ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

INGEGNERIA E TECNOLOGIA DELL'INFORMAZIONE PER IL MONITORAGGIO STRUTTURALE E AMBIENTALE E LA GESTIONE DEI RISCHI – EIT4SEMM

Ciclo 36

Settore Concorsuale: 01/B1 - INFORMATICA

Settore Scientifico Disciplinare: INF/01 - INFORMATICA

ARCHITECTING THE EDGE-CLOUD CONTINUUM FOR IOT-BASED MONITORING APPLICATIONS

Presentata da: Ivan Zyrianoff

Supervisore

Prof. Marco Di Felice

Coordinatore Dottorato

Prof. Luca De Marchi

Co-supervisore

Prof. Tullio Salmon Cinotti

Esame finale anno 2024

Abstract

Monitoring the status and ensuring the integrity of appliances in civil and industrial scenarios in real-time and over time is a top priority worldwide. To achieve this goal, it is necessary to have a strong synergy between multiple tools, disciplines, and approaches, which can be attained through a joint hardware-software co-design of the different components in an Internet of Things (IoT) system. Layered IoT architectures facilitate understanding of the different software components, hardware, sensing, and networking roles of smart IoT applications. These architectures are inherently distributed, spanning from devices installed in the field up to the cloud, passing through intermediary stages at different levels of edge computing infrastructure – constituting a computational infrastructure known as the edge-cloud continuum. IoT software platforms based on layered architectures are expected to be adaptable to scenarios with varying characteristics, requirements, and constraints from stakeholders and applications. However, they still face challenges, such as the lack of interoperability and managing data across the edge-cloud continuum. A fine balance exists between providing data with minimal delay and satisfying data freshness constraints. The lack of generality also hampers using the same architecture in multiple deployment scenarios. This thesis proposes, implements, and evaluates a multi-layer IoT architecture that is infrastructure-agnostic and designed to meet the challenges of interconnecting system components. The architecture seamlessly interfaces devices, applications, and subsystems through abstractions. It efficiently manages data across the edge-cloud continuum, enabling timely access to data by utilizing customizable proactive edge-caching techniques. Finally, we demonstrate the architecture's versatility by deploying it in different structural health monitoring (SHM) settings with varying requirements, sensing units, and infrastructure configurations.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Smart applications leverage the benefits provided by the massive amount of data generated by thousands or millions of sensors in the Internet of Things (IoT) [2]. In pair, the next generation of engineered structures will be equipped with *intelligent sensor systems* featuring on–board and advanced *decision-making functionalities*. Hence, implementing monitoring architectures requires perfect coordination among the sensing, communication, and decision subsystems to achieve a timely and reliable diagnosis [3]. To realize the potential gains of such coordination, they have to deal with the inherently distributed nature of real-world IoT infrastructures [4]. The end-to-end data path starts with the acquisition by sensors in the field, spanning multiple edge computing nodes for processing, storage, and communication tasks, ending up in a cloud data center that executes physical or data-driven models and can act as long-term storage. Indeed, a processing, storage, and communication *continuum* between sensors and the cloud composed of various infrastructure elements became clear [5]. It covers the end-to-end path of data acquisition, processing, storage, and transmission, starting with devices and spanning different intermediate stages, ending up in the user interface.

Requirements and Challenges

The effectiveness of monitoring systems is based on the optimal integration between the required hardware resources for signal acquisition, conditioning, and digitalization and the associated software infrastructure in charge of data management, data analytics, and visualization [6]. Such integration must consider the requirements posed by the application domains. We highlight the following challenges that currently hinder such integration:

1. **lack of interoperability**: the developed architectures must handle the heterogeneity of sensing units, which may differ based on the type of sensed signals, data formats, and acquisition protocols [7].

2. **scalability struggles**: resilient monitoring strategies are needed to ensure operational serviceability in burdensome workloads. Managing IoT data is a pivotal challenge since the acquired information may exhibit all the four dimensions of big data (volume, variety, velocity, and veracity) [8].

3. **data management issues**: data management policies could be implemented at any stage of the computational continuum. For instance, damage-sensitive features can be processed on *the extreme edge*, directly on the sensors' board, on *the edge* (i.e., on computational units nearby the sensors' board) or on *the cloud* for scenarios involving onerous post-processing phases [9]. Additionally, due to the sheer volume produced and requested by IoT systems, the transmission of requests to and from the data producers introduces augmented latency, overuse of network traffic, and increased costs – certain web services operate on a utilization-based payment model. When utilizing proprietary cloud services, costs are calculated also based on the usage of the volume of downloaded data. It is necessary to optimize data management throughout the continuum [10].

4. **inadaptability**: Deployment scenarios often entail non-compatible infrastructure, posing challenges regarding adaptability, and the distribution of software components across the edge-cloud continuum presents trade-offs. Executing processing at the edge reduces the workload for subsequent components, resulting in bandwidth and processing efficiencies. However, edge devices may have lower robustness compared to cloud resources [11]. Indeed, human expertise is needed to analyze each scenario and develop a custom-designed solution to attend to the constraints and requirements of the analyzed use case [12].

To meet these requirements, some research studies on smart structures propose multi–layered, IoT-based architectures involving both smart sensor devices in charge of measuring, pre-processing, and forwarding physical data and remote processing units, which merge and handle the huge data volume, finally executing structural assessment algorithms [13–17]. While the referred solutions have achieved considerable results, open points still need to be tackled. Firstly, some research works fail to consistently deploy all the cyber-physical components, as is the case of [13, 15, 17], or vice versa; they fail in test-fielding the software components on real-world scenarios [16]. In the case of complete deployments, the system

often has custom-made components or assumptions that match the specific use case or particularities of the scenario. Indeed, the lack of generality makes architectures barely extendable to support different sensing, processing, and monitoring tasks. The deployment of IoT architectures is a complex task. Real use cases often impose requirements, and the rapid and straightforward solution is to make an ad-hoc modification that solves it. However, this approach narrows the generality of the architecture. Replicate architectural solutions that support the base data acquisition and processing process to enable versatile IoT architectures.

Research Objectives

The main objective of this thesis is to *design and implement a customizable IoT monitoring solution that leverages the edge-cloud continuum and is adaptable to a variety of structural health monitoring (SHM) scenarios with heterogeneous infrastructure*. Four research questions (RQ) arise to accomplish the mentioned objective:

i *How can a seamless interconnection be established among diverse devices, applications, and systems in the context of IoT monitoring applications?*

ii *What strategies can be employed to enhance the efficient management of data within the edge-cloud continuum, emphasizing enabling low latency while considering data freshness?*

iii *How can decentralized systems be integrated to enhance the trustworthiness of data collected from diverse IoT devices in scenarios demanding reliability?*

iv *How can we deploy the same architecture across diverse scenarios with different edge-continuum configurations while considering different system end-goals and requirements?*

Contributions

To achieve the main objective, we advance the state of the art by introducing a novel, infrastructure-agnostic, multi-layer IoT architecture. This architecture is a general-purpose platform for SHM, leveraging cutting-edge technologies from the information, software, and industrial engineering communities. Its design should be modular and customizable. Based on the scenario requirements, components could be included to support a given feature, such as a specific interoperability integration or a latency requirement. Based on the architectural design and its implementation, we define our main macro-contributions:

1. Solution to RQ (i): Our approach to interoperability focuses on creating solutions for seamlessly connecting heterogeneous devices, applications, and systems. We avoid introducing new models that could contribute to fragmentation within the current IoT landscape. Instead, we opt for an integration strategy based on well-established and widely adopted open standards and technologies. We adopted the W3C Web of Things (WoT) standard [1] as a foundation for building interconnected IoT ecosystems. First, we establish a bridge between devices and applications by integrating WoT with the NGSI data model [18], which is utilized by the FIWARE Platform [19], and seamlessly connecting Web services documented through the OpenAPI Specification [20]. Subsequently, we extend this integration to encompass System-of-System solutions, enabling interfacing with legacy subsystems by leveraging the Arrowhead Framework [21].

2. Solution to RQ (ii): Edge caching is a potential solution to meet latency constraints [22]. In particular, proactive edge caching explores the pattern of user requests to predict and prefetch data to satisfy latency constraints while meeting information freshness requirements [23]. We designed a distributed framework for proactive edge caching called `CACHE-IT`. It considers the particularities of IoT scenarios and explores the proximity of edge infrastructure to devices with cloud resources to optimize data processing and response time. It decouples the caching strategy algorithm from the underlying architecture to ensure customization based on application-specific requirements.

3. Solution to RQ (iii): We have integrated a trustworthiness layer into a decentralized system by leveraging blockchain technology, where clients pay for queries to devices that are rewarded based on the returned data quality. Our proposal employs distributed applications that link the blockchain to the off-chain entities referred to as *oracles* [24]. These oracles mediate the data access interface among IoT devices, the blockchain, and clients.

4. Solution to RQ (iv): Our architecture enables custom choices, allowing for fine-tuning trade-offs to cater to specific use cases due to its infrastructure-agnostic nature. This claim was validated through its utilization in two different projects: the EU Arrowhead Tools[1] project and the MAC4PRO project [25], both of which required the architecture's deployment in real-world use cases. MAC4PRO is a national project that aims to integrate Industry 4.0 monitoring with advanced models for cost-effective,

[1] https://tools.arrowhead.eu/home/

safety-driven maintenance of components and infrastructure. The Arrowhead Tools Project involves approximately 90 partners from 17 EU countries and aims to provide digitalization and automation solutions for the European industry. From the conceptualization of the base architecture to its practical application in MAC4PRO and its further customization for the SHM Arrowhead Tools demonstrator, we provide a comprehensive guide to the practicalities of the proposed architecture.

Methodologies

We utilize several research methodologies to achieve the objectives described, including:

- ***Literature review***: To characterize the current state of the art of IoT architectures for monitoring applications and identify the main challenges and requirements for the proposed scenario.
- ***Architectural design***: we adopted an architectural-approach to problem-solving. Thus, designing and structuring the system's components, interfaces, and interactions systematically to specific challenges and requirements inherent to the IoT. This approach creates a framework that solves immediate problems and provides flexibility, adaptability, and future-proofing for evolving IoT landscapes.
- ***Performance evaluation***: we systematically assess and measure the designed solutions' efficiency, reliability, and scalability. It involves rigorous testing and analysis to ensure that the proposed solutions meet the immediate functional requirements and demonstrate their efficiency and trade-offs under various conditions, considering varying workloads and system configurations.
- ***Real-world use case***: we performed practical implementation and execution of the proposed solutions in live, operational environments. This approach moves beyond theoretical or simulated scenarios in IoT systems, allowing the architecture to interact with actual devices, networks, and end-users. Indeed, real-world deployments provide valuable insights into the robustness and adaptability of the architectures under authentic conditions, enabling the validation of theoretical findings in practical settings.

Thesis Structure

The thesis structure mirrors the proposed IoT architecture. Chapter 2 provides an overview of the multi-layer IoT architecture proposed for monitoring scenarios. The following chapters

are dedicated to the thesis contributions for each specific architectural layer. Chapter 3 deals with interoperability and addresses RQ (i), while Chapter 4 covers data management and addresses RQ (ii). Chapter 5 concludes by presenting the service layer and responding to RQ (iii). Next, we demonstrate the applicability and versatility of the architecture in various real-world scenarios, which is the theme of Chapter 6. Finally, Chapter 7 concludes the thesis, summarizes the contributions made, and draws future directions. Each chapter follows a structured methodology. We begin with an introduction to the motivation and subjects that will be addressed in that chapter. The first section covers the *literature review* of the specific theme, comparing the innovations presented in the chapter with the state-of-the-art. Following this, each component proposed presents an *architectural design* to solve the problem, which was implemented and validated through a *performance evaluation*. In many cases, a *real-world use case* is also provided to illustrate the proposal's applicability.

Chapter 2

IoT Edge-Cloud Continuum Architecture

Section 2.1 describes the current state-of-art of IoT architectures for monitoring scenario. Finally, Section 2.2 presents the architecture, which is composed of four layers: Sensing, Interoperability, Data Management, and Service. The subsequent chapters of this thesis build upon the foundational architecture established in here. Chapter 3, Chapter 4 and Chapter 5 delve into specific aspects of interoperability, data management, and services, respectively. Within these chapters, we explore in depth the contributions made in this thesis within each vertical, while considering the edge-cloud continuum scenario.

2.1 Background

In this section we present the state-of-the-art on IoT monitoring architectures that leverage the edge-cloud continuum. Subsection 2.1.1 defines the concept of the edge-cloud continuum, while in Subsection 2.1.2 we present the main architectural solutions found in literature for IoT monitoring applications, focusing in the SHM domain.

2.1.1 The IoT Edge-Cloud Continuum

Many authors acknowledge the existence of a processing, storage, and communication continuum composed of infrastructure elements located between the cloud and the sensors and actuators [5, 26, 10]. The continuum receives slightly different names (e.g., cloud continuum, cloud-edge-continuum, IoT-cloud-continuum) but always refers to the availability of resources far from the cloud and close to the devices and the location where the computation takes place. In the scope of this thesis, we refer to it as the edge-cloud continuum, which we outline stages in the continuum between the cloud and the end-device as follow: *Cloud* $\leftrightarrow$ *External Edge* $\leftrightarrow$ *Internal Edge* $\leftrightarrow$ *End-Device*, as illustrated by Figure 2.1. A trade-off

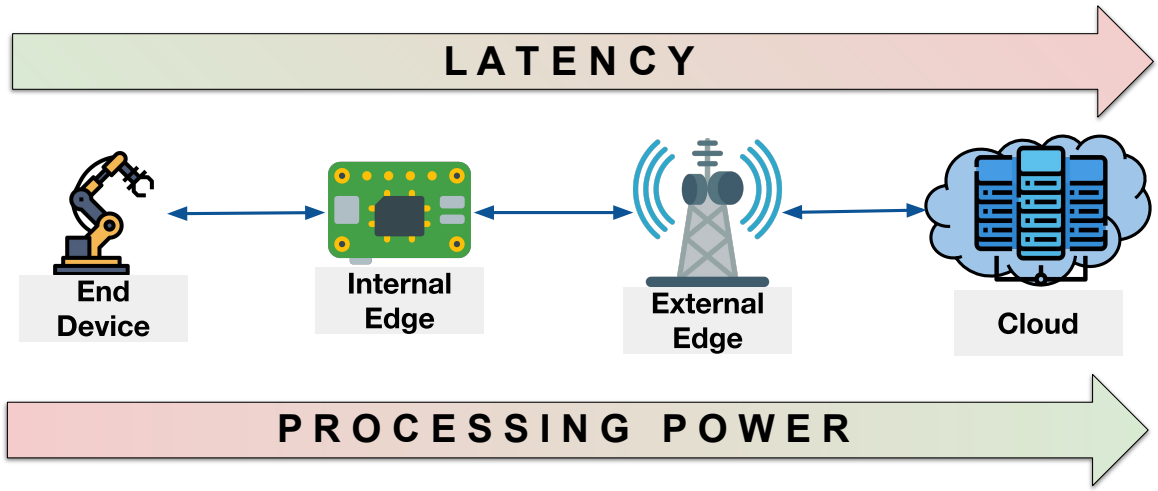


Fig. 2.1 Edge-cloud continuum regarding latency and processing power.

emerges in determining the optimal location for performing computation, whether closer to devices or to the cloud. Elements closer to the devices present low delay but lower computing capacity, whereas external edge devices are closest to the cloud, providing more powerful computing resources with higher delays [27, 28]. Besides the latency considerations, moving applications closer to edge has the potential to save bandwidth utilization and reduce network traffic [29].

Although the cloud-edge continuum is a novel paradigm, there are efforts in literature to define it. Moreschini [5] is one of the most impactful works in that regard. They reviewed the term cloud continuum and define it as "an extension of the traditional Cloud towards multiple entities (e.g., Edge, Fog, IoT) that provide analysis, processing, storage, and data generation capabilities.". They address the cloud continuum definitions, how they evolved, and which types of infrastructures it spans. Other researchers, consider also mobile nodes, such as unmanned aerial vehicles (UAV), part of the continuum [30]. Khalyeyev [31] acknowledge the widespread use of the edge-cloud continuum, highlighting the consensus gap about its meaning and properties, causing the lack of models and tools for reasoning about application development and deployment. They address this issue by proposing a reference component model for reasoning about smart applications running in the continuum.

A challenge when exploring the continuum is the vast heterogeneity of computation nodes, which difficult the deployment of systems when considering diverse infrastructural configurations. These concerns primarily revolve around the extensive variety of devices, spanning from single-board computers like Raspberry Pis to robust multiprocessor servers [26].

2.1.2 IoT Multi-layer Architectures

In this thesis, we proposed a four-layer architecture IoT architecture for IoT monitoring scenarios. There is a vast literature of layer architectures for IoT-based systems; however, there is little convergence – or standards –, which led to a myriad of heterogeneous approaches and taxonomies [32]. Hence, we identify several layered IoT architectures that can be applied to monitoring that stand out, and we compare these to our proposed solution.

The three-layer architecture was proposed in the early development stages of IoT [33, 34], and it represents the most simple and generic definition of an IoT-based architecture. Thus, it can be implemented in all IoT application domains. It is composed of: (*i*) Perception or Sensing Layer, which includes all the sensors and acquisition devices which are necessary to collect data; (*ii*) Network Layer, which serves as the intermediate layer between the edge devices and the central unit. Among its primary functions, it is responsible for transferring data while ensuring a secure connection in a sensor–to–cloud direction; (*iii*) Application Layer: it includes all the essential services the monitoring process requires, such as data processing and visualization.

Despite its ease of implementation, such an IoT approach lacks consistency in that it cannot capture the inherent complexities of the current condition monitoring requirements. A variation of the traditional three-layer architecture for SHM was proposed by Zonzini et al. [35]. The architecture encompasses the particular features of SHM-based sensors (e.g., accelerometers) and introduced computing capabilities in the network edge, which is responsible for exposing the IoT devices through an interoperable interface. The same system was further enhanced in [36], where the importance of the edge was highlighted, making it a layer of its own. However, this framework mixes architectural layers with computing locations.

A four-layer variant of the IoT architecture has been proposed to distribute better the tasks between the architectural resources of the monitoring system, thus favoring the development of a more versatile and timely responsive framework. Compared to the three-layer architectures, the four-layer variant introduces a purposely–devoted *Processing* or *Support layer* in between the networking substrate and the terminal application domain[37]. Regarding architectures that focus on SHM domains, we highlight Lamonaca *et al.* [38], which defines application, event detection, signal processing, and sensing as the layers of the SHM Systems. Its goal is to create a framework where all IoT systems for SHM can fit. Finally, five-layer IoT architectures introduce an additional level above the application services known as *Business layer*, since it orchestrates the entire IoT system as a whole [39]. In the scope of SHM, there is a known five-steps architectural guideline for implementing IoT-based systems for SHM. The steps – or layers – are sensing, gateway, network, control, and graphic interface [40, 41].

Diverging from the discussed approaches, other researchers deployed monitoring systems that do not fit in any of the mentioned layer-based IoT architectures. Wang *et al.* [42] proposes and implements an architecture to perform continuous monitoring. It comprises three tiers: edge, platform, and enterprise. The edge tier, via the IIoT gateway, connects to the cloud and platform tier, and the enterprise tier accesses data from the platform tier. Qian *et al.* [43] presents an IoT-based approach to condition monitoring of the wave power generation system, though, they do not present a software architecture to support it. Yang *et al.* [44] proposes a cloud-based monitoring platform for industrial applications, their system also encompasses edge computing nodes for performing tasks that require low latency.

2.2 IoT Architecture for IoT-Based Monitoring Systems

We took inspiration from the 5-layer IoTecture defined in [39]. The objective is to present a high-level generic and flexible architecture capable of supporting deployments comprised of numerous heterogeneous devices, different communication technologies, applications with non-uniform interfaces, and multiple end-user roles, ranging from managers to data scientists and system administrators. Figure 2.2 depicts the layered architecture adopted. Each layer defines a specific function, which can be performed by one or multiple applications, that connects with the other adjacent layers. The functionalities of the upper three layers are provided by multiple architectural components, which comprise the following chapters of the thesis. The architecture layers are:

1. *Sensing* layer, encompasses devices responsible for interacting with the physical world. This category comprises all sensors and actuators, such as accelerometers, gas, and piezoelectric sensors. This layer also includes the physical communication medium of the network stack –e.g., Wi-Fi, LTE, LoRa, etc. The sensing layer is out of the scope of the current thesis and our work involves the integration of it to the upper-layers.

2. *Interoperability* layer, offering uniform and standard interfaces inter-layer interactions and on-boarding tasks, as automatic discovered of applications and devices. This category is comprised of two sets of tools:

   (a) Communication-enablers: the tools responsible for data to be delivered, such as protocol-specific message brokers (e.g., MQTT Brokers) and network layer enablers, (e.g., the LoRaWAN server stack);

   (b) Homogenization tools: provide uniform interfaces to consume data (e.g., the WoT standard) and bridge different data structures or protocols.

Chapter 3 explores the state-of-art advances made in this thesis which addresses interoperability.

3. *Data Management* layer, encompasses tools that are responsible for the data storing, filtering, processing, and transformation operations. A notable aspect within this is the efficient data caching in the edge-cloud continuum. Chapter 4 presents this thesis contributions regarding data management.

4. *Service* layer, is comprised of services for end-users and integration with third-party applications or systems. This includes data visualization as well as interactions that can potentially lead up to commands that change the current state of the system. Chapter 5 illustrates this layer by describing one system integration.

The architectural design is decoupled from the deployment plan. Based on the requirements and the available resources, the software components can be variously configured and deployed through the edge-cloud continuum. For instance, they can be assigned entirely to edge nodes near the monitored structure or distributed among the cloud and the edge nodes. We emphasize that the flexibility in distributing software components through the continuum pertains to deployment time rather than run-time. Nevertheless, our architecture inherently supports run-time migration since the migration of services between edge and cloud was already explored in literature [45–47]. We stress that our innovations rely not on specific algorithms but on the architectural system combining hardware and software components. We provide evidence of such decoupling in Chapter 6, in which we deploy the architecture in different configurations.

Each architectural component designed and incorporated in the architecture is domain-agnostic, ensuring versatility across various deployments. This characteristic enables the components to be employed in different contexts and scenarios, showcasing the adaptability of the architecture to diverse application domains within the IoT landscape. The architecture follows a modular design, where applications operate independently of each other. The independence of applications, coupled with interoperability components, enables easy integration of new devices or applications. Those aspects promote to easily customize the architecture to meet specific requirements of given use cases.

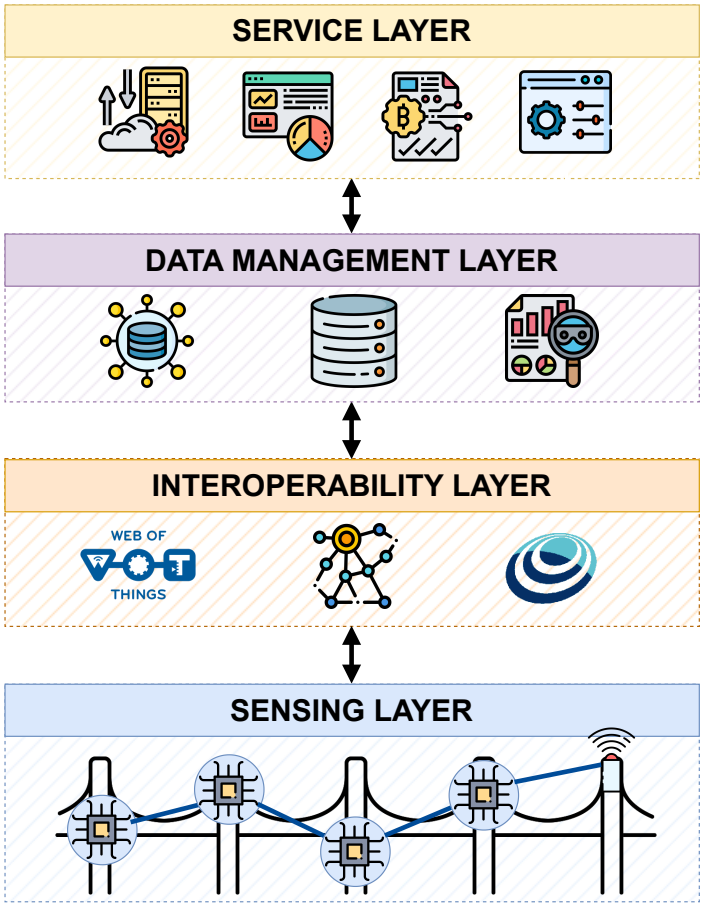


Fig. 2.2 High-level four-layer IoT Architecture

Chapter 3

Interoperability Layer: Web of Things in the IoT Edge-Cloud Continuum

This Chapter answers to the RQ (i) "*How can a seamless interconnection be established among diverse devices, applications, and systems in the context of IoT monitoring applications?*" by presenting the interoperability advances made in the scope of the thesis, which focus on the creation of seamless cross-perspective interoperability solutions based on well-established and widespread open standards and technologies. Most solutions designed to overcome interoperability challenges focus on a specific perspective, which we categorize into three distinct domains: device, application and system perspectives.

Figure 3.1 illustrates the relations of the different IoT interoperability perspectives, a set of corresponding solutions and which interoperability levels their address. Despite the availability of multiple IoT solutions addressing interoperability challenges, they typically are restricted to address the issues in their specific perspective. Consequently, creating an IoT ecosystem that seamlessly enables interoperability from devices to systems remains a leading challenge in IoT research. In this chapter, we adopted the W3C Web of Things (WoT) standard [1] as a foundation to build upon by designing and implementing solutions to create more integrated ecosystems.

Section 3.1 presents the current state-of-art of the interoperability solutions. The remainder of the chapter focus on interoperability integration. Specifically, we are focusing on two integration efforts: bridging the device perspective with the application perspective, which is addressed in Section 3.2; and connecting both device and application perspectives to the system perspective, addressed in Section 3.3.

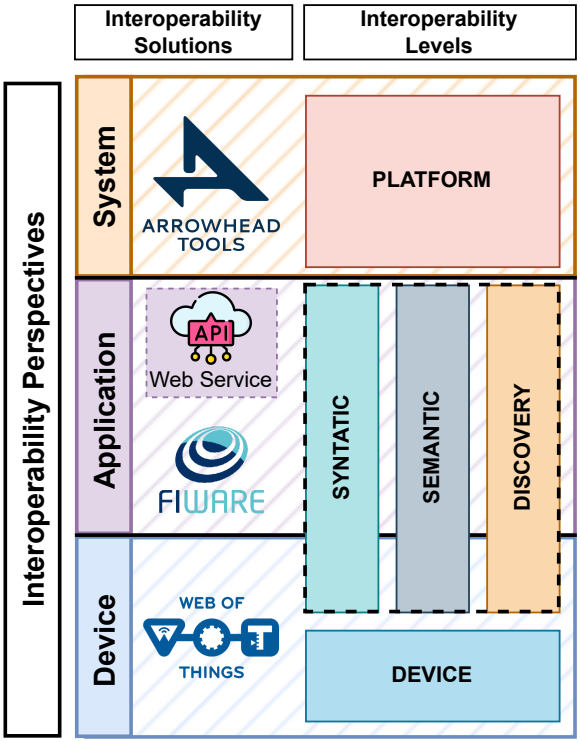


Fig. 3.1 IoT interoperability perspectives, solutions and levels addressed

3.1 Background

Studies, such as [48], have categorized various levels of IoT interoperability, including device interoperability, network interoperability, syntactical interoperability, semantic interoperability, and platform interoperability. We expand that definition by adding IoT discoverability. We present what each level encompass:

- *Device Interoperability*: IoT encompasses a wide range of devices, from high-end resources such as Raspberry Pi and smartphones to resource-constrained counterparts such as RFID tags, sensors, and actuators. These devices exhibit variations in micro-controller architecture, system characteristics, and communication technologies. The presence of multiple communication protocols, including Wi-Fi, 4G/5G, Bluetooth, NFC, ZigBee, Z-Wave, and proprietary solutions such as LoRa and SIGFOX, necessitates device interoperability. This level ensures the seamless integration of heterogeneous devices with different communication protocols, facilitating the exchange of information and the integration of new devices into any IoT platform.

- *Network Interoperability*: the networks supporting IoT devices are heterogeneous, multi-service, multi-vendor, and distributed. Unlike desktop computers, IoT devices

rely on a variety of short-range wireless communication and networking technologies that can be intermittent and unreliable. Network interoperability addresses the challenges associated with ensuring seamless message exchange between systems across different networks. This includes addressing, routing, resource optimization, security, QoS, and mobility support in the dynamic and heterogeneous IoT network environment.

- *Syntactical Interoperability*: refers to the interoperability of both format and data structure in the exchange of information or services between heterogeneous entities. To achieve this, interfaces must be defined for resources, exposing structures according to specific schemas such as WSDL and REST APIs. Message content is serialized for transmission, typically using formats such as XML or JSON. Challenges arise when the encoding rules of the message sender are incompatible with the decoding rules of the receiver, resulting in mismatched message parse trees.

- *Semantic Interoperability*: IoT involves enabling different agents, services, and applications to meaningfully exchange information, data, and knowledge. Challenges arise from different data models, incompatible data schemas, different units of measurement, and semantic inconsistencies. As a result, IoT systems have difficulty interoperating dynamically and automatically, even as they expose their data and resources. Achieving semantic interoperability requires addressing the differences in how IoT systems describe and understand resources and operations.

- *Platform Interoperability*: exists due from the existence of different operating systems, programming languages, data structures, architectures, and access mechanisms across different devices and platforms. IoT-specific operating systems such as Contiki[1], RIOT[2], TinyOS[49], and OpenWSN[50], and platform providers such as FIWARE, ThingsBoard, Amazon AWS IoT, and IBM Watson contribute to this disparity. Developers face barriers to building cross-platform, cross-domain IoT applications that require in-depth knowledge of platform-specific APIs and information models for customization and integration.

- *Discover Interoperability*: the representation of device capabilities, automatic indexing mechanisms, and clients' search capabilities vary widely. Effective discoverability interoperability addresses the challenge of harmonizing how device capabilities are represented and providing mechanisms for clients to efficiently search and index

[1] www.contiki-os.org
[2] https://riot-os.org

devices. This involves standardizing approaches for device representation, indexing protocols, and search mechanisms to enhance the overall discoverability experience in the IoT ecosystem.

The interoperability challenges identified are distributed across diverse perspectives, as shown in Figure 3.1. This proposed categorization serves as a complementary framework to the highlighted interoperability issues. One aspect of the IoT is the classification of heterogeneity, while the other aspect examines the challenges and their impact on devices, applications, and systems. Solutions for IoT interoperability are tailored to specific levels and perspectives. Solutions such as the W3C Web of Things (WoT) [1] standard addressed the inter-connectivity challenges from the *device* perspective, by proving a standardize way to describe and interact with devices. On the other hand, open IoT Platforms and web services aim to establish seamless connective between *application*, facilitating the process of integrating new applications. Finally, complex IoT scenarios usually require interfacing with multiple other subsystem, which usually involve the integration with legacy IoT ecosystems. Solutions such as the Arrowhead Tools project[3], enable automatic cross-platform and cross-domain integration between *systems*, including legacy ones.

The following subsections examine the a subset of the most promising interoperability solutions which were utilized in the remainder of this thesis. Specifically, Subsection 3.1.1 discusses the device perspective, while Subsection 3.1.2 presents the application perspective and presents a set of experiments that compared different IoT Platforms. Subsection 3.1.3 outlines the system perspective.

3.1.1 Device Perspective: the W3C Web of Things

Concepts and early ideas surrounding the WoT have been in development since 2007 [51]. However, it wasn't until 2013 that the World Wide Web Consortium (W3C) initiated its standardization efforts in the WoT domain. This began with the establishment of the Web of Things Community Group and later with the Web of Things Interest and Working Group in 2017. Unlike traditional approaches, which often propose the creation of new protocols or middleware layers, the W3C WoT approach revolves around a descriptive information model capable of representing diverse solutions. The core of this proposal is the definition of a Web Thing (WT), which indicates any "*physical or a virtual entity whose metadata and interfaces are described by a Thing Description (TD)*' '[1]. The latter denotes a sequence of standardized, machine-understandable metadata encoded in JSON-LD [4] that models the

[3]https://cordis.europa.eu/project/id/826452
[4]json-ld.org

capabilities of an IoT device. The TD contains detailed instructions about the nature of

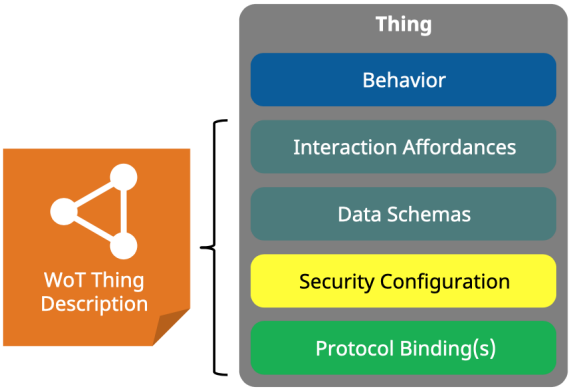


Fig. 3.2 W3C Web Thing architecture proposed in [1].

a device or service – i.e., a WT. In practice, the information model standardized by W3C contains a set of interaction patterns (or affordances) that a service or device is capable of supporting. The affordances group together a set of atomic functions, called operations, that clients can use to interact with the represented WT. There are three types of affordances:

- **properties**: represent the inner state of a WT – e.g., the current temperature of a smart thermometer or the configuration parameters of a coffee machine. A client can perform the following operations: write read, observe, and unobserved.
- **actions**: high-level functions that WTs offer to their clients. Usually, actions operate on the physical world or modify the WT's state. Examples include: toggle a smart lamp or move a robotic arm to the desired position. A client can invoke the action and if it is a long-running function it can later query its state or cancel its execution.
- **events**: data sources that asynchronously push data to clients. Typically, alarms or expected states are modeled as events, but not regular property changes that are modeled with the Property affordances. The operations grouped under the events affordances are: subscribe and unsubscribe.

The Listing 3.1 illustrates a TD of a Accelerometer sensor that has only one property, which returns a object composed of three properties, each one related to an axis. The affordances enable clients to leverage on a stable high-level interface while the *Protocol Binding Template* defines how clients can perform the operations described in the TD. It describes low-level concepts like, the protocol to use and its configuration parameters, in an ontology. Then these terms can be imported inside a TD and used to describe how to

perform a specific operation. These low-level communication requirements can be paired with security constraints enabled by the *Security Schemas*. Security Schemas are common Web authentication paradigms that can be declared to be used when invoking an action or any affordance operation. The TD only contains the methodology to access the affordance, not the sensitive information (e.g., username and password) required to interact with the WT. As written in the security guidelines[52] a TD should never contain sensitive information. Figure 3.2 depicts the main WT architecture components encompassed by the TD described.

A run-time software named *Servient* implements the software object described by the TD. The Servient allows to host and *expose* a WT (i.e., to make the TD available over a network) and to interact with a remote WT by *consuming* the TD. According to W3C, [1], a WT functionality should be available in all available protocols. Thus, Servients bind multiple protocols and data models to enable interactions with different platforms. The W3C description does not clarify if there is a one-to-one relationship between WTs and Servients. Throughout this thesis, we assume that a Servient can host several WTs.

```
{
    "title":"Accelerometer Sensor",
    "properties":{
       "Acceleration":{
          "type":"object",
          "description":"Current acceleration result",
          "observable":false,
          "readOnly":true,
          "writeOnly":false,
          "properties":{
             "type":"object",
             "properties":{
                "x":{ "type":"number" },
                "y":{ "type":"number" },
                "z":{ "type":"number" }
             }
          }
       }
    }
 }
```

Listing 3.1 Example of accelerometer sensor TD

Discovery

A crucial WoT feature is to dynamically discover TDs at runtime. This aspect of discovery is addressed by the normative specification of the W3C Web of Thing Discover [53]. The document focuses on the normative steps required to obtain and publish TDs over the Web. The approach to acquiring TDs adopts a two-phase architectural model, balancing the dual demands of openness and controlled access to metadata, ensuring that only authorized entities can access the necessary information. The first phase, known as *"Introduction,"* is employed to discover one or more candidate URLs. These URLs are treated as opaque strings, deliberately devoid of any substantial metadata. During this phase, the process remains entirely open, with no restrictions applied to consumers. The candidate URLs are acquired through one of the defined introduction methods. Presently, there exist five introduction methods: well-known URLs, Direct (e.g., QR codes or manual URL provision), DNS-Based Service Discovery, CoRE Link Format and CoRE Resource Directory, and DID Documents.

Upon obtaining a set of candidate URLs, the Discoverer proceeds to the second phase, denoted as *"Exploration."* This phase encompasses the operations necessary to retrieve the TD referenced by the URLs and further processing to extract additional information. Typically, these operations are protected by security mechanisms, such as authentication tokens, ensuring that the TDs remain inaccessible to unauthorized users. Notice that the URL obtained via one introduction mechanism invariably directs to a single TD hosted by an exploration service. Discoverers must be capable of interacting with different types of exploration services:

- **Thing Description Server**: "Any web service that can be referenced by a URL and returns a TD with appropriate authentication and access controls can be used as an exploration mechanism" [53].
- **Thing Description Directory (TDD)**: serves as a WT that offers services for managing a collection of TDs describing other WTs [53]. The TDD facilitates a broader set of APIs for filtering and searching for the desired TDs.

Thing Description Directory

A TDD is an exploration service that can be used to retrieve and filter a list of TDs. Currently, the specification is focusing on TDD based on HTTP but, in the future, it might support other non-web-native protocols like CoAP or MQTT. Implementers of a TDD are required to support a set of compliant APIs exposed as HTTP endpoints. Currently, those APIs are

grouped into three categories: things, events, and filtering. Things endpoints are further subdivided into creation, retrieval, update, deletion, and listing. Those functions represent the CRUDL operations for the set of TDs stored inside the service. The specification recommends protecting relevant resources with secure protocols and credentials. The events API allows the client to subscribe to the basic events fired by the TDD like the creation of a TD, an update, or a deletion. Finally, the filtering API comprehends three different querying technologies that implementers can choose to support or not: JSONPath [54], XPath [55], and SPARQL [56].

WoT State-of-the-art

In this subsection, we introduce the current state-of-the-art research on WoT and focus on the modern solutions of indexing and searching WTs.

One of the most prominent verticals of WoT research is to design and develop a mechanism to enable the W3C WoT standard to integrate with non-WoT components seamlessly. Indeed, a known shortcoming of the W3C WoT standard is the lack of out-of-the-box conversion methods to dissonant interfaces to its ecosystem. Implementation efforts are often needed to integrate third-party Web services or other standard interfaces into the WoT. Recent advances filled that gap, providing seamless integration of RESTful Web services [57] and NGSI-based interfaces [58] to the W3C WoT ecosystem. Other efforts are on the live migration of WTs [45] to cope with the intrinsic dynamicity of IoT environments in terms of time-varying network and computational loads.

The WoT standard enables abstracting the device's physical properties and creating interoperable interfaces that facilitate seamless communication within IoT systems. However, efficient indexing and searching of WTs are fundamental aspects for the widespread adoption of WoT [59]. Numerous techniques have been proposed to address WT search challenges [60], varying in the adopted query language and overall technology. Among these, IoT-SVKSearch [61] stands out as a promising approach, supporting searches based on both spatial-temporal attributes and value-based criteria, effectively incorporating the dynamism of IoT environments into the search mechanism. GOLDIE [62] offers a hierarchical location-based WoT directory architecture that includes federated identity management and IoT-specific features like discoverability, aggregation, and geospatial queries. DBAC [63] innovates in the access-control vertical, enabling decentralized attributed access without the need for complete trust or credential provision while preserving user privacy. Other efforts have focused on indexing WTs for specific scenarios, such as indoor devices [64]. In [64], device features are automatically extracted using machine learning techniques and clustered to group similar devices. Although these works advance the state-of-the-art in WT indexing

and searching, they do not align with current W3C standards for discoverability, leading to an increasingly fragmented landscape with multiple heterogeneous solutions for indexing and querying devices.

There are two other W3C-compatible implementations, namely TinyIoT [65] and WoT Hive [66]. TinyIoT holds a historical significance as the first implementation of the APIs outlined within the specification. Originating as a research project within the Fraunhofer Institute, it has since evolved into an independent open-source endeavor. The service, implemented in Go, uses an integrated LevelDB instance for the storage and querying of TDs. It supports a comprehensive feature set, including DNS-SD as an introduction service for the TDD, complete implementation of all mandatory APIs, and a JSON-Path query endpoint. While the software solution is robust, the queries are performed entirely in memory, which could potentially pose challenges when deploying TinyIoT in large-scale environments.

WoT Hive has been developed inside the European project AURORAL[5] and wants to be the most feature-rich implementation of the TDD APIs. The service is written in Java with the help of the Spark framework and it supports SPARQL endpoints as storage for the list of TDs. In contrast to TinyIoT, WoT Hive boasts more robust semantic and syntactic capabilities due to its backend support for Triple stores, supporting both JSONPath, SPARQL-based discovery and semantic validation. On the other hand, the expanded feature set compromises scalability with a large set of TDs as demonstrated in [66].

3.1.2 Application Perspective: Open IoT Platforms

Currently, there is a plethora of new and different IoT Platforms, each with its own set of features, requirements, and trade-offs – the Unify-IoT [67] project identified more than 300 different IoT platforms. In this myriad of options, stakeholders often make a choice based on assumptions not supported by reliable data, as "*the more interoperability features an IoT platform provides, the worse its overall performance and scalability*." We aim to provide a preliminary analysis of open-source IoT Platforms qualitatively in terms of interoperability and comparing them with a quantitative performance analysis that asses the scalability of each platform. to Due to the vast number of IoT platforms, a single study cannot encompass a detailed evaluation of each of them. First, we restrict our analysis to only *open-source* platforms. Second, we identify three classes of platforms and select a single representative from each. We categorized IoT Platforms if they are maintained by (1) a collaborative community of independent developers (e.g., FIWARE Platform[19] founded and fostered by the European Commission); (2) a start-up (e.g., Konker[6] supported by a Brazilian company);

[5] https://www.auroral.eu/

[6] https://www.konkerlabs.com/index-en.html

(3) an already established company (e.g., ThingsBoard[7] is one of the market-leading IoT platforms). Our goal is to provide an initial assessment through a small-scale performance evaluation to understand if the selected platforms have a trade-off regarding its interoperability features. Further studies are required to in fact proof the hypothesis stated.

FIWARE

The FIWARE platform [19] is an open-source IoT framework fostered and funded by the European Commission under Horizon 2020 program. It comprises software modules that perform functions needed in various IoT-based applications – Generic Enablers (GE). There are another set of applications that integrate with FIWARE and usually are domain-specific. Those are labeled as *Powered by FIWARE*.

IoT Agent applications are data model-specific, so each different data structure requires a new IoT Agent. There are currently agents for the following data models and protocols: LWM2M over CoAP, JSON or UltraLight over HTTP/MQTT, OPC-UA, Sigfox LoRaWAN [68]. Further, a NodeJS library to enable IoT Agent development allows developers to build custom agents to connect non-support data structures/network protocols to the FIWARE ecosystem.

Applications in the FIWARE ecosystem adopt a standard NGSI (Next Generation Service Interface) data exchange model that enables communication between them. IoT Agents are components that handle IoT data heterogeneity in FIWARE, translating IoT-specific protocols into the NGSI context information protocol [68]. Additionally, IoT Agents map NGSI information as virtual representations of the IoT devices in JSON entities, stored and managed by Orion, a publish/subscribe context broker.Applications can consume and publish IoT data through Orion using NGSI REST-based web interfaces. Hence, interoperability is granted once applications are in the FIWARE ecosystem and use the NGSI data model. IoT Agents have a standard API that enables CRUD (Create, Read, Update and Delete) operations of devices - and defines the corresponding translation to the NGSI model. IoT Agents store device metadata in a database.

IoT Agent applications are data model-specific, so each different data structure requires a new IoT Agent. There are currently agents for the following data models and protocols: LWM2M over CoAP, JSON or UltraLight over HTTP/MQTT, OPC-UA, Sigfox LoRaWAN [68]. Further, a NodeJS library to enable IoT Agent development is available to build custom agents to connect non-support data-structures/network protocols to the FIWARE ecosystem.

[7] https://thingsboard.io

ThingsBoard

ThingsBoard is an open-source platform that enables device management, data collection, and visualization for IoT-based systems. It enables connectivity via industry standard IoT protocols – HTTP, MQTT, and CoAP. ThingsBoard allows users to build dashboards and offers multiple options of widgets and graphs to improve IoT data visualization.

ThingsBoard IoT Gateways integrate devices connected to legacy and third-party systems with the IoT platform. There are gateways for external MQTT brokers, OPC-UA servers, Sigfox Back-end, Modbus slaves, or CAN nodes. Additionally, ThingsBoard offers a guide for developers to build custom IoT Gateway to integrate not supported protocols. A critical ThingsBoard feature is to connect several different data sources, enabling users to build a data processing rule chain – based on Node-Red[8] – capable of transforming, processing, triggering actions to devices, and integrating with other third-party applications – only in paid versions.

Konker

Konker is a cloud-based open-source IoT Platform that collects data and connects devices. It aims to offer intuitive and simple features to enable users to prototype solutions within minutes. Konker supports HTTP and MQTT devices and adds a security layer on top – each device is connected to a unique ID and password-authenticated by the platform. Devices and applications are connected in Konker through a processing pipeline which the actors can the following roles:

- ***Device***: a JSON representation of a physical or virtual device that transmits or receives data;
- ***Channel***: groups message with similar content to be processed as a single unit;
- ***Route***: connects input devices to output applications or devices;
- ***Transformation***: enables payload structure manipulation of messages connected through a *channel*.

A Qualitative Comparison of IoT Open Platforms

Our analysis of platform interoperability perspective focuses on how well platforms bridge different network protocols and data models from heterogeneous IoT-based devices. We

[8] https://nodered.org/

consider the out-of-the-box supported integration and whether the platform provides a framework or a generic gateway to assist in interfacing non-compatible devices. Additionally, we investigate whether platform interfaces can connect to third-party applications or other platforms for mashing IoT data. Table 3.3 summarizes the comparison of the mentioned characteristics of the analyzed platforms.

One core aspect of any IoT Platform is being able to bring heterogeneous device data to the platforms. Hence, all platforms provide a way to bridge IoT devices that communicate in different network protocols and data structures. FIWARE and ThingsBoard adopted a similar approach, both utilized middleware applications that translate IoT-specific protocols to data that the platform can process. However, an essential difference between the two is that FIWARE IoT Agents are tied to the network protocol and data model, and the ThingsBoard IoT Gateways are only associated with the network protocol and the user needs to provide a code-snipped to integrate a given data model. As Table 3.3 showcases, ThingsBoard is the platform that supports more different network protocols, followed by FIWARE. However, both provide a framework to assist the development of new translation middleware applications – Generic Gateway column of Table 3.3.

In contrast with the other platforms, Konker adopts restrictions to receive data. Each device must send a payload message with an individual user and a password automatically created by the platform. The additional overhead to configure each specific device with an individual authentication excludes several devices, such as those that do not allow changes in the payload; or low-powered IoT devices with payload size limitations. Compared to the other platforms, Konker offers the least amount of support to different network protocols (MQTT and HTTP only), and it only receives structured as JSON.

A common feature of all platforms is the representation of IoT devices as JSON-based virtual entities that model the capabilities of that device. Other components of the system can be represented as a virtual entity to describe its meta-information as static data or relationships with other physical entities – i.e., a room temperature can be inferred as the average of all the temperature sensors located there. The entity-based model is a characteristic that hinders the fragmentation in IoT environment at the *platform level*. It provides a platform-agnostic way to model data and represent devices; thus, combining data from different platforms with few efforts is possible.

Another ubiquitous feature of IoT Platforms is that they provide a REST API to access and manipulate services. However, each platform has its custom endpoints, specific data models, and specific capabilities. FIWARE allows users to query time series data with several parameters and aggregations. In contrast, ThingsBoard does not provide a direct way to

query and aggregate time-series data. The lack of uniformity hinders the development of cross-platform applications, requiring development efforts to implement such solutions.

Although the analyzed platforms lack standard interfaces, they offer built-in integration with other industry-established platforms. FIWARE and ThingsBoard offer integration with know LPWANs platforms, particularly those that support LoRaWAN. ThingsBoard also integrates with other proprietary platforms like AWS IoT, IBM Watson, Microsoft Azure. However, all the platform integration supported by ThingsBoard is only available in its paid Professional Edition (indicate by the * in Table 3.3). There is also the possibility in ThingsBoard to create custom integration with other applications, but that requires coding efforts and pipeline modeling in a graphical interface.

Table 3.1 Summary of the interoperability comparison between the IoT Platforms

	Application Protocols	**Generic Gateway**	**Platform Integration**	**Entity Representation**	**Data Exchange Protocol**
FIWARE	HTTP, MQTT, CoAP, OPC-UA	yes	*LPWANs*: ChirpStack, The Things Network, SigFox	JSON	NGSI
Konker	HTTP, MQTT	no	No Integrations Available	JSON	none
ThingsBoard	HTTP, MQTT, CoAP; *Supported throught IoT Gateways*: OPC-UA, Modbus, BLW, CAN, BACnet, ODBC, SNMP, FTP	yes	*IoT Platforms*: AWS IoT, IBM Watson, Microsoft Azure*;*LPWANs*: ThingPask, The Things Network, TEKTELIC, LORIOT, SigFox, NB-IoT Network *	JSON	none

Konker does not provide direct integration with other platforms, but with the *transformation* concept, it is feasible to connect Konker data streams to third-party applications. Hence, a *transformation* is a code-snipped that can be connected to a data stream to format the data in a structure that comply with other interfaces. Thus, it is possible to convert to the format of other platforms and trigger an action to send data to that platform or through an MQTT broker or the application REST API.

FIWARE provides two ways of integrating non-NGSI-based applications in its ecosystem: (*i*) IoT Agent-based: FIWARE provides an IoT Agent Node.js library to enable developers to build custom agents to connect applications or devices to the FIWARE ecosystem by translating the data structure and network protocol to NGSI on top of HTTP – to communicate with RESTful APIs; (*ii*) Wrapper-based: build a GE that encapsulates the third-party service with an NGSI interface, for instance: Cygnus[9] that encapsulates Apache Flume[10], and

[9]https://github.com/telefonicaid/fiware-cygnus
[10]https://flume.apache.org/

Draco[11] that encapsulates Apache NiFi[12]. One advantage of FIWARE is that the numerous GEs often offered capabilities not well supported by the other platforms, as querying and aggregating time-series data through APIs – three GEs enable such features: QuantumLeap[13], STH Comet[14], Cygnus. Performing time-series query operations in Konker or ThingsBoard is more complex and limited than in FIWARE QuantumLeap.

Although the three platforms provide a way to assist developers in integrating with other non-supported platforms it is still required for developers to know the specific data structures and interfaces to integrate third-party applications and services.

A Quantitative Comparison of IoT Open Platforms

We designed and conducted experiments to quantitative compare the performance and scalability of each platforms in close to real IoT scenarios. One goal was to answer the hypothesis: "*the more interoperability features an IoT platform provides, the worse its overall performance and scalability*." In every experiment, we recorded : *(i)* CPU and RAM usage from the IoT Platform; *(ii)* end-to-end processing time. In the experiments, a synthetic IoT sensor workload generator (SenSE)[69] transmits sensor data through the HTTP protocol to the IoT Platform, responsible for forwarding the data to an application consumer. The consumer is responsible for handling the data, recording, and calculating the processing time. Each component was executed in a different virtual machine to guarantees that the execution of one component does not hinder the performance of others.

Table 3.2 presents factors and levels used in each scenario, consisting of 36 possibilities. The scenarios consider the implementation of all three analysed IoT Platforms in two different IoT smart applications: Smart City (SC) and Smart Health (SH).

In the Smart Cities scenarios, we used a 25-byte payload to simulate a simple sensor sending a single data, such as temperature or humidity. In the Smart Health scenarios, the payload is increased to represent better typical medical data such as patient identification, glucose levels, heart rate, steps, location, medicines, ECG, and blood pressure, totaling 3.865 bytes [70, 71]. To evaluate how each platform manages system resources, we analyze the performance of each platform on two separate infrastructures based on AWS[15] standards, an AWS.Medium (2 vCPUs and 4Gb RAM) and an AWS.Large (4 vCPUs and 8Gb RAM) virtual machines.

[11] https://github.com/ging/fiware-draco
[12] https://nifi.apache.org/
[13] https://github.com/orchestracities/ngsi-timeseries-api
[14] https://github.com/telefonicaid/fiware-sth-comet
[15] https://aws.amazon.com

Table 3.2 Factors and Levels

Factor	**Level**
Payload Size	smart city: 25 bytes smart health: 3685 bytes
Infrastructure	AWS.Medium 2CPUs 4Gb RAM AWS.Large 4CPUs 8Gb RAM
Workload (messages per second)	
AWS.Medium	8,16,24
AWS.Large	24,32,40
Platforms	FIWARE, ThingsBoard and Konker

Another factor analyzed was how each scenario performs with different workloads. In AWS.Medium scenarios, we use a workload of 8, 16, and 24 messages per second (mps). On the other hand, in the AWS.Large scenarios, the workload is increased to 24, 32, and 40 mps. For all evaluations, the asymptotic confidence intervals were computed at the level of 95%.

Figure 3.3 depicts the processing time of each platform in an AWS.Medium virtual machine, while Figure 3.4 shows the processing time in an AWS.Large virtual machine. Both in Figure 3.4 and Figure3.3 ThingsBoard had a better overall performance compared to the other analyzed solutions, as it presented the lowest processing times in almost all experiments. Further, ThingsBoard utiliz computational resources in a efficient manner – as show Figure 3.17 and Figure 3.18. ThingsBoard has a stable and regular increase in processing time, CPU and memory, and is compatible with the workload and payload.

Even though the Konker platform does not have the best performance, it shows to be stable since its processing time did not suffer meaningful variations when we altered the workload, scenario, and infrastructure. However, Konker stability comes with a trade-off regarding resource usage. Figure 3.5 depicts that Konker demands 10% of CPU at the lowest workload – i.e., 8mps. The same behavior can be seen in Figure 3.18, illustrating that Konker demands more than half memory available in all AWS.Medium scenarios while using almost all resources available in the AWS.Large.

FIWARE platform was the most impacted by the workload increase. FIWARE had the worst performance when analyzing the processing time in Figure 3.3 and Figure 3.4, even operating with the workload of 8mps. In terms of scalability, FIWARE could not complete the 40mps scenario, crashing before the end of the experiments – depicted by the white bars in Figure 3.4. Analyzing the usage of hardware resources in Figure 3.5, Figure 3.6, Figure 3.7 and Figure 3.8, we can conclude that FIWARE depends more on CPU than memory. Comparing the workload of 24mps in AWS.Medium with AWS.Large in both SC and SH, it is clear that the CPU is more demanded while the memory usage is still the same.

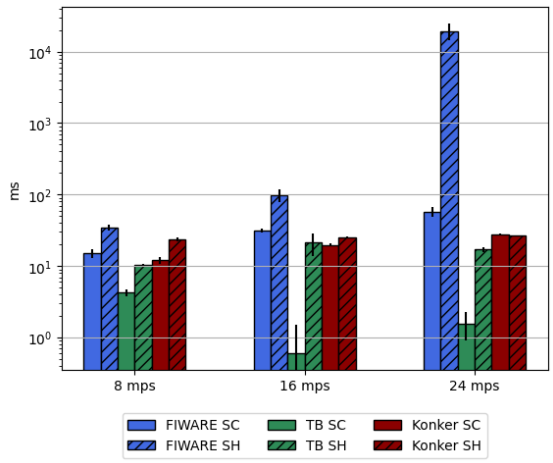


Fig. 3.3 Processing Time (ms) in AWS.Medium

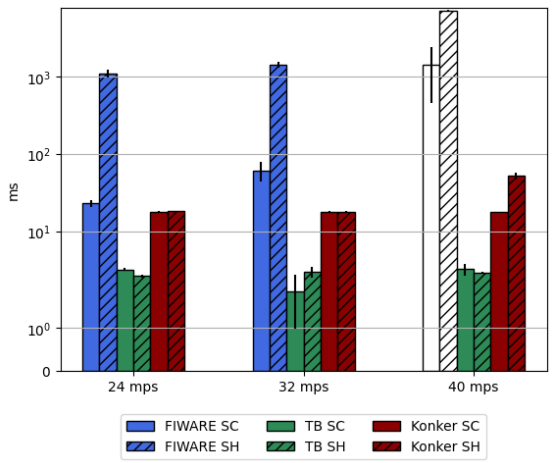


Fig. 3.4 Processing Time (ms) in AWS.Large

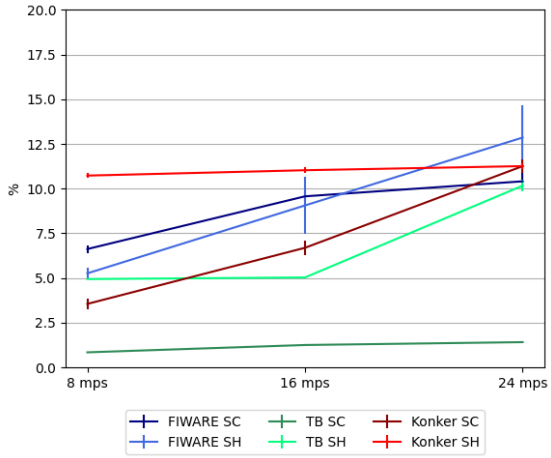


Fig. 3.5 CPU Usage (%) in AWS.Medium

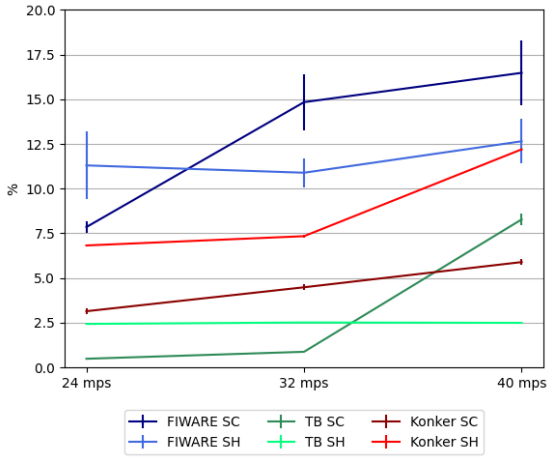


Fig. 3.6 CPU Usage (%) in AWS.Large

The experiment's results invalidate our initial hypothesis that related interoperability features provided by an IoT Platform with its performance. As stated in Section 3.1.2, ThingsBoard provides different interoperability features both in the device- and platform-level, being the platform that offers more out-of-the-box integration. Despite all these features, ThingsBoard presented the overall best results in terms of processing time. On the other hand, Konker's lack of interoperability features did not reflect better performance or more efficient usage of computational resources. There is no insights that a specific interoperability feature originated FIWARE's poor performance under high workloads. The results did not indicate that there is a trade-off between interoperability features and performance. However, the platform's ability to develop, maintain and support impacted both interoperability features

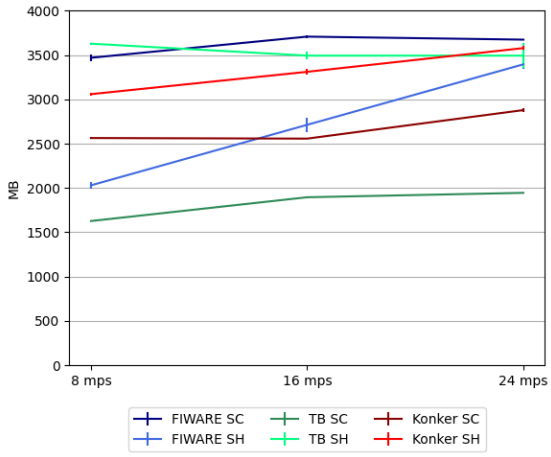


Fig. 3.7 Memory Usage (MB) in AWS.Medium

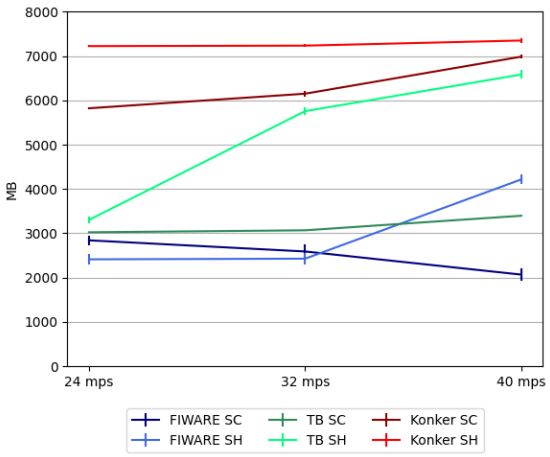


Fig. 3.8 Memory Usage (MB) in AWS.Large

and scalability. ThingsBoard platform is a product of an already settled company. Hence, there have dedicated teams to support and develop different aspects of their platform – as implement integration with third-party applications. On the other hand, Konker is supported by a small start-up that primary business model is to provide the entire IoT system to other companies. Thus, it is probable that they cannot spend time and resources to develop new integration with different IoT devices. Finally, FIWARE is an open-source platform supported and developed by a community of independent developers or companies that utilize the tool. Hence, an integration with a protocol or platform is often only available if one of the developers had that same necessity – and it is only supported by a long period if there is still interest in the community in that integration.

WoT and FIWARE Comparison

Since both FIWARE and the W3C WoT emphasizes open standards and community-driven development, we compared their interoperability solutions. Both FIWARE and WoT handle the heterogeneity in IoT environments through a common philosophy: map IoT devices as virtual entities with well-defined data structures that other applications can consume to interact with IoT devices. However, they differ in many aspects, as their interoperability perspective. FIWARE, which adhere to the application perspective, connects multiple entities through its context broker, while the WoT focus on increasing the connectivity of single entities, which are WTs. Figure 3.9 illustrates the interoperability approaches of WoT (A) and FIWARE (B). The architectural differences may also introduce qualitative differences on system deployments, reflected by the following aspects:

Implementation efforts: the FIWARE interoperability solution is an out-off-shelf application. Thus, programming efforts are only required if there is the need to implement a new IoT Agent for a previously unsupported data model or protocol. Opposite to that, there is not an out-of-the-shelf WoT application. However, there exist frameworks (e.g., the `node-wot`[72] tool) for assisting the development and the run-time execution of WTs.

Interfacing other applications: IoT Agents communicate only to FIWARE-based context brokers. However, FIWARE provides a vast catalog of generic and specific enablers that easily interface to any application with its ecosystem, third-party applications that do not communicate via NGSI standard require a connector to be bridged to FIWARE. A similar issue emerges in the WoT context: WT can be consumed through a Servient or by processing the WoT-specific format that is often not compatible with other IoT Platforms.

Flexibility: Both solutions require the development efforts for dealing with unknown protocols. However, the W3C WoT architecture descriptions provides clear guidelines that assist the implementation of new solutions.

Adaptability - i.e., adapt to new situations minimizing the need of a new deployment: IoT devices can be created, read, updated, and deleted to IoT Agent at run-time using its API. There is no specification in the WoT architecture of a similar interface. Although it is possible to implement such a feature in a Servient, it requires programming efforts.

Based on the aforementioned comparison, we can conclude that both solutions could gain from integrating one with the other. Although FIWARE already has an interoperability solution, it might extend the supported protocols and data formats, especially towards IoT devices that the IoT Agents do not already support. For WoT, integrating with FIWARE would vastly expand the support towards applications/platforms working with the NGSI model.

3.1.3 System Perspective: the Arrowhead Framework

The last decade has been dominated by a fast-paced industrial revolution, particularly affecting IoT-based ecosystems. In particular, Industry 4.0 no longer relies on legacy and monolithic SCADA/DCS systems, instead, they are supported by flexible Service-Oriented Architectures (SOA), where modular systems consume or provide services, ensuring loose coupling between modules and their reusability across multiple domains[73]. The Eclipse Arrowhead Framework is a software platform released as an open source product of the Arrowhead Project[16] that structures closed environments as Local Clouds: controlled ecosystems that implement the base concepts of SOA – loose coupling, late binding, and discovery

[16]http://www.arrowheadproject.eu/

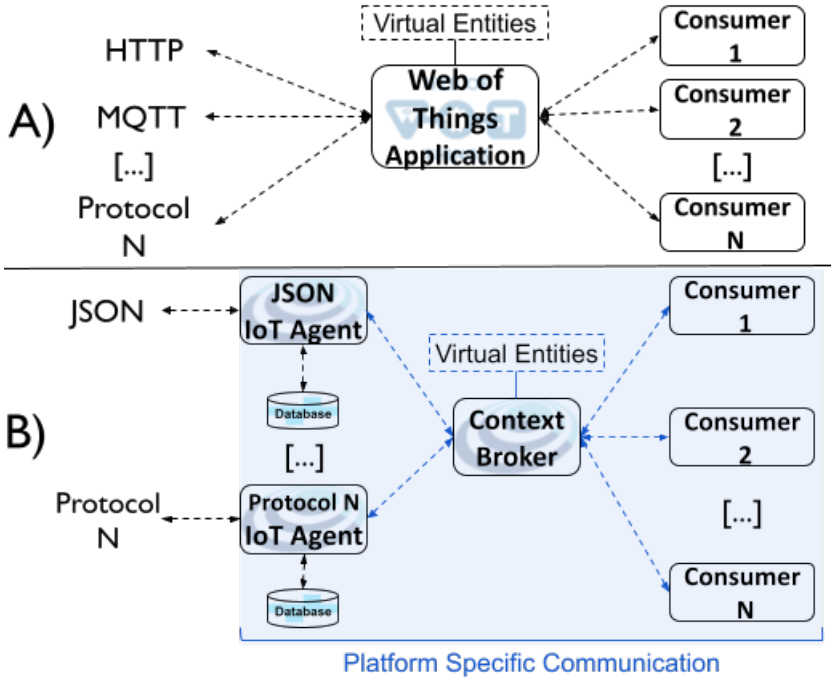


Fig. 3.9 WoT and FIWARE architectural definitions

– and hosts a single instance of a central coordination entity, called the Core Services [21]. Then, each Local Cloud acts as a System-of-Systems in which each system is either an *application system* (if an integral part of the baseline) or a *tool* . Regardless, they behave as service providers or service consumers. Service consumption is supervised and managed by the Core Services, which must be deployed in the Local Cloud in a minimum set. The latter defines the "mandatory" Core Services in order to be Arrowhead-compliant; which are the Service Registry, the Authorization, and the Orchestration. The **Service Registry** retains a service record – a set of metadata – for each of the services in the Local Cloud acting as a registrar which enables discovery and loose coupling. Service providers can register themselves or a modeling facility may be used in the design phase of the Local Cloud, such as SysML [74]. Each service record contains the essential details for interacting with such service (i.e., the endpoint and the service name) as well as additional details in case the service is annotated via a well-known standard (e.g., OAS). Each service provider all its offered services independently all its offered services in the Service Registry via its API, so that service consumers can subsequently fetch the necessary reference to the services of interest. The **Authorization** is a storage of a set of authorization rules that specify whether a consumer is authorized to use a certain service. In addition, it provides a token-based authentication mechanism. Finally, the **Orchestration** is the enabler of late binding, as it allows an additional actor, the cloud manager, to associate directly consumers to providers

at run-time. This way, consumers cannot autonomously decide which service provider to query, instead, they query the Orchestration service to obtain the provider that was assigned to them.

3.2 Bridging Device to Application Perspective

Referring to the IoT interoperability perspectives Figure 3.1, this section describes our contributions to bridge the device perspective to the application perspective using the WoT standard.

Subsection 3.2.1 presents the design and implementation of ZION, an open source scalable W3C TDD. Its scalability is supported by a performance evaluation that demonstrates its efficiency compared to other TDDs. The discovery allows applications to automatically find and register WTs as such, serving as a first enabler to bridge both perspectives.

The following two subsections aim to provide connectivity from the WoT standard to another ecosystem. Since the W3C WoT addresses do not provide methods or guidelines for converting dissonant interfaces to their ecosystem, development effort and component knowledge are required to integrate an application to the WoT standard, and such a developed solution is often strongly tied to the specific application interface, operations, and data models. The subsection 3.2.3 outlines the design and implementation of a connector that bridges the WoT architecture to the FIWARE [19] ecosystem, leveraging the strengths of both solutions. The seamless connection of both ecosystems creates a cross-perspective ecosystem. The subsection 3.2.4 presents a technique to seamlessly integrate RESTful web services into the WoT ecosystem. Our solution enables the translation of *any* RESTful interface – provided its OpenAPI Specification (OAS) [17] – into a WT description and the procedures to instantiate the translated description into a WT that acts as a proxy for the actual Web application. Our mechanism decouples the application interface from the underlying network logic.

3.2.1 ZION: A Scalable W3C Web of Things Directory

ZION is an scalable, open-source TDD fully aligned with the W3C Discovery standard [53]. Its core values revolve around speed, flexibility, and user-friendliness. It comprises a standard API that facilitates CRUDL (Create, Read, Update, Delete, List) operations managing TDs. The same interface supports querying TDs metadata through JSONPath, following the IETF JSONPath standard [54], and offers robust pagination capabilities. To validate ZION's scalability, we conducted a comparative analysis with two other open-source

[17] https://spec.openapis.org/oas/latest.html

TDD implementations, specifically, WoT Hive [66], and TinyIoT [65]. We evaluated the querying performance of each TDD while indexing various quantities of WT.

Architectural Design

ZION's software architecture is modular, ensuring scalability and maintainability. The architecture is divided into:the API Reference, API Experimental, Authentication, Introduction, and Persistence modules. Each of these modules serves a distinct purpose, ensuring that the software can handle the diverse requirements of an IoT device directory. ZION's architecture is illustrated by Figure 3.10

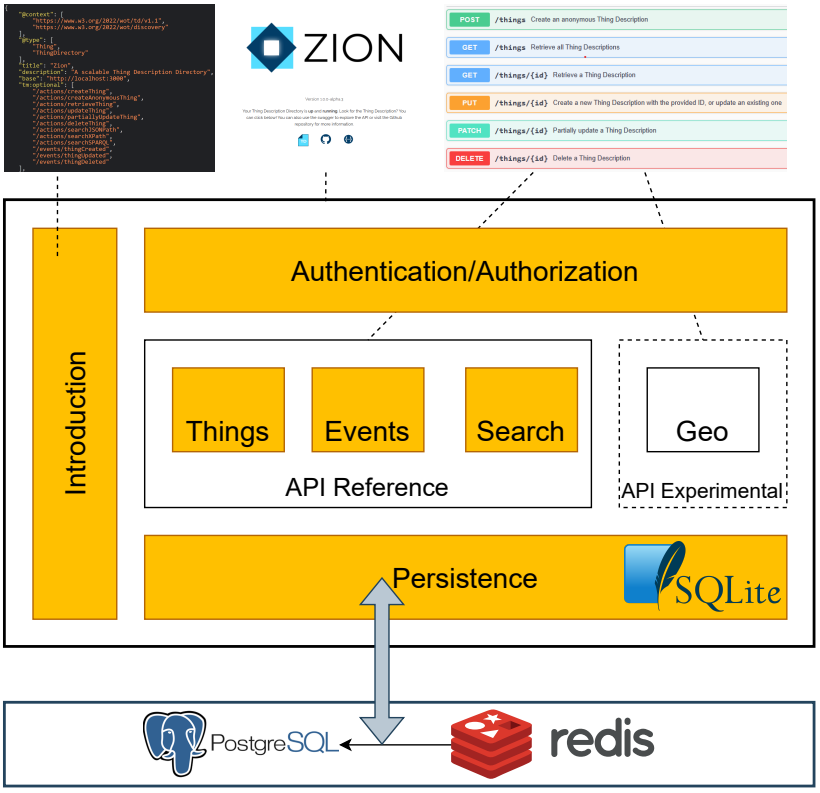


Fig. 3.10 ZION architectural design

API Reference

The API Reference offers a comprehensive implementation of the functionalities outlined in the W3C WoT Discovery document. It is structured into three distinct sub-modules: Events, Search, and Things.

The Events module handles API endpoints related to device events, allowing the tracking and monitoring of device activities. Clients can subscribe to all events or a specific event,

such as when a device is added, modified, or deleted. This real-time subscription mechanism is implemented using Server-Sent Events (SSE), ensuring immediate updates and efficient communication between the server and clients.

Within the W3C-specified Search API, advanced searching mechanisms are proposed, including JSONPath, XPath, and SPARQL. However, in our Search module, we prioritized the JSONPath implementation due to its flexibility and intuitive nature for querying JSON data structures which is the default encoding format for TDs.

The Things module provides comprehensive endpoints for TD operations, including creation, retrieval, modification, and deletion. Authentication mechanisms secure these operations, limiting modifications to authorized clients. Advanced querying options are available, allowing clients to filter and enrich TD listings with additional metadata and related information. The module's design adheres to RESTful API principles, offering intuitive endpoints and standard HTTP methods for simplified client integration and consistent interactions.

3.2.2 API Experimental

The API Experimental module is designed to host innovative features that extend beyond the current W3C WoT Discovery standard. While these features are still in the conceptual phase, the module's architecture is primed to accommodate them, ensuring that when they are developed, integration will be seamless. Among the anticipated features for this module are the geospatial API and a high-level tagging system for TDs. The geospatial API is envisioned to offer capabilities for managing and querying devices based on their geographical locations. On the other hand, the tagging system aims to provide a sophisticated mechanism for categorizing and organizing TDs, enhancing search and retrieval efficiency.

Authentication

The authentication module supports token-based authentication via username and password. This self-contained support model doesn't necessitate the use of an external service. However, we are actively working on refactoring this feature to adopt a more extensible approach. In the future, users will have the flexibility to select their preferred authentication mechanisms to suit their specific needs. For instance, they can opt for username and password authentication for smaller setups or utilize OIDC (OpenID Connect) for cloud-based deployments, ensuring enhanced security and user convenience.

Introduction

The introduction module implements the first phase of the WoT Discovery proccess and it currently supports DNS-SD, CoRE-RD, and the Well-known introduction methods. Its architecture is designed to be adaptable and open to further extensions in the future, enabling the inclusion of additional introduction methods as needed.

Persistence

The Persistence module serves as an abstraction layer for data storage and retrieval. It utilizes the Knex.js[18] query builder to establish a robust connection with the PostgreSQL database and generate the needed queries. This module not only ensures the efficient management of user data and TDs but also adeptly handles TD lifecycle events. The *AbstractRepository* offers a generic blueprint for basic CRUDL operations promoting modularity and reusability, with specialized repositories like the *UserRepository* and *ThingDescriptionRepository* extending these operations for their specific needs. The *ThingDescriptionRepository*'s capability to process JSONPath queries is particularly noteworthy. To achieve this, a dedicated library[19] was developed to translate JSONPath queries into SQL/JSON Path, the language natively supported by PostgreSQL. This translation covers 90% of the language, and it's sufficient to support the querying of almost every TD. The *TDLifecycleEventRepository* is currently in-memory, but there are considerations to migrate to more persistent storage solutions like Redis to enhance scalability.

Performance Analysis

We conducted a performance analysis to assess the scalability of the various implementations of TDDs compatible with the W3C WoT Discovery standard; namely: ZION, TinyIoT, and WoT Hive. Our focus was on characterizing the query resolution time under different workloads, as it represents the most critical metric in this context. Unlike the less frequent operations of inserting, removing, and updating TDs, querying TDs occurs more frequently under real-world conditions – as searching and consuming internet content is a more common activity than the tasks of creating or removing content.

In our experimental setup, we performed experiments where we systematically varied the quantity of TDs stored in the database across different scenarios. In detail, we conducted experiments for 10, 100, 1,000, 10,000, and 100,000 TDs. We categorized the TDs into different complexity levels according to their number of lines: simple, medium, and complex.

[18] https://knexjs.org
[19] https://github.com/vaimee/jsonpath-to-sqljsonpath

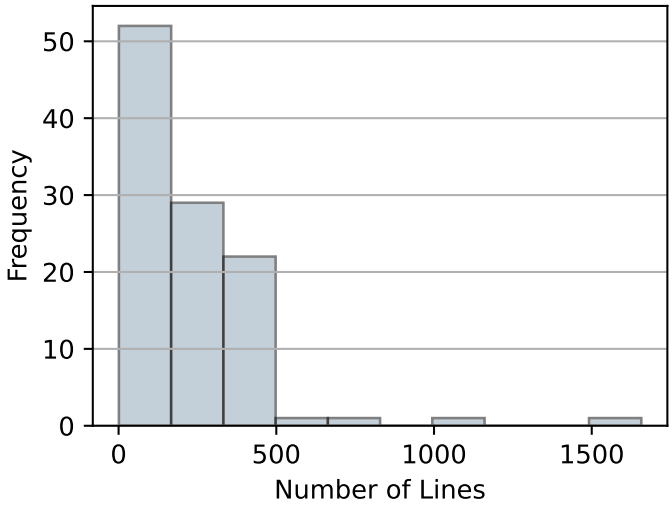


Fig. 3.11 Histogram that shows the distributions of WT TDs per number of lines

To guarantee a greater similarity with real-world scenarios, we distributed the categories of TDs stored in each experiment replication mimicking a Pareto distribution (the Pareto distribution is commonly employed to model the sizes of files on the internet) as Figure 3.11 depicts. Hence, from the collected dataset of TDs, we ordered the TDs by quantity of lines and the 5% percent with most lines were consider complex, the 15% below where considered medium TDs, and the 80% remaining ones were classified as simple. We emphasize that the TDs utilized in the experiments represented real devices and are publicly available in a GitHub repository[20]. Before each experiment, we populated the TDD with the TDs according to the specified distribution and complexity levels.

In each experiment, we conducted a hundred sequential calls to the TDD JSONPath search endpoint. For each call, we randomly select a query from the following options:

- `$[?(@.properties.lightColor)]`: this query searched for TDs containing the `lightColor` property.;
- `$[?(@.properties.lightColor && @.properties.brightness)]`: This query searches TDs featuring both the `lightColor` and `brightness` properties.
- `$[?(@.properties.sensorInformation)]. properties..state`: This query searches TDs with `state` as a sub-property within the `sensorInformation` property.property.

[20]https://github.com/vaimee/tdd-workload-generator/tree/main/src/populate-db/examples-td

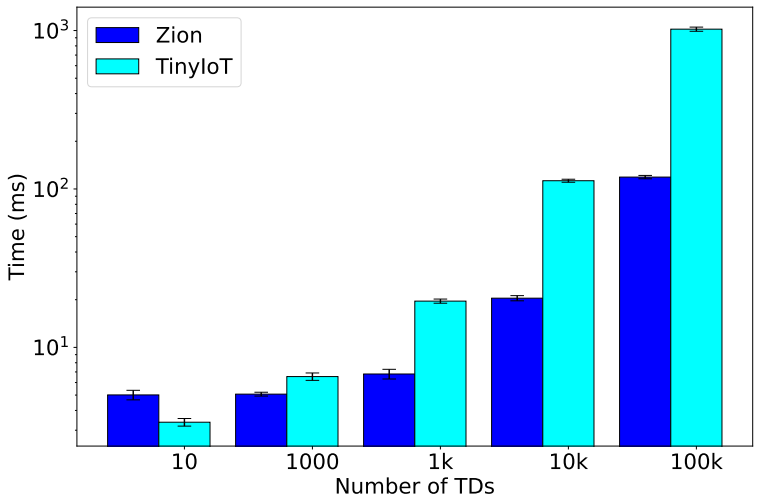


Fig. 3.12 Processing times for ZION and TinyIoT with y-axis in logarithm scale.

Preliminary experiments unveiled that all three queries have similar processing times. In each replication, we reset the database and re-populated it. The workload generator and the analyzed TDD were deployed in the same machine (12GB of RAM and an Intel Core i5-7200U CPU running at 2.50GHz with 4 cores). We containerized ZION and TinyIoT using Docker. Each experiment was replicated 30 times and asymptotic confidence intervals were computed at the level of 99%.

The results are depicted in Figure 3.12, note that the y-axis is in logarithm scale. We omitted WoT Hive from the graphs due to its unfeasible high processing time in all tested workloads. In the lowest workload scenario with 10 TDs, the average processing time was 0.94 seconds. This average increased to 1.85 seconds with 100 TDs and further extended to 8.61 seconds with 1,000 TDs. During all WoT Hive experiments, we encountered numerous error replies and experienced denial of service from the server. Our experiments support that WoT Hive is not suitable for the use cases defined in this study. Regarding the performance comparison for TinyIoT with ZION, we can note that the effect of increasing workload on ZION results in a steady, linear rise in its processing time. In contrast, TinyIoT experiences an exponential surge in processing time as the workload intensifies.

3.2.3 WoT-FIWARE Integration

We developed a generic application that bridges the WoT and the FIWARE ecosystems by translating the WoT data to the NGSI format. We opted to develop a standalone WoT

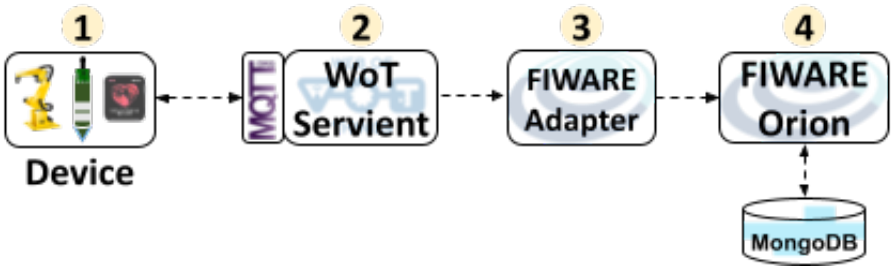


Fig. 3.13 WoT connection to the FIWARE ecosystem dataflow

mash-up application instead of implementing a direct connection from a WoT Servient to FIWARE due to two main reasons: 1) Generality: developing the Adapter as an application enables the communication between FIWARE and WoT for any system, not only for our scenario. 2) WoT best practices: the WoT Architecture [1] does not have a way of actively sending data in a specific network protocol to another software module. Instead, the mash-up application subscribes to a TD event, which transfers the data via a WoT-specific interface when triggered.

We developed the WoT-FIWARE Adapter in JavaScript, using NodeJS[21] on top of the node-wot framework [72]. Our Adapter subscribes to the `ngsiOutput` event in the TD and maps the WT as an Orion entity. The WT triggers this event whenever there is an update in one of its properties. The Adapter iterates through such properties and encapsulates them in a JSON object represented in NGSI. We virtualized the WoT-FIWARE Adapter as a Docker container, available as an open-source project[22].

Figure 3.13 illustrates the WoT connection to the FIWARE ecosystem dataflow. The complete steps depicted in Figure 3.13 are:

1. an IoT device sends a message using a WoT supported protocol (e.g., MQTT) to the WoT Servient and then to the WT associated with the device;

2. the WT processes the IoT message and updates the TD properties related to that device. In turn, it triggers the `ngsiOutput` event, which notifies the WoT-FIWARE Adapter;

3. The Adapter maps the received WT to a corresponding FIWARE Orion entity. If this entity does not exist in Orion, the WoT-FIWARE Adapter creates it;

4. Orion receives the message and stores the entity information in MongoDB.

[21] https://nodejs.org/
[22] https://github.com/UniBO-PRISMLab/WoT-FIWARE-adapter

Performance Analysis

In the following, we present a performance evaluation of the two interoperability solutions on a real-world IoT deployment. The goal of the evaluation is twofold: (*i*) to validate the operations of the WoT-FIWARE Adapter; (*ii*) to investigate further the performance trade-offs, scalability, and requirements of IoT Agent and WoT solutions. The quantitative comparison is influenced by the current implementations of the interoperability solutions.

We consider a performance analysis scenario based on a real IoT environment using the SWAMP Platform [75] for smart irrigation. In detail, sensor probes obtain soil data and transmit it to the SWAMP FIWARE-based Platform through LoRaWAN, where a set of mathematical and data-driven models are processed to generate an irrigation prescription map [76]. Figure 3.15 illustrates the complete dataflow from an infrastructural point-of-view, with real pictures of a SWAMP Pilot located in a Brazilian agriculture frontier [12].

Although the SWAMP reference sensor probe communication technology is LoRaWAN[23], some off-the-shelf soil sensors use the basic LoRa modulation. Those probes transmit data to a simple LoRa gateway that sends the sensor payload directly to the platform in a raw MQTT structure, thus bypassing ChirpStack [77], which is a LoRaWAN server implementation responsible for handling networking, authorization, and authentication issues. Soil probes sense the soil and transmit data to the LoRa Gateway every 10 minutes, structuring the payload according to the UltraLight2.0 (UL) protocol - a lightweight text-based protocol for constrained devices and communications where bandwidth and device memory may be limited [78].

Figure 3.19 depicts the core SWAMP Platform dataflow, including the two LoRa-based transmission methods. In the *Interoperability Solution* block, we tested two alternatives: 1) WoT software layer: composed of a Servient and the WoT-FIWARE Adapter that enables the communication with the FIWARE Orion. To this aim, we developed WTs for each SWAMP soil probes; the WTs are fed by the UL data and from the ChirpStack Server; 2) Native FIWARE environment: We used the UL IoT Agent in the experiments with basic LoRa modulation and the SWAMP LoRaWAN IoT Agent [12] for the LoRaWAN experiments. Currently, there is an official version of the LoRaWAN IoT Agent, but it is not fully integrated into the ChirpStack Server.

Regarding our testing environment, we used SenSE (Sensor Simulation Environment) as the synthetic IoT sensor workload generator that can abstract real devices and model complex scenarios [69].As the performance analysis focuses primarily on the interoperability solution, we emulated all the data flow modules prior to the interoperability application. In the experiments, SenSE produces synthetic data (i.e., simulating the soil probes transmitting

[23]https://resources.lora-alliance.org/technical-specifications/lorawan-specification-v1-1

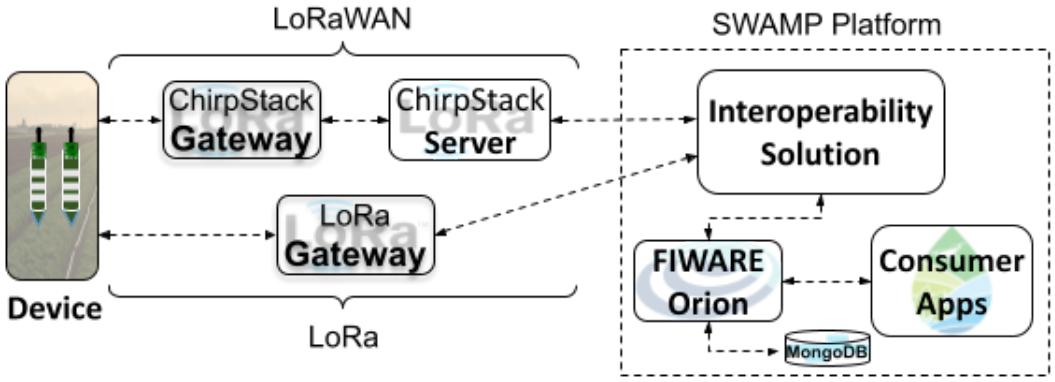


Fig. 3.14 Dataflow of the SWAMP IoT-based Platform

data) and sends it to the interoperability solution. The corresponding soil probe entity is updated in Orion, which sends a notification with the data to the Consumer. The software components used in the experiments were deployed as Docker containers. The Consumer represents a generic application that consumes sensor data - e.g., data visualization tool, mobile app, or a third-party application. It is implemented as a simple data sink.

The number of sensors and the sensor message periodicity - each emulated sensor sends a message each 10 min - does not vary during the experiment. We utilized two Virtual Machines (VM) to perform the experiments. In VM #1, we deployed the modules that enable the performance analysis - SenSE and Consumer - and in VM #2, we deployed the applications under test: the interoperability solution, FIWARE Orion, and MongoDB. We configured both VMs as the standard Amazon AWS t2.medium instance configuration (2vCPU - 4GB of RAM).

We conducted 18 experiments, varying the levels of three factors - workload, protocol, and interoperability solution - as depicted by Table 3.3. We evaluated WoT and FIWARE IoT Agent as interoperability solutions and UL and LoRaWAN as protocols - mixed protocol as appears in Table 3.3 refers to experiments where both UL and LoRaWAN protocols were used simultaneously. Also, the workload was varied from low (1,000 sensors), medium (5,000 sensors), and high (10,000 sensors). SenSE emulates SWAMP sensor probes generating one packet every 10 minutes. Each experiment was replicated 30 times, and asymptotic confidence intervals were computed at the level of 99%.

We focused on the following metrics in the analysis: end-to-end delay (i.e. the average time taken since a sensor data point is generated until the Consumer application receives it), percentage of delivered messages, and system metrics (i.e. CPU and RAM usage per Docker container of the evaluated modules, collected every five seconds.).

Table 3.3 Experiment Factors and Levels

Factor	Level
Interoperability Solution	Web of Things - FIWARE IoT Agent
Protocol	Ultralight2.0 - LoRaWAN - Mixed
Number of Sensors	1000 - 5000 - 10000

Fig. 3.15 SWAMP dataflow from infrastructure point-of-view

Figure 3.16 summarizes the key results of the performance evaluation, depicting the total experiment delay for the IoT Agent and the WoT software layer in low, medium, and high workloads for the three different types of traffic: LoRaWAN messages, UL messages, and mixed traffic - half of the sensors sending UL messages and the other half sending LoRaWAN messages. The y-axis is expressed on a logarithmic scale.

When comparing the IoT Agent and the WoT software layer's performance, it is important to stress that we utilized two different IoT Agent implementations. Thus, when observing Figure 3.16, we can conclude that the WoT (plus Connector) delay is similar to the official FIWARE IoT Agent. Nevertheless, it is worst than the SWAMP LoRaWAN implementation. In the experiments with mixed traffic, the WoT interoperability solution improved its performance regarding delay compared to its performance in experiments using LoRaWAN traffic. However, the IoT Agent performed better because both applications divided the processing between them, acting as a workload balancing.

Analyzing the delay for high workload, we conclude that neither application can keep up with 10,000 sensors, considering the computer resources allocated, since the experiments overall had delays from 18s to 51s. All messages were delivered in the IoT Agent experiment; however, some messages were lost when utilizing WoT. This loss is reported in Table 3.4, which shows the delivered message rate for all the experiments and reveals packet loss events under high workloads.

The computer resources usage are shown in Figure 3.17 - CPU usage - and Figure 3.18 - memory usage. We can observe that MongoDB is the application with the highest demand for CPU, caused by Orion. Each Orion entity is stored in MongoDB, and if an attribute is updated, Orion will also update that attribute in MongoDB, thus increasing the processing demand for the database. Regarding memory usage, there is a significant difference between

Table 3.4 Delivered Messages

Interoperability App.	Traffic	Workload	Delivered Messages
Web of Things	LoRaWAN	High	84.04 ± 6.10%
Web of Things	UltraLight	High	90.13 ± 3.57%
Web of Things	Mix	Medium	98.32 ± 1.14%
Web of Things	Mix	High	89.81 ± 2.80%

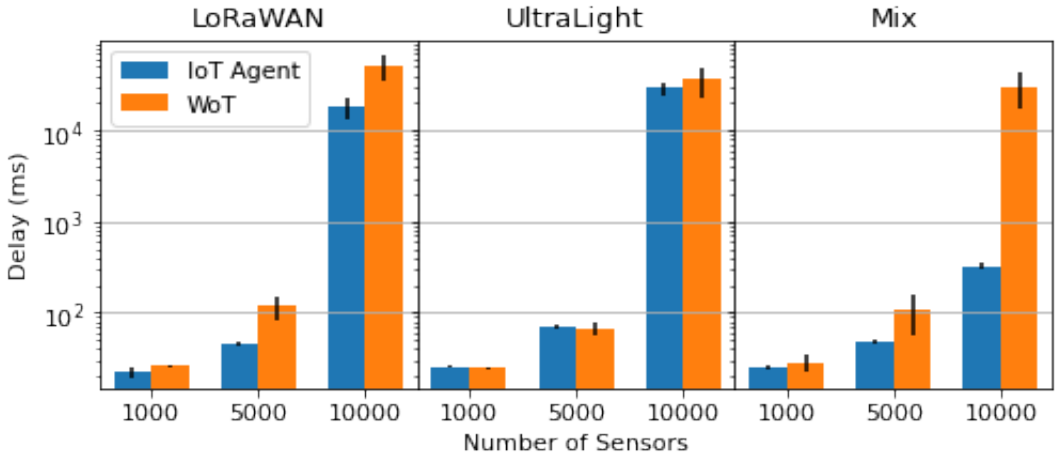


Fig. 3.16 Experimental delay

the WoT solution and the IoT Agents, which we believe is caused by the soil probe TD. A TD is verbose and individual to each different soil probe. Also, the `node-wot` implementation does not use a database. Thus all information needed is stored in the RAM. The depletion of RAM in the WoT experiments is the cause of losing messages. The usage of a DB is also reflected in the RAM usage, which was higher for WoT since the Servient allocated the TDs in memory.

The higher overall delay of the WoT software layer is expected since the latter is composed of two modules (WT and the Adapter); this solution may introduce more processing and networking steps that are not needed in the IoT Agent implementation. However, this is not true when comparing the WoT software layer with the official IoT Agent - i.e., in experiments solely with UL traffic - in this case, the delay is similar.

Moreover, the WoT requirement of not having protocol/platform-specific endpoints prevents integration with other IoT Platforms that have well-defined interfaces and protocols. A direct consequence of this decision is implementing a specific adapter for bridging the generic WoT interface to specific ones, such as the WoT-FIWARE Adapter. Although the evaluation analysis demonstrated the effectiveness and validated the correct operations of our software, the need for an adapter may introduce a possible system failure point or a

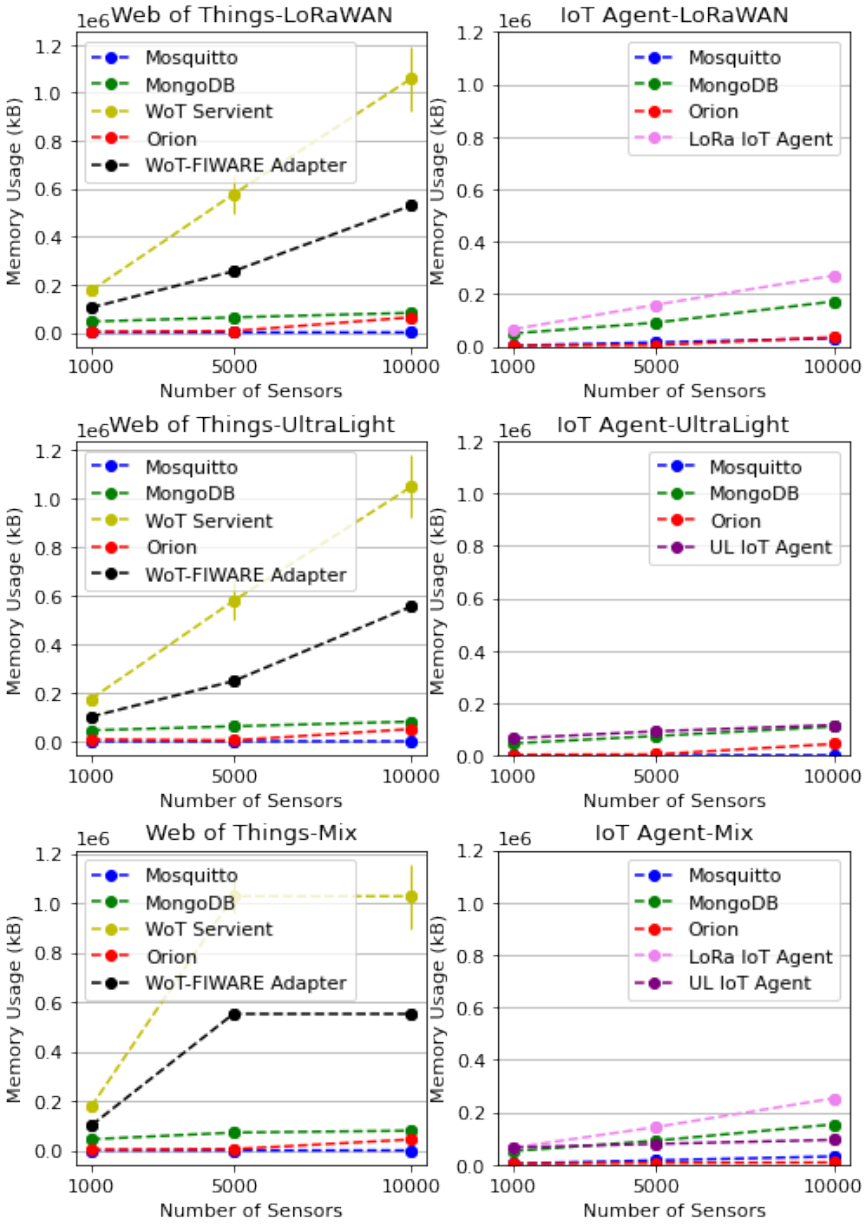


Fig. 3.17 Experimental CPU Usage

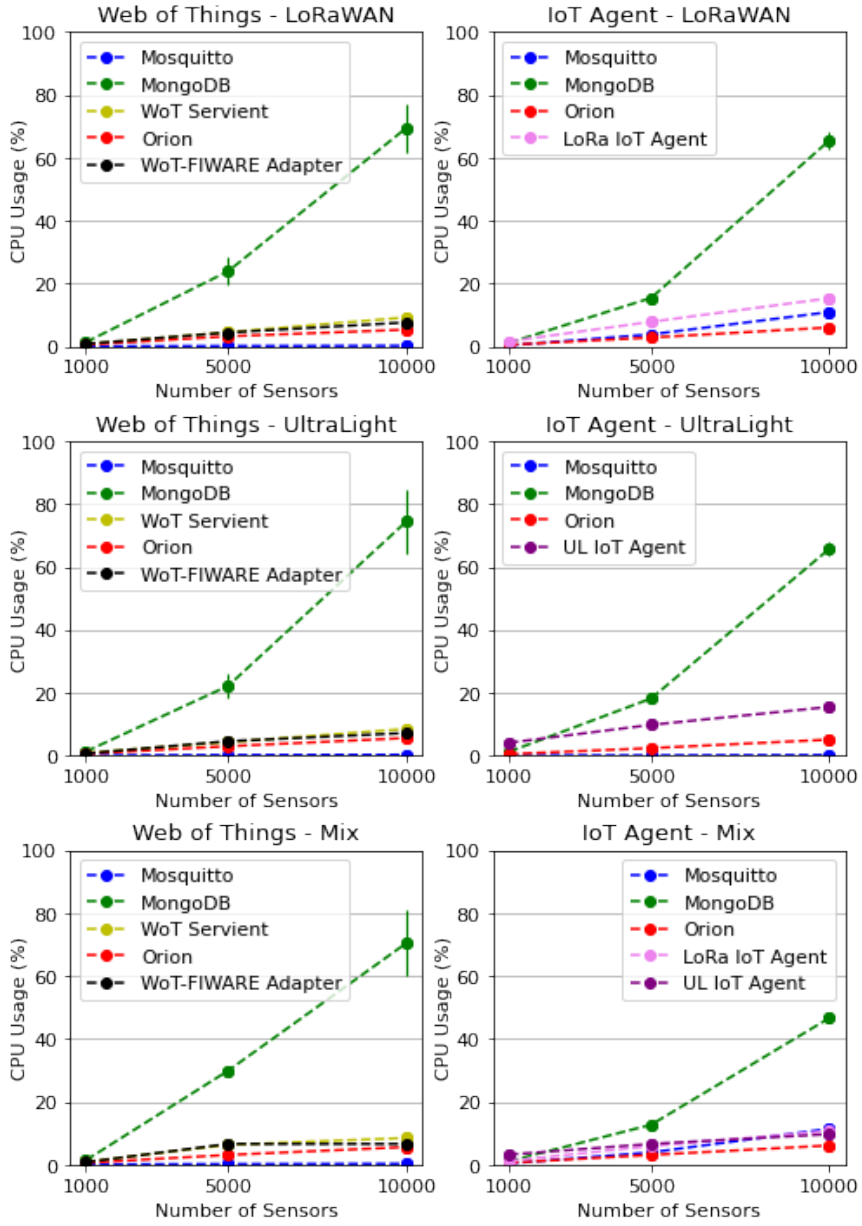


Fig. 3.18 Experimental Memory Usage

bottleneck. Similar concerns about the usage of the adapter can be raised from a software architectural perspective. Nonetheless, in our scenario, interoperability is not guaranteed by WoT but by the WoT-FIWARE Adapter. Thus, the conceptual requirements of WoT impacted its performance and required additional programming efforts compared to the IoT Agent solution: WoT enables interoperability only if all applications that communicate with it adopt the TD standard interfaces.

3.2.4 Seamless Integration of RESTful Web Services with the Web of Things

This Subsection details the proposed method to translate OAS into WoT TD and instantiated the generated TD as a WT proxy of the real application.

Syntaxical Translation of OAS to WoT TD

A formal description of the RESTful API is required to be eligible for a translation. We utilized the OAS to translate the API documentation in a TD, but our technique is extensible to other methods. The OAS defines a standard, language-agnostic JSON-based description interface to RESTful APIs (Swagger[24] adopted standard). We opted for OAS since it is widely adopted in commercial and academic applications. Additionally, it partially adopts the JSON schema[79] to structure its document, as does the WoT TD. As both support the JSON schema, the information described in the OAS regarding the request and response data models and their descriptions is also added to the WoT TD with minimal re-structuring. Far from providing the complete endpoints description, the OAS is necessary to obtain some critical values required by the WoT TD, namely: the WT name – the former OAS title –and the server that hosts the web service described by the OAS. The server address is essential to instantiate the WT proxy since it needs to know the URL to forward the requests.

The conversion process of the RESTful interface in a WoT TD is challenging since there is no exact match between the RESTful architecture and the WoT affordances. Further, the W3C TD[80] is decoupled from the underlying network protocols. However, both are virtual interfaces that allow users to access and manage resources. Hence, some similarities can be found:

- The `GET` method requests a representation of the specified resource, as the reading a WoT property affordance;

[24] https://swagger.io/specification/

Table 3.5 Correspondent WoT affordances of HTTP methods

HTTP Method	WoT Affordance
GET	readable property or readable and writable property
POST	action
PUT	writable property
PATCH	writable property
DELETE	action

- The `PUT` method aims to replace a specified resource, and the PATCH method to partially update the resource, similarly as writing to a WoT property;
- The `POST` method request and the WoT action both submit data to a specified resource;
- The `DELETE` method represents removing a specific resource, thus invoking an action.

For instance, an endpoint that implements both PUT and POST methods are translated into two different affordances – one action and one writable property. The naming of the converted affordance is composed by the `HTTP method-endpoint name` – e.g., `HTTP GET /weather` → `get-weather` affordance. Table 3.5 summarises the WoT correspondent of each translatable HTTP method.

Parameters are a key aspect of RESTful web interfaces that are strongly tied to the HTTP protocol. Several HTTP endpoints implement parameters as optional or required fields. Four types of parameters are supported by HTTP APIs: path parameters (e.g. /device/{id}), query parameters (e.g. /weather?city=Bologna), header parameters (e.g. X-MyHeader:Value) and cookie parameters. As the TD is independent of the underlying network protocol, the WoT does not support HTTP-specific parameters. Hence, we include the parameters in the request body, as an object where each of its keys is a parameter containing at least two keys: `in` that can assume the value of `(query|header|path|cookie)`, and `required`, a Boolean attribute. GET endpoints that have parameters are translated as a readable *and* writable property.

Unfortunately, not all features of RESTful services can be easily translated into a WoT equivalent since there are some mismatches between generic REST interfaces and the W3C WoT interface [1]. The hierarchical tree structure of paths in a REST API does not have an equivalent in WoT. Hence, all endpoints are mapped as a plain affordance – *e.g.* a GET endpoint as `/sensor/moisture` is translated to a `get-sensor.moisture` WoT property.

WT Proxy Instantiation

The translated TD is instantiated in a *servient* as a WT that acts as a proxy of the actual service. When a property is queried or an action is evoked, the WT makes the correspondent HTTP request to the service – it reassembles the URI from its affordance title and adds parameters when present. Next, it forwards the server reply in a WoT-understandable way. The communication only involves the instantiated WT and the proxied service. Consequently, the interactions of the instantiated WTs are entirely decoupled from the translation process.

C3PO: A Tool to instantiate RESTful services into WT

C3PO (**C**onverter of O**P**en A**P**I S**P**ecification to WoT **O**bjects)[25] is an open-source application developed in JavaScript using the NodeJS v10 engine. The `node-wot`[26] framework supports the creation of WTs. The tool is lightweight virtualized as a Docker container and its container image is publicly available at DockerHub[27].

Through C3PO, users can instantiate WT proxies of *any* web service that has an OAS – both OAS version 2 and 3. The tool provides two main features: (*i*) *Convert OAS to TD*: Users provide an OAS and our tool replies with the correspondent conversion to WoT TD (this increases the tool flexibility, as it enables the translated TD to be used in other implementations not directly tied with C3PO); (*ii*) *Instantiate WT proxies of Web Applications*: C3PO instantiates a WT proxy from the web service described through an OAS and returns the URL of the WT proxy.

Table 3.7 presents a complete list of all C3PO API endpoints and their description. Figure 3.19 illustrates the operation of C3PO, specifically the service interactions performed by the tool when it receives a request on /deployWoT/url endpoint. The numbered steps depicted in Figure 3.19 represent:

1. A user performs a request on C3PO API with a URL as the input;
2. CP3O fetches the OAS from the web service API at the given URL;
3. The tool checks if the retrieved description complies with the OAS specification. C3PO returns the user a report of the errors found in case of a mismatch. Then, it converts the OAS into a WoT TD and deploys a WT proxy of the RESTful web service;
4. The user can connect their sensors and WoT-based applications with the web service API through the WT proxy.

[25] https://github.com/UniBO-PRISMLab/c3po
[26] https://github.com/eclipse/thingweb.node-wot/
[27] https://hub.docker.com/repository/docker/ivanzy/c3po

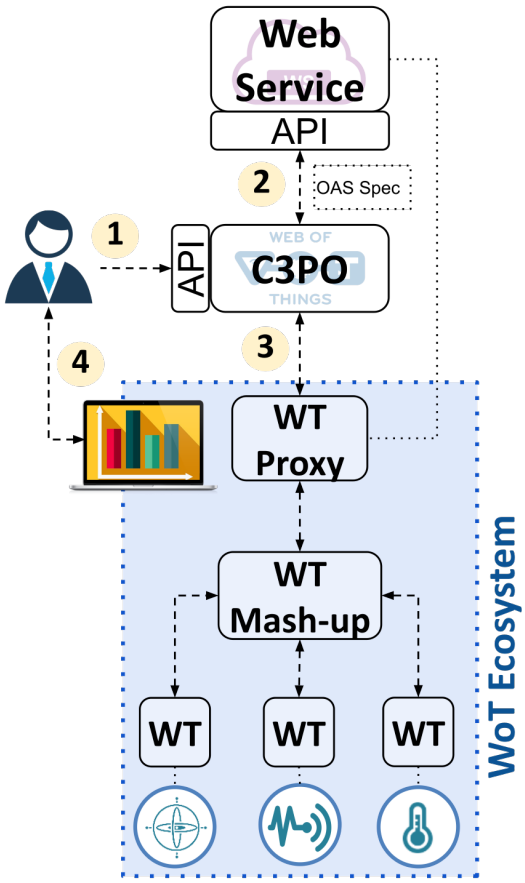


Fig. 3.19 C3PO process of deploying proxy WTs dataflow

C3PO has an open-source[28] graphic interface exposed on the Internet at https://wot-translator.iot-prism-lab.cs.unibo.it/. The GUI allows users to translate their OAS to TD easily and rapidly, without the need to install or deploy any software. Users need to provide an OAS in the left-side editor, and the WoT TD correspondent appears on the right side of the screen, as depicted by Figure 3.22.

Performance Analysis

Our proposed conversion method of RESTful services to WTs is based on instantiating a WT proxy of the real application, inherently adding a latency overhead to the overall system. Many IoT-based systems have strict delay restrictions, and the addition of another processing layer – i.e., resulting in a latency increase – must be carefully considered. Further, IoT real-world deployments are characterized by their complexity, since they are usually

[28]https://github.com/UniBO-PRISMLab/wot-translator-front

Table 3.6 C3PO's RESTful API endpoints

Name	Method	Description
/translateOpenApi	POST	receives an OAS and returns its conversion to a WoT TD
/translateOpenApi/url	POST	receives an URL that hosts a OAS and returns its conversion to a WoT TD
/deployWoT	POST	receives an OAS and instantiates a WT proxy of the service
/deployWoT/url	POST	receives an URL that hosts a OAS and instantiates a WT proxy of the service
/health	GET	returns the status of the service and the current up-time
/api-docs	GET	C3PO Swagger GUI interface
/openapi	GET	returns C3PO OAS specification in JSON

composed of numerous IoT devices transmitting data continuously. Hence, applications need to be scalable to suit such constraint-driven environments.

Based on those considerations, we conducted a performance analysis of our conversion method and its implementation (i.e., the C3PO tool) with two goals: (*i*) demonstrating the scalability of our application, even in high workloads; (*ii*) quantifying the latency impact imposed by our solution.

For supporting the experiments, we developed the Professor[29], an open-source tool for synthetically generating workload. The Professor performs HTTP requests to previously configured endpoints. We assume that the arrival process of HTTP requests occurs according to Poisson distribution. Consequently, the interval between requests follows the exponential distribution, and users can configure the λ variable – i.e., the number of requests per second.

In the experiments, we assume that the distribution of HTTP methods in the requests follows the occurrence of HTTP operations in commercial RESTful APIs, as mapped by [81]. Hence, the requests in the experiments were: 46% GET, 31% POST, 10% DELETE, 7.3% PUT, and 5.7% PATCH.

The experiments were performed in two different servers in the same network – hence, the network delay is negligible. We deployed the C3PO and the Professor in one server, and in the other, we deployed a generic RESTful web service that exposes its OAS in one of its endpoints. C3PO converted the OAS to a WT TD and instantiated a proxy of the generic service. Then, we performed experiments in two different scenarios: (*i*) *Scenario A*: the Professor performed request directly to the generic web service API; (*ii*) *Scenario B*: the Professor made requests to the WT proxy that forwards the queries to the generic web server.

[29]https://github.com/ivanzy/professor

The described experimental environment is illustrated by Figure 3.20. All the applications utilized in the experiments were lightweight virtualized as Docker containers. The maximum computation resources available to each container were defined to mimic an AWS t2.medium instance (2 vCPU and 4GB of RAM).

The generic web server[30] utilized in the experiment was developed using NodeJS, and it is also open-source to assure that experiment reproducibility. The server has a single endpoint – in addition to the one that exposes its OAS –, in which we implemented five different HTTP operations: GET, POST, DELETE, PUT, and PATCH. The server does not perform any computation, it just replies to requests with a successful status code. In case the request has an input, the server replies with the request body as payload and the successful status code.

We performed experiments under three different workloads – 25 requests/s, 50 requests/s, and 100 requests/s – in both scenarios. Each experiment was replicated 30 times, and each replication lasted 3 minutes with 6s of cool-down between replications, totalling 9.3 hours of experiment execution. The asymptotic confidence intervals were computed at the level of 99%.

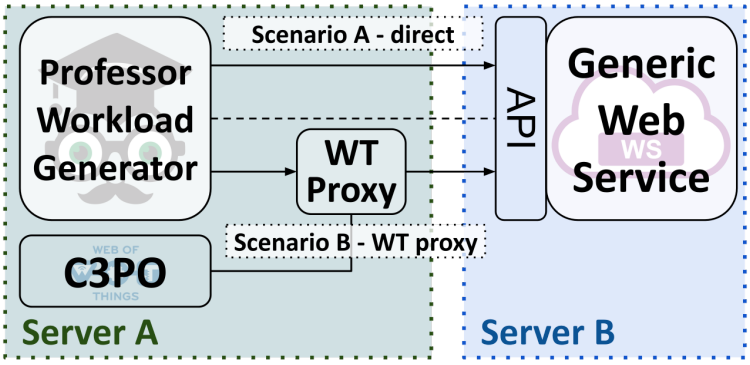


Fig. 3.20 Experiment Environment

Figure 3.21 summarizes the results of the performance evaluation, depicting the total experience delay for both evaluated scenarios in low, medium, and high workloads. The results showcase that both scenarios are scalable, as the workload increase did not generate a correspondent rise in the delay. The significantly low delay time in both scenarios can be explained by three factors: (*i*) the web server did not perform any computation; (*ii*) the server and the WT proxy were implemented in NodeJS[31], an engine designed to build scalable network applications; (*iii*) the two servers were in the same local network and connected through a high-speed link.

[30]https://github.com/UniBO-PRISMLab/mockApi
[31]https://nodejs.org/en/about/

The experiments unveiled that the additional delay imposed by querying applications through a WT proxy is approximately 0.4 ms – even in high workloads. The average network latency and operation time of real applications is much more than the added amount by the WT proxy. Therefore, **the overhead impact by adding a WT proxy layer is negligible to the system's overall performance**.

Fig. 3.21 Latency Results

3.3 Bridging Device to System Perspective

Referring to the IoT interoperability perspectives Figure 3.1, this Section describes our contributions to bridge the device perspective to the system perspective.

To achieve this goal, we propose a tool that enables a two-way translation between a WoT ecosystem and a System-of-Systems composed of well-described Web services. In detail, we present the following contributions:

- We propose and implement a middleware, namely, WoT-Arrowhead Enabler (WAE), capable of discovering and converting Arrowhead services into WTs and vice-versa. Thus, seamlessly connecting both ecosystems. For this aim, we leverage C3PO presented in Section 3.2.4 to convert Arrowhead services into WoT TDs and deploy those as fully functional WTs.

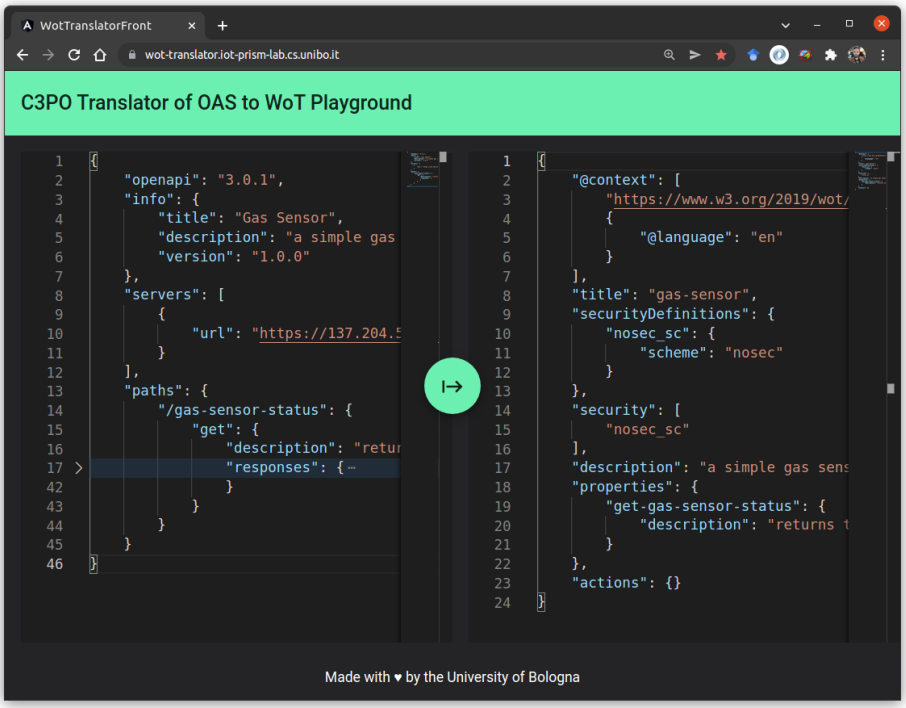


Fig. 3.22 A screenshot of the C3PO interface.

- We validate the proposed solution through a series of performance analysis experiments that enlighten the scalability of the application when tested in a close-to-real environment under high workloads.
- We implemented a rule-based engine that bridges different systems thought the Arrowhead framework and the WAE tool. It reasons by evaluating the values of WTs which actions within a legacy system.

3.3.1 Architectural Design

Not all systems can communicate directly with WTs, due to the strict WoT interfaces defined by the W3C [1]. Further, applications may not know the location of those WTs, unless implementing a discovery service. The Arrowhead SR solves both problems. It can expose the location and interface – as a REST API – of WTs. Thus, our application first automatically registers WTs as Arrowhead services; a detailed explanation of those interactions can be found at [82]. On the other hand, the Arrowhead ecosystem encompasses different services for several purposes and application domains. Hence, it will be a significant advantage for WoT-based applications to interact with those services. This feature enables seamless

integration of both ecosystems, i.e., from WoT to Arrowhead and vice-versa, to reduce the fragmentation issue among interoperability solutions previously mentioned.

Therefore, we propose the WAE, an application that spawns a WT proxy for each Arrowhead service. In this manner, WoT applications can interact with a variety of applications that do not follow the W3C standard architecture and interfaces [1]. In a typical Arrowhead implementation, there are numerous services registered onto the SR. Our proposal does not aim to convert all services to individual WTs since it would generate unnecessary computational resource usage. Instead, WAE monitors an array of services using a specific identifier. Whenever a new service is registered with the monitored identifier, our solution automatically detects it and attempts to convert it into a new WT. The instantiation of a WT into a service proxy leverages the C3PO, which convert Arrowhead services interfaces into TDs and deploy them as WTs. Clearly, the approach can be extended to other methods, provided that appropriate translators – other than C3PO – are developed. All WTs created by WAE are automatically registered in a TDD (e.g., ZION), ensuring that the WTs can be discovered and managed within the WoT ecosystem.

3.3.2 Service Interaction

This subsection details the interactions between the software modules from an architectural standpoint to enable the two-sided integration: (*i*) the automatic discovery of new WT and their registration in Arrowhead SR, and (*ii*) the automatic discovery and conversion of Arrowhead services into WTs.

WT Discovery and Registration in Arrowhead

Figure 3.23 depicts the service interactions performed by the WAE to discover new WTs and register them in the Arrowhead SR. The WAE application periodically queries the TDD, monitoring if a new WT was created. Figure 3.23 illustrates this 3-step process:

1. The WAE retrieves the list of all current WTs in the TDD;
2. The WAE checks if each WT is registered and up-to-date in the Arrowhead SR. Hence, the WAE issues a GET request in the Arrowhead SR API with the metadata information for each WT. An empty reply means that the WT is not registered. Further, the WAE compares the TD of the WT with the one register in the Arrowhead. If any difference is detected, there is the need to update the WT in the Arrowhead.
3. The WAE registers or updates the WTs identified in the previous step.

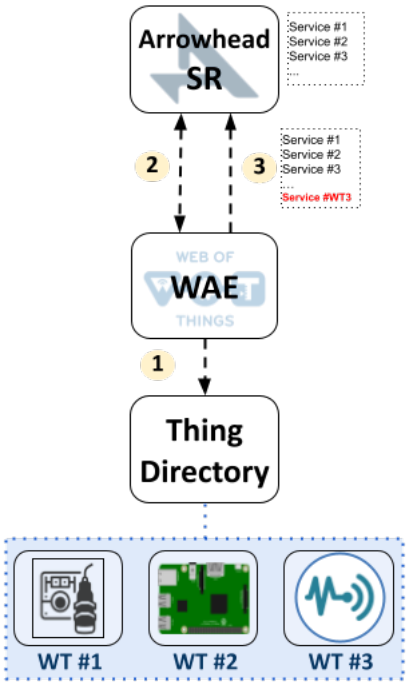


Fig. 3.23 Discovery and registration of WoT in the Arrowhead SR

Discovery and Conversion of Arrowhead Services into Web Things

Figure 3.24 depicts the service interactions performed by the WAE to filter and convert Arrowhead services into WTs. To this aim, the WAE periodically queries the Arrowhead SR and checks if a new service was created matching the service names it is currently monitoring. If so, the WAE instantiates a WT that acts as a proxy of that service. The detailed steps illustrated in Figure 3.24 are:

1. The user specifies one or more service names in a JSON format via the WAE API. Then, the WAE starts monitoring all services with such names.

2. The WAE obtains all services in Arrowhead SR and filters those that match the service names currently being monitored. Next, it checks if the filtered services have not been already deployed as WTs.

3. If the WAE identifies one or more applications deployed as WT, it gets the application OAS specification and converts it to a TD, applying the translation rules previously mentioned in Section 3.3.1.

4. The WAE instantiates a WT that acts as a proxy of the real service with the converted TD.

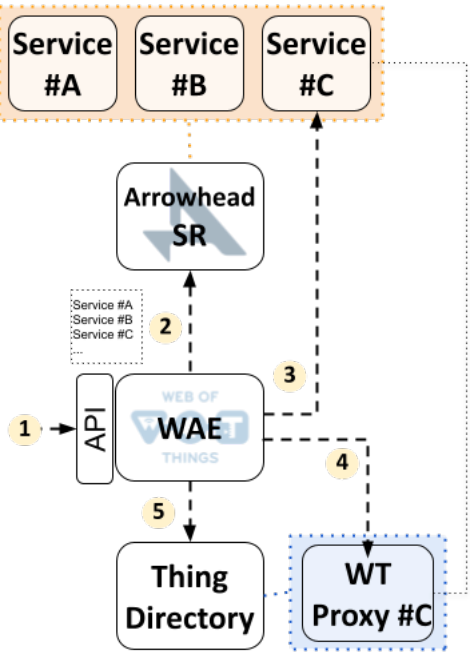


Fig. 3.24 Conversion of Arrowhead services into Web Things.

5. The WAE registers the new WT in the TDD.

WAE is an open-source application (available at [83]) developed in JavaScript using the NodeJS v10 engine. Table 3.7 presents a complete list of WAE API endpoints and their descriptions.

Table 3.7 WAE's RESTful API endpoints

Name	Method	Description
/arrowhead	GET	returns metadata regarding the Arrowhead polling for the conversion of Arrowhead services to WTs
	POST	adds a new serviceName to be monitored and translated as a Web Thing
/wotRepository	GET	returns metadata regarding the Thing Repository polling for the discovery and registration of WTs to Arrowhead
/management	GET	returns the metadata in both /arrowhead and /wotRepository endpoints in a single object
/health	GET	returns the status of the service and the current uptime
/api-docs	GET	Swagger GUI interface of OAS specification
/openapi	GET	returns WAE OAS specification in JSON

3.3.3 Performance Analysis

We conducted a performance analysis study with a twofold scope: (*i*) to validate WAE conversion of Arrowhead Services to WTs; (*ii*) to investigate the application's scalability in scenarios in which thousands of services need to be translated and instantiated.

Data Analysis: REST API Statistical Inference and OAS Generation

To run and control the experiments, we developed the OpenAPIGenerator [84], an open-source application for generating a configurable number of random OAS and exposing them in predetermined URIs. The OpenAPIGenerator also registers each generated OAS in the Arrowhead SR as a service.

A synthetic OAS needs to have size and complexity mimicking a real-world set of service APIs to approximate the experiments to a real scenario. Hence, we made statistical inferences in the public directory of REST API definitions available at APIs.guru[32] in OAS format. The APIs.guru directory filters out private and non-reliable APIs, thus consists of public, persistent, and helpful APIs –i.e., that provide useful functions not only for its owner. The dataset is composed of 2,283 different APIs, resulting in 52,203 endpoints and 77,171 methods.

Figure 3.25 depicts the occurrence percentage of each operation, i.e., HTTP method, in the directory. We filter operations that are not IANA-valid HTTP methods [85] – specific to a particular domain or company – and methods that represent less than 0.5% of the total. The histograms in Figure 3.26 depict the probability distributions of API Endpoints and GET and POST methods. Both distributions are similar and follow a long tail behavior. The x-axis in both histograms was limited to 100 to improve the graph visualization, encompassing 96.2% of the API Endpoints data and 95.6% of the GET and POST method data. Thus, we utilize the dataset to create an empirical probability distribution of the occurrence of those operations. The OpenAPIGenerator uses such distribution to generate OAS in the experiments synthetically.

Experimental Design

In a single server, we instantiated the three services utilized in the experiments: the WAE, the OpenAPIGenerator, and the Arrowhead SR, each virtualized as a Docker container. Preliminary to each experiment, the OpenAPIGenerator creates a set of OAS, exposes them, and registers them in the Arrowhead SR. A subgroup of the services has the same *service name*. Each experiment consists of the WAE converting the OAS of that subgroup of

[32]https://apis.guru/

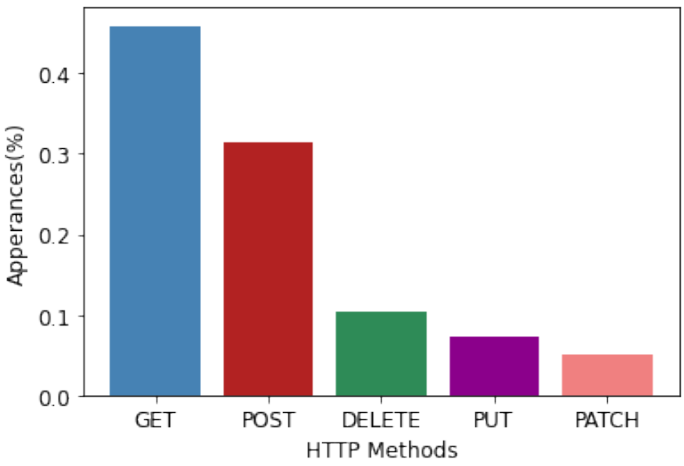


Fig. 3.25 Percentage of the valid HTTP Methods in the analysed dataset

Arrowhead services to TDs and deploying them as WTs. We performed four experiments, varying the number of services of the subgroup, starting from a single service, then increasing by a factor of 10 for each experiment (i.e., 1, 10, 100, 1,000). In every experiment, we record the time that the WAE takes to: (*i*) **fetch the OAS** once detects an Arrowhead service that needs to be converted; (*ii*) **translate** the OAS to a TD; (*iii*) **deploy** the service as a WT.

We set the initial time reference for all the recorded metrics as the last Arrowhead SR response time - the timestamp of the last poll operation performed by WAE.

Results

Figure 4.4 summarizes the key results of the performance analysis, depicting the processing time for all recorded metrics in the different evaluated workloads. The *y*-axis is expressed on a logarithmic scale. Overall, the experiments validated the WAE conversion of Arrowhead services to WTs and showcased that it can scale to convert a significant amount of services in a suitable time. The WAE can convert a batch of 1,000 services to WT in less than 16s, and, for a single service, it takes less than 50ms. The step taking more processing time in all workloads is the deployment of a new WT. This behavior is expected, especially for WTs that comply with the specifications of the W3C standard, thus needing to bind different protocols and support complex interactions. The fastest process is the translation, as it parses the OAS and maps it to another format via plain string manipulations. The process of fetching the OAS can potentially take more time in a real environment due to network latency issues and real application times.

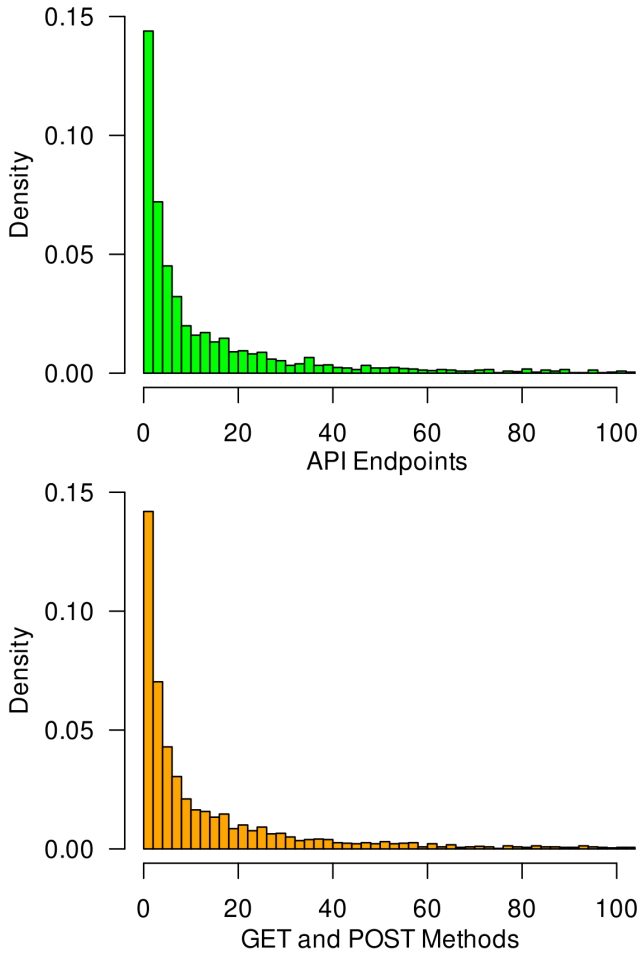


Fig. 3.26 Histograms of API Endpoints and GET and POST methods

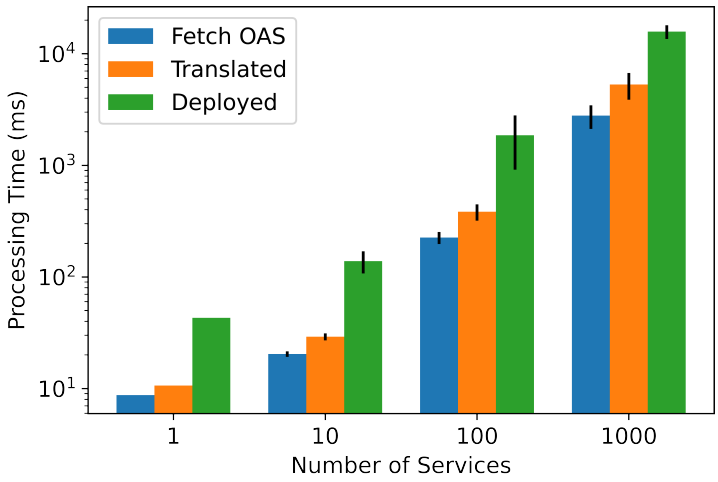


Fig. 3.27 Experimental Processing Times

A WoT and Arrowhead Closed-Loop Automation

We propose a system that exploits the WAE to enable closed-loop automation between legacy monitoring systems and actuation systems in an industrial scenario. In particular, we focus on the integration of an SHM scenario with a power control actuation subsystem. This system consists of three main blocks connected by the Arrowhead Tool Framework: (i) an SHM subsystem, (ii) a power control subsystem, and (iii) a control logic. The SHM subsystem can monitor the inertial and physical state of a building to assess its health or predict potential threats to the structure. The power control subsystem can monitor the power consumption of industrial equipment and turn it on and off remotely and on demand. We enable automatic discovery of system components through WAE. Through a set of well-defined rules, the control logic is able to automatically manage the state of the power control subsystem components based on the sensor values from the SHM subsystem.

The SHM subsystem is a monitoring environment that is divided into various layers, from edge to cloud. Sensors collect data regarding the monitored structure and produce data streams for the components above. These sensors are abstracted by the W3C WoT standard as WTs. The power control subsystem is used as an actuation subsystem within the SHM loop. The system consists of a legacy system, a custom module for translating WoT to legacy devices, and a TDD – e.g., ZION.). The legacy system is responsible for power monitoring and control through a set of custom power plugs that behave like smart circuit breakers. Real-time measurements of active and reactive power, voltage, and current are possible through power monitoring, while power control enables remote powering on or off of an appliance through a latching relay

The control logic depends on the WoT Arrowhead Drawbridge (WAD), an application that interfaces with other system components and interacts with them based on a set of well-defined rules. These rules dictate how the system should respond based on observed conditions in the monitored components. We propose a JSON-based syntax that utilizes WoT affordances to enable a standardized way of describing rules. These rules are based on monitoring a set of WTs `properties` that trigger a set of `actions`. Further details on the WAD can be found in [86].

Chapter 4

Data Management Layer: Caching in the IoT Edge-Cloud Continuum

This Chapter encompasses the contributions made in this thesis concerning the data management layer according to the reference architecture (Section 3.3.1). In particular, it addresses the RQ (ii), which is: *What strategies can be employed to enhance the efficient management of data within the edge-cloud continuum, emphasizing in enabling low latency while considering data freshness?*. To answer to this research question, we choose to concentrate on a specific facet of the data management layer, which is particularly relevant when considering the latency aspects in the IoT edge-cloud continuum: *edge caching*.

Section 4.1 presents a comprehensive review of the state-of-art in IoT edge caching and proposes a novel taxonomy focused on five orthogonal features of edge caching: placement, distance , strategy, metrics, and design. The literature review highlighted proactive edge caching [87–89] as a further improvement of edge caching. However, the existing literature has two noticeable shortcomings: (*i*) the solutions proposed in current literature are tailored to domains other than IoT; (*ii*) While caching strategies, models, and optimizations are the main focus of researchers, there is a lack of proposals for caching frameworks that facilitate the design and deployment of such strategies [90]. To address these challenges, Section 4.2 propose `CACHE-IT` (**C**onnected **A**rchitecture for **C**aching **HE**terogeneous **I**o**T**), a distributed framework for proactive edge caching in IoT scenarios. `CACHE-IT` decouples the caching strategy algorithm from the underlying architecture, enabling customization based on application-specific requirements. Subsequently, Section 4.3 explore how `CACHE-IT` can be effectively applied in privacy-constrained scenarios, where the transmission of sensitive user log data to the cloud is restricted. To address such issue, we leverage Federated Learning (FL) techniques.

4.1 Background

IoT and cloud computing have emerged as complementary paradigms and experienced joint, exponential growth. Most existing IoT deployments are cloud-based architectures with (big)-data produced by the smart devices and transferred towards remote infrastructures for storage and analytics. Nonetheless, several new applications demand more flexible resource allocation schemes. This is the case of next-generation time-sensitive applications envisaged *e.g.* in healthcare, industrial [91] and automotive scenarios [92] that rely on high automation and low-latency operations.

Edge computing is emerging as a viable solution to minimize the networking latency in IoT scenarios by pushing cloud services and data to the proximity of smart devices [93]. By leveraging the client proximity of edge devices combined with the vast resources of the cloud, the edge-cloud continuum paradigm [94] aims to optimize data processing and storage, providing a balance between latency-sensitive applications and resource-intensive cloud computations. However, Not all tasks executed in the cloud can be offloaded to the edge due to limited processing power and the lack of robustness of edge devices [95]. Further, real IoT systems commonly require to interface with third-party services that are not under system administrators' control [96]. In order to achieve better exploitation of cloud-edge continuum integration with IoT, edge caching constitutes a potential solution to satisfying latency constraints [22]. Storing frequently accessed data at the edge reduces redundant computation, which leads to cost savings in terms of power and cloud resources – e.g., serverless functions, cloud data transfer, and paid APIs.

This section discusses the state-of-the-art and the future directions of IoT edge caching. We propose a taxonomy and a consequential survey of the existing studies based on five orthogonal features of caching systems: placement, distance, strategy, metrics, and design. Next, we analyze five different domains concerning the proposed taxonomy. The rest of the section is structured as follows. Subsection 4.1.1 introduces the taxonomy and literature review of IoT edge caching systems. Subsection 4.1.2 discusses the existing edge caching deployments in different IoT scenarios. Finally, Subsection 4.1.3 concludes by comparing our caching framework to other approaches in literature.

4.1.1 IoT Edge Caching: Taxonomy and Review

Caching mechanisms are well investigated in the literature on networking systems. In [93], readers can find a broad review of the main building blocks of edge computing, including edge learning, offloading, and caching. In the following, we provide a concise review of the state of the art of IoT edge caching solutions based on five orthogonal features: cache

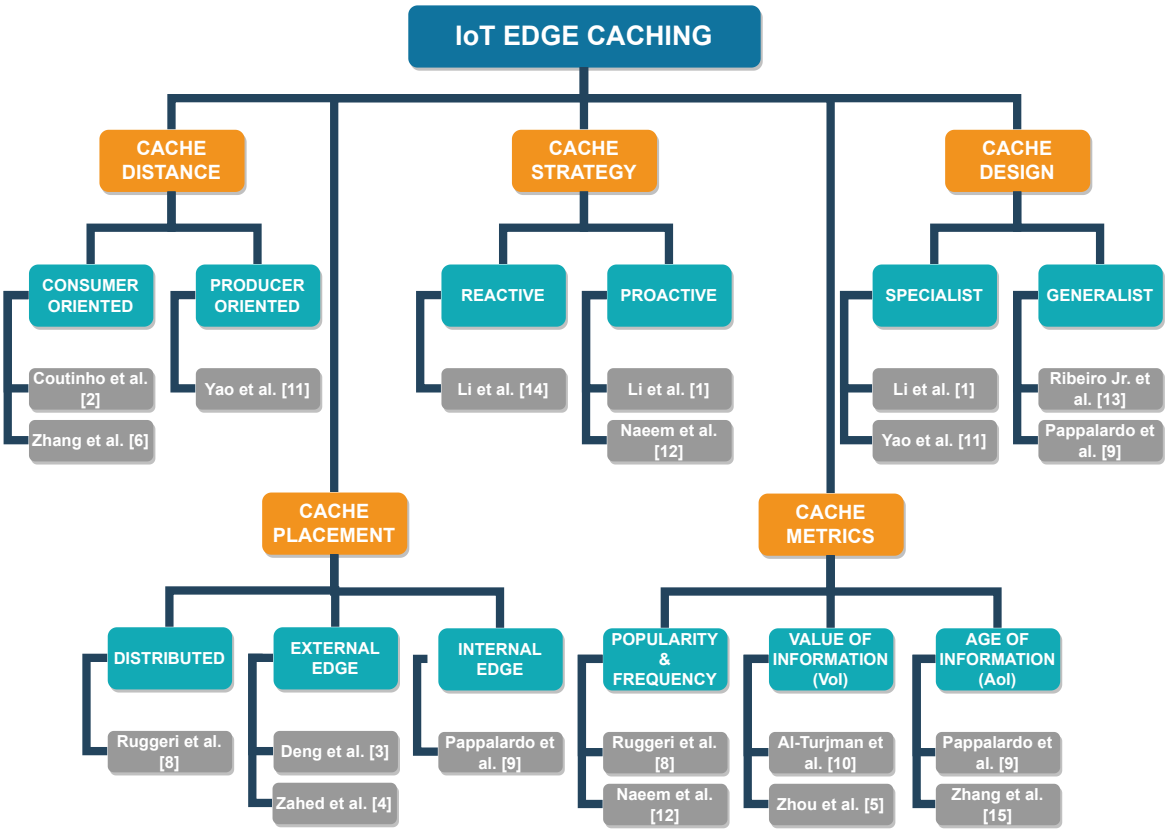


Fig. 4.1 The studies reviewed in this thesis, classified according to the features introduced in Subsection 4.1.1

.

placement, caching distance, caching strategy, caching metrics, and caching design. Figure 4.1 summarizes the classification of reviewed studies based on these features.

Cache Placement in the IoT Compute Continuum

The integration of edge and cloud computing solutions has led to the creation of an IoT compute continuum that spans from the connected devices to the public/private cloud and enables the seamless orchestration of services over it.

Although no clear definition of the components of the IoT compute continuum exists, many recent papers consider layered architecture involving smart devices and cloud services as terminal nodes and Base Stations along with fog nodes as intermediate layers. We outline the cache placement in the continuum between the cloud and the end-device as follow: *Cloud* $\leftrightarrow$ *External Edge* $\leftrightarrow$ *Internal Edge* $\leftrightarrow$ *End-Device*, as previous mentioned in Chapter 2. Edge caching mechanisms can be deployed on the smart *end-devices* directly or on intermediate edge nodes geographically and topologically close to them that can be either: *internal* to the local network – i.e., cluster heads, proxy nodes–; or *external* – i.e., BS. However, given the significant differences in computing capabilities, where to cache is crucial since it can impact both the communication latency and the amount of data cachable.

The presence of cached IoT data at the BSs can also drive the offloading of computational tasks. For instance, in [97], the authors investigate a joint caching/computation optimization problem to select the tasks to execute at each BS and the IoT data to cache so that the associated energy cost is minimized. Fog-layer caching systems work as data proxies, receiving requests from the applications and deciding whether to fetch the last update from the sensor or to send the cached data [98]. Device-layer caching systems exploit local computational/storage resources and fetch contents from other peers through the cooperative approaches reviewed in [99]. However, while these solutions have been widely investigated in the case of edge servers and smartphones in MEC scenarios [93], they can be unfeasible for most resource-constrained smart devices. Finally, recent studies investigate the possibility of dynamically *distributing cached* data over multiple layers of the continuum so that the associated distance between the data source and the requesting devices is minimized. An example is the caching orchestration mechanism in [100], which is deployed on top of a software-defined networking controller and is in charge of distributing the data to cache over multiple edge nodes in order to limit the average retrieval delay.

Caching Distance: Consumer-oriented vs Producer-oriented

We can further classify IoT edge caching systems based on the distance of the cache from the data consumer and the data producer. More specifically, we distinguish between consumer-oriented and producer-oriented solutions. In the first case (consumer-oriented), data is cached in the edge and utilized by the devices and mash-up applications nearby. This is the case of IoT scenarios where smart devices request remote data or API services. Here, the caching system stores the cloud response to minimize the latency of subsequent requests. Vice versa, in producer-oriented caching, data produced from smart devices are cached at nearby nodes of the continuum, with the applications fetching data from the edge cache rather than from the devices directly. This approach may reduce the number of activation and hence the energy consumption of battery-constrained devices, though impacting the freshness of information retrieved. In [101], the producer-oriented cache is embedded within the gateway. When the user request arrives, the requested data can be fetched directly from the gateway cache if the requested data is cached. Otherwise, the corresponding IoT sensor is awakened and recharged via energy transmitters. Producer-oriented caching is suitable for specific scenarios where sensing features do not change rapidly and do not need constant monitoring.

Caching Strategy: Proactive vs Reactive

Existing IoT edge caching systems mainly fall into two categories of data retrieval, i.e., reactive or proactive strategies. Reactive caching solutions update their storage when a new request-response from the cloud or the smart device originates in the system. In [102], reactive in-network caching strategies are proposed for Named Data Networking (NDN) IoT scenarios. Here, contents are selected based on their popularity, i.e., how many times they have been fetched; in a second phase, contents are distributed over nodes further selected based on their number of connections. Vice versa, proactive caching strategies determine which contents should be cached *before* they are effectively requested; hence they are based on the prediction of future requests. For instance, in [91], an industrial IoT scenario is considered with edge servers caching big data on behalf of mobile clients. A proactive caching strategy based on the distributed Hungarian algorithm is proposed to fetch data in advance by taking into account the users' mobility patterns and the edge servers' computational capacities.

Caching Metrics

Several metrics have been proposed for content fetching and replacement in networking caching systems. *Content popularity* measured in terms of requests is used, among others, in

[100] and [102]. While such a metric can be suitable for consumer-oriented caching, it is unsuitable for IoT scenarios, as pointed in Section I. For these reasons, other adopted metrics proposed in the literature are the *Age of Information (AoI)* and the *Value of Information (VoI)*. The first metric reflects the freshness of cached data and can be measured as the time elapsed since the data was fetched from the IoT data source. In [98] the authors propose an analytical formulation of the AoI and investigate its relationship with the network overhead of the smart device, which sends its periodic updates to a proxy server. The VoI has a less precise characterization and is interpreted in many studies as a generic function reflecting the importance of each cached data. In [103] the authors propose a caching replacement strategy for wireless body area network where the VoI function combines three parameters related to delay-based content, age-based content, and demand-based content. Therefore, through a single VoI metric, different caching services for multiple WBAN applications can be supported.

Caching Design: Generalist vs. Specialist

Despite the generality of the IoT paradigm, data belonging to different IoT applications are quite different in acquisition mechanisms, contents, and scopes. Such differences may reflect in the caching operations leading to customization and specialization driven by the use case. We can classify the existing IoT-based edge caching solutions as generalist or specialist approaches. The first category includes the studies proposing caching solutions for IoT scenarios without considering the data content and data source characteristics. Most of the works reviewed so far fall within this category. Machine Learning (ML) techniques for proactive caching, e.g., the one reviewed in smart agriculture scenarios [104], can be considered generalist in the way traditional ML algorithms are used. However, the trained model is dependent on the local data. Conversely, specialist solutions are tailored to a specific domain and typically feature higher performances, while they may underperform in other conditions. In [91] the authors include their proactive caching strategy features of industrial mobile nodes. Further, producer-oriented caching solutions as [98] and [101] have unique features that are only suitable for low-power sensor network scenarios.

4.1.2 IoT Edge Caching Use Cases

This section presents several use cases in which IoT edge caching plays a crucial role in the applications' outcomes.

Industry 4.0

The application of IoT concepts in Industry 4.0 (IIoT) is changing how industrial production occurs. Industrial appliances in smart manufacturing highly interact with their environment. Industrial processes demand a continuous and low-latency exchange of data that needs to be processed by different components of the IIoT ecosystem. Due to the intrinsic characteristics of the latter, the deployed smart devices can be both data generators – e.g., sensors – and data consumers – e.g., actuators. To reduce the amount of exchanged data, IIoT edge caching solutions must be deployed close to the data origin (producer-oriented) to reduce data acquisition and close to the data users (consumer-oriented) to make task execution faster. In time-critical industrial scenarios, such as motion control in closed loop systems and massive wireless sensor networks, the applications necessitate of time constraints and hard deadlines for detection and response to events [92]. Legacy solutions based on caching systems for cloud services cannot reduce the burden of transferring data back and forth from the local and the remote locations. Moreover, existing IoT edge caching solutions in Industry 4.0 mainly rely on fixed external edge devices (usually placed in 5G network small cells) that fail in achieving the real-time requirements of IIoT systems. For this reason, and due to the increasing complexity of the environments and the privacy concerns, a more pervasive internal edge caching deployment must be taken into account [92]. Proactive caching systems can fit the characteristics of IIoT applications well due to the factory production process that must follow a specific path to complete a task, such as a workpiece conveyance along the plant [91]. Hence, using IoT edge cache nodes along the assembly line permits proactive caching to fulfil the time-critical applications' requests at the IoT terminal devices.

Smart Grids

Nowadays, Smart Grids (SGs) are composed of sophisticated tools and devices used to control and monitor the power system to connect power stations (producers) and end-users (consumers). Moreover, thanks to the advances in IoT technologies, a social movement has been promoting more participative energy processes that redefine citizens' roles as energy prosumers, i.e., both consumers and producers/co-owners of energy facilities. Given the complexity in the SG management, the legacy cloud approach may no longer be able to fulfill the system requirements, like latency, bandwidth, reliability, and scalability [105]. Due to prosumers paradigm shift, where the energy trade and exchange in the SG moves from centralized management to a distributed one, the inclusion of service caching in the edge/fog paradigm shows attractive advantages in handling the massive amount of data traffic inside the SGs [106]. Indeed, edge devices frequently need to collect data from sensors

and cloud services – e.g., weather forecast and energy prices for intelligent monitoring tasks. If the external services are not adequately cached, the latency for the task execution increases drastically due to the continuous requests to remote servers. In this case, a proactive caching strategy is pivotal to an efficient data distribution system. IoT edge caching is a key feature in SGs for time-sensitive applications like energy load balancing for smart meters and micro-grids. Depending on energy demand, availability, and price, the micro-grid devices are able to automatically select alternative energy sources [105]. Additionally, due to the large amount of data coming from the SG sensors and the limited caching capacity of each edge device, a specialized caching system must be considered for data distribution and service deployment design.

Smart Agriculture

IoT applications for smart agriculture may face severe networking challenges due to the lack of high-speed or reliable Internet connection. The caching system fulfills a different role in those environments than in the previous domains. The cache can store the acquired sensory data and transmit them opportunistically whenever the network connection is re-established. The caching system should support both consumer-oriented and producer-oriented placements due to sensor devices and actuators. Furthermore, it needs to face the limited storage capacity that characterizes the network devices – terminal nodes and internal edge servers – used in smart agriculture. Several solutions can be used to reduce the data dimensionality, from ML approaches to optimization techniques based on AoI and VoI metrics. Also, a specialist cache design can assist the development of ad-hoc solutions better exploiting the characteristics of smart agriculture environments [104].

Vehicular Networks

Vehicular networks are often characterized by the high speed and the autonomous mobility of the network nodes. Hence, the data orchestration services must face the dynamicity of the network topology. In-ground vehicular networks, the caching of multimedia content delivery has been largely investigated in the last years due to the high demand for content delivery [107]. Since the main caching distance is on the consumer-oriented side, a significant challenge is constituted by choosing the best edge locations, which must be as close as possible to the requesting vehicles. However, in the case of IoT edge caching, the generated and requested data can derive both from bandwidth-consuming infotainment services (IS), such as media entertainments, as well as from time-critical driving-related context information services (CIS), such as sensory data for assisted/autonomous mobility or emergency applications.

The management of CIS data differs from the IS data due to the time-critical nature of the driving environment that imposes short delay for the context-related data delivery. [108]. In cache-enabled vehicular networks, the vehicles may become information producers and consumers of local data and, consequently, the cache distance can be both consumer-oriented and producer-oriented. In these applications, the cache placement for real-time data management and delivery can be an issue due to the high mobility of the vehicles. For the same reason, the AoI metric can be used as a key index for deciding content management and delivery strategies [108]. Moreover, a hierarchical caching system deployment – that exploits the presence of edge nodes on RSUs (Roadside Units) and cellular base stations, i.e. from end devices to the cloud – can support the operations of vehicular networks due to the high flexibility offered by the presence of different cache units [92].

Low Power Sensors

Several IoT applications rely on low-power devices with limited computing capabilities and constrained lifetime. Caching techniques are paramount to improving the system performance while enabling multiple applications on top. Most of the studies related to low-power IoT sensor networks assume a producer-oriented edge caching approach. Here, the acquired sensory data are stored at the internal edge layer to make them available to third entities, hence avoiding continuous requests to the sensors. As a result, the number of activation of the sensor devices can be reduced and their energy power consumption prolonged. The general placement of the caching system is close to the data source – i.e., on the network gateway[101]. Depending on the deployed vertical application, both proactive and reactive caching strategies can be applied, while a key factor is constituted by the storage capacity and computational power of the gateway which may manage multiple low-power sensors.

4.1.3 Frameworks for Proactive Edge Caching

In this subsection, we review the recent literature and highlight the differences concerning our work by focusing on the following aspects:

- The **goal** of the caching strategy, i.e., what the caching strategy aims to optimize.
- The overall caching **strategy**, focusing on the underlying scientific method adopted.
- Whether the caching solution is **domain-agnostic** or is particularly tied to one application, e.g., video streaming.

- Whether the caching solution is **infrastructure-agnostic** or is particularly tied to a certain physical deployment, e.g., 5G macro-cells.
- Whether the caching solution is **proactive**, i.e., it tries to cache the content before it is requested.
- Whether the caching solution is **cooperative**, meaning that the edge Cache Nodes exchange content or information for tuning their decisions.

We do not utilized the taxonomy created because it refers to the classification of caching solutions and it is not useful to compare *frameworks*. Indeed, the caching placement or the its distance are not relevant as features to be compared when analysing different architectures. Related information about each of the analyzed papers is collected and summarized in Table 4.1, while the rest of the section is dedicated to their description and the analysis of the gaps our work aims to tackle.

First, a substantial quantity of works proposed in the literature are bound to a definite application scenario or deployment. It is the example of [91], which deals specifically with industrial scenarios where mobile nodes need data from a central server and wander near wireless sensors. Here, the authors propose a centralized optimization algorithm where all the computation takes place in the cloud to proactively offload data to edge servers by predicting client mobility. Prediction of mobility patterns is also employed in [109], where proactive caching is used in urban scenarios to deliver content to vehicles promptly. In [110], authors cache and transcode video content on UAVs to alleviate the backhaul load through a heuristic model. Authors in [111] propose a deep learning model for proactive caching that predicts the content popularity of videos and music only. In contrast with our work, the mentioned approaches are vertical in one application context and lack adaptability.

One recurrent scenario for which a consistent number of works have been proposed is 5G cellular networks. In [112], authors overview the state-of-the-art and the current challenges and potential solutions for 5G edge caching. They also propose a caching framework that leverages blockchain transactions and non-negative matrix factorization. In [113], authors inspect cooperative micro-caching as a network function that should be embedded in 5G networks to minimize latency. A similar 5G scenario is analyzed in [92], where results aim to motivate the introduction of new caching policies to deal with high mobility nodes (e.g., vehicles). Again, in [114], authors focus on mobile content caching for 5G networks, tackling the problems of edge caching and radio resource allocation separately, using a deep reinforcement leaning for the first one and a branch & bound approximation algorithm for the second. The work in [88] proposes DeepCacheNet, a deep learning-based framework for proactive caching that uses stacked denoising autoencoders to classify content popularity and

instruct base stations to cache such content. In [115], the authors present a framework for proactive and cooperative edge caching that aims to cluster base stations to establish intra-cluster collaboration through a deep reinforcement learning technique. Caching-as-a-Service [116] is a caching virtualization framework along with the development of Cloud-based Radio Access Networks (C-RAN). It focuses on virtualizing the cache through Network Function Virtualization (NFV) and exploring the possibilities of detaching the software application from the underlying hardware. These approaches, unlike ours, are tied to the 5G infrastructure, requirements, and characteristics, which do not reflect many IoT systems.

Most edge caching works in literature present a similar core structure: they aim to optimize several parameters (e.g., the hit rate and the backhaul load) of a specific caching policy. Therefore, they propose a mathematical structure to model the system, an algorithmic or learning-based solution, and numerical results supported by simulations. Some solutions are numerical or probabilistic, such as Smart-Edge-CoCaCo [117], or [118], which proposes a cooperative caching mechanism for scenarios where edge nodes offload computing tasks to edge cloudlets. A D2D solution is adopted in [119], where authors consider data freshness and client mobility. However, most of the solutions in the literature rely on deep learning, as [87], where content popularity in the future is predicted by using bidirectional deep recurrent neural networks, or [120], which uses a distributed deep learning algorithm to predict the content demand by single users, or in [121], where deep reinforcement learning is adopted by each caching agent so that they learn to cooperate and share cached resources. Other related deep learning-based works use federated learning [122] or deep learning in conjunction with attention mechanisms [123]. In [124], authors present a very different solution: proactive caching is done locally in each user's equipment, and cache hit can occur locally or with direct D2D communication, eliminating the structural constraints enforced by cellular networks. With such a focus on the single caching strategy, very few efforts are dedicated to proposing an edge caching architecture for IoT scenarios capable of accommodating different caching policies. Furthermore, most of the solutions proposed are not complemented by a real implementation or guidelines of the software components.

It is also important to mention Information-Centric Networking (ICN), which, in the past decade, has established itself as an alternative to conventional TCP/IP networks. ICN, in particular in its guise of Named Data Networking (NDN), fulfills a content-centric design and a location-independent naming mechanism, giving this paradigm several advantages in the scope of mobility and efficiency. One powerful and native feature of ICN is its in-network caching in intermediate network routers [125]. The recent study in [126] presents an all-encompassing solution based on NDN, encompassing network topology, data freshness,

Table 4.1 Comparison of CACHE-IT with the works in literature

Paper	**Goal**	**Strategy**	**Domain**	**Infrastructure**	**Proactive**	**Cooperative**
[88]	Backhaul Load	Deep Learning	✓	✗	✓	✗
[87]	Hit Rate	Deep Learning	✗	✗	✓	✗
[117]	Delay	No strategy	✓	✓	✗	✗
[120]	Delay, Hit rate	Deep Learning	✓	✓	✓	✓
[118]	Response Time	Heuristic	✗	✓	✗	✓
[121]	Hit Rate	Deep R. Learning	✓	✓	✗	✓
[91]	Delay, Goodput	Optimization	✗	✗	✓	✗
[113]	Delay	Fuzzy inference	✓	✗	✗	✓
[92]	Delay	Statistical	✓	✗	✗	✗
[89]	Hit Rate	LSTM & Ensemble	✓	✗	✓	✗
[114]	Hit Rate & Delay	Deep R. Learning	✓	✗	✓	✓
[109]	Hit Rate & Delay	Probabilistic	✗	✗	✓	✗
[110]	Hit Rate & Delay	Heuristic	✗	✗	✗	✗
[124]	Hit Rate & Delay	LSTM	✓	✓	✓	✓
[122]	Hit Rate & Profit	Federated Learning	✓	✓	✓	✓
[123]	Traffic Load	Deep Learning	✗	✗	✓	✗
[126]	Hit Rate, Delay	Heuristic	✓	✗	✗	✗
[119]	User Utility	Probabilistic	✗	✗	✗	✓
[112]	Delay	Optimization	✓	✗	✓	✗
[115]	Hit Rate	Deep R. Learning	✓	✗	✓	✓
[111]	Hit Rate, Load	Deep Learning	✗	✗	✓	✗
[116]	Traffic Load	Greedy	✓	✗	✗	✓
CACHE-IT	Flexible	Flexible	✓	✓	✓	✓

and content popularity, thus attempting to design a common solution to all previous works on caching in ICN.

A final aspect to consider is modeling the pattern of requests from clients. A meaningful amount of the mentioned works assume time-invariant content popularity and primarily cater to human clients. This may differ if the system actors are IoT devices, such as sensors or robots. Furthermore, they disregard the system aspects of how to collect, share, store, and process client traces to obtain the popularity values of each resource. In [89], authors apply proactive caching to mobile edge networks at the base stations based on content popularity; however, they evaluate the method over a movie dataset, assuming to know the demographic information of the clients through the usage of cameras. In contrast to existing approaches, our work focuses on crucial aspects of IoT environments, including time-variant conditions and interoperability. Additionally, we evaluated `CACHE-IT` under different client behaviors, which was modeled based on the real behavior of agents in IoT scenarios.

4.1.4 Federated Learning support in Edge Caching

The evolution of IoT technologies and their requirements has prompted researchers to address the limitations of traditional caching techniques by moving the cache to the network edge. Alongside the developments in edge caching, there have been recent advancements in proactive edge caching, which focus on exploring the relationships between client and data to predict and preemptively store the next cached requests. Most solutions in the literature aim to optimize a given feature of the caching system, such as the cache hit rate, and minimize the backhaul load by proposing a specific caching strategy that leverages a given machine-learning algorithm. From those, we highlight [120], which uses a distributed deep learning algorithm to predict the content demand by single clients, and [87], which the future content popularity predicted using bidirectional deep recurrent neural networks. A hierarchical deep learning-based content caching strategy for mobile edge computing in IoT networks was proposed in [89]. However, they evaluate the method over a movie dataset, assuming to know the demographic information of the clients through the usage of cameras.

The initial attempt to combine edge computing with FL was proposed by Yu et al. [127] as a proactive caching technique to predict file popularity and cache them in advance. Another noteworthy contribution was made by Wang et al [128], as a framework to combine Deep Reinforcement Learning and FL to decide which resources to cache. However, they did not consider proactive caching. Relevant progress was made by Qiao et al. [129], who proposed an adaptive FL-based proactive content caching strategy for 5G edge networks while considering non-identically distributed (Non-IID) client data and constrained edge

resources. Other recent advances focus on specific domains, such as in intelligent Connect Vehicles (ICV)[130], self-driving cars[131] and video data[132]. They design and implement specific caching strategies considering domain-specific conditions – such as high-speed movement – are considered. To our knowledge, no FL solution considered the Age of Information of the cached content as an essential constraint.

Given the predominant emphasis on developing caching strategies, none of the listed studies proposes an edge caching architecture that allows for the easy modification or change of the caching policy. `CACHE-IT` [10] proposes a proactive edge caching architecture agnostic of the underlying caching strategy. This thesis extends this work by introducing the possibility of executing FL-based strategies and leveraging the edge node capabilities of computation.

4.2 `CACHE-IT`: Proactive Edge Caching in Heterogeneous IoT Scenarios

This section presents `CACHE-IT`, a proactive edge caching framework designed for IoT scenarios. `CACHE-IT` allows rapid deployment, modification, and replacement of the caching strategy. The framework feeds the history of clients requests to the caching strategy, handled as modular component, which is responsible to determined what and where to cache. `CACHE-IT` components are based on current Web technology standards, ensuring compatibility and easy integration with existing systems. The contributions of this section are:

- **Flexibility**: `CACHE-IT` allows the deployment of customized and flexible caching in the C2T continuum by decoupling the edge caching architecture from the caching strategy, enabling use-case-driven caching strategy customization. Consequently, it provides versatility in the technique utilized (e.g., Deep Learning, Heuristic) and the caching goal (e.g., minimize latency, minimize AoI). The framework allows handling different specific users or requests that impose specific constraints.

- **IoT oriented design**: we address interoperability issues of IoT by providing a dedicated device abstraction layer that seamlessly integrates heterogeneous IoT devices through a standard and well-defined interface. We tackle the dynamism of IoT environments regarding the volatile nature of sensors by incorporating mechanisms that adapt the caching strategy to reflect these changes. Finally, `CACHE-IT` components were designed to be distributed in the C2T continuum, considering edge nodes ranging from base stations to constrained computational devices.

- **Advanced Caching Mechanism**: CACHE-IT utilizes cooperative and proactive caching mechanisms for increased performance with no additional complexity. It optimizes the capacity of the system to share resources in a twofold manner. First, caching strategies are able to redirect specific Cache Node requests to query directly other edge nodes. Second, when a cache miss occurs, the Cache Node queries its nearest neighbors. We intentionally avoided mechanisms that might be bound to specific caching strategies to ensure CACHE-IT flexibility and adaptability.

- **Validation**: we performed extensive simulations varying numerous parameters to understand the impact of CACHE-IT features under different scenarios and client behaviors – modeled according to patterns commonly found in IoT scenarios.

In the remainder of this section, Subsection 4.2.1 introduces the CACHE-IT framework, providing a comprehensive overview of its architecture, while Subsection 4.2.2 details its operations. Subsection 4.2.3 describes CACHE-IT implementation. In Subsection 4.2.4, we evaluate the framework performance through large-scale simulations.

4.2.1 Architectural Design

CACHE-IT is an architectural framework for proactive edge caching in heterogeneous IoT scenarios that provides high customization and flexibility. Following, we describe the guidelines that support CACHE-IT design, then we detail its high-level architecture.

Designing Guidelines

Our architecture aims to provide proactive caching capabilities through a fully customized framework. Additionally, our solution aims to be deployed on top of operational IoT systems. Four general guidelines supported our architectural design:

1. **Interoperability**: we adopt standards-based approaches and open interface solutions that enable seamless communication between heterogeneous devices and data providers.

2. **Generality**: IoT systems are employed in a wide range of domains, each with unique requirements and characteristics. These domains include areas that are orthogonally different from each other, such as Industry 4.0 and smart agriculture. Specific scenarios may have their own particularities that must be addressed. The architecture should be domain-agnostic and capable of handling different constraints.

3. **Adoption**: The fundamental elements of the architecture must rely on established, real-world technologies that are widely adopted in the industry. Nonetheless, the architecture description must remain independent of the underlying technology

4. **Flexibility**: The architecture must be flexible regarding the caching deployment in the C2T continuum by decoupling it from the caching strategy regarding its goal, constraints, and requirements. The ability to easily switch between caching policies allows organizations to react rapidly to alterations in their requirements or constraints.

`CACHE-IT` High Level Architecture

`CACHE-IT` is a distributed microservice-oriented architecture; each software component is modular and independent. This design addresses scalability concerns by allowing multiple replicas of the same services to be instantiated simultaneously, effectively managing a high demand of requests to the framework. Figure 4.2 illustrates `CACHE-IT` high-level architecture, which comprises clients – at the bottom – that request data from providers – at the top. Requests are routed through the caching framework, which checks if the requested content is cached and valid. If so, data is returned to the client without forwarding the request to the provider.

`CACHE-IT` comprises two classes of computational nodes: a single Cache Controller and N_C Cache Nodes. The set of Cache Nodes is denoted as $C = \{c_1, c_2, \ldots, c_{N_C}\}$. The Cache Manager – a Cache Controller component – generates a set of instructions, called caching orders, for each Cache Node that determine: *what*, *when*, and *how long* to cache. The Cache Workers – one per Cache Node – are responsible for carrying out the caching orders – i.e., performing requests and storing data – and managing clients' requests, thus returning the cached content or forwarding their request to the specified provider. Typically, Cache Nodes are located at the edge, and the Cache Controller is deployed in the cloud. Any device with an operating system and storage capabilities is suitable as a Cache Node. It can be deployed in a dedicated configuration, such as on a Raspberry Pi, or as an independent and isolated process within a shared environment. The latter setup is better suited for more powerful and stable equipment (e.g., base stations). However, we highlight that the Cache Node does not engage in resource-intensive processing tasks (which are executed by the Cache Manager), resulting in minimal impact on the node's computational metrics. A caching order is a well-defined structure that contains an instruction to be performed by the specified Cache Node at a certain execution time, and its output is valid until the specified expiration time. A caching order has two different modes:

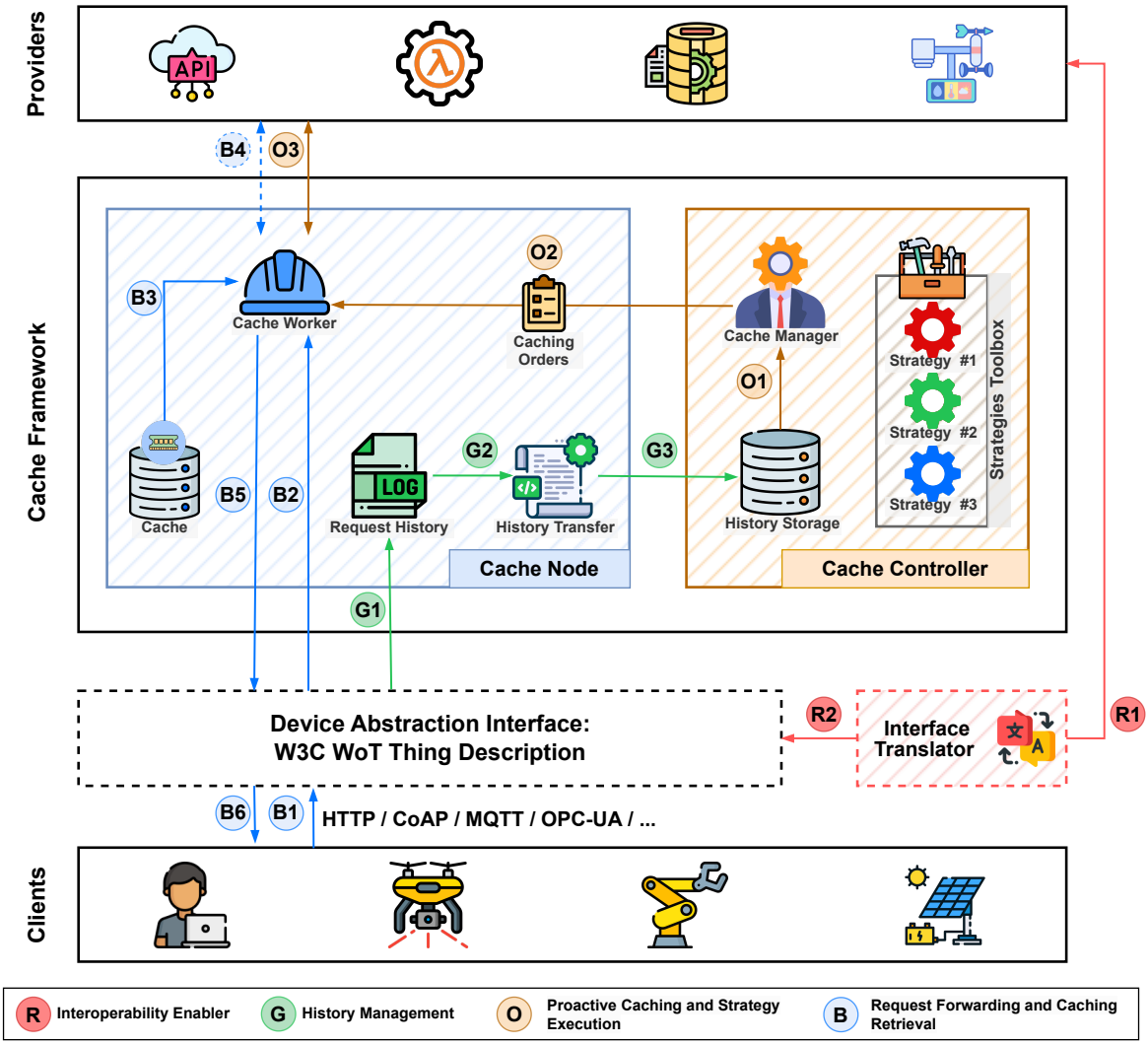


Fig. 4.2 CACHE-IT High Level Architecture

- *Standalone*: the caching order contains the provider to be requested and all the metadata needed to formulate the request. Each cache order makes the Cache Worker perform a request to a given provider; the corresponding response is cached for a predetermined time.

- *Cooperative*: besides the content in standard mode, the caching order includes an additional parameter: a Cache Node address. In this case, instead of requesting data directly to the service provider, the Cache Worker checks if the Cache Node specified has the requested resource; if so, the requests to the specified provider will be fetched from that Cache Node until the defined expiration time.

By employing cooperative caching orders, the Cache Manager can minimize the number of requests made to the providers. This is achieved by storing a particular resource in one Cache Node and directing the remaining Cache Nodes to retrieve it from that specific Cache Node. Before initialization, the user defines the *Caching Template*, a well-defined structure that holds all the system configurations – including the caching strategy. Table 4.2 lists the attributes included in the Caching Template along with a brief description. There are four independent data flows in `CACHE-IT` (as depicted in Figure 4.2 legend), and for each Caching Template attribute, we denote the data flow it impact.

Table 4.2 Caching Template

Attribute	**Description**	**Data Flow**
`providers`	data providers to be translated to a seamless interface	interoperability
`record attributes`	what attributes to register from clients' requests	history management
`logs longevity`	retention time of log files	history management
`t time slot duration`	the time slot in which a new set of caching orders is generated	caching strategy execution
`caching strategy`	a set of three functions that generate caching orders	caching strategy execution
`cache node storage`	the maximum storage available in each cache node	caching strategy execution
`cache replacement`	resource replacement strategy	request forwarding and data retrieval
`cNN`	the number of neighbor Cache Nodes visited if the resource is found locally	request forwarding and data retrieval
`reactive caching`	if the data resulting from a cache miss is stored	request forwarding and data retrieval
`cache expiration time`	The default time that cached resources are valid in the cache in case that it is not specified by the caching order	request forwarding and data retrieval

4.2.2 Operations

After initialization, `CACHE-IT` has a short bootstrap phase for enabling interoperability, followed by the caching strategy execution, which is also triggered every t time slots (the

time slot duration set in the Caching Template). Both the history manager, the request forward, and data retrieval data flows (Subsection 13) are triggered by the arrival of the client's requests.

Interoperability Enabler

An important aspect to consider when designing an IoT caching framework is the heterogeneous nature of the devices, which vary greatly in software and communication protocols. As illustrated by Figure 4.2, CACHE-IT clients can be diverse, ranging from constrained sensors to robots and even regular humans. IoT devices do not usually adopt compatible interfaces with current Web technologies, which makes integration with non-IoT solutions difficult. To address this issue, CACHE-IT provides a standardized and interoperable interface, allowing clients to abstract the specific interface adopted by the data providers.

CACHE-IT bootstrap operation aims to enable interoperability, as illustrated in red in Figure 4.2. The active component in this data flow is the Interface Translator, which bridges dissonant ecosystems, namely the traditional Web, with a standard and interoperable interface for IoT devices.

We adopt the W3C WoT as a interoperability standard; and, as WoT does not provide methods or guidelines to convert dissonant interfaces to its ecosystem, we utilized C3PO presented in 3.2.4 as the current implementation of the Interface Translator. It converts traditional Web services interfaces into TDs and instantiates them in WTs that act as a proxy of the original provider. *Step R1* in Figure 4.2 refers to the syntactical translation of the provider interface into a TD, and *Step R2* depicts its instantiation into a proxy WT. The set of data providers to translate are defined by the `providers` attribute of the Caching Template.

History Management

CACHE-IT defines a pipeline to manage the clients' request history, which the caching strategies may use. Each request performed by a client – through the interoperability layer – is registered in a `log` file, as depicted in the *Step G1* of the dataflow. The lightweight nature of `log` files makes them well-suited for resource-constrained IoT environments. Additionally, the system also supports a configurable data-pruning mechanism to ensure the efficient management of the `log` files. This functionality allows the user to define record retention parameters, namely `logs longevity` defined in the Caching Template. Logs surpassing a pre-set age threshold are automatically pruned based on these parameters, ensuring the lightweight nature of the `log` files over an extended duration.

Each record's core information is its arrival time (i.e., its timestamp) and the Cache Node where the record was registered. The complete content of a record is depicted in Table 4.3. Some record properties are optional. Namely, the returned data, which might be too burdensome to store, manage, and process, and the client identification, which might be concealed for privacy reasons. The user defines through the Caching Template which optional properties will be registered; this choice is based on the information and metadata that the employed caching strategy utilizes.

Table 4.3 Properties of a record

Property	Description	Optional
`request`	IP Address or DNS, port, and URI	✗
`parameters`	query, path, header, and cookie parameters	✗
`cached`	boolean value that is true if the requested content was retrieved from the cache	✗
`delay`	time elapse to retrieve the response (from cache or the response time from performing the request to the provider)	✗
`timestamp`	request arrival timestamp	✗
`cache node`	the Cache Node received and registered the record	✗
`client`	client identification (IP and MAC address)	✓
`data`	returned request data	✓

Step G2 and *Step G3* refer to transferring the request history of a single Cache Node to the centralized Cache Controller storage, a process that occurs in batches. The History Transfer is a lightweight shipper for forwarding and centralizing `log` data. In detail, *Step G2* illustrates the operation of finding and managing all the `log` files that will be transferred to the Cache Controller. The History Transfer needs to keep track of the state of each `log` file and what portion of the file was not sent already. *Step G3* refers to the periodical data transfer of the collected request history to the centralized storage in the Cache Controller. We define as D_{past} the subset of data points stored in the last *past* time. The History Transfer needs to attend to some requirements:

- **Disconnection Handling**: as the Cache Nodes often are unreliable computation nodes in the network edge, they might suffer occasional disconnections from the Internet; hence, they lose communication with the Cache Controller. In such cases, the History Transfer should keep track of the unsent batches and send them once the connection is re-established.
- **Lightweight implementation**: the History Transfer should use limited system resources and not impact the other process executing in the Cache Node.

- **Multi-environment deployment**: there are virtually no constraints on what devices can fulfill the role of a Cache Node. The History Transfer should be able to support multi-heterogeneous deployment environments.
- **History Update Management**: as common scenarios comprise multiple Cache Nodes, those can potentially overwhelm the History Storage. The History Transfer must employ techniques to find an adequate pace to transmit data to the History Storage.

Caching Strategy Execution

The Cache Manager comprises three distinct operations, each with different execution periodicity and detailed in its own Subsection. The modification and deployment of a Caching Template occur upon direct user intervention, and it is a *long-term* operation. Adapting the caching strategy to new environmental conditions is a *medium-term* operation. Lastly, the generation and transmission of caching orders are considered a *short-term* operation. Figure 4.3 depicts each operation and its interactions. Additionally, CACHE-IT *short-term* operations are depicted through the orange dataflow in Figure 4.2. Although they are explained in detail in the following text, an introduction is necessary to understand the big picture, as these operations are a core system component: every t time slot (the duration of which is specified in the Caching Template) the Cache Manager – located in the Cache Controller – invokes a function that generates and transmits (*Step O1* and *Step O2*) instructions (i.e., caching orders) to each Cache Worker. We count the time slots as $t_0, t_1, \ldots, t_n$. When a Cache Worker receives a new batch of caching orders, it reports its caching accuracy in the previous time slot (i.e., t_{i-1}) to potentially trigger *medium-term* operations that adjust the generation of such caching orders (such as retraining a model), meaning that the conditions may have changed.

Long-Term: User Intervention Before system initialization, the user defines the caching strategy in the Caching Template. In proactive edge caching, the strategies can employ diverse techniques that impose different data flows and operations [133]. For instance, a statistical-based strategy requires re-execution for generating each new set of caching orders, while a machine learning-based strategy relies on a pre-trained model, which requires an understanding of the timing for training (and retraining) the model. To encompass variability in the strategies processing steps, we define a generic workflow comprising three functions that compose the caching strategy in CACHE-IT. Those functions are:

- *gen* **function**: the core function that sets the strategy goal and implements a technique (e.g., Deep Learning). It is a higher-order function that produces a function *cOrder*,

which generates a set of caching orders in each time slot (*short-term* operation). The *gen* function is analogous with the machine learning process of *training a model*, while the *cOrder* function pairs with the model execution itself in inference mode. Other techniques that do not involve pre-computing steps can be integrated into the framework by defining *gen* as a deterministic function that always outputs the same *cOrder* function.

- *period* **function**: the function accesses the historical storage and outputs a time slot, determining the size of the data chunk utilized by the *gen* function upon execution. The aim is to determine the last homogeneous time segment of historical storage data. It is possible to utilize a simple rolling window method (e.g., always get the last 1-month of data) or an advanced technique (e.g., an auto-regression model that, based on the prediction errors, determines the last time segment of homogeneous data).

- *trigger* **function**: it is executed every time slot (i.e., *short-term*), taking as input the accuracy of all the caching nodes in the last time window and returns a `Boolean` value. If `true`, it prompts the execution of the *period* and the *gen* functions to regenerate the *cOrder*. The output of the function can be bounded to the Cache Nodes accuracy (e.g., if less than 20%), time (e.g., every 12h), or both.

Upon system initialization or when the user uploads a new Caching Template, `CACHE-IT` executes the newly defined *period* and *gen* functions to generate an updated *cOrder* function considering the newly set configurations, such as the maximum available storage for each Cache Node. Figure 4.2 omits the component in charge of managing the Cache Nodes and gathering data related to their status – i.e., online or offline – and their connectivity. However, we adopted IoTManA[134] as our chosen management system, which provides tools for managing, controlling, and monitoring software, hardware, and communication components. Specifically, it facilitates monitoring network delay and system entities' availability – i.e., the Cache Controller and Cache Nodes.

Medium-Term: Caching Strategy Adaption Due to changes in the context, such as the inclusion, removal, or behavior change of IoT devices, a historical system snapshot may not represent its current state. To periodically adapt the generation of caching orders to reflect the system conditions better, `CACHE-IT` performs a check (e.g., executes *trigger* function) each time slot to determine if conditions have changed. From `CACHE-IT` point-of-view, an alteration in system conditions means that the output of *trigger* function is `true` or that a new Caching Template was uploaded. In such cases, `CACHE-IT` regenerates the *cOrder* function by executing the *period* function followed by the *gen* function.

Short-Term: Generation of Caching Orders The *short-term* operations encompass the generation of caching orders in each time slot. Algorithm 1 is executed in fixed slots of t time to generate and transmit a new set of caching orders, detailing *Step O1* and *Step O2* from Figure 4.2. Line 2 denotes the generated function *cOrder* producing caching orders for the next t_{i+1} time slot. Then, the Cache Manager transmits to each Cache Worker its set of caching orders, which reply with their caching accuracy in the last t_{i-1} time slot (lines 3 to 5). A Cache Worker accuracy is defined as the total amount of cache hits divided by the total number of requests. The algorithm checks if *gen* is already in execution (line 6). Then, if the *trigger* function outputs `true` (lines 7 and 8) the *medium-term* operations are triggered. Namely, the execution of *period* (line 9), followed by *gen* (line 10); which updates the caching order generating function *cOrder*.Figure 4.3 illustrates an example of the three distinct caching operations. The upload of a new Caching Template (*long-term*) triggers the execution of *period* and *gen* to generate a new *cOrder* (*medium-term*). In the example, the *gen* function performs the retraining of an LSTM Neural Network model, which becomes the updated *cOrder*. Then, the Neural Network (i.e., *cOrder*) generates a new batch of caching orders in every time slot (e.g., 2h), which are transmitted to each Cache Node (*short-term*). The Cache Nodes reply with their accuracy. The *trigger* function then takes the set of accuracies as input and, if their average is below 20%, prompts the execution of the *medium-term* operations to retrain the Neural Network.

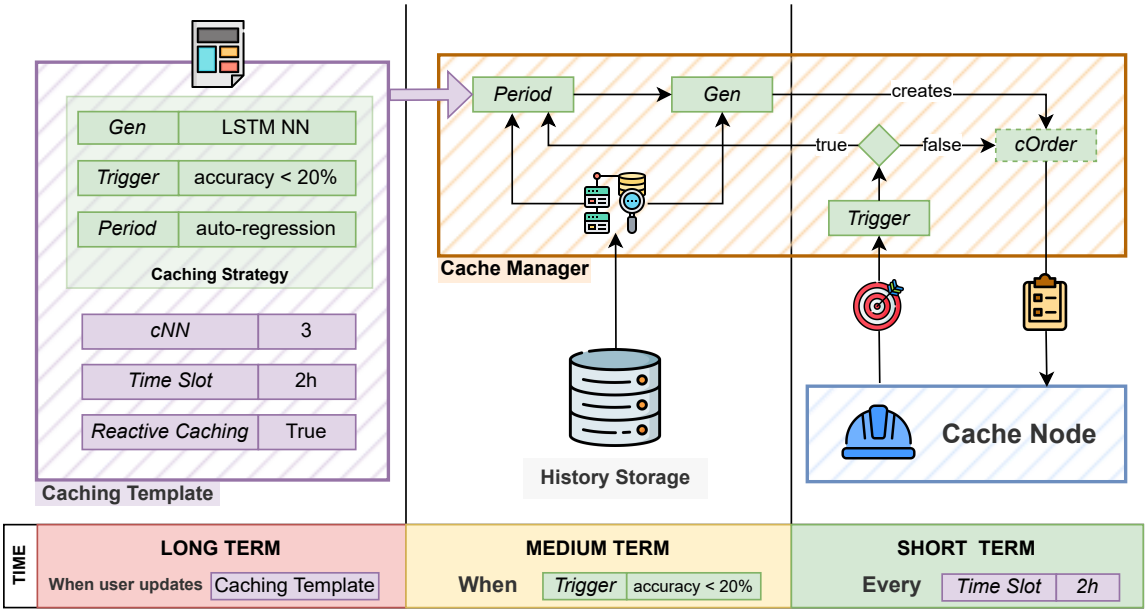


Fig. 4.3 CACHE-IT operations and their timings.

Algorithm 1: The CACHE-IT caching orders generation

Every t_i **time passed:**
2 $caching\_orders_{t_i} \leftarrow cOrder(D_{past})$
3 **for** c *from* 0 **to** N_c **do**
4 $orders_c \leftarrow$ filter $caching\_orders_{t_i}$ for caching orders that match c
5 transmit to the c Cache Node the $orders_c$ and receive $accuracy_{t_{i-1},c}$ of all Cache Workers
6 **if** *gen is not executing* **then**
7 $isTrigger \leftarrow trigger(\bigcup_{c \in C} accuracy_{t_{i-1},c})$
8 **if** *isTrigger* **then**
9 $past \leftarrow period(D)$
10 $cOrder \leftarrow gen(D_{past})$
11 **end**
12 **end**
13 **end**

Request Forwarding and Caching Retrieval

The request forwarding and caching retrieval process employed by each Cache Node is triggered upon receiving a request; the overall flow is depicted in blue in Figure 4.2. A client requests data from a given provider in any protocol supported by the Device Abstraction Interface, as depicted by *Step B1*, which translates the request and forwards it to the Cache Worker. In turn, the Cache Worker checks if the requested data is cached (*Step B3*). If so, the data is returned to the client (*Step B5* and *Step B6*). The role of *Step B6* is mapping the data to the format and protocol used by the client. When a cache miss occurs (*Step B4*), the Cache Worker forwards the request to the provider.

Algorithm 2 is executed by each Cache Worker to handle client data requests and to perform caching orders. One particular feature described in the Algorithm 2 is the cNN (CACHE-IT Nearest Neighbors) strategy. In cNN, when a cache miss occurs, the Cache Worker checks if its N nearest neighbors (i.e., other Cache Nodes) have the resource requested cached. This feature can be integrated into any caching strategy and capitalizes on the assumption that geographic proximity influences the request pattern. The amount of N of neighbors visited is defined in the Caching Template.

Algorithm 2 keeps track of the received caching orders, the local cache hit rate (i.e., its accuracy), and the set of N neighboring Cache Nodes and their latency towards the originating Cache Node. The Algorithm 2 uses two event handlers for dealing with the arrival of caching orders and client requests. For the sake of simplicity, we adopt the dot notation (.) to access the properties of a caching order – e.g., *order.type*.

When a set of caching orders arrives (line 5), the algorithm checks if any client requests were made since the last batch of caching orders was received (line 6); if so, it computes the cache accuracy by dividing the number of cached requests by the total number and transmits

it to the Cache Controller (lines 7-9). For each caching order in the set, the algorithm checks whether the order is of *standalone* or *cooperative* (lines 11-17). For standalone orders, the algorithm performs the request at the designated execution time and caches the response data until its expiration time – if a given caching order did have its expiration time specified by the caching strategy, it uses the default expiration time set in the Caching Template; if the caching order is *cooperative*, the Cache Worker adds it to a list of cooperative orders (line 15). The list of cooperative orders is periodically updated to remove expired orders – omitted from the algorithm for clarity.

When a request arrives from a client (line 18), the algorithm first checks if the content is present in the local cache (line 20). If so, the corresponding data is returned to the client, and the *cachedRequests* variable is incremented (line 21). If not, the Cache Worker checks if there is a valid cooperative caching order that matches the request (lines 22 - 27). If so, the Cache Worker tries to retrieve data from the Caching Node specified in the cooperative order, returning it to the client and incrementing *cachedRequests* in case of success. If the data is not found, the algorithm performs the cNN strategy (lines 28 - 31) by verifying if the content requested is cached in any of the *N* nearest Cache Nodes. Finally, if data is not found, the algorithm performs the request directly to the data provider and returns the data to the client (lines 32 - 36). The returned resource is only cached if the variable *reactiveCaching* is set to `true`. This behavior considers the reliability of the prediction solution, as a cache miss may be an outlier, and caching its returned resource might be unnecessary.

CACHE-IT default cache replacement strategy is Least Recently Used (LRU). However, users can configure it to use other strategies such as Least Frequently Used (LFU), random, or maximum idle time.

4.2.3 Implementation

This Section describes CACHE-IT implementation. We utilize industry-adopted applications to fulfill some framework components while the others perform specific CACHE-IT tasks, which we implemented ourselves. A first version of an **Interface Translator** was proposed in [96] and its implementation, namely C3PO (**C**onverter of O**P**en A**P**I S**P**ecification to WoT **O**bjects), was detailed in [135]. It can convert RESTful Web services APIs documented through the OpenAPI Specification (OAS) [20] into WTs. The current implementation requires a formal description of the provider interface (including its endpoints, inputs, outputs, and parameters) for the translation process. The OAS provides a language-independent standard to describe RESTful interfaces using a JSON-based description, and it is the *de facto* standard for API documentation.

Algorithm 2: The Cache Worker caching retrieval and request forwarding.

```
1  cooperativeOrders ← []
2  cacheNodes ← list of CacheNodes and their respective latency
3  totalClientRequests ← 0
4  cachedRequests ← 0
5  upon the arrival of a set of caching orders orders: (every t intervals)
6      if totalClientRequests is not 0 then
7          accuracy ← cachedRequests/totalClientRequests
8      else
9          accuracy ← 1
10     end
11     transmit accuracy to Cache Controller and reset totalClientRequests and cachedRequests
12     for order in orders do
13             if order.type = standard then
14                 data ← performRequest(order)
15                 store(data, order.expirationTime)
16             else
17                 cooperativeOrders.push(order)
18             end
19     end
20 upon the arrival of a request r from client:
21     increment totalClientRequests
22     data ← getLocalCache(r)
23     if data is not null increment cachedRequests and return data to the client
24     for order in cooperativeOrders do
25         if match(order, r) then
26             data ← getFromCacheNode(order.cacheNode, r)
27             if data is not null increment cachedRequests and return data to the client
28         end
29     end
30     for cacheNode in cacheNodes do
31         data ← getFromCacheNode(cacheNode, r)
32         if data is not null increment cachedRequests, return data to the client and break
33     end
34     data ← performRequest(r)
35     if reactiveCaching then
36         store(data, presetTime)
37     end
38     return data to client
```

We adopted the ELK stack[1] to fulfill the role of the **History Transfer** and **the History Storage**, as it satisfies the requirements listed. The ELK stack is a set of open-source tools that provide a flexible and scalable platform for collecting and storing distributed `log` data. ELK is an acronym built with the union of its three main components – i.e., Elasticsearch, Logstash, and Kibana. Elasticsearch is employed for indexing and storing data; Logstash is a data processing pipeline that collects and parses the `log` data to be stored in Elasticsearch.

[1] https://www.elastic.co

Logstash and Elasticsearch together fulfill the role of History Storage. The History Transfer is implemented by Filebeat, a lightweight shipper that is used to collect and transfer log data to Logstash. It can be instantiated in practically any computational environment and utilizes a back-pressure sensitive protocol to send data to Logstash, thus preventing overloading.

While the ELK stack was chosen for its scalability and flexibility, many other alternatives follow different approaches, like Fluentd[2] and Graylog[3]. Fluentd is built around an extensible architecture. Like Logstash, it offers a unified logging layer, but where it distinguishes itself is in its pluggable architecture. Fluentd supports numerous input and output sources via plugins, enabling it to integrate with various systems without core modifications. Also, it uses a lightweight core and is written in Ruby and C, which might provide efficiency benefits in specific environments. Graylog is centered on simplicity and ease of use. While Elasticsearch can be used as a backend for both ELK and Graylog, Graylog provides a more streamlined setup process and a centralized management interface for various logging pipelines. A notable feature of Graylog is its built-in alerting and reporting capabilities, allowing users to generate insights from their log data more seamlessly. However, based on the particular requirements and design principles of CACHE-IT, the ELK stack was determined to be most aligned with our goals.

The Cache Manager invokes the caching strategy functions through a POST request towards a user-specified address (defined in the Caching Template). The caching strategy needs to be wrapped in an application that exposes a REST API. That way, we preserve CACHE-IT separation of concerns and provide the user a rapid way to deploy and test different versions of their caching strategy since REST API can be hosted through a lightweight virtualization technique. This design allows caching strategy functions to implement more complex operations, including components tailored to specific scenarios or domains, such as mobility prediction. Finally, **Cache Worker** and the **Cache Manager** were developed as NodeJS[4] applications to be executed in a container environment [136].

As for the cache storage, we adopted Redis[5], considered the *de facto* standard database for caching. It is a lightweight key-value store implementation that operates entirely in memory, making it suitable for use cases demanding low latency. Additionally, Redis currently supports various operating systems and hardware architectures, and its low resource requirements make it compatible with edge scenarios. The framework is designed to be deployed using lightweight virtualization (e.g., Docker containers[6]).

[2]https://www.fluentd.org
[3]https://graylog.org
[4]https://nodejs.org
[5]https://redis.io
[6]https://www.docker.com/

4.2.4 Performance Analysis

This section evaluates different CACHE-IT features under different client behaviors. Specifically, we aim to quantify the effects of (1) the number of neighbors visited by the cNN ranging from zero to two, (2) the caching strategy accuracy, and (3) the usage of cooperative caching orders as opposed to standalone orders. The objective is to understand how these factors impact the overall system performance, demonstrating the flexibility, applicability, and performance of CACHE-IT.

We designed and implemented an open-source simulator [137] to perform large-scale simulations by modeling the Cache Workers, the Cache Manager, and the network behavior.

In the experiments, we do not implement an explicit caching strategy. Instead, we emulate the caching strategy accuracy as a percentage that dictates the number of requests it could predict. Introducing a specific caching strategy would divert the evaluation toward the effectiveness of that particular strategy, overshadowing the evaluation of the cache architecture as a whole. By abstracting the specific caching strategy operations from our evaluation, we isolate the impact of the unique features and benefits provided by CACHE-IT. To emulate the caching strategy, the Cache Manager probabilistically selects a number of requests according to the caching strategy accuracy, sets them as a simulation input, and transforms those into standard caching orders. The caching order execution time is determined by the request execution time subtracted from a Gaussian time to represent the lack of precision in determining the specific request arrival time. The request expiration time is set to a default value of 10 min. If the experiment utilizes cooperative orders, the Cache Manager analyzes the caching orders and identifies duplicate resources cached in the same time slot; in those cases, it converts one of the orders to cooperative, pointing to the Cache Node that holds the resource in that time. Finally, the Cache Manager performs a redundancy check to eliminate duplicate caching orders. Standalone caching orders that store resources used by cooperative orders are excluded from this process.

For simplicity and without loss of generality, the data providers are abstracted as entities that generate resources over time. Its application and network behavior are modeled based on real datasets. We utilized [138] to model the network latency between the edge nodes and the data providers. The mentioned work characterizes cloud-to-user latency from different geographically distributed vantage points towards several data centers in distinct locations on the infrastructures of Amazon Web Services[7] and Microsoft Azure[8]. Upon initialization, each data provider randomly selects a specific cloud data center and emulates its network latency. We utilized the encrypted web traffic dataset as a base to model the data providers'

[7]https://aws.amazon.com/
[8]https://azure.microsoft.com/

behavior [139], specifically, the returned data size in bytes and the application processing time. The dataset comprises 800 real web services monitoring data – including the mentioned metrics. We select three significant web applications to represent high, medium, and low values regarding the returned data size of the data providers since this feature had non-overlapping distributions as opposed to processing time, which is similar to all services. The characterization of the selected web services is shown in Figure 4.4 and in Figure 4.13 regarding processing time and bytes returned. Data providers were assigned to each category equally. We disregard the latency added by the Device Abstraction Layer since a previous work [96] demonstrates it is insignificant.

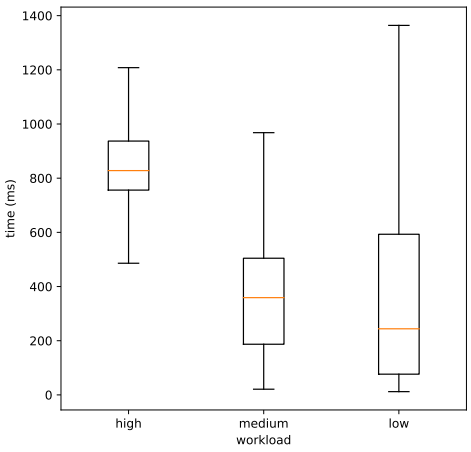


Fig. 4.4 Data characterization of processing time for the three categories of data providers.

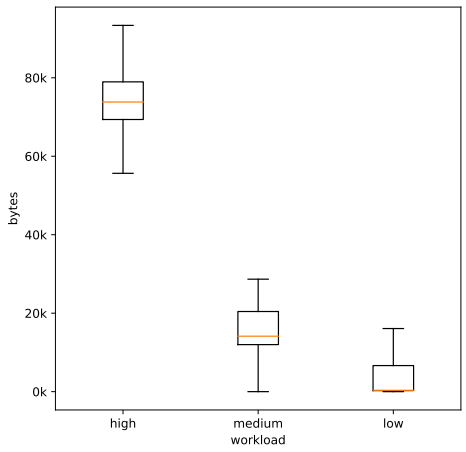


Fig. 4.5 Data characterization of response size in bytes for the three categories of data providers.

The simulator deploys a configurable number of edge nodes within the experiment dimensions in random positions. These edge nodes serve as network gateways – e.g., base stations. Each edge node is equipped with a dedicated cache storage, and the caching replacement strategy employed in our experiments is the Least Recently Used (LRU) algorithm. We modeled the network between the client and the edge node following the wireless latency dataset present in [140], and the inter-edge nodes communications were modeled using the wired LAN dataset of the same source [140].

Clients in IoT scenarios often are mobile, ranging from industry 4.0 devices [141] to intelligent vehicles [109], so we add client mobility in the evaluation scenario. Each client moves in a trajectory determined by a list of random reference points within the experiment dimensions. The simulator generates a list of requests for each client for the experiment duration, and clients perform these requests by querying its closest edge node. The inter-

arrival times between requests follow an exponential distribution corresponding to a Poisson process. The choice of which provider to query follows a Zipf popularity distribution since the latter is widely used to simulate the popularity of requests to data providers in the context of edge caching [142, 88, 121, 92, 109, 110, 119, 116]. We utilized the Zipf parameter as 1.1 [121] independent of the client category. Additionally, we modeled three different patterns of client behavior based on observation in IoT systems; each client category alters how providers are selected. The popularity distribution in this context means that the popularity order of each data provider was randomly shuffled while still adhering to the same underlying Zipf distribution. The categories are:

- **ID**: clients in this category are independent of each other and follow their particular popularity distribution.
- **type**: Each client in this category is assigned a specific type with its own popularity distribution. This behavior was modeled assuming that devices of the same type (e.g., a specific brand and model of UAV) tend to consume data from similar providers.
- **location**: clients in this category have popularity distributions associated with their respective areas. The simulation considers the total area and divides it into a parameterized number of subareas, each with its popularity distribution.

Each category is illustrated in Figure 4.6. We perform experiments for each client type and an experiment configuration in which we deploy all client types in the same experiment equally distributed – this experiment configuration is called “mix”.

During the simulation, we track and record the following metrics:

- **Latency**: captures the time taken for each request to be returned to the client.
- **AoI**: represents the time difference between the generation of a resource and its arrival to the clients.
- **Number of Requests to Providers**: the total number of requests sent to the data providers.
- **Cache Hit Rate**: the percentage of requests retrieved from the cache as opposed to those forwarded to the data provider.

Finally, Table 4.6 lists the experiment parameters kept constant in all simulations. Table 4.7 depicts the factors and levels utilized in the performance evaluation. We perform experiments combining all the levels and factors. The caching order type denotes experiments executed only with standalone orders instead of the ones in which cooperative orders were

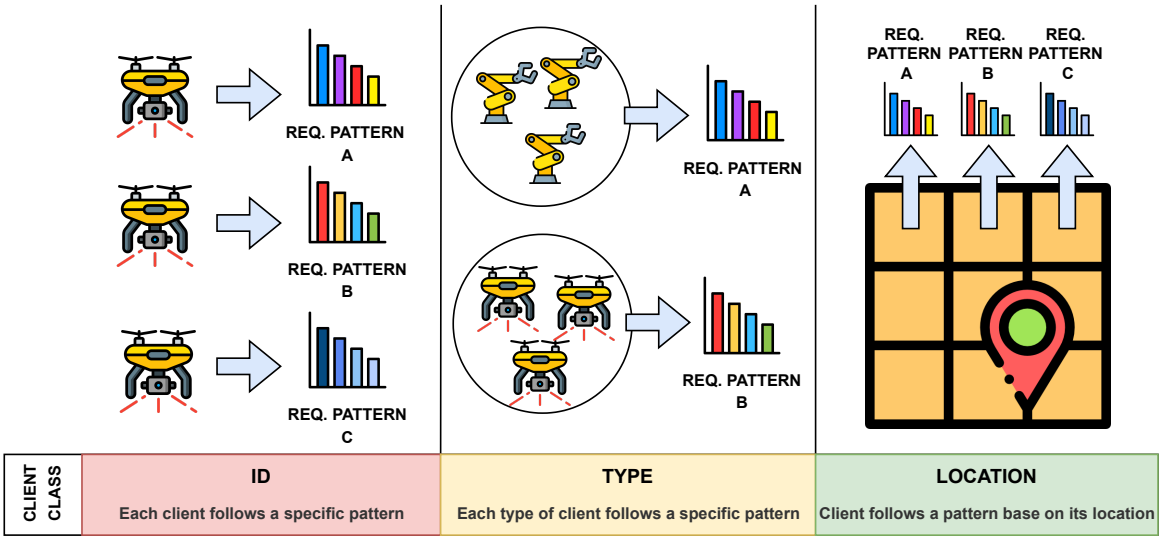


Fig. 4.6 Representation of the different categories of client behavior modeled in the experiments.

used. For clarity, we refer to the experiments utilizing standard and cooperative orders as "cooperative." Each experiment was replicated 30 times, and the calculated confidence interval was 99%. In each replication, all entities involved were re-instantiated, increasing the inter-experiment variability – e.g., the clients' trajectories and initial positions, the edge nodes placement, the assignment of providers to their category, and a network trace.

Figure 4.7 depicts a handful selection of the most significant experiment configurations, which allows for a comprehensive analysis of the performance and behavior of our proposed framework. Those results were all executed with "mix" as the client category. We adopt a consistent naming pattern for our experiments: "accX-NY-Z," where X represents the accuracy percentage, Y denotes the number of neighbors visited (for the c-NN strategy), and Z indicates the caching order employed, "s" for standalone and "c" for cooperative. Exceptions are "baseline," which denotes the experiments without cache, and "regular-cache," which refers to experiments only applying the current reactive caching. The results showcase that the greater the caching strategy accuracy and the number of neighbors visited, the better the outcome for all metrics, except for AoI – since cached resources are less fresh than those fetched directly from the data providers.

It is noteworthy that CACHE-IT impacts positively in terms of latency, hit rate, and the number of requests sent to providers. The framework features enhance caching strategy accuracy, and we numerically show that the c-NN mechanism meaningfully improves the system. One experiment parameter that influences the results is the default time to keep resources cached (10 min), as the clients tend to query the same provider, which increases

Table 4.4 Experiment Parameters

Property	**Value**
Experiment Duration	1h
Area dimensions	10 000 units^2
Number of edge nodes	10
Edge storage size	4 GB
Minimum distance between edge nodes	1000 units
Number of clients	500
Client Speed	10 units/s
Number of client types	5
Number of subareas	5
Default resource expiration time	10 min
Reactive caching	✓
Number of providers	750
Rate of requests per client (Poisson λ)	0.1 event/s
Popularity distribution (Zipf α)	1.1

Table 4.5 Factors and Levels

Factor	**Level**
Caching Strategy Accuracy	0%, 20%, 40%
Caching Orders Type	standalone, cooperative
c-NN (N neighbors visited)	0, 1, 2
Client Category	location, ID, type, mix

the overall hit rate, though it also increases the AoI. The baseline configuration value in Figure 4.7, approximately 0.25 seconds, reflects the minimal time difference from resource generation to client delivery, as the resource was not cached for a period of time. System administrators deploying `CACHE-IT` must consider the trade-off between lower latency and data freshness. In scenarios where it is key to have low AoI, the amount of time to keep resources cached should be bounded by the maximum AoI.

Figure 4.8 deepens the comparison between the experiments that use cooperative orders and experiments that do not. In general, the outcome of those experiments differs for the number of requests sent to providers and AoI. The number of requests is lower due to the cooperative aspect of the system sharing the predicted resource without performing additional requests. Consequently, the resources returned are less fresh, which increases the AoI. The difference in latency and hit rate between cooperative and standalone caching is attributed to removing redundant caching orders. Removing redundant caching orders does not eliminate

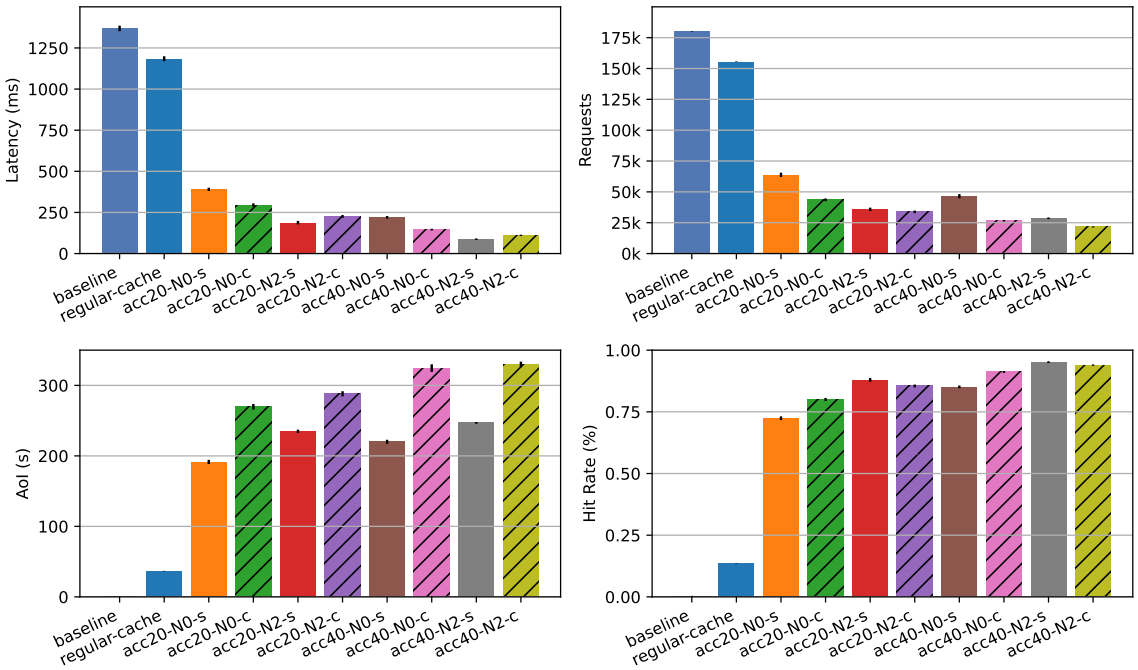


Fig. 4.7 Overall Simulation results for CACHE-IT comparing different configurations. Hatched bars represent experiments in which cooperative caching orders were used.

caching orders that serve as pointers for cooperative ones, resulting in a higher number of requests and a greater average number of cached resources per Cache Node. However, when the number of visited neighbors (N) increases, this aspect is mitigated as resources are shared with the closest nodes.

That behavior is observed by comparing the different Figure 4.8 rows – each row corresponds to a number of neighborhoods visited, ranging from zero to two. This disparity in latency arises due to the additional one-hop requirement for cooperative orders to retrieve the requested data. When comparing the experiments with cNN equals two, both cooperative and standalone caching strategies exhibit similar high hit rates. However, the incremental latency introduced by cooperative orders (i.e., the additional hop to retrieve a resource) becomes more noticeable. In our experiments with ten edge nodes, configurations with two visited neighbors had access to approximately 30% of the total cached resources.

Figure 4.9 shows sets of graphs characterizing the performance of the different client categories: each row represents a c-NN value, starting from zero until two. All the experiments were performed using standalone orders only. The results show that the client categories "type" and "location" have the best overall results since clients share a similar requesting pattern. In contrast, the "id" client category had slightly worse results since each client is independent of the other. Due to the heterogeneity of clients' behaviors, the experiments with

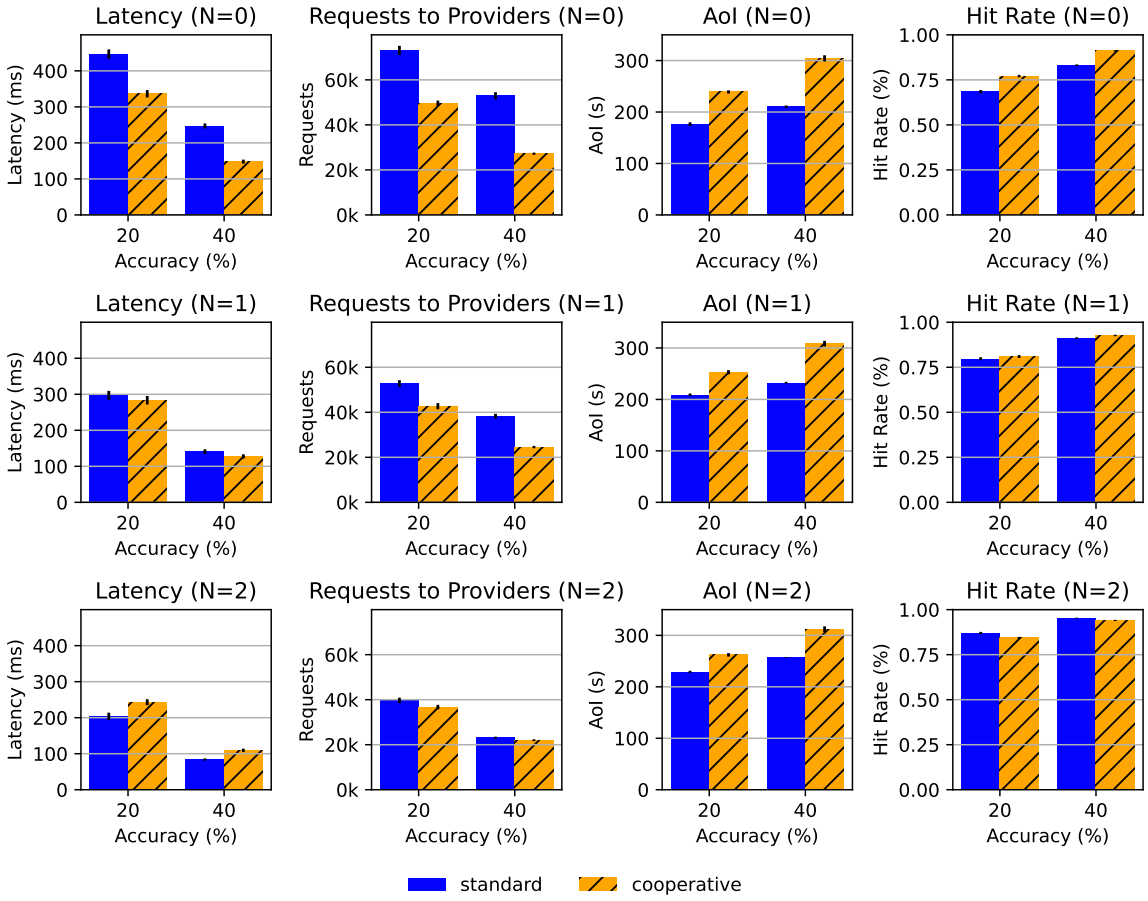


Fig. 4.8 Simulation results for CACHE-IT comparing standard and cooperative caching orders. Each row represents a different cNN configuration, denoted as N.

all the types – i.e., "mix" – follow the same pattern as observed in the experiments performed with the "id" client category, which means that in those scenarios, the clients' popularity distribution was, in practice, independent from each other. However, Figure 4.9 makes evident that the client category differences are minor when compared to the impact wielded by other simulation configurations, as the caching strategy accuracy. This behavior indicates that CACHE-IT is versatile enough to be exploited in different contexts regarding client interactions. The results presented in this section are open-source to guarantee transparency and replicability; they can be found in the project's GitHub repository [137].

4.3 CACHE-IT support for Federated Learning

This section details how CACHE-IT can be utilized in privacy-aware scenarios. We denote as CACHE-IT FL the extension of CACHE-IT that allows it to perform Federated Learning

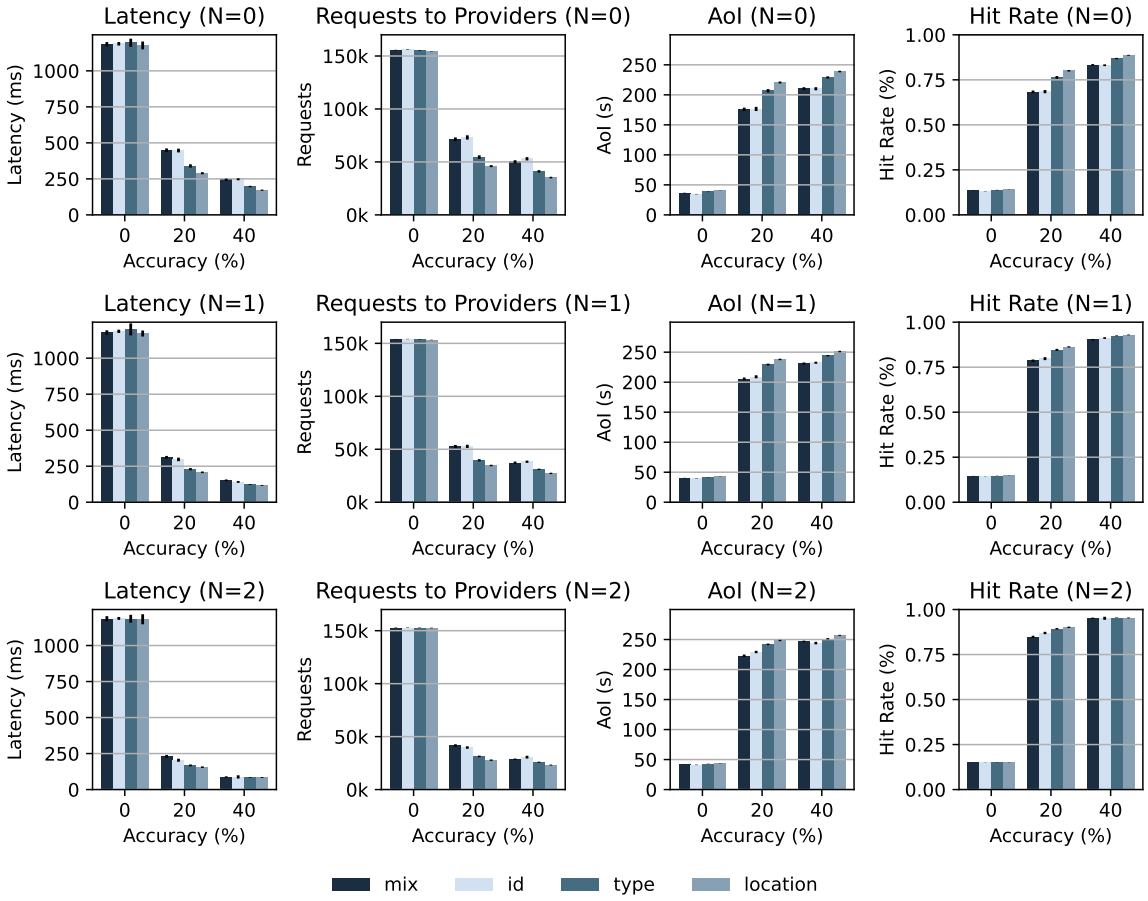


Fig. 4.9 Simulation results for CACHE-IT for different client types. Each row represents a different cNN caching configuration, denoted as N.

strategies. Based on the privacy requirements, CACHE-IT FL supports three categories of edge-caching strategies, i.e., local, global or federated, while being agnostic of the specific prediction algorithm in use. Two main contributions are provided in this section:

- **Privacy awareness**: our architecture seamlessly accommodates various forms of data sharing between edge and cloud nodes, enabling proactive caching strategies. This capability allows the system owners to strike the wanted balance between optimizing the Age of Information and cache hit rate (enhancing application performance) while also prioritizing privacy preservation (improving client performance) through federated strategies.

- **IoT-cloud computing continuum exploitation**: The location where the data is processed to generate the information exchanged with the central node can be in the edge

node, the cloud, or somewhere inside the continuum; allowing optimal adaptation to diverse scenario and domain requirements.

We simulate the CACHE-IT FL architecture and some proposed caching strategies with a dataset generated according to IoT common patterns. For each device in the IoT system, the strategies forecast the most appropriate time to update the cache to guarantee that the AoI meets the application requirements whenever the data is needed. The CACHE-IT FL architecture is detailed in Section 4.3.1, which also encompasses a FL approach for CACHE-IT FL, and Section 4.3.2 shows performance results.

4.3.1 Architectural Design

CACHE-IT FL is a set of extensions applied in CACHE-IT to support a spectrum of heterogeneous caching strategies, from the traditional centralized machine learning-based techniques to FL-based ones. Further, it allows to balance the processing between edge and cloud instead of offloading the computing tasks solely to the cloud.

Figure 4.10 illustrates the CACHE-IT FL architecture, which is based on a simplified depiction of the architecture of Figure 4.2 in Section 4.2. CACHE-IT FL introduces an additional layer of computational component: the cache steward, which have a 1:1 relationship to cache workers. The naming comes from labor relationships, in which workers carry out the instructions determined by managers, and stewards manage the communication between managers and workers. In CACHE-IT FL the cache strategy is collaboratively computed between the Cache Manager and the Cache Stewards. Typically, the Cache Stewards are located together with the Cache Workers, i.e., at the edge. However, other configurations are possible, such as placing the Stewards in the cloud or somewhere within the continuum between the edge and the cloud. The `log` file generated by each Cache Worker are periodically transferred to its corresponding Cache Steward – instead of the Cache Manager.

On system initialization, the system administrator must specify a *caching template* defining CACHE-IT FL configurations with certain attributes: CACHE-IT FL adds an additional attribute to the Caching Template. The **Model Execution Layer (MEL)**, a binary configuration specifying whether the generation of cache orders is centralized by the Cache Manager or distributed to the Cache Stewards. While CACHE-IT FL offers greater customization regarding privacy and the choice of where to execute the Caching Strategy, it constraints the type of strategy to those based on ML that produce a model. This limitation arises due to the requirement of having a model to operate in a FL setting. In this context, the *gen* function is divided into two model generation functions (the *steward model generation* and the *manager*

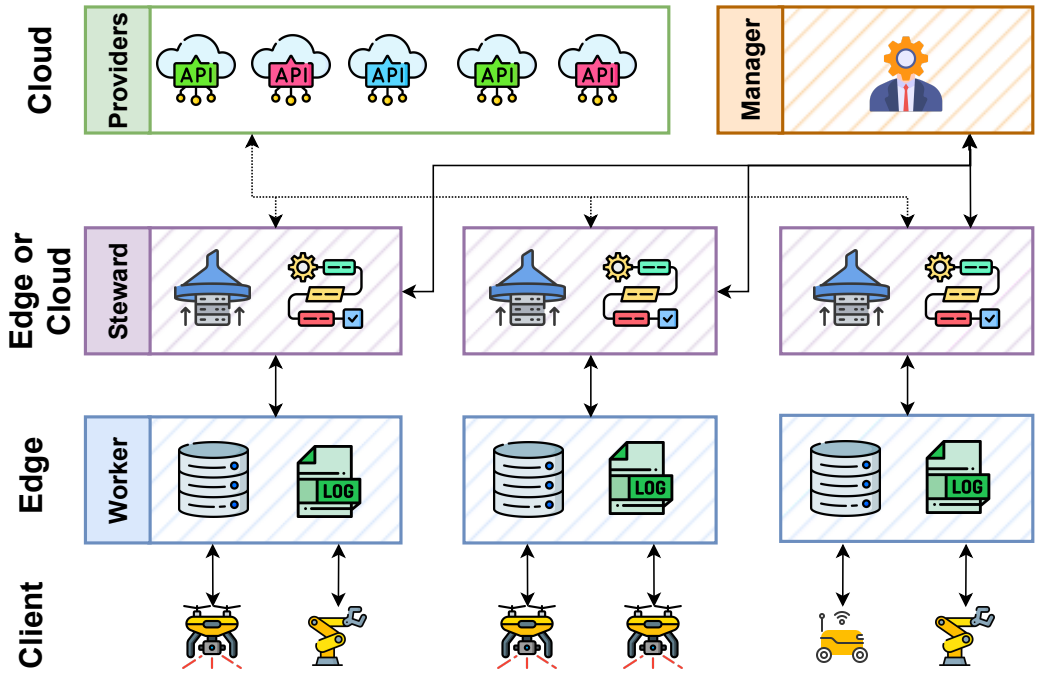


Fig. 4.10 CACHE-IT FL High Level Architecture

model generation) which collaborate to produce a model (*cOrder* function) that generates caching orders for each Cache Worker.

The activity diagram in Figure 4.11 summarizes the operations of CACHE-IT FL– as a simplification, we represent only one Cache Worker and one Cache Steward in the diagram. The colored blocks represent the configured functions that compose the model generation functions (i.e., CACHE-IT *gen* function). The activity diagram executes in each t time slot (from the *caching template*) and initializes with the caching order generation (*cOrder*), which executes the *cOrder* function generated by the model generation functions. Depending on the MEL, the orders are generated by the Stewards or the Manager. Following this operation, the set of caching orders is transmitted to each Cache Worker, which triggers two parallel activities:

1. Iterate through the set of caching orders, awaiting the next order execution time to make a request to the specific provider and cache the returned content – i.e., the Cache Worker performs the operations described in the caching orders;

2. The calculation of the cache accuracy in the last t time slot is transmitted to the Cache Manager.

The Cache Manager averages the cache accuracy of all Cache Workers and executes of the *trigger* function passing as input the aggregated accuracy result. If *trigger* returns `true`, the

model generation functions are re-executed to generate an updated model, which will replace the current *cOrder* function. Otherwise, the procedure finishes. To retrain the model, the first function executed is the Steward Model Generation. According to the Caching Template, the function can execute a local model (if the caching strategy is FL-based), transmit the request historical data to the Cache Manager, or execute data transformation and preprocessing steps before sending it to the Cache Manager. Following the Steward Model Generation, the Manager Model Generation function is executed using the data received from the Stewards to execute the centralized model training, validation, and evaluation. The output of this function is a new model to generate caching orders that will update the current Caching Order Generation – in the Steward or the Manager, depending on the MEL. The execution of the Steward Model Generation and the Manager Model Generation functions is repeated until a stopping criterion is reached, such as a maximum number of rounds, achieving a desired performance level, or when convergence is reached.

Federated Learning in `CACHE-IT FL`

This subsection details how the `CACHE-IT FL` can support a FL-based architecture. Figure 4.12 presents a visual representation of the operations to generate a model using FL through the two functions that compose a caching strategy deployed across the `CACHE-IT FL` components. The diagram of Figure 4.12 depicts the main operations as colored boxes – the ones that have continuous lines are the functions that compose the caching strategy.

The operation initializes with the Local Model execution in the Steward (i.e., Caching Order Generation function as Figure 4.12 depicts), resulting in the generation and transmission of caching orders to the Cache Workers. When a Cache Worker receives a new batch of orders, it transfers the cache accuracy from the most recent time window to the Cache Steward. In case that the *trigger* function returns `true`, the model is retrained. This operation comprises the following steps (illustrated in 4.12):

1. Steward Model Gen.: each Cache Steward trains its local model with new data and transfer it to the Cache Manager;

2. Manager Model Gen.: the Cache Manager performs the aggregation of the new local models and update the global model according with the defined strategy (e.g., FedAvg).

Then, the Stewards download the model generated by the Manager and update their local models. The execution of those two functions continues until the stopping criteria is met.

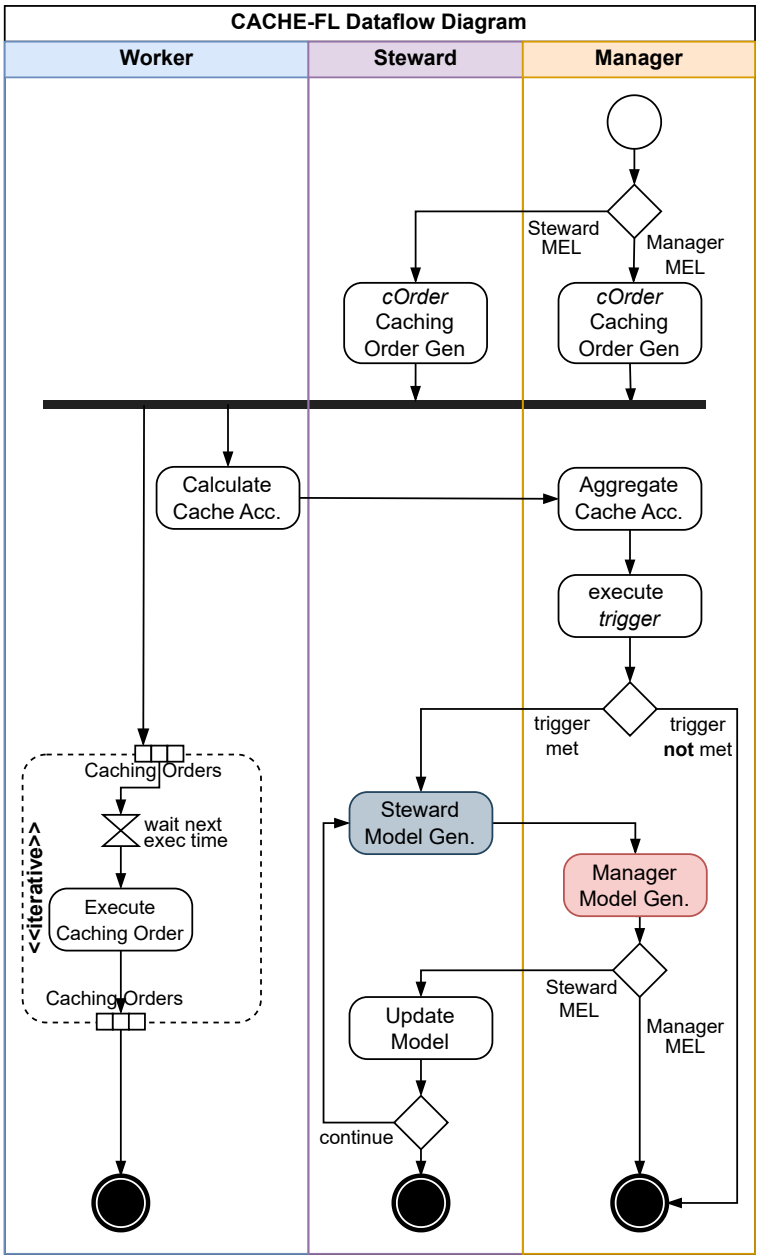


Fig. 4.11 Model Generation Activity Diagram

4.3.2 Performance Analysis

Experimental Setup

We validate CACHE-IT FL by implementing the model proposed in Section 6.3.1 in three different categories of edge-caching strategy:

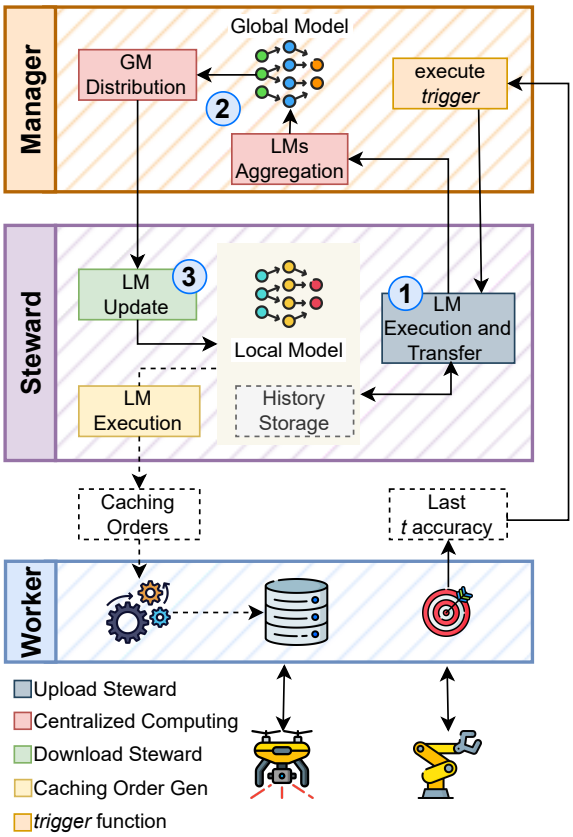


Fig. 4.12 CACHE-IT FL Federated Learning Setup

- **centralized**: when the MEL is the Cache Manager, the complete dataset is processed by the Manager Model Generation function.
- **local**: when the MEL is the Cache Steward and each local dataset is processed by the Steward Model Generation function.
- **federated**: when the MEL is the Cache Steward and the Stewards and the Manager collaborate to create the model through FL.

We aim to quantify the trade-offs of each CACHE-IT FL configuration regarding privacy levels and performance. In the next subsection, we will delve deeper into the specifics of the learning model, providing a comprehensive understanding of its architecture and functionality.

We modeled a scenario where clients' request pattern is associated with their distance from a Point of Interest (PoI), assuming a direct correlation between their location and their request frequency. This assumption is based on the observation that clients near a certain PoI tend to have similar data needs and request patterns. In our experiments, we randomly

placed a single PoI in the area dimensions, that act as hubs that attract clients' requests. The closer devices are to a PoI, the more frequent their requests become. To perform the experiments, we utilized the open-source IoT caching simulator presented in Section 4.2 which was modified to simulate the pattern of requests described, based on the distance to a PoI.

Table 4.6 lists the parameters kept constant in all experiments. An important parameter is the data expiration time, which defines when the cached resource becomes invalid due to modifications in the request content in the provider. For simplicity, we model it as a constant time. Table 4.7 depicts the factors and levels utilized to understand the effect of different configurations. Different experiments vary all levels and factors, with 30 replications, and results are presented with asymptotic confidence intervals at the level of 99%.

During the experiments, we collected metrics of three categories:

- **Model Analysis Metrics**: we focused on comparing the Mean Absolute Error (MAE) across the three caching strategies for the different system configurations.
- **Caching Related Metrics**: the set of domain metrics extracted from the simulator, which are the average AoI and latency per client request and the cache hit rate of the system.
- **Continuum Evaluation Metrics**: we analyzed the data transferred between edge and cloud in all configurations to understand how different strategies can impact bandwidth-limited scenarios.

Table 4.6 Experiment Parameters

Property	**Value**
Experiment Duration	12h
Area dimensions	1 km^2
Edge storage size	4 GB
Number of clients	100
client Speed	10 m/s
cache expiration time	100s
N_{back}	10

Learning Model and Caching Strategy

The learning model in the CACHE-IT FL architecture is designed for adaptability and robustness. Its main role is to forecast caching orders using historical data and real-time

Table 4.7 Factors and Levels

Factor	Level
Caching Strategy Category	centralized, local, federated
Number of Edge Nodes	3, 6, 9, 12

location from client requests. We designed a strategy that forecasts the next user requests and proactively caches the anticipated resources, reducing the need to fetch real-time data from remote servers. The steps involved in this caching strategy are the following:

- *Data collection*: gather historical data from the `logs` of the Cache Stewards, including intervals between client requests and their geographical locations which – in our use case, we modeled a location-based service, which provides the advantage of utilizing location information to implement proactive caching. For the **centralized** strategy, the model has access to the collects data from all the Stewards, thus accessing the data from all clients. In contrast, for the **local** and **federated** strategies, the model only has access to local data, specific to a subset of clients. This distinction ensures that the centralized strategy benefits from a holistic view, while the local and federated strategies operate with more constrained, edge-specific datasets.
- *Forecast*: use the ML model to forecast a client's next request.
- *Resource caching*: based on predictions, proactively cache resources that are likely to be requested next.

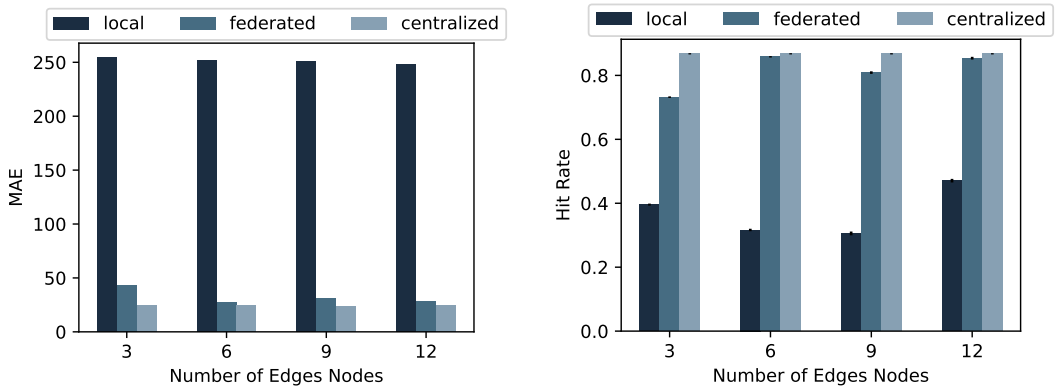
It is important to clarify that the goal of our analysis is showcasing the `CACHE-IT FL` benefits rather than creating a novel neural network design. While we provide a general architecture for the neural network, we remark that any ML technique can be used depending on the specific application. Our findings center around the effectiveness of `CACHE-IT FL` as a system and the impact of its different configurations in the system performance. The model used for the analysis is a feed-forward neural network whose architecture includes:

- *Input features*: divided into two categories. The first one comprises the historical data that consists of intervals between requests from a specific client. The model considers N_{back} historical intervals. The second category is the geographical location of the client request, represented by two features – latitude and longitude.
- *Output value*: the model predicts a single continuous value as a regression task, representing the time of the next request for each client, which latter are transformed in a set of caching orders transferred to the Cache Workers.

Results

Figure 4.13(a) depicts the MAE for each configuration. As expected, the local configuration has worse performance than federated and centralized since it only has access to a subset of the dataset. Remarkably, the federated solution achieves results that closely match those of the centralized approach but without sharing clients' data.

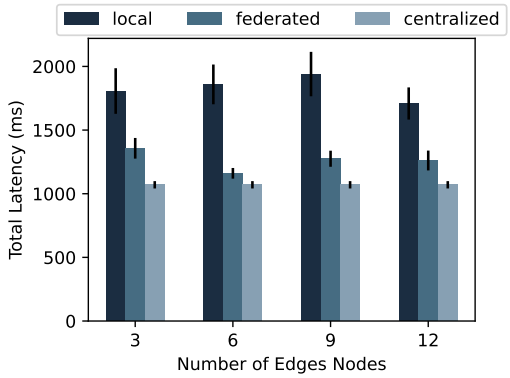
Regarding caching metrics, Figure 4.13(b) illustrates the cache hit rate while Figure 6.5 depicts the average client latency. The two metrics are correlated since the bigger the hit rate, the lower the latency, as more requests are returned from the local cache rather than from the provider. The same performance pattern from Figure 4.13(a) can be observed in these experiments since the accuracy in forecasting requests directly impacts the cache hit rate. However, this pattern is not observed in Figure 4.13(d), which depicts AoI box plots of the aggregate replications. These results demonstrate that the experiment setup does not impact AoI distributions. This behavior is due to the cache expiration time set in the Caching Template, which constrains the AoI to a maximum value. The cache hit rate results are also connected with the cache expiration time since increasing it affects the time that a request remains in the cache. In turn, this increases the probability of getting data retrieved from the cache.



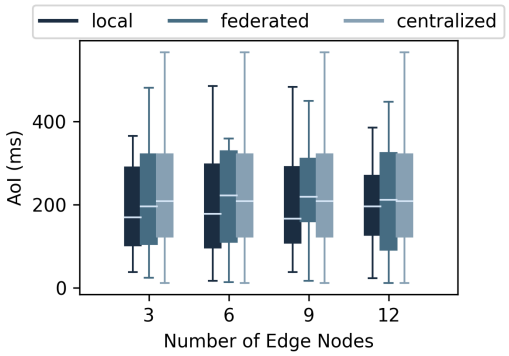
(a) CACHE-IT FL Mean Absolute Error

(b) CACHE-IT FL Cache Hit Rate

Finally, Figure 4.13 shows the network impact of federated and centralized approaches – local is omitted since there is no traffic between edge and cloud in that configuration. We considered the amount of bytes transferred for the strategy and not the data traffic that is generated by requests from clients toward providers. For centralized, the amount of transferred data is constant as it represents the request history of clients being transferred to the cloud – the number of clients is constant. On the other hand, federated linearly scales according to the number of edge nodes since each edge node needs to transfer its model to the



(c) CACHE-IT FL average client latency



(d) CACHE-IT FL consolidated AoI distributions

Manager in the cloud and download the centralized model from it. Other factors that impact the data exchange between edge and cloud in federated mode are the number of epochs and the model size. Noteworthy, the amount of data transferred in all configurations is considered small when compared with the capabilities of existing network technologies.

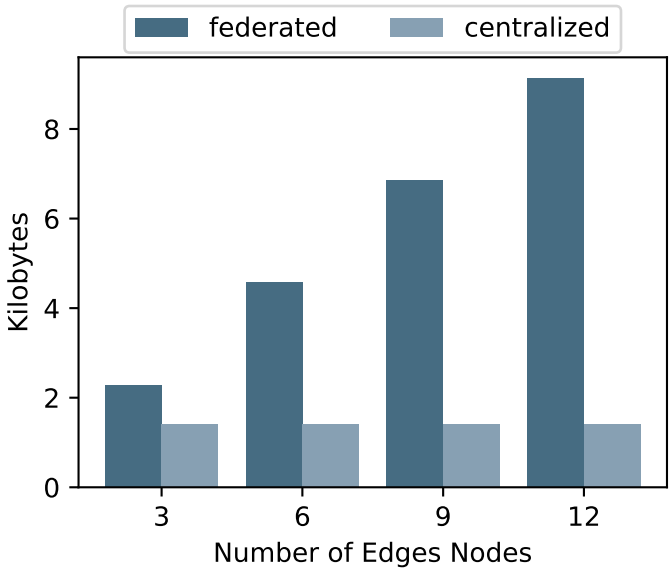


Fig. 4.13 CACHE-IT FL Edge-Cloud Bandwidth

In summary, this study has evaluated different CACHE-IT FL configurations in various scenarios. The local configuration showcases inherent privacy advantages but at the expense of performance, while the centralized approach leans towards superior performance with limited privacy preservation. Notably, the federated approach offers a balanced compromise, maintaining client data privacy while achieving performance results closely aligned with the centralized configuration. This flexibility underscores CACHE-IT FL's applicability to

a range of use cases, enabling organizations to tailor their caching strategies to specific privacy-performance trade-offs while exploring the edge-cloud continuum computation features.

Chapter 5

Services Layer: Trustworthiness in the IoT Edge-Cloud Continuum

This Chapter addresses the RQ (iii) of the thesis, namely: *How can decentralized systems be integrate to enhance the trustworthiness of data collected from diverse IoT devices in scenarios demanding reliability?* To tackle this issue we integrate blockchain technologies in the IoT domain. Since centralized architectures face challenges of transparency and the creation of data silos, the objective is to create a distributed and trustworthy system that allows clients to pay for data, and device owners are rewarded for providing it, based on the data quality of the returned valued. The system leverages interoperability solutions presented in Chapter 3, such as the W3C WoT and ZION, to seamlessly connect devices and applications. Our solution is implemented through the Service layer of our proposed architecture, functioning as a bridge that enables transparent interactions between the architecture and its users. This layer provides integration with external systems and it offers graphical user interfaces (GUIs) to enhance the user experience within in the ecosystem. The implementation of visualization tools lacks a dedicated section in this Chapter due to its limited scientific contribution.

In the subsequent sections, Section 5.1 introduce the readers to the fundamentals of the blockchain technology that overlap with IoT and the current state-of-the-art of oracle architectures to collect IoT data, while Section 5.2 delves into our architectural integration proposed.

5.1 Background

This section provides the background for contributions related the blockchain integration to provide trustworthiness to IoT. We shortly explain the basis of blockchain technology and we delve into the main oracle architectures for IoT systems.

Blockchains are innovative technologies to store and share data between a network of nodes without relying on a single centralized authority. They are instances of Distributed Ledger Technologies (DLTs) that use cryptography to create a secure, immutable, and transparent record of transactions. These technologies have undergone substantial changes recently, evolving dramatically from the first Bitcoin-related implementations. Blockchains are an innovative way to store and share data between a network of nodes without relying on a single centralized authority. Blockchains are Distributed Ledger Technologies (DLTs) that use cryptography to create a secure, immutable, and transparent record of transactions. This technology has undergone substantial changes recently, evolving dramatically from its first Bitcoin-related implementation. Among the most significant advances, we cite the emergence of blockchain-based platforms such as Ethereum [143], which introduced the concept of "smart contracts" as executable programs that enforce an agreement between two or more parties in a secure and verifiable way. Thanks to their pre-defined functions, they can store information, process inputs, and write outputs. Their capabilities have opened the doors to exploring new synergies not only related to decentralized finance but to other sectors such as supply chain management, insurance, voting systems, health care, and IoT.

Smart contracts only have access to data stored on the chain. This limitation is because their execution must be deterministic to be fully verifiable by other nodes in the network. Therefore, injecting external data into the blockchain requires an off-chain component, the *oracle*. An oracle is a software entity capable of retrieving external data and making it available on-chain for smart contracts[24]. Hence, the oracle is not the data source *per se*, but the layer that queries, verifies, and authenticates external data sources and then forwards such information. In other words, it is a bridge connecting the blockchain to the outside world, enabling the consumption of data from APIs, sensors, devices, and more. The use of oracles brings back the centralization that blockchain was supposed to remove from the equation, reintroducing issues related to architectures with a single point of failure (e.g., bringing corrupt, malicious, and incorrect data to the chain). This dilemma is known as *"The Oracle Problem"*. It deals with the issue of finding a balance between efficiency and decentralization when data is retrieved from the outside through these systems [144]. Numerous approaches have been proposed to solve this problem, and they can be grouped into two macro-categories: centralized and decentralized solutions.

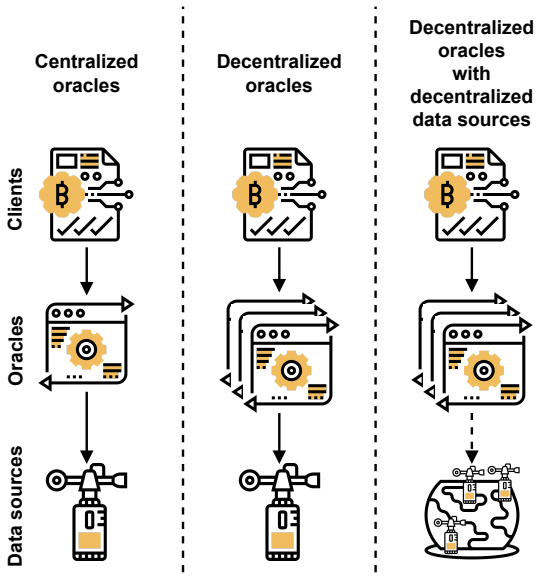


Fig. 5.1 Different oracle architectures and their relationship with data sources

Examining Figure 5.1, we can notice that a centralized oracle is based on a single-server architecture and relies on a single data source. Typically, these solutions employ Trusted Execution Environments to secure the critical processes of oracles, in combination with technologies, such as TLSNotary [145] , for generating proofs of data authenticity. An example of an oracle that leverages this type of architecture is TownCrier[146]. On the other hand, a distributed oracle is implemented through multiple nodes providing data to the blockchain. Each node usually relies on one or more specific data sources to fulfill the requests. Witnet[147], for instance, is a decentralized oracle network based on reputation points. Augur[148] is another reputation-based decentralized oracle designed to resolve the outcome of markets. ChainLink[149], for instance, is a decentralized oracle network based on reputation mechanisms and can be considered a general-purpose system. ASTRAEA[150] uses a voting mechanism to establish the authenticity of a specific assessment: submitters enter propositions into the system, while voters and certifiers play a game to determine the truth value of each proposition. Aeternity[151] is an oracle network that uses a system of stake-weighted voting to ensure the reliability and accuracy of the data provided by the oracles. Although some of these oracles can be used to retrieve IoT data, there are specialized solutions that are designed for this task. DiOr-SGX [152] uses multiple oracle servers to minimize the risk of a single point of failure while ensuring data integrity. STB[153] is a distributed and hierarchical blockchain architecture with a peer-to-peer oracle network. It has a lightweight consensus algorithm for IoT-constrained devices and specialized components for scaling and verifying the reliability of external information before storing them on-chain.

Finally, OIB[154] is a system to facilitate the deployment of smart contracts-based Industrial IoT applications. The core of the system lies in a distributed oracle network that extends the computing capabilities of the contracts.

Looking at the solutions presented, it is possible to realize how the risk of centralization is always around, especially when it comes to selecting and managing data sources[24]. Furthermore, in the specific case of oracle architectures for the IoT, the heterogeneity of devices is not considered, even if a clear methodology for integration and communication would be needed, since IoT devices do not usually have compatible interfaces with current Web technologies, as they employ a different stack of protocols. This Section advances state-of-the-art by proposing a novel, fully decentralized oracle system specialized for retrieving IoT data. Differently from the cited studies, our solution detaches the oracles layer entirely from the data sources, pushing to the limit the concept of decentralization and trust. Our objectives are threefold: (*i*) maintain the highest possible level of decentralization of the system enabling multiple oracle nodes to retrieve data from a set of different data sources each time; (*ii*) treat the data sources as first-class citizens of the system, keeping track of their reputation and rewarding them for the provided data; (*iii*) unify device discovery and communication through a well-defined interface that can serve as a homogeneous abstraction layer for the various actors in our system.

5.2 Blockchain-based Oracle Architecture for IoT

A decentralized IoT global market would allow the end users to easily and seamlessly gather data from IoT devices connected to a global network. Additionally, anyone could effortlessly connect its device and expose its capabilities throughout the network – monetizing its usage [155]. For instance, smart insurance companies can utilize reliable sensory data as automatic triggers to release contract compensation, inherently benefiting from blockchain security and fraud-proof features – further description of this use case is provided later in the manuscript.

Many approaches have been proposed to fulfill such a vision. However, they all share similar pitfalls. On the one hand, centralized approaches inherently bind users to trust a single entity and to stick to specific standards and technologies[24]. This unavoidably creates silos that are hardly interoperable with one another and do not guarantee the necessary transparency to the final users. On the other hand, decentralized solutions fail to provide the required trust for users to consume data. Trust plays an essential role in the IoT global market since clients need to ensure the quality of the queried data.

Blockchain technology has the potential to become a pivotal enabler for a global IoT market[153], as it creates trustworthiness in totally decentralized systems by sharing among

all the nodes of a network a single and immutable history of transactions. However, many challenges still impede unleashing the full potential of blockchain for IoT-based applications. In this Section, we tackle four of them:

1. IoT devices have limited processing and storing capabilities and are often battery-powered, imposing energy efficiency constraints. Thus, it is unfeasible for IoT devices to be blockchain network nodes since they cannot spend energy and computation to verify other transactions and cannot store the transaction history. Furthermore, blockchains themselves are not able to actively access external IoT data.

2. IoT devices are unreliable per design; they are made to be numerous, inexpensive, and interchangeable. Moreover, they could be more susceptible to tampering than traditional devices since computational-intensive security mechanisms cannot be supported. Hence, a substantial overhead – imposed by the blockchain – is in place to query unreliable data from cheap sensors.

3. A common and well-known problem of IoT is the lack of interoperability. IoT devices comply with several different communication protocols, data structures, and interfaces, which contribute to a fragmented landscape that hinders the adoption of IoT global markets.

4. Blockchain and related consensus mechanisms rely upon deterministic outputs. IoT sensor measurements are inherently nondeterministic, meaning that IoT data sources may return different results even if operating in the same conditions. This calls for an accurate method for selecting multiple data sources and safely aggregating their results to ensure trustworthiness.

To address such challenges, we propose DESMO, a novel architecture – founded by the ONTOCHAIN European project [1] – to enable an IoT Global Market based on distributed data oracles paired with decentralized IoT data sources. *Oracles* are special applications that connect blockchains to the off-chain world[154]. In order to increase their trustworthiness and maintain decentralization, we adopted a layer of distributed oracles and enabled the client applications to retrieve data from multiple data sources that share the same features in a specified geolocation. We need to trust not only the oracles but – and mainly – the data coming from the IoT devices. For this reason, our architecture includes reputation algorithms for the ranking and automatic selection of trustworthy data sources. DESMO is able to connect with the reference IoT architecture since it is a integration on the service level. Finally, to address IoT inherent heterogeneity, we utilize the the W3C WoT [156].

[1] https://ontochain.ngi.eu/

In the following subsections, we provide further details of our proposal. We outline its unique features and technical characteristics compared to existing blockchain solutions. Next, we describe its architectural design and showcase two applications enabled by the DESMO IoT global market. To exemplify its operation, we conducted a case study that demonstrated the system's robustness to malicious nodes while maintaining data quality.

5.2.1 Architectural Design

In this section, we analyze the DESMO architecture and how it supports the concept of global IoT market. As we will see, the DESMO architecture spans from on-chain components (smart contracts) to off-chain components (oracles, indexers, and data sources). All these concur in achieving a fully decentralized system organized in the following layers:

- The **Clients** are the buyers of IoT data and can be on-chain or off-chain components.
- The **DESMO Protocol** layer is composed of smart contracts that register requests, store responses and payments and detain the reputation ranking of data sources.
- The **Decentralized oracles** layer collects the requests from smart contracts and queries the designated sources.
- The **Decentralized IoT data sources** layer contains the devices that provide the data and the directories that index them. Users can register new directories by staking tokens. In this way, if a directory behaves fraudulently, the protocol can punish it by draining from these funds.

As illustrated by Figure 6.9, the top layer of the platform contains the clients, which can be different: smart contracts – e.g., protocols – that need to access IoT data to perform automatic actions, Dapps implementing IoT use cases, server-side applications in need of retrieving IoT data in a trusted and distributed way. Immediately below, we find the DESMO distributed protocol, a set of smart contracts that cooperate to manage the various aspects of the system. The Portal contract is the entry point of the client, receiving requests for data and initiating the data retrieval process. The Hub is the repository for registered data sources and their reputation score, which quantifies the trustability of each source. Finally, the Token contract (omitted in Figure 6.9) holds the currency that powers the system's economy. The amount paid by each client is distributed between the oracle nodes and the data sources participating in the process. The oracles layer contains the worker nodes capable of executing the DESMO oracle application, which queries a defined set of data sources and computes

a result to the client request through a consensus algorithm between oracles, as explained later. The bottom layer includes the data sources, and its structure is divided into two distinct blocks: directories and end devices. Directories are responsible for indexing the metadata of physical devices and making them discoverable via a well-defined interface that supports both semantic and geo-spatial queries. On the other hand, to be compatible with the system, devices must expose a semantic descriptor that allows both directories and oracles to interact with them. We decided to use the W3C WoT standard in our implementation as it provides a homogeneous interface to access IoT devices that abstract from their particular interfaces and heterogeneous network protocols. Specifically, we adopted in the system architecture W3C WoT directories to index the devices, they have to implement the W3C WoT Discovery specification[156] with the addition of support for spatial queries – ZION is an example of a suitable TDD. To store off-chain, accessible, nondeterministic data we make use of IPFS (InterPlanetary File System) [157]. IPFS is a decentralized, peer-to-peer file system that provides storage and retrieval of data on the Internet. It replaces traditional methods with a content-based addressing system for files and their versions. IPFS allows for saving arbitrarily big files and, by writing only their hash on the actual blockchain, we ensure always to store a constant amount of data on-chain.

The complete flow of a single data request is then depicted in Figure 6.9.

Step 1 A client submits a new request to the system by calling a function of the Portal smart contract. The payload of a request includes the semantic query for identifying the desired data type and the geospatial filter – e.g., the temperature in the metropolitan area of New York City.

Step 2 The Portal contract asks the Hub for the TDDs to associate with the request. TDDs, as said, can become part of the system (i.e., included in the list detained by the *Hub*) by staking a certain amount of tokens. Each of them starts with a neutral reputation value – we set it to 0 – and gets selected by the Hub to reply to a data request through a round-robin selection process. The process uses reputations as weights, so high-reputation TDDs are more likely to be selected.

Step 3 and 4 A subset of oracle instances takes on the request and queries the selected TDDs. It is essential to highlight that each oracle makes the same query on each of the elected TDDs and collects the descriptors of all devices that semantically match the request.

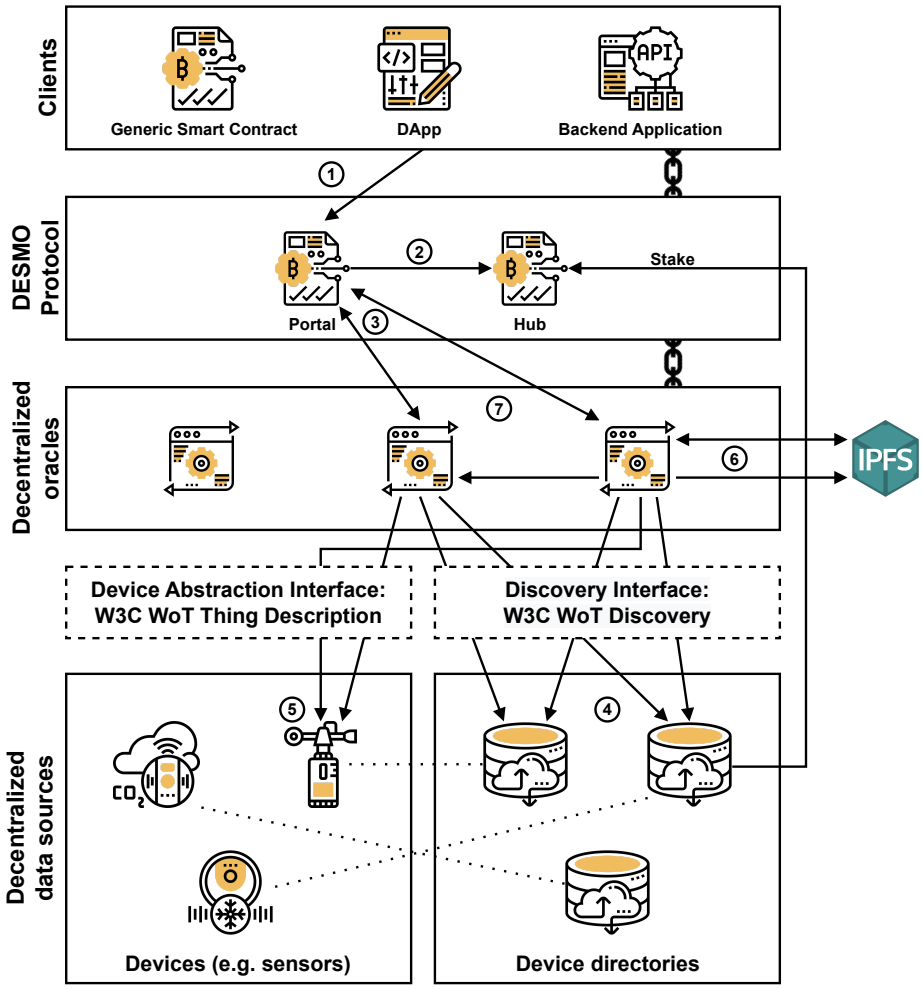


Fig. 5.2 The *DESMO* layered architecture with the steps required in the query resolution process

Step 5 The oracles retrieve sensor data from the end devices and need to reach a consensus on which data point best represents reality and how good and reliable the sensor data is. The process that implements the above actions (the "consensus process") is depicted over five phases in Figure 5.3, and supports the explanation of steps 5, 6, and 7 of the query resolution process. We can abstract from the concept of TDD and assume that each oracle queries the same data sources – i.e., the sensors – and obtains from each of them a single data point corresponding to the sensor reading. In Figure 5.3, data points generated by different sources – phase I of the consensus process – are depicted as squares of different colors. In this case, each oracle, within a single request round, ends up with as many data points as the number of queried data sources, represented in phase II of the consensus process. Note that two oracles may obtain two different data points from the same source. This inherently belongs to the nature of IoT and can be dictated by several factors: for instance, oracle queries could have

taken place at two different moments in time, thus triggering two different sensor readings, causing nondeterminism. Nonetheless, within a single request round, we expect a data source to reply consistently to multiple oracles, meaning that data points shall differ negligibly. The subsequent steps will address the nondeterminism problem.

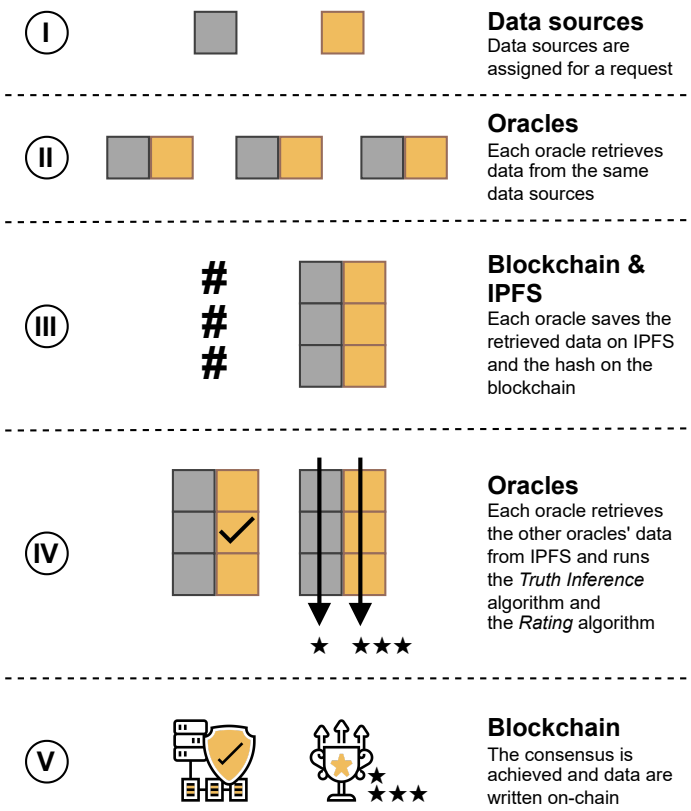


Fig. 5.3 DESMO data gathering and consensus process.

Step 6 Oracles store the reply they obtained from data sources onto IPFS. In phase III of the consensus process, the collection of replies on IPFS is represented as a matrix, where each row is associated with a single oracle and each column with a single data source. Upon saving the reply on IPFS, each oracle also hashes its content and saves the hash onto the blockchain. This ensures the amount of space that a single client request occupies on-chain to be solely dependent on the number of selected oracles (which is constant), not on the number of queried sources (which can be arbitrarily high, depending on how much the client pays). In phase IV of the consensus process, oracles can use the hashes stored on the Portal to retrieve the full matrix from IPFS. At this point, each of them executes two actions: (*i*) The **Truth Inference algorithm**, which outputs a single data point in the matrix – the "inferred

truth" – that is believed to be the closest to the ground truth.; (*ii*) The **Rating algorithm**, which outputs a score for each data source on top of their output with respect to the inferred truth. The score will be integrated with the overall reputation of the data source. The Truth Inference algorithm is a function τ that takes in input the matrix of results M and returns a single sensor reading $t \in M$. The Rating algorithm is a function ρ that takes in input M and returns an array of n scores S, where n is the number of queried data sources. Each score must be constrained to a definite interval, in our case we defined scores to be between -1 and 1.

Step 7 Finally, in phase V of the consensus process, oracles need to reach a consensus by all executing $\tau(M)$ and $\rho(M)$ and perform majority voting, then the two results are written on the Portal contract. Note that both functions are deterministic, which means that, if all oracles execute them onto the same matrix M, they should end up with the same results. The Portal then stores the result of $\tau(M)$, which is the reply to the client request, and uses the result of $\rho(M)$ to update the reputation of the sources in the Hub contract. This is done, for each source, by linearly combining its score with its previous reputation, which is constrained between -1 and 1 as well, in order to obtain again a reputation value between -1 and 1. Upon performing this operation, the Hub contract may choose to blacklist a source, if its reputation falls below a certain threshold. This operation, as we will see in the section about the Case Study, is necessary to mitigate the chances for attacks as well as untrusted or defective data sources.

The interaction of DESMO with the blockchain is a crucial aspect of its architecture, specifically regarding the transactions generated from data queries. When a user makes an initial query, both the query and the final result are recorded on the blockchain, resulting in two separate transactions. Further, each oracle involved in a query stores the hash of the queried data on the blockchain to make it available for other oracles to calculate the Truth Inference algorithm – as outlined in *Step 6*. The output generated by each oracle is also recorded in the blockchain to enable the process described in *Step 7*. Current scalable blockchains have a high throughput of thousands of transactions per second (TPS) and are pushing to keep increasing those numbers – e.g., Solana can process more than 8,000 TPS [158]. Considering those numbers and an average of five oracles involved in each query, *theoretically* DESMO could support millions of daily queries. A first implementation of the different components of the architecture can be found on GitHub[159].

5.2.2 Use Cases

In this section, we explore new applications that are enabled by DESMO. We selected one insurance scenario that leverages sensor data and one urban scenario related to noise pollution.

Smart Insurances

Currently, insurance claims are a long and costly process. Insurances companies desire to combat fraud, at the same time, to reduce time and costs by automating the claim-related administrations and execution process. On the other hand, customers want rapid compensation and clear insurance terms. A trustworthy automatic process for insurance will be beneficial for both parties.

An example application enabled by DESMO is insurance in the intelligent farming domain. Figure 5.4 depicts an example application in the intelligent farming domain enabled by DESMO. In the illustrated example, suppose that a given company commercializes insurance for farmers to protect their crops against extreme environmental hazards – *e.g.*, storms. The customer and the company agreed on specific terms to trigger the farmers' payment. Those conditions and their associated triggers are established on a smart contract that acts as a client of DESMO – interacting with the Portal contract. –as the first step in Figure 5.4 showcases.

The insurance company smart contract defines features – *e.g.*, wind speed , temperature – to be queried in a specific geolocation without specifying the set of devices that provide such data. The data sources providers could be both sensors deployed by a trusted partner of the insurance company, as well as already deployed devices in the farm and nearby locations – as step 2 of Figure 5.4 depicts, in which three sensors act as the data source, two deployed in farms and one in the wilds that was deployed for a different purpose but still provides the requested features. DESMO Truth Inference and Rating algorithm will assure data quality and prioritize the selections of trustworthy data sources to avoid malicious users tampering with the system – *e.g.*, falsifying a storm's conditions. In case the conditions are met, the smart contract automatically releases the tokens (i.e., funds) to the farmers to keep their businesses running. – step 3 of Figure 5.4.

Figure 5.4 illustrates an example application in the intelligent farming domain. First, the farmer and the company established the policies of a smart contract based on the damage that a crop is likely to suffer under specific environmental conditions – e.g., 100 mph winds – and the appropriated values to be paid in case of damage. Then, the smart contract deployed on-chain reads sensor data through DESMO, and in case of the conditions are met, the

farmers automatically receive the funds to keep their business running. There are other initiatives of blockchain-based insurance that would also leverage our system from different domains, such as transportation[160].

Urban Noise Pollution Monitoring

Noise pollution is a common hazard in urban environments, and its sources range from social activity to construction. It is estimated that the cost of noise-related health issues in the EU is between 0.3% to 0.4% GDP[161]. Public authorities commonly incentivize noise mitigation by imposing fines on noise emitters above a certain threshold. However, it is challenging to enforce noise regulations in large urban environments. We envision such systems to leverage the usage of DESMO. The appointed public authority would hire several small and medium enterprises to cover a city with noise monitoring sensors. Different set companies could cover different neighborhoods, with some overlap. Another third-party actor – employed by the public authorities – would deploy a smart contract that queries the DESMO Portal contract to get noise pollution data in different parts of the city. Once a code violation is detected, the smart contract automatically warns the competent authorities to take appropriate action. The advantage of using DESMO in such a setup is threefold: (*i*) DESMO geo-spatial filter allows clients to utilize different granularity of boundaries in successive queries to narrow the precise location of the noise pollution source; (*ii*) Given the truth inference and rating algorithm, malfunctioning data sources would not hinder the overall system data quality, and their data would be requested less often. Consequently, reliable data sources would be queried frequently and – as each request in DESMO is paid – be more profitable, incentivizing the continuous maintenance of the system; (*iii*) Different companies would deploy sensors with diverse technologies regarding communication protocols and data structure. DESMO, by design, handles IoT heterogeneity via W3C WoT solutions.

5.2.3 Performance Evaluation

We developed a case study to assess the proposed system's resilience and effectiveness in a theoretical scenario in presence of malicious data sources. As we motivate our work on trustworthiness, we need to punish or ban possible malicious data sources while maintaining satisfactory data quality. The case study comprises clients interested in environmental temperature values and 1,000 sensor sources capable of producing temperature readings registered to various TDDs.

These systems can be prone to collusion attacks, where a group of malicious sources may cooperate in injecting false data on the blockchain. In our scenario, we represent this

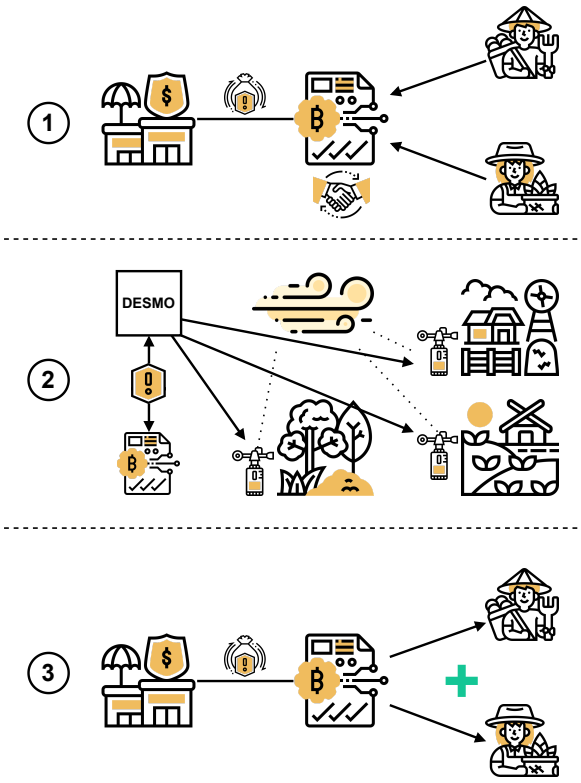


Fig. 5.4 Smart Insurance use case applied in the agricultural domain

attack as follows: the temperature ground truth is set to 25°C, and the response from the Truth Inference algorithm is considered to be *accurate* if it falls within a tolerance of ±3°C. Honest sources produce temperature values according to a normal distribution, with the mean equal to the ground truth and the standard deviation uniformly generated between 0 and $\frac{4}{3}$ of the tolerance, representing a few honest sources making occasional mistakes. We modeled malicious sources to generate temperature readings by replacing the ground truth with a fairly distant common value (10°C) to simulate a collusion attack, where all attackers concur in redirecting the output to a fake verdict. We then simulated a period composed of 5,000 epochs; a single epoch corresponds to a client request, a response by the system, and a single run of the Truth Inference and the Rating algorithms. We experimentally found that all configurations running more than a few hundred of epochs yield consistent results, hence our choice.

During the first 2,500 epochs, we assume that 50% of the sources are already registered in the system, and all of them are honest – this assumption of a "warm start" is realistic because most of the blockchain-based applications run an initial *genesis phase*, where the chain is populated with a solid history of valid blocks in a controlled way. During the remaining 2,500 epochs, we add the remaining sources, including the malicious ones. These can join

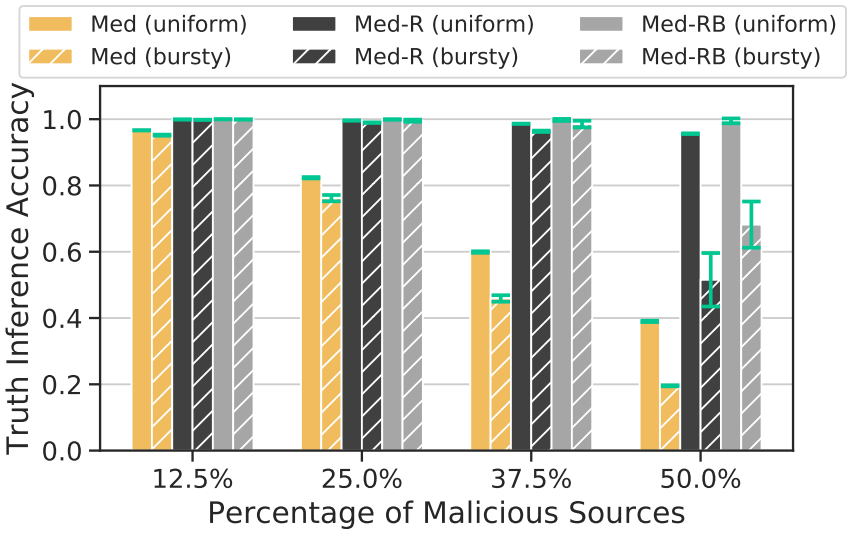


Fig. 5.5 Truth Inference Accuracy: the percentage of requests by a client that get satisfied within a tolerance threshold.

the network following either a **uniform** distribution, thus randomly picking an epoch, or a **bursty** distribution. In the latter case, we ensure that *all* malicious sources join the network at once, simulating the worst case of colluding sources operating simultaneously.

Our first experiment aims to validate the solidity of the Rating algorithm and the usefulness of blacklisting sources that are identified as malicious. For this reason, we ran simulations adopting a single Truth Inference metric: the median – i.e., we consider $\tau(M)$ to return the median of all sensor values in M, obtained with a single request. We name this baseline algorithm "**Med**". We then compare its performance against "**Med-R**", in which we additionally perform the Rating algorithm. In particular, our implementation of $\rho(M)$ outputs, for each $t \in M$, the distance of t from $\tau(M)$, normalized as $\frac{1}{1+((t-\tau(M))/TOL)^2} - 1$, to output a value between 1 and -1 (in our case, if the distance is more than 3°C, the score will be below 0), picking the minimum value for each source. Next, the reputation of each source is calculated as a convex combination of R and $\rho(M)$, tuned by a parameter α, $(1-\alpha)R + \alpha\rho(M)$ where R is the old reputation value, and α is a parameter that tunes how much the new score affects the reputation – in the simulations, we set it to 0.5. Finally, we show the performance of "**Med-RB**", where we perform both the above Rating algorithm and the blacklisting step, excluding sources with a reputation below a threshold.

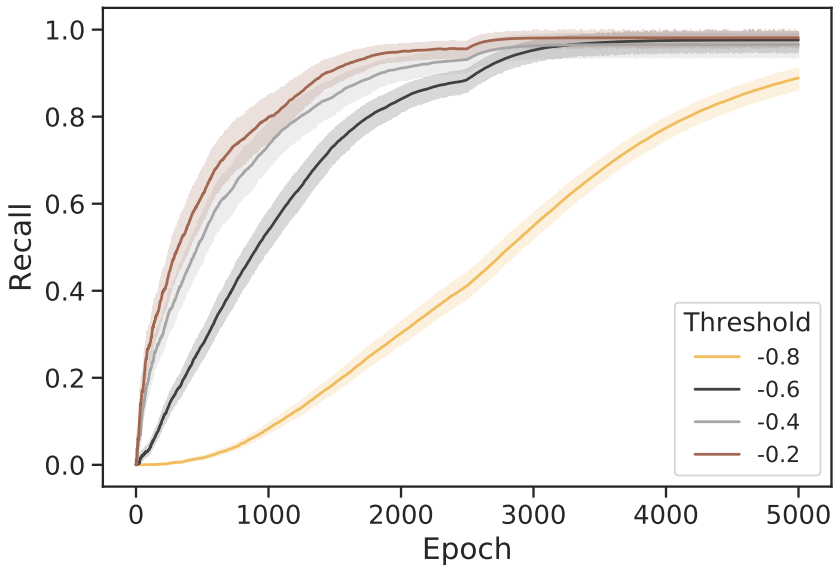


Fig. 5.6 Blacklisting Recall: the percentage of all the malicious sources that the system is able to detect and ban over time.

Figure 5.5 shows the results obtained by executing the three algorithms over different percentages of malicious sources, with the maximum being set to 50%, as a higher value, according to the well-known 51% problem, would compromise the entire network [162]. The bar chart shows the Truth Inference accuracy, the ratio of client requests that get an accurate response, for uniform and bursty arrival rates. Results show how the rating step significantly affects resilience against a certain number of malicious sources. We also expect a much better effect against defective or low-quality sources, as in our simulations, malicious sources are performing a joint attack, which is the worst possible scenario. The chart also depicts how blacklisting always has a better impact on inference accuracy. In particular, it allows the system to hold up as many as 50% of malicious sources joining all at once, still yielding a high accuracy – around 0.7. Simulations were performed taking into account different values of blacklisting threshold and multiple repetitions.

Furthermore, we study the impact of our Med-RB strategy in terms of "blacklisting precision and recall". The rationale is that different use cases may privilege certain requirements over others. For instance, it may be more important to timely get rid of all malicious sources, while tolerating a reasonable number of honest sources to be kicked out as well (high recall). Conversely, it may be affordable to keep few malicious sources, while ensuring that all banned sources are malicious (high precision). The first case is shown in Figure 5.6 over

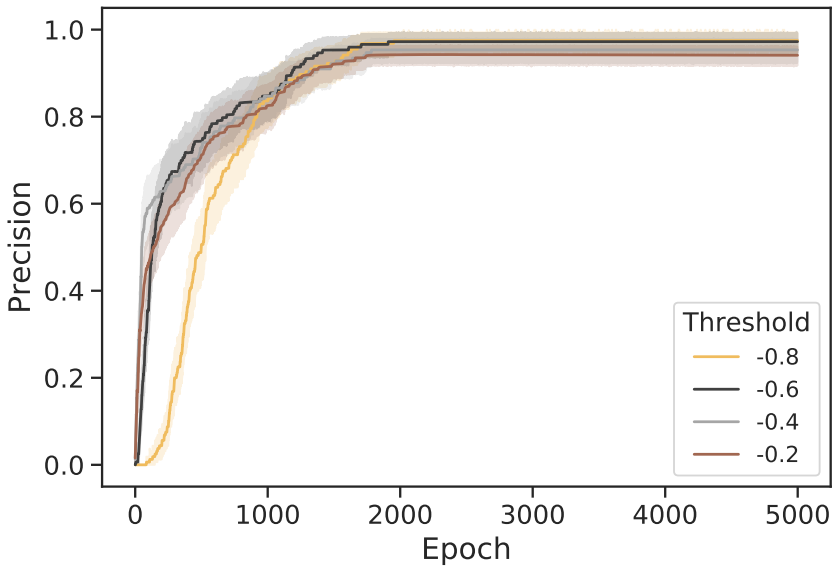


Fig. 5.7 Blacklisting Precision: the percentage of all the banned sources over time that are actually malicious.

time for different suitable values of "blacklisting threshold". We can see the recall values to be consistent with the expectations: higher thresholds mean less permissive scenarios, thus higher recall. However, we notice that only the most permissive threshold underperforms and, yet, yields a high recall value – above 0.8 – at the end of the simulation. An even better result is shown in Figure 5.7, where, no matter the threshold, the precision stabilizes between 0.9 and 1.0, with only tiny differences that are in line with the expectations: the more permissive the policy, the higher the precision and vice versa. High precision scores indicate high reliability and the ability of the technique to identify and kick out malicious data sources with high accuracy, regardless of the threshold. The evaluation confirms that Med-RB is a solid baseline for our scenario, as its only additional cost is to store the result of the Rating algorithm on-chain, which can be easily controlled by setting an upper bound to the number of selected sources. Med-RB can be used as a first building block for further evolution, including more parameters.

Chapter 6

Use Cases: Deployment in the IoT Edge-Cloud Continuum

This Chapter demonstrates the versatility and extensibility of the architecture by using it in different projects , namely: the **Arrowhead Tools**[1] and the **MAC4PRO** [25]. The use of the architecture in different conditions addresses RQ (iv): *How can we deploy the same architecture in different scenarios that have different edge-continuum configurations, while considering different system end goals and requirements?*.

Section 6.1 provides a comprehensive review of the state-of-the-art of modern SHM systems. Our literature review shows that the majority of proposals feature a tailor-made deployment for a specific use case, and the system components are bound to a specific location (i.e., the cloud or the edge). The remainder of this chapter demonstrates how the proposed architecture addresses these issues. Section 6.2 illustrates the use of the architecture in the MAC4PRO project. In the context of the project, the challenge is to deploy the same architecture across different SHM pilots, each with diverse edge-cloud continuum configurations. Section 2.2 exemplifies how the the architecture adapts to the requirements of the Arrowhead Tools project. The Arrowhead Tools project encapsulate different engineering pipelines that must be cohesively encompassed by the proposed architecture. The architecture was deployed in the Arrowhead SHM pilot to achieve the following goals: reduction of engineering costs, integration of legacy systems, and interoperability with IoT frameworks. To address these issues, we leverage most of the solutions presented in the Chapter 3 that address interoperability challenges. We also illustrate the caching solutions from Chapter 4 applied in the SHM Arrowhead pilot.

[1] `https://tools.arrowhead.eu/home/`

6.1 Background

This section provides the background on the use of IoT systems for monitoring tasks. We describe the current state of the art of IoT-based system for SHM, then we compared our solution with the others in the literature.

Several recent studies investigate the integration of the IoT paradigm for IoT monitoring applications. In addition to connecting smart sensors to the Internet, such integration consists of deploying software architectures in the continuum to collect, store, and analyze monitoring data. A reference architecture is discussed in [13], including four different components: (*i*) smart objects, (*ii*) gateway, (*iii*) cloud, and (*iv*) remote station for data access and visualization. Two IoT-SHM use cases related to the safety and protection of masonry buildings are presented, although the development of the software component is still in a preliminary stage. As previously observed, structural assessment through non-destructive techniques can involve a large amount of data collection due to high frequency sensor sampling and long measurement periods necessary for high quality information retrieval [163]. For this reason, cloud infrastructures often provide the storage and computing resources of IoT SHM platforms. In [14], the authors use the AWS IoT cloud platform to manage smart sensors installed on a single-track railroad bridge; the raw sensor data is stored using DynamoDB and displayed through a custom web interface. Similar sensor-to-cloud workflows are proposed and tested in [15] and [164]. Finally, in [17], the authors propose a Digital Twin framework for SHM that uses edge nodes for data cleaning and preprocessing tasks. However, the paper focuses on the modeling of the DT rather than the underlying platform that supports it. We emphasize that the aforementioned approaches propose architectures tailored to a specific use case; moreover, the application components are usually bound to a specific processing location - e.g., the cloud. Much less attention has been paid to the design of general-purpose monitoring software architectures that can abstract from domain-specific industrial and civil engineering information and support heterogeneous sensor devices. We review the most promising solutions available in the literature, focusing on those that address problems related to SHM (see Table 6.1), and highlight the differences with respect to our work by focusing on the following features:

- Interoperability: the ability to easily integrate dissonant interfaces and data structures from sensors and software components into the system.
- Modularity: The ability to divide the system into independent modules that can be developed, maintained, and managed separately.
- Agnostic design:

  - Infrastructure: The ability to remain agnostic to the underlying hardware infrastructure across different deployment scenarios to accommodate different edge and cloud computing node configurations.
  - Application: the flexibility to work with different software configurations without being locked into specific technologies or implementations.
- Edge-Cloud Continuum Support: The ability to work seamlessly across edge and cloud computing environments.

In their study, Seongwoon et al. (2018) [16] examine the benefits of using Service Oriented Architectures (SOA) for monitoring systems. They argue that the ability to compose independent and reusable software components is advantageous for this purpose. The architecture developed in this thesis follows a similar approach, but considers the entire edge-cloud continuum rather than centralizing computation solely in the cloud. In [165], an SHM architecture is proposed where software components are distributed across the edge-cloud continuum. They adopt a modular design that can accommodate different infrastructures. Our architecture differs by prioritizing interoperability and maintaining the flexibility of software components that are not tied to specific technology implementations. A collection and classification of several SHM systems deployed on bridges has been presented in [166]. The authors analyze the different sensing and communication technologies and the most common data processing algorithms for early warning. Although many aspects were covered, an architectural design that allows replicability of the proposed solution was not considered, nor was the potential of the edge-cloud continuum. The work in [167] presents the integration of a bridge SHM system with a BIM model, allowing a 3D bridge model to directly access bridge health data in real time. IoT sensors were deployed and communicated via WiFi to an IoT web platform. The limitation of this work is that it focuses on the BIM aspects of the system and the specifics of bridge modeling, rather than the software architecture to support it. The work presented in [168] designed and implemented an architecture to support fiber optic sensors for SHM. This architecture enables real-time data processing using an Apache Kafka-based stream processing system. Their system was custom-built to meet stringent time constraints while processing large amounts of data. As a result, the architecture is tightly coupled to its specific technological implementation and underlying infrastructure, with limited consideration for interoperability. Our approach prioritizes interoperability, which simplifies the integration of new sensors and software components and enables efficient updates. This design greatly enhances the deployment of our system across different surveillance structures, which often require different hardware and software components.

Table 6.1 Comparison of the proposed architecture to the literature

Solution	interoperability	modularity	agnostic-design		continuum support
			infrastructure	application	
[16]	✗	✓	✗	✓	✗
[165]	✗	✓	✓	✗	✓
[166]	✗	✓	✗	✓	✗
[167]	✗	✗	✓	✗	✗
[168]	✗	✓	✗	✗	✓
Our proposal	✓	✓	✓	✓	✓

6.2 MAC4PRO

In this section, we described how the proposed architecture was implemented and deployed in the context of the MAC4PRO[2] project, a research effort aimed at developing an infrastructure-agnostic and general-purpose monitoring platform for the condition assessment of industrial and civil infrastructures leveraging the ultimate technologies delivered by the Information, Software, and Industrial Engineering communities.

We present a complete implementation of the abstract architecture, illustrated in Figure 6.1, which meets the aforementioned requirements and validate the architecture versatility in two distinct experimental campaigns related to the condition monitoring of real facilities in their operative environment. The first test-bed refers to the monitoring of a concrete building during seismic events simulated through a shaking table. The second use case pertains to identifying leakage in hydraulic circuits via acoustic emissions. For both campaigns, we discuss the deployment of the architectural components in the edge-cloud continuum and present diagnostic results. Additionally, we conducted a comprehensive performance evaluation to assess the architecture capabilities.

The design of each architectural layer are within MAC4PRO are presented in their respective subsections. Thus, Subsection 6.2.1 delves into the sensing layer, Subsection 6.2.2 provides a comprehensive overview of the interoperability layer, Subsection 6.2.3 details the data management layer, and Subsection 6.2.4 expounds on the service layer. Subsection 6.2.5 presents the performance evaluation conducted for the proposed deployment. Finally, Subsection 6.2.6 and Subsection 6.2.7 are dedicated to an extensive experimental validation phase on two representative benchmark scenarios for condition monitoring.

6.2.1 Sensing layer

The *Sensing* layer corresponds to the Sensor Networks (SNs) in charge of data gathering. As a general observation, for a large-scale structure, we expect that the deployment of a

[2]https://site.unibo.it/mac4pro/

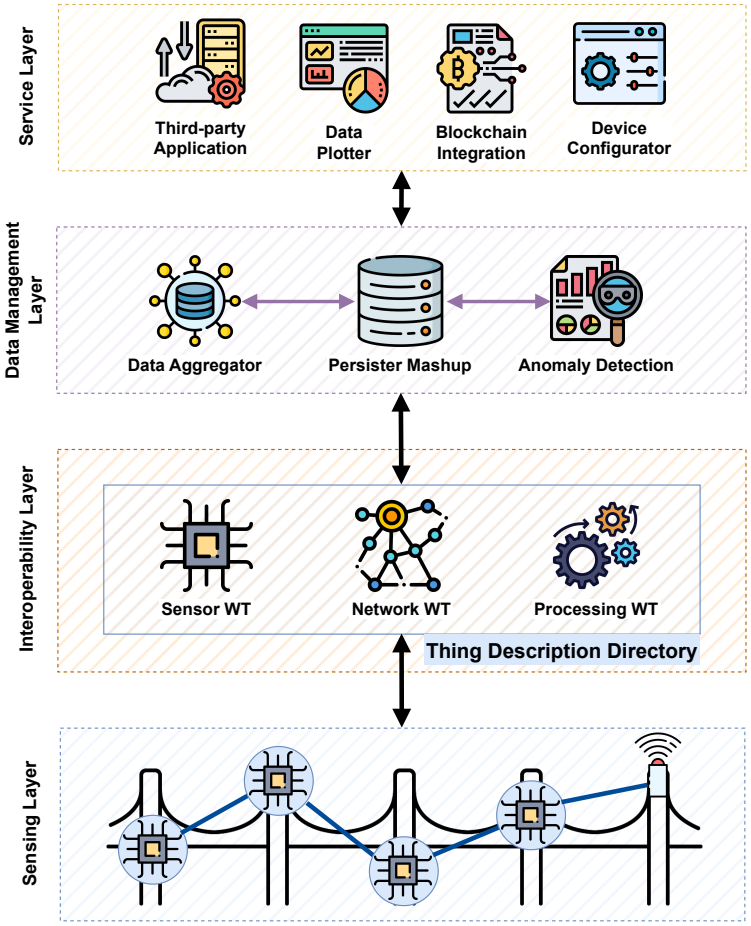


Fig. 6.1 MAC4PRO implementation of the reference architecture.

single, multi-hop SN will introduce some pitfalls in terms of both end-to-end performance and reliability, as largely discussed in the literature [169]. For this reason, we assume that the same structure can be instrumented with multiple SNs consisting of Extreme Edge Nodes (EENs), geographically isolated or with some spatial redundancy. These EENs may be heterogeneous regarding sensing and computational capabilities, communication protocols, and generated data formats. This heterogeneity is important for full-scale structural inspection to overcome the limitations of individual sensing technologies and their operative ranges. To capitalize on that, we have designed a distributed SN consisting of small-footprint, low-power, and light-weight EENs, i.e., peripheral devices integrating, in a thin form factor, all the circuitry necessary for heterogeneous data sampling, conditioning, and pre- and post-processing [170].

Despite the advantages in terms of electrical characteristics [171], the designed EEN is unique in that it offers computing functionalities implementing sensor-near feature extraction for structural diagnostics. The embedded algorithms comprise, among others, an exhaustive list of parameters (e.g., amplitude, energy, count) for acoustic and vibration data processing.

6.2.2 Interoperability Layer

The *Interoperability* layer allows the platform to abstract as much as possible from the characteristics of the sensing units. The interoperability support takes advantage of the WoT standard. In MAC4PRO, the *Interoperability* layer is composed of three classes of WTs as shown in Figure 6.1:

- *Sensor-related* WTs. We associate a WT to each sensing unit of the SN, exposing the data produced from that EEN as readable properties, the configuration settings as writable properties, and supported commands as actions. For instance, for the case of tri-axis accelerometers illustrated in Section **??**, the properties include the raw signal values in each direction and the sampling frequency, while the actions include the possibility to turn on/off the data acquisition on a specific axis. Thanks to the WT abstraction, we utilize the MODRON platform [**?**] to establish a bidirectional, logical communication channel with each device of the *Sensing* layer through a uniform and well-defined software interface. MODRON is an IoT platform developed by researchers at the University of Bologna. It provides features such as automatic integration with WTs, time-series storage, and visualization.

- *Network-related* WTs. We assign a WT to each SN, modeled as a whole. In such case, the WT includes links to the *Sensor* WTs composing that SN. In addition, it may expose aggregated properties (e.g., the average network performance) and global

commands (e.g., turning on/off the SN by issuing the same command on each sensor WT).

- *Processing-related* WTs. We associate WTs with software tasks in charge of processing the sensor data, extracting second-layer information from the monitored structure, and enabling error-handling capabilities. In such case, the WT is not connected to any physical device but acquires data from multiple Sensor WTs, acting as virtual sensors. However, since it exposes a TD with its own properties, actions, and events, it can be displayed and controlled by tools such as the Device Configurator as a real device. For instance, the implementation of vibration data analysis tasks, such as natural frequency identification, can be provided with a dedicated WT, acquiring data from Sensor WT (i.e., the accelerometers) and providing peak spectral values in output as TD properties.

The users and applications must be able to discover the WTs of an SHM scenario in order to interact with them. To this aim, the *Interoperability Layer* includes a TDD (namely, ZION), which is a register of available WTs that provide search capabilities upon the metadata description of the devices (i.e., their TD) to the upper layer.

6.2.3 Data Management Layer

The *Data Management* layer collects the SHM data, aggregates it, and supports the analytics. To this aim, it leverages the facilities of the underlying *Interoperability* layer, specifically the WT abstractions, to gather data from heterogeneous sensing units. The data acquisition is performed via the *Persister Mashup* through a three-stage procedure called *task*. First, a task retrieves the TDs of the WTs of interest from the TDD. Then, it establishes a direct connection to each WT. Finally, it issues a sampling sequence of actions and saves the returned data to the designated databases. The user can fully configure each task through a REST API; for instance, it is possible to specify the start time and frequency of execution, as well as its type (one-shot or periodic). In addition, the user must indicate the data source(s) to be queried (i.e., the WTs), the interaction affordance necessary for sampling the sensory value (e.g., a property), and a proper mapping function if the received data must fit a predefined database scheme.

Finally, SHM data are stored in the target database. To increase performance and scalability, the process of persisting SHM data to the target database(s) is queued during execution and the queue is processed using the maximum number of available threads. It can also be distributed by splitting the workload across multiple instances, allowing tasks to be parallelized, increasing system throughput and reducing processing time.

The collected data are immediately available to the *Service* layer above and, simultaneously, are also used by the other two components on the *Data Management* layer shown in Figure 6.1. The *Data Aggregator* extracts features from the time-series data retrieved from the *Persister*. It includes standard statistical aggregation methods (e.g., mean, maximum, and minimum of a time-series), allowing for easy integration of context- and sensor-dependent aggregation methods through an extendable interface. The extracted features are stored to be quickly accessible for later processing or visualization steps.

The Data Management pipeline includes collaboration between the Data Aggregator and the Anomaly Detection module: the first extracts features that the second utilizes to perform its computation. Both components use raw and feature data to assess the condition of the monitored structure, detect anomalies, and provide insights for maintenance operations. Different technical appliances may require specific analysis strategies and models to identify defects accurately; for this reason, the component is designed to be highly modular, allowing for the dynamic loading of new algorithms. This need for modularity is justified because the resource utilization of the involved diagnostic algorithms may vary depending on multiple causes: the nature of processed waveforms (vibrations vs. AEs) and the global/local value of the computed damage-sensitive features. Global damage indexes are employed for the analysis of vibration data, which can be computed only in a post-processing phase upon aggregating EEN-related information. Therefore, vibration data's data aggregation and anomaly detection components work asynchronously, with periodic queries extracting data from the databases, processing it on the cloud, and storing the computed outputs. Conversely, local processing is suitable for AE data. AE techniques are recommended when the primary source of defects is intrinsically related to energy release. The key AE parameters for real-time assessment are signal peak amplitude, signal energy and AE count, which are strictly EEN-dependent. Time analysis of these parameters allows early detection and localization of incipient faults such as growing cracks in reinforced concrete, corrosion processes in metal structures, or leaks from pipelines [172].

6.2.4 Service Layer

The *Service* layer uses the data access APIs of the *Data Management* layer to provide user functionalities. Third parties can develop these or be custom implemented directly by the MODRON platform. The built-in services include the *Data Plotter*, MODRON's dashboard for visualizing the stored data (raw and processed), the *Device Configurator*, a graphic interface to manage devices, and the *Blockchain Integration*, which guarantees data transparency and immutability.

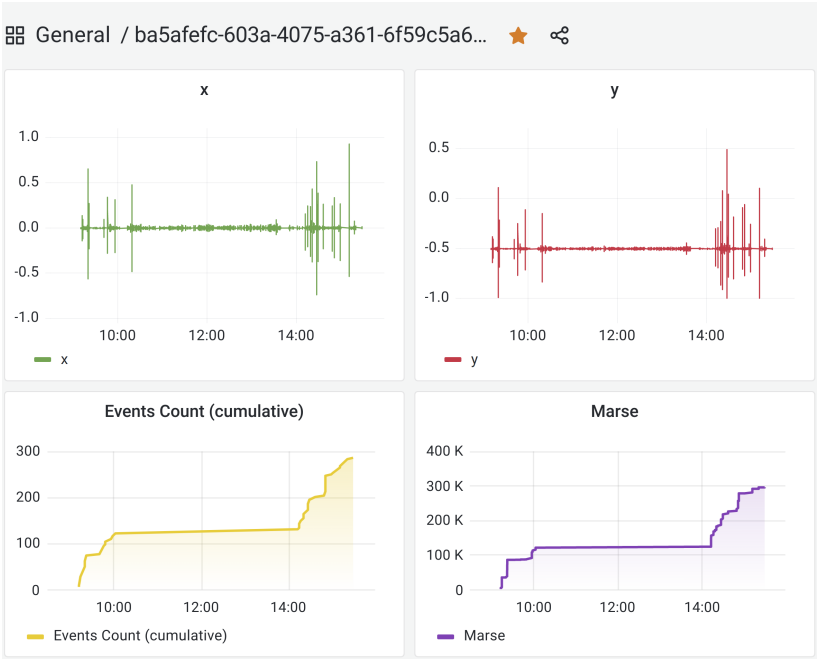


Fig. 6.2 The *Data Plotter* depicting vibration and AE sensor data from the experimental campaigns.

The *Data Plotter* offers a wide range of functionalities allowing users to create custom charts, filter data based on the WTs time interval, and apply various transformations to the displayed data. A representative screenshot of the *Data Plotter* is reported in Figure 6.2, considering real-world data of the two experimental campaigns presented in Section **??**. In addition, it supports exporting charts and data in multiple formats, making it easy to embed it into other applications and share it with different collaborators.

Through the *Device Configurator*, which is also utilized in the Arrowhead Tools project, end users can observe the value of a property, issue an action for modifying WT configurations, and subscribe to events to receive real-time updates from the WT.

Finally, the *Blockchain Integration* service provides an additional layer of security and trust to the architecture in SHM scenarios of critical facilities (e.g., buildings, bridges, industrial plants, as described in Section 5.2. The data logging system exploits the blockchain's feature of maintaining a permanent and unalterable history of transactions to guarantee the immutability and transparency of the SHM data. Specifically, events related to detecting an anomaly in the structure are stored on the chain, making them easily verifiable by external auditors.

6.2.5 Performance Analysis

This section aims to comprehensively characterize the proposed architecture and its implementation by examining its key variables and identifying potential bottlenecks. We conducted a thorough evaluation encompassing the most significant metrics at each infrastructure level – i.e., EEN, edge, and cloud. We examined the effects of task execution under different edge-cloud continuum configurations, focusing on the trade-offs associated with the feature extraction task detailed in Subsection 6.2.3. This task holds particular significance as it reduces the data dimensionality. To evaluate its impact, we analyzed the deployment of this task on both the EEN and in the cloud.

Concerning the EEN, we analyzed the impact on energy consumption when performing feature extraction onboard or not. To this end, the number of collected samples per sensor (single acquisition) was varied (1024, 4096, and 16384), and the energy spent to compute these features and transmit them (or raw data) has been analyzed, encompassing different wireless transmission technologies [173]. Figure 6.3 summarizes the results and highlights that performing feature extraction on the EEN is more efficient: this is coherent with the fact that the energy expenditure in data processing is minimal if compared to the transmission costs of wireless modulus. This pattern characterizes all the evaluated communication technologies and is enhanced when the data payload increases. An IoT analyzer[174] has been used for energy profiling, considering the hardware and software components of our specific architecture. In particular, in computational terms, we have assumed the electrical properties of the STM32F3 family of microprocessors (maximum clock frequency of 72 MHz, 40 mA and 10 µA in run and sleep mode, respectively).

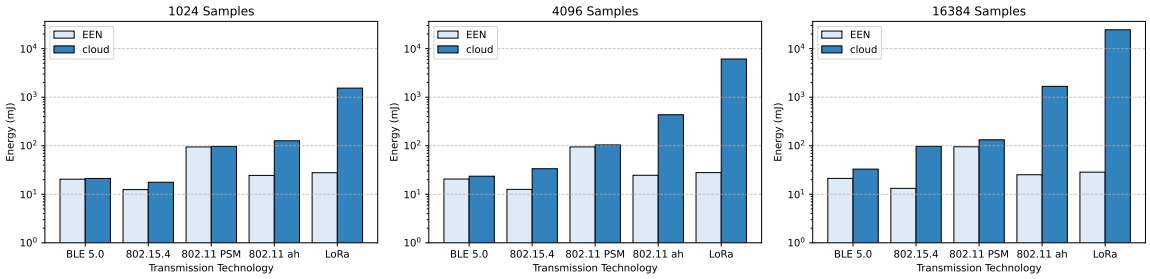


Fig. 6.3 Energy consumption analysis of feature extraction on EEN.

The placement of feature extraction within the edge-cloud continuum directly impacts the amount of data transferred between the edge and the cloud. Edge-cloud data transfer is a key variable in SHM scenarios, where poor network connectivity is a common challenge. Structures under monitoring often lack dedicated networking infrastructure and are exposed to various environmental hazards, including adverse weather conditions. Therefore, we

evaluated data payload sizes when performing feature extraction at the EEN or in the cloud. We assume that each data acquisition generates 2,500 samples, which are transformed into human- and machine-readable formats through the WT abstractions. Each edge device is equipped with a tri-axial acceleration sensor and with three-channel AE SNs. In the AE case, we extract eleven features (summing all the energy and time-related features), while in the accelerometer case, we extract six features – the number of frequencies of interest. Consequently, the feature extraction payload size varies between these two types of SNs. The results of this evaluation, depicted in Figure 6.4, illustrate the byte size of the payload for different configurations. Notably, performing feature extraction in the cloud results in a higher workload on the network.

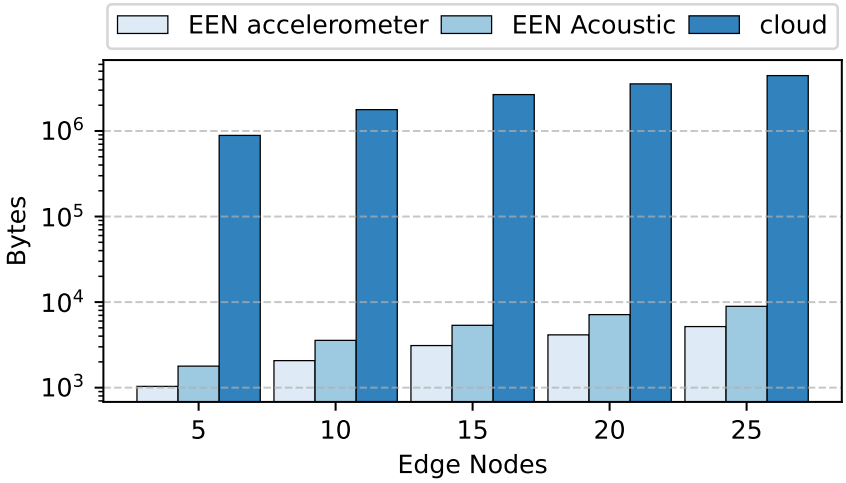


Fig. 6.4 Data payload size comparison when performing feature extraction in EEN versus in the cloud.

Finally, we aim to understand how the increasing workload impacts the cloud, especially when receiving raw data and performing feature extraction. As the same cloud application can monitor several structures, we scaled up the number of monitored structures to analyze how the cloud application performs under these conditions. We assume that each monitored structure is equipped with 10 edge nodes. As such, we are modeling dense and realistically complicated geometries, which require a fine sensor installation plan to capture different damaging phenomena. We realize that in real use cases, this number is different depending on the physical properties of each structure. For our experiments, we simulated data transmission frequencies based on real-world scenarios. Each edge node transmits an accelerometer payload every hour and an AE payload every minute. This higher AE sampling rate is typically necessary when monitoring degraded structures, we opt for evaluating the system in this configuration as the deployed cloud applications needs to able to support the system in

critical scenarios. To scale the data generation process, we developed a workload generator that emulates the edge transmission of accelerometer and AE data. The cloud application in this scenario comprised two components: the Data Aggregator and the Persister Mashup, consisting of a NodeJS and InfluxDB Docker container, respectively. Each container had access to one logical CPU (Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz) and 4 GB of RAM. The workload generator is connected via LAN to accurately assess the performance of the cloud application under controlled and consistent network conditions. Each experiment represents a time window of 10 minutes of continuing transmitting data. Figure 6.5 shows the processing latency from data generation to its inclusion in the database and it demonstrates that the feature extraction component is scalable, as increasing the workload does not significantly impact its execution time. Additionally, the system bottleneck is associate to managing multiple connections and efficiently transforming and storing data in the database. Notably, the system showcases high scalability, as even with constrained resources, it can handle multiple monitored structures. To further enhance the system scalability, scaling the cloud computational nodes horizontally or vertically is a viable option.

Summarizing the conducted evaluation, performing feature extraction in the EEN decreases its energy consumption the amount of bytes transferred between the edge and the the cloud. On the other hand, the additional computation imposed by performing the feature extraction in the cloud is minimal, and the cloud offers greater stability compared to the edge. Moreover, conducting feature extraction in the edge increases the complexity of the EEN, which may require more expensive equipment. In conclusion, there is no universally optimal component placement within the continuum. Instead, a careful analysis of trade-offs is essential to address each scenario's unique characteristics. Fortunately, our architecture versatility enables the exploration of various deployment configurations, empowering users to adapt it to their specific needs.

6.2.6 Use Case #1: concrete frame under seismic excitation

The structure

The specimen used within this experimental campaign is a two-story reinforced concrete (RC) frame, 3×3 m (x and y in-plane directions), 4 m tall (z direction), having columns and beams with cross-sections 20×20 cm and 20×30 cm, respectively. The frame was purposely designed according to non-seismic codes, namely without joint resources for ductility as requested by recent seismic provisions, to represent a bulk of existing non-recent buildings. The structure is built upon two one-way ribbed floor slabs lightened by hollow clay blocks,

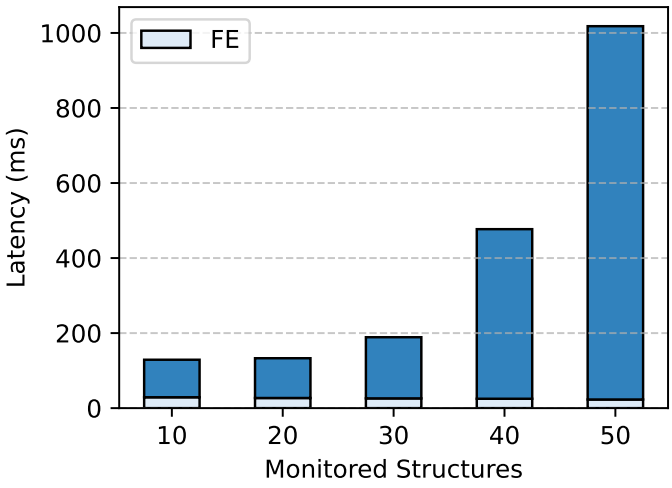


Fig. 6.5 Cloud application scalability.

with the possibility of applying additional masses by steel plates at different points: a picture is shown at the bottom of Figure 6.6.

Testing was conducted at the seismic hall of the ENEA Casaccia Research Center, which is equipped with a 4×4 m shaking table capable of applying seismic inputs on large mock-ups of structures up to 30 t of weight. To this end, the seismic acceleration recorded in Norcia at Savelli Station on 30 October 2016 was selected as the input shaking force since it provoked a disastrous earthquake in Italy. Signals were applied with increasing levels of maximum Peak Ground Acceleration (PGA), from 0.1 to 0.8 g, to damage the frame progressively.

The sensor network

Two SNs comprising three EEN devices (one per floor) were installed on two opposite columns of the frame: the positions were selected after a preliminary numerical simulation of the elasto-mechanical properties of the structure. Splitting the EENs in two different SNs has been preferred over the deployment of one single network in order to minimize the length and the number of cables to be deployed, which would have otherwise been affected by high electromagnetic noise and interference. Two EENs (one per SN) were connected to three G150 AE transducers: the bonding to the structure was realized via metal platforms for landing magnetic sensor holders. All EENs were programmed to acquire, at the same time, tri-axial accelerations; for those connected to AE transducers, the sensing and on–board processing of AE features was also enabled.

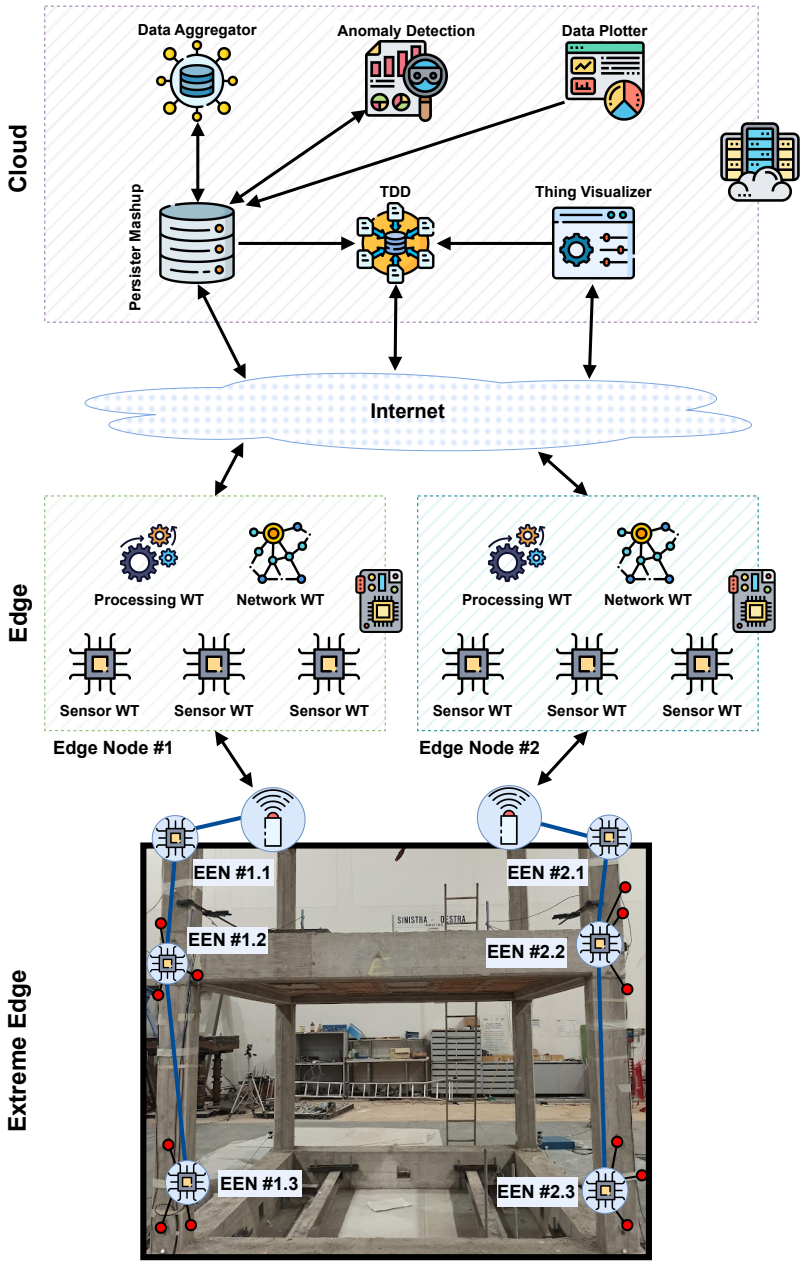


Fig. 6.6 The MAC4PRO deployment plan on the reinforced concrete frame (red dots indicate the position of the AE transducers).

MAC4PRO Deployment Plan

Figure 6.6 illustrates the deployment plan of MAC4PRO to enable the monitoring of the concrete structure. In this scenario, each SN was connected to an edge device (i.e., Raspberry Pi), which abstracts the particularities of the SN and EENs, instantiating them as WTs. Additionally, a processing WT was utilized to perform edge computation tasks. Specifically,

it reads raw sensor data – as a binary data stream – and converts it to a well-structured human- and machine-understandable format, including additional metadata (i.e., collected data timestamp, unique universal sensor id). It performs the first step of data cleaning and error handling. The processing WT handles exceptions and ill-formed data (generated by sensor or communication errors) not to jeopardize the whole processing pipeline. It is strategically placed at the edge for a twofold reason: (*i*) there is intense data communication between the processing WT and the other WTs that could occupy a large portion of the available bandwidth between cloud and edge; (*ii*) some error handling strategies trigger device commands, which need to be executed with low latency not to propagate errors and, thus, minimize the data loss. Finally, the edge is the first infrastructure component of our platform connected to the Internet. For this reason, it implements security mechanisms. The data transmission from the cloud to the edge is encrypted through the HTTPS protocol. Additionally, we leverage the usage of unique identifiers assigned to authenticated applications and users to control and monitor the access for the exposed WTs.

Regarding the continuum configuration, the computation-intensive tasks were deployed in the cloud server – namely, the *Data Management* and *Service* layers components. The TDD is the only component of the *Interoperability* layer (see Section 6.2.2) deployed in the cloud since it indexes the WTs from all edges nodes, and it is the entry point enabling the discovery of the MAC4PRO WTs. In this scenario, the vibration data analysis – performed by the data aggregator and the anomaly detection – was not performed in real-time since it was not the test-bed goal. The vibration data was first acquired through the SNs, then stored, and finally processed to investigate the effects of the emulated earthquake. On the other hand, feature extraction of AE data was computed directly at acquisition time by exploiting the unique on-sensor computing capabilities of the developed EEN devices. The EEN processing is crucial to diminish the burden of the subsequent applications in the computing pipeline, saving considerable bandwidth and storage. Considering only AE data, the performed EEN computation imposes a **reduction of 99.8%** in the transmitted payload [175].

6.2.7 Use Case #2: hydraulic circuit under Acoustic Emission leakage

The structure

The second experimental campaign aims to validate the ability of MAC4PRO architecture to detect fluid leakage that might arise during the pressurization of industrial facilities, such as vessels. To this purpose, the hydraulic circuit in Figure 6.7 has been built and exploited as a target structure. The circuit comprises a pipe loop that can independently pressurize two

1000-liter vessels; moreover, it can be controlled from a dedicated control room or remotely through a dashboard.

The sensor network

As depicted in Figure 6.7, one SN consisting of three EENs, each connected to one AE transducer (i.e., S1, S2, and S3) installed by magnetic connections along the pipeline of the pressure circuit. One sensor (S2) was attached in proximity to the opening valve (inset in the center of the figure) used to simulate leakage, while the remaining two are far apart along the pipe components.

MAC4PRO Deployment Plan

Figure 6.7 depicts the placement of the MAC4PRO architectural components in the hydraulic circuit. Compared to the previous test-bed presented in Section 6.2.6, we introduced the following changes in the deployment plan. First, we deployed a single edge node since only one SN exists in the scenario. Second, the *Data Aggregator* component was deployed only in each EEN, while in the previous test-bed the feature extraction was performed in the cloud and the EEN. Moreover, the *Thing Visualizer* component was not deployed in this test-bed since updating sensor configurations and metadata at run-time was unnecessary. We highlight that such changes to the deployment plan were possible thanks to the modularity and versatility of the MAC4PRO architecture can be easily customized to support many SHM scenarios.

6.3 Arrowhead Tools Project

Condition Monitoring (CM) is a critical application of IoT in Industry 4.0 and Smart City scenarios, especially following the recent energy crisis. CM aims to monitor the status of a physical appliance *over-time* and in *real-time* in order to react promptly when anomalies are detected, as well as perform predictive maintenance tasks. Current deployments suffer from both interoperability and management issues within their engineering process at all phases – from their design to their deployment, to their management –, often requiring human intervention. Furthermore, the fragmentation of the IoT landscape and the heterogeneity of IoT solutions hinder a seamless onboarding process of legacy devices and systems.

In this section, we demonstrate how Arrowhead Tools project tackle those issues by integrating the architecture proposed (Section 2.2) into a toolchain context for CM of industrial scenarios. The architecture was slighted adapted to leverage the concept of

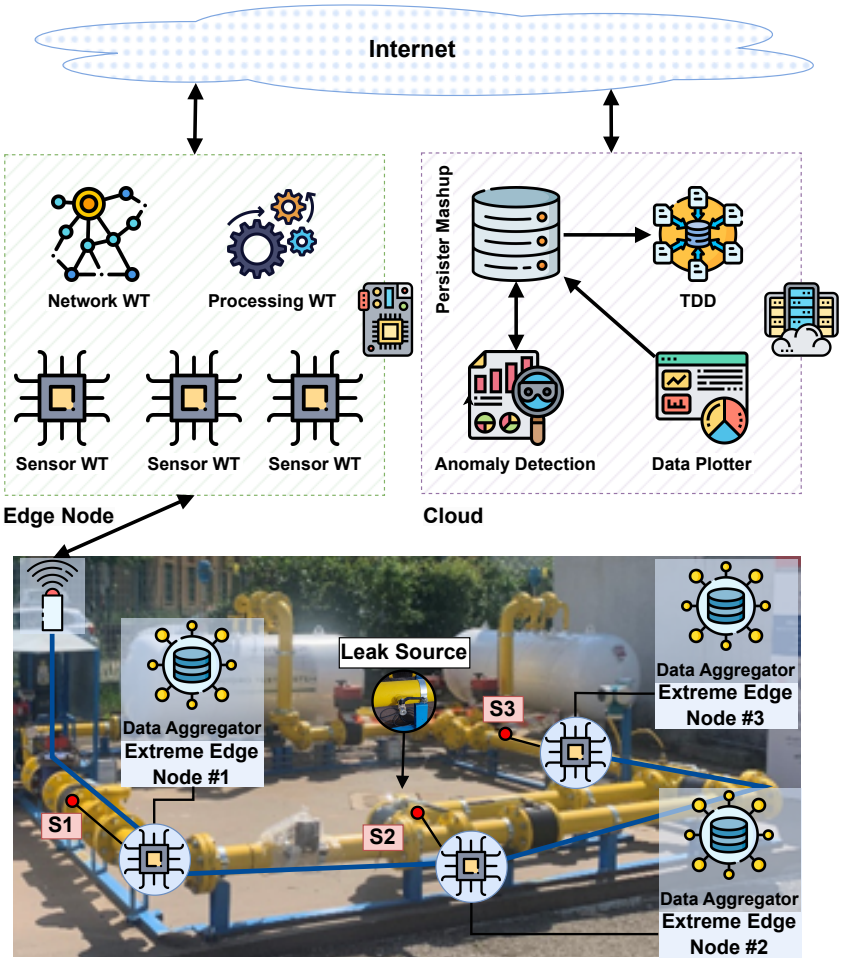


Fig. 6.7 The MAC4PRO hydraulic circuit deployment plan.

toolchains, in order to meet certain project objectives. In particular, we focused into three different and generalizable toolchains, altogether concurring in achieving the following three specific goals:

1. integrating third-party and legacy devices through a seamless onboarding procedure supported by translation of standards;
2. automatically adjusting the working conditions of the devices in order to optimize their maintenance and management costs according to the conditions of the environment;
3. supporting the integration with well-established IoT frameworks and cloud services to lower the need for on-site intervention.

Within our proposal, the main enabler of toolchains is the Eclipse Arrowhead framework, which provides local clouds with service-oriented capabilities, such as loose coupling,

discovery, and orchestration, and represents a single fruition channel for service providers and consumers.

As a proof-of-concept implementation, we then discuss a use-case related to SHM, which is also a demonstrator for the Arrowhead Tools project, also available in a video[3]. The current pilot setting is composed of an SHM sensor network, hosting a number of inertial sensors that monitor the vibrations of a model building. We firstly provided sensors with a degree of interoperability by leveraging the W3C WoT standard. The use case is enriched by the development of a number of engineering tools that comply with our proposed toolchain architecture, making the use case an instance of our proposal.

The section is structured as follows: Subsection 6.3.1 presents the modifications made on the reference architecture to encompass the toolchains. Subsection 6.3.2 presents our demonstrator pilot, the sensor used, and the tools for their interoperability. Subsection 6.3.3 details the implemented tools and components for the three proposed toolchains within our demonstrative implementation. Subsection 6.3.4 details the implementation of caching – i.e., `CACHE-IT` from Chapter 4 – within the system. Subsection 6.3.5 describes the experiments and the validation results with a particular focus on the project objectives and discusses takeaway messages.

6.3.1 Architectural Adaptations to a Toolchain-Oriented System

We set three project objectives for a CM scenario, in order to satisfy a number of KPIs that are necessary for the current standards. In order to fulfill said objectives, we propose three different toolchains which are compatible with the thesis overall architecture – as illustrated in Figure 6.8 –, each one enabling a distinct functionality and composing software tools, that perform tasks determined by their own layer, into pipelines.

Notably, the architecture utilized in Arrowhead Tools is an adaption of the reference architecture to suit a toolchain setting. The first layer is shared among the toolchains since the physical components are the base of any IoT-based system. The Interoperability layer operates as a toolbox, providing general-purpose tools to each of the toolchains, each of which selects a subset of components in order to enable its own interoperability. Tools from the interoperability layer up to service layer may be deployed in the cloud, or in a local cloud (i.e., locally with the structure). Interactions among tools and application systems in a local cloud are mediated by a local instance of the Eclipse Arrowhead framework, which also holds a connection with the outer world (i.e., the cloud). Following is the description of our proposed three toolchains:

[3]https://youtu.be/1nEOJpXu9l8

- **Data Toolchain**: it is the traditional IoT pipeline – i.e., data acquition –, enabling the main goal of the system. In the specific case of SHM, it provides end-users with information about the physical health of the structures over time and can potentially identify structural damages, which can lead to predictive maintenance and the avoidance of accidents.

- **Device Toolchain**: it supports the management of physical devices of the system. The toolchain utilizes device metadata to support system administrators with information about device location, capabilities, version, and current configuration. Also, it allows users or tools to alter the current devices' settings in order to enhance or optimize a certain feature of the system.

- **Energy Toolchain**: energy is a crucial factor for IoT-based systems. Devices are numerous and designed to last for years, thus, a delicate balance exists between battery lifetime and device operation. Improper use of the energy in the system can lead up to a meaningful increase in costs related to maintenance. We leverage the energy management of the system bringing it to the application level. The energy toolchain monitors the energy harvesters, the consumption of each device, and the current and future environmental conditions that may impact the energy.

The high-level design of each toolchain is a guideline to assist the implementation of IoT-based systems for the CM domain and can be accomplished with a myriad of different implementations that perform the roles described. However, in the next sections, we exemplify and illustrate how we implemented each toolchain in the scope of our use case.

6.3.2 The SHM Pilot: Multi-Chain Components

As an instance of the above described toolchain architecture, we present here its concrete implementation into the SHM pilot use case within the Arrowhead Tools project, reported in Figure 6.9. The baseline of the use case features a structure to monitor and a set of inertial sensors for SHM permanently attached to it. Each sensor cluster is managed by a Cluster Head (CH) that acts as a sink for the data and interacts with any external actor. Sensors are abstracted into WTs. Other legacy sensors, in our case a gas sensor, may be added. Compliance between the WoT and Arrowhead is achieved through WAE, presented in Section 3.2. All elements of our pilot that we identify with the sensing and interoperability layer of our toolchain architecture are described in this section.

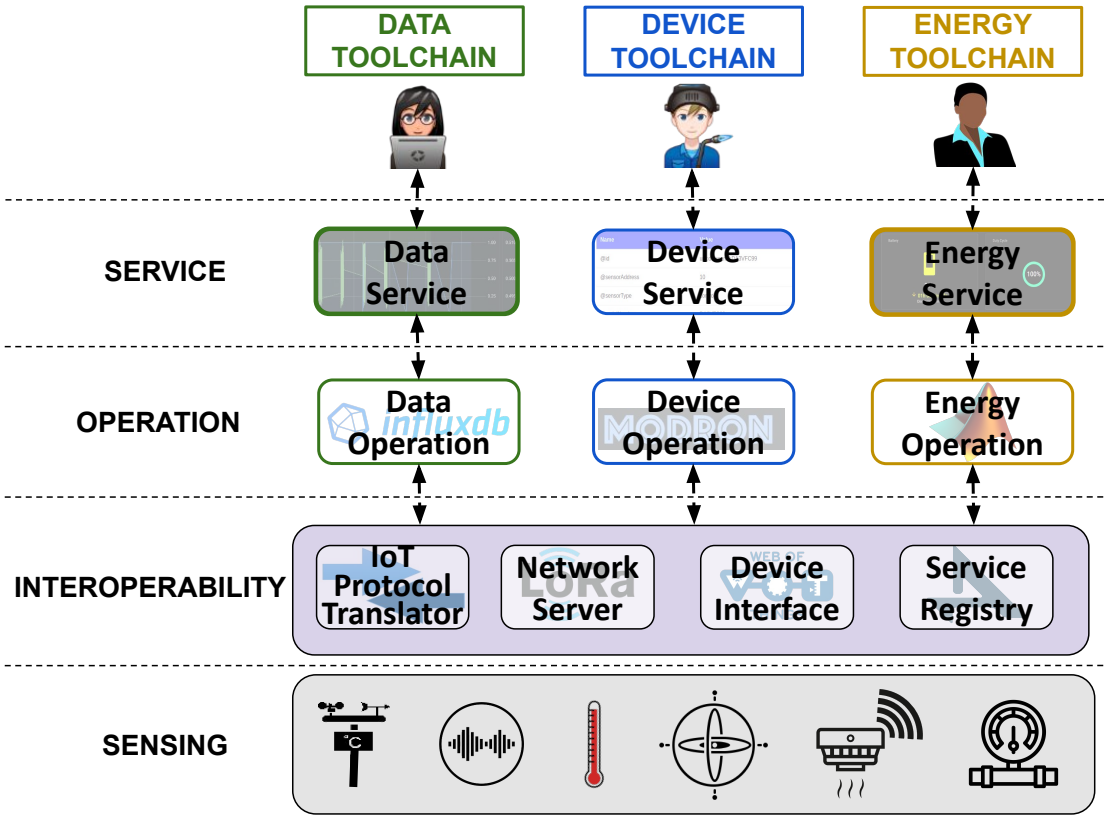


Fig. 6.8 Extension of base architecture to support toolchains

The description of each toolchain is presented in Subsection 6.3.3. All the interactions taking place among tools and application systems are mediated and overseen by an instance of the Arrowhead Core Services.

We assume that the monitored structure is in a poorly connected environment, in which providing cable connection to all the sensors is hard, therefore edge devices communicate with each other via wireless connections and are powered by batteries. Sensing layer devices employed in the pilot are either connected to a sensor network tailored for SHM or third-party devices not easily integrated to an already deployed network. The devices that compose this layer are utilized by the three toolchains, and are:

- **SHM Sensor Network**: The adopted monitoring network consists of intelligent sensor nodes (SN) and related network interfaces. Each peripheral device is a micro-system unit equipped with a microcontroler and sensors that capture triaxial accelerations and angular velocities in a broad frequency range. Once signals are sensed by the peripheral node, information is then transmitted to the CH controller through the companion gateway network interface (referred to as SAN interface in [170] and in Figure 6.9).

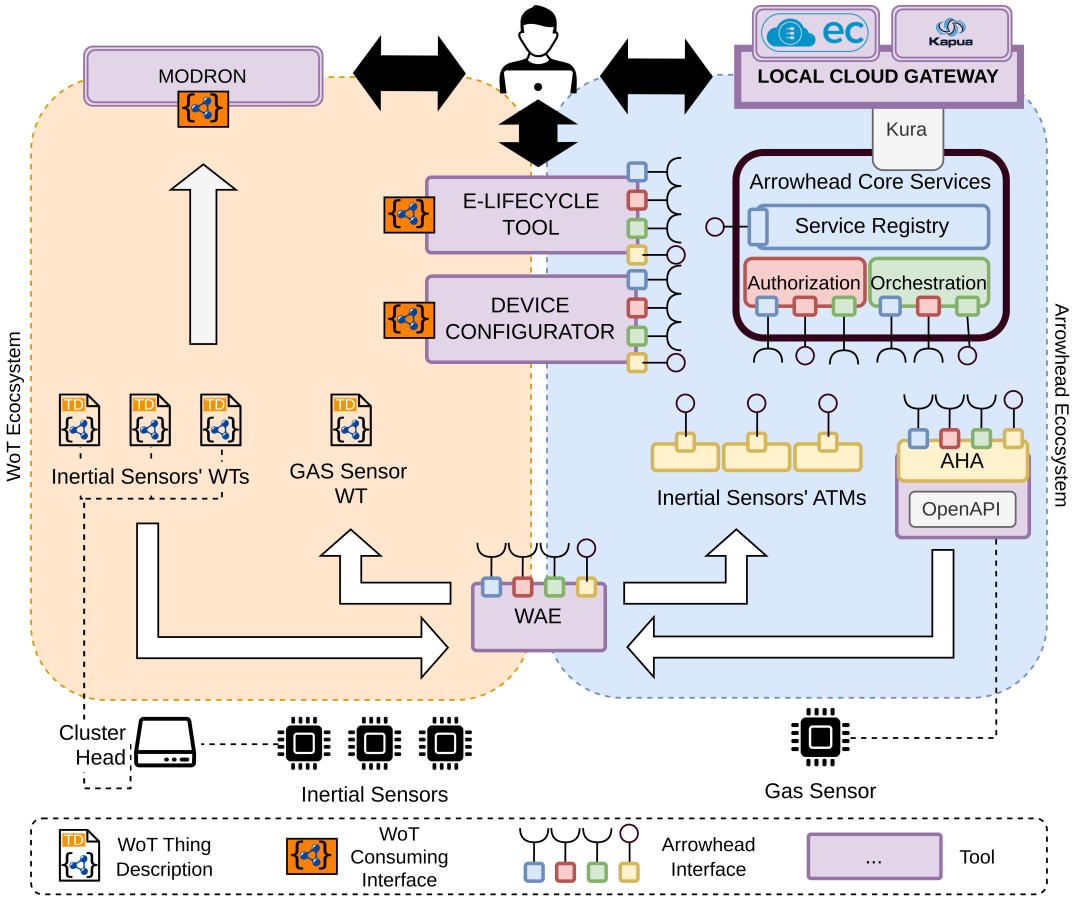


Fig. 6.9 Toolchain architecture of the whole System-of-Systems, with a focus on the separation and the interoperability between the Arrowhead and the WoT ecosystems.

- **The Gas Sensor Node**: The gas sensor node is an autonomous end-device that embeds an experimental gas sensor by STMicroelectronics [176]. The sensor supports SHM scenarios in a number of ways – e.g., detecting the presence of heavy traffic load. In addition to the sensor, the node integrates: a micro-controller unit, a LoRa transceiver, and all the circuitry needed for the smart power supply system. Additionally, it is equipped with a solar energy harvester, a battery management system, and a module that monitors the condition in which the energy harvester works.

The Arrowhead framework is a key interoperability enabler; and, due to its pervasive presence in all toolchains, it was omitted for clarity in all the following diagrams that illustrate each toolchain, namely Figure 6.10, Figure 6.11, and Figure 6.13. Besides, the major tools that enable interoperability are the ones presented and discussed in Chapter 3. In detail, each device that composed the SHM network was abstracted as a WT, which had its TD indexed by ZION. Besides, the WAE was responsible to bridge the communication between WT and

Arrowhead services. Additionally, the Arrowhead Adapter (AHA) was developed by other partners of the project to facilitated the integration of legacy devices (e.g., the gas sensor) with the Arrowhead ecosystem.

6.3.3 The SHM Pilot: Toolchains

This subsection presents the implementation of the three toolchains described in Subsection 6.3.1. The toolchains deployed in the pilot are modular and independently from each other. However, it is common for them to be deployed together since their functionalities are complementary.

Data Toolchain

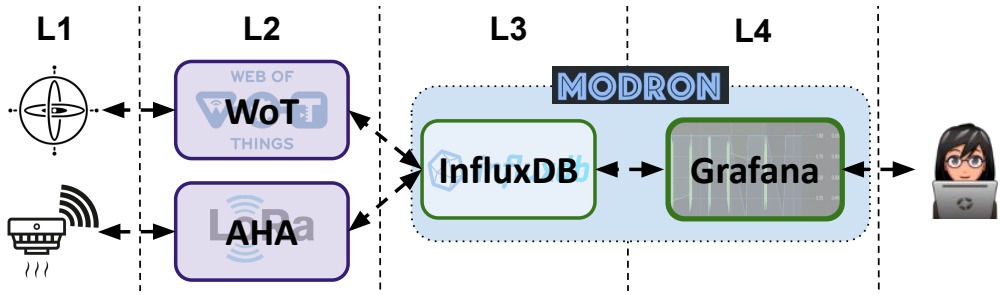


Fig. 6.10 Data Toolchain

The Data Toolchain is the main functional toolchain in the use case, as its goal is to acquire data from the sensors deployed on the structure and report it to the internet for analysis. In this toolchain the data of both typology of sensors are send to MODRON through their interoperability abstractions and store in a InfluxDB, which is connected to a Grafana(i.e., the Data Plotter), providing live visualization of the sensor data. Data can also be queried by third-party applications through MODRON's interface.

Device Toolchain

The Device Toolchain enables a user to interact directly with the devices and issue them with commands that change their operating conditions at run-time. The toolchain includes the baseline application systems (i.e., the SHM sensor network), the gas sensor, AHA, the WAE, ZION, and the Device Configurator. While both category of devices are wrapper through their respective adapters (i.e., WAE and AHA), WAE is the component responsible to manage and integration with ZION. To do so, it registers the WTs directly in ZION and it transforms the AHA Arrowhead Service in a TD to be stored and managed by ZION. Connected to the

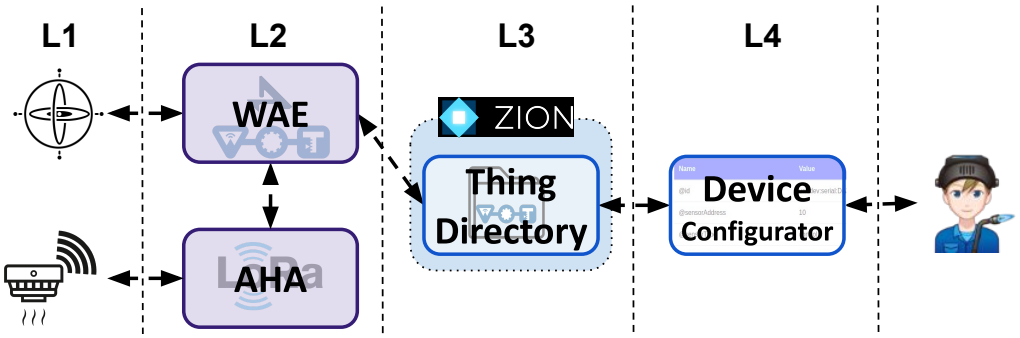


Fig. 6.11 Device Toolchain

indexer, is the Device Configurator, which lists all the WTs affordances and allows to the user to trigger a specific actions that will change the targeted device behavior. A screenshot of the Configurator is depicted by Figure 6.12.

Energy Toolchain

Wireless sensors are one of the key elements for many IoT applications because they are easy to install and simple to connect to existing networks and infrastructures. One of the main hardware enhancements to the initial baseline is a shift from cable-connected edge devices to wireless ones. However, the latter are traditionally battery-powered, and this is the main limiting factor to their extensive use in real applications [177]. To be compliant with application-specific requirements (e.g., data sampling and transmission rate), the node power consumption often does not guarantee an acceptable recharge rate of the battery leading to not sustainable maintenance costs and too frequent short interruption of service. The adoption of Energy Harvesting (EH) solutions helps to extend the battery life or even obtain battery-less autonomous devices.Unfortunately, the design of an energy harvesting module is usually a complex and strictly application-specific task requiring a lot of design efforts in terms of both time and costs. It is in this context that an EH toolchain has been developed, composed of the energy harvesters and a number of software tools: Dr. Harvester and the E-Lifecycle Tool.

An EH circuit is comprised of three main functional blocks: i) the energy source module (e.g., solar panel) to collect energy from the environment where the device works; ii) the energy conversion and management circuit to efficiently power supply the device by using the collected energy; iii) the energy buffer (e.g., rechargeable battery) to store the collected energy. Customization based on application-specific requirements is needed to obtain effective EH solutions. For example, the two considered case studies – the Gas Sensor and the SHM sensor network (entirely powered by the CH, which is hosting the harvester) – have very

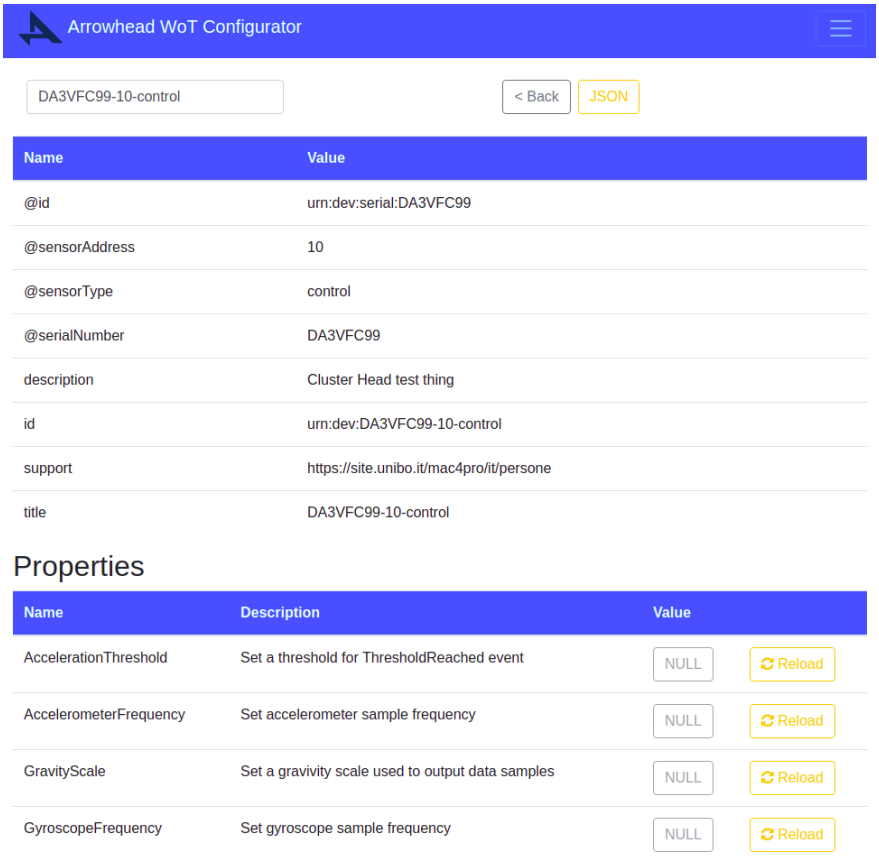


Fig. 6.12 A screenshot of the WoT–Arrowhead Device Configurator

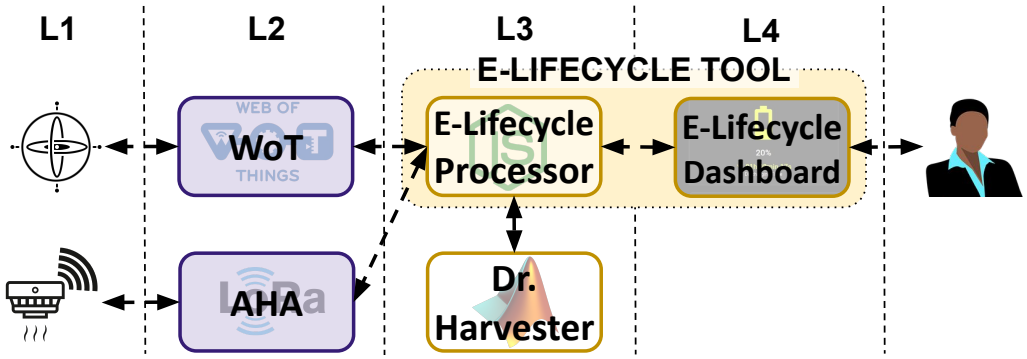


Fig. 6.13 Energy Toolchain

different power consumption and consequently, they need two different EH circuits. In this work, the design effort to obtain suitable solutions for both case studies has been reduced thanks to the developed Dr. Harvester tool.

Dr. Harvester

The tool implements a DT of the energy harvest system. It is comprised of a MATLAB-based core application able to interact with an electrical circuit simulator. It is based on a set of pre-designed EH circuits, each circuit is designed to match a given set of environmental working conditions (i.e., solar irradiance, thermal gradients, and mechanical vibrations) and a range of load power consumption (i.e., power consumption of the edge device).

Dr. Harvester is used to forecast the behavior of a remote-controlled edge device (e.g., Gas Sensor or SHM sensor network) under the provided working conditions. Here the main outputs of Dr. Harvester are the remaining battery lifetime or the remaining time to fully recharge the battery depending on the simulated working conditions. All the tasks are triggered by the E-Lifecycle Tool through an Arrowhead-enabled REST interface. To decouple the interaction with other tools, Dr. Harvester was converted into an Arrowhead Service Provider. The tool, by registering services on the Arrowhead Framework of the local cloud, allows remote submissions of simulations via a REST interface and retrieves results on the remaining battery life that will then be used for optimization. Each request causes the input to be validated, which is then forwarded to the MATLAB process to start the job. Since a simulation might take time, it is handled asynchronously.

First, the E-Lifecycle Tool transforms sensor data in a data structure that complies with Dr. Harvester's simulation input, gathering the EH system meta-information (e.g., battery storage capacity) and the working conditions of interest retrieved from the edge device (i.e., the actual electrical and environmental working conditions and the actual device's battery state of charge). Second, the E-Lifecycle Tool launches Dr. Harvester which executes the

simulation of the EH circuit corresponding to the input parameters. Third, Dr. Harvester generates an output containing the simulation results. More precisely, the output generated by Dr. Harvester contains the battery status (i.e., the battery is actually charging/discharging), the rate of variation of the battery state-of-charge and the simulation status to inform the E-Lifecycle Tool that the simulation has been completed correctly or not, and consequently if the results are valid or have to be discarded. Figure 6.20 depicts the E-Lifecycle GUI that supports the energy management of devices. The E-Lifecycle Tool is available as an open–source application[4].

As each simulation in Dr. Harvester can take time to execute, we deployed a caching service within the E-Lifecycle Processor. This way, we decrease the significant waiting time of future requests – enhancing user experience – and avoid redundant computation on the server side [90]. The deployment of the `CACHE-IT` framework into the Arrowhead Tools system is detailed in Subsection 6.3.4.

6.3.4 `CACHE-IT` Deployment

The Dr. Harvester has multi-thread limitations and requires considerable processing power and time to respond to requests – a response is given after an average of $\sim$25s. Applications – both GUI and back-end services – must perform several simulations encompassing all possible duty cycles for each device, frequently leading to waiting times in the order of hours. Adding new devices to the systems dramatically impacts its performance. The use case presented is not time-critical compared to applications that need milliseconds of precision. Although, high latency impacts the system in a twofold way: *(i)* Frequently, the environmental conditions had already changed when all the simulations finished their execution; *(ii)* The high screen loading time is a significant issue for user experience. Additionally, the evaluated scenario implies having several identical sensors deployed. Without caching, it leads to redundant computation performed by Dr. Harvester, wasting computation resources and power. Consequently, we deployed the `CACHE-IT` framework which is responsible for caching the outputs from the harvester DT. We detail the Caching Template utilized, providing insights into the careful design choices and trade-offs we made to optimize performance.

This testbed caching strategy relies on domain knowledge; the data requests to Dr. harvester contain information such as the device's electronic features, battery percentage, and the current solar irradiance, which is logged at the edge and transferred to the Cache Controller by the History Transfer. We designed the caching strategy to leverage those data to estimate each device's battery and solar irradiance for the next period and produce

[4]https://github.com/UniBO-PRISMLab/arrowhead-optimizer

caching orders instructing those resources' caching. The other relevant Caching Template configurations are the functions that define the caching strategy:

- *gen* function: we choose the Prophet algorithm as the underlying technique since it is capable of forecasting time series data [178]. The *gen* function triggers the `fit` process, which outputs a model capable of generating predictions. The function extracts the historical solar irradiance and the battery level data points from the `log` and fits them into the Prophet model. This fitting process allows Prophet to understand the trends and seasonality present in our data. Once fitted, the model can generate a forecast for a future period. This forecast becomes the function *cOrders* that we utilize to generate caching orders.

- *trigger* function: this function is implemented to simply return a *true* value every 12 hours. The periodic reevaluation is chosen to ensure that our caching strategy could respond to changes in demand patterns that might occur from day to day but without causing unnecessary computational overhead by recomputing the strategy too frequently.

- *period* function: this function is designed to output the values from the last 7 days. This choice was based on the results obtained in [90]. This time frame balances the need for a sufficiently large dataset to capture trends and patterns with the need to keep the computational demands of the strategy manageable.

6.3.5 Results and Discussion

The effectiveness of the SHM application is validated on an existing use case, which consists of an experimental model of a bridge (Figure 6.14) located at the Laboratory of Structural and Geotechnical Engineering (LISG) of the University of Bologna. The structure represents a 1:4 scale reproduction of a real composite steel-concrete bridge crossing the A14 highway in Italy, near the city of Bologna. The scale replica preserves the materials and their properties; changes concern only the dimensions. More information about the model bridge manufacturing process and material properties can be found in [179].

The deployed SHM sensor network and its main components are displayed in the middle callout of Figure 6.14, in which one CH (red point) and seven SNs (green points) are noticeable, placed at strategic positions for vibration analysis, i.e., in correspondence of the anti-nodal points of the first modal components. A snapshot of one SN installed at the bridge deck joint has also been enclosed in the upper part of Figure 6.14. Besides, a vibrodyne (Figure 6.14 blue point) has been applied to the bridge in order to apply a variable-intensity

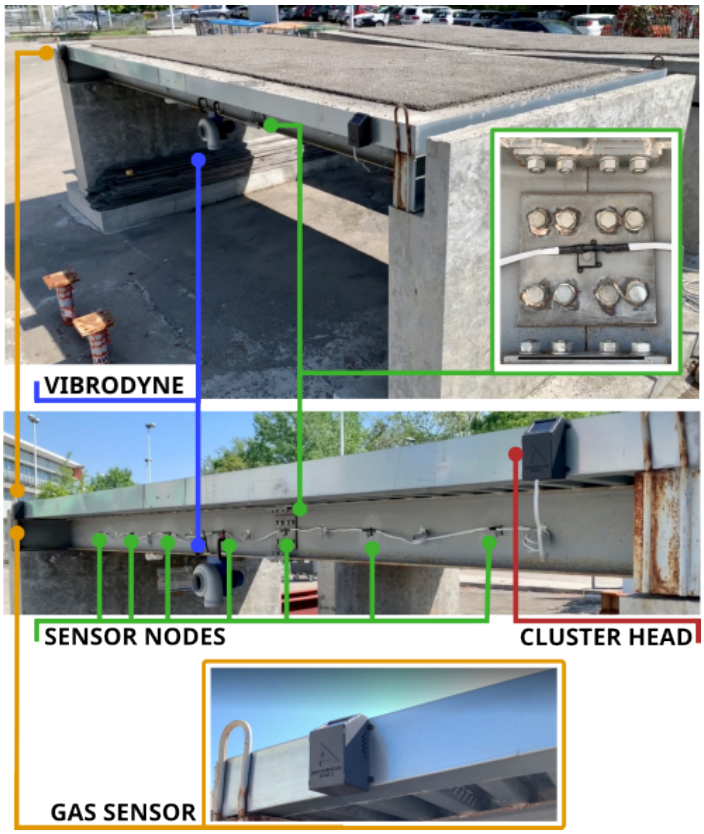
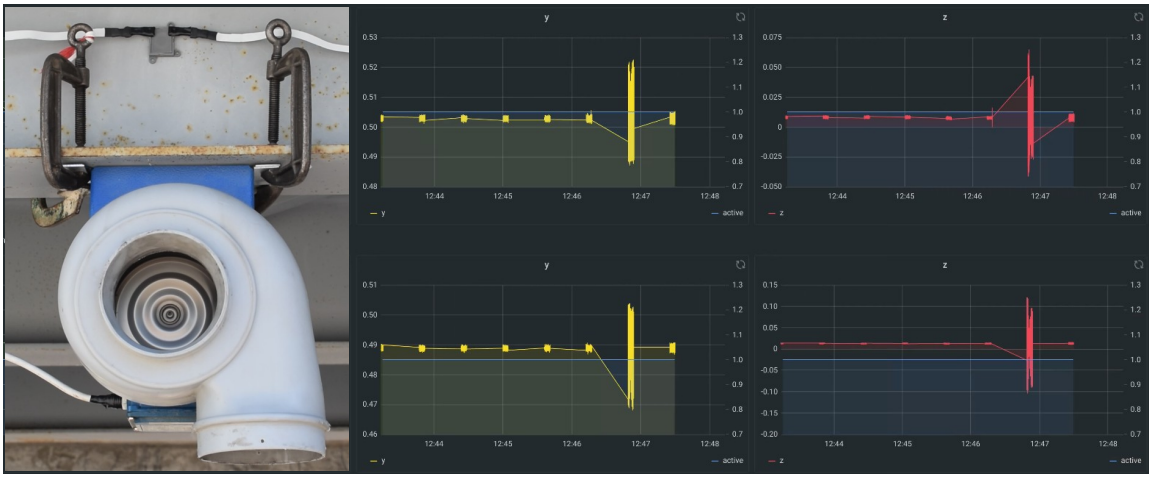


Fig. 6.14 Bridge model under test.

harmonic excitation up to its power supply settings. In addition to the SHM network outlined above, the deployment includes the gas sensor (Figure 6.14 orange points). Both the gas sensor and the CH are powered by an EH circuit attached to a solar panel.

Integration with Legacy Systems

The three toolchains are composed of several tools that altogether concur in achieving the first project objective: the seamless interoperability and the integration of legacy systems. This is most evident for the tools that compose the Data and the Device Toolchains. The demo environment illustrated above contains a number of off-the-shelf legacy devices and systems that need an integration step in order to be fully operational in our heterogeneous scenario. The SHM sensors were originally abstracted into WTs, however, they lacked interoperability towards the Arrowhead Local clouds services. Hence, only with the WAE, we achieved a fully interoperable layer, so that the interaction with the devices from the upper layers is totally agnostic of the standard they comply with.



(a) Sensor readings of the SHM sensors on MODRON with the vibrodyne on.

(b) Sensor readings of the gas sensor on MODRON.

Fig. 6.15 The upper figure shows the accelerometer bursts that change after the vibrodyne is turned on. The lower figure shows how the data from the gas sensor changes before and after the gas sensor is sprayed with gas.

The validation of this aspect has taken place by means of the sensor values shown through the Grafana dashboard via the Data Toolchain. Figure 6.15 shows the sensor values reported into said dashboard. In particular, the upper figure shows how we solicit sensor readings by turning on the vibrodyne and, thus, injecting vibrations into the structure. These are captured by the accelerometers and reported in one of the last bursts on their Y and Z axes, also visible in the picture (the X axis is not very significant as the bridge does not vibrate much longitudinally). The exact same system is used to detect variations in the concentration of gas, as captured by the gas sensor in the lower part of the figure. The validation shows a small real-world fully operational system, however, the main advantage here is the interoperability: in fact, adding more physical sensors or other systems does not cause the complexity to increase, as the "hub" systems such the proposed architecture do not need any additional interface to cope with onboarding processes, rather they take place seamlessly.

Performance Evaluation: Caching in an SHM

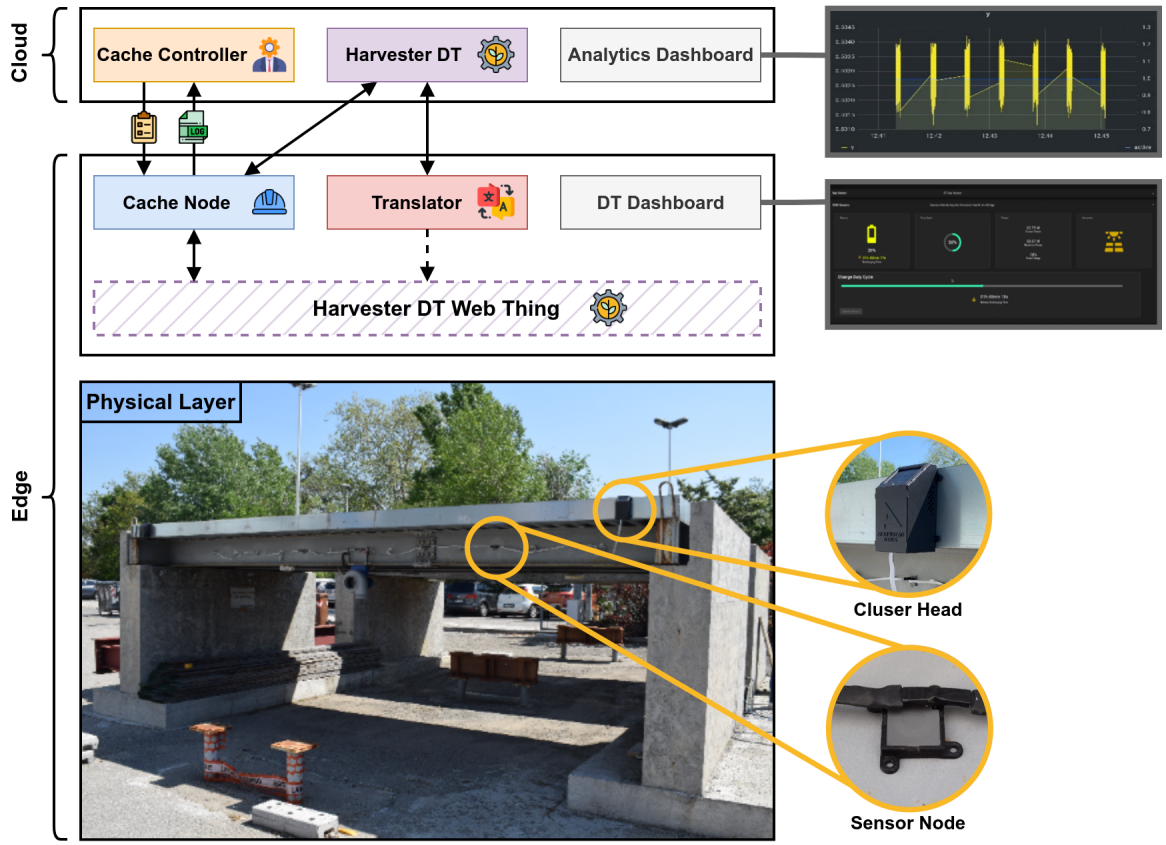


Fig. 6.16 The CACHE-IT framework deployed in a SHM case-study

To validate the impact of the caching added to the Energy toolchain, we validate our design of edge caching experimented with a subset of the taxonomy features mentioned in Section 4.1, and the CACHE-IT default approach of proactive edge caching. The architecture is illustrated in Figure 6.16, which illustrates the scenario and the deployment of the CACHE-IT framework. We evaluated the system with different configurations of cache design – specialist and generalist – and caching strategy – proactive and reactive – and analyzed their impact in terms of latency and cache hit rate. The four caching configurations developed for the experiments are the following:

- ***Baseline***: no-cache was deployed, and no edge device was used.
- ***Configuration #1*** - **Reactive Generalist Cache**: it represents the traditional cache system: once a request is made to Dr. Harvester, its response is then cached on the edge.

- ***Configuration #2*** - **Reactive Specialist Cache**: this is a simple algorithm that analyzes the similarities between the input request data and the requests that it has cached. If equivalent, then it replies with the cached data.

- ***Configuration #3*** - **Proactive Specialist Cache**: Throught `CACHE-IT` we employed an algorithm to forecast future requests and proactively cache them for a specific period. Besides the static electronic features of the input payload, two attributes change over time: battery percentage and solar irradiance. As described into Subsection 6.3.4, we utilize the cache size period as the training dataset (the reason being that it is not possible to evaluate the proactive cache with zero cache size), and we use the Prophet data forecast algorithm to generate the predictions. Then we proactively cache twelve hours of data – the duration of a single replication. Figure 6.17 depicts the forecast and real values by Prophet for a single replication when we have 7-days of training data, the dotted line separates the training set from the actual forecast.

We evaluated the configurations above against three different cache sizes – i.e., empty cache, one day of cache, and 7-days of cache. Each experiment emulated ten cluster heads for 12 and was replicated twenty times.

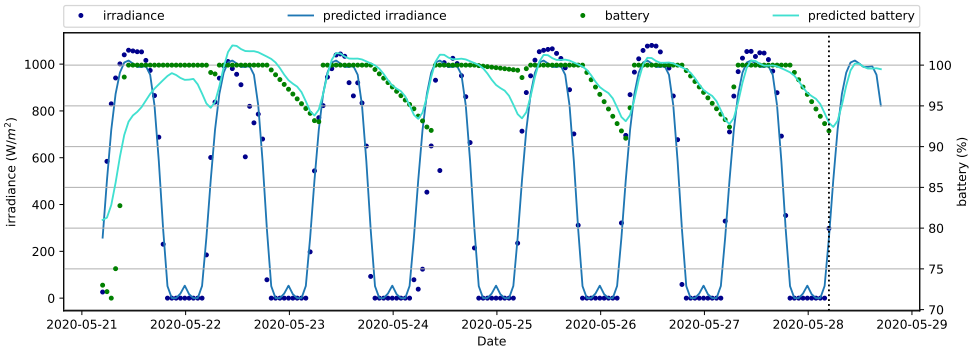


Fig. 6.17 Proactive caching (Configuration #4) predictions of battery life and solar irradiance for seven days of training.

We deployed the Dr. Harvester in a server in our private cloud. The edge caching was deployed in a computational node with low processing power – a Raspberry Pi –, and we synthetically generated requests following a Poisson distribution from a personal computer. The traffic generator and the edge caching devices were in the same LAN, connected through WiFi. The traffic generator emulated the cluster head battery depletion, and the irradiance data was provided by the National Solar Radiation Database[5]. In each experiment replication,

[5]https://joint-research-centre.ec.europa.eu/pvgis-photovoltaic-geographical-information-system/pvgis-data-download/nsrdb-solar-radiation_en

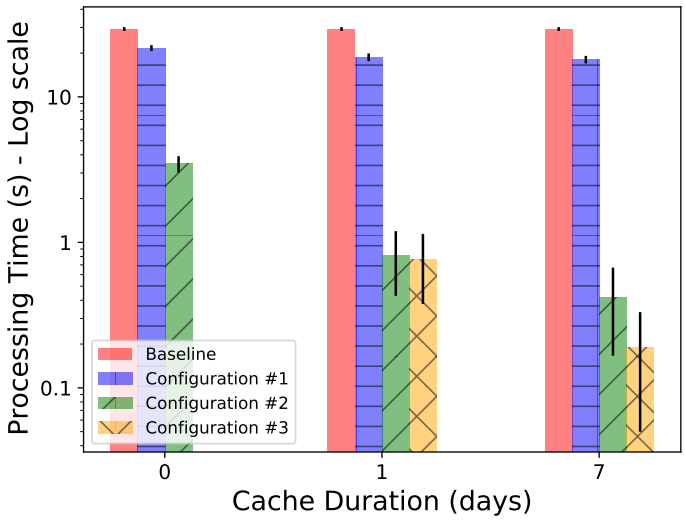


Fig. 6.18 Processing time results in logarithmic scale for different edge cache configurations

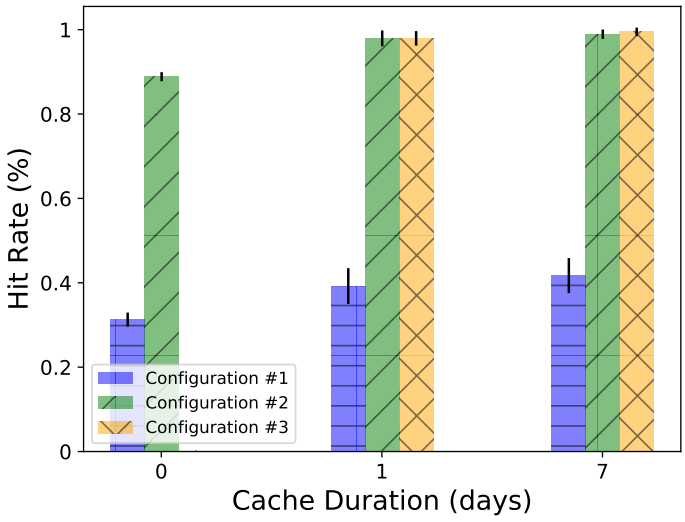


Fig. 6.19 Cache hit rate results for different edge cache configurations

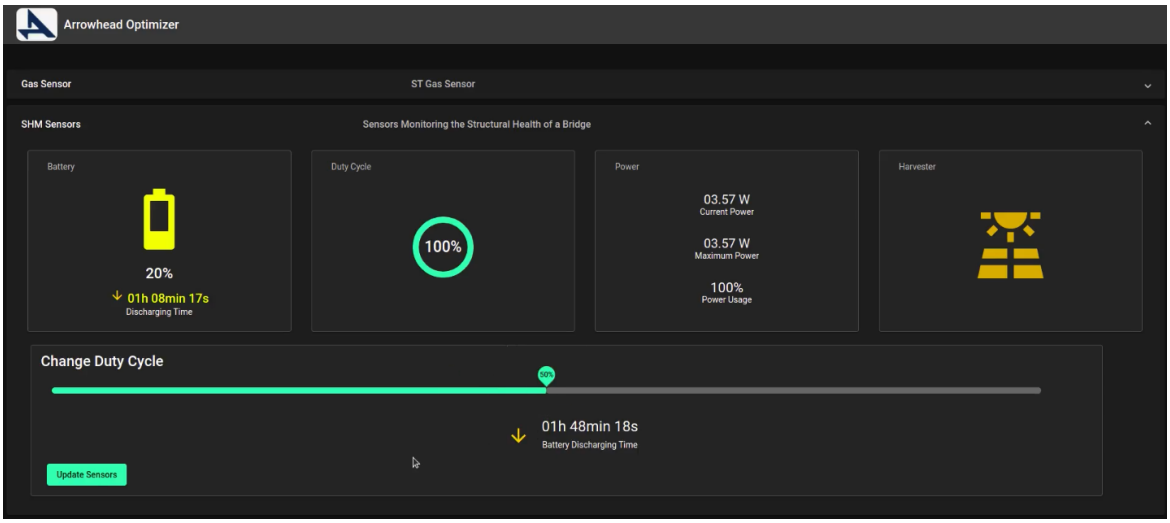
a random start date and an hour were selected from the solar irradiance database. Although the edge device has limited storage capabilities, the data cached is a small payload. The cache storage did not fill the device's storage in any of the experiments. Consequently, we did not need to utilize caching replace techniques. The impact of the different configurations through different cache sizes is depicted in Figure 6.18 and Figure 6.19. As the time difference between a cached response – a few milliseconds – and a response from the DT – $\sim$25s – is substantial, the y-axis is on a logarithmic scale. Such a time difference further highlights the need for a cache mechanism.

Configuration #1 is the simplest solution; though it has the worst performance, it still significantly improves the latency compared to not using a cache. Configuration #2 is a significant improvement metric-wise from the generalist – configuration #1 – solution, obtaining more than 90% hit rate in all experiments and, consequently, a low response time. However, its design has restricted a subset of solutions and requires higher engineering effort to be adapted. Finally, the configuration #3 shows similar results to the configuration #2. The main difference is that the proactive cache made requests beforehand rather than replying with previously cached results. Both methods are effective; however, the proactive cache is potentially more advantageous in scenarios where either AoI is essential.

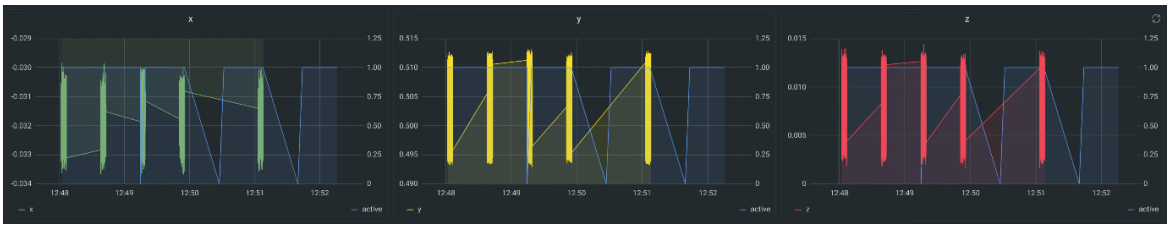
Reduction of Engineering Costs

All illustrated toolchains increase the automation of the baseline as well as reduce the need for human intervention, which implies a significant reduction in the cost of manual effort. However, the major evidence of cost reduction was brought in by the Energy Toolchain, presented in Subsection 6.3.3, which introduces a number of decision-making tools that support the engineering process both at design time and run-time, greatly simplifying the interaction with the physical components as well as guiding their parameters towards an optimal resource consumption.

Controlling the data rate at which information is gathered is of utmost importance in energy-efficient monitoring scenarios, especially when this parameter can be changed remotely.Figure 6.20 shows the E-Lifecycle Dashboard with two tabs: one for the Gas Sensor and one for the SHM sensors. The bottom of Figure 6.20 shows the Grafana-based dashboard after the duty cycle was changed to 50%, it depicts the sensor bursts and their wake-up and sleep state via the blue line. The caching system deployed was fundamental to enable the operation of the Energy Toolchain, since it enable a decrease in more than 90% in the delay time. The customization features of `CACHE-IT` enabled the energy toolchain deployment in the scenario.



(a) The E-Lifecycle Dashboard.



(b) Sensor readings and duty cycles on MODRON.

Fig. 6.20 Screenshots showing the E-Lifecycle Dashboard (above) during the operation of changing the duty cycle of the SHM sensors from 100% to 50%. The result of the duty cycle change is shown through on three axes of a single SHM sensor (below): the blue line identifies the wakeup intervals of the sensors. When the line is not set to 1 the sensors do not perform any read operation.

The possibility to forecast the amount of energy available at a given time and for a given set of working conditions and reconfigure all the devices of a remote complex system is the key factor to obtaining a context-aware dynamic optimization of the performance of both single edge device and the whole system. This methodology can be applied in many IoT and IIoT emerging applications and contributes significantly to the engineering, deploying, commissioning and maintenance costs reduction.

Chapter 7

Conclusions

This thesis aimed to design, develop, and deploy a flexible IoT monitoring system that utilizes the edge-cloud continuum for SHM scenarios. This solution is intended to be customizable and easily adaptable to diverse use cases. To reach such a goal, we divided our objective into four main research questions (RQ): (i) *How can a seamless interconnection be established among diverse devices, applications, and systems in the context of IoT monitoring applications?*; (ii) *What strategies can be employed to enhance the efficient management of data within the edge-cloud continuum, emphasizing in enabling low latency while considering data freshness?*; (iii) *How can tools be designed and implemented to enable end-user usability and integration with other systems?*; (iv) *How can we deploy the same architecture across diverse scenarios, which have different edge-continuum configurations, while considering different system end-goals and requirements?*.

In the sequence of this chapter, Section 7.1 summarizes the contributions of this thesis. Section 7.2 presents directions for the future. Finally, Section 7.3 presents the final thoughts and conclusions.

7.1 Summary of Contributions

We developed a flexible multi-layer IoT architecture that is infrastructure-agnostic. This architecture is a versatile platform for SHM projects. It utilizes cutting-edge technologies from information technology, software, and industrial engineering. It comprises four layers: Sensing, Interoperability, Data Management, and Service. Each of those layers – except the sensing layer, which is out of the scope of this work – was designed to respond to one of the raised questions, while the RQ (iv) was addressed by the deployment and usage of the architecture in different domains.

7.1.1 RQ (i) – Interoperability

To answer the RQ (i), the interoperability layer integrates current solutions to establish an IoT ecosystem that facilitates seamless connectivity between devices, applications, and systems, which involves designing integration between widely adopted interoperability solutions, including the W3C Web of Things standard, which enables connectivity at the device perspective, the FIWARE IoT Platform, which seamlessly connects applications, and the Arrowhead framework, which facilitates the onboarding and integration of systems, including legacy ones. Analyzing the contributions discussed in Chapter 3, we identified two key contributions that have had a significant impact:

- **FIWARE integration**: WoT and FIWARE boast large communities advocating for their adoption and usage. The integration of these two ecosystems merges the advantages of both worlds. On the one hand, FIWARE does not have strict specifications for defining entities compared to WoT's specifications for TDs, and it faces difficulties in dealing with various network protocols and data formats. On the other hand, the WoT does not have mechanisms to: search WTs by localization, to orchestrates flows based on property values; and to automatically integrates time-series storage — all features enabled by FIWARE.

- **Web Service integration**: one of the goals of the *Web* of Things is to seamlessly interconnect IoT devices with the Web by utilizing concepts already familiar in that context. However, the W3C WoT initiative does not provide methods or guidelines to convert dissonant interfaces to its ecosystem. Integrating third-party applications into the W3C WoT demands custom code, significantly burdening application developers. In response, we introduced techniques that automate converting RESTful web services –comprising most of the current Web- into WTs. This approach separates the business logic of web applications from the underlying network stack and establishes a uniform interface for all actors within an IoT system.

Interoperability is a crucial enabler to reach the main objective of this thesis, which is to establish a *versatile* architecture. A versatile system must adapt to different settings, which require an interface with diverse components. Interoperability prevents the proposed architecture from becoming overly restrictive to a specific scenario or use case.

7.1.2 RQ (ii) – Edge Caching

To optimize data management throughout the continuum, specifically addressing RQ (ii), we designed and developed the `CACHE-IT` framework within the Data Management layer. It is a

distributed framework that enables efficient, proactive edge caching strategies in IoT-based scenarios. CACHE-IT represents a step forward of the current IoT edge caching landscape by:

- **IoT oriented design**: Unlike network-oriented proactive caching, IoT application-oriented caching must consider varying constraints, such as low latency and constraints regarding AoI. CACHE-IT addresses interoperability issues of IoT by providing a dedicated device abstraction layer that integrates heterogeneous IoT devices. We tackle IoT environments' dynamism regarding sensors' volatile nature by incorporating mechanisms that adapt the caching strategy to reflect these changes.
- **Easy design and implementation of caching strategies**: we separate the caching strategy from the underlying architecture, allowing for customization based on the distinctive requirements of individual applications. The other features and caching optimization mechanism enabled by CACHE-IT are orthogonal to the strategy chosen.

CACHE-IT is transparent to the system that utilizes it, which enables it to adhere to basically any IoT system. We have included the possibility of utilizing FL-based strategies in the framework. This approach allows for the maintenance of client privacy since there is no need to centralize all clients' data. The results showed that the FL strategy's performance was slightly below the global one, still a privacy versus performance trade-off to be explored.

7.1.3 RQ (iii) – Trustworthiness

To answer RQ (iii) effectively, we developed a blockchain-based oracle framework that provides trustworthiness for IoT data. The proposed system is modular and suitable for critical applications that need trustworthiness but do not require real-time data collection. Two of its notable characteristics are:

- **no specific hardware required**: We have demonstrated that a fully decentralized system can be achieved with sufficient trustworthiness without requiring authenticated hardware, such as Trusted Execution Environments. This lack of specific hardware greatly expands the number of nodes that can join the system.
- **distributed data producers**: IoT devices are designed to be numerous, inexpensive, and interchangeable, which makes them inherently unreliable. We rely on distributed oracles and multiple data sources that share the same features in a specified geolocation to retrieve trustworthiness data. We need to trust not only the oracles but also— and mainly– the data coming from the IoT devices. Our architecture includes reputation algorithms for ranking and automatically selecting trustworthy data sources.

7.1.4 RQ (iv) – Real-world deployments

We address RQ (iv) by demonstrating the architecture's applicability in real-world deployments. The architecture was utilized in two diverse projects, MAC4PRO and Arrowhead Tools, which impose diverse infrastructure and requirements. In MAC4PRO, we deployed the same architecture in different monitoring domains with distinct edge-cloud continuum configurations. In the Arrowhead Tools project, we further explore the architecture to support multiple engineering pipelines, such as device management and energy optimization, which must be cohesive and encompassed by the proposed architecture. Our field experiments demonstrate the versatility of the architecture in addressing contrasting IoT monitoring scenarios.

The MAC4PRO project allowed us to explore different edge-cloud continuum configurations to demonstrate that the architectural layers are not bound to infrastructural locations (e.g., the cloud). We move parts of the data management processing to the extreme edge – the device microcontroller – which decreases the energy consumption and the number of bytes transferred between the edge and the cloud. On the other hand, the additional computation imposed by performing the feature extraction in the cloud is minimal, and the cloud offers greater stability compared to the edge. There is no universal optimal placement, but a careful trade-off should be analyzed in each scenario.

In the Arrowhead Project, it was necessary to interface with the subsystem developed by other partners, which greatly increased the system's heterogeneity. Components developed in this thesis, such as ZION (Section 3.2) and WAE (Section 3.3), were utilized to interconnect those ecosystems. Caching was proved a crucial enabler of the demonstrator – and it was the motivation for the development of `CACHE-IT` –, since without it, we could not produce a live demo in which we showed the energy and battery forecast of the devices by utilizing the Dr. Harvester simulator.

The architecture deployment in both projects demonstrated that architectural solutions are needed to support the data acquisition, processing, and visualization pipeline when deploying systems in real environments. When applying the architectural design to the field, many unexpected issues arise – such as the mentioned Dr. Harvester problem that sparked the development of `CACHE-IT` – and robust solutions are needed to overcome those problems.

7.1.5 Minor Contributions

Four minor contributions were produced as a side-effect of the main objective. Those are:

- **Quantitative comparison of IoT platforms**: In the background section of Chapter 3, we performed a small-scale performance evaluation to understand the behavior of

different open IoT platforms. We had an assumption that there was a clear trade-off between interoperability features offered by a given platform and its performance. However, the experiments demonstrated that such a relationship does not exist. Our results indicated that the more well-structured and organized the community or company that fosters the development of the platforms, the better its scalability.

- **ZION**: a reliable TDD implementation is a key element for real WoT-based systems as it enables efficient indexing and searching of WTs. ZION's usage throughout the thesis reflects this aspect. We developed ZION because the other implementation fell short of the feature that we valued the most: querying TDs. The WoT Hive utilized a semantic approach, which lacked performance, as demonstrated by the experiments in Subsection 3.2.1. TinyIoT resolves JSONPath queries by loading all the TDs into memory and searching through them, which is inefficient. Through ZION, we leverage the efficient JSONPath implementation of Postgres to achieve high scalability. We believe that ZION is a strong candidate to become the default TDD implementation.
- **IoT edge caching taxonomy**: caching IoT data on the edge has its challenges and features that deviate from the standard methods of network caching, such as the content dynamicity and the power constraints. In section 4.1 we proposed a novel IoT edge caching taxonomy to fulfill this gap. Subsequently, in Section 6.3.4, we conduct a performance evaluation of edge caching deployed in a real SHM scenario, investigating the impact of different taxonomy classes on the system performance.
- **Cache simulator**: we developed an open-source caching simulator. Currently, caching strategies, models, and optimizations are the main focus of researchers. Researchers typically evaluate proposed techniques for algorithmic efficiency and accuracy. However, there is a lack of studies that showcase domain metrics. Notably, `CACHE-IT` can calculate metrics such as latency and AoI for each data request, considering network delay and application time. On the network side, it models the delay from the user to the base station, the LAN communication between base stations, and the delay from the base station to the cloud hosting the application. It also supports client mobility, allows users to modify various configurations and parameters, and extends its functionality.

7.2 Current and future research directions

The work presented in this thesis provides a foundation for further research in the field of multi-layer architectures for IoT monitoring scenarios that consider the edge-cloud continuum. Possible future studies that can be pursued to build upon this work include the following:

- **Automatic Distribution of System Across the Edge-Cloud Continuum**: a prospective research direction involves exploring the automated distribution of applications throughout the edge-cloud continuum. The concept is to define the system as a monolith – instead of the typical microservices distributed architectures –and have a dedicated component that would analyze the code produced and the available infrastructure to partition the system and deploy it across computational nodes. This optimizes resource utilization while cutting development efforts.

- **Harnessing Artificial Intelligence for IoT Interoperability:** the goal is to harness the potential of artificial intelligence, particularly capitalizing on the advancements in Large Language Models (LLMs). The idea is to employ LLMs to parse the documentation or manuals associated with IoT devices and applications to automatically generate integration to other systems. An example would be to provide the documentation of a commercial sensor to this LLM, and it would return the respective Thing Description that maps the capabilities of the device as a W3C Web Thing.

- **Development of an IoT Caching Strategy Marketplace**: Another prospective study involves the development of a community-driven marketplace dedicated to IoT edge caching strategies that would be deployed utilizing `CACHE-IT`. This platform could be a repository for various caching strategies applicable across domains. We are particularly interested in exploring more complex strategies, potentially incorporating deep-learning neural networks to enhance caching efficiency. This research avenue seeks to optimize the current performance of caching mechanisms while enabling customization to each user's particular scenario.

- **Expansion of Performance Evaluation**: To further validate the proposed architecture, future investigations should expand the scope of evaluations. This involves extensive experiments under varied conditions and scenarios, including edge-cloud continuum configurations. Additionally, evaluations should consider scenarios where the same system monitors multiple sensing structures. The objective is further to assess the robustness and adaptability of the architecture, identifying specific areas for improvement and optimization in real-world deployment scenarios.

These research directions have the potential to significantly expand upon the work presented in this thesis and contribute to the adaptability of IoT architectures in various application domains.

7.3 Final Remarks

The work presented in this document establishes a sound foundation for further studies on flexible IoT monitoring architectures that leverage the edge-cloud continuum. The solutions built and developed in this thesis support the end applications of the traditional IoT pipeline, composed of sensor data collection, data processing, and visualization. As demonstrated in Chapter 6, deploying and operating an architecture in real-world scenarios requires other components due to the emerging complexities. The solutions developed in each architectural layer worked as a toolbox, which, depending on the requirements of the scenario, could be seamlessly included.

Our architecture enables us to connect heterogeneous devices, applications, and subsystems. The ability to easily integrate components proved to be a *must* in real systems and enables versatility since other deployments have requirements to interconnect with other components. Regarding data management, it is still a challenge in a distributed edge-cloud continuum setting to have the data in the exact location and time it needs to be. Edge caching solutions can improve this problem, and we believe that `CACHE-IT` represents a step closer to actually implementing complex caching solutions in real systems. Finally, our blockchain integration enables the retrieval of trustworthiness data of unreliable (but numerous) IoT devices.

In conclusion, this thesis advances the state-of-the-art IoT architecture by proposing and implementing a flexible architecture suitable for IoT monitoring scenarios that address key problems of the IoT landscape. The proposed architecture was validated by deploying it in real scenarios that leverage the edge-cloud continuum.

References

[1] W3C Working Group. Wot reference architecture (w3c recommendation 9 april 2020). http://www.w3.org/TR/wot-architecture, . Accessed on Jun 1, 2023.

[2] Maggi Bansal, Inderveer Chana, and Siobhán Clarke. A survey on iot big data: current status, 13 v's challenges, and future directions. *ACM Computing Surveys (CSUR)*, 53 (6):1–59, 2020.

[3] Md Zakirul Alam Bhuiyan, Jie Wu, Guojun Wang, and Jiannong Cao. Sensing and decision making in cyber-physical systems: The case of structural event monitoring. *IEEE Transactions on Industrial Informatics*, 12(6):2103–2114, 2016. doi: 10.1109/TII.2016.2518642.

[4] Yang Zhang, Junliang Chen, and B. Cheng. Integrating events into soa for iot services. *IEEE Communications Magazine*, 55:180–186, 2017. doi: 10.1109/MCOM.2017.1600359.

[5] Sergio Moreschini, Fabiano Pecorelli, Xiaozhou Li, Sonia Naz, David Hästbacka, and Davide Taibi. Cloud continuum: The definition. *IEEE Access*, 10:131876–131886, 2022. doi: 10.1109/ACCESS.2022.3229185.

[6] Gang Wang, M. Nixon, and Mike Boudreaux. Toward cloud-assisted industrial iot platform for large-scale continuous condition monitoring. *Proceedings of the IEEE*, 107:1193–1205, 2019. doi: 10.1109/JPROC.2019.2914021.

[7] Ren Li, Tianjin Mo, Jianxi Yang, Shixin Jiang, Tong Li, and Yiming Liu. Ontologies-based domain knowledge modeling and heterogeneous sensor data integration for bridge health monitoring systems. *IEEE Transactions on Industrial Informatics*, 17 (1):321–332, 2021. doi: 10.1109/TII.2020.2967561.

[8] Jianqing Fan, Fang Han, and Han Liu. Challenges of big data analysis. *National science review*, 1(2):293–314, 2014.

[9] Mayank Mishra, Paulo B Lourenço, and Gunturi Venkata Ramana. Structural health monitoring of civil engineering structures by using the internet of things: A review. *Journal of Building Engineering*, 48:103954, 2022.

[10] Ivan Zyrianoff, Lorenzo Gigli, Federico Montori, Luca Sciullo, Carlos Kamienski, and Marco Di Felice. Cache-it: A distributed architecture for proactive edge caching in heterogeneous iot scenarios. *Available at SSRN 4481856*.

[11] Guangming Cui, Qiang He, Feifei Chen, Hai Jin, and Yun Yang. Trading off between user coverage and network robustness for edge server placement. *IEEE Transactions on Cloud Computing*, 10:2178–2189, 2020. doi: 10.1109/TCC.2020.3008440.

[12] Ivan Zyrianoff, Alexandre Heideker, Dener Silva, João Kleinschmidt, Juha-Pekka Soininen, Tullio Salmon Cinotti, and Carlos Kamienski. Architecting and deploying iot smart applications: A performance–oriented approach. *Sensors*, 20(1):84, 2020.

[13] C. Scuro, F. Lamonaca, S. Porzio, G. Milani, and R.S. Olivito. Internet of things (iot) for masonry structural health monitoring (shm): Overview and examples of innovative systems. *Construction and Building Materials*, 290:123092, 2021. ISSN 0950-0618.

[14] Visvesh Naraharisetty, Venkat Surendar Talari, Sairam Neridu, Prafulla Kalapatapu, and Venkata Dilip Kumar Pasupuleti. Cloud architecture for iot based bridge monitoring applications. In *2021 International Conference on Emerging Techniques in Computational Intelligence (ICETCI)*, pages 39–42, 2021. doi: 10.1109/ICETCI51973.2021.9574044.

[15] Marco Claudio De Simone, Angelo Lorusso, and Domenico Santaniello. Predictive maintenance and structural health monitoring via iot system. In *2022 IEEE Workshop on Complexity in Engineering (COMPENG)*, pages 1–4, 2022. doi: 10.1109/COMPENG50184.2022.9905441.

[16] Seongwoon Jeong and Kincho Law. An iot platform for civil infrastructure monitoring. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 746–754, 2018. doi: 10.1109/COMPSAC.2018.00111.

[17] Hung V. Dang, Mallik Tatipamula, and Huan X. Nguyen. Cloud-based digital twinning for structural health monitoring using deep learning. *IEEE Transactions on Industrial Informatics*, 18(6):3820–3830, 2022. doi: 10.1109/TII.2021.3115119.

[18] Gilles Privat and A Medvedev. Guidelines for modelling with ngsi-ld. *ETSI White Paper. Available online: https://www. etsi. org/images/files/ETSIWhitePapers/etsi_wp_42_NGSI_LD. pdf (accessed on 21 June 2022)*, 2021.

[19] FIWARE Foundation. Fiware: The open source platform for our smart digital future, . www.fiware.org, Accessed Mar. 10, 2021.

[20] Swagger. Openapi specification 3.1.0. https://swagger.io/specification/, Feb 2021. Accessed on Jun 1, 2023.

[21] Jerker Delsing. *Iot automation: Arrowhead framework*. Crc Press, 2017.

[22] Jingjing Yao, Tao Han, and Nirwan Ansari. On mobile edge caching. *IEEE Communications Surveys Tutorials*, 21(3):2525–2553, 2019. doi: 10.1109/COMST.2019.2908280.

[23] Lorenzo Gigli, Ivan Zyrianoff, Federica Zonzoni, Denis Bogomolov, Nicola Testoni, Luca De Marchi, Giuseppe Augugliaro, Canio Mennutti, Alessandro Marzani, and Marco Di Felice. Proactive caching in the edge-cloud continuum with federated learning. accepted for publication in the CCNC 2024 proceedings, 2024.

[24] Hamda Al-Breiki, Muhammad Habib Ur Rehman, Khaled Salah, and Davor Svetinovic. Trustworthy blockchain oracles: Review, comparison, and open research challenges. *IEEE Access*, 8:85675–85685, 2020. doi: 10.1109/ACCESS.2020.2992698.

[25] Lorenzo Gigli, Ivan Zyrianoff, Federica Zonzoni, Denis Bogomolov, Nicola Testoni, Luca De Marchi, Giuseppe Augugliaro, Canio Mennutti, Alessandro Marzani, and Marco Di Felice. Next generation edge-cloud continuum architecture for structural health monitoring. under review in IEEE Transactions on Industrial Informatics, 2023.

[26] Dragi Kimovski, Roland Mathá, Josef Hammer, Narges Mehran, Hermann Hellwagner, and Radu Prodan. Cloud, fog, or edge: Where to compute? *IEEE Internet Computing*, 25(4):30–36, 2021. doi: 10.1109/MIC.2021.3050613.

[27] Luiz Bittencourt, Roger Immich, Rizos Sakellariou, Nelson Fonseca, Edmundo Madeira, Marilia Curado, Leandro Villas, Luiz DaSilva, Craig Lee, and Omer Rana. The internet of things, fog and cloud continuum: Integration and challenges. *Internet of Things*, 3:134–155, 2018.

[28] Amine Abouaomar, S. Cherkaoui, Zoubeir Mlika, and A. Kobbane. Resource provisioning in edge computing for latency-sensitive applications. *IEEE Internet of Things Journal*, 8:11088–11099, 2021. doi: 10.1109/JIOT.2021.3052082.

[29] Lianhai Wang, Yannan Li, Qiming Yu, and Yong Yu. Outsourced data integrity checking with practical key update in edge-cloud resilient networks. *IEEE Wireless Communications*, 29(3):56–62, 2022. doi: 10.1109/MWC.002.2100597.

[30] Zhipeng Cheng, Zhibin Gao, Minghui Liwang, Lianfen Huang, Xiaojiang Du, and Mohsen Guizani. Intelligent task offloading and energy allocation in the uav-aided mobile edge-cloud continuum. *IEEE Network*, 35(5):42–49, 2021. doi: 10.1109/MNET.010.2100025.

[31] Danylo Khalyeyev, Tomáš Bureš, and Petr Hnětynka. Towards a reference component model of edge-cloud continuum. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, pages 91–95, 2023. doi: 10.1109/ICSA-C57050.2023.00030.

[32] Sarah A Al-Qaseemi, Hajer A Almulhim, Maria F Almulhim, and Saqib Rasool Chaudhry. Iot architecture challenges and issues: Lack of standardization. In *2016 Future technologies conference (FTC)*, pages 731–738. IEEE, 2016.

[33] Miao Wu, Ting-Jie Lu, Fei-Yang Ling, Jing Sun, and Hui-Ying Du. Research on the architecture of internet of things. In *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, volume 5, pages V5–484–V5–487, 2010. doi: 10.1109/ICACTE.2010.5579493.

[34] Omar Said and Mehedi Masud. Towards internet of things: Survey and future vision. *International Journal of Computer Networks*, 5(1):1–17, 2013.

[35] Federica Zonzini, Cristiano Aguzzi, Lorenzo Gigli, Luca Sciullo, Nicola Testoni, Luca De Marchi, Marco Di Felice, Tullio Salmon Cinotti, Canio Mennuti, and Alessandro Marzani. Structural health monitoring and prognostic of industrial plants and civil structures: A sensor to cloud architecture. *IEEE Instrumentation & Measurement Magazine*, 23(9):21–27, 2020.

[36] Cristiano Aguzzi, Lorenzo Gigli, Luca Sciullo, Angelo Trotta, Federica Zonzini, Luca De Marchi, Marco Di Felice, Alessandro Marzani, and Tullio Salmon Cinotti. Modron: A scalable and interoperable web of things platform for structural health monitoring. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–7, 2021.

[37] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE internet of things journal*, 4(5):1125–1142, 2017.

[38] Francesco Lamonaca, Carmelo Scuro, Domenico Grimaldi, Renato Sante Olivito, Paolo Francesco Sciammarella, and Domenico Luca Carnì. A layered iot-based architecture for a distributed structural health monitoring system system. *Acta Imeko*, 8(2):45–52, 2019.

[39] Ivan Zyrianoff, Alexandre Heideker, Dener Silva, João Kleinschmidt, Juha-Pekka Soininen, Tullio Salmon Cinotti, and Carlos Kamienski. Architecting and deploying iot smart applications: A performance–oriented approach. *Sensors*, 20(1), 2020. ISSN 1424-8220. doi: 10.3390/s20010084. URL https://www.mdpi.com/1424-8220/20/1/84.

[40] Carmelo Scuro, Paolo Francesco Sciammarella, Francesco Lamonaca, Renato Sante Olivito, and Domenico Luca Carni. Iot for structural health monitoring. *IEEE Instrumentation & Measurement Magazine*, 21(6):4–14, 2018.

[41] C Scuro, F Lamonaca, S Porzio, G Milani, and RS Olivito. Internet of things (iot) for masonry structural health monitoring (shm): Overview and examples of innovative systems. *Construction and Building Materials*, 290:123092, 2021.

[42] Gang Wang, Mark Nixon, and Mike Boudreaux. Toward cloud-assisted industrial iot platform for large-scale continuous condition monitoring. *Proceedings of the IEEE*, 107(6):1193–1205, 2019. doi: 10.1109/JPROC.2019.2914021.

[43] Peng Qian, Bo Feng, Dahai Zhang, Xiange Tian, and Yulin Si. Iot-based approach to condition monitoring of the wave power generation system. *IET Renewable Power Generation*, 13(12):2207–2214, 2019.

[44] Hanbo Yang, Zheng Sun, Gedong Jiang, Fei Zhao, Xufeng Lu, and Xuesong Mei. Cloud-manufacturing-based condition monitoring platform with 5g and standard information model. *IEEE Internet of Things Journal*, 8(8):6940–6948, 2020.

[45] Cristiano Aguzzi, Lorenzo Gigli, Luca Sciullo, Angelo Trotta, and Marco Di Felice. From cloud to edge: Seamless software migration at the era of the web of things. *IEEE Access*, 8:228118–228135, 2020. doi: 10.1109/ACCESS.2020.3045632.

[46] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, and Minyi Guo. Adaptive resource efficient microservice deployment in cloud-edge continuum. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1825–1840, 2022. doi: 10.1109/TPDS.2021.3128037.

[47] Yan Chen, Yanjing Sun, Chenyang Wang, and Tarik Taleb. Dynamic task allocation and service migration in edge-cloud iot system based on deep reinforcement learning. *IEEE Internet of Things Journal*, 9(18):16742–16757, 2022. doi: 10.1109/JIOT.2022.3164441.

[48] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. Interoperability in internet of things: Taxonomies and open challenges. *Mobile networks and applications*, 24:796–809, 2019.

[49] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. *Ambient intelligence*, pages 115–148, 2005.

[50] Thomas Watteyne, Xavier Vilajosana, Branko Kerkez, Fabien Chraim, Kevin Weekly, Qin Wang, Steven Glaser, and Kris Pister. Openwsn: a standards-based low-power wireless development environment. *Transactions on Emerging Telecommunications Technologies*, 23(5):480–493, 2012.

[51] Dominique Guinard. *A Web of Things Application Architecture. Integrating the Real-World into the Web*. Doctoral thesis, ETH Zurich, Zürich, 2011.

[52] Elena Reshetova and Michael McCool. Web of things (wot) security and privacy guidelines. W3C recommendation, November 2019. https://www.w3.org/TR/wot-security/.

[53] Andrea Cimmino, Michael McCool, Farshid Tavakolizadeh, and Kunihiko Toumura. Web of things (wot) discovery. Proposed recommendation, World Wide Web Consortium (W3C), July 11 2023. URL https://www.w3.org/TR/2023/PR-wot-discovery-20230711/.

[54] Stefan Gössner and C. Bormann. Jsonpath – xpath for json. Ietf internet draft, January 2021. https://www.ietf.org/archive/id/draft-goessner-dispatch-jsonpath-00.html.

[55] Jonathan Robie, Michael Dyck, and Josh Spiegel. Xml path language (xpath) 3.1. W3C recommendation, March 2017. https://www.w3.org/TR/xpath-31/.

[56] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. Sparql 1.1 query language. W3C recommendation, March 2013. https://www.w3.org/TR/sparql11-query/.

[57] Ivan Zyrianoff, Lorenzo Gigli, Federico Montori, Carlos Kamienski, and Marco Di Felice. Two-way integration of service-oriented systems-of-systems with the web of things. In *IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6, 2021. doi: 10.1109/IECON48115.2021.9589619.

[58] Ivan Zyrianoff, Alexandre Heideker, Luca Sciullo, Carlos Kamienski, and Marco Di Felice. Interoperability in open iot platforms: Wot-fiware comparison and integration. In *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 169–174, 2021. doi: 10.1109/SMARTCOMP52413.2021.00043.

[59] Luca Sciullo, Lorenzo Gigli, Federico Montori, Angelo Trotta, and Marco Di Felice. A survey on the web of things. *IEEE Access*, 10:47570–47596, 2022. doi: 10.1109/ACCESS.2022.3171575.

[60] Yuchao Zhou, Suparna De, Wei Wang, and Klaus Moessner. Search techniques for the web of things: A taxonomy and survey. *Sensors*, 16(5), 2016. ISSN 1424-8220. doi: 10.3390/s16050600. URL https://www.mdpi.com/1424-8220/16/5/600.

[61] Zhiming Ding, Zhikui Chen, and Qi Yang. Iot-svksearch: a real-time multimodal search engine mechanism for the internet of things. *International Journal of Communication Systems*, 27(6):871–897, 2014. doi: https://doi.org/10.1002/dac.2647. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/dac.2647.

[62] Luoyao Hao and Henning Schulzrinne. Goldie: Harmonization and orchestration towards a global directory for iot. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021. doi: 10.1109/INFOCOM42981.2021.9488752.

[63] Luoyao Hao, Vibhas Naik, and Henning Schulzrinne. Dbac: Directory-based access control for geographically distributed iot systems. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 360–369, 2022. doi: 10.1109/INFOCOM48880.2022.9796804.

[64] Muhammad Rehan Faheem, Tayyaba Anees, Muzammil Hussain, Allah Ditta, Hani Alquhayz, and Muhammad Adnan Khan. Indexing in wot to locate indoor things. *IEEE Access*, 11:53497–53517, 2023. doi: 10.1109/ACCESS.2023.3272691.

[65] Farshid Tavakolizadeh and Shreekantha Devasya. Thing directory: Simple and lightweight registry of iot device metadata. *Journal of Open Source Software*, 6 (60):3075, 2021.

[66] Raul Garcıa-Castro Andrea Cimmino. Wothive: Enabling syntactic and semantic discovery in the web of things. *Open Journal of Internet of Things (OJIOT)*, 8(1): 54–65, 2022.

[67] A Gluhak, O Vermesan, R Bahr, F Clari, TM Maria, T Delgado, A Hoeer, F Bösenberg, M Senigalliesi, and V Barchett. Bdeliverable d03. 01 report on iot platform activities-unify-iot, 2016.

[68] FIWARE Foundation. Iot agents. https://fiware-academy.readthedocs.io/en/latest/iot-agents/idas/index.html, Accessed Mar. 9, 2021.

[69] Ivan Zyrianoff, Fabrizio Borelli, Gabriela Biondi, Alexandre Heideker, and Carlos Kamienski. Scalability of real-time iot-based applications for smart cities. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00688–00693, 2018. doi: 10.1109/ISCC.2018.8538451.

[70] Alessandra Galli, Giada Giorgi, and Claudio Narduzzi. Multi-user ecg monitoring system based on ieee standard 802.15.6. pages 1–6, 2019. doi: 10.1109/IWMN.2019.8805046.

[71] Elisa Spanò, Stefano Di Pascoli, and Giuseppe Iannaccone. Low-power wearable ecg monitoring system for multiple-patient remote monitoring. *IEEE Sensors Journal*, 16 (13):5452–5462, 2016. doi: 10.1109/JSEN.2016.2564995.

[72] W3C Working Group. Eclipse thingweb node-wot, . https://github.com/eclipse/thingweb.node-wot, Accessed Mar. 9, 2021.

[73] Diego GS Pivoto, Luiz FF de Almeida, Rodrigo da Rosa Righi, Joel JPC Rodrigues, Alexandre Baratella Lugli, and Antonio M Alberti. Cyber-physical systems architectures for industrial internet of things applications in industry 4.0: A literature review. *Journal of Manufacturing Systems*, 58:176–192, 2021.

[74] Géza Kulcsár, Kadosa Koltai, Szvetlin Tanyi, Bálint Péceli, Ákos Horváth, Zoltán Micskei, and Pál Varga. From models to management and back: Towards a system-of-systems engineering toolchain. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6. IEEE, 2020.

[75] Carlos Kamienski, Juha-Pekka Soininen, Markus Taumberger, Ramide Dantas, Attilio Toscano, Tullio Salmon Cinotti, Rodrigo Filev Maia, and André Torre Neto. Smart water management platform: Iot-based precision irrigation for agriculture. *Sensors*, 19(2), 2019. ISSN 1424-8220. doi: 10.3390/s19020276. URL https://www.mdpi.com/1424-8220/19/2/276.

[76] Rodrigo Togneri, Carlos Kamienski, Ramide Dantas, Ronaldo Prati, Attilio Toscano, Juha-Pekka Soininen, and Tullio Salmon Conic. Advancing iot-based smart irrigation. *IEEE Internet of Things Magazine*, 2(4):20–25, 2019.

[77] ChirpStack. Chirpstack, open-source lorawan® network server stack. https://www.chirpstack.io/, Accessed Mar. 9, 2021.

[78] FIWARE Foundation. Fiware iot agent ultralight, . https://fiware-iotagent-ul.readthedocs.io/en/latest/usermanual, Accessed Mar. 9, 2021.

[79] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273, 2016.

[80] Sebastian Kaebisch, Takuki Kamiya, Michael McCool, Victor Charpenay, and Matthias Kovatsch. Web of things (wot) thing description. W3C recommendation, April 2020. https://www.w3.org/TR/wot-thing-description/.

[81] Ivan Zyrianoff, Lorenzo Gigli, Federico Montori, Carlos Kamienski, and Marco Di Felice. Two-way integration of service-oriented systems-of-systems with the web of things. In *IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6, 2021. doi: 10.1109/IECON48115.2021.9589619.

[82] Luca Sciullo, Lorenzo Gigli, Angelo Trotta, and Marco Di Felice. Wot store: Managing resources and applications on the web of things. *Internet of Things*, 9:100164, 2020. ISSN 2542-6605. doi: https://doi.org/10.1016/j.iot.2020.100164. URL https://www.sciencedirect.com/science/article/pii/S254266052030007X.

[83] Ivan Dimitry Zyrianoff. Wot-arrowhead adapter, . https://github.com/UniBO-PRISMLab/wot-arrowhead-adapter, Accessed Jul. 22, 2021.

[84] Ivan Dimitry Zyrianoff. Openapi generator, . https://github.com/UniBO-PRISMLab/openAPIgenerator, Accessed Jul. 22, 2021.

[85] IANA. Hypertext transfer protocol (http) method registry. https://www.iana.org/assignments/http-methods/http-methods.xhtml, Accessed Jul. 27, 2021.

[86] Carlo Puliafito, Lorenzo Gigli, Ivan Zyrianoff, Federico Montori, Antonio Virdis, Stefano Di Pascoli, Enzo Mingozzi, and Marco Di Felice. Joint power control and structural health monitoring in industry 4.0 scenarios using eclipse arrowhead and web of things. In *2022 IEEE 5th International Conference on Industrial Cyber-Physical Systems (ICPS)*, pages 1–6, 2022. doi: 10.1109/ICPS51978.2022.9816975.

[87] Laha Ale, Ning Zhang, Huici Wu, Dajiang Chen, and Tao Han. Online proactive caching in mobile edge computing using bidirectional deep recurrent neural network. *IEEE Internet of Things Journal*, 6(3):5520–5530, 2019. doi: 10.1109/JIOT.2019.2903245.

[88] Shailendra Rathore, Jung Hyun Ryu, Pradip Kumar Sharma, and Jong Hyuk Park. Deepcachnet: A proactive caching framework based on deep learning in cellular networks. *IEEE Network*, 33(3):130–138, 2019. doi: 10.1109/MNET.2019.1800058.

[89] The-Vi Nguyen, Nhu-Ngoc Dao, Van Dat Tuong, Wonjong Noh, and Sungrae Cho. User-aware and flexible proactive caching using lstm and ensemble learning in iot-mec networks. *IEEE Internet of Things Journal*, 9(5):3251–3269, 2022. doi: 10.1109/JIOT.2021.3097768.

[90] Ivan Zyrianoff, Angelo Trotta, Luca Sciullo, Federico Montori, and Marco Di Felice. Iot edge caching: Taxonomy, use cases and perspectives. *IEEE Internet of Things Magazine*, 5(3):12–18, 2022. doi: 10.1109/IOTM.001.2200112.

[91] Xiaomin Li and Jiafu Wan. Proactive caching for edge computing-enabled industrial mobile wireless networks. *Future Generation Computer Systems*, 89:89–97, 2018. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2018.06.017. URL https://www.sciencedirect.com/science/article/pii/S0167739X17329588.

[92] Rodolfo W. L. Coutinho and Azzedine Boukerche. Modeling and analysis of a shared edge caching system for connected cars and industrial iot-based applications. *IEEE Transactions on Industrial Informatics*, 16(3):2003–2012, 2020. doi: 10.1109/TII.2019.2938529.

[93] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y. Zomaya. Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet of Things Journal*, 7(8):7457–7469, 2020. doi: 10.1109/JIOT.2020.2984887.

[94] Mung Chiang, Sangtae Ha, Fulvio Risso, Tao Zhang, and I. Chih-Lin. Clarifying fog computing and networking: 10 questions and answers. *IEEE Communications Magazine*, 55(4):18–20, 2017. doi: 10.1109/MCOM.2017.7901470.

[95] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE Access*, 8:85714–85728, 2020. doi: 10.1109/ACCESS.2020.2991734.

[96] Ivan Zyrianoff, Lorenzo Gigli, Federico Montori, Cristiano Aguzzi, Sebastian Kaebisch, and Marco Di Felice. Seamless integration of restful web services with the web of things. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 427–432, 2022. doi: 10.1109/PerComWorkshops53856.2022.9767531.

[97] M. Ishtiaque A. Zahed, Iftekhar Ahmad, Daryoush Habibi, and Quoc Viet Phung. Green and secure computation offloading for cache-enabled iot networks. *IEEE Access*, 8:63840–63855, 2020. doi: 10.1109/ACCESS.2020.2982669.

[98] Martina Pappalardo, Enzo Mingozzi, and Antonio Virdis. A model-driven approach to aol-based cache management in iot. In *2021 IEEE 26th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 1–6, 2021. doi: 10.1109/CAMAD52502.2021.9617772.

[99] Jingjing Yao, Tao Han, and Nirwan Ansari. On mobile edge caching. *IEEE Communications Surveys Tutorials*, 21(3):2525–2553, 2019. doi: 10.1109/COMST.2019.2908280.

[100] Giuseppe Ruggeri, Marica Amadeo, Claudia Campolo, Antonella Molinaro, and Antonio Iera. Caching popular transient iot contents in an sdn-based edge infrastructure. *IEEE Transactions on Network and Service Management*, 18(3):3432–3447, 2021. doi: 10.1109/TNSM.2021.3056891.

[101] Jingjing Yao and Nirwan Ansari. Caching in energy harvesting aided internet of things: A game-theoretic approach. *IEEE Internet of Things Journal*, 6(2):3194–3201, 2019. doi: 10.1109/JIOT.2018.2880483.

[102] Muhammad Ali Naeem, Tu N. Nguyen, Rashid Ali, Korhan Cengiz, Yahui Meng, and Tahir Khurshaid. Hybrid cache management in iot-based named data networking. *IEEE Internet of Things Journal*, 9(10):7140–7150, 2022. doi: 10.1109/JIOT.2021.3075317.

[103] Fadi M Al-Turjman, Muhammad Imran, and Athanasios V Vasilakos. Value-based caching in information-centric wireless body area networks. *Sensors*, 17(1):181, 2017.

[104] Franklin M. Ribeiro Junior, Reinaldo A.C. Bianchi, Ronaldo C. Prati, Kari Kolehmainen, Juha-Pekka Soininen, and Carlos A. Kamienski. Data reduction based on machine learning algorithms for fog computing in iot smart agriculture. *Biosystems Engineering*, 2022. ISSN 1537-5110. doi: https://doi.org/10.1016/j.biosystemseng.2021.12.021. URL https://www.sciencedirect.com/science/article/pii/S1537511021003299.

[105] Mengyu Li, LanLan Rui, Xuesong Qiu, Shaoyong Guo, and Xiuzhi Yu. Design of a service caching and task offloading mechanism in smart grid edge network. In *2019 15th International Wireless Communications Mobile Computing Conference (IWCMC)*, pages 249–254, 2019. doi: 10.1109/IWCMC.2019.8766672.

[106] Huan Zhou, Zhenyu Zhang, Dawei Li, and Zhou Su. Joint optimization of computing offloading and service caching in edge computing-based smart grid. *IEEE Transactions on Cloud Computing*, pages 1–1, 2022. doi: 10.1109/TCC.2022.3163750.

[107] Ke Zhang, Jiayu Cao, Sabita Maharjan, and Yan Zhang. Digital twin empowered content caching in social-aware vehicular edge networks. *IEEE Transactions on Computational Social Systems*, 9(1):239–251, 2022. doi: 10.1109/TCSS.2021.3068369.

[108] Shan Zhang, Hongbin Luo, Junling Li, Weisen Shi, and Xuemin Shen. Hierarchical soft slicing to meet multi-dimensional qos demand in cache-enabled vehicular networks. *IEEE Transactions on Wireless Communications*, 19(3):2150–2162, 2020. doi: 10.1109/TWC.2019.2962798.

[109] Genghua Yu, Yixin He, Jian Wu, Zhigang Chen, and Jianping Pan. Mobility-aware proactive edge caching for large files in the internet of vehicles. *IEEE Internet of Things Journal*, pages 1–1, 2023. doi: 10.1109/JIOT.2023.3240423.

[110] A.H.M. Ahmadullah Chowdhury, Irfanul Islam, M. Ishtiaque A. Zahed, and Iftekhar Ahmad. An optimal strategy for uav-assisted video caching and transcoding. *Ad Hoc Networks*, 144:103155, 2023. ISSN 1570-8705. doi: https://doi.org/10.1016/j.adhoc.2023.103155.

[111] Kyi Thar, Thant Zin Oo, Yan Kyaw Tun, Do Hyeon Kim, Ki Tae Kim, and Choong Seon Hong. A deep learning model generation framework for virtualized multi-access edge cache management. *IEEE Access*, 7:62734–62749, 2019. doi: 10.1109/ACCESS.2019.2916080.

[112] Dinh Thai Hoang, Dusit Niyato, Diep N. Nguyen, Eryk Dutkiewicz, Ping Wang, and Zhu Han. A dynamic edge caching framework for mobile 5g networks. *IEEE Wireless Communications*, 25(5):95–103, 2018. doi: 10.1109/MWC.2018.1700360.

[113] Muhammad Umar Farooq, Muhammad Zeeshan, Muhammad Talha Jahangir, and Muhammad Asif. A novel cooperative micro-caching algorithm based on fuzzy inference through nfv in ultra-dense iot networks. *Journal of Network and Systems Management*, 30(1):20, Oct 2021. ISSN 1573-7705. doi: 10.1007/s10922-021-09632-6.

[114] Fan Zhang, Guangjie Han, Li Liu, Miguel Martínez-García, and Yan Peng. Joint optimization of cooperative edge caching and radio resource allocation in 5g-enabled massive iot networks. *IEEE Internet of Things Journal*, 8(18):14156–14170, 2021. doi: 10.1109/JIOT.2021.3068427.

[115] Siya Xu, Xin Liu, Shaoyong Guo, Xuesong Qiu, and Luoming Meng. Mecc: A mobile edge collaborative caching framework empowered by deep reinforcement learning. *IEEE Network*, 35(4):176–183, 2021. doi: 10.1109/MNET.011.2000663.

[116] Xiuhua Li, Xiaofei Wang, Chunsheng Zhu, Wei Cai, and Victor C. M. Leung. Caching-as-a-service: Virtual caching framework in the cloud-based mobile networks. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 372–377, 2015. doi: 10.1109/INFCOMW.2015.7179413.

[117] Yixue Hao, Yiming Miao, Long Hu, M. Shamim Hossain, Ghulam Muhammad, and Syed Umar Amin. Smart-edge-cocaco: Ai-enabled smart edge with joint computation, caching, and communication in heterogeneous iot. *IEEE Network*, 33(2):58–64, 2019. doi: 10.1109/MNET.2019.1800235.

[118] Chang Kyung Kim, TaeYoung Kim, SuKyoung Lee, Seungkyun Lee, Anna Cho, and Mun-Suk Kim. Delay-aware distributed program caching for iot-edge networks. *Plos one*, 17(7):e0270183, 2022.

[119] Xu Zhao and Qi Zhu. Mobility-aware and interest-predicted caching strategy based on iot data freshness in d2d networks. *IEEE Internet of Things Journal*, 8(7):6024–6038, 2021. doi: 10.1109/JIOT.2020.3033552.

[120] Yuris Mulya Saputra, Dinh Thai Hoang, Diep N. Nguyen, Eryk Dutkiewicz, Dusit Niyato, and Dong In Kim. Distributed deep learning at the edge: A novel proactive and cooperative caching framework for mobile edge networks. *IEEE Wireless Communications Letters*, 8(4):1220–1223, 2019. doi: 10.1109/LWC.2019.2912365.

[121] Yuming Zhang, Bohao Feng, Wei Quan, Aleteng Tian, Keshav Sood, Youfang Lin, and Hongke Zhang. Cooperative edge caching: A multi-agent deep learning based approach. *IEEE Access*, 8:133212–133224, 2020. doi: 10.1109/ACCESS.2020.3010329.

[122] Tan Li and Linqi Song. Federated online learning aided multi-objective proactive caching in heterogeneous edge networks. *IEEE Transactions on Cognitive Communications and Networking*, pages 1–1, 2023. doi: 10.1109/TCCN.2023.3262243.

[123] Yin Zhang, Yujie Li, Ranran Wang, Jianmin Lu, Xiao Ma, and Meikang Qiu. Psac: Proactive sequence-aware content caching via deep learning at the network edge. *IEEE Transactions on Network Science and Engineering*, 7(4):2145–2154, 2020. doi: 10.1109/TNSE.2020.2990963.

[124] Dongyang Li, Haixia Zhang, Hui Ding, Tiantian Li, Daojun Liang, and Dongfeng Yuan. User preference learning-based proactive edge caching for d2d-assisted wireless networks. *IEEE Internet of Things Journal*, pages 1–1, 2023. doi: 10.1109/JIOT.2023.3244621.

[125] Haibo Wu, Jun Li, and Jiang Zhi. Could end system caching and cooperation replace in-network caching in ccn? In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 101–102, 2015.

[126] Haibo Wu, Yaogong Xu, and Jun Li. Ptf: Popularity-topology-freshness-based caching strategy for icn-iot networks. *Computer Communications*, 204:147–157, 2023.

[127] Zhengxin Yu, Jia Hu, Geyong Min, Haochuan Lu, Zhiwei Zhao, Haozhe Wang, and Nektarios Georgalas. Federated learning based proactive content caching in edge computing. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2018. doi: 10.1109/GLOCOM.2018.8647616.

[128] Xiaofei Wang, Yiwen Han, Chenyang Wang, Qiyang Zhao, Xu Chen, and Min Chen. In-edge ai: Intelligentizing mobile edge computing, caching and communication by federated learning. *IEEE Network*, 33(5):156–165, 2019. doi: 10.1109/MNET.2019.1800286.

[129] Dewen Qiao, Songtao Guo, Defang Liu, Saiqin Long, Pengzhan Zhou, and Zhetao Li. Adaptive federated deep reinforcement learning for proactive content caching in edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4767–4782, 2022. doi: 10.1109/TPDS.2022.3201983.

[130] Chunlin Li, Yong Zhang, and Youlong Luo. A federated learning-based edge caching approach for mobile edge computing-enabled intelligent connected vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 24(3):3360–3369, 2023. doi: 10.1109/TITS.2022.3224395.

[131] Subina Khanal, Kyi Thar, and Eui-Nam Huh. Route-based proactive content caching using self-attention in hierarchical federated learning. *IEEE Access*, 10:29514–29527, 2022. doi: 10.1109/ACCESS.2022.3157637.

[132] Yijing Li, Shihong Hu, and Guanghui Li. Cvc: A collaborative video caching framework based on federated learning at the edge. *IEEE Transactions on Network and Service Management*, 19(2):1399–1412, 2022. doi: 10.1109/TNSM.2021.3135306.

[133] Mohammad Reiss-Mirzaei, Mostafa Ghobaei-Arani, and Leila Esmaeili. A review on the edge caching mechanisms in the mobile edge computing: A social-aware perspective. *Internet of Things*, 22:100690, 2023. ISSN 2542-6605. doi: https://doi.org/10.1016/j.iot.2023.100690.

[134] Dener Silva, Alexandre Heideker, Ivan D. Zyrianoff, João H. Kleinschmidt, Luca Roffia, Juha-Pekka Soininen, and Carlos A. Kamienski. A management architecture for iot smart solutions: Design and implementation. *Journal of Network and Systems Management*, 30(2):35, Jan 2022. ISSN 1573-7705. doi: 10.1007/s10922-022-09648-6. URL https://doi.org/10.1007/s10922-022-09648-6.

[135] Ivan Zyrianoff, Lorenzo Gigli, Federico Montori, Cristiano Aguzzi, Sebastian Kaebisch, and Marco Di Felice. Artifact: C3po - converter of open api specification to wot objects. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 185–186, 2022. doi: 10.1109/PerComWorkshops53856.2022.9767293.

[136] Ivan Zyrianoff. Cache-it cache worker. https://github.com/ivanzy/cache-worker, .

[137] Ivan Zyrianoff. Cache-it simulator. https://github.com/UniBO-PRISMLab/cache_simulator, .

[138] Fabio Palumbo, Giuseppe Aceto, Alessio Botta, Domenico Ciuonzo, Valerio Persico, and Antonio Pescapé. Characterization and analysis of cloud-to-user latency: The case of azure and aws. *Computer Networks*, 184:107693, 2021. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2020.107693.

[139] Stanislav Špaček, Petr Velan, Pavel Čeleda, and Daniel Tovarňák. Encrypted web traffic dataset: Event logs and packet traces. *Data in Brief*, 42:108188, 2022. ISSN 2352-3409. doi: https://doi.org/10.1016/j.dib.2022.108188.

[140] Gary A. Stafford. Lan network stability. https://www.kaggle.com/datasets/garystafford/ping-data. "Accessed on May 31, 2023".

[141] Marco Pettorali, Francesca Righetti, Carlo Vallati, Sajal K. Das, and Giuseppe Anastasi. Mobility management in industrial iot environments. In *2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 271–280, 2022. doi: 10.1109/WoWMoM54355.2022.00046.

[142] Zihao Sang, Songtao Guo, Quyuan Wang, and Ying Wang. Gcs: Collaborative video cache management strategy in multi-access edge computing. *Ad Hoc Networks*, 117: 102516, 2021. ISSN 1570-8705. doi: https://doi.org/10.1016/j.adhoc.2021.102516.

[143] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.

[144] Giulio Caldarelli. Understanding the blockchain oracle problem: A call for action. *Information*, 11(11):509, 2020.

[145] Pawel Szalachowski. Blockchain-based tls notary service. *ArXiv*, abs/1804.00875, 2018. URL https://api.semanticscholar.org/CorpusID:4626463.

[146] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*, pages 270–282, 2016.

[147] Adán Sánchez de Pedro, Daniele Levi, and Luis Iván Cuende. Witnet: A decentralized oracle network protocol. *arXiv preprint arXiv:1711.09756*, 2017.

[148] Jack Peterson, Joseph Krug, Micah Zoltu, Austin K Williams, and Stephanie Alexander. Augur: a decentralized oracle and prediction market platform (v2. 0). *Whitepaper, https://augur. net/whitepaper. pdf*, 2019.

[149] Lorenz Breidenbach, Christian Cachin, Benedict Chan, Alex Coventry, Steve Ellis, Ari Juels, Farinaz Koushanfar, Andrew Miller, Brendan Magauran, Daniel Moroz, et al. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. *Chainlink Labs*, 2021.

[150] John Adler, Ryan Berryhill, Andreas Veneris, Zissis Poulos, Neil Veira, and Anastasia Kastania. Astraea: A decentralized blockchain oracle. In *2018 IEEE international conference on internet of things (IThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData)*, pages 1145–1152. IEEE, 2018.

[151] Zackary Hess, Yanislav Malahov, and Jack Pettersson. Æternity blockchain. *Online]. Available: https://aeternity. com/aeternity-blockchainwhitepaper. pdf*, 2017.

[152] Sangyeon Woo, Jeho Song, and Sungyong Park. A distributed oracle using intel sgx for blockchain-based iot applications. *Sensors*, 20(9):2725, 2020.

[153] Hajar Moudoud, Soumaya Cherkaoui, and Lyes Khoukhi. Towards a scalable and trustworthy blockchain: Iot use case. In *ICC 2021-IEEE International Conference on Communications*, pages 1–6. IEEE, 2021.

[154] Yu Du, Jun Li, Long Shi, Zhe Wang, Taotao Wang, and Zhu Han. A novel oracle-aided industrial iot blockchain: Architecture, challenges, and potential solutions. *IEEE Network*, pages 1–8, 2022. doi: 10.1109/MNET.103.2100395.

[155] Shaimaa Bajoudah, Changyu Dong, and Paolo Missier. Toward a decentralized, trust-less marketplace for brokered iot data trading using blockchain. In *2019 IEEE International Conference on Blockchain*, pages 339–346, 2019. doi: 10.1109/Blockchain.2019.00053.

[156] W3C. Web of things. URL https://www.w3.org/WoT/documentation. (accessed: 14.12.2022).

[157] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[158] Giuseppe Antonio Pierro and Roberto Tonelli. Can solana be the solution to the blockchain scalability problem? In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1219–1226, 2022. doi: 10.1109/SANER53432.2022.00144.

[159] Desmo main repository. URL https://github.com/vaimee/desmo. (accessed: 13.12.2022).

[160] Zhe Xiao, Zengxiang Li, Yechao Yang, Piao Chen, Ryan Wen Liu, Wei Jing, Yauheni Pyrloh, Ekanut Sotthiwat, and Rick Siow Mong Goh. Blockchain and iot for insurance: A case study and cyberinfrastructure solution on fine-grained transportation insurance. *IEEE Transactions on Computational Social Systems*, 7(6):1409–1422, 2020.

[161] Markus Maibach, Christoph Schreyer, Daniel Sutter, HP Van Essen, BH Boon, Richard Smokers, Arno Schroten, Claus Doll, Barbara Pawlowska, and Monika Bak. Handbook on estimation of external costs in the transport sector. *Ce Delft*, 336, 2008.

[162] Muhammad Saad, Jeffrey Spaulding, Laurent Njilla, Charles Kamhoua, Sachin Shetty, DaeHun Nyang, and David Mohaisen. Exploring the attack surface of blockchain: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(3):1977–2008, 2020.

[163] Pietro D'Antuono, Wout Weijtjens, and Christof Devriendt. On the minimum required sampling frequency for reliable fatigue lifetime estimation in structural health monitoring. how much is enough? In *European Workshop on Structural Health Monitoring: EWSHM 2022-Volume 1*, pages 133–142. Springer, 2022.

[164] Juan Wang, Jinhua Yang, and Zhentao Zhang. Design of cloud computing platform based accurate measurement for structure monitoring using fiber bragg grating sensors. In *2021 IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, pages 807–811, 2021. doi: 10.1109/ICBAIE52039.2021.9389950.

[165] Cristian Martín, Daniel Garrido, Luis Llopis, Bartolomé Rubio, and Manuel Díaz. Facilitating the monitoring and management of structural health in civil infrastructures with an edge/fog/cloud architecture. *Computer Standards Interfaces*, 81:103600, 2022. ISSN 0920-5489. doi: https://doi.org/10.1016/j.csi.2021.103600.

[166] Zhiguo He, Wentao Li, Hadi Salehi, Hao Zhang, Haiyi Zhou, and Pengcheng Jiao. Integrated structural health monitoring in bridge engineering. *Automation in Construction*, 136:104168, 2022. ISSN 0926-5805. doi: https://doi.org/10.1016/j.autcon.2022.104168.

[167] Muhammad Fawad, Marek Salamak, Grzegorz Poprawa, Kalman Koris, Marcin Jasinski, Piotr Lazinski, Dawid Piotrowski, Muhammad Hasnain, and Michael Gerges. Automation of structural health monitoring (shm) system of a bridge using bimification approach and bim-based finite element model development. *Scientific Reports*, 13(1): 13215, Aug 2023. ISSN 2045-2322. doi: 10.1038/s41598-023-40355-7.

[168] Ambarish G. Mohapatra, Jaideep Talukdar, Tarini Ch. Mishra, Sameer Anand, Ajay Jaiswal, Ashish Khanna, and Deepak Gupta. Fiber bragg grating sensors driven structural health monitoring by using multimedia-enabled iot and big data technology. *Multimedia Tools and Applications*, 81(24):34573–34593, Oct 2022. ISSN 1573-7721. doi: 10.1007/s11042-021-11565-w.

[169] Shweta Sharma and Amandeep Kaur. Survey on wireless sensor network, its applications and issues. In *Journal of Physics: Conference Series*, volume 1969, page 012042. IOP Publishing, 2021.

[170] Nicola Testoni, Cristiano Aguzzi, Valentina Arditi, Federica Zonzini, Luca De Marchi, Alessandro Marzani, and Tullio Salmon Cinotti. A sensor network with embedded data processing and data-to-cloud capabilities for vibration-based real-time shm. *Journal of Sensors*, 2018, 2018.

[171] Federica Zonzini, Michelangelo Maria Malatesta, Denis Bogomolov, Nicola Testoni, Alessandro Marzani, and Luca De Marchi. Vibration-based shm with upscalable and low-cost sensor networks. *IEEE Transactions on Instrumentation and Measurement*, 69(10):7990–7998, 2020.

[172] Martine Wevers and Kasper Lambrighs. *Applications of Acoustic Emission for SHM: A Review*, pages 9–15. 09 2009. ISBN 9780470061626. doi: 10.1002/9780470061626.shm011.

[173] Federica Zonzini, Vasilis Dertimanis, Eleni Chatzi, and Luca De Marchi. System identification at the extreme edge for network load reduction in vibration-based monitoring. *IEEE Internet of Things Journal*, 9(20):20467–20478, 2022. doi: 10.1109/JIOT.2022.3176671.

[174] Élodie Morin, Mickael Maman, Roberto Guizzetti, and Andrzej Duda. Comparison of the device lifetime in wireless networks for the internet of things. *IEEE Access*, 5: 7097–7114, 2017. doi: 10.1109/ACCESS.2017.2688279.

[175] D Bogomolov, N Testoni, F Zonzini, M Malatesta, L de Marchi, and A Marzani. Acoustic emission structural monitoring through low-cost sensor nodes. In *Proceedings of the 10th International Conference on Structural Health Monitoring of Intelligent Infrastructure, Porto, Portugal*, volume 30, 2021.

[176] Claudia Bruno, Antonella Licciardello, Giuseppe Antonio Maria Nastasi, Fabio Passaniti, Carmen Brigante, Francesco Sudano, Alessandro Faulisi, and Enrico Alessi. Embedded artificial intelligence approach for gas recognition in smart agriculture applications using low cost mox gas sensors. In *2021 Smart Systems Integration (SSI)*, pages 1–5. IEEE, 2021.

[177] Kyriakos Georgiou, Samuel Xavier-de Souza, and Kerstin Eder. The iot energy challenge: A software perspective. *IEEE Embedded Systems Letters*, 10(3):53–56, 2017.

[178] Sean J Taylor and Benjamin Letham. Forecasting at scale. *The American Statistician*, 72(1):37–45, 2018.

[179] Mirco Tarozzi, Giacomo Pignagnoli, and Andrea Benedetti. Identification of damage-induced frequency decay on a large-scale model bridge. *Engineering Structures*, 221: 111039, 2020.

Chapter 8

Research Publications

This is a list of publications resulting from the research conducted during the pursuit of the PhD thesis.

Journal Articles

1. **Zyrianoff, I.**, Gigli, L., Montori, F., Sciullo, L., Kamienski, C., & Di Felice, M. (2024). Cache-it: A Distributed Architecture for Proactive Edge Caching in Heterogeneous IoT Scenarios. *Elsevier Ad Hoc Networks*.

2. Gigli, L., **Zyrianoff, I.**, Zonzoni, F., Bogomolov, D., Testoni, N., De Marchi, L., ... Di Felice, M. (2023). Next Generation Edge-Cloud Continuum Architecture for Structural Health Monitoring. *IEEE Transactions on Industrial Informatics*. doi: 10.1109/TII.2023.3337391.

3. Gigli, L., **Zyrianoff, I.**, Montori, F., Aguzzi, C., Roffia, L., & Di Felice, M. (2023). A decentralized oracle architecture for a blockchain-based IoT global market. *IEEE Communications Magazine*, 61(8), 86–92. doi:10.1109/MCOM.007.2200906

4. Montori, F., **Zyrianoff, I.**, Gigli, L., Calvio, A., Venanzi, R., Sindaco, S., ... Cinotti, T. S. (2023). An IoT toolchain architecture for planning, running and managing a complete condition monitoring scenario. *IEEE Access*, 11, 6837–6856. doi: 10.1109/ACCESS.2023.3237971

5. **Zyrianoff, I.**, Trotta, A., Sciullo, L., Montori, F., & Di Felice, M. (2022). IoT edge caching: Taxonomy, use cases and perspectives. *IEEE Internet of Things Magazine*, 5(3), 12–18. doi:10.1109/IOTM.001.2200112

6. Silva, D., Heideker, A., **Zyrianoff, I.** D., Kleinschmidt, J. H., Roffia, L., Soininen, J.-P., & Kamienski, C. A. (2022). A management architecture for IoT smart solutions: Design and implementation. *Journal of Network and Systems Management*, 30(2), 35. doi:10.1007/s10922-022-09648-6

Conference Proceedings

1. **Zyrianoff, I.**, Montecchiari, L., Trotta, A., Gigli, L., Kamienski, C., & Di Felice, M. (2024). Proactive Caching in the Edge-Cloud Continuum with Federated Learning.*CCNC 2024 proceedings*.

2. Aguzzi, C., Gigli, L., **Zyrianoff, I.**, & Roffia, L. (2024). Zion: A Scalable W3C Thing Description Directory. *CCNC 2024: 3rd International Workshop on IoT Interoperability and the Web of Things (IIWOT'24)*.

3. Brunelli, C., Pappacoda, G., **Zyrianoff, I.**, Bononi, L., & Felice, M. D. (2024). Water Wastage Detection in Smart Homes Through IoT and Machine Learning. *CCNC 2024 proceedings* as a work-in-progress.

4. Sciullo, L., Montori, F., **Zyrianoff, I.**, Gigli, L., Tinti, D., & Di Felice, M. (2023). Designing a hybrid push-pull architecture for mobile crowdsensing using the Web of Things. In *2023 IEEE International Conference on Smart Computing (SmartComp)* (pp. 332–337). doi:10.1109/SMARTCOMP58114.2023.00081

5. Kamienski, C., Cavalcanti, D., Batista, D., **Zyrianoff, I.**, & Virdis, A. (2022). The 1st International Workshop on the Internet of Time-Critical Things (IoTime 2022). In *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)* (pp. 1–2). doi:10.1109/WF-IoT54382.2022.10152076

6. Ottolini, D., **Zyrianoff, I.**, & Kamienski, C. (2022). Interoperability and scalability trade-offs in open IoT platforms. In *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)* (pp. 1–6). doi:10.1109/CCNC49033.2022.9700622

7. Puliafito, C., Gigli, L., **Zyrianoff, I.**, Montori, F., Virdis, A., Di Pascoli, S., ... Di Felice, M. (2022). Joint power control and structural health monitoring in Industry 4.0 scenarios using Eclipse Arrowhead and Web of Things. In *2022 IEEE 5th International Conference on Industrial Cyber-Physical Systems (ICPS)* (pp. 1–6). doi:10.1109/ICPS51978.2022.9816975

8. **Zyrianoff, I.**, Gigli, L., Montori, F., Aguzzi, C., Kaebisch, S., & Di Felice, M. (2022a). Artifact: C3PO - Converter of Open API Specification to WoT Objects. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events (PerCom Workshops)* (pp. 185–186). doi:10.1109/PerComWorkshops53856.2022.9767293

9. **Zyrianoff, I.**, Gigli, L., Montori, F., Aguzzi, C., Kaebisch, S., & Di Felice, M. (2022b). Seamless integration of RESTful Web services with the Web of Things. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events (PerCom Workshops)* (pp. 427–432). doi:10.1109/PerComWorkshops53856.2022.9767531

10. Montori, F., **Zyrianoff, I.**, Gigli, L., Venanzi, R., Sindaco, S., Aguzzi, C., ... Cinotti, T. S. (2021). A toolchain architecture for condition monitoring using the Eclipse Arrowhead framework. In *IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society* (pp. 1–6). doi:10.1109/IECON48115.2021.9589532

11. Prati, R. C., Borelli, F., **Zyrianoff, I.**, Silva, D., Togneri, R., & Kamienski, C. (2021). Irrigasim: An irrigation simulation, processing, and animation environment. In *2021 IEEE International Workshop on Metrology for Agriculture and Forestry (MetroAgriFor)* (pp. 305–309). doi:10.1109/MetroAgriFor52389.2021.9628455

12. Sciullo, L., **Zyrianoff, I.**, Trotta, A., & Felice, M. D. (2021). WoT Micro Servient: Bringing the W3C Web of Things to Resource Constrained Edge Devices. In *2021 IEEE International Conference on Smart Computing (SmartComp)* (pp. 161–168). doi:10.1109/SMARTCOMP52413.2021.00042

13. **Zyrianoff, I.**, Gigli, L., Montori, F., Kamienski, C., & Felice, M. D. (2021). Two-Way Integration of Service-Oriented Systems-of-Systems with the Web of Things. In *IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society* (pp. 1–6). doi:10.1109/IECON48115.2021.9589619

14. **Zyrianoff, I.**, Heideker, A., Sciullo, L., Kamienski, C., & Di Felice, M. (2021). Interoperability in Open IoT Platforms: WoT-Fiware Comparison and Integration. In *2021 IEEE International Conference on Smart Computing (SmartComp)* (pp. 169–174). doi:10.1109/SMARTCOMP52413.2021.00043

15. **Zyrianoff, I.**, Neto, A. T., Silva, D., Cinotti, T. S., Di Felice, M., & Kamienski, C. (2021). A Soil Moisture Calibration Service for IoT-Based Smart Irrigation. In *2021*

IEEE International Workshop on Metrology for Agriculture and Forestry (MetroAgriFor) (pp. 315–319). doi:10.1109/MetroAgriFor52389.2021.9628393

16. Augusto Sales Dantas, R., Vasconcelos da Gama Neto, M., **Zyrianoff, I.**, & Alberto Kamienski, C. (2020). The Swamp Farmer App for IoT-Based Smart Water Status Monitoring and Irrigation Control. In *2020 IEEE International Workshop on Metrology for Agriculture and Forestry (MetroAgriFor)* (pp. 109–113). doi:10.1109/MetroAgriFor50201.2020.9277588

17. Heideker, A., Ottolini, D., **Zyrianoff, I.**, Neto, A. T., Salmon Cinotti, T., & Kamienski, C. (2020). IoT-Based Measurement for Smart Agriculture. In *2020 IEEE International Workshop on Metrology for Agriculture and Forestry (MetroAgriFor)* (pp. 68–72). doi:10.1109/MetroAgriFor50201.2020.9277546

18. Queté, B., Heideker, A., **Zyrianoff, I.**, Ottolini, D., Kleinschmidt, J. H., Soininen, J.-P., & Kamienski, C. (2020). Understanding the Tradeoffs of LoRaWAN for IoT-Based Smart Irrigation. In *2020 IEEE International Workshop on Metrology for Agriculture and Forestry (MetroAgriFor)* (pp. 73–77). doi:10.1109/MetroAgriFor50201.2020.9277566