ML on Production

Ensuring Reliability and Scalability

Muhammad Nanda S.

Content

Introduction

Project Overview

This project demonstrates an end-to-end machine learning **for production purposes**. While not a full MLOps implementation, it incorporates key practices to ensure the model is **scalable**, **reliable**, and **maintainable in production**.

Key Objectives

The main objectives of this project are to:

1. Develop & deploy a machine learning model.
2. Ensure the model's reliability and scalability.
3. Implement version control and monitoring for the model in production
4. Utilize related tools.

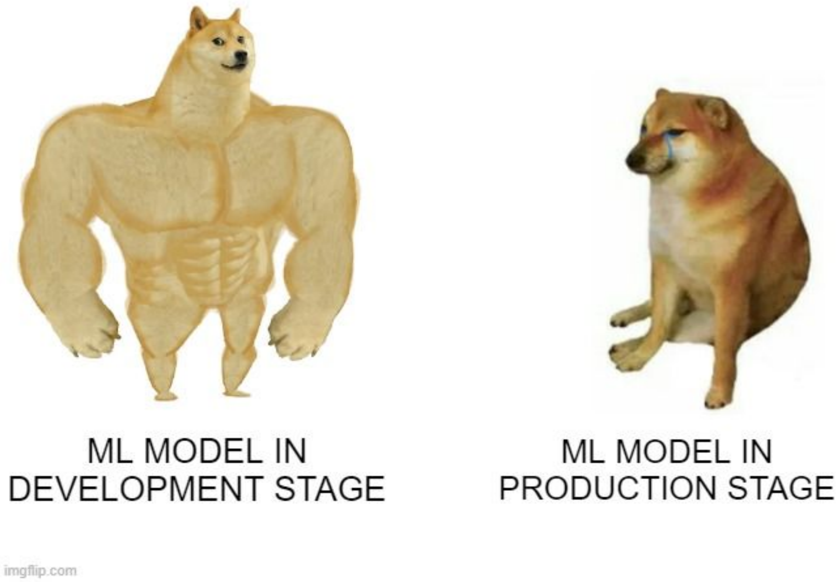
Note: *everything will be done proportionally so that it can become a comprehensive process unit*

Why ML on Production?

Deploying machine learning models into production is a critical step in deriving value from ML. It bridges the gap between model development and its effective use in business by:

- Ensuring the model performs consistently in real-world scenarios.
- Tracking model performance over time.
- Maintaining version control to manage updates and improvements.

However, this requires considerations beyond model development, including perspectives from other teams.



Even if we create a good model in the development phase, it does not necessarily mean that the model will perform well in the production phase.

Tools & Technologies

Here are some of the tools and technologies used in working on the project:

Python Programming Language
For developing the model and pipeline.

Neptune.ai
Provides a centralized platform for tracking model performance and experiments.

Git
Used for code and data versioning.

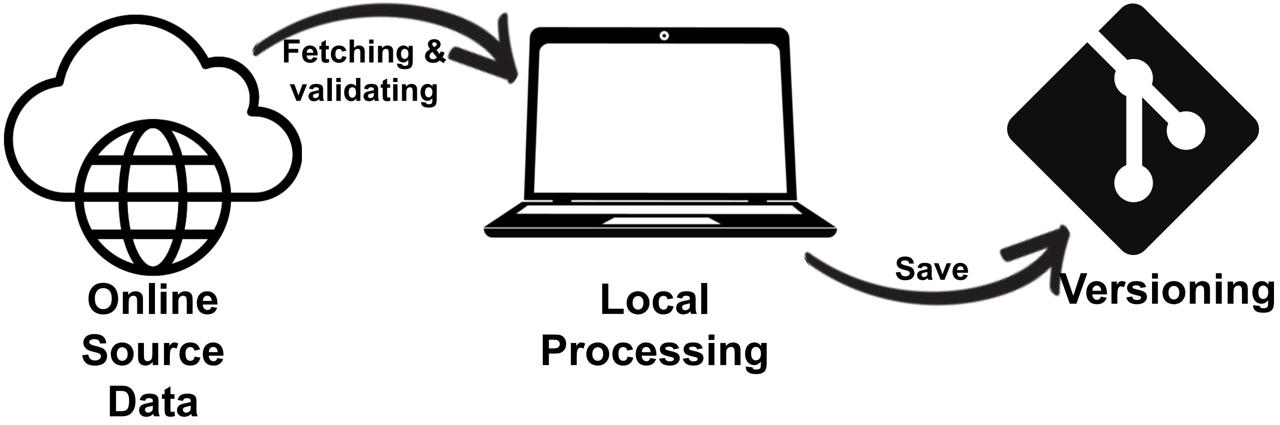
Docker
To containerize the model and ensure consistent deployment across environments.

AWS (Amazon Web Services):
To deploy the model in a scalable cloud environment.

Data Management & Processing

Overview

In this project, the dataset used is the Boston Housing dataset, a classic dataset used for regression tasks. The data provides various features about houses in the Boston area, and our goal is to predict the median value of owner-occupied homes (MEDV) using these features.



The data management process included **fetching, validating, and preprocessing** the data to prepare it for model training and deployment.

Data Description

Each record in the dataset describes a house in Boston suburb or town. The data was drawn from the Boston Standard Metropolitan Statistical Area (SMSA) in 1970.

The dataset consists of **506 rows non-null data** & has **13 attributes**/columns. The detail shown in the right

- **CRIM**: Per capita crime rate by town
- **ZN**: Proportion of residential land zoned for lots over 25,000 sq.ft.
- **INDUS**: Proportion of non-retail business acres per town
- **CHAS**: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- **NOX**: Nitric Oxide concentration (parts per 10 million)
- **RM**: The average number of rooms per dwelling
- **AGE**: Proportion of owner-occupied units built before 1940
- **DIS**: Weighted distances to five Boston employment centers
- **RAD**: Index of accessibility to radial highways
- **TAX**: Full-value property-tax rate per 10,000 dollars
- **PTRATIO**: Pupil-teacher ratio by town
- **LSTAT**: % lower status of the population
- **MEDV**: Median value of owner-occupied homes in 1000 dollars

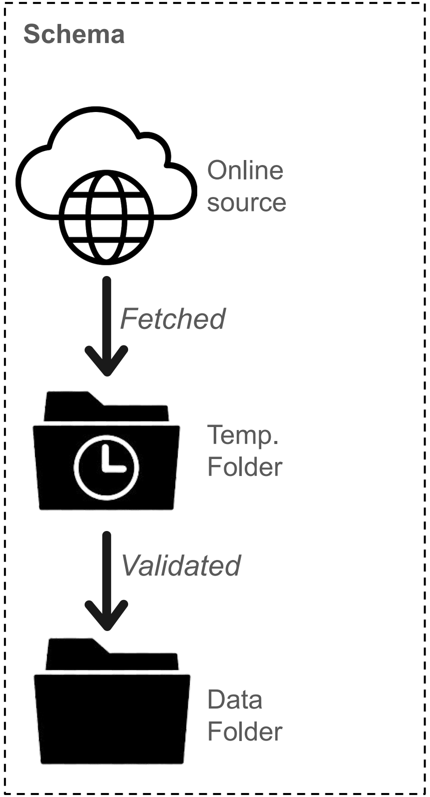
Data Fetching & Validation

Steps:

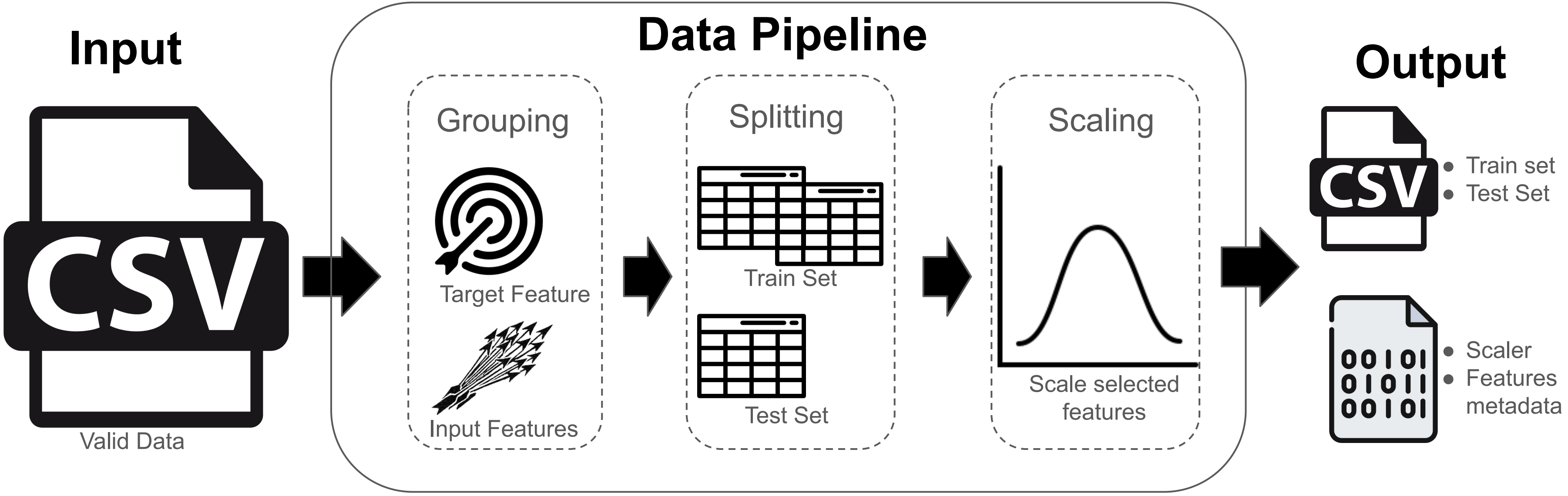
- Fetched the online data into a temporary folder.
- Each data point was validated in terms of data type, and invalid data points were removed from the dataset.
- The valid data was then exported to a data folder in CSV format.

Sample data overview

1	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	LSTAT	MEDV
2	0.00632	18	2.31	0	0.538	6.575	65.2	4.09	1	296	15.3	4.98	24
3	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	9.14	21.6
4	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	4.03	34.7
5	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	2.94	33.4



Data Processing Overview



Note: *Focused on the full ML pipeline, this project uses a simple processing flow, reflecting the clean and straightforward nature of the data.*

Grouping

- The target variable name (**TARGET**) is defined as MEDV (from data description).
- Feature sets are identified by separating features by data type (float and integer).
  - **FLOAT_FEATURES**: CRIM, ZN, INDUS, NOX, RM, AGE, DIS, PTRATIO, LSTAT
  - **INT_FEATURES**: CHAS, TAX, RAD
- A dictionary called **FEATURES** was created to store these feature sets for future reference.

Splitting & Scaling

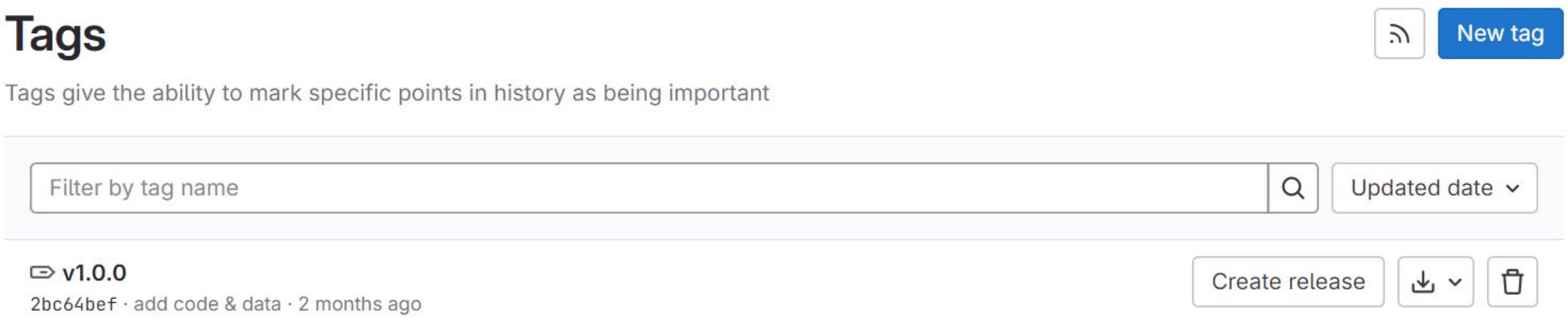
The data was split into training and testing sets to evaluate model performance. We ensured reproducibility by using a fixed random state:

- Train set: 80% (404 data points)
- Test set: 20% (102 data points)

Float features were scaled using a Standard Scaler to normalize the data, helping to improve model performance. Also to ensure there is no data leakage, the scaling process is carried out as follows:

- **Fit** the scaler **Only** to **FLOAT_FEATURES** in Train set
- **Apply** the fitted scaler to **FLOAT_FEATURES** in Test set

Versioning



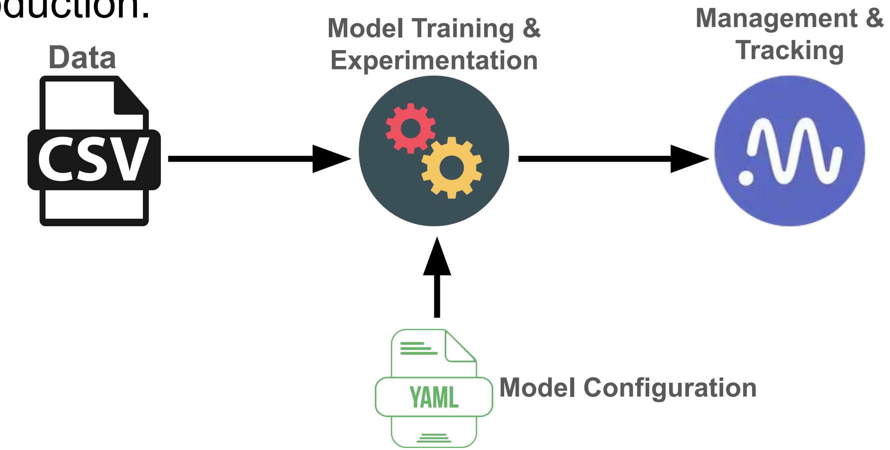
To ensure reproducibility and traceability of the data used in this project, data versions were tracked using Git/GitLab. This ensures that the exact version of the data used in model training can be retrieved at any point, improving model reliability in production environments.

Model Development

Overview

The scopes of model development in this project are:

- Develop model to predict house prices using various regression models.
- Incorporates model management and tracking with Neptune.ai in the training process
- All relevant metrics, hyperparameters, and configurations are logged for seamless analysis and reproduction.



Model Development Schema

Regressor Models (Main)

Main Regressor: Regressor to predict house price

Model Used:

1. **Linear Regression**: Simple model, used as baseline predictions.
2. **Ridge Regression**: Adds regularization to reduce overfitting.
3. **Huber Regressor**: Robust to outliers.
4. **Random Forest Regressor**: Ensemble model for more robust options.

Regressor Models (Secondary)

Secondary Regressor: Predicting House Price Intervals

Why?

- This project implements uncertainty detection. The secondary regressor model will be used as one approach to determine the uncertainty. When the main regressor's prediction falls within the interval generated by the secondary regressor, it is labeled 'certain.' Predictions outside this interval are labeled 'uncertain.

Model used:

1. **Quantile Regression**: Predicts intervals for upper and lower quantiles.
2. **Gradient Boosting Regressor**: Optimized through gradient descent for accurate interval prediction.

Flexible Model Configurations

The model has been configured via a **yaml** file.

This **allows for flexible and easy adjustment of hyperparameters**, such as the quantile that the model tries to predict in Quantile Regressor, number of trees in the Random Forest, etc.

```yaml
model_versions: "v1.0.0"
Methods:
  - name: "Upper Estimator_QR"
    config:
      quantile: .9
      alpha: 0
      solver: "highs"
  - name: "Linear Regression"
    config:
      fit_intercept: True
```

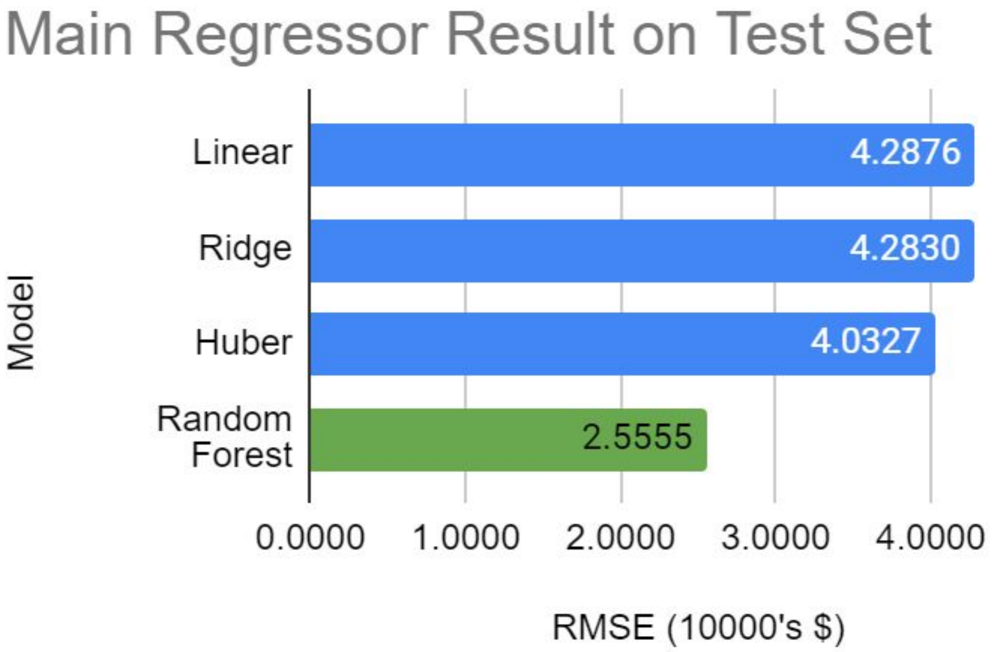
Example of model_config.yaml

Metrics

- **RMSE** (the main metric for the main regressor) gives more weight to larger errors due to the squaring and is more interpretable in terms of the original units (house prices). It captures the standard deviation of errors, giving a sense of how far predictions deviate from actual values.
- **Pinball Loss** (main metric for secondary regressor) evaluates how close predicted quantiles are to the actual outcomes, penalizing overestimation and underestimation differently depending on the quantile being predicted.
- **MAE** (optional) gives a straightforward interpretation of average error. It doesn't emphasize larger errors as much, which might be important when predicting expensive items like houses.
- **MSE** (optional) similar to RMSE but can be harder to interpret due to the squared scale of errors.

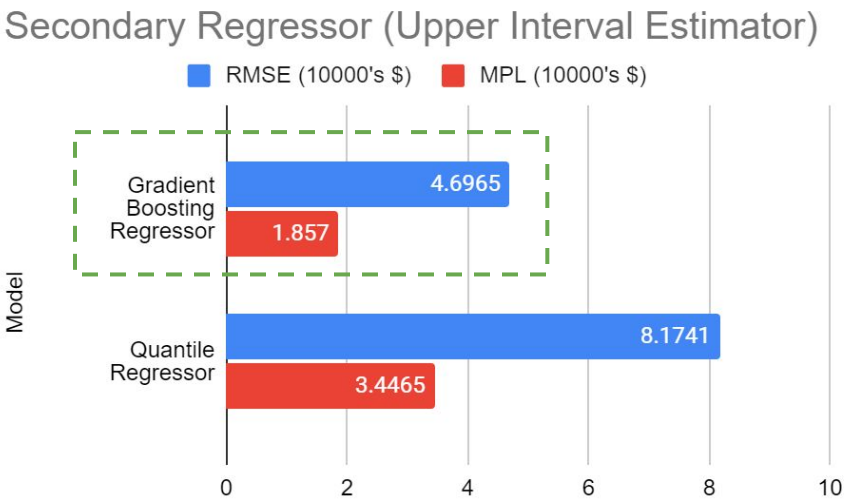
$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \qquad \text{Pinball Loss} = \frac{1}{n}\sum_{i=1}^{n}\rho_\tau(y_i - \hat{y}_i)$$

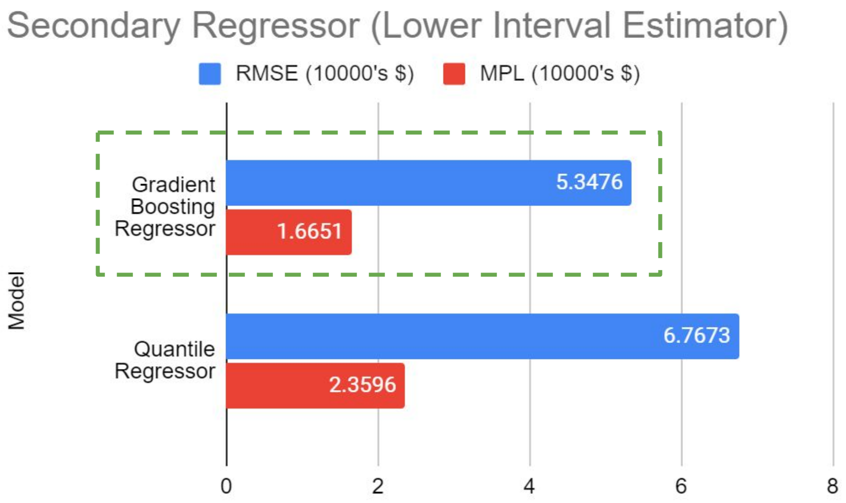
Metrics



- The RMSE is measured in 10000's dollars ($), with lower values indicating better performance.
- **Random Forest** outperforms the others with an RMSE of **2.5555**

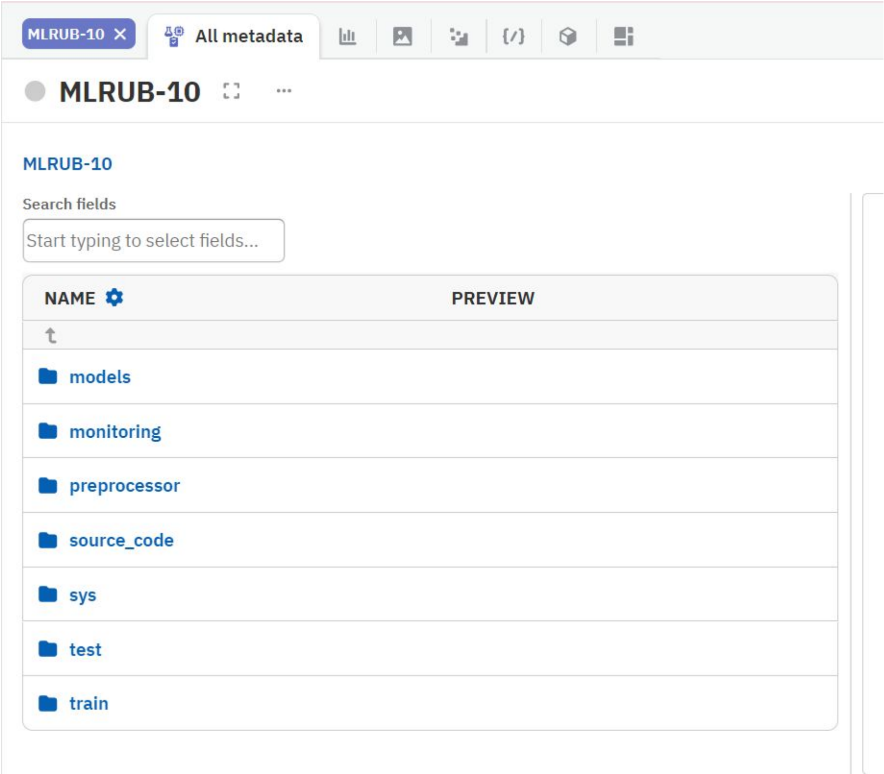
Metrics





- For a balance between accuracy and reliable interval prediction, Gradient Boosting Regressor selected to estimate the interval

Model Management with Neptune.ai



By integrating Neptune.ai, this project captures essential elements such as models, monitoring metrics, and training processes. The folder structure in the image exemplifies the systematic tracking and versioning, which are critical for transparency and model reproducibility.

Integration Steps

Preparation:

1. Create a Neptune.ai Account
2. Retrieve the API Token to authenticate Neptune from local environment or script.

Code Implementation (Simplified):

```python
# 1. Initialize Neptune
run = neptune.init_run(project=PROJECT_NAME, api_token=TOKEN)

# 2. Train the model
model.fit(x_train, y_train)

# 3. Save and Log Model Artifacts
model_filename = f"{model_name}_{model_version}.bin"
joblib.dump(model, model_filename)
run[f"models/{model_name}/{model_version}/artifact"].upload(model_filename)

# 4. End the Neptune run
run.stop()
```

Model Serving and Deployment

Deployment Process Overview

Step 1

Local Deployment

- **Easier Debugging**: Identifies issues with model usage and API before scaling.
- **Validation**: Ensures the model behaves as expected with real data and the prediction flow is correct.

Step 2

Cloud Deployment

- **Scalability**: The cloud enables handling more traffic and requests than local environments.
- **Cost-Effectiveness**: Resources are only used when needed, making cloud deployment efficient.

Local Deployment of House Price Prediction API

- The deployed API is built using **FastAPI**, known for its high performance and flexibility.
- The project includes **uncertainty detection** in predictions, where the model provides intervals (upper and lower bounds) for house prices. If the prediction falls within this interval, it's marked as "certain"; otherwise, it's "uncertain."
- **Docker** is used for containerization, ensuring the application can be run across any environment without setup inconsistencies.

Technical Setup for Local Deployment

- **.env File**: Key environment variables, including the model version and Neptune API credentials, are managed using a .env file
- **Docker Integration**: prepare mandatory files before creating a docker container:
  - requirements.txt (to match dependencies in development)
  - Dockerfile (text document that contains all the commands a user could call on the command line to assemble an image)

```
MODEL_VERSION=<your_model_version>
NEPTUNE_API_TOKEN=<your_api_token>
```

Example .env file

```
# Build the Docker image
docker build -t prediction-
engine:latest .

# Run the Docker container
docker run -itd --name prediction-
engine --restart unless-stopped --
memory="3G" -p 8000:8000 -d
prediction-engine:latest
```

Example docker command to build & run image

Usage

Call API & Give Input

```
curl -X 'POST' \
  'http://localhost:8000/predict/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "CRIM": 0.00632,
  "ZN": 18,
  "INDUS": 2.31,
  "CHAS": 0,
  "NOX": 0.538,
  "RM": 6.575,
  "AGE": 65.2,
  "DIS": 4.09,
  "RAD": 1,
  "TAX": 296,
  "PTRATIO": 15.3,
  "LSTAT": 4.98
}'
```

Output

```
{
  "input_data": {
    "CRIM": 0.00632,
    "ZN": 18,
    "INDUS": 2.31,
    "CHAS": 0,
    "NOX": 0.538,
    "RM": 6.575,
    "AGE": 65.2,
    "DIS": 4.09,
    "RAD": 1,
    "TAX": 296,
    "PTRATIO": 15.3,
    "LSTAT": 4.98
  },
  "prediction": {
    "Lower_Interval": 24,
    "Prediction": 25.4,
    "Upper_Interval": 30.25
```

Cloud Deployment of the API using AWS

The API is deployed on **AWS** for scalability and reliability.
Services used:

AWS ECR: To store Docker images.

AWS Lambda: For running serverless functions.

DynamoDB: To store and manage prediction data.

AWS Glue: For running scheduled tasks (invoking API simulation).

AWS Technical Workflow

Step 1: DynamoDB Integration

- Set up a DynamoDB table to store prediction results and data logs.

Step 2: Modify local deployment code

- Add Connection to AWS & DynamoDB to store Prediction
- Download model & preprocessor object from neptune.ai
- Make sure the script use the downloaded model & preprocessor
- Update Dockerfile

Step 3: Push Docker Image to AWS ECR

- Build the Docker image locally.
- Push the Docker image to AWS Elastic Container Registry (ECR).

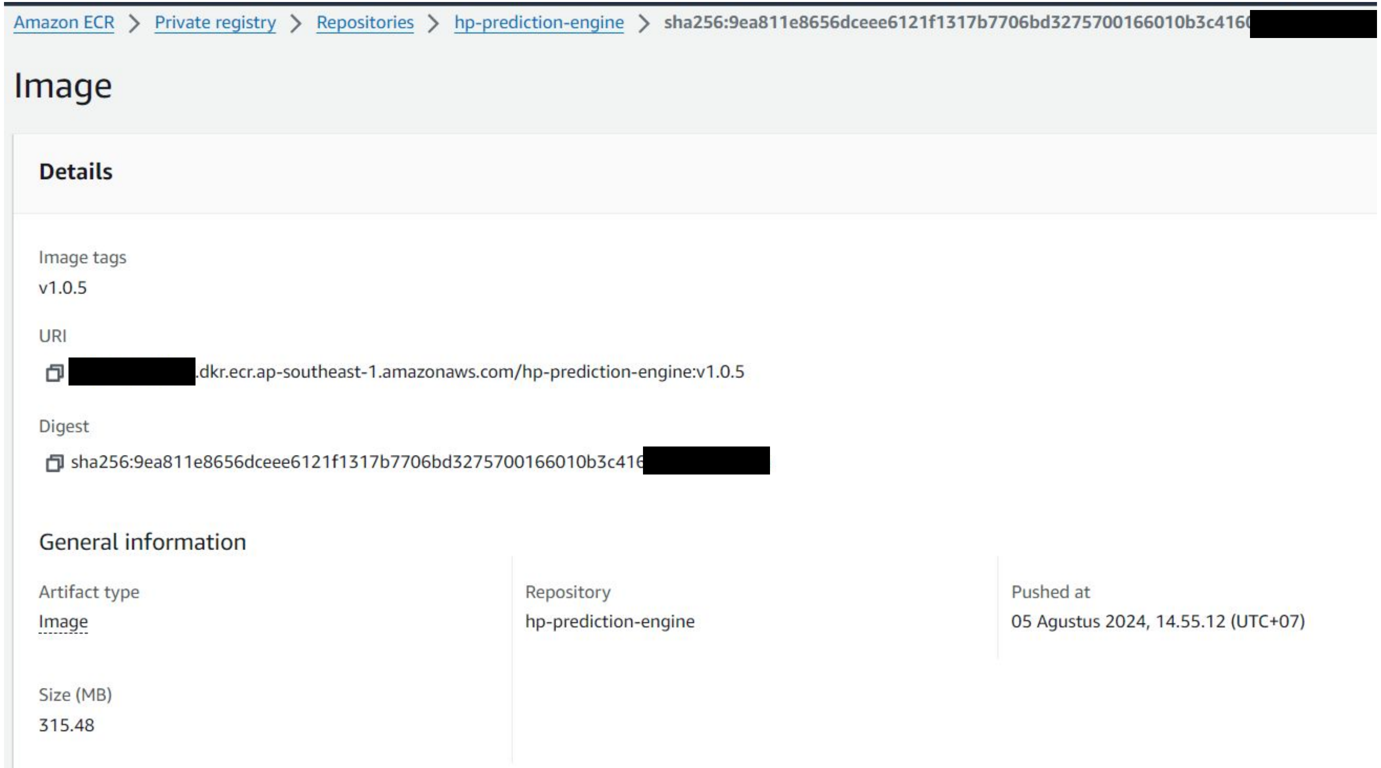
Step 4: Deploy with AWS Lambda

- Create an AWS Lambda function that uses the Docker image for serverless execution.

Step 5: Simulate Invoking API with AWS Glue

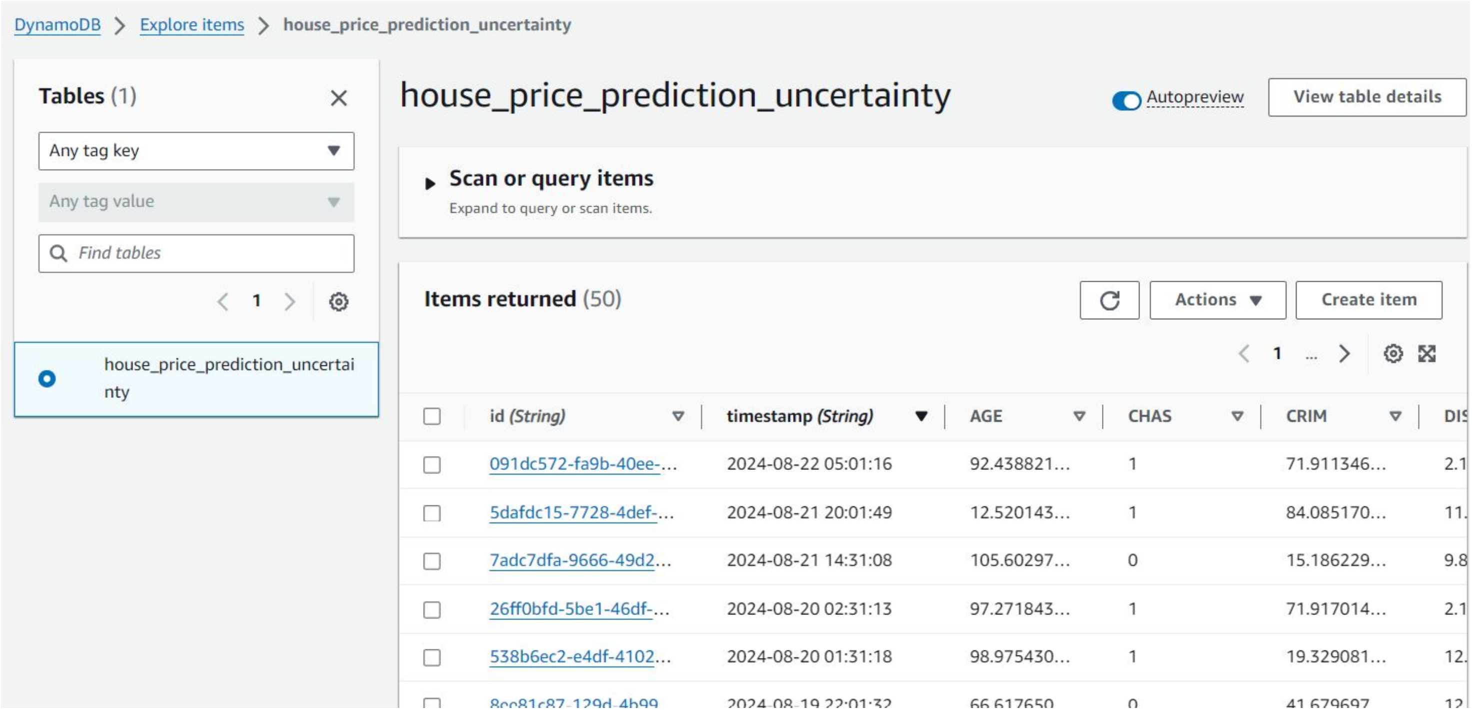
- Create scheduled script to invoke created API from AWS Lambda

AWS ECR Preview in AWS Console



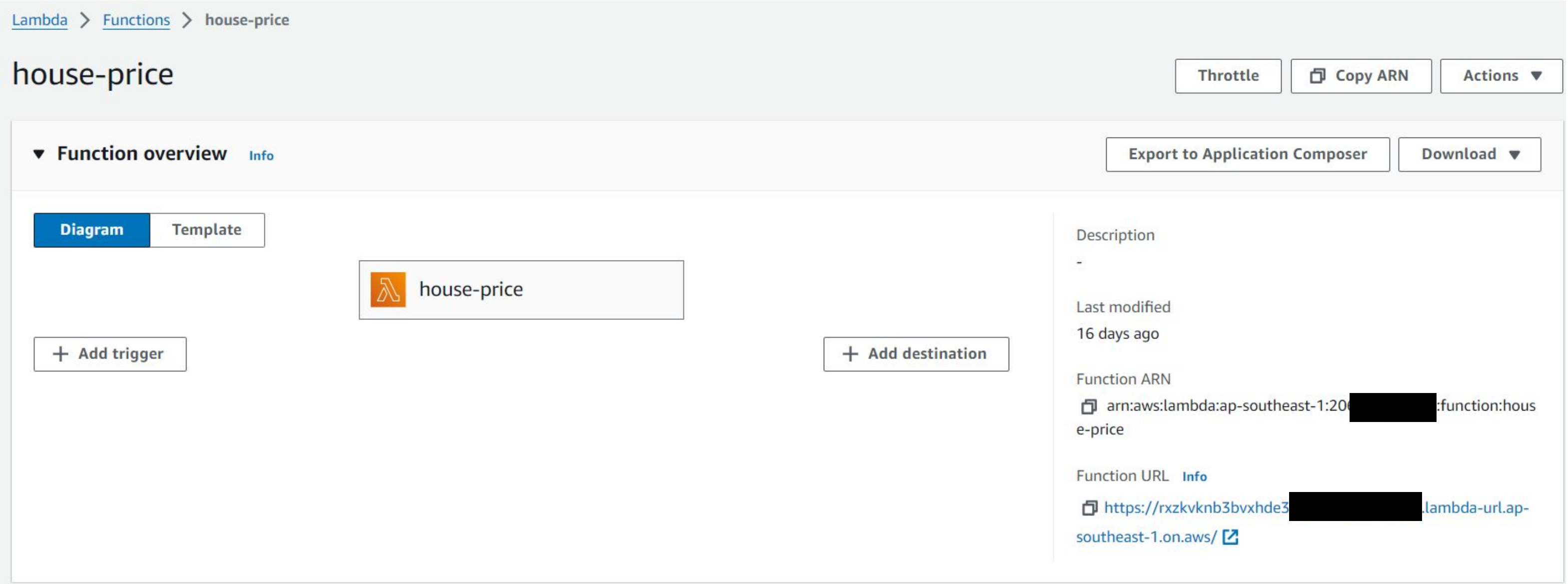
The Docker image of the ML model, stored in AWS ECR, includes tag, URI, and artifact information for production deployment.

DynamoDB Preview in AWS Console



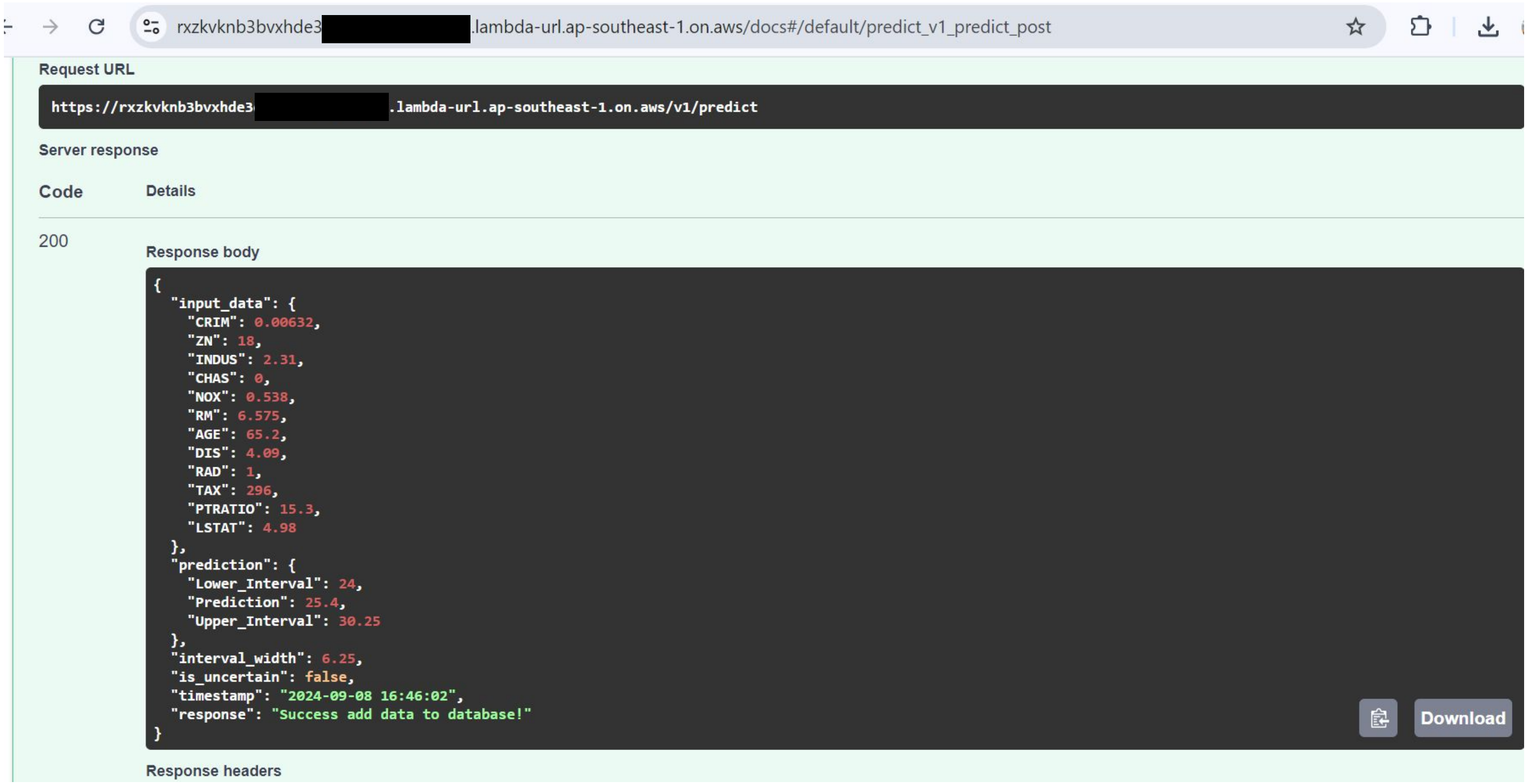
DynamoDB houses prediction data, store uncertainty values and feature inputs to the ML model.

AWS Lambda Preview in AWS Console



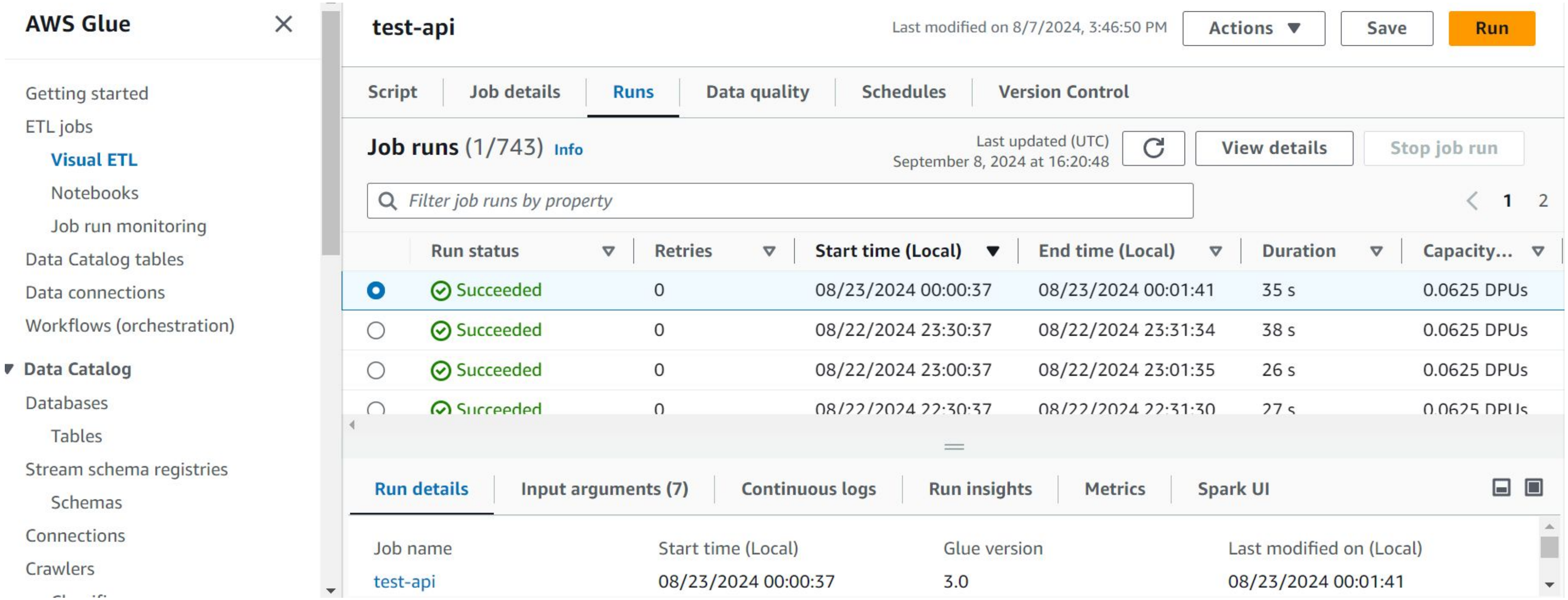
AWS Lambda view showcasing the deployed 'house-price' function, complete with ARN and function URL for API triggers.

Testing API from AWS Lambda



Successful prediction via AWS Lambda API endpoint.

AWS Glue Preview in AWS Console



Scheduling & tracking successful Invoke API via AWS Glue

Monitoring Prediction

Monitoring Model Performance and Uncertainty

Why Monitoring Matters?

- To ensure the deployed model continues to perform as expected in real-world conditions.
- Monitoring helps identify if the model's predictions are accurate and reliable, and whether uncertainty in predictions is increasing.

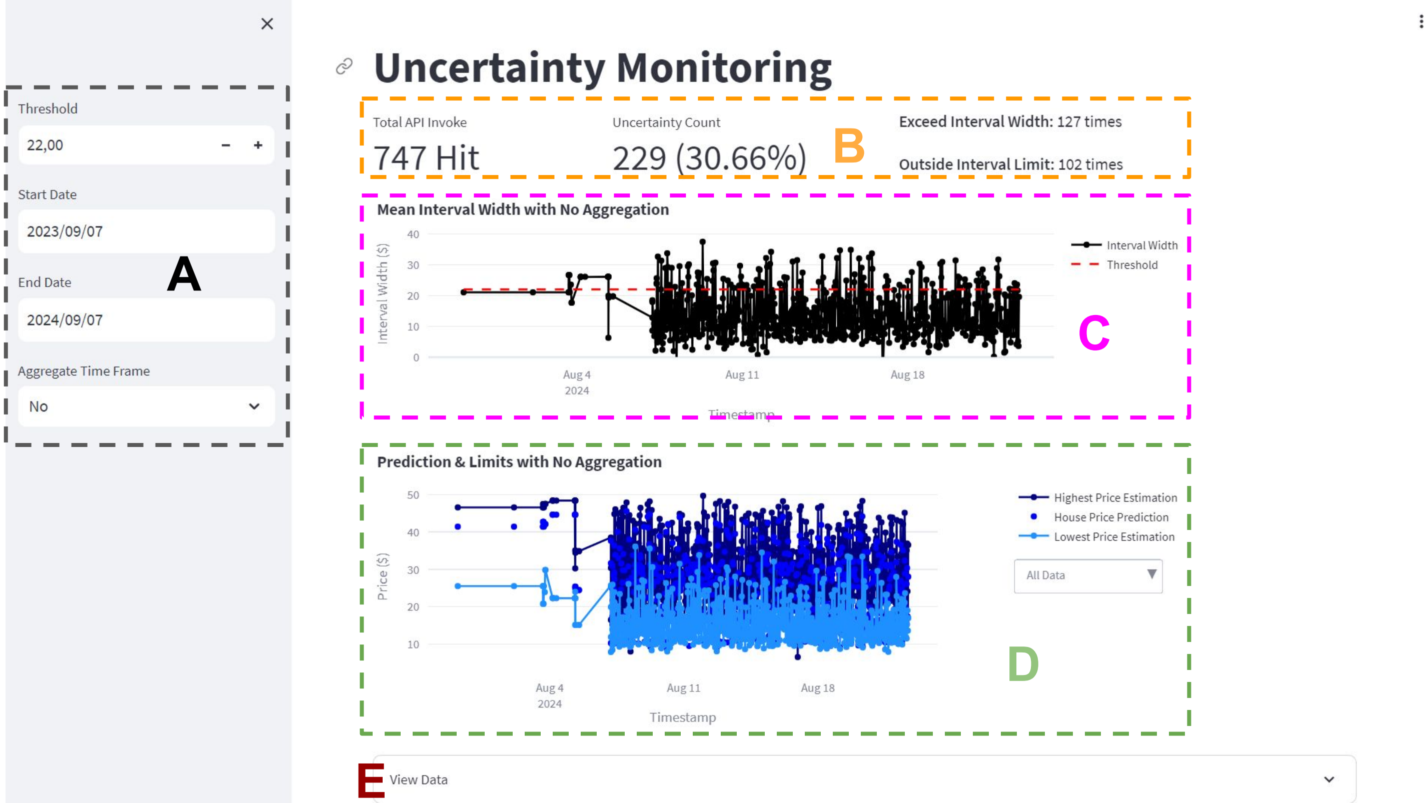
Focus on Uncertainty:

- Tracking the uncertainty of predictions allows us to know when the model might be unreliable, prompting further action or investigation.
- Monitoring involves reviewing intervals and identifying predictions labeled as “uncertain.”

Setting Up the Monitoring System

1. Pre-requisites:
   - The machine learning model already deployed on AWS.
   - Have at least one successful API invocation.
2. Creating the Environment:
   - Have access to AWS DynamoDB to retrieve prediction data
3. Design the Dashboard
   - Select the appropriate elements to focus on monitoring the prediction results and uncertainty.
   - Keep the dashboard simple & interactive

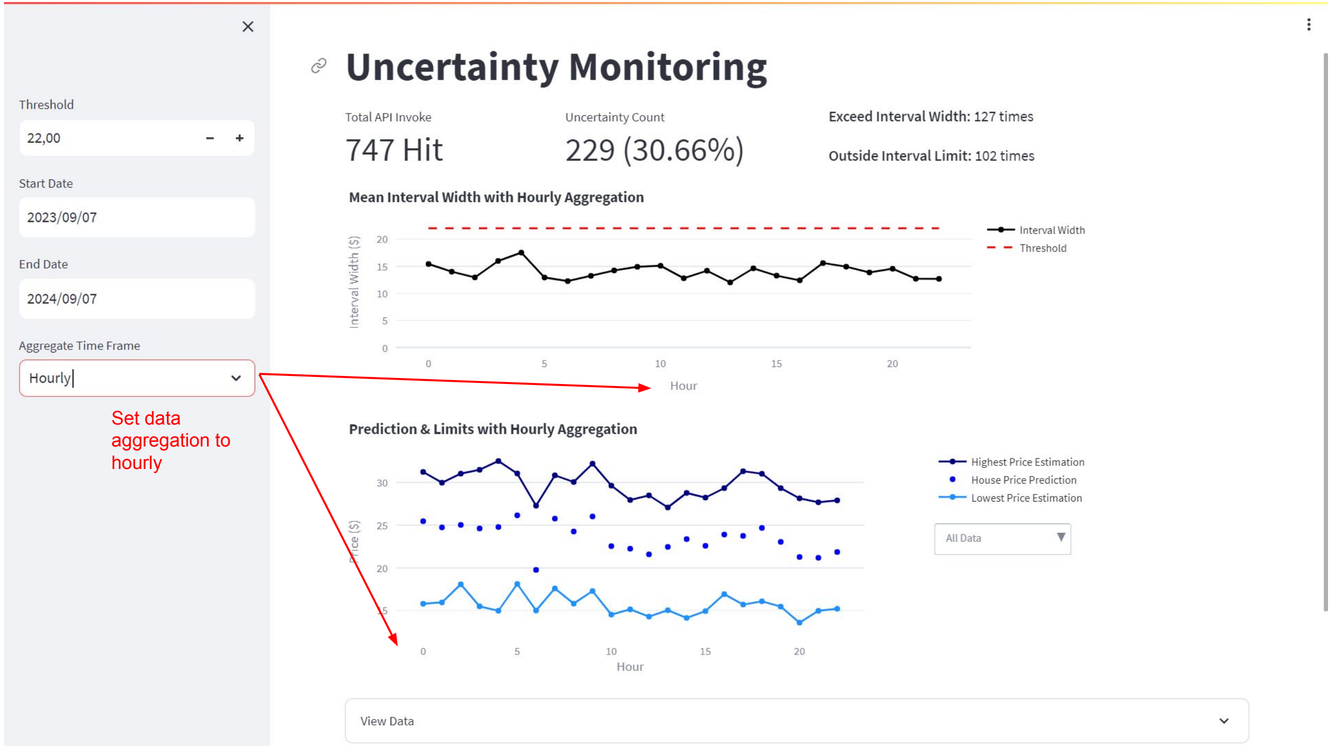
Dashboard: Overview



Dashboard Key Features

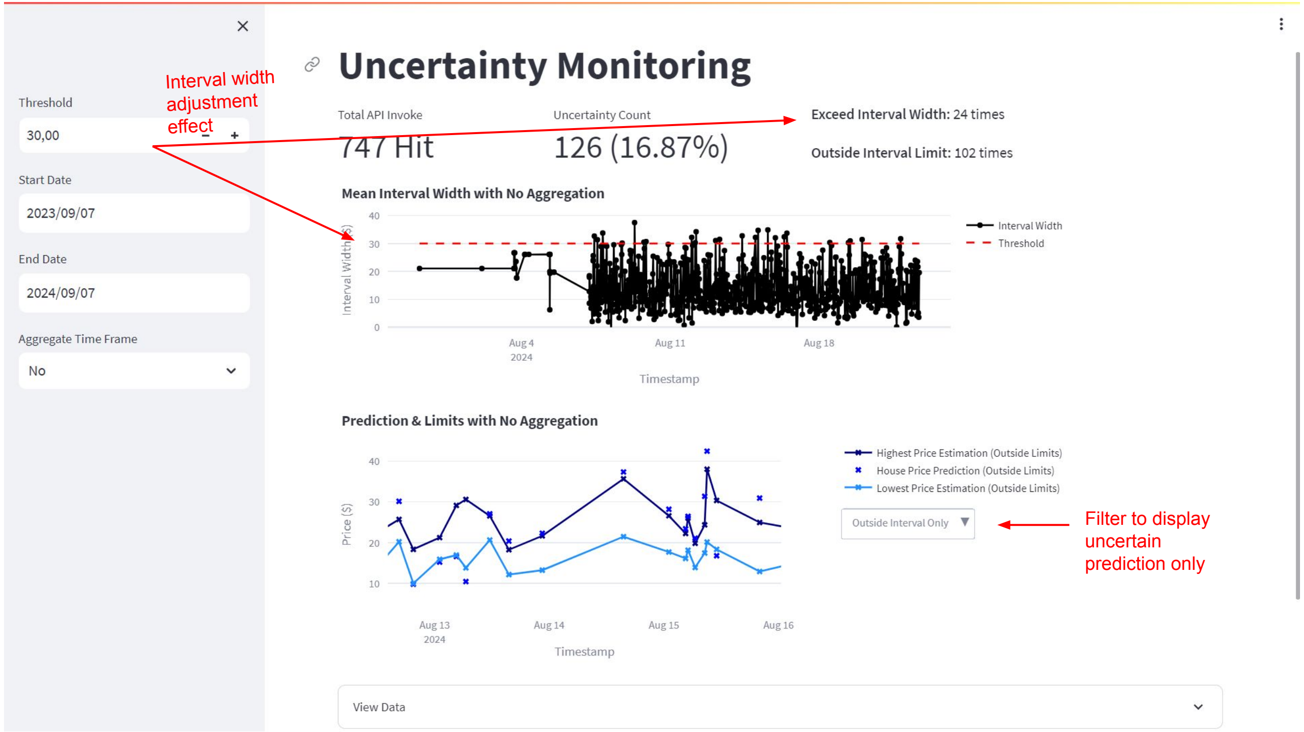
A. **Data Selector & Aggregation Settings**: Allows the user to filter predictions based on date ranges and set aggregation preferences (e.g., time frames or thresholds)
B. **Simple Scorecard**, as a quick snapshot of key metrics:
   - Total API invocations (how many predictions were made).
   - The percentage of uncertain predictions (highlighting the level of model reliability).
   - Counts for exceeded interval widths and outside interval limits.
C. **Interval Width Chart**: Display the width of the prediction intervals over time, compared to a set threshold (shown in red)
D. **Prediction & Limits Chart**: Display the predicted house price and its upper and lower limits for a given time period.
E. **Data Viewer:** Display the raw data retrieved from the database

Dashboard: Hourly Display



Real-time uncertainty and prediction monitoring with hourly aggregation settings.

Dashboard: Uncertain Display



Utilize the dashboard features to examine uncertainty details.

Future Tasks & Improvements

Future Tasks & Improvements

Review & Analyze Input and Predictions

- Regularly reviewing the input data and the model’s predictions (especially the intervals) is crucial to ensure that the model maintains high accuracy and relevance.

Add Input Data Monitoring from API

- Monitoring incoming data in real-time ensures that the data quality remains high and that any anomalies or unexpected inputs are caught early. This is especially important in production environments where input data can change over time.

Increase Data Pipeline Complexity

- Adding more data sources or preprocessing steps can improve the model's ability to generalize across different situations. Also, remember that this project currently implements a simple pipeline.

Model Tuning & Optimization

- This is a last resort, because retraining the model is not necessarily going to fix it right away when there is an error/inaccuracy in the prediction. We need to find the root cause first.

Thank You

Attachements

- Data repo
- Modeling repo
- Deployment (Local / testing) repo
- Deployment (for cloud) repo
- Monitoring repo