Module 8: Integrations & Pipelines

In this module, you will:

1. **Learn about data integration patterns** with TimescaleDB
2. **Set up a realistic data ingestion pipeline** from REST APIs
3. **Export and analyze data with Python & Pandas**
4. **Create visualizations** with Matplotlib
5. **Build a complete data pipeline** from API to insights
6. **Clean up**

What are Integrations & Pipelines?

Simple Explanation

Data Integration = Getting data INTO TimescaleDB from external sources
Data Pipeline = The complete flow: Collect → Store → Process → Analyze → Visualize

Common Integration Patterns

📥 **Data Sources (Ingestion):**

- REST APIs (weather, sensors, stock prices)
- File uploads (CSV, JSON)
- Direct database connections
- Scheduled data imports

📤 **Data Destinations (Export):**

- Python/Pandas for analysis
- CSV/JSON exports for other tools
- Machine learning pipelines
- Real-time dashboards

Step 1: Project Setup

Create Project Directory

```
mkdir timescale-integrations
cd timescale-integrations
```

Create docker-compose.yml

```
services:
  timescaledb:
    image: timescale/timescaledb:latest-pg16
    container_name: timescale-integrations
    environment:
      - POSTGRES_USER=admin
      - POSTGRES_PASSWORD=admin123
      - POSTGRES_DB=integrations_db
    ports:
      - "5432:5432"
    volumes:
      - timescale_data:/var/lib/postgresql/data
      - ./init-scripts:/docker-entrypoint-initdb.d

volumes:
  timescale_data:
```

Create Database Schema

Create `init-scripts/01-setup.sql`:

```
-- Enable TimescaleDB extension
CREATE EXTENSION IF NOT EXISTS timescaledb;

-- Weather data table
CREATE TABLE weather_data (
    time TIMESTAMPTZ NOT NULL,
    city TEXT NOT NULL,
    temperature DOUBLE PRECISION,
    humidity INTEGER,
    pressure DOUBLE PRECISION,
    wind_speed DOUBLE PRECISION,
    description TEXT,
    api_source TEXT DEFAULT 'openweather'
);

-- Create hypertable
SELECT create_hypertable('weather_data', 'time');

-- Stock price data table
CREATE TABLE stock_prices (
    time TIMESTAMPTZ NOT NULL,
```

```
    symbol TEXT NOT NULL,
    open_price DOUBLE PRECISION,
    high_price DOUBLE PRECISION,
    low_price DOUBLE PRECISION,
    close_price DOUBLE PRECISION,
    volume BIGINT,
    api_source TEXT DEFAULT 'alphavantage'
);

-- Create hypertable
SELECT create_hypertable('stock_prices', 'time');

-- Sensor data table (for simulated IoT)
CREATE TABLE sensor_readings (
    time TIMESTAMPTZ NOT NULL,
    device_id TEXT NOT NULL,
    sensor_type TEXT NOT NULL,
    value DOUBLE PRECISION,
    unit TEXT,
    location TEXT,
    metadata JSONB
);

-- Create hypertable
SELECT create_hypertable('sensor_readings', 'time');

-- Create some indexes for better query performance
CREATE INDEX idx_weather_city_time ON weather_data (city, time DESC);
CREATE INDEX idx_stock_symbol_time ON stock_prices (symbol, time DESC);
CREATE INDEX idx_sensor_device_time ON sensor_readings (device_id, time DESC);
```

Start the Database

```
# Start TimescaleDB
docker compose up -d

# Wait for startup
sleep 10

# Verify it's running
docker compose ps
```

Step 2: API Data Ingestion

Install Python Dependencies

```
# Create virtual environment
python3 -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install required packages
pip install psycopg2-binary requests pandas matplotlib seaborn python-dotenv schedule
```

Create Environment Configuration

Create `.env` file:

```
# Database connection
DB_HOST=localhost
DB_PORT=5432
DB_NAME=integrations_db
DB_USER=admin
DB_PASSWORD=admin123

# API Keys (Optional - we'll use free endpoints)
OPENWEATHER_API_KEY=your_api_key_here
ALPHAVANTAGE_API_KEY=your_api_key_here
```

Create Weather API Ingestion

Create `weather_ingestion.py`:

```
#!/usr/bin/env python3
import requests
import psycopg2
import json
import time
from datetime import datetime
from dotenv import load_dotenv
import os

load_dotenv()

class WeatherIngestion:
```

```
def __init__(self):
    self.db_config = {
        'host': os.getenv('DB_HOST', 'localhost'),
        'port': os.getenv('DB_PORT', 5432),
        'database': os.getenv('DB_NAME', 'integrations_db'),
        'user': os.getenv('DB_USER', 'admin'),
        'password': os.getenv('DB_PASSWORD', 'admin123')
    }

    # Using free weather API (no key required)
    self.base_url = "https://api.open-meteo.com/v1/forecast"

    # Cities to monitor
    self.cities = {
        'New York': {'lat': 40.7128, 'lon': -74.0060},
        'London': {'lat': 51.5074, 'lon': -0.1278},
        'Tokyo': {'lat': 35.6762, 'lon': 139.6503},
        'Sydney': {'lat': -33.8688, 'lon': 151.2093},
        'Mumbai': {'lat': 19.0760, 'lon': 72.8777}
    }

def connect_db(self):
    """Connect to TimescaleDB"""
    try:
        conn = psycopg2.connect(**self.db_config)
        return conn
    except Exception as e:
        print(f"❌ Database connection failed: {e}")
        return None

def fetch_weather_data(self, city, coordinates):
    """Fetch weather data from Open-Meteo API"""
    try:
        params = {
            'latitude': coordinates['lat'],
            'longitude': coordinates['lon'],
            'current': 'temperature_2m,relative_humidity_2m,surface_pressure,wind_speed_10m',
            'timezone': 'auto'
        }

        response = requests.get(self.base_url, params=params, timeout=10)
        response.raise_for_status()

        data = response.json()
```

```
            current = data['current']

            return {
                'temperature': current.get('temperature_2m'),
                'humidity': current.get('relative_humidity_2m'),
                'pressure': current.get('surface_pressure'),
                'wind_speed': current.get('wind_speed_10m'),
                'description': f"Temperature: {current.get('temperature_2m')}°C",
                'timestamp': datetime.fromisoformat(current['time'].replace('Z', '+00:00'))
            }

        except Exception as e:
            print(f"❌ Failed to fetch weather for {city}: {e}")
            return None

    def insert_weather_data(self, city, weather_data):
        """Insert weather data into TimescaleDB"""
        conn = self.connect_db()
        if not conn:
            return False

        try:
            cursor = conn.cursor()

            insert_query = """
                INSERT INTO weather_data (time, city, temperature, humidity, pressure, wind_speed,
description)
                VALUES (%s, %s, %s, %s, %s, %s, %s)
            """

            cursor.execute(insert_query, (
                weather_data['timestamp'],
                city,
                weather_data['temperature'],
                weather_data['humidity'],
                weather_data['pressure'],
                weather_data['wind_speed'],
                weather_data['description']
            ))

            conn.commit()
            print(f"☑ Inserted weather data for {city}: {weather_data['temperature']}°C")
            return True
```

```
            except Exception as e:
                print(f"❌ Failed to insert data for {city}: {e}")
                return False
            finally:
                conn.close()

    def run_ingestion_cycle(self):
        """Run one complete ingestion cycle for all cities"""
        print(f"🌤️  Starting weather ingestion at {datetime.now()}")

        success_count = 0
        for city, coordinates in self.cities.items():
            weather_data = self.fetch_weather_data(city, coordinates)

            if weather_data:
                if self.insert_weather_data(city, weather_data):
                    success_count += 1
                time.sleep(1)  # Be nice to the API

        print(f"📊 Ingestion complete: {success_count}/{len(self.cities)} cities updated")
        return success_count

    def run_continuous(self, interval_minutes=15):
        """Run continuous ingestion"""
        print(f"🔄 Starting continuous weather ingestion (every {interval_minutes} minutes)")
        print("Press Ctrl+C to stop")

        try:
            while True:
                self.run_ingestion_cycle()
                print(f"💤 Sleeping for {interval_minutes} minutes...")
                time.sleep(interval_minutes * 60)
        except KeyboardInterrupt:
            print("\n🛑 Stopping weather ingestion")

if __name__ == "__main__":
    ingestion = WeatherIngestion()

    # Run a single cycle first
    print("🧪 Testing single ingestion cycle...")
    ingestion.run_ingestion_cycle()

    # Ask user if they want continuous ingestion
```

```
    print("\n" + "="*50)
    response = input("Run continuous ingestion? (y/n): ").lower()

    if response == 'y':
        ingestion.run_continuous(interval_minutes=5)  # Every 5 minutes for demo
    else:
        print("☑ Single ingestion complete!")
```

Create Simulated IoT Sensor Ingestion

Create `sensor_ingestion.py`:

```
#!/usr/bin/env python3
import psycopg2
import random
import time
import json
from datetime import datetime, timedelta
from dotenv import load_dotenv
import os
import threading

load_dotenv()

class SensorIngestion:
    def __init__(self):
        self.db_config = {
            'host': os.getenv('DB_HOST', 'localhost'),
            'port': os.getenv('DB_PORT', 5432),
            'database': os.getenv('DB_NAME', 'integrations_db'),
            'user': os.getenv('DB_USER', 'admin'),
            'password': os.getenv('DB_PASSWORD', 'admin123')
        }

        # Simulated sensor configurations
        self.sensors = {
            'factory_floor': {
                'devices': ['TEMP_001', 'TEMP_002', 'HUMID_001', 'PRESS_001'],
                'location': 'Factory Floor A',
                'sensors': {
                    'TEMP_001': {'type': 'temperature', 'unit': 'celsius', 'range': (18, 26)},
                    'TEMP_002': {'type': 'temperature', 'unit': 'celsius', 'range': (20, 28)},
                    'HUMID_001': {'type': 'humidity', 'unit': 'percent', 'range': (40, 70)},
```

```
                'PRESS_001': {'type': 'pressure', 'unit': 'hPa', 'range': (1010, 1025)}
            }
        },
        'warehouse': {
            'devices': ['TEMP_003', 'MOTION_001', 'LIGHT_001'],
            'location': 'Warehouse B',
            'sensors': {
                'TEMP_003': {'type': 'temperature', 'unit': 'celsius', 'range': (15, 25)},
                'MOTION_001': {'type': 'motion', 'unit': 'boolean', 'range': (0, 1)},
                'LIGHT_001': {'type': 'light', 'unit': 'lux', 'range': (100, 800)}
            }
        }
    }

    self.running = False

def connect_db(self):
    """Connect to TimescaleDB"""
    try:
        conn = psycopg2.connect(**self.db_config)
        return conn
    except Exception as e:
        print(f"❌ Database connection failed: {e}")
        return None

def generate_sensor_reading(self, device_id, sensor_config, location):
    """Generate realistic sensor reading"""
    sensor_type = sensor_config['type']
    unit = sensor_config['unit']
    value_range = sensor_config['range']

    if sensor_type == 'motion':
        # Motion sensor: mostly 0, occasionally 1
        value = 1 if random.random() < 0.1 else 0
    elif sensor_type == 'temperature':
        # Temperature with some natural variation
        base_temp = sum(value_range) / 2
        variation = (value_range[1] - value_range[0]) * 0.3
        value = base_temp + random.uniform(-variation, variation)
        value = round(value, 2)
    elif sensor_type == 'humidity':
        # Humidity with gradual changes
        value = random.uniform(*value_range)
        value = round(value, 1)
```

```
        elif sensor_type == 'pressure':
            # Pressure with small variations
            base_pressure = sum(value_range) / 2
            variation = (value_range[1] - value_range[0]) * 0.2
            value = base_pressure + random.uniform(-variation, variation)
            value = round(value, 2)
        else:
            # Default: random value in range
            value = round(random.uniform(*value_range), 2)

        # Add some metadata
        metadata = {
            'battery_level': random.randint(75, 100),
            'signal_strength': random.randint(-80, -30),
            'firmware_version': '1.2.3'
        }

        return {
            'device_id': device_id,
            'sensor_type': sensor_type,
            'value': value,
            'unit': unit,
            'location': location,
            'metadata': json.dumps(metadata),
            'timestamp': datetime.now()
        }

    def insert_sensor_reading(self, reading):
        """Insert sensor reading into TimescaleDB"""
        conn = self.connect_db()
        if not conn:
            return False

        try:
            cursor = conn.cursor()

            insert_query = """
                INSERT INTO sensor_readings (time, device_id, sensor_type, value, unit, location,
metadata)
                VALUES (%s, %s, %s, %s, %s, %s, %s)
            """

            cursor.execute(insert_query, (
                reading['timestamp'],
```

```
            reading['device_id'],
            reading['sensor_type'],
            reading['value'],
            reading['unit'],
            reading['location'],
            reading['metadata']
        ))

        conn.commit()
        print(f"📡 {reading['device_id']}: {reading['value']} {reading['unit']}")
        return True

    except Exception as e:
        print(f"❌ Failed to insert sensor reading: {e}")
        return False
    finally:
        conn.close()

def simulate_location_sensors(self, location_name, location_config):
    """Simulate all sensors for a specific location"""
    while self.running:
        for device_id, sensor_config in location_config['sensors'].items():
            reading = self.generate_sensor_reading(
                device_id,
                sensor_config,
                location_config['location']
            )
            self.insert_sensor_reading(reading)
            time.sleep(0.5)  # Small delay between readings

        time.sleep(5)  # Delay between cycles

def start_simulation(self, duration_minutes=10):
    """Start sensor simulation for all locations"""
    print(f"🏭 Starting IoT sensor simulation for {duration_minutes} minutes")
    print("📡 Sensors active:")

    for location, config in self.sensors.items():
        print(f"  📍 {config['location']}: {', '.join(config['devices'])}")

    self.running = True
    threads = []
```

```
        # Start a thread for each location
        for location_name, location_config in self.sensors.items():
            thread = threading.Thread(
                target=self.simulate_location_sensors,
                args=(location_name, location_config),
                daemon=True
            )
            thread.start()
            threads.append(thread)

        try:
            # Run for specified duration
            time.sleep(duration_minutes * 60)
        except KeyboardInterrupt:
            print("\n🛑 Stopping sensor simulation...")
        finally:
            self.running = False
            print("☑ Sensor simulation stopped")

if __name__ == "__main__":
    simulation = SensorIngestion()

    print("🩹 Testing sensor ingestion...")
    simulation.start_simulation(duration_minutes=2)  # Run for 2 minutes
```

Run the Data Ingestion

```
# Terminal 1: Start weather ingestion
python weather_ingestion.py

# Terminal 2: Start sensor simulation (run this after weather starts)
python sensor_ingestion.py
```

Step 3: Data Export and Analysis with Pandas

Create Data Analysis Module

Create `data_analysis.py`:

```
#!/usr/bin/env python3
import pandas as pd
import psycopg2
```

```
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta
import numpy as np
from dotenv import load_dotenv
import os

load_dotenv()

class TimescaleAnalyzer:
    def __init__(self):
        self.db_config = {
            'host': os.getenv('DB_HOST', 'localhost'),
            'port': os.getenv('DB_PORT', 5432),
            'database': os.getenv('DB_NAME', 'integrations_db'),
            'user': os.getenv('DB_USER', 'admin'),
            'password': os.getenv('DB_PASSWORD', 'admin123')
        }

        # Set up plotting style
        plt.style.use('seaborn-v0_8')
        sns.set_palette("husl")

    def connect_db(self):
        """Create database connection"""
        try:
            conn = psycopg2.connect(**self.db_config)
            return conn
        except Exception as e:
            print(f"❌ Database connection failed: {e}")
            return None

    def query_to_dataframe(self, query, params=None):
        """Execute query and return pandas DataFrame"""
        conn = self.connect_db()
        if not conn:
            return None

        try:
            df = pd.read_sql_query(query, conn, params=params)
            return df
        except Exception as e:
            print(f"❌ Query failed: {e}")
            return None
```

```
    finally:
        conn.close()

def analyze_weather_data(self):
    """Analyze weather data and create visualizations"""
    print("🌤️ Analyzing weather data...")

    # Query recent weather data
    query = """
        SELECT
            time,
            city,
            temperature,
            humidity,
            pressure,
            wind_speed
        FROM weather_data
        WHERE time >= NOW() - INTERVAL '24 hours'
        ORDER BY time DESC, city
    """

    df = self.query_to_dataframe(query)

    if df is None or df.empty:
        print("❌ No weather data found")
        return

    print(f"📊 Found {len(df)} weather records")
    print(f"🏙️ Cities: {', '.join(df['city'].unique())}")

    # Convert time to datetime
    df['time'] = pd.to_datetime(df['time'])

    # Create visualizations
    fig, axes = plt.subplots(2, 2, figsize=(15, 12))
    fig.suptitle('Weather Data Analysis', fontsize=16, fontweight='bold')

    # Temperature by city over time
    for city in df['city'].unique():
        city_data = df[df['city'] == city]
        axes[0, 0].plot(city_data['time'], city_data['temperature'],
                        marker='o', label=city, linewidth=2)

    axes[0, 0].set_title('Temperature Over Time by City')
```

```
axes[0, 0].set_xlabel('Time')
axes[0, 0].set_ylabel('Temperature (°C)')
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)
axes[0, 0].tick_params(axis='x', rotation=45)

# Current temperature comparison
latest_data = df.groupby('city').last()
bars = axes[0, 1].bar(latest_data.index, latest_data['temperature'])
axes[0, 1].set_title('Current Temperature by City')
axes[0, 1].set_ylabel('Temperature (°C)')
axes[0, 1].tick_params(axis='x', rotation=45)

# Add value labels on bars
for bar in bars:
    height = bar.get_height()
    axes[0, 1].text(bar.get_x() + bar.get_width()/2., height,
                    f'{height:.1f}°C',
                    ha='center', va='bottom')

# Humidity vs Temperature scatter
colors = plt.cm.viridis(np.linspace(0, 1, len(df['city'].unique())))
for i, city in enumerate(df['city'].unique()):
    city_data = df[df['city'] == city]
    axes[1, 0].scatter(city_data['temperature'], city_data['humidity'],
                       label=city, alpha=0.7, s=60, color=colors[i])

axes[1, 0].set_title('Humidity vs Temperature')
axes[1, 0].set_xlabel('Temperature (°C)')
axes[1, 0].set_ylabel('Humidity (%)')
axes[1, 0].legend()
axes[1, 0].grid(True, alpha=0.3)

# Weather summary statistics
summary_stats = df.groupby('city').agg({
    'temperature': ['mean', 'min', 'max'],
    'humidity': 'mean',
    'wind_speed': 'mean'
}).round(2)

# Create a heatmap of average values
heatmap_data = df.groupby('city')[['temperature', 'humidity', 'wind_speed']].mean()
im = axes[1, 1].imshow(heatmap_data.T, cmap='YlOrRd', aspect='auto')
axes[1, 1].set_title('Average Weather Metrics Heatmap')
```

```
    axes[1, 1].set_xticks(range(len(heatmap_data.index)))
    axes[1, 1].set_xticklabels(heatmap_data.index, rotation=45)
    axes[1, 1].set_yticks(range(len(heatmap_data.columns)))
    axes[1, 1].set_yticklabels(heatmap_data.columns)

    # Add colorbar
    plt.colorbar(im, ax=axes[1, 1])

    # Add text annotations
    for i in range(len(heatmap_data.index)):
        for j in range(len(heatmap_data.columns)):
            text = axes[1, 1].text(i, j, f'{heatmap_data.iloc[i, j]:.1f}',
                                   ha="center", va="center", color="black")

    plt.tight_layout()
    plt.savefig('weather_analysis.png', dpi=300, bbox_inches='tight')
    plt.show()

    # Print summary statistics
    print("\n📊 Weather Summary Statistics:")
    print(summary_stats)

    return df

def analyze_sensor_data(self):
    """Analyze IoT sensor data"""
    print("\n📡 Analyzing sensor data...")

    # Query recent sensor data
    query = """
        SELECT
            time,
            device_id,
            sensor_type,
            value,
            unit,
            location,
            metadata
        FROM sensor_readings
        WHERE time >= NOW() - INTERVAL '2 hours'
        ORDER BY time DESC
    """

    df = self.query_to_dataframe(query)
```

```
if df is None or df.empty:
    print("❌ No sensor data found")
    return

print(f"📊 Found {len(df)} sensor readings")
print(f"🏭 Locations: {', '.join(df['location'].unique())}")

# Convert time to datetime
df['time'] = pd.to_datetime(df['time'])

# Create sensor-specific visualizations
fig, axes = plt.subplots(2, 2, figsize=(15, 12))
fig.suptitle('IoT Sensor Data Analysis', fontsize=16, fontweight='bold')

# Temperature sensors over time
temp_data = df[df['sensor_type'] == 'temperature']
if not temp_data.empty:
    for device in temp_data['device_id'].unique():
        device_data = temp_data[temp_data['device_id'] == device]
        axes[0, 0].plot(device_data['time'], device_data['value'],
                        marker='o', label=device, linewidth=2)

    axes[0, 0].set_title('Temperature Sensors Over Time')
    axes[0, 0].set_xlabel('Time')
    axes[0, 0].set_ylabel('Temperature (°C)')
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)
    axes[0, 0].tick_params(axis='x', rotation=45)

# Sensor type distribution
sensor_counts = df['sensor_type'].value_counts()
bars = axes[0, 1].bar(sensor_counts.index, sensor_counts.values)
axes[0, 1].set_title('Sensor Reading Count by Type')
axes[0, 1].set_ylabel('Number of Readings')
axes[0, 1].tick_params(axis='x', rotation=45)

# Add value labels on bars
for bar in bars:
    height = bar.get_height()
    axes[0, 1].text(bar.get_x() + bar.get_width()/2., height,
                    f'{int(height)}',
                    ha='center', va='bottom')
```

```
# Humidity sensor data (if available)
humidity_data = df[df['sensor_type'] == 'humidity']
if not humidity_data.empty:
    axes[1, 0].plot(humidity_data['time'], humidity_data['value'],
                    'g-o', linewidth=2)
    axes[1, 0].set_title('Humidity Sensor Readings')
    axes[1, 0].set_xlabel('Time')
    axes[1, 0].set_ylabel('Humidity (%)')
    axes[1, 0].grid(True, alpha=0.3)
    axes[1, 0].tick_params(axis='x', rotation=45)
else:
    axes[1, 0].text(0.5, 0.5, 'No humidity data available',
                    ha='center', va='center', transform=axes[1, 0].transAxes)

# Latest sensor values by location
latest_readings = df.loc[df.groupby(['location', 'sensor_type'])['time'].idxmax()]
pivot_data = latest_readings.pivot_table(
    index='location',
    columns='sensor_type',
    values='value',
    aggfunc='mean'
)

if not pivot_data.empty:
    sns.heatmap(pivot_data, annot=True, fmt='.1f',
                cmap='viridis', ax=axes[1, 1])
    axes[1, 1].set_title('Latest Sensor Values by Location')
else:
    axes[1, 1].text(0.5, 0.5, 'Insufficient data for heatmap',
                    ha='center', va='center', transform=axes[1, 1].transAxes)

plt.tight_layout()
plt.savefig('sensor_analysis.png', dpi=300, bbox_inches='tight')
plt.show()

# Print summary statistics
print("\n📈 Sensor Summary Statistics:")
summary = df.groupby(['location', 'sensor_type']).agg({
    'value': ['count', 'mean', 'min', 'max']
}).round(2)
print(summary)

return df
```

```
def create_combined_dashboard(self):
    """Create a combined dashboard with multiple data sources"""
    print("\n📊 Creating combined data dashboard...")

    # Query data from multiple tables
    queries = {
        'weather': """
            SELECT
                time_bucket('1 hour', time) as bucket,
                city,
                AVG(temperature) as avg_temp,
                AVG(humidity) as avg_humidity
            FROM weather_data
            WHERE time >= NOW() - INTERVAL '24 hours'
            GROUP BY bucket, city
            ORDER BY bucket DESC
        """,
        'sensors': """
            SELECT
                time_bucket('15 minutes', time) as bucket,
                sensor_type,
                location,
                AVG(value) as avg_value,
                COUNT(*) as reading_count
            FROM sensor_readings
            WHERE time >= NOW() - INTERVAL '2 hours'
            GROUP BY bucket, sensor_type, location
            ORDER BY bucket DESC
        """
    }

    data = {}
    for name, query in queries.items():
        df = self.query_to_dataframe(query)
        if df is not None and not df.empty:
            df['bucket'] = pd.to_datetime(df['bucket'])
            data[name] = df
        else:
            print(f"⚠️  No data found for {name}")

    if not data:
        print("❌ No data available for dashboard")
        return
```

```
# Create dashboard
fig = plt.figure(figsize=(16, 10))
fig.suptitle('TimescaleDB Integration Dashboard', fontsize=18, fontweight='bold')

# Create grid layout
gs = fig.add_gridspec(3, 3, hspace=0.3, wspace=0.3)

# Weather data visualizations
if 'weather' in data:
    weather_df = data['weather']

    # Temperature trends
    ax1 = fig.add_subplot(gs[0, :2])
    for city in weather_df['city'].unique():
        city_data = weather_df[weather_df['city'] == city]
        ax1.plot(city_data['bucket'], city_data['avg_temp'],
                 marker='o', label=city, linewidth=2)
    ax1.set_title('Weather Temperature Trends (24h)')
    ax1.set_ylabel('Temperature (°C)')
    ax1.legend()
    ax1.grid(True, alpha=0.3)
    ax1.tick_params(axis='x', rotation=45)

    # Current weather summary
    ax2 = fig.add_subplot(gs[0, 2])
    latest_weather = weather_df.groupby('city').last()
    bars = ax2.bar(range(len(latest_weather)), latest_weather['avg_temp'])
    ax2.set_title('Latest Temperatures')
    ax2.set_xticks(range(len(latest_weather)))
    ax2.set_xticklabels(latest_weather.index, rotation=45)
    ax2.set_ylabel('Temperature (°C)')

    # Add value labels
    for i, bar in enumerate(bars):
        height = bar.get_height()
        ax2.text(bar.get_x() + bar.get_width()/2., height,
                 f'{height:.1f}°C',
                 ha='center', va='bottom')

# Sensor data visualizations
if 'sensors' in data:
    sensors_df = data['sensors']

    # Temperature sensor trends
```

```
    ax3 = fig.add_subplot(gs[1, :2])
    temp_sensors = sensors_df[sensors_df['sensor_type'] == 'temperature']
    if not temp_sensors.empty:
        for location in temp_sensors['location'].unique():
            loc_data = temp_sensors[temp_sensors['location'] == location]
            ax3.plot(loc_data['bucket'], loc_data['avg_value'],
                     marker='s', label=location, linewidth=2)
        ax3.set_title('IoT Temperature Sensors (2h)')
        ax3.set_ylabel('Temperature (°C)')
        ax3.legend()
        ax3.grid(True, alpha=0.3)
        ax3.tick_params(axis='x', rotation=45)
    else:
        ax3.text(0.5, 0.5, 'No temperature sensor data',
                 ha='center', va='center', transform=ax3.transAxes)

    # Sensor activity summary
    ax4 = fig.add_subplot(gs[1, 2])
    activity = sensors_df.groupby('sensor_type')['reading_count'].sum()
    wedges, texts, autotexts = ax4.pie(activity.values, labels=activity.index,
                                       autopct='%1.1f%%', startangle=90)
    ax4.set_title('Sensor Activity Distribution')

# Combined metrics
ax5 = fig.add_subplot(gs[2, :])

# Create a timeline showing data ingestion rates
timeline_data = []

if 'weather' in data:
    weather_timeline = data['weather'].groupby('bucket').size().reset_index()
    weather_timeline['source'] = 'Weather API'
    weather_timeline.columns = ['time', 'count', 'source']
    timeline_data.append(weather_timeline)

if 'sensors' in data:
    sensor_timeline = data['sensors'].groupby('bucket')['reading_count'].sum().reset_index()
    sensor_timeline['source'] = 'IoT Sensors'
    sensor_timeline.columns = ['time', 'count', 'source']
    timeline_data.append(sensor_timeline)

if timeline_data:
    combined_timeline = pd.concat(timeline_data, ignore_index=True)
```

```
            for source in combined_timeline['source'].unique():
                source_data = combined_timeline[combined_timeline['source'] == source]
                ax5.plot(source_data['time'], source_data['count'],
                         marker='o', label=source, linewidth=2)

            ax5.set_title('Data Ingestion Timeline')
            ax5.set_xlabel('Time')
            ax5.set_ylabel('Records Count')
            ax5.legend()
            ax5.grid(True, alpha=0.3)
            ax5.tick_params(axis='x', rotation=45)
        else:
            ax5.text(0.5, 0.5, 'No timeline data available',
                     ha='center', va='center', transform=ax5.transAxes)

        plt.savefig('combined_dashboard.png', dpi=300, bbox_inches='tight')
        plt.show()

        # Print data summary
        print("\n📊 Data Integration Summary:")
        if 'weather' in data:
            print(f"   🌤️  Weather: {len(data['weather'])} hourly averages from
{data['weather']['city'].nunique()} cities")
        if 'sensors' in data:
            print(f"   📡 Sensors: {len(data['sensors'])} readings from
{data['sensors']['location'].nunique()} locations")

        return data

    def export_data_samples(self):
        """Export sample data to various formats for external use"""
        print("\n💾 Exporting data samples...")

        # Export weather data
        weather_query = """
            SELECT * FROM weather_data
            WHERE time >= NOW() - INTERVAL '24 hours'
            ORDER BY time DESC
        """

        weather_df = self.query_to_dataframe(weather_query)
        if weather_df is not None and not weather_df.empty:
            weather_df.to_csv('weather_export.csv', index=False)
            weather_df.to_json('weather_export.json', orient='records', date_format='iso')
```

```
            print(f"  ☑ Weather data exported: {len(weather_df)} records")

        # Export sensor data
        sensor_query = """
            SELECT * FROM sensor_readings
            WHERE time >= NOW() - INTERVAL '2 hours'
            ORDER BY time DESC
        """

        sensor_df = self.query_to_dataframe(sensor_query)
        if sensor_df is not None and not sensor_df.empty:
            sensor_df.to_csv('sensor_export.csv', index=False)
            sensor_df.to_json('sensor_export.json', orient='records', date_format='iso')
            print(f"  ☑ Sensor data exported: {len(sensor_df)} records")

        # Create summary report
        if weather_df is not None and sensor_df is not None:
            summary_report = {
                'export_timestamp': datetime.now().isoformat(),
                'weather_records': len(weather_df),
                'sensor_records': len(sensor_df),
                'weather_cities': weather_df['city'].nunique() if not weather_df.empty else 0,
                'sensor_locations': sensor_df['location'].nunique() if not sensor_df.empty else 0,
                'time_range': {
                    'start': min(weather_df['time'].min() if not weather_df.empty else datetime.now(),
                                 sensor_df['time'].min() if not sensor_df.empty else
datetime.now()).isoformat(),
                    'end': max(weather_df['time'].max() if not weather_df.empty else datetime.now(),
                               sensor_df['time'].max() if not sensor_df.empty else
datetime.now()).isoformat()
                }
            }

            import json
            with open('data_export_summary.json', 'w') as f:
                json.dump(summary_report, f, indent=2)

            print(f"  ☑ Summary report created")

        print("💾 Export complete! Files created:")
        print("  - weather_export.csv/json")
        print("  - sensor_export.csv/json")
        print("  - data_export_summary.json")
```

```
if __name__ == "__main__":
    analyzer = TimescaleAnalyzer()

    print("🔍 Starting TimescaleDB Data Analysis")
    print("=" * 50)

    # Run all analyses
    try:
        weather_data = analyzer.analyze_weather_data()
        sensor_data = analyzer.analyze_sensor_data()
        combined_data = analyzer.create_combined_dashboard()
        analyzer.export_data_samples()

        print("\n☑ Analysis complete! Check the generated PNG files and exported data.")

    except Exception as e:
        print(f"❌ Analysis failed: {e}")
```

Create Simple Pipeline Runner

Create `run_pipeline.py`:

```
#!/usr/bin/env python3
import subprocess
import time
import sys
import os
from datetime import datetime

def run_command(command, description):
    """Run a command and handle errors"""
    print(f"\n🚀 {description}")
    print(f"💻 Running: {command}")

    try:
        result = subprocess.run(command, shell=True, check=True,
                                capture_output=True, text=True)
        print(f"☑ {description} completed successfully")
        if result.stdout:
            print(f"📄 Output: {result.stdout.strip()}")
        return True
    except subprocess.CalledProcessError as e:
```

```
        print(f"❌ {description} failed")
        print(f"Error: {e.stderr.strip() if e.stderr else str(e)}")
        return False

def check_dependencies():
    """Check if all required packages are installed"""
    print("🔍 Checking dependencies...")

    required_packages = ['psycopg2', 'pandas', 'matplotlib', 'seaborn', 'requests']
    missing_packages = []

    for package in required_packages:
        try:
            __import__(package)
            print(f"  ☑ {package}")
        except ImportError:
            print(f"  ❌ {package} - MISSING")
            missing_packages.append(package)

    if missing_packages:
        print(f"\n⚠  Missing packages: {', '.join(missing_packages)}")
        print("💡 Run: pip install psycopg2-binary pandas matplotlib seaborn requests python-
dotenv")
        return False

    print("☑ All dependencies satisfied")
    return True

def main():
    print("🎯 TimescaleDB Integration Pipeline Runner")
    print("=" * 50)

    # Check dependencies
    if not check_dependencies():
        sys.exit(1)

    # Check if database is running
    print("\n🔍 Checking database connection...")
    if not run_command("docker compose ps timescaledb", "Database status check"):
        print("💡 Start the database with: docker compose up -d")
        sys.exit(1)
```

```
print("\n📋 Pipeline Options:")
print("1. Run complete pipeline (ingestion + analysis)")
print("2. Run data ingestion only")
print("3. Run data analysis only")
print("4. Export data samples")
print("5. Check data status")

choice = input("\nSelect option (1-5): ").strip()

if choice == "1":
    # Complete pipeline
    print("\n🎯 Running complete data pipeline...")

    # Start weather ingestion in background
    print("🌤️  Starting weather data ingestion...")
    weather_process = subprocess.Popen([sys.executable, 'weather_ingestion.py'])

    # Wait a bit for weather data to start
    time.sleep(10)

    # Start sensor simulation in background
    print("📡 Starting sensor simulation...")
    sensor_process = subprocess.Popen([sys.executable, 'sensor_ingestion.py'])

    # Wait for data collection
    print("⏳ Collecting data for 3 minutes...")
    time.sleep(180)  # 3 minutes

    # Stop ingestion processes
    print("🛑 Stopping data ingestion...")
    weather_process.terminate()
    sensor_process.terminate()

    # Wait a moment for cleanup
    time.sleep(5)

    # Run analysis
    print("📊 Running data analysis...")
    run_command(f"{sys.executable} data_analysis.py", "Data analysis")

elif choice == "2":
    # Ingestion only
    print("\n📥 Starting data ingestion...")
```

```
        print("🌤️  Weather ingestion will run for 2 minutes")
        print("📡 Sensor simulation will run for 2 minutes")
        print("Press Ctrl+C to stop early")

        try:
            # Start both ingestion processes
            weather_process = subprocess.Popen([sys.executable, 'weather_ingestion.py'])
            time.sleep(5)
            sensor_process = subprocess.Popen([sys.executable, 'sensor_ingestion.py'])

            # Wait for data collection
            time.sleep(120)  # 2 minutes

            # Stop processes
            weather_process.terminate()
            sensor_process.terminate()
            print("☑️ Data ingestion completed")

        except KeyboardInterrupt:
            print("\n🛑 Stopping ingestion processes...")
            try:
                weather_process.terminate()
                sensor_process.terminate()
            except:
                pass

    elif choice == "3":
        # Analysis only
        run_command(f"{sys.executable} data_analysis.py", "Data analysis")

    elif choice == "4":
        # Export only
        print("\n💾 Exporting data samples...")
        run_command(f"{sys.executable} -c \"from data_analysis import TimescaleAnalyzer;
TimescaleAnalyzer().export_data_samples()\"", "Data export")

    elif choice == "5":
        # Status check
        run_command(f"{sys.executable} -c \"from data_analysis import TimescaleAnalyzer;
analyzer = TimescaleAnalyzer(); analyzer.query_to_dataframe('SELECT COUNT(*) as
weather_records FROM weather_data WHERE time >= NOW() - INTERVAL \\'24 hours\\';');
analyzer.query_to_dataframe('SELECT COUNT(*) as sensor_records FROM sensor_readings
WHERE time >= NOW() - INTERVAL \\'2 hours\\';')\"", "Data status check")
```

```
    else:
        print("❌ Invalid option selected")
        sys.exit(1)

    print(f"\n🎉 Pipeline execution completed at {datetime.now()}")
    print("📁 Check the current directory for generated files:")
    print("  - *.png (visualization files)")
    print("  - *.csv (exported data)")
    print("  - *.json (exported data)")

if __name__ == "__main__":
    main()
```

Step 4: Complete Pipeline Demo

Run the Complete Pipeline

```
# Make sure you're in the project directory and virtual environment is active
python run_pipeline.py
```

Manual Step-by-Step Execution

If you prefer to run each component manually:

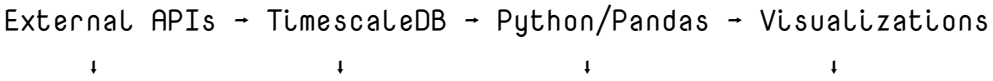
```
# Terminal 1: Start weather ingestion (let it run for a few minutes)
python weather_ingestion.py

# Terminal 2: Start sensor simulation (let it run for a few minutes)
python sensor_ingestion.py

# Terminal 3: After collecting data, run analysis
python data_analysis.py
```

Step 5: Understanding the Integration Patterns

Integration Architecture



```
Weather API     Hypertables     DataFrames      Matplotlib
IoT Sensors     Time-series     Analysis        Dashboards
REST APIs       Compression     Aggregation     Exports
File Imports    SQL Queries     Statistics      CSV/JSON
```

Key Integration Benefits

✅ **TimescaleDB Advantages:**

- **Time-series optimization**: Automatic partitioning and compression
- **SQL compatibility**: Use familiar SQL for complex queries
- **Scalability**: Handle millions of time-series data points
- **Real-time ingestion**: Insert data as it arrives

✅ **Python/Pandas Benefits:**

- **Rich analysis**: Statistical functions and data manipulation
- **Visualization**: Matplotlib, Seaborn for charts and graphs
- **Machine learning**: Easy integration with scikit-learn, TensorFlow
- **Export flexibility**: CSV, JSON, Excel, and more

Common Integration Patterns

📥 **Ingestion Patterns:**

1. **Polling**: Regular API calls (weather example)
2. **Streaming**: Real-time data feeds (sensor simulation)
3. **Batch**: Scheduled bulk uploads
4. **Event-driven**: Triggered by external events

📤 **Export Patterns:**

1. **Scheduled reports**: Daily/weekly summaries
2. **Real-time dashboards**: Live data visualization
3. **File exports**: CSV, JSON for external analysis
4. **Python analysis**: Feed data science workflows

Step 6: Production Considerations

```
# Connect to TimescaleDB using psql
docker exec -it timescale-integrations psql -U admin -d integrations_db
```

Performance Optimization

```
-- Create BRIN index
CREATE INDEX idx_weather_city_time_brin
ON weather_data USING BRIN (city, time);

-- Create continuous aggregate for real-time analytics
CREATE MATERIALIZED VIEW weather_hourly
WITH (timescaledb.continuous) AS
SELECT
    time_bucket('1 hour', time) AS bucket,
    city,
    AVG(temperature) as avg_temp,
    MIN(temperature) as min_temp,
    MAX(temperature) as max_temp,
    AVG(humidity) as avg_humidity
FROM weather_data
GROUP BY bucket, city;


-- Refresh policy for real-time updates
SELECT add_continuous_aggregate_policy('weather_hourly',
    start_offset => INTERVAL '1 day',
    end_offset => INTERVAL '1 hour',
    schedule_interval => INTERVAL '1 hour');
```

Error Handling and Monitoring

Create `monitoring.py`:

```
#!/usr/bin/env python3
import psycopg2
from datetime import datetime, timedelta
import json

def check_data_health():
    """Check data pipeline health"""
    checks = {
        'weather_data_recent': """
            SELECT COUNT(*) as count
            FROM weather_data
            WHERE time >= NOW() - INTERVAL '1 hour'
        """,
        'sensor_data_recent': """
            SELECT COUNT(*) as count
```

```
            FROM sensor_readings
            WHERE time >= NOW() - INTERVAL '15 minutes'
        """,
        'total_records': """
            SELECT
                (SELECT COUNT(*) FROM weather_data) as weather_count,
                (SELECT COUNT(*) FROM sensor_readings) as sensor_count
        """
    }

    # Implementation would check each query and alert if thresholds not met
    print("🔍 Data pipeline health check would run here")
    print("☑ All checks passed (demo)")

if __name__ == "__main__":
    check_data_health()
```

Security Best Practices

```
# Example: Environment-based configuration
import os
from urllib.parse import quote_plus

# Secure database connection
DB_CONFIG = {
    'host': os.getenv('DB_HOST', 'localhost'),
    'port': int(os.getenv('DB_PORT', 5432)),
    'database': os.getenv('DB_NAME'),
    'user': os.getenv('DB_USER'),
    'password': quote_plus(os.getenv('DB_PASSWORD')),
    'sslmode': os.getenv('DB_SSL_MODE', 'require')
}

# API key management
API_KEYS = {
    'weather': os.getenv('WEATHER_API_KEY'),
    'stock': os.getenv('STOCK_API_KEY')
}
```

Step 7: Cleanup

Stop All Processes

```
# If running continuous ingestion, stop with Ctrl+C

# Stop the database
docker compose down -v

# Deactivate virtual environment
deactivate

# Remove project directory (optional)
cd ..
rm -rf timescale-integrations
```

Verify Cleanup

```
# Check no containers remain
docker ps -a | grep timescale

# Check system usage
docker system df
```

Summary: What You Learned

🎯 **Core Concepts Mastered:**

- **Data ingestion patterns**: REST API polling, IoT simulation, real-time streaming
- **TimescaleDB integration**: Hypertables, time-series optimization, SQL queries
- **Python data pipeline**: Pandas for analysis, Matplotlib for visualization
- **Export mechanisms**: CSV, JSON, and programmatic data access

🛠️ **Hands-On Skills:**

- Built weather data ingestion from public APIs
- Simulated IoT sensor data streams
- Created comprehensive data analysis with Pandas
- Generated professional visualizations with Matplotlib
- Exported data in multiple formats for external use

🚀 **Production-Ready Patterns:**

1. **Modular architecture**: Separate ingestion, analysis, and visualization
2. **Error handling**: Robust connection management and graceful failures
3. **Configuration management**: Environment variables and secure secrets
4. **Monitoring**: Data quality checks and pipeline health monitoring

Key Integration Benefits:

- **Real-time insights**: From raw data to visualizations in minutes
- **Scalable architecture**: Handle growing data volumes efficiently
- **Flexible exports**: Data available in multiple formats for different tools
- **Time-series power**: Leverage TimescaleDB's optimization for time-based data

Next Steps for Your Projects:

- Start with simple API integrations using the patterns shown
- Build incrementally: ingestion → storage → analysis → visualization
- Monitor data quality and pipeline performance from day one
- Scale up gradually as your data volume and complexity grows

You now have a complete understanding of building data integration pipelines with TimescaleDB and can create production-ready systems for your own time-series data projects!