

Article

A Five-Layer Open-Source IoT Platform Architecture for Smart City Applications

Nikolaos Monios ^{*}, Panagiotis Papageorgas , Dimitrios Piromalis , Vasileios Cheimaras 
and Georgios Sarigiannis 

Department of Electrical and Electronics Engineering, University of West Attica, 12244 Athens, Greece; ppapag@uniwa.gr (P.P.); piromali@uniwa.gr (D.P.); vcheimaras@uniwa.gr (V.C.); g.sarigiannis@uniwa.gr (G.S.)

^{*} Correspondence: n.monios@uniwa.gr

Abstract

The rapid growth of smart cities has highlighted the need for robust, flexible, and cost-effective platforms that can support the integration and management of Internet of Things (IoT) devices, data streams, and analytics pipelines. However, recent studies have revealed a significant gap in the availability of open-source IoT platforms tailored for smart cities, creating a barrier for researchers and developers seeking to experiment, innovate, and validate connected urban applications. This article proposes a lightweight yet comprehensive open-source smart city platform architecture to address this gap. By integrating essential components such as real-time data ingestion, big data storage, messaging protocols, and data analytics, the platform offers a solid foundation for developing and testing smart city solutions. The architecture focuses on modularity, interoperability, and scalability, ensuring that the system can grow alongside the demands of urban environments. We aim to foster an open, collaborative ecosystem where researchers and developers can freely access, modify, and enhance the platform, driving forward the next generation of smart city innovations. The proposed solution serves as a bridge between theoretical research and practical deployment, reducing barriers to entry and accelerating the development of smarter, more sustainable urban systems.

Keywords: IoT; smart cities; platform; open-source; framework; edge computing; cloud computing



Academic Editors: Charalabos Skianis, Philippe Krief, Enric Pages Montanera and John Soldatos

Received: 2 July 2026

Revised: 22 July 2026

Accepted: 29 July 2026

Published: 1 August 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

The rapid evolution of Smart Cities (SC) has highlighted the need for accessible, scalable, and interoperable Internet of Things (IoT) platforms to support the development and testing of connected urban applications. However, recent research has identified a significant gap in the availability of open-source IoT platforms specifically designed for smart city ecosystems, as well as limitations of proprietary IoT platforms for smart city, or smart industry, or smart agriculture use cases, regarding multi-protocol and multi-technology support, cyber-security features, device analytics, and more [1]. This lack of accessible tools impedes innovation, particularly for researchers and developers who require flexible, cost-effective solutions for prototyping and evaluating smart city applications.

IoT devices utilise their integrated sensors and communication modules to facilitate real-time data acquisition, monitoring, and—if necessary—control across a wide range of infrastructures. Nonetheless, exclusive reliance on a centralised IoT network architecture

presents significant challenges, including increased latency, bandwidth limitations, and reduced system resilience.

To address these issues, various frameworks have been proposed over the years, to segment and classify the key components of IoT platforms, thereby facilitating the design of IoT solutions across diverse environments (e.g., urban, industrial, agricultural). This segregation is typically achieved through abstraction pillars, commonly referred to as *layers*, while classification is performed using *blocks* representing core components (e.g., sensors, services, databases). Researchers and developers can then model data flows either laterally between components or vertically across layers, depending on the use case.

SC deployments span a broad spectrum of applications, including wireless connectivity infrastructure, smart homes and buildings, e-governance and public services, urban management systems, environmental monitoring, intelligent transportation, smart parking, healthcare digitisation, smart tourism, energy grid optimisation, waste management, and public surveillance, among others. Notably, IoT platforms developed for adjacent domains—such as smart agriculture, industrial IoT, and asset tracking—share a largely equivalent functional architecture with their smart city counterparts, encompassing ETL data pipelines, data storage, visualisation, and analytics capabilities. The primary distinction between these domains lies not in their underlying platform functionalities, but in the specific use cases they address and the users they serve.

The main contributions of this work are as follows: (i) the synthesis of existing smart-city IoT reference architectures into a minimal five-layer architecture, comprising the Physical, Edge, Cloud, Application, and Security layers; (ii) the implementation of this architecture using exclusively open-source software components, including EdgeX Foundry (Odessa), Apache NiFi (v1.27.0), Telegraf (v1.30), Mosquitto (v2.0), InfluxDB (v1.80), and Grafana (v10.2.2); (iii) the deployment of the resulting platform on two heterogeneous edge devices, namely a Raspberry Pi (Raspberry Pi Ltd., Cambridge, UK) and an NVIDIA Jetson Nano (NVIDIA Corporation, Santa Clara, CA, USA), to demonstrate portability and resource adaptability across different hardware profiles; (iv) the functional validation of the platform through end-to-end testing of data transmission, cloud synchronisation, dashboard-based visualisation, and baseline security mechanisms, including TLS encryption, X.509 certificate-based authentication, and user-level access control; and (v) the provision of the platform as both a reference architecture and a reproducible implementation guide, with the former offering a reusable, layer-based conceptual model adaptable to different smart city use cases and technology stacks, and the latter providing a concrete, publicly available deployment based on open-source tools and low-cost hardware, accompanied by all source code, firmware, and configuration artefacts in a public repository.

State-of-the-Art Reference Architectures for IoT Platforms

Panagiotakopoulos et al. present a FIWARE-based IoT framework for a smart water distribution management use case within a SC environment. The framework is structured into three layers: the Physical layer, the Middleware layer, and the Application layer. The blocks within the Physical layer concern the actual “things” involved in the use case—namely, sensors, actuators, and the communication technologies that facilitate data transmission to the subsequent layer. In the Middleware layer, the authors include blocks representing the FIWARE data broker, databases, and a FIWARE data agent that bridges different communication protocols. Finally, the Application layer contains blocks dedicated to data visualisation for authorised users [2].

Similarly, Ajao and Apeh, in their work on developing a secure mechanism for edge computing in smart cities, propose a three-layer architecture comprising the Edge, Fog layer, and Cloud layers. In this architecture, data aggregation from IoT devices takes place

at the Edge layer. Logical connectivity, data analysis, and control responses are handled within the Fog layer. Finally, big data processing and business logic intelligence are carried out in the Cloud layer [3].

In [4], the authors propose a Vehicle-to-Everything (V2X) Software-Defined Internet of Vehicles (SDIoV) architecture based on 5G—an additional use case within the IoT domain. Their proposed architecture is structured into three layers: the Infrastructure layer (which includes the vehicles), the Control layer (which encompasses the network and associated protocols), and the Application layer.

Whaiduzzaman et al. propose a SC architecture comprising three layers: the “Data Monitoring and Actuation” level, the “Data Transfer” level, and the “Data Processing” level. In the first level, the authors include people, physical objects, infrastructure, and vehicles. The second level encompasses both wired and wireless communication technologies, while the final level consists of the smart services corresponding to each SC use case [5].

In [6], the authors expand the architecture of a SC framework into four layers: the *Sensing* layer, the *Network* layer, the *Cognition* layer, and the *Applications* layer. The key addition compared to previous architectures is the *Cognition* layer, which hosts all Machine Learning (ML) and Artificial Intelligence (AI) functionalities, including configuration, training, and optimisation.

Mahmudov and Temirova [7] also present a four-layer framework, comprising the *Sensing* layer, the *Transmission* layer, the *Data Processing* layer, and the *Application* layer. The *Sensing* layer includes the sensors relevant to the SC use case, while the *Transmission* layer consists of the access and transmission networks. In the *Data Processing* layer, the authors incorporate databases, AI and decision-making models. Finally, the *Application* layer contains the smart applications that utilise data from both the *Sensing* and *Data Processing* layers.

Faliagka et al. further expand the SC framework to five layers in their work on smart mobility within SC environments. Their proposed architecture includes a *Data Acquisition* layer, an *Edge Computing* layer, a *Data Storage* layer, a *Digital Twin* layer, and an *Application* layer. Data are collected from various sources, including IoT devices, vehicles, and human inputs via mobile applications, simulators, and third-party services. The *Data Acquisition* layer manages this intake and channels the information through real-time processing systems for filtering, routing, and transformation prior to storage. The *Edge Computing* layer ensures the execution of decisions made by higher-level systems by directly interacting with devices. At the core, the *Digital Twin* layer comprises interconnected microservices responsible for system intelligence, decision-making, and actuation. This layer leverages AI/ML techniques and simulation models to analyse data, predict outcomes, and issue commands to the *Edge Computing* layer. Finally, the *Application* layer encompasses user-facing interfaces that support interaction with system analytics, scenario simulations, and decision-support tools [8].

Lynn and Wood [9], propose a SC framework for smart streets, identifying five fundamental entities: *Social Actors*, *Artifacts*, *Networks*, *Places*, and *Infrastructure*. *Social Actors* include individuals, groups, or computational agents that can share a cognitive framework and engage meaningfully with other entities. These actors may assume multiple roles (e.g., residents, travellers) and operate within diverse organisational structures, whether public or private. *Artifacts* refer to physical or digital objects (e.g., sensors, vehicles, wearables) that enable value creation through interaction with social actors. *Networks* are interconnected systems, whether social or technological, that facilitate communication and coordination, with streets serving as essential connectors within these systems. *Places* are socially constructed, meaningful domains formed through repeated interactions and experiences, and are distinct from purely spatial locations; they may exist in both physical

and digital realms. *Infrastructure* encompasses the underlying physical, digital, and organisational systems that support the sustainable operation of the street, and often collect data passively. Each entity may exist in either physical or cyber form, and may itself represent a system. Together, these entities define the smart street as a system of systems, underscoring its complexity and interdependence.

In [10], the authors propose a five-layer SC framework consisting of: (i) the *Sensing* layer, which includes IoT devices, smartphones, and gateways responsible for data collection; (ii) the *Network* layer, which handles communication protocols; (iii) the *Middleware* layer, comprising databases, microservices, and event management systems; (iv) the *Application* layer, which provides user interfaces through mobile apps, web portals, and third-party systems; and (v) the *Security* layer, responsible for authentication, encryption, and overall data protection. Notably, the *Security* layer spans all other layers, directly interacting with each, underscoring its central importance to the framework's design.

Garcia-Font, in his proposal for SC IoT framework utilising Distributed Ledger Technologies (DLTs) to enable decentralised and user-centric systems, presents a five-layer architecture. This framework includes the *Application* layer, the *Network* layer, the *Data Processing* layer, and two DLT-related layers—the first being the *Blockchain* layer, and the second comprising scalability solutions built on top of the foundational DLT layer [11].

Corallo et al., in their work on a reference IoT architecture for smart industry—incorporating digital twins—propose a six-layer architecture designed to facilitate the integration of the two fundamental domains of such use cases: the physical and digital spaces. At the bottom of the physical space lies the *Physical* layer. In the digital space, the *Network* layer extends from the top to the bottom, acting as a bridge between the two domains. The digital space also includes the *Data Integration* layer, the *Model* layer, and the *Knowledge* layer. The architecture concludes with the *Application* layer at the top [12].

Bokolo and Petersen propose a seven-layer reference architecture for SCs, comprising the following layers: *Context*, *Service*, *Business*, *Application and Data Processing*, *Data Space*, *Technologies*, and *Physical Infrastructures*. The architecture integrates the following components: The *Physical Infrastructures* layer includes citywide physical assets such as sensors, IoT devices, and RFID tags that generate real-time data. The *Technologies* layer provides the necessary hardware and software infrastructure—such as cloud computing platforms and big data tools—for the collection, processing, and storage of this data. The *Data Space* layer manages all data types, including real-time, processed, historical, and third-party data, supporting both relational and non-relational databases. The *Application and Data Processing* layer hosts applications that convert raw data into actionable insights and provide digital services. The *Business* layer involves enterprises and partners responsible for service delivery and operational coordination. The *Service* layer defines the city's strategic action plans, allocates resources, and outlines high-level services to meet performance objectives. Lastly, the *Context* layer captures stakeholder needs, goals, key performance indicators (KPIs), and guiding principles to ensure that smart city initiatives align with citizen interests [13].

Cumbajin et al., in their work on the application of digital twins in smart industry environments for quality control purposes, present a four-layer architecture. At the *Real World* level, the architecture includes machines, robots, assets, and human operators. The *Edge* level is responsible for aggregating data from the Real World level, processing it, and transferring it to the *Cloud* level. Finally, the *Digital* level is responsible for data visualisation and analytics [14].

Kolhar et al. present a three-layer architecture designed to support the enforcement of restrictions during the COVID-19 pandemic in SCs. A key component of their proposal is the *Edge* layer, which aggregates data from the *Device* layer, stores it locally, applies AI

algorithms, and controls actuators. The collected data are visualised within the *Application* layer, which directly queries the databases hosted on the edge devices [15].

Jana, Pal, and Kumar propose a five-layer architecture for SCs, comprising the *Edge with IoT Devices* layer, the *Fog* layer, the *Cloud* layer, the *Application* layer, and the *Security* layer. The implementation of the architecture is based on microservices, enhancing both the scalability and flexibility of the platform [16].

Khalil et al., in their work on the authentication of IoT-enabled devices in SCs using blockchain technologies, propose a four-layer architecture comprising: (i) the *Sensing* layer, which effectively combines the functionalities of both the *Edge* and *Fog* layers and includes data collection from smart devices as well as actuator control; (ii) the *Transmission* layer, responsible for handling inbound and outbound data traffic through various wireless communication technologies; (iii) the *Blockchain* layer, where smart contracts and transaction processing take place; and (iv) the *Application* layer, in which the smart city use cases are implemented [17].

The inclusion of the *Edge* layer within smart city architectures is a growing trend, as highlighted in the comprehensive review conducted by Khan et al. in their work on edge computing in smart cities [18], where the authors analyse the evolution of edge computing paradigms and their role in SC applications. Furthermore, this trend is reinforced in the recent review by Sanchez-Callegos et al. on emerging technologies in smart cities, where numerous multi-layered SC architectures incorporating edge computing are presented and discussed [19].

In conclusion, the design of a reference architecture for an IoT SC platform is an important part of the solution, facilitating the integration of key components and identifying missing interactions or components. In a reference architecture, the solution is segregated into layers, and each layer features various components, either physical or digital, that—when combined—form a data flow that realises the solution.

2. Platform Architecture

In this section, we propose a reference architecture for a basic SC IoT platform, using minimal weather stations as a use case, based on open-source software, hardware, and low-cost sensors that monitor basic environmental data (temperature and humidity) within an area of interest. The proposed architecture is derived from the state-of-the-art review discussed earlier.

The surveyed architectures presented earlier, reveal a set of recurring functional layers that can be considered a superset from which any individual SC IoT architecture draws. These layers are: (i) the *Physical/Device* layer, encompassing sensors, actuators, and embedded devices; (ii) the *Edge/Fog* layer, responsible for local data aggregation and real-time processing; (iii) the *Network/Transmission* layer, handling communication protocols and routing; (iv) the *Middleware/Data Integration* layer, comprising brokers, event management, and microservices; (v) the *Cloud* layer, providing remote storage and scalable compute; (vi) the *Data Processing/Cognition* layer, hosting AI/ML models and decision-making logic; (vii) the *Digital Twin* layer; (viii) the *Blockchain/DLT* layer; (ix) the *Application* layer, delivering user-facing services and dashboards; and (x) the *Security* layer, responsible for authentication, encryption, and data protection. Figure 1 illustrates this superset, highlighting the layers incorporated in the proposed architecture.

The proposed five-layer architecture, comprising the *Physical*, *Edge*, *Cloud*, *Application*, and *Security* layers, draws from this superset while deliberately omitting layers that are beyond the scope of a general-purpose prototype deployment, such as dedicated *Network*, *Middleware*, *AI/ML* processing, *Digital Twin*, and *Blockchain* layers. The latter three are identified as directions for future work, particularly the integration of *AI/ML*

inference at the Edge layer. The primary architectural novelty of the proposed platform lies in treating security as a first-class, omnipresent layer. Rather than addressing security as an implicit concern or an afterthought, the Security layer in this architecture spans and enforces authorisation and encryption across every data transaction; from sensor-to-broker communication secured via mutual TLS, through authenticated ingestion at the synchronisation services, to access-controlled storage and visualisation at the cloud layer. This design philosophy is shared by only two of the nineteen surveyed works, making it a distinguishing characteristic of the proposed platform.

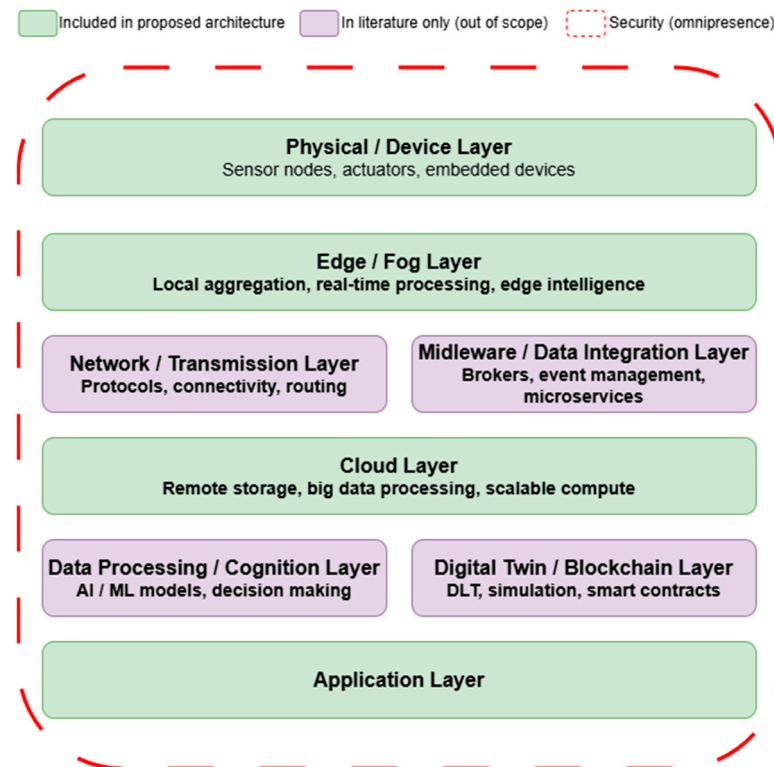


Figure 1. Superset of layers in the proposed architecture and of the architectures found in the literature. The Security layer is explicitly present in 2 of 19 surveyed architectures. A key novelty of the proposed platform 10 distinct functional layers identified across surveyed works.

It is worth clarifying how the proposed Security layer relates to the two surveyed works discussed in Section “State-of-the-Art Reference Architectures for IoT Platforms” that also incorporate a dedicated security layer. In [10], the Security layer spans all other layers of a five-layer SC framework, a structural property that is conceptually identical to the omnipresent role assigned to the Security layer in the proposed architecture. However, the work in [10] remains a theoretical framework: no implementation is provided, no concrete security mechanisms are specified at the component level, and no validation of the security posture is reported. In contrast, the proposed platform operationalises this concept through specific, verifiable mechanisms (mutual TLS with per-device X.509 certificates at the MQTT broker, authenticated data ingestion at the synchronisation services, and access-controlled storage and visualisation at the cloud layer) all of which are validated through a live deployment and made reproducible via a public repository. The second work in [16] proposes a five-layer microservices-based architecture that also includes a Security layer. However, in that work the Security layer is listed as one of five parallel layers rather than as a transversal, omnipresent concern, and again no implementation or security validation is provided. The proposed platform therefore advances beyond

both works by combining the architectural intent of an omnipresent Security layer with a concrete, deployable, and validated realisation of that intent.

Building on the superset presented above, we propose a five-layer reference architecture tailored to the SC IoT platform described in this work. The five layers (Physical, Edge, Cloud, Application, and Security) represent the minimal structure required for a basic deployment of the described use case, as illustrated in Figure 2. Within this architecture, the Security layer is not confined to a single tier; instead, it spans all remaining layers, enforcing authentication and encryption at every point of data exchange. The architecture is designed to be extensible: as the use case grows in complexity, additional layers from the superset, most notably a dedicated AI/ML processing layer at the Edge, can be incorporated without restructuring the existing five-layer foundation.

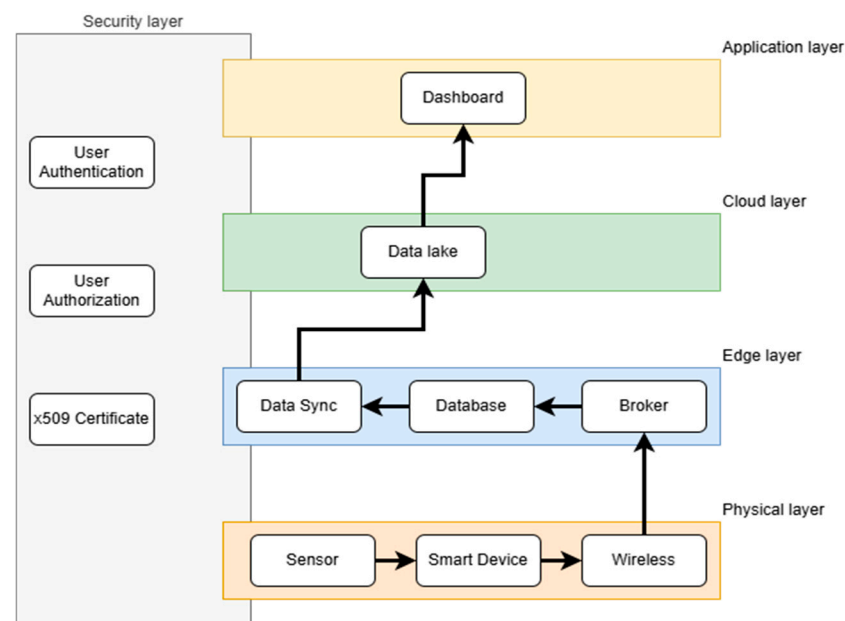


Figure 2. Reference Architecture for a minimal SC IoT Platform.

In the *Physical* layer, we define the smart device (comprising both hardware and software), the sensors, and the wireless communication module of the sensor node in our platform. Each smart device will include, at least, these three components. We also designed the basic data flow; from the moment the sensor generates data to the point of transmission to the next layer. In a more complex use case, for instance, one that involves not only a smart device monitoring temperature and humidity but also another device measuring air quality, the Physical layer could be restructured to group components by function, forming separate subsystems within the layer. Wired Internet connectivity is also supported if a wireless module is unavailable.

The *Edge* layer receives data wirelessly from the smart devices. It includes a broker component responsible for ingesting data from the smart devices and storing them in a local database. A Data Sync service is responsible for securely transferring data from the database to the Cloud layer. The local database can be organised as a data warehouse, ready for use by the edge device and/or its connected devices. These three basic blocks constitute the Edge Stack; a combination of services that facilitate the aggregation of data from neighbouring devices, the storage of this data into a local database, and the transfer of batches of data from the database to a remote data lake.

The Cloud layer incorporates the data storage component, which aggregates time-series data from all edge devices and organises it into a centralised repository, ready for use by applications.

In this use case, the *Application* layer serves as a basic dashboard. It retrieves data from the cloud layer's data lake and presents it to users through visualisations.

The *Security* layer spans across all other layers, emphasising the critical importance of secure data transmission throughout the architecture. We identified three key components that provide basic security functionalities: a *user authentication* component, a *user authorisation* component, and an *X.509 certificate* component.

This architecture addresses the most basic smart city use case, monitoring environmental data (temperature and humidity) and visualising it on a dashboard. For each layer, we identified the necessary components that, when combined, enable the solution's implementation.

Open-Source Implementation Proposal

In this section, we propose an implementation of the platform that relies entirely on open-source software and tools, and widely available low-cost hardware. In Figure 3, we present the proposed implementation architecture, in which the abstract blocks from the reference model are replaced with specific open-source components.

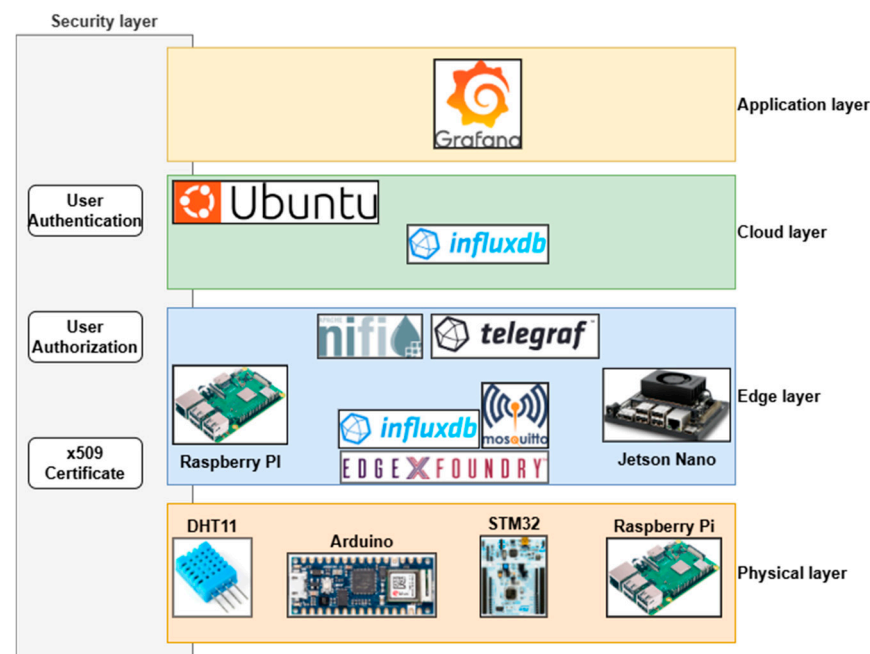


Figure 3. A SC IoT architecture proposal using only open-source software and tools (Platform and Sensor Nodes).

At the *Physical* layer, we suggest the popular DHT11 temperature and humidity sensor for the sensor block of the proposed architecture. We also suggest several candidate development boards that can interface with the sensor to construct the sensor node, including the Arduino Nano ESP32 (Arduino S.r.l., Monza, Italy), a STMicroelectronics (STM32) Discovery board (STMicroelectronics, Geneva, Switzerland), and even a Raspberry Pi single-board computer (SBC); even though the Raspberry Pi was not used in the demonstration that will be presented later, we still use it here to show that any widely used low-cost hardware can be utilised. These examples are indicative and do not limit the selection of other compatible development boards.

The sensor nodes will periodically sample data from the environmental sensors and transmit the samples to the *Edge* layer wirelessly using the MQTT protocol. A popular open-source MQTT library for embedded systems is Paho, which is used in this implementation on the STM32 Discovery board.

The edge device is a key component of the *Edge* layer. Although numerous candidates exist for the role of the edge device, we present here two examples: the Raspberry Pi single-board computer (SBC) and the NVIDIA Jetson Nano. On both edge devices, we suggest the EdgeX Foundry software, a collection of open-source, interconnected, containerised (Docker) microservices that form a fully functional Edge Stack in the IoT domain.

EdgeX Foundry facilitates both horizontal data transfer (sensor-to-cloud) and lateral data transfer (edge-to-edge) by providing a suite of microservices that enable these functionalities. For instance, it offers a fully customisable Mosquitto MQTT broker, a Redis database, a Consul container for service discovery, and service mesh capabilities, among others.

EdgeX Foundry is a hardware and operating-system-agnostic edge platform for IoT, with minimal deployment resource requirements. It requires at least 1 GB of RAM and 3 GB of storage, and supports the following operating systems [20]:

- Windows 7, or later.
- Ubuntu Desktop/Server 14, or later.
- Ubuntu Core 16, or later.
- MAC OS X.

For these reasons, EdgeX Foundry is proposed as the edge component. Its open-source nature, combined with its hardware- and OS-agnostic capabilities, makes it an excellent tool for a fully open-source smart city IoT platform, facilitating the design, development, and deployment of IoT-based solutions.

Although Redis can be used directly, we propose the open-source time-series database InfluxDB for our implementation, as it better serves our purpose for this minimal scenario. It is worth mentioning, though, that any database can be used within the EdgeX Foundry ecosystem. In more complex use cases, other databases can be used as well, e.g., MongoDB, and PostgreSQL.

Furthermore, we propose two data synchronisation services responsible for copying data from the local InfluxDBs to the Cloud Layer's data lake: Telegraf and Apache NiFi. Telegraf is a data forwarding service for InfluxDB; it listens for incoming data and pushes it to predefined databases, data brokers, endpoints, etc. Apache NiFi is an open-source Extract–Transform–Load (ETL) tool that facilitates the flow of data between various systems. Other possible Data Sync services for this implementation could be Airbyte and Apache Airflow.

In the *Cloud* layer, the proposed host OS for the architecture's data lake is Ubuntu. Here, there are numerous other Operating Systems that could be used, but Ubuntu offers great long-term support and a vibrant community that helps anyone facing issues to overcome them. InfluxDB is proposed as the data lake component due to the simplicity of the use case: simple sensor nodes transmit a small payload consisting of 2 measurable physical units; there is no metadata or media (sound, photos, videos, etc.) in to the payload. Again, any other database can be used as a data lake, for example MongoDB or Hadoop, but it all depends on the use case and its scale; for a minimal scenario like this, InfluxDB is more than sufficient.

Finally, in the *Application* layer, Grafana is proposed as the dashboard interface, enabling secure, seamless integration and interaction with InfluxDB instances. The dashboard will query the InfluxDB data lake from the *Cloud* layer and visualise the data for authorised users.

In Table 1 we summarise the hardware, software, tools, and services we propose for implementation. Furthermore, we offer alternative open-source options to highlight that our suggestions do not limit the implementation.

Table 1. Open-source components proposals and suggested alternatives.

Component	Open-Source Suggestion	Open-Source Alternative
Sensor Node Devices	Arduino Nano ESP32, STM32 Discovery, Raspberry Pi	ESP8266, TI Launchpad, NXP FRDM, Raspberry Pi Pico, Orange Pi, Beaglebone Black
Communication Protocol	MQTT	Apache Kafka, REST API, CoAP, OPC-UA
Edge Devices	Raspberry Pi, NVIDIA Jetson Nano	Google Coral, UP Edge, Asus Tinker Edge R, LattePanda Sigma
Messages Broker	Mosquitto	Apache Kafka, EMQX
Data Warehouse Databases	InfluxDB	Redis, MongoDB, PostgreSQL
Data Sync Mechanism	Telegraf, Apache NiFi	Airbyte, Apache Airflow
Edge Stack	EdgeX Foundry	Eclipse Kura, thin-edge.io
Cloud OS	Ubuntu	Linux Mint, Fedora, Debian
Data Lake Databases	InfluxDB	Redis, MongoDB, PostgreSQL, Hadoop
UI/Dashboard	Grafana	OpenObserve, SigNoz

The *Security* layer spans the entire architecture, from bottom to top, signifying that every data transfer must be secured and authorised; otherwise, it is prohibited. For example, sensor nodes will utilise X.509 certificates when transmitting data to the MQTT broker, and data transfers to the data lake will be conducted over HTTPS using username and password authentication.

To further contextualise the contribution of the proposed platform, Table 2 presents a feature-level comparison against three widely adopted open-source IoT platforms, ThingsBoard, FIWARE, and OpenRemote. These platforms were selected as representative comparators due to their prominence in the smart city literature and their frequent citation as reference implementations [1]. While all three platforms offer robust device management, data collection, and visualisation capabilities, they differ from the proposed platform in several key dimensions. Firstly, none of the three platforms explicitly model security as a dedicated architectural layer; instead, security is handled through external identity providers, reverse-proxy TLS termination, or optional certificate-based authentication modes. Secondly, their deployment complexity and opinionated integrated-stack design make it less straightforward to substitute individual components, whereas the proposed platform is intentionally composable, allowing each block to be replaced with alternatives (as shown in Table 1). Thirdly, none of the surveyed platforms provide a fully reproducible, end-to-end deployment targeting heterogeneous low-cost edge hardware, together with sensor node firmware for multiple MCU families, in a single public repository. These distinctions position the proposed platform not as a competing solution for large-scale deployments, for which ThingsBoard or FIWARE remain more suitable, but as a minimal, reproducible, and security-conscious foundation for research prototyping and educational deployment in smart city contexts.

Table 2. Comparison of the proposed platform against established open-source IoT platforms.

	ThingsBoard	FIWARE	OpenRemote	Proposed Platform
Fully open-source	+ (CE edition)	+	+	+
Smart cities focus	+	+	+	+
Dedicated edge layer	+ (ThingsBoard Edge)	Partial (FogFlow)	+ (Edge gateway)	+ (EdgeX Foundry)
Heterogeneous hardware support	Partial	Partial	Partial	+
mTLS/X.509 per device authentication	Partial (token/certificate)	- (application-level only)	Partial (TLS via proxy)	+ (enforced at broker)

Table 2. Cont.

	ThingsBoard	FIWARE	OpenRemote	Proposed Platform
Security as architectural layer	-	-	-	+
Component-level composability	- (integrated stack)	+	Partial	+
Docker-based deployment	+	+	+	+
Public reproducible deployment	-	-	-	+(GitHub repository)
Setup complexity	High	High	Medium	Low-Medium

3. Platform Deployment and Testing

Having discussed the selected open-source components of the proposed architecture, this section presents the data flow from the sensor nodes to the User Interface (UI), as well as the deployment and functional validation of the proposed platform.

3.1. Experimental Setup

The experimental deployment was conducted using two edge devices: a Raspberry Pi single-board computer and an NVIDIA Jetson Nano. Both devices ran Linux-based operating systems and were connected to the same local network via Wi-Fi; in the case of Nano, a USB Wi-Fi dongle was used to enable internet connectivity. The setup includes two sensor nodes; an Arduino Nano ESP32 board and an STM32 Discovery board. The Arduino Nano ESP32 Sensor node periodically (every 15 s) samples temperature and humidity data from a DHT11 sensor and transmits the measurements to the Raspberry PI edge device via MQTT. On the other hand, an STM32 Discovery board utilises its embedded digital sensors periodically (15 s), which include temperature and humidity amongst others, and transmits its data to the NVIDIA Jetson Nano edge device via MQTT again. In the Cloud, we have deployed an InfluxDB that receives data from edge devices into a single collection, and a Grafana dashboard that visualises the aggregated data from the cloud database. The entire data flow is depicted in Figure 4.

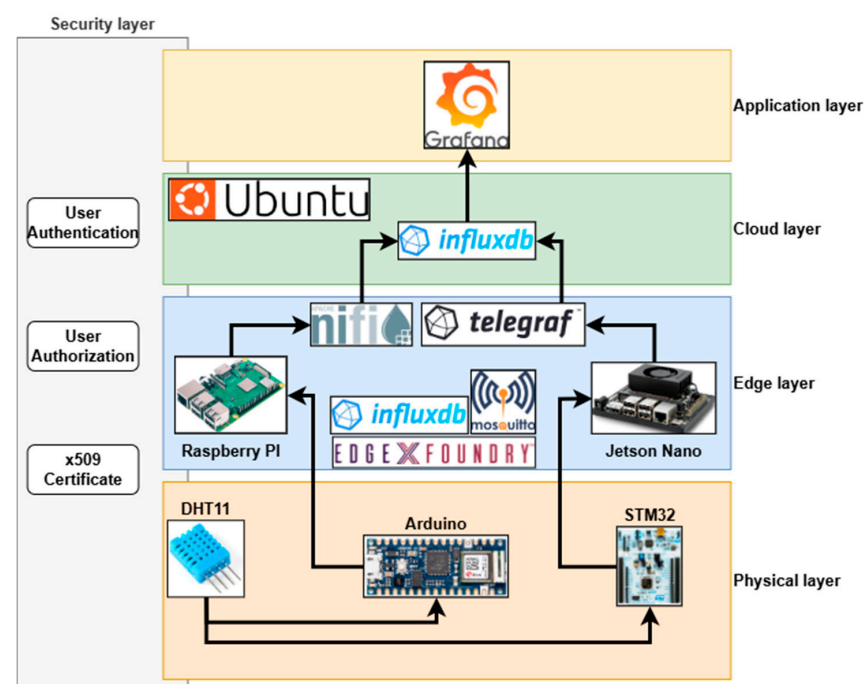


Figure 4. Data flow in a simple SC scenario.

The Cloud layer was hosted on a VirtualBox virtual machine configured with a single vCPU and 4 GB of RAM, running Ubuntu 18.04 LTS, on which InfluxDB and Grafana were deployed as Docker containers.

Table 3 summarises the key hardware specifications of the four devices used in this deployment. The reader may notice that the NVIDIA Jetson Nano lacks wireless capabilities; thus, we used a USB Wi-Fi dongle to enable the Nano to communicate wirelessly on the network. Alternatively, the Nano can connect to the internet via its Ethernet port.

Table 3. Technical specifications of the hardware used in the experimental setup.

	Arduino Nano ESP32	STM32 Discovery B-L475E-IOT01A2C	Raspberry Pi 3 Model B+	NVIDIA Jetson Nano
Role	Sensor node	Sensor node	Edge device	Edge device
OS/Firmware framework	Arduino (Platform.IO)	bare-metal/HAL	Debian GNU/Linux 12 (bookworm)	JetPack/L4T
MCU/SoC	u-blox NORA-W106-10B	STM32L4—ARM Cortex M4	ARM Cortex A53	NVIDIA Tegra X1 (Arm Cortex A57)
GPU	N/A	N/A	N/A	128-core NVIDIA Maxwell
Frequency	240 MHz	80 MHz	1.4 GHz	1.43 GHz
Cores count	2	1	4	4
RAM	8 MB	128 KB	1 GB	4 GB
Flash	16 MB	1 MB	N/A	N/A
SD/eMMC	N/A	SD	32 GB SD	32 GB SD
Power supply	5V USB	5V USB	5V charger	5V charger
Power consumption	~0.07–0.8 W	0.07 W	5.1 W	10 W
USB	USB-C	Micro USB	4 × USB 2.0	4 × USB 3.0
Wireless	1 × Wi-Fi 4 (802.11b/g/n) 1 × Bluetooth 5.0 (BLE support)	1 × Wi-Fi 4 (802.11b/g/n) 1 × 868 MHz RF 1 × Bluetooth 4.1 1 × NFC	1 × Dual-band 802.11 b/g/n/ac 1 × Bluetooth 4.2	N/A (requires external Wi-Fi adapter)
Wired	2 × UART 1 × I2C 1 × SPI	Arduino Pinout 1 × UART 2 × I2C 1 × SPI	1 × Ethernet 300 mbps	1 × Gigabit Ethernet port
Extras	8 × analog pins 5 PWM pins	2 × digital omnidirectional microphones 1 × digital temperature & humidity sensor 1 × 3-axis magnetometer 1 × 3D accelerometer & 3D gyroscope 1 × digital barometer 1 × Time-of-Flight and gesture-detection sensor	40 GPIO pins HDMI port 3.5 mm audio jack CSI interface Display interface	40 GPIO pins HDMI port DisplayPort CSD interface

Table 3 also reports the nominal power consumption of each device as specified by the respective manufacturer datasheets. The sensor nodes exhibit low power draw in their active state (~0.07–0.8 W), with transient peaks during Wi-Fi transmission. The edge devices consume significantly more power (~5.1–10 W), reflecting their role as always-on processing nodes. These figures are provided as indicative values for deployment planning purposes; accurate in situ power profiling under representative workloads would be required for energy-constrained deployments.

To demonstrate the low-cost nature of the proposed platform, Table 4 presents approximate retail prices for each hardware component at the time of writing, sourced from various major electronics distributors. The total hardware cost for a minimal single-edge-device deployment—comprising one sensor node (Arduino Nano ESP32 with DHT11 sensor) and one edge device (Raspberry Pi 3B+)—is estimated at approximately €62–95, excluding cloud hosting costs, which depend on the chosen provider and deployment scale. Cloud hosting costs were not included in this estimate, as they are highly dependent on the chosen provider, virtual machine configuration, and deployment duration.

Table 4. Hardware bill of materials cost for the experimental setup.

	Arduino Nano ESP32	STM32 Discovery B-L475E-IOT01A2C	Raspberry Pi 3 Model B+	NVIDIA Jetson Nano	DHT11
Approximate Retail Price (Euros)	16–20	50–70	45–55	90–130	2–5
Total (Euros)	203–280				

The source code for both sensor nodes has been uploaded to a dedicated GitHub repository created to validate the proposed architecture. The source code for the Arduino Nano ESP32 was compiled using the popular Platform.IO (v6.1.19) plugin for Visual Studio Code (v1.129.1), whereas the source code for the STM32 Discovery board was compiled on the STM32CubeIDE (v2.1.1), which is based on Eclipse. We hope this repository will enhance the reproducibility of our proposal and will enable readers to replicate it on their own premises. The link for the GitHub repository is the following: https://github.com/nikmonios/Open_Source_Smart_City_Platform (accessed on 21 July 2026).

The repository also includes a certificate generation script (certs/generate_certs.sh) that automates the creation of the CA, broker, and per-device client certificates required for the mutual TLS configuration (described later in Section 3.3), enabling complete replication of the security setup from a fresh environment.

3.2. Platform Deployment

As discussed earlier, EdgeX Foundry is a fully open-source platform. For deployment, the EdgeX Foundry organisation provides a set of YAML configuration files hosted on GitHub (<https://github.com/edgexfoundry/edgex-compose>, accessed on 12 February 2026) that are readily available for use.

For our scenario, we adopted the docker-compose-zero-trust.yml deployment file as a baseline and extended it by incorporating the additional services proposed in our architecture. Unnecessary services for our minimal use case (e.g., Redis) were removed, new services were added (InfluxDB, NiFi, Portainer), user credentials were configured, and mounted volumes were defined to include the necessary certificates for the MQTT broker.

Given that two different edge devices with distinct service requirements were employed, two separate deployment files were created—one for each. In the Jetson Nano deployment, Telegraf was added as a service to enable data synchronisation with the Cloud layer. In contrast, in the Raspberry Pi deployment, Apache NiFi was used to provide equivalent functionality. Following this configuration, each YAML file was deployed, triggering Docker to download the required service images and execute them as containers. The structural diagram depicting the layered dependencies is illustrated in Figure 5.

The selection of two distinct data synchronisation tools (Telegraf on the NVIDIA Jetson Nano and Apache NiFi on the Raspberry Pi) was motivated by the differing operational characteristics of each edge device and the design goal of demonstrating the platform's component-level composability.

Telegraf was selected for the Jetson Nano due to its lightweight, single-binary deployment model and configuration-file-driven operation, which minimises resource overhead on a device whose computational budget is primarily reserved for potential future AI/ML inference workloads at the Edge layer.

Apache NiFi was selected for the Raspberry Pi owing to its visual flow-based programming interface and its native support for flexible data transformation pipelines, which proved advantageous in handling the more complex payload structure of the STM32 Discovery board (a JSON object comprising thirteen fields which will be presented in Section 3.3) without requiring custom preprocessing code. It is important to note that the two tools

were not deployed as competing solutions for the same task; rather, each was matched to the edge device and sensor node combination for which it was best suited.

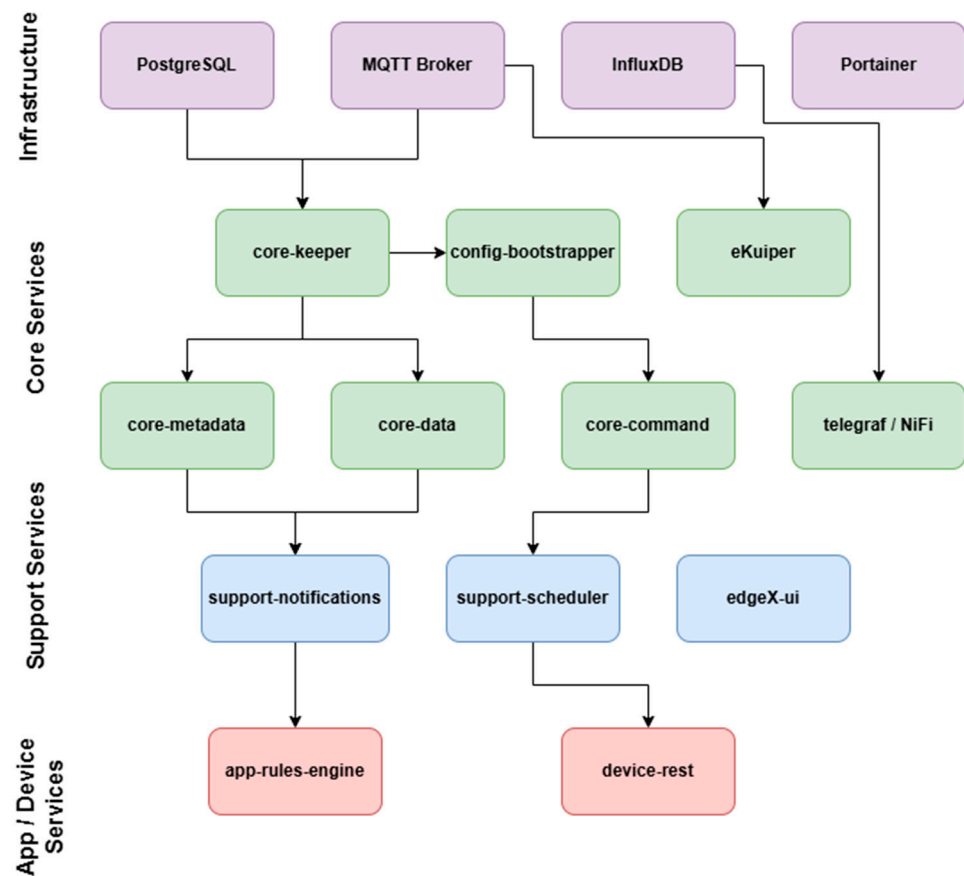


Figure 5. Architecture of the deployed EdgeX Foundry stack in the edge devices.

A formal performance comparison between Telegraf and NiFi (in terms of throughput, latency contribution, and resource consumption) was outside the scope of this prototype deployment and is identified as a direction for future work, alongside the evaluation of additional synchronisation alternatives such as Node-RED or custom MQTT-to-InfluxDB bridge implementations.

The YAML deployment files, the Mosquitto broker configuration files, the Telegraf configuration file, and the NiFi pipeline were also uploaded to the GitHub repository that was created for this architecture. In Table 5 we present the parameters used for each service (image, and port) that was deployed on the edge devices.

Table 5. Parameter table of deployed services.

Service	Image	Port	Notes
PostgreSQL	postgres:17.5-alpine	5432	Persistent volume
MQTT broker	eclipse-mosquitto:2.0	8883	TLS
core-keeper	EdgeX arm64	59890	Registry & config store
core-metadata	EdgeX arm64	59881	
core-data	EdgeX arm64	59880	
core-command	EdgeX arm64	59882	
eKuiper	lfedge/ekuiper:2.1	59720	Rules engine
app-rules-engine	EdgeX arm64	59701	Profile: rules-engine

Table 5. *Cont.*

Service	Image	Port	Notes
device-rest	EdgeX arm64	59986	
support-notifications	EdgeX arm64	59860	
support-scheduler	EdgeX arm64	59863	
InfluxDB	influxdb:1.8	8086	Sensor data
Telegraf	telegraf:1.30		Forwards to Cloud
NiFi	apache/nifi	8443	Forwards to Cloud
Portainer	portainer/portainer	9000	Services inspection

The deployed services can be remotely inspected, though this is not mandatory, using the widely adopted *Portainer* (v2.39.3) tool, which provides a graphical user interface for monitoring the status and health of Docker containers, networks, and volumes.

3.3. Functional Validation and Observations

During testing, the platform exhibited stable behaviour under the evaluated workload. Sensor data were successfully transmitted from the physical layer to the edge layer, stored locally, synchronised with the cloud data lake, and visualised at the application layer, with no observable data loss. Table 6 summarises the quantitative results obtained during a 24 h continuous test on both edge devices.

Table 6. Quantitative Validation. Cloud host: VirtualBox VM, 1 vCPU, 4 GB RAM, Ubuntu 18.04 LTS.

Metric	Raspberry Pi	NVIDIA Jetson Nano	Cloud
Test Duration	24 h	24 h	24 h
Sampling Interval	15 s	15 s	Near Real-Time
Total Messages Sent	5760	5760	11,520
Total Messages Stored Locally	5760	5760	11,520
Messages Synchronised to Cloud	5760	5760	11,520
Data Loss Rate	0%	0%	0%
Mean End-to-End Latency	<1 s	<1 s	<1 s
CPU Usage	~1–5%	~1–5%	~70–80%
RAM Usage	~3%	~3%	~10%
Database Size	96 Kbytes	176 Kbytes	272 Kbytes

A sampling interval of 15 s was used on both sensor nodes, thus each device generated and published a total of 5760 MQTT messages in one day, all of which were successfully stored in the local database and subsequently synchronised to the cloud data lake, resulting in a combined total of 11,520 messages received at the cloud layer. A 0% data loss rate was recorded across both devices, confirming reliable end-to-end data delivery from the physical layer through to cloud storage. This figure was obtained by comparing the number of messages published by each sensor node, as recorded via serial console logging, with the corresponding counts in the local and cloud databases at the end of the test period; the counts matched exactly in both cases. Both devices maintained near real-time synchronisation with the cloud, with negligible delay between message publication and its availability on the dashboard.

Both sensor nodes publish their JSON-formatted payloads to the MQTT topic/sensor, which is consumed locally by the corresponding data-synchronisation service; Telegraf on

the Jetson Nano, and Apache NiFi on the Raspberry Pi, before being forwarded to the cloud-hosted InfluxDB instance. The end-to-end data flow followed the expected path: DHT11 sensor/embedded digital sensor → MQTT publisher → EdgeX broker → local database → data synchronisation service → cloud data lake → Grafana dashboard. This confirms the correct integration of the individual components across all layers of the architecture.

Security mechanisms were also validated during deployment. Certificate-based authentication was enforced at the MQTT broker, ensuring that only authorised sensor nodes could publish data.

Specifically, the broker was configured to require mutual TLS (mTLS), where both the broker and each sensor node authenticate one another using X.509 certificates issued by a certificate authority (CA) created for this deployment. On the Arduino Nano ESP32 node, the CA certificate, client certificate, and private key are embedded in the firmware as raw string literals at compile time. On the STM32 Discovery node, the corresponding certificate and key files are provisioned interactively at first boot, by pasting them into the device over a serial terminal (e.g., Tera Term), after which they are retained in the device's non-volatile storage. The broker and the remaining edge- and cloud-side services obtain their certificates and private keys via Docker volume mounts configured at deployment time.

Beyond the transport-level security provided by mutual TLS, the platform incorporates certificate management and credential provisioning procedures tailored to the heterogeneous nature of the deployed hardware. Certificate generation follows a centralised authority model: a self-signed CA is generated once per deployment using OpenSSL, and all broker and device certificates are signed by this CA, as documented in the certificate generation script provided in the public GitHub repository which was mentioned earlier.

Broker-side certificates and keys are provisioned at deployment time via Docker volume mounts, ensuring that private key material never appears in the container image or the version-controlled configuration files. Device-side certificate provisioning differs between the two sensor node implementations: on the Arduino Nano ESP32, the CA certificate, client certificate, and private key are embedded directly in the firmware as raw string literals at compile time; on the STM32 Discovery board, the corresponding material is provisioned interactively at first boot via a serial terminal connection, after which it is retained in the device's non-volatile storage. Access control at the storage and visualisation layers is enforced through InfluxDB's built-in HTTP authentication, which requires valid credentials for all read and write operations, and through Grafana's user management system, which restricts dashboard access to authenticated users.

It is acknowledged that the current access-control model represents a baseline implementation suitable for a research prototype: a production deployment would require additional hardening, including role-based access control (RBAC), credential rotation policies, and (as discussed below) end-to-end payload-level protection independent of the broker.

It should be acknowledged that the mutual TLS implementation adopted in this work provides hop-by-hop security: the TLS session terminates at the MQTT broker, which therefore decrypts and has full access to the payload content before forwarding it to the subscribing services. This architecture implicitly assumes a trusted broker; an assumption that may not hold in all smart city deployment scenarios. A malicious or honest-but-curious broker could, in principle, inspect or modify the exchanged payloads without the knowledge of the publishing sensor nodes or the subscribing data consumers. End-to-end confidentiality and integrity, enforced at the application payload level independently of the intermediate broker, are not implemented in the current prototype and represent a recognised limitation.

Two relevant approaches addressing this problem in MQTT-based systems are worth noting. PICADOR [21] proposes a Proxy Re-Encryption (PRE) scheme for topic-based publish–subscribe systems, whereby the broker re-encrypts payloads from publishers into a form decryptable only by authorised subscribers, without itself being able to access the plaintext. More recently, MQTT-E [22] extends this concept to smart city vehicular scenarios, employing serverless edge devices as PRE proxy brokers to achieve end-to-end encryption while also addressing broker overload and Denial-of-Service resilience. Integrating such payload-level encryption mechanisms into the proposed platform (potentially at the Edge layer, prior to publication) is identified as a direction for future work, particularly as the platform is extended to use cases involving sensitive data.

Additionally, access to the cloud data lake was restricted to authenticated clients over HTTPS, confirming the correct application of access control policies throughout the data pipeline.

The elevated CPU utilisation recorded at the cloud layer (~70–80%) may appear disproportionate relative to the modest data load generated by two sensor nodes at a 15 s sampling interval. This figure reflects the aggregate resource consumption of all containerised services running concurrently on the cloud host (a VirtualBox virtual machine configured with a single vCPU and 4 GB of RAM running Ubuntu).

On a single-core host, Docker’s container scheduler must time-share the CPU across all active containers simultaneously, including PostgreSQL, InfluxDB, and Grafana, as well as the host OS itself. This scheduling contention produces a high aggregate CPU utilisation figure that is fundamentally a function of the constrained virtual hardware configuration rather than of the data processing demands imposed by the platform. In particular, Grafana’s default periodic polling of the InfluxDB data source contributes a recurring baseline CPU load that is independent of incoming data volume.

It should therefore be emphasised that the reported utilisation is not indicative of a scalability bottleneck intrinsic to the platform architecture: deploying the cloud layer on a multi-core host (as would be standard in any production smart city scenario) would distribute this load across multiple cores and yield substantially lower per-core utilisation.

To further investigate the source of the CPU load, per-container resource consumption was measured using Docker’s native statistics API (docker stats) while the platform was running under idle conditions; stack fully operational, no active dashboard user, and no data being ingested. The results, presented in Table 7, reveal that Grafana accounts for the dominant share of CPU utilisation (~30.84%), while InfluxDB contributes a negligible ~0.06%. The remaining container (Portainer) accounts for 0.01%. Critically, this measurement was taken during an idle period with no active data ingestion, confirming that Grafana’s CPU consumption is driven entirely by its default periodic polling of the InfluxDB data source rather than by the volume of incoming sensor data. These findings prove that the aggregate CPU utilisation reported in Table 6 is attributable to Grafana’s background behaviour on a constrained single-vCPU host, and is not indicative of a scalability bottleneck intrinsic to the platform’s data ingestion pipeline.

Table 7. Per-container resource consumption on the cloud host (VirtualBox VM, 1 vCPU, 4 GB RAM, Ubuntu 18.04 LTS) measured under idle conditions using ‘docker stats’.

Container	CPU %	Memory Usage	Memory %
Portainer	0.01%	30.02 MB	0.76
Grafana	30.84%	126.9 MB	3.23
InfluxDB	0.06%	82.9 MB	2.11

The difference in the sizes of the two local databases of the edge devices shown in Table 6 is attributed to the different payloads used in each sensor node. For example, for the case of the Arduino Nano ESP32 we used the following payload:

```
{
  "measurement": "sensor_data",
  "tags": {
    "ID": "esp32"
  },
  "time": "%lu", # long unsigned int
  "fields": {
    "Temperature": "%.1f", # float, 1 precision
    "Humidity": "%.1f" # float, 1 precision
  }
}
```

In contrast, for the case of the STM32 Discovery board, which offers more built-in sensors on its board by default (as shown in Table 3), we decided to use them all so the following payload was created:

```
{
  "measurement": "sensor_data",
  "tags": {
    "ID": "%s" # string
  },
  "time": "%lu", # long unsigned int
  "fields": {
    "temperature": "%.2f", # float 2 precision
    "humidity": "%.2f", # float 2 precision
    "pressure": "%.2f", # float 2 precision
    "proximity": "%ld", # long int
    "acc_x": "%d", # int
    "acc_y": "%d", # int
    "acc_z": "%d", # int
    "gyr_x": "%.0f", # float zero precision
    "gyr_y": "%.0f", # float zero precision
    "gyr_z": "%.0f", # float zero precision
    "mag_x": "%d", # int
    "mag_y": "%d", # int
    "mag_z": "%d" # int
  }
}
```

In both payload formats, the time field is a Unix timestamp generated locally on the sensor node and synchronised via the Network Time Protocol (NTP) before each transmission, ensuring that timestamps reflect the actual measurement time rather than the time of arrival at the edge or cloud layer.

Figure 6 presents a demo visualisation of aggregated data collected from the Arduino Nano ESP32 board and the STM32 Discovery board over a given time period, as displayed in Grafana. The dashboard can be accessed via any standard web browser on a desktop computer, tablet, or smartphone. As a result, a simple yet complete scenario for visualising basic environmental data within a defined area of interest is realised.

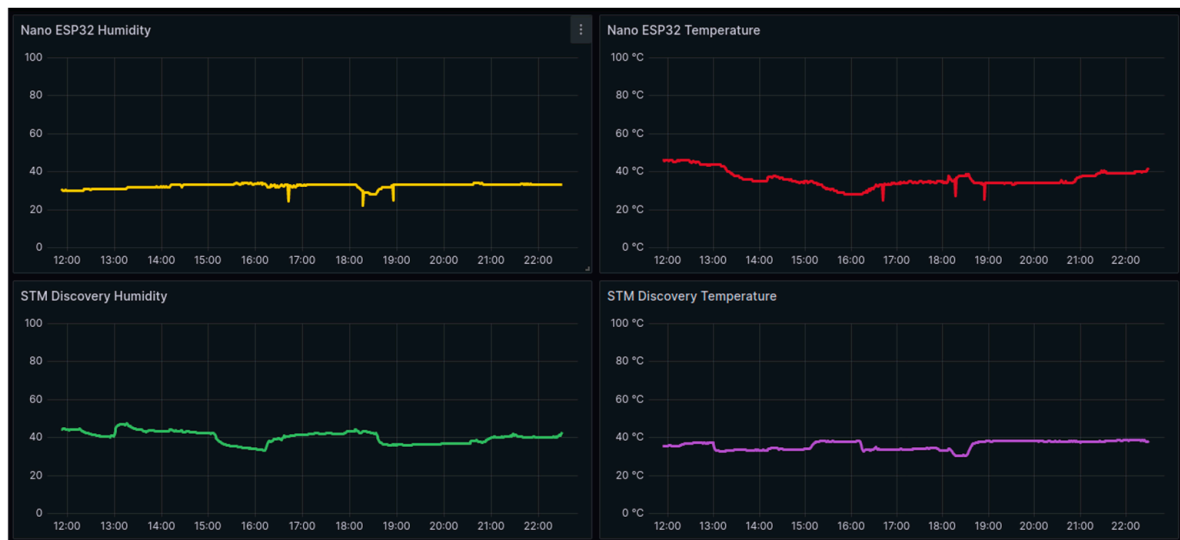


Figure 6. Visualisation of aggregated data in Grafana.

The Grafana configuration has been provided as JSON in the GitHub repository, for any reader who needs to replicate the implementation.

To evaluate the platform's behaviour beyond the original 24 h baseline test, two additional experimental scenarios were conducted using a virtual (Python-based) sensor node simulator. The simulator replicates the behaviour of both the Arduino Nano ESP32 and the STM32 Discovery board, generating pseudorandom temperature and humidity readings, signing communications with valid X.509 client certificates, and publishing to the MQTT broker at the same 15 s interval, and was configured to emulate 20 extra sensor nodes (10 Arduino Nano ESP32, 10 STM32 Discovery) publishing data over a five-hours period.

Under stable network conditions, no message loss was observed in the new 22-sensor-node (S2) scenario, demonstrating that the platform sustains reliable data delivery as the number of connected nodes increases by an order of magnitude relative to the original deployment. To evaluate behaviour under degraded network conditions, a third scenario (S3) was conducted in which artificial packet loss of 5% was introduced at the network interface of the edge device using the Linux Traffic Control utility (tc netem). Under these conditions, a message loss rate of 5% was recorded over a five-hour test period, consistent with the expected behaviour of MQTT QoS level 0 under unreliable connectivity.

These results, presented in Table 8, empirically validate the QoS 0 limitation discussed below in Section 3.4: while the platform performs reliably under stable conditions, the absence of delivery acknowledgement and retransmission makes it susceptible to message loss when network conditions degrade, reinforcing the case for adopting QoS level 1 in future deployments.

Table 8. Messages loss rate in stable and degraded networks, with QoS set to zero. Both real sensor-noded and simulated were used.

Scenario	Nodes	Duration	Network Condition	Messages Sent	Loss Rate
S1	2 (real hardware)	24 h	stable	11,520	0%
S2	22 (2 real hardware + 20 virtual)	5 h	stable	1200	0%
S3	22 (2 real hardware + 20 virtual)	5 h	degraded	1200	5%

3.4. Limitations

The evaluation presented in this section focuses primarily on functional validation rather than performance benchmarking or large-scale stress testing. Consequently, aspects such as system scalability, throughput under high data rates, and long-term resource utilisation were not assessed. These aspects are considered important directions for future experimentation and are discussed further in the next section.

Furthermore, the current implementation relies on MQTT Quality of Service (QoS) level 0 for both publication and consumption, providing at-most-once delivery without acknowledgement, retransmission, or duplicate-detection mechanisms. While no message loss was observed during the reported 24 h test under stable network conditions, this configuration does not guarantee delivery under less reliable connectivity, and the platform does not currently implement explicit handling of failed or duplicate messages. Beyond reliability, this choice carries specific security consequences in a smart city context. The absence of delivery acknowledgement means that packet loss induced by a network-level attacker (for example, through selective jamming or traffic interception) is indistinguishable from benign network degradation, providing no means of detecting deliberate data suppression at the transport layer. In a smart city scenario where sensor readings inform safety-critical decisions, such as environmental threshold alerts or traffic monitoring, silent message loss could be exploited to mask anomalous conditions from platform operators. Future work will explore the use of QoS level 1 (at-least-once delivery) at the sensor nodes and synchronisation services, combined with sequence-number- or timestamp-based deduplication at the ingestion stage, to provide stronger end-to-end delivery guarantees and to mitigate the risk of undetected message suppression.

4. Discussion

This work presented the design, implementation, and validation of a fully open-source IoT platform tailored to basic smart city (SC) use cases, using environmental monitoring as a representative example. By combining a layered reference architecture with a concrete deployment exclusively based on open-source tools, the proposed platform demonstrates that functional, secure, and scalable IoT solutions can be developed without relying on proprietary ecosystems.

One of the key contributions of this study lies in the synthesis of existing state-of-the-art reference architectures into a minimal yet extensible five-layer model. While many existing SC frameworks introduce numerous layers and components to address complex scenarios, the proposed architecture focuses on identifying the essential building blocks for entry-level smart city applications. This approach lowers the barrier to adoption for researchers, students, and developers, while preserving the ability to scale horizontally (by adding devices and edge nodes) and vertically (by enriching the architecture with additional layers, such as AI or Digital Twin components).

The implementation and deployment results validate the feasibility of the proposed architecture. The use of EdgeX Foundry as the edge stack proved effective in handling data ingestion, local storage, and secure data synchronisation with the cloud. Its hardware and OS-agnostic nature makes it particularly suitable for heterogeneous smart city environments, where devices with varying computational capabilities are expected to coexist. Similarly, the combination of MQTT, InfluxDB, and Grafana provided a lightweight yet robust data pipeline from sensor nodes to user-facing dashboards.

Security was treated as a transversal concern rather than an isolated component, reflecting best practices observed in contemporary SC architectures. The integration of X.509 certificates, authentication, and authorisation mechanisms across all layers ensures that even simple deployments adhere to fundamental security principles. While the imple-

mented security measures are sufficient for the presented use case, they also serve as a foundation for incorporating more advanced mechanisms, such as role-based access control, zero-trust networking, or hardware-based security modules.

Despite its successful deployment, the proposed platform exhibits several limitations that warrant further investigation. First, the evaluation focused on a minimal use case with a limited number of sensor nodes and relatively low data rates. As the number of devices increases, performance aspects such as broker throughput, edge storage limits, and cloud ingestion latency should be systematically assessed. Second, while InfluxDB was adequate for the scenario presented, more complex use cases involving heterogeneous data types (e.g., multimedia streams, metadata-rich payloads) would require alternative data lake technologies.

Future work will therefore focus on extending the platform in multiple directions. At the architectural level, the integration of AI components at the edge and/or cloud layers will be explored to support advanced analytics, anomaly detection, and predictive decision-making. At the communication level, additional protocols and transport technologies (e.g., CoAP, OPC UA, LPWANs, or 5G-based connectivity) will be evaluated to assess their suitability for different SC scenarios. Finally, more complex smart city use cases—such as traffic monitoring, air quality analysis, or energy management—will be implemented to further validate the generality and robustness of the proposed platform.

Finally, while the current deployment handles non-sensitive environmental data, specifically temperature and humidity readings, the platform's modular and extensible design means it could, in future iterations, be adapted to process more sensitive data types, such as video feeds, location tracking data, or biometric information collected in urban environments. In such scenarios, the existing security mechanisms (mutual TLS, certificate-based authentication, and access-controlled storage) would represent a necessary but insufficient baseline, and additional privacy-by-design measures would need to be incorporated. These would include, but are not limited to, data minimisation at the Edge layer (collecting only the data strictly necessary for the intended purpose), anonymisation or pseudonymisation of personally identifiable information prior to cloud transmission, purpose limitation enforced at the Application layer, and compliance with applicable data protection regulations, including the General Data Protection Regulation (GDPR) in the European Union context. The authors acknowledge these considerations as an important dimension of responsible smart city platform design, and identify the integration of privacy-by-design principles as a direction for future work as the platform evolves towards more complex and sensitive use cases.

Overall, this work demonstrates that open-source technologies can serve as the basis for practical, secure, and extensible SC IoT platforms, bridging the gap between conceptual reference architectures and deployable real-world systems.

Author Contributions: Conceptualization, N.M. and P.P.; methodology, N.M.; software, N.M.; validation, N.M., P.P., D.P. and V.C.; formal analysis, N.M. and P.P.; investigation, N.M.; resources, N.M., V.C. and G.S.; data curation, N.M.; writing—original draft preparation, N.M.; writing—review and editing, N.M., P.P. and D.P.; visualization, N.M., G.S. and V.C.; supervision, P.P. and D.P.; project administration, P.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data and configuration files supporting the findings of this study are available at https://github.com/nikmonios/Open_Source_Smart_City_Platform (accessed 21 July 2026).

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

SC	Smart City
IoT	Internet of Things
ETL	Extract–Transform–Load
AI	Artificial Intelligence
V2X	Vehicle-to-Everything
DLT	Distributed Ledger Technologies
OS	Operating System

References

- Monios, N.; Peladarinos, N.; Cheimaras, V.; Papageorgas, P.; Piromalis, D.D. A Thorough Review and Comparison of Commercial and Open-Source IoT Platforms for Smart City Applications. *Electronics* **2024**, *13*, 1465. [\[CrossRef\]](#)
- Panagiotakopoulos, T.; Vlachos, D.P.; Bakalakos, T.V.; Kanavos, A.; Kameas, A. A FIWARE-based IoT Framework for Smart Water Distribution Management. In Proceedings of the 2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA), Chania Crete, Greece, 12–14 July 2021; pp. 1–6.
- Ajao, L.A.; Apeh, S.T. Secure edge computing vulnerabilities in smart cities sustainability using petri net and genetic algorithm-based reinforcement learning. *Intell. Syst. Appl.* **2023**, *18*, 200216. [\[CrossRef\]](#)
- Storck, C.R.; Duarte-Figueiredo, F. A 5G V2X Ecosystem Providing Internet of Vehicles. *Sensors* **2019**, *19*, 550. [\[CrossRef\]](#) [\[PubMed\]](#)
- Whaiduzzaman, M.; Barros, A.; Chanda, M.; Barman, S.; Sultana, T.; Rahman, M.S.; Roy, S.; Fidge, C. A Review of Emerging Technologies for IoT-Based Smart Cities. *Sensors* **2022**, *22*, 9271. [\[CrossRef\]](#) [\[PubMed\]](#)
- Udayakumar, R.; Mahesh, B.; Sathiyakala, R.; Thandapani, K.; Choubey, A.; Khurramov, A.; Alzubaidi, L.H.; Sravanthi, J. An Integrated Deep Learning and Edge Computing Framework for Intelligent Energy Management in IoT-Based Smart Cities. In Proceedings of the 2023 International Conference for Technological Engineering and its Applications in Sustainable Development (ICTEASD), Al-Najaf, Iraq, 14–15 November 2023; pp. 32–38.
- Mahmudov, S.; Temirova, D. Smart Cities: Concepts, Standarts, Frameworks, Models, Technologies. In Proceedings of the 2022 International Conference on Information Science and Communications Technologies (ICISCT), Tashkent, Uzbekistan, 28–30 September 2022; pp. 1–7.
- Faliagka, E.; Christopoulou, E.; Ringas, D.; Politi, T.; Kostis, N.; Leonardos, D.; Tranoris, C.; Antonopoulos, C.P.; Denazis, S.; Voros, N. Trends in Digital Twin Framework Architectures for Smart Cities: A Case Study in Smart Mobility. *Sensors* **2024**, *24*, 1665. [\[CrossRef\]](#) [\[PubMed\]](#)
- Lynn, T.; Wood, C. Smart Streets as a Cyber-Physical Social Platform: A Conceptual Framework. *Sensors* **2023**, *23*, 1399. [\[CrossRef\]](#) [\[PubMed\]](#)
- Peralta Abadía, J.J.; Walther, C.; Osman, A.; Smarsly, K. A systematic survey of Internet of Things frameworks for smart city applications. *Sustain. Cities Soc.* **2022**, *83*, 103949. [\[CrossRef\]](#)
- Garcia-Font, V. Conceptual Technological Framework for Smart Cities to Move towards Decentralized and User-Centric Architectures Using DLT. *Smart Cities* **2021**, *4*, 728–745. [\[CrossRef\]](#)
- Corallo, A.; Del Vecchio, V.D.; Lezzi, M.; Morciano, P. Shop Floor Digital Twin in Smart Manufacturing: A Systematic Literature Review. *Sustainability* **2021**, *13*, 12987. [\[CrossRef\]](#)
- Jnr, B.A.; Petersen, S.A. Validation of a Developed Enterprise Architecture Framework for Digitalisation of Smart Cities: A Mixed-Mode Approach. *J. Knowl. Econ.* **2022**, *14*, 1702–1733. [\[CrossRef\]](#)
- Cumbajin, E.; Rodrigues, N.; Frazão, L.; Fernández-Caballero, A.; Pereira, A. Digital twin architecture for Ceramics 5.0. *Future Gener. Comput. Syst.* **2026**, *183*, 108529. [\[CrossRef\]](#)
- Kolhar, M.; Al-Turjman, F.; Alameen, A.; Abualhaj, M.M. A Three Layered Decentralized IoT Biometric Architecture for City Lockdown During COVID-19 Outbreak. *IEEE Access* **2020**, *8*, 163608–163617. [\[CrossRef\]](#) [\[PubMed\]](#)
- Jana, D.; Pal, P.; Kumar, P. Real-Time Agile Software Management for Edge and Fog Computing Based Smart City Infrastructure. In *International Conference on Computing and Communication Networks*; Springer Nature: Cham, Switzerland, 2025; pp. 113–121.
- Khalil, U.; Mueen, U.; Malik, O.A.; Hussain, S. A Blockchain Footprint for Authentication of IoT-Enabled Smart Devices in Smart Cities: State-of-the-Art Advancements, Challenges and Future Research Directions. *IEEE Access* **2022**, *10*, 76805–76823. [\[CrossRef\]](#)
- Khan, L.U.; Yaqoob, I.; Tran, N.H.; Kazmi, S.A.; Dang, T.N.; Hong, C.S. Edge-computing-enabled smart cities: A comprehensive survey. *IEEE Internet Things J.* **2020**, *7*, 10200–10232. [\[CrossRef\]](#)
- Sanchez-Gallegos, D.D.; Carrizales-Espinoza, D.E.; Torres-Charles, C.; Carretero, J. Smart Cities: A Systematic Review of Emerging Technologies. *Smart Cities* **2025**, *8*, 173. [\[CrossRef\]](#)

20. Platform Requirements. 2025. Available online: <https://docs.edgexfoundry.org/4.1/general/PlatformRequirements/> (accessed on 12 February 2026).
21. Borcea, C.; Gupta, A.B.D.; Polyakov, Y.; Rohloff, K.; Ryan, G. PICADOR: End-to-end encrypted Publish–Subscribe information distribution with proxy re-encryption. *Future Gener. Comput. Syst.* **2017**, *71*, 177–191. [[CrossRef](#)]
22. Buccafurri, F.; De Angelis, V.; Lazzaro, S.; Vangala, A. MQTT-E: E2E encryption in MQTT via proxy re-encryption avoiding broker overloading. *Ad Hoc Netw.* **2025**, *176*, 103878. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.