Blockchain-based Framework for Secure Data Streams Dissemination in Federated IoT Environments

Jakub Sychowiec * and Zbigniew Zieliński

Posted Date: 2 April 2025

doi: 10.20944/preprints202504.0237.v1

Keywords: Internet of Things; Blockchain; Distributed Ledger; Device and Data Authentication

 doi:10.20944/preprints202504.0237.v1

Article

Blockchain-Based Framework for Secure Data Streams Dissemination in Federated IoT Environments

Jakub Sychowiec * and Zbigniew Zieliński

Military University of Technology, Warsaw, Poland
* Correspondence: jakub.sychowiec@wat.edu.pl

Abstract: An industrial-scale increase of applications of the Internet of Things, a significant number of which are based on the concept of federation, presents unique security challenges due to their distributed nature and the need for secure communication between components from different administrative domains. The heterogeneity of devices, protocols, and security requirements in different domains further complicates the requirements for the secure distribution of data streams in Federated IoT Environments. The effective dissemination of data streams in federated environments also ensures the flexibility to filter and search for patterns in real-time to detect critical events or various types of threats (e.g., fires, hostile objects) with changing information needs of end users. Some known solutions and best practices for the secure distribution of data streams in such environments include end-to-end encryption, authentication of both IoT devices and data, and the use of blockchain (Distributed Ledger Technology, DLT). This paper presents a novel and practical framework for the secure, reliable, and dynamic dissemination of data streams within a multi-organizational federation environment. The framework integrates different technologies, such as Hyperledger Fabric (to implement a resilient authentication and authorization mechanism), Apache Kafka as data queuing technology, and microservice processing logic for verifying and disseminating data streams. However, integrating DLT, Kafka brokers, and streams microservices within the federation raises important questions regarding performance, security, and reliability. In the paper, we focus mainly on the scalability, throughput, latency, and potential bottlenecks. We thoroughly validated the effectiveness of the proposed framework by conducting extensive performance tests in two setups: the cloud-based and the resource-constrained environments.

Keywords: Internet of Things; blockchain; distributed ledger; device and data authentication

1. Introduction

We are observing an industrial-scale increase in the use of the Internet of Things (IoT) in both the civilian and military spheres, a significant number of which are multi-domain. Multi-domain IoT environments such as intelligent transportation, smart power grids, resilient smart cities, intelligent healthcare, or hybrid military operations are increasingly based on the concept of federation. The aim of creating a federation is to enable different entities to use shared resources and exchange information securely and efficiently without relying on a central authority, thus facilitating cooperation and increasing the resilience of the entire system. IoT federated environments are distributed, with their components and IoT devices located in different places and belonging to various entities. One example is the creation of a federation formed of NATO countries and non-NATO mission actors (Federated Mission Networking, FMN) [1], where each actor retains control over its capabilities and operations while accepting and meeting the requirements outlined in pre-negotiated and agreed-upon arrangements, such as the security policy. Another example is the interaction of civilian services and the military, which form a federation when providing humanitarian assistance in eliminating natural disasters (Humanitarian Assistance and Disaster Relief, HADR).

There are many situations in which federation partners need to exchange information, for example, about the location of each other's troops, detected threats (image object detection), etc. IoT

 doi:10.20944/preprints202504.0237.v1

in a federation environment enhances the ability to get an accurate real-time [2] picture of the situation during an operation, e.g., by deploying mobile IoT devices such as unmanned aerial vehicles (UAVs). Hence, IoT devices operated by different federation partners must securely communicate with each other. To ensure the timely transmission of situational awareness data, the UAV may need to use a partner's resources within the communication range. In this case and many others, it is necessary to establish a trust relationship through mutual authentication of devices belonging to different federation partners, such as between a specific UAV and the partner's data distribution system. In the case of information exchange in federated environments, where the military participates jointly with civilian services in HADR operations in an urbanized area, it is most often assumed that mobile radio networks will be utilized as the main communication medium.

Federated IoT environments present unique security challenges due to their distributed nature and the need for secure communication between components from different administrative domains. To an even greater extent, fulfilling secure data stream distribution requirements in federated IoT environments is a complex challenge due to the heterogeneity of devices, protocols, and security requirements across domains. One example of a use case is the secure distribution of data streams in smart cities. In a smart city scenario, data streams from traffic sensors, surveillance cameras (CCTV), and environmental sensors must be securely distributed across multiple domains (e.g., transportation, public safety, and utilities).

Some known solutions and best practices for ensuring secure data stream distribution in such environments include end-to-end encryption (preventing unauthorized access during transmission), authentication (for device and data), and blockchain mechanism (to ensure data integrity and accountability). These solutions also involve processing data locally at edge gateways to reduce latency and improve security.

Another requirement for the effective distribution of data streams in federated environments is the ability to process them (filtering, pattern searching) in real-time to detect critical events (e.g., traffic congestion) or various types of threats (e.g., fire, hostile objects). It is essential to ensure flexibility and dynamic data distribution, which can be reflected in the changing demand for information from end users. This means it allows systems to dynamically subscribe/unsubscribe to data streams based on their current needs (Contextual Dissemination). One possibility is to rank information and services using the Value of Information (VoI), a subjective measure that quantifies information's value to its users. Platforms such as VOICE [3] use VoI to rank Information Objects (IOs) and data producers based on their relevance and usefulness.

By combining aformentioned solutions, multi-domain IoT environments can achieve secure and efficient data stream distribution while addressing the unique challenges posed by IoT devices and heterogeneous networks. Moreover, the optimal performance and resource utilization can be achieved by enabling context-dependent data dissemination based on its importance and relevance.

The presented needs for ensuring the reliability and security of data exchange bring challenges, the solution of which determines the implementation of IoT. The basic problem remains: *how to carry out the acquisition and fusion of data from various sources with different levels of trust and operate in computing environments with varying degrees of reliability and security?* To solve it, it is necessary to know the answer to the sub-questions in the first place:

- *Security gap* - How to implement secure data distribution among participants in a federated environment, adhering to the Data-Centric Security paradigm [4]? This involves ensuring robust data protection starting from the point of origin, maintaining integrity throughout its entire life cycle, and facilitating granular access control mechanisms to enforce strict data permissions.
- *Identity Management gap* - How to manage the identity of devices? How to identify devices?
- *Real-Time Data Access gap* - How to enable the processing and sharing of relevant IOs based on their VoI, importance and relevance within a specified time range?

 doi:10.20944/preprints202504.0237.v1

- *Resource Allocation gap* - How to dynamically allocate resources to effectively manage trade-offs in a data dissemination environment while taking customer key performance indicators (KPIs) into account?
- *Network Integration and Interoperability gap* - How to organize interconnections, especially between unclassified systems (civilian systems) and military systems?
- *Resilience and Centralization gap* - How to ensure data availability in constrained (partially isolated) environments?

The paper addressed several highlighted gaps, including Security, Identity Management, Interoperability, and Resilience. To fulfill these issues effectively, it was essential to integrate various concepts and technologies while considering the security requirements of Federated IoT Environments. This integration involves implementing a data-centric authentication mechanism that employs a unique identity (fingerprint) that can be reliably stored within a DLT. Additionally, establishing a data stream processing system built on a lightweight and manageable pool of microservices, complemented by context-based real-time data dissemination technologies. Consequently, we proposed a multi-layered framework architecture aimed at ensuring the secure and reliable dissemination of data streams within a multi-organizational federation environment. Our framework implements data-centric authentication based on the unique identities of IoT devices. We consider our primary contributions to be as follows:

1. We have developed a novel and practical framework for the secure and dynamic dissemination of data streams within a multi-organizational federation environment utilizing Hyperledger Fabric [5], Apache Kafka as data queuing technology, along with a microservice processing logic for verifying and disseminating data streams (by utilizing the Kafka Streams API library in Java [6] and the Sarama library in Go [7]);
2. We have integrated a hardware-software IoT gateway with a DLT (Hyperledger Fabric) to authenticate IoT devices and verify data streams, which involves the deployment of the Fingerprint Enrichment Layer in conjunction with the Protocol Forwarder (proxy) component;
3. We validated the effectiveness of the proposed framework by conducting extensive performance tests in two Setups: the Amazon Web Services cloud-based and the Raspberry Pi resource-constrained environment.

The remainder of the article is structured as follows: Section II provides an overview of the relevant research that serves as the foundation for our solution. Section III details our multi-layered framework architecture, highlighting its key components and the security and reliability mechanisms employed to enhance confidentiality, integrity, availability, and accountability for data: in-process, in-transit, and at-rest. Section IV outlines the proposed message types and key operations of our experimental framework. Section V introduces two configurations of our environment: one cloud-based and the other resource-constrained, along with benchmarks for latency metrics. Section VI presents a thorough discussion of the results we obtained. Section VII summarizes our conclusions and delineates our goals for future work. Lastly, the abbreviations used throughout this publication are defined at the end.

2. Related Work

This section presents related works that have had the greatest impact on the proposed framework architecture for secure and reliable data stream dissemination in the Federated IoT Environments. These works address the basic problems related to:

- securing data processed by IoT devices with the usage of distributed ledger technology and blockchain mechanism;
- behavior-based IoT device identification (IoT distinctive features);
- the integration of heterogeneous military and civilian systems based on IoT devices, where specific KPIs must be achieved (e.g., zero-day interoperability).

 doi:10.20944/preprints202504.0237.v1

Additionally, at the end of this section, we have briefly discussed our solution against the analyzed works.

2.1. Blockchain-Based Device and Data Protection Mechanisms

The literature presents numerous attempts to integrate the IoT and blockchain (distributed ledger) technology. The work [8] describes the challenges and benefits of integrating blockchain with the IoT and its impact on the security of processed data. Similarly, in the works [9,10], where a proposal for a 4-tier structural model of Blockchain and the IoT is presented.

Guo et al. [11] proposed a mechanism for authenticating IoT devices in different domains, where cooperating DLTs operating in the master-slave mode were used for data exchange. Xu et al. [12] presented the DIoTA framework based on a private Hyperledger Fabric blockchain, which was used to protect the authenticity of data processed by IoT devices.

The work [13] proposed an access control mechanism for devices, which used the Ethereum public blockchain placed in the Fog layer and public key infrastructure based on elliptic curves. Furthermore, NIST has defined Attribute-Based Access Control (ABAC) [14] as a logical access control method that authorizes actions based on the attributes of both the publisher and the subscriber, the requested operations, and the specific context (current situational awareness).

H. Song et al. [15] proposed a blockchain-based ABAC system that employs smart contracts to manage dynamic policies, which are crucial in environments with frequently changing attributes, such as the IoT. Additionally, Lu Ye et al. [16] introduced an access control scheme that integrates blockchain with ciphertext-policy attribute-based encryption, featuring fine-grained attribute revocation specifically designed for cloud health systems.

2.2. Fingerprint Sampling Techniques

Apart from classification methods for identifying a group or type of similar IoT devices [17], an interesting area of research is fingerprint techniques [18,19], which aim to identify a unique image of a device's identity through appropriate selection of its distinctive features. The fundamental premise of fingerprint methods is the occurrence of manufacturing errors and configuration distinctions, which implies the non-existence of two identical devices. Subsequently, the main challenge associated with fingerprinting techniques is the selection of non-ephemeral parameters that make it possible to distinguish devices uniquely. Generally, three main fingerprint method classes can be identified for IoT devices as a result of distinction:

- *Hard/Soft-ware class* - hardware and software features of the device;
- *Flow class* - characteristics of generated network traffic;
- *RF class* - characteristics of generated radio signals.

The authors of the LAAFFI framework [20] presented a protocol designed to authenticate devices in federated environments based on unique hardware and software parameters extracted from a given IoT device.

Concerning distinctive radio features, Sanogo et al. [21] evaluated the Power Spectral Density parameter. The work [22] indicates a proposal to use neural networks to identify devices based on the Physical Unclonable Function in combination with radio features: frequency offset, in-phase (I) and quadrature (Q) imbalance, and channel distortion.

Charyyev et al. [23] proposed the LSIF fingerprint technique, where the Nilsimsa hash function was used to determine a unique IoT device network flow. In contrast, the work of [24] demonstrated the Inter-Arrival Time (IAT) differences between successively received data packets as a unique identification parameter.

2.3. Reliable Data Streams Dissemination

In addressing one of the primary challenges of deploying various IoT devices within Federated Environments, which necessitates the processing of vast data streams in a secure, reliable, and context-

 doi:10.20944/preprints202504.0237.v1

dependent manner, we undertook a thorough analysis of relevant literature on this subject. Our emphasis was also on developing solutions to the sub-questions outlined in Section 1, particularly those concerning the gaps in Real-Time Data Access, Network Integration, Interoperability, and Security, all of which can be effectively tackled with a Data-Centric approach.

Notably, NATO countries continually refine their requirements (KPIs) to address mentioned challenges. Moreover, they establish research groups dedicated to identifying optimal solutions for coalition data dissemination systems within the framework of Federated Mission Networking.

Jansen et al. [25] developed an experimental environment involving four organizations, where data is distributed across two configurations. The first configuration employs two types of MQTT brokers: Mosquito and VerneMQ. In contrast, the second configuration operates without brokers, broadcasting MQTT streams using a connection-less protocol (e.g., User Datagram Protocol, UDP).

Suri et al. [26] performed an analysis and performance evaluation of eight data exchange systems utilized in mobile tactical networks, revealing that the DisService author's protocol significantly outperforms alternatives such as Redis and RabbitMQ. Additionally, another study [27] suggests a data exchange system for IoT devices based on the MQTT protocol, incorporating elliptic curve cryptography for data security.

Furthermore, Yang et al. [28] introduced a system architecture designed for anonymized data exchange among participants, leveraging the Federation-as-a-Service cloud service model and built upon the Hyperledger Fabric.

2.4. Discussion

Although the previously aforementioned publications provide solutions for data access management, data exchange systems, and device/data authentication methods, a notable gap exists for a data dissemination system tailored for dynamic and distributed environments, such as the Federated IoT. Furthermore, there is a lack of systems that incorporate device authentication techniques based on unique fingerprints while ensuring data protection in accordance with the Data-Centric paradigm.

In our literature analysis, we have not identified a solution that effectively combines components of a distributed ledger (specifically, Hyperledger Fabric), data queue systems (like Apache Kafka), and stream processing microservices to address the identified gap in data dissemination.

Most of the reviewed publications rely on a trusted third-party infrastructure and a private DLT to enhance the security of processed data. In contrast to the approaches taken by Guo et al. (master-slave chain) and Xu et al. (DIoTA framework), our solution utilizes a single global instance of the distributed ledger. This allows for the seamless transfer of devices between organizations within the federation, enabling these devices to use another organization's infrastructure for secure data dissemination.

The studies [25,26] do not assess data queue technologies like Apache Kafka. Moreover, they concentrate solely on efficient data exchange and overlook the security of data streams. In our proposed solution, it is feasible to implement Attribute-Based Access Control while concurrently adhering to the Data-Centric paradigm [4].

Furthermore, we have prioritized interoperability between military and civilian systems, particularly considering the limitations of such environments. To this end, our proposed system incorporates recommendations from the NATO IST-150 working group [25], which examined disconnected, intermittent, and limited (DIL) tactical networks. Our system employs a publish-subscribe model and utilizes Commercial Off-The-Shelf (COTS) components that are widely available, thereby minimizing operational costs, which is crucial for ensuring immediate interoperability.

Additionally, our work distinguishes the key used for securing the communication channel of IoT devices from the key used for data authenticity protection. Unlike the DIoTA framework, which employs an HMAC-based commitment scheme with randomly generated keys for message authentication, we propose using the unique fingerprint samples of the device. Specifically, we propose to utilize a sealing key as a hybrid identity image based on a combination of several fingerprint method classes.

 doi:10.20944/preprints202504.0237.v1

Finally, our framework (specifically, streams microservices) can be enhanced with components that analyze, classify, and share data streams based on the specific VoI that IOs provide to consumers (context), as well as their relevance [3].

3. Framework Design

This section outlines our multilayered framework architecture for the reliable dissemination of data streams (messages) within a multi-organizational federation environment that requires the Data-Centric Security approach. Figure 1 depicts the system's overall structure of a federation comprising two organizations: Org1 and Org2. As part of our pluggable architecture, we have identified the following layers:

- *Publishers Layer* – encompasses entities that facilitate the data-centric protection of generated (produced) data streams through the sealing process, where the sealing key is derived from the device's fingerprint (identity) defined during the registration operation managed by the Device Maintain Layer;
- *Subscribers Layer* – is made up of authenticated and authorized entities that read (consume) available and verified data streams from the Data Queue Layer according to the access policy (e.g., one to many);
- *Data Queue layer* - is composed of distributed data queues that facilitate the acquisition, merging, storage, and replication of data streams transmitted from the Publisher's Layer, subsequently making it accessible to the Subscribers Layer;
- *Fingerprint Enrichment Layer* - can transport connection-less protocols like UDP by employing a Protocol Forwarder that converts data streams into a connection-oriented format (e.g., Transmission Control Protocol, TCP). This layer is essential due to the constraints of the connection types supported by the technologies available for the Data Queue Layer. Moreover, it facilitates device behavior-based authentication by utilizing analyzers to gather various features, including network and radio characteristics, and by enriching messages with entity fingerprint samples;
- *Distributed Ledger Layer* - enables interchangeable deployment of various DLT. Furthermore, it reliably stores the identities of devices belonging to organizations within the federation and retains information about entity Data Quality and subscribers Data Context Dissemination [29];
- *Communication Layer* - enables the Streams Microservice and the Device Maintain Layer to communicate with the Distributed Ledger via a hardware-software IoT gateway;
- *Streams Microservice Layer* - is tasked with verifying sealed data streams originating from entities within the Publisher's Layer. Additionally, it has the capability to analyze, categorize, and disseminate streams pertinent to these entities, and enrich them (e.g., by detecting objects during image processing);
- *Device Maintain Layer* – manages device registration operation, with additional responsibilities including updating and revoking identities. The registration process initiates with the establishment of the entity's identity through the use of fingerprint methods.

In this article, we proposed the adoption of specific technologies to address the various layers of our system. Figure 2 illustrates our architecture, which takes advantage of open-source platforms and libraries. For the Distributed Ledger Layer, we selected the Hyperledger Fabric solution. In the Data Queue Layer [5], we deployed the Apache Kafka stream-processing platform [6]. Finally, for the Streams Microservice Layer, we implemented two the same streams processing logic utilizing the Kafka Streams API library in Java [6] and the Sarama library in Go [7].

Our architecture is thoughtfully designed to enhance interoperability, facilitating the seamless transfer of devices between the Publishers and Subscribers Layers across diverse organizations (Figure 1). This approach enables these devices to utilize various organization Data Queue Layers, ensuring secure and reliable data dissemination in any configuration that aligns with predefined policies, such as one-to-many relationships.

 doi:10.20944/preprints202504.0237.v1

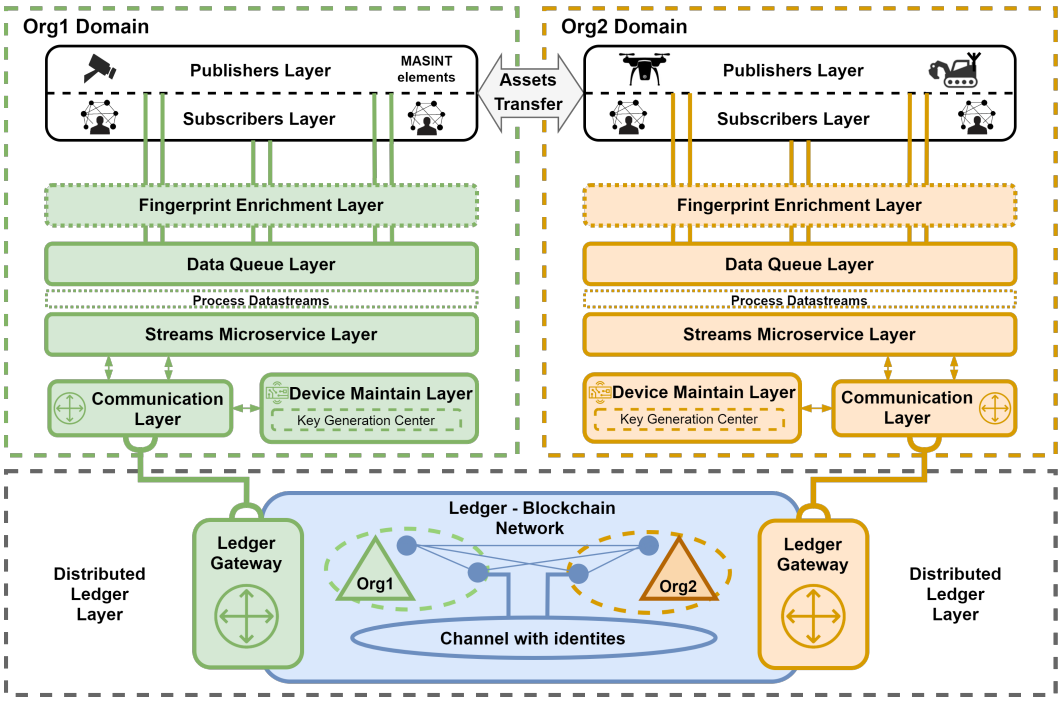

Figure 1. Proposed framework general overview.

Entities representing the Publisher's Layer ensure the security of their data streams by sealing them with identities registered within the Device Maintenance Layer. These sealed streams are transmitted using the available (supported) communication protocols and mediums to the Data Queue Layer (Kafka Cluster). In this layer, the streams are stored under a specific topic, such as *topic_1-in*. Additionally, the Fingerprint Enrichment Layer participates (proxies) in transmission utilizing transformation of connection-less messages into a connection-oriented message format.

The subsequent step occurs in the Streams Microservice Layer, which sequentially retrieves messages from the brokers to verify the sealed messages. The Verify Streams Microservice queries the Distributed Ledger Layer (Hyperledger Fabric) through the Communication Layer (IoT gateway) to obtain an image of the device's identity. Next, the identity extracted from the message is compared with the one stored in the ledger.

Once a message is verified and approved, it is written back to the Kafka Cluster under a dedicated topic, such as *topic_1-out*, making it accessible to entities representing the Subscribers Layer. Optionally, any rejected messages during this process can be directed to a separate topic to aid in identifying and detecting potentially malicious devices.

As previously mentioned, a device identity image is critical to the verification process. The Device Maintenance Layer encompasses a registration operation through the Key Generation Center, during which the device identity is established using hybrid fingerprint techniques. These techniques determine the specific hardware and software features, generated network characteristics, and radio signals related to the device. Once the identity is defined, it is redundantly added (stored) in the Distributed Ledger Layer through the execution of chaincode (group of smart contracts). Successful registration is contingent upon achieving consensus among the participating federated organizations.

Moreover, within our framework, we have delineated two categories of data stores: the on-chain store and the off-chain store. The on-chain store refers to the ledger, while the off-chain store encompasses local storage utilized by applications and microservices, such as those leveraging the Kafka Streams API library. One of our key proposals is to employ the off-chain store as a micro-caching mechanism to store identities from the on-chain store temporarily. This mechanism can significantly mitigate the delays associated with ledger queries and improve the overall performance of an individual instance of the Verify Streams Microservice.

 doi:10.20944/preprints202504.0237.v1

To realize this, we proposed a minor extension to our environment. In the Publisher's Layer, a dedicated listening application known as Blockchain Event Listeners can be utilized to manage events emitted from the Distributed Ledger Layer. These events will pertain to ledger operations, including device registration, updates, and revocations. Furthermore, a dedicated topic such as *dev_id*, specifically for storing these events, can be established. Subsequently, the Identity Streams Microservice can be deployed to add or update identities in the off-chain store. Additionally, by enforcing appropriate retention policies for this designated topic, we can preserve historical event-based records of device identity changes, functioning as a blockchain-topic.

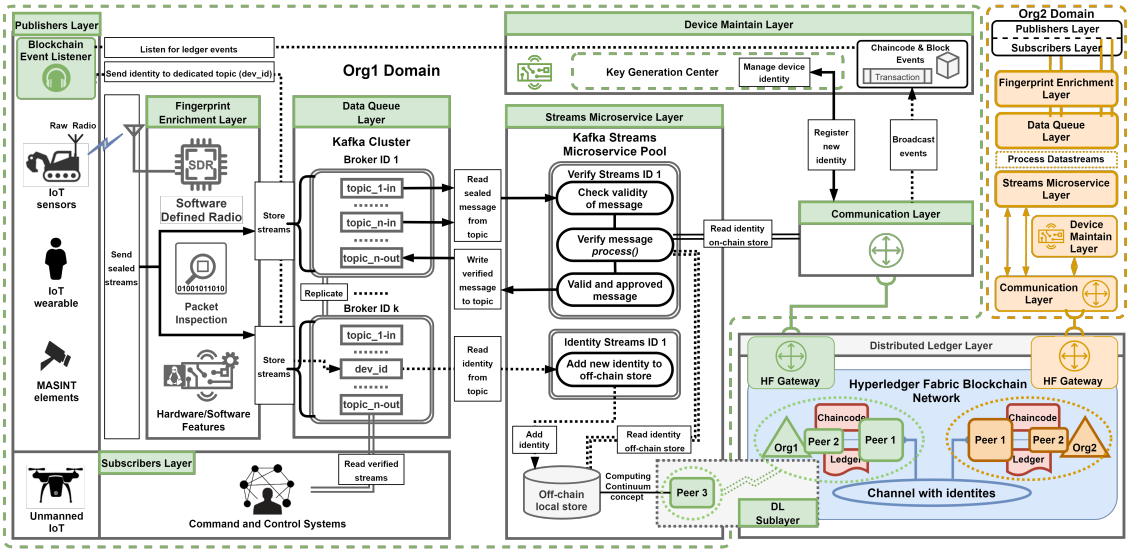

Figure 2. Proposed framework detailed overview.

Alternatively, the off-chain store can be integrated with the Continuum Computing concept (CC) [3] which involves providing computing capabilities across the diverse layers of an IoT system (edge, fog, and cloud). The fundamental notion of this concept is to relocate high-performance cloud-based services to lower layers. While this escalates resource management's complexity, it facilitates deploying services that demand ubiquitous and efficient computing capabilities. The concept also aims to optimize resource allocation, ensuring that devices perform service tasks as close as possible to data sources.

In the realm of computing modeling, our system is designed to process incoming requests sequentially to minimize the decay of Information Objects. Our architecture facilitates the dynamic deployment of microservices at all layers of the IoT system, offering both horizontal and vertical scaling for resource allocation. It also accommodates various deployment contexts.

The upcoming headings will provide an in-depth overview of the mentioned layers, along with mechanisms that enhance confidentiality, integrity, availability, and accountability for data: in-process, in-transit, and at-rest. Furthermore, a thorough analysis of the built-in security and reliability features associated with each specified technology will be provided.

3.1. Publishers Layer

The Publisher Layer encompasses entities that facilitate the data-centric protection of generated data streams through symmetric encryption and sealing operation. This leverages digital signatures and the device's hybrid identity. Although the selection of encryption and digital signature algorithms is not the primary focus of this study, it invites consideration of the potential applications of lightweight and quantum-resilient cryptography. In 2022, NIST evaluated and approved two quantum-resistant algorithms for digital signatures [30]: FALCON and SPHINCS+. These algorithms utilize distinct mathematical approaches: lattice and hash systems, respectively. In our research, however, we opted for well-established algorithms such as AES for encryption and HMAC for signatures. Devices

 doi:10.20944/preprints202504.0237.v1

with limited resources, like the Raspberry Pi, have adequate computational power to perform the cryptographic operations required by these algorithms.

3.2. Subscribers Layer

The Subscribers Layer comprises entities that have been granted authorized access to the Data Queue Layer. Our framework's architecture is designed to ensure the reliable dissemination of data across all levels of command: tactical, operational, and strategic. Additionally, the framework enables seamless data sharing across different domains within a federation, which includes allied forces and civilian institutions, fostering collaboration and coordinated efforts toward common objectives. Moreover, our system adeptly manages data security along the path from producer to consumer, while also implementing fine-grained access control mechanisms.

3.3. Data Queue Layer

The layer in the headline holds a pivotal position within our system, performing a multitude of important functions such as:

- storing and replicating sealed data streams within the layer;
- storing invalid records to trigger a detection mechanism of potentially malicious entities;
- intermediating within the micro-caching mechanism by linking the Publisher Layer (Blockchaine Event Listeners) with the Streams Microservice Layer.

The Publisher's layer comprises numerous data sources, each generating real-time data streams that require processing. To facilitate the seamless processing of these messages, the Apache Kafka solution has been used. For example, Kul et al. [31] have introduced a framework that leverages Kafka and neural networks to monitor (track) vehicles. In their study, the dataset was represented as data streams captured by CCTV.

Apache Kafka is a stream-message system that utilizes a producer-broker-consumer (publish-subscribe) model and classifies messages based on their topics. The Kafka Cluster is composed of message brokers that acquire, merge, and store data generated from the Publishers Layer (producers), and make it available to the Subscribers Layer (consumers). The layer is designed for the availability and reliability of data records, thanks to built-in synchronization and distributed data replication between brokers. Furthermore, it leverages serialization and compression mechanisms, such as lz4 and gzip, making messages payload format- and protocol-independent.

Furthermore, Kafka technology incorporates built-in components and mechanisms for defining and managing entity authorization through Access Control Lists (ACLs). Essentially, ACLs outline which entities can access specific resources (topics) and delineate allowed operations (e.g, READ, WRITE) to perform on those resources. Establishing a distinct principal for each entity (device) and assigning only the necessary ACLs, enables the processes of debugging and auditing leading to the identification of which entity executes each operation.

However, large Kafka Cluster topologies that involve multiple publishing and subscribing entities (numerous topics) often encounter significant challenges in managing entity authorization. Although it is feasible to implement a more intricate authorization hierarchy within the cluster. This can imply an additional operational burden. In the previous article [4], we explored the application of ABAC in our environment. This solution can alleviate the authorization burden on the Kafka Cluster, thereby freeing its internal resources and allowing the Streams Microservice Layer to manage access control more efficiently.

3.4. Fingerprint Enrichment Layer

A notable limitation of Kafka technology is its dependence on connection-oriented protocols, specifically the TCP. In contrast, our framework requires the ability to communicate with entities utilizing connection-less protocols, such as the UDP. To address this challenge, we propose the introduction of a Fingerprint Enrichment Layer. A fundamental component of this layer is a Protocol

 doi:10.20944/preprints202504.0237.v1

Forwarder, which is designed to handle connection-less messages and convert them into a connection-oriented message format.

Furthermore, the mentioned layer can contribute to device behavior-based (fingerprint-compliance) authentication. This can be accomplished through the utilization of dedicated components, referred to as analyzers, which gather radio signal, network flows, and hardware features. For the radio fingerprinting analyzer, we propose employing a Software Defined Radio. This system replaces or supplements traditional hardware components, such as mixers, filters, amplifiers, and detectors, with software-based Digital Signal Processing techniques. Providing flexibility, cost-effectiveness, versatile wideband reception, and enhanced interoperability within our architecture and radio communication systems. For the network analyzer, we can capture distinctive features from the headers of TCP/IP layers through comprehensive packet inspection. A detailed description of the proposed analyzers subsystem is beyond the scope of this publication and will be the subject of our future research.

3.5. Distributed Ledger Layer

Our experimental framework features a pluggable structure that allows for the interchangeable deployment of different DLTs. The Distributed Ledger Layer indicates several functions such as:

- redundant and reliable storing of all identities of devices belonging to organizations that participate in the federation;
- redundant and reliable storing information about entities regarding the Value of Information and the Context Dissemination;
- secure handling of the chaincode (smart contracts) execution (transaction steps) during the device registration operation;
- obtaining approvals (transaction authorizations) under endorsement policy from participating organizations;
- generating events related to actions on distributed ledger (blockchain);
- being an integrated part of the verification process of devices and sealed messages.

Based on a performance comparison, we have opted to integrate the Hyperledger Fabric technology with our system. This particular technology has achieved a transaction throughput of 10,000 tps, as documented in [8]. It is noteworthy that the Ethereum ledger had a lower throughput. However, for Ethereum, only the Proof of Work consensus protocol was examined. Whereas the currently less energy-intensive and scalable Proof of Stake consensus was not covered in these experiments.

The Fabric DLT adopts the Practical Byzantine Fault Tolerance consensus algorithm, which mandates that all participating parties know of one another. As a consequence, it is a permission blockchain where public key infrastructure is deployed. To register the identity of devices, complex business logic is executed using multilingual chaincode (Go, Java, Node.js). Moreover, the target execution environment for chaincode is a Docker container and its resources can be controlled (e.g., limited, isolated) through a Linux kernel feature *cgroups*. Also, private data channels can be created between organizations, where the identity of selected devices can be hidden from other organizations.

The on-chain store used in Fabric technology consists of systematically organized structures: world state and transaction log. The world state serves as the ledger's current state database, while the transaction log acts as a Change Data Capture mechanism. This mechanism incrementally records both approved and rejected transactions, ensuring that data at-rest is secure and accountable.

3.6. Communication Layer

The hardware-software IoT gateway manages communication between the Device Maintain Layer, the Streams Microservice Layer, and the Distributed Ledger Layer via an interface to the Hyperledger Fabric Gateway services [5] that run on the ledger nodes. The Communication Layer facilitates the seamless communication exchange for:

- performing queries to the on-chain store to read the examined identity from the Distributed Ledger Layer during the verification process;

 doi:10.20944/preprints202504.0237.v1

- participating in entity identities registering, updating, and revoking operations called by the Device Maintain Layer;
- broadcasting of events generated by the Distributed Ledger Layer as a result of approved transactions and blocks.

The IoT gateway utilizes a dynamic connection profile with the Distributed Ledger Layer. This profile leverages the internal capabilities of ledger nodes to detect alterations in the network topology in real-time, thus ensuring the seamless functioning of the Streams Microservices Layer, even in the event of node failure. Furthermore, the checkpointing mechanism proves to be beneficial, as it enables uninterrupted monitoring of ledger events without the risk of data loss due to connection interruptions.

3.7. Streams Microservice Layer

The Streams Microservice Layer is responsible for verifying sealed messages originating from entities within the Publisher's Layer. This layer is notable for executing complex operations on individual messages (records) in a sequential manner. To enhance the efficiency of message (stream) processing, selecting an appropriate framework or library is essential. Evaluations conducted by Karimov et al. [32] and Poel et al. [33] assessed various solutions designed for this purpose. Both studies concluded that the Apache Flink framework outperformed alternatives such as Kafka Streams API, Spark Streaming, and Structured Streaming, earning the highest overall ranking.

However, our proposed system architecture leverages the Streams API library to define custom verification processing logic. This choice is based on the inherent advantages of Kafka technology, including its failover and fault tolerance features. The Streams API library offers two approaches for implementing processing logic: the high-level Streams DSL and the low-level Processor API. We chose the Processor API for its pluggable architecture, which facilitates the deployment of various types of local off-chain stores. Moreover, the library employs a semantic guarantee pattern that ensures each message is processed exactly once from start to finish, thereby preventing any loss or duplication of messages in the event of a stream processor failure.

We opted against using the Spark and Structured Streaming frameworks as they rely on micro-batching, which processes messages within fixed time windows. Similarly, we did not consider the Apache Flink framework because it requires a separate processing cluster, which would increase operational costs for infrastructure deployment and negatively impact interoperability.

It is essential to highlight that our experimental system can further leverage the Kafka Streams API to support a range of tasks, including Data Enrichment, Data Quality Assessment, and Data Context Dissemination. In particular, this capability can be applied to object detection and classification during image processing. Figure 3 showcases the integration of our system for the mentioned use case.

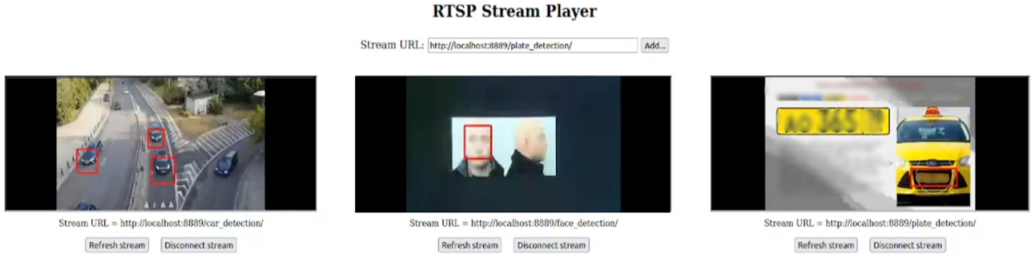

Figure 3. Message enrichment within our experimental system.

3.8. Device Maintain Layer

The primary function of the Device Maintain Layer is to manage device registration operations, with additional responsibilities including updating and revoking device identity. The registration process begins with establishing the entity's identity through hybrid fingerprint techniques. Entities of the Publisher's Layer will use these identities during the sealing process, where the device identity

 doi:10.20944/preprints202504.0237.v1

image working as a key will be used with digital signatures to seal data streams sent to the Data Queue Layer.

In the process of defining entity identity, we advocate for the use of a Confidential Computing strategy [34] that incorporates the Defense-in-Depth and Hardware Root of Trust concepts. This strategy involves the implementation of multiple heterogeneous security layers (countermeasures) that are built on highly reliable hardware, firmware, and software components. These countermeasures are essential for executing critical security functions, such as session key generation and the secure storage of cryptographic materials, ensuring that any adverse operations not detected by one technology can still be identified and mitigated by another.

To safeguard data across its various states – whether in-process, in-transit, or at-rest – secure enclaves, including Trusted Execution Environments (TEEs), Hardware Security Modules (HSMs), or Trusted Platform Modules (TPMs), can be employed. TEEs provide a secure area within the processor, whereas HSMs are specialized hardware created specifically for key storage. Meanwhile, TPMs are hardware chips that offer a range of security functions, including secure key storage and platform (entity) integrity checks.

The specific procedures for key management fall outside the scope of this article. Instead, we propose a general procedure for defining the entity key (identity). During the registration operation, the device administrator places the device in an RF-shielded chamber to minimize potential interference that could impact the radio waves emitted by the device. Using specialized software and measurement equipment, the distinct characteristics of the device undergo a series of tests to establish a unique identity profile. In this study, we proposed a hybrid approach that combines several fingerprinting methods, primarily based on the parameters of the generated radio signals. The rationale for this selection is:

- limitations arising from the heterogeneity of the environment and the need to maintain the mobility of IoT devices;
- devices' vulnerability to extreme environmental factors (e.g., temperature, humidity);
- autonomy from the protocols used in the network.

The complete entity management process is conducted through the Communication Layer. Once the identity is established, it can be securely stored using the designated storage solutions on the device. Furthermore, the identity will also be recorded in the Distributed Ledger Layer and may optionally be retained in the off-chain stores of the Streams Microservice Layer.

4. Framework Basic Operations

This section delineates the main operations of our experimental framework, conscientiously examining the interrelationships among system layers and the flow of messages. Notably, we have integrated certain elements conceptualized by Jarosz et al. concerning a novel LAAFFI protocol [20].

4.1. Security Mechanisms and Message Types

As outlined in Section 3.4, the Fingerprint Enrichment Layer utilizes the Protocol Forwarder Component, which is specifically designed to manage connection-less streams and convert them into a connection-oriented format. This process employs an ETL (Extract, Transform, Load) mechanism, where a series of functions is applied to the extracted data, allowing it to be transformed into a standardized format. Moreover, the producer utilizing a data serialization mechanism, is solely responsible for determining how to convert the data from a specific protocol (such as MQTT) into byte representation. In contrast, the consumer defines how to interpret the byte string received from the broker through the deserialization process.

Within our environment, we proposed two primary types of messages that can be assigned to a single data stream. The first type is the data stream authentication message, referred to as AUTH_MSG (Figure 4). The second type is the session-related data stream message, known as DATA_MSG (Figure 5). Specific message fields are described below:

 doi:10.20944/preprints202504.0237.v1

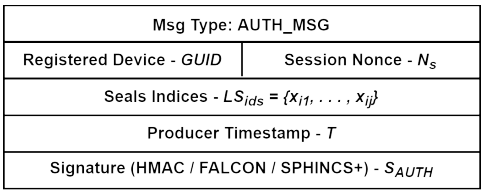

Figure 4. AUTH_MSG structure.

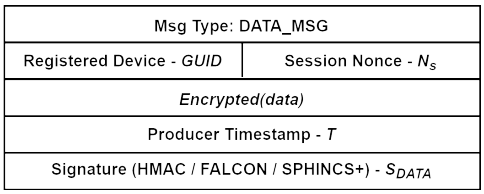

Figure 5. DATA_MSG structure.

- Globally Unique Identifier, $GUID$ - is assigned to the entity during the registration process in the Device Maintain Layer. It functions as a unique identifier, ensuring that each device can be distinctly recognized within a set of registered devices. For example, it may be represented as a human-readable combination of the federated organization name, type, and number, such as ORG1-SENSOR-0001;
- Session Nonce, N_s - is a unique pseudo-random value generated by the entity that identifies a specific data stream, thus facilitating the correlation between the AUTH_MSG and the DATA_MSG.
- Seals Indices, $LS_{ids} = \{x_{i_1}, \ldots, x_{i_j}\}$ - consist of a subset of indexes for the Seals selected from the Secure Seal Store - $Sl_{store} = \{Sl_1, Sl_2, \ldots, Sl_k\}$, where Seal - $Sl_x = H(FID_x \oplus HSMV_k)$. For each specific Seal we proposed to use a hashed $H()$ exclusive OR multiplication $\oplus$ of the entity Fingerprint Sample - FID_x recorded in the Entity Features Store - $EF_{store} = \{FID_1, FID_2, \ldots, FID_n\}$ combined with internal parameters from the Hardware Security Module - $HSMV_k$;
- Timestamp, T - is used as a protection mechanism against replay attacks, when the Kafka topic is configured to use *CreateTime* for timestamps, the timestamp of a record will be the time that the producer sets when the record (message) is created.
- Message Signatures, S_{AUTH}, S_{DATA} - are utilized to guarantee both the integrity and authenticity of the data streams. These signatures are compared to signature values calculated during the processing of messages within the Streams Microservice Layer. The Seal - Sl_x or Subset of Seals - $SSl_{store} \subseteq Sl_{store}$ and Session Nonce works in conjunction with a key-derivation function to generate the key for the signature function $Seal_{sk} = Key_{gen}(SSl_{store}, N_s)$. The use of signatures fits naturally into the architecture of our system because of the Kafka message format autonomy (independence).

The message structures of AUTH_MSG and DATA_MSG that the Kafka Broker can handle (store, queue) are illustrated in Figures 6 and 7. The *Kafka_Key* and *Kafka_Value* consist of sequences of bytes that form the message. The *Kafka_Key* plays a crucial role in directing a message to a specific partition within the Kafka topic. When a key is provided, all messages associated with that key are directed to the same partition, ensuring they are processed sequentially. The *Kafka_Value* contains the data that consumers will read and process. Additionally, we have included optional header fields related to the implementation of specialized feature (behavior-based) analyzers deployed within the Fingerprint Enrichment Layer (see Figure 6).

 doi:10.20944/preprints202504.0237.v1

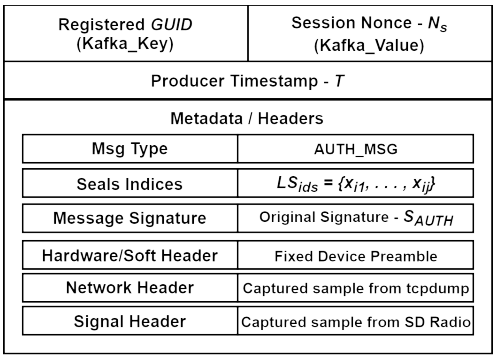

Figure 6. AUTH_MSG Kafka structure.

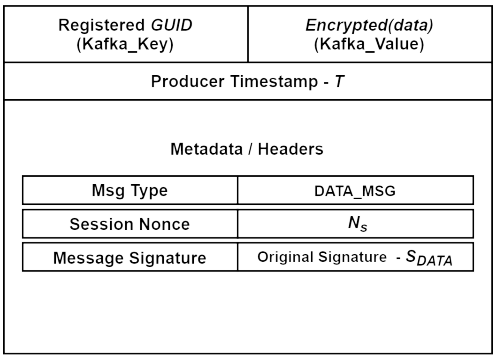

Figure 7. DATA_MSG Kafka structure.

We acknowledge the critical need to safeguard data throughout its various states, whether in-process, in-transit, or at-rest. In our deployment, we have employed SSL/TLS communication solely between the Streams Microservice Layer and the Distributed Ledger Layer. Our framework follows a data-centric approach, which facilitates the encryption of data stream payloads using AES, along with data authentication through device fingerprints. This approach allows us to bypass the need for communication protection within the Data Queue Layer, particularly among producers, subscribers, and brokers. This decision reduces additional overhead affecting microservice verification latency.

The aforementioned approach, in conjunction with the Kafka serialization and deserialization mechanism, facilitates independent data exchange between producers and subscribers. It is essential to highlight that microservice is unable to access the data payload, as it may be encrypted using a key that has been exchanged in advance between producers and subscribers. Nonetheless, this limitation does not impact the verification process of the data streams. A detailed discussion of this topic was provided in our previous work [4].

Additionally, to safeguard the EF_{store} and Sl_{store} during transit, we propose utilizing an encryption mechanism that employs one-time pre-shared keys, which will be securely maintained solely within the entity (e.g., Event Listener) and the microservice instance. To protect these stores in an off-chain environment (at-rest state), we recommend implementing the Confidential Computing strategy and Secure Enclaves, as detailed in Section 3.8.

4.2. Entity Registration

The top-level sequence diagram (Figure 8) outlines the message flow and the specific actions (steps) that are related to the operation of registering entity identity:

- Step 1: the device registration process begins with the Device Maintain Layer defining an entity's identity through hybrid fingerprint techniques. This operation is conducted within a secure enclave (e.g., TEE);
- Step 2: the entity's identity, represented by fingerprint samples is recorded in the Entity Features Store - $EF_{store} = \{FID_1, FID_2, \ldots, FID_n\}$;

 doi:10.20944/preprints202504.0237.v1

- Step 3: a Set of Seals - $Sl_{store} = \{Sl_1, Sl_2, \ldots, Sl_k\}$ is generated based on a subset of fingerprint samples from the Features Store. For each specific Seal - Sl_x, we proposed to use a hashed $H()$ exclusive OR multiplication $\oplus$ of the Feature Sample - FID_x combined with internal parameters from the Hardware Security Module - $HSMV_p$;
- Step 4: the chaincode, a component of the Distributed Ledger Layer that manages the secure transaction of adding a new identity to the ledger, is invoked. This transaction incorporates the Entity Features and Secure Seal Stores as part of its payload (EF_{store}, Sl_{store}).
- Step 5: upon receiving the necessary approvals from the organizations specified by the endorsement policy, the transaction is executed. This step is crucial for ensuring the integrity and transparency of the ledger, as it guarantees that all identities are accurately and securely recorded;
- Step 6: the Entity Features and Secure Seal Store are recorded in the distributed ledger through separate channels: an Entity Features Channel - $EF_{channel}$ and a Secure Seal Channel - $Sl_{channel}$, respectively;
- Step 7: a confirmation of the registration is sent back to the Device Maintain Layer;
- Step 8: considering the Confidential Computing Strategy, the Sl_{store} is written to the Entity's Secure Enclave. Any cached values related to the registration process should be cleared (wiped out).

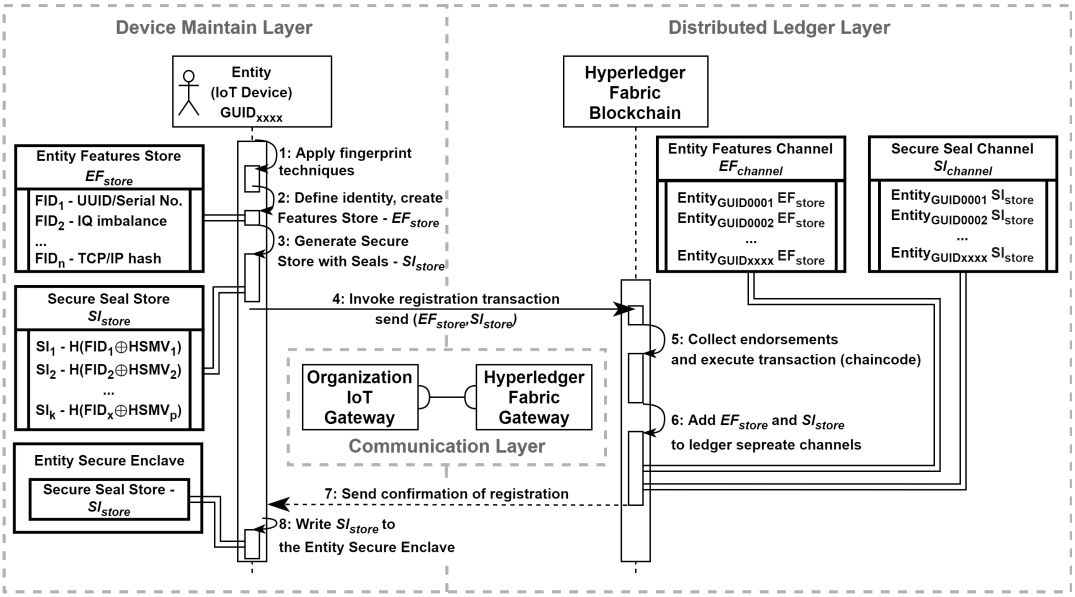

Figure 8. Top-level sequence diagram for registering entity identity.

4.3. Blockchain Event Listener Application

As an enhancement to the entity registration operation, we proposed to incorporate device identity into the local off-chain data store, which is a component of the Streams Microservice Layer. This improvement aims to reduce time delays during message verification by eliminating the need for a ledger query step (micro-caching mechanism). Figure 9 presents a top-level sequence diagram for the operation, illustrating the flow of messages and interactions involved (steps):

- Step 1-4: remain as for registering entity identity top-sequence diagram (Figure 8);
- Step 5: upon receiving the necessary approvals from the organizations specified by the endorsement policy, the transaction is executed (the Distributed Ledger Layer);
- Step 6: the Entity Features and Secure Seal Stores are recorded in the distributed ledger through separate channels: $EF_{channel}$, $Sl_{channel}$;
- Step 7: an application called Blockchain Event Listener monitors events that are emitted by the Distributed Ledger Layer. This application represents a special entity within the Publisher's Layer.

 doi:10.20944/preprints202504.0237.v1

As a result of the approved and executed transaction, the *GUID* and the Seal Store are written to the Event payload. Then the Event is emitted by the Distributed Ledger Layer.

- Step 8: a confirmation of the registration is sent back to the Device Maintain Layer;
- Step 9: the Secure Seal Store is written to the entity's secure enclave (the Device Maintain Layer);
- Step 10: the Blockchain Event Listener (the Publisher's Layer) interprets the occurrence of the Event and the Seal Store is extracted from the Event payload;
- Step 11: the Seal Store extracted from the Event payload is written to the Data Queue Layer (Kafka Cluster). The dedicated Kafka topic is utilized for this purpose;
- Step: 12: the Streams Microservice Layer reads the Seal Store from the cluster in a sequential manner;
- Step 13: during *process()* method handled by the Streams Microservice Layer the Seal Store is added to the local off-chain data store and can be utilized by other streams microservices within the pool.

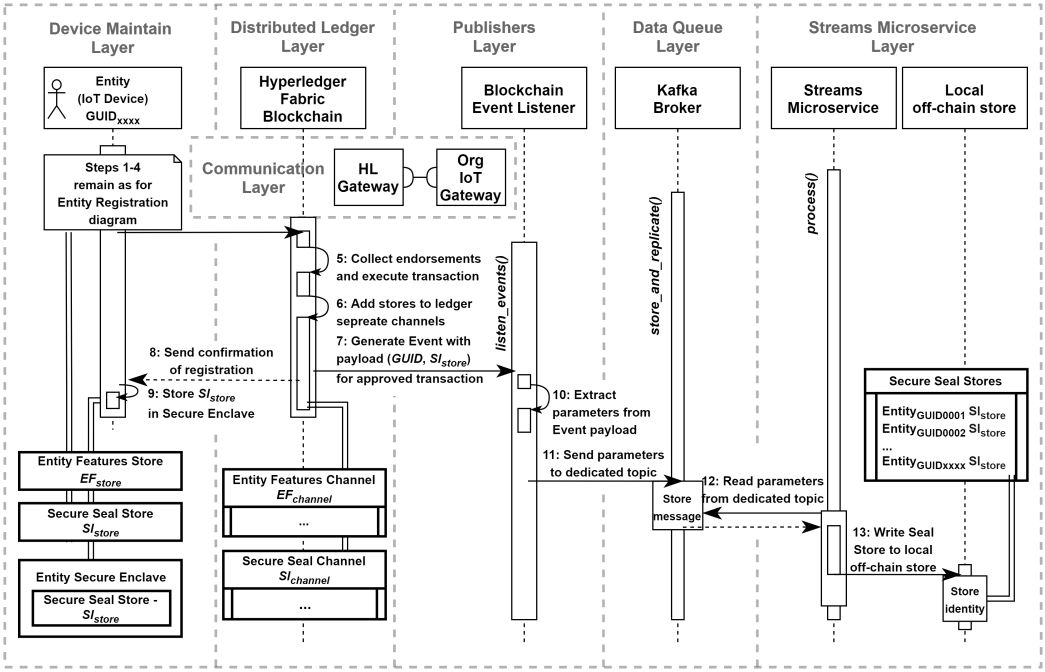

Figure 9. Top-level sequence diagram for adding identities using a Blockchain Event Listener.

4.4. Data Streams Verification

A tailored stream processing algorithm has been proposed. The top-level sequence diagram (Figure 10) outlines the message flow and the specific actions (steps) that are related to the operation of data streams verification. Additionally, the local off-chain data store has been incorporated into the Streams Microservice Layer, which can enhance the overall performance of the system:

- Step 1: when an entity (IoT device) from the Publishers Layer intends to transmit a data stream, it invokes the *generate_data_stream()* method. During this execution, initial parameters for the cryptographic primitive (sealing) are chosen from the Secure Seal Store - $Sl_{store} = \{Sl_1, Sl_2, \ldots, Sl_k\}$, where Seal - $Sl_x = H(FID_x \oplus HSMV_k)$ or Subset of Seals - $SSl_{store} \subseteq Sl_{store}$ is selected along with its corresponding indices - $LS_{ids} = \{x_{i_1}, \ldots, x_{i_j}\}$, and a Session Nonce - N_s is generated;
- Step 2a: for the chosen Seal and the Session Nonce, a Session Seal Key is generated $Seal_{sk} = Key_{gen}(SSl_{store}, N_s)$. Next, the AUTH_MSG is crafted and sealed using a signature algorithm founded on the specified parameters: $S_{AUTH_{N_s}} = Sign(AUTH_{MSG}, Seal_{sk})$;
- Step 2b: subsequently, the session-related data stream messages are sealed using the Session Seal Key generated in Step 2a: $S_{DATA_{N_s i}} = Sign(DATA_{MSG}, Seal_{sk})$;

 doi:10.20944/preprints202504.0237.v1

- Step 3a: the sealed AUTH_MSG is transmitted through a reliable communication channel via the Fingerprint Enrichment Layer to the Data Queue Layer (Kafka Cluster), where it is stored under a designated topic. The AUTH_MSG includes: $(GUID, LS_{ids}, N_s, T, S_{AUTH_{N_s}})$;
- Step 3b: the sealed DATA_MSG messages are transmitted in the same manner as described in Step 3a. The DATA_MSG contains the $(GUID, N_s, T, S_{DATA_{N_s i}})$;
- Step 4: optionally, within the Fingerprint Enrichment Layer, specialized behavior-based analyzers handle the *sampling()* method to capture fingerprint samples associated with a specific device's AUTH_MSG;
- Step 5a: the *handle_auth()* method transforms the raw AUTH_MSG message into a structure suitable for loading into the Kafka Broker;
- Step 5b: the *handle_data()* method transforms the raw DATA_MSG message into a format that can be processed by the Data Queue Layer;
- Step 6a: the data stream authentication message - AUTH_MSG is forwarded to the Data Queue Layer;
- Step 6b: the session-related messages - DATA_MSG are forwarded;
- Step 7: the *process()* method of the Streams Microservice Layer sequentially reads the AUTH_MSG message from the specified topic;
- Step 8: the parameters $(GUID, LS_{ids}, N_s, T, S_{AUTH_{N_s}})$ are extracted from the AUTH_MSG for subsequent verification.
- Step 9a: the micro-caching mechanism is utilized. The Verify Streams Microservice queries the local off-chain storage to retrieve the device fingerprint (identity) that sealed the message. The query is composed of the GUID and the Seals Indices $(GUID, LS_{ids})$. As an extension, a stored procedure or a trigger-like mechanism can be employed with the local off-chain storage for generating the Session Seal Key $Seal_{sk_x} = Key_{gen}(SSl_{store_x}, N_s)$. In this case, the body query is extended with the Session Nonce parameter (N_s), allowing for a reduction in the number of steps necessary before proceeding to Step 12. The Confidential Computing Strategy should be implemented to ensure the strict protection of data in transit.
- Step 9b: the appropriate identity $(GUID, SSl_{store})$ is returned, or an identity *Not Found Error* is generated. If the extension mentioned in Step 9a is applied, an alternate response of $(GUID, Seal_{sk_x})$ is returned.
- Step 10: if the Session Seal Key - $Seal_{sk_x}$ is successfully generated (or obtained), the steps related to querying the Distributed Ledger Layer (Hyperledger Fabric) are omitted, and Step 12 is executed instead.
- Step 11a: otherwise, the identity *Not Found Error* results in a query via the Communication Layer to the Hyperledger Fabric Gateway service of the Distributed Ledger Layer.
- Step 11b: the chaincode (transaction) is executed to generate $Seal_{sk_x}$ based on the device GUID, a list of seal indices, and the Session Nonce $(GUID, LS_{ids}, N_s)$.
- Step 11c: the device GUID along with the Session Seal Key is returned from the ledger $(GUID, Seal_{sk_x})$, or a relevant error is produced.
- Step 12: entity identities are compared by verifying the extracted signature $S_{AUTH_{N_s}}$ that sealed the AUTH_MSG against the signature - $S_{AUTH_{N_s x}} = Sign(AUTH_{MSG}, Seal_{sk_x})$ with the Session Seal Key returned from Step 10, Step 11c, or optionally Step 9b. If the signatures match $(S_{AUTH_{N_s}} = S_{AUTH_{N_s x}})$, the AUTH_MSG undergoing the verification process will be preserved. If they do not match, the message may either be discarded or stored in a separate queue for the identification of potentially malicious or faulty devices.
- Step 13: the Streams Microservice sequentially reads and verifies the session-related messages - DATA_MSG_{Nsi}.
- Step 14: the Session Seal Key from either Step 10 or Step 11c is used, and the signatures of the DATA_MSG are compared.
- Step 15: verified DATA_MSG is sent to the appropriate topic.

 doi:10.20944/preprints202504.0237.v1

- Step 16: an entity representing the Subscribers Layer reads the verified data streams *read_streams()* based on the subscribed topics and the authorization policy.

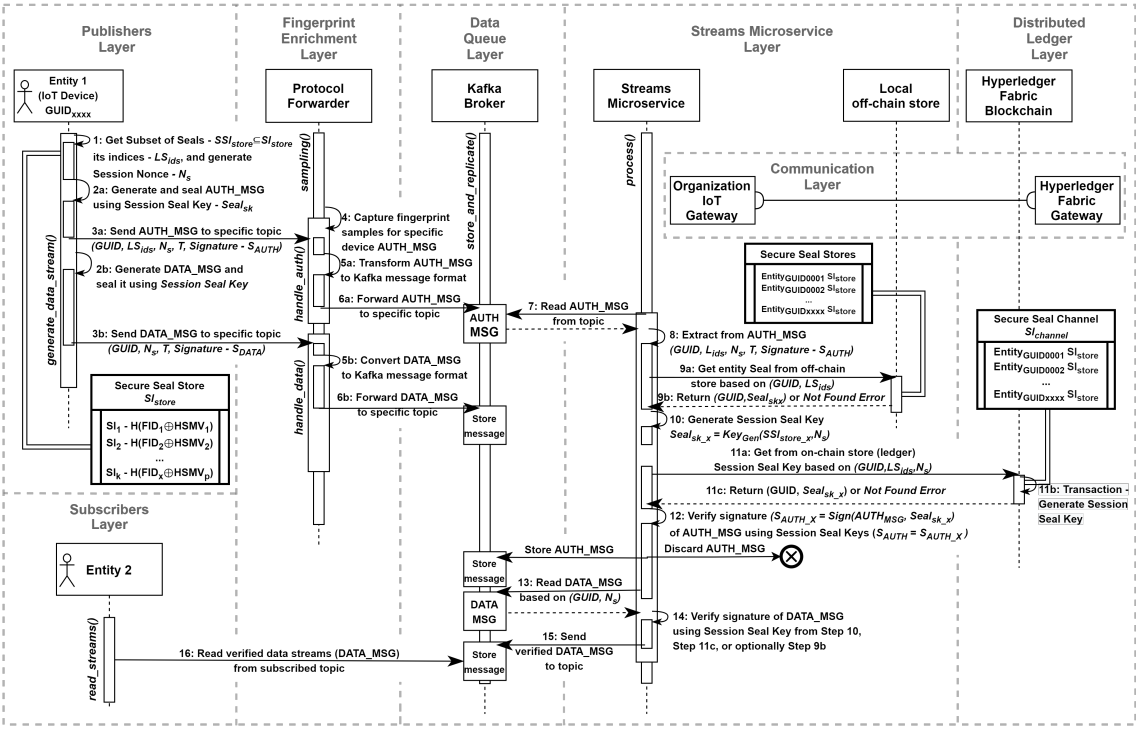

Figure 10. Sequence diagram for verification of data streams operation.

5. Framework Evaluation

Benchmarking the performance of streaming data processing systems poses a considerable challenge due to the complexities related to the global concept of time. This section provides benchmarks for our framework in two setups: the Amazon Web Services (AWS) cloud environment and the Raspberry Pi device-based environment. In the AWS environment, we measured the average times for consumers to read the data stream that was processed through microservices utilizing the Java Kafka Streams API. Our objective was to validate the applicability of our framework in the context of audiovisual streams and to assess its computational stability. Conversely, the Raspberry Pi setup focused on gathering the latency metrics of key internal operations on resource-constrained devices. Additionally, we compared the operation latencies of implementations in both Java (Kafka Streams API library) and Go (Sarama library) programming languages.

5.1. Cloud Setup

Our experiment seeks to verify our framework's capability to process audiovisual streams in a distributed and federated cloud environment. Consequently, we deployed the Data Queue (Kafka Cluster) and Distributed Ledger in separate locations (AWS regions). To ensure durability (fault tolerance), we deployed three Kafka Brokers and established two ledger peers for each federated organization. Communication between the layers including the Streams Microservice Layer is enabled through a single AWS Private Link. Figure 11 illustrates the various components of our experimental framework deployed using AWS technology (Setup I), comprising:

- two AWS regions to simulate geographical distances: MSK Region that belongs to Org1, and AMB Region for Org1 and Org2. The Multi-Region link was set using VPC Peering Connection;
- a single Amazon Managed Streaming for Apache Kafka version 2.8.1, deployed in the MSK Region isolated (availability) zones, where three kafka.t3.small brokers (vCPU: 2, Memory: 2GiB, Network Bandwidth: 5Gbps) were set. Each broker has a default configuration with a single partition and a replication factor of 3;

 doi:10.20944/preprints202504.0237.v1

- a single Amazon Managed Blockchain Starter Edition for Hyperledger Fabric version 2.2, deployed in the AMB Region, where a single channel for identities was created within the blockchain network. Also, each member (Org1, Org2) of the channel has two nodes (peers) running of type bc.t3.small (vCPU: 2, Memory: 2GiB, Network Bandwidth: 5Gbps);
- two Amazon Elastic Compute Cloud (EC2) virtual machines of type t2.micro (vCPU: 1, Memory: 1GiB), deployed in the MSK Region in two isolated zones to simulate data stream producer and consumer;
- a single EC2 instance (t2.micro) deployed in the MSK Region with a streams microservice to verify messages;
- a single AWS PrivateLink interface (VPC Endpoint) that enables the communication between the microservice and elements of the Amazon Managed Blockchain Hyperledger Fabric.

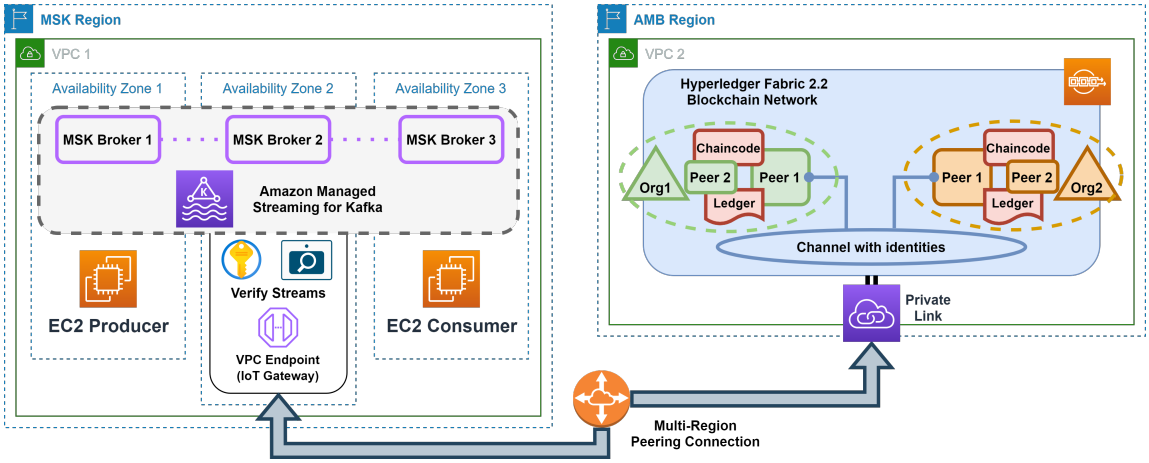

Figure 11. Setup I - overview of the AWS cloud-based environment.

The AWS cloud due to its pay-as-you-go model and pluggable architecture for the COTS services: Amazon Managed Streaming - Apache Kafka and Amazon Managed Blockchain - Hyperledger Fabric, enables efficient deployment of our framework. Simultaneously minimizing the operational costs associated with the provisioning, configuration, and maintenance of its various components. As a consequence, our framework is suitable for federated environments for which it is required to ensure zero-day interoperability.

5.2. Resource-Constrained Setup

In the second benchmark (Setup II), we chose resource-constrained devices to evaluate their utilization and computational power. We positioned the specific layers of our framework within the IoT environment locations: Actuators/Sensors, Edge, Fog, Data Center (Cloud). The Kafka Cluster was deployed on the Raspberry Pi 5 (RPi5) devices in the Edge Processing component, and virtualization technology was used to manage the distributed ledger in the Data Center location. For the Streams Microservice Layer placed in Fog, we used the Raspberry Pi 3B+ (RPi3), the RPi5, and a Standalone Laptop (SLap) platforms as a reference for microservice benchmarking. The Communication Layer included a Gigabit switch and an IEEE 802.11 router. Our setup simulates infrastructure placement in tactical military networks and can be adapted for scenarios involving civil components during HADR operations. Figure 12 represents the device-based environment, which includes:

- the Apache Kafka Cluster (Data Queue Layer) based on RPi5 devices (CPU: 2.4GHz Arm Cortex A76, Memory: 8GiB). Each Kafka Broker has a default configuration with a single partition and a replication factor of 3. Moreover, each broker was deployed at the Tactical level (Unit Area of Responsiblity, AoR) within the specific Subunit Command Posts to simulate geographical distances.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 April 2025 doi:10.20944/preprints202504.0237.v1

- the Streams Microservice Layer is an integral part of the Fog Processing component. It has been designed to work with three different devices: RPi3 (CPU: 1.4GHz Arm Cortex A53, Memory: 1GiB), RPi5 (CPU: 2.4GHz Arm Cortex A76, Memory: 8GiB), and SLap (CPU: 3.3GHz i5-12450H, Memory: 16GB).
- the Hyperledger Fabric Blockchain Network is deployed on a server using virtualization technology (Data Centers Component). Within the blockchain network, a single identity channel was created. Each member (Org1, Org2) of the channel has two nodes (peers) running on virtual machines (vCPU: 2, Memory: 2GiB);
- three RPi3 devices were deployed at each Subunit AoR (Tactical level) to be used as producers and consumers.

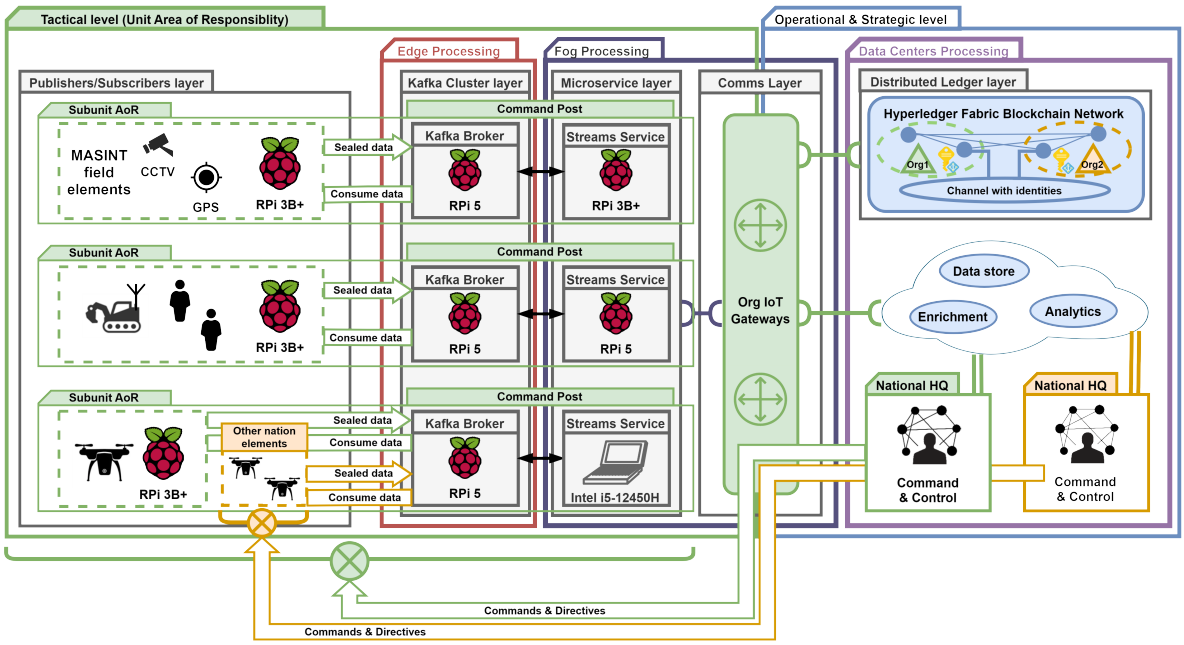

Figure 12. Setup II - overview of the Raspberry Pi device-based environment.

Furthermore, in Section 3 we presented the meaning of the Computing Continuum concept. By relocating the Ledger Peer from the Data Center to the Fog location, we refined Setup II to facilitate benchmarking with the mentioned concept (Figure 13).

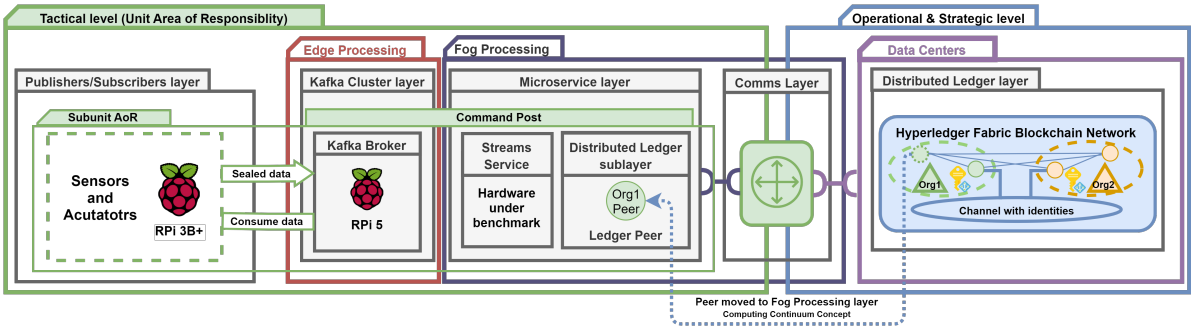

Figure 13. Setup II - the Computing Continuum concept enabled.

The Raspberry Pi device-based environment facilitates straightforward out-of-the-box deployment of our framework while maintaining low operational costs. Moreover, the presented environment promotes the Computing Continuum concept, which can enhance the deployment of services at the tactical level in settings characterized by DIL networks, as well as in federated environments that require zero-day interoperability.

 doi:10.20944/preprints202504.0237.v1

Additionally, we are considering integrating information classification approaches with the CC concept to enable the processing and sharing of relevant IOs based on their Value of Information. The versatility of Raspberry Pi devices allows for integration with a Neural Processing Unit, a specialized processor designed to accelerate artificial intelligence and machine learning tasks. This feature can facilitate parallel processing of large data volumes within the Fog component, making it especially well-suited for applications involving Contextual Dissemination through image recognition or natural language processing, as well as for Data Quality Assessment.

Moreover, the hardware of these devices is compatible with various operating systems, including Windows 10 IoT Core and Linux OS. Lastly, the general-purpose input/output (GPIO) pins provide the flexibility to experiment with a range of communication protocols such as Sigfox, LoRaWAN, NB-IoT, Zigbee, and BLE.

5.3. Processing Systems Benchmarking

In conducting performance studies (benchmarks) of streaming data processing systems, it is necessary to consider following key metrics [32], [33]: latency, throughput, the usage of hardware-software resources (CPU, RAM), and power consumption (PC). Furthermore, the overall performance evaluation can be affected by the input parameters (e.g., system configuration) and processing scenarios (workloads) [31]. In the context of the proposed framework, several parameters are listed below:

- configuration of the Data Queue Layer: number of brokers, partitions, and data streams replication factor;
- parallelization (horizontal-scaling) of stream processors (microservices);
- kind (e.g., windowed aggregation) and type of operations (e.g., stateless);
- number of organizations that joined a Federated IoT Environment and registered devices (identity count);
- number of peers of the Distributed Ledger Layer;
- selected programming language for microservices and chaincodes.

Generally, latency defines as the interval of time it takes for a system (platform) under test (SUT) to process a message, calculated from the moment the input message is read from the source until the output message is written by SUT. Hence, it is important to distinguish the latency metric [32] into its two types: *event-time latency* and *processing-time latency*. The first mentioned refers to the interval between a timestamp assigned to the input message by the source and the time the SUT generates an output message. The second one refers to the interval calculated between the time when an input message is ingested (read) by the SUT, and the time the SUT generates an output message.

Data Dissemination System Benchmarking

In the context of the Apache Kafka, latency (end-to-end, E2E) is defined as the total time from when application logic produces a record using *KafkaProducer.send()* to when that record can be consumed by the application logic through *KafkaConsumer.poll()*. This E2E latency includes various intermediate operations that can affect the overall duration. The publish operation time encompasses flying time, queuing time, and internal record processing time. Flying time refers to the duration for transmitting the record to the leader broker, queuing time pertains to the time taken to add the request to the broker queue by network thread, and record processing time involves the reading of the request from the queue and the appending of logs (records). Furthermore, the replication factor and cluster load impact commit time, which is the time required for the slowest in-sync follower broker to retrieve a record from the leader. Moreover, the Catch-Up operation refers to the duration a consumer takes to read the latest record ($Offset_N$) while its offset pointer is lagging ($Offset_K$, where $K < N$). Lastly, fetch operation time impacts Kafka consumer record read latency, as the consumer continually polls the leader broker for more data from its subscribed topic partition.

As previously noted, various factors can influence latency. In our experiments, we focused on the detailed configuration of the Kafka parameters for our microservice that operates both as a producer

 doi:10.20944/preprints202504.0237.v1

and a consumer during single record processing. Consequently, the following configuration was uniformly applied across all scenarios:

- the produce operation was set without artificial delay, and a record will be sent immediately to the Kafka Cluster;
- microservice reliability was configured to complete operations only after all in-sync replicas have received the record and sent back an acknowledgment. This setting includes both publish and commit times, which adds latency overhead to our microservice;
- the replication factor of a single record was set to three and the number of replica fetcher threads per source broker was set to the default value of one;
- idempotence was set. This enables the Kafka mechanism to identify and eliminate duplicate messages by comparing a unique sequence number of each record sent to a partition;
- there was no separate listener for replication traffic on brokers and for client (producer/consumer) traffic;
- the microservice record consumer has been configured to operate without incurring any additional latency, while ensuring that the processing guarantee is established as exactly once. Upon submission of a fetch request by microservice, a response is provided immediately when a single byte of data (record) becomes available from the Kafka Broker.

Distributed Ledger System Benchmarking

Regarding the distributed ledger operations benchmarking, we focused on collecting latency metrics from two sources: one from the microservice ledger query operation (DL Query), and another directly from the log entries of the Ledger Peer server. By analyzing the peer logs, we focused on two types of entry: the duration of gRPC call and the time required to evaluate chaincode (Chaincode Execution), specifically to acquire the device's seal from the ledger. The first mentioned metric is essential for monitoring and performance analysis, as it offers valuable insights into the efficiency and responsiveness of the services involved in gRPC communication. Conversely, the second metric pertains to a peer node that endorses transactions, reflecting the computation burden.

In examining potential peer (database) microcaching mechanisms, we can differentiate between two phases of the Ledger Peer: the Cold Peer (CP) and the Hot Peer (HP). The CP occurs when the peer is restarted each time before the microservice is launched. Conversely, the HP refers to a state in which the peer has already been queried for all identities present in the ledger, and then the microservice under test is launched. These phases can significantly affect the performance of the chaincode latency [35,36]. Notably, we have selected Apache CouchDB as the primary backend database for our ledger peers. This NoSQL database employs a schema-free JSON format and a B-tree indexing system, making it ideal for complex, read-heavy queries, although it is not optimized for write operations like a Log-Structured Merge-tree. Fortunately, our framework primarily focuses on ledger read operations.

Microservice Benchmarking

In the case of the Go programming language, it is statically compiled into a machine code, resulting in a single executable that includes all necessary dependencies and can be run directly on the operating system. This approach eliminates the two-step compilation process utilized by Java, which embodies the concept of "write once, run anywhere".

At the onset of a Java microservice, the Java Virtual Machine (JVM) performs Just-In-Time (JIT) dynamic compilation, leading to fluctuations in internal operation latencies during the initial execution phase. This phase, commonly referred to as the Warm-Up State (WS), involves the JIT analyzes the bytecode (pre-compiled form) and translates it into machine code. It identifies frequently executed methods and loops, omits unused code segments, applies constant folding, and manages object allocation. Once this optimization process concludes, the microservice attains a Steady State (SS) of performance [37].

 doi:10.20944/preprints202504.0237.v1

We recognize the impact of JIT optimization on our metrics, so as a mitigation strategy during our benchmarks, we focused on capturing latencies for microservices operating in both the WS and the SS. Nevertheless, fluctuations in performance during the WS state can lead to notable latencies, especially on resource-constrained platforms.

Our implementation involves collecting the durations of key microservice operations: Cryptographic Primitives Initialization, Distributed Ledger Query, HMAC Validation, AES Decryption, and Message Forwarding. Each operation will be executed a fixed number of times based on the selected message volume. Lastly, the main method *process()*, which includes these operations, will be repeated multiple times using fresh JVM instances (forks). A similar approach will be employed for the Go microservice.

5.4. Resource Utilization

To effectively manage device resource utilization during benchmarks and identify potential anomalies, we employed the real-time monitoring system based on the Node Exporter service. This service allows for accumulation of an extensive range of metrics related to hardware and operating systems, such as RAM, I/O disk, and CPU usage. The collected data can be then ingested and stored in Prometheus [38], an open-source monitoring system that implements a pull model. For in-depth analysis, we utilized Grafana, a web application that provides interactive visualizations and customizable dashboards, allowing us to derive valuable insights regarding the tested platforms.

5.5. Power Consumption

During benchmarking, it is crucial to monitor potential voltage spikes that may decrease CPU instructions per clock (IPC), as these issues can adversely affect microservice execution. Moreover, maintaining a focus on power consumption (PC) facilitates the identification of high-power instructions (operations) within an actively running microservice under specific workloads, allowing for further optimization.

It is also important to detect throttle events that could result in instability and crashes of microservices. For the Raspberry Pi platforms, the risk of such events occurs when the CPU temperature, as measured by the built-in sensor, ranges from 80 to 85°C. As a result, this leads to a gradual throttling of the CPU cores, causing a reduction in their frequencies.

We recognize that PC can fluctuate, and the current draw may vary based on usage. Therefore, during the benchmarking process, we took the following preliminary steps:

- devices for testing were deployed in their unmodified state, without any overclocking;
- the lower limit for CPU temperature (throttle protection) has not been modified;
- no additional devices were connected to the GPIO and USB ports; an SSH connection was used to control the devices.

To monitor power consumption during our tests, we employed the RPi5 Power Management Integrated Circuit (PMIC, Renesas DA9091 “Gilmour”) [39], which integrates eight switch-mode power supplies to deliver various voltages required by the PCB components, including the Cortex-A76 CPU cores (VDD_CORE). We integrated a revision-agnostic software tool called *vcgencmd*, which provides access to information about the voltage and current for each component managed by the PMIC. This tool is particularly versatile for monitoring various parameters, including the CPU core’s status, temperature, and throttle state (represented as a bit-pattern, flag: 0x40000). Additionally, we implemented the Prometheus Scraper that operates on a pull model to collect power metrics from devices under test. These metrics are then ingested into a Prometheus instance for further visualization in Grafana (Figure 14).

 doi:10.20944/preprints202504.0237.v1

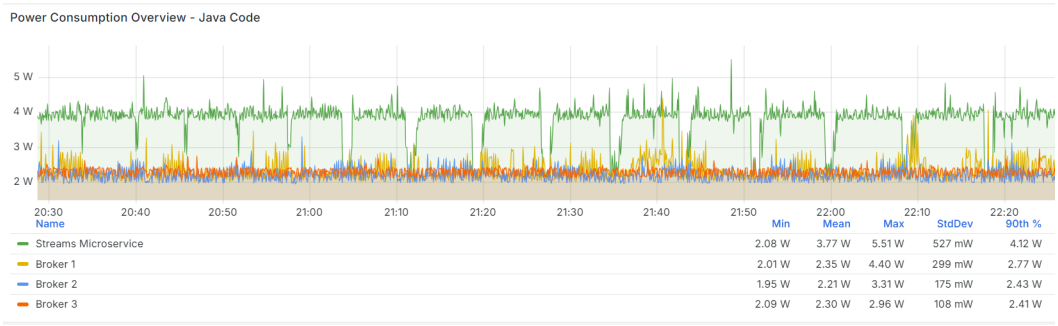

Figure 14. Power consumption visualization in the Grafana tool.

5.6. Processing Scenarios

Scenarios of Setup I - Cloud-Based Environment

In this paper, we examined the processing time latency of microservices developed using the Java Kafka Streams API and deployed in the AWS cloud-based environment (Setup I). Our objective was to validate the applicability of our framework in the context of audiovisual streams and to assess its computational stability. We designed two scenarios for this purpose:

- Setup I Scenario I: involved verifying the input (sealed) message by performing a comparison operation between the extracted device identity, with the identity stored in the distributed ledger.
- Setup I Scenario II: involved verifying the sealed message by comparing the extracted device identity with the identity stored in the off-chain data store. In this scenario, all device identities from the ledger were synchronized and stored in the off-chain store.

In the scenarios outlined, the burst-at-startup technique was employed [31]. This process involved generating and sealing each input message with a pseudo-random device identity in advance. Once a predetermined number of input messages had been created, a single instance of the microservice responsible for their verification was activated. Furthermore, we aimed to extend workloads utilizing varying quantities of registered identities within the distributed ledger.

Scenarios of Setup II - Resource-Constrained Environment

For the Raspberry Pi resource-constrained environment (Setup II) we benchmarked microservices developed in Java (Kafka Streams API) and the Go (Sarama) programming language. Furthermore, beyond the full processing-time, we measured the duration of microservices' significant (time-consuming) operations: Cryptographic Primitives Initialization, Distributed Ledger Query, HMAC Validation, AES Decryption, and Message Forwarding. Moreover, we took an insightful look at hardware and software utilization through real-time monitoring of the whole Setup II. We planned to conduct bellow scenarios:

- Setup II Scenario I: involved verifying the sealed message by comparing the extracted device identity with the identity stored in the distributed ledger located in the Data Center component, while also applying the burst at startup technique.
- Setup II Scenario II: involved verifying the sealed message by comparing the extracted device identity with the identity stored in the Ledger Peer relocated to the Fog Processing component (the Continuum Computing concept enabled). The burst at startup technique was applied.
- Setup II Scenario III: involved verifying the sealed message by comparing the extracted device identity with the identity stored in the distributed ledger within the Data Center component. Throughout this scenario, continuous writing operations were conducted to the Data Queue Layer, maintaining a consistent message throughput of 400 messages per second [33].

 doi:10.20944/preprints202504.0237.v1

5.7. Reference Points for Experiments

Processing Scenarios:

To compare the RPi3 and the RPi5 platforms we present results from Setup II Scenario I (Steady State, Hot Peer phase, 10 000 identities, 10 000 messages), derived from microservice benchmarking on the SLap platform (Table 1 – Java, Table 2 – Go). For consistency during further reading, the tables detailing microservice latency present averaged results with the average absolute deviation in brackets, calculated from 10 launches (forks) of the microservice. The following metrics were employed to evaluate the results:

- average latency (Avg) and average absolute deviations of data points from their mean value (Avg Dev);
- minimum (Min) and maximum (Max) latency;
- standard deviation based on the entire population (Pop Std Dev);
- quantiles of order: p90, p95, p99 (Percentiles [p=0.9; p=0.95; p=0.99]).

The Java-based Verify Streams Microservice took an average of 5.76 (±0.10) ms, with the minimum delay recorded at 3.59 (±0.07) ms, and an average latency deviation of 0.90 (±0.08) ms. The long-tail latencies of the p90 and p95 quantiles were 7.04 (±0.17) and 7.81 (±0.24) ms, respectively. The most time-intensive operation was the DL Query, which averaged 5.69 (±0.10) ms. Based on the log entries, the Chaincode Execution operation took an average of 1.10 (±0.02) ms, while full gRPC call lasted 2.47 (±0.11) ms. For the Message Forwarding operation - p90 percentiles did not exceed 0.051 (±0.0016) ms.

Table 1. Setup II Scenario I - Processing-time latency metrics (in ms) for Java code.

Operation	Avg	Avg Dev	Min	Max	Pop Std Dev	Percentiles [p=0.9; p=0.95; p=0.99]
Crypt. Prim. Init	0.003 (0.0001)	0.001 (0.0001)	0.001 (0.0001)	0.489 (0.0237)	0.005 (0.0003)	[0.004 (0.0003); 0.006 (0.0002); 0.009 (0.0005)]
DL Query	5.69 (0.10)	0.89 (0.08)	3.56 (0.07)	57.36 (19.42)	1.74 (0.36)	[6.95 (0.17); 7.71 (0.23); 10.96 (0.78)]
HMAC Validation	0.006 (0.0002)	0.002 (0.0001)	0.001 (0.0001)	0.129 (0.0801)	0.004 (0.0004)	[0.009 (0.0003); 0.012 (0.0004); 0.022 (0.0009)]
AES Decryption	0.024 (0.0006)	0.009 (0.0005)	0.006 (0.0004)	1.238 (0.7858)	0.021 (0.0064)	[0.041 (0.0015); 0.052 (0.0015); 0.083 (0.0028)]
Msg Forwarding	0.030 (0.0007)	0.015 (0.0005)	0.005 (0.0008)	8.578 (0.7048)	0.089 (0.0073)	[0.051 (0.0016); 0.077 (0.0017); 0.117 (0.0031)]
Full Processing	5.76 (0.10)	0.90 (0.08)	3.59 (0.07)	57.53 (19.41)	1.76 (0.36)	[7.04 (0.17); 7.81 (0.24); 11.05 (0.79)]
Chaincode Exec (logs)	1.10 (0.02)	0.19 (0.03)	0.00 (0.00)	19.30 (5.50)	0.49 (0.06)	[1.10 (0.18); 2.00 (0.00); 3.00 (0.00)]
gRPC Call (logs)	2.47 (0.11)	0.34 (0.07)	1.51 (0.02)	26.9 (22.99)	0.70 (0.19)	[2.93 (0.27); 3.29 (0.42); 4.71 (0.69)]

The Go-based microservice fork averaged 6.48 (±0.24) ms, with a minimum delay of 4.30 (±0.17) ms and an average latency deviation of 0.95 (±0.13) ms. The p90 and p95 latencies were measured at 7.87 (±0.47) and 9.08 (±0.63) ms, respectively. The most time-intensive operation was the DL Query, which averaged 4.23 (±0.23) ms. Chaincode Execution took 1.31 (±0.12) ms, while gRPC call had an average duration of 2.81 (±0.16) ms. For the p90 quantile, the latency for Message Forwarding operations was 2.53 (±0.08) ms.

Table 2. Setup II Scenario I - Processing-time latency metrics (in ms) for Go code.

Operation	Avg	Avg Dev	Min	Max	Pop Std Dev	Percentiles [p=0.9; p=0.95; p=0.99]
Crypt. Prim. Init	0.009 (0.0003)	0.002 (0.0001)	0.001 (0.0002)	0.061 (0.0134)	0.004 (0.0005)	[0.012 (0.0004); 0.015 (0.0011); 0.029 (0.0046)]
DL Query	4.23 (0.23)	0.81 (0.15)	2.64 (0.11)	20.13 (2.92)	1.40 (0.25)	[5.44 (0.54); 6.56 (0.60); 9.56 (1.31)]
HMAC Validation	0.011 (0.0003)	0.002 (0.0004)	0.002 (0.0002)	0.194 (0.1815)	0.008 (0.0049)	[0.012 (0.0005); 0.016 (0.001); 0.031 (0.0064)]
AES Decryption	0.009 (0.0006)	0.003 (0.0009)	0.001 (0.0002)	0.289 (0.2250)	0.012 (0.0074)	[0.013 (0.001); 0.018 (0.0009); 0.037 (0.0038)]
Msg Forwarding	2.197 (0.0914)	0.276 (0.0321)	1.499 (0.0590)	18.194 (2.8001)	0.760 (0.1696)	[2.526 (0.0845); 2.688 (0.0866); 3.813 (0.6615)]
Full Processing	6.48 (0.24)	0.95 (0.13)	4.30 (0.17)	25.18 (1.89)	1.67 (0.20)	[7.87 (0.47); 9.08 (0.63); 13.12 (1.55)]
Chaincode Exec (logs)	1.31 (0.12)	0.48 (0.13)	0.00 (0.00)	12.30 (4.04)	0.86 (0.18)	[1.90 (0.36); 2.9 (0.36); 4.9 (0.9)]
gRPC Call (logs)	2.81 (0.16)	0.58 (0.13)	1.72 (0.06)	14.44 (4.30)	0.99 (0.20)	[3.67 (0.41); 4.60 (0.51); 6.69 (0.86)]

Resource Utilization:

In light of Kingman's formula [40], which indicates that queue latencies increase exponentially with resource utilization, we assessed the performance of the Ledger Peer, Kafka Cluster, and microservice execution on the SLap platform. We did not observe any significant increases in resource utilization during our assessment. There were no events such as CPU idling, decreases in IPC, or throttling. Additionally, we recorded a stable temperature of 30°C. Furthermore, cluster I/O disk

 doi:10.20944/preprints202504.0237.v1

utilization and queuing delays did not significantly dominate the time taken for Message Forwarding operation. Lastly, the network traffic monitoring revealed that the microservice producer's limit on the number of unacknowledged requests sent to the leader broker remained within the configured upper limit of five, which did not affect the latency associated with the record publish.

Power Consumption:

According to the official documentation for the Raspberry Pi platforms [41], all models require a 5.1V power supply. The current requirements generally increase based on the specific model. Notably, the RPi5 consumes between 1.2W and 1.6W when connected to a power source, even when powered off. When connected to a power supply unit that delivers 5A, the RPi5 can provide up to 1.6A of power to downstream USB peripherals. Typical active current consumption for the RPi5 is 800mA, while for the RPi3, it is 500mA.

The average power consumption (PC) for the RPi3 platform during Idle state was 1.53W. Under Stress Conditions, the average PC increased to 4.335 (max. 6.834)W. The documentation does not include the PC for the RPi5 platform. However, we applied our own measurements, and the following tables present the average PC on the RPi5. Table 3 details the PC for the device under test (SM) and the Kafka Cluster (B1, B2, B3) during Idle state, where all associated services were inactive. In contrast, Table 4 illustrates the PC when all related services were operational in a listening state, and the CC concept was enabled, the Ledger Peer was running on the device under test (SM+CC). Although, there were no records processing or chaincode execution.

By comparing both states, we recorded a slightly different average PC: for Idle state (SM) - 2.12 (±0.18)W, and for Running state (SM+CC) - 2.15 (±0.22)W. The PC for Kafka Cluster during Running state did not exceeded 2.9W. Furthermore, during the Catch-Up state, where Ledger Peer full synchronization was required, PC was found to be 51% greater than during the Concurrent state (Table 5).

Table 3. Power Consumption - Idle state.

Device name	CPU idle (%)	RAM usage (%)	PC (W)
SM	98.6	9.8	2.12 (0.18)
B1	98.7	5.5	2.20 (0.14)
B2	98.2	5.6	2.25 (0.18)
B3	98.1	5.7	2.30 (0.16)

Table 4. Power Consumption - Running state.

Device name	CPU idle (%)	RAM usage (%)	PC (W)
SM + CC	98.1	20.6	2.15 (0.22)
B1	95.2	16.9	2.83 (0.51)
B2	94.8	17.1	2.71 (0.48)
B3	94.8	17.1	2.87 (0.49)

Table 5. Different Ledger Peer syncing states.

Sync state	CPU idle (%)	RAM usage (%)	PC (W)
No Sync	98.1	20.6	2.15 (0.22)
Catch Up	62.8	21.0	3.30 (0.72)
Concurrent	97.1	21.0	2.18 (0.22)

5.8. Benchmarks

The following section of the article provides the results achieved for workloads described in Section 5.6 along with resource utilization and power consumption of the platforms under test.

 doi:10.20944/preprints202504.0237.v1

Processing Scenarios:

Setup I Scenario I

Table 6 presents the results from the Setup I Scenario I, where we benchmarked the microservice utilizing the burst at startup technique. Regarding the results, changing the number of registered identities did not affect the processing-time latency associated with the verification of a single message. An average delay of approximately 39 ms was measured. The minimum delay was 31.7 (±1.30) ms. In contrast, the average deviations for quantiles of the order p90 and p99 do not consecutively exceed 2 ms and 3 ms. For 100 000 registered identities, quantiles of p90 latencies were 44.7 (±0.56) ms, and for p99 - 55.8 (±1.64) ms.

Table 6. Setup I Scenario I - Processing-time latency metrics (in milliseconds).

Identity Count	Avg	Avg Dev	Min	Max	Pop Std Dev	Percentiles [p=0.9; p=0.95; p=0.99]
10000	38.3 (0.76)	3.0 (0.00)	32.7 (0.56)	133.8 (7.80)	5.5 (0.70)	[44.3 (0.76); 47.7 (1.24); 56.7 (2.16)]
20000	39.5 (1.20)	4.0 (0.00)	32.2 (1.28)	143.7 (9.90)	6.6 (0.60)	[46.7 (1.90); 50.7 (1.70); 61.4 (2.00)]
35000	39.9 (1.12)	3.7 (0.42)	32.6 (1.08)	131.2 (5.64)	5.9 (0.54)	[46.5 (1.10); 50.5 (1.30); 60.2 (2.16)]
50000	38.2 (1.28)	3.3 (0.42)	31.7 (1.30)	130.8 (7.76)	5.5 (0.60)	[44.6 (1.72); 47.8 (2.00); 55.7 (2.70)]
100000	38.6 (0.72)	3.3 (0.42)	32.0 (0.40)	139.3 (5.16)	5.5 (0.50)	[44.7 (0.56); 47.8 (0.64); 55.8 (1.64)]

Setup I Scenario II

Table 7 illustrates the average time required for consumers to read the data stream while the SUT was simultaneously processing it. Importantly, every second message referenced an identity not registered in the DLT. This method was designed to minimize the influence of optimization mechanisms (caching). The use of the off-chain data store reduced the reading time of the data streams by nearly half, with an average delay of approximately 21 ms recorded.

Table 7. Time of data stream (1000 messages) retrieval by consumer (in seconds).

	Identity Count				
	10000	20000	35000	50000	100000
Scenario I	38.86 (0.64)	39.95 (1.39)	40.19 (0.98)	38.59 (1.22)	39.15 (0.61)
Scenario II	20.42 (0.77)	21.03 (1.20)	21.15 (1.06)	20.39 (1.12)	20.06 (0.69)

Setup II Scenario I

The upcoming tables details the metrics for the microservice during the Hot Peer Phase and Java Steady State. Tables 8 (Java) and 9 (Go) provide results for the RPi5 platform, while Tables 10 (Java) and 11 (Go) focus on the RPi3. Our analysis revealed that the full processing latency for the Java-based microservice on the RPi5 was 7.38 (±0.18) ms, approximately 1.62 ms longer than that on the Standalone Laptop. Remarkably, 98.7% of this processing time was associated with the DL Query operation. Additionally, the microservice on the RPi3 demonstrated a latency of 21.14 (±0.28) ms, nearly three times greater than that of the RPi5.

Our comparison of metrics from Go-based microservice execution revealed that the RPi5 consistently surpassed the performance of both the SLap and the RPi3 platforms. The average time to verify a single message was 6.18 (±0.29) ms, with a minimum delay of 4.14 (±0.17) ms. The recorded p90 and p95 latencies were 8.30 (±0.44) ms and 9.32 (±0.56) ms, respectively. Notably, 68.6% of the processing time was attributed to the DL Query, while 31.2% was related to the Message Forwarding operation.

 doi:10.20944/preprints202504.0237.v1

Table 8. Setup II (RPi5) Scenario I - Processing-time latency metrics (in ms) for Java code.

Operation	Avg	Avg Dev	Min	Max	Pop Std Dev	Percentiles [p=0.9; p=0.95; p=0.99]
Crypt. Prim. Init	0.007 (0.0004)	0.003 (0.0003)	0.002 (0.0003)	2.207 (1.5388)	0.024 (0.0156)	[0.010 (0.0005); 0.014 (0.0004); 0.027 (0.0005)]
DL Query	7.20 (0.18)	1.10 (0.19)	5.29 (0.06)	150.7 (185.65)	3.21 (2.66)	[8.80 (0.33); 9.99 (0.43); 13.83 (0.76)]
HMAC Validation	0.015 (0.0004)	0.006 (0.0003)	0.006 (0.0010)	0.533 (0.2455)	0.013 (0.0017)	[0.022 (0.0004); 0.03 (0.0007); 0.062 (0.0016)]
AES Decryption	0.057 (0.0017)	0.027 (0.0005)	0.024 (0.0037)	3.053 (1.3175)	0.061 (0.0100)	[0.098 (0.0034); 0.136 (0.0021); 0.244 (0.0041)]
Msg Forwarding	0.082 (0.0021)	0.044 (0.0021)	0.027 (0.0038)	24.701 (4.5436)	0.262 (0.0435)	[0.144 (0.0058); 0.193 (0.0046); 0.354 (0.0247)]
Full Processing	7.38 (0.18)	1.16 (0.18)	5.38 (0.05)	157.56 (184.31)	3.31 (2.64)	[9.08 (0.32); 10.34 (0.41); 14.42 (0.74)]
Chaincode Exec (logs)	1.19 (0.09)	0.34 (0.14)	0.00 (0.00)	23.00 (10.00)	0.77 (0.19)	[1.50 (0.60); 2.30 (0.48); 3.70 (0.84)]
gRPC Call (logs)	2.55 (0.11)	0.47 (0.11)	1.57 (0.09)	27.68 (11.60)	0.91 (0.16)	[3.28 (0.4); 3.89 (0.52); 5.59 (0.68)]

Table 9. Setup II (RPi5) Scenario I - Processing-time latency metrics (in ms) for Go code.

Operation	Avg	Avg Dev	Min	Max	Pop Std Dev	Percentiles [p=0.9; p=0.95; p=0.99]
Crypt. Prim. Init	0.004 (0.0002)	0.002 (0.0003)	0.002 (0.0001)	0.150 (0.1292)	0.006 (0.0038)	[0.006 (0.0001); 0.007 (0.0003); 0.013 (0.002)]
DL Query	4.24 (0.27)	1.26 (0.16)	2.50 (0.07)	27.63 (10.54)	1.84 (0.28)	[6.25 (0.39); 7.13 (0.52); 10.12 (1.03)]
HMAC Validation	0.003 (0.0002)	0.001 (0.0004)	0.002 (0.0001)	0.225 (0.2114)	0.008 (0.0063)	[0.004 (0.0001); 0.005 (0.0003); 0.011 (0.0019)]
AES Decryption	0.005 (0.0006)	0.002 (0.0010)	0.003 (0.0001)	0.376 (0.3774)	0.015 (0.0131)	[0.006 (0.0002); 0.008 (0.0006); 0.027 (0.0086)]
Msg Forwarding	1.926 (0.0622)	0.252 (0.0387)	1.463 (0.0193)	15.582 (3.0110)	0.678 (0.1895)	[2.221 (0.0706); 2.391 (0.1051); 3.691 (0.8053)]
Full Processing	6.18 (0.29)	1.33 (0.16)	4.14 (0.11)	26.90 (4.30)	1.91 (0.24)	[8.30 (0.44); 9.32 (0.56); 13.00 (1.46)]
Chaincode Exec (logs)	1.79 (0.21)	1.05 (0.16)	0.00 (0.00)	19.00 (11.80)	1.48 (0.31)	[3.50 (0.50); 4.40 (0.48); 6.60 (0.72)]
gRPC Call (logs)	3.24 (0.24)	1.13 (0.16)	1.73 (0.05)	23.68 (11.83)	1.62 (0.3)	[5.13 (0.31); 5.87 (0.48); 8.24 (0.67)]

Table 10. Setup II (RPi3) Scenario I - Processing-time latency metrics (in ms) for Java code.

Operation	Avg	Avg Dev	Min	Max	Pop Std Dev	Percentiles [p=0.9; p=0.95; p=0.99]
Crypt. Prim. Init	0.039 (0.0015)	0.013 (0.0017)	0.015 (0.0013)	6.523 (1.581)	0.071 (0.017)	[0.052 (0.0016); 0.067 (0.0018); 0.108 (0.0022)]
DL Query	20.28 (0.29)	3.70 (0.29)	14.50 (0.13)	541.19 (71.92)	11.28 (2.42)	[25.49 (0.36); 29.99 (0.50); 42.24 (1.51)]
HMAC Validation	0.078 (0.0012)	0.019 (0.0014)	0.046 (0.0063)	4.260 (3.4290)	0.063 (0.0288)	[0.097 (0.0019); 0.133 (0.0031); 0.236 (0.0035)]
AES Decryption	0.280 (0.0061)	0.109 (0.0068)	0.148 (0.0100)	21.179 (10.3614)	0.321 (0.0980)	[0.457 (0.0214); 0.586 (0.0152); 0.952 (0.0144)]
Msg Forwarding	0.389 (0.0103)	0.182 (0.0056)	0.185 (0.0157)	91.299 (9.5987)	0.968 (0.0928)	[0.647 (0.0172); 0.866 (0.0355); 1.636 (0.1188)]
Full Processing	21.14 (0.28)	3.97 (0.28)	15.02 (0.14)	542.44 (71.97)	11.56 (2.37)	[26.9 (0.33); 31.54 (0.45); 44.62 (1.55)]
Chaincode Exec (logs)	1.14 (0.04)	0.25 (0.07)	0.00 (0.00)	19.7 (3.04)	0.65 (0.21)	[1.40 (0.48); 2.00 (0.00); 2.90 (0.72)]
gRPC Call (logs)	2.58 (0.05)	0.36 (0.05)	1.37 (0.17)	28.2 (2.52)	0.77 (0.18)	[3.06 (0.07); 3.39 (0.15); 4.70 (0.58)]

Table 11. Setup II (RPi3) Scenario I - Processing-time latency metrics (in ms) for Go code.

Operation	Avg	Avg Dev	Min	Max	Pop Std Dev	Percentiles [p=0.9; p=0.95; p=0.99]
Crypt. Prim. Init	0.030 (0.0005)	0.008 (0.0007)	0.021 (0.0002)	0.325 (0.1910)	0.017 (0.0058)	[0.043 (0.0013); 0.054 (0.0014); 0.085 (0.0072)]
DL Query	5.10 (0.18)	0.76 (0.15)	3.72 (0.06)	23.96 (9.70)	1.40 (0.40)	[6.14 (0.31); 7.08 (0.54); 10.49 (2.01)]
HMAC Validation	0.030 (0.0009)	0.008 (0.0012)	0.023 (0.0001)	0.550 (0.2487)	0.023 (0.0071)	[0.042 (0.0013); 0.052 (0.0035); 0.086 (0.0073)]
AES Decryption	0.062 (0.0018)	0.014 (0.0026)	0.048 (0.0003)	1.493 (1.4497)	0.053 (0.0436)	[0.080 (0.0016); 0.091 (0.0023); 0.126 (0.006)]
Msg Forwarding	2.591 (0.1124)	0.303 (0.0225)	1.947 (0.0196)	16.971 (4.1936)	0.703 (0.1376)	[2.988 (0.1207); 3.204 (0.1204); 4.407 (0.3333)]
Full Processing	7.86 (0.21)	0.94 (0.14)	6.04 (0.07)	29.8 (8.75)	1.65 (0.37)	[9.22 (0.33); 10.21 (0.62); 14.18 (2.01)]
Chaincode Exec (logs)	1.20 (0.08)	0.33 (0.11)	0.00 (0.00)	14.09 (6.84)	0.70 (0.19)	[1.64 (0.46); 2.18 (0.30); 3.45 (0.68)]
gRPC Call (logs)	2.67 (0.15)	0.44 (0.09)	1.71 (0.07)	16.39 (6.57)	0.84 (0.22)	[3.32 (0.25); 3.87 (0.34); 5.61 (0.81)]

Figure 15 presents the comparison of the Message Forwarding operation latencies on different platforms under test. During the Java Warm-Up State, we recorded the following operation times: SLap – 0.09 (±0.004) ms, RPi5 – 0.24 (±0.008) ms, RPi5 with CC enabled – 0.24 (±0.009) ms, and RPi3 – 0.99 (±0.031) ms. In the Java Steady State, the times were 0.03 (±0.001) ms, 0.08 (±0.002) ms, 0.09 (±0.002) ms, and 0.39 (±0.010) ms, respectively. For the Go-based microservice, we measured: SLap – 2.25 (±0.017) ms, RPi5 – 1.97 (±0.016) ms, RPi5 with CC enabled – 2.12 (±0.016) ms, and RPi3 – 2.70 (±0.020) ms. Notably, the Ledger Peer state (Hot, Cold) did not influence the latency of the examined operation. The variations in latencies are largely attributable to the handling of synchronous operations and the wait time associated with acknowledgement responses. In contrast, we found that the metric latency for the Go implementation using the non-blocking asynchronous producer (*sarama.AsyncProducer*) was significantly lower: SLap – 0.007 (±0.001) ms, RPi5 – 0.002 (±0.001) ms, RPi5 with CC enabled – 0.002 (±0.001) ms, and RPi3 – 0.032 (±0.003) ms. Furthermore, it is noteworthy that for the Go implementation, the latencies for the RPi5 were, on average, about 0.3 ms lower than those recorded for the SLap platform.

 doi:10.20944/preprints202504.0237.v1

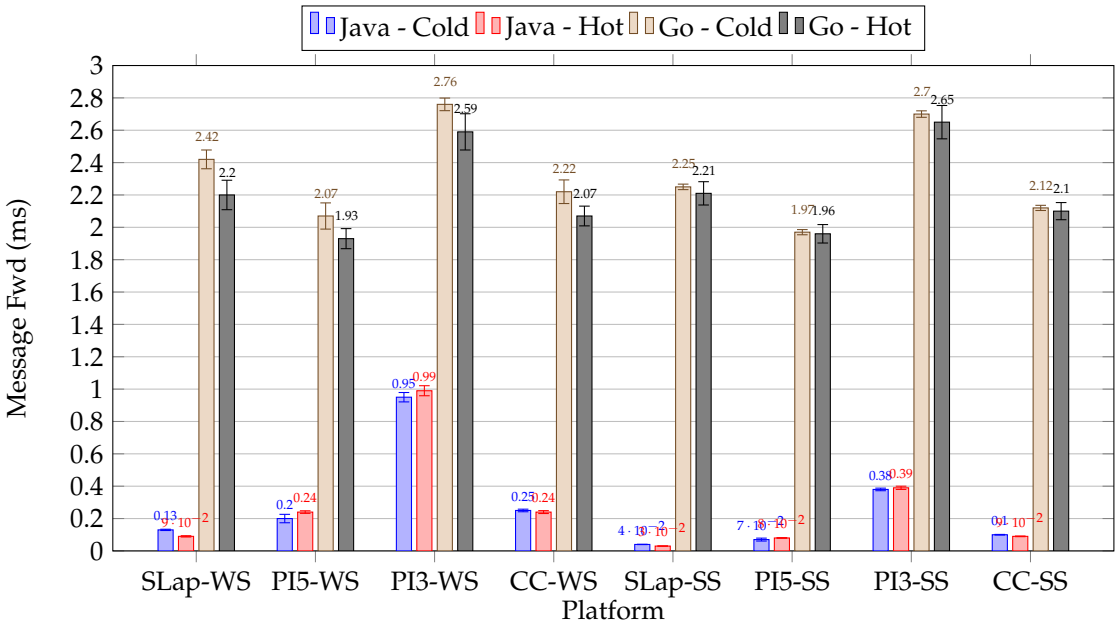

Figure 15. Message Forwarding latencies on different platforms under test.

Setup II Scenario II

Tables 12 (Java) and 13 (Go) outline the results from Setup II Scenario II, in which the Ledger Peer was relocated to the Fog Processing component. The benchmarking within this scenario concentrated on comparing latencies for the RPi5 platform with the Continuum Computing concept enabled (CC). In this configuration, both the Ledger Peer and the microservice were running on the same device platform – Raspberry Pi 5.

Table 12. Setup II (RPi5) Scenario II - Processing-time latency metrics (in ms) for Java code.

Operation	Avg	Avg Dev	Min	Max	Pop Std Dev	Percentiles [p=0.9; p=0.95; p=0.99]
DL Query	7.13 (0.19)	1.10 (0.28)	5.77 (0.02)	138.04 (153.11)	1.40 (0.84)	[8.30 (0.05); 9.33 (0.09); 13.04 (0.20)]
Chaincode Exec (logs)	0.79 (0.02)	0.41 (0.01)	0.00 (0.00)	8.40 (1.28)	0.57 (0.01)	[1.00 (0.00); 1.00 (0.00); 2.20 (0.32)]
gRPC Call (logs)	2.67 (0.02)	0.32 (0.02)	2.04 (0.1)	91.02 (82.52)	1.2 (0.8)	[3.11 (0.03); 3.43 (0.06); 4.85 (0.11)]

Table 13. Setup II (RPi5) Scenario II - Processing-time latency metrics (in ms) for Go code.

Operation	Avg	Avg Dev	Min	Max	Pop Std Dev	Percentiles [p=0.9; p=0.95; p=0.99]
DL Query	3.23 (0.24)	0.22 (0.06)	2.83 (0.21)	7.15 (1.05)	0.35 (0.06)	[3.59 (0.15); 3.79 (0.15); 4.51 (0.25)]
Chaincode Exec (logs)	0.53 (0.26)	0.35 (0.08)	0.00 (0.00)	2.90 (0.72)	0.43 (0.05)	[1.00 (0.00); 1.00 (0.00); 1.10 (0.18)]
gRPC Call (logs)	2.60 (0.20)	0.17 (0.05)	2.27 (0.18)	6.09 (0.71)	0.27 (0.05)	[2.90 (0.13); 3.04 (0.12); 3.51 (0.18)]

The results (Figures 16 and 17) showed closely aligned metrics for the Ledger Peer during HP phase and both the WS and SS states, with latencies averaging approximately 11 ms for WS and 7.3 ms for SS. Furthermore, the DL Query operation latency for RPi5 during WS was recorded at 14.55 (±0.95) ms, while for the RPi5 with CC enabled was 18.62 (±3.66) ms. Lastly, we observed a significant error of 1.65 ms for WS phase and HP state, associated with the DL query. Figure 18 presents the results obtained for Go-based microservice tested on different platforms and with both Ledger Peer states.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 April 2025 doi:10.20944/preprints202504.0237.v1

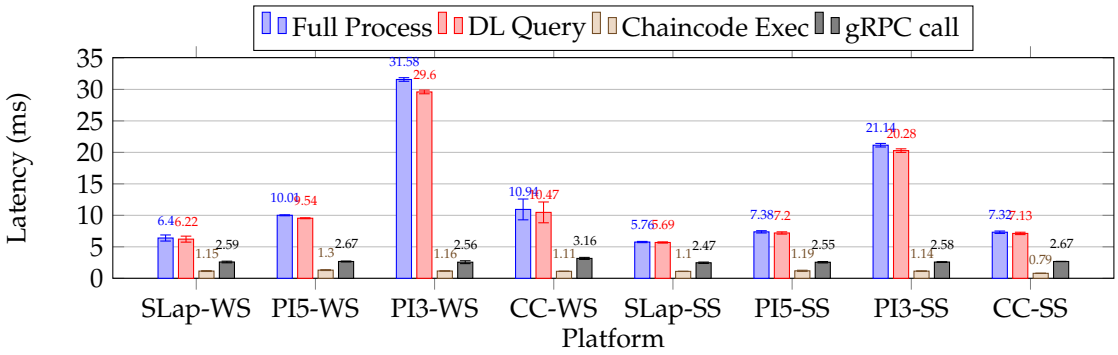

Figure 16. Java microservice latencies on different platforms under test - Ledger Hot Peer phase.

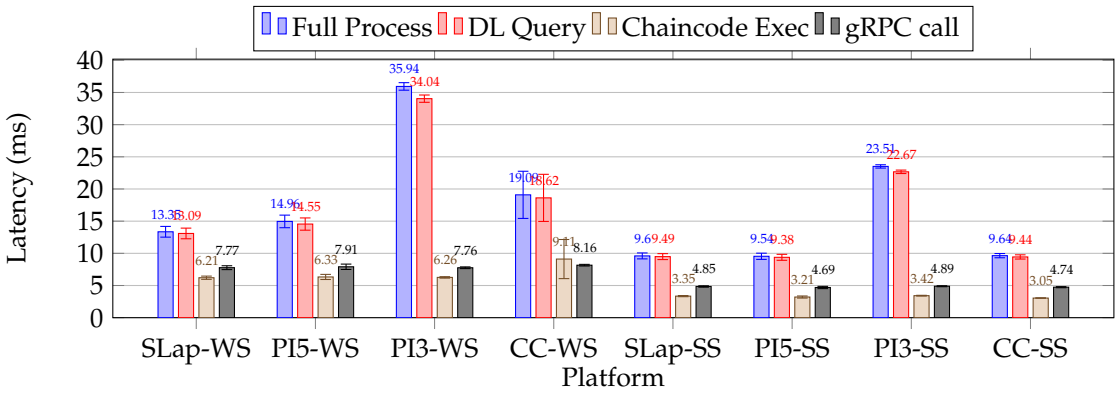

Figure 17. Java microservice latencies on different platforms under test - Ledger Cold Peer phase.

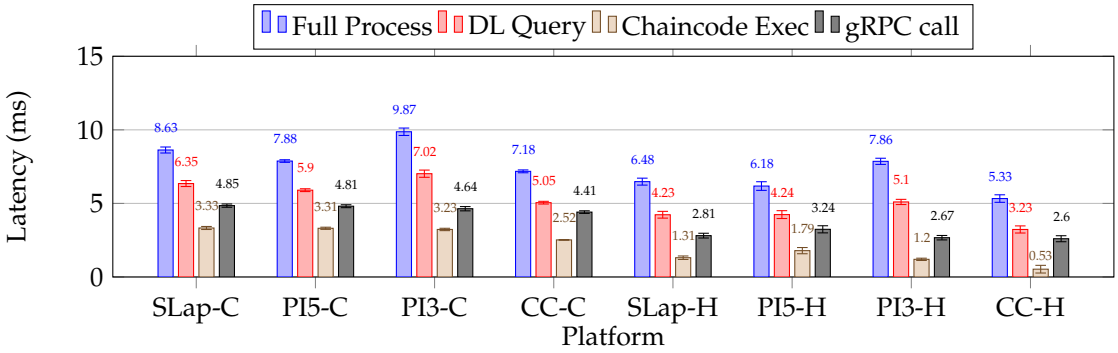

Figure 18. Go microservice latencies on different platforms under test - HP and CP Ledger Peer states.

Resource Utilization:

In Section 5.7, we presented reference results from our experiments evaluating the performance (resource utilization) of the Ledger Peer, Kafka Cluster, and microservice execution on the SLap platform. We applied similar evaluation criteria for the measurements collected from the RPi5 and RPi3 platforms. Notably, we did not observed significant increases in resource utilization throughout our assessment. There were no instances of CPU reductions in instructions per clock, or throttling events observed. Additionally, we recorded a stable temperature of about 52°C on both devices. Moreover, the ratio of CPU usage to I/O wait time for the Kafka Cluster did not exceed 10%, and disk utilization (queuing delays) did not substantially affect the time required for record publish operations. Lastly, our network traffic monitoring revealed that the microservice producer's limit on the number of unacknowledged requests sent to the leader broker remained within the configured upper limit of five, thereby ensuring that latency associated with the Message Forwarding operation was not impacted.

Power Consumption:

 doi:10.20944/preprints202504.0237.v1

Tables 14 and 15 provide an analysis of the power consumption for the RPi5 platform during microservice execution. The findings indicate that the Java microservice (SM) exhibits a higher power consumption (PC) compared to its Go counterpart. Specifically, we recorded an average PC of 3.11 (±0.63) W for Java and 2.27 (±0.29) W for Go. When comparing these results to the reference PC, we observed an increase of approximately 1.00 W for Java and 0.15 W for Go. With the CC concept enabled (SM + CC), the power consumption values remained stable at 3.88 (±0.58) W for Java and 3.17 (±0.59) W for Go. During the Java microservice execution, the Ledger Peer consumed approximately 0.77 W, while the consumption for Go was about 0.87 W. The average PC for the Kafka cluster during both microservice executions did not exceed 3.4 W. Finally, no voltage spikes that could potentially impact CPU IPC were detected. Only drops associated with logging computation statistics were identified.

Table 14. Cluster and Peer - Java.

Device name	CPU idle (%)	RAM usage (%)	PC (W)
SM	85.2	16.0	3.11 (0.63)
SM + CC	60.1	31.6	3.88 (0.58)
B1	86.0	17.7	3.34 (0.68)
B2	83.3	17.8	3.32 (0.68)
B3	91.4	17.7	3.40 (0.66)

Table 15. Cluster and Peer - Go.

Device name	CPU idle (%)	RAM usage (%)	PC (W)
SM	96.1	11.0	2.27 (0.29)
SM + CC	71.2	27.4	3.17 (0.59)
B1	86.4	17.9	3.29 (0.71)
B2	85.9	18.1	3.13 (0.66)
B3	92.4	18.0	3.21 (0.57)

6. Discussion

Deploying our framework within AWS Cloud infrastructure revealed its suitable for processing audiovisual streams within environments that demand immediate interoperability. The results shown in Table 6 for the Setup I Scenario I are promising as the average time for consumer to read verified data stream was approximately 39 seconds. In the context of audiovisual streams, the measured time ($1000msg/39s$) represents about 25 frames per second (fps). Notably, the work of [42] demonstrated that CCTV cameras with a minimum 8 fps frame rate are required to correctly identify objects on video. By evaluating the average deviations (Table 6), we can confidently affirm the computational stability of our framework. This stability is essential for upcoming performance studies, including those focused on maximum throughput. In addition, the results for the Setup I Scenario II confirmed the rightness of local off-chain data store applicability as a mirco-caching mechanism for streams verifying. The usage of mentioned store almost doubly reduced the reading time of data streams (Table 7).

Furthermore, we have confirmed that our environment can be deployed on resource-constrained COTS platforms while maintaining low operational costs. The latency metrics from the Setup II Scenario I (Tables 8 and10) for key internal operations demonstrate the computational stability of the framework. Additionally, the metrics for Chaincode Execution and gRPC call indicated that benchmarking was conducted under stable Ledger Peer conditions. Our analysis found that the overall processing latency for the Java-based microservice on the RPi5 was 7.38 (±0.18) ms, which is approximately 1.62 ms longer than that on the SLap platform. Notably, 98.7% of this processing time was associated with the DL Query operation. The RPi3 microservice execution exhibited a nearly three times greater latency than the RPi5. For Go-based microservice execution (Tables 9 and 11), we observed that the RPi5 consistently outperformed both the SLap and the RPi3. The average time required to verify a single message on the RPi5 was 6.18 (±0.29) ms. And, 68.6% was attributed to the DL Query, while 31.2% was related to the Message Forwarding operation.

 doi:10.20944/preprints202504.0237.v1

Notably, we observed (Figure 15) that when uniformly applying the configuration for the Message Forwarding operation, the Kafka Streams API outperforms the Sarama library. The discrepancies in latencies for stream forwarding can largely be attributed to the handling of synchronous operations and the wait time for acknowledgment responses. In contrast, the use of asynchronous data flow adversely affects reliability, which is not in alignment with the key performance indicators we have established.

Moreover, we analyzed metrics from our Go-based microservice. We found that, for both Cold and Hot phases with the Computing Continuum concept enabled, the DL Query operation exhibited the lowest latencies among all platforms, averaging 5.05 (±0.09) ms for CP and 3.23 (±0.24) ms for HP. Moreover, the latency for Chaincode Execution was more than twice as low on the CC platform compared to others. In contrast, latencies for gRPC calls remained largely consistent across the various platforms.

During our experiments, we confirmed that the phase of the Ledger Peer – whether it is Cold or Hot, combined with JVM dynamic compilation and garbage collection, results in fluctuations in internal operation latencies during the microservice Warm-Up State. We especially observed significant instability (deviations) in both Chaincode Execution and gRPC calls during testing that involved the Warm-Up State and the Cold Peer phase. We noted particularly concerning latency of Chaincode Execution, which averaged 9.11 (±3.04) ms – this was higher than the gRPC call average of 8.16 (±0.13) ms. This anomaly in the metrics can be attributed to duration received from log entries where the execution time for the chaincode is rounded to whole milliseconds.

To mitigate this, we can estimate the duration of the Warm-Up State statically using tools such as the Java Microbenchmark Harness. However, one drawback of this approach is its dependence on the researcher's coding experience, which can lead to inaccurate estimates. Alternatively, dynamic estimation methods can be employed, such as a sliding window to compare the most recent iterations [37]. This approach can effectively reduce the duration of performance testing while maintaining an acceptable accuracy level.

As we noted earlier, several factors can influence our environment's metrics, including the cluster's configuration, resource usage, the settings for producing/consuming applications, and the communication medium used. Through detailed monitoring of resource utilization and power consumption with the Prometheus environment integrated into our setup, we identified issues such as CPU IPC decrease, queuing delays, and PC spikes. Fortunately, none of these events occurred during our experiments.

Lastly, upon comparing our results with those detailed in Bekaroo et al. [43], we found that the PC for the RPi5 under test during the Idle state was nearly identical (2.20W). Furthermore, the results for the CC concept associated with the Java microservice can be likened to activities that necessitate the use of a persistent network connection and graphic rendering (3.80W).

7. Conclusions and Future Work

One of the ongoing challenges is acquiring, analyzing, and disseminating the considerable volumes of data generated by various entities, especially IoT devices, while ensuring that distribution is secure and reliable. Furthermore, to effectively manage trade-offs in a data dissemination environment while considering key performance indicators, it is imperative to implement thorough monitoring for potential adjustments.

To address the identified challenges, we proposed and evaluated our experimental framework architecture, which is designed for secure and fault-tolerant distribution of data streams within the multi-organizational federation environment. Our research primarily emphasized data durability, fault tolerance, and low-latency KPIs. We also considered Kingman's formula, which indicates that increasing the number of producers or consumers can exponentially increase the load on brokers as they accommodate more connections (queue burden). This can result in request bursts and long-tail latencies, underscoring the necessity of effectively scaling the Data Queue Layer.

 doi:10.20944/preprints202504.0237.v1

After thoroughly comparing metrics for the SLap, RPi5, and RPi3 platforms, we have determined that adopting horizontal scaling will be more advantageous for our environment. This strategy will allow us to distribute workloads across multiple devices, thereby enhancing resilience and adaptability to fluctuating demands of the IoT environments. On the one hand, by increasing the pool of microservices, we can improve the latency-throughput trade-off. On the other hand, we proposed deploying the Fingerprint Enrichment Layer in conjunction with the Protocol Forwarder (proxy) component. This component can reduce network bandwidth consumption while simultaneously alleviating the load on the cluster (the Data Queue Layer).

Moreover, this article identified and measured the negative impact of Java's Just-in-Time compilation mechanism on the stream processing microservice latencies. Fortunately, techniques exist to minimize Warm-Up State by employing suitable JVM flags, selecting the appropriate compilation tier, applying Ahead-of-Time Compilation, coordinating restoration at checkpoints, or utilizing ahead-of-time fake traffic mirroring. In our upcoming research, we intend to implement the latter technique and apply dynamic estimation [37].

Additionally, we validated the effectiveness of the Computing Continuum concept within our framework. The Ledger Peer arrangement utilized a mechanism similar to a local off-chain store, enhancing efficiency by positioning the peer and the microservice in close proximity within the Fog component. An intriguing avenue for future research is integrating the Kafka Streams API off-chain store handling mechanism with the on-chain peer, which could prove advantageous for DIL network deployments.

Furthermore, our research will thoroughly evaluate security and reliability risk assessments, considering a diverse range of threats across the Application, Network, and Perception layers of the IoT system. We aim to establish a formal multi-level security model that effectively addresses data confidentiality and integrity protection requirements. In addition, we will conduct experiments aimed at maximizing throughput by applying various configurations to the components defining each layer of our environment. We will also undertake more in-depth studies focusing on the Value of Information, Data Quality, and Context Dissemination. Additionally, we plan to deploy lightweight and delay-tolerant consensus algorithms within the Distributed Ledger Layer, which employs a directed-acyclic graph structure for achieving consensus.

Although the proposed implementation of the Streams Microservice Layer enables handling single record batches (individual messages in the data stream), our pluggable architecture supports integration with resource orchestration and automation platforms, allowing for the microservice dynamic allocation based on varying KPIs. Moreover, our framework offers significant potential as a foundation for various applications. For instance, it could be incorporated into systems designed to detect and neutralize UAVs. Such integration would empower both civilian and military organizations to maintain comprehensive oversight of air defense operations, enabling IoT devices within smart city and army infrastructures to relay data regarding UAV locations securely via our solution. Another possible application involves establishing an ad-hoc system for coordinating international efforts focused on HADR operations. Our system would facilitate reliable data dissemination from CCTV cameras and health devices, such as SOS wristbands, in this scenario. This capability would significantly reduce response times for individuals needing assistance and enhance decision-making through improved information sharing.

Author Contributions: Conceptualization, J.S. and Z.Z.; methodology, Z.Z.; software, J.S.; validation, J.S.; investigation, J.S. and Z.Z.; resources, Z.Z.; data curation, J.S.; writing—original draft preparation, J.S. and Z.Z.; writing—review and editing, J.S. and Z.Z.; visualization, J.S.; supervision, Z.Z.; project administration, Z.Z.; funding acquisition, Z.Z. All authors have read and agreed to the published version of the manuscript.

Data Availability Statement: Data is contained within the article.

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

 doi:10.20944/preprints202504.0237.v1

Abbreviations

The following abbreviations are used in this manuscript:

ABAC	Attribute-Based Access Control
ACL	Access Control List
AES	Advanced Encryption Standard
AWS	Amazon Web Services
AoR	Area of Responsibility
CC	Compute Continuum concept
CCTV	City Surveillance Camera
COTS	Commercial Off-The-Shelf
CP	Cold Ledger Peer Phase
DIL	Disconnected, Intermittent, Limited
DLT	Distributed Ledger Technology
FMN	Federated Mission Networking
gRPC	Remote Procedure Calls
GUID	Globally Unique Identifier
HADR	Humanitarian Assistance And Disaster Relief
HMAC	Keyed-Hash Message Authentication Code
HP	Hot Ledger Peer Phase
IOs	Information Objects
IPC	CPU Instructions per Clock
IoT	Internet of Things
JIT	Just-In-Time
JVM	Java Virtual Machine
KPIs	Key Performance Indicators
MQTT	MQ Telemetry Transport
NATO	North Atlantic Treaty Organization
PC	Power Consumption
RPi3	Raspberry Pi 3
RPi5	Raspberry Pi 5
SLap	Standalone Laptop
SS	Java JIT Steady State
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
VoI	Value of Information
WS	Java JIT Warm-Up State

References

1. Johnsen, F.T.; Hauge, M. Interoperable, adaptable, information exchange in NATO coalition operations. *Journal of Military Studies* **2022**, *11*, 49 – 62. https://doi.org/10.2478/jms-2022-0005.
2. Kopetz, H.; Steiner, W. Real-Time Systems: Design Principles for Distributed Embedded Applications. *Real-Time Systems* **2022**. https://doi.org/10.1007/978-3-031-11992-7_13.
3. Zaccarini, M.; Cantelli, B.; Fazio, M.; Fornaciari, W.; Poltronieri, F.; Stefanelli, C.; Tortonesi, M. VOICE: Value-of-Information for Compute Continuum Ecosystems. In Proceedings of the 2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN), 2024, pp. 73–80. https://doi.org/10.1109/ICIN60470.2024.10494500.
4. Pióro, L.; Sychowiec, J.; Kanciak, K.; Zieliński, Z. Application of Attribute-Based Encryption in Military Internet of Things Environment. *Sensors (Basel, Switzerland)* **2024**, *24*. https://doi.org/10.3390/s24185863.
5. Hyperledger Fabric documentation. https://hyperledger-fabric.readthedocs.io/. Accessed: 2025-03-03.

 doi:10.20944/preprints202504.0237.v1

6. Apache Kafka Streams API documentation. https://kafka.apache.org/documentation/streams/. Accessed: 2025-03-03.
7. Sarama Go library. https://pkg.go.dev/github.com/shopify/sarama. Accessed: 2025-03-03.
8. Wang, X.; Zha, X.; Ni, W.; Liu, R.P.; Guo, Y.J.; Niu, X.; Zheng, K. Survey on blockchain for Internet of Things. *Computer Communications* **2019**, *136*, 10–29. https://doi.org/https://doi.org/10.1016/j.comcom.2019.01.006.
9. Kumar, R.; Khan, F.; Kadry, S.N.; Rho, S. A Survey on blockchain for industrial Internet of Things. *Alexandria Engineering Journal* **2021**. https://doi.org/10.1016/j.aej.2021.11.023.
10. Alfandi, O.; Khanji, S.I.R.; Ahmad, L.; Khattak, A.M. A survey on boosting IoT security and privacy through blockchain. *Cluster Computing* **2020**, *24*, 37 – 55. https://doi.org/10.1007/s10586-020-03137-8.
11. Guo, S.; Wang, F.; Zhang, N.; Qi, F.; song Qiu, X. Master-slave chain based trusted cross-domain authentication mechanism in IoT. *J. Netw. Comput. Appl.* **2020**, *172*, 102812. https://doi.org/10.1016/j.jnca.2020.102812.
12. Xu, L.; Chen, L.; Gao, Z.; Fan, X.; Suh, T.; Shi, W.L. DIoTA: Decentralized-Ledger-Based Framework for Data Authenticity Protection in IoT Systems. *IEEE Network* **2020**, *34*, 38–46. https://doi.org/10.1109/MNET.001.1900136.
13. Khalid, U.; Asim, M.; Baker, T.; Hung, P.C.K.; Tariq, M.A.; Rafferty, L. A decentralized lightweight blockchain-based authentication mechanism for IoT systems. *Cluster Computing* **2020**, *23*, 2067 – 2087. https://doi.org/10.1007/s10586-020-03058-6.
14. Chung.; Ferraiolo, D.; Kuhn, D.; Schnitzer, A.; Sandlin, K.; Miller, R.; Scarfone, K. Guide to Attribute Based Access Control (ABAC) Definition and Considerations, 2019. https://doi.org/https://doi.org/10.6028/NIST.SP.800-162.
15. Song, H.; Tu, Z.; Qin, Y. Blockchain-Based Access Control and Behavior Regulation System for IoT. *Sensors* **2022**, *22*. https://doi.org/10.3390/s22218339.
16. Lu, Y.; Feng, T.; Liu, C.; Zhang, W. A Blockchain and CP-ABE Based Access Control Scheme with Fine-Grained Revocation of Attributes in Cloud Health. *Computers, Materials & Continua* **2024**, *78*, 2787–2811. https://doi.org/10.32604/cmc.2023.046106.
17. Sivanathan, A.; Gharakheili, H.H.; Loi, F.; Radford, A.; Wijenayake, C.; Vishwanath, A.; Sivaraman, V. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing* **2019**, *18*, 1745–1759. https://doi.org/10.1109/TMC.2018.2866249.
18. Xu, Q.; Zheng, R.; Saad, W.; Han, Z. Device Fingerprinting in Wireless Networks: Challenges and Opportunities. *IEEE Communications Surveys & Tutorials* **2015**, *18*, 94–104. https://doi.org/10.1109/COMST.2015.2476338.
19. Jagannath, A.; Jagannath, J.; Kumar, P.S.P.V. A Comprehensive Survey on Radio Frequency (RF) Fingerprinting: Traditional Approaches, Deep Learning, and Open Challenges. *Comput. Networks* **2022**, *219*. https://doi.org/10.36227/techrxiv.17711444.
20. Jarosz, M.; Wrona, K.; Zieliński, Z. Distributed Ledger-Based Authentication and Authorization of IoT Devices in Federated Environments. *Electronics* **2024**, *13*. https://doi.org/10.3390/electronics13193932.
21. Sanogo, L.; Alata, E.; Takacs, A.; Dragomirescu, D. Intrusion Detection System for IoT: Analysis of PSD Robustness. *Sensors (Basel, Switzerland)* **2023**, *23*. https://doi.org/10.3390/s23042353.
22. Chatterjee, B.; Das, D.; Maity, S.; Sen, S. RF-PUF: Enhancing IoT Security Through Authentication of Wireless Nodes Using In-Situ Machine Learning. *IEEE Internet of Things Journal* **2018**, *6*, 388–398. https://doi.org/10.1109/JIOT.2018.2849324.
23. Charyyev, B.; Gunes, M.H. IoT Traffic Flow Identification using Locality Sensitive Hashes. *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)* **2020**, pp. 1–6. https://doi.org/10.1109/ICC40277.2020.9148743.
24. Neumann, C.; Heen, O.; Onno, S. An Empirical Study of Passive 802.11 Device Fingerprinting. *2012 32nd International Conference on Distributed Computing Systems Workshops* **2012**, pp. 593–602. https://doi.org/10.1109/ICDCSW.2012.8.
25. Jansen, N.; Manso, M.; Toth, A.; Chan, K.S.; Bloebaum, T.H.; Johnsen, F.T. NATO Core Services profiling for Hybrid Tactical Networks — Results and Recommendations. *2021 International Conference on Military Communication and Information Systems (ICMCIS)* **2021**, pp. 1–8. https://doi.org/10.1109/ICMCIS52405.2021.9486415.
26. Suri, N.; Fronteddu, R.; Cramer, E.; Breedy, M.R.; Marcus, K.M.; in 't Velt, R.; Nilsson, J.; Mantovani, M.; Campioni, L.; Poltronieri, F.; et al. Experimental Evaluation of Group Communications Protocols for Tactical Data Dissemination. *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)* **2018**, pp. 133–139. https://doi.org/10.1109/MILCOM.2018.8599749.

 doi:10.20944/preprints202504.0237.v1

27. Rango, F.D.; Potrino, G.; Tropea, M.; Fazio, P. Energy-aware dynamic Internet of Things security system based on Elliptic Curve Cryptography and Message Queue Telemetry Transport protocol for mitigating Replay attacks. *Pervasive Mob. Comput.* **2020**, *61*. https://doi.org/10.1016/j.pmcj.2019.101105.
28. Yang, M.; Margheri, A.; Hu, R.; Sassone, V. Differentially Private Data Sharing in a Cloud Federation with Blockchain. *IEEE Cloud Computing* **2018**, *5*, 69–79. https://doi.org/10.1109/MCC.2018.064181122.
29. Byabazaire, J.; O'Hare, G.M.P.; Collier, R.W.; Delaney, D.T. IoT Data Quality Assessment Framework Using Adaptive Weighted Estimation Fusion. *Sensors (Basel, Switzerland)* **2023**, *23*. https://doi.org/10.3390/s23135993.
30. Post-Quantum Cryptography PQC - Selected Algorithms: Digital Signature Algorithms. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. Accessed: 2025-03-03.
31. Kul, S.; Tashiev, I.; Sentas, A.; Sayar, A. Event-Based Microservices With Apache Kafka Streams: A Real-Time Vehicle Detection System Based on Type, Color, and Speed Attributes. *IEEE Access* **2021**, *9*. https://doi.org/10.1109/ACCESS.2021.3085736.
32. Karimov, J.; Rabl, T.; Katsifodimos, A.; Samarev, R.S.; Heiskanen, H.; Markl, V. Benchmarking Distributed Stream Data Processing Systems. *2018 IEEE 34th International Conference on Data Engineering (ICDE)* **2018**. https://doi.org/10.1109/ICDE.2018.00169.
33. van Dongen, G.; den Poel, D.V. Evaluation of Stream Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems* **2020**, *31*, 1845–1858. https://doi.org/10.1109/TPDS.2020.2978480.
34. Bartock, M.; Souppaya, M.; Savino, R.F.; Knoll, T.; Shetty, U.; Cherfaoui, M.; Yeluri, R.; Malhotra, A.; Banks, D.; Jordan, M.; et al. Hardware-Enabled Security: Enabling a Layered Approach to Platform Security for Cloud and Edge Computing Use Cases. 2021. https://doi.org/10.6028/NIST.IR.8320.
35. Nakaike, T.; Zhang, Q.; Ueda, Y.; Inagaki, T.; Ohara, M. Hyperledger Fabric Performance Characterization and Optimization Using GoLevelDB Benchmark. In Proceedings of the 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), 2020, pp. 1–9. https://doi.org/10.1109/ICBC48266.2020.9169454.
36. P., S.; Venkatesan, M. Scalability improvement and analysis of permissioned-blockchain. *ICT Express* **2021**, *7*, 283–289. https://doi.org/https://doi.org/10.1016/j.icte.2021.08.015.
37. Traini, L.; Cortellessa, V.; Pompeo, D.D.; Tucci, M. Towards effective assessment of steady state performance in Java software: are we there yet? *Empirical Software Engineering* **2022**, *28*. https://doi.org/10.1007/s10664-022-10247-.
38. Monitoring Linux with the Node Exporter. https://prometheus.io/docs/guides/node-exporter/. Accessed: 2025-03-03.
39. Extra PMIC features: RPi 4, RPi 5, and Compute Module 4. https://pip.raspberrypi.com/categories/685-whitepapers-app-notes/documents/RP-004340-WP/Extra-PMIC-features-on-Raspberry-Pi-4-and-Compute-Module-4.pdf. Accessed: 2025-03-03.
40. Kingman, J.F.C.; Atiyah, M.F. The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society* **1961**, *57*, 902 – 904. https://doi.org/10.1017/S0305004100036094.
41. Raspberry Pi hardware - Power supply: Typical power requirements. https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#typical-power-requirements. Accessed: 2025-03-03.
42. Keval, H.U.; Sasse, M.A. to catch a thief – you need at least 8 frames per second: the impact of frame rates on user performance in a CCTV detection task. In Proceedings of the ACM Multimedia, 2008. https://doi.org/10.1145/1459359.1459527.
43. Bekaroo, G.; Santokhee, A. Power consumption of the Raspberry Pi: A comparative analysis. In Proceedings of the 2016 IEEE International Conference on Emerging Technologies and Innovative Business Practices for the Transformation of Societies (EmergiTech), 2016, pp. 361–366. https://doi.org/10.1109/EmergiTech.2016.7737367.