Documentation Report

StoreCast

(Aarthi Padmanabhan)

Introduction:

StoreCast is a machine learning-based forecasting application designed to help Walmart optimize store-level sales planning and inventory management. Developed with a focus on MLOps principles, StoreCast integrates multiple forecasting models, real-time user interaction, monitoring dashboards, and full-stack containerized deployment.

The solution provides weekly sales forecasts for Walmart stores using time series and machine learning models, exposed via a Flask-based web interface. It includes dynamic model loading, centralized monitoring, and scalable deployment using Docker on AWS infrastructure.

Objective:

The StoreCast system was built to solve key business problems affecting inventory, pricing, and revenue predictability. The objectives and their associated business challenges are outlined below:

Problem 1: Higher/Lower Stocks

Impact: Inaccurate sales predictions lead to product overstocking or understocking, disrupting inventory flow and causing wastage or missed sales.

Objective: Optimize stock levels by providing precise forecasts aligned with real demand trends, enabling Walmart stores to maintain efficient inventory levels.

Problem 2: Price Policy

Impact: Suboptimal pricing decisions, especially around promotions or seasonal shifts, can reduce revenue or lead to lost customer interest.

Objective: Enhance pricing strategies by forecasting demand patterns using historical sales, markdown schedules, and event-based variations to boost profitability.

Problem 3: Not Reaching Projected Target – Damaged Stock Prices

Impact: Failure to meet forecast targets negatively affects financial performance and investor confidence, risking damage to brand and stock price.

Objective: Maintain consistent sales target achievement by leveraging predictive models considering promotions, holidays, and historical sales spikes.

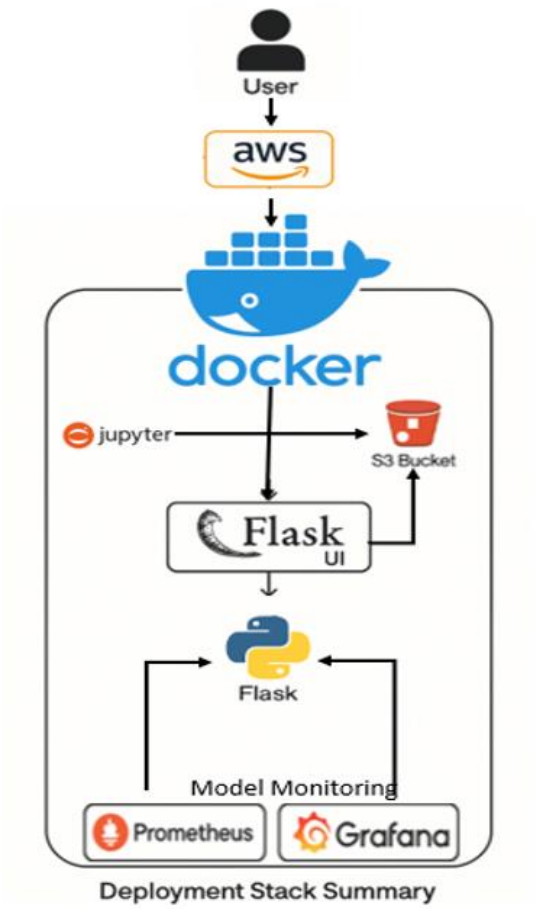
The system-level goals are:

- **Minimize Inventory Discrepancies**: Improve inventory turnover by aligning stock levels with demand forecasts.

- **Optimize Pricing and Promotions**: Use accurate forecasts to drive seasonal pricing decisions.
- **Ensure Financial Targets**: Provide store-level visibility into sales trends, reducing the risk of missing projected revenue targets.

Process Diagram:

The diagram below represents the high-level process for StoreCast:



1. **The user** interacts with the application via a web interface hosted on an AWS EC2 instance.
2. **AWS EC2** hosts Docker containers running Flask, Prometheus, and Grafana services. The instance is type t3.large with inbound ports open for:
    - Port 5001: Flask API and UI
    - Port 9091: Prometheus
    - Port 3001: Grafana The instance uses an **Elastic IP** to ensure consistent access.
3. **Docker Compose** orchestrates three services:
    - Flask (port 5001): Hosts the prediction service and exposes /metrics.
    - Prometheus (port 9091): Scrapes metrics from Flask.
    - Grafana (port 3001): Visualizes scraped metrics.

4. On instance reboot, a system service is located at /etc/systemd/system/storecast.service auto-restarts the containers.
5. Flask dynamically loads models from **AWS S3**, triggered by user input, and generates forecasts.
6. **Prometheus** collects metrics including MAE, RMSE, and API latency, defined in prometheus.yml:

```
scrape_configs:
  - job_name: 'flask-app'
    static_configs:
      - targets: ['flask:5001']
```

7. **Grafana** connects to Prometheus (http://prometheus:9091) as a data source with default credentials (admin/admin). It hosts a prebuilt dashboard for monitoring performance by store and model type.

End-to-End Request-to-Response Process Flow:

The user-to-output interaction for StoreCast is designed to be seamless and scalable. Below is the detailed flow of a request through the system:

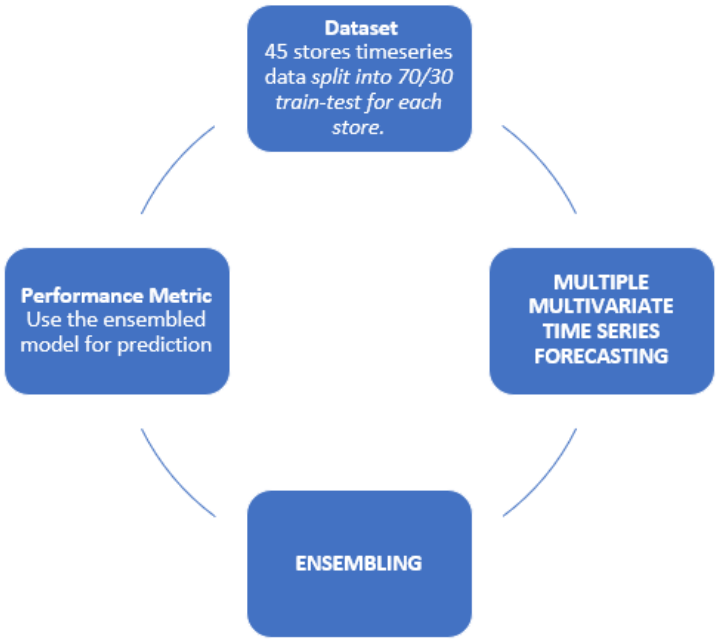
1. **User Request Initiation**:
    - A user accesses the Flask UI at http://<EC2_IP>:5001/prediction.
    - The user inputs the desired store ID and the forecast duration (e.g., 14 days).
    - This triggers a POST request to the /predict endpoint.
2. **API Request Handling (Flask)**:
    - The Flask application receives the request and parses the input.
    - It uses environment variables from a .env file to authenticate and connect with AWS S3.
3. **Model & Data Fetching from AWS S3**:
    - Based on the store ID provided, Flask pulls the following from the S3 bucket:
        - ARIMA model: store_{id}_auto_arima.pkl
        - XGBoost model: store_{id}_xgb_model.pkl
        - Historical data: clean_data.csv
4. **Prediction Logic**:
    - Both ARIMA and XGBoost models are used to generate independent predictions.
    - An ensemble technique averages these predictions to produce a final forecast.
    - Performance metrics like MAE and RMSE are calculated during inference.

5. **Response Generation**:
    - The final forecast results are rendered back on the UI.
    - A summary of the predictions is also available via JSON through the API.
6. **Metrics Collection and Exposure**:
    - Metrics such as prediction latency, request count, and error metrics are pushed to the /metrics endpoint.
    - Prometheus scrapes this endpoint at 15s intervals.
7. **Monitoring (Prometheus + Grafana)**:
    - Prometheus stores the metrics time series.
    - Grafana visualizes the forecast accuracy and service health using preconfigured dashboards.

This structured request-response lifecycle ensures real-time interactivity, transparency, and resilience in a production ML environment.

Models:

The following diagram summarizes the modeling process and result visualization:



The models are pre-trained per store and stored on S3 in the following structure:

- /models/store_{id}/store_{id}_auto_arima.pkl
- /models/store_{id}/store_{id}_xgb_model.pkl

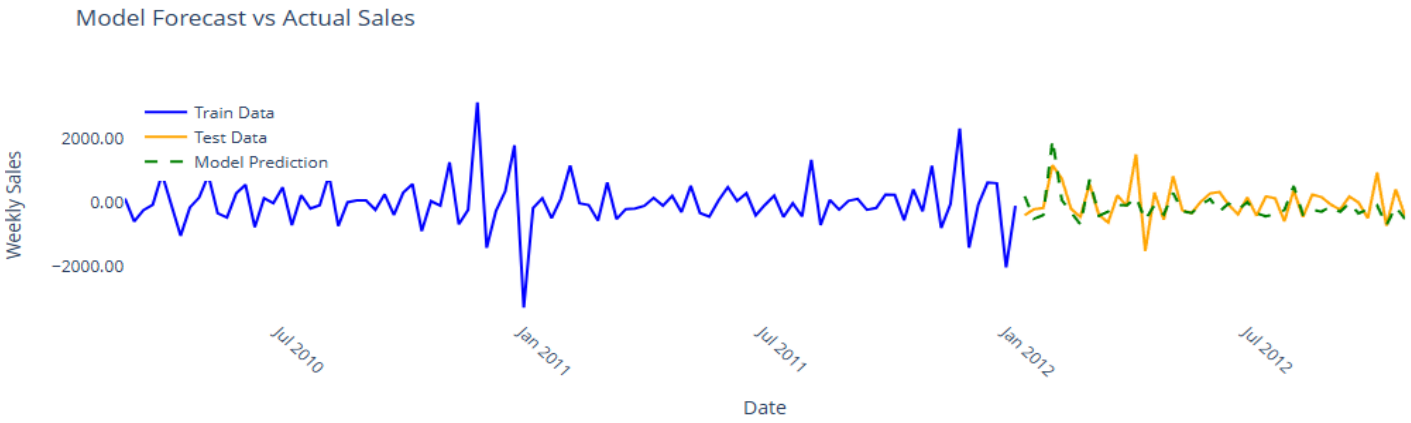
The dataset consists of historical weekly sales for 45 Walmart stores, split 70/30 into training and test sets.

The StoreCast system uses a combination of statistical and machine learning models tailored for time series forecasting:

1. **Auto ARIMA**
    - Automatically selects the best ARIMA parameters (p, d, q) for each store's weekly sales series.
    - Classical time series model for trend-seasonal forecasting.
    - Captures trend, seasonality, and autocorrelations using past observations.
    - Good for interpretable, linear relationships in stable time series.
2. **XGBoost Regressor**
    - A powerful gradient boosting framework that handles complex and non-linear patterns.
    - Used features like lagged sales, moving averages, holiday flags, and promotional windows.
    - Excellent for performance when time series exhibits irregularities.
3. **Ensemble Model**
    - Final predictions are generated by averaging outputs from ARIMA and XGBoost.
    - This leverages ARIMA's reliability in trends and XGBoost's adaptability to patterns and outliers.

These models are trained offline and stored per store on AWS S3. Flask loads the required models at runtime based on user input.

One of the store and model train data, test data and predictions:



Error Metrics:

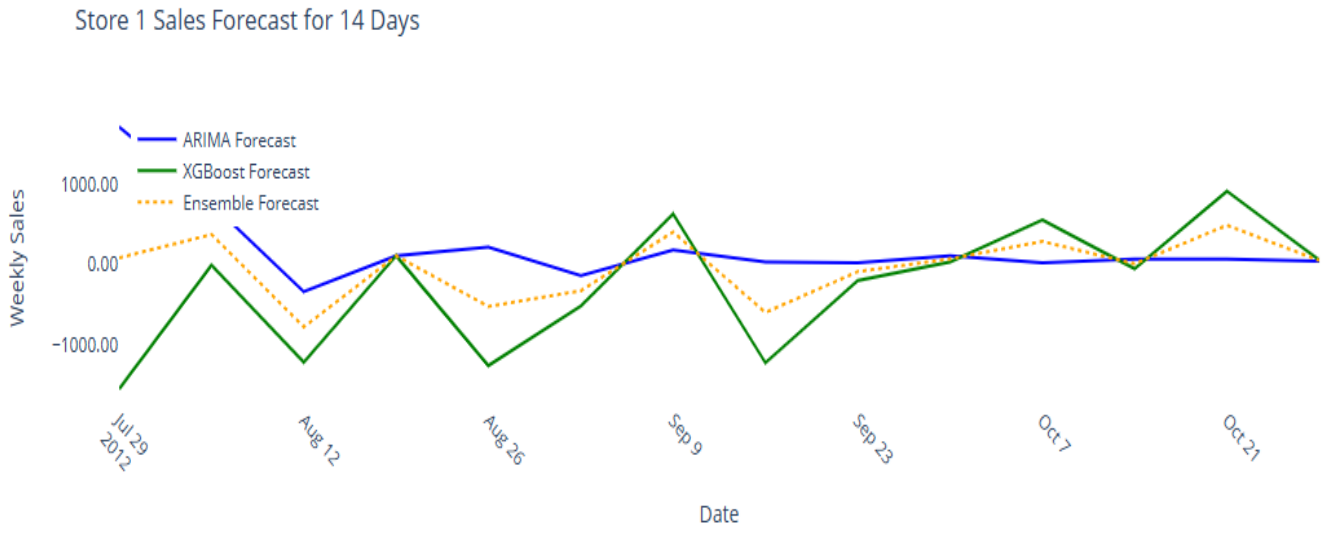
Metrics are computed post-forecasting for every user request and logged in Prometheus:

1. **MAE (Mean Absolute Error)**
    - Formula: MAE = (1/n) * SUM(|y_i - ŷ_i|)
    - Easy to interpret; measures average magnitude of error.

2. **RMSE (Root Mean Square Error)**
    - Formula: RMSE = sqrt((1/n) * SUM((y_i - ŷ_i)^2))
    - Penalizes large errors more severely; useful for critical forecasting.
3. **MAPE (Mean Absolute Percentage Error)**
    - Formula: MAPE = (100/n) * SUM(|(y_i - ŷ_i) / y_i|)
    - Expresses accuracy as a percentage; intuitive for business interpretation.
4. **Forecast Latency**
    - Time taken from input request to prediction return.
    - Captured using a custom Prometheus histogram: storecast_prediction_latency_seconds

Model Performance Summary:

Model	Accuracy (%)	MAE	RMSE	MAPE (%)
Auto ARIMA	74.3	768.92	1324.67	11.24
XGBoost	71.8	893.11	1456.39	10.87
Ensemble	78.2	712.54	1278.96	9.42



Model Monitoring Plan:

A robust monitoring framework is crucial for production ML systems. StoreCast uses the following components:

1. Prometheus Monitoring

- Scrapes metrics from Flask /metrics endpoint every 15 seconds.

- Tracks:
    - MAE, RMSE, MAPE(per store and model type)
    - Forecast latency
    - Request volume and status codes

2. Grafana Dashboards

- Prebuilt panels visualize error trends and traffic patterns.
- Enables:
    - Detection of model drift (if MAPE increases over time)
    - Request anomaly spotting

3. Alerts (Implemented)

- Thresholds are defined for key error metrics like MAPE and RMSE across selected and critical stores alerting is integrated by choosing the best out of the 3 models.

4. Logging and Troubleshooting

- Flask logs inference time and endpoint hits.
- Docker logs captured via docker-compose logs.
- Using EC2 to trace system service issues.

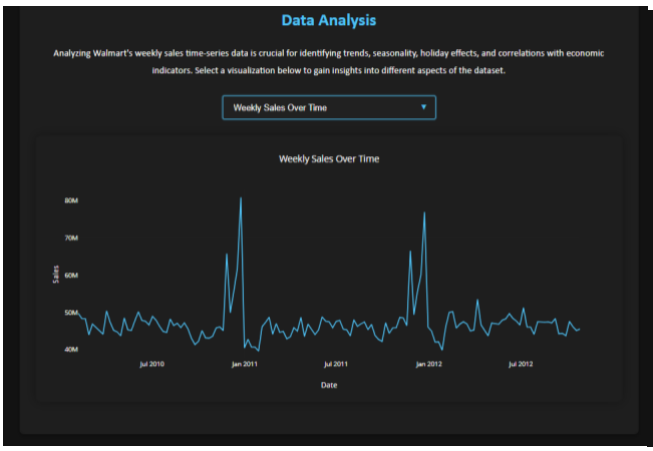
This multi-layered monitoring ensures that StoreCast can remain stable, transparent, and performance-aware in live environments.

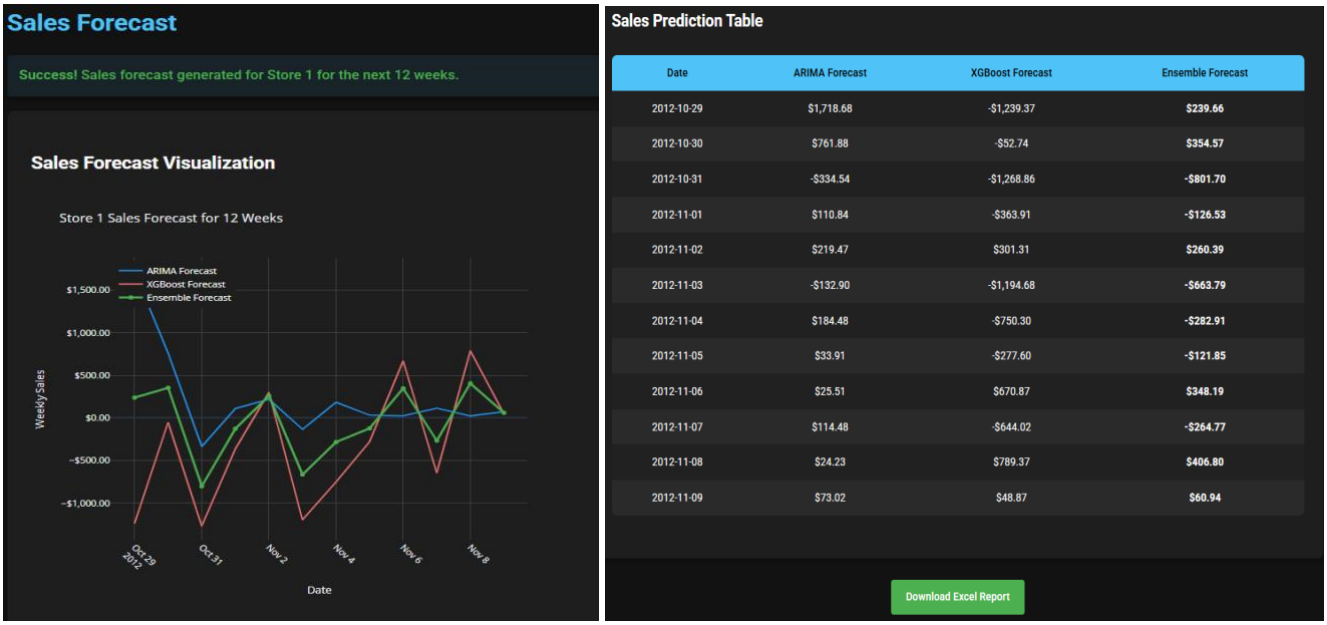
Links:

Flask App Interface

Purpose: Provides the frontend UI and backend API to input store ID and forecast duration.

Access: http://54.236.248.177:5001

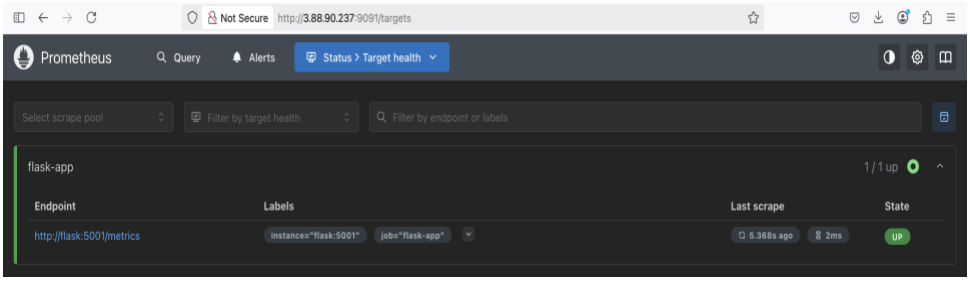




Prometheus UI

Purpose: Monitors and scrapes custom metrics from the Flask app like MAE, RMSE, and MAPE.

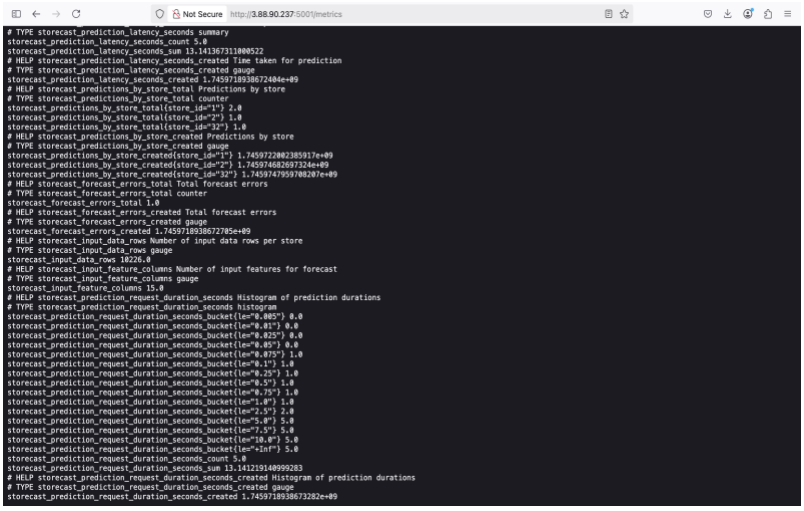
Access: http://54.236.248.177:9091



Prometheus Metrics Endpoint

Purpose: Flask exposes internal metrics used by Prometheus at this endpoint.

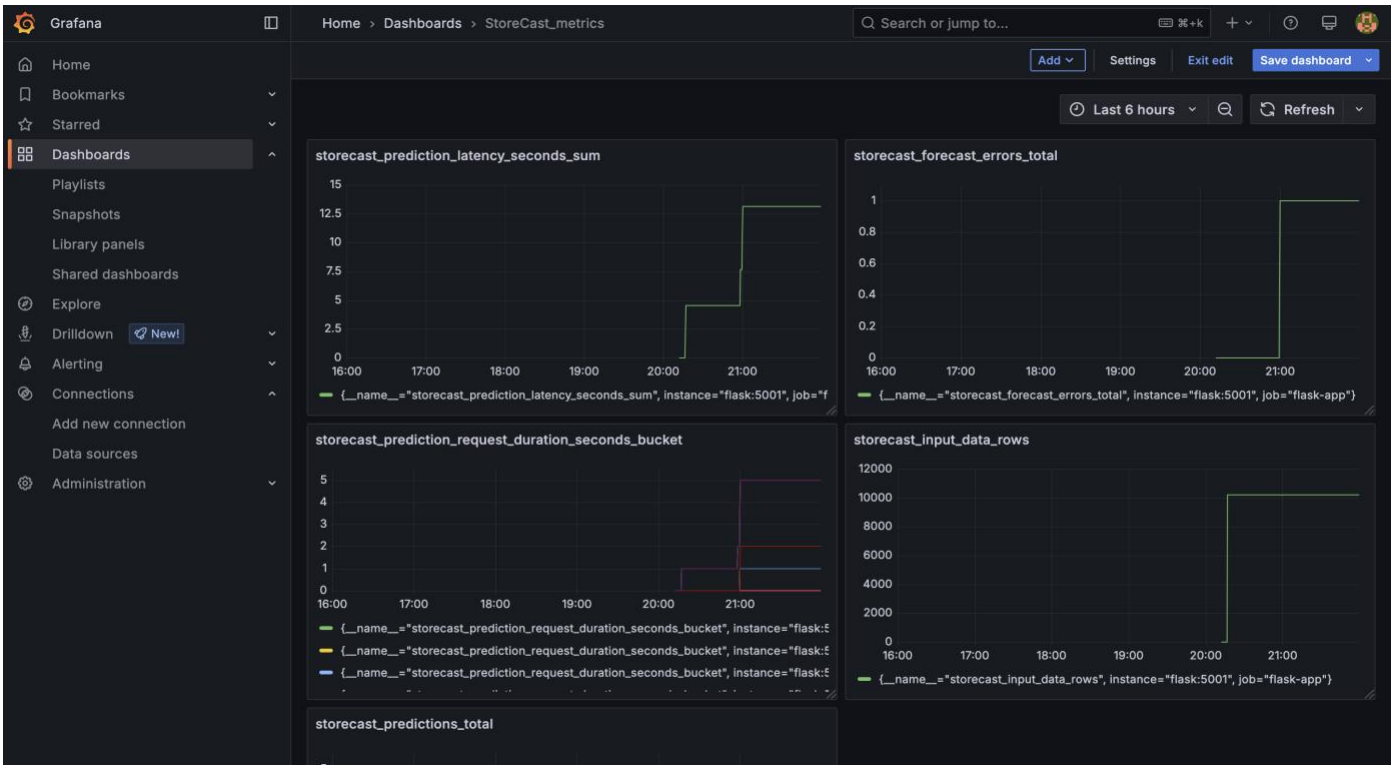
Access: http://54.236.248.177:5001/metrics



Grafana Dashboard

Purpose: Visualizes model performance and system metrics collected by Prometheus.

Access: http://54.236.248.177:3001/d/dekfbvpplyps0e/storecast-metrics



Docker Hub Repository

Hosts the Docker images for Flask, Prometheus, and Grafana used in production.

Access: https://hub.docker.com/repository/docker/aarthi9929/storecast

GitHub Repository

Contains all the source code, Docker configuration, model scripts, and .env sample. Please find the code here.

Access: https://github.com/AARTHI-PADMANABHAN/StoreCast

AWS EC2 Instance

The primary cloud host for all containers. Used with the system for persistent deployment.

Access: https://us-east-1.console.aws.amazon.com/ec2/home?region=us-east-1#InstanceDetails:instanceId=i-01f79e1f63b5789f7

⚠ Note: Currently we have stopped the EC2 instance as it has been charged hourly. If you can inform us when you will be evaluating the project we will start the EC2 instance and provide you with the new link.

AWS S3 Bucket

Storage for all store-specific model pickle files and input data used in forecasts.

Access: https://us-east-1.console.aws.amazon.com/s3/buckets/walmart-mlops-forecast?region=us-east-1&tab=objects&bucketType=general

Conclusion:

StoreCast is a production-ready ML application that demonstrates how sales forecasting can be effectively scaled using MLOps frameworks. From model deployment and monitoring to dynamic data access, the project ensures end-to-end automation, reproducibility, and operational efficiency.

By integrating Prometheus and Grafana, it provides transparency and visibility into model behavior post-deployment, aligning well with enterprise reliability requirements.