Forecasting Dev Handbook

Engineering handoff and operating guide

Owner: Ganesh

1. Introduction

1.1 What is the Forecasting System?

The **Forecasting System** is a machine-learning–driven platform designed to predict future retail sales performance in terms of **revenue (monetary value)** and **demand (units sold)**. It enables retailers and internal analytics teams to make informed decisions related to inventory planning, promotions, staffing, and financial forecasting.

At a high level, the system consists of **two clearly separated but tightly aligned components**:

1. Training Pipeline

The training pipeline is an **offline, batch-oriented system** responsible for building and versioning forecasting models. It:

- Extracts historical sales data from multiple client databases
- Enriches raw sales data with **weather**, **holiday**, and **store-location** features
- Performs large-scale feature engineering using Apache Spark
- Trains gradient-boosting models (LightGBM) using Ray for distributed execution
- Outputs versioned models and feature schemas for downstream inference

This pipeline is compute-intensive, runs on demand or on a schedule, and is designed for scalability and reproducibility.

2. Inference API

The inference component is a **real-time, stateless API service** built with FastAPI. It:

- Loads the latest trained models at startup
- Accepts forecast requests over REST APIs
- Performs real-time feature engineering that mirrors training logic
- Generates hourly or daily forecasts for future dates
- Supports scenario-based forecasting (promotions, holidays, weather overrides)

The inference API is optimized for **low-latency responses**, horizontal scalability, and safe production usage (read-only database access).

Together, these components allow the system to support both **accurate long-term model training** and **fast, reliable real-time forecasting**.

1.2 Purpose of This Handbook

This handbook serves as the **authoritative technical reference** for the Sales Forecasting System. Its purpose is to:

- Explain the **architecture and design decisions** behind the system
- Document the **training pipeline**, including data flow and model lifecycle
- Describe all **API endpoints**, request/response formats, and validation rules
- Provide guidance on **configuration, deployment, and environment setup**
- Enable effective **troubleshooting, testing, and maintenance**

The handbook is written to support both **day-to-day development work** and **long-term system ownership**, ensuring knowledge continuity across teams.

1.3 Intended Audience

This handbook is intended for a broad technical audience involved in building, operating, or integrating with the Sales Forecasting System:

- **Backend Developers**

Working on API endpoints, business logic, feature engineering, or performance optimizations.

- **Data Engineers / ML Engineers**

Responsible for running and modifying the training pipeline, managing data enrichment, and improving model quality.

- **DevOps / Platform Engineers**

Deploying the system, managing environments, configuring cloud services, monitoring health, and scaling infrastructure.

- **System Administrators**

Maintaining databases, credentials, access control, and production stability.

- **QA and Validation Engineers**

Testing API behavior, validating forecast correctness, and ensuring regression safety.

While the document is technical, it assumes **basic familiarity with Python, REST APIs, and relational databases**, and it gradually introduces system-specific concepts.

1.4 How to Use This Handbook

The handbook is organized into **three progressive parts**, each serving a specific purpose:

Part 1: Foundations and Setup

- System overview and design philosophy
- Technology stack and dependencies
- Repository structure and code navigation
- Local development and environment setup

➡️ *Recommended starting point for all new contributors.*

Part 2: Architecture and API Details

- End-to-end system architecture
- Training pipeline internals
- Inference request flow and feature engineering
- Complete API endpoint reference and sample payloads

➡️ *Primary reference for development and integration work.*

Part 3: Operations and Maintenance

- Database schemas and data stores
- Configuration reference and environment variables
- Error handling and testing strategies
- Troubleshooting guides and operational checklists

➡️ *Primary reference for production support and system ownership.*

Readers are encouraged to:

- Read **Part 1 sequentially** for onboarding

- Use **Parts 2 and 3 as reference material** when implementing features or diagnosing issues
- Jump directly to specific sections as needed during development or operations

This structure ensures the handbook remains both **approachable for newcomers** and **efficient for experienced engineers**.

2. TL;DR – What to Know in 5 Minutes

This section provides a **high-level, practical overview** of the Sales Forecasting System for engineers who need to understand the system quickly before working with it.

2.1 System at a Glance

What this system does:

- Predicts **future retail sales** for products
- Outputs forecasts as:
    - **Revenue** (dollars)
    - **Demand** (units sold)

How it's delivered:

- A **real-time REST API** for forecasting
- An **offline training pipeline** for building models

Key design idea:

Training and inference are completely decoupled but share identical feature logic.

2.2 Two Repositories You Must Know

The system is split into **two independent repositories**, each with a single, well-defined responsibility.

1. `AI_Inventory_Optimization_Demand_Forecasting_Training` – Training Pipeline

Purpose:

- Build and version forecasting models

Responsibilities:

- Extract historical sales data from client databases
- Enrich data with:
  - Weather
  - Holidays
  - Store geolocation
- Perform large-scale feature engineering (Spark)
- Train models using LightGBM (distributed with Ray)
- Save versioned models and feature schemas

Output artifacts:

- `revenue_model_*.pkl`
- `units_model_*.pkl`
- Feature schema JSON files
- Model registry metadata

Runs:

- Offline

- On-demand or scheduled
- Can take hours depending on data size

2. `AI_Inventory_Optimization_Demand_Forecasting` – Inference API

Purpose:

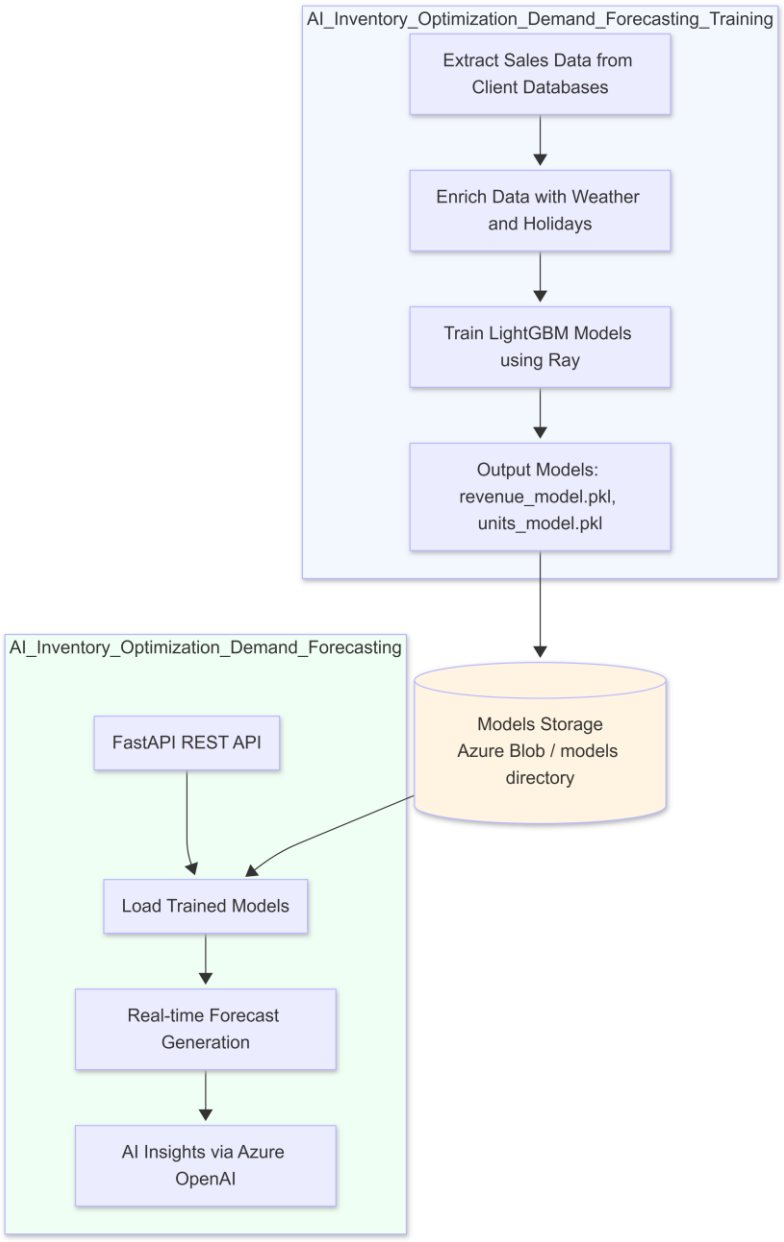
- Serve forecasts in real time

Responsibilities:

- Load trained models at startup
- Accept forecast requests over HTTP (JSON)
- Recreate training-time features at inference time
- Perform recursive multi-day forecasting
- Return structured forecast responses

Key properties:

- Stateless
- Horizontally scalable
- Read-only database access
- Safe for production workloads



2.3 The Only Endpoints You Really Need to Remember

Forecast Endpoints (Core)

- **POST `/forecast_revenue`**

Returns forecasted **sales revenue** in dollars

- **POST `/forecast_demand`**

Returns forecasted **sales demand** in units

Both endpoints:

- Share the same request structure
- Support date ranges, time ranges, promotions, and weather overrides
- Return hourly forecasts by default

Utility Endpoints

- **GET `/health`**

Confirms API health and model load status

- **GET `/docs`**

Interactive Swagger UI for testing APIs

2.4 Optional AI-Powered Endpoint

These endpoints enhance usability but are **not required** for core forecasting.

- **POST `/api/ai/insights`**

Generates human-readable explanations of forecast results

- **POST `/api/ai/chart_callouts`**

Produces annotations for significant forecast spikes or dips

- **POST `/api/ai/parse_promo_event`**

Converts unstructured promotional text into structured data

If Azure OpenAI is not configured, these endpoints degrade gracefully.

2.5 Typical Developer Workflow

1. **Train models** (data/ML engineers)

```
python run_full_pipeline.py
```

2. **Start the inference API** (backend/devops)

```
uvicorn app:app --host 0.0.0.0 --port 8000 --reload
```

3. **Call a forecast endpoint**

```
curl -X POST http://localhost:8000/forecast_revenue \
  -H "Content-Type: application/json" \
  -d
'{"client_code":"571","store_id":6,"product_id":817522014428
,"next_days":7}'
```

4. **Receive forecasts**
    a. Hourly predictions
    b. Aggregated totals and averages
    c. Ready for dashboards or downstream systems

2.6 Key Things to Remember

- Training and inference are **separate systems**
- Models are **versioned artifacts**, not code changes
- Feature engineering must stay **identical** between training and inference
- The API is **stateless and safe to scale**

- Missing data is handled gracefully using defaults

If you understand this section, you already understand **80% of how the system works**.

3. Design Principles

This section outlines the **core architectural and engineering principles** that guide the design of the Sales Forecasting System. These principles are critical to preserving system correctness, scalability, and long-term maintainability.

3.1 Strict Separation of Training and Inference

Model training and model inference are treated as **independent lifecycles**:

- Training is an **offline, batch-oriented process**
- Inference is an **online, real-time service**
- Models are the *only* contract between the two

This separation ensures:

- Predictable API performance
- No accidental retraining in production
- Independent scaling of compute-heavy training workloads

3.2 Stateless Inference API

The inference service is intentionally **stateless**:

- No user sessions
- No request-to-request dependency

- No database writes during inference

Each request is fully self-contained, enabling:

- Horizontal scaling
- Safe restarts and rolling deployments
- Simple load-balancing strategies

3.3 Read-Only Data Access During Inference

During inference, the system **only reads data** from client databases:

- Historical sales (for lag features)
- Profile averages (for baseline behavior)

The API never modifies client data, which:

- Eliminates data corruption risk
- Simplifies security and compliance reviews
- Allows safe access to production databases

3.4 Recursive Forecasting for Multi-Day Predictions

Multi-day forecasts are generated using a **recursive strategy**:

- Predictions for earlier time steps are reused as inputs for later ones
- Lag and rolling features for future dates come from prior predictions

This approach:

- Preserves feature realism across forecast horizons
- Avoids unrealistic flat or repeated forecasts

- Aligns inference behavior with training assumptions

3.5 Feature Engineering Consistency

Feature engineering logic is treated as **part of the model**, not an implementation detail:

- Feature names, types, and encodings must match training exactly
- Feature schemas are versioned alongside models
- Validation occurs before every prediction

Any mismatch between training and inference features is considered a **critical error**.

3.6 Graceful Degradation and Safe Defaults

The system is designed to continue operating even with incomplete data:

- Missing weather data falls back to historical averages
- Unknown products or stores use profile-level defaults
- Optional AI endpoints return empty results if unavailable

This ensures:

- Forecasts are always returned when possible
- Partial outages do not cascade into full system failures
- Downstream systems can rely on predictable behavior

3.7 Production-First Design Mindset

All design decisions prioritize production reliability:

- Explicit validation and clear error messages
- Defensive coding around external dependencies
- Deterministic model loading and versioning
- Observability through health checks and logs

The result is a system that is **easy to operate, safe to extend, and difficult to misuse**.

4. Technologies Used

This section describes the **technology choices** behind the Sales Forecasting System and, more importantly, **why each technology exists** in the stack. The goal is not to catalog dependencies, but to explain architectural intent.

4.1 Stack Overview

The system is built entirely in Python and is optimized for:

- Large-scale data processing (training)
- Low-latency, high-throughput inference (serving)
- Operational simplicity in production

The stack is intentionally split between **training-time technologies** and **inference-time technologies**, with a small set of shared libraries.

4.2 Training Pipeline Technologies

Apache Spark

Role: Distributed ETL and feature engineering

Spark is used to:

- Extract large volumes of historical sales data
- Join enrichment data (holidays, weather, stores)
- Perform windowed aggregations and feature generation

It was chosen because:

- It scales well across clients and years of data
- It provides deterministic, reproducible transformations
- It integrates cleanly with SQL-based data sources

Ray

Role: Distributed model training and parallel execution

Ray is responsible for:

- Parallelizing LightGBM training workloads
- Managing CPU resources efficiently
- Supporting hyperparameter tuning when enabled

Ray allows training to scale independently of Spark and avoids monolithic single-node model training.

LightGBM

Role: Core forecasting model

LightGBM is used because it:

- Handles mixed numerical and categorical features well
- Trains quickly on large tabular datasets
- Produces compact models with fast inference

Two independent models are trained:

- Revenue forecasting model
- Demand (units) forecasting model

Geopy, Meteostat, and Holidays

Role: Data enrichment

These libraries provide:

- Store geocoding (latitude/longitude)
- Historical and forecast weather data
- Holiday calendars by country and state

They are used during training and inference to ensure **feature parity** between both stages.

4.3 Inference API Technologies

FastAPI

Role: API framework

FastAPI is used to:

- Define REST endpoints with strong typing
- Enforce request and response validation
- Auto-generate interactive API documentation

Its async-friendly design enables high throughput with minimal boilerplate.

Pydantic

Role: Data validation and schema enforcement

Pydantic ensures:

- Invalid requests fail fast
- Clear, actionable validation errors
- Strong guarantees on request shape and types

This significantly reduces defensive code inside business logic.

Uvicorn

Role: ASGI server

Uvicorn runs the FastAPI application in both development and production, supporting:

- Async execution
- Multiple worker processes
- Clean startup and shutdown behavior

LightGBM (Inference)

The same LightGBM models trained offline are loaded into memory at API startup. This guarantees:

- No model state mutation during inference
- Deterministic predictions
- Fast per-request execution

4.4 Shared and Infrastructure Technologies

Pandas and NumPy

Used for:

- Final-stage feature assembly
- Lightweight transformations during inference
- Model input formatting

SQLAlchemy and ODBC

Used for:

- Read-only database access
- Profile and historical data lookups
- Safe, parameterized queries

Azure Services (Optional)

The system integrates optionally with:

- Azure Blob Storage for model artifacts
- Azure Key Vault for secret management
- Azure OpenAI for AI-powered insights

All Azure integrations are **non-blocking** and degrade gracefully when unavailable.

5. Repository and File Structure

This section explains **how the codebase is organized** and where engineers should make changes. Understanding this layout is essential before modifying behavior.

5.1 High-Level Layout

The system is split into two repositories:

- `AI_Inventory_Optimization_Demand_Forecasting_Training` – Model training and data preparation
- `AI_Inventory_Optimization_Demand_Forecasting` – Real-time inference and API serving

Each repository is independently deployable and versioned.

5.2 Training Repository (AI_Inventory_Optimization_Demand_Forecasting_Training)

Structure and Intent

The training repository is organized as a **linear pipeline**, where each step has a single responsibility and can be run independently.

```
AI_Inventory_Optimization_Demand_Forecasting_Training /
├── step_0_1_extract_training_data.py
├── step_1_1_geocode_stores.py
├── step_1_2_populate_holidays.py
├── step_1_3_backfill_weather.py
├── step_2_1_enrich_training_data.py
├── step_2_2_model_training.py
├── run_full_pipeline.py
├── model_versioning.py
├── kv_helper.py
├── requirements.txt
├── env_template.txt
├── README.md
└── SQL Scripts/
    ├── CREATE_TBL_ETL_TRAININGDATA.sql
    ├── CREATE_TBL_STORECOORDINATES.sql
    ├── CREATE_TBL_CALENDARHOLIDAYS.sql
    ├── CREATE_TBL_WEATHERHISTORY.sql
    └── CREATE_VIEW_VW_FORECASTING.sql
```

Core Pipeline Scripts

- `step_0_1_extract_training_data.py`

Extracts raw sales data from client databases

- `step_1_1_geocode_stores.py`

Resolves store addresses to coordinates

- `step_1_2_populate_holidays.py`

Builds holiday calendars for all locations

- `step_1_3_backfill_weather.py`

Fetches historical weather data

- `step_2_1_enrich_training_data.py`

Joins enrichment data into training records

- `step_2_2_model_training.py`

Trains and versions forecasting models

Orchestration and Utilities

- `run_full_pipeline.py`

Executes the full pipeline end-to-end

- `model_versioning.py`

Handles model naming, versioning, and storage

- `kv_helper.py`

Loads secrets from Key Vault or environment variables

5.3 Inference Repository (AI_Inventory_Optimization_Demand_Forecasting)

Structure and Intent

The inference repository is structured around a **single application entry point**, with supporting modules grouped by responsibility.

```
AI_Inventory_Optimization_Demand_Forecasting/
├── app.py
├── model_versioning.py
├── kv_helper.py
├── ai_helper.py
├── ai_insights.py
├── ai_chart_callouts.py
├── ai_lift_composer.py
├── ai_promo_parser.py
├── models/
│   ├── revenue_model_latest.pkl
│   ├── units_model_latest.pkl
│   ├── feature_schema_revenue.json
│   ├── feature_schema_units.json
│   └── model_registry.json
├── requirements.txt
```

```
├── env_template.txt
└── README.md
```

Core Application

- `app.py`

Contains:

  - API endpoints
  - Feature engineering logic
  - Database access helpers
  - Model prediction flow

This file is intentionally centralized to make request flow easy to trace.

Supporting Modules

- `model_versioning.py` – Model loading and registry handling
- `kv_helper.py` – Credential management
- `ai_helper.py` – Azure OpenAI client initialization

AI-related behavior is isolated into:

- `ai_insights.py`
- `ai_chart_callouts.py`
- `ai_lift_composer.py`
- `ai_promo_parser.py`

Models and Schemas

The `models/` directory contains:

- Versioned model artifacts
- Latest-model pointers
- Feature schema definitions
- Model registry metadata

These files are treated as **deployable assets**, not source code.

5.4 Where to Make Changes

To avoid unintended side effects:

- **API behavior:** modify `app.py`
- **Feature engineering:** modify feature creation logic in `app.py`
- **Training logic:** modify scripts in `AI_Inventory_Optimization_Demand_Forecasting_Training`
- **Model lifecycle:** modify `model_versioning.py`
- **AI behavior:** modify `ai_*.py` modules

Understanding this separation is critical before introducing new features.

6. Local Development Setup

This section explains how to set up the Sales Forecasting System **locally for development and testing**. Training and inference have different requirements and should be set up independently.

6.1 Prerequisites

6.1.1 Common Requirements

Required for both repositories:

- **Python 3.12**
- **SQL Server ODBC Driver**
    - Windows: ODBC Driver 17 for SQL Server
    - Linux/macOS: Microsoft ODBC Driver for SQL Server

Optional but recommended:

- Git
- A Unix-like shell (Git Bash, WSL, or macOS/Linux terminal)

6.1.2 Training Pipeline Requirements

Additional requirements for `AI_Inventory_Optimization_Demand_Forecasting_Training`:

- **Java JDK 7 or higher** (required for Apache Spark)
- **Apache Spark (via PySpark)**
- **Ray** for distributed training

Optional:

- Microsoft SQL Server JDBC driver (for Spark JDBC performance)

6.1.3 Inference API Requirements

Additional requirements for `AI_Inventory_Optimization_Demand_Forecasting`:

- Trained model artifacts (`.pkl` files)
- Feature schema JSON files

Optional:

- Azure OpenAI credentials (AI endpoints)
- Azure Blob Storage access (remote model storage)
- Azure Key Vault access (secret management)

6.2 Environment Setup

The training pipeline and inference API should be installed in **separate virtual environments**.

6.2.1 Training Pipeline Setup

```
cd AI_Inventory_Optimization_Demand_Forecasting_Training

python -m venv venv
source venv/bin/activate   # Windows: venv\Scripts\activate

pip install -r requirements.txt
```

Verify dependencies:

```
java -version
python -c "import pyspark; print(pyspark.__version__)"
python -c "import ray; print(ray.__version__)"
```

6.2.2 Inference API Setup

```
cd AI_Inventory_Optimization_Demand_Forecasting

python -m venv venv
```

```
source venv/bin/activate   # Windows: venv\Scripts\activate

pip install -r requirements.txt
```

Verify dependencies:

```
python -c "import fastapi; print(fastapi.__version__)"
python -c "import lightgbm; print(lightgbm.__version__)"
```

6.3 Environment Variables

Each repository includes an `env_template.txt` file.

Setup Steps

1. Copy the template:

```
cp env_template.txt .env
```

2. Populate required variables
3. Export or load `.env` using your preferred method

Required Variables (Both Repositories)

- `SQL_SERVER_MAIN`
- `SQL_DATABASE_MAIN`
- `SQL_USERNAME_MAIN`
- `SQL_PASSWORD_MAIN`
- `SQL_ODBC_DRIVER`

Training-Specific Variables

- `ETL_CLIENT_CODES`
- `ETL_DATA_YEARS`
- `TRAINING_DATA_YEARS`
- `RAY_NUM_WORKERS`

Inference-Specific Variables

- `MAX_FORECAST_DAYS`

Optional:

- Azure OpenAI variables
- Azure Blob Storage variables
- Azure Key Vault variables

6.4 Database Preparation

Training Repository

Before running the pipeline:

1. Connect to the **main ETL database**
2. Execute SQL scripts in order:
    a. `CREATE_TBL_ETL_TRAININGDATA.sql`
    b. `CREATE_TBL_STORECOORDINATES.sql`
    c. `CREATE_TBL_CALENDARHOLIDAYS.sql`
    d. `CREATE_TBL_WEATHERHISTORY.sql`
3. For each client database:

a. Run `CREATE_VIEW_VW_FORECASTING.sql`

Inference Repository

- No schema changes are required
- Uses client databases in **read-only mode**

6.5 Running Locally

6.5.1 Run Training Pipeline

Options:

- Full pipeline: `python run_full_pipeline.py`
- Individual steps:

```
python step_0_1_extract_training_data.py
python step_1_1_geocode_stores.py
python step_1_2_populate_holidays.py
python step_1_3_backfill_weather.py
python step_2_1_enrich_training_data.py
python step_2_2_model_training.py
```

6.5.2 Run Inference API

Ensure models exist in `models/` directory, then:

```
uvicorn app:app --host 0.0.0.0 --port 8000 --reload
```

Verify:

- Health check: `GET /health`
- API docs: http://localhost:8000/docs

6.6 Local Development Notes

- Training can be resource-intensive; start with small datasets
- Inference API should start in under a few seconds
- Always retrain models after changing feature logic
- Never test inference using partially trained models

With this setup complete, developers can confidently move from **local experimentation** to **production deployment**.

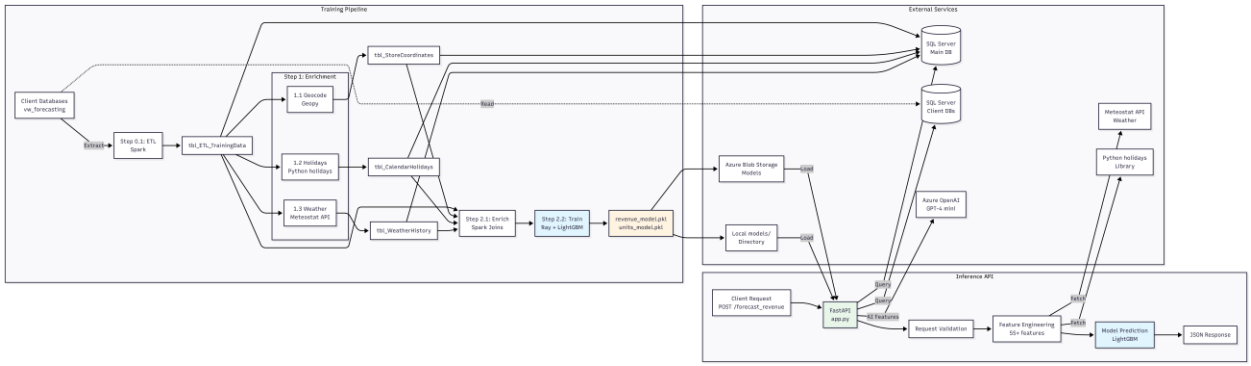
7. System Architecture

This section describes the **end-to-end architecture** of the Sales Forecasting System. It explains how data flows through the system, how major components interact, and how responsibilities are separated to support scalability, reliability, and maintainability.

The architecture is intentionally designed around **clear boundaries**:

- Offline vs online workloads
- Compute-heavy vs latency-sensitive paths
- Model lifecycle vs request lifecycle

7.1 High-Level Architectural Overview



At the highest level, the system consists of three major layers:

1. **Client Layer** – External systems consuming forecasts
2. **Inference Layer** – Real-time forecasting API
3. **Training Layer** – Offline model generation pipeline

These layers communicate only through **well-defined contracts**, primarily REST APIs and versioned model artifacts.

7.2 End-to-End Flow (Conceptual)

Inference Path (Online):

Client Application

→ HTTP REST Request (JSON)

→ Inference API (FastAPI)

→ Feature Engineering

→ Model Prediction (LightGBM)

→ JSON Forecast Response

Training Path (Offline):

Client Databases

→ ETL & Enrichment (Spark)

→ Feature Engineering

→ Model Training (Ray + LightGBM)

→ Versioned Model Artifacts

→ Deployment to Inference Environment

The two paths are intentionally decoupled and interact only through **model artifacts and feature schemas**.

7.3 Inference Architecture

The inference system is designed as a **stateless, horizontally scalable service**.

Core Components

API Layer

- Handles HTTP routing and request parsing
- Validates inputs using strict schemas
- Returns structured JSON responses

Business Logic Layer

- Resolves filters (store, product, category)
- Calculates forecast date and time ranges

- Coordinates feature generation and prediction loops

Feature Engineering Layer

- Reconstructs training-time features in real time
- Pulls historical data and profiles as needed
- Applies promotions, holidays, and weather logic

Model Layer

- Loads models once at startup
- Validates feature schema alignment
- Executes predictions deterministically

Data Access Layer

- Performs read-only queries against client databases
- Caches frequently used profile and lookup data
- Integrates with external data sources when required

7.4 Forecast Execution Flow (Inference)

For each forecast request, the API executes the following sequence:

1. **Request Validation**
    a. Ensure required fields are present
    b. Validate date and time ranges
    c. Enforce safety limits (e.g., max forecast days)
2. **Entity Resolution**
    a. Resolve product identifiers if names are provided
    b. Normalize filters into a canonical internal form
3. **Range Expansion**
    a. Expand date ranges into individual days

    b. Expand time ranges into hourly slots
4. **Profile Initialization**
    a. Fetch historical averages for the requested scope
    b. Cache profiles for reuse within the request
5. **Recursive Forecast Loop**

For each (date, hour):

    a. Build feature vector
    b. Execute model prediction
    c. Store prediction for future lag features
6. **Response Assembly**
    a. Aggregate forecasts
    b. Compute totals and averages
    c. Format response payload

7.5 Recursive Forecasting Strategy

Multi-day forecasts rely on **recursive prediction**:

- Early predictions are reused as inputs for later time steps
- Lag and rolling features for future timestamps are derived from prior predictions

This strategy ensures:

- Temporal consistency across the forecast horizon
- Alignment with how lag features were learned during training
- Realistic forecast trajectories rather than static repetitions

7.6 Training Architecture

The training pipeline is a **batch-oriented, multi-stage workflow** optimized for throughput and reproducibility.

Major Stages

Data Extraction

- Pulls raw sales data from multiple client databases
- Applies filtering by date, client, and business rules

Data Enrichment

- Augments sales data with:
    - Store geolocation
    - Holiday calendars
    - Historical weather data

Feature Engineering

- Applies window functions and aggregations
- Produces the same feature set expected at inference time

Model Training

- Trains independent models for revenue and demand
- Distributes training across CPUs using Ray
- Optionally performs hyperparameter tuning

Artifact Generation

- Produces versioned model files
- Writes feature schema definitions
- Updates model registry metadata

7.7 Separation of Concerns

The architecture enforces separation at multiple levels:

- **Training vs Inference** – Different runtimes, scaling models, and failure modes
- **Feature Logic vs Model Logic** – Features are validated independently of predictions
- **Data Access vs Business Logic** – Database queries are isolated from forecast rules

This separation:

- Simplifies reasoning about system behavior
- Reduces blast radius of changes
- Enables independent evolution of components

7.8 External Dependencies and Boundaries

External systems are treated as **optional, non-blocking dependencies**:

- Weather services
- Holiday libraries
- Cloud storage
- AI/LLM services

Failures or latency in these systems:

- Do not prevent core forecasting
- Trigger fallbacks or defaults
- Are isolated from the main prediction loop

7.9 Architectural Guarantees

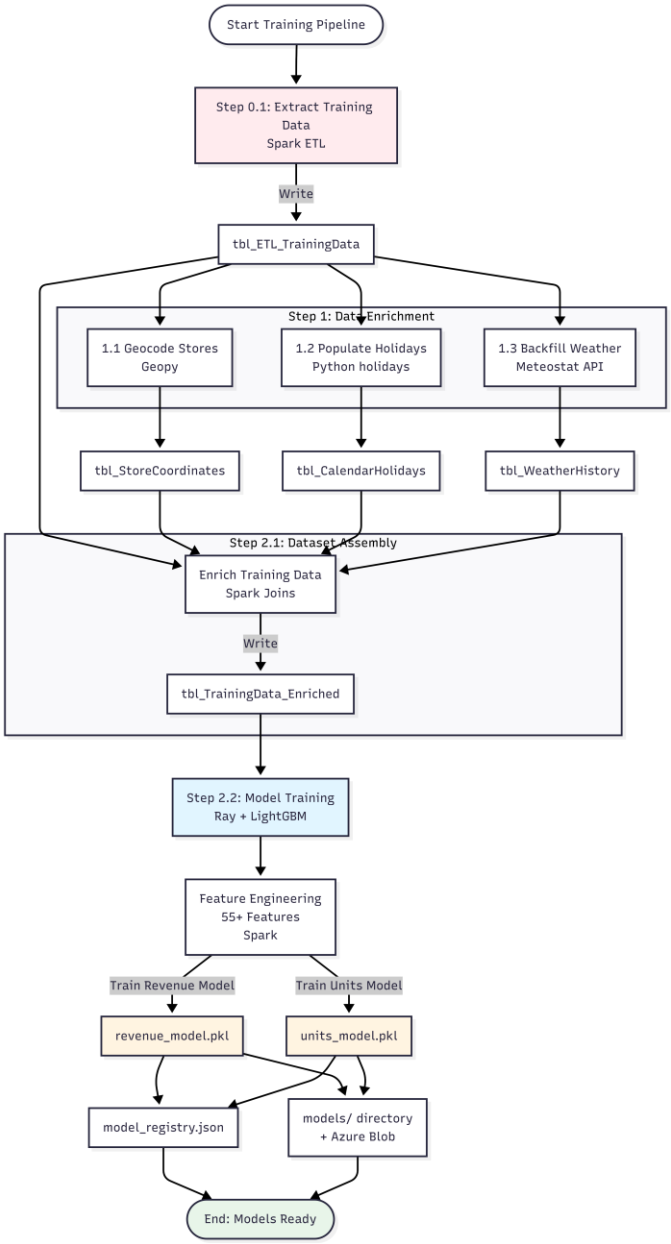
The architecture is designed to guarantee:

- Deterministic forecasts for identical inputs
- No side effects during inference
- Predictable performance under load
- Safe horizontal scaling
- Clear ownership of responsibilities

These guarantees are foundational and should be preserved as the system evolves.

8. Training Pipeline

This section provides a **detailed architectural and operational view** of the training pipeline. It explains how raw sales data is transformed into production-ready forecasting models, and why each step exists.



8.1 Pipeline Overview

The training pipeline is a **multi-stage, sequential workflow** executed offline. Each stage produces durable outputs that are either persisted to the database or written as model artifacts.

High-level stages:

1. Data extraction (ETL)

2. Store geocoding
3. Holiday calendar generation
4. Weather backfilling
5. Training data enrichment
6. Model training and versioning

Each stage can be run independently, but is typically executed end-to-end via an orchestrator.

8.2 Step 0.1 – Extract Training Data (ETL)

Purpose

To create a **canonical, centralized training dataset** from multiple client databases.

Inputs

- Client databases (read-only)
- Forecasting views exposed by each client
- Environment-driven filters (client codes, date ranges)

Processing

- Connects to each client database
- Queries forecasting views using Spark JDBC
- Filters data by:
    - Client code
    - Business type (if configured)
    - Date range

Outputs

- Writes normalized records to the central training table
- Establishes a consistent schema for downstream steps

This step isolates all client-specific data complexity from the rest of the pipeline.

8.3 Step 1.1 – Geocode Stores

Purpose

To associate each store with **stable geographic coordinates** required for weather enrichment.

Processing

- Extracts unique store addresses from training data
- Resolves latitude and longitude using a geocoding service
- Persists coordinates in a dedicated lookup table

Design Notes

- Geocoding is idempotent and cached
- Failures are retried conservatively to respect rate limits

8.4 Step 1.2 – Populate Holidays

Purpose

To generate **location-aware holiday signals** for feature engineering.

Processing

- Identifies all countries and states present in the data
- Generates holiday calendars across relevant years
- Stores holiday metadata centrally

Design Notes

- Holiday data is deterministic and reproducible
- Centralization avoids recomputation across runs

8.5 Step 1.3 – Backfill Weather

Purpose

To enrich historical sales with **weather conditions** aligned by store, date, and hour.

Processing

- Iterates over stores and date ranges
- Queries a historical weather provider
- Persists temperature, precipitation, and humidity

Design Notes

- Weather is fetched once and reused across runs
- Missing data is explicitly handled and flagged

8.6 Step 2.1 – Enrich Training Data

Purpose

To produce a **fully enriched training dataset** that mirrors inference-time feature availability.

Processing

- Joins training data with:
  - Store coordinates
  - Holiday calendar
  - Weather history
- Updates existing records rather than creating duplicates

Design Notes

- Enrichment is performed using Spark for scalability
- This step defines the final data surface for feature engineering

8.7 Step 2.2 – Model Training

Purpose

To train forecasting models that generalize across stores, products, and time.

Processing

- Loads enriched data into Spark
- Generates features using window functions and aggregations
- Splits data into training and validation sets
- Trains models using distributed LightGBM via Ray

Two models are trained independently:

- Revenue forecasting model
- Demand (units) forecasting model

8.8 Feature Engineering Strategy (Training)

Feature engineering during training establishes the **contract** that inference must obey.

Key feature categories include:

- Time-based features (cyclical, seasonal)
- Boolean calendar flags
- Promotion indicators
- Holiday indicators
- Weather signals
- Lag and rolling statistics
- Profile-level aggregates
- Categorical identifiers

All feature names, types, and defaults are captured in versioned schemas.

8.9 Model Artifacts and Versioning

Each training run produces immutable artifacts:

- Versioned model binaries
- Feature schema definitions
- Training metrics and metadata

Artifacts are:

- Timestamped
- Registered centrally
- Promoted explicitly for inference use

This ensures full traceability from prediction back to training data.

8.10 Operational Characteristics

The training pipeline is designed to:

- Run for hours without human intervention
- Resume safely after partial failures
- Support incremental experimentation
- Scale with data volume and client count

Failures in training **never impact live inference**, preserving production stability.

8.11 Key Guarantees

The pipeline guarantees:

- Reproducible model generation
- Strict alignment with inference features
- No dependency on live production traffic
- Auditable and explainable model versions

These guarantees are foundational to maintaining trust in forecast outputs.

9. Information Flow

This section explains **how information moves through the system**, from raw client data to final forecast responses. It focuses on *data dependencies, transformation boundaries, and flow direction* rather than implementation details.

Understanding this flow is critical for:

- Debugging incorrect forecasts
- Reasoning about feature correctness
- Safely modifying training or inference logic

9.1 Training Data Flow

The training data flow is **one-directional and batch-oriented**. No step depends on live inference traffic.

Flow:

Client Databases

→ Forecasting Views

→ Central Training Tables

→ Enrichment Tables

→ Feature Engineering

→ Trained Models

9.1.1 Source: Client Databases

- Each client exposes a read-only forecasting view
- Views normalize transactional sales data into a consistent shape
- No client-side data is modified at any point

These views are the **single source of truth** for historical sales behavior.

9.1.2 Centralization: Training Tables

Extracted data is written into a central training database where:

- Schemas are enforced
- Cross-client joins are possible
- Enrichment steps can operate uniformly

This layer decouples downstream processing from client-specific storage.

9.1.3 Enrichment Flow

Training records are enriched by joining:

- Store coordinates (geocoding)
- Holiday calendars
- Historical weather observations

Each enrichment source is:

- Deterministic
- Persisted independently
- Reusable across training runs

9.1.4 Feature Engineering and Model Output

After enrichment:

- Windowed features are computed
- Lag and rolling statistics are generated
- Categorical fields are normalized

The output of this flow is:

- Feature matrices
- Versioned models
- Feature schemas defining the inference contract

9.2 Inference Request Flow

Inference operates on **request-driven, real-time data flow** with no persistent side effects.

Flow:

Client Request

→ Validation

→ Feature Construction

→ Model Prediction

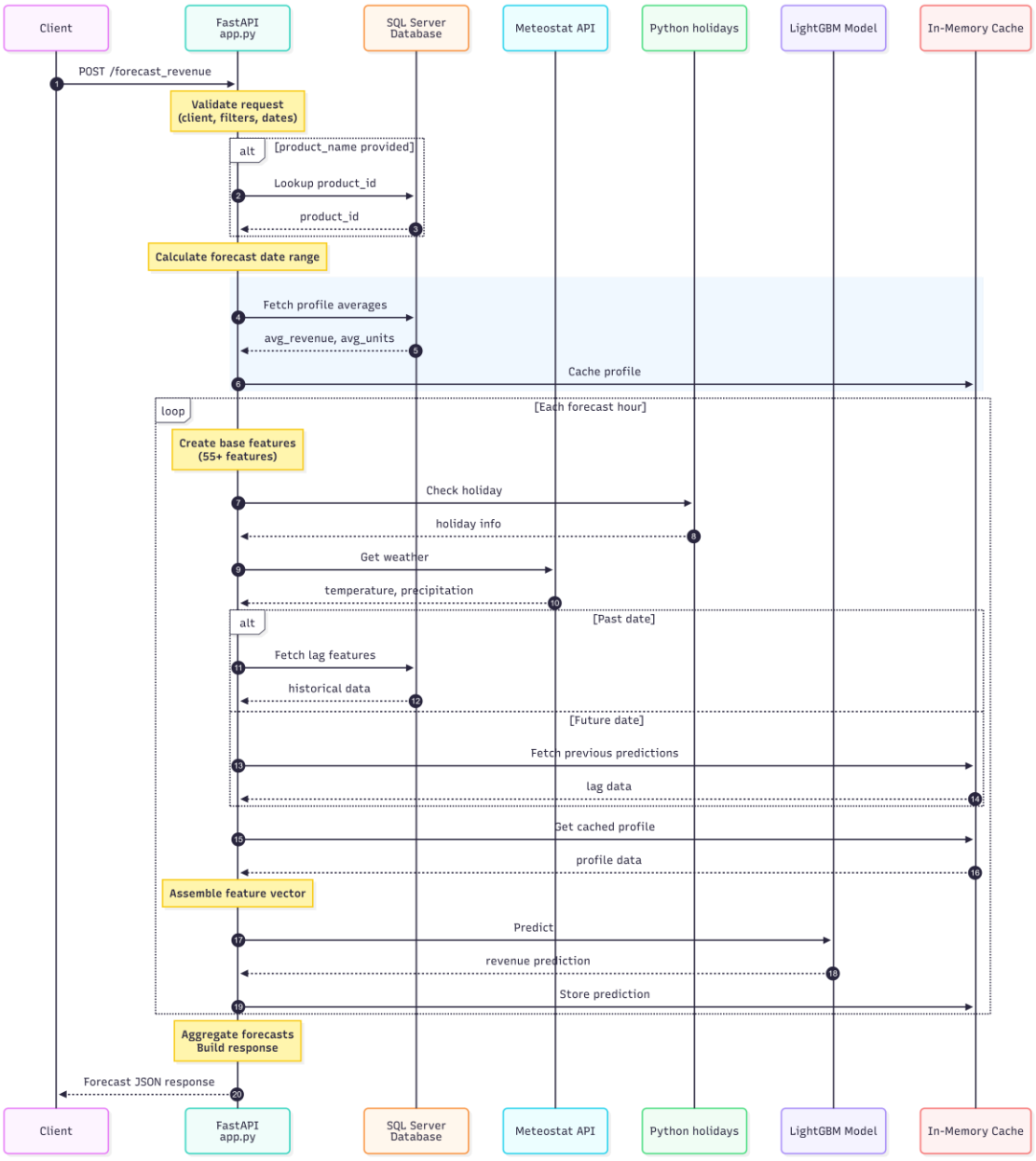
→ Response Assembly

9.2.1 Request Ingress

Incoming requests provide:

- Filter scope (store, product, category, etc.)
- Forecast horizon (dates and/or hours)
- Optional scenario inputs (promotions, weather overrides)

No historical data is supplied directly by the client.



9.2.2 Data Lookup and Caching

The inference layer retrieves:

- Historical sales (for lag features)
- Profile-level aggregates (baselines)

To reduce latency:

- Profiles are cached per request
- Historical lookups are reused across forecast steps

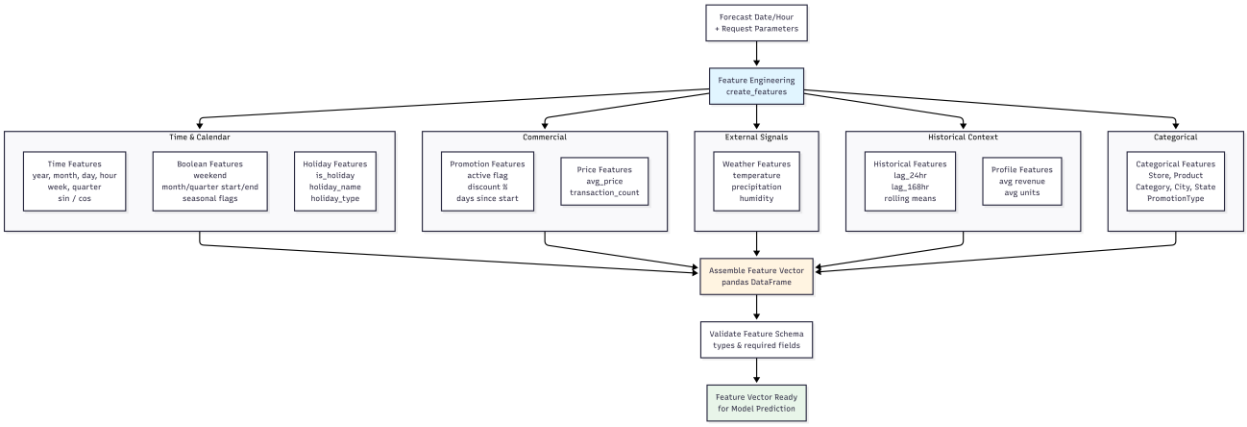
All data access is **read-only**.

9.2.3 Feature Construction Flow

For each forecasted timestamp:

1. Time features are derived from the target date/hour
2. Calendar and seasonal flags are applied
3. Promotion logic is evaluated
4. Holiday indicators are resolved
5. Weather data is fetched or overridden
6. Lag and rolling features are computed
7. Profile features are injected

This mirrors training-time feature construction exactly.



9.2.4 Recursive Prediction Flow

Predictions are generated sequentially:

- Earlier predictions are stored in-memory
- Future lag features reference prior predictions
- No future ground-truth data is ever assumed

This creates a **causal prediction chain** across the forecast horizon.

9.2.5 Response Assembly

Once all timestamps are processed:

- Individual forecasts are aggregated
- Totals and averages are computed
- Metadata (filters, ranges, model version) is attached

The final response is returned as structured JSON.

9.3 AI and Optional Data Flows

AI-driven features operate on **derived forecast outputs**, never on raw transactional data.

Flow:

Forecast Results

→ Summary Extraction

→ Prompt Construction

→ AI Service Call

→ Parsed Insights

Key properties:

- AI failures do not block forecasts
- Outputs are advisory, not authoritative
- Core prediction logic is unaffected

9.4 Flow Boundaries and Isolation

The system enforces strict boundaries:

- Training never depends on inference
- Inference never mutates data
- AI never influences numeric predictions
- External services never become single points of failure

These boundaries prevent cascading failures and preserve correctness.

9.5 Information Flow Guarantees

Across both training and inference, the system guarantees:

- Directional, acyclic data flow
- Deterministic transformations
- Clear ownership of each data source
- No hidden state across requests

Understanding and preserving these flows is essential when extending or refactoring the system.

10. API Endpoints

This section documents the **public-facing API surface** of the Sales Forecasting System. It focuses on endpoint responsibilities, behavioral guarantees, and how each endpoint fits into the overall architecture.

Detailed request and response schemas are covered in later sections; this section explains *what each endpoint is for and how it should be used*.

10.1 Endpoint Categories

All API endpoints fall into one of three categories:

1. **Forecast Endpoints** – Core business functionality
2. **AI / Insight Endpoints** – Optional, explanatory features
3. **Utility Endpoints** – Health and discovery

This separation helps consumers understand which endpoints are **critical**, which are **value-added**, and which are **operational**.

10.2 Forecast Endpoints (Core)

Forecast endpoints are the **primary contract** of the system. They:

- Perform real-time predictions
- Are deterministic for identical inputs
- Never modify persistent data

All forecast endpoints:

- Accept JSON payloads
- Enforce strict validation
- Return structured, machine-consumable responses

10.2.1 POST `/forecast_revenue`

Purpose

Generate revenue forecasts expressed in monetary value.

Behavior

- Produces hourly forecasts by default
- Supports single dates, date ranges, or rolling future windows
- Applies optional scenario inputs such as promotions or weather overrides

Guarantees

- Revenue values are non-negative
- Identical inputs yield identical outputs (given the same model version)
- Forecasts do not depend on request history

10.2.2 POST `/forecast_demand`

Purpose

Generate demand forecasts expressed as units sold.

Behavior

- Shares identical request structure with `/forecast_revenue`
- Uses a dedicated demand model
- Returns integer-based forecasts

Guarantees

- Demand values are non-negative
- Output structure matches revenue forecasts exactly
- Scenario behavior is consistent with revenue forecasting

10.3 Scenario and Override Support

Forecast endpoints support **scenario modeling** without retraining:

- Promotional discounts and durations
- Holiday effects
- Weather overrides

Scenario inputs:

- Affect feature construction only
- Do not persist beyond the request
- Do not mutate trained models

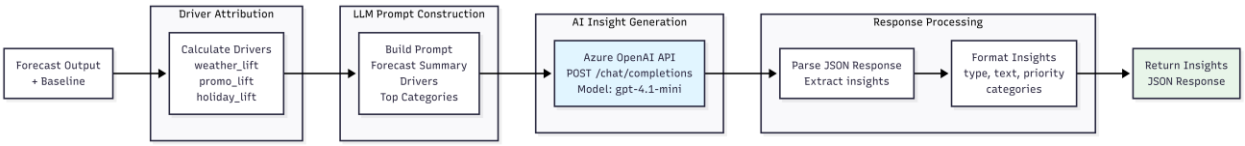
This allows safe experimentation and what-if analysis.

10.4 AI and Insight Endpoints (Optional)

AI endpoints provide **interpretability and usability enhancements**. They never influence numeric forecasts.

If AI services are unavailable, these endpoints:

- Return empty or fallback responses
- Do not impact forecast endpoints



10.4.1 POST `/api/ai/insights`

Purpose

Generate human-readable explanations for forecast behavior.

Behavior

- Consumes aggregated forecast results
- Highlights key drivers such as promotions or weather
- Returns prioritized insight statements

10.4.2 POST `/api/ai/chart_callouts`

Purpose

Generate annotations for significant forecast movements.

Behavior

- Identifies spikes or dips beyond thresholds

- Produces short labels and tooltips
- Designed for frontend chart overlays

10.4.3 POST `/api/ai/parse_promo_event`

Purpose

Convert unstructured promotional text into structured inputs.

Behavior

- Extracts dates, discount levels, and categories
- Infers relative date expressions
- Returns confidence scores

10.5 Utility Endpoints

Utility endpoints support **operability and discoverability**.

10.5.1 GET `/health`

Purpose

Expose service and model readiness.

Behavior

- Confirms API availability
- Reports model load status
- Indicates degraded vs healthy state

This endpoint is safe for:

- Load balancer checks
- Monitoring systems

10.5.2 GET /

Purpose

Provide a minimal service descriptor.

Behavior

- Returns service name and version
- Lists key endpoint paths

10.5.3 GET /docs

Purpose

Expose interactive API documentation.

Behavior

- Auto-generated from endpoint definitions
- Allows live request testing in development

10.6 Error Handling and Contracts

Across all endpoints:

- Validation errors return client-readable messages
- Server errors never leak internal stack traces
- HTTP status codes reflect error category

No endpoint:

- Persists request data
- Mutates client databases
- Depends on hidden server state

10.7 API Stability Guarantees

The API is designed to guarantee:

- Backward-compatible request evolution where possible
- Stable response structures per endpoint
- Explicit versioning through deployment, not URL changes

Consumers can safely integrate without coupling to internal implementation details.

11. Request and Response Reference

This section defines the **formal request and response contracts** for the Sales Forecasting API. It specifies required fields, optional parameters, validation rules, and response structures.

These contracts are authoritative. Any change to them must be treated as a **breaking API change** unless explicitly versioned.

11.1 Forecast Request (Common Structure)

All forecast endpoints (`/forecast_revenue` and `/forecast_demand`) share the same request structure. The meaning of the request is independent of the metric being predicted.

11.1.1 Required Fields

- **client_code** (string)

Identifies the client dataset to query. This field is mandatory for all requests.

11.1.2 Scope Filters (At Least One Required)

At least one of the following filters must be provided to define the forecast scope:

- **store_id** (integer)
- **product_id** (integer)
- **product_name** (string)
- **department** (string)
- **department_type** (string)
- **product_category** (string)

Providing multiple filters narrows the forecast scope.

11.1.3 Location Fields (Optional)

These fields are used primarily for weather resolution:

- **city** (string)
- **state** (string)
- **state_code** (string)
- **zip_code** (string)
- **country** (string, default: "US")
- **latitude** (float)

- **longitude** (float)

If latitude and longitude are provided, geocoding is skipped.

11.1.4 Date and Time Specification (At Least One Required)

A request must specify **what period to forecast** using one of the following patterns:

- **date** – forecast a single date
- **next_days** – forecast N days forward from today
- **date_from** and **date_to** – forecast a date range

Optional time scoping:

- **hour** – forecast a single hour
- **hour_from** and **hour_to** – forecast a time range within each day

11.1.5 Promotion Parameters (Optional)

Promotion fields allow scenario modeling:

- **promotion_type** (integer)
- **promotion_discount_pct** (float, 0–100)
- **promotion_start_date** (YYYY-MM-DD)
- **promotion_end_date** (YYYY-MM-DD)

Promotions affect feature construction only and do not persist.

11.1.6 Weather Overrides (Optional)

Weather overrides allow explicit scenario control:

- **temperature_override** (float)
- **precipitation_override** (float ≥ 0)
- **humidity_override** (float, 0–100)
- **weather_condition_override** (string)

Overrides take precedence over external weather data.

11.2 Validation Rules

Requests are validated before any processing occurs. Common rules include:

1. `client_code` is mandatory
2. At least one scope filter must be provided
3. At least one date specification must be provided
4. `next_days` must be within configured limits
5. `hour_to` must be greater than `hour_from`
6. All dates must use `YYYY-MM-DD` format

Invalid requests fail fast with descriptive error messages.

11.3 Forecast Response (Per Timestamp)

Each forecasted timestamp produces a single forecast record.

Fields

- **store_id** (integer | null)
- **product_id** (integer | null)
- **product_name** (string | null)
- **date** (string, YYYY-MM-DD)
- **hour** (integer, 0–23)

- **forecast** (number)
- **weather_condition** (string | null)
- **model_version** (string)

The meaning of `forecast` depends on the endpoint:

- Revenue (currency) for `/forecast_revenue`
- Units (integer) for `/forecast_demand`

11.4 Aggregated Forecast Response

Forecast responses are returned as an aggregate object.

Structure

- **filters** – normalized request filters
- **forecasts** – array of per-timestamp forecast records
- **total_forecast** – sum of all forecasts
- **average_forecast** – average across timestamps
- **date_range** – formatted date range string
- **time_range** – formatted time range or null

This structure is stable across forecast endpoints.

11.5 AI Endpoint Requests and Responses

AI endpoints define their own contracts but follow consistent principles:

- Inputs reference forecast outputs or summaries
- Outputs are advisory and non-authoritative
- Failures never affect forecast endpoints

Each AI endpoint documents:

- Required and optional fields
- Confidence or priority indicators
- Fallback behavior when AI services are unavailable

11.6 Error Response Format

All error responses follow a consistent structure:

- **detail** (string) – human-readable error description

Errors are returned with appropriate HTTP status codes:

- `400` – malformed or incomplete request
- `422` – validation failure
- `500` – internal processing error
- `503` – service unavailable (e.g., models not loaded)

11.7 Contract Guarantees

The request/response layer guarantees:

- Deterministic behavior for identical inputs
- Stable response structures per endpoint
- No hidden or undocumented fields
- Explicit validation failures

Consumers should rely only on documented fields and guarantees defined in this section.

12. Sample Payloads

This section provides **practical, real-world examples** of request and response payloads for the Sales Forecasting API. These samples are intended for:

- Developers integrating with the API
- QA engineers validating behavior
- Troubleshooting and debugging

All examples assume a development environment and omit authentication for clarity.

12.1 Basic Revenue Forecast (Next N Days)

Request

```
POST /forecast_revenue
Content-Type: application/json
{
  "client_code": "571",
  "store_id": 6,
  "product_id": 817522014428,
  "next_days": 7
}
```

Response (Truncated)

```
{
  "filters": {
    "client_code": "571",
    "store_id": 6,
```

```
    "product_id": 817522014428,
    "product_name": null,
    "department": null,
    "department_type": null,
    "product_category": null
  },
  "forecasts": [
    {
      "store_id": 6,
      "product_id": 817522014428,
      "product_name": null,
      "date": "2025-01-15",
      "hour": 0,
      "forecast": 125.50,
      "weather_condition": "NORMAL",
      "model_version": "ray_lightgbm"
    }
  ],
  "total_forecast": 14735.00,
  "average_forecast": 87.65,
  "date_range": "2025-01-15 to 2025-01-22",
  "time_range": null
}
```


12.2 Demand Forecast for a Date Range

Request


```
{
  "client_code": "571",
  "store_id": 6,
  "product_id": 817522014428,
```

```
  "date_from": "2025-01-15",
  "date_to": "2025-01-18"
}
```

Notes

- Uses the demand model internally
- Response structure matches revenue forecasting
- Forecast values represent units sold

12.3 Forecast with Promotion Scenario

Request

```
{
  "client_code": "571",
  "store_id": 6,
  "product_id": 817522014428,
  "date_from": "2025-01-15",
  "date_to": "2025-01-22",
  "promotion_discount_pct": 25.0,
  "promotion_start_date": "2025-01-15",
  "promotion_end_date": "2025-01-18"
}
```

Notes

- Promotion effects apply only within the specified window
- Outside dates revert to baseline behavior
- No retraining is required

12.4 Forecast with Weather Override

Request

```
{
  "client_code": "571",
  "store_id": 6,
  "product_id": 817522014428,
  "date": "2025-07-04",
  "temperature_override": 102.0,
  "weather_condition_override": "HOT"
}
```

Notes

- Overrides bypass external weather services
- Useful for scenario planning and stress testing

12.5 Time-Window Forecast (Hourly Subset)

Request

```
{
  "client_code": "571",
  "store_id": 6,
  "product_id": 817522014428,
  "date": "2025-01-15",
  "hour_from": 14,
  "hour_to": 18
```

```
}
```

Notes

- Forecasts only selected hours per day
- Reduces computation and response size

12.6 Product Name Resolution

Request

```
{
  "client_code": "571",
  "store_id": 6,
  "product_name": "Pepsi",
  "next_days": 3
}
```

Notes

- Product name is resolved internally to product ID
- Resolution happens once per request

12.7 AI Insights Example

Request

```
{
  "forecast_data": {
    "forecasted_revenue": 15000,
```

```
    "baseline_revenue": 12000,
    "lift_pct": 25.0
  },
  "drivers": {
    "promo_lift": 20.0,
    "weather_lift": 5.0
  }
}
```

Response (Example)

```
{
  "insights": [
    {
      "type": "promo",
      "text": "Promotion is driving a significant uplift in
sales.",
      "priority": "high"
    }
  ]
}
```

12.8 Common Validation Error Example

Request (Invalid)

```
{
  "client_code": "571",
  "next_days": 7
}
```

Response

```
{
  "detail": "Must provide at least one filter"
}
```

12.9 Usage Guidance

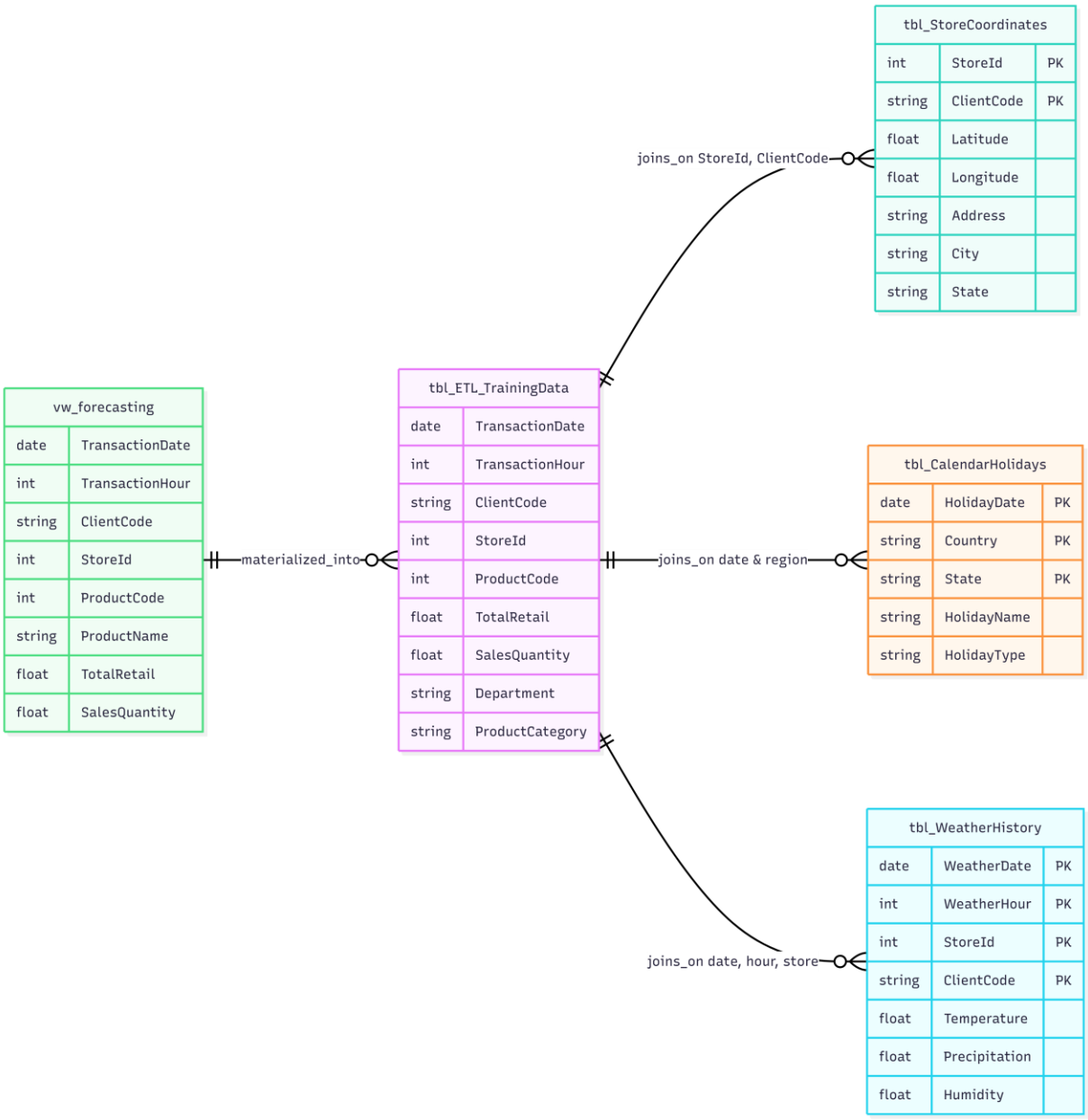
When building integrations:

- Start with basic forecast requests
- Add scenario parameters incrementally
- Validate against `/health` before load testing
- Use sample payloads as regression baselines

These examples represent **supported and expected usage patte

13. Data Stores and Schemas

This section documents all **persistent data stores and schemas** used by the Sales Forecasting System. These structures form the backbone of both training and inference workflows.



13.1 Training Data Tables

13.1.1 Central Training Table

Purpose

Stores normalized historical sales data used for model training.

Characteristics

- Append-only during ETL
- Updated during enrichment
- Never accessed by inference APIs

Key Data Domains

- Transaction timestamps
- Client, store, and product identifiers
- Revenue and unit measures
- Location and category metadata

13.1.2 Store Coordinates Table

Purpose

Maps stores to stable geographic coordinates.

Usage

- Weather enrichment during training
- Weather resolution during inference

This table acts as a long-lived lookup cache.

13.1.3 Holiday Calendar Table

Purpose

Stores country- and state-specific holiday metadata.

Usage

- Training feature enrichment
- Inference-time holiday flags

Holiday data is deterministic and reusable across runs.

13.1.4 Weather History Table

Purpose

Stores historical weather observations aligned to stores.

Usage

- Training enrichment
- Inference lagged weather features

This table prevents repeated external API calls.

13.2 Client-Side Forecasting View

Purpose

Provides a read-only abstraction over transactional sales data for inference.

Design Guarantees

- No writes from the forecasting system
- Stable schema across clients
- Single source of truth for historical sales

13.3 Model Artifacts

Model artifacts are treated as **deployable assets**, not source code.

Artifacts include:

- Revenue model binaries
- Demand model binaries
- Feature schema definitions
- Model registry metadata

These artifacts define the contract between training and inference.

14. Configuration Reference

This section defines all configuration inputs required to operate the system.

14.1 Configuration Sources

Configuration is resolved in the following priority order:

1. Secret management service (if enabled)
2. Environment variables
3. Built-in defaults

No secrets are hardcoded in source code.

14.2 Database Configuration

Required for all environments:

- Database server
- Database name
- Username and password
- ODBC driver

These values must be valid for both training and inference.

14.3 Training Configuration

Controls ETL scope and training behavior:

- Client selection
- Date range selection
- Training window length
- Distributed execution settings

Changing these values affects model behavior and must trigger retraining.

14.4 Inference Configuration

Controls API behavior:

- Maximum forecast horizon
- Model storage location
- Optional AI feature enablement

Inference configuration must never change model semantics.

15. Error Handling

This section defines how errors are surfaced and handled across the system.

15.1 Error Categories

Errors fall into four categories:

- Validation errors
- Dependency errors
- Runtime errors
- Service availability errors

Each category maps to a specific HTTP status code.

15.2 Error Response Contract

All errors return a consistent structure:

- Human-readable description
- No internal stack traces
- No sensitive data

This contract is stable across endpoints.

15.3 Failure Isolation

Failures are isolated to prevent cascading impact:

- Training failures do not affect inference
- AI failures do not affect forecasts
- External API failures trigger fallbacks

16. Testing Guide

This section outlines recommended testing practices.

16.1 Training Pipeline Testing

Recommended checks:

- Database connectivity
- ETL correctness
- Enrichment completeness
- Model artifact generation

Testing should be performed incrementally by pipeline stage.

16.2 Inference API Testing

Recommended checks:

- Health endpoint availability
- Basic forecast correctness
- Scenario handling
- Validation error behavior

Automated tests should use sample payloads as baselines.

17. Troubleshooting

This section provides guidance for diagnosing common issues.

17.1 Training Issues

Common root causes:

- Missing dependencies
- Insufficient memory or CPU
- Database connectivity failures

Training issues never impact live forecasts.

17.2 Inference Issues

Common root causes:

- Missing or incompatible model artifacts
- Database timeouts
- Misconfigured environment variables

Health checks should be the first diagnostic step.

18. Versioning and Change Management

This section defines how system changes are tracked and managed.

18.1 Version Scope

Versions apply to:

- Model artifacts

- Feature schemas
- API behavior

Code changes alone do not imply model changes.

18.2 Change Discipline

All changes must:

- Preserve documented contracts
- Be backward-compatible where possible
- Be traceable through version metadata

This discipline ensures long-term system stability.

19. Model Evaluation & Promotion Criteria

19.1 Overview

Model evaluation ensures that trained models meet defined quality standards before being promoted to production. This section documents the evaluation metrics, validation strategy, promotion criteria, retraining triggers, and rollback procedures used to manage the model lifecycle safely.

19.2 Evaluation Metrics

The training pipeline computes the following metrics for **both revenue and demand models**.

19.2.1 Mean Absolute Error (MAE)

- **Definition:** Average absolute difference between predicted and actual values
- **Formula:** MAE = mean(|y_pred − y_actual|)
- **Interpretation:** Lower is better

- **Units:** Same as target (currency for revenue, units for demand)

Example: An MAE of $50 indicates the model's average hourly error is $50.

19.2.2 Root Mean Squared Error (RMSE)

- **Definition:** Square root of the average squared prediction error
- **Formula:** RMSE = sqrt(mean((y_pred − y_actual)^2))
- **Interpretation:** Lower is better; penalizes large errors more heavily
- **Primary Metric:** Used for hyperparameter tuning and model selection

Example: An RMSE of $75 indicates occasional larger forecast errors.

19.2.3 R^2 Score (Coefficient of Determination)

- **Definition:** Proportion of variance explained by the model
- **Formula:** R^2 = 1 − (SS_res / SS_tot)
- **Range:** Can be negative to 1.0
- **Interpretation:** Higher is better

Example: R^2 = 0.85 indicates the model explains 85% of observed variance.

19.3 Validation Strategy

19.3.1 Train / Validation Split

- **Method:** Chronological time-based split
- **Validation Window:** Last 90 days (configurable via `VALIDATION_DAYS`)
- **Rationale:** Prevents data leakage and simulates real-world forecasting

Split date = max(training_date) − VALIDATION_DAYS

19.3.2 Time-Series Cross-Validation (Optional)

- **Method:** Rolling-window cross-validation
- **Configuration:** `CV_NUM_SPLITS`, `CV_VALIDATION_DAYS`
- **Purpose:** Robust evaluation across multiple historical periods

Enabled only when `CV_NUM_SPLITS > 0`.

19.4 Model Evaluation Process

Performed during `step_2_2_model_training.py`:

1. **Hyperparameter Tuning**
    a. Ray Tune searches hyperparameter space
    b. Optimizes RMSE on validation set
    c. Early stopping after 10 non-improving rounds
2. **Final Training**
    a. Trains on full training set using best parameters
    b. Uses optimal iteration count from validation
3. **Validation Evaluation**
    a. Computes MAE, RMSE, R^2
    b. Metrics calculated separately for revenue and demand
4. **Model Registry Update**
    a. Metrics stored with model version
    b. Location: `models/model_registry.json`

19.5 Promotion Criteria

19.5.1 Minimum Acceptance Thresholds

Revenue Model

- RMSE < current production model
- MAE < current production model

- $R^2 \geq 0.79$

Demand Model

- RMSE < current production model
- MAE < current production model
- $R^2 \geq 0.75$

19.5.2 When to Promote

Promote a model only when:

- All metrics meet minimum thresholds
- Metrics outperform current production model
- No significant cross-validation degradation
- Training completed without errors
- Feature schema matches inference expectations

19.5.3 When NOT to Promote

Do not promote if:

- RMSE or MAE increases
- R^2 below minimum threshold
- Validation metrics regress
- Training warnings or errors exist
- Feature schema mismatch detected
- Cross-validation variance is high

19.6 Model Comparison

Model versions are compared using `model_registry.json`.

Comparison steps:

1. Confirm new metrics exist
2. Compare RMSE and MAE (must improve)
3. Verify R^2 threshold
4. Check logs for training issues
5. Promote only if all checks pass

19.7 Retraining Triggers

Retrain models when:

- New data is available (monthly recommended)
- Data distribution shifts detected
- Production accuracy degrades
- New features introduced
- Business scope changes (stores, products)

19.8 Rollback Criteria

Rollback when:

- Production accuracy degrades
- User complaints increase
- Error rates spike
- Feature incompatibility detected
- Performance regressions observed

Rollback steps:

1. Identify last stable version
2. Restore previous model files
3. Update registry
4. Restart inference API
5. Monitor post-rollback stability

20. Monitoring & Observability

20.1 Overview

Monitoring ensures the system remains reliable, performant, and accurate in production. This section defines metrics, thresholds, alerts, and operational responses.

20.2 API Health Monitoring

20.2.1 Health Endpoint

`GET /health`

Reports:

- Overall status
- Model load state
- Database connectivity

Check every 1–5 minutes.

20.2.2 Health States

- **Healthy:** All checks pass
- **Degraded:** Partial failures; forecasts continue with fallbacks
- **Unhealthy:** API unreachable or returning 5xx

20.3 Performance Monitoring

20.3.1 Latency Targets (p95)

- Forecast endpoints: < 2s

- AI endpoints: < 5s
- `/health`: < 100ms

Alerts:

- Warning: > 2× target
- Critical: > 5× target

20.3.2 Traffic Monitoring

Track:

- Requests per minute
- Endpoint usage
- Client distribution

Alert on:

- Sudden drops (>50%)
- Spikes (>200%)
- No traffic > 1 hour

20.4 Error Rate Monitoring

Target thresholds:

- 4xx < 5%
- 5xx < 1%
- Timeouts < 0.5%

Critical alerts triggered when exceeded.

20.5 Model Performance Monitoring

Monitor:

- Forecast vs actual accuracy (when available)
- Output distributions
- Drift indicators

Manual review recommended weekly.

20.6 External Dependency Monitoring

- Weather APIs: alert if failure > 10%
- AI services: alert if failure > 20%

System degrades gracefully when unavailable.

20.7 Logging & Alerts

Log levels:

- INFO: normal operations
- WARNING: fallbacks, degraded mode
- ERROR: prediction or system failures

Critical alerts:

- Models not loaded
- API unavailable
- Error rate spikes

20.8 Monitoring Tools

Recommended:

- Application Insights / Datadog / Prometheus
- ELK / Log Analytics for logs

Key metrics:

- Latency percentiles
- Error rates
- Prediction times
- DB query latency

20.9 Operational Runbooks

Defined responses for:

- High error rates
- Development: http://localhost:8000
- Dev on cloud: https://io-df-api-dev.modisoft.com
- Production: https://io-df-api-prod.modisoft.comHigh latency
- Model degradation

Runbooks prioritize fast diagnosis, rollback, and service stability.

21. REBUILD FROM SCRATCH CHECKLIST

This section provides a **step-by-step procedure to rebuild the entire Sales Forecasting System from a clean environment**, including database setup, model training, and inference API deployment.

It is intended for **new environment provisioning, disaster recovery, or ownership transitions**.

21.1 Prerequisites

Before starting, ensure the following requirements are met:

Software

- Python **3.9+** (recommended: **3.10 or 3.11**)
- SQL Server **ODBC Driver 17** installed
- Git installed and available in PATH

Access

- Main database access (`SQL_SERVER_MAIN`)
- Client databases (via stored procedure or direct read access)
- Azure Key Vault (optional – secrets management)
- Azure Blob Storage (optional – model storage)
- Azure OpenAI (optional – AI/LLM features)

Repositories

- Access to:
    - Training pipeline repository (`df_model_training`)
    - Inference API repository (`df_model_apis`)

21.2 Step 1: Clone Repositories

Clone both repositories into separate directories:

```
# Clone training pipeline
git clone <training_repo_url>
cd df_model_training

# Clone inference API
cd ..
git clone <inference_repo_url>
cd df_model_apis
```

21.3 Step 2: Set Up Training Pipeline Environment

```
cd df_model_training

# Create virtual environment
python -m venv venv
```

Activate environment:

```
# Windows
venv\Scripts\activate

# Linux / Mac
source venv/bin/activate
```

Install dependencies:

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

Verify installation:

```
python -c "import pandas, lightgbm, pyspark, ray; print('All packages
installed')"
```

21.4 Step 3: Configure Training Pipeline

Copy environment template and configure credentials:

```
cp env_template.txt .env
```

Update `.env` with:

- SQL_SERVER_MAIN
- SQL_DATABASE_MAIN
- SQL_USERNAME_MAIN
- SQL_PASSWORD_MAIN
- Azure credentials (if applicable)

Verify database connectivity:

```
python -c "from kv_helper import get_database_credentials;
print(get_database_credentials())"
```

21.5 Step 4: Run Database Setup Scripts

Execute the following SQL scripts **on the Main Database**, in order:

1. CREATE_TBL_ETL_TRAININGDATA.sql
2. CREATE_TBL_STORECOORDINATES.sql
3. CREATE_TBL_CALENDARHOLIDAYS.sql
4. CREATE_TBL_WEATHERHISTORY.sql

Execute the following **on each Client Database**:

5. CREATE_VIEW_VW_FORECASTING.sql

21.6 Step 5: Run Training Pipeline

Run the full pipeline:

```
python run_full_pipeline.py
```

Optional: Skip steps if data already exists.

Examples

```
# Skip ETL
SKIP_ETL=true python run_full_pipeline.py
```

```
# Skip geocoding
SKIP_GEOCODE=true python run_full_pipeline.py

# Skip holidays
SKIP_HOLIDAYS=true python run_full_pipeline.py

# Skip weather
SKIP_WEATHER=true python run_full_pipeline.py

# Skip enrichment
SKIP_ENRICHMENT=true python run_full_pipeline.py

# Skip training
SKIP_TRAINING=true python run_full_pipeline.py
```

Verify Output

- Models created in `models/` directory (or Azure Blob)
- Log file generated:

```
pipeline_run_YYYYMMDD_HHMMSS.log
```

21.7 Step 6: Set Up Inference API Environment

```
cd ../df_model_apis

python -m venv venv
```

Activate environment and install dependencies:

```
# Windows
venv\Scripts\activate

# Linux / Mac
source venv/bin/activate
```

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

Verify installation:

```
python -c "import fastapi, lightgbm, pandas, sqlalchemy; print('All packages installed')"
```

21.8 Step 7: Configure Inference API

Copy environment template:

```
cp env_template.txt .env
```

Configure:

- Database credentials (same as training)
- Model paths (local or Azure Blob)
- Azure Key Vault (optional)
- Azure OpenAI credentials (optional)

21.9 Step 8: Copy Models to Inference API

If using **local models**, copy from training to inference:

```
# Windows
copy df_model_training\models\*.pkl df_model_apis\models\
copy df_model_training\models\*.json df_model_apis\models\

# Linux / Mac
cp df_model_training/models/*.pkl df_model_apis/models/
cp df_model_training/models/*.json df_model_apis/models/
```

If using **Azure Blob Storage**, models will load automatically at startup.

21.10 Step 9: Start Inference API

Development mode:

```
uvicorn app:app --host 0.0.0.0 --port 8000 --reload
```

Production mode:

```
uvicorn app:app --host 0.0.0.0 --port 8000 --workers 4
```

Verify:

- Swagger UI: http://localhost:8000/docs
- Health check: http://localhost:8000/health

21.11 Step 10: Test with Sample Payloads

Revenue Forecast

```
curl -X POST "http://localhost:8000/forecast_revenue" \
  -H "Content-Type: application/json" \
  -d '{
    "client_code": "571",
    "store_id": 6,
    "product_id": 817522014428,
    "next_days": 7
  }'
```

Demand Forecast

```
curl -X POST "http://localhost:8000/forecast_demand" \
  -H "Content-Type: application/json" \
  -d '{
```

```
    "client_code": "571",
    "store_id": 6,
    "product_id": 817522014428,
    "next_days": 7
  }'
```

Health Check

```
curl http://localhost:8000/health
```

21.12 Common Rebuild Issues

Issue	Resolution
`Module not found`	Activate virtual environment, reinstall dependencies
ODBC driver error	Install ODBC Driver 17 for SQL Server
Database connection failed	Verify credentials, firewall, and connectivity
Models not found	Confirm models exist and paths are correct
Port 8000 in use	Start API on alternate port (`--port 8001`)

22. DATABASE TABLES AND VIEWS DEPENDENCIES

22.1 Overview

The Sales Forecasting System depends on multiple database tables and views spanning **two database layers**:

- **Main Database (ETL / Storage)**
  Central database used for training data, enrichment artifacts, and inference fallback.
- **Client Databases (Read-only)**
  Client-specific databases queried during real-time inference.

This section documents **required vs optional objects**, expected fields, and **system behavior when data is missing**.

22.2 Main Database Tables (Training & Inference Fallback)

22.2.1 tbl_ETL_TrainingData

Purpose
Central training data table populated by the ETL pipeline.

Required Fields

- `StoreId` (INT) – Store identifier
- `ProductCode` (NVARCHAR / BIGINT) – Product identifier
- `TransactionDateTime` (DATETIME) – Transaction timestamp
- `TotalRetail` (DECIMAL / FLOAT) – Revenue per transaction
- `SalesQuantity` (DECIMAL / FLOAT) – Units sold per transaction
- `ClientCode` (NVARCHAR) – Client identifier

Optional Fields (Enhance Accuracy)

- `City`, `State`, `StateCode` – Location attributes
- `Department`, `ProductCategory` – Product classification
- `Latitude`, `Longitude` – Store coordinates (weather lookup)

Default Behavior When Missing

- Table missing → **Training pipeline fails** (hard dependency)
- Optional fields missing → NULL / 0 defaults applied
- No historical data → Profile averages default to:
  `(revenue=0.0, units=0.0, avg_revenue=10.0, avg_units=1.0)`

22.2.2 tbl_StoreCoordinates

Purpose
Stores geocoded store locations for weather feature generation.

Required Fields

- `StoreId` (INT) – Store identifier

- `Latitude` (FLOAT) – Store latitude
- `Longitude` (FLOAT) – Store longitude

Optional Fields

- `Address`, `City`, `State` – Address components (geocoding support)

Default Behavior When Missing

- Table missing → System attempts geocoding from address data
- Coordinates missing → Weather defaults applied:
  - Temperature = 70°F
  - Precipitation = 0
  - Humidity = 50
- Geocoding fallback → Uses **Geopy** if address available

22.2.3 tbl_CalendarHolidays

Purpose
Holiday calendar used for feature engineering.

Required Fields

- `HolidayDate` (DATE) – Holiday date
- `Country` (NVARCHAR) – Country code (e.g., `"US"`)
- `HolidayName` (NVARCHAR) – Holiday name

Optional Fields

- `State`, `HolidayType` – Additional classification

Default Behavior When Missing

- Table missing → Python `holidays` library used at runtime
- Holiday missing → Library fallback lookup
- No holiday data →
  `is_holiday = 0`, `holiday_name = "NONE"`

22.2.4 tbl_WeatherHistory

Purpose
Historical weather data used during training.

Required Fields

- `StoreId` (INT) – Store identifier
- `WeatherDate` (DATE) – Weather date
- `Hour` (INT, 0–23) – Hour of day
- `Temperature` (FLOAT) – Temperature (°F)
- `Precipitation` (FLOAT) – Inches
- `Humidity` (FLOAT) – Percentage

Default Behavior When Missing

- Table missing → Training pipeline fetches from **Meteostat API**
- Partial data missing → Normal weather defaults applied
- Inference → **Does not use this table** (queries Meteostat directly)

22.2.5 tbl_Clients (Reference Table)

Purpose
Client registry for multi-client training orchestration.

Required Fields

- `Client_Code` (NVARCHAR) – Client identifier

Default Behavior When Missing

- Table missing → Clients provided manually via configuration
- Single-client training supported without this table

22.3 Client Database Views (Inference – Real-time)

22.3.1 vw_forecasting

Purpose
Primary real-time data source for inference.

Required Fields (Minimum)

- `ClientCode` (NVARCHAR) – Client identifier
- `StoreId` (INT) – Store identifier
- `ProductCode` (NVARCHAR / BIGINT) – Product identifier
- `ProductName` (NVARCHAR) – Product name lookup
- `TransactionDateTime` (DATETIME) – Transaction timestamp
- `TotalRetail` (DECIMAL / FLOAT) – Revenue
- `SalesQuantity` (DECIMAL / FLOAT) – Units

Optional Fields (Enhance Accuracy)

- `City`, `State`, `StateCode`
- `Department`, `ProductCategory`
- `Latitude`, `Longitude`

Default Behavior When Missing

- View missing → Fallback to `tbl_ETL_TrainingData`
- Required fields missing → Inference request fails
- Optional fields missing → Defaults or feature omission
- Missing coordinates → Weather defaults applied

Core Queries Used by System

Product lookup by name

```
SELECT ProductCode
FROM vw_forecasting
WHERE ProductName = :product_name
  AND ClientCode = :client_code
```

Historical features (lag / rolling)

```
SELECT SUM(TotalRetail) AS revenue,
       SUM(SalesQuantity) AS units
FROM vw_forecasting
WHERE StoreId = :store_id
  AND ProductCode = :product_id
  AND TransactionDateTime >= :start_dt
  AND TransactionDateTime < :end_dt
```

Profile averages

```
SELECT AVG(TotalRetail) AS avg_revenue,
       AVG(SalesQuantity) AS avg_units
FROM vw_forecasting
WHERE StoreId = :store_id
  AND ProductCode = :product_id
```

22.4 Main Database Stored Procedures (Inference)

22.4.1 USP_GetClientDBConnection

Purpose

Resolves client-specific database connection details.

Required Parameters

- @code (NVARCHAR) – Client code

Returns

- Server name
- Database name
- Username (if stored)
- Password (if stored)

Default Behavior When Missing

- Procedure missing → System uses main database only
- Client not found → Fallback to main DB
- Incomplete credentials → Warning logged, fallback used

22.5 Data Freshness Requirements

Training Pipeline

- `tbl_ETL_TrainingData` → Refresh before training
- `tbl_CalendarHolidays` → Annual update sufficient
- `tbl_WeatherHistory` → Backfill once, update as needed
- `tbl_StoreCoordinates` → Static (update only on store changes)

Inference API

- `vw_forecasting` → Real-time or near-real-time
- `tbl_ETL_TrainingData` → Acceptable as slightly stale fallback

22.6 Dependency Summary

Table / View	Database	Required	Purpose	Default When Missing
tbl_ETL_TrainingData	Main	Yes (Training)	Training data storage	Training fails
tbl_StoreCoordinates	Main	Optional	Store locations	Geocode or defaults
tbl_CalendarHolidays	Main	Optional	Holiday calendar	Python holidays library
tbl_WeatherHistory	Main	Optional (Training)	Historical weather	Meteostat API

vw_forecasting	Client	Yes (Inference)	Real-time data	Fallback to ETL
USP_GetClientDBConnection	Main	Optional	Client DB mapping	Main DB only

23. DESIGN CAVEATS AND LIMITATIONS

23.1 Overview

This section documents known limitations, assumptions, and edge cases that developers and operators should be aware of. These are **design constraints** of the current implementation (not bugs) and may require operational guardrails or future enhancements.

23.2 Low-Volume Products

23.2.1 Issue

Products with very low sales volume (e.g., **< 1 sale/week**) provide limited historical signal, making forecasting inherently unstable.

23.2.2 Impact

- Forecast accuracy may be lower for low-volume items
- Historical averages can be noisy or not representative
- Lag features (`24hr`, `168hr`) may frequently evaluate to **0**

23.2.3 Mitigation

- Fallback to **profile averages** (store-level or category-level)
- Default values used when history is missing:
  - revenue/units = `0.0`
  - avg_revenue = `10.0`
  - avg_units = `1.0`
- Consider aggregation at **category/department level**

23.2.4 Recommendations

- If `< 5 historical transactions`: prefer **category-level** forecasts
- Track accuracy separately for high-volume vs low-volume products
- Retrain more frequently if low-volume mix changes materially

23.3 New Products (Cold Start)

23.3.1 Issue

Products with **no historical transactions** cannot leverage lag features or historical averages.

23.3.2 Impact

- Lag features default to `0.0`
- Profile averages fall back to category/department averages (if available)
- Predictions rely more heavily on:
    - time features (day-of-week, hour, season)
    - store attributes and location
    - category/department (if provided)
    - weather and holidays

23.3.3 Mitigation

- Use category/department profile averages when product history is unavailable
- Use time + location features as baseline predictors
- Promotions may override baseline behavior via promo features

23.3.4 Recommendations

- For new products, always pass **category/department** when available
- Consider manual override for the first **1–2 weeks** after launch
- Use category-level forecasting until sufficient product history exists

23.4 Weather Data Assumptions

23.4.1 Issue

Weather is sourced from **Meteostat**, which may:

- have gaps for certain locations/dates
- map stores to nearest station (not exact)
- vary in historical accuracy by region
- provide weaker fidelity beyond **7–10 days**

23.4.2 Impact

- Weather features may not reflect true store conditions
- Missing weather defaults to:
  - Temperature = `70°F`
  - Precipitation = `0`
  - Humidity = `50`
- Longer horizons (e.g., **>30 days**) rely on less reliable weather assumptions

23.4.3 Mitigation

- Manual weather overrides supported via request payload
- Defaults are intentionally "normal" to avoid extreme bias
- Weather condition classification (`NORMAL`, `HOT`, `RAINY`, etc.) can still be used even if exact values are missing

23.4.4 Recommendations

- Monitor weather API latency and failure rates
- For critical planning, supply weather overrides
- Consider multi-source weather integration for redundancy

23.5 Holiday Data Assumptions

23.5.1 Issue

Holiday detection uses:

- Python `holidays` library (runtime) or `tbl_CalendarHolidays`
- country/state codes must match library conventions (e.g., `US`, `IN`)
- regional/local holidays may be missing

23.5.2 Impact

- Holidays not in table/library are treated as non-holidays
- Store-specific events (anniversaries, local closures) not modeled
- Holiday effects can vary by category, but category-specific holiday impact is not explicitly modeled

23.5.3 Mitigation

- Fallback to Python library when table missing
- Custom holidays can be added manually to `tbl_CalendarHolidays`
- Promotions can capture certain store-specific events

23.5.4 Recommendations

- Populate holiday table fully before training
- Review holiday uplift by category and consider category-specific features if needed
- Maintain documentation of regional holidays and encode them via the table

23.6 Recursive Forecasting Limitations

23.6.1 Issue

For multi-day forecasts, **Day 2+ predictions depend on Day 1 predictions** for lag features (recursive forecasting), which can amplify errors.

23.6.2 Impact

- Forecast accuracy generally degrades as horizon increases
- Errors can propagate across future days
- Long horizons (e.g., **>30 days**) may drift more noticeably

23.6.3 Mitigation

- Actual historical data is used for past timestamps
- Future timestamps use `previous_predictions` cache
- Forecast horizon limited by `MAX_FORECAST_DAYS` (default: `90`)

23.6.4 Recommendations

- Prefer **7–14 days** for highest reliability
- Re-run forecasts daily (rolling horizon) so Day 2 uses real Day 1 actuals when available
- Monitor accuracy vs horizon and alert on degradation trends

23.7 Database Dependency

23.7.1 Issue

Database access is required for:

- lag/rolling historical features
- profile averages
- product name resolution
- store location lookups

23.7.2 Impact

- Database outages force default/fallback behavior (reduced accuracy)
- Network latency increases end-to-end response time
- Higher request volumes increase DB load

23.7.3 Mitigation

- Connection pooling reduces overhead
- Caching (`profiles_cache`, `historical_cache`) reduces repeated queries
- Graceful degradation when DB unavailable (defaults applied)
- Fallback to ETL table if client DB is unavailable

23.7.4 Recommendations

- Monitor connection pool utilization and query latency
- Keep cache TTLs aligned to business needs (`historical_cache_ttl_seconds = 600`)
- Consider read replicas for high scale
- Add DB query timeouts to avoid hanging requests

23.8 Model Version / Feature Schema Mismatch

23.8.1 Issue

Feature schemas may change between training runs. If inference and training schemas diverge, prediction quality can degrade.

23.8.2 Impact

- Missing features default to `0` / `NULL`
- Extra features are ignored/pruned
- Model performance may degrade if feature logic changed but model not retrained

23.8.3 Mitigation

- Feature schema validation at startup (warnings logged)
- Model registry tracks `feature_schema_version`
- Extra features are automatically pruned

23.8.4 Recommendations

- Retrain models whenever feature engineering changes

- Record feature changes in registry metadata
- Validate new models in staging prior to production rollout
- Monitor forecast distributions after model upgrades

23.9 Time Zone Assumptions

23.9.1 Issue

The system uses a fixed timezone (`API_TIMEZONE = "Asia/Kolkata"`) for:

- date calculations
- hour extraction
- time-based feature generation

23.9.2 Impact

- Assumes all stores operate in the same timezone
- Hour features (0–23) align to server timezone, not store-local time
- DST transitions are not handled

23.9.3 Mitigation

- `API_TIMEZONE` can be changed per deployment
- Store-specific timezone handling is not implemented

23.9.4 Recommendations

- Deploy separate instances per timezone if needed
- Or extend feature engineering to incorporate store-local timezone offsets

23.10 Summary of Limitations

Limitation	Severity	Impact	Mitigation
Low volume products	Medium	Lower accuracy	Use category-level forecasts

New products (cold start)	Medium	Limited features	Use category/department averages
Weather data gaps	Low	Defaults used	Manual overrides supported
Holiday coverage	Low	Regional holidays missed	Populate holiday table manually
Recursive forecasting	Medium	Error propagation	Limit forecast horizon
Database dependency	High	Degraded accuracy if unavailable	Caching + fallbacks
Model/schema mismatch	High	Incorrect/unstable predictions	Schema validation + staging tests
Time zone assumption	Medium	Multi-timezone not supported	Separate deployments / timezone offsets

24. LOCAL DEBUGGING GUIDE

This section provides **practical troubleshooting guidance** for developers and operators when running the training pipeline or inference API locally or in non-production environments.

24.1 Logging Configuration

24.1.1 Default Logging Setup

The system uses Python's built-in `logging` module with the following defaults:

- **Log Level:** `INFO`
- **Format:**
  `%(asctime)s - %(levelname)s - %(message)s`
- **Output:** Console (`stdout` / `stderr`)

Log Locations

- **Inference API:** Console output when running via `uvicorn`

- **Training Pipeline:**
  - Console output
  - File: `pipeline_run_YYYYMMDD_HHMMSS.log`

24.1.2 Enabling Verbose (DEBUG) Logging

Enable DEBUG logging to diagnose feature engineering, database, or model issues.

Inference API

```
# Windows (PowerShell)
$env:LOG_LEVEL="DEBUG"
uvicorn app:app --host 0.0.0.0 --port 8000 --reload

# Linux / Mac
LOG_LEVEL=DEBUG uvicorn app:app --host 0.0.0.0 --port 8000 --reload
```

Alternative (temporary code change)

In `app.py`, change:

```
logging.basicConfig(level=logging.INFO, ...)
```

to:

```
logging.basicConfig(level=logging.DEBUG, ...)
```

Training Pipeline

In `run_full_pipeline.py`, update:

```
logging.basicConfig(level=logging.DEBUG, ...)
```

24.1.3 Log File Locations

Training Pipeline

- File: `pipeline_run_YYYYMMDD_HHMMSS.log`
- One log file per execution
- Retention: **manual cleanup**

Inference API

- Default: Console only
- File logging must be explicitly configured

24.1.4 Adding File Logging to Inference API

To enable rotating file logs, update `app.py`:

```
import logging
from logging.handlers import RotatingFileHandler

file_handler = RotatingFileHandler(
    "api.log",
    maxBytes=10 * 1024 * 1024,  # 10 MB
    backupCount=5
)

file_handler.setLevel(logging.DEBUG)
file_handler.setFormatter(
    logging.Formatter(
        "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
    )
)

logger = logging.getLogger(__name__)
logger.addHandler(file_handler)
```

24.2 Common Error Categories and Resolutions

24.2.1 Database Connection Errors

Typical Errors

- `Failed to initialize database connection`
- `ODBC Driver not found`
- `Login failed for user`

Common Causes

- Invalid credentials
- ODBC driver missing
- Firewall restrictions
- Database server unavailable

Resolution Steps

1. Verify `.env` values:
    a. `SQL_SERVER_MAIN`
    b. `SQL_DATABASE_MAIN`
    c. `SQL_USERNAME_MAIN`
    d. `SQL_PASSWORD_MAIN`
2. Test connection manually:

```
import pyodbc
conn = pyodbc.connect(
    "DRIVER={ODBC Driver 17 for SQL Server};"
    "SERVER=your-server.database.windows.net,1433;"
    "DATABASE=your_database;"
    "UID=your_username;"
    "PWD=your_password"
)
```

3. Install ODBC driver:

- Windows: Microsoft installer
- Linux: `sudo apt-get install unixodbc`
- Mac: `brew install unixodbc`

4. Check firewall rules and network access:

```
telnet your-server.database.windows.net 1433
```

24.2.2 Model Loading Errors

Typical Errors

- `Model not found`
- `Revenue model not loaded`
- `Error loading models`

Resolution Steps

1. Verify model files exist:

   ```
   ls models/revenue_model_latest.pkl
   ls models/units_model_latest.pkl
   ```

2. Confirm paths in `.env`:

   ```
   MODELS_PATH=models
   MODEL_REGISTRY_PATH=models/model_registry.json
   ```

3. Validate Azure Blob configuration (if used)
4. Test manual model loading:

   ```
   import pickle
   with open("models/revenue_model_latest.pkl", "rb") as f:
              model = pickle.load(f)
   print(type(model))
   ```

5. Re-run training pipeline if needed

24.2.3 Feature Engineering Errors

Typical Errors

- `Error creating features`
- `Feature schema mismatch`
- Runtime calculation errors

Resolution Steps

1. Enable DEBUG logging
2. Verify feature schema files:

```
ls models/feature_schema_revenue.json
ls models/feature_schema_units.json
```

3. Validate input payload:
- Required identifiers present
- Correct date format (`YYYY-MM-DD`)
- Numeric fields contain numbers
4. Verify optional dependencies:

```
pip install meteostat holidays geopy
```

5. Inspect feature engineering logic for:
- Division by zero
- NULL handling
- Date arithmetic issues

24.2.4 API Request Validation Errors

Typical Errors

- `422 Validation Error`
- `Missing required field`
- `Invalid date format`

Resolution Steps

1. Confirm payload meets API contract:
    a. `client_code` required
    b. At least one filter: `store_id`, `product_id`, or `product_name`
    c. Valid date specification
2. Validate JSON format:

```
import json
json.loads(request_payload)
```

3. Use Swagger UI for testing: http://localhost:8000/docs

24.2.5 External Service Errors

Services Affected

- Meteostat (weather)
- Azure OpenAI (AI features)
- Geocoding services

Resolution Steps

1. Test connectivity: `ping api.meteostat.net`
2. Verify credentials (Azure OpenAI)
2. Confirm graceful degradation:
    - Weather → defaults
    - Geocoding → defaults
    - AI endpoints → empty responses
4. Check rate limits and quotas

24.2.6 Performance Issues

Symptoms

- Slow responses (>5s)

- High memory usage
- Database timeouts

Resolution Steps

1. Reduce forecast horizon
2. Review database query performance
3. Monitor cache hit/miss ratios
4. Scale API workers:
   ```
   uvicorn app:app --workers 4
   ```
5. Profile code:
   ```
   python -m cProfile -o profile.stats app.py
   ```

24.3 Standard Debugging Checklist

When diagnosing issues, follow this sequence:

1. Check health endpoint: `curl http://localhost:8000/health`
2. Verify environment variables:
   ```
   python -c "import os; print([k for k in os.environ if 'SQL' in k or 'MODEL' in k])"
   ```
3. Enable DEBUG logging
4. Inspect logs:
   - Training: `pipeline_run_*.log`
   - Inference: Console or `api.log`
5. Validate database connectivity
6. Confirm models are loaded
7. Test with minimal payload
8. Validate external services
9. Review recent code/config changes
10. Isolate issue by simplifying requests

24.4 Getting Help

If issues persist:

1. Collect diagnostics:
    a. `/health` response
    b. Recent logs (50–100 lines)
    c. Full error stack traces
    d. Request payload
    e. Sanitized environment config
2. Review documentation:
    a. This handbook
    b. Repository READMEs
    c. Swagger API docs
3. Escalate with:
    a. Steps to reproduce
    b. Expected vs actual behavior
    c. Supporting logs

APPENDIX A: ENVIRONMENT VARIABLES – QUICK REFERENCE

This appendix lists all environment variables used by the **training pipeline** and **inference API**.
Variables are grouped by scope and marked **Required** or **Optional**.

A.1 Required (Training & Inference)

Variable	Description
`SQL_SERVER_MAIN`	Main SQL Server hostname and port
`SQL_DATABASE_MAIN`	Main database name
`SQL_USERNAME_MAIN`	Database username
`SQL_PASSWORD_MAIN`	Database password
`SQL_ODBC_DRIVER`	ODBC driver (e.g., `ODBC Driver 17 for SQL Server`)

A.2 Training-Specific

Variable	Description
ETL_CLIENT_CODES	Comma-separated client codes
ETL_DATA_YEARS	Years of data to extract
ETL_BUSINESS_TYPE_ID	Business type identifier
TRAINING_DATA_YEARS	Number of years used for training
TRAIN_BUSINESS_TYPE_ID	Business type used during training
RAY_NUM_WORKERS	Parallel workers for Ray training

A.3 Inference-Specific

Variable	Description
MAX_FORECAST_DAYS	Maximum forecast horizon (default: 90)

A.4 Azure Blob Storage (Optional)

Variable	Description
AZURE_USE_BLOB	Enable Azure Blob model storage
AZURE_STORAGE_CONNECTION_STRING	Blob storage connection string
AZURE_STORAGE_ACCOUNT	Storage account name
AZURE_MODEL_CONTAINER	Container holding trained models

A.5 Azure Key Vault (Optional)

Variable	Description
AZURE_KEY_VAULT_NAME	Key Vault name for secret management

A.6 Azure OpenAI (Optional – AI Features)

Variable	Description
AZURE_OPENAI_API_KEY	OpenAI API key
AZURE_OPENAI_ENDPOINT	Azure OpenAI endpoint
AZURE_OPENAI_API_VERSION	API version
AZURE_OPENAI_DEPLOYMENT_NAME	Model deployment name

A.7 Pipeline Skip Flags (Optional)

Used to bypass steps during development or recovery.

Variable	Skipped Step
SKIP_ETL	Data extraction
SKIP_GEOCODE	Store geocoding
SKIP_HOLIDAYS	Holiday population
SKIP_WEATHER	Weather backfill
SKIP_ENRICHMENT	Feature enrichment
SKIP_TRAINING	Model training

APPENDIX B: API ENDPOINTS – QUICK REFERENCE

Forecast Endpoints

Endpoint	Method	Purpose
/forecast_revenue	POST	Forecast revenue
/forecast_demand	POST	Forecast demand (units)

AI / LLM Endpoints

Endpoint	Method	Purpose
/api/ai/insights	POST	Generate insights

/api/ai/chart_callouts	POST	Chart annotations
/api/ai/compose_lift	POST	Scenario lift calculation
/api/ai/parse_promo_event	POST	Parse promotion text

Utility Endpoints

Endpoint	Method	Purpose
/health	GET	Health check
/	GET	API metadata
/docs	GET	Swagger UI

Base URLs

- Development: http://localhost:8000
- Dev on cloud: https://io-df-api-dev.modisoft.com
- Production: https://io-df-api-prod.modisoft.com

APPENDIX C: TRAINING PIPELINE – QUICK REFERENCE

Pipeline Steps

Step	Script	Purpose	Output
0.1	step_0_1_extract_training_data.py	Extract training data	tbl_ETL_TrainingData
1.1	step_1_1_geocode_stores.py	Geocode stores	tbl_StoreCoordinates
1.2	step_1_2_populate_holidays.py	Populate holidays	tbl_CalendarHolidays
1.3	step_1_3_backfill_weather.py	Backfill weather	tbl_WeatherHistory

2.1	`step_2_1_enrich_training_data.py`	Feature enrichment	Enriched ETL table
2.2	`step_2_2_model_training.py`	Train models	`.pkl` models

Run Full Pipeline

```
python run_full_pipeline.py
```

Skip Steps

Set any `SKIP_*` environment variable to `true`.

APPENDIX D: CONTACTS AND OWNERSHIP

Area	Primary Contact	Backup Contact
Forecasting UI	Ketan	[Name]
Forecasting Backend	Nitin	[Name]
Database	Dalbir	Sandeep
AKS / DevOps	Hirshant	[Name]
Product Owner	Ebuka	[Name]

END OF HANDBOOK

This completes the Sales Forecasting System Technical Handbook.

For updates and latest documentation, refer to the repository README files.

Version: 2.0.0

Last Updated: January 2025