# GitHub Copilot Prompt: EV Battery Digital Twin System

## Project Overview

Build a production-grade Electric Vehicle Battery Digital Twin system with:

- Real-time telemetry simulation
- AI-powered RUL (Remaining Useful Life) prediction using XGBoost
- Time-series data storage with TimescaleDB
- Message streaming via Kafka (Redpanda) and MQTT
- Live monitoring with Prometheus and Grafana
- MLflow for model tracking
- 3D battery visualization

## Architecture Requirements

### Tech Stack

- **Language**: Python 3.12+
- **ML Framework**: XGBoost, scikit-learn, NumPy, Pandas
- **Database**: TimescaleDB (PostgreSQL 15)
- **Message Brokers**: Redpanda (Kafka), Eclipse Mosquitto (MQTT)
- **Monitoring**: Prometheus, Grafana
- **ML Ops**: MLflow, MinIO (S3-compatible)
- **Containers**: Docker Compose

### Project Structure

```
ev-battery-digital-twin/
├── src/
│   ├── __init__.py
│   ├── simulator/
│   │   ├── __init__.py
│   │   └── publisher.py        # Telemetry generator
│   ├── inference/
│   │   ├── __init__.py
│   │   └── live_predictor.py   # Real-time ML predictions
│   ├── models/
│   │   ├── __init__.py
```

```
│   │   └── train.py            # XGBoost training pipeline
│   └── common/
│       ├── __init__.py
│       └── utils.py            # Shared utilities
├── models/                     # Trained model artifacts
├── datasets/                   # Training data (Kaggle dataset)
├── notebooks/                  # Jupyter notebooks
├── infra/
│   ├── grafana/provisioning/
│   │   ├── datasources/
│   │   │   └── datasource.yml
│   │   └── dashboards/
│   │       ├── dashboard.yml
│   │       └── battery_dashboard.json
│   ├── prometheus/
│   │   └── prometheus.yml
│   ├── mosquitto/
│   │   └── mosquitto.conf
│   └── create_telemetry_table.sql
├── tests/                      # Unit tests
├── docker-compose.yml
├── requirements.txt
├── .env.example
└── README.md
```

---

## Implementation Instructions

### 1. Docker Compose Infrastructure

Create `docker-compose.yml` with these services:

**TimescaleDB** (port 5432):

- Image: `timescale/timescaledb:latest-pg15`
- Environment: POSTGRES_USER=twin, POSTGRES_PASSWORD=twin_pass, POSTGRES_DB=twin_data
- Volume: Init script at `/docker-entrypoint-initdb.d/init.sql`

**Redpanda** (ports 9092, 29092):

- Image: `docker.redpanda.com/redpandadata/redpanda:v24.2.4`
- Kafka-compatible message broker
- Topic: `ev-telemetry`

**MinIO** (ports 9000, 9001):

- Image: `minio/minio:latest`
- S3-compatible storage for ML artifacts

**MLflow** (port 5000):

- Image: `ghcr.io/mlflow/mlflow:v2.9.2`
- Backend: SQLite
- Artifact store: MinIO S3

**Mosquitto** (ports 1883, 9002):

- Image: `eclipse-mosquitto:2.0`
- MQTT broker for IoT devices

**Prometheus** (port 9090):

- Image: `prom/prometheus:latest`
- Scrapes metrics from predictor service

**Grafana** (port 3000):

- Image: `grafana/grafana:latest`
- Default credentials: admin/admin
- Auto-provision TimescaleDB datasource

Network: `twin-net` (bridge)

---

## 2. Database Schema (`infra/create_telemetry_table.sql`)

Create TimescaleDB hypertable:

```sql
```

```sql
CREATE EXTENSION IF NOT EXISTS timescaledb;

CREATE TABLE IF NOT EXISTS ev_telemetry (
    time TIMESTAMPTZ NOT NULL,
    battery_id VARCHAR(50) NOT NULL,
    soc DOUBLE PRECISION,
    soh DOUBLE PRECISION,
    voltage DOUBLE PRECISION,
    current DOUBLE PRECISION,
    temperature DOUBLE PRECISION,
    charge_cycles INTEGER,
    power_consumption DOUBLE PRECISION,
    rul_prediction DOUBLE PRECISION,
    failure_probability DOUBLE PRECISION,
    PRIMARY KEY (time, battery_id)
);

SELECT create_hypertable('ev_telemetry', 'time', if_not_exists => TRUE);
CREATE INDEX idx_battery_id ON ev_telemetry(battery_id, time DESC);
SELECT add_retention_policy('ev_telemetry', INTERVAL '90 days', if_not_exists => TRUE);
```

---

### 3. ML Model Training (`src/models/train.py`)

Implement XGBoost training pipeline:

**Features** (7 inputs):

1. SoC (State of Charge) %
2. SoH (State of Health) %
3. Battery_Voltage (V)
4. Battery_Current (A)
5. Battery_Temperature (°C)
6. Charge_Cycles (count)
7. Power_Consumption (W)

**Target Variables**:

- RUL (Remaining Useful Life) - Regression
- Failure_Probability - Binary Classification

**Pipeline**:

1. Load dataset from `datasets/EV_Predictive_Maintenance_Dataset_15min.csv`
2. Split data (80/20 train/test)
3. StandardScaler for feature normalization
4. Train XGBRegressor for RUL (target $R^2 > 0.90$)
5. Train XGBClassifier for failure probability
6. Log experiments to MLflow
7. Save models using joblib:
   - `rul_xgb_model.joblib`
   - `failure_xgb_model.joblib`
   - `scaler.joblib`
   - `feature_names.joblib`

**Hyperparameters**:

- n_estimators: 200
- max_depth: 8 (RUL), 6 (Failure)
- learning_rate: 0.1
- subsample: 0.8
- colsample_bytree: 0.8

---

## 4. Telemetry Simulator (`src/simulator/publisher.py`)

Implement realistic EV battery simulation:

**Behavior**:

- Discharge: SoC decreases 0.2-0.8% per cycle
- Charge: SoC increases 0.5-1.5% per cycle
- Switch to charging at SoC < 20%
- Complete charge at SoC > 99%
- SoH degrades 0.01% per cycle
- Temperature rises with current draw

**Publishing**:

1. PostgreSQL/TimescaleDB - Store telemetry

2. Kafka topic `ev-telemetry` - Stream processing
3. MQTT topic `ev/battery/telemetry` - IoT devices

**Interval**: Every 2 seconds (configurable)

**Output format** (JSON):

```json
{
  "timestamp": "2025-10-20T10:30:15.123Z",
  "battery_id": "BATTERY_001",
  "soc": 85.5,
  "soh": 95.2,
  "voltage": 380.5,
  "current": 105.3,
  "temperature": 32.1,
  "charge_cycles": 150,
  "power_consumption": 40.1
}
```

---

### 5. Live Prediction Service (`src/inference/live_predictor.py`)

Implement real-time ML inference:

**Workflow**:

1. Load trained models from `models/` directory
2. Every 5 seconds:
   - Fetch latest telemetry from TimescaleDB
   - Scale features using saved scaler
   - Predict RUL and failure probability
   - Update database with predictions
   - Expose Prometheus metrics

**Prometheus Metrics** (port 9100):

- `battery_rul_prediction` (Gauge)
- `battery_failure_probability` (Gauge)
- `battery_soc` (Gauge)
- `battery_soh` (Gauge)

- `battery_temperature` (Gauge)
- `predictions_total` (Counter)

**Dependencies**:

- joblib for model loading
- psycopg2 for database
- prometheus_client for metrics

---

### 6. Prometheus Configuration (`infra/prometheus/prometheus.yml`)

```yaml
global:
  scrape_interval: 5s

scrape_configs:
  - job_name: 'ev_predictor'
    static_configs:
      - targets: ['host.docker.internal:9100']
```

---

### 7. Grafana Dashboard Configuration

Create datasource (`infra/grafana/provisioning/datasources/datasource.yml`):

```yaml
apiVersion: 1
datasources:
  - name: TimescaleDB
    type: postgres
    url: timescaledb:5432
    database: twin_data
    user: twin
    secureJsonData:
      password: twin_pass
    jsonData:
      sslmode: disable
      timescaledb: true
```

Create dashboard with panels:

1. **SoC Time Series** - Line chart

2. **SoH Trend** - Line chart with threshold
3. **Temperature Heatmap** - Color-coded
4. **RUL Gauge** - Current prediction
5. **Failure Risk** - Percentage gauge
6. **Charge Cycles Counter** - Stat panel
7. **Voltage/Current Graph** - Dual-axis
8. **Power Consumption** - Area chart

Auto-refresh: 5 seconds

---

## 8. Dependencies (`requirements.txt`)

```
# Core ML & Data Science
xgboost>=2.0.0
numpy>=1.24.0
pandas>=2.0.0
scikit-learn>=1.3.0
joblib>=1.3.0

# ML Ops
mlflow>=2.9.0
boto3>=1.28.0

# Database
psycopg2-binary>=2.9.0

# Messaging
kafka-python>=2.0.0
paho-mqtt>=1.6.0

# Monitoring
prometheus-client>=0.18.0

# Web Framework
flask>=3.0.0
flask-cors>=4.0.0

# Utilities
python-dotenv>=1.0.0
pyyaml>=6.0
requests>=2.31.0
```

```
# Development
jupyter>=1.0.0
matplotlib>=3.7.0
seaborn>=0.12.0
```

---

## 9. Environment Configuration (`.env.example`)

```bash
# Database
PG_HOST=localhost
PG_PORT=5432
PG_USER=twin
PG_PASSWORD=twin_pass
PG_DATABASE=twin_data

# Kafka
KAFKA_BOOTSTRAP_SERVERS=localhost:9092
KAFKA_TOPIC=ev-telemetry

# MQTT
MQTT_HOST=localhost
MQTT_PORT=1883
MQTT_TOPIC=ev/battery/telemetry

# MLflow
MLFLOW_TRACKING_URI=http://localhost:5000
MLFLOW_S3_ENDPOINT_URL=http://localhost:9000
AWS_ACCESS_KEY_ID=minioadmin
AWS_SECRET_ACCESS_KEY=minioadmin

# Model paths
MODEL_DIR=models
```

---

## 10. Utility Functions (`src/common/utils.py`)

Implement helper functions:

```python
```

```python
def get_db_connection():
    """Create PostgreSQL connection with retry logic"""
    pass

def create_kafka_producer():
    """Initialize Kafka producer with error handling"""
    pass

def create_mqtt_client():
    """Setup MQTT client with connection callbacks"""
    pass

def load_models(model_dir):
    """Load all trained models and scaler"""
    pass

def validate_telemetry(data):
    """Validate telemetry data structure"""
    pass

def setup_logging(name, log_file='logs/app.log'):
    """Configure structured logging"""
    pass
```

---

## 11. Testing (`tests/test_simulator.py`, `tests/test_predictor.py`)

Create unit tests:

- Test telemetry generation logic
- Test database insertion
- Test model prediction accuracy
- Test metric exposition
- Mock external services

---

## 12. Documentation (`README.md`)

Include:

- Architecture diagram
- Quick start guide

- Service endpoints table
- Model performance metrics
- Troubleshooting section
- API documentation
- Development setup

---

## Execution Flow

### Setup & Deployment

```bash
# 1. Clone and setup
git clone <repo>
cd ev-battery-digital-twin
python3.12 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt

# 2. Download dataset from Kaggle
# Place in: datasets/EV_Predictive_Maintenance_Dataset_15min.csv

# 3. Start infrastructure
docker-compose up -d

# 4. Wait for services (30 seconds)
sleep 30

# 5. Train models
python src/models/train.py

# 6. Start simulator (Terminal 1)
python src/simulator/publisher.py

# 7. Start predictor (Terminal 2)
python src/inference/live_predictor.py

# 8. Access dashboards
# Grafana: http://localhost:3000 (admin/admin)
# MLflow: http://localhost:5000
# Prometheus: http://localhost:9090
# MinIO: http://localhost:9001
```

---

## Advanced Features to Implement

### 1. REST API (`src/api/server.py`)

Flask application with endpoints:

- `GET /api/telemetry/latest` - Latest battery data
- `GET /api/telemetry/history?minutes=60` - Historical data
- `GET /api/predictions/current` - Current RUL and failure risk
- `GET /api/health` - Service health check
- `POST /api/battery/configure` - Update simulation parameters

### 2. WebSocket Real-time Stream

Live data streaming to web clients using Flask-SocketIO

### 3. Anomaly Detection

Implement Isolation Forest for detecting abnormal patterns in telemetry

### 4. Alert System

Trigger alerts when:

- SoC < 15% (critical low battery)
- Temperature > 60°C (overheating)
- Failure probability > 0.7 (high risk)
- RUL < 100 cycles (maintenance needed)

Send notifications via:

- Email (SMTP)
- Slack webhook
- Database log table

### 5. Multi-Battery Fleet Management

Extend to handle multiple batteries:

- Array of battery_ids
- Fleet-level dashboard
- Comparative analytics

### 6. Model Retraining Pipeline

Scheduled retraining:

- Collect new telemetry data
- Retrain models monthly
- Compare performance metrics
- Auto-deploy if improved

### 7. Data Export Service

Export telemetry to:

- CSV files
- Parquet format
- Cloud storage (S3, Azure Blob)

### 8. Load Testing

Simulate 100+ batteries with stress testing tools

---

## Code Quality Requirements

### Style Guide

- Follow PEP 8
- Type hints for all functions
- Docstrings (Google style)
- Maximum line length: 100 characters

### Error Handling

- Try-except blocks for all I/O operations
- Custom exception classes
- Graceful degradation
- Detailed error logging

### Logging

- Structured JSON logs
- Log levels: DEBUG, INFO, WARNING, ERROR, CRITICAL

- Separate log files per service
- Log rotation (max 10MB per file)

### Security

- No hardcoded credentials (use environment variables)
- SQL injection prevention (parameterized queries)
- Input validation on all external data
- Rate limiting on API endpoints

---

## Performance Targets

- **Telemetry Generation**: 30+ records/minute
- **Prediction Latency**: < 100ms per inference
- **Database Write**: < 50ms per insert
- **Model Loading**: < 3 seconds on startup
- **Memory Usage**: < 500MB per service
- **API Response Time**: < 200ms (p95)

---

## Monitoring & Observability

### Metrics to Track

1. **System Metrics**:
   - CPU usage per service
   - Memory consumption
   - Disk I/O
   - Network throughput
2. **Application Metrics**:
   - Telemetry generation rate
   - Prediction frequency
   - Model inference time
   - Database query duration
   - Message queue lag

3. **Business Metrics**:
   - Average battery SoH
   - Predicted RUL distribution
   - Failure risk trends
   - Charge cycle statistics

### Grafana Alerts

Configure alert rules:

- High failure probability (> 0.8)
- Low SoH (< 75%)
- Service downtime
- Database connection failures

---

## Deployment Considerations

### Production Readiness

1. **Kubernetes Migration**:
   - Convert Docker Compose to K8s manifests
   - Use StatefulSets for databases
   - Implement HorizontalPodAutoscaler
2. **High Availability**:
   - TimescaleDB replication
   - Kafka cluster (3+ nodes)
   - Predictor service replicas
3. **Security Hardening**:
   - TLS/SSL for all connections
   - Secrets management (HashiCorp Vault)
   - Network policies
   - RBAC for services
4. **CI/CD Pipeline**:
   - GitHub Actions / GitLab CI
   - Automated testing

- Docker image building
- Staging environment

5. **Backup Strategy**:
    - Daily database backups
    - Model artifact versioning
    - Configuration backups
    - Disaster recovery plan

---

## Dataset Information

**Source**: https://www.kaggle.com/datasets/datasetengineer/eviot-predictivemaint-dataset

**Expected Columns**:

- SoC (State of Charge)
- SoH (State of Health)
- Battery_Voltage
- Battery_Current
- Battery_Temperature
- Charge_Cycles
- Power_Consumption
- RUL (Remaining Useful Life) - Target
- Failure_Probability - Target

**Preprocessing**:

- Handle missing values (forward fill for time series)
- Remove outliers (IQR method)
- Feature scaling (StandardScaler)
- No data augmentation needed (large dataset)

---

## Expected Output

After implementation, the system should:

1. ✅ Generate realistic battery telemetry every 2 seconds
2. ✅ Store time-series data in TimescaleDB
3. ✅ Stream data via Kafka and MQTT
4. ✅ Predict RUL with 90%+ accuracy ($R^2 > 0.90$)
5. ✅ Classify failure probability
6. ✅ Update predictions every 5 seconds
7. ✅ Expose Prometheus metrics
8. ✅ Display live Grafana dashboards
9. ✅ Track experiments in MLflow
10. ✅ Run containerized services

**Visual Outputs**:

- Real-time line charts of SoC, SoH, Temperature
- RUL gauge showing remaining cycles
- Failure risk percentage
- 3D battery visualization (bonus)
- Alert notifications

---

## Additional Resources

### Configuration Files to Create

**Mosquitto Config** (`infra/mosquitto/mosquitto.conf`):

```
listener 1883
allow_anonymous true
listener 9002
protocol websockets
```

**Grafana Dashboard JSON** (`infra/grafana/provisioning/dashboards/battery_dashboard.json`): Create panels for:

- Time series: SoC, SoH, Temperature, Voltage, Current
- Gauge: RUL, Failure Probability
- Stat: Charge Cycles, Power Consumption

---

## Success Criteria

The implementation is complete when:

1. ✅ All 7 Docker services start successfully
2. ✅ Models train with $R^2 > 0.90$ for RUL
3. ✅ Simulator publishes to all 3 channels (DB, Kafka, MQTT)
4. ✅ Predictor makes predictions every 5 seconds
5. ✅ Grafana displays 8+ panels with live data
6. ✅ Prometheus scrapes metrics from predictor
7. ✅ MLflow tracks training experiments
8. ✅ System runs continuously without crashes
9. ✅ Documentation is complete
10. ✅ Code passes linting (pylint, mypy)

---

## Copilot-Specific Instructions

When implementing this project with GitHub Copilot:

1. **Start with infrastructure**: Generate docker-compose.yml first
2. **Database schema next**: Create SQL initialization script
3. **Training pipeline**: Implement model training with MLflow integration
4. **Simulator**: Build telemetry generator with realistic physics
5. **Predictor**: Create inference service with Prometheus metrics
6. **Configuration files**: Generate Grafana, Prometheus configs
7. **Documentation**: Auto-generate README with architecture diagrams
8. **Tests**: Create unit tests for all modules

**Use these Copilot comments in your code**:

```python
```

```
# TODO: Implement EV battery telemetry simulator with charge/discharge cycles
# TODO: Train XGBoost model for RUL prediction with R² > 0.90
# TODO: Create TimescaleDB connection with retry logic and connection pooling
# TODO: Publish telemetry to Kafka topic 'ev-telemetry' with error handling
# TODO: Expose Prometheus metrics on port 9100 for monitoring
```

---

## Final Notes

This is a **production-grade** system designed for:

- Real-world EV battery monitoring
- Predictive maintenance applications
- IoT data pipeline demonstrations
- ML Ops best practices
- Time-series database optimization

The architecture is **scalable** and **maintainable**, following industry standards for observability, reliability, and performance.

**Estimated Development Time**: 2-3 days with Copilot assistance

**License**: MIT (or your choice) **Author**: Your Name **Last Updated**: October 20, 2025