# Smart IoT Bolt for Pipelines: Analytics Microservice Documentation

## 1. Introduction to the Analytics Microservice

The Analytics Microservice is a critical component of the Smart IoT Bolt for Pipelines system. It serves as the intelligent brain of the platform, responsible for processing time-series data from pipeline sensors, predicting future values, detecting anomalies, and triggering appropriate responses to ensure pipeline safety and stability.

This microservice utilizes advanced statistical models (ARIMA) and machine learning techniques to provide:

- Predictive analysis of temperature and pressure values
- Detection of anomalous sensor readings
- Identification of potentially catastrophic cascading failures
- Automated control signals to actuators (valves) when thresholds are exceeded

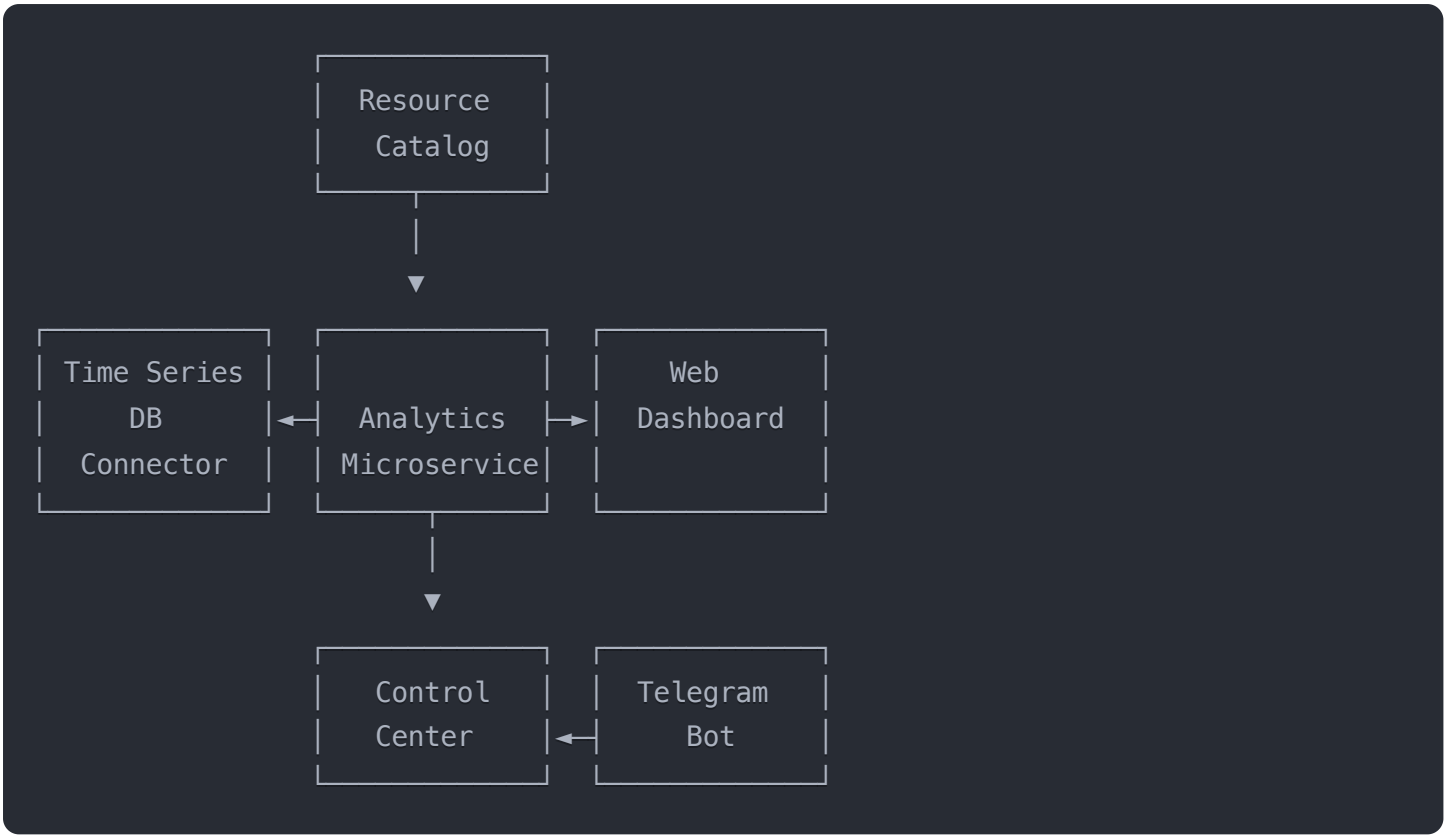
The Analytics Microservice adheres to the microservice architecture principles, communicates with other system components through REST APIs, and follows the development guidelines specified for the Smart IoT Bolt system.

## 2. System Architecture & Integration

### 2.1 Position in Overall System Architecture

The Analytics Microservice fits into the Smart IoT Bolt system architecture as follows:

Copy




### 2.2 Integration Points

The Analytics Microservice integrates with:

1. **Time Series DB Connector**: Retrieves historical sensor data (temperature and pressure) via REST API
2. **Resource Catalog**: Registers itself and discovers other services via REST API
3. **Web Dashboard**: Provides predictions and anomaly alerts via REST API
4. **Telegram Bot**: Sends critical alerts via REST API
5. **Control Center**: Sends valve actuation commands when thresholds are exceeded via REST API

### 2.3 Communication Protocols

- Exclusively uses **REST APIs** for all service interactions
- Implements endpoints using **CherryPy** as required
- Follows RESTful principles with appropriate HTTP methods (GET, POST)

## 3. Key Components & Structure

The Analytics Microservice is organized according to the specified folder structure:

Copy

```
MS_Analytics/
├── __init__.py
├── main.py                 # CherryPy server & startup
├── .env                    # Environment variables
├── requirements.txt        # Dependencies
├── controllers/
│   ├── __init__.py
│   └── analytics_controller.py  # API endpoints
├── models/
│   ├── __init__.py
│   └── prediction_model.py  # ARIMA model implementation
├── services/
│   ├── __init__.py
│   ├── prediction_service.py    # Prediction logic
│   └── anomaly_service.py   # Anomaly detection
└── utils/
    ├── __init__.py
    └── data_processor.py    # Data preprocessing
```

## 4. Detailed Component Explanations

### 4.1 Prediction Model (models/prediction_model.py)

#### 4.1.1 ARIMAModel Class

This class provides the core time-series forecasting capability using the Seasonal ARIMA (SARIMAX) model from statsmodels:

- **Purpose**: Create, fit, and use ARIMA models for temperature and pressure forecasting
- **Key Methods**:
  - `fit(time_series)`: Fits the model to historical sensor data
  - `predict(steps)`: Forecasts future values for a specified number of steps
  - `save_model(filepath)` and `load_model(filepath)`: Persistence functions

#### 4.1.2 find_best_arima_params Function

- **Purpose**: Automatically determines optimal ARIMA parameters (p, d, q)
- **Process**: Tests multiple parameter combinations and selects the one with lowest AIC value
- **Benefit**: Eliminates manual parameter tuning, improving prediction accuracy

### 4.2 Data Processor (utils/data_processor.py)

Provides utilities for preparing time-series data for analysis:

#### 4.2.1 preprocess_time_series Function

- **Purpose**: Cleans and formats raw sensor data for modeling
- **Operations**:
  - Converts to pandas DataFrame
  - Handles timestamps
  - Interpolates missing values
  - Resamples to regular intervals (1-minute)

#### 4.2.2 Stationarity Functions

- `check_stationarity(time_series)`: Tests if a time series is stationary using Augmented Dickey-Fuller test
- `make_stationary(time_series)`: Differentiates a time series until it becomes stationary
- `inverse_transform(predictions, original_series, d)`: Converts differenced predictions back to original scale

### 4.3 Prediction Service (services/prediction_service.py)

Provides high-level prediction functionality:

#### 4.3.1 PredictionService Class

- **Key Methods**:
  - `_get_historical_data(sensor_type, hours)`: Retrieves sensor data from Time Series DB Connector
  - `train_models()`: Creates and trains ARIMA models for temperature and pressure
  - `predict_temperature(hours_ahead)` & `predict_pressure(hours_ahead)`: Generate future predictions
  - `will_exceed_threshold(prediction_data)`: Determines if predicted values will exceed safety thresholds

#### 4.3.2 Prediction Process Flow

1. Retrieve historical data from Time Series DB Connector
2. Preprocess the data for analysis
3. Find optimal ARIMA parameters
4. Train the model on historical data
5. Generate predictions for a specified time horizon
6. Check predictions against thresholds

7. Return formatted prediction data and threshold flags

## 4.4 Anomaly Service (services/anomaly_service.py)

Provides anomaly detection and alert capabilities:

### 4.4.1 AnomalyService Class

- **Key Methods**:
  - `_get_recent_data(sensor_type, minutes)`: Retrieves recent sensor data
  - `detect_anomalies(sensor_type)`: Identifies anomalous sensor readings using Isolation Forest
  - `_is_critical_value(sensor_type, value)`: Checks if a value exceeds critical thresholds
  - `detect_cascading_failures()`: Analyzes correlation between temperature and pressure for risk assessment
  - `send_alert(alert_type, message, data)`: Sends alerts to Web Dashboard and Telegram Bot

### 4.4.2 Anomaly Detection Process

1. Retrieve recent sensor data
2. Apply Isolation Forest algorithm to identify outliers
3. Flag critical anomalies that exceed safety thresholds
4. For cascading failures:
   - Analyze correlation between temperature and pressure
   - Check for simultaneous increasing trends
   - Identify when both values approach their respective thresholds

## 4.5 Analytics Controller (controllers/analytics_controller.py)

Implements the REST API endpoints:

### 4.5.1 API Endpoints

- `/` (GET): Service information
- `/predict_temperature` (GET): Temperature prediction endpoint
- `/predict_pressure` (GET): Pressure prediction endpoint
- `/temperature_anomalies` (GET): Temperature anomaly detection endpoint
- `/pressure_anomalies` (GET): Pressure anomaly detection endpoint
- `/cascading_failures` (GET): Cascading failure risk detection endpoint
- `/trigger_valve` (GET/POST): Valve actuation endpoint

#### 4.5.2 Controller Structure

The controller uses CherryPy's dispatching mechanism with nested classes for each endpoint:

- Main `AnalyticsController` class initializes services
- Each endpoint is implemented as a nested class with appropriate HTTP methods (GET/POST)
- Service registration with Resource Catalog is handled during initialization

### 4.6 Main Application (main.py)

- Initializes the CherryPy application
- Configures server settings from environment variables
- Registers the controller and starts the server

## 5. Data Flow & Decision Making

### 5.1 Prediction Workflow

1. **Data Retrieval**: The microservice requests historical temperature and pressure data from the Time Series DB Connector.
2. **Model Training**: ARIMA models are trained with optimal parameters based on historical data.
3. **Forecasting**: The trained models predict future temperature and pressure values.
4. **Threshold Checking**: Predictions are compared against safety thresholds.
5. **Alert Generation**: If thresholds are predicted to be exceeded, alerts are sent to the Web Dashboard and Telegram Bot.
6. **Valve Control**: In critical cases, commands are sent to the Control Center to adjust valve settings.

### 5.2 Anomaly Detection Workflow

1. **Data Retrieval**: Recent sensor data is retrieved from the Time Series DB Connector.
2. **Anomaly Analysis**: The Isolation Forest algorithm identifies outlier data points.
3. **Criticality Assessment**: Anomalies are evaluated for their severity based on threshold proximity.
4. **Cascading Failure Analysis**: Correlation and trend analysis between temperature and pressure identify systemic risks.
5. **Alert Generation**: Critical anomalies trigger alerts to the Web Dashboard and Telegram Bot.
6. **Automated Response**: In high-risk scenarios, valve actuation commands are sent to the Control Center.

## 6. Technical Implementation Details

### 6.1 Environment Variables

The microservice uses environment variables for configuration:

- `SERVICE_NAME`: Name of the service for registration
- `SERVICE_PORT`: Port on which the service runs
- `RESOURCE_CATALOG_URL`: URL of the Resource Catalog service
- `TIMESERIES_CONNECTOR_URL`: URL of the Time Series DB Connector
- `PRESSURE_THRESHOLD`: Safety threshold for pressure readings
- `TEMPERATURE_THRESHOLD`: Safety threshold for temperature readings
- `WEB_DASHBOARD_URL`: URL of the Web Dashboard service
- `TELEGRAM_BOT_URL`: URL of the Telegram Bot service

### 6.2 Error Handling

- Comprehensive try-except blocks for robust operation
- Detailed error messages with appropriate HTTP status codes
- Graceful degradation when dependent services are unavailable

### 6.3 Performance Considerations

- Efficient data retrieval with time filtering
- Optimized model training with parameter selection
- Resampling of data to reduce computational load
- Caching of trained models to avoid redundant calculations

## 7. Conclusion

The Analytics Microservice is the intelligent core of the Smart IoT Bolt for Pipelines system. It processes sensor data to provide predictive insights, detect anomalies, and trigger automated responses to ensure pipeline safety. By leveraging statistical models and machine learning techniques, it enables a proactive approach to pipeline maintenance and risk management.

The microservice adheres to the project's architecture principles, using CherryPy for REST APIs, communicating with other components through standard protocols, and avoiding hardcoded values. It is designed for scalability, reliability, and ease of integration with the broader system.