Smart City Monitoring System

University Of L'Aquila, Italy

Professor: Davide Di Ruscio

Course: Software Engineering for Internet of Things

Members: Capricci Federico, Di Giuseppe Edoardo

Table of contents

1.Introduction

Rapid urbanization poses significant challenges to infrastructure stability, environmental quality, and the general well-being of citizens. The sustainability of modern metropolitan areas depends heavily on efficient resource management and the ability to monitor urban dynamics in real-time.

In this report, we describe a **Smart City monitoring system** based on the **Internet of Things (IoT)**, in which critical urban parameters—such as traffic flow, temperature, humidity, air quality, and noise levels—are monitored proactively across various city districts. The proposed architecture incorporates data from heterogeneous sensors, real-time processing pipelines, and advanced visualization capabilities.

Its purpose is to provide city administrators with immediate situational awareness, enabling timely interventions to mitigate traffic congestion or environmental hazards, with the ultimate goal of improving the quality of urban life. The implementation of this system aims at taking a step towards data-driven urban planning and the development of more sustainable and resilient cities.

2. System Goals

2.1. Scalable IoT infrastructure for real-time monitoring of urban environments.

> Develop a robust, containerized IoT architecture capable of tracking traffic density, noise pollution, and air quality parameters (such as PM2.5, CO, and NO2) across multiple city districts in real-time.

2.2. Automated alert mechanism for rapid response to city events.

> Integrate an instant notification system (e.g., via Telegram) that alerts city administrators or traffic control centers upon detecting specific threshold breaches, enabling timely intervention.

2.3 Centralized visualization platform for decision support.

> Create an interactive, web-based Grafana dashboard that provides authorities with real-time situational awareness of urban conditions and historical trends to support data-driven decision-making.

3. Functional Requirements:

3.1. Data Generation

The system must simulate a realistic urban sensor network by generating synthetic data streams representing various environmental and traffic parameters. A dedicated simulation component (implemented in Python) shall concurrently emulate multiple sensors—including temperature, humidity, PM2.5, PM10, CO, NO2, O3, noise levels, and traffic density—across distinct city districts and streets. This component must ensure the data is randomized within realistic operating ranges to mimic natural fluctuations and must publish these readings as JSON payloads via the MQTT protocol, utilizing a hierarchical topic structure (e.g., smartcity/district/street/sensor) to ensure logical data organization.

3.2. Data Collecting

The system requires a robust ingestion pipeline capable of handling high-frequency data streams from the sensor network with low latency. A centralized MQTT broker shall act as the entry point for all sensor messages, ensuring decoupled communication between producers and consumers. To ensure data persistence and integrity, a telemetry agent (Telegraf) must subscribe to the broker, parse the incoming JSON payloads, extract metadata (tags such as district and street), and write the formatted metrics into a time-series database (InfluxDB). This process must guarantee that high volumes of time-stamped data are efficiently indexed and stored for historical retrieval.

3.3 Dynamic configuration

The system must support real-time adjustments to its operational parameters without requiring a restart of the simulation service. Specifically, the data generation interval (the "sleep time" between sensor readings) shall be dynamically configurable via the MQTT protocol. The simulation component must subscribe to a dedicated configuration topic (smartcity/config) and listen for incoming JSON control messages. Upon receiving a payload such as {"time_sleep": x}, the system shall immediately update its internal global settings and apply the new sampling frequency across all active sensor threads. This functional requirement ensures that administrators can adjust the granularity of data collection in response to specific urban events or testing needs on the fly.

3.4. Air Quality Detection and Analysis

The system must actively analyze incoming environmental data streams in real-time to assess urban air quality levels. Utilizing a logic processing engine (Node-RED), the system shall evaluate specific pollutants (such as PM2.5, PM10, CO, NO2 and O3) against pre-defined safety thresholds. This requirement entails the implementation of rule-based algorithms that continuously monitor sensor readings; if a value exceeds the established "safe" or "moderate" limits, the system must immediately classify the event as a hazard or warning state, triggering the subsequent alert workflow without human intervention.

3.5. Monitoring Interface

To provide city administrators with immediate situational awareness, the system must provide a centralized, web-based visualization dashboard (Grafana). This interface shall connect directly to the time-series database to render interactive graphs, gauges, and stats representing the current status of the city. The dashboard must support dynamic filtering, allowing users to drill down into specific districts or streets, and must visualize both real-time metrics and historical trends. This capability ensures that decision-makers can visually identify congestion patterns, pollution hotspots, or environmental anomalies at a glance.

3.6. Notification and Alerting

The system must include an automated notification mechanism to ensure rapid response to critical events. Upon detecting an anomaly during the data analysis phase (e.g., a potential fire indicated by high temperature and low humidity, or severe pollution levels), the system must instantly generate and transmit an alert message to relevant authorities. This integration shall utilize an external messaging API (Telegram Bot) to push concise, actionable notifications containing the location (district/street), the specific sensor value, and the timestamp of the event, ensuring that maintenance or emergency teams are informed regardless of their physical location.

4. Non Functional Requirements:

4.1. Portability

The system must ensure high portability to allow seamless deployment across different environments (Development, Testing, and Production) regardless of the underlying host operating system. This is achieved by utilizing **Docker** to containerize every microservice within the architecture. By encapsulating the Python simulator, MQTT broker, Telegraf agent, InfluxDB, Node-RED, and Grafana into isolated containers, the project eliminates the "it works on my machine" problem. The entire infrastructure can be orchestrated and launched with a single command, ensuring consistent behavior across diverse hardware and software setups.

4.2 Scalability and Modularity

The architecture is designed with a high degree of horizontal and vertical scalability. Thanks to the decoupled nature of the MQTT protocol and the modularity of the microservices, the system can easily expand to accommodate urban growth. New city districts, additional streets, or new types of IoT sensors (e.g., waste management or smart lighting) can be integrated by simply updating the configuration files without requiring structural changes to the core logic. This ensures the system can scale from a single neighborhood to a full-scale metropolitan area monitoring platform.

4.3 Resilience and fault tollerance

To satisfy the requirement for high availability and data consistency, the system implements multiple layers of error handling and recovery. **Communication Resilience** is achieved through a buffering mechanism within the messaging pipeline; if a downstream service (like the mqtt comunication) becomes unavailable, the system buffers outgoing metrics in memory to prevent data loss. At the application level, **Robust Error Handling** is implemented using connection retry loops and try-catch blocks in the Python and Node-RED logic. These measures ensure that temporary network failures or service restarts do not halt execution, allowing the system to self-heal and resume operations automatically.

4.4 User friendly UI

The system must provide an intuitive and accessible user experience for city administrators who may not have a technical background. This is addressed by leveraging **Grafana** for the front-end layer. The monitoring interface is designed with a "User-First" approach, utilizing ergonomic gauges, color-coded alerts, and interactive time-series graphs. By providing a web-based, responsive dashboard, the system ensures that complex environmental and traffic data are translated into clear, actionable insights that can be easily interpreted for rapid decision-making.

4.5 Security

The system must maintain high security standards to protect urban data and infrastructure. This is implemented through **Strict Authentication** protocols for every inter-service connection; the MQTT broker, InfluxDB, and Grafana are all protected by mandatory credentials, disabling anonymous access. Furthermore, the system follows security best practices by utilizing **Environment Variables (.env files)** to manage sensitive information such as passwords, API tokens (Telegram), and administrative keys. This approach prevents hardcoding credentials in the source code, reducing the risk of accidental exposure and ensuring a secure deployment pipeline.

5. Technologies Used

The following technologies were used for the development of this project:

Python

Python is utilized to generate high-fidelity synthetic data simulating a network of urban IoT sensors. The **paho-mqtt** library is employed as the MQTT client to publish these metrics to the broker. Messages are published using a hierarchical topic structure: smartcity/{district}/{street}/{sensor}. This design allows for granular subscriptions and future-proof scalability, enabling the system to monitor an arbitrary number of districts and streets. The simulated sensors include **temperature, humidity, noise levels, traffic density, and air quality parameters (PM2.5, PM10, CO, NO2, O3)**.

MQTT - Mosquitto

The **Mosquitto** broker serves as the central messaging hub. We implemented advanced security by disabling anonymous access and requiring **username/password authentication** for all clients. This ensures that only authorized sensors and services can publish or subscribe to urban data streams. Mosquitto's lightweight MQTT protocol is ideal for the asynchronous, real-time communication required by this Smart City infrastructure.

Telegraf

Telegraf acts as the primary data ingestion agent. It subscribes to the MQTT broker using the inputs.mqtt_consumer plugin. A specialized **Regex Processor** was implemented within Telegraf to parse the hierarchical MQTT topics and transform them into searchable metadata (tags) such as "district", "street", and "sensor". To optimize the database schema, Telegraf is configured to exclude technical metadata (like the raw topic string) before writing the final metrics to InfluxDB.

The system uses wildcard subscriptions to remain agnostic of the city's size:

- smartcity/+/+/+

For example, the topics for a "Downtown" district would be structured as follows:

- smartcity/downtown/main_street/temperature
- smartcity/downtown/second_ave/traffic

InfluxDB

The continuous stream of urban metrics is stored in **InfluxDB**. This time-series database is optimized for high-write throughput and rapid retrieval of time-stamped data. We utilized the **Flux query language** to perform real-time data analysis and to power the visualization layer. The database is initialized via Docker environment variables to automatically create the organization and the storage bucket (smartcity-bucket) upon first deployment.

Grafana

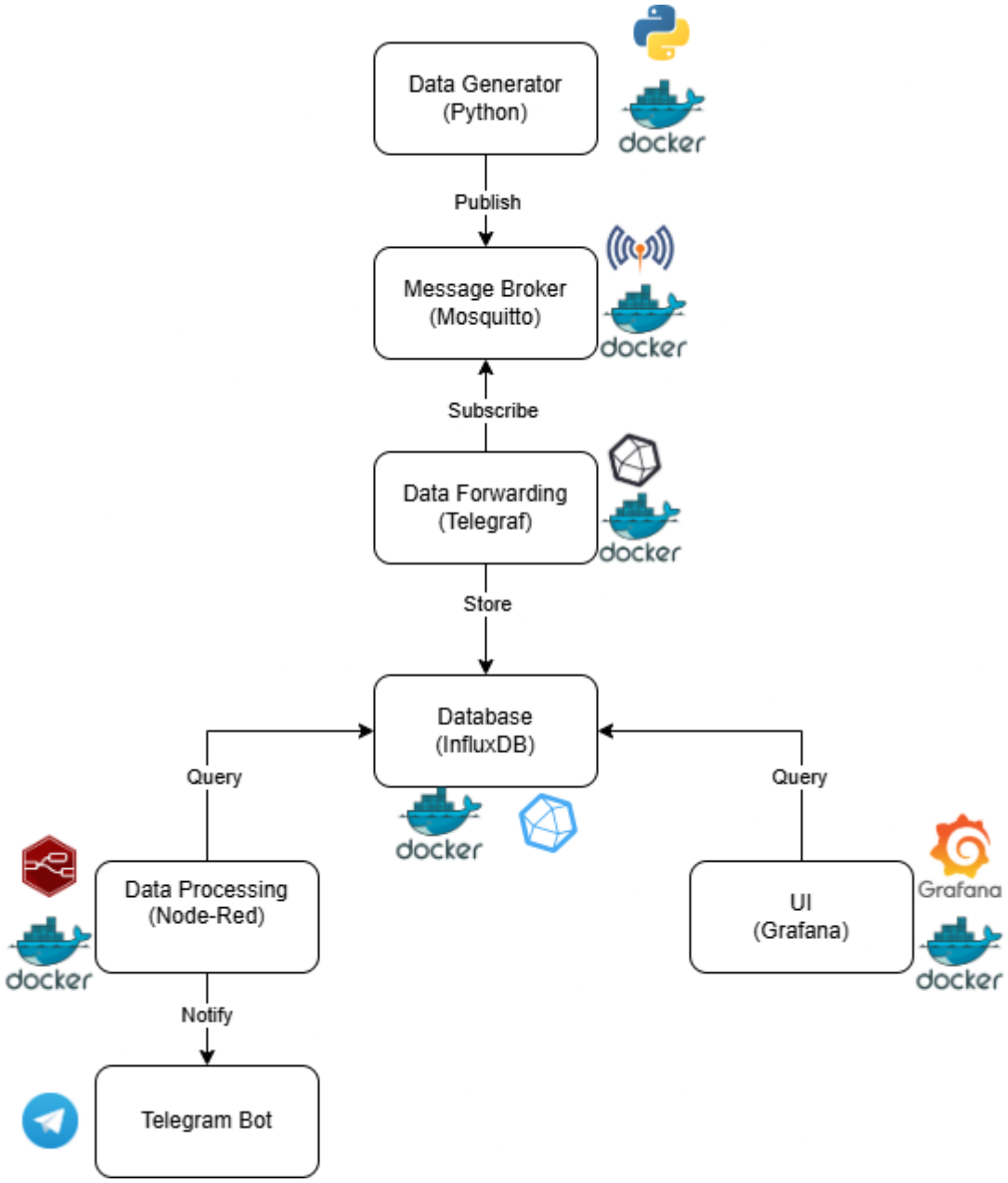
Grafana was utilized to create an advanced dashboard for real-time urban monitoring. We implemented **Dashboard-as-Code (Provisioning)**, allowing the dashboard configurations to be stored as JSON files. This approach facilitates collaborative development and ensures that the monitoring interface is automatically deployed and synchronized across different environments. The dashboard provides a comprehensive overview of city health through gauges, time-series graphs, and district-based filtering.

Nodered + Telegram

We employed **Node-RED** as the middleware logic engine. It is configured to periodically query InfluxDB for critical threshold breaches (e.g., hazardous CO2 levels or extreme traffic congestion). When an anomaly is detected, Node-RED triggers a workflow that formats a descriptive alert and transmits it via a **Telegram Bot**. The bot was created via Telegram's

"BotFather" and is integrated into Node-RED using specialized nodes. For security, the Bot Token and Chat ID are managed as environment variables, ensuring that sensitive credentials are never hardcoded into the flow logic.

6. System Architecture



The system follows a modular microservices architecture, where each component is isolated within a Docker container to ensure portability and scalability. The data flow follows a pipeline from edge simulation to storage, processing, and finally visualization.

Data Generation Layer

The "edge" of the system consists of a **Data Generator** implemented in **Python**. This component is responsible for the simulation logic of the IoT sensor network. It generates synthetic measurements and publishes them as JSON payloads to the **Mosquitto MQTT Broker**. The use of MQTT ensures asynchronous, decoupled communication between the sensors and the rest of the infrastructure.

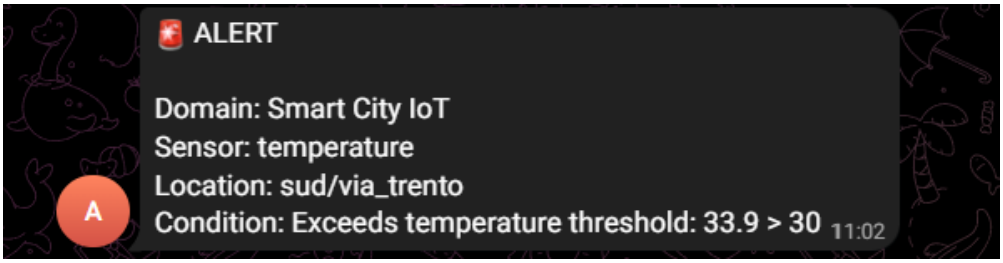
Data Ingestion and Storage Layer

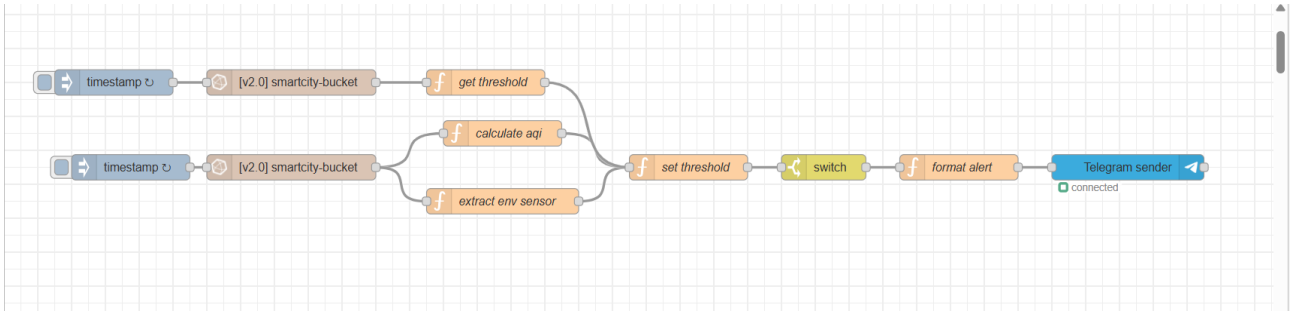
Data forwarding is handled by **Telegraf**, which acts as a specialized agent between the messaging broker and the database. Telegraf subscribes to the relevant MQTT topics, parses the incoming data, and performs tag extraction to organize the metrics. These processed data points are then persisted in **InfluxDB**, a high-performance time-series database optimized for handling rapid streams of sensor telemetry.

Data Processing and Logic Layer

The intelligence of the system resides in **Node-RED**. This component interacts with InfluxDB to query historical and real-time data. It performs two primary functions:

- **Air Quality Analysis:** It calculates complex environmental indices based on raw pollutant data and writes the results back to the database.
- **Alerting Logic:** It retrieves user-defined thresholds from the database and compares them against current readings. If a threshold is exceeded, it triggers an external integration to send instant notifications via a **Telegram Bot**.





Visualization Layer (UI)

The User Interface is implemented via **Grafana**. It serves as the centralized monitoring dashboard, providing city administrators with a real-time visual representation of the sensor data stored in InfluxDB. Through various panels, gauges, and time-series graphs, the UI translates raw database entries into actionable insights regarding city health and traffic conditions.

7. Data Visualization

The visualization layer is powered by **Grafana**, which serves as the primary interface for translating raw time-series data into actionable urban intelligence. The system utilizes a dynamic, multi-panel dashboard designed to provide city administrators with both high-level overviews and granular technical details.

The dashboard is built around a **hierarchical filtering system**, allowing users to refine the view based on the city's logical structure:

- **District Selection:** Users can isolate data for specific metropolitan areas (e.g., Centro, Nord, Sud).
- **Street-Level Drilldown:** Once a district is selected, the interface dynamically updates to allow filtering by specific streets.
- **Sensor-Specific Views:** Monitoring can be narrowed down to individual metrics like temperature, humidity level, traffic density, ambient noise or all air quality parameters.

Beyond real-time monitoring, the system implements **statistical aggregations** to provide deeper insights into urban trends. Every dashboard panel is configured to display:

- **Current Value:** The most recent reading from the IoT network.
- **Mean Value:** The average reading over a selected timeframe, useful for identifying baseline pollution or traffic levels.
- **Max/Min Values:** Essential for identifying peak congestion hours or dangerous environmental spikes.

As regards the air quality data we have a panel with the 5 main data for the calculation of the air quality which represent the average of the value of that sensor for the specified period of time.

8. Conclusion

This project demonstrates the design and implementation of a scalable Smart City monitoring system based on IoT technologies. The proposed architecture integrates real-time data generation, efficient ingestion pipelines, time-series storage, automated analysis, and centralized visualization. The use of containerized microservices, MQTT-based communication, and InfluxDB ensures portability, reliability, and scalability. Grafana provides an intuitive interface for real-time and historical data analysis, while Node-RED enables automated alerting for critical events.

Overall, the system represents a solid foundation for data-driven urban monitoring and can be easily extended to support additional Smart City services and sensors.