# Digital Twin Architecture Documentation

## System Overview

This document explains the complete Digital Twin architecture for Robot Reinforcement Learning in Logistics, detailing each component's role and how they interact.

---

## Architecture Diagram

```
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■                    DIGITAL TWIN ECOSYSTEM                        ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

    ■■■■■■■■■■■■■■■■■■■■                    ■■■■■■■■■■■■■■■■■■■■
    ■  NVIDIA Isaac    ■                    ■  Reinforcement   ■
    ■  Simulation      ■■■■■■■■■■■■■■■■■■■■■■  Learning Policy ■
    ■  (Future)        ■                    ■  (Future)        ■
    ■■■■■■■■■■■■■■■■■■■■                    ■■■■■■■■■■■■■■■■■■■■
             ■                                       ■
             ■ Publishes State                      ■ Reads State
             ■ Updates                              ■ Sends Actions
             ■                                       ■
             ▼                                       ▼
    ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    ■          FIWARE ORION CONTEXT BROKER (NGSI-LD)              ■
    ■                    Port: 1026                                ■
    ■         Central Hub for Real-time Entity Management          ■
    ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
                   ■                              ■
         Stores    ■                              ■ Queries
         Entities  ■                              ■ Real-time Data
                   ■                              ■
    ■■■■■■■■■■■■■■■▼■■■■■■■■■■    ■■■■■■■■■■■■■■■▼■■■■■■■■■■■■■■■
    ■    Digital Twin         ■    ■   Monitoring Dashboard      ■
    ■    Entities (JSON-LD)   ■    ■   (Web Interface)           ■
    ■                         ■    ■   Port: 8081                ■
    ■  • Robot:001           ■    ■                             ■
    ■  • Environment:001     ■    ■  • Real-time Display        ■
    ■  • Episode:001         ■    ■  • Battery Chart            ■
    ■■■■■■■■■■■■■■■■■■■■■■■■■■■    ■  • Auto-refresh (5s)        ■
                                   ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
                   ■
                   ■ Notifications
                   ■ on Changes
                   ■
    ■■■■■■■■■■■■■■■▼■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    ■         Telemetry Service (Flask API)                       ■
    ■                 Port: 8668                                  ■
    ■         Receives & Processes Notifications                  ■
    ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
                   ■
```

```
                    ■ Stores Historical
                    ■ Data with Timestamps
                    ▼
     ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
     ■            TimescaleDB (PostgreSQL)                          ■
     ■                  Port: 5432                                  ■
     ■       Time-series Optimized Historical Storage               ■
     ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
```

---

## Component Details

### *1. FIWARE Orion Context Broker (NGSI-LD)*

**Role:** Central nervous system of the digital twin architecture

**Technology:**

• FIWARE Orion-LD v1.5.1
• NGSI-LD (Next Generation Service Interface - Linked Data)
• MongoDB 4.4 backend

**Responsibilities:**

**Entity Management**: Stores and manages digital twin entities

- Robot entities (position, velocity, battery, sensors)

- Environment entities (temperature, lighting, obstacles)

- Episode entities (training metrics, rewards, steps)

**Real-time Updates**: Provides instant access to current state

- RESTful API endpoints for CRUD operations

- NGSI-LD format ensures semantic interoperability

- JSON-LD context for standardized vocabulary

**Subscription System**: Notifies subscribers when entities change

- Publishes notifications to telemetry service

- Supports attribute-level filtering

- Enables event-driven architecture

**CORS Support**: Allows dashboard to fetch data
- Configured with `-corsOrigin __ALL`

- Enables cross-origin requests from web browser

**Port:** 1026

**Key Endpoints:**

• `GET /version` - Check Orion version and status
• `POST /ngsi-ld/v1/entities` - Create new entity
• `GET /ngsi-ld/v1/entities/{id}` - Retrieve entity
• `PATCH /ngsi-ld/v1/entities/{id}/attrs` - Update attributes
• `POST /v2/subscriptions` - Create subscription for notifications

**Data Flow:**

```
Isaac Sim → Updates Robot State → Orion Context Broker
RL Policy → Reads State from Orion → Computes Action → Updates Orion
Dashboard → Queries Orion → Displays Real-time Data
```

---

## *2. Digital Twin Entities (JSON-LD)*

**Role:** Semantic representation of physical and virtual components

**Technology:**

• NGSI-LD data model
• JSON-LD with custom context
• Standardized vocabulary

**Entity Types:**

#### Robot Entity (`urn:ngsi-ld:Robot:001`)

**Purpose:** Digital representation of the physical/simulated robot

**Attributes:**

• `position` (GeoProperty): 3D coordinates [x, y, z]
• `velocity` (Property): Speed vector {vx, vy, vz}
• `orientation` (Property): Quaternion {x, y, z, w}
• `battery` (Property): Charge level (0-100%)
• `status` (Property): Operational state (idle, moving, charging, error)
• `sensorData` (Property):
- IMU: Acceleration and gyroscope readings

- Lidar: Point cloud data

- Camera: Image status

**Use Case:** Isaac Sim publishes robot state updates; RL policy reads state to decide actions

#### Environment Entity (`urn:ngsi-ld:Environment:001`)

**Purpose:** Represents simulation environment conditions

**Attributes:**

• `temperature` (Property): Ambient temperature
• `lighting` (Property): Light configuration
• `obstacles` (Property): List of obstacles in scene
• `dimensions` (Property): Environment size

**Use Case:** Defines training environment parameters for RL agent

#### Episode Entity (`urn:ngsi-ld:Episode:001`)

**Purpose:** Tracks reinforcement learning training progress

**Attributes:**

• `episodeNumber` (Property): Current training episode
• `reward` (Property): Accumulated reward
• `steps` (Property): Number of timesteps
• `metrics` (Property): Performance indicators
- Success rate

- Collision count

- Distance traveled

**Use Case:** Monitors training progress and performance metrics

**JSON-LD Context:**

```
{
  "@context": [
    "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld",
    {
      "Robot": "https://example.org/digitaltwin/Robot",
      "position": "https://example.org/digitaltwin/position",
      "velocity": "https://example.org/digitaltwin/velocity"
    }
  ]
}
```

---

## *3. Telemetry Service (Custom Flask Application)*

**Role:** Historical data collection and storage bridge

**Technology:**

• Python Flask web framework
• Flask-CORS for cross-origin support
• psycopg2 for PostgreSQL connection

**Responsibilities:**

**Notification Receiver**: Listens for Context Broker notifications

- Endpoint: `POST /v2/notify`
- Receives entity change notifications via subscriptions
- Processes NGSI-LD attribute updates

**Data Transformation**: Converts real-time data to time-series format

- Extracts entity ID, type, attribute name, and value
- Adds timestamp for temporal ordering
- Stores metadata (dateCreated, dateModified)

**Database Writer**: Persists data to TimescaleDB

- Batch inserts for efficiency
- Uses hypertable for time-series optimization
- Indexes on entity_id and time for fast queries

**Query API**: Provides historical data access

- Endpoint: `GET /v2/entities/{id}?lastN=X`
- Returns last N records for an entity
- Supports time-range filtering

**Health Monitoring**: Status endpoint for system checks
- Endpoint: `GET /health`
- Returns database connection status

**Port:** 8668

**Docker Configuration:**

```
telemetry-service:
  build: ./telemetry_store
  ports: ["8668:8668"]
  environment:
    - DB_HOST=timescaledb
    - DB_PORT=5432
```

**Subscription Example:**

```
{
  "description": "Notify Telemetry Store of Robot changes",
  "subject": {
    "entities": [{ "idPattern": ".*", "type": "Robot" }],
    "condition": { "attrs": ["battery", "position", "status"] }
  },
  "notification": {
    "http": { "url": "http://telemetry-service:8668/v2/notify" },
    "attrs": ["battery", "position", "status"]
  }
}
```

**Data Flow:**

```
Context Broker → Subscription Triggers → POST /v2/notify →
Extract Attributes → Format for TimescaleDB → INSERT INTO telemetry_data
```

---

## 4. TimescaleDB (Time-series Database)

**Role:** Long-term storage for historical telemetry data

**Technology:**

• PostgreSQL 15 with TimescaleDB extension
• Hypertable optimization for time-series data
• Automatic partitioning by time

**Responsibilities:**

**Time-series Storage**: Efficiently stores timestamped data

- Automatic chunking by time intervals
- Compression for older data
- Fast time-range queries

**Data Retention**: Manages historical data lifecycle

- Configurable retention policies
- Automatic data aggregation
- Space optimization

**Analytics Support**: Enables historical analysis
- Aggregate functions (AVG, MIN, MAX, STDDEV)
- Time-bucket queries
- Trend analysis

**Port:** 5432

**Database Schema:**

```
CREATE TABLE telemetry_data (
    id SERIAL,
    time TIMESTAMPTZ NOT NULL,          -- When data was recorded
    entity_id TEXT NOT NULL,            -- Which entity (Robot001)
    entity_type TEXT NOT NULL,          -- Type of entity (Robot)
    attribute_name TEXT NOT NULL,       -- What changed (battery)
    attribute_value JSONB NOT NULL,     -- New value (85)
    metadata JSONB,                     -- Additional context
    PRIMARY KEY (id, time)
);

-- Convert to hypertable for time-series optimization
SELECT create_hypertable('telemetry_data', 'time');

-- Indexes for fast queries
CREATE INDEX idx_entity_id ON telemetry_data (entity_id, time DESC);
CREATE INDEX idx_entity_type ON telemetry_data (entity_type, time DESC);
```

**Example Query:**

```
-- Get last 10 battery readings for Robot001
SELECT time, attribute_value
FROM telemetry_data
WHERE entity_id = 'Robot001'
  AND attribute_name = 'battery'
ORDER BY time DESC
LIMIT 10;
```

**Benefits:**

• 10-100x faster than regular PostgreSQL for time-series queries
• Automatic data compression
• Native support for time-based partitioning
• SQL compatibility

---

## *5. Monitoring Dashboard (Web Interface)*

**Role:** Real-time visualization and monitoring interface

**Technology:**

• Pure HTML/CSS/JavaScript
• Chart.js for data visualization
• Fetch API for REST calls
• Responsive design with CSS Grid

**Features:**

**Real-time Display**:

- Robot status (name, operational state)

- Battery level with visual bar indicator

- 3D position coordinates (X, Y, Z)

- Velocity vector (vX, vY, vZ)

- Sensor readings (IMU, Lidar, Camera)

**Training Metrics**:

- Current episode number

- Accumulated reward

- Step count

- Performance indicators

**Environment Info**:

- Temperature

- Lighting configuration

- Environment dimensions

**Historical Charts**:

- Battery level over time (Chart.js)

- Last 20 data points from telemetry

- Interactive visualization

**Auto-refresh**:
- Updates every 5 seconds

- Checks service health status

- Non-blocking async calls

**Port:** 8081 (HTTP server)

**API Integration:**

```
// Fetch robot data from Context Broker
const robot = await fetch(
  "http://localhost:1026/ngsi-ld/v1/entities/urn:ngsi-ld:Robot:001",
  { headers: { Accept: "application/ld+json" } }
);

// Fetch battery history from Telemetry Service
const history = await fetch(
  "http://localhost:8668/v2/entities/Robot001?lastN=20"
```

```
    );
```

**CORS Configuration:**

• Orion: Returns `Access-Control-Allow-Origin: *`
• Telemetry Service: Uses flask-cors library
• Enables browser to fetch from different ports

**User Interface:**

• Purple gradient background
• Card-based layout
• Color-coded status badges
• Responsive grid system
• Battery indicator (green/yellow/red)

---

## Data Flow Scenarios

### *Scenario 1: Robot State Update*

```
1. Isaac Sim detects robot moved to new position
   ■■> Publishes update to Orion Context Broker

2. Orion receives PATCH request
   ■■> Updates Robot:001 entity attributes
   ■■> Stores in MongoDB

3. Orion checks active subscriptions
   ■■> Finds telemetry subscription for Robot entities
   ■■> Sends POST /v2/notify to telemetry-service:8668

4. Telemetry Service receives notification
   ■■> Extracts entity_id, attributes, timestamp
   ■■> Inserts into TimescaleDB telemetry_data table

5. Dashboard auto-refresh triggers (every 5s)
   ■■> Fetches latest state from Orion
   ■■> Fetches battery history from Telemetry Service
   ■■> Updates UI with new data
```

### *Scenario 2: Training Episode Completion*

```
1. RL Policy completes episode
   ■■> Calculates final reward and metrics
   ■■> Updates Episode:001 entity in Orion

2. Orion notifies Telemetry Service
   ■■> Episode metrics stored in TimescaleDB

3. Dashboard displays
   ■■> Current episode number
   ■■> Total reward
   ■■> Success rate from metrics

4. Data available for analysis
   ■■> Query historical episode performance
   ■■> Track learning progress over time
```

### *Scenario 3: Historical Data Query*

```
1. User wants to analyze battery usage patterns
   ■■> Opens dashboard battery chart

2. Dashboard queries Telemetry Service
   ■■> GET /v2/entities/Robot001?lastN=20

3. Telemetry Service queries TimescaleDB
   ■■> SELECT * FROM telemetry_data
       WHERE entity_id='Robot001'
       AND attribute_name='battery'
       ORDER BY time DESC LIMIT 20

4. Returns time-series data
   ■■> Dashboard renders Chart.js line graph
   ■■> Shows battery level over time
```

---

## Docker Network Architecture

All services run in Docker containers within the `fiware-network`:

```
fiware-network (Bridge Network)
■
■■ fiware-mongo:27017
■   ■■> Backend for Orion Context Broker
■
■■ fiware-orion:1026
■   ■■> NGSI-LD API (public)
■   ■■> Connects to mongo-db
■
■■ telemetry-timescaledb:5432
■   ■■> PostgreSQL + TimescaleDB
■
■■ telemetry-service:8668
   ■■> Flask API (public)
   ■■> Connects to timescaledb
```

**Port Mappings:**

• `1026:1026` - Orion accessible from host
• `5432:5432` - TimescaleDB accessible from host (for admin)
• `8668:8668` - Telemetry Service accessible from host

**Container Communication:**

• Uses Docker DNS (container names as hostnames)
• No need for localhost or IP addresses
• Automatic service discovery

---

## Technology Stack Summary

| Component | Technology | Version | Purpose |
| ---------------- | ---------------- | ---------- | -------------------------- |

|  |  |  |  |
|---|---|---|---|
| Context Broker | FIWARE Orion-LD | 1.5.1 | Real-time entity management |
| Database (Orion) | MongoDB | 4.4 | Context Broker persistence |
| Time-series DB | TimescaleDB | latest-pg15 | Historical data storage |
| Telemetry Service | Python Flask | 3.0.0+ | Notification processing |
| CORS Support | flask-cors | 4.0.0+ | Cross-origin requests |
| Dashboard | HTML/CSS/JS | - | Web visualization |
| Charting | Chart.js | 3.9.1 | Data visualization |
| Containerization | Docker Compose | - | Service orchestration |
| Data Format | NGSI-LD / JSON-LD | - | Semantic interoperability |

---

## Key Design Decisions

### *1. Why NGSI-LD?*

• **Semantic Interoperability**: Linked Data ensures standardized vocabulary
• **FIWARE Ecosystem**: Integrates with many FIWARE components
• **Context Management**: Built for IoT and digital twin use cases
• **Subscription Model**: Event-driven architecture for real-time updates

### *2. Why Custom Telemetry Service (not QuantumLeap)?*

• **Simplicity**: Lightweight Flask app easier to customize
• **Control**: Full control over data format and storage
• **Debugging**: Easier to troubleshoot and add logging
• **Learning**: Educational value in understanding the flow

### *3. Why TimescaleDB?*

• **Performance**: 10-100x faster than PostgreSQL for time-series
• **SQL Compatible**: Use familiar SQL queries
• **Compression**: Automatic data compression saves space
• **Analytics**: Built-in functions for time-series analysis

### *4. Why Docker?*

• **Isolation**: Each service runs in its own container
• **Portability**: Works on Windows, Linux, macOS
• **Networking**: Automatic DNS and service discovery
• **Reproducibility**: Same environment everywhere

---

## System Requirements

**Hardware:**

• CPU: 4+ cores recommended
• RAM: 8GB minimum, 16GB recommended
• Disk: 10GB+ free space

**Software:**

• Docker Desktop with WSL2 (Windows) or Docker Engine (Linux)
• Python 3.12+ (for local development)

• Modern web browser (Chrome, Firefox, Edge)
• Git for version control

**Network:**

• Ports 1026, 5432, 8668, 8081 available
• Internet access for Docker image downloads

---

## Future Integration

### *NVIDIA Omniverse (Isaac Sim)*

**Role:** Physics-based robot simulation

**Integration Plan:**

Install Isaac Sim
Create Python connector script
Read robot state from simulation
Publish to Orion Context Broker via REST API
Subscribe to action commands from RL policy

**Example Code:**

```
import requests
from omni.isaac.core import World

world = World()
robot = world.scene.get_object("robot")

while True:
    # Get robot state
    position = robot.get_world_pose()[0]
    velocity = robot.get_linear_velocity()

    # Update Context Broker
    requests.patch(
        "http://localhost:1026/v2/entities/Robot001/attrs",
        json={
            "position": {"value": position.tolist(), "type": "Array"},
            "velocity": {"value": velocity.tolist(), "type": "Array"}
        }
    )
```

### *Reinforcement Learning Policy*

**Role:** AI agent learning optimal robot behavior

**Integration Plan:**

Implement RL algorithm (PPO, SAC, DQN)
Define observation space (from robot state)
Define action space (motor commands)
Read state from Context Broker
Compute action using trained model
Send action to Isaac Sim via Context Broker

**Example Code:**

```
import requests
import numpy as np
from stable_baselines3 import PPO
```

```
model = PPO.load("trained_model.zip")

while True:
    # Get current state
    response = requests.get(
        "http://localhost:1026/ngsi-ld/v1/entities/urn:ngsi-ld:Robot:001"
    )
    state = extract_observation(response.json())

    # Compute action
    action = model.predict(state)

    # Send action to simulation
    requests.patch(
        "http://localhost:1026/v2/entities/Robot001/attrs",
        json={"action": {"value": action.tolist(), "type": "Array"}}
    )
```

---

## Troubleshooting Guide

### *Common Issues*

#### 1. Context Broker not responding

```
# Check if container is running
docker ps | grep fiware-orion

# View logs
docker logs fiware-orion --tail 50

# Restart services
docker-compose restart orion
```

#### 2. Telemetry notifications not working

```
# Check subscription status
curl http://localhost:1026/v2/subscriptions

# Verify telemetry service is running
curl http://localhost:8668/health

# Check telemetry service logs
docker logs telemetry-service --tail 50
```

#### 3. Dashboard not showing data

```
# Check CORS headers
curl -H "Origin: http://localhost:8081" http://localhost:1026/version

# Verify data exists
curl http://localhost:1026/ngsi-ld/v1/entities/urn:ngsi-ld:Robot:001

# Check browser console for errors (F12)
```

#### 4. Database connection issues

```
# Check if TimescaleDB is running
docker ps | grep timescaledb
# Connect to database
```

```
docker exec -it telemetry-timescaledb psql -U postgres -d telemetry

# Check table contents
SELECT COUNT(*) FROM telemetry_data;
```

---

## Performance Considerations

**Context Broker:**

• MongoDB indexes on entity IDs
• Connection pooling
• Query optimization with attribute filters

**Telemetry Service:**

• Batch inserts to reduce database round-trips
• Async processing for high-throughput scenarios
• Connection pool to TimescaleDB

**TimescaleDB:**

• Hypertable automatically partitions by time
• Compression policies for old data
• Appropriate chunk interval (default: 7 days)

**Dashboard:**

• Debounced refresh (5 seconds)
• Efficient Chart.js rendering
• Limits on historical data points (last 20)

---

## Security Considerations

**Current Implementation:**

• ■■ No authentication on APIs (development only)
• ■■ CORS set to `*` (allows all origins)
• ■■ Database passwords in docker-compose (not encrypted)

**Production Recommendations:**

Enable authentication on Orion (OAuth2, API keys)
Use HTTPS/TLS for all communications
Restrict CORS to specific origins
Use Docker secrets for sensitive data
Implement API rate limiting
Add network segmentation
Regular security updates

---

## Monitoring & Maintenance

**Health Checks:**

```
# Orion
curl http://localhost:1026/version

# Telemetry Service
curl http://localhost:8668/health
```

```
# TimescaleDB
docker exec telemetry-timescaledb pg_isready -U postgres
```

**Backup Strategy:**

```
# Backup MongoDB (Orion data)
docker exec fiware-mongo mongodump --out /backup

# Backup TimescaleDB
docker exec telemetry-timescaledb pg_dump -U postgres telemetry > backup.sql
```

**Log Monitoring:**

```
# View all container logs
docker-compose logs -f

# Specific service
docker logs -f fiware-orion
docker logs -f telemetry-service
```

---

## Conclusion

This Digital Twin architecture provides a **robust, scalable, and extensible foundation** for robot reinforcement learning in logistics applications.

**Key Strengths:**

- ■ Real-time state management with NGSI-LD
- ■ Historical data retention with time-series optimization
- ■ Event-driven architecture via subscriptions
- ■ Web-based monitoring and visualization
- ■ Containerized for easy deployment
- ■ Standards-based (FIWARE, NGSI-LD)

**Ready for:**

- Integration with NVIDIA Isaac Sim
- Deployment of RL training pipelines
- Scalability to multiple robots
- Extension with additional FIWARE components

**Next Steps:**

Integrate Isaac Sim simulation
Implement RL training loop
Add authentication and security
Scale to multiple robot entities
Deploy to production environment

---

**Project Repository:** https://github.com/fedih/Digital-Twin-NVIDIA-Omniverse-Robot-RL-for-Logistics

**Documentation Date:** November 24, 2025