just ran is running in a development environment in Flask. To run it in a production environment, we use gunicorn. We first install gunicorn

```
pipenv install gunicorn
```

Then start the service using:

```
gunicorn --bind 0.0.0.0:9696 predict:app
```

Then test the the service and make prediction with **`python test.py`**.

Packaging the app to Docker

To package the app to `Docker`, first we need to create a "Dockerfile". In the Dockerfile, we add the following command:

```
FROM python:3.9.18-slim

RUN pip install -U pip
RUN pip install pipenv

WORKDIR /app

COPY ["Pipfile", "Pipfile.lock", "./"]

RUN pipenv install --deploy --system

COPY ["predict.py", "lin_reg.bin", "./"]

EXPOSE 9696

ENTRYPOINT ["gunicorn", "--bind=0.0.0.0:9696", "predict:app"]
```

Next we build the docker image using the Dockerfile

```
docker build -t ride-duration-prediction-service:v1 .
```

And then we can test it in interactive mode ("–it") and remove it after running ("-p") using:

```
docker run -it --rm -p 9696:9696 ride-duration-prediction-service:v1
```

In a separate shell, test the the service and make prediction with **`python test.py`**. With the docker container, we can serve our model and deploy the service anywhere docker is supported .e.g., using AWS Elastic Beanstalk or Kubernetes.

4.2.2 Web services: Getting the models from the model registry (MLflow)

— Not up to date

1. **Overview:**
   - Discussed combining model deployment using Flask with model registry.
   - Deployed a linear regression model as a web service.
   - Exposed predictive function as an endpoint for making predictions.
2. **Model Registry Integration:**
   - Utilized model registry to fetch models (e.g., random forest) using a run ID.
   - Demonstrated setting up MLflow server with SQLite and S3 as the artifact root.
3. **Implementation Steps:**
   - Prepared functions for data conversion and prediction.
   - Created Flask application to deploy model fetched from MLflow registry.
   - Demonstrated setting up virtual environment and installing dependencies.
4. **Simplifying Model Handling:**
   - Emphasized using scikit-learn pipelines to combine preprocessing and model into a single entity.
   - Demonstrated logging entire pipeline to MLflow, reducing complexity in fetching artifacts.
5. **Testing and Validation:**
   - Validated the new pipeline-based model by running predictions and comparing outputs.
   - Highlighted the importance of specifying model version in responses.
6. **Independence from Tracking Server:**

- Addressed potential issues with dependency on tracking servers.
- Demonstrated fetching models directly from S3 to avoid dependency on MLflow tracking server.
- Suggested using environment variables for dynamic configuration in deployments (e.g., Kubernetes).

7. **Code and Configuration:**
   - Provided code snippets for setting up and testing Flask application.
   - Showed how to replace hardcoded paths with environment variables for flexible deployment.

Example Code Snippets:

Creating a Pipeline:

```
from sklearn.pipeline import make_pipeline

pipeline = make_pipeline(vectorizer, RandomForestRegressor())
pipeline.fit(X_train, y_train)
```

Logging Pipeline to MLflow:

```
mlflow.sklearn.log_model(pipeline, "model")
```

Fetching Model from S3:

```
model_uri = f"s3://my-bucket/mlflow/{run_id}/artifacts/model"
model = mlflow.sklearn.load_model(model_uri)
```

Using Environment Variables:

```
import os

run_id = os.getenv('RUN_ID')
model_uri = f"s3://my-bucket/mlflow/{run_id}/artifacts/model"
model = mlflow.sklearn.load_model(model_uri)
```

4.2.3 Batch: Preparing a scoring script

Assuming we have built a model to predict the ride duration and we want to see how our drivers deviate from the actual duration, we can deploy the model for predicting ride duration in batch mode. In deploying our model using Batch deployment, we have to train the model in a notebook, apply the model and then turn the notebook into a script.

To convert our `score.ipynb` notebook to a script, use:

```
jupyter nbconvert --to script score.ipynb
```

After turning the notebook into a script, we have to clean the script, parametrize the script and make it self-contained in such a way that we can just run the script and get the predictions. After this step, the next step is to deploy the model or schedule the script with Prefect.

Once the script is ready, we can create an environment with pipenv and specify all the dependencies that the script has. We also want to save the result to S3 in this case, we would need credentials to do that. after that, you package the script in a docker container and schedule your docker container with AWS Batch or ECS or as a kubernetes job.

MEAP

Model Monitoring

> *Model monitoring is the ongoing process of tracking, analyzing, and evaluating the performance and behavior of machine learning models in real-world, production environments. It involves measuring various data and model metrics to help detect issues and anomalies and ensure that models remain accurate, reliable, and effective over time.*
>
> — EvidentlyAI Blog

5.1 Introduction to ML monitoring

Fun Fact: Challenges in ML Model Deployment

Implementing a machine learning model is challenging, but building production services with that model is even harder. Over time, a model's performance may degrade, necessitating monitoring to maintain quality.

Once we've built a machine learning model, after sometime, the model's performance starts to degrade. ML model monitoring involves tracking a predefined set of metrics to detech issues and answer questions like "What happened?" and "Is the system working as expected?" This module discusses how to monitor machine learning models in production, focusing on maintaining model performance over time. Machine learning monitoring is done to monitor:

- Service Health
  - Latency
  - Memory
  - Uptime
- Data Health

  - Data Drift
  - Broken Pipelines
  - Data Outage
  - Schema Change
  - Underperforming Segments
- Model Health
  - Model Accuracy
  - Concept Drift
  - Model Bias

Service Health Monitoring - Service health monitoring is essential to ensure the service operates correctly. Key metrics include uptime, memory usage, and latency.

Model Performance Metrics - Additional Layers for ML Models: Include metrics related to data and model performance. Model performance metrics monitors model performance and is dependent on the problem statement. For ranking algorithms, use ranking metrics; for regression problems, use metrics like mean absolute error (MAE) or mean absolute percentage error (MAPE). For classification problems, use metrics like log loss, precision, and recall.

Data Quality and Integrity Metrics - In cases where ground truth is delayed, use proxy metrics related to data quality and integrity. Monitor metrics like the share of missing values, value ranges, and data distribution changes.

Data Drift and Concept Drift - Monitor data drift and concept drift to detect changes in data distributions over time. We compare input data distributions and model outputs over time to detect drifts.These metrics serve as early signals of potential problems with the model.

Extended Metrics for Monitoring

- Depending on the sensitivity of your case and risks, consider additional metrics such as:
  - Quality metrics by segments (e.g., category-specific performance).
  - Model bias and fairness, especially in sensitive domains like healthcare or finance.
  - Outlier monitoring to reduce risks by sending specific objects to manual review.
  - User trust metrics for recommender systems, especially for explaining recommendations.

5.1.1 Batch and Online Model Monitoring

Importance of Monitoring

Monitoring `production services` is crucial to ensure they function correctly. For machine learning models, additional layers of monitoring related to data and model performance are required. Important aspects to monitor include model performance, data drift, and concept drift.

Batch Monitoring

- Involves running evaluation jobs periodically or based on triggers.
- Daily monitoring jobs query model prediction logs to compute model and data quality metrics.
- Suitable for batch data pipelines and online ML services.
- Easier to maintain than continuous monitoring services.
- Combines immediate and delayed monitoring within the same architecture.
- Can run data validation jobs alongside monitoring.
- Introduces delays in metric computation.
- Requires expertise in workflow orchestration and data engineering resources.

Real-Time Monitoring

- Data is sent directly from the ML service to the monitoring service.
- Continuously computes and publishes ML quality metrics.
- Suitable for online ML prediction services.
- Allows near real-time detection of issues like missing data.
- More costly in terms of engineering resources.
- Batch monitoring pipelines may still be needed for delayed ground truth.

Use existing monitoring tools and dashboards (like Grafana) to monitor ML models. For batch models, metrics like data drift and precision can be calculated on batches of data. For online models, use window functions to generate small batches for calculating metrics.

5.1.2 Monitoring Architecture

- We can build a general monitoring scheme applicable to both batch and non-batch models.
- Start with prediction logs from the service (batch or online).
- Create a monitoring pipeline that reads prediction logs, calculates metrics, and stores them in a database.
- Use a dashboard tool like Grafana to visualize and monitor these metrics.

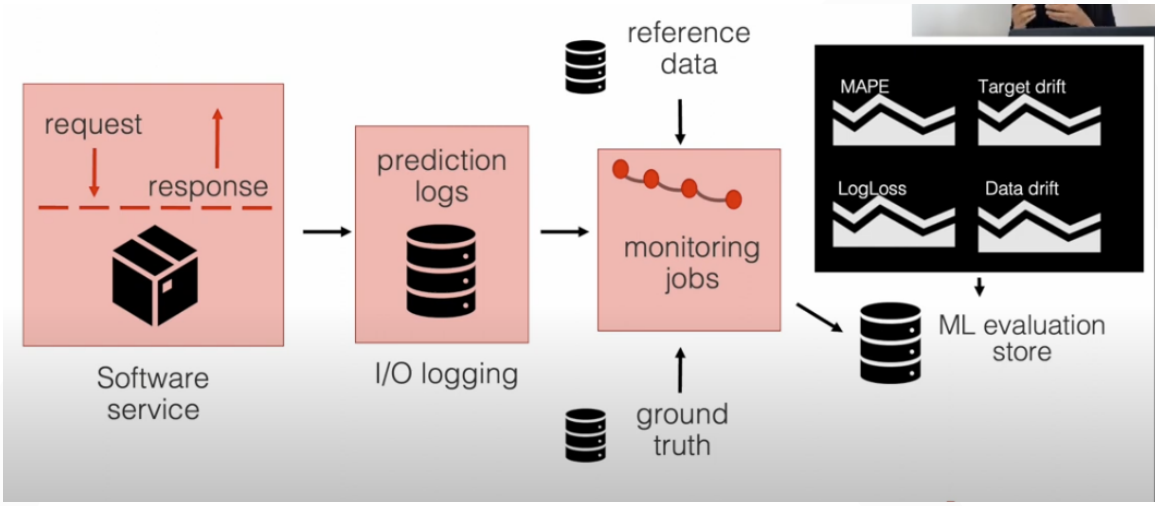

Figure 5.1: Monitoring Scheme

5.1.3 Practical Implementation

For batch models, use BI tools like Tableau or Looker for initial analysis. For online models, simulate production usage and build monitoring on top of logs using tools like Prefect and Evidently. Store metrics in a PostgreSQL database and visualize them using Grafana.

- Logging and Monitoring Pipeline:
  - Prediction Logs: Collect logs from the service.
  - Batch Analysis: Analyze logs in batches to calculate metrics.
  - Database Storage: Store metrics in a database.
  - Dashboarding: Use tools like Grafana to visualize and monitor metrics.
- Tools Used:
  - Prefect: For scheduling and running monitoring jobs.
  - Evidently Library: For calculating and comparing metrics.
  - PostgreSQL: For storing calculated metrics.
  - Grafana: For creating dashboards and alerts.

5.2 ML Monitoring Environment Setup

To setup environment for monitoring for our project, in the project folder,

```
mkdir taxi-monitoring
```

Change directory into the “taxi-monitoring” folder and create a virtual environment.

```
conda create -n monitoring-env python=3.11
```

Next we activate the virtual environment

```
conda activate monitoring-env
```

Create a `requirements.txt` file to list all the packages and install it. Once the requirements have been listed in the requirement file, install it in the terminal with:

```
pip install -r requirements.txt
```

Next we create a docker compose file. Docker compose is a tool that allows us to build multi-container docker application. Docker compose "yaml" file is a file where you can list all your services. The docker compose file is created below.

```
version: '3.7'

volumes:
  grafana_data: {}

networks:
  front-tier:
  back-tier:
  .....
```

In the "taxi-monitoring" directory, create a "config" folder and a "grafana_datasources.yaml" file. Docker compose is going to allow us to create all the services. Run the docker compose build command using:

```
docker-compose up --build
```

If you get the error below when building the docker file, find and kill the process using the port.

```
Error response from daemon: Ports are not available: exposing port TCP
    0.0.0.0:5432 -> 0.0.0.0:0: listen tcp 0.0.0.0:5432: bind: address
    already in use
```

Run the following one after the other

```
sudo lsof -i :5432
sudo kill -9 <PID>
```

Once docker compose is up and running, we can try to access grafana using `localhost:3000/login`, default username and password is `admin`. We can access adminer at `http://localhost:8080/`. We have successfully created our working environment and made sure our services starts. Once the service starts, we can prepare reference data and model.

5.3 Evidently Metric Calculation

To perform metric calculation in Evidently, we need to create a report. A report is an object which allows us to select what metrics we want to calculate and group them all together. The code snippet provided defines a `Report` object with specific metrics to evaluate in a dataset. Below is a detailed breakdown of what each part does:

Code Snippet

```
report = Report(metrics=[
    ColumnDriftMetric(column_name='prediction'),
    DatasetDriftMetric(),
    DatasetMissingValuesMetric()
])
```

Explanation

- `report = Report(metrics=[...])`: This initializes a `Report` object with a list of metrics that will be calculated when the report is generated.
- `ColumnDriftMetric(column_name='prediction')`: This metric checks for drift in the specified column, which is `'prediction'` in this case. Drift indicates that the statistical properties of a column have changed over time, which can signal issues such as changes in the underlying data distribution or model degradation.
- `DatasetDriftMetric()`: This metric evaluates the entire dataset to detect drift. It assesses if the overall data distribution has changed, which can affect the performance and reliability of models trained on the dataset.
- `DatasetMissingValuesMetric()`: This metric checks for missing values in the dataset. It identifies the presence and possibly the extent of missing data, which is crucial for maintaining data quality and ensuring the accuracy of analysis and model training.

Example Usage

This setup is typically used in monitoring data quality and model performance over time. Here is how you might use it:

1. Initialize the Report

```
report = Report(metrics=[
    ColumnDriftMetric(column_name='prediction'),
    DatasetDriftMetric(),
    DatasetMissingValuesMetric()
])
```

2. Generate the Report with Data

Assuming you have current and reference datasets (for drift comparison):

```
current_data = ...  # Load or define your current dataset
reference_data = ...  # Load or define your reference dataset

report.run(current_data=current_data, reference_data=reference_data)
```

3. View the Report

After running the report, you can inspect the results:

```
print(report.show())
```

Practical Application

In a real-world scenario, you would use this type of report to continuously monitor your data pipeline. For instance:

- **Column Drift Monitoring**: Ensuring that your model predictions remain consistent over time and are not being affected by unforeseen changes in data distribution.
- **Dataset Drift Monitoring**: Detecting shifts in your overall data, which could indicate changes in user behavior, data collection issues, or other external factors.
- **Missing Values Monitoring**: Keeping track of missing data to maintain data integrity and avoid biases in your analysis or model training.

We can create the report object as a python dictionary. This allows us to derive any value for the dictionary.We could derive information such as the prediction drift, number of drifted columns, and about the shares of missing value. By regularly generating and reviewing such reports, you can proactively address data quality issues and maintain the performance and reliability of your data-driven applications.

5.4 Evidently Monitoring Dashboard

An example of the Evidently monitoring dashboard can be found here. Once the dashboard have been created in jupyter notebook, we can test iot in the terminal by calling Evidently UI. In the "taxi-monitoring" directory, run `evidently ui` which starts the monitoring service. You can copy the URL, and paste in the browser to see the project list. We notice that we don't have any visible dashboard. In this case, we have to configure it using the code snippet below:

```
# configure the dashboard
```

```
project.dashboard.add_panel(
    DashboardPanelCounter(
        filter=ReportFilter(metadata_values={}, tag_values=[]),
        agg=CounterAgg.NONE,
        title="NYC taxi data dashboard"
    )
)

project.dashboard.add_panel(
    DashboardPanelPlot(
        filter=ReportFilter(metadata_values={}, tag_values=[]),
        title="Inference Count",
        values=[
            PanelValue(
                metric_id="DatasetSummaryMetric",
                field_path="current.number_of_rows",
                legend="count"
            ),
        ],
        plot_type=PlotType.BAR,
        size=WidgetSize.HALF,
    ),
)

project.dashboard.add_panel(
    DashboardPanelPlot(
        filter=ReportFilter(metadata_values={}, tag_values=[]),
        title="Number of Missing Values",
        values=[
            PanelValue(
                metric_id="DatasetSummaryMetric",
                field_path="current.number_of_missing_values",
                legend="count"
            ),
        ],
        plot_type=PlotType.LINE,
        size=WidgetSize.HALF,
    ),
)

project.save()
```

Once we've completed the configuration, we save and run it. We can go back to the monitoring service URL to see the dashboard. Our dashboard looks like the figure below.

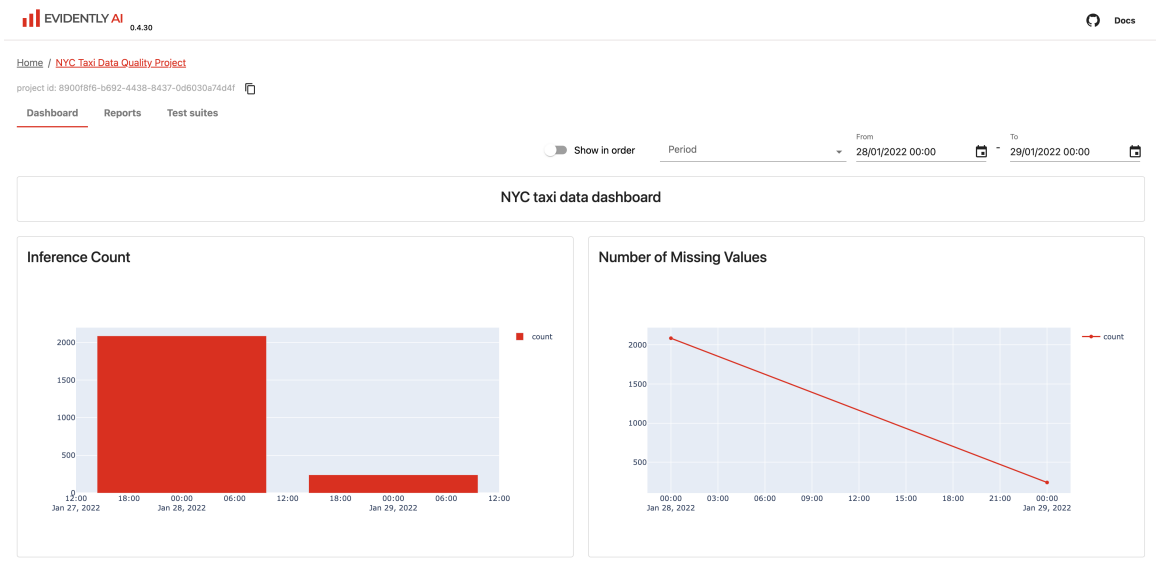

Figure 5.2: Evidently Monitoring Dashboard

With the dashboard, we can build pretty nice reports to monitor your data and models especially for batch models deployed in production very quickly.

5.5 Data Quality Monitoring

Here we describes ways to modify a script to create a data monitoring dashboard.

Load Reference Data and Model

- Script now includes functions to load reference data and a model.

Daily Data Processing

- We read all data initially.
- Simulates daily usage by processing data in 10-second intervals.
- Creates a current data set for each simulated day.

Report Generation

- Existing report generation code is reused to transform data.
- Script calculates metrics like prediction drift, drifted columns, and missing value share.

Database Update

- We then insert calculated metrics into a database table.

We can also convert the script into a Prefect pipeline using decorators. In data quality monitoring, we used Grafana to visualize the metrics. We populate the dashboard with data for prediction drift, number of drifted columns, and missing value share.

5.6 Saving and Reusing Grafana Dashboards

In order to save and reuse Grafana dashboards, first we create a configuration file, update docker compose, and configure the dashboard.

Steps:

1. **Configuration Files:**
   - Create a YAML file (`grafana_dashboards.yaml`) in the **`config`** directory to specify general dashboard settings.
   - Create a directory named **`dashboard`** to store individual dashboard configurations.
   - Inside **`dashboard`**, create a JSON file (e.g., `data_drift.json`) for each specific dashboard. This file will store detailed configurations for that dashboard.
2. **Update Docker Compose:**
   - Modify the `volumes` section of your Grafana service in `docker-compose.yml`.
   - Add volume mounts for the `grafana_dashboards.yaml` file and the **`dashboard`** directory. This allows Grafana to access the configuration files during startup.

```
    volumes:
      - ./config/grafana_dashboards.yaml:/etc/grafana/
   provisioning/dashboards/dashboards.yaml:ro
      - ./dashboards:/opt/grafana/dashboards
```

3. **Configure Dashboard JSON:**
   - In Grafana, access the settings for your desired dashboard.
   - Export the dashboard configuration as a JSON model.
   - Paste the exported JSON code into the corresponding JSON file (e.g., `data_drift.json`) within the **`dashboard`** directory.

Once, we've edited the code, we then build docker-compose. In the terminal, run `docker-compose up -build`. In a second terminal, run `python evidently_metrics_calculation.py`. Once docker is up and running, check the dashboard by login into Grafana at `http://localhost:3000`. We should see a dashboard like **Fig. 5.3**.

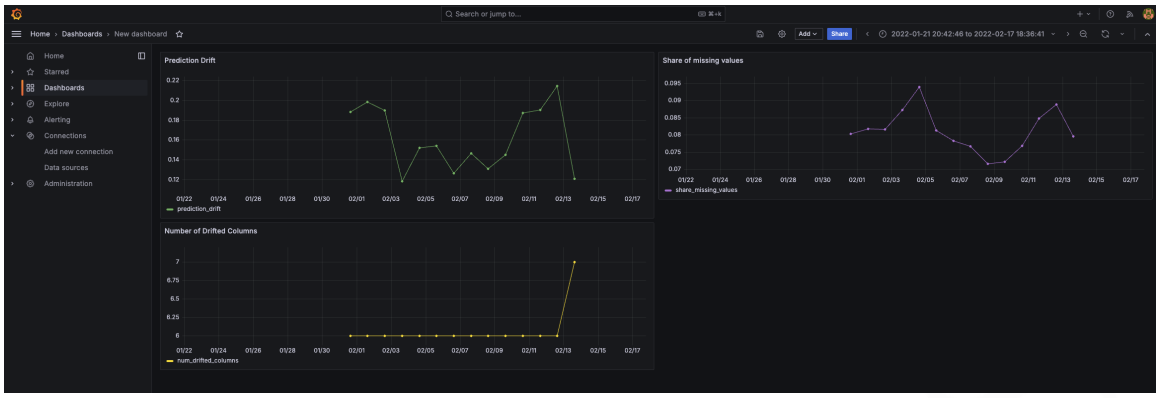

Figure 5.3: Grafana Saved Dashboard

5.7 Debugging with Evidently Library

This document outlines the debugging process using Evidently library for data and model issues identified through monitoring.

5.7.1 Benefits of Evidently

Evidently offers various functionalities for data monitoring and debugging:

1. **Visual Reports:** Generate visual dashboards to understand data drift and other issues.
2. **Test Suites:** Provide pre-defined tests to check for specific data or model problems.
3. **Detailed Analysis:** Offer in-depth analysis of data changes and potential causes.

5.7.2 Using Evidently for Debugging

The following steps demonstrate how to use Evidently for debugging:

1. **Import Libraries:** Import necessary libraries like ‘evidently‘ for reports and test suites, specific metric libraries, and test preset libraries.
2. **Load Data:** Load separate reference data (used for training) and current data.
3. **Load Model:** Load the model you intend to debug.
4. **Prepare Data:** Select the problematic part of the current data for analysis.
5. **Using Test Suites:**
    - Define ‘column_mapping‘ to specify feature mapping between reference

and current data.
- Exclude target columns if not relevant (e.g., 'Target=None').
- Use 'evidently.test_presets.DataDriftPreset' for data drift tests.
- Run the test suite with 'suite.run(reference_data, current_data, column_mapping)'.
- Visualize results using 'suite.show(inline=True)'.

6. **Using Reports:**
    - Create a report object using 'evidently.Report' and specify metrics (e.g., 'evidently.Report(evidently.MetricPreset.DATA_DRIFT)').
    - Run the report with 'report.run(reference_data, current_data, column_mapping)'.
    - Visualize results using 'report.show(inline=True)' or save as HTML for offline viewing ('report.save_html("report.html")').

Evidently's test suites and reports provide valuable tools for debugging data and model issues. They offer clear visualizations and detailed analysis to identify and address data drift and other performance problems.