High level documentation

Full System Design Document: Demand Forecasting & Monitoring System

1. System Overview

This document describes a **Demand Forecasting & Monitoring System** consisting of:

- **Backend**: FastAPI-based forecasting service with drift detection
- **Frontend**: Streamlit-based visualization dashboard
- **Monitoring**: Prometheus metrics + MLflow tracking

The system provides:
✔
Real-time demand forecasting

✔ **Automated drift detection**

✔ **Interactive visualization**

✔ **Model performance tracking**

---

2. High-Level Architecture

```
graph TD
    A[Frontend - Streamlit] -->|HTTP| B[Backend - FastAPI]
    B --> C[(Model Storage)]
    B --> D[MLflow Tracking]
    B --> E[Prometheus Metrics]
    F[User] --> A
```

2.1 Component Responsibilities

Component	Responsibilities
Streamlit Frontend	Visualization, simulation control, model initialization
FastAPI Backend	Forecasting, drift detection, model management
MLflow	Model versioning, experiment tracking
Prometheus	System/API metrics collection

---

3. Backend (FastAPI) Design

3.1 Core Components

1. Model Management

- `ForecastingModel` **Class**: Wrapper for Prophet model
    - Handles serialization/deserialization
    - Standardizes prediction interface
- **Model Storage**: Local files (`model.pkl`, `metadata.json`)

2. Drift Detection

- `DriftDetector` **Class**:
    - Window-based error monitoring
    - Configurable threshold triggering
    - Stateful error tracking

3. API Endpoints

Endpoint	Method	Description
`/model_init`	POST	Initialize model with training data
`/last_trained_date`	GET	Get model metadata
`/predict_for_date`	GET	Get prediction + trigger drift check

4. Observability

- **Prometheus Metrics**:

    - Endpoint latency/counters
    - System resource usage
- **MLflow Integration**:
    - Model versioning
    - Drift event logging

3.2 Key Design Decisions

Decision	Rationale	Trade-offs
Global state management	Simple implementation	Not horizontally scalable
Error-based drift detection	Direct business impact	Less sophisticated than statistical tests
Immediate retraining	Minimizes prediction degradation	Computationally expensive

3.3 Data Flow

```
sequenceDiagram
    Frontend->>Backend: POST /model_init (CSV + params)
    Backend->>Backend: Train initial model
    Backend->>MLflow: Log training run
    Frontend->>Backend: GET /predict_for_date?date=X
    Backend->>DriftDetector: Check for drift
    alt Drift detected
        Backend->>Backend: Retrain model
        Backend->>MLflow: Log drift event
    end
    Backend->>Frontend: Return prediction
```

4. Frontend (Streamlit) Design

4.1 Core Features

1. **Model Initialization**
    - CSV upload
    - Parameter configuration (window size, threshold)
2. **Simulation Control**
    - Start/Pause/Reset
    - Scrolling time window
3. **Visualization**
    - Predicted vs actual values
    - Drift event markers
4. **Monitoring**
    - Drift event counter
    - Current date tracking

4.2 Key Design Patterns

Pattern	Implementation	Purpose
Session State	`st.session_state`	Maintain simulation state
Scrolling Window	Plotly range slider	Handle long time series
Reactive Updates	`st.rerun()`	Live simulation updates

4.3 UI Components Layout

```
graph LR
    A[Sidebar] --> B[Model Configuration]
    C[Main Area] --> D[Forecast Visualization]
    C --> E[Simulation Controls]
    C --> F[Raw Data Toggle]
```

---

5. Integration Design

5.1 API Contracts

`/model_init` (POST)

```
Request:
  - multipart/form-data:
    - file: CSV
    - window_size: int
    - threshold: float

Response:
  - {"message": "Model initialized successfully"}
```

`/predict_for_date` (GET)

```
Request:
  - Query params:
    - current_date: "YYYY-MM-DD"

Response:
  - {
      "date": "YYYY-MM-DD",
      "predicted": float,
      "actual": float,
      "error": float,
      "drift_detected": bool
    }
```

5.2 Error Handling

Scenario	Frontend Behavior	Backend Response
Model not initialized	Show warning	500 + error message
Invalid date format	Input validation	400 Bad Request
Drift detection	Highlight in red	Include in response

---

6. Operational View

6.1 Deployment Architecture

```
graph TD
    A[Streamlit] --> B[FastAPI:8000]
    B --> C[Prometheus:8001]
    B --> D[MLflow:5001]
```

6.2 Monitoring Dashboard

Prometheus Metrics to Alert On:

1. `predict_for_date_request_latency_seconds > 1s`
2. `system_memory_usage_percent > 90%`
3. `api_requests_per_ip_total` (abnormal spikes)

6.3 Scaling Considerations

Aspect	Current Limitation	Mitigation Strategy
State management	In-memory globals	Redis-backed storage
Horizontal scaling	Not supported	Stateless redesign
Large datasets	Memory-bound	Chunked processing

---

7. Future Enhancements

7.1 Short-Term (Next Release)

- [ ] Model version comparison
- [ ] Prediction confidence intervals
- [ ] Alerting integration (Slack/Email)

7.2 Long-Term

- [ ] Batch prediction API

- [ ] Multiple model support
- [ ] Kubernetes deployment

---

8. Conclusion

This design provides:
✅
End-to-end forecasting pipeline

✅ **Proactive model monitoring**

✅ **Interactive analysis capabilities**

The trade-offs prioritize:

- **Rapid iteration** (Streamlit + FastAPI)
- **Operational simplicity** (single-node deployment)
- **Immediate observability** (built-in metrics)

Recommended next steps:

1. Implement Prometheus alerting rules
2. Add model version rollback capability
3. Stress test with large datasets

Design Choices and Rationales for Demand Forecasting System

1. Backend Architecture (FastAPI)

1.1 Framework Selection: FastAPI

Choice:

Built the backend using FastAPI instead of Flask/Django.

Rationale:

- **Performance**: Native async support (ASGI) handles concurrent prediction requests efficiently
- **Type Safety**: Automatic request validation via Pydantic models reduces bugs
- **API Documentation**: Built-in OpenAPI/Swagger UI simplifies integration
- **Modern Features**: Dependency injection, background tasks ideal for ML services

Trade-offs:

- Less mature ecosystem than Django for admin UIs
- Requires explicit middleware for advanced features

1.2 State Management

Choice:

Used global variables (`model`, `df_global`) instead of database/Redis.

Rationale:

- **Simplicity**: No external dependencies for single-node deployment
- **Latency**: In-memory access faster than network calls
- **Prototyping**: Faster iteration during development

Trade-offs:

- Not horizontally scalable
- State lost on server restart

Mitigation:

Added model persistence to disk (`model.pkl`) for recovery.

1.3 Drift Detection Implementation

Choice:

Window-based error thresholding instead of statistical tests (KS, Chi-square).

Rationale:

- **Interpretability**: Business teams understand "prediction error > X"
- **Computational Efficiency**: O(1) complexity per update vs O(n) for statistical tests
- **Direct Impact**: Tied directly to model performance degradation

Trade-offs:

- Less sensitive to distribution changes not affecting error
- Requires manual threshold tuning

1.4 Observability Stack

Choices:

1. **Prometheus for Metrics**
    - Endpoint latency
    - System resources
    - Custom business metrics (drift events)
2. **MLflow for Model Tracking**
    - Model versions
    - Training parameters
    - Drift metadata

Rationale:

- **Separation of Concerns**: Metrics vs model tracking
- **Standard Tools**: Widely supported in ML ecosystems
- **Temporal Analysis**: Prometheus excels at time-series data

Alternative Considered:

ELK stack was rejected due to higher operational overhead.

---

2. Frontend Architecture (Streamlit)

2.1 Framework Selection

Choice:

Streamlit over Dash/Plotly Flask.

Rationale:

- **Rapid Prototyping**: Build UI with pure Python
- **Built-in Components**: Sliders, file uploaders, metrics out-of-the-box
- **State Management**: `st.session_state` sufficient for this use case
- **Visualization**: Tight Plotly integration

Trade-offs:

- Less customizable than Dash
- Not ideal for complex SPAs

2.2 Visualization Approach

Choice:

Interactive Plotly chart with:

- Scrolling time window
- Drift event markers
- Unified hover tooltips

Rationale:

- **User Experience**: Analysts can inspect specific dates
- **Performance**: Partial rendering avoids overloading browser
- **Interpretability**: Color-coded actual vs predicted

Key Implementation:

```
fig.update_layout(
    xaxis=dict(
        range=[min_date, max_date],  # Dynamic window
        rangeslider=dict(visible=True)  # Scrollbar
```

```
    )
)
```

2.3 Simulation Control

Choice:

Imperative loop with `st.rerun()` instead of async callbacks.

Rationale:

- **Simplicity**: Avoids complex event handling
- **Predictable State**: Full refresh prevents stale UI
- **Progress Visibility**: Clear "running" state

Trade-off:

Brief UI flicker during updates.

3. Integration Design

3.1 API Contracts

Choice:

Simple JSON endpoints with minimal nesting.

Example:

```
@app.get("/predict_for_date")
async def predict(current_date: str):
    return {
        "date": "2023-01-01",
        "predicted": 42.1,
        "actual": 40.2,
        "drift_detected": False
    }
```

Rationale:

- **Frontend Simplicity**: Easy to consume in Streamlit
- **Debugging**: Human-readable responses
- **Extensibility**: Add fields without breaking changes

3.2 Error Handling

Choice:

HTTP status codes + JSON error details.

Implementation:

```
try:
    predict()
except Exception as e:
    return JSONResponse(
        status_code=500,
        content={"error": str(e), "type": type(e).__name__}
    )
```

Rationale:

- **Standard Practice**: Follows REST conventions
- **Frontend Handling**: Streamlit can show toast notifications
- **Monitoring**: Prometheus alerts on 5xx codes

---

4. Operational Choices

4.1 Deployment Architecture

Choice:

Single-node design with co-located services.

Rationale:

- **Development Speed**: No Kubernetes overhead
- **Resource Efficiency**: Shared memory for model/data

- **Debugging**: All logs in one place

Future-Proofing:

Code structured to allow:

- Containerization (Docker)
- Separate metric server

4.2 Monitoring Strategy

Choices:

1. **Process-Level Metrics** (CPU/Memory)
2. **Endpoint-Specific Latency**
3. **Business Metrics** (Drift Events)

Rationale:

- **Triage Efficiency**: Distinguish system vs model issues
- **Capacity Planning**: Identify resource bottlenecks
- **SLA Compliance**: Track prediction latency

---

5. Key Trade-offs Summary

Area	Choice	Rationale	Trade-off
State	In-memory globals	Simplicity	No horizontal scaling
Drift Detection	Error thresholding	Interpretability	Less statistically rigorous
Frontend	Streamlit	Rapid development	Less customizable
Metrics	Prometheus + MLflow	Standard tools	Operational overhead

6. Evolution Path

1. **Immediate**: Add model version rollback
2. **Mid-term**: Redis for shared state
3. **Long-term**: Kubernetes deployment

This architecture prioritizes **developer velocity** and **operational simplicity** while providing clear upgrade paths for production scaling.