# Sales Forecast Product — Python + FastAPI + Prophet + XGBoost/LightGBM + SQL Server

Complete implementation guide, architecture, folder structure, and sample code to build a territory/area/region-wise sales forecasting product using Python, FastAPI, Prophet, and tree-based models (LightGBM/XGBoost) with SQL Server as the datastore.

## Overview

This document guides you through building a production-ready Sales Forecasting system. It covers: - Data model and SQL Server schema - ETL & preprocessing pipeline - Baseline forecasting with Prophet - Gradient-boosted models (LightGBM / XGBoost) - Model training, evaluation, and persistence - FastAPI service (endpoints: train, predict, status) - Containerization (Docker), scheduling, and MLOps suggestions

## Requirements

* Python 3.10+
* SQL Server (express or full)
* Docker & Docker Compose (optional but recommended)

Python packages (sample requirements.txt):

fastapi  
uvicorn[standard]  
pandas  
numpy  
sqlalchemy  
pyodbc  
pymssql  
prophet  
lightgbm  
xgboost  
scikit-learn  
pydantic  
joblib  
mlflow  
alembic  
python-dotenv

Note: Prophet installs as prophet (or fbprophet in older setups). Use the variant that matches your environment.

## High-level Architecture

+--------------------+ +-----------------+ +----------------+  
| SQL Server (Raw) | ---> | ETL / Features | ---> | Model Training |  
+--------------------+ +-----------------+ +----------------+  
 | |  
 v v  
+--------------------+ +----------------+  
| Forecast DB (SQL) | <--- FastAPI ---> | Model Store |  
+--------------------+ +----------------+  
 |  
 v  
+--------------------+  
| Dashboard / BI |  
+--------------------+

Components: - **Extraction/ETL**: Pull data from SQL Server, create features and aggregates (region/territory/month). - **Training module**: Train Prophet (time-series) and LightGBM/XGBoost (tabular with lags & exogenous features). - **API**: FastAPI exposes endpoints for forecast queries and retraining. - **Model store**: Filesystem, SQL table, or MLflow to store trained artifacts and metadata. - **Dashboard**: PowerBI / Streamlit / Metabase for visualization.

## Data Model & SQL Schema (recommended)

Tables: 1. sales\_raw — raw transactional or aggregated monthly sales

CREATE TABLE sales\_raw (  
 id INT IDENTITY PRIMARY KEY,  
 sale\_date DATE NOT NULL,  
 region VARCHAR(100),  
 territory VARCHAR(100),  
 area VARCHAR(100),  
 product\_id INT,  
 sales\_amount FLOAT,  
 sales\_qty FLOAT,  
 channel VARCHAR(50)  
);

1. sales\_agg\_monthly — pre-aggregated per month for faster training

CREATE TABLE sales\_agg\_monthly (  
 id INT IDENTITY PRIMARY KEY,  
 year INT,  
 month INT,  
 yearmonth CHAR(7), -- e.g. '2023-01'  
 region VARCHAR(100),  
 territory VARCHAR(100),  
 area VARCHAR(100),  
 product\_id INT,  
 total\_sales FLOAT,  
 total\_qty FLOAT  
);

1. model\_registry — store metadata for models

CREATE TABLE model\_registry (  
 id INT IDENTITY PRIMARY KEY,  
 model\_name VARCHAR(100),  
 model\_type VARCHAR(50), -- prophet, lgbm, xgb  
 version VARCHAR(50),  
 path VARCHAR(500),  
 metrics JSON, -- store RMSE, MAE  
 trained\_at DATETIME  
);

## Folder Structure

sales-forecast/  
├─ app/  
│ ├─ main.py # FastAPI app  
│ ├─ api/  
│ │ ├─ endpoints.py # endpoints (train, predict, status)  
│ ├─ db/  
│ │ ├─ db.py # SQLAlchemy engine + session  
│ │ ├─ models.py # ORM models (optional)  
│ ├─ services/  
│ │ ├─ etl.py # ETL and feature engineering  
│ │ ├─ prophet\_train.py # Prophet training utilities  
│ │ ├─ tree\_train.py # LGBM/XGB training utilities  
│ │ ├─ inference.py # prediction helpers  
│ ├─ utils/  
│ │ ├─ metrics.py # evaluation metrics  
│ │ ├─ persistence.py # save/load model  
├─ notebooks/  
├─ scripts/  
│ ├─ run\_etl.py  
│ ├─ train\_all.py  
├─ docker/  
│ ├─ Dockerfile  
│ ├─ docker-compose.yml  
├─ requirements.txt  
├─ README.md

## ETL & Feature Engineering (app/services/etl.py)

Key steps: - Pull data from sales\_raw or sales\_agg\_monthly. - Ensure continuous months per series (fill missing with zero). - Create lags: lag\_1, lag\_3, lag\_6, rolling means. - Add time features: month, quarter, is\_month\_start, seasonality flags. - Optionally add external regressors: promotions, holidays, weather, macro.

# app/services/etl.py  
import pandas as pd  
from sqlalchemy import text  
  
def load\_monthly\_sales(engine, start\_date=None, end\_date=None):  
 q = "SELECT yearmonth, region, territory, area, product\_id, total\_sales FROM sales\_agg\_monthly"  
 df = pd.read\_sql(q, engine)  
 df['ds'] = pd.to\_datetime(df['yearmonth'] + '-01')  
 df = df.sort\_values(['region','territory','area','product\_id','ds'])  
 return df  
  
  
def create\_lags(df, group\_cols, target\_col='total\_sales', lags=[1,3,6]):  
 df = df.copy()  
 for lag in lags:  
 df[f'lag\_{lag}'] = df.groupby(group\_cols)[target\_col].shift(lag)  
 df['rolling\_3'] = df.groupby(group\_cols)[target\_col].shift(1).rolling(3).mean().reset\_index(level=group\_cols, drop=True)  
 return df

## Prophet (Time-Series) — Baseline

Prophet expects a DataFrame with ds (date) and y (value). For region/territory-wise forecasts you’ll run one Prophet model per series (e.g., per territory or per (region,product) pair).

Example training loop:

# app/services/prophet\_train.py  
from prophet import Prophet  
import pandas as pd  
  
def train\_prophet(df\_series, yearly\_seasonality=True, weekly\_seasonality=False):  
 m = Prophet(yearly\_seasonality=yearly\_seasonality, weekly\_seasonality=weekly\_seasonality)  
 # Add holidays or regressors if available  
 m.fit(df\_series)  
 return m  
  
  
def forecast\_prophet(m, periods=6, freq='MS'):  
 future = m.make\_future\_dataframe(periods=periods, freq=freq)  
 forecast = m.predict(future)  
 return forecast[['ds','yhat','yhat\_lower','yhat\_upper']]

**Notes**: - If you have thousands of time-series, training a Prophet model per series may be costly. Consider grouping or using global models (TFT or deep learning) for scale.

## LightGBM / XGBoost (Tabular Approach)

Instead of one model per time series, train a single model that takes series identifiers (region/territory) as categorical features plus lags and time features. This scales much better.

Key ideas: - Input: ds, region, territory, area, product\_id, month, year, lag\_1, lag\_3, rolling\_3, is\_holiday, promo\_flag. - Target: total\_sales for the next month.

Sample training code (LightGBM):

# app/services/tree\_train.py  
import lightgbm as lgb  
from sklearn.model\_selection import TimeSeriesSplit  
from sklearn.metrics import mean\_squared\_error  
  
  
def train\_lgb(train\_df, features, target='total\_sales'):  
 X = train\_df[features]  
 y = train\_df[target]  
 # Use a time-based split or custom split  
 tscv = TimeSeriesSplit(n\_splits=3)  
 models = []  
 for train\_idx, val\_idx in tscv.split(X):  
 X\_tr, X\_val = X.iloc[train\_idx], X.iloc[val\_idx]  
 y\_tr, y\_val = y.iloc[train\_idx], y.iloc[val\_idx]  
 dtrain = lgb.Dataset(X\_tr, y\_tr)  
 dval = lgb.Dataset(X\_val, y\_val, reference=dtrain)  
 params = {'objective': 'regression', 'metric': 'rmse', 'verbosity': -1}  
 bst = lgb.train(params, dtrain, valid\_sets=[dtrain,dval], early\_stopping\_rounds=50, num\_boost\_round=1000)  
 preds = bst.predict(X\_val)  
 print('Fold RMSE:', mean\_squared\_error(y\_val, preds, squared=False))  
 models.append(bst)  
 return models

Inference: use recent lags to forecast next month(s). For multi-step forecasting, either re-generate lags iteratively or train direct multi-output models.

## Model Persistence & Registry

* Save models with joblib or pickle to a model directory, and store metadata in model\_registry table.
* For production-grade, use MLflow: register models, track experiments, and serve models.

# app/utils/persistence.py  
import joblib  
from pathlib import Path  
  
def save\_model(obj, name, version='v1'):  
 path = Path('models') / f"{name}\_{version}.pkl"  
 path.parent.mkdir(parents=True, exist\_ok=True)  
 joblib.dump(obj, path)  
 return str(path)  
  
  
def load\_model(path):  
 return joblib.load(path)

## FastAPI App

Endpoints to include: - POST /train — triggers a training job (prophet or lgbm) - GET /predict?region=&territory=&area=&product\_id=&periods= — returns forecasts - GET /models — list available models and metadata - GET /health — health check

Sample main.py:

# app/main.py  
from fastapi import FastAPI  
from app.api.endpoints import router  
  
app = FastAPI(title='Sales Forecast API')  
app.include\_router(router)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 import uvicorn  
 uvicorn.run('app.main:app', host='0.0.0.0', port=8000, reload=True)

Sample endpoints (simplified):

# app/api/endpoints.py  
from fastapi import APIRouter, BackgroundTasks  
from app.services.etl import load\_monthly\_sales, create\_lags  
from app.services.tree\_train import train\_lgb  
from app.services.prophet\_train import train\_prophet, forecast\_prophet  
from app.utils.persistence import save\_model, load\_model  
from sqlalchemy import create\_engine  
  
router = APIRouter()  
  
@router.get('/health')  
async def health():  
 return {'status':'ok'}  
  
@router.post('/train')  
async def train(model\_type: str = 'lgbm', background\_tasks: BackgroundTasks = None):  
 # Start background training task or run sync  
 # For simplicity run sync here (but prefer background tasks or external scheduler)  
 engine = create\_engine('mssql+pyodbc://...')  
 df = load\_monthly\_sales(engine)  
 df = create\_lags(df, ['region','territory','area','product\_id'])  
 if model\_type == 'prophet':  
 # example train for single series  
 sub = df[(df.region=='North') & (df.product\_id==1)][['ds','total\_sales']].rename(columns={'total\_sales':'y'})  
 m = train\_prophet(sub)  
 path = save\_model(m, 'prophet\_north\_p1')  
 return {'model\_path': path}  
 else:  
 features = ['month','year','lag\_1','lag\_3','rolling\_3','region','territory']  
 models = train\_lgb(df.dropna(), features)  
 path = save\_model(models[-1], 'lgbm\_global')  
 return {'model\_path': path}  
  
@router.get('/predict')  
async def predict(region: str, territory: str = None, periods: int = 3):  
 # Load latest model and run inference  
 model = load\_model('models/lgbm\_global\_v1.pkl')  
 # build feature row for last known month and generate predictions  
 return {'status':'ok', 'predictions': []}

**Important**: For long-running training, use BackgroundTasks, Celery, or an external scheduler. Avoid running heavy training inside the main FastAPI process.

## Evaluation & Metrics

* Use time-series cross-validation (rolling origin) for robust evaluation.
* Metrics: RMSE, MAE, MAPE (with caution when near-zero values).
* Keep baseline models (last-year same month, naive average) for benchmarking.

## Containerization & Docker

Sample Dockerfile (simplified):

FROM python:3.10-slim  
WORKDIR /app  
COPY requirements.txt ./  
RUN pip install --no-cache-dir -r requirements.txt  
COPY ./app ./app  
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

docker-compose.yml should include your SQL Server container (mcr.microsoft.com/mssql/server) and the app service.

## Scheduling & Retraining

* Automate ETL + training using Airflow, Prefect, or simple cron.
* Retrain cadence: monthly/weekly depending on business volatility.
* Implement model drift monitoring: track performance on holdout set and raise alerts.

## Monitoring & Observability

* Store predictions and actuals in forecast\_results table for backtesting.
* Monitor prediction errors per territory/region.
* Log model versions used for each prediction for traceability.

## Example: Multi-step forecasting strategy (practical)

1. **One-step direct**: Train model to predict next-month sales (works well, simpler). For forecasting multiple months, iteratively predict month+1, append to features, then predict month+2.
2. **Direct multi-output**: Train models that predict 3/6 month horizons directly using appropriate targets.

## Quick Tips & Best Practices

* Always use time-based validation, not random train/test split.
* Keep deterministic preprocessing pipelines (use scikit-learn Pipeline if possible).
* Make categorical encodings consistent between training and inference (store encoders).
* When using Prophet with many series, parallelize training using multiprocessing.
* Use LightGBM with categorical features properly (pass categorical\_feature).

## Next Steps I can do for you

1. Create a runnable Git repo with the above structure and example data.
2. Implement the FastAPI endpoints fully (train, predict) with working SQL Server connection string template.
3. Build a notebook that trains Prophet and LightGBM on a sample of your data (you can upload a CSV).
4. Add MLflow tracking and a simple Streamlit dashboard.

Tell me which of the above you’d like me to build next — I can scaffold the code and provide runnable examples.

*End of document.*