Distributed reasoning for the autonomous coordination of smart object networks

Politecnico di Bari

Department of Electrical and Information Engineering

Electrical and Information Engineering Ph.D. Program

SSD: IINF-05/A – Information Processing Systems

Final dissertation

Distributed Reasoning for the Autonomous Coordination of Smart Object Networks

by

Filippo Gramegna

Ph.D. program Coordinator:

Prof. Mario Carpentieri

Supervisor:

Prof. Michele Ruta

Course n. 37, 01/11/2021 – 31/10/2024

Politecnico di Bari

Department of Electrical and Information Engineering

Electrical and Information Engineering Ph.D. Program

SSD: IINF-05/A – Information Processing Systems

Final dissertation

Distributed Reasoning for the Autonomous Coordination of Smart Object Networks

by

Filippo Gramegna

Ph.D. program Coordinator:

Prof. Mario Carpentieri

Supervisor:

Prof. Michele Ruta

Referees:

Prof. Francesco Maria Donini

Prof. Hasan Ali Khattak

Course n. 37, 01/11/2021 – 31/10/2024

Abstract

The evolution of the *Internet of Things* (IoT) into the *Internet of Everything* (IoE) reflects the evolution of connected devices and computing technologies, encompassing not only things, but also environments, people, processes, and data. It enables large-scale data collection and analytics, with the potential to transform the interactions between many kinds of human activities and the physical world. Although this transition improves operational efficiency and data-driven decision-making, its full realization requires overcoming issues concerning network bandwidth, energy consumption, data security, and privacy. Most importantly, in the IoE interoperability and smart information management become essential for supporting flexible autonomous processing and sophisticated, flexible service-oriented architectures for extensive machine-to-machine and human-machine interactions.

A key strategy for addressing these challenges is *edge computing*, which brings computational tasks closer to data sources. This transformation is essential for managing the large volumes and rapid pace of data generated in the IoE, while also mitigating latency and bandwidth issues associated with centralized processing systems. An early example of a smart framework that leverages edge computing is the *Semantic Web of Things* (SWoT). Here, ontology-based descriptions of devices, objects, and events are dealt with locally by pervasive intelligent agents through automated reasoning, enabling autonomous operations towards specific objectives.

The advancement of SWoT towards a *Semantic Web of Everything* (SWoE) requires deeper embedding of semantic technologies in pervasive computing interactions. This vision requires pervasive knowledge representation and automated reasoning abilities, even on devices with stringent processing, memory, and energy limitations. Local inference mechanisms on devices are essential in the SWoE, considering the high volatility and restricted accessibility of more powerful devices.

The deployment of SWoE architectures discloses considerable difficulties from a scientific and technological standpoint. Current Semantic Web reasoners and Knowledge Base Management Systems (KBMS) are primarily tailored for high-performance computing environments such as servers and workstation clusters, making them unsuitable for nano-scale devices. Reasoning engines that might work on smaller devices frequently lack essential inference support, thus limiting their practicality. For this reason, creating SWoE platforms requires a reassessment of evaluation and benchmarking methodologies to consider the unique constraints of this new paradigm.

This dissertation presents several innovative contributions to the field of distributed reasoning in SWoE scenarios, focusing on applying knowledge representation and automated inferences to the coordination of networks of smart agents embodied into resource-constrained devices. To this aim, this work covers system architectures and optimization strategies for various essential components frameworks, such as: *Cowl*, a lightweight and versatile knowledge representation library designed for devices with limited resources, overcoming the restrictions of current KBMS in embedded and IoT contexts; *Tiny-ME*, an innovative multi-platform reasoner and matchmaking engine tailored for the SWoE, providing efficient reasoning capabilities appropriate for cloud, desktop, mobile, and edge devices; *evOWLuator*, a cross-platform evaluation framework that is mindful of energy consumption for Semantic Web reasoners, emphasizing power usage estimation and supporting inferences on remote devices; a *Cloud-Edge Intelligence* (CEI) framework for multi-agent systems and sensor-based application, exploiting serverless computing for data management and machine learning tasks.

Great emphasis is placed on the assessment of the developed technologies through extensive experimental campaigns, which provide insights into performance, efficiency, and applicability in SWoE settings. In addition, practical applications are demonstrated through case studies in various contexts. The first scenario demonstrates a framework for adapting Quality of Experience (QoE) in Web multimedia streaming, using the WebAssembly port of Tiny-ME as reasoning engine. The second highlights a privacy-focused local event finder, showing a client-side Web reasoning use case in data retrieval and personalization for Web applications. The third case study explores how Tiny-ME manages semantically annotated resources in peer-to-peer networks, improving negotiation and discovery explanations. Finally, a smart city example shows how Cowl can be integrated in nano-scale sensors to exchange semantically enriched data, enhancing urban mobility. Together, the mentioned experiments and applications underscore the flexibility and wide-ranging usability of the presented methods and technologies, highlighting the extensive potential of the SWoE.

Contents

Introduction

In a world that is becoming more connected, the complex network of the *Internet of Things* (IoT) is deeply embedded in industries, environments, and everyday life. This large-scale ecosystem, in which devices communicate and interact over the Internet, represents a significant milestone in technological progress. The IoT impacts various sectors, from home and building automation to industrial processes and environmental monitoring. It reflects the evolution of connected devices and computing technologies, enabling large-scale data collection and insightful analytics that are transforming the interactions of many kinds of human activities with the physical world.

The integration of low-cost devices with non-negligible computational power and small sizes into everyday objects now allows multiple devices to interact simultaneously, even if they differ from each other. These smart objects, which populate the environment, can collect data in real-time during daily activities. As a result, it is possible to detect events, classify ongoing interactions, and recognize contextual situations. Data collected by heterogeneous micro-devices scattered throughout a specific environment generate real-time streams from which valuable information can be extracted for decision support or automated control processes, creating new opportunities in various economic and social fields [58]. This growing pervasiveness is pushing the Internet of Things (IoT) to evolve into what is known as the Internet of Everything (IoE), a broader concept encompassing not just things, but also people, processes, and data. However, this shift amplifies some well-known IoT challenges, such as network bandwidth usage and power consumption, and introduces new issues, particularly concerning data protection and privacy.

The concept of *edge computing* [189], along with the exploitation of *cloud-edge* infrastructures, aims to address some of these problems by moving computation and storage closer to sources of data generation, specifically at the network's *edge*. By distributing intelligence and processing capabilities across the network, edge computing reduces reliance on centralized cloud infrastructures and allows faster response times, lower energy consumption, and enhanced data privacy. The *Semantic Web* [23] is progressively embracing this edge-centric approach, evolving into the *Semantic Web of Things* (SWoT) [181]. According to the SWoT vision, ontology-based tags are used to annotate devices, objects, and phenomena in a detailed, organized, and clear manner. These metadata enable intelligent agents to autonomously pursue their objectives by processing and deducing implicit information through automated reasoning.

The evolution of the IoT towards the IoE foreshadows a similar shift for the SWoT, paving the way for the *Semantic Web of Everything* (SWoE). In this expanded framework, semantic technologies enable seamless interactions between things, people, processes, and data at any scale, from the Web to nanodevices with limited computational resources. This vision requires the integration of general-purpose knowledge representation and automated reasoning techniques and methodologies on tiny devices, enabling autonomous decision-making and self-cooperation of multiple (homogeneous or heterogeneous) agents for the execution of a specific task or mission. Inference processes in the SWoE need to run locally, as access to more powerful auxiliary devices acting as semantic facilitators cannot always be guaranteed. Moreover, due to the high volatility of devices, resources, and data, rapid on-the-fly processing capabilities are crucial, as relying on external devices or handling the delays of wireless low-power network links for communication is often impractical. Semantic-based approaches enable structured and unambiguous information annotation capabilities, facilitating interoperability and high-level data fusion. Agents and protocols running on networked objects can be designed to automatically infer the required or available features in a completely decentralized and collaborative way.

Attempting to realize such a vision with current technology, if feasible, results in less-than-ideal outcomes for numerous reasons. Indeed, most existing tools for manipulating ontologies and knowledge bases in the Semantic Web are primarily designed for desktop and server environments. As will be discussed in Section 3.2, only a few systems are beginning to address the needs of the SWoE. However, these tools remain impractical due to hardware constraints or challenges with platform interoperability. Moreover, reasoning engines that could potentially be deployed on smaller devices often lack the support for non-standard inference services, which – as motivated in Section 2.3.2 – are crucially useful in the SWoT and, consequently, the SWoE. In addition, adapting existing systems or designing new ones tailored for the SWoE requires thorough evaluation methodologies, benchmarks, and software frameworks, as the use of computational resources becomes a key limiting factor. Deploying Semantic Web technologies on mobile and embedded devices also raises concerns about energy consumption, a factor that has traditionally been overlooked in desktop-oriented systems. As will be detailed in Section 3.3, current evaluation tools for Semantic Web technologies lack essential features feature for the SWoE, such as the ability to estimate power consumption and support for executing inference tasks on remote devices.

This dissertation presents several innovative contributions to the field of distributed reasoning in SWoE scenarios, focusing on applying knowledge representation and automated reasoning techniques to the auto coordination of networks of smart agents in resource-constrained environments. Agents and protocols running on networked objects can be designed to automatically infer the required or available features in a completely decentralized and collaborative way. This introduces additional complexities related to maintaining network reliability, consistency, and scalability. To address these issues, this dissertation introduces a series of tools and frameworks specifically designed to enable lightweight reasoning on devices with limited resources. In detail, the core academic contributions involve the conceptualization and practical execution of:

- a list of tools developed to achieve the SWoE vision, including: (i) *Cowl* [25, 24], a lightweight knowledge representation library specifically designed for resource-constrained device; *Tiny-ME* [164, 119], an innovative multi-platform reasoner and matchmaking engine for the SWoE; (iii) *evOWLuator* [178], a cross-platform, energy-aware evaluation tool for Semantic Web reasoners that can execute inference tasks on remote devices.

- a novel *Cloud-Edge Intelligence* (CEI) framework for multi-agent systems and sensor-based application [116], exploiting serverless computing for data management and Machine Learning (ML) tasks. Data gathering and preprocessing functions run on field sensors, while edge devices and cloud infrastructure nodes execute functions for information storage, model training and prediction in a flexible and opportunistic fashion.

The structure of the remainder of this dissertation is as follows: Chapter 2 presents the research context and provides a review of the technological background; Chapter 3 delineates the requirements for a knowledge representation stack suitable for the SWoE, covering the design and implementation of the Cowl library, detailing the theoretical foundation for effective reasoning on edge devices, and the architecture and optimizations of the Tiny-ME reasoner. Additionally, it introduces evOWLuator and demonstrates its effectiveness through a comparative evaluation involving various state-of-the-art Semantic Web reasoners; Chapter 4 explains the architecture and components of the proposed CEI framework for sensor-based applications, validating the approach with a complete prototypical implementation and an experimental evaluation and comparison with an existing microservice architecture on ML training and inference tasks; Chapter 5 details practical applications and case studies that support the proposal, concluding with a discussion on future perspectives.

Chapter 2

Distributed Artificial Intelligence for Edge Computing: state of the art

This chapter provides an insight into current trends in Edge Computing (EC) along with the implementation of knowledge representation and automated reasoning methods for the automatic coordination of smart object networks. Unlike traditional cloud solutions, this approach places processing and storage resources nearer to data sources, potentially lowering processing times and bandwidth consumption. In these scenarios, multiple homogeneous or heterogeneous smart objects in the environment can gather, process, and share information independently and collaboratively to modify their actions. Knowledge Representation and Reasoning languages and tools in the Semantic Web of Things can be utilized to offer automated reasoning abilities in situations that demand the cooperation of various agents. Their core technologies and visions are discussed to give a theoretical and technical foundation for the following chapters. For each specific topic, the limitations of the current state-of-the-art will be examined to highlight the issues tackled in the research work for this thesis.

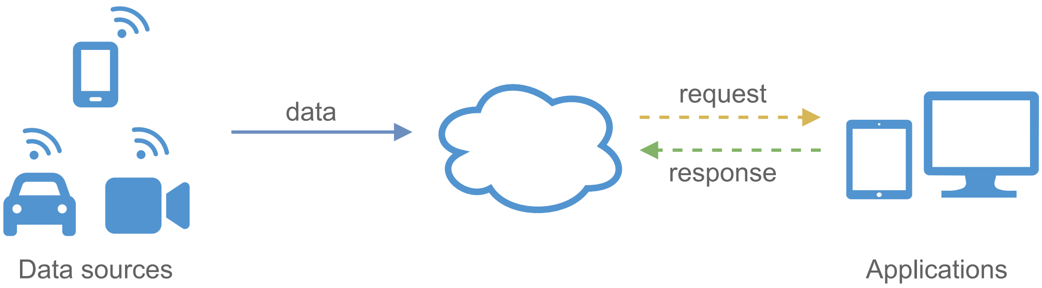


Figure 2.1: Cloud Computing paradigm.

2.1 Edge Computing

In current IoT landscapes, the edge of the network emerges as the primary site for data generation, demanding ever shorter response times. This shift toward edge-centric data processing carries significant benefits, notably enhancing performance efficiency and reinforcing security measures.

Historically, centralizing processing tasks in the cloud has proven effective for analyzing huge data volumes, leveraging its superior computational resources compared to common smart objects adopted at the edge of the network. However, the expanding rate of data production and the growing strain on existing network capacities are highlighting data transmission speed as the critical bottleneck for cloud solutions [189]. Figure 2.1 illustrates the typical architecture of conventional cloud computing solutions. Here, end devices act as *data sources*, generating raw data from field processes and sending them to the cloud platform. Meanwhile, *applications* interface with the cloud to request and utilize information on behalf of users.

This type of architecture appears unsuitable for IoT applications where (i) latency is mission-critical, (ii) the sheer volume of data generated leads to excessive bandwidth consumption at the network edge, and (iii) stringent data privacy requirements are in place. To address these limitations, alternative architectural paradigms such as micro-datacenters [1], cloudlets [111], and fog computing [11] have been introduced. These paradigms aim to enhance data processing efficiency for IoT device networks by bringing computational resources closer to the data source. Furthermore, in highly dynamic environments with unpredictable device and resource availability, *opportunistic computing* [48] has emerged as a paradigm designed to maximize the utilization of available resources for executing distributed computing tasks. The primary challenge in opportunistic computing lies in effectively managing volatile connections to ensure seamless data accessibility and the provision of collaborative computing services to both applications and users. Middleware services in this context must conceal disconnections and delays while managing diverse computing resources, services, and data to maintain a consistent and efficient operational environment.

Edge Computing (EC) is an emerging paradigm that places substantial computing and storage resources at the local network's edge, near field devices and sensors [176]. The term "edge" encompasses any computing and network resource along the path from the data source to the cloud data center. For instance, in the healthcare sector, a smartphone acts as the edge between wearable devices and a cloud service. Similarly, in a home automation network, a gateway serves as the edge between smart sensors and a cloud platform. The primary objective of the EC paradigm is to execute computational tasks closer to the data sources. A generic EC architecture, illustrated in Figure 2.2, features an intermediate layer of edge nodes positioned between end devices and the cloud. An edge node can (i) collect data from sensors and field devices, (ii) perform pre-processing activities and/or propagate data to the cloud, and (iii) request services and exchange contents with the cloud. In an IoT application design, exploiting edge nodes facilitates the offloading, storing, caching, and processing of data, as well as the distribution of the computational load within the local network.

2.1.1 Edge Intelligence

Edge Computing is recognized as a disruptive and rapidly evolving architectural paradigm in the contemporary technological landscape [94]. By strategically placing computing and communication resources at the net-

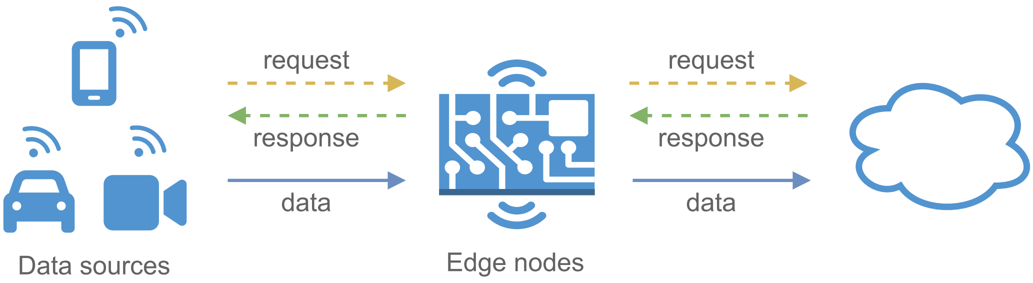


Figure 2.2: Edge Computing paradigm.

work's edge, where devices generate data, this approach significantly reduces communication latency and enhances responsiveness for end users. This is particularly beneficial in pervasive scenarios, from digital healthcare to smart manufacturing [2, 156]. Meanwhile, Artificial Intelligence (AI), through its various techniques, is profoundly transforming our daily lives. Machines are becoming increasingly proficient at tasks traditionally reserved for human intelligence, such as planning, learning, reasoning, problem-solving, knowledge representation, perception, navigation, object manipulation, language processing, social intelligence, and content generation. AI's ability to analyze vast volumes of data and extract higher-level insights is particularly valuable in Cyber-Physical Systems (CPS), leading to an emerging demand to integrate Edge Computing with AI. This integration has given rise to the Edge Intelligence (EI) paradigm [53, 128], that represents more than just the convergence of Edge Computing and AI. It encompasses a myriad of interconnected concepts and technologies. By bringing AI capabilities to the edge of the network, Edge Intelligence enhances the ability to perform real-time data analysis, enabling smarter, faster, and more efficient decision-making processes. This paradigm not only optimizes resource utilization but also facilitates the deployment of sophisticated AI applications in environments where low latency and high responsiveness are critical.

Researchers have explored EI from various perspectives. The work in [53] proposes a comprehensive vision by dividing EI into two distinct areas: "*AI for edge*" and "*AI on edge*".

- AI for edge focuses on enhancing Edge Computing by applying advanced AI technologies to solve constrained optimization problems more effectively. By integrating AI, the edge becomes more intelligent and efficient, leading to what can be termed Intelligence-enabled Edge Computing (IEC).

- AI on edge explores the deployment and execution of AI models directly at the edge. It allows the training and inference of AI models through a device-edge-cloud synergy. AI on edge seeks to derive insights from huge, distributed datasets while meeting requirements for algorithm performance, cost efficiency, privacy, reliability, and overall operational productivity.

Currently, there is no formal definition of Edge Intelligence, though [230] describes it as the paradigm that facilitates the local execution of AI algorithms on terminal devices using the data and signals they generate. While this definition aligns with the most common approach to EI in real-world applications, it significantly limits the paradigm's scope. Running computationally intensive algorithms on embedded devices necessitates substantial computing power, which not only increases the cost of an EI solution but is also often incompatible with the limited processing capabilities of current end devices.

In this context, authors in [232] assert that the EI should not be restricted to executing AI models solely on local devices or edge servers. They advocate for a paradigm where AI models are deployed with coordinated interaction between edge and cloud computing. This approach aims to minimize both end-to-end latency and power consumption compared to the conventional local execution approach. Such operational advantages underscore the importance of incorporating a collaborative hierarchy in designing effective EI solutions. Typically, AI models primarily execute the inference phase at the network edge, while the training phase is predominantly delegated to cloud datacenters leveraging substantial computational resources. This approach necessitates transferring extensive training datasets to the cloud, entailing high communication expenses and raising issues regarding data privacy. Employing edge-based data preprocessing and reduction techniques proves advantageous for efficiently managing massive datasets before their storage, transmission, and analysis. Nonetheless, challenges persist concerning the scalability and latency of these methods, which remain active areas of research [151].

In contrast, Edge Intelligence capitalizes on the combined data and computational resources spanning terminal devices, edge nodes, and cloud datacenters to optimize the overall training performance and effectively deploy trained models for inference. Thus, employing the EI paradigm does not strictly confine AI model training or inference exclusively to the edge of the network, but these operations can be orchestrated in a coordinated manner across cloud and edge environments via data offloading. Zhou *et al.* [232] categorize Edge Intelligence into six levels based on the volume and distance of data offloading, as illustrated and detailed in Table 2.1.

Table 2.1: Levels of EI as categorized by Zhou et al. [232].

Level	Description
0	*Cloud Intelligence.* Training the AI model and inferencing fully in the cloud.
1	*Cloud-Edge co-inference and Cloud training.* Training the AI model in the cloud, but inferencing through edge-cloud cooperation, which entails a partial data offload to the cloud.
2	*In-Edge co-inference and Cloud training.* Training the AI model in the Cloud, but inferencing at the network edge, by completely or partially offloading the data to the edge nodes or nearby devices.
3	*On-Device inference and Cloud training.* Training the AI model in the Cloud but inferencing in a fully local on-device manner. This implies that no data would need to be offloaded.
4	*Cloud-Edge co-training and inference.* Both training and inferencing occur in the edge-cloud cooperative way.
5	*All In-Edge.* Training and inferencing fully occur at the edge.
6	*All On-Device.* Both training and inferencing occur on local (field) devices.

As the level of Edge Intelligence increases, there is a corresponding reduction in both the volume and distance of data offloading. Consequently, this trend enhances data privacy while concurrently lowering transmission latency and bandwidth costs. However, this advancement is accompanied by heightened computational latency. Within the local network, there exists a variable trade-off between power consumption due to communication and computational processing. This analysis emphasizes the absence of a universally optimal level of EI. Instead, the determination of the most suitable level for each application requires a comprehensive evaluation of factors including latency, energy efficiency, privacy considerations, and bandwidth costs.

2.1.2 Open issues and limitations

The adoption of the Edge Computing architectural paradigm facilitates data processing and storage at the network's peripheral nodes, which are near the end user. In contrast to traditional cloud solutions with centralized servers, EC fosters a transition towards more distributed methods. This approach reduces processing delays by distributing the computational tasks, lowers energy consumption, and ensures development flexibility. Despite the inherent potential and benefits of this architecture model, it does present some limitations and ongoing research challenges, which are outlined below.

Naming. An ordinary EC infrastructure encompasses a vast number of nodes that need to be managed. Architecturally, multiple applications operate on top of the edge nodes, each with its unique distributed computing framework, delivering specific services to the end users. Similar to all computer systems, the *naming scheme* in EC is essential for programming, addressing, identifying objects, and transmitting data. Regrettably, a dependable and uniform naming system for this type of architectural paradigm has not yet been developed and standardized. An effective scheme must handle the movement of objects within a highly dynamic network topology while ensuring privacy and security.

Conventional identification mechanisms such as the Domain Name Service (DNS) and the Uniform Resource Identifier (URI) address the needs of most contemporary networks with high efficiency; however, they lack the flexibility to accommodate the dynamic nature of EC architectures. Emerging naming methods for EC include Named Data Networking (NDN) [229] and MobilityFirst [159]. Specifically, the NDN mechanism (i) offers a hierarchical, well-organized addressing system for dynamic networks based on content/data, (ii) simplifies the management/configuration of services, and (iii) provides excellent scalability for Edge solutions. However, existing NDN solutions are predominantly focused on Internet Protocol (IP)-based networks and do not provide mechanisms to configure additional proxies to integrate other IoT-oriented communication protocols such as Bluetooth [27] or ZigBee [61]. Security is another concern with NDN, as differentiating hardware information from service providers remains a significant challenge [46]. MobilityFirst can maintain the name linked to an edge device distinct from its network address through a dynamic mapping method. This enables primary edge services to be supported even when nodes exhibit significant mobility changes. However, the primary challenge of MobilityFirst is the complex and non-intuitive configuration process, making it hard to manage and deploy services.

Privacy and Security. In edge solutions, managing privacy and ensuring data security are essential requirements. A home or work environment equipped with numerous IoT devices can produce significant amounts of sensitive information that can be deduced through data mining and analysis. In these situations, architectures focused on security and privacy, along with the support for monitoring and management services, pose significant challenges. Moving computational tasks from cloud infrastructures to the network edge can effectively enhance the protection of data privacy and security. Preprocessing data –particularly when it is sensitive– at the network edge to aggregate information can diminish the data streams sent to the cloud via the Internet, thus reducing the risks of eavesdropping and tampering [224]. Nonetheless, several issues remain unresolved:

- The community is still not fully aware that privacy and security are crucial in such architectures [8]. Service providers, system integrators, application developers, and end users need to recognize the potential for unexpected privacy breaches, as exemplified by vulnerabilities in IP cameras, health monitoring devices, and smart home automation systems.

- Data ownership at the edge can be compromised. Like mobile applications, smart objects in pervasive contexts collect data that is stored, transmitted, and analyzed by a service provider. For enhanced security, it is advisable to retain data at or near the collection point. Data collected at the edge should remain on the edge node and only be transferred to a service provider for processing after user or data protection officer evaluation [109], with sensitive data filtered out during this authorization process.

- Effective tools for handling privacy and security at the data collection stages are scarce, primarily due to the significant resource limitations of most end devices, which cannot handle usual computation-heavy security measures. Implementing IoT security solutions on an edge node is advantageous because: (i) it has greater resources compared to IoT end devices; (ii) it can meet the real-time requirements necessary for security designs; (iii) the vast amount of data collected from different sources improves the accuracy of security algorithms [185]. Additionally, the mobility of nodes further complicates the task of securing the network.

Optimization metrics. The design of an infrastructure based on the EC paradigm must consistently consider and optimize the following performance metrics.

Latency. Latency is a key metric for evaluating the effectiveness of a computing architecture, especially for real-time applications in pervasive computing. Cloud computing infrastructures boast extensive computing resources and can manage large and complex tasks, such as image processing or speech recognition, in a notably short duration. However, overall latency also depends on end-to-end communication delays in a Wide Area Network

(WAN), which can greatly affect the performance of real-time interactive applications. To minimize response times, computations should occur as close as possible to the devices collecting data and requests. For example, in a manufacturing plant, an edge node can be utilized to locally process product images for automated quality inspection, sending only faulty component data to the cloud. However, the physical layer closest to the data is not always the best solution due to limited computational capabilities, especially when executing concurrent tasks. An optimal balance can be achieved by appropriately distributing the available computing resources and the maximum acceptable waiting time for an application [9].

Bandwidth. Besides latency, bandwidth plays a crucial role in transmission times, particularly for applications that transfer significant amounts of data. Handling the workload at the network edge can significantly enhance latency in comparison to processing in the cloud. This also reduces the necessary bandwidth and enhances transmission reliability. Additionally, if an edge node lacks the computational power to meet a certain computing request, various data pre-processing techniques (e.g., data integration, data normalization, data transformation, data reduction, feature selection, instance selection) [65, 151] can be employed to reduce the volume of data that needs to be transferred to the cloud.

Energy. Energy is the most crucial resource for battery-operated wireless nodes in IoT-based pervasive computing. An endpoint can save energy by offloading its workload to edge devices [129]. However, to determine whether it is more efficient to (partially) offload the workload to the edge or handle it locally, one must consider the trade-off between the energy used for computation and that used for information transmission. Generally, to objectively estimate the energy impact of a specific task, one must consider the required computing resources, network signal strength, data size, and available bandwidth [88]. Utilizing edge computing is advisable only if the overhead of data transmission is lower than the cost of local processing. However, when considering the entire network architecture instead of just the endpoints involved, the total energy consumption is the sum of the energy costs of each device from data collection to processing. Per-device energy costs can be estimated as the sum of local computation and transmission costs [228]. In this scenario, the optimal strategy for load distribution may vary, depending on whether a local node is engaged in other tasks or not. In the first case, the computational task progressively shifts to edge (or cloud) resources, and multi-hop data transmission might increase the load on network nodes, leading to higher power consumption.

Cost. Lower cost is one of the advantages that motivate companies to adopt EC technologies and architectures. In 2021, *Gartner* forecasted that by 2022, over 50% of enterprise-generated data would be created and processed outside a conventional data center or cloud [52]. Although expensive cloud datacenters have been replaced with on-demand pay-per-use IT resources, the operating expenses for companies using this architectural model remain high. In an EC framework, the expenses mainly involve infrastructure such as sensors, computing systems, and network [211]. Specifically, utilizing the local area network for data processing offers enterprises greater bandwidth and storage at a lower cost compared to cloud computing. Moreover, since processing occurs at the edge, less data needs to be sent to the cloud or data center for additional processing. Consequently, both the amount of data that must be transferred and the overall solution cost are reduced.

AI integration. The rapid development of innovative AI and ML techniques has led the scientific community to combine AI with Edge Computing, forming the Edge Intelligence (EI) paradigm [128]. EI presents multiple advantages:

- pre-processing raw data at the network edge reduces the dimensionality of data that needs to be sent to the Cloud for in-depth analysis, thus lowering bandwidth usage;
- heterogeneous sensor data fusion algorithms at the network edge decrease the computational load on cloud platforms;
- utilizing the computing capabilities of edge nodes enhances the responsiveness of distributed AI algorithms in Big Data environments.

However, the EI framework adds complexity to edge computing in significant ways [221]: (i) the integration of structured approaches and paradigms to integrate ML within a cloud-edge continuum; (ii) the requirement to modify AI/ML algorithms for distributed computing and storage systems; (iii) the management of computing and memory resources needed for model training and data prediction, as well as for training and prediction services themselves.

Several unresolved issues restrict the broader use of EC solutions: the heterogeneity of devices, services, and information that emerge in pervasive contexts; the accuracy of the gathered information; the dependability of decisions independently suggested by pervasive devices.

This thesis presents a pioneering general-purpose Cloud-Edge Intelligence (CEI) distributed framework (refer to Chapter 4) to address the aforementioned challenges. It utilizes an Osmotic Computing paradigm-based microservice architecture that facilitates opportunistic resource utilization by dynamically distributing service modules across devices at the network edge and/or in the cloud. The machine learning model's training and prediction tasks can be executed on edge or cloud nodes, or via cloud-edge cooperation, which is a crucial aspect of the proposed approach.

2.2 Smart objects

The concept of the *Internet of Things* describes a system that is loosely coupled, decentralized, and dynamic, where a vast number of everyday objects are interconnected worldwide, equipped with intelligence, and actively participate in business, logistics, information, and social processes [231]. These *things* are generally known as smart objects (SOs), and when supported by an "anywhere, anytime, and anything connection" [12], they form the essential elements of the IoT [103], and consequently, the foundational CPS paradigms. Smart objects are capable of offering extensive cyber-physical capabilities and services to both humans and machines due to their abilities in communication, sensing, actuation, embedded processing, and even reasoning.

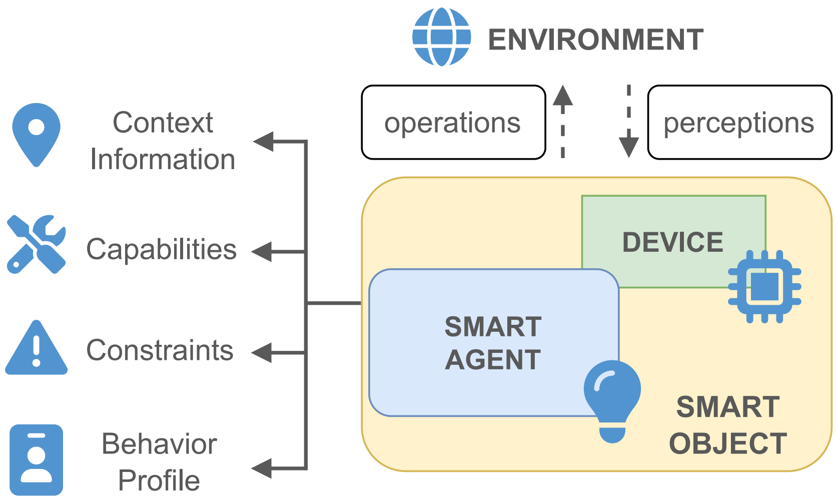


Figure 2.3: Smart object descriptions.

According to literature, a *smart object* [212] refers to an intelligent agent operating on a device integrated with sensors, actuators, communication interfaces, and typically limited computational, storage, and power capabilities. These smart objects can collect data streams concerning both internal and external factors, adapt to their surroundings, and/or take actions to alter the environment.

There are primarily four types of information handled, as illustrated in Figure 2.3:

- *Context Information*: an outline of the environment in which the object functions, based on data gathered from the integrated sensors;
- *Capabilities*: the sensing and actuation functionalities of the smart object, referred to as perception and operation capabilities;
- *Constraints*: boundaries and characteristics imposed by itself or other entities, shaping the preferred state of the environment. These constraints guide the smart object's behavior to ensure adaptation in accordance with them;
- *Adaptation/Behavior Profiles*: context-sensitive reactive patterns within the smart object, defined using a rule-based structure consisting of two

segments:

- *Preconditions*: current requirements regarding the state of the environment that must be satisfied for the profile to be triggered;
- *Postconditions*: intended patterns in the future state of the environment that need to be met by the agent to adhere to the profile. Smart objects must determine how to fulfill the postcondition criteria, based on their internal logic and built-in actuators.

This research builds on existing definitions and introduces them in an innovative manner, as further clarified in Chapter 5. Specifically, information is articulated in a semantically rich and structured format based on the *Attributive Language with unqualified Number restrictions* ($\mathcal{ALN}$) Description Logic, which is a subset of the *Web Ontology Language* (OWL) 2 [146] standard used for creating ontologies and annotating resources in the Semantic Web. Section 2.3 provides background information on knowledge representation languages and technologies. In a typical SWoE scenario, each smart object incorporates a reasoning engine that carries out automated inference processes to uncover implicit knowledge from environmental semantic-based data. This enables the object to dynamically identify necessary tasks for adjusting its configuration or interacting with the environment, and share potentially useful information with nearby objects in a collaborative way.

2.2.1 Current issues and limitations

Essentially, the operations of Context-Aware Systems (CAS) are organized according to a so-called *context information life cycle*, which determines where information is generated and where it is utilized [4]. As Figure 2.4 illustrates, it includes four stages:

- *Context acquisition*: perception components (*e.g.*, embedded sensors, network interfaces) gather low-level data from the cyber-physical environment; this data may be preprocessed and/or stored to make it

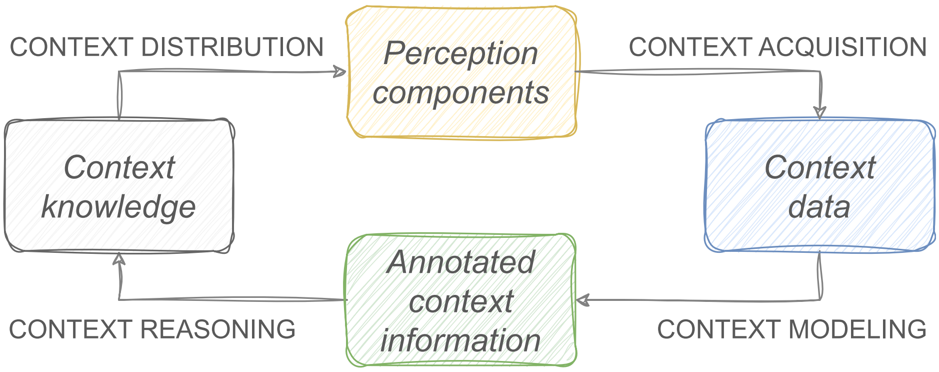


Figure 2.4: Context information life cycle [4].

accessible for further processing.

- *Context modeling*: this stage involves multi-sensor fusion and semantic-based annotation of the gathered data w.r.t. reference domain ontologies, to create a formal model of the observed context.

- *Context reasoning*: at this stage, inference procedures are applied to the available annotations to derive implicit knowledge, with the goal of achieving a deeper understanding of the context and supporting decision-making.

- *Context distribution*: during this stage, context knowledge is disseminated; there are various dissemination models, primarily *pull*-based (*e.g.*, *request/response*) and *push*-based (*e.g.*, *publish/subscribe*).

This thesis explores a research area focused on embedding intelligence in micro-devices situated in pervasive environments, interconnected wirelessly within unstructured smart object networks. According to the CAS framework, cognitive *smart objects* should be capable of: (i) autonomously extracting and processing environmental data; (ii) creating semantically rich, compact descriptions about themselves and their operational context; (iii) achieving steady-state with coherent situational awareness by integrating high-level information they detect and receive into a micro-*log*; (iv) making the acquired knowledge easily accessible to other network nodes by publishing it to a micro-*blog*. In pervasive environments, objects operate as independent *social* *agents*, engaging autonomously and synchronizing in response to environmental updates. These environments include various mobile micro-devices, where the environmental knowledge of each agent, obtained through its own or accessible perception tools, can differ significantly. In addition, unpredictable conditions influencing sensor-derived data, such as unreliable sensor technology or frequent wireless communication disruptions, require agents to manage the resulting incomplete or inconsistent information. Despite having an inherently limited view, agents aim to construct a more accurate and comprehensive environmental context *snapshot* by incorporating perspectives from other agents. Enabling such *fusion* capability in a smart object requires the ability to integrate ontology-based context annotations from both self-generated and received data, managing inconsistencies, and mitigating erroneous information. By doing so, a smart object can dynamically determine the task(s) necessary to adjust its own setup or the environment's state and automatically deduce what essential functions it can offer or needs from other entities to respond appropriately, in a decentralized and cooperative manner. The historical and inferred context is shared over a mobile ad-hoc network –and on the Web, if Internet access is available– to promote intelligent collaboration.

In literature, there is a fundamental distinction between High-Level Information Fusion (HLIF) and Low-Level Information Fusion (LLIF) [29]. Although both types focus on error correction and information enhancement, they operate at distinctly different levels and use varying tools and techniques. Specifically, the aforementioned *Context modeling* step utilizes LLIF, whereas *Context reasoning* is directly associated with HLIF. LLIF focuses on numerical data derived from physical phenomena and is commonly associated with multi-sensor fusion used in identification, tracking, and classification tasks. In this context, *aggregation functions* [40, 101, 43] facilitate the fusion of individual device measurements and help minimize the size of data being exchanged. Likewise, [43] describes a cooperative surveillance Multi-Agent System (MAS) that forms coalitions of agents with overlapping fields of view. Upon receiving data transformed into a common reference system, a local fusion agent can monitor moving objects and detect inconsistencies within the coalition. *Distributed particle filtering* [80], a highly popular and flexible technique for estimating environmental states, is particularly well-suited for large-scale, nonlinear systems. The subfamily of *Aggregation Chain* methods shares two key characteristics with the approach presented here: (i) only estimated probability distributions are exchanged, not the raw measurements, and (ii) probability estimations are iteratively merged in a chain. However, the purely statistical nature of the method distinguishes it from semantic-based approaches, as the managed data lack high-level formal meaning. HLIF, on the other hand, handles abstract symbolic information and is typically used for refining knowledge and maintaining situational awareness. As categorized by the knowledge fusion patterns in [193], it can be seen as *"fusing knowledge of knowledge workers and knowledge accumulated in repositories"*, as the semantic annotations provided by independent smart objects are consolidated and aligned to resolve detected conflicts, using a core domain knowledge encoded in a reference ontology. Consequently, the fusion of sensor data falls outside the scope of this thesis, as it assumes that smart agents are equipped with on-board sensing and data mining capabilities, allowing LLIF to generate logic-based annotated descriptions of their observations, as seen in [168, 174].

Semantic-based information fusion leverages operators and inference mechanisms on formulas expressed in logical language. Epistemic logic was employed for peer-to-peer (P2P) information integration in [39], although it mainly focused on schema mappings for queries within a heterogeneous P2P system as a unified entity, rather than on distributed intelligence. The belief merging framework discussed in [44] utilizes the concept of *weak disagreement* –*i.e.*, information that is neither affirmed nor refuted by different agents– to enhance knowledge toward a more accurate overall truth approximation. However, [44] emphasized the majority consensus among multiple peers. In contrast, the decentralized iterative knowledge fusion strategy presented in [169] aims to eliminate explicit disagreements rather than to achieve majority agreement, treating all arguments equally. Consequently, the approach in [44] aligns with logical *arbitration* [112], in contrast to *belief revision* methods [66], which give more importance to recent arguments. Relevant applications of semantic-based information fusion encompass ambient intelligence [141], home and building automation [170], wireless sensor networks [49], and vehicular networks [169]. In these contexts, alternative popular methods rely on Bayesian [160] or Dempster-Shafer (also known as belief function) theory. Unfortunately, integrating these approaches with semantically explicit frameworks for context modeling is not straightforward. A significant effort is described in [68] to improve situation awareness in vehicular networks by utilizing Multi-Entity Bayesian Networks, which integrate semantic relationships with entities in a Bayesian network to enhance probabilistic event assessment.

A review of data management and analytics in the IoT can be found in [5]. Recently, the trend of centralizing cloud-based computation has been shifting toward more decentralized models like *Fog Computing* [50] and *Edge Computing* [189], which bring storage and processing closer to data sources. For example, the approach proposed in Chapter 4 aligns with these architectures, offering benefits such as reduced processing latency, enhanced scalability, and improved privacy, especially in scenarios involving extensive applications. In such contexts, *semantic data aggregation* is frequently associated with approximate techniques or lossy compression to consolidate data from various sources in the IoT and wireless sensor network domains. However, some studies employ semantic-based knowledge representation frameworks to enhance not only query processing but also data aggregation [175, 10].

An intelligent HLIF method, exploiting a knowledge-fusion strategy, has been introduced in [169]. The developed *Concept Integration* (CI) inference service is intended to consolidate the viewpoints of various independent entities into a unified context observation. The individual knowledge is derived from a local CI process that handles detected and received semantic annotations regarding the current environment state. It is organized as a 4-tuple of annotations, each offering a distinct perspective on the context: (i) elements recognized in both local and other agents' observations; (ii) local observation conflicting with other observations; (iii) elements identified in the local observation, but not by other agents; (iv) elements noticed by other agents, but not locally. This method is appropriate for MAS such as vehicular networks, characterized mainly by uniform mobile agents with fairly abundant computational power. The potential advantages of employing semantics and automated reasoning for the self-regulation of intelligent agent teams were also highlighted in [171], where a knowledge-driven framework was suggested for the autonomous control of mobile robots in Search And Rescue (SAR) operations. Semantic-based methods facilitate interoperability among various devices, enabling high-level situation description, evaluation, and inference. However, decentralized and dynamic IoT structures are challenged by the unpredictability of nodes due to the instability of actors and devices. Furthermore, scalability presents a significant challenge when trust and coordination among smart objects are required.

To address these challenges, Cowl is presented in Section 3.1 of this dissertation. It is an OWL manipulation library that aims to simplify knowledge management. This library is designed to be versatile for general-purpose architecture and well-suited for large-scale IoT environments or devices with limited resources. By integrating axiom stream serialization and parsing capabilities, this tool is particularly useful in situations where devices need to generate, coordinate, and transfer semantically annotated data, such as in interconnected multi-agent systems.

2.3 Knowledge Representation and Reasoning

In ubiquitous and pervasive environments, smart software agents operating on personal devices can gather, process, and share meaningful information fragments to facilitate adaptive, context-aware behaviors across a wide range of applications. As these agents engage with each other and the surrounding environment in a typical perception-decision-action-evaluation cycle, they must seamlessly integrate the information they detect and receive, and collaboratively manage any agreements or disagreements on their perceptions. In addition, a high degree of autonomic capability is required for these agents to initiate actions or interventions in the environment based on the detected context, all without human involvement. Techniques and technologies in Knowledge Representation and Reasoning facilitate information modeling through formal and precise semantic interpretation. This not only improves interoperability across various hardware and software systems but also supports reasoning tasks, such as making inferences to unveil hidden insights from explicitly stated data in a Knowledge Base (KB). Together, *Knowledge Representation and Reasoning* (KRR) techniques are crucial for transforming the WWW into a smarter, more responsive environment. They enable a novel level of interaction where machines can efficiently process, analyze, and even predict user needs based on the rich semantic structure of annotated web resources.

Of the numerous KRR languages and tools available, those emerging from the Semantic Web initiative have been widely accepted due to their well-known algorithmic characteristics and highly efficient implementations. The following subsections provide a detailed overview of the Description Logics (DLs) family and the reasoning tasks pertinent to this research.

2.3.1 Description Logics

Description Logics (DLs) [14] are a family of Knowledge Representation (KR) languages situated within a decidable subset of First Order Logic (FOL) [33, 60]. DLs facilitate the representation of knowledge through:

- *concepts*, also called *classes*, representing collections of objects;
- *roles*, also called *properties*, which outline relationships between pairs of concepts;
- *individuals*, *i.e.*, the named examples of classes.

These components can be unified through *constructors* to form DL expressions, with their formal semantics defined by an *interpretation* $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ that links each term to a subset of the discourse universe (the *domain* Δ). Concept *conjunction* is interpreted as set intersection: $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$. Concept *disjunction* is interpreted as set union: $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$. The operator $\neg$, when used, represents *complement.* There are additional constructs that define each language in the DL family, influencing both the expressiveness and the computational complexity of inference tasks. An *ontology* (a.k.a. terminology, terminological box, or TBox) [195] comprises two types of *assertions* related to classes and properties: *inclusion*, which enables the definition of *is-a* relationships between classes ($A \sqsubseteq D$ where A and D are concept expressions); and *equivalence*, which assigns a name to a specific concept expression ($A \equiv D$). The *assertion box* (ABox) consists of individuals. A Knowledge Base (KB) consists of a pair ⟨TBox, ABox⟩.

DLs are characterized by the constructors they offer. This dissertation uses the *Attributive Language with unqualified Number restrictions* ($\mathcal{ALN}$) DL as the benchmark for expressiveness. It achieves sufficient expressiveness while maintaining polynomial complexity for both standard and non-standard inferences [56]. The constructs of $\mathcal{ALN}$ are presented below and Table 2.2 provides a summary of the syntax and semantics of constructors and assertions in $\mathcal{ALN}$.

- $\top$, *universal concept.* Every object within the domain.
- $\bot$, *bottom concept.* The empty set.
- A, *atomic concepts.* All objects belonging to the set A.
- $\neg A$, *atomic negation.* All objects not belonging to the set A.
- $C \sqcap D$, *intersection.* Objects that are part of both C and D.
- $\forall R.C$, *universal restriction.* The objects involved in the relation R whose range are all the objects included in set C.
- $\exists R$, *unqualified existential restriction.* There exists at least one object participating in the relation R.
- $(\geq nR)$[1], $(\leq nR)$, $(= nR)$[2], *unqualified number restrictions.* Respectively, the minimum, the maximum and the exact number of objects participating in the relation R.

[1]Notice that $\exists R$ is equivalent to $(\geq 1R)$.
[2]Notice that $(= nR)$ is a shortcut for $(\geq nR) \sqcap (\leq nR)$.

Table 2.2: Syntax and semantics of $\mathcal{ALN}$.

Name	Syntax	Semantics
Top	$\top$	$\Delta^{\mathcal{I}}$
Bottom	$\bot$	$\emptyset$
Intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Atomic negation	$\neg A$	$\Delta^{\mathcal{I}} \backslash A^{\mathcal{I}}$
Universal quantification	$\forall R.C$	$\{d_1 \mid \forall d_2 : (d_1, d_2) \in R^{\mathcal{I}} \to \ d_2 \in C^{\mathcal{I}}\}$
Number restrictions	$\geq nR$	$\{d_1 \mid \sharp\{d_2 \mid (d_1, d_2) \in R^{\mathcal{I}}\} \geq n\}$
	$\leq nR$	$\{d_1 \mid \sharp\{d_2 \mid (d_1, d_2) \in R^{\mathcal{I}}\} \leq n\}$
Inclusion	$A \sqsubseteq D$	$A^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
Equivalence	$A \equiv D$	$A^{\mathcal{I}} = D^{\mathcal{I}}$

2.3.1.1 Web Ontology Language (OWL)

The Web Ontology Language (OWL) [146] is a recommendation by the World Wide Web Consortium (W3C) that extends the RDF Schema to create ontologies and represent metadata about resources in the Semantic Web. OWL employs Internationalized Resource Identifiers (IRI) for unique resource naming and is built upon the Resource Description Framework (RDF) [78] knowledge model. The initial version of OWL was divided into three levels of complexity, ranging from the least to the most expressive:

- *OWL Lite*: allows the definition of simple taxonomies and basic constraints;
- *OWL DL*: provides substantial expressiveness while maintaining computational feasibility. The name indicates a direct equivalence between OWL and DLs;
- *OWL Full*: offers maximum flexibility and expressiveness at the cost of computational decidability.

OWL Lite is contained within *OWL DL*, which itself is a subset of *OWL Full*. The elements of *OWL DL* that can represent the $\mathcal{ALN}$ DL are shown in

Table 2.3 using *RDF/XML* (RDF embedded in *eXtensible Markup Language*) syntax [177].

Table 2.3: Mapping between OWL RDF/XML and DL syntax.

OWL RDF/XML syntax	**DL syntax**
<owl:Thing>	$\top$
<owl:Nothing>	$\bot$
<owl:Class rdf:ID="C">	C
<owl:ObjectProperty rdf:ID="R">	R
<rdfs:subClassOf>	$\sqsubseteq$
<owl:equivalentClass>	$\equiv$
<owl:disjointWith>	$\neg$
<owl:intersectionOf>	$\sqcap$
<owl:allValuesFrom>	$\forall$
<owl:someValuesFrom>	$\exists$
<owl:maxCardinality>	$\leq$
<owl:minCardinality>	$\geq$
<owl:cardinality>	$=$

In dynamic scenarios characterized by changing nodes that interact opportunistically to achieve a common goal, *OWL Full* is unsuitable due to its complexity, while appropriate fragments of *OWL DL* offer a good balance between expressiveness and complexity. This dissertation models the domain of interest using the latest OWL language specification published by the W3C in 2012, namely OWL 2 [146]. This specification enhances language expressiveness, expands datatype support, and annotation capabilities, and offers a variety of syntaxes for reading, storing, and exchanging knowledge conceptualizations among applications. Ontologies can be serialized in any triple-based RDF syntax, such as *RDF/XML* [177] or *Turtle* [155]. Additional OWL-specific syntaxes, like *Functional* [145] and *OWL/XML* [133], directly encode OWL axioms without converting them into RDF triples. Lastly, the *Manchester syntax* [82] is designed for better readability by users without a logic background. Tools are available to convert between different syntaxes, providing flexibility and adaptability for various use cases and user preferences.

OWL 2 contains three sublanguages known as *profiles*. These are simplified versions of the language designed to address particular industrial needs, like improved reasoning scalability and efficient query processing.

- *OWL 2 EL*, a segment that enables reasoning within polynomial time complexity. It is specifically tailored for ontologies with numerous properties or classes. It is particularly useful for domains such as life sciences, due to its ability to manage large hierarchies.
- *OWL 2 QL*, intended to simplify the access and querying of data stored within databases. It supports seamless integration with relational databases, making it perfect for use cases where accessing data through ontologies is essential.
- *OWL 2 RL*, a subset of rules in OWL 2, aimed at applications that need scalable reasoning while maintaining a significant degree of expressiveness.

Each profile addresses particular practical use cases and presents a unique balance between expressiveness and reasoning efficiency. Unlike earlier OWL sublanguages, OWL 2 profiles are independent of each other. Moreover, knowledge-based systems do not need to fully adhere to any specific OWL profile: the supported OWL constructs can be selectively restricted to match DLs that ensure favorable computational complexity and practical performance characteristics, while still being sufficiently expressive for beneficial applications.

2.3.2 Reasoning services

Algorithms for $\mathcal{ALN}$ concept descriptions that exhibit polynomial-complexity *structural* reasoning can be exploited to facilitate distributed information aggregation and discovery within resource-constrained agent networks and MASs

[182]. For this purpose, when an $\mathcal{ALN}$ KB is loaded, it undergoes preprocessing through *unfolding* and *Conjunctive Normal Form* (CNF) *normalization* [163] steps. Specifically, given a TBox $\mathcal{T}$ and a concept C, unfolding recursively incorporates references to axioms from $\mathcal{T}$ within the concept expression itself. As a result, $\mathcal{T}$ becomes unnecessary for subsequent inferences. To achieve finite unfoldings and maintain polynomial time and space complexity for inference procedures, $\mathcal{ALN}$ TBoxes must adhere to the following constraints [163]:

- the left-hand side (LHS) of inclusion ($\sqsubseteq$) and definition ($\equiv$) axioms must be atomic, *i.e.*, *general concept inclusions* are not allowed;
- if an atomic concept A is on the LHS of a definition axiom, it must not appear on the LHS of any other inclusion or definition axiom;
- TBoxes must be *acyclic*, meaning that the concept on the LHS of inclusion and definition axioms must not appear in the unfolding of the right-hand side (RHS) of the same axiom. This condition can be somewhat mitigated by allowing *told subsumption cycles* [206].

The CNF translation is achieved by performing a series of predefined substitutions. Any concept expression C in CNF can be represented as:

$$C \equiv C_{CN} \sqcap C_{\leq} \sqcap C_{\geq} \sqcap C_{\forall}$$

with

- C_{CN}: conjunction of (possibly negated) concept names;
- $C_{\leq}$: conjunction of maximum cardinality constraints, at most one per role;
- $C_{\geq}$: conjunction of minimum cardinality constraints, at most one per role;
- $C_{\forall}$: conjunction of universal restrictions, at most one per role; fillers are recursively in CNF.

The normalization process maintains semantic equivalence with respect to the models derived from the TBox; moreover, CNF is unique (considering the commutativity of the conjunction operator) [56]. The normalized form of an unsatisfiable concept is simply represented as $\perp$.

Considering a DL ontology $\mathcal{T}$, where S and R are two satisfiable concepts within $\mathcal{T}$, the standard inference services of *satisfiability* and *subsumption* offered by DL-based systems [14] can be expressed as:

- *Satisfiability*: verifies if S can have instances w.r.t. the ontology $\mathcal{T}$, *i.e.*, $\mathcal{T} \not\models S \sqsubseteq \perp$. A class that is unsatisfiable contains a contradiction, which means that it cannot have instances.
- *Subsumption*: checks if S is more specific than R w.r.t. the ontology $\mathcal{T}$, *i.e.*, $\mathcal{T} \models S \sqsubseteq R$. In this case, all instances of S are also instances of R.

After $\mathcal{ALN}$ concept expressions have been unfolded and normalized, subsumption and satisfiability can be determined by examining the structure of the expressions. This involves comparing them as if they were sets of primitive atoms that represent the $\mathcal{ALN}$ constructs shown in Table 2.2. The algorithmic execution of these services, including illustrative examples of their application on a sample ontology, is detailed in Section 3.2.2. Typically, knowledge-based applications require more advanced and complex inference services over ontologies, such as *ontology coherence*, *consistency*, and *classification*:

- **Ontology Coherence** entails verifying that every named concept within the TBox is satisfiable [130]. An ontology that contains an unsatisfiable class is considered incoherent, although it can still provide valuable inferences. Therefore, incoherent ontologies are commonly used in various applications.
- **Ontology Consistency** assesses whether the axioms in the ontology can be interpreted such that at least one class has an instance. If the ontology lacks consistency, every class is interpreted as the empty set, leading to no valuable conclusions. This is typically seen as a critical

error in ontology modeling, and most reasoners will halt inferences when facing this situation.

- **Ontology Classification** determines the complete hierarchy of concepts based on the subsumption relation, ranging from $\top$ to $\bot$. In essence, classification is logically the same as determining all subsumption relations between every pair of concepts within the TBox. Consequently, it is a significantly complex reasoning task that demands meticulous optimization to execute efficiently in terms of time and space, especially for extensive ontologies.

In practical application scenarios that involve detailed and frequently conflicting descriptions, conventional inference services such as *Subsumption* and *Satisfiability* fall short because they only offer a Boolean result. More sophisticated settings that handle different information from multiple independent sources require an explanation of the outcome. Non-monotonic reasoning tasks initially designed for belief revision are necessary. The non-standard inferences mentioned below are particularly important for the aims of this work:

- *Concept Contraction*: if $\mathcal{T} \models S \sqcap R \sqsubseteq \bot$, *i.e.*, S and R are not compatible with each other, Concept Contraction (CC) is able to identify a pair of concepts $\langle G, K\rangle$ such that $\mathcal{T} \models R \equiv G \sqcap K$, and $K \sqcap S$ is satisfiable in $\mathcal{T}$. Here, K is referred to as a *contraction* of R according to S and $\mathcal{T}$. G (for *Give up*) represents "why" R is incompatible with S *i.e.*, which element of R is in conflict with S and must be retracted to obtain an expression K (for *Keep*) such that $K \sqcap S$ is satisfiable in $\mathcal{T}$. Hence, Concept Contraction Problem (CCP) extends the notion of (in)satisfiability and provides a method to move from a partial match to a *potential match* situation.

- *Concept Abduction*: if $\mathcal{T} \models S \sqcap R \not\sqsubseteq \bot$ and $\mathcal{T} \models S \not\sqsubseteq R$, *i.e.*, S and R are compatible but S does not subsume R, then Concept Abduction (CA) identifies a concept H (for *Hypothesis*) such that $\mathcal{T} \models S \sqcap H \sqsubseteq R$ and $S \sqcap H$ is satisfiable in $\mathcal{T}$. Basically, H captures the aspects of R that are not covered by S. In particular, solving a Concept Abduction Problem (CAP) offers an extension and rational for (missed) subsumption and provides a path from a potential to a *full match* (a.k.a. *subsume match* [108, 143]), which is the desired outcome of the matchmaking framework and occurs when $S \sqsubseteq R$, *i.e.*, all features in the request are fulfilled by the resource.[3]

- *Concept Bonus* (CB): derives a concept B from S, which denotes what S offers but is not stated in R. In formal terms, identifying the Bonus B of S with regard to R is equivalent to find H in a Concept Abduction problem where R and S are swapped.

- *Concept Difference* (CD): defines a method to subtract information from one concept description using another: according to Teege's original definition [202], if $\mathcal{T} \models R \sqsubseteq S$, then the resulting difference $D = CD(R, S)$ is a concept $D \in \mathcal{L}$ such that $R \equiv S \sqcap D$.

- *Concept Covering* (CCov): can be exploited in matchmaking and negotiation scenarios to compose a set of elementary expressions to answer complex requests [183]. Basically, CCov involves a set of resources and a request with the goal to (i) cover (*i.e.*, satisfy) the features mentioned in the request as much as possible through the conjunction of resources, and (ii) provide an explanation for the portion that may remain uncovered. In formal temrs, given a concept expression R (request) and a set of concept expressions $\mathcal{S} = \{S_1, S_2, ..., S_n\}$ in a KB (available resources), where R and S_1, S_2, ..., S_n are satisfiable in the reference ontology $\mathcal{T}$, the output of CCov is a pair $\langle \mathcal{S}_c, H \rangle$ where $\mathcal{S}_c \subseteq \mathcal{S}$ contains concepts in $\mathcal{S}$ forming a (possibly incomplete) covering of R w.r.t. $\mathcal{T}$ and concept H represents the (potential) part of R not addressed by the elements in $\mathcal{S}_c$.

The $\mathcal{ALN}$ CNF describing concept expressions establishes a metric space definition featuring a *norm* operator $\|\cdot\|$. For example, the CNF norm of

[3] *Exact match*, occurring when $S \equiv R$, is the ideal outcome, but full match is equally desirable from the point of view of requesters, since all their preferences are met.

G and H denotes a semantic distance **penalty** for CC and CA, respectively, which is used to rank resources in relation to a specific request. Likewise, $\|B\|$ serves as a relevance indicator for the Bonus. According to [163], proposed penalty values include:

1. every atomic concept, whether negated or not, counts as 1;
2. for each role, number restrictions are weighted as the relative difference between cardinality values in R and S w.r.t. the cardinality value in R (if R has no number restriction on that role, the penalty is 1);
3. for each value restriction, the penalty is computed recursively as above.

For CA, CB, and CC, minimality criteria are established, as one generally aims to hypothesize or give up the least amount possible. These criteria generate numerical distance (penalty) functions derived from the CNF norm of expressions H and G. On the other hand, a maximality criterion is applied for CD because, in this scenario, one usually aims to subtract as much as possible.

These penalties can be combined using a *utility function* (a.k.a. a *score combination function*) to assess the quality of match approximation. This approach is analyzed and utilized in *semantic matchmaking* issues [47]. Semantic matchmaking essentially involves identifying the most suitable matches to a *request* (denoted as R in the previous definitions) from the available *resource* descriptions (denoted as S earlier), with both the request and the resources semantically annotated in relation to a shared reference ontology [47]. In these situations, the outcome is an ordered list of relevant resources in relation to the request. It is crucial to note that all inference services used in semantic matchmaking are based on the following principles:

- *Open World Assumption.* The lack of information in a concept expression does not imply negation; rather, it denotes an unspecified constraint, possibly due to being unknown or considered irrelevant.

- *Non-symmetric evaluation.* A semantic matchmaking system can produce different evaluations for the match between two concept expressions S and R, depending on whether it aims to match S with R or R with S.

Table 2.4: Set of possible match classes.

Match class	Description	Semantics
Exact	S ia semantically equivalent to R. All the requirements specified in R are in S and S does not expose any additional feature w.r.t. R.	$\mathcal{T} \models R \equiv S$
Full	S is more specific than R. All the requirements expressed in R are provided by S and S includes extra features that are both not required by R and not in conflict with the ones in R.	$\mathcal{T} \models S \sqsubseteq R$
Plug-in	R is more specific than S. All the characteristics expressed in S are requested by R and R exposes also other requirements both not exposed by S and not in conflict with characteristics in S.	$\mathcal{T} \models R \sqsubseteq S$
Potential	R is compatible with S. Nothing in R is logically in conflict with anything in S and vice-versa.	$\mathcal{T} \not\models S \sqcap R \sqsubseteq \bot$
Partial	R is not compatible with S. There is at least one requirement in R that contradicts some feature in S.	$\mathcal{T} \models S \sqcap R \sqsubseteq \bot$

As a result of the matchmaking process, five different match classes (categories) [55, 108], summarized in Table 2.4, have been identified. Although the most desirable match is clearly the exact match, a requester would find a full match equally satisfactory. However, potential and partial matches are most frequently encountered in complex real-world scenarios. Through CC and CA, one can advance from a partial match to a full match by leveraging a query refinement process.

2.3.3 State of the art

The classical Semantic Web is characterized by the steady and extensive availability of considerable computational and networking resources. In contrast, SWoT scenarios face severe hardware limitations, with information sources spread across tiny devices in physical environments, leading to various challenges. These challenges include highly unpredictable connectivity, unexpected disconnections, location dependency, and the constraints of wireless communication links and energy supply, making pervasive scenarios distinct from traditional wired and reliable computing contexts. All such issues are exacerbated in the SWoE. To support user activities and offer general-purpose innovative services, mobile agents with rapid decision support, query answering, and stream reasoning capabilities are necessary [63, 110]. In such contexts, reasoning engines are used as decisional and organizational systems, where specific non-standard inference services might be more appropriate than standard ones [183]. Furthermore, since mobile and embedded devices typically have limited resources, they need inference engines optimized for efficient performance on low-resource platforms, whereas conventional reasoners usually demand considerable hardware and software requirements, particularly regarding main memory.

Essentially, there are several types of automated reasoning algorithms, which include:

- *Structural algorithms* focus on the arrangement and form of logical expressions, often transforming them into simpler or standardized forms. These algorithms are especially useful in scenarios where the logical structure is complex or significantly affects the reasoning process. Structural algorithms are highly efficient and typically operate within polynomial time. However, they are incomplete for highly expressive DLs, *i.e.*, they cannot identify every existing relationship.

- *Tableaux-based algorithms* operate by building a *tableau*, which is a tree-based structure that represents various interpretations of the ontology. By systematically expanding this tableau according to specific rules and checking for contradictions, these algorithms can determine the consistency of an ontology or the satisfiability of a concept expression. While they can handle more expressive DLs, they incur considerably higher computational costs, with time and space complexity often being exponential or even double exponential.

- *Rule-based algorithms* apply predefined rule sets, typically in the *if-then* form, on a collection of asserted facts to infer new facts. In *forward chaining*, rules are repeatedly applied to the expanding set of facts until a specific fact is derived or no further facts can be inferred. Conversely, *backward chaining* begins with the query and works backward to determine the necessary facts to satisfy it. Rule-based algorithms are generally efficient, operating within polynomial time relative to the input size, but they tend to be inflexible and do not support *non-monotonic* reasoning, meaning they cannot revoke conclusions if new conflicting information is introduced.

Due to the architectural limitations and computational complexity of DLs reasoning, most of the early mobile inference engines only supported rule processing for entailment materialization in knowledge bases. Examples of these engines are *3APL-M* [100], *COROR* [199], *MiRE4OWL* [96], *Delta-Reasoner* [132], and the system described in [184], which were inadequate for applications requiring non-standard inference tasks and extensive ontology reasoning. *Pocket KRHyper* [190] was the first reasoning engine, based on hyper tableau calculus, specifically designed for mobile devices [98]. However, frequent "out of memory" errors significantly restricted the size and complexity of logic expressions it could handle. To address these limitations, memory consumption was reduced through tableaux optimizations as introduced in [197] and applied in *mTableaux*, an enhanced version of the Java Standard Edition (SE) *Pellet* reasoner [191]. Despite performance improvements, Java

SE technology was not designed for mobile platforms, because they depend on Java class libraries that are incompatible with most widespread mobile operating systems (*e.g.*, Android). In [225], four Semantic Web reasoners were successfully migrated to the Android platform (Pellet, *CB* [91], *Hermit* [67], and *JFact*, a Java version of *Fact++* [206]), though significant rewriting or restructuring was required in some instances. Similarly, in [92], the *ELK* reasoner was optimized and tested on Android. However, all ported systems were mainly intended for batch processing of large ontologies and/or expressive languages, which makes their usage on mobile devices less ideal due to computational and memory limitations.

Moreover, the aforementioned reasoners provide only basic inference services such as satisfiability and subsumption, which are not enough for pervasive scenarios. Non-monotonic reasoning tasks are essential to develop software agents capable of delivering rapid decision support and dynamic organization in inherently unpredictable environments. *Mini-ME* [182] was an Android-oriented *matchmaker* and reasoner, compatible with Java SE. It supports both standard and non-standard semantic matchmaking inferences. The rule engine for Android in [208] introduced an innovative *RETEpool* algorithm for OWL 2 RL rulesets, effectively balancing memory consumption and performance. *Mini-ME Swift* [167] was the pioneering OWL reasoner for iOS, redesigned based on the earlier Mini-ME, using the OWL API for iOS [166]. *Tiny-ME* (Tiny Matchmaking Engine) [164] is a matchmaking and reasoning engine designed and built with a compact and portable C core. It offers a wider array of standard and non-standard inference services compared to its predecessor Mini-ME in a moderately expressive OWL 2 fragment aligned with the $\mathcal{ALN}$ DL. Its multiplatform framework, combined with various Applications Programming Interfaces (APIs), ensures native compatibility with Windows, Linux, macOS, Docker containers, Android, iOS, and embedded systems like Apache NuttX. This work will delve into the Tiny-ME in greater detail in Section 3.2.

2.3.4 Open issues and limitations

By combining the Semantic Web vision of an interpretable and functional Web with the widespread networking capability of the IoT, the SWoT has achieved notable progress in semantics-enabled digital ecosystems. However, as the SWoT evolves towards the more extensive and inclusive SWoE, it encounters a new array of challenges. This transition is not just a matter of increasing scale or scope, but entails a fundamental change in how KRR technologies are deployed and utilized. The main challenge resides in the necessity to implement these technologies at an even finer level of detail, progressing from mobile devices to microcontrollers and nano-scale devices that constitute the core of the IoE. This progress is fraught with challenges, as the limitations of these smaller devices concerning computational power, memory, and energy are considerably more restrictive. The difficult task of embedding semantic functionalities into such constrained environments, while maintaining conformity with traditional Semantic Web and SWoT frameworks, calls for innovative solutions and a re-evaluation of current paradigms. Furthermore, the intrinsic instability of devices, the rapid speed of data streams, the variety of platforms, and the essential requirement for cross-platform compatibility further complicate this transition. Consequently, the transition from the SWoT to the SWoE uncovers numerous unresolved issues, calling for a dedicated effort to create robust, efficient, and scalable solutions capable of supporting new machine-to-machine and human-to-machine interaction models, applications, and services to fully achieve the envisioned potential.

Volatility. One of the main issues in the IoE is the high volatility of devices and network connections. Devices in typical IoE setups are frequently mobile, resulting in constantly changing network topologies and sporadic Internet access. In this context, depending on powerful external devices for semantic support is impractical, so local availability of KB manipulation and automated inference tools is necessary. Furthermore, relying on battery power for many of these devices poses limitations on their operational lifespan and performance. Solving these problems calls for the creation of lightweight, energy-efficient KRR technologies that can adjust to the highly dynamic nature of SWoE settings.

Velocity. Speed of the generated data presents a major challenge for conventional Semantic Web systems, particularly in real-time IoT processing scenarios. As noted earlier, these systems typically concentrate on extensive and expressive KBs, along with complex and resource-intensive boolean inferences, which often do not align with applications dealing with fast-paced data streams. Instead, SWoE systems should prioritize rapid queries on compact and moderately expressive KBs, potentially through more adaptable non-monotonic inferences, to address quickly changing conditions with prompt insights and actions.

Platform heterogeneity. The SWoE exhibits significant platform variety due to the wide assortment of hardware setups and software ecosystems within the IoE domain, where devices vary from robust servers to the most restricted microcontrollers, each with distinct capabilities, operating systems, and communication protocols. Although numerous Java-based Semantic Web reasoners have been adapted for Android [30], there is a notable scarcity for iOS devices [167] and embedded real-time operating systems. A functional SWoE infrastructure requires the development of new cross-platform inference frameworks, services, and tools that can operate efficiently on high-performance workstations and mobile devices while also being light enough to function on nano-devices with minimal computational power.

Resources constraints. The resource limitations at the lower end of the SWoE device spectrum impose significant restrictions on deploying traditional Semantic Web technologies. Most current Semantic Web reasoners and KBMS are created for use on the WWW and are expected to operate on plugged-in servers or at least on sufficiently capable mobile devices like tablets and smartphones. Transferring existing tools is often impractical due to substantial hardware and software platform disparities, and even when it is feasible, it requires major effort and does not always yield performance benefits. Specifically, energy consumption has rarely been prioritized in reasoner design for conventional computing systems. There is an urgent need for developing lightweight, energy-efficient semantic technologies specifically crafted for the SWoE

In conclusion, to truly determine if new or modified solutions are suitable for the SWoE, robust evaluation frameworks and methodologies are essential. As discussed in Section 3.3, the current OWL reasoner evaluation tools and frameworks lack comprehensive support for both standard and non-standard reasoning services, broad platform compatibility, energy consumption measurement, and the capability to perform inference tasks on remote devices, all within a single package.

Chapter 3

A distributed knowledge representation and reasoning infrastructure

This chapter introduces the tools developed during the Ph.D. program to achieve the SWoE vision:

- **Cowl** [25, 24]: a compact knowledge representation library, tailored specifically for devices with limited resources (Section 3.1).
- **Tiny-ME** [164, 119]: a cutting-edge, multi-platform reasoning and matchmaking engine for the SWoE (Section 3.2).
- **evOWLuator** [178]: an energy-efficient, cross-platform evaluation tool for Semantic Web reasoners that can execute inference tasks on remote devices (Section 3.3).

Each tool will be described in detail, including its architectural design, implementation concepts, and a comprehensive experimental analysis.

3.1 Cowl: OWL at (and beyond) the Edge

This section presents *Cowl* [25, 24], a cutting-edge OWL manipulation library designed for managing ontologies across a broad range of platforms, from desktop computers and smartphones to embedded systems with limited processing power and storage capacity. Its source code[1] is freely accessible under a highly permissive license[2], aiming to promote usage in both academic and commercial settings. The library is capable of parsing, querying, editing, and serializing OWL ontologies, with a focus on processing efficiency and minimal memory usage through specific design choices, features, and optimizations. The toolkit also introduces the *axiom streams* method, an innovative approach to ontology processing at the axiom level that avoids the use of an intermediate data store, thus optimizing resource usage for environments with stringent resource limitations. An extensive evaluation on a popular embedded platform confirms its practicality on the most constrained SWoE devices, marking the first successful deployment and assessment of an all-encompassing OWL library on a microcontroller with less than 100 KiB of RAM.

The rest of the paragraph is organized as follows: Section 3.1.1 covers related work and tools, while Section 3.1.2 and Section 3.1.3 explain the capabilities and architecture of the Cowl library, respectively; Section 3.1.3.1 introduces the innovative axiom stream method for parsing and serializing ontologies; Section 3.1.3.2 highlights peculiarities and optimizations that make it adaptable for any device category; finally, experiments and their results are detailed in Section 3.1.4.

3.1.1 Background on OWL toolkits

The software tools available for manipulating OWL usually depend on high-level abstractions, allowing applications to handle ontologies without delving into the specifics of the serialization formats. Historically, the development of OWL software has been shaped by the close relationship between OWL and RDF. Consequently, the initial OWL toolkits utilized a *triple-based* abstraction, viewing ontologies as collections of RDF triples, and provided

[1]Cowl source code: `https://github.com/sisinflab-swot/cowl`

[2]Eclipse Public License v2.0: `https://www.eclipse.org/legal/epl-2.0/`

APIs to assist users in managing them according to the OWL specification.

A notable tool is *Apache Jena* [127], a Java framework that offers manipulation primitives for RDF, *RDF Schema* (RDFS) [37], and OWL models, as well as an inference Application Programming Interface (API) to assist reasoning and rule engines. Another remarkable example is the *Protégé-OWL Plugin* [99] for the *Protégé* ontology editor [134], which is based on Jena and excels in the development of graphical applications. The *owlcpp* [107] open-source library in C++ enables the parsing of OWL ontologies in RDF/XML format into an RDF triple store built upon the *Redland* [20] application framework, with additional use of the *Fact++* [206] engine for reasoning purposes.

Triple-based abstraction tools typically utilize RDF statements and triple stores as their foundational data architectures, focusing on RDF-based OWL serializations. This approach faces several issues: (i) it often requires varying degrees of understanding regarding how OWL ontologies map to RDF graphs, making them more technical and difficult to use; (ii) they encode OWL axioms as one or more RDF triples, complicating updates and queries and generally adding memory overhead; (iii) RDF-based serializations require the inclusion of parsers and renderers for RDF syntaxes, and sometimes XML, thereby increasing code size and memory demands.

The *OWL API* [81], a widely used Java toolkit for OWL, introduced an innovative approach to OWL abstractions by viewing ontologies as sets of axioms and making APIs more consistent with the OWL specification. This *axiom-based* abstraction hides all serialization details, enabling the adoption of different memory storage options and reducing the learning curve for users. The *OWL API for iOS* [166], a partial port of the OWL API to Objective-C for Apple devices, can be integrated with the *Mini-ME Swift* reasoner [167] to enable semantically enhanced applications and services on iOS. A similar approach is employed by *Horned-OWL* [118], an early-stage Rust library for OWL 2 manipulation aimed at handling large-scale ontologies, providing full support for OWL-XML and nearly full support for OWL-RDF parsing.

An additional abstraction, known as the *ontology-oriented programming* technique, embeds ontologies directly in a runtime setting, treating ontology entities as objects in programming languages. This approach capitalizes on the similarity between ontology frameworks and object-oriented programming (OOP) models to streamline the integration of ontologies and reasoning in software development by employing a familiar syntax. However, it often demands considerable introspection support at the language level. *Owlready* [104], a pioneer of this concept, is a widely-used toolkit for managing OWL ontologies and their constructs as Python objects, and also for reasoning over them through modified versions of the *HermiT* [67] and *Pellet* [191] engines. Another similar method is used by *OWLOOP* [38], an extension of the *OWL-API* designed to reduce the coding required for ontology manipulation by allowing interactions with OWL entities as objects within the OOP framework, although it currently supports only a subset of OWL 2 axioms.

Table 3.1: Overview and comparison of OWL toolkits.

Toolkit	**Language**	**Updated**	**Memory usage**
Apache Jena [127]	Java	2023	High [22]
Protegé-OWL [99]	Java	2020	N.A.
owlcpp [107]	C++	2016	Medium [25, 24]
OWL API [81]	Java	2023	High [25, 24]
horned-owl [118]	Rust	2023	N.A.
OWL API for iOS [166]	Objective-C	2020	Medium [166]
Owlready2 [104]	Python	2023	High [25, 24]
OWLOOP [38]	Java	2022	N.A.
Cowl	C	2023	Low [25, 24]

Although numerous OWL manipulation tools have been introduced in this brief literature review, many of them present substantial challenges that limit their suitability for resource-constrained environments, as described in Table 3.1, such as:

- reliance on Java and/or Python runtime environments, which are unsuitable for embedded devices[3];

- designs optimized for traditional computing environments, assuming abundant processing power and memory resources;
- absence of detailed performance evaluations, particularly regarding memory usage and focusing on resource-limited devices;
- projects that are not actively maintained.

A preliminary version of the Cowl library was presented in [25], where preliminary evaluations showed a strong fit with the needs of the Semantic Web of Everything. Later, essential features have been incorporated, including ontology editing and serialization support, as well as the handling of ontologies as continuous axiom streams. Furthermore, the tool has been extensively evaluated on an embedded platform.

3.1.2 Features

Cowl fully implements the *OWL 2 Structural Specification* [145]. It supports OWL 2 ontologies without any constraints on the configuration of axioms and expressions. In general, the library offers the following functionalities: (i) parsing ontology documents; (ii) executing programmatic queries on ontologies; (iii) creating and editing ontologies; (iv) serializing and writing ontology documents.

Ontology documents can be processed into an efficient in-memory store, enabling knowledge extraction through queries. The repository allows for updates by adding and deleting axioms and supports serialization to preserve changes permanently. Ontologies can also be managed as *axiom streams*, an

[3]The presence of Python interpreters for embedded systems, such as *MicroPython* [21], does not fully address this issue, as they only support a subset of the standard Python runtime environment libraries, requiring at least partial rewriting of existing OWL manipulation tools.

innovative method (cfr. Section 3.1.3.1) that processes documents axiom-by-axiom without requiring an intermediate store, making it highly suitable for platforms with limited memory. Cowl currently handles ontology documents using the *functional syntax* [145]. However, its architecture is designed to accommodate additional readers and writers in future updates. The functional syntax is preferred as the reference format due to several useful features:

- straightforward, clear syntax that can be interpreted with minimal resources;
- no need for base serialization formats like RDF or XML, as these would require specific readers and writers;
- ensure that the byte encoding of one axiom does not mix with the encoding of another. As described below, this allows for notable memory efficiency when handling OWL knowledge graphs as streams of axioms.

3.1.3 Design principles

The primary design decision involves selecting a suitable programming language that is both a systems language to enable library deployment on embedded devices and flexible enough to allow easy integration with other higher-level languages and environments, supporting mobile devices, desktops, and cloud servers. The C language is a logical choice for its widespread use in embedded systems [79] and its high interoperability [28]. Consequently, the library has been developed in standard C11, avoiding any compiler-dependent extensions or platform-specific API calls, to enable straightforward deployment on any platform with a C compiler and integration into any programming environment supporting C interoperability. The option to use the *ontology-oriented* programming abstraction has been dismissed because the C language lacks the necessary object-oriented features and introspection capabilities. Similarly, the *triple-based* abstraction has been set aside because it would require additional software components and computational resources. The design of the Cowl library is outlined in Figure 3.1. It employs the *axiom-based*

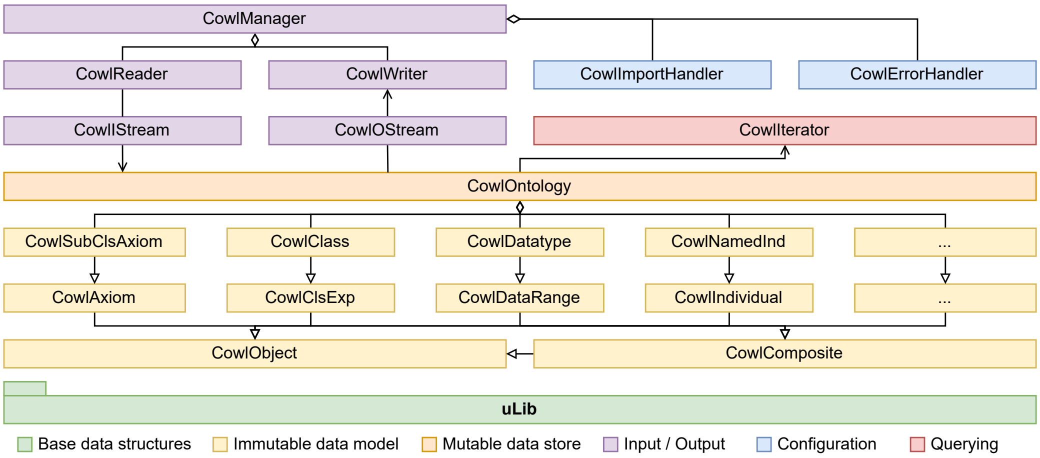


Figure 3.1: Structure of the Cowl library.

abstraction, and its data model and APIs are crafted to closely align with the OWL 2 Structural Specification [145] W3C Recommendation. Reproducing the specification in C presents a design challenge: OWL construct relationships are often hierarchical and cannot be straightforwardly expressed with C types due to the language's purely procedural nature, lacking support for object-oriented concepts like inheritance. In Cowl, hierarchical relationships are emulated using the *pseudo-inheritance* technique, which leverages specific aspects of the C language to mimic inheritance at the structure level: the C11 specification ([86], 6.7.2.1-15) ensures that there is no padding at the start of a *struct*, guaranteeing its memory address matches that of its first member. By embedding *structs* in such a way that the base type (the "superstruct") is the first member of the derived type (the "substruct"), pointer casts between the two types, generally forbidden by the C standard, become allowable.

According to the OWL 2 specification [145], OWL constructs are categorized into families. The library represents these as base types and subsequently extends them to concrete types using the pseudo-inheritance method:

- **Axioms** (*CowlAxiom*): statements that define the truths within the domain described by the ontology;
- **Class expressions** (*CowlClsExp*): sets of individuals defined by for-

mally specified constraints on their properties;

- **Data ranges** (*CowlDataRange*): similar to class expressions, but related to sets of *literals*, *i.e.*, data values including strings or numbers;

- **Individuals** (*CowlIndividual*): specific instances within the knowledge domain, which can be either *named* or *anonymous*;

- **Object property expressions** (*CowlObjPropExp*): relationships between pairs of individuals;

- **Data property expressions** (*CowlDataPropExp*): relationships between individuals and literals.

The data model has been meticulously designed to ensure that OWL constructs are *immutable*, *i.e.*,, they remain unaltered once created. This not only simplifies software design, but also allows various performance optimizations crucial for devices with limited resources, as discussed in Section 3.1.3.2. The data model is based on *uLib*[4], an open-source library offering foundational data structures and utilities aimed at computational efficiency and low memory usage.

Cowl employs the *reference counting* method for memory management: each construct in the data model has a reference count attribute that is incremented whenever a new reference is made, and decremented when a reference is no longer required, *e.g.*, when a construct that was referencing it is destroyed. The principle is that a construct can be safely freed when its reference count hits zero, as this indicates that no references are pointing to it, hence it is no longer needed. Pseudo-inheritance simplifies the implementation of reference counting by enabling all constructs to have a common ancestor, *CowlObject*, which provides the necessary state and primitives.

In Cowl, ontologies are represented using the *CowlOntology* object, which fundamentally serves as a collection of axioms. Internally, a *CowlOntology* operates as an optimized, self-organizing in-memory database, indexing axioms

[4]uLib source code: `https://github.com/ivanobilenchi/ulib`

by their types and referenced entities to enhance query efficiency. The *CowlOntology* API allows for the addition and removal of axioms, annotations, and other elements, enabling users to modify existing ontologies or to create and populate new ones. Ontologies can be accessed via specific endpoints in the *CowlOntology* API, which take *CowlIterator* instances as parameters. A *CowlIterator* is a C-based implementation of the *closure* pattern [105], a concept in functional programming where a function and its environment are packaged together, allowing the function to use local variables outside its direct scope. In C, this is achieved by manually passing the environment through a pointer to a general *context* object. The *CowlIterator* function is used for each element found in the query, and the inclusion of a user-defined state enables the creation of complex queries. Iteration can be halted by returning `false`, useful for stopping early when a specific condition is met. The object *CowlManager* enables the reading and writing of ontologies, managing various input sources and output targets. This entire I/O system is based on uLib streams, which provide unified access to files, memory buffers, network streams, and more, as byte streams that can be read and written. Cowl's design supports multiple readers and writers, either built-in or user-supplied. *CowlReader* objects convert the byte streams into sequences of OWL constructs, while *CowlWriter* objects perform the reverse transformation. The parsing and rendering of OWL constructs depend on the specific format of the ontology document.

The fundamental mechanism for ontology modularization in OWL 2 is the *imports* system, where ontologies can import other ontologies to access their entities, expressions, and axioms. Imported ontologies can be sourced from storage devices, network streams, or other input sources. Cowl assigns the responsibility of managing imports to the user through the *CowlImportLoader* object, which must be implemented to return the *CowlOntology* corresponding to a specific IRI. A similar strategy is used to monitor parsing and serialization errors, where a user-designated *CowlErrorHandler* instance handles failures as appropriate for the application.

3.1.3.1 Axiom streams

A key innovation of Cowl lies in its method for reading and writing ontologies [24]. Many current OWL toolkits access knowledge within an ontology document by first deserializing it into an intermediate RDF data store. Similarly, serializing knowledge involves initially constructing the data store, then converting it into an OWL ontology document. This limitation is due to the tight coupling between OWL and RDF, which allows OWL ontologies to be expressed as RDF graphs. Given RDF's simplicity, a single OWL axiom might require multiple RDF statements, potentially scattered throughout the document, since there is no requirement for triples encoding the same axiom to be adjacent, as shown in Figure 3.2. In the illustrated example, three axioms are serialized using the *Functional* [145] and the RDF triple-based *Turtle* [155] syntaxes: the former maintains consecutive byte sequences for each axiom, whereas the latter allows them to be mixed together. This requires building an intermediate data store, which is then populated and converted into axioms when handling OWL ontologies serialized as RDF documents. Due to its simplicity and broad applicability, and to avoid API fragmentation, this *store-based* approach has traditionally been used by OWL tools for both RDF-based and OWL-specific serializations. Nevertheless, this approach is inefficient in several situations: for instance, if a data model is already available and needs to be rendered as an OWL document, or if only a small subset of axioms from an existing OWL document is required, bypassing the creation of a potentially large intermediate representation is preferable to save memory and reduce CPU usage.

The OWL functional syntax, along with all standard OWL serialization formats, permits ontologies to be serialized in a manner that ensures the resulting byte sequence adheres to the following structure:

- **Header:** including IRI prefix definitions, the ontology IRI, import directives, and ontology annotations.
- **Axioms:** a series of axiom definitions, typically constituting the largest part of the byte sequence.

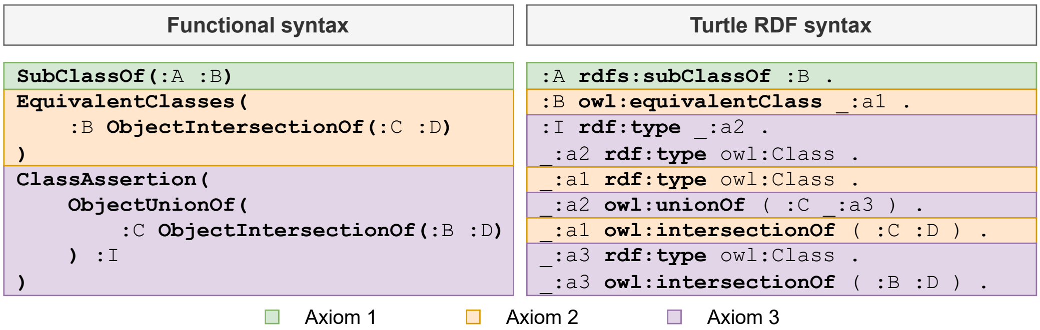


Figure 3.2: Valid encodings of three axioms in both Functional and Turtle syntax. RDF does not ensure that the triples corresponding to each axiom appear sequentially in the byte stream.

- **Footer:** a typically small and fixed set of bytes that marks the end of the document.

In addition to the header and footer, it is evident that any data model can typically be serialized into an ontology document by generating an appropriate sequence of axioms, *i.e.*,, an *axiom stream*. Cowl offers axiom stream serialization via the *CowlOStream* API, which enables user code to manage the serialization process by writing the ontology header, individual axioms, and finally the footer, without the need to generate an intermediate axiom repository. In particular, if the selected OWL document format ensures that the byte sequence serialization of two different axioms cannot be mixed, then ontologies can be *read* as axiom streams. OWL-specific serializations, such as the functional, OWL/XML, and Manchester syntaxes, comply with this rule. Consequently, the library uses a separate API, *CowlIStream*, to implement this parsing method: the user supplies a state pointer and handler functions, which the parser calls when it identifies the relevant construct in the incoming byte stream. This parsing approach is lightweight, since it bypasses populating the *CowlOntology* data store, as shown in the experiments in Section 3.1.4. However, axiom stream parsing does not support ontology editing and programmatic queries, which require a *CowlOntology* store, and it generally cannot be used for RDF serializations. However, if the stated property is maintained during serialization, *i.e.*, all triples forming each axiom are written consecutively, then RDF documents could also be processed as axiom streams. This requires informing the parser that the RDF document adheres to this property, possibly through an optional flag or a special comment at the beginning of the document. Therefore, axiom stream parsing is not entirely excluded for RDF and could be effectively implemented in future library versions.

Merging axiom stream serialization and parsing with the byte stream functionality of the *uLib* library is particularly advantageous in contexts where devices need to exchange semantically annotated data, such as in networked multi-agent systems. In these situations, the sender and receiver can share OWL knowledge graphs at the axiom level without demanding the full data store on either side. An example of the interactions supported by Cowl is presented in Section 5.3.

3.1.3.2 Optimizations for embedded platforms

The majority of the design and development decisions made during the creation of the library aim at optimizing hardware resource utilization, particularly focusing on memory usage and code size, which are the most restrictive factors in embedded systems. The methods outlined below capitalize on three fundamental features utilized across the library, as discussed in Section 3.1.3: data model immutability, pseudo-inheritance, and reference counting.

Instance sharing. Objects expected to be referenced by multiple axioms are stored in a shared instance pool. Constructor functions are designed to provide instances from this pool, preventing duplicate instances. This method is also selectively applied to strings that represent common language constructs, ensuring they are not redundantly generated.

Optimized IRI storage. Each IRI can be divided into a namespace and a local name. Since namespaces are frequently shared among several IRIs, Cowl uses two strings to represent IRIs internally: the namespace and the remainder. When an IRI appears in the byte stream, Cowl extracts the namespace based on the *XML Namespaces* specification [36] and incorporates it into the string instance pool. This approach enables multiple IRIs to reference the same namespace instance, thereby reducing the memory usage, as illustrated in Figure 3.3.

Faster equality checks. Instance sharing and immutability accelerate equality checks. Due to the nature of instance sharing, identical objects must be the same instance, allowing equality checks to be done via pointer comparisons. This avoids the need for more expensive comparisons of object contents. This is crucial since the library heavily relies on hash-based data structures that require equality checks for all CRUD (Create, Read, Update, Delete) operations. An additional optimization is applied to strings; their hashes are cached and employed to speed up checks. With an appropriate hash function[5], identical strings have equivalent hashes, avoiding to compare strings with different hash values. This greatly minimizes the need for numerous string comparisons.

Code size reduction. Although Cowl's data model covers the entire OWL 2 structural specification regarding types, most of the types and their corresponding APIs are merely placeholders or lightweight wrappers for a common low-level implementation. Most OWL constructs are represented as structured tuples of other constructs, encapsulated within a single *CowlComposite* structure containing a variable number of *CowlObject* fields, as illustrated in Figure 3.1. All types derived from *CowlComposite* share the same implementation for operations such as field retrieval, equality assessments, hash calculations, and field iteration. This approach significantly minimizes code size while maintaining type safety at the API level.

Low-level optimizations. Significant memory savings can be achieved through various low-level type optimizations. Numeric type sizes can be specified at compile time, with integers ranging from two, four, to eight bytes, and floats from four to eight bytes, which is particularly advantageous for devices with limited resources. This leads to certain trade-offs in implementation, such as constraints on the maximum size of collections or the maximum

[5]Cowl employs the DJBX33A [64] hash function for strings.

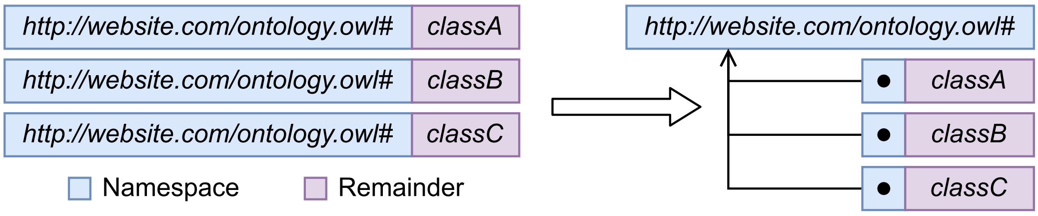


Figure 3.3: Optimized IRI storage: namespace instances are shared among IRIs.

number of references any object can hold. Strings and vectors employ the *small buffer optimization*, whereby buffers smaller than the structure they are part of are stored within it, eliminating the need for additional heap memory allocation. Moreover, Cowl uses the *pseudo-generics* strategy for collections, where type definitions and collection codes for various data types are created using preprocessor directives, similar to a template system. This balances runtime performance, including speed and memory usage, and code size, hence specialized collection versions are developed only for commonly used and critical data structures.

3.1.4 Experiments

Assessing Cowl's performance is essential to confirm its suitability for the various environments and platforms demanded by the SWoE. A preliminary experimental campaign was conducted in [25], where an alpha version of the library was compared to leading OWL ontology manipulation toolkits. The findings of this assessment are summarized in Section 3.1.4.1, followed by the results of an additional evaluation carried out with a stable version of the library on a widely-used microcontroller platform [24], as described in Section 3.1.4.2. In both evaluation campaigns, the EVOWLUATOR framework (refer to Section 3.3) was exploited for test automation, taking advantage of its capability to execute command line utilities on various OWL datasets and collect performance data, specifically regarding execution time and memory consumption.

3.1.4.1 Laptop tests

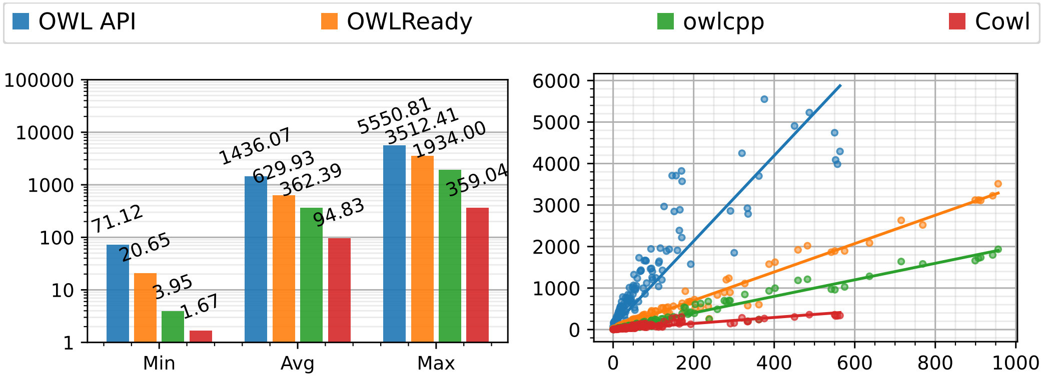
A preliminary version of the library has been evaluated against the following cutting-edge OWL tools: OWL API (5.1.20), Owlready2 (0.37), and owlcpp (0.3.3). These libraries were adapted into simple command-line tools for intregration into evOWLuator, taking the ontology path as an argument. The tests were conducted on a 2021 Apple MacBook Pro 16"[6]. The performance metrics reported are the mean of 5 cold runs.

The dataset used for the experiments is composed of 109 knowledge bases, derived from the 2014 *OWL Reasoner Evaluation Workshop* (ORE2014) competition corpus[7] as follows: all functionally-styled ontologies were considered and arranged by size, with one sample taken at each MiB boundary (if available). The tests focused on ontology parsing and querying times, as well as peak memory consumption. The following queries were chosen for benchmarking: **(Q1)** retrieval of all axioms within the ontology; **(Q2)** retrieval of all classes within the ontology and, for each class, retrieval of all subclass axioms directly referring to it as the subclass or superclass.

Prior to showcasing the outcomes of the experimental campaign, it is essential to highlight some key distinctions between the evaluated tools:

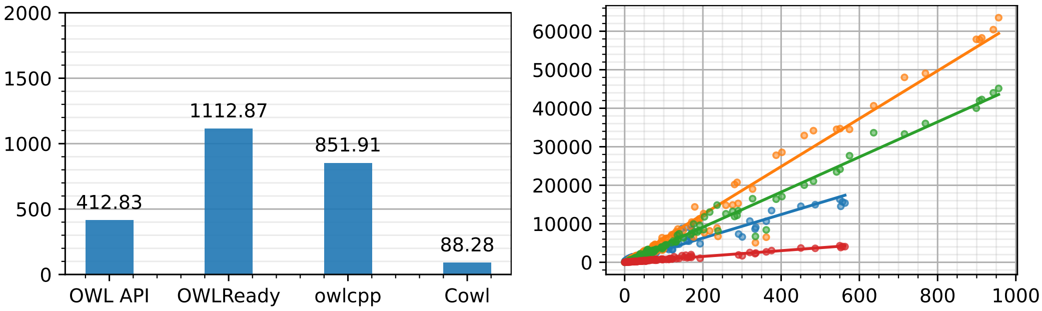
- **Serialization Support**: OWL API support most syntaxes in literature; Owlready2 supports RDF/XML, OWL/XML, and Turtle; owlcpp is restricted to RDF/XML, and Cowl currently has only a functional syntax parser. Consequently, Owlready2 and owlcpp have been tested on RDF/XML documents, while the OWL API and Cowl have been used with their functional syntax versions. This is evident from the scatterplot data points, which show larger ontology sizes for tools using the RDF/XML serialization.
- **Data Model Architectures**: OWL API and Cowl provide a direct correspondence to OWL axioms and structures, whereas Owlready2

[6] Apple M1 Max System-on-Chip with 64 GB RAM, 1 TB SSD, macOS Monterey 12.3.
[7] ORE2014 corpus: `http://dl.kr.org/ore2014`



(a) Dataset-wide min, avg and max (*MiB*). **(b)** Memory peak by ontology size (*MiB*).

Figure 3.4: Comparison of peak memory usage.

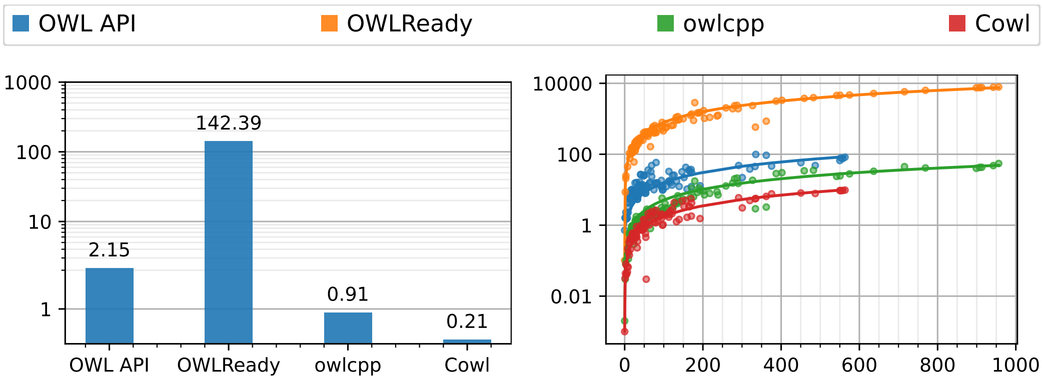


(a) Cumulative time (*s*). **(b)** Time (*ms*) by ontology size (*MiB*).

Figure 3.5: Comparison of parsing time.

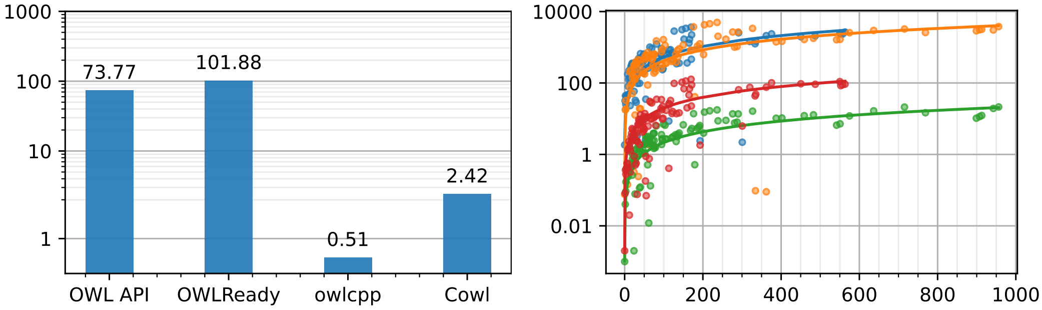
and owlcpp keep ontologies in RDF triple stores. This leads to significant performance differences in ontology queries due to the distinct implementation methods.

Figure 3.4 presents the peak memory usage metrics, determined by EvOWLuator as the *maximum resident set size* (MRSS) of the process. Cowl exhibits a notably smaller memory footprint compared to all other tools, with a particularly significant difference when compared to the non-native OWL API and Owlready2 libraries. All libraries show an approximately linear increase in memory usage relative to the ontology size, though the absolute values on highly resource-limited nodes –such as those classified as Class 0 and Class 1 by the Internet Engineering Task Force (IETF) [35]– may differ from those on the 64-bit test workstation.



(a) Cumulative time (s). **(b)** Time (ms) by ontology size (MiB).

Figure 3.6: Comparison of time taken to retrieve all axioms.



(a) Cumulative time (s). **(b)** Time (ms) by ontology size (MiB).

Figure 3.7: Comparison of time taken to retrieve all subclass axioms for all classes.

Figure 3.5 presents data on ontology parsing turnaround times. Similar to memory usage, Cowl outperforms the other tools, with all libraries exhibiting linear behavior concerning ontology size. The OWL API demonstrates exceptional parsing speed, even surpassing the native owlcpp library, although it consistently requires more memory compared to the other evaluated toolkits.

Figure 3.6 illustrates the retrieval times for all axioms. Cowl remains the quickest among the frameworks, with the OWL API showing performance close to that of owlcpp. Since the query involves iterating through all constructs in the knowledge base, triple store-based tools are disadvantaged because they deal with a higher volume of constructs, as OWL axioms are typically represented using multiple RDF statements.

Figure 3.7 displays the turnaround times for retrieving all subclass axioms

```
SELECT *
WHERE {
    { ?a_class rdfs:subClassOf ?b_class . }
    UNION
    { ?b_class rdfs:subClassOf ?a_class . }
    FILTER (!isBlank(?a_class))
}
```

Listing 1: Retrieval of all subclass axioms for each class from a triple store.

for all classes. owlcpp leads in performance, followed by Cowl, while OWL API and Owlready2 show similar and significantly higher values. The query was intentionally selected to be efficiently handled by triple stores (*e.g.*,, via the SPARQL query in Listing 1) due to their strong reliance on constructs that reference specific entities. In contrast, axiom-based tools need to create auxiliary indices to expedite reference checks for specific OWL entities and might still end up evaluating unrelated axioms based on the indexing approach. Enhancing the performance of such queries is feasible with additional indices or more advanced data structures. However, this often leads to increased memory consumption, which was not considered a beneficial trade-off given Cowl's emphasis on memory-restricted environments.

3.1.4.2 Embedded board tests

Although previous findings demonstrated cutting-edge performance compared to other well-known OWL toolkits, these evaluations were carried out on a laptop testbed. To validate that the proposed system can truly penetrate the SWoE device spectrum, it is necessary to perform assessments on an appropriately small nano-scale device. Moreover, it is essential to compare the innovative *axiom-based* method for ontology manipulation with the conventional *store-based* technique to fully grasp its advantage in SWoE scenarios.

Arduino[8] boards, known for their open-source design, extensive community support, and wide range of compatible hardware and software libraries, are suitable for various applications, from simple automation tasks to complex

[8] Arduino home: `https://arduino.cc`

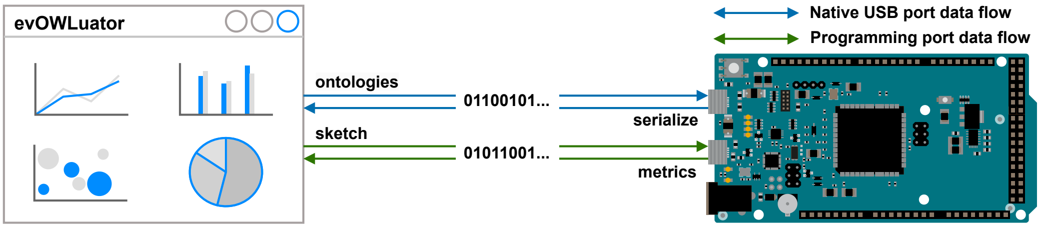


Figure 3.8: Experimental setup for tests on Arduino Due.

robotics and IoT infrastructure. Therefore, the Arduino Due board[9] has been selected as the testbed for an extensive experimental campaign. To our knowledge, this is the first instance of deploying and evaluating a fully capable OWL library on a microcontroller of this size.

The experimental configuration is depicted in Figure 3.8. The PC has been connected to the board, using the EVOWLUATOR framework to supervise all experiments. Due to the modular design of the framework and its ability to handle reasoning tasks on remote devices (refer to Section 3.3.4), a plugin has been developed to enable PC-board communication through the UART[10] and USB[11] 2.0 ports. Specifically, the UART programming port has been used to upload Arduino "sketches" responsible for conducting on-board tests involving the Cowl library and reporting performance data back to the evaluation framework. Meanwhile, the faster USB port has been employed for larger data exchanges, such as transferring serialized ontology documents to and from the board. Given that Cowl's ontology I/O functionalities are based on *uLib* streams, reading and writing ontologies through the USB port has merely required custom input/output streams that utilize Arduino's built-in *USBSerial* class.

The dataset for the experiments comprises the entire 2014 OWL Reasoner Evaluation Workshop (ORE2014) competition corpus [16], which includes 16555 ontologies in functional syntax, with sizes ranging from 10 KiB to more than 500 MiB. The tests focus on ontology parsing, serialization, and querying

[9] Featuring a 32-bit ARM Cortex M3 CPU running at 84 MHz, with 96 KiB of SRAM and 512 KiB of flash storage.

[10] Universal Asynchronous Receiver-Transmitter.

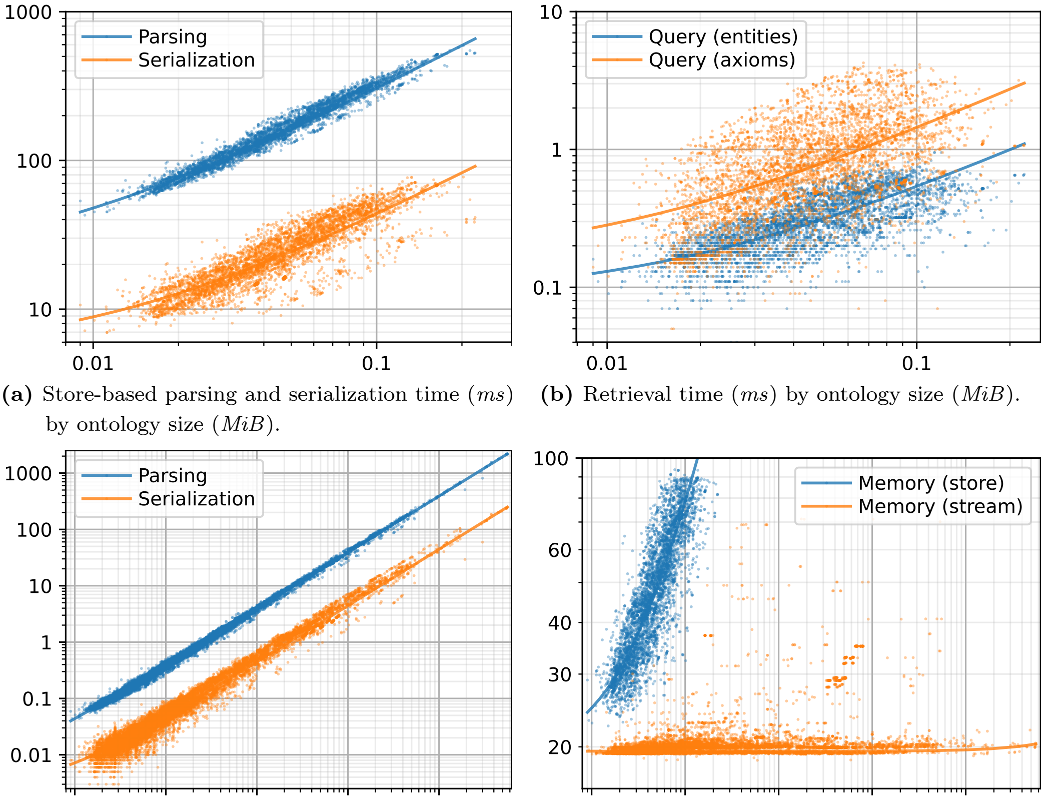
[11] Universal Serial Bus.

times, along with peak memory usage. Parsing and serialization times exclude the duration needed to transfer the ontology document via USB port in the results presented, as this depends solely on the USB port's transfer speed and affects the overall time, making it challenging to determine the exact processing time of the ontology.

The initial test focused on loading ontologies from the USB port into the in-memory data store, executing queries, and serializing them back to the USB port. Figure 3.9a illustrates, on a logarithmic scale, the time required to parse and serialize ontologies based on their size. Both parsing and serialization times exhibit a linear trend, as expected. The board successfully processed 9543 ontologies (57.6% of the entire dataset), with the largest ontology being 228 KiB. The remaining ontologies could not be processed due to memory limitations.

Figure 3.9b shows the time needed to retrieve: (i) all the entities in an ontology; (ii) for each ontology class, all subclass axioms that directly cite it as either subclass or superclass. The first query executes in under 1 ms even for the largest ontologies, whereas the more complex second query completes in under 3 ms. This performance reflects the efficiency of the *CowlOntology* store, along with the enhancements mentioned in Section 3.1.3.2, even on highly resource-constrained devices.

The second experiment involved processing ontologies as a stream of axioms, concurrently serializing detected axioms. In this scenario, because the USB 2.0 port does not support full-duplex communication, serialization was performed over a dummy byte stream that discards its output. In this configuration, Cowl can process almost the entire ontology dataset (16464 ontologies, 99.5%), as illustrated in Figure 3.9c: the largest processed ontology is also the largest ontology in the dataset, having a size of 563 MiB. This efficiency is mainly because ontologies are handled one axiom at a time, eliminating the need for an intermediate data store, thereby significantly reducing memory usage. This benefit becomes more evident when comparing peak memory usage, as shown in Figure 3.9d. The conventional method is constrained by the size of the ontology being processed, whereas the stream-



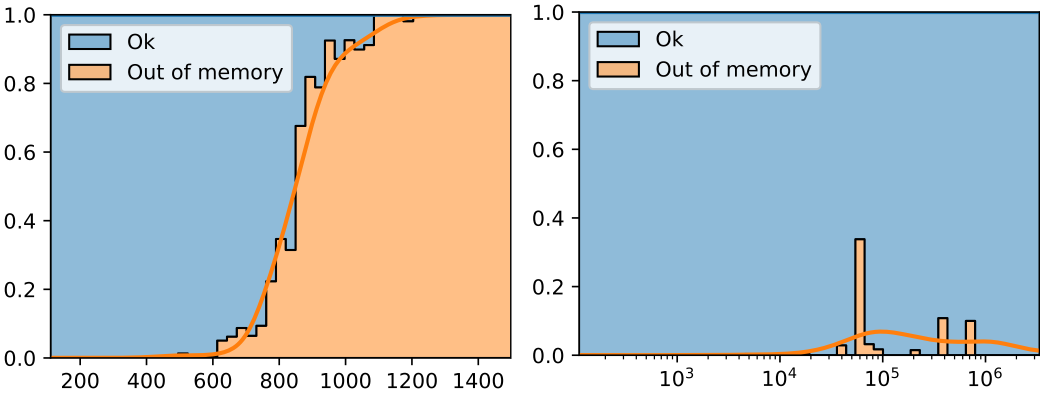
(a) Store-based parsing and serialization time (ms) by ontology size (MiB).

(b) Retrieval time (ms) by ontology size (MiB).

(c) Stream-based parsing and serialization time (s) by ontology size (MiB).

(d) Store- vs stream-based memory usage (KiB) by ontology size (MiB).

Figure 3.9: Performance metrics on Arduino Due.



(a) Store-based approach.

(b) Stream-based approach.

Figure 3.10: Ratio of out-of-memory events by axiom count in ontology parsing.

based approach is limited only by the size of the largest axiom in the ontology. Consequently, the trend in memory usage changes from linear to roughly constant, with a few exceptions, allowing for the potential management of much larger ontologies.

A significant observation arises when comparing the rate of out-of-memory errors between the store- and stream-based methods. Figure 3.10 displays a histogram of ontologies organized by axiom count (x-axis) against the percentage of ontologies that experience an out-of-memory (OOM) event within each bin (y-axis). As depicted in Figure 3.10a, the store-based method shows a steady increase in the ratio of OOM events approaching the 1.0 threshold for ontologies containing more than approximately 1100 axioms. This trend aligns with the limited availability of SRAM memory, which is quickly exhausted by the in-memory store. On the other hand, as illustrated in Figure 3.10b, the stream-based method allows for the processing of ontologies with a substantial number of axioms, significantly reducing the likelihood of memory saturation.

The experimental campaign validates the Cowl library's ability to manage OWL knowledge graphs on embedded devices, highlighting its efficiency in handling and processing ontologies despite the limited resources typically available. Thanks to its architecture and optimizations, it can manage a significant portion of the dataset using a classical store-based approach, allowing embedded applications to query and edit moderately sized ontologies. Additionally, its stream-based technique, useful for some specific use cases, serves as a practical fallback when memory constraints prevent larger ontologies from being stored entirely in memory, enabling the tool to handle graphs of virtually any size. These findings collectively demonstrate that Cowl's design, and innovative processing methods effectively address the challenges of managing OWL ontologies on devices with limited capacity, thus expanding the scope of SWoE applications.

3.2 Tiny-ME: a multiplatform reasoning engine for the SWoE

This section presents *Tiny-ME*[12] [164] (the Tiny Matchmaking Engine), a SWoE-oriented matchmaking and reasoning engine. It boasts a cross-platform architecture built from scratch, with a shared core in C language ensuring both portability and efficient execution of KB management operations and reasoning capabilities. Above the low-level core, various high-level APIs are available to ease integration across numerous platforms and technology stacks, from the Web to small, resource-limited devices.

The reasoner offers both standard (Ontology Classification, Ontology Coherence, Concept Subsumption, and Concept Satisfiability) and non-standard (Concept Abduction, Contraction, Bonus, Difference, and Covering) inference services within an OWL 2 subset that aligns with the $\mathcal{ALN}(\mathcal{D})$ Description Logic (*Attributive Language with unqualified Number restrictions and Datatypes*). This system is fully compatible with all major desktop and mobile operating systems, and also works seamlessly with all leading browser platforms via a port to the *WebAssembly*[13] runtime, [161], enabling its use in client-side Web applications. In addition, Tiny-ME can function for reasoning within microservice architectures for Web and cloud applications; to this end, the standard *OWLlink* [115] protocol for remote reasoner invocation has been enhanced to include support for the non-standard inference services. The architectural and technological decisions made for this system also enable deployment on nano-scale devices, including embedded drone autopilots and microcontrollers. Such platform versatility is unprecedented in the OWL reasoning domain, positioning the system as a contender for establishing a practical SWoE infrastructure.

The structure of this paragraph is as follows: Section 3.2.1 provides the background; Section 3.2.2 elaborates on the supported inference services;

[12]Tiny-ME: `http://swot.sisinflab.poliba.it/tinyme/`
[13]WebAssembly: `https://webassembly.org/`

Section 3.2.3 explains the core principles behind the system's design along with its architecture and key data structures; Section 3.2.4 describes the platform-specific and Web APIs for enhanced cross-platform integration; finally, Section 3.2.5 details the experimental results on desktop, mobile, browser, and microcontroller platforms.

3.2.1 Background

Classical Semantic Web environments typically benefit from consistent access to extensive computational and networking resources. In contrast, the SWoE operates under significantly different conditions, characterized by hardware and micro-devices with limited resources distributed across various physical locations, serving as information sources. The availability of these devices is highly variable and unpredictable, influenced by factors such as user and device mobility, limitations of wireless communication links, and constraints on energy supply. As a result, processing batch workloads on highly expressive and complex Knowledge Bases is not suitable for SWoE contexts. Instead, inference engines need to quickly process queries on smaller and less complex annotations. This shift necessitates a reevaluation of the expressiveness of the logical languages used, adapting them to these specific operational constraints. The historically intractable worst-case complexity of OWL DL led to the creation of OWL 2 *profiles*. These profiles simplify the language and the permissible KB axioms, allowing for efficient algorithms while retaining sufficient expressiveness for practical applications. $\mathcal{EL}^{++}$ [13] extended the basic $\mathcal{EL}$ (*Existential Language*) DL to support a range of applications, especially those involving large ontologies with moderate expressiveness. The proposal introduced a polynomial-complexity Ontology Classification algorithm, enabling the development of high-performance $\mathcal{EL}^{++}$ classifiers like *ELK* [93]. Similarly, one of the first methods for adapting non-standard inferences such as Concept Contraction and Concept Abduction to pervasive computing [162] involved using structural algorithms on acyclic TBoxes within the $\mathcal{AL}$ (Attributive Language) DL through a mobile Relational Database Management System (RDBMS) query layer.

The first generation of inference engines tailored for mobile and IoT environments focused on more efficient methods compared to traditional Semantic Web reasoners to mitigate the constrained memory capacity of devices. *Pocket KRHyper* [190], a Java Micro Edition library for theorem proving and model generation using the hyper tableau calculus, was the earliest instance of a lightweight inference engine, although it struggled with significant memory limitations of the target platform. The μOR [6] reasoner utilized a straightforward resolution and pattern matching algorithm for a subset of OWL-Lite. Similarly, *MiRE4OWL* [96] was a rule-based mobile inference engine employing OWL-DL semantics, resolved with the well-known RETE algorithm. Recent efforts have focused on downsizing reasoning engines for use in embedded devices. Reference [74] involved the adaptation of consequence-driven $\mathcal{EL}^+$ reasoning to a Programmable Logic Controller (PLC) aimed at industrial automation. The rule-based reasoner described in [200] employed selective rule loading in conjunction with a two-stage RETE algorithm, showing good performance on the *Sun SPOT* sensor platform for ontologies of small and medium sizes.

Another category of reasoners includes those originally created for conventional computer platforms and later adapted for mobile devices, leveraging the increasing computational power in smartphones and tablets. As mentioned in [30], five Java-based OWL reasoners were customized for use on Android, although this adaptation required a substantial effort. In [92], the ELK reasoner was redesigned to transition from Java Standard Edition (SE) to Android, reducing memory usage and adding support for multi-core CPU architectures. Apart from adaptations, native mobile inference engines have also been built. For example, *Mini-ME 2.0* [183] is specifically crafted for Android, serving both as a *matchmaker* and a reasoner, and remains compatible with Java SE. In [208], a rule engine for Android incorporates the innovative $RETE_{pool}$ algorithm on OWL 2 RL rulesets, capable of balancing memory usage and performance. *Mini-ME Swift* [167] is the first OWL reasoner for iOS, re-engineered from the aforementioned Mini-ME using the OWL API for iOS [166], an iOS-specific knowledge manipulation library.

The World Wide Web is a widespread platform for deploying new software tools and applications, with browsers often acting as the primary interface for many relevant scenarios [201]. Web applications frequently rely on remote services for DL reasoning. The JavaScript (JS) language, which is inherently oriented towards the Web, has rarely been selected for developing reasoning engines, mainly due to its lower performance compared to other programming languages, and because invoking DL inferences through client-server protocols like OWLlink [115], supported by many Semantic Web reasoners, is considered simpler. Many semantic-enabled Web applications for traditional desktop clients follow this approach across various domains. However, design and technological choices often hinder their adaptation to mobile and ubiquitous environments. Notable semantic-enabled mobile-oriented Web applications include location-based discovery clients for mobile devices, such as *DBpedia Mobile* [19] and *LOD4AR* [214]. *OntoWiki Mobile* [62] is a mobile knowledge management application that uses HTML5 and jQuery Mobile to store user-annotated knowledge locally, even without an Internet connection, and synchronize it with a remote server once a connection is available. A survey in [226] on semantic-enabled mobile applications found that only 4 out of 36 apps were developed using Web-based cross-platform technologies.

The use of client-side Web technologies for developing cross-platform reasoning engines is not very common. An significant example is the *EYE* (Euler Yet another proof Engine) [213], which can perform forward and backward rule chaining over Euler paths. This engine has been adapted to JS-based environments such as the Web and *Node.js*[14] through a JS port of the *SWI Prolog* [220] engine. The *MobiBench* framework for evaluating mobile semantic rule engines [209] supports the integration of JS-based rule engines for the Web and other platforms. This integration is facilitated by the *Apache Cordova*[15] Software Development Kit (SDK). *HyLAR+*[16] [203], a hybrid OWL 2 EL reasoner, can execute reasoning tasks on both the server and the client side using the same JS code with Node.js and *AngularJS*[17],

[14]Node.js: `https://nodejs.org/`
[15]Apache Cordova home: `https://cordova.apache.org/`
[16]HyLAR GitHub repository: `https://github.com/ucbl/HyLAR-Reasoner`
[17]AngularJS home: `https://angularjs.org/`

respectively. It leverages the *JSW* semantic technology framework for JS[18], which includes the *BrandT* browser-based OWL 2 EL inference engine [198]. WebAssembly [161] offers a feasible method to create or migrate reasoning engines to Web applications and standalone runtime environments such as Node.js with minimal performance loss. However, the proposals are still in the early stages. *EYE JS*[19] represents the first WebAssembly port of the EYE engine.

Table 3.2 provides a summary of key features of the reasoning systems under consideration. Regarding semantic matchmaking functionality, as discussed in Section 2.3.2, the label "*Full*" denotes systems that support only *exact* or full/subsume match types [108], where resources must be subsumed by the request. The label "*Approximated*" identifies systems capable of handling additional match types like *potential/intersection* and *partial/disjoint*, depending on whether subsumption does not hold and when the conjunction of request and resource is unsatisfiable. In conclusion, the *Explanation* column indicates whether the systems are capable of providing formal justifications for their conclusions, such as the reasons why subsumption does not hold between two concept expressions.

[18] JSW GitHub repository: `https://github.com/JS-WindowFramework/JSW`
[19] EYE JS GitHub repository: `https://github.com/eyereasoner/eye-js`

Table 3.2: Features of related reasoning systems

	Features				Platforms				
Name	**DL**	**Algorithm family**	**Matchmaking**	**Explanation**	**Language**	**Web/Cloud**	**Desktop**	**Mobile**	**Embedded**
μOR [6]	OWL-Lite$^-$	Pattern matching	Full		Java		JVM		JamVM
COROR [200]	$\mathcal{SHOIN}(\mathcal{D})$	RETE	Full		Java		JVM		Sun SPOT
ELK [92]	$\mathcal{EL}^+$	Consequence-based			Java		JVM	Android	
HermiT [67, 30]	$\mathcal{SROIQ}(\mathcal{D})$	Tableaux	Full		Java	OWLLink	JVM	Android	
JFact [30]	$\mathcal{SROIQ}(\mathcal{D})$	Tableaux	Full		Java	OWLLink	JVM	Android	
Konclude [196]	$\mathcal{SROIQV}(\mathcal{D})$	Hybrid	Full		C++	OWLLink	Native		
Mini-ME [183]	$\mathcal{ALN}$	Structural	Full, Approx.	Yes	Java	OWLLink	JVM	Android	
Mini-ME Swift [167]	$\mathcal{ALN}$	Structural	Full, Approx.	Yes	Swift		macOS	iOS	
MiRE4OWL [96]	$\mathcal{SHOIN}(\mathcal{D})$	RETE	Full		C++		Native	Windows	
Pellet [30]	$\mathcal{SROIQ}(\mathcal{D})$	Hybrid	Full	Yes	Java		JVM	Android	
PLC-based [74]	$\mathcal{EL}^+$	Consequence-based			SCL				SIMATIC
Pocket KRHyper [190]	$\mathcal{ALC}^+$	Tableaux	Full		Java		JVM	J2ME	
RETEpool [208]	OWL-RL	RETE	Full		Java		JVM	Android	
Tiny-ME	$\mathcal{ALN}(\mathcal{D})$	Structural	Full, Approx.	Yes	C, Obj-C, Java	OWLLink, JavaScript	Native, JVM	Android, iOS	Any with C support

3.2.2 Inference services

The first release of Tiny-ME [164] focused on the $\mathcal{ALN}$ DL, as discussed and justified in Section 2.3.1. It features structural algorithms with polynomial complexity for $\mathcal{ALN}$ concept expressions. This section addresses the algorithmic implementation of the inference services, which were formally presented in Section 2.3.2. As mentioned there, in the context of $\mathcal{ALN}$ DL, the process of comparing unfolded and normalized concept expressions essentially reduces to set operations. For instance, let's examine this $\mathcal{ALN}$ TBox:

$$
\begin{aligned}
&A \equiv \forall P.D \sqcap \geq 3P \\
&B \sqsubseteq \forall P.D \\
&C \sqsubseteq B \sqcap \geq 2P \\
&E \sqsubseteq D \\
&F \equiv E \sqcap \neg D
\end{aligned}
$$

After applying unfolding and CNF normalization, the resulting concept expressions (CEs) are:

$$
\begin{aligned}
&A \rightarrow \forall P.D \sqcap \geq 3P \\
&B \rightarrow B \sqcap \forall P.D \\
&C \rightarrow B \sqcap C \sqcap \forall P.D \sqcap \geq 2P \\
&D \rightarrow D \\
&E \rightarrow E \sqcap D \\
&F \rightarrow \bot
\end{aligned}
$$

It is noted that B and C are present as conjuncts in their respective unfolded concept expressions, whereas A is not. According to [14, §9.2.3], in $\mathcal{ALN}$ DL, an atomic concept is included in its own unfolding if and only if it is on the LHS of an inclusion axiom; otherwise, it is excluded if it is on the LHS of an equivalence axiom. An important point to mention regards F, whose concept expression turned into $\bot$: Initially, F was defined as the conjunction of E and $\neg D$, which expands to $E \sqcap D \sqcap \neg D$, leading to an obvious contradiction. Hence, its concept expression reduces to $\bot$ when computing the CNF, implying that the F concept is not **satisfiable**.

Finally, let us consider the task of checking whether a **subsumption** relationship exists between A and C: once unfolded and normalized, only the CEs of A and C are needed, *i.e.*, the entire TBox is no longer necessary. In this scenario, $A \sqsubseteq C$ holds because $\forall P.D$ appears in both the CEs of A and C, and $(\geq 3P)$ is more specific than $(\geq 2P)$. The detailed algorithm for subsumption will be discussed next.

Preprocessing

Unfolding and **CNF normalization** are essential preprocessing steps, and thus their optimization has been given significant focus. According to [167], improvements were made by caching fully unfolded and normalized concepts. However, while developing Tiny-ME, it became evident that caching could also be applied to *intermediate unfolded concepts* [164]. Given an acyclic concept B, any other concept C that is recursively unfolded as part of unfolding B is also completely unfolded, making it a candidate for caching. Nevertheless, C is not yet in normal form, so the concept cache must distinguish between concepts that have only been unfolded and those that have been both unfolded and normalized. This approach has two advantages: (i) it allows for the reuse of the unfolded description of C during the normalization of other concepts; (ii) it reduces computation when C needs to be normalized, since the unfolding step is performed only once.

Satisfiability and Subsumption

Satisfiability can be easily verified through CNF normalization. As mentioned in Section 2.3.2, a concept expression A in CNF form is either $\perp$ or the conjunction of an arbitrary number of supported constructs. To determine if A is satisfiable, simply ensure that it is not $\perp$.

Subsumption leverages the classic structural algorithm described in [14]. Given two CNF-normalized concept expressions $R \equiv R_{CN} \sqcap R_{\leq} \sqcap R_{\geq} \sqcap R_{\forall}$ and $S \equiv S_{CN} \sqcap S_{\leq} \sqcap S_{\geq} \sqcap S_{\forall}$, to assess if $R \sqsubseteq S$ holds:

1. if $R \equiv \perp$, then $R \sqsubseteq S$.

2. for each atomic concept $A \in S_{CN}$, if $A \notin R_{CN}$, then $R \not\sqsubseteq S$.

3. for each negated atomic concept $\neg A \in S_{CN}$, if $\neg A \notin R_{CN}$, then $R \not\sqsubseteq S$.

4. for each role P such that $\leq xP \in S_{\leq}$, if $\leq yP \in R_{\leq}$ with $x < y$, then $R \not\sqsubseteq S$.

5. for each role P such that $\geq xP \in S_{\geq}$, if $\geq yP \in R_{\geq}$ with $x > y$, then $R \not\sqsubseteq S$.

6. for each role P such that $\forall P.E \in S_{\forall}$, if $\forall P.F \in R_{\forall}$ with $F \not\sqsubseteq E$, then $R \not\sqsubseteq S$.

7. otherwise, $R \sqsubseteq S$.

Concept Contraction

Having $R \equiv R_{CN} \sqcap R_{\leq} \sqcap R_{\geq} \sqcap R_{\forall}$ and $S \equiv S_{CN} \sqcap S_{\leq} \sqcap S_{\geq} \sqcap S_{\forall}$ unfolded and CNF-normalized concept expressions in $\mathcal{ALN}$, $CC(R, S)$ is computed by means of the following structural algorithm:

1. set $K \equiv K_{CN} \sqcap K_{\leq} \sqcap K_{\geq} \sqcap K_{\forall} := R$ and $G \equiv G_{CN} \sqcap G_{\leq} \sqcap G_{\geq} \sqcap G_{\forall} := \top$;

2. for each atomic concept $A \in K_{CN}$, if $\neg A \in S_{CN}$, then move A from K_{CN} to G_{CN};

3. for each negated atomic concept $\neg A \in K_{CN}$, if $A \in S_{CN}$, then move $\neg A$ from K_{CN} to G_{CN};

4. for each role P such that $\geq xP \in K_{\geq}$ and $\leq yP \in S_{\leq}$ with $y < x$, replace $\geq xP$ in $K_{\geq}$ with $\geq yP$ and put $\geq xP$ in conjunction with the concept expression for $G_{\geq}$;

5. for each role P such that $\leq xP \in K_{\leq}$ and $\geq yP \in S_{\geq}$ with $y > x$, replace $\leq xP$ in $K_{\leq}$ with $\leq yP$ and put $\leq xP$ in conjunction with the concept expression for $G_{\leq}$;

6. for each role P s.t. $\forall P.E \in K_\forall$ and $\forall P.F \in S_\forall$, if $\exists \geq xP$ with $x > 0$ either in $K_\geq$ or in $S_\geq$, then compute Contraction recursively on the fillers: $\langle G', K' \rangle = CC(E, F)$, put $\forall P.G'$ in conjunction with the concept expression for $G_\forall$ and replace $\forall P.E$ with $\forall P.K'$ in the concept expression for $K_\forall$.

For example, referring to the above example TBox:
$\langle G, K \rangle = CC(A, \leq 2P) = \langle \geq 3P, \forall P.D \sqcap \geq 2P \rangle$.

Concept Abduction and Bonus

Having $R \equiv R_{CN} \sqcap R_\leq \sqcap R_\geq \sqcap R_\forall$ and $S \equiv S_{CN} \sqcap S_\leq \sqcap S_\geq \sqcap S_\forall$ unfolded and CNF-normalized concept expressions in $\mathcal{ALN}$, the structural algorithm for $CA(R, S)$ is:

1. set $H \equiv H_{CN} \sqcap H_\leq \sqcap H_\geq \sqcap H_\forall := \top$;
2. for each (possibly negated) atomic concept $A \in R_{CN}$, if $A \notin S_{CN}$, then put A in conjunction with the concept expression for H_{CN};
3. for each role P such that $\geq xP \in R_\geq$, if $\geq yP \notin S_\geq$ or $\geq yP \in S_\geq$ with $y < x$, then put $\geq xP$ in conjunction with the concept expression for $H_\geq$;
4. for each role P such that $\leq xP \in R_\leq$, if $\leq yP \notin S_\leq$ or $\leq yP \in S_\leq$ with $y > x$, then put $\leq xP$ in conjunction with the concept expression for $H_\leq$;
5. for each role P s.t. $\forall P.E \in R_\forall$ and $\forall P.F \in S_\forall$, then compute Abduction recursively on the fillers: $H' = CA(E, F)$ and put $\forall P.H'$ in conjunction with the concept expression for $H_\forall$.

For example, referring to the above example TBox: $H = CA(A, B) = \geq 3P$.

The algorithm for finding the Bonus B of S w.r.t. R is identical the CA problem where the roles of R and S are reversed, *i.e.*, R acts as resource and S as request.

Concept Difference

Given two unfolded and CNF-normalized $\mathcal{ALN}$ concept expressions $R \equiv R_{CN} \sqcap R_{\leq} \sqcap R_{\geq} \sqcap R_{\forall}$ and $S \equiv S_{CN} \sqcap S_{\leq} \sqcap S_{\geq} \sqcap S_{\forall}$, $CD(R, S)$ is computed structurally as in what follows:

1. if $R \sqcap S \sqsubseteq \bot$, *i.e.*, R and S are not consistent, then use Concept Contraction to retrieve the part K of S that is consistent with R. Otherwise, $K := S$.
2. return the Concept Bonus between K and R: $D := CB(R, K)$.

For example, referring again to the example TBox:
$CD(B, A) = B$ and $CD(A, B) = \geq 3P$.

Concept Covering

The structural algorithm for CCov, given a set $\{R, S_1, S_2, ..., S_n\}$ of unfolded and CNF-normalized concept expressions, is:

1. set $\mathcal{S} := \emptyset$ and $H := R$;
2. repeat the following steps until $S_{max} \equiv \top$:
   (a) set $r_{min} := \|H\|$ and $S_{max} := \top$;
   (b) for each S_i in $\mathcal{S}$, if $S_i \sqcap R \not\sqsubseteq \bot$ (*i.e.*, S_i and R are consistent) and $CD(S_i, H) \not\equiv \top$ (*i.e.*, S_i covers H), then compute $H_i, r_i := CA(H, S_i)$. If $r_i < r_{min}$, update $r_{min} := r_i$, $S_{max} := S_i$ and $H_{max} := H_i$;
   (c) if $S_{max} \not\equiv \top$, add S_{max} to $\mathcal{S}_c$, remove it from $\mathcal{S}$, and set $H := H_{max}$;
3. return $\mathcal{S}_c, H$.

During step 2, each pass through the loop computes H_i and r_i as the CA hypothesis and penalty relative to the remaining uncovered part H. The resource with the smallest penalty, that is, the one that covers the most of H, is included in the set $\mathcal{S}_c$ until no more resources can expand the coverage.

Ontology Classification

Tiny-ME employs a modified version of the *enhanced traversal* algorithm described in [15], which also considers subsumption cycles identified during the ontology preprocessing phase. The results of subsumption checks are *cached*, a common practice among OWL reasoners, although much effort was made to avoid checks whenever possible: classification is conducted following the concept's *definition order* [15], enabling the omission of the expensive *bottom search* stage of the traversal algorithm for primitive concepts with acyclic descriptions. The use of *told disjoints* [206] and *told subsumers* is implemented by pre-loading the subsumption cache based on explicitly stated subclass, class equivalence, and class disjointness axioms. Furthermore, the implementation of *synonym merging* (*e.g.*, if it is deduced that $B \equiv C$, then the taxonomy nodes for B and C are merged) minimizes memory usage and search time by reducing the size of the tree.

Ontology Coherence

A straightforward method involves performing CNF normalization for every concept in the TBox [183]. However, previous studies [167] have shown that this approach can be time consuming, particularly for larger TBoxes. One possible solution to alleviate this problem is to adopt aggressive caching policies for unfolded concepts, although this increases memory consumption. Tiny-ME tackles this issue by performing the Coherence check with a modified version of the Classification algorithm, which stops immediately upon identifying an unsatisfiable concept. In Tiny-ME, normalization is *lazy*, meaning it occurs only when necessary for an inference task. Because the Classification algorithm systematically avoids costly subsumption checks whenever possible, the overall number of required normalizations is also reduced. Therefore, this method generally provides a significant improvement in both time and memory efficiency compared to the straightforward approach. Moreover, the Coherence check of a TBox $\mathcal{T}$ can be omitted in the following scenarios: (i)

$\mathcal{T}$ is *trivially incoherent* if $\exists$ a concept expression C in $\mathcal{T} \mid C \sqsubseteq \bot$; (ii) $\mathcal{T}$ is *trivially coherent* if $\mathcal{T}$ contains no *disjoint concept* axioms and *number restrictions* are either missing or all of the same type (*i.e.*, either minimum or maximum cardinality). Both conditions can be verified cost-effectively during the KB loading phase, as it simply involves checking the presence of specified constructors in the TBox.

Semantic Matchmaking

The standard inference services outlined in Section 2.3.2 are profitable for conventional semantic-enabled applications. However, they often fail in situations that demand more sophisticated responses beyond simple binary (Boolean) decisions, such as in matchmaking or negotiation, which are common in SWoT/SWoE scenarios. In these cases, non-standard and *non-monotonic* inferences are more useful. As noted in [163], these inferences provide explanations for the results and support the retraction of conflicting information under the Open World Assumption.

Semantic matchmaking involves identifying the most relevant element from a set of *resources* based on a given *request*, where both are defined as satisfiable concept expressions within a shared set of axioms $\mathcal{T}$ in a DL. The output includes a set of concepts, each scored for its semantic relevance to the submitted query. Beyond resource discovery, matchmaking also aids in data stream analytics by transforming statistical classification problems into semantic matchmaking tasks. For instance, in [173], sample features were annotated with conjunctive concept expression fragments, and target classes were defined with concept expressions within a common ontology. These applications are particularly pertinent in SWoT/SWoE scenarios, which are characterized by dynamic data and fragmented information [51, 165, 226]. The present work builds on the classification of match types from [56], differing from earlier works [108, 143] by preferring the *full/subsume* match category, which does not require to hypothesize additional information about the resource. Moreover, the proposed framework allows for ranking different resources within the same match category. More recent studies [163, 150] focus on resource discovery in the SWoT, combining reasoning with quantitative contextual

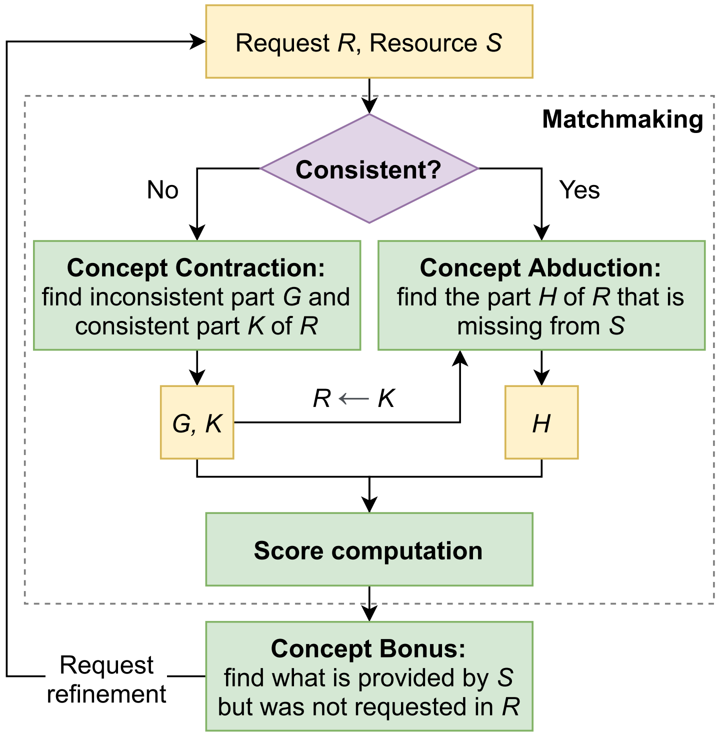


Figure 3.11: Semantic matchmaking framework.

attributes in *utility functions*.

The *semantic matchmaking* framework, as depicted in the flow chart of Figure 3.11, includes both standard and non-standard inference services as follows. Consider a set of axioms $\mathcal{T}$ in $\mathcal{ALN}$, and two concepts R, S in $\mathcal{L}$ – representing a *request* and a *resource*, respectively – both of which are satisfiable in $\mathcal{T}$. Semantic matchmaking begins with an initial **Consistency check** to determine if $R \sqcap S$ is satisfiable w.r.t. $\mathcal{T}$; formally, $\mathcal{T} \models R \sqcap S \not\sqsubseteq \bot$. If this check fails, the resource is deemed a *partial match* to the request, and **Concept Contraction** (CC) can be performed. The outcome of CC is a pair of concepts $\langle G, K \rangle$ such that $\mathcal{T} \models R \equiv G \sqcap K$, and $\mathcal{T} \models K \sqcap S \not\sqsubseteq \bot$. Essentially, CC identifies the part of R that conflicts with S. By removing only the conflicting requirements G (for *Give up*) from R, a new expression K (for *Keep*) is left, representing a contracted version of the original request. The solution G from CC clarifies "why" the conjunction of R and S is unsatisfiable, enabling the shift from a partial match to a *potential match* scenario.

3.2.3 Architecture

When designing a versatile reasoner for a wide range of use cases, several challenges must be addressed: adaptability, ease of use, maintainability, and resource efficiency. Failing to meet any of these requirements can hinder deployment in SWoE scenarios. To achieve these objectives, Tiny-ME has been developed based on the following core principles:

1. **Shared implementation across platforms**. Inference services should be implemented only once and designed to operate on various platforms. This approach simplifies maintainability, development, and ensure consistent and efficient performance across different environments.

2. **Efficiency in memory usage**. The implementation of inference services must be highly memory-efficient to enable deployment on small, resource-limited devices, ensuring effective support for useful inferences.

3. **Simple, language-specific APIs**. Applications should have access to simple, language-specific APIs to interact with the reasoner. New language APIs should be easily developable, leveraging the existing implementation of inference algorithms, ensuring flexibility and user-friendliness.

4. **Modular separation of KR and reasoning**. Knowledge representation (KR) and reasoning capabilities should be separated into different modules. KR is computationally costly due to extensive OWL data models and parser. Common Semantic Web syntaxes use memory-intensive strings, often unnecessary for reasoning tasks. Additionally, the OWL 2 specification requires support for RDF/XML [177] serialization, requiring the inclusion of XML parsers, which further increase code size and memory requirements.

High-level architecture

The comprehensive high-level architecture of Tiny-ME is shown in Figure 3.12 and explained as follows:

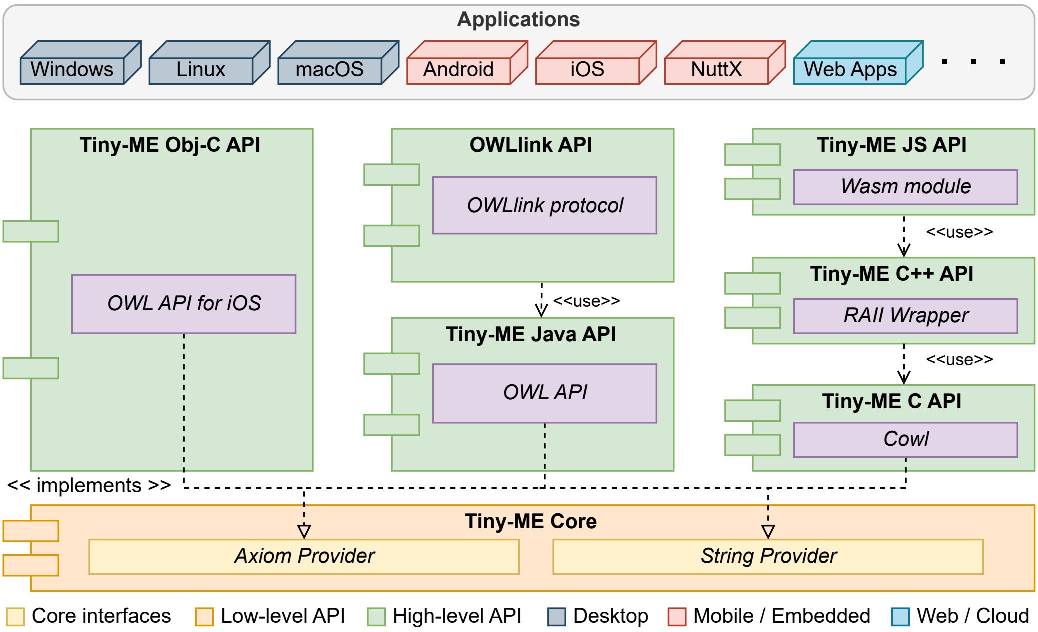


Figure 3.12: Tiny-ME high-level architecture.

- **Core layer:** designed according to the C11 standard without using compiler extensions or platform-specific APIs. This layer offers highly optimized standard and non-standard inference algorithms, along with their supporting data structures, and is designed to remain independent of how knowledge is represented or stored. $\mathcal{ALN}$ OWL entities (*i.e.*,, classes, object properties, and individuals) are represented as numerical identifiers, called *entity pointers*, and their string representations are never required during the reasoning process. The transformation from structured KB axioms to entity pointer expressions occurs via the *Axiom provider* interface, which the reasoner uses to populate its data. An optional *writer* API can perform the reverse transformation, utilizing the *String provider* interface. In a distributed SWoE architecture, some devices might not need or have the memory capacity for string representation. In such cases, they can only implement the Axiom provider API and still perform reasoning tasks effectively on (unlabeled) entities.

- **High-level APIs:** the modular architecture supports the creation

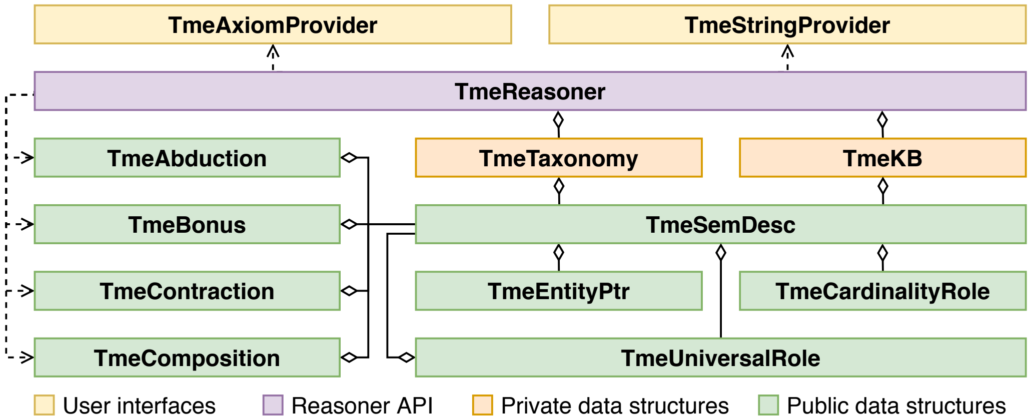


Figure 3.13: Tiny-ME core architecture.

of multiple APIs in various programming languages, as described in Section 3.2.4. Typically, the use of C11 for the reasoning core enables the possibility of many high-level APIs, primarily because most programming language runtimes offer basic interoperability with C code.

Core architecture

The reasoning core of Tiny-ME can be compiled either as a static or dynamic linking library, and it is compatible with any platform that has a C compiler available. Although the C language does not have built-in object-oriented features, highly cohesive components can be created by logically associating structured data with the functions that manipulate them, resulting in a modular and maintainable codebase. The core architecture, illustrated in Figure 3.13, comprises the following key elements:

- `TmeAxiomProvider`: this interface abstracts the retrieval of axioms from KBs. Implementors need to map $\mathcal{ALN}$ OWL class expressions to `TmeSemDesc` structures, as described later.
- `TmeStringProvider`: provides string representations of entity pointers, enabling visualization of class expressions when using the built-in writer API.

- `TmeReasoner:` carries out reasoning tasks over ontologies, such as Classification and Coherence, and offers a facade API for inference services on class expressions provided by lower-level components.

- `TmeKB:` manages KB operations, including loading and preprocessing class expressions through *lazy* unfolding and CNF normalization, preventing redundant computations [167].

- `TmeTaxonomy:` manages the concept hierarchy resulting from Classification, supporting operations like insertion, deletion, merging, and retrieval of a class's ancestors and successors.

- `TmeSemDesc:` represents $\mathcal{ALN}$ class expressions in CNF numerically. It models the conjunction of various components such as atomic classes, cardinality restrictions, and universal object property restrictions. Class expression elements are stored in vectors, with types depending on the kind of atom. Specifically, atomic classes and their negation are represented by `TmeEntityPtr`, a `typedef` for an integer type; `TmeCardinalityRole` represents unqualified number restrictions with a property identifier (`TmeEntityPtr`) and a cardinality (of integer type); universal quantifiers are represented by `TmeUniversalRole`, using an integer type for the property identifier and a pointer to the filler. Hence, the entire class description is composed only of integers, ensuring a compact memory representation and reduced computational overhead.

- `TmeAbduction, TmeContraction, TmeBonus, TmeComposition:` represent the outcomes of CA, CB, CC, and CCov, respectively. All these structures also include a penalty score, as detailed in Section 3.2.2.

3.2.4 High-level APIs

As highlighted in Section 3.2.3, the integration of Tiny-ME across various platforms and technological stacks is enabled by several high-level APIs, which are detailed in the following:

- **C:** the native interface of the system, providing access to the public API of the reasoning core by implementing the axiom and string providers through the *Cowl* library, discussed in Section 3.1. This interface is compatible with any platform that supports a C compiler, making it ideal for performance-critical and embedded software development due to C's efficiency [79].

- **Java:** tailored for Java SE and Android, this interface uses the OWL API [81] end the *Java Native Interface* (JNI)[20] to map Java classes and methods to corresponding C structures and functions. For instance, inference services are accessible through the `Reasoner` class, which encapsulates the native `TmeReasoner` structure and conforms to the OWL API's `OWLReasoner` interface. Class expressions are represented by the `SemanticDescription` class, linking to the native `TmeSemDesc` structure. This pattern is replicated for other logic constructors. Significant attention has been given to the management of native memory: Java objects that are backed by native structures are monitored by the `NativeMemoryManager`, exploiting *phantom references*[21] to track objects on the verge of being garbage-collected, allowing for the timely invocation of native deallocators.

- **Objective-C:** designed for iOS and macOS applications, axiom and string providers are implemented via the OWL API for iOS [166]. As in the Java API, class instances and methods map lower-level C structures and functions, although the wrapping logic is thinner: since Objective-C is an extension of C, it does not require additional interfaces, leading to a simpler architecture and enhanced performance, as detailed in Section 3.2.5. Memory management in this environment is also more straightforward, as *Automatic Reference Counting* (ARC)[22] and the existence of reliable finalizers[23] enable the synchronization of the lifespan

[20] Java Native Interface: `https://docs.oracle.com/en/java/javase/13/docs/specs/jni/index.html`
[21] PhantomReference: `https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/ref/PhantomReference.html`
[22] ARC: `https://clang.llvm.org/docs/AutomaticReferenceCounting.html`
[23] NSObject: `https://developer.apple.com/documentation/objectivec/nsobject`

of C allocations with their Objective-C wrappers.

- **OWLlink API**: to facilitate client-server interactions in Web environments and microservice architectures for cloud and edge computing, Tiny-ME extends the standard OWLlink protocol [115]. OWLlink provides a declarative interface for OWL reasoners, allowing the assertion of axioms in KBs and executing standard inference tasks via HTTP requests. Tiny-ME supports key reasoning services like Subsumption, Satisfiability, Classification, and Coherence checks and introduces a novel extension to the OWLlink protocol, adhering to the official protocol extension guidelines [114], to incorporate non-standard reasoning capabilities outlined in Section 3.2.2. A fork of the Java-based *OWLlink API*[24] [139] compatible with OWL API version 5 has been developed to implement the extended interface, exploiting the Tiny-Me Java API (see Figure 3.12)[25]. Currently, Tiny-ME does not support `Tell` OWLlink requests for asserting axioms to a KB, but it does support `Load Ontologies` requests for loading a KB from a URL. A *Dockerfile* is available on Tiny-ME's homepage to build a Docker[26] container featuring Tiny-ME as an OWLlink server.

- **Client-side Web API**: enables client-side reasoning in Web contexts via WebAssembly [119], making Tiny-ME runnable on all major modern desktop and mobile Web browsers. Web developers can interact with the reasoner through a simple JavaScript (JS) API, which maps the low-level native API and is generated by the *Emscripten*[27] toolchain. An object-oriented C++ API wraps the Tiny-ME core component and the Cowl library to manage and parse OWL 2 ontologies, as shown in Figure 3.15. With appropriate binding annotations and a properly configured *CMake*[28] build system, Emscripten cross-compiles the system for the WebAssembly runtime, as shown in Figure 3.14. The

[24]OWLlink adapter: `https://github.com/ignazio1977/owllink-owlapi`
[25]OWLlink matchmaking extension: `https://github.com/sisinflab-swot/owllink-matchmaking-extension`
[26]Docker: `https://www.docker.com/`
[27]Emscripten: `https://emscripten.org`
[28]CMake home: `https://cmake.org`

output is represented by a Wasm module and a JS file that can be imported into any web application, enabling the use of the reasoner as a regular JS library. The Tiny-Me C++ API is also responsible for standardizing object lifecycles using the *RAII* paradigm (Resource Acquisition Is Initialization), encapsulating C pointers in a `CPtr<T>` template type, which implements type-specific memory management logic. This is necessary because the underlying libraries use different memory management paradigms: Tiny-Me core follows the traditional `malloc/free` approach, while Cowl adopts `reference counting`. This approach simplifies dynamic memory handling by delegating it to the C++ runtime and ensures the JS binding correctly disposes of unneeded allocations. Compatibility of the JS API has been tested on the following browser/platform combinations: (i) Chrome (version 90.0.4430) and Firefox (version 88.0.1) on Windows 10 May 2020 Update, Ubuntu 20.10, macOS Big Sur and Android 10; (ii) Edge (version 88.0.705.74) on Windows 10; (iii) Safari (version 14.0) on macOS Big Sur, iPadOS 14 and iOS 14. Full compatibility has been observed in all tests. The usefulness of the client-side API was demonstrated in a Web application for semantic-based quality of experience adaptation in Web multimedia streaming [120], and in a privacy-conscious mobile web application for personalized event recommendations ranked according to the user preferences [119]. More details will be provided in Section 5.1.

3.2.5 Evaluation

This section outlines the outcomes of multiple experimental studies performed with Tiny-ME on various testbeds and inference tasks. The objective is to assess Tiny-ME's flexibility in managing different technological and resource constraints, as well as to compare its performance against other advanced reasoning systems.

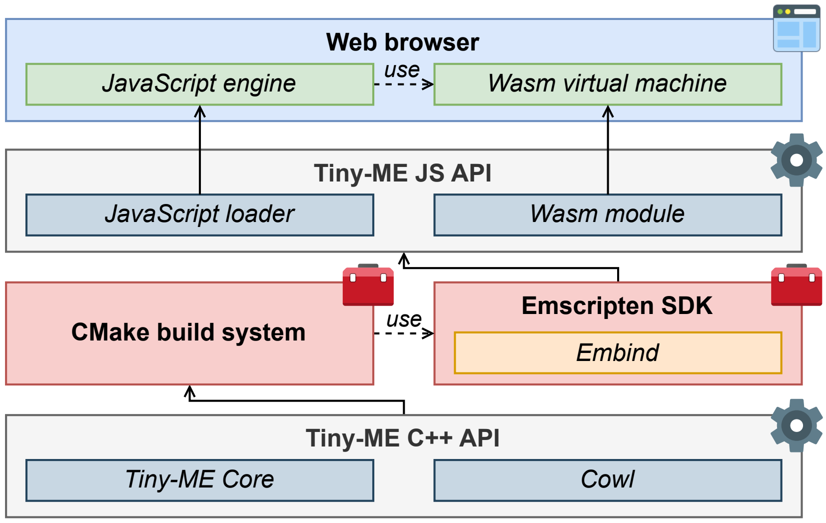


Figure 3.14: Porting workflow and architecture of Tiny-ME for WebAssembly.

3.2.5.1 PC and mobile device tests

The initial experimental campaign, as detailed in [164], was designed to evaluate the computational efficiency of Tiny-ME 1.0 for Ontology Classification and non-standard inference tasks. These evaluations were performed on both a small workstation and mobile devices using the evOWLuator framework (refer to Section 3.3). The workstation setup included an *Apple Mac Mini (2014)*[29], while mobile tests were conducted on an *Apple iPhone 7*[30] and a *HTC/Google Nexus 9* tablet[31].

The dataset for the classification tests consisted of 1364 KBs selected from the 2014 *OWL Reasoner Evaluation Workshop* competition[32]. These KBs, representing 8.24% of the total of 16555 in the DL classification corpus, were chosen based on their expressiveness, limited to $\mathcal{ALN}$. As noted above,

[29] Intel i7 4578u dual-core CPU at 3.0 GHz, 16 GB DDR3 RAM at 1600 MT/s, 1 TB HDD + 128 GB SSD (Fusion Drive), macOS Mojave 10.14.5

[30] Apple A10 CPU (2 high-performance cores at 2.34 GHz and 2 energy-efficient cores), 2 GB LPDDR4 RAM, 32 GB flash storage, iOS 10.1.1

[31] Nvidia Tegra K1 dual-core CPU at 2.3 GHz, 2 GB LPDDR3 RAM at 1600 MT/s, 32 GB flash storage, Android 7.1.1 Nougat, patch level 5 October 2017

[32] ORE 2014 corpus: `http://dl.kr.org/ore2014`

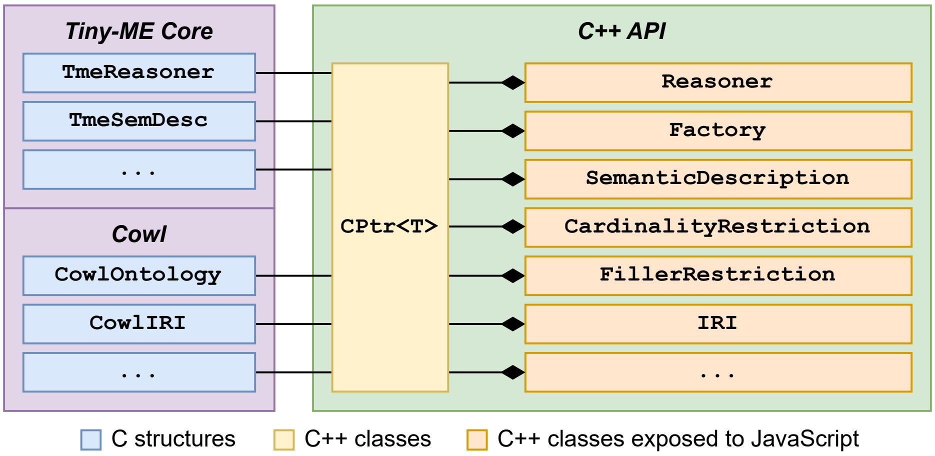
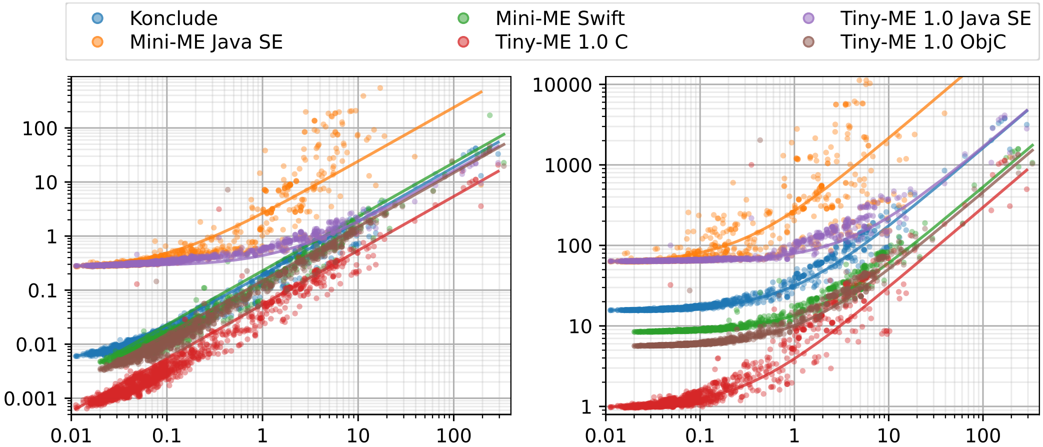


Figure 3.15: Low-level architecture of Tiny-ME for WebAssembly.

the tests were conducted on both the workstation and the mobile platforms. The correctness of Tiny-ME's outputs was verified by comparing them to the results from Konclude and Mini-ME Swift, which served as test oracles for Ontology Classification and the non-standard Matchmaking task, respectively. Tiny-ME consistently produced accurate and comprehensive inferences for all supported ontologies and reasoning tasks.

All performance metrics were averaged over five separate runs, with *Peak memory usage* measured as the *maximum resident set size* (MRSS) of the reasoner process. This was tracked using the `getrusage` POSIX call for iOS and Android, and assessed by EVOWLUATOR in desktop and workstation environments. Standard deviations for both time and memory metrics are omitted to avoid clutter in tables and graphs, as they remain consistently small (around 5% and 1% of the mean values, respectively).

Konclude was selected as the reference reasoner due to its proven reliability and superior performance in Ontology Classification, as demonstrated in previous studies [144, 167]. While the performance report does not directly compare Konclude with Tiny-ME –given the different levels of expressiveness and features of the DLs on which they are based– Konclude's performance remains a essential benchmark. However, since no other actively maintained reasoner specifically targets the $\mathcal{ALN}$ DL (with only Mini-ME allowing a



(a) Time (s) by ontology size (MiB).

(b) Memory peak (MiB) by ontology size (MiB).

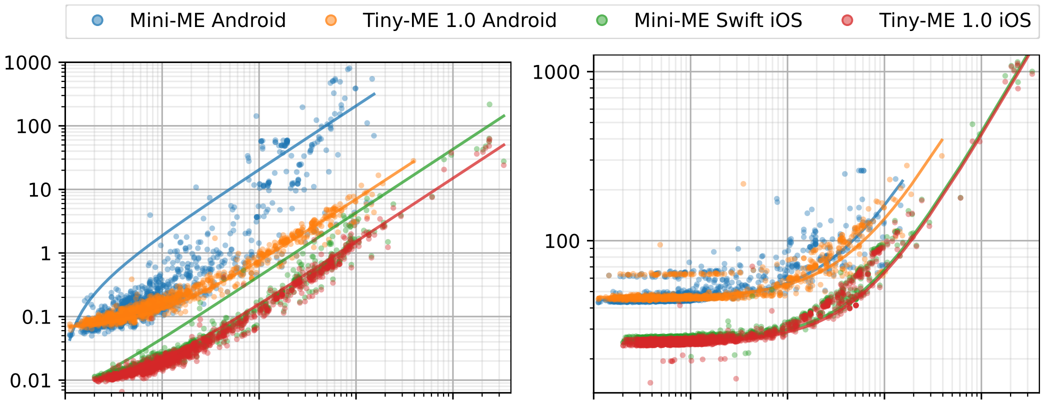
Figure 3.16: Comparison of classification time and memory peak on workstation.

fair comparison), Konclude performance must be considered when evaluating the system. Figures 3.16a and 3.16b depict the inference turnaround time and maximum memory usage relative to the ontology size. This comparison includes the interfaces for Tiny-ME C, Java SE, and Objective-C (ObjC), as well as Mini-ME 2.0 for Java SE and Mini-ME Swift for macOS.

The cumulative performance metrics presented in Table 3.3, focus on 1168 ontologies that all reasoners successfully classified within a 20 minute wall-clock timeout without runtime errors, ensuring a fair comparison of execution times. Scatterplots, on the other hand, show all data points for each reasoner.

As shown in Table 3.3, Tiny-ME C has outperformed all other reasoners in terms of both time and memory efficiency, particularly with ontologies under 1 MiB in size. This is a significant finding for resource-constrained SWoE environments.

The Objective-C API exhibits shorter processing times and lower memory consumption compared to the Java API. This is expected due to the more streamlined Objective-C wrappers, as described in Section 3.2.4. Tiny-ME ObjC and Mini-ME Swift have comparable performance, with Tiny-ME ObjC having a slight advantage due to its more efficient C core, despite both



(a) Time (s) by ontology size (MiB). **(b)** Memory peak (MiB) by ontology size (MiB).

Figure 3.17: Comparison of classification time and memory peak on mobile.

exploiting the same OWL parser and data model. Both Mini-ME 2.0 and Tiny-ME Java SE exhibit similar trends in terms of processing time and memory usage for small ontologies, but their behaviors diverge as input sizes increase (see Table 3.3). Both systems use the OWL API 3.x library for ontology parsing, which mainly influences the classification time for smaller ontologies. However, as the size of the ontology increases, the reasoning time becomes more significant, highlighting the benefits of the C core. Similar conclusions are drawn from memory peak analysis, as reported in Table 3.3.

Figure 3.17 displays classification results obtained by the Mini-ME and Tiny-ME variants on both Android and iOS platforms. The performance analysis for both mobile platforms supports the findings from the workstation tests. On iOS, reasoners have successfully processed all ontologies in the dataset within the set time limit and without encountering errors. In contrast, Android reasoners faced memory issues when classifying ontologies larger than approximately 20 MiB. This problem is evident in Figure 3.17, where data points for larger ontologies are missing on Android, and in Table 3.3, which shows lower peak memory usage for Android reasoners compared to their iOS counterparts due to memory exhaustion errors.

All Tiny-ME variants have also been tested on non-standard inference tasks using four $\mathcal{ALN}$ KBs, which are detailed in Table 3.4. Following the

Table 3.3: Dataset-wide evaluation results for classification.

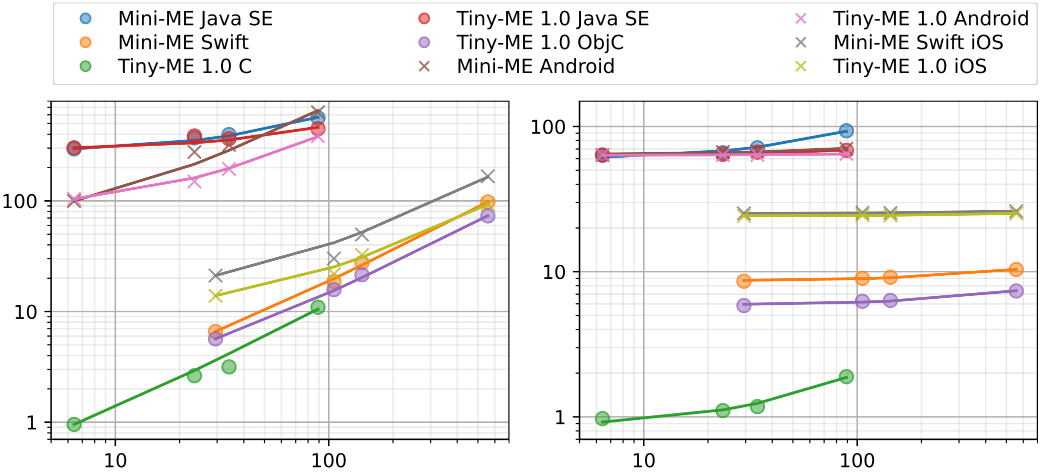
Platforms	Reasoners	Errors and Timeouts	Parsing Time (s)	Classification Time (s)	Minimum Memory Peak (MiB)	Maximum Memory Peak (MiB)
Workstation	**Konclude**	0	82.16	64.37	15.53	1559.52
	Mini-ME Java SE	196	443.01	4516.83	62.49	11237.72
	Mini-ME Swift	0	131.80	30.08	8.41	1065.92
	Tiny-ME 1.0 C	0	27.26	8.41	0.96	307.55
	Tiny-ME 1.0 Java SE	0	447.96	67.02	62.98	1408.6
	Tiny-ME 1.0 ObjC	0	131.14	12.31	5.62	1032.87
Mobile platforms	**Mini-ME Android**	237	326.71	8084.61	43.09	259.88
	Tiny-ME 1.0 Android	9	343.05	96.54	44.18	133.71
	Mini-ME Swift iOS	0	450.07	344.95	24.94	1140.18
	Tiny-ME ObjC iOS	0	446.41	113.68	23.95	1117.37

Table 3.4: Features of KBs used for non-standard inference tests.

Knowledge Base	Toy	Agriculture	Building	MatchAndDate
Size (KiB)	30.43	128.35	142.08	590.54
#concepts	48	134	180	157
#roles	8	17	27	11
#instances	7	16	29	100
#matchmaking	28	48	493	10000

methodology from [167], a matchmaking task starts with a *compatibility* check between a ⟨*request, resource*⟩ pair. If their conjunction is satisfiable, Concept Abduction is carried out; if not, Concept Contraction is applied first, followed by Concept Abduction on the contracted request. This approach aligns with the matchmaking framework described in Section 3.2.2, excluding the final Concept Bonus calculation.

Figure 3.18 summarizes the main results. It is important to note that the ontology size differ between reasoners using the OWL API for iOS and Cowl, which only support RDF/XML and functional serializations, respectively.



(a) Time (ms) by ontology size (KiB) **(b)** Memory peak (MiB) by ontology size (KiB)

Figure 3.18: Overall processing time and memory peak for the matchmaking task.

The latter is also used for reasoners using the Java OWL API. The results are consistent with the Classification tests: Tiny-ME C significantly outperforms the other reasoners, while Mini-ME Swift and Tiny-ME Obj-C demonstrate similar performance and are both more efficient than the Java-based solutions. Figure 3.18a shows shorter turnaround times on Android compared to Java SE, due to the fast mass storage installed on the Nexus 9 compared to the Mac Mini, where parsing is slower. Figure 3.18b reveals that the memory requirements for Android and Java SE versions are comparable, with slightly higher peaks on Java SE. This likely occurs because mobile devices trigger the garbage collector more frequently as a result of stricter resource management practices. In contrast, on iOS, a different pattern is observed because, as noted in [167], where iOS reasoner variants experience consistent memory overhead due to their graphical user interface, unlike other versions that can operate as command-line tools.

3.2.5.2 Client-side WebAssembly

A series of experiments has been carried out [119] to assess the performance of the WebAssembly version of the reasoner with respect to the response time for both standard and non-standard inference services. The following metrics were recorded for each test:

- **Fetching**: the time taken by the client to request and download the specified ontology. This metric depends on the device, the size of the ontology, and the network connection between the client and the server, but not on the reasoner itself.
- **Parsing**: the time required by the reasoner to convert the ontology into its internal data model.
- **Reasoning**: the time taken to perform the requested inference task.

Tests where performed on a 2021 MacBook Pro,[33] an iPhone 12 Pro[34] and a OnePlus 7T[35] to ensure a sufficiently fair and diverse representation of Web content delivery platforms in terms of both hardware and software.

The reference dataset is consistent with those used in Section 3.2.5.1, but only includes ontologies with a file size under 1 MiB in OWL 2 functional-style syntax [145]. Consequently, the classification corpus comprises 1140 ontologies, without affecting the dataset exploited for non-standard tests.

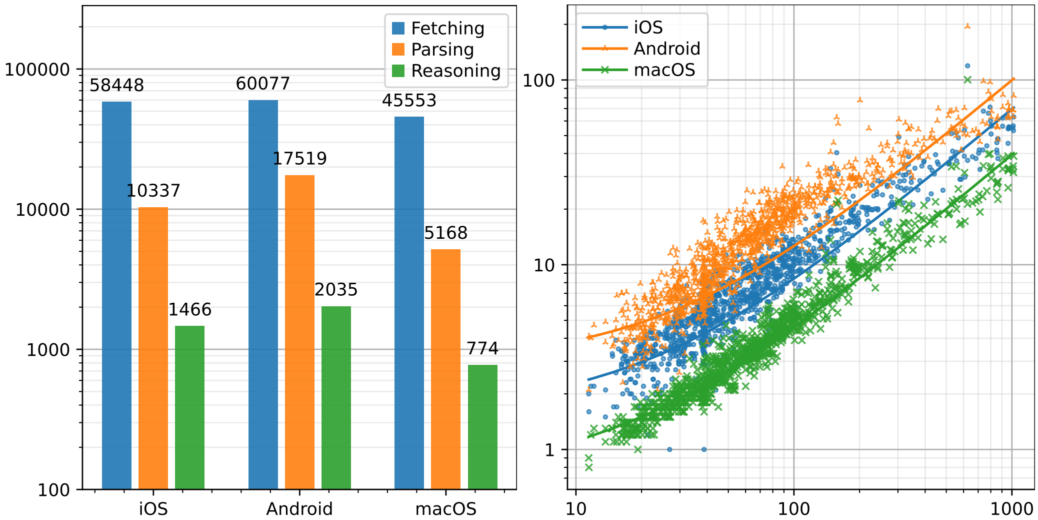
The experimental approach involves accessing a Web page containing client-side JS code responsible for executing all test tasks. For each ontology in the dataset, the script downloads the ontology document and then uses the Tiny-ME JS API to execute the specified inference. The results are stored in a Comma-Separated Values (CSV) file, which is subsequently processed by the EVOWLUATOR framework to generate visualizations (see Section 3.3.2.3). The test Web pages and ontologies are served by an *NGINX*[36] instance

[33]Apple M1 Max SoC, 8 performance cores @3.2GHz and 2 efficiency cores @2.0GHz, 64 GB UM RAM, 1 TB SSD, macOS Ventura 13.3.1, Safari 16.4 browser.

[34]Apple A14 Bionic, 2 high-power cores @3.1 GHz and 4 low-power cores @1.8 GHz, 6 GB LPDDR4X RAM running at 4266 MT/s, 128 GB NVMe SSD, iOS 15.1, Safari Mobile 15.1 browser.

[35]Qualcomm Snapdragon 855+, 1 core @2.96 GHz, 3 cores @2.42 GHz and 4 cores @1.8 GHz, 8 GB LPDDR4X RAM, 128 GB UFS 3.0 storage, Android 12, Chrome 112.0.5615.47 browser

[36]`https://www.nginx.com/`



(a) Cumulative Reasoning Time (*ms*). **(b)** Reasoning Time (*ms*) by ontology size (*KiB*).

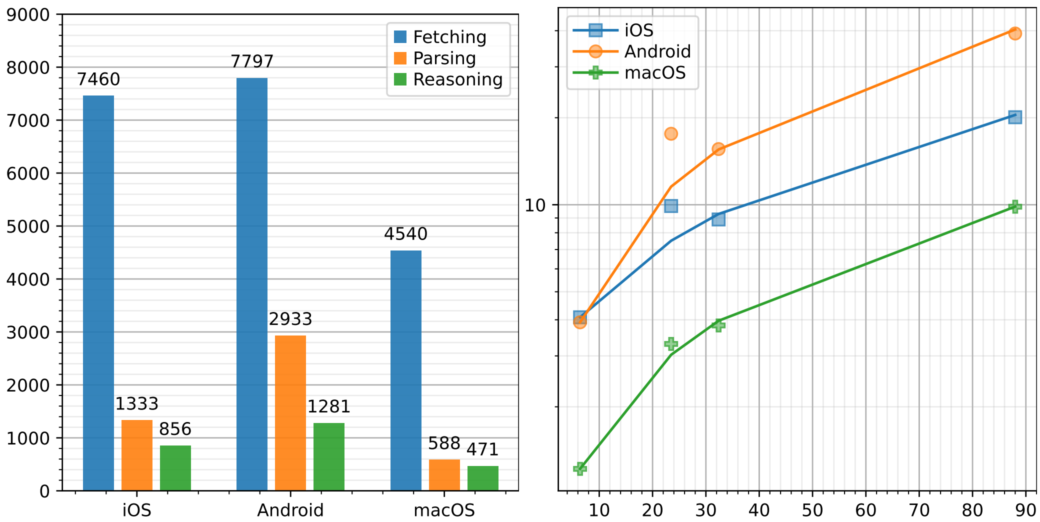
Figure 3.19: Ontology classification results for the JS API.

running on a desktop computer,[37] configured to handle all requests without caching. All devices are connected to the same IEEE 802.11n WLAN.[38]

Times are recorded using the `performance.now()`[39] JS API function, which typically provides timestamps with a precision of 1-2 milliseconds to prevent timing attacks and fingerprinting. While this precision is sufficient for measuring fetching and parsing operations, it is often too imprecise for inference tasks. To address this, reasoning times were calculated by executing multiple consecutive iterations of each task (10 for the two mobile devices and 50 for the laptop). The combined fetching and parsing times were subtracted from the total execution time, and the results were averaged over all iterations.

Figure 3.19 displays the results of the standard *ontology classification* inference service. The bar chart in Figure 3.19a indicates the overall time needed for fetching, parsing, and reasoning across all tested ontologies. The scatter plot in Figure 3.19b demonstrates the correlation between the ontology

[37]Intel Core i7-3770k CPU, 4 cores @3.5GHz, 12 GB DDR3 RAM @1600 MT/s, 2 TB SATA SSD, Windows 10.
[38]Hosted by a TP-Link TN-WR841N router.
[39]`https://developer.mozilla.org/en-US/docs/Web/API/Performance/now`



(a) Cumulative Reasoning Time (ms). **(b)** Reasoning Time (ms) by ontology size (KiB).

Figure 3.20: Semantic matchmaking results for the JS API.

size and the total parsing and reasoning time, with the least capable device requiring up to 200 ms in the worst scenario. The results showcase the satisfactory performance of inference services geared towards ontology management, which is crucial for interactive semantic-enabled Web applications in mobile environments. Although the fetching time is the most significant factor, the domain ontology can generally be fetched once and then cached to enhance the application responsiveness.

Figure 3.20 illustrates the performance metrics for the *matchmaking* non-standard service. The Wasm reasoner executes this inference within a few milliseconds, with 40 ms being the maximum time recorded for the largest ontology in the corpus. These results align with those observed in Ontology Classification and they are particularly significant as non-standard inferences are frequently more applicable in SWoE scenarios, which Tiny-ME id specifically designed to handle.

An preliminary memory usage test for matchmaking has been conducted on the aforementioned MacBook Pro, exploiting the snapshotting features available in the Mozilla Firefox browser developer tools [120]. Typically, the Emscripten compiler allocates a fixed 16 MiB block of memory for the Wasm module, which obscures the actual memory requirements for the matchmaking task. As a result, the starting memory allocation was reduced to 192 KiB, the smallest size that allows code compilation, and the Wasm module was recompiled with the `ALLOW_MEMORY_GROWTH` option[40] enabled, which allows the browser to allocate additional memory as needed. Based on this setup, the following memory snapshots were captured: (i) *Baseline:* a blank page before loading the Wasm module; (ii) *Wasm module:* after loading the Wasm module; (iii) *Runtime:* memory usage peak while perfroming matchmaking.

Loading the Wasm module results in a 1.16 MiB memory increase over the baseline, which includes the Wasm memory buffer initially sized at 192 KiB. During matchmaking, an additional 0.31 MiB of memory is allocated, and the browser dynamically resizes the Wasm memory buffer to 320 KiB. These results indicate an acceptable memory overhead for the matchmaking process, validating the practicality of the suggested approach.

Overall, this thorough evaluation suggests that the JS API of the system can be effectively integrated into interactive web applications, providing both standard and specialized client-side inference services without compromising performance or user experience.

3.3 evOWLuator: a multiplatform benchmarking framework for OWL reasoners

This section introduces evOWLuator [178], a cross-platform framework designed to assess OWL reasoners. Distinguished by its high flexibility, expandability, and scalability, it stands out due to its unique architectural features. evOWLuator operates on GNU/Linux, macOS, and Windows (via the *Windows Subsystem for Linux*, WSL), supports any ontology corpus, and can evaluate standard reasoning tasks such as ontology consistency and classification, along with more specialized ones like semantic matchmaking.

[40] https://emscripten.org/docs/optimizing/Optimizing-Code.html

Furthermore, users can extend the range of supported inference services through a plug-in system.

One of the main attributes of the framework is its capability to execute tests either locally or on remote devices, facilitating integration with mobile and embedded platforms, as shown in the evaluation campaigns detailed in the previous sections. The tool is proficient in assessing various reasoning metrics, such as correctness, turnaround time, memory usage, and energy consumption, a pioneering effort in the OWL benchmarking field. evOWLuator's modular architecture enhances its versatility by enabling the addition of new target reasoners, platforms, and reasoning tasks. It can also create interactive visualizations of results with highly customizable plots, making it an invaluable asset for research endeavors. To encourage its adoption in both academic and industrial settings, its source code[41] is provided under a highly permissive, commercially-friendly license.[42]

The rest of the paragraph is structured as follows: Section 3.3.1 covers the state of the art; Section 3.3.2 illustrates how evOWLuator can be used to perform experiments and visualize outcomes; Section 3.3.3 outlines the framework's architecture, with customizable interfaces detailed in Section 3.3.4; lastly, Section 3.3.5 presents the results of a small experimental study comparing six OWL reasoners, showcasing the tool's and approach's effectiveness.

3.3.1 Background

When searching for the most suitable tool for a specific application, it is essential to go beyond functional requirements and platform compatibility by performing a thorough, data-driven analysis of performance and scalability. The approach helps to avoid strategic errors and ensures that the software components are suitable for the intended platforms and features [227]. In the domain of Semantic Web technologies, this has led to the development of

[41] evOWLuator source code: `https://github.com/sisinflab-swot/evowluator`
[42] Eclipse Public License 2.0: `https://www.eclipse.org/legal/epl-2.0/`

numerous OWL reasoning benchmarks and automated evaluation frameworks. Selecting the right evaluation tool is itself a challenging task, influenced by factors such as the types of performance metrics provided, platform compatibility, supported inference services, ease of reasoner integration, and test automation abilities, among others. Furthermore, the recent surge in mobile and pervasive computing has expanded the application of knowledge representation and reasoning to new and unconventional contexts.

Before 2010, reasoner evaluations relied primarily on benchmarks with a limited number of ontologies and small sets of manually created queries [85]. A notable case is the *Lehigh University Benchmark (LUBM)* [77], known for its focus on a single university-related ontology, featuring fourteen extensional queries and a data generator for scalable ABoxes. LUBM, along with three other ontologies and query sets across four OWL fragments, was used in [31] to evaluate five reasoners, identifying the most suitable ones for each ontology class and inference task. Likewise, [54] analyzed eight reasoners using three large ontologies from the OWL 2 EL profile, focusing on classification, consistency, concept satisfiability, and subsumption tests. Furthermore, [138] enhanced LUBM to support SPARQL-based stream reasoning. Recent advances in the availability of knowledge bases and graphs have led to the development of larger and more diverse datasets for OWL reasoner benchmarking, aiming to evaluate systems in practical scenarios. A pioneering 2012 study [89] evaluated the classification time of four reasoners using a dataset of more than 300 real-world ontologies, including ontology metrics as machine learning features to predict processing time. The *OWL Reasoner Evaluation* (ORE) workshop series, held annually from 2012 to 2016, expanded the scope with additional ontologies, reasoning tasks, and participants [69, 144]. Reasoners were scored based on the number of problems solved in each competition track, with time serving as the tiebreaker for the final rankings. Unfortunately, other performance metrics such as memory or energy consumption were not considered.

The increase in datasets and test cases underscored the need for automated benchmarking tools. A key solution for traditional computing platforms was the framework from [144]. Before starting the evOWLuator project, there was consideration of adapting this framework to include additional metrics and platforms. However, it was ultimately deemed too complex. Designed primarily for live competitions and ensuring inference correctness, the framework required extensive modifications, complicating an already large project with over 200 Java files and 15000 lines of code, excluding the Web interface. It was not built for easy expansion or the inclusion of features like memory and energy assessment or mobile platform support. In contrast, evOWLuator provides a more streamlined and flexible solution, comprising about 4,000 lines of code across 40 Python files. It supports various customizations through plug-ins without having to modify the existing codebase.

There are specialized frameworks available for evaluating non-standard inferences. For instance, *JustBench* [17] is tailored to assess reasoner performance in verifying *justifications*, which are minimal ontology subsets required for an entailment. Performance evaluations for Mini-ME on Java/Android [183] and its reengineered Swift version [167] for iOS focused on the matchmaking task. As the use of semantic technologies in ubiquitous computing grows, there is an increasing emphasis on benchmarking mobile-specific OWL profiles and developing new mobile reasoners. An important study [30] assessed six reasoners with OWL API [81] support on Android, using the ORE 2013 dataset [69], focusing on ontology classification and consistency. To simplify the testing process, an Android app was developed, allowing users to select the reasoner, ontology set (based on OWL profile sublanguages), and inference task, with results stored in an embedded database.

Mobile and ubiquitous computing are characterized by platform heterogeneity and the need for strict energy consumption management, driving the development of cross-platform, energy-aware benchmarking frameworks. A framework discussed in [210] was specifically designed to evaluate mobile semantic rule engines and was developed in JavaScript using the *PhoneGap*[43]

[43] PhoneGap home: `https://phonegap.com/`. It is based on the open source *Apache Cordova* engine: `https://cordova.apache.org/`

Software Development Kit (SDK). This approach allowed for the use of rule engines written in JavaScript or natively for any platform supported by PhoneGap, including Android, iOS, and Windows 8.1. An enhanced version of this framework, called *MobiBench* [209], also includes support for OWL 2 RL reasoning, benchmark automation, and Java reasoners through the *Nashorn* JavaScript engine bundled with Java SE 8 and later versions. Future work aims to incorporate energy usage profiling. While energy-aware mobile benchmarks exist, they are typically developed for specific research projects. The study in [148] evaluated Android reasoners by using a hardware power monitor in place of the device's battery to achieve precise measurements. However, this approach is complex due to the varying electrical parameters across different mobile models and the prevalence of non-removable batteries in modern smartphones. Alternatively, [102] used an *ODROID XU3* single-board computer with built-in power monitoring circuitry to test six reasoners, offering a more practical approach than [148], though it did not fully reflect typical mobile and ubiquitous computing environments and focused solely on Java-based reasoners. A software-centric approach was taken in [57], where a profiler used Android APIs to correlate battery charge and time metrics, achieving accuracy within 5% of hardware monitors for apps with minimal network or sensor usage. This method was also applied in [207] to compare the energy consumption of Android reasoners. Similarly, [76] developed a model predicting energy consumption for mobile ontology reasoning, revealing that energy use is influenced by battery charge levels and that the relationship between task duration and power consumption is not always linear These findings highlight the need for a scalable evaluation framework that efficiently adapts to different mobile devices and platforms, offering comprehensive insights into energy consumption in mobile reasoning tasks.

3.3.2 How to use evOWLuator

This section provides guidance on how to configure and use evOWLuator to perform evaluations and visualize results. Installation instructions and additional technical details, including command-line options and flags, are available in the online documentation[44]. Once the tool is installed and configured correctly, it can be used by calling the `evowluate` command line utility, followed by a *subcommand* specifying the task to be executed by the framework, as shown in Figure 3.21. The following sections introduce the available subcommands.

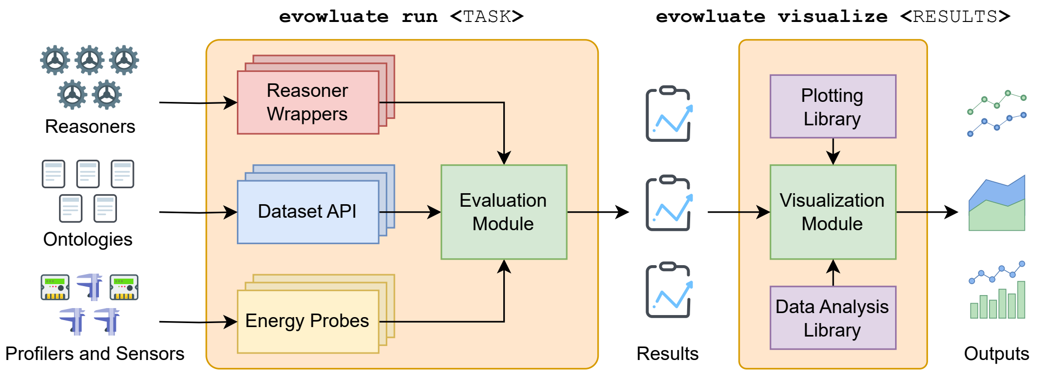


Figure 3.21: EVOWLUATOR high-level architecture and data flow.

3.3.2.1 Setup

After installing the framework, some initial setup is necessary before running an evaluation:

- **Datasets:** ontology corpora should be placed in the `data` directory[45]. Each dataset must be stored in a root folder, named appropriately so that EVOWLUATOR recognizes it as the dataset name, with subfolders for each supported syntax[46]. These subfolders should contain ontologies serialized in their respective syntaxes.

- **Reasoners, reasoning tasks, energy probes:** configure these by adding Python modules that implement the `Reasoner`, `ReasoningTask`,

[44] EVOWLUATOR documentation: `http://swot.sisinflab.poliba.it/evowluator`

[45] All filesystem paths are relative to the EVOWLUATOR root directory.

[46] Supported syntaxes include: `dl`, `functional`, `krss`, `krss2`, `manchester`, `obo`, `owlxml`, `rdfxml`, `turtle`.

and `EnergyProbe` interfaces into the `evowluator/user/reasoners`, `evowluator/user/tasks`, and `evowluator/user/probes` directories, respectively.

Missing dataset serializations can be automatically generated by evOWLuator using the `convert` subcommand, which allows the conversion of datasets into any format supported by the framework. Additional information on the plugin mechanism for reasoners, tasks, and energy probes is available in Section 3.3.4.

3.3.2.2 Running evaluations

Once all necessary components are configured, evaluations can be initiated using the `run` subcommand, specifying the reasoning task to be evaluated and any required parameters, such as the dataset. evOWLuator provides built-in support for various tasks, including standard ones like *classification* and *consistency*, as well as non-standard tasks like *matchmaking*. Custom reasoning tasks can be seamlessly integrated by implementing the `ReasoningTask` interface, as detailed in Section 3.3.4. Each inference task supports two evaluation modes:

- **Correctness**: verifies the validity of inference results. Users can employ a single reasoner as a correctness oracle, evaluate correctness through consensus achieved by a randomized majority vote, or assume all reasoners provide correct results and only consider runtime errors or timeouts. In this evaluation mode, reasoner outputs are collected instead of merely correctness results, allowing the correctness strategy to be adjusted *a posteriori* during result visualization. For reasoning tasks that generate extensive outputs, such as ontology classification, a hash of the output is maintained.
- **Performance**: gathers metrics like inference time, peak memory consumption, and optionally, energy consumption if one or more *energy*

probes are defined using command-line flags. The framework can capture and display multiple time recordings from each reasoner, corresponding to different reasoning stages (e.g., parsing, preprocessing, reasoning, etc.).

Test execution can be managed using various flags that allow users to customize different aspects of the evaluation process: (i) evaluations are, by default, conducted for all reasoners capable of handling the specified reasoning task, but users can limit the evaluation to specific reasoners if needed; (ii) a timeout can be set for inferences, causing the termination of the reasoner process if exceeded; (iii) performance tests can be executed multiple times, with the framework averaging the results for visualizations; (iv) correctness tests can be run in parallel by specifying the number of worker processes.

Evaluations can be interrupted by sending the `SIGINT` POSIX signal to the framework, *e.g.*, by hitting `CTRL+C` in the terminal. Incomplete evaluations can be resumed using the `resume` subcommand. Once a test is complete, the framework generates and saves the following items in a new folder within the `results` directory: (i) a human-readable *log* of the assessment; (ii) a summary of the *configuration* used for the evaluation, including selected reasoners, dataset, syntaxes, *etc.*; (iii) machine-processable test *outcomes*, with content and structure depending on the configuration.

3.3.2.3 Visualizing results

In addition to providing raw evaluation results, evOWLuator can also create summary reports and visual graphs. The type of visualizations available depends on the test configuration:

- For **correctness** tests, the tool provides statistics about the number of accurate and inaccurate results, runtime errors, and timeouts. These metrics are displayed in a grouped bar plot (*e.g.*, Figure 3.25).

- For **performance** evaluations, the tool computes times per ontology and across the entire dataset, along with the minimum, maximum, and average memory peaks detected for each reasoner. If energy probes are included, it also summarizes energy consumption metrics. The visualizations include grouped bar charts for cumulative results (*e.g.*, Figure 3.26a, Figure 3.26c) and scatterplots showing the evaluated metrics relative to ontology size (*e.g.*, Figure 3.26b, Figure 3.26d).

The plots are displayed in an interactive window that allows users to navigate, zoom, and crop. They can be saved as vector or raster graphics files. Additionally, various plot features can be adjusted via specific command-line arguments, such as plot dimensions, titles, labels, axis scales and limits, time and memory units, legend placement and format, marker and bar types and colors, polyline fitting for scatterplots, and the degree of the fitted polynomial.

3.3.3 Architecture

evOWLuator is designed with flexibility in mind, particularly for testing different reasoning engines and running inference services on mobile and embedded devices. The framework uses an *object-oriented* approach, where users configure it by extending Python's abstract base classes with custom subclasses that implement the required interfaces. While this *programmatic* method may be more detailed compared to a *declarative* approach, such as using structured configuration files, it offers greater expressiveness by allowing users to fully leverage Python and its standard library. This approach significantly enhances the framework's adaptability and future-proofing, making it versatile enough to integrate with any reasoner interfaces.

The core components of the framework are described below and illustrated in the Unified Modeling Language (UML) component diagram in Figure 3.22. The *Data* module provides the `Dataset` and `Ontology` classes, which simplify access to user-provided datasets and the ontologies they contain. The *Evalua-*

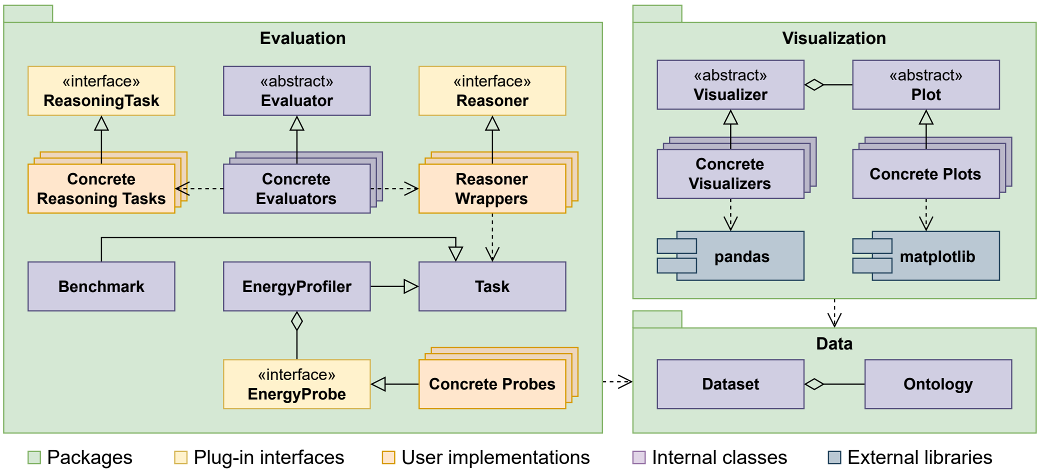


Figure 3.22: UML diagram of the main components of EVOWLUATOR.

tion engine interfaces with reasoners, ontologies, and energy profilers through customizable software endpoints. Most functionalities are contained within *Evaluators*, which are subclasses of the `Evaluator` abstract class responsible for implementing the primary business logic for different evaluation types, such as correctness or performance. Inference task details are represented by `ReasoningTask` and implemented through concrete classes that adhere to this interface. The `Task` class acts as the main API for initiating processes and collecting their output: its subclasses enable energy consumption profiling and the benchmarking of execution times and memory usage.

The *Evaluation Engine* component triggers inference tasks executed by reasoners through user-defined subclasses of the `Reasoner` abstract class. This class allows tests to be run both locally and on remote devices, with particular support for mobile and embedded platforms like iOS and Android (refer to Section 3.3.4). For performance and energy consumption evaluations, the engine can execute multiple test iterations as specified by command line arguments. If more than one iteration is performed, the framework averages the results to provide a unified output for further analysis. In addition, users can also specify a *timeout* for each reasoning task, after which the reasoner process is automatically stopped. EVOWLUATOR is designed to detect *runtime errors* in two cases: the reasoner exits with a non-zero code, or it fails to produce necessary output such as computation times for

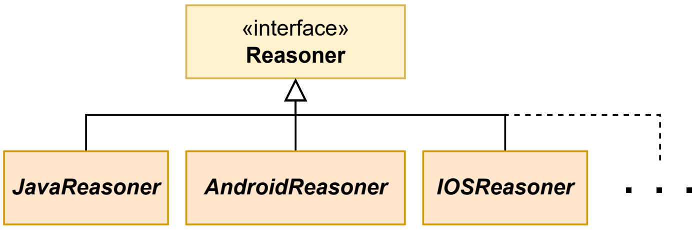


Figure 3.23: *Reasoner* interface and template classes.

performance evaluations or inference results for correctness checks. Once a test is complete, the *Evaluation Engine* logs the results, which can then be used for visualization. The *Visualization Engine* can generate tabular summaries and graphical representations, providing clear recaps of the raw data. This component includes subclasses of the `Visualizer` abstract class, with charting and plotting features provided by subclasses of the `Plot` abstract class.

3.3.4 Available interfaces

Reasoners

To integrate with evOWLuator, reasoners must have a command line interface for executing reasoning tasks on specific ontologies, requiring inputs to specify at minimum the reasoning task and the ontology file path. The details of these arguments depend on the reasoner and are defined in the implementation of the `Reasoner` interface. The framework dynamically loads all the subclasses of the `Reasoner` abstract class, which must provide metadata (*e.g.*, name, executable path, supported OWL syntaxes) and command line arguments required for each supported reasoning task. For correctness results, reasoner command line tools should either follow the output format expected by evOWLuator or provide a custom output parsing implementation by subclassing the `ResultsParser` class. This approach allows the framework to accommodate a wide range of reasoners and their output formats.

Alongside the `Reasoner` base class, the proposed framework offers several templates to facilitate the integration of inference engines across major platforms, as depicted in Figure 3.23 and described below.

- **Java SE:** this template simplifies the process of integrating Java-based reasoning engines packaged in *jar* files by managing the initialization of the *Java Virtual Machine* (JVM). The configuration of the JVM is handled through specific flags set with designated methods.
- **iOS:** the template supports the execution and testing of iOS reasoners by encapsulating inference engines within *Xcode* projects as `XCTestCase`[47] subclasses. Reasoning tasks are defined as test case methods, which are deployed to the target device along with the necessary evaluation datasets. evOWLuator triggers test cases via `xcodebuild`, the command line interface for Xcode, providing any required data through environment variables. Users only need to supply relevant project details, such as the Xcode project path and the names of the test methods implementing each supported reasoning task.
- **Android:** the template enables the execution and testing of reasoning tasks on Android devices. Users must provide Android-specific details, such as the reasoner app's package identifier, and install reasoners as Android wrapper applications that include an `EVOWLUATE` intent filter. evOWLuator automatically sets up a *launcher* application to activate and close reasoner apps by sending the appropriate `EVOWLUATE` intents[48], leveraging the Android `Instrumentation` class[49]. Communication between evOWLuator and the launcher application occur via the *Android Debug Bridge* (`adb`), which must be installed on the host machine. *USB debugging* must also be enabled on the target mobile device by accessing the hidden *developer menu*[50].

[47] XCTest: `https://developer.apple.com/documentation/xctest`
[48] Android intents: `https://developer.android.com/reference/android/content/Intent`
[49] Android Instrumentation: `https://developer.android.com/reference/android/app/Instrumentation`
[50] Android developer menu: `https://developer.android.com/studio/debug/dev-options`

Reasoning tasks

The `ReasoningTask` interface defines the specific requirements for each reasoning task, allowing the specification of expected inputs and outputs for various inference services. For example, tasks such as ontology classification and consistency require only a single input file (the ontology to be processed), while other inferences, such as matchmaking, may need additional inputs, such as an OWL ontology with individual requests. This interface also manages the expected outputs from the reasoner, as detailed below:

- in terms of **correctness**, results can be output as text files, standard output, or ontology documents. If ontology documents are used, they must be valid OWL ontologies in any serialization format supported by the OWL API [81]. For example, reasoners should produce results on the standard output for the consistency task, while they should deliver the inferred taxonomy as an OWL ontology for the classification task;
- regarding **performance**, the interface specifies which results reasoners are expected to produce on the standard output, *i.e.*, reasoning phases along with their respective time measurements. It also allows for the specification of memory usage data, which is essential for reasoning tasks on remote devices. For local reasoners, the framework automatically calculates the memory usage metric as the *maximum resident set size* (MRSS) of the reasoner process, using the cross-platform `psutil`[51] Python library. Remote reasoners, however, must determine this metric themselves, typically using platform-specific methods, such as the `getrusage`[52] POSIX call for iOS and Android devices.

Energy footprint

The `EnergyProfiler` class estimates energy usage by running the reasoner and continuously querying a specified *energy probe* instance while the corresponding process is active. Energy probes are required to implement the

[51]Process and system utilities (psutil): `https://psutil.readthedocs.io`
[52]`getrusage` man page: `https://linux.die.net/man/2/getrusage`

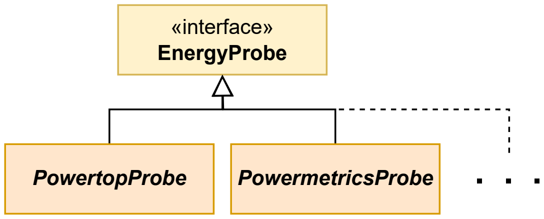


Figure 3.24: *EnergyProbe* interface and built-in probes.

`EnergyProbe` interface, as depicted in Figure 3.24. The collected N samples are then used to compute an *energy footprint* score, which represents the estimated energy consumption of the engine during its operation:

$$score = sampling\_interval * \sum_{i=1}^{N} sample_i \tag{3.1}$$

The method is designed for broad compatibility with both software- and hardware-based energy measurement solutions. `EnergyProbe` instances can interact with the operating system's built-in power management tools, use data from energy profilers (as discussed in Section 3.3.1), or even incorporate readings from external hardware, such as the Monsoon High Voltage Power Monitor[53]. EVOWLUATOR inherently supports *PowerMetrics*[54] (developed by Apple Inc. for macOS) and *PowerTOP*[55] (created by Intel Corp. for GNU/Linux and also usable on Microsoft Windows via WSL) software-based energy profilers. To add additional energy probes, users can define classes that implement the `EnergyProbe` interface, which is responsible for calculating and recording power usage samples to reflect average energy consumption between consecutive polls. Energy consumption is reported as an absolute *score* without any specific measurement unit, ensuring compatibility with some energy profilers like PowerMetrics that do not provide power usage in standard physical units. However, when power samples are measured in watts, as per Equation 3.1, the score can be interpreted as energy consumption in

[53]Monsoon HV Monitor: `https://www.msoon.com/high-voltage-power-monitor`

[54]PowerMetrics: `https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx`

[55]PowerTOP: `https://www.intel.com/content/www/us/en/developer/articles/tool/powertop-primer.html`

joules. This understanding extends to tools such as PowerTOP and potentially to hardware-based power meters.

For battery-powered devices, it is essential to maintain consistent power sources (either grid or battery) and the same charge level at the beginning of each test. Fluctuations in these conditions can influence energy measurements, thus affecting the reproducibility of the tests. To perform energy comparisons across different devices or operating systems, the framework should ideally be used alongside an external hardware power meter. Although this approach offers the most precise, device-agnostic energy readings, it is crucial to carefully interpret the results, keeping in mind the potential biases or confounding factors arising from the hardware and software differences among the devices being tested.

3.3.5 Case study: benchmarking classification and consistency

To validate evOWLuator's functionality and demonstrate its utility, a targeted experimental campaign has been conducted. This campaign assessed the accuracy, performance, and energy consumption of ontology classification and consistency inference services provided by selected advanced OWL reasoners, tested on both desktop and mobile devices. In order to evaluate the framework's benchmarking capabilities, the campaign has been purposefully limited in scope, involving a smaller selection of reasoners and datasets compared to the larger-scale tests mentioned in Section 3.3.1. This strategy enables a focused evaluation of evOWLuator's core features and efficiency in a controlled environment.

3.3.5.1 Testbed, reasoners and datasets

Desktop evaluations have been conducted on a *Linux* machine[56] and a *Mac Mini (2014)*[57], while mobile tests have been performed on an *Apple iPhone 7*[58] and a *HTC/Google Nexus 9* tablet[59]. Tested desktop reasoners include: *Fact++* (version 1.6.5) [206], *HermiT* (1.3.8) [67], *Konclude* (0.6.2-544) [196], *Mini-ME* (2.0) [183], *Mini-ME Swift* (1.0) [167] and *TrOWL* (1.5) [204]. Mini-ME and Mini-ME Swift have also been employed for tests on Android and iOS, respectively. In addition, the *JFact* (1.2.1), *HermiT* (1.3.8) and *Pellet* (2.3.1) [191] Android versions from [30] have been evaluated. Tests have been carried out on the following datasets:

- **ORE 2014**: a total of 1398 ontologies were chosen from the *2014 OWL Reasoner Evaluation Workshop* competition dataset[60], selecting only those with a reference expressiveness of no more than $\mathcal{ALN}$. This ensures compatibility with Mini-ME, which is not capable of handling more expressive DL.
- **ORE 2014 Energy**: the set includes the 50 largest ontologies from the ORE 2014 collection (average size 1781.28 ± 14063.8 KiB, ranging from 11.44 KiB to 291.33 MiB in functional syntax), which were specifically chosen for energy consumption testing on macOS. Large ontologies are prioritized because they ensure sufficiently long-running processes to obtain accurate measurements with software energy profilers such as *PowerMetrics* and *PowerTOP*. These profilers typically cannot capture data from brief processes that end before power sampling can begin, even if those processes are very energy-demanding. By using this selective

[56] AMD Ryzen 5 3600 CPU at 3.6 GHz, 32 GB DDR4 RAM at 3000 MT/s, 1 TB SSD, Ubuntu 18.04 LTS x64.
[57] Intel i7 4578u dual-core CPU at 3.0 GHz, 16 GB DDR3 RAM at 1600 MT/s, 1 TB HDD + 128 GB SSD (Fusion Drive), macOS Mojave 10.14.5.
[58] Apple A10 CPU (2 high-performance cores at 2.34 GHz and 2 low-energy cores), 2 GB LPDDR4 RAM, 32 GB flash storage, iOS 10.1.1
[59] Nvidia Tegra K1 dual-core CPU at 2.3 GHz, 2 GB LPDDR3 RAM at 1600 MT/s, 32 GB flash memory, Android 7.1.1 Nougat, patch level 5 October 2017
[60] `http://dl.kr.org/ore2014`

approach, the limitations of the profilers are addressed, leading to reliable energy usage readings for longer-duration tasks.

- **BioPortal**: a collection of 20 ontologies from BioPortal[61] [219] without limitations on DL expressiveness (average size 130.79 ± 104.57 MiB, ranging from 27.92 MiB to 450.71 MiB in functional syntax). BioPortal has been selected due to the significant size and popularity of life sciences ontologies within the Semantic Web community. The selection process involved identifying the 100 largest ontologies in BioPortal, classifying them using four high-expressiveness reasoners (Konclude, Fact++, HermiT, and TrOWL), and retaining only those where all reasoners provided accurate and complete inferences within 2 hours. This data set has been used to showcase energy profiling capabilities on both Linux and macOS platforms.

3.3.5.2 Setup

The experimental setup followed the procedure described in detail in the EVOWLUATOR documentation.

Datasets

After installing EVOWLUATOR on both desktop computers within the `evowluator` base directory, the relevant datasets have been organized by placing the ontologies into the appropriate subdirectories of the `data` folder within the installation path. Each dataset should have a root directory named as its identifier, along with a subdirectory for each supported OWL syntax, containing the ontology files. Any missing ontology formats required by some reasoners have been generated using the `convert` subcommand[62].

[61] `https://bioportal.bioontology.org`

[62] The ORE 2014 dataset is exclusively available in functional OWL syntax, although Mini-ME Swift requires ontologies in RDF/XML format.

Desktop reasoners

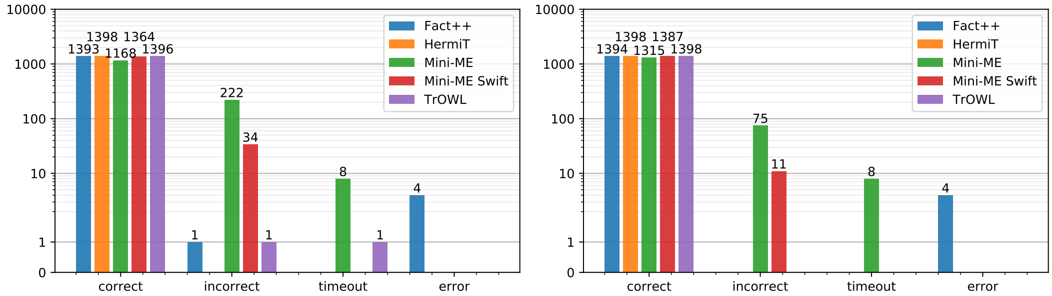
The next phase involved the integration of reasoners, which, as detailed in Section 3.3.4, requires creating Python modules that implement the `Reasoner` interface. A wrapper has been developed for Java-based reasoners that exploit the OWL API to handle classification and consistency inference tasks, ensuring a standardized command line interface across all reasoners, including Fact++, HermiT, TrOWL, and Mini-ME. Integration on the Python side was achieved by developing a generic `Reasoner` template subclass for all OWL API-based reasoners, which was further extended to include specific reasoner metadata. Since Mini-ME Swift and Konclude already have a built-in command line interface, integration was accomplished by simply creating appropriate `Reasoner` subclasses.

iOS reasoners

On the host machine, after installing Xcode and its command line tools, an `XCTest` project must be created, with reasoning task calls encapsulated within distinct methods of an `XCTestCase` subclass to support iOS reasoners like Mini-ME Swift. To integrate the Python component, the `IOSReasoner` template class has been subclassed, providing the relevant metadata. Ontologies are transferred to the target device using the *copy bundle resources* phase during the Xcode build process.

Android reasoners

The host computer must have the `adb` tool installed to enable communication with the connected Android device. Simple wrapper applications for Android have been developed for Mini-ME, JFact, HermiT, and Pellet, which handle specific intents and initiate the respective reasoning tasks. On the Python side, the setup required subclassing the `AndroidReasoner` template provided by evOWLuator. Before running the tests, ontologies were uploaded to the device's storage memory.



(a) Classification. **(b)** Consistency.

Figure 3.25: Correctness results of various OWL reasoners.

Table 3.5: Summary of classification correctness tests.

Reasoner	Correct	Incorrect	Timeout (s)	Error	Ratio
Fact++	1393	1	0	4	1.00
HermiT	1398	0	0	0	1.00
Mini-ME	1168	222	8	0	0.84
Mini-ME Swift	1364	34	0	0	0.98
TrOWL	1396	1	1	0	1.00

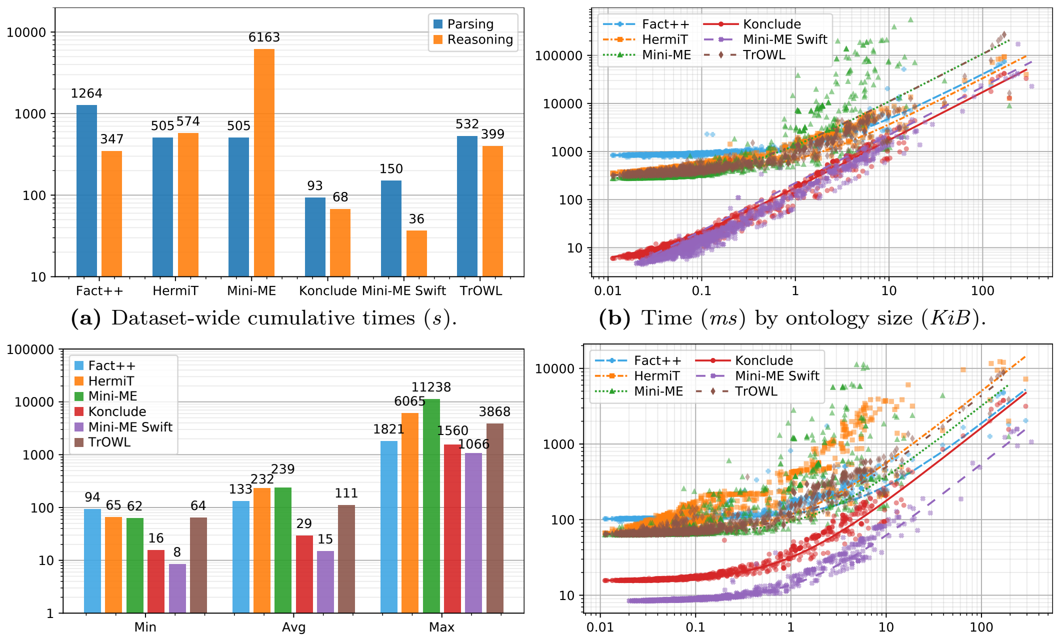
3.3.5.3 Results

Correctness

Correctness has been verified across the entire ORE 2014 dataset using Konclude as the test oracle, recognized as the most reliable reasoner for ontology classification and consistency in the latest OWL reasoner competition [144]. The results, shown in Figure 3.25, illustrate the number of correct and incorrect results for each reasoner with respect to the test oracle (which is not depicted in the plot). The figure also indicates the frequency of timeouts and

Table 3.6: Summary of consistency correctness tests.

Reasoner	Correct	Incorrect	Timeout (s)	Error	Ratio
Fact++	1394	0	0	4	1.00
HermiT	1398	0	0	0	1.00
Mini-ME	1315	75	8	0	0.94
Mini-ME Swift	1387	11	0	0	0.99
TrOWL	1398	0	0	0	1.00



(a) Dataset-wide cumulative times (s).

(b) Time (ms) by ontology size (KiB).

(c) Dataset-wide memory peaks (MiB).

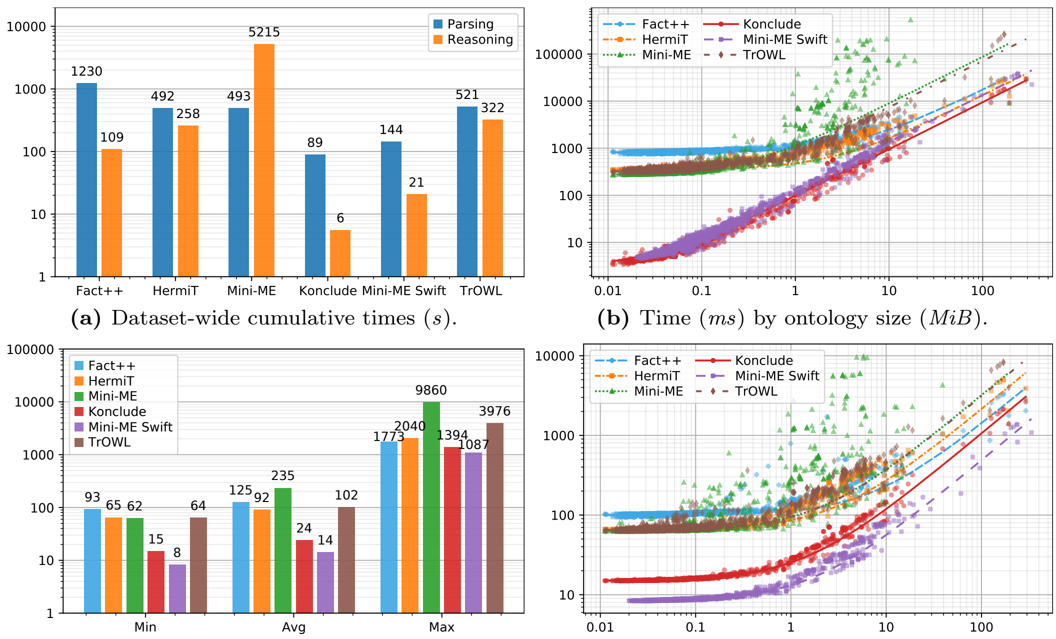
(d) Memory peak by ontology size (MiB).

Figure 3.26: Classification performance tests on desktop.

runtime errors for each reasoner. Detailed information is provided in Tables 3.5 and 3.6, which also include a correctness ratio, computed as the number of correct results divided by the total number of ontologies in the dataset. It can be observed that Mini-ME and Mini-ME Swift have lower ratios compared to other reasoners; this is attributed to unsupported constructs in the ontologies and, in the case of Mini-ME, more frequent timeouts on the largest ontologies in the dataset.

Desktop performance

The performance evaluation metrics focus on the ontologies from the ORE 2014 dataset that were accurately classified by all the previously mentioned reasoners within the specified timeout on the macOS testbed. Figures 3.26 and 3.27 illustrate the results for classification and consistency, respectively, showcasing EVOWLUATOR's ability to generate histograms and scatterplots. Specifically: (i) Figures 3.26a and 3.27a show histogram plots of dataset-wide cumulative parsing and reasoning times in seconds; (ii) Figures 3.26b



(a) Dataset-wide cumulative times (s).

(b) Time (ms) by ontology size (MiB).

(c) Dataset-wide memory peaks (MiB).

(d) Memory peak by ontology size (MiB).

Figure 3.27: Consistency performance tests on desktop.

and 3.27b show the relationship between processing time and ontology size; (iii) Figures 3.26c and 3.27c illustrate dataset-wide minimum, average, and maximum memory peaks for each reasoner; (iv) Figures 3.26d and 3.27d plot memory peak as a function of ontology size.

The aggregated results produced by the framework are summarized in Tables 3.7 and 3.8. These tables show the total parsing and reasoning time, along with the minimum, average, and maximum memory peak across the entire dataset for each reasoner.

Mobile performance

Comparable graphs are displayed for mobile tests, highlighting the performance metrics of Android reasoners (Figures 3.28 and 3.29) and Mini-ME Swift on iOS (Figures 3.30 and 3.31). The results for both desktop and mobile tests are consistent with those from earlier experimental studies [167].

It is clear that EVOWLUATOR employs *Matplotlib* to dynamically adjust

Table 3.7: Summary of classification performance tests on desktop.

Reasoner	Parsing time (s)	Reasoning time (s)	Total time (s)	Min mem. peak (MiB)	Avg mem. peak (MiB)	Max mem. peak (MiB)
Fact++	1247.31	347.48	1594.79	93.57	131.00	1728.90
HermiT	491.35	573.57	1064.92	65.45	227.37	6064.55
Konclude	76.28	67.64	143.91	15.53	27.39	455.15
Mini-ME	491.44	6162.98	6654.41	62.49	233.34	11237.72
Mini-ME Swift	121.37	36.40	157.77	8.41	13.91	189.50
TrOWL	516.93	399.39	916.32	63.88	105.78	1353.89

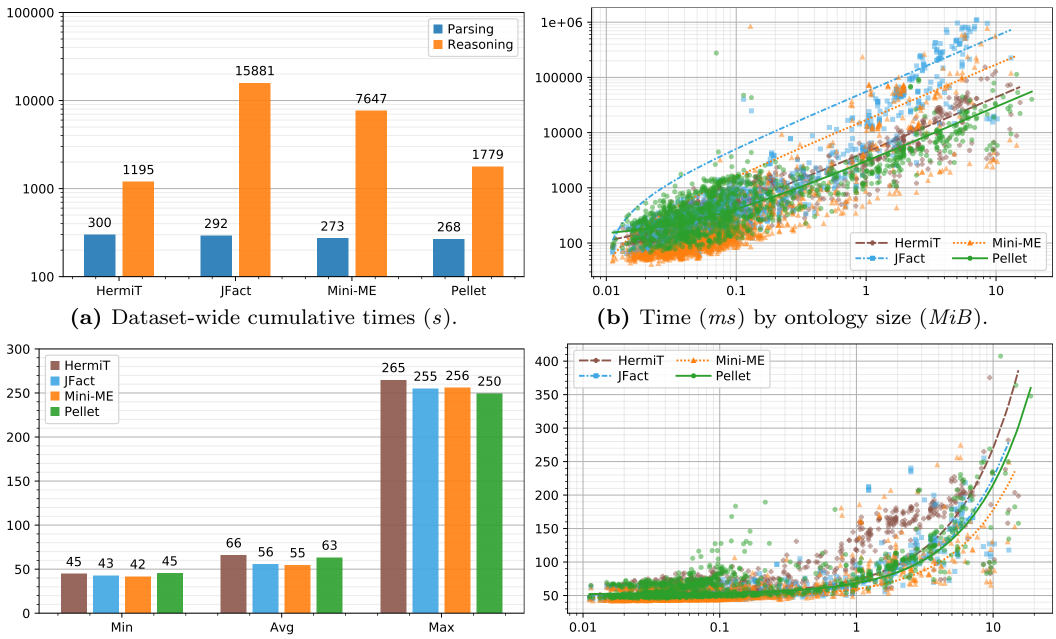
Table 3.8: Summary of consistency performance tests on desktop.

Reasoner	Parsing time (s)	Reasoning time (s)	Total time (s)	Min mem. peak (MiB)	Avg mem. peak (MiB)	Max mem. peak (MiB)
Fact++	1212.18	108.55	1320.73	92.98	122.94	1718.54
HermiT	477.92	258.24	736.16	64.95	89.34	649.09
Konclude	72.57	5.58	78.15	14.96	22.19	245.55
Mini-ME	479.05	5214.83	5693.88	62.38	229.72	9859.68
Mini-ME Swift	114.93	20.69	135.62	8.36	13.16	167.49
TrOWL	506.20	321.51	827.71	63.71	97.32	1278.56

and distinguish graphical elements, such as size and color, in the reports based on the number of reasoners tested. The figures presented in this section have been created in Portable Document Format (PDF) and seamlessly integrated into the LaTeX project of this dissertation. Additionally, they can also be converted to Scalable Vector Graphics (SVG) or Portable Network Graphics (PNG) formats for online publication.

ORE 2014 Energy footprint

These experiments evaluate the energy consumption for reasoning tasks using the ORE 2014 Energy dataset, performed on the macOS testbed. The results, shown in Figure 3.32, reflect the energy usage, with lower scores indicating better performance. Specifically: (i) Figures 3.32a and 3.32c display the minimum, average and maximum energy footprint for each reasoner across the dataset; (ii) Figures 3.32b and 3.32d illustrate the energy footprint as a function of the ontology size; (iii) Tables 3.9 and 3.10 provide a summary of the energy evaluation results in tabular form, replicating the data from Figures 3.32a and 3.32b. Notably, Fact++ and HermiT show a considerably lower energy footprint for the consistency task compared to classification,



(a) Dataset-wide cumulative times (s).

(b) Time (ms) by ontology size (MiB).

(c) Dataset-wide memory peaks (s).

(d) Memory peak by ontology size (MiB).

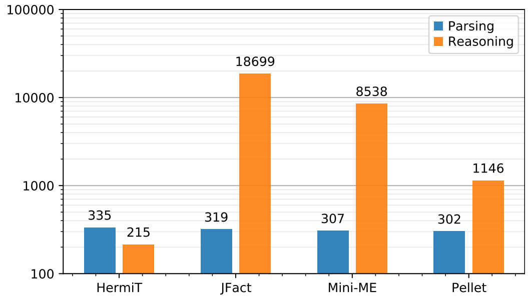
Figure 3.28: Classification performance tests on Android.

Table 3.9: ORE 2014 - Summary of classification energy footprint tests on macOS.

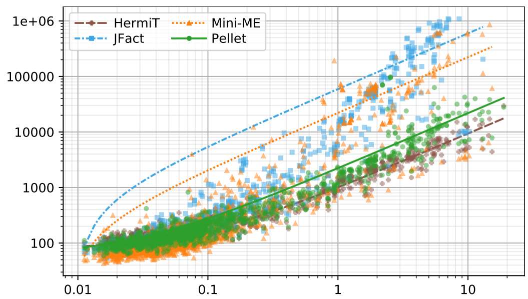
Reasoner	Min energy	Avg energy	Max energy
Fact++	475.44	1406.19	7533.51
HermiT	274.50	1273.30	3588.50
Konclude	47.48	149.18	408.60
Mini-ME	141.93	9547.86	55859.39
Mini-ME Swift	55.74	169.42	445.29
TrOWL	208.21	1174.35	3582.45

whereas for the other reasoners the two scores are closer.

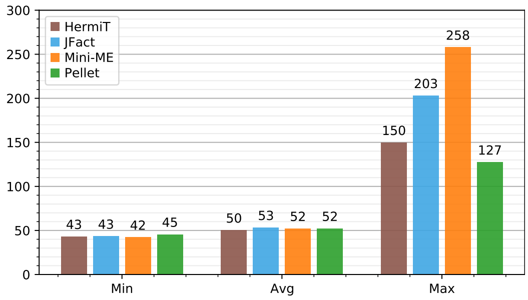
The results, available in CSV format, simplify further analysis employing the *pandas* Python library. For example, Table 3.11 was generated by computing the Pearson correlation coefficients between time-energy, memory-energy, and time-memory for each reasoner in the classification task, using the ORE 2014 Energy dataset. This process involved loading and merging the average performance and energy results with the `DataFrame.merge()` method and



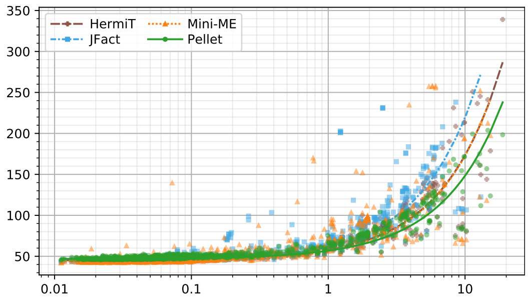
(a) Dataset-wide cumulative times (s).



(b) Time (ms) by ontology size (MiB).



(c) Dataset-wide memory peaks (s).



(d) Memory peak by ontology size (MiB).

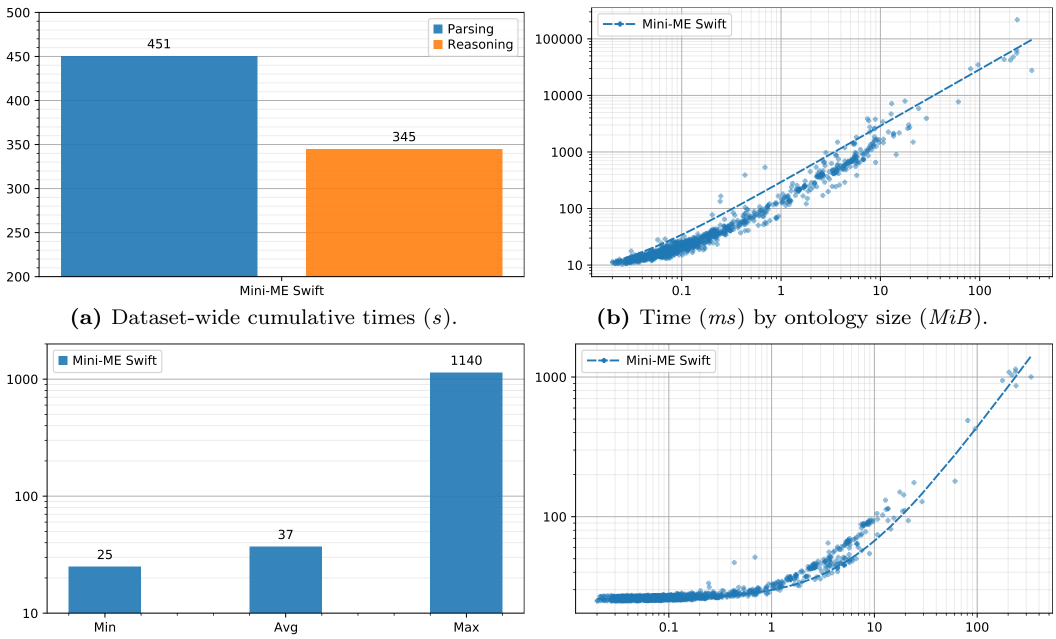
Figure 3.29: Consistency performance tests on Android.

Table 3.10: ORE 2014 - Summary of consistency energy footprint tests on macOS.

Reasoner	Min energy	Avg energy	Max energy
Fact++	332.82	624.91	1241.54
HermiT	269.42	545.51	1115.08
Konclude	37.33	79.82	184.28
Mini-ME	143.05	7655.57	56533.58
Mini-ME Swift	55.75	137.92	373.81
TrOWL	228.31	987.98	2987.28

computing pairwise column correlations via the `DataFrame.corr()` method[63]. The analysis indicates a strong linear correlation between energy consumption and time for all reasoners, consistent with [148] but partially differing from [76], as mentioned in Section 3.3.1. Additionally, most reasoners showed a linear correlation between energy and memory usage, with the exception of Fact++ and Mini-ME, whose unique behavior suggests the need for further targeted investigation.

[63] Pandas DataFrame documentation: `https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html`



(a) Dataset-wide cumulative times (s).

(b) Time (ms) by ontology size (MiB).

(c) Dataset-wide memory peaks (s).

(d) Memory peak by ontology size (MiB).

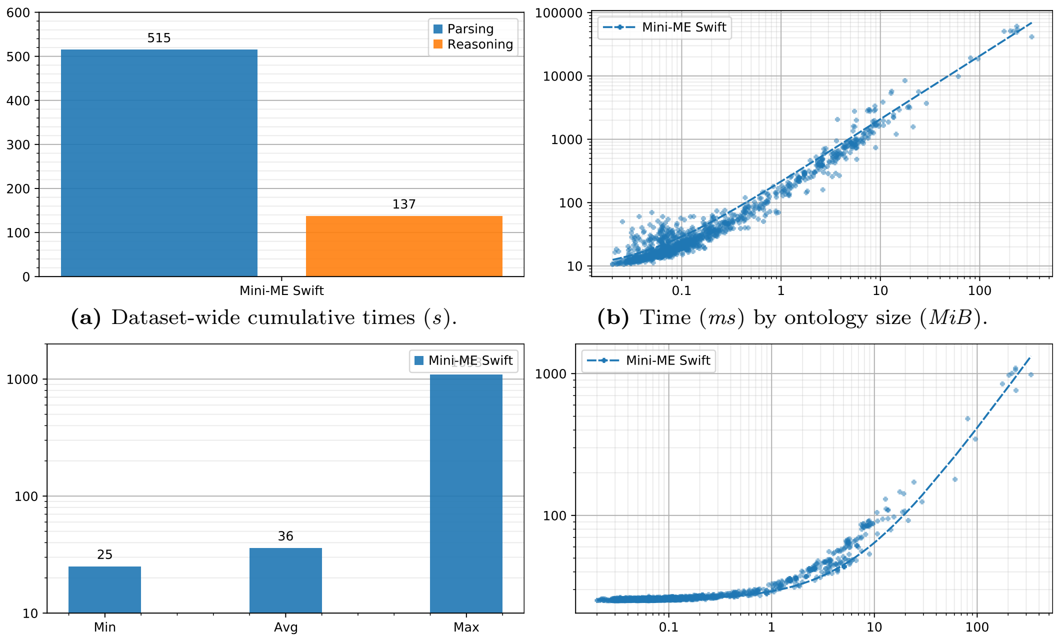
Figure 3.30: Classification performance tests on iOS.

BioPortal Energy Footprint

To assess EVOWLUATOR functionalities across multiple platforms and on extensive and expressive ontologies, an extra experimental campaign has been performed, focusing on energy consumption on both Linux and macOS using the BioPortal Energy dataset. Mini-ME Swift has been excluded from this test due to its expressiveness being limited to $\mathcal{ALN}$. The results are presented as follows: (i) Figure 3.33 shows energy scores recorded on Linux

Table 3.11: ORE 2014 - Correlation between time, memory peak and energy footprint score on macOS.

Reasoner	Energy-Time	Energy-Memory	Time-Memory
Fact++	0.98	0.27	0.19
HermiT	0.99	0.96	0.97
Konclude	1.00	0.99	0.99
Mini-ME	1.00	0.28	0.24
Mini-ME Swift	1.00	0.93	0.93
TrOWL	0.99	0.97	0.98



(a) Dataset-wide cumulative times (s). **(b)** Time (ms) by ontology size (MiB).

(c) Dataset-wide memory peaks (s). **(d)** Memory peak by ontology size (MiB).

Figure 3.31: Consistency performance tests on iOS.

using `powertop`; (ii) Figure 3.34 illustrates energy scores measured on macOS with `powermetrics`; (iii) Tables 3.12 and 3.13 provide a detailed comparison of Linux and macOS results in tabular form;

Fact++ and HermiT demonstrate significantly lower energy consumption during consistency checks compared to classification tasks, whereas the other reasoners exhibit more uniform behavior across both inference types, consistent with previous observations. Table 3.14, constructed similarly to Table 3.11 from earlier experiments, details the Pearson correlation coefficients for time-energy, memory-energy, and time-memory for each reasoner. These results show remarkable consistency, even though different operating systems and energy profilers were used. Specifically, the correlation between *powertop* scores on Linux and *powermetrics* measurements on macOS, documented in Table 3.15, is quite high. When comparing Table 3.11 with Table 3.14, it is evident that for larger ontologies, HermiT's correlation values become more similar to those of Fact++. This suggests that lower energy-time and energy-memory correlations may result from weaker time-memory correlation,

Table 3.12: BioPortal - Summary of classification energy footprint tests on Linux and macOS.

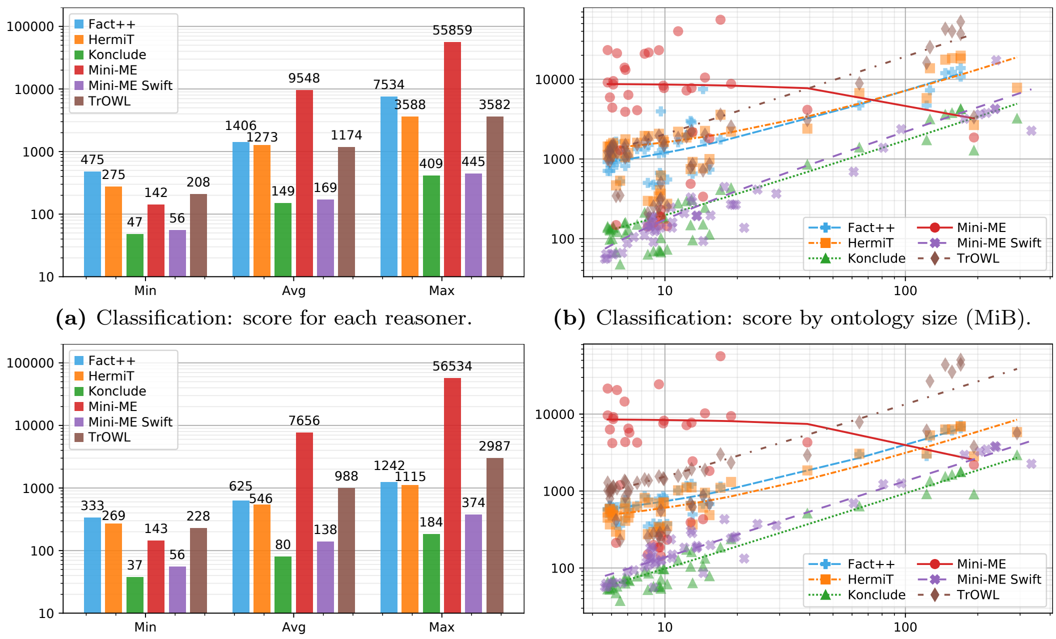
Platform	Reasoner	Min energy	Avg energy	Max energy
Linux	Fact++	23.87	1606.24	6747.2
	HermiT	32.65	236.07	1244.58
	Konclude	3.61	21.78	56.34
	TrOWL	30.7	80.47	171.51
macOS	Fact++	1472.49	131134.95	563359.07
	HermiT	2524.55	17081.62	95200.06
	Konclude	395.70	2031.57	5641.29
	TrOWL	2442.32	6483.57	13857.09

Table 3.13: BioPortal - Summary of consistency energy footprint tests on Linux and macOS.

Platform	Reasoner	Min energy	Avg energy	Max energy
Linux	Fact++	21.77	45.8	94.99
	HermiT	20.51	62.66	133.28
	Konclude	2.76	16.75	51.4
	TrOWL	26.5	79.65	162.73
macOS	Fact++	1282.61	2949.97	6136.69
	HermiT	1436.54	4305.14	9269.16
	Konclude	243.23	1703.41	4976.11
	TrOWL	1991.29	5736.65	12179.98

Table 3.14: BioPortal - Correlation between time, memory peak and energy footprint score on Linux and macOS.

Platform	Reasoner	Energy-Time	Energy-Memory	Time-Memory
Linux	Fact++	0.61	0.70	0.42
	HermiT	0.69	0.53	0.36
	Konclude	0.89	0.91	0.97
	TrOWL	0.97	0.97	0.92
macOS	Fact++	0.61	0.63	0.37
	HermiT	0.70	0.63	0.69
	Konclude	0.99	0.98	0.96
	TrOWL	0.98	0.98	0.98



(a) Classification: score for each reasoner. **(b)** Classification: score by ontology size (MiB).

(c) Consistency: score for each reasoner. **(d)** Consistency: score by ontology size (MiB).

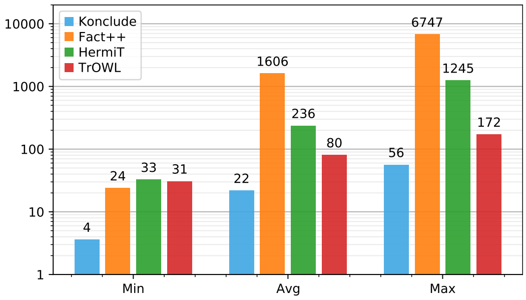
Figure 3.32: ORE 2014 - Energy footprint tests on macOS.

Table 3.15: BioPortal - Correlation between energy scores on Linux and macOS.

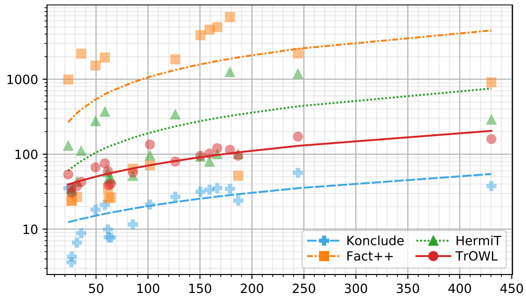
Reasoner	Energy Correlation
Fact++	0.97
HermiT	0.99
Konclude	0.90
TrOWL	0.97

which are influenced by the interaction between inference algorithms and the size and structure of the ontology. Further in-depth analysis is recommended in future experimental studies.

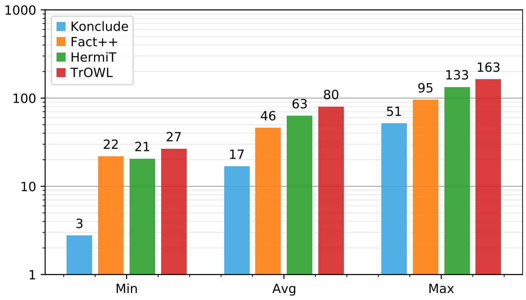
Drawing from this case study and early feedback from thesis students and interns at the research group laboratory, EVOWLUATOR is easy to use and highly adaptable to test real-world reasoners. Integrating six desktop reasoners along with five additional mobile test configurations required just 250 lines of Python code. Visualization capabilities have been particularly valuable for quick and easy performance evaluations.



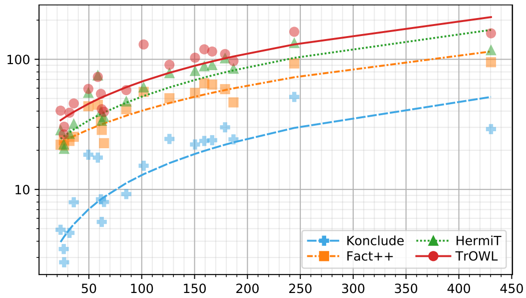
(a) Classification: score for each reasoner.



(b) Classification: score by ontology size (MiB).

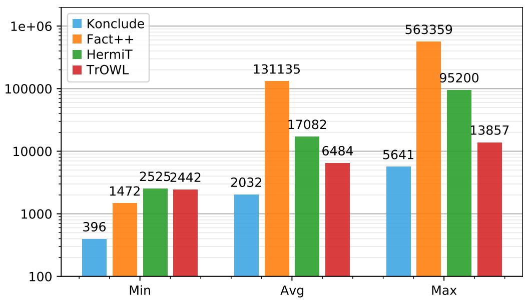


(c) Consistency: score for each reasoner.

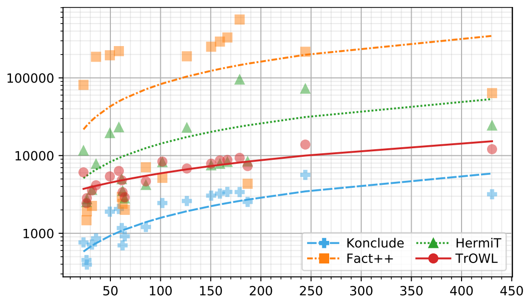


(d) Consistency: score by ontology size (MiB).

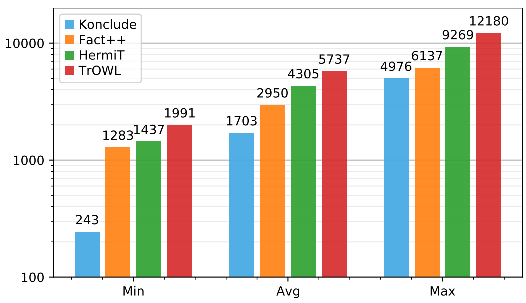
Figure 3.33: BioPortal Energy footprint tests on Linux desktop.



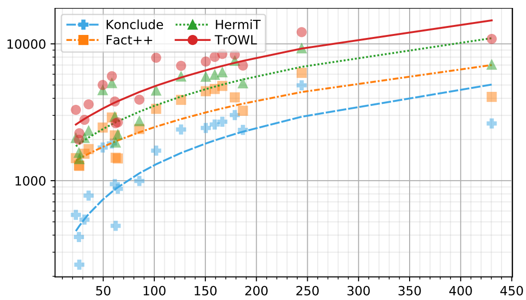
(a) Classification: score for each reasoner.



(b) Classification: score by ontology size (MiB).



(c) Consistency: score for each reasoner.



(d) Consistency: score by ontology size (MiB).

Figure 3.34: BioPortal Energy footprint tests on Mac Mini.

Chapter 4

Serverless microservice architecture for distributed cloud-edge intelligence

This chapter presents a *Cloud-Edge Intelligence* (CEI) framework for sensor-based decentralized coordination applications, exploiting serverless computing for data management and ML tasks [116]. Event-driven stateless functions implement the logical components of a full cloud-edge framework for sensor data gathering, preprocessing, storage, ML model training, and inference. IoT-based field devices provide functions for sensor data gathering and basic preprocessing as Real-Time OS (RTOS) tasks; data storage, model training and inference are implemented in functions running either on edge devices or on cloud infrastructure nodes, sharing a single codebase and avoiding duplication of development effort. The node selection for responding to a given function invocation is managed opportunistically, based on application requirements, network proximity, and available bandwidth and computing resources.

The overall architecture enables an automatic management of the inherent trade-off between model accuracy and computational complexity that exists in CEI applications. Cloud endpoints can process larger amounts of sensor data and thus train more accurate ML models; this not only implies the use of larger amounts of computation, memory and storage resources, but in some scenarios can be prevented by data transmission bandwidth limitations from the local network to the cloud or by (industrial or regulatory) privacy restrictions. Conversely, edge nodes can train models from local data, preserving data privacy and reducing bandwidth usage and prediction response times, at the expense of reduced model accuracy due to smaller training datasets. The proposed architecture supports both training and inference at cloud as well as at edge layer, enabling developers to take advantage of the benefits specific to each context whenever possible. At the same time, in order to mitigate the accuracy limitations of training at the edge, models trained in the cloud are then disseminated to edge nodes periodically, thus improving the overall accuracy progressively during the lifetime of the CEI application. Data privacy is ensured by limiting data transmission to the cloud and prioritizing local processing on edge nodes when sensitive information is involved.

The rest of the chapter is organized as follows: Section 4.1 covers the related work. Section 4.2 provides a detailed explanation of the framework architecture and its individual components, including the reference hardware, software, and cloud off-the-shelf components used for implementation. Lastly, Section 4.3 outlines the reference testbed and experiments.

4.1 Background

In recent years, the progress made in developing increasingly accurate and robust AI models on centralized cloud-based servers has paralleled the rapid expansion of edge computing solutions. *Edge Computing* (EC) refers to a paradigm where significant computing and storage resources are placed at the local network's edge, close to field devices and sensors [176]. This approach reduces communication latency and enhances responsiveness in various applications, from digital healthcare to smart manufacturing [2, 156]. Instead of relying entirely on cloud execution, the CEI paradigm has emerged to enable cooperative AI model execution between the edge and the cloud, thus minimizing both end-to-end latency and energy consumption [128]. CEI applications typically operate in a coordinated cloud-edge device environment, according to the six-level classification of architectures discussed in [232]. Shifting from the *Cloud Intelligence* toward the *All On-Device* paradigm [232] enhances data privacy, while simultaneously reducing data offloading, path length, transmission latency, and bandwidth costs. Most current CEI platforms adopt the *In-Edge co-inference* paradigm, where model training takes place exclusively in the cloud and inference (prediction) is executed at the edge. A notable example is the energy management platform described in [205], which uses the cloud infrastructure for model training and evaluation before regularly deploying models to edge nodes for daily operations in power plants. Similarly, the cloud-edge collaboration framework for cognitive services in [59] features a deep model deployed on the cloud continuously supporting a shallow model on the edge in training and real-time inference tasks. The architectures enabling collaborative learning in a cloud-edge context are comprehensively reviewed in [222].

Osmotic Computing is a promising approach to address the challenges of coordinating AI tasks across edge and cloud layers [136]. The main concerns in designing and implementing AI-based applications within an OC environment are outlined in [131]. OC has been applied in many works in a variety of contexts. For example, the trust management framework in [188] leverages OC to efficiently distribute computation among several nodes in Pervasive Online Social Networks (POSNs). The pollution monitoring platform in [117] uses OC for opportunistic data filtering and integration from a variety of IoT sensors in urban areas. Similarly, orchestrating microservices for data analysis in statistical computing is tackled in [75], while [142] proposes an OC architecture for smart classrooms, exploiting deep learning models for IoT device control and computer vision tasks such as chalkboard handwriting recognition. However, the lack of containerization and dynamic orchestration in these early OC implementations limits their capabilities. Recent developments have extended the foundational OC features to incorporate enhanced capabilities. For example, [158] proposes an OC-based mechanism for message-oriented middleware in IoT environments. [90] focuses on energy-aware resource management, by deploying a hyper-heuristic for optimal service dispatch on incoming workloads and an extended Kalman filter for monitoring edge datacenters. The framework in [216] targets microservice deployment across *cloud-to-thing* environments, encompassing cloud, edge and IoT layers: an osmotic smart orchestrator is designed to migrate *MicroELements* (MELs) composed of microservices along with their associated data. This architecture is also applied in [41] to develop a distributed healthcare system and a Body Area Network (BAN) case study. Furthermore, OC is used in [187] to schedule and deploy services across multiple nodes in distributed mobile augmented/virtual reality applications. A CEI microservice architecture based on the OC paradigm is also described in [121], enabling the collection of data streams from sensors deployed in the environment, data preprocessing, AI model training and inference for ML classification and regression tasks. This architecture has been tailored to CEI applications involving IoT-based sensor networks [124]. In [124], each atomic functionality (storage, DSMS, AI, etc.) of the CEI framework is containerized in a separate microservice. This solution adopts the same core set of services, but distributes them in a novel way, as explained in Section 4.2: application-independent services (such as storage and DSMS) are directly deployed to the host device, while application-specific features are implemented and distributed as serverless functions. This approach reduces overhead by eliminating containers, making deployment of components simpler and more flexible, as allowed by the underlying serverless function runtime.

Despite the progress made, significant challenges remain in developing effective CEI solutions, particularly in terms of scalability, low communication latency, energy efficiency, and cost savings. *Serverless computing* [186] aims to enable dynamic environments where application microservices are encapsulated in stateless *functions*, relieving users from directly managing the underlying platforms and the infrastructure. The adoption of serverless architectures in cloud-edge applications promises significant opportunities for sensor networks [42, 157]. Preliminary data analytics functions can be executed on the edge devices with low latency, by leveraging their proximity to sensors generating the data, while computationally expensive tasks like model training on large data sets produced by field sensors can be handled in the cloud. *Rural AI* [147] exemplifies the potential of serverless computing in

Table 4.1: Comparison of serverless CEI frameworks

Framework	Software stack description	Physical testbed description	ML model type(s)	Edge MPU support
Rural AI [147]	Only FaaS platform	None	ANN	No
FedLess [71]	Complete	Complete	ANN	No
Hierarchical [137]	None	None	ANN	No
Proposed framework	Complete	Complete	multiple[1]	Yes

challenging environments, using the *funcX* federated serverless platform [45] to introduce a framework supported by a Function-as-a-Service (FaaS) structure. This approach shows promise for federated learning in precision farming, often hindered by inconsistent and poor network infrastructure. Comparable strategies can be found in *FedLess* [71], a serverless platform tailored for the training and application of Deep Neural Network (DNN) models on different FaaS platforms, and in the serverless hierarchical framework in [137]. The latter employs a two-level architecture, where cluster heads supervise sets of nodes that can exchange local machine learning model parameters.

As shown in Table 4.1, the architecture presented in this chapter distinguishes itself from the aforementioned serverless CEI frameworks for several key reasons:

- it provides a comprehensive overview of its software stack, with a specific emphasis on the use of Commercial Off-The-Shelf (COTS) components;
- it details targeted devices and platforms, focusing on the lower end of the cloud-edge continuum, along with results from an experimental evaluation conducted on a physical testbed;
- it supports a wide array of machine learning models, providing flexibility with respect to platform constraints and application requirements;
- it explicitly allows model training and inference on edge MPU devices, accounting for their hardware resource limitations.

4.2 Cloud-Edge Infrastructure Framework

This section describes the proposed CEI framework, providing a detailed overview of the architecture and examining the features of the reference device targets and the cloud platform that support its implementation.

4.2.1 Architecture

Figure 4.1 depicts the proposed framework infrastructure. Similarly to [124], each logical component is encapsulated in a building block. The architecture supports a wide range of composition patterns, enabling the dynamic provisioning of each microservice to a specified device, while taking into account its capabilities and constraints. It integrates both long-lived microservices, hosted in containers, and serverless functions, invoked through an asynchronous event-driven pattern. In Figure 4.1, containers are marked by gray gear icons, within OS virtualization environments displayed as yellow blocks. Conversely, serverless functions are represented by a blue λ icon, inside serverless function runtimes shown as orange blocks.

The proposed architecture includes three kinds of nodes:

- **Field Devices**: local IoT sensors with severely constrained processing capabilities, powered by MicroController Units (MCUs). They are primarily used for data collection and basic preprocessing tasks.
- **Edge Nodes**: located at the border of the local network of *Field Devices*, these intermediary nodes are equipped with MicroProcessor Units (MPUs), offering resources similar to those of single-board computers or fog computing *small servers* [154]. They are capable of performing more complex tasks, such as edge AI training and inference on local data.
- **Cloud Node**: remote datacenter hosting the cloud infrastructure. It provides extensive computational and networking resources, supporting

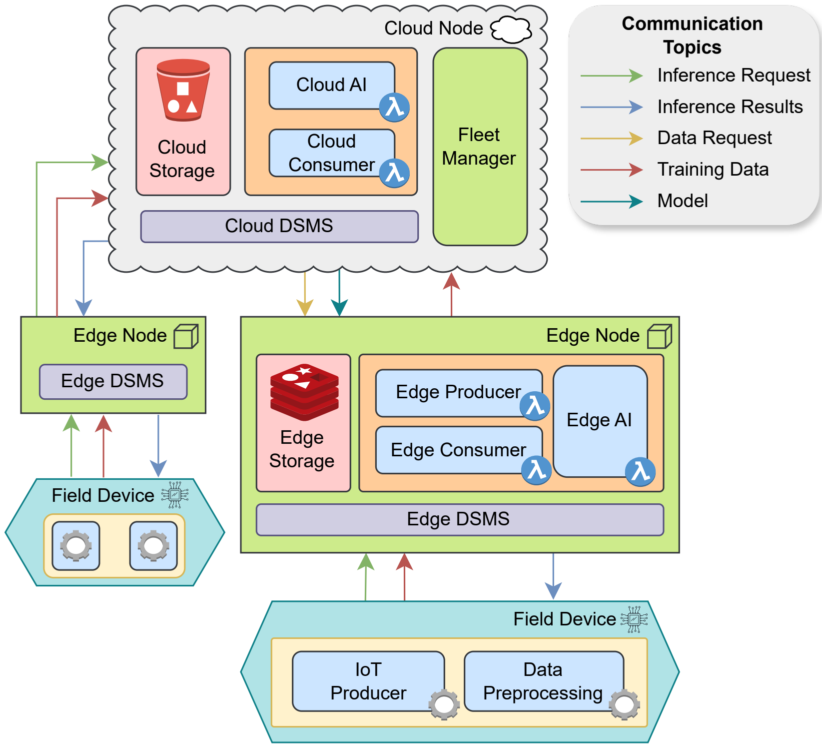


Figure 4.1: CEI framework architecture

large-scale machine learning model training and inference, and serving the underlying network of devices.

The logical components of the architecture are encapsulated in microservices, described in what follows.

Data Stream Management System (DSMS): serving as a connection interface for each node, this component enables seamless communication and message exchange across the network using a *publish/subscribe* pattern, which is well-suited for scalable, loosely-coupled microservice architectures [158]. Notably, in the proposed framework each node exposes its DSMS service (denoted in Figure 4.1 as either *Edge DSMS* or *Cloud DSMS*), and these services can be federated. This marks a departure from [124], where only one DSMS was employed in a central orchestrator node. By distributing DSMS instances across nodes, the system achieves resource replication and improved fault tolerance. In the event of connectivity issues or device faults, *Field Devices* and third-party applications can temporarily connect to another available DSMS as a fallback option. Furthermore, having multiple DSMS instances allows for further decoupling of nodes from each other, leading to a more robust and flexible overall architecture. The DSMS is compatible with *Message Queuing Telemetry Transport* (*MQTT*) [18], a widely-used communication protocol for IoT-based sensor networks. In the proposed architecture, nodes establish a connection with the nearest message broker. For each message received, the DSMS invokes the associated service –possibly encapsulated in a serverless function– to process a given request, following an event-driven approach. If the local node lacks the necessary service, the DSMS relays the message to the *Cloud Node* for processing, ensuring continuity in the application flow. Communication is organized via MQTT *topics*, a structured channeling that helps in message categorization and dispatch. Defined topics include: (i) `Inference Request`, for publishing requests for inference (prediction); (ii) `Inference Results`, where the outcomes of inference tasks are dispatched; (iii) `Data Request`, employed by the cloud to prompt *Edge Nodes* to transfer data from local to Cloud Storage; (iv) `Training Data`, a channel for nodes to share data intended for model training; (v) `Model`, the channel through which the cloud disseminates updated models to the *Edge Nodes*. The separation of data into distinct topics simplifies the configuration, since it allows a direct association of specific topics with serverless functions invocation.

Fleet Manager: oversees the provisioning of edge devices and the deployment of architectural software components. Differently from the *Orchestrator* component in [124], it adopts a hybrid deployment strategy: some software components are deployed as containers, while a serverless approach is used for deploying functions. This strategy optimizes the deployment process by tailoring it to the specific nature of each software component and the capabilities of the edge devices, resulting in a more balanced and efficient resource management within the architecture.

Edge Storage: serves as a local, temporary repository for data collected from field sensors and IoT devices. Its proximity to data processing microservices enhances both latency and bandwidth optimization.

Cloud Storage: acting as the cloud counterpart to *Edge Storage*, this is a persistent, fault-tolerant storage solution for all data collected within the network. It enables other network nodes to access stored data via its interface and supports integration with third-party analysis tools, *e.g.*, a visualization dashboard. It also stores and provides access to the latest ML models, enabling *Edge Nodes* to retrieve them and update their local models with a more advanced version [124].

Consumer: this service –available both on *Edge Nodes* and *Cloud Nodes*– is responsible for dispatching incoming messages via the associated DSMS. This includes reading the training data from sensing *Field Device(s)* and storing them into the closest local storage, being the *Edge Storage* for *Edge Nodes* and *Cloud Storage* in the cloud datacenter, as well as invoking *Cloud AI* or *Edge AI* –whichever is closer– to process incoming inference requests.

Producer: this service –on *Field Devices* as well as on *Edge Nodes*– is responsible for publishing preprocessed training data or inference requests. In case of field IoT sensor devices, this is implemented as a lightweight RTOS task sending data towards the closest *Edge Node*, while the *Edge Producer* is a serverless function triggered by `Data Request` messages to upload data from the edge to the Cloud Storage.

Cloud AI and **Edge AI**: they handle model training and prediction tasks on sensor network data streams, collected at the cloud layer from various Producer instances. This enable continuous model refinement through a feedback loop: a less accurate model may be trained and deployed on edge devices using the subset data available to each *Edge Node*, while a more accurate model is trained in the cloud by aggregating larger volumes of information. This enhanced model can then be retrieved by *Edge Nodes* to replace their local models. The training and prediction tasks are executed on individual nodes, independently from the rest of the network.

Figure 4.1 shows two *Edge Nodes* connecting to the cloud. The left node is a lightweight *Edge Node*, which lacks support for AI services, and only executes the basic *Edge DSMS* service. Upon receiving a message, this node forwards to the cloud for request fulfillment. In contrast, the right node is equipped to handle AI, storage, data production, and data consumption services; it is also able to address requests originating from *Field Devices*' sensor data collection events, without requiring cloud intervention. This highlights the flexibility of the proposed framework. It should be noted that the architecture is not limited to these two cases, as it allows the deployment of a subset of services to a particular *Edge Node* and a corresponding configuration of the DSMS to relay messages only when the receiving end lacks the required service to process the request. Data can undergo preliminary processing –*e.g.*, for noise reduction, smoothing, extraction of basic statistical features, etc.– on *Field Devices* via a lightweight *Data Preprocessing* RTOS task, before being transmitted to the nearest *Edge Node* for more sophisticated processing and mining.

4.2.2 Targets and platforms

Table 4.2: Reference development boards for Micro Controller Units

Board	MCU	Architecture	Processor Speed	Memory (SRAM)	Storage (Flash)
Arduino Uno	ATmega328P	Atmel 8-bit AVR RISC-based	20 MHz Single Core	2 KB	32 KB
NUCLEO-C031C6	STM32C031C6T6	ARM 32-bit Cortex-M0+	48 MHz Single Core	12 KB	32 KB
Blue Pill	STM32F103C8T6	ARM 32-bit Cortex-M3	72 MHz Single Core	20 KB	64 KB
Arduino Due	AT91SAM3X8E	ARM 32-bit Cortex-M3	84 MHz Single Core	96 KB	512 KB
Raspberry Pi Pico	RP2040	ARM 32-bit Cortex-M0+	133 MHz Single Core	264 KB	16 MB
ESP32-Gateway	Xtensa LX6	Tensilica 32-bit RISC-based	240 MHz Dual Core	520 KB	4 MB
B-L4S5I-IOT01A	STM32L4S5VIT6	ARM 32-bit Cortex-M4	120 MHz Single Core	640 KB	2 MB

The proposed framework takes advantage of heterogeneous deployment and composition strategies of the aforementioned logical components to support different kinds of devices, including MPUs and MCUs, for task distribution

[1]Supported ML models: `https://scikit-learn.org/stable/user_guide.html`

Table 4.3: Reference development boards for Micro Processor Units

Board	MPU	Architecture	Processor Speed	Memory (DRAM)	Storage
Raspberry Pi B	Broadcom BCM2835	ARM1176JZF-S 32-bit	0.7 GHz Single Core	512 MB	MicroSD SDXC
BeagleBone Black	AM335x	ARM 32-bit Cortex-A8	1.0 GHz Single Core	512 MB	4 GB eMMC
Raspberry Pi 3B	Broadcom BCM2837	ARM 64-bit Cortex-A53	1.2 GHz Quad-Core	1 GB	MicroSD SDXC
Raspberry Pi 3B+	Broadcom BCM2837B0	ARM 64-bit Cortex-A53	1.4 GHz Quad-Core	1 GB	MicroSD SDXC
Asus Tinker Board	Rockchip RK3288	ARM 32-bit Cortex-A17	1.8 GHz Quad-Core	2 GB	MicroSD SDXC
Jetson Nano	Tegra X1 SoC	ARM big.LITTLE Cortex-A53 + A57	1.9 GHz Quad-Core 1.3 GHz Quad-Core	4 GB	MicroSD SDXC
ROCKPro64	Rockchip RK3399	ARM big.LITTLE Cortex-A72 + A53	1.8 GHz Dual-Core 1.4 GHz Quad-Core	4 GB	128 GB eMMc
Raspberry Pi 4B	Broadcom BCM2711	ARM 64-bit Cortex-A72	1.7 GHz Quad-Core	8 GB	MicroSD SDXC
Variscite DART-MX8M	i.MX 8M Plus	ARM Cortex-A53	1.5 GHz Quad-Core	8 GB	64 GB eMMC

across the local network. MCUs, with their embedded on-chip flash memory, excel in quick start-up times and short computational bursts, making them ideal for real-time tasks such as data acquisition. Conversely, MPUs relay on external non-volatile memory for program and data storage, offering larger capacity but requiring longer start-up times due to code transfer to an external Dynamic Random-Access Memory (DRAM) before execution starts. The memory capacity in MPUs typically spans from hundreds of megabytes to few gigabytes, while MCUs generally have up to 2 MB of program memory and several hundred kilobytes of embedded Static Random-Access Memory (SRAM). Both categories of devices are usually provided as development boards as well as standalone chips for integration into final products. Table 4.2 lists reference COTS development boards for embedded micro-controllers as supported by the proposed framework, whereas Table 4.3 includes MPU-based single-board computers. As depicted in the architecture in Section 4.2.1, MCUs are used for data collection and basic data processing tasks in sensing devices, while edge nodes are equipped with more capable MPUs allowing the execution of containerized microservices as well as hosting serverless functions.

Table 4.4: Reference software technologies

Node	Component	Technologies
Cloud	Storage	AWS S3
	Consumer	AWS Lambda, Boto3
	AI	AWS Lambda, Boto3, scikit-learn
	DSMS	AWS IoT Core
	Fleet manager	AWS IoT Greengrass cloud
Edge	Storage	AWS IoT Greengrass, Redis
	Consumer	AWS IoT Greengrass Lambda, Boto3
	Producer	AWS IoT Greengrass Lambda, Boto3
	AI	AWS IoT Greengrass Lambda, scikit-learn
	DSMS	AWS IoT Greengrass, Moquette, MQTT bridge
Field	Producer	FreeRTOS task, coreMQTT
	Preprocessing	FreeRTOS task

A fully functional prototype of the previously detailed logical architecture has been implemented on top of the *Amazon Web Services* (AWS)[2] platform. The mapping of the logical components in Section 4.2.1 to AWS services and software technologies is detailed in Table 4.4 and described hereafter. In the Cloud Node, the DSMS is based on *AWS IoT Core*[3], a platform designed to enable secure and scalable device connectivity to the AWS cloud. IoT Core provides essential features such as message broker services that allow devices and applications to send and receive messages using protocols such as MQTT, HTTP, and WebSocket. It also includes more advanced features, such as a rule engine to route, transform, and filter messages along the way, and a *Device Shadow* service which maintains a persistent representation of device state for when devices are offline. Furthermore, IoT Core can be easily integrated with other AWS services, like *AWS Lambda* and *Amazon S3*, which are used in the proposed architecture to implement the *Serverless Function Runtime* and *Cloud Storage*, respectively.

Amazon Simple Storage Service[4] (S3) is a scalable object storage service designed to store and retrieve large amounts of data from anywhere within the AWS infrastructure. Data in S3 are organized in buckets (similar to directories) and objects (comparable to files). Each object is assigned a

[2]Amazon Web Services: `https://aws.amazon.com`
[3]AWS IoT Core: `https://aws.amazon.com/iot-core`
[4]AWS S3: `https://aws.amazon.com/s3`

unique identifier key provided by the user, allowing for easy data retrieval. S3 provides an event notification feature triggering responses to given events, enabling automated workflows which can be beneficial in sensor network applications for real-time or near real-time data processing and analysis.

Lambda[5] is the AWS serverless computing platform, which allows developers to execute code in response to events without provisioning or managing servers or containers. It automatically scales applications by running code in response to each trigger from a few requests per day to thousands per second, depending on the size of the workload. Lambda runs code in a managed runtime environment and supports several programming languages, including Node.js, Python, Ruby, Java, and .NET 8; as technology evolves, further runtimes may be added. The *Cloud Consumer* and both AI components are built on top of Lambda, leveraging the builtin Python 3.9 runtime. The *Boto3*[6] library is used to interface with AWS services such as S3, and *scikit-learn*[7] [149] is exploited for ML model training and inference.

Edge nodes are managed by the *Fleet Manager* using the *AWS IoT Greengrass*[8] cloud service. Each logical block in an edge node is implemented as an AWS IoT Greengrass component, *i.e.*, a software module that can be deployed to any device in the fleet. AWS IoT Greengrass extends cloud capabilities to edge devices, enabling them to process data locally while still using the cloud for management, analytics, and durable storage. Greengrass supports local Lambda functions, allowing code to be run locally on Greengrass devices while using the same programming model as AWS Lambda. The Greengrass Python Lambda runtime is employed for the *Producer*, *Consumer*, and *Edge AI* components, mirroring the serverless functions on the cloud, which allows for low-latency local data processing and inference.

The *Edge DSMS* component is based on the *Moquette* MQTT broker for local communication with field devices and an *MQTT bridge* for data exchange

[5]AWS Lambda: `https://aws.amazon.com/lambda`
[6]AWS Boto3: `https://aws.amazon.com/sdk-for-python`
[7]scikit-learn: `https://scikit-learn.org`
[8]AWS IoT Greengrass: `https://aws.amazon.com/greengrass/`

with cloud nodes: both are provided by AWS Greengrass. The *Edge Storage* is based on *Redis*[9], an open-source, in-memory data store. It supports a wide range of data structures including strings, hashes, lists, sets, and geospatial indexes. By storing data in memory rather than on mass storage, Redis enables high-throughput operations with sub-millisecond latency, making it well-suited for real-time processing applications.

Field devices leverage the *FreeRTOS*[10] open-source RTOS for embedded systems, which provides essential multitasking features, such as scheduling, inter-process communication, and timing operations, and it is portable across various microcontroller platforms. Additional libraries are available as part of the *FreeRTOS-Plus* package, which provides support for TCP/IP networking, security protocols, and file I/O. Additional specialized libraries such as *corePKCS11* are used for encryption and secure element access, while *coreMQTT* handles communication with edge MQTT brokers.

Integrating COTS services and components enhances the framework's capabilities, reliability, security, and robustness, thus improving its overall technical quality. However, this effort goes beyond engineering; it involves designing a completely new federated learning architecture with custom orchestration logic for serverless functions across heterogeneous cloud and edge target platforms. This integration required defining training and inference workflows, by optimizing serverless function implementations to support resource-constrained nodes and by designing the management of models and of training data.

[9]Redis: `https://redis.com`
[10]FreeRTOS: `https://freertos.org`

4.3 Experiments

4.3.1 Testbed configuration

The experimental setup complies with the architecture described in Section 4.2.1 and utilizes hardware from the OC-based testbed in [124] to facilitate a direct comparison. In fact, the setup consists of:

- two **Field IoT Devices** using two B-L4S5I-IOT01A boards equipped with STM32L4S5VIT6 MCU, which simulate sensor data acquisition;
- two **Edge nodes**, respectively based on a Raspberry Pi 3B (Edge 1) and a 3B+ (Edge 2), each running the AWS IoT Greengrass client software;
- the **Cloud Infrastructure** provided by AWS, supplying the cloud services as specified by the architecture.

The connectivity configuration reflects the setup used in [124]: edge nodes and field devices are interconnected through a promiscuous –*i.e.*, not dedicated exclusively to the testbed devices– IEEE 802.11 wireless link, while communication between the local network and the cloud is provided by an Internet connection [11].

In this testbed *scikit-learn* has replaced *Keras*[12] with *TensorFlow*[13], as used in [124], since the latter exceeds the 50 MB function code size limit currently imposed by AWS Lambda. Despite this change, the framework remains flexible: complex training tasks can still be deployed as persistent containerized services, while serverless functions can handle more lightweight training or inference tasks. The test uses the same machine learning model and training procedure as in [124]: a multi-layer perceptron regressor [70] with

[11]Fiber To The Cabinet, 100/30 Mbps downstream/upstream bandwidth.

[12]Keras: `https://keras.io/`

[13]TensorFlow: `https://www.tensorflow.org/`

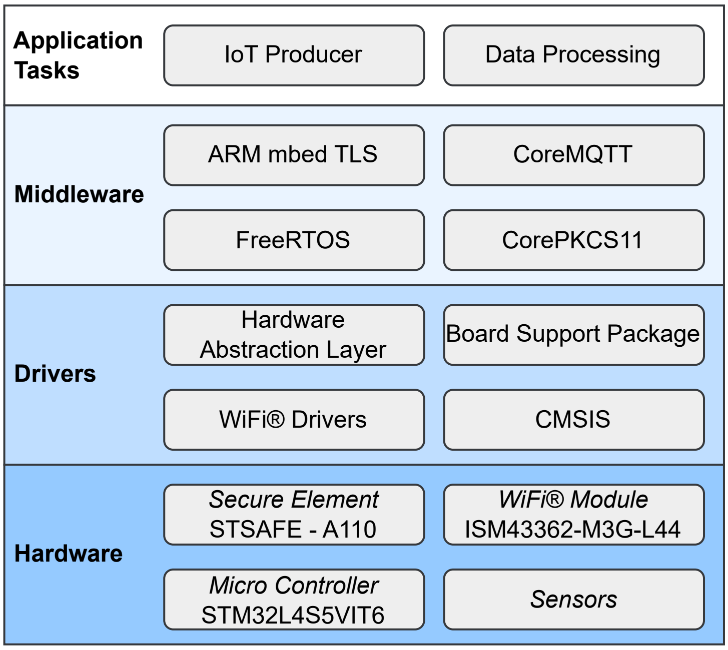


Figure 4.2: Field device firmware architecture

5 hidden layers and 200 neurons each, trained with the *Adam* [97] optimizer over 10 epochs. The same public dataset for ore quality prediction in mining [125] has been adopted for tests. With respect to the original dataset, the "date" column used for timestamping has been removed, resulting in a total of 22 features. The dataset comprises $N = 737453$ samples, and the prediction label is "% Silica Concentrate".

The **Field Devices** are equipped with a custom firmware designed to support both data preprocessing and MQTT communication. The layered firmware architecture is sketched in Figure 4.2. The hardware level is given by the B-L4S5I-IOT01A board, detailed in Table 4.2, which provides on-board sensors and an embedded *ST-LINK* programmer interface. The development environment, *STM32CubeIDE*, along with the *X-Cube-AWS* expansion pack, provides the necessary software components for the *Drivers* and *Middleware* layers. The Drivers layer encapsulates:

- *Hardware Abstraction Layer (HAL)* APIs for processor-specific access;

- *Board Support Package (BSP)* APIs for board-specific component access;
- drivers for the on-board Wi-Fi module;
- *Common Microcontroller Software Interface Standard (CMSIS)*, which exposes common interfaces to board drivers for upper layers across various ARM microcontrollers, essential for porting FreeRTOS kernel to the device.

Above the drivers, **FreeRTOS+** operates as the core of the upper software layers, enabling task concurrency and synchronization. This layer contains various utility libraries like *corePKCS11* for accessing encryption and secure element capabilities and the *coreMQTT* MQTT client. Mutual authentication and encrypted communication –mandatory for AWS IoT Core and GreenGrass CoreDevice support– are implemented through the *ARM mbedTLS* library.

The **Edge Nodes** have been configured by installing Raspberry Pi OS Lite[14] version 12 (*bookworm*), along with OpenJDK version 11.0.20, Redis 6.0, and the Greengrass IoT Client Software. AWS provides the needed services to deploy and configure the remaining components of the architecture. The setup requires the registration of *Field Devices*' and *Edge Nodes*' certificates within the AWS Console. Certificates for *Field Devices* are obtained from the boards' secure element, whereas *Edge Node certificates* can be directly generated from the AWS Console.

The developed **serverless functions** are integrated into AWS Lambda and AWS IoT Greengrass Cloud. Subsequently, triggers can be configured to automatically invoke these serverless functions when a message is received on the corresponding topic. Acting as a **Fleet Manager**, AWS IoT Greengrass Cloud manages the deployment of these functions and Greengrass Components to *Edge Nodes* suitable according to their computational capabilities.

[14] Raspberry Pi OS: `https://www.raspberrypi.com/software/operating-systems/`

Table 4.5: Model training performance: Edge and Cloud

Node	Training (s)	Evaluation (s)
Cloud	571.77	2.08
Edge1	2068.51	9.89
Edge2	2083.69	10.36

4.3.2 Results and discussion

The main results of the experimental campaign are presented below, comparing the proposed serverless architecture with the container-based OC one from [124]. Two main case studies have been addressed, named *Training* and *Inference* and described later on.

Training. This scenario evaluates both capability and performance of the proposed framework to train and deploy the machine learning model outlined in Section Section 4.3.1. In this process, the *Edge Consumer* within each *Edge Node* component accumulates historical labeled data over time, storing it in the respective *Edge Storage* via the *Training Data* topic. After this data gathering phase, each of the two *Edge Nodes* owns half of the full dataset. The subsequent interactions proceed as follows:

1. Each *Edge Node*, following a periodic schedule, autonomously trains its model instance using the locally stored data;
2. On a different, extended schedule, the *Cloud Node* sends a `Data Request` to each *Edge Node*. As a response, *Edge Nodes* compress and upload their data to the *Cloud Storage*;
3. Once all data are collected in the *Cloud Storage*, training begins using the full dataset. Upon completion, the model becomes available for *Edge Nodes* to download and replace their existing models with this –presumably more accurate– version.

Figure 4.3 presents the training and validation times, along with accuracy parameters such as the coefficient of determination (R^2) and the Mean Squared

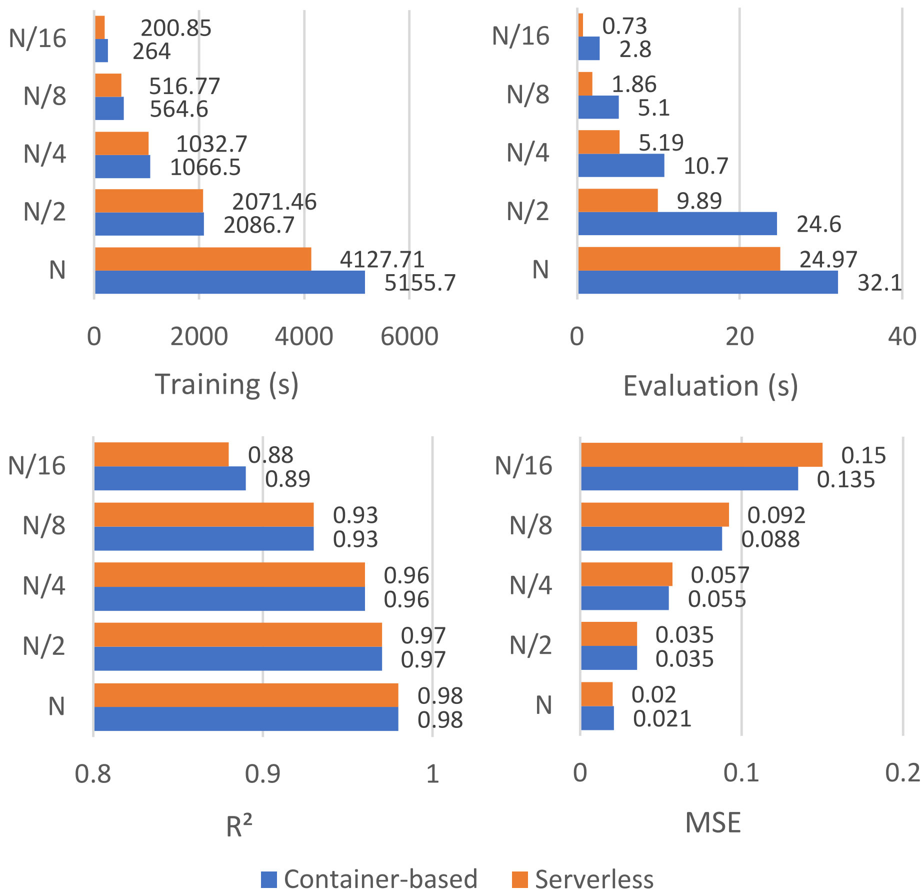


Figure 4.3: Comparing container-based and serverless architectures: model metrics by dataset size at the edge

Error (MSE). The orange bars illustrate the results obtained from the proposed serverless solution, while the blue bars report on the outcomes from the container-based architecture in [124]. Outcomes are taken across varying dataset sizes on the Raspberry Pi 3B+ Edge 2 device. Observations suggest that with smaller data subsets, like $N/8$ or $N/16$, prediction accuracy is not as effective as in the model trained on the full set of N samples, but it remains acceptable depending on the specific requirements and constraints of the application scenario, particularly if the provided subset adequately represents the comprehensive data [7]. In each real usage context, application developers and users will have to identify (un)acceptable accuracy levels, taking also into account cloud computation and bandwidth costs as well

Table 4.6: Dataset upload to cloud - Time and network activity

Node	Upload Time (s)	Upstream (KB)	Downstream (KB)
Edge 1	12.45692	36202.56	277.1318
Edge 2	11.96553	36193.16	286.3869

Table 4.7: Bandwidth consumption for Inference

Node	Upstream (KB)	Downstream (KB)
Cloud	6.02	4.29
Edge 1	2.65	1.36
Edge 2	2.66	1.36

as regulatory or industrial privacy requirements limiting data transmission to the cloud, *e.g.*, when sensor data are measured from patients through wearable healthcare devices or they may reveal the details of processes under trade, state or military secret. Generally speaking, however, experimental findings highlight the suitability of edge devices for making inferences using locally-trained models. While the MSE and R^2 values are closely matched in the two approaches, there is a noticeable processing time reduction with the new one. This improvement is likely due to the efficiency of scikit-learn in CPU usage compared to TensorFlow and Keras, given that the edge devices used in both testbeds are the same.

Table 4.5 reports on the training times of the *Cloud AI* and *Edge AI* functions. Notably, the *Cloud Node* is able to train a more precise model using the full dataset in a shorter time compared to the *Edge Nodes*, even considering the additional time required to upload the dataset to the cloud. However, the latter is dependent on bandwidth availability. Table 4.6 details the time needed for each *Edge Node* to completely upload their data to the *Cloud Storage*, along with the amount of sent (Upstream) and received (Downstream) information.

Inference. This scenario evaluates the time and latency involved in executing a machine learning prediction. Two *Field Devices* establish con-

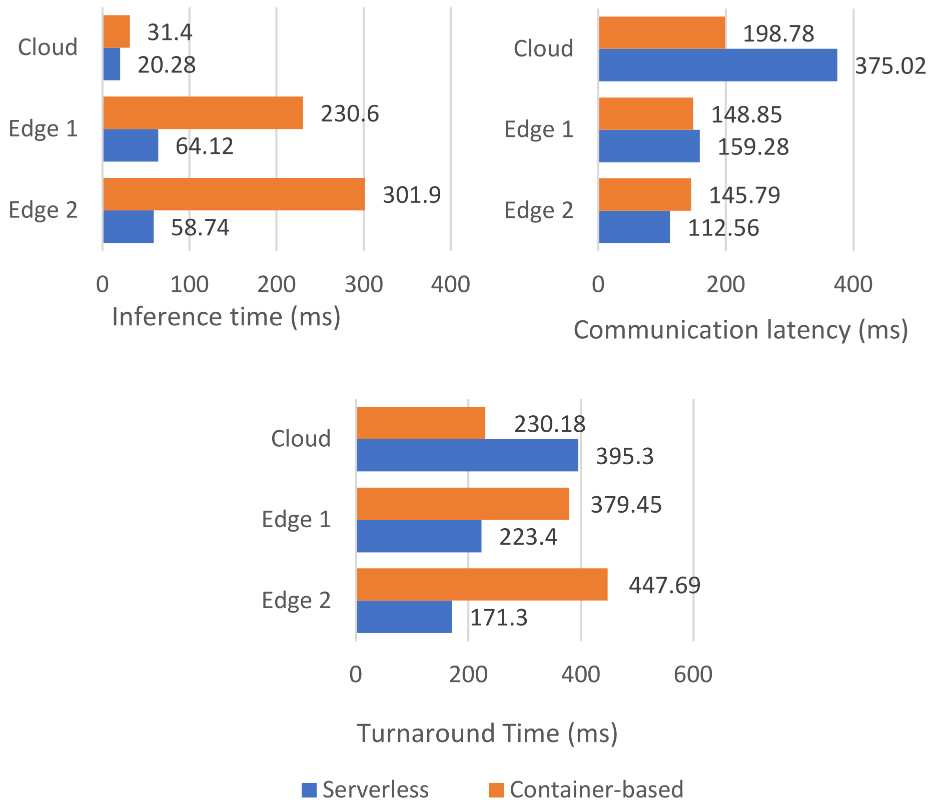


Figure 4.4: Comparing container-based and serverless architectures: inference time and communication latency

nections with the *Edge Nodes* and send 10 samples each to an *Edge Node's Data Consumer* function, which internally retrieves the ML model from the nearest storage and computes the regression value. The process is performed in three configurations:

1. **Edge 1**: *Field Device* connecting to the Raspberry PI 3B Edge Device;
2. **Edge 2**: *Field Device* connecting to the Raspberry PI 3B+ Edge Device;
3. **Cloud**: *Field Device* connecting to the most capable *Edge Node* (Raspberry PI 3B+) without local inference capabilities; in this case, the edge device acts only as a gateway to the cloud.

To enhance execution speed, a caching mechanism has been implemented to prevent repeatedly querying the storage to retrieve the machine learning model during each function execution. Inference times and communication latency are measured per sample, with the average results shown in Figure 4.4, while Table 4.7 refers to network bandwidth consumption. The blue bars represent the findings from the new experimental setup, while the orange bars provide results from [124] for a direct comparison. As expected, locally deployed serverless function optimizes communication bandwidth and reduces latency, but it requires sligthly more time for inference compared to cloud-based functions. Again, the new testbed exhibits a significant reduction in inference time across the board w.r.t. the container-based framework. On the other hand, for what concerns bandwidth, communication latency, and overall turnaround time, the performance enhancement from leveraging edge computing w.r.t. only using the cloud infrastructure becomes more evident for the proposed approach than the previous one. It can be deemed, even on the adopted small-scale testbed, that serverless computing makes the edge layer more efficient for prediction tasks, and thus serverless CEI can be considered more advantageous than cloud-only AI and containerized CEI approaches.

Chapter 5

Semantic-enhanced coordination in smart object network contexts

To assess the practicality, flexibility, and resilience of the proposed SWoE infrastructure across different scenarios, a series of carefully chosen case studies has been developed. These case studies cover a broad spectrum, ranging from widespread Web services to multi-domain service and resource marketplaces, and even down to nano-scale devices in smart cities. Each case study presented in this chapter demonstrates various aspects of the overall proposal, showcasing its adaptability, scalability, and utility in a wide range of circumstances.

The initial scenario introduces a cutting-edge semantic-based framework for dynamically adapting Quality of Experience (QoE) in Web multimedia streaming. In this context, reasoning is executed within the client Web Application via the WebAssembly port of the Tiny-ME reasoning engine, a significant new feature expected to have extensive applications in the future. The second investigation introduces a Web-based local event finder designed with privacy in mind, showcasing the SWoE stack suitability for client-side data retrieval and personalization functionalities in Rich Internet Applications (RIA). This example illustrates how the SWoE improves user experience by offering semantic features and intelligent information processing while maintaining user privacy with respect to conventional Web and social media applications. Privacy management is achieved through client-side data processing, which minimizes the need to share personal information with external servers, and by adopting secure data-handling practices to protect user identities and preferences during information retrieval and customization. The third case study explores the use of reasoning capabilities offered by Tiny-ME to handle crucial stages in the lifecycle of semantically annotated resource descriptions within a peer-to-peer network of intelligent agents. This example underscores the role of semantic web technologies in providing explanations during negotiations and ensuring the comprehensibility of discovery results. The final study examines a scenario within a smart city, illustrating how the SWoE, aided by Cowl's low memory usage, allows nano-scale sensors disseminated in urban areas to actively participate in a knowledge-oriented system. This integration supports improved urban mobility and land management by facilitating the collection, processing, and dissemination of semantically rich data, thus making city environments more efficient and responsive. Collectively, these case studies highlight SWoE's flexibility, demonstrating its practical implementation across a wide spectrum of applications from nano-sensors to the global Internet, emphasizing its extensive potential and impact on the broader information technology industry.

5.1 In the Web of Things

To demonstrate the feasibility and advantages of the SWoE technological stack in Web environments, two case studies have been developed: one centered on a knowledge-driven Quality of Experience (QoE) adaptation framework, and the other on the retrieval and preference-based ranking of local events.

5.1.1 Semantic-based QoE adaptation

Semantic web technologies can also be leveraged to dynamically adjust the QoE during the fruition of web multimedia streams. The standard for Dynamic Adaptive Streaming over HTTP, known as *MPEG-DASH* [194], has certain limitations, as it primarily focuses on network parameters without considering contextual factors that can impact QoE. To address this limitation and provide greater flexibility, a novel knowledge-driven QoE adaptation approach has been proposed. This approach relies on dynamic ontology-based annotation of the streaming context and semantic matchmaking with DASH reference profiles. A complete prototype of the proposed framework has been implemented to demonstrate its feasibility and a case study has been conducted to validate the concept.

Framework description. The reference architecture for the proposed system is depicted in Figure 5.1. At the core, the *resource server* is a standard HTTP-based Web server hosting the media content segmented according to the MPEG-DASH specification. It supports various video quality profiles and encodings. The *streaming client* application runs a MPEG-DASH compliant *media player*. In a Web-oriented scenario, the client is typically a browser equipped with a plug-in media player, usually implemented in JavaScript. This architectural approach also allows for seamless integration of QoE adaptation functionalities into applications across multiple platforms, *e.g.*, desktop PCs, notebooks, smartphones, tablets and smart TVs.

In greater detail, the client also comprises the following components: (i) a *context manager* responsible for gathering information that characterizes the viewing conditions; (ii) an *HTTP daemon* that handles communication with the resource server; (iii) a *QoE adaptation module*, which includes the matchmaking engine providing non-standard inferences to select the appropriate QoE profile. This adaptation module receives both context and network parameters from the streaming client and builds an annotated description according to the modeling guidelines defined later. It has been implemented using WebAssembly 1.0 (Wasm), a binary application format and portable compilation target for various programming languages. Wasm enables the deployment of Web components in both client and server applications. Its key advantages include full compatibility with major browser engines, standardized Web API interactions, optimized performance through efficient use of host hardware, and strict adherence to browser security policies, such

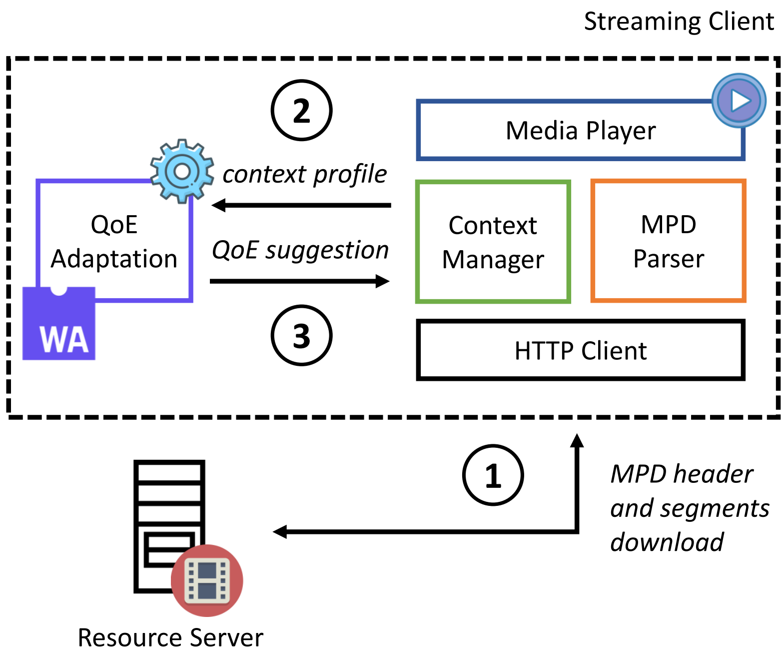


Figure 5.1: Semantic-based framework architecture for adaptation of QoE

as same-origin and permissions.

The client-side graphical user interface (GUI) is designed for simplicity and ease of use, presenting inference results clearly and enabling the user to manually select video quality settings. If the chosen quality is not suitable for the current context, the user is alerted with a warning message and an explanation of the conflicting factors.

Figure 5.1 also sketches the basic workflow for semantic-based QoE adaptation, which can be summarized as follows:

1. the streaming client loads the reference Web page and the media application begins video playback, either triggered by the user or automatically if `autoplay` is enabled;

2. the media player retrieves the MPD (Media Presentation Description) file from the resource server and parses it to get information about the segments composing the multimedia content;

3. contextual and network parameters are collected and sent to the QoE adaptation module. An OWL-based annotation is dynamically created according to the reference ontology and submitted to a matchmaking process against a set of QoE template profiles. To avoid unnecessary computation, a reasoning task is only started if the client detects a change in the observed parameters;

4. the matchmaking process returns the most suitable profile along with: (i) a relevance score depending on the semantic distance between the current context and the selected QoE profile; (ii) the settings values associated with the selected profile (*e.g.*, resolution, brightness, contrast, saturation); (iii) a list of compatible and contrasting features as identified by the applied inference;

5. according to the above results, the media player retrieves the next segment at the appropriate resolution and adjusts the video panel settings if necessary. An additional MPD header is downloaded from the resource server when a new video quality is recommended for the first time;

6. the explanation section of the interface displays information about the compatible and conflicting characteristics that contributed to the matchmaking outcome.

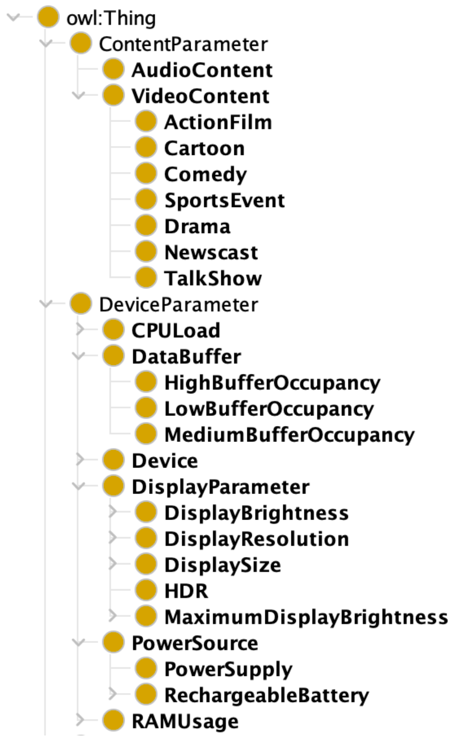
Steps 3-6 are repeated for all segments until the video playback ends.

Knowledge Graph modeling. To support semantic-based data annotation and understanding, a reference ontology $\mathcal{T}$, accessible at `http://swot.sisinflab.poliba.it/onto/semantic-qoe`, has been defined to represent the domain conceptualization described below. Specifically, two well-known RDF vocabularies are used as upper-ontologies, *i.e.*, DBpedia [106] (which models essential concepts related to media content and device/network parameters) and DogOnt [32] (which covers concepts for home environments, home/building automation, and appliances). The $\mathcal{T}$ is designed to be acyclic and is expressed in the moderately expressive $\mathcal{ALN}$ DL. The resulting Knowledge Graph (KG) includes: (i) a formal conceptual model of the media streaming domain (Terminological Box $\mathcal{T}$); (ii) assertions concerning *instances* (Assertion Box, ABox), including streaming profiles that define ideal QoE under specific usage conditions.

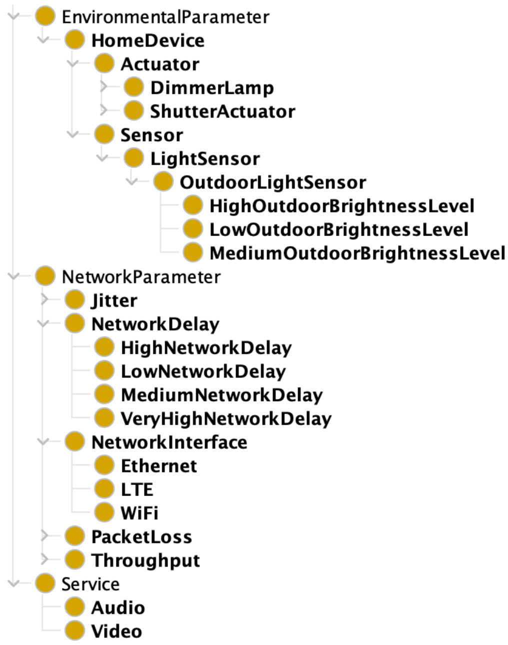
As shown in Figure 5.2, the proposed ontology models QoE elements based on the conceptualization proposed in [192]. Concepts are grouped into four main categories: content, device, environment, and network parameters. For each measurable feature (*e.g.*, buffer occupancy, network delay, battery power), $\mathcal{T}$ includes a hierarchical structure that forms a partonomy of the topmost concept. In this way, each parameter is modeled through a taxonomy that represents all significant value ranges and configurations within the domain of interest. When expanding the model with additional features, the depth of the hierarchy and the breadth of each level are determined by the knowledge modeler. These factors are typically proportional to either value ranges or specific subclasses relevant to each parameter, as well as the required level of detail in data representation.

In the proposed model, QoE guidelines are translated into a set of OWL individuals, with each streaming profile being linked to an instance within the reference KG. As depicted in Figure 5.3, each individual is defined by means of: (i) an OWL-based description expressed as the conjunction of *concepts*, usually combined using different *object properties* (*e.g.*, hasDeviceParameter) through existential and universal role quantification, as well as number restrictions; (ii) a set of *annotation properties* representing extra-logical attributes related to the suggested video settings (*e.g.*, brightness, resolution) aimed at ensuring optimal QoE. The KG currently includes an initial set of configuration parameters that have provided control over the modeling effort while enabling a full effectiveness evaluation of the proposed knowledge-based approach. By extending the KG, additional concepts and annotation properties can be modeled in order to further detail QoE profiles without changing the overall framework or the inference-based matchmaking process.

Automated reasoning. The QoE Adaptation module exploits the WebAssembly port of the Tiny-ME reasoner, as described in Section 3.2.4, to perform its reasoning tasks. Specifically, the matchmaker applies the *semantic*



(a) Content and device parameters



(b) Context and network parameters

Figure 5.2: Conceptual model of the media streaming domain

matchmaking framework (see Figure 3.11 in Section 3.2.2) to compute a score that reflects the semantic *affinity* between the OWL-based annotation of current client-side context and the available QoE profiles. Basically, a DL concept expression of contextual profile R is compared to QoE description S as in the flowchart in Figure 3.11, by means of the *Concept Contraction* (CC) and *Concept Abduction* (CA) non-standard, non-monotonic reasoning services. Both the results of CC ($\langle G, K \rangle$) and CA (H) can also be used to rank QoE profiles and to enable a refinement process where the user could act to adapt the context conditions and improve the media streaming experience, by *e.g.*, closing a concurrent application or switching from cellular to Wi-Fi connection. This procedure is applied iteratively across all QoE descriptions defined within the KG, comparing each description to the current context profile R. The QoE description with the highest rank is selected. Hence, the adopted approach: (i) produces a fine-grained logic-based profile ranking identifying the most appropriate QoE settings; (ii) detects possible

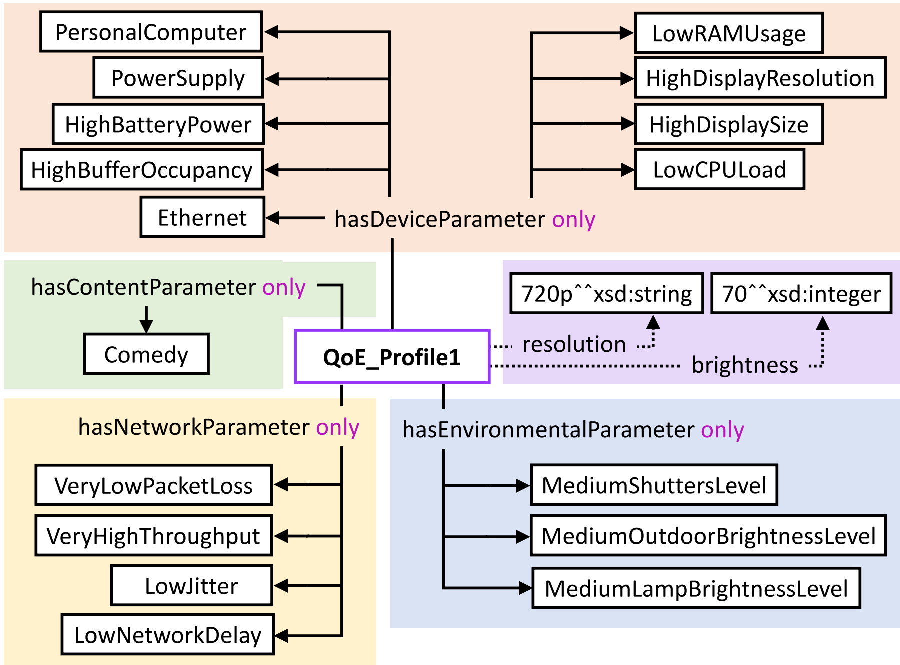


Figure 5.3: Example of QoE profile annotation

incompatibilities between a proposed streaming profile and contextual conditions; (iii) provides matchmaking outcomes which are both human-readable and machine-understandable, along with a rigorous logic-based explanation consisting of the concept expressions G and H.

Application scenario. An illustrative example is proposed hereafter to clarify the semantic-based inferences by presenting the QoE adaptation procedure in a basic usage scenario. Semantic-based annotations related to context profiles and QoE individuals are always verified by the matchmaking engine in order to detect inconsistencies within and between concept descriptions.

The system implementation is based on the following tools. The open-source Apache HTTP server (`https://httpd.apache.org`, version 2.4.46) has been used as resource server hosting the *Big Buck Bunny* (`https://peach.blender.org`) short movie, an open video project developed by the Blender Foundation and freely available under the Creative Commons Attribution 3.0 license. In particular, the server exposes a segmented version of the movie avail-

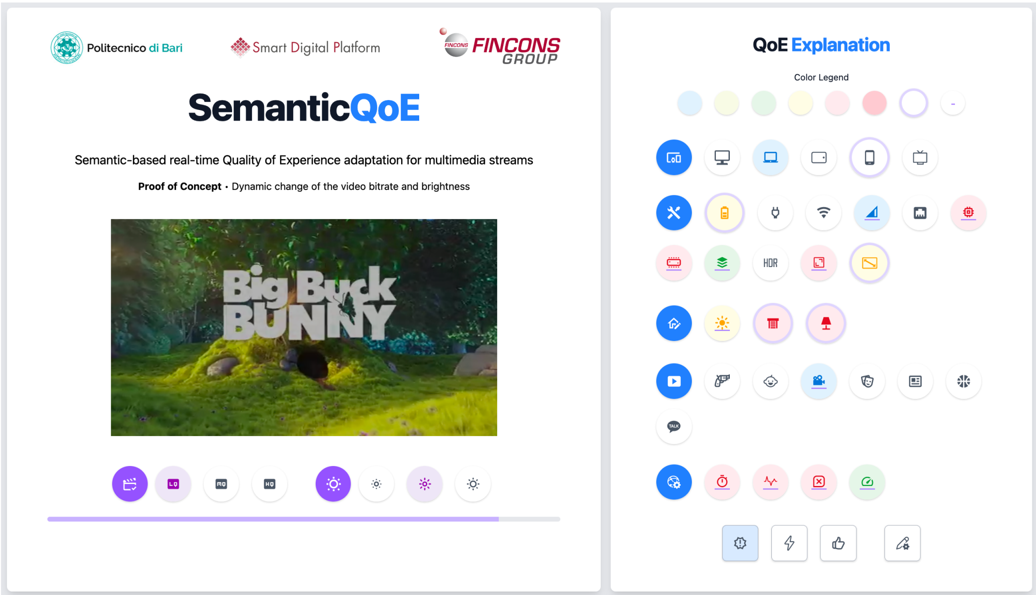


Figure 5.4: Streaming client GUI

able at `http://ftp.itec.aau.at/datasets/mmsys12/BigBuckBunny/`, structured according to the MPEG-DASH specification. It consists of segments with a length of 2 seconds and an increasing bit rate from 200 to 2000 kbps. Mozilla Firefox desktop edition (`https://www.mozilla.org/firefox`, version 88.0.1) has been selected as the reference browser for the client application.

The user interacts with the GUI, shown in Figure 5.4, via the browser. This interface is an HTML page linking the modules described in Figure 5.1 (*i.e.*, HTTP client, MPD parser, and context manager), developed in JavaScript. The QoE adaptation module is implemented in WebAssembly as previously described.

The GUI consists of two main sections. On the left hand side, it includes: a Web player supporting the MPEG-DASH protocol; two sets of violet icons representing resolution and brightness related to the QoE profile selected through the matchmaking procedure; a rank bar that displays the semantic-based similarity score. On the right hand side, a simple explanation section summarizes parameters of the current context profile. They are associated to illustrative icons and grouped by category (blue background) following the same taxonomy of the ontology. Only the device type is shown as a separated element, in order to improve its visibility on the GUI. The explanation section serves a dual purpose. Basically, it shows the status of various parameters exploiting a color-coded legend: icons with a light blue background indicate active on/off functionalities (*e.g.*, Wi-Fi network access) or equipment/stream properties (*e.g.*, device type), while a gradient color palette (ranging from light green to red) highlights detected features with a value range from very low to very high. In both cases, the *context manager* manages the correspondences between GUI elements and concepts modeled in the KG. Hovering the mouse pointer over an icon, a tooltip text also appears showing the name of the related OWL class. For example, following the scenario in Figure 5.4, the context profile R is expressed (in OWL 2 Manchester syntax [82] and with respect to the reference KG) as:

R ≡ (hasContentParameter **only** Comedy) **and** (hasDeviceParameter **only** (MediumBatteryLevel **and** LTE **and** HighCPULoad **and** HighDisplayResolution **and** HighRAMUsage **and** LowBufferOccupancy **and** MediumDisplaySize **and** Notebook)) **and** (hasEnvironmentalParameter **only** (HighShutterLevel **and** HighLampBrightnessLevel **and** MediumOutdoorBrightnessLevel)) **and** (hasNetworkParameter **only** (HighPacketLoss **and** HighNetworkDelay **and** HighJitter **and** LowThroughput))

Basically, the client detects high local processing and memory usage, along with network congestion and high room brightness while the user is watching a comedy video. The original QoE individual reported in Figure 5.3 includes several missing or conflicting features that do not align well with the current context. As a result, the semantic-based matchmaking process is invoked, which identifies the following QoE annotation S with the highest logic affinity (about 89%) for the detected context.

S ≡ (hasContentParameter **only** Comedy) **and** (hasDeviceParameter **only** (LowBatteryLevel **and** LTE **and** HighCPULoad **and** HighDisplayResolution **and** HighRAMUsage **and** LowBufferOccupancy **and** LowDisplaySize **and** Smartphone)) **and** (hasEnvironmentalParameter **only** (MediumOutdoorBrightnessLevel **and** Medi-

`umLampBrightnessLevel and MediumShutterLevel)) and (hasNetworkParameter only (HighPacketLoss and HighNetworkDelay and HighJitter and LowThroughput))`

The QoE profile S ensures an improved video experience, although a partial match is obtained also in this case. In particular, S recommends a low video quality suitable for smartphones and devices with limited hardware capabilities and constrained network bandwidth. Additionally, it suggests a medium video brightness level, taking into account the room's internal and outdoor light conditions, as well as partially opened window shutters. Matchmaking results, detailed below, are also reported in the GUI of Figure 5.4 through the aforementioned components.

`G (Give-up)` $\equiv$ `(hasEnvironmentalParameter only (MediumShutterLevel and MediumLampBrightnessLevel)) and (hasDeviceParameter only (LowBatteryLevel and LowDisplaySize))`

`K (Keep)` $\equiv$ `(hasContentParameter only Comedy) and (hasDeviceParameter only (LTE and HighCPULoad and HighRAMUsage and LowBufferOccupancy and HighDisplayResolution and Smartphone)) and (hasNetworkParameter only (HighNetworkDelay and HighJitter and HighPacketLoss and LowThroughput)) and (hasEnvironmentalParameter only MediumOutdoorBrightnessLevel)`

`H (Hypothesis)` $\equiv$ `(hasDeviceParameter only Smartphone)`

In the GUI icons, a horizontal bar highlights active features that are compatible with the selected QoE profile, *i.e.*, the K part of the related OWL annotation. For example, in Figure 5.4 the icons for CPU load and memory occupancy are both red –as a high level has been detected by the client– and underlined, since they match the reference profile. The buffer occupancy icon is also underlined but displayed in green, reflecting a low level. Missing or contrasting features (*i.e.*, H and G, respectively) detected through the Concept Abduction or Concept Contraction algorithms are depicted using a bold violet border. Tooltips provide additional information, detailing the inference results and explaining which concepts would ensure a full match with the QoE annotation. A legend at the top of the explanation section summarizes the meaning of all graphical elements.

For experimental purposes, the right-hand section of the GUI also functions as an interactive configuration panel in the prototype. Users can click on a feature to open a popup window where they can adjust its value from a list of available options. This makes it possible to simulate variations of the context parameters and observe how the QoE adaptation module reacts to the updated conditions. In the bottom part of the section, four buttons enable users to activate a set of predefined scenarios that require different QoE configurations or to apply a given user-defined context profile (via a button with a pencil icon).

Finally, a user can also set a particular video resolution or brightness by clicking on the corresponding icons in the left-hand section. In this case, the logic-based automatic adaptation of the QoE will be temporarily disabled, and the matchmaking process will only be used to compare the configuration selected by the user with the client context conditions, displaying the relevance rank outcome in the bottom horizontal bar. In practical scenarios, users often select inappropriate streaming settings. This feature provides useful support by immediately highlighting the reasons for the configuration mistake via the explanation GUI section. To re-enable the automatic QoE adaptation, users can simply click on the pencil icon.

5.1.2 Privacy-conscious (mobile) Web

On modern Web platforms, users frequently share their profile details to receive personalized recommendations for events, products, and services. The proposed approach, depicted in Figure 5.5, exploits the WebAssembly version of the Tiny-ME reasoner –outlined in Section 3.2.4– to deliver a flexible solution through four key steps:

1. a mobile Web application collects generic data from the Web;

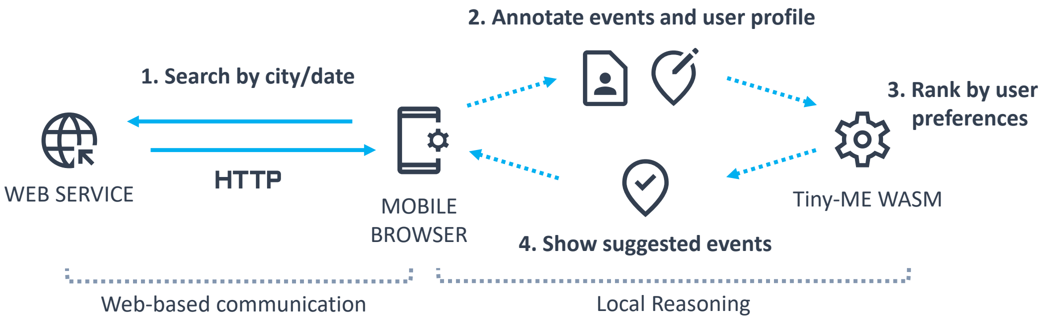


Figure 5.5: Architecture and workflow of the proposed Web application.

2. this data is subsequently annotated according to an OWL domain ontology;

3. the annotated data is used to identify resources that best align with the user's interests and preferences;

4. the results are then displayed in the user interface, enabling the user to make selections or refine their queries.

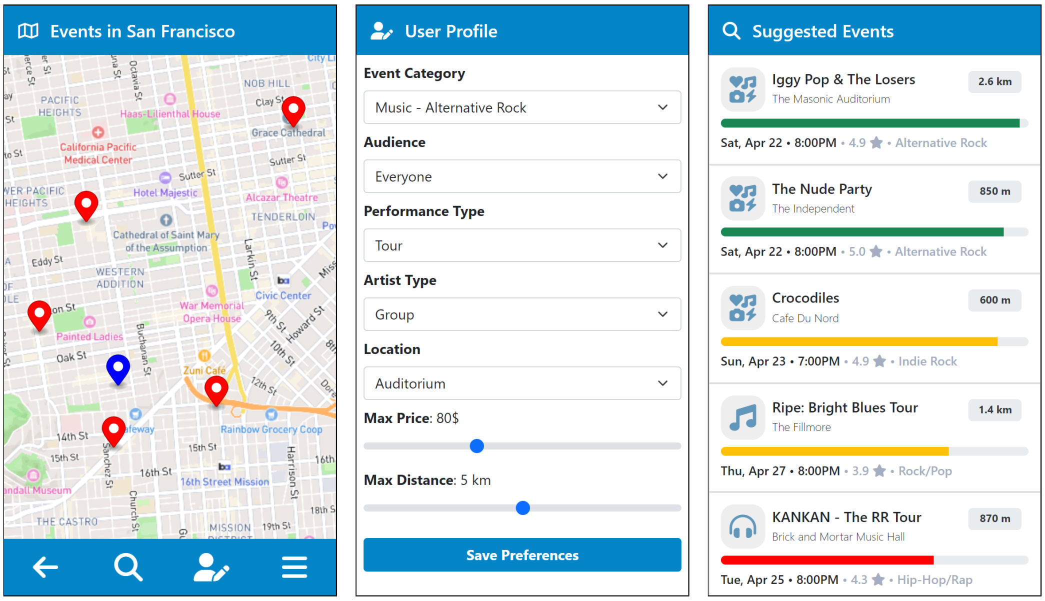
A significant advantage of this approach is that it requires a remote server query only during the initial step, with all subsequent processing carried out locally on the user's mobile device. This method not only guarantees the protection of private data, but also removes the typical latency associated with interacting with a remote reasoning engine.

Consider the following scenario: *While on vacation in San Francisco, Martina is interested in purchasing a ticket for a musical event, spending no more than $80.00. She prefers alternative rock bands and would like to attend a tour date instead of a one-time concert.*

The framework illustrated in Figure 5.5 allows users to obtain information about local events through a Web application on their mobile browser. The data for this case study originates from *Ticketmaster*[1] using their public RESTful API[2]. Initially, as depicted in Figure 5.6a, users retrieve a list of

[1]https://www.ticketmaster.com

[2]https://developer.ticketmaster.com/products-and-docs/apis/getting-started/



(a) Available events. **(b)** User profile. **(c)** Suggested events.

Figure 5.6: Screenshots of the Web application.

nearby events based on basic filters such as city and date range. This information is generic and does not reflect individual preferences. To personalize this data, users can select various features and preferences to create their private personal profile (Figure 5.6b) on the device, which is then converted into an ontology-based semantic annotation. Figure 5.7 shows the TBox concepts corresponding to the categories provided by the service API. Below are the specific details of the selected preferences:

```
User_Profile: Event and (hasAudience only EveryOne) and (hasPrice only
float[<=80.0]) and (hasStyle only Tour) and (hasCategory only Alterna-
tive_Rock) and (hasType only Group) and (hasCategory min 1) and (hasStyle
min 1) and (hasAudience min 1)
```

After applying an initial filter to events within a 5 km radius of the mobile device's location and considering only those occurring within the current week, the user profile is then compared to the following semantically annotated event descriptions through semantic matchmaking:

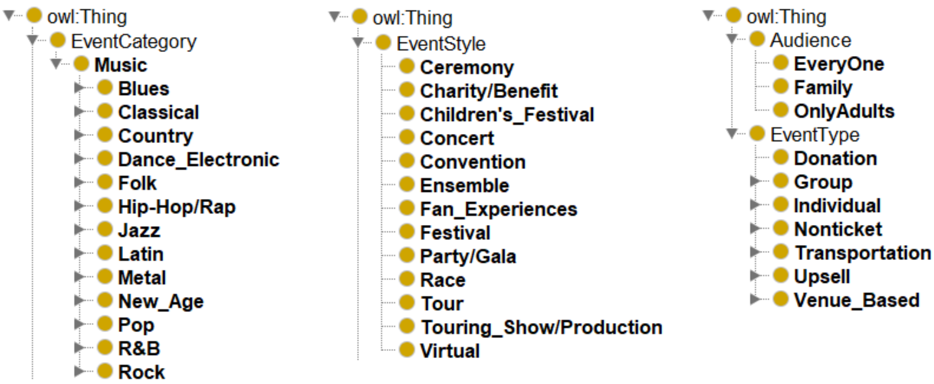


Figure 5.7: Reference ontology for the event finder application.

Crocodiles: Event and (hasAudience only EveryOne) and (hasPrice only float[>=163.0,<=180.0]) and (hasStyle only Tour) and (hasType only Band) and (hasCategory only Indie_Rock)

Iggy_Pop_and_The_Losers: Event and (hasAudience only EveryOne) and (hasPrice only float[>=50.0,<=100.0]) and (hasStyle only Tour) and (hasType only Band) and (hasCategory only Alternative_Rock)

KANKAN-The_RR_Tour: Event and (hasAudience only EveryOne) and (hasPrice only float[>=25.0,<=105.0]) and (hasStyle only Concert) and (hasType only Performer) and (hasCategory only Hip_Hop_Rap)

Ripe-Bright_Blues_Tour: Event and (hasAudience only EveryOne) and (hasPrice only float[>=27.0,<=27.0]) and (hasStyle only Tour) and (hasType only Band) and (hasCategory only Pop_Rock)

The_Nude_Party: Event and (hasAudience only EveryOne) and (hasPrice only float[>=20.0,<=20.0]) and (hasStyle only Concert) and (hasType only Band) and (hasCategory only Alternative_Rock)

The returned list of events, shown in Figure 5.6c, is ranked according to the following *utility function*:

$$u(R, C) = 100 \cdot [1 - \frac{s\_penalty(R, C)}{s\_penalty(R, \top)} \cdot (1 + \frac{distance(R, C)}{max\_distance})] \tag{5.1}$$

where $s\_penalty(R, C)$ is the result of the semantic matchmaking task (detailed in Section 3.2.2) and it represents the semantic distance between user profile R and event annotation C. This distance is normalized by $s\_penalty(R, \top)$, which measures the semantic distance between R and *top*, and is only dependent on the ontology structure. The $distance(R, C)$ context parameter is included to exploit the event geographical distance as weighting factor. The utility function also converts the semantic distance value into a more intuitive percentage score, as shown in Table 5.1. It can be seen that if $s\_penalty(R, C) = 0$ then the final score is 100%, regardless of the geographical distance. According to user preferences and event descriptions, the top-ranked event, as listed in Figure 5.6c, is the individual `Iggy_Pop_and_The_Losers`, despite its greater physical distance compared to other available options.

Table 5.1: Musical events ranked by the utility function.

Musical events	***s_penalty(R,C)***	***distance(R,C)***	***u(R,C)***
Iggy Pop & The Losers	0.20	2.60 km	97.97 %
The Nude Party	1.00	0.85 km	93.22 %
Crocodiles	1.51	0.60 km	89.81 %
Ripe Bright Blues Tour	4.00	1.40 km	72.59 %
KANKAN - The RR Tour	6.24	0.87 km	57.68 %

5.2 In peer-to-peer electronic marketplaces

The technological stack developed and outlined in Chapter 3 is also well suited for multi-domain service/resource marketplace environments. In such environments, Semantic Web technologies - grounded on formal Knowledge Representation and Reasoning - can enhance information interoperability and empower smart software agents distributed in a peer-to-peer network to automate data-driven activities. To illustrate this, a prototypical semantic-enanched *ridesharing* marketplace named *RideMATCHain*[3] has been developed. This platform supports comprehensive specifications of vehicle and driver attributes, as well as passenger preferences [73].

Let us suppose a heterogeneous group of people is sending requests from different places and at various times to a ridesharing service in order to reach their destinations. For example, suppose that there are currently five drivers available for ridesharing through the service with their cars. The task is to match each passenger with the vehicle that best meets their preferences and requirements, while also considering compatibility factors. Both passengers and vehicles are described according to a prototypical ontology, whose conceptual taxonomy is depicted in Figure 5.8. Whenever feasible, the classes have been reused from *SUMO-OWL* (the OWL translation of the Suggested Upper Merged Ontology) [126], *e.g.*, `Automobile`, `AgeGroup`.

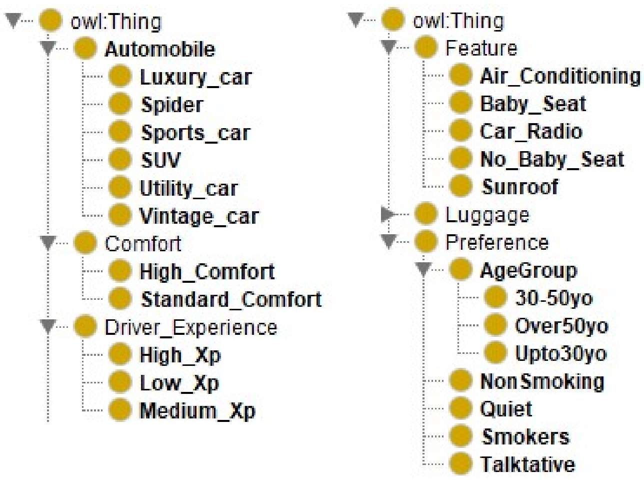


Figure 5.8: Reference ontology top-level classes

Each available ride is considered a resource, represented as a distributed asset with a semantic annotation split into two parts: the first one is constant over time and summarizes the set of vehicle features relevant to the service and the driver's profile; the second part provides details on the available space

[3] `https://github.com/sisinflab-swot/ridematchain`

for passengers and their luggage. An excerpt of the annotations linked to each car is shown below in Manchester syntax [82]:

```
SUV_Car: SUV and (accepts only NonSmoking) and (hasFeature only (Car_Radio and Air_Conditioning and Baby_Seat)) and (comfortLevel only High_Comfort) and (driverExperience only High_Xp) and (availableSeat only int[>=6,<=6]) and (availableCapacity only float[>=650.0,<=650.0])

Big_CityCar: Utility_Car and (accepts only (NonSmoking and Quiet)) and (hasFeature only (Car_Radio and Air_Conditioning and No_Baby_Seat)) and (comfortLevel only Standard_Comfort) and (driverExperience only High_Xp) and (availableSeat only int[>=4,<=4]) and (availableCapacity only float[>=320.0,<=320.0])

Small_CityCar: Utility_Car and (hasFeature only (Sunroof and No_Baby_Seat)) and (comfortLevel only Standard_Comfort) and (driverExperience only High_Xp) and (availableSeat only int[>=1,<=1]) and (availableCapacity only float[>=200.0,<=200.0])

Luxury_Sedan: Luxury_Car and (accepts only NonSmoking) and (hasFeature only (Sunroof and Baby_Seat and Car_Radio)) and (comfortLevel only High_Comfort) and (driverExperience only Low_Xp) and (availableSeat only int[>=4,<=4]) and (availableCapacity only float[>=490.0,<=490.0])

Spider_Car: Spider and (accepts only Talktative) and (hasFeature only (Car_Radio and No_Baby_Seat)) and (comfortLevel only Standard_Comfort) and (driverExperience only Medium_Xp) and (availableSeat only int[>=1,<=1]) and (availableCapacity only float[>=250.0,<=250.0])
```

Resource descriptions have been registered, distributed and synchronized as assets among all participants in the peer-to-peer network. For the sake of simplicity, to show how the ridesharing service works, let us consider a single step iteration. *A user, headed downtown with her baby, is looking for a car with two seats, one of which equipped with baby seat. She would like to travel* *comfortably in an air-conditioned, quiet and non-smoking environment.* In formulas:

```
R ≡ (accepts only (NonSmoking and Quiet)) and (hasFeature only (Baby_Seat and Air_Conditioning)) and (comfortLevel only High_Comfort) and (driverExperience only High_Xp) and (availableSeat only int[>=2])
```

To accommodate both passenger-car and passenger-passenger constraints, the framework invokes the semantic matchmaking task detailed in Section 3.2.2. After a pre-filtering step, exploited to discard cars with a heading diverging from that of the passenger, the matchmaking identifies which vehicles of the fleet best meet the specified requirements. The returned list of suitable resources is ranked according to the utility function described by Equation 5.1. Also in this case, the $distance(R, C)$ contextual data is used to take into account the car geographical distance, combined as weighting factor. Outcomes, ranked according to the utility function, are summarized in Table 5.2.

Table 5.2: Matchmaking output for the first iteration

	SUV_Car	Small_CityCar	Big_CityCar	Luxury_Sedan	Spider_Car
$u(R, C)$	**90.2**	76.2	64.3	56.1	31.5

As evident, SUV_Car and passenger semantic descriptions are very similar: all atomic concepts, universal quantifiers, and unqualified number restrictions on roles are compatible. On the other hand, passenger requirements are not compatible with the Small_CityCar: the user can get an explanation for this by means of the *Concept Contraction* reasoning task:

```
G (Give up): (hasFeature only (Baby_Seat)) and (availableSeat only int[>=2])

K (Keep): (accepts only (NonSmoking and Quiet)) and (hasFeature only Air_Conditioning) and (comfortLevel only High_Comfort) and (driverExperience only High_Xp)
```

The best match for the passenger is with SUV_Car, so she is assigned to this vehicle. Upon confirmation of the passenger/car association, the passenger's preferences (modelled in the filler of the `accepts` object property) are appended in conjunction with the SUV annotation. As a consequence, also the semantic description of the selected resource is updated (modifications are underlined):

```
SUV_Car: SUV and (accepts only (NonSmoking and Quiet)) and (hasFeature
only (Car_Radio and Air_Conditioning and Baby_Seat)) and (comfortLevel
only High_Comfort) and (driverExperience only High_Xp) and (availableSeat
only int[>=4,<=4]) and (availableCapacity only float[>=350.0,<=350.0])
```

The selection transaction triggers a renewal of the peer-to-peer network internal status, enabling the system to re-evaluate and update the resource allocations. Following this update, the allocation task is repeated for the remaining passengers, ensuring that any incompatibility constraints between passengers and vehicles are identified and addressed. This process continues until all passengers are matched with suitable vehicles that meet their specified requirements and preferences.

5.3 In smart city road management

Intelligent road management systems are transforming the way traffic is monitored and controlled on roads and urban streets [153, 152]. This case study illustrates how Cowl can aid in managing and sharing data from various smart devices to optimize road management, improve traffic flow, and enhance public safety. In the reference smart city scenario represented in Figure 5.9, urban areas are equipped with smart devices embedded in the road surface, including accelerometer, temperature, pressure, and vibration sensors, designed to detect traffic density [223] and environmental conditions. These road sensors also communicate with additional sensor-equipped IoT devices, *e.g.*, smart tires of vehicles, pedestrians' smart shoes, smartphones and wearables, and Unmanned Aerial Vehicles (UAVs, a.k.a. drones), to collect

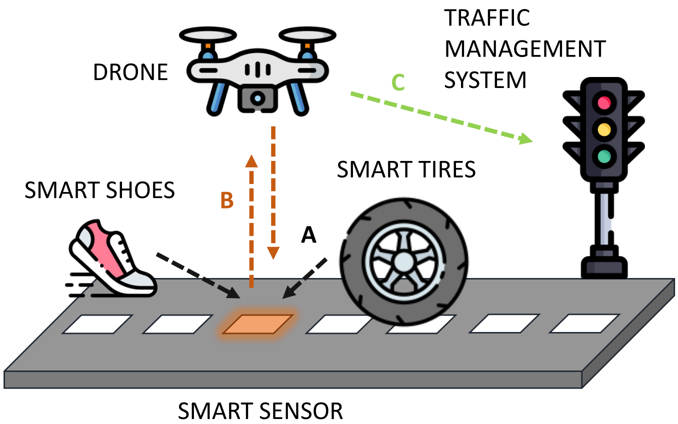


Figure 5.9: Reference scenario for smart road management.

and exchange real-time contextual data using a standardized communication protocol. The *Constrained Application Protocol* (CoAP) [34] has been chosen for this case study, as it is an application layer protocol specifically designed for networks of objects with limited computational, memory, and bandwidth resources. It was preferred over MQTT due to its lower bandwidth requirements and suitability for non-time-critical applications, as highlighted in [215, 95]. CoAP is particularly useful in SWoT/SWoE scenarios, such as collaborative sensor networks, where heterogeneous data needs to be aggregated from distributed nodes to infer events [174, 172]. The three-step process illustrated in Figure 5.9 is described in detail below.

Step A: interactions with road surfaces. As both pedestrians and vehicles move along the equipped road, smart sensors continuously gather data from the road surface. Specifically, self-sustaining triboelectric pressure-velocity nanosensors [217], along with vibration sensors triggered by the motion of vehicles, can be used to detect traffic type and intensity using Machine Learning algorithms [140]. Periodically, Cowl updates and annotates detected events and related contextual data within each smart sensor. This process generates numerous dynamic, ubiquitous, geo-referenced fragments of a knowledge base, each containing OWL axioms that describe the local state of specific road sections at a given time. Figure 5.10 illustrates an instance of data annotation involving a smart sensing node equipped with a temperature

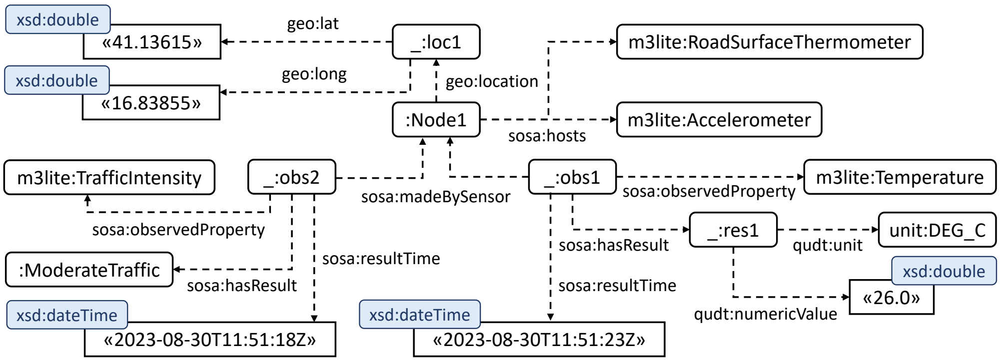


Figure 5.10: Example of road context annotation.

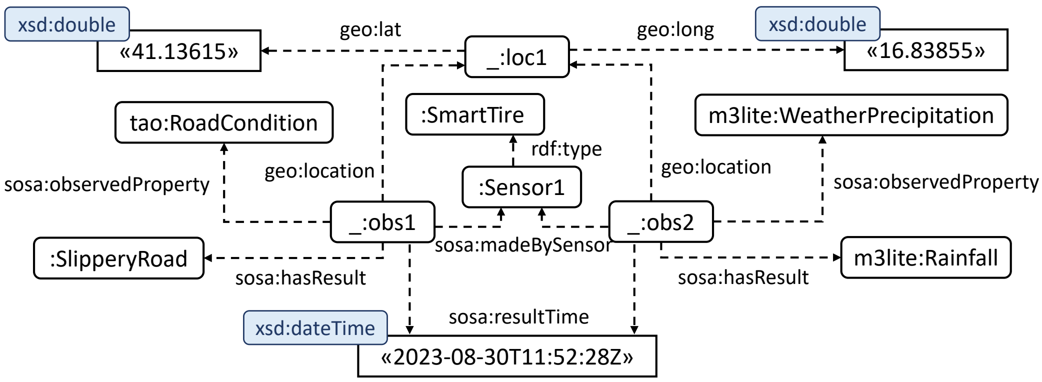


Figure 5.11: Example of annotation received from a smart tire.

sensor and an accelerometer. The *M3-lite* [3] ontology serves as the upper ontology for modeling device types, measurement parameters, and properties. The *W3C Geospatial* vocabulary [113] specifies the geographic coordinates of each node, while the *Sensor, Observation, Sample, and Actuator (SOSA)* ontology [87] provides the reference specification for defining observations and detected events.

The data collected by external smart devices passing by a road node can further enhance event annotations. In particular, several manufacturers are currently working on developing intelligent tires equipped with communication capabilities to share sensor data with both roadside infrastructures and the Electronic Control Unit (ECU) of the car. Smart tires generally have the capability to detect important road information (*e.g.*,, road surface conditions, weather conditions, and the risk of aquaplaning) to help avoid critical driving

```
// 1. Request from the road node to the tire
CON [id=<0xbc90>, token=<0x71>]
GET /sensor_data
```

```
// 2. Response from the tire
ACK [id=<0xbc90>, token=<0x71>]
CODE 2.05 Content
PAYLOAD
...
ObjectPropertyAssertion(sosa:observedProperty _:obs2 m3lite:Rainfall)
ObjectPropertyAssertion(sosa:hasResult _:obs2 _:res1)
ObjectPropertyAssertion(qudt:unit _:res1 unit:MilliM-PER-DAY)
DataPropertyAssertion(sosa:resultTime _:obs2
    "2023-08-30T11:51:23Z"^^xsd:dateTime)
DataPropertyAssertion(qudt:numericValue _:res2 "5.0"^^xsd:double)
...
```

Listing 2: CoAP-based interaction between a road node and a sensorized tire.

situations. When a vehicle equipped with smart tires approaches a road section with an integrated smart node, the pressure sensor identifies the presence of the vehicle, triggering the road node to initiate a CoAP-based data exchange to retrieve information from the tire. Figure 5.11 illustrates an example of an OWL-based annotation received from a sensorized tire, which also includes concepts related to the transportation domain modeled by the *Transport Administration Ontology (TAO)* [135]. This annotation is sent by the tire when it receives a CoAP GET request for the relevant sensor data resource, as illustrated in Listing 2. The smart road node collects data from various vehicles and IoT devices as they pass through the area. Then it processes and integrates this information using Cowl, creating a progressively detailed representation of the surrounding traffic and road conditions. This processed data is subsequently sent back to pedestrians and vehicles through CoAP, enhancing their knowledge graphs and enabling more accurate real-time inferences for automated decision-making.

Step B: interaction between road and drone. As illustrated in Figure 5.9, smart road nodes can also communicate with urban monitoring drones equipped with high-definition cameras, environmental sensors, and communication systems. Each drone follows a predetermined flight path, guided by Global Navigation Satellite Systems (GNSS) and remote control.

```
// 1. Original knowledge graph of the road sensor
...
ObjectPropertyAssertion(sosa:observedProperty _:obs2 m3lite:Rainfall)
ObjectPropertyAssertion(sosa:hasResult _:obs2 _:res1)
ObjectPropertyAssertion(qudt:unit _:res1 unit:MilliM-PER-DAY)
DataPropertyAssertion(sosa:resultTime _:obs2
    "2023-08-30T11:51:23Z"^^xsd:dateTime)
DataPropertyAssertion(qudt:numericValue _:res2 "5.0"^^xsd:double)
...
```

```
// 2. Request from the UAV to the road sensor
CON [id=<0xbc91>, token=<0x72>]
POST /sensor_data
PAYLOAD
...
ObjectPropertyAssertion(sosa:observedProperty _:obs2 m3lite:Rainfall)
ObjectPropertyAssertion(sosa:hasResult _:obs2 _:res1)
ObjectPropertyAssertion(qudt:unit _:res1 unit:MilliM-PER-DAY)
DataPropertyAssertion(sosa:resultTime _:obs2
    "2023-08-30T11:53:15Z"^^xsd:dateTime)
DataPropertyAssertion(qudt:numericValue _:res2 "10.0"^^xsd:double)
...
```

```
// 3. Response from the road sensor
ACK [id=<0xbc91>, token=<0x72>]
2.04 Changed
```

```
// 4. Updated knowledge graph of the road sensor
...
ObjectPropertyAssertion(sosa:observedProperty _:obs2 m3lite:Rainfall)
ObjectPropertyAssertion(sosa:hasResult _:obs2 _:res1)
ObjectPropertyAssertion(qudt:unit _:res1 unit:MilliM-PER-DAY)
DataPropertyAssertion(sosa:resultTime _:obs2
    "2023-08-30T11:53:15Z"^^xsd:dateTime)
DataPropertyAssertion(qudt:numericValue _:res2 "7.5"^^xsd:double)
...
```

Listing 3: CoAP-based interaction between the UAV and a road node.

When a drone passes over individual road sensors, it acquires real-time road condition via CoAP. Simultaneously, the UAV's onboard sensors capture additional contextual data –acquired by processing images from the onboard cameras and environmental sensors– which can be annotated and transmitted to road sensors through CoAP messages. An instance of this interaction can be seen in Listing 3, where one of the smart road nodes initially measures a rainfall intensity of 5 mm per day. As the UAV flies over, it detects a rainfall intensity of 10 mm per day and sends a POST request to the road sensor, which then updates its reading by combining the two values (in this case, by averaging them).

Step C: interaction between drones and traffic management systems. Finally, once the UAV comes within range of a road-side unit part of the distributed urban traffic management system (TMS), it sends collected data for real-time urban condition analysis. An example of this interaction is shown in Listing 4, where the drone sends a CoAP PUT request to a TMS unit, which updates the system's geographic database (such as a PostGIS[4] instance), exposed as a CoAP endpoint. By using Cowl to query its internal knowledge graph, the drone can obtain the geographic coordinates of the collected data: all gathered information –regarding things like traffic level and type, road surface condition and wetness, and weather conditions– is extracted and transmitted. Advanced TMS devices can then perform more complex (and computationally intensive) analyses on the overall data, producing reports and alerts which are then distributed to relevant city department policy-makers, emergency responders, and connected vehicles. This analysis helps optimize traffic flow through connected traffic lights and signals, reduces congestion, and enhances road safety. Additionally, it enables rapid deployment of emergency services when necessary.

Data recorded by road sensors and drones can be leveraged in an Urban Digital Twin [218] control center for long-term analysis and urban planning. This integration offers valuable insights for optimizing traffic flow and prioritizing preventive road maintenance. The interaction between smart road

[4]PostGIS home: `https://postgis.net`

```
// 1. Request from the UAV to the TMS
CON [id=<0xbc92>, token=<0x73>]
PUT /geo_data/41.13615/16.83855
PAYLOAD
...
ObjectPropertyAssertion(sosa:observedProperty _:obs1 m3lite:TrafficIntensity)
ObjectPropertyAssertion(sosa:hasResult _:obs1 :ModerateTraffic)
DataPropertyAssertion(sosa:resultTime _:obs1
    "2023-08-30T11:51:18Z"^^xsd:dateTime)
...
```

```
// 2. Response from the TMS
ACK [id=<0xbc92>, token=<0x73>]
2.01 Created
```

Listing 4: CoAP-based interaction between the UAV and a TMS unit.

infrastructures and Cowl-powered drones demonstrates the advantages of combining pervasive data collection and annotation with distributed aerial surveillance to develop a comprehensive and adaptable urban monitoring system. This approach can ultimately alleviate road congestion, reduce traffic-related delays, and minimize atmospheric and noise pollution, thereby improving the overall safety and quality of life of city residents.

Conclusion and perspectives

Semantic-based approaches provide structured and unambiguous information annotation capabilities, which enhance interoperability and enable advanced data integration. Agents and protocols running on networked objects can be designed to autonomously infer the required or available features in a fully decentralized and collaborative way, leveraging algorithms and frameworks that operate across a spectrum of devices, from powerful cloud infrastructures to highly constrained micro- and nano-devices.

In this context, the proposed research has focused on the development and application of general-purpose knowledge representation and automated reasoning techniques and methodologies in environments that require the co-operation of multiple (homogeneous or heterogeneous) agents for the execution of a specific task or mission. The developed technology stack, including Cowl for knowledge access and manipulation, Tiny-ME for pervasive reasoning, evOWLuator for methodical evaluations, and a sensor-driven cloud-edge intelligence framework for machine learning tasks collaboration, showcases a holistic strategy for achieving the vision of the Semantic Web of Everything. Each component is crucial in linking theoretical concepts with practical implementations in various computing platforms.

Cowl exemplifies how semantic technologies can be effectively employed in resource-constrained environments through innovative methodologies, architectural designs, and optimizations, while remaining flexible enough to provide high-performance access to knowledge graphs on more capable platforms. *Tiny-ME* provides a robust reasoning framework built on a moderately expressive fragment of OWL 2, ensuring efficient functionality across a range of platforms, from cloud-based microservices to smaller edge devices and embedded boards. This adaptability allows Tiny-ME to provide essential reasoning capabilities for the SWoE, regardless of the computational resources available on each device. evOWLuator complements this technology stack by offering a structured framework for assessing the performance and efficiency of Semantic Web technologies, with a particular focus on energy consumption and remote inference capabilities. The proposed CEI framework enables both training and inference of ML models at cloud as well as edge layer, allowing developers to take full advantage of the strengths of each context, while agents can collaborate and interact via event-driven publish/subscribe messaging. This infrastructure sets the stage for future advancements in the field and further refinement of the developed technologies, leading to more intelligent, interconnected, and autonomous systems within the Internet of Everything.

Looking ahead, the development of Cowl will expand to include support for parsing and serializing RDF-based OWL formats such as RDF/XML and Turtle. This enhancement aims to increase compatibility with a wider array of existing Semantic Web tools and datasets. Furthermore, there are plans to integrate inference capabilities through interfaces that work with renowned OWL reasoners. Ultimately, Cowl's minimal memory footprint and distinctive ability to handle ontologies as streams of axioms pave the way to build a large-scale cross-platform distributed knowledge graph framework. This framework would allow nodes with limited processing power to engage in the generation and use of semantically enriched data, supporting truly collaborative, pervasive, and knowledge-aware applications.

As the development of Tiny-ME progresses, efforts will focus on improving language expressiveness and expanding its reasoning capabilities. The main objective is to support reasoning in OWL 2 profiles such as $\mathcal{EL}$, $\mathcal{EL}^{+}$, and $\mathcal{EL}^{++}$, which requires both the updating of data structures and the implementation of new reasoning algorithms within its C-based core. Due to its unique architectural framework, these improvements in inference capabilities will only require a one-time coding effort, which will automatically benefit all related APIs. Furthermore, Tiny-ME's modular datatype architecture will be leveraged to integrate non-standard datatypes, broadening its applicability across a broader spectrum of practical scenarios. These updates will boost the system's versatility and adaptability, enabling it to address the diverse and evolving needs of the Semantic Web of Everything. An additional research direction is the integration of the proposed KRR infrastructure with Large Language Models to enhance their logic-based inference abilities, opening new possibilities for advanced AI-driven applications.

The next enhancements to evOWLuator aim to significantly improve its functionality and user experience. A key initiative involves conducting a comprehensive survey of actively developed reasoning engines for desktop and mobile platforms, with a specific focus on evaluating energy consumption. Moreover, evOWLuator's visualization features will be expanded, offering a wider range of visualization types and options. This will greatly enhance its utility in academic research and data analysis contexts by providing more advanced tools for representing data, enabling deeper insights and more intuitive performance analyses of semantic web tools. In addition, the development of a web-based front-end will simplify the process of conducting evaluations. This intuitive interface will make evOWLuator more accessible and user-friendly, broadening its appeal across multiple platforms, and significantly expanding its user base.

Regarding the proposed CEI microservice architecture, future work will focus on exploring and integrating performance optimization algorithms for serverless function-based application workflows. Furthermore, the integration of semantic-enanched IoT-oriented ML approaches will be investigated to improve the explainability of models and inference outcomes. Expanding the testbed implementation and conducting further testing will also be essential to evaluate critical aspects of cloud-edge serverless computing platforms, including cloud cost profiles, edge device energy efficiency, and overall scalability. These efforts will provide the final validation of the approach in terms of technology readiness and ensure its applicability in real-world scenarios.

In moving forward with the SWoE vision, the shift from academic research to practical and production ready technologies is crucial. This progression demands thorough validation and enhancement of the suggested frameworks in real-world applications, along with productive interactions with both academic and industry partners. Achieving this vision involves refining the proposed case studies, emphasizing their importance and utility in industry-relevant contexts. Additionally, building collaborations that simplify the technology tranfer and joint innovation will be key to bridging the gap between theoretical advancements and practical implementation, ensuring the SWoE vision becomes a reality. A key element of this advancement is raising the *Technology Readiness Level* (TRL) of the proposed solutions, which is crucial for guaranteeing their usability and efficiency in practical scenarios. Achieving this goal requires cooperative efforts with industry and academic partners to align technologies with market demands and research objectives, incorporate them into current systems, and tackle the issues faced by various industries and research domains. Such partnerships are essential for speeding up the adoption of innovative technologies and paradigms and for their assimilation into the expansive IoT/IoE environment.

Therefore, the achievement of the SWoE relies on effective community engagement. The actions taken so far, such as the public release of tools and source code under permissive licenses, along with detailed documentation, have sparked promising discussions with researchers and professionals from both the academic and industrial sectors. Looking ahead, it is imperative to expand these efforts by engaging more thoroughly with the community to gather feedback, encourage collaboration, and promote the exchange of ideas. This will not only lead to improvements and advancements in current methods and solutions but may also inspire the development of new tools, frameworks, and applications grounded in the work of this dissertation. Ideally, these efforts will lay the foundation for a future where semantic technologies can transform the realm of IoT into a more widespread, intelligent, interconnected, and accessible field.

Bibliography

[1] Mohammad Aazam and Eui-Nam Huh. Fog computing micro datacenter based dynamic resource estimation and pricing model for iot. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 687–694. IEEE, 2015.

[2] Alaa Awad Abdellatif, Amr Mohamed, Carla Fabiana Chiasserini, Mounira Tlili, and Aiman Erbad. Edge computing for smart health: Context-aware approaches, opportunities, and challenges. *IEEE Network*, 33(3):196–203, 2019.

[3] Rachit Agarwal, David Gomez Fernandez, Tarek Elsaleh, Amelie Gyrard, Jorge Lanza, Luis Sanchez, Nikolaos Georgantas, and Valerie Issarny. Unified IoT ontology to enable interoperability and federation of testbeds. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 70–75, 2016.

[4] Jose Aguilar, Marxjhony Jerez, and Taniana Rodríguez. CAMeOnto: Context awareness meta ontology modeling. *Applied Computing and Informatics*, 14(2):202–213, 2018.

[5] Ejaz Ahmed, Ibrar Yaqoob, Ibrahim Abaker Targio Hashem, Imran Khan, Abdelmuttlib Ibrahim Abdalla Ahmed, Muhammad Imran, and Athanasios V Vasilakos. The role of big data analytics in Internet of Things. *Computer Networks*, 129:459–471, 2017.

[6] Safdar Ali and Stephan Kiefer. μOR–A micro OWL DL reasoner for ambient intelligent devices. In *4th International Conference on Advances in Grid and Pervasive Computing*, pages 305–316. Springer, 2009.

[7] Alhanoof Althnian, Duaa AlSaeed, Heyam Al-Baity, Amani Samha, Alanoud Bin Dris, Najla Alzakari, Afnan Abou Elwafa, and Heba Kurdi. Impact of Dataset Size on Classification Performance: An Empirical Evaluation in the Medical Domain. *Applied Sciences*, 11(2):796, 2021.

[8] Abdulmalik Alwarafy, Khaled A. Al-Thelaya, Mohamed Abdallah, Jens Schneider, and Mounir Hamdi. A survey on security and privacy issues in edge-computing-assisted Internet of Things. *IEEE Internet of Things Journal*, 8(6):4004–4022, 2020.

[9] Amira A. Amer, Ihab E. Talkhan, Reem Ahmed, and Tawfik Ismail. An optimized collaborative scheduling algorithm for prioritized tasks with shared resources in mobile-edge and cloud computing systems. *Mobile Networks and Applications*, 27(4):1444–1460, 2022.

[10] L. Andrade, M. Serrano, and C. Prazeres. The Data Interplay for the Fog of Things: A Transition to Edge Computing with IoT. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–7, May 2018.

[11] Hany F. Atlam, Robert J. Walters, and Gary B. Wills. Fog computing and the Internet of Things: A review. *big data and cognitive computing*, 2(2):10, 2018.

[12] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer networks*, 54(15):2787–2805, 2010.

[13] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the EL envelope. In *International Joint Conference on Artificial Intelligence*, volume 5, pages 364–369, 2005.

[14] Franz Baader, Diego Calvanese, Deborah L McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2002. 2nd Ed.

[15] Franz Baader, Bernhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, and Enrico Franconi. An empirical analysis of optimization

techniques for terminological representation systems. *Applied Intelligence*, 4(2):109–132, 1994.

[16] Samantha Bail, Birte Glimm, Ernesto Jimenez-Ruiz, Nicolas Matentzoglu, Bijan Parsia, and Andreas Steigmiller. ORE 2014 OWL Reasoner Evaluation Live Competition. `http://dl.kr.org/ore2014`. Accessed: 2023-03-20.

[17] Samantha Bail, Bijan Parsia, and Ulrike Sattler. JustBench: a framework for OWL benchmarking. In *International Semantic Web Conference*, pages 32–47. Springer, 2010.

[18] Andrew Banks, Ed Briggs, Ken Borgendale, and Rauhl Gupta. MQTT Version 5.0. Technical report, OASIS: Burlington, MA, USA, March 2019.

[19] Christian Becker and Christian Bizer. Exploring the Geospatial Semantic Web with DBpedia mobile. *Journal of Web Semantics*, 7(4):278–286, 2009.

[20] David Beckett. The design and implementation of the Redland RDF application framework. *Computer Networks*, 39(5):577–588, 2002.

[21] Charles Bell. *MicroPython for the Internet of Things*. Springer, 2017.

[22] Alexandre Bento, Lionel Médini, Kamal Singh, and Frédérique Laforest. Do Arduinos Dream of Efficient Reasoners? In Paul Groth, Maria-Esther Vidal, Fabian Suchanek, Pedro Szekley, Pavan Kapanipathi, Catia Pesquita, Hala Skaf-Molli, and Minna Tamper, editors, *The Semantic Web*, pages 289–304, Cham, 2022. Springer International Publishing.

[23] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):28–37, 2001.

[24] Ivano Bilenchi, Filippo Gramegna, Giuseppe Loseto, Saverio Ieva, Floriano Scioscia, and Michele Ruta. Cowl: Pushing OWL 2 over the Edge. *Internet of Things*, 29:101439, 2025.

[25] Ivano Bilenchi, Floriano Scioscia, and Michele Ruta. Cowl: A Lightweight OWL Library for the Semantic Web of Everything. In Agapito *et al.*, editor, *Current Trends in Web Engineering. ICWE 2022.*, pages 100–112, Cham, 2023. Springer.

[26] Ivano Bilenchi, Arnaldo Tomasino, Filippo Gramegna, Saverio Ieva, Agnese Pinto, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. Knowledge Representation and Reasoning for Unmanned Aerial Vehicle Intelligence. In *7th Italian Workshop on Embedded Systems (IWES 2022)*, 2022.

[27] Chatschik Bisdikian. An overview of the Bluetooth wireless technology. *IEEE Communications Magazine*, 39(12):86–94, 2001.

[28] Tegawendé F. Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 303–312, 2013.

[29] Erik P Blasch, Dale A Lambert, Pierre Valin, Mieczyslaw M Kokar, James Llinas, Subrata Das, Chee Chong, and Elisa Shahbazian. High level information fusion (HLIF): Survey of models, issues, and grand challenges. *IEEE Aerospace and Electronic Systems Magazine*, 27(9):4–20, 2012.

[30] Carlos Bobed, Roberto Yus, Fernando Bobillo, and Eduardo Mena. Semantic reasoning on mobile devices: Do Androids dream of efficient reasoners? *Journal of Web Semantics*, 35:167–183, 2015.

[31] Jürgen Bock, Peter Haase, Qiu Ji, and Raphael Volz. Benchmarking OWL Reasoners. In *ARea2008 – Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*, 2008.

[32] Dario Bonino and Luigi De Russis. DogOnt as a viable seed for semantic modeling of AEC/FM. *Semantic Web*, 9(6):763–780, 2018.

[33] Alexander Borgida. Description logics in data management. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):671–682, 1995.

[34] Carsten Bormann, Angelo P Castellani, and Zach Shelby. Coap: An application protocol for billions of tiny internet nodes. *Internet Computing, IEEE*, 16(2):62–67, 2012.

[35] Carsten Bormann, Mehmet Ersue, and Ari Keranen. Terminology for Constrained-Node Networks. RFC 7228, Internet Engineering Task Force, May 2014.

[36] Tim Bray, Dave Hollander, Andrew Layman, Richard Tobin, and Henry S. Thompson. Namespaces in XML 1.0. Recommendation, W3C, December 2009. http://www.w3.org/TR/xml-names/.

[37] Dan Brickley and Ramanathan V. Guha. RDF Schema 1.1. Recommendation, W3C, February 2014. https://www.w3.org/TR/rdf-schema/.

[38] Luca Buoncompagni, Syed Yusha Kareem, and Fulvio Mastrogiovanni. Owloop: A modular api to describe owl axioms in oop objects hierarchies. *SoftwareX*, 17:100952, 2022.

[39] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Logical foundations of peer-to-peer data integration. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 241–251. ACM, 2004.

[40] Tomasa Calvo and Gleb Beliakov. Aggregation functions based on penalties. *Fuzzy sets and Systems*, 161(10):1420–1436, 2010.

[41] Lorenzo Carnevale, Antonio Celesti, Antonino Galletta, Schahram Dustdar, and Massimo Villari. Osmotic computing as a distributed multiagent system: The Body Area Network scenario. *Internet of Things*, 5:130–139, 2019.

[42] Gustavo André Setti Cassel, Vinicius Facco Rodrigues, Rodrigo da Rosa Righi, Marta Rosecler Bez, Andressa Cruz Nepomuceno, and

Cristiano André da Costa. Serverless computing for internet of things: A systematic literature review. *Future Generation Computer Systems*, 128:299–316, 2022.

[43] Federico Castanedo, Jesús García, Miguel A Patricio, and José M Molina. Data fusion to improve trajectory tracking in a Cooperative Surveillance Multi-Agent Architecture. *Information Fusion*, 11(3):243–255, 2010.

[44] Gustavo Cevolani. Truth approximation, belief merging, and peer disagreement. *Synthese*, 191(11):2383–2401, 2014.

[45] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. Funcx: A federated function serving fabric for science. In *Proceedings of the 29th International symposium on high-performance parallel and distributed computing*, pages 65–76, 2020.

[46] Tanusree Chatterjee, Sushmita Ruj, and Sipra Das Bit. Security issues in named data networks. *Computer*, 51(1):66–75, 2018.

[47] Simona Colucci, Tommaso Di Noia, Agnese Pinto, Azzurra Ragone, Michele Ruta, and Eufemia Tinelli. A Non-Monotonic Approach to Semantic Matchmaking and Request Refinement in E-Marketplaces. *International Journal of Electronic Commerce*, 12(2):127–154, 2007.

[48] Marco Conti and Mohan Kumar. Opportunities in opportunistic computing. *Computer*, 43(01):42–50, 2010.

[49] Orhan Dagdeviren, Ilker Korkmaz, Fatih Tekbacak, Kayhan Erciyes, et al. A survey of agent technologies for wireless sensor networks. *IETE Tech Rev*, 28:168–184, 2011.

[50] Amir Vahid Dastjerdi and Rajkumar Buyya. Fog Computing: Helping the Internet of Things Realize its Potential. *Computer*, 49(8):112–116, 2016.

[51] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Science*, 1(1-2):59–83, 2017.

[52] Paul Delory, Lydia Leong, Tony Iams, Matthew Brisse, Douglas Toombs, Angelina Troy, Stanton Cole, Fintan Quinn, Mohini Dukes, and Marco Meinardi. 2022 planning guide for cloud and edge computing. Technical report, Gartner, October 2021.

[53] 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.

[54] Kathrin Dentler, Ronald Cornet, Annette Ten Teije, and Nicolette De Keizer. Comparison of reasoners for large ontologies in the OWL 2 EL profile. *Semantic Web*, 2(2):71–87, 2011.

[55] T. Di Noia, E. Di Sciascio, F.M. Donini, and M. Mongiello. A System for Principled Matchmaking in an Electronic Marketplace. *International Journal of Electronic Commerce*, 8(4):9–37, 2004.

[56] Tommaso Di Noia, Eugenio Di Sciascio, and Francesco M. Donini. Semantic matchmaking as non-monotonic reasoning: A description logic approach. *Journal of Artificial Intelligence Research (JAIR)*, 29:269–307, 2007.

[57] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Software-based energy profiling of android apps: Simple, efficient and reliable? In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pages 103–114. IEEE, 2017.

[58] Ergin Dinc, Murat Kuscu, Bilgesu Arif Bilgin, and Ozgur Baris Akan. Internet of Everything: A Unifying Framework Beyond Internet of Things. In *Harnessing the Internet of Everything (IoE) for Accelerated Innovation Opportunities*, pages 1–30. IGI Global, 2019.

[59] Chuntao Ding, Ao Zhou, Yunxin Liu, Rong N Chang, Ching-Hsien Hsu, and Shangguang Wang. A cloud-edge collaboration framework for cognitive service. *IEEE Transactions on Cloud Computing*, 10(3):1489–1499, 2020.

[60] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Reasoning in description logics. *Principles of Knowledge representation*, 1:191–236, 1996.

[61] Sinem Coleri Ergen. Zigbee/ieee 802.15. 4 summary. *UC Berkeley, September*, 10(17):11, 2004.

[62] Timofey Ermilov, Norman Heino, and Sören Auer. Ontowiki mobile: knowledge management in your pocket. In *Proceedings of the 20th International Conference Companion on World Wide Web*, pages 33–34, 2011.

[63] Timofey Ermilov, Ali Khalili, and Sören Auer. Ubiquitous semantic applications: a systematic literature review. *International Journal on Semantic Web and Information Systems*, 10(1):66–99, 2014.

[64] César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience*, 44(6):681–698, 2014.

[65] Salvador García, Julián Luengo, and Francisco Herrera. *Data preprocessing in data mining.* Springer, 2015.

[66] Peter Gärdenfors. *Belief revision*, volume 29. Cambridge University Press, 2003.

[67] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. HermiT: an OWL 2 reasoner. *Journal of Automated Reasoning*, 53(3):245–269, 2014.

[68] Keyvan Golestan, Bahador Khaleghi, Fakhri Karray, and Mohamed S Kamel. Attention assist: A high-level information fusion framework for situation and threat assessment in vehicular ad hoc networks. *IEEE Transactions on Intelligent Transportation Systems*, 17(5):1271–1285, 2016.

[69] Rafael Gonçalves, Samantha Bail, Ernesto Jiménez-Ruiz, Nicolas Matentzoglu, Bijan Parsia, Birte Glimm, and Yevgeny Kazakov. OWL

reasoner evaluation (ORE) workshop 2013 results. In *ORE*, pages 1–18, 2013.

[70] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press Cambridge, 2016.

[71] Andreas Grafberger, Mohak Chadha, Anshul Jindal, Jianfeng Gu, and Michael Gerndt. FedLess: Secure and Scalable Federated Learning Using Serverless Computing. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 164–173, 2021.

[72] Filippo Gramegna, Arnaldo Tomasino, Saverio Ieva, Ivano Bilenchi, Agnese Pinto, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. RideMATCHain: a Semantic-enhanced Blockchain Marketplace for Ridesharing. In *8th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2022)*, 2022.

[73] Filippo Gramegna, Arnaldo Tomasino, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. Features and capabilities of a blockchain-based ridesharing enhanced with semantics. In Agapito *et al.*, editor, *Current Trends in Web Engineering. ICWE 2022.*, pages 113–124, Cham, 2023. Springer.

[74] Stephan Grimm, Michael Watzke, Thomas Hubauer, and Falco Cescolini. Embedded $\mathcal{EL}+$ Reasoning on Programmable Logic Controllers. In *International Semantic Web Conference*, pages 66–81. Springer, 2012.

[75] Dominik Grzelak, Johannes Mey, and Uwe Aßmann. Design and Concept of an Osmotic Analytics Platform based on R Container. In *Proceedings of the International Conference on Foundations of Computer Science*, pages 29–35. UCMSS, 2018.

[76] Isa Guclu, Yuan-Fang Li, Jeff Z Pan, and Martin J Kollingbaum. Predicting energy consumption of ontology reasoning over mobile devices. In *International Semantic Web Conference*, pages 289–304. Springer, 2016.

[77] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

[78] Guus Schreiber and Yves Raimond. RDF 1.1 Primer. W3C Recommendation, W3C, June 2014. `https://www.w3.org/TR/rdf11-primer/`.

[79] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. Operating systems for low-end devices in the Internet of Things: a survey. *IEEE Internet of Things Journal*, 3(5):720–734, 2015.

[80] Ondrej Hlinka, Franz Hlawatsch, and Petar M Djuric. Distributed particle filtering in agent networks: A survey, classification, and comparison. *Signal Processing Magazine, IEEE*, 30(1):61–81, 2013.

[81] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL Ontologies. *Semantic Web*, 2(1):11–21, 2011.

[82] Matthew Horridge and Peter Patel-Schneider. OWL 2 Web Ontology Language Manchester Syntax (Second Edition). W3C note, W3C, December 2012. `http://www.w3.org/TR/owl2-manchester-syntax`.

[83] Saverio Ieva, Ivano Bilenchi, Filippo Gramegna, Floriano Scioscia, Michele Ruta, and Giuseppe Loseto. Integrating AI-based fleet optimization with mixed reality to enhance last-mile delivery. In *10th Italian Conference on ICT for Smart Cities and Communities (I-CiTies 2024)*, 2024.

[84] Saverio Ieva, Davide Loconte, Agnese Pinto, Filippo Gramegna, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. Semantic-enhanced Network Orchestration to Support Smart City Services. In *8th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2022)*, 2022.

[85] Martha Imprialou, Giorgos Stoilos, and Bernardo Cuenca Grau. Benchmarking ontology-based query rewriting systems. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 779–785, 2012.

[86] International Organization for Standardization. C11 Standard, ISO/IEC 9899:2011. Draft, ISO, 2011.

[87] Krzysztof Janowicz, Armin Haller, Simon JD Cox, Danh Le Phuoc, and Maxime Lefrançois. SOSA: A lightweight ontology for sensors, observations, samples, and actuators. *Journal of Web Semantics*, 56:1–10, 2019.

[88] Congfeng Jiang, Tiantian Fan, Honghao Gao, Weisong Shi, Liangkai Liu, Christophe Cérin, and Jian Wan. Energy aware edge computing: A survey. *Computer Communications*, 151:556–580, 2020.

[89] Yong-Bin Kang, Yuan-Fang Li, and Shonali Krishnaswamy. A Rigorous Characterization of Classification Performance – A Tale of Four Reasoners. In Ian Horrocks, Mikalai Yatskevich, and Ernesto Jimenez-Ruiz, editors, *OWL Reasoner Evaluation Workshop (ORE 2012)*, volume 858 of *CEUR Workshop Proceedings*, pages 88–99. CEUR-WS, 2012.

[90] Kuljeet Kaur, Sahil Garg, Georges Kaddoum, Syed Hassan Ahmed, and Dushantha Nalin K Jayakody. En-OsCo: Energy-aware Osmotic Computing Framework using Hyper-heuristics. In *Proceedings of the ACM MobiHoc Workshop on Pervasive Systems in the IoT Era*, pages 19–24, 2019.

[91] Yevgeny Kazakov. Consequence-driven reasoning for Horn SHIQ ontologies. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[92] Yevgeny Kazakov and Pavel Klinov. Experimenting with ELK Reasoner on Android. In *2nd International Workshop on OWL Reasoner Evaluation (ORE-2013)*, pages 68–74, 2013.

[93] Yevgeny Kazakov, Markus Krötzsch, and František Simančík. The incredible ELK. *Journal of automated reasoning*, 53(1):1–61, 2014.

[94] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235, 2019.

[95] Hasan Ali Khattak, Michele Ruta, and Eugenio Di Sciascio. CoAP-based healthcare sensor networks: A survey. In *Applied Sciences and Technology (IBCAST), 2014 11th International Bhurban Conference on*, pages 499–503. IEEE, 2014.

[96] Taehun Kim, Insuk Park, Soon J Hyun, and Dongman Lee. MiRE4OWL: Mobile Rule Engine for OWL. In *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, pages 317–322. IEEE, 2010.

[97] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[98] Thomas Kleemann and Alex Sinner. User Profiles and Matchmaking on Mobile Phones. In Oscar Bartenstein, editor, *Proc. of 16th International Conference on Applications of Declarative Programming and Knowledge Management INAP2005, Fukuoka*, 2005.

[99] Holger Knublauch, Ray W. Fergerson, Natalya Fridman Noy, and Mark Alan Musen. The Protégé OWL plugin: An open development environment for Semantic Web applications. In *International Semantic Web Conference*, pages 229–243. Springer, 2004.

[100] Fernando Koch. 3APL-M platform for deliberative agents in mobile devices. In *Proceedings of Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, page 154. ACM, 2005.

[101] Sébastien Konieczny and Ramón Pino Pérez. Logic based merging. *Journal of Philosophical Logic*, 40(2):239–270, 2011.

[102] Patrick Koopmann, Marcus Hähnel, and Anni-Yasmin Turhan. Energy-Efficiency of OWL Reasoners – Frequency Matters. In *Joint International Semantic Technology Conference*, pages 86–101. Springer, 2017.

[103] Gerd Kortuem, Fahim Kawsar, Vasughi Sundramoorthy, Daniel Fitton, et al. Smart Objects as Building Blocks for the Internet of Things. *IEEE Internet Computing*, 14(1):44–51, 2009.

[104] Jean-Baptiste Lamy. Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies. *Artificial intelligence in medicine*, 80:11–28, 2017.

[105] Peter John Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[106] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic web*, 6(2):167–195, 2015.

[107] Mikhail K. Levin and Lindsay G. Cowell. owlcpp: a C++ library for working with OWL ontologies. *Journal of Biomedical Semantics*, 6(1):35, Sep 2015.

[108] Lei Li and Ian Horrocks. A Software Framework for Matchmaking Based on Semantic Web Technology. *International Journal of Electronic Commerce*, 8(4), 2004.

[109] Xinghua Li, Ting Chen, Qingfeng Cheng, Siqi Ma, and Jianfeng Ma. Smart applications in edge computing: Overview on authentication and data security. *IEEE Internet of Things Journal*, 8(6):4063–4080, 2020.

[110] Yuan-Fang Li, Jeff Z Pan, Manfred Hauswirth, and Hai Nguyen. The Ubiquitous Semantic Web: Promises, Progress and Challenges. In *Web-Based Services: Concepts, Methodologies, Tools, and Applications*, pages 272–289. IGI Global, Hershey, PA, USA, 2016.

[111] Yujin Li and Wenye Wang. The unheralded power of cloudlet computing in the vicinity of mobile devices. In *2013 IEEE Global Communications Conference (GLOBECOM)*, pages 4994–4999. IEEE, 2013.

[112] Paolo Liberatore and Marco Schaerf. Arbitration (or how to merge knowledge bases). *IEEE Transactions on Knowledge and Data Engineering*, 10(1):76–90, 1998.

[113] Joshua Lieberman, Raj Singh, and Chris Goad. W3C Geospatial Vocabulary. W3C Incubator Group Report, W3C Geospatial Incubator Group (GeoXG), Oct 2007.

[114] Thorsten Liebig, Marko Luther, and Olaf Noppens. OWLlink: Structural Specification. Member Submission, W3C, July 2010. `https://www.w3.org/submissions/owllink-structural-specification`.

[115] Thorsten Liebig, Marko Luther, Olaf Noppens, and Michael Wessel. Owllink. *Semantic Web*, 2(1):23–32, January 2011.

[116] Davide Loconte, Saverio Ieva, Filippo Gramegna, Ivano Bilenchi, Corrado Fasciano, Agnese Pinto, Giuseppe Loseto, Floriano Scioscia, Michele Ruta, and Eugenio Di Sciascio. Serverless Microservice Architecture for Cloud-Edge Intelligence in Sensor Networks. *IEEE Sensors Journal*, 2024.

[117] Antonella Longo, Andrea De Matteis, and Marco Zappatore. Urban pollution monitoring based on Mobile Crowd Sensing: an osmotic computing approach. In *2018 IEEE 4th International Conference on Collaboration and Internet Computing*, pages 380–387. IEEE, 2018.

[118] Phillip Lord and Jennifer D. Warrender. Horned-owl: Building ontologies at big data scale. In *Proceedings of the International Conference on Biomedical Ontologies 2021 (ICBO 2021), Bozen-Bolzano, Italy, 16-18 September, 2021*, volume 3073 of *CEUR Workshop Proceedings*, pages 134–136. CEUR-WS.org, 2021.

[119] Giuseppe Loseto, Ivano Bilenchi, Filippo Gramegna, Davide Loconte, Floriano Scioscia, and Michele Ruta. Tiny-ME Wasm: Description Logics Reasoning in Your Browser. In Casteleyn *et al.*, editor, *Current Trends in Web Engineering. ICWE 2023.*, pages 114–126, Cham, 2024. Springer Nature Switzerland.

[120] Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, and Ivano Bilenchi. Semantic-based adaptation of quality of experience in web multimedia streams. In *38th ACM/SIGAPP Symposium On Applied Computing (SAC 2023)*, pages 1821–1830. ACM, ACM Press, March 2023.

[121] Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, Saverio Ieva, Corrado Fasciano, Ivano Bilenchi, and Davide Loconte. Osmotic cloud-edge intelligence for iot-based cyber-physical systems. *Sensors*, 22(6):2166, 2022.

[122] Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, Saverio Ieva, Corrado Fasciano, Ivano Bilenchi, and Davide Loconte. Osmotic Cloud-Edge Intelligence for IoT-based Cyber-Physical Systems. *Sensors*, 22(6):2166, 2022.

[123] Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, Saverio Ieva, Corrado Fasciano, Ivano Bilenchi, Davide Loconte, and Eugenio Di Sciascio. A Cloud-Edge Artificial Intelligence Framework for Sensor Networks. In *9th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI 2023)*, pages 149–154, 2023.

[124] Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, Saverio Ieva, Corrado Fasciano, Ivano Bilenchi, Davide Loconte, and Eugenio Di Sciascio. A cloud-edge artificial intelligence framework for sensor networks. In *2023 9th International Workshop on Advances in Sensors and Interfaces (IWASI)*, pages 149–154. IEEE, 2023.

[125] Eduardo Magalhães Oliveira. Quality Prediction in a Mining Process. `https://www.kaggle.com/edumagalhaes/quality-prediction-in-a-mining-process`. Last accessed: 2024-04-03.

[126] Viviana Mascardi, Angela Locoro, and Paolo Rosso. Automatic ontology matching via upper ontologies: A systematic evaluation. *IEEE Transactions on knowledge and data engineering*, 22(5):609–623, 2009.

[127] Brian McBride. Jena: A Semantic Web toolkit. *IEEE Internet computing*, 6(6):55–59, 2002.

[128] Javier Mendez, Kay Bierzynski, M. P. Cuéllar, and Diego P. Morales. Edge intelligence: Concepts, architectures, applications and future directions. *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[129] Antti P. Miettinen and Jukka K. Nurminen. Energy efficiency of mobile clients in cloud computing. *HotCloud*, 10(4-4):19, 2010.

[130] Martín O. Moguillansky, Renata Wassermann, and Marcelo A. Falappa. An argumentation machinery to reason over inconsistent ontologies. In Guillermo R Simari Angel Kuri-Morales, editor, *Advances in Artificial Intelligence–IBERAMIA 2010*, pages 100–109, Berlin, Germany, 2010. Springer.

[131] Ahsan Morshed, Prem Prakash Jayaraman, Timos Sellis, Dimitrios Georgakopoulos, Massimo Villari, and Rajiv Ranjan. Deep Osmosis: Holistic Distributed Deep Learning in Osmotic Computing. *IEEE Cloud Computing*, 4(6):22–32, 2017.

[132] Boris Motik, Ian Horrocks, and Su Myeon Kim. Delta-Reasoner: a Semantic Web reasoner for an intelligent mobile platform. In *Proceedings of the 21st International Conference on World Wide Web*, pages 63–72, New York, NY, USA, 2012. ACM.

[133] Boris Motik, Bijan Parsia, and Peter Patel-Schneider. OWL 2 Web Ontology Language XML Serialization (Second Edition). Recommendation, W3C, December 2012. `https://www.w3.org/TR/owl2-xml-serialization`.

[134] Mark Alan Musen. The Protégé project: a look back and a look forward. *AI matters*, 1(4):4–12, 2015.

[135] Bojan Najdenov, Goran Petkovski, Milos Jovanovik, Riste Stojanov, and Dimitar Trajanov. Automated linked data generation from the

transport administration domain. In *2015 23rd Telecommunications Forum Telfor (TELFOR)*, pages 827–830, 2015.

[136] Benazir Neha, Sanjaya Kumar Panda, Pradip Kumar Sahu, Kshira Sagar Sahoo, and Amir H Gandomi. A systematic review on osmotic computing. *ACM Transactions on Internet of Things*, 3(2):1–30, 2022.

[137] Jer Shyuan Ng, Wei Yang Bryan Lim, Zehui Xiong, Xianbin Cao, Jiangming Jin, Dusit Niyato, Cyril Leung, and Chunyan Miao. Reputation-Aware Hedonic Coalition Formation for Efficient Serverless Hierarchical Federated Learning. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):2675–2686, 2022.

[138] Tu Ngoc Nguyen and Wolf Siberski. SLUBM: An Extended LUBM Benchmark for Stream Reasoning. In *2nd International Workshop on Ordering and Reasoning, in the 12th International Semantic Web Conference (ISWC 2013)*, volume 1059 of *CEUR Workshop Proceedings*, pages 43–54, 2013.

[139] Olaf Noppens, Marko Luther, and Thorsten Liebig. The OWLlink API: Teaching OWL Components a Common Protocol. In *Proceedings of the 7th International Workshop on OWL: Experiences and Directions (OWLED 2010)*, pages 13.1–13.4, 2010.

[140] Tomoki Okuro, Yumiko Nakayama, Yoshitada Takeshima, Yusuke Kondo, Nobuya Tachimori, Makoto Yoshida, Hiromu Yoshihara, Hirohiko Suwa, and Keiichi Yasumoto. Vehicle Detection and Classification using Vibration Sensor and Machine Learning. In *2022 18th International Conference on Intelligent Environments (IE)*, pages 1–8, 2022.

[141] Andrei Olaru, Adina Magda Florea, and Amal El Fallah Seghrouchni. A context-aware multi-agent system as a middleware for ambient intelligence. *Mobile Networks and Applications*, 18(3):429–443, 2013.

[142] Alberto Pacheco, Pablo Cano, Ever Flores, Edgar Trujillo, and Pedro Marquez. A smart classroom based on deep learning and osmotic iot computing. In *2018 Congreso Internacional de Innovación y Tendencias en Ingeniería (CONIITI)*, pages 1–5. IEEE, 2018.

[143] Massimo Paolucci, Takahiro Kawamura, Terry R Payne, and Katia Sycara. Semantic Matching of Web Services Capabilities. In *International Semantic Web Conference*, pages 333–347. Springer, 2002.

[144] Bijan Parsia, Nicolas Matentzoglu, Rafael Gonçalves, Birte Glimm, and Andreas Steigmiller. The OWL reasoner evaluation (ORE) 2015 competition report. *Journal of Automated Reasoning*, 59(4):455–482, 2017.

[145] Bijan Parsia, Boris Motik, and Peter Patel-Schneider. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). Recommendation, W3C, December 2012. `http://www.w3.org/TR/owl2-syntax/`.

[146] Bijan Parsia, Sebastian Rudolph, Markus Krötzsch, Peter Patel-Schneider, and Pascal Hitzler. OWL 2 Web Ontology Language Primer (Second Edition). Recommendation, W3C, December 2012. `http://www.w3.org/TR/owl2-primer`.

[147] P. Patros, M. Ooi, V. Huang, M. Mayo, C. Anderson, S. Burroughs, M. Baughman, O. Almurshed, O. Rana, R. Chard, K. Chard, and I. Foster. Rural AI: Serverless-Powered Federated Learning for Remote Applications. *IEEE Internet Computing*, 27(02):28–34, mar 2023.

[148] Evan W. Patton and Deborah L. McGuinness. A power consumption benchmark for reasoners on mobile devices. In *International Semantic Web Conference*, pages 409–424. Springer, 2014.

[149] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[150] Charith Perera, Arkady Zaslavsky, Chi Harold Liu, Michael Compton, Peter Christen, and Dimitrios Georgakopoulos. Sensor Search Techniques for Sensing as a Service Architecture for the Internet of Things. *Sensors Journal, IEEE*, 14(2):406–420, 2014.

[151] Laércio Pioli, Carina F. Dorneles, Douglas D. J. de Macedo, and Mario A. R. Dantas. An overview of data reduction solutions at the edge of IoT systems: a systematic mapping of the literature. *Computing*, 104(8):1867–1889, 2022.

[152] Andrea Pompigna and Raffaele Mauro. Smart roads: A state of the art of highways innovations in the smart age. *Engineering Science and Technology, an International Journal*, 25:100986, 2022.

[153] Filippo Giammaria Praticò, Gaetano Bosurgi, Dario Bruneo, Salvatore Cafiso, Fabrizio De Vita, Alessandro Di Graziano, Rosario Fedele, Orazio Pellegrino, and Giuseppe Sollazzo. Innovative smart road management systems in the urban context: Integrating smart sensors and miniaturized sensing systems. *Structural Control and Health Monitoring*, 29(10):e3044, 2022.

[154] Cássio Prazeres and Martin Serrano. Soft-IoT: Self-organizing Fog of Things. In *2016 30th international conference on advanced information networking and applications workshops (WAINA)*, pages 803–808. IEEE, 2016.

[155] Eric Prud'hommeaux and Gavin Carothers. RDF 1.1 Turtle. Recommendation, W3C, February 2014. `https://www.w3.org/TR/turtle/`.

[156] Tie Qiu, Jiancheng Chi, Xiaobo Zhou, Zhaolong Ning, Mohammed Atiquzzaman, and Dapeng Oliver Wu. Edge computing in Industrial Internet of Things: Architecture, advances and challenges. *IEEE Communications Surveys & Tutorials*, 22(4):2462–2488, 2020.

[157] Philipp Raith, Stefan Nastic, and Schahram Dustdar. Serverless edge computing—where we are and what lies ahead. *IEEE Internet Computing*, 27(3):50–64, 2023.

[158] Thomas Rausch, Schahram Dustdar, and Rajiv Ranjan. Osmotic Message-Oriented Middleware for the Internet of Things. *IEEE Cloud Computing*, 5(2):17–25, 2018.

[159] Dipankar Raychaudhuri, Kiran Nagaraja, and Arun Venkataramani. Mobilityfirst: a robust and trustworthy mobility-centric architecture for the future Internet. *ACM SIGMOBILE Mobile Computing and Communications Review*, 16(3):2–13, 2012.

[160] Mohsen Rohani, Denis Gingras, Vincent Vigneron, and Dominique Gruyer. A new decentralized Bayesian approach for cooperative vehicle localization based on fusion of GPS and VANET based inter-vehicle distance measurement. *IEEE Intelligent Transportation Systems Magazine*, 7(2):85–95, 2015.

[161] Andreas Rossberg. WebAssembly Specification Release 2.0 (Draft 2023-04-08). `https://webassembly.github.io/spec/core/`. Accessed: 2023-04-14.

[162] Michele Ruta, Tommaso Di Noia, Eugenio Di Sciascio, Giacomo Piscitelli, and Floriano Scioscia. Semantic-based mobile registry for dynamic RFID-based logistics support. In *10th International Conference on Electronic Commerce, ICEC 08*, pages 1–9. ACM Press, 2008.

[163] Michele Ruta, Eugenio Di Sciascio, and Floriano Scioscia. Concept Abduction and Contraction in Semantic-based P2P Environments. *Web Intelligence and Agent Systems*, 9(3):179–207, 2011.

[164] Michele Ruta, Floriano Scioscia, Ivano Bilenchi, Filippo Gramegna, Giuseppe Loseto, Saverio Ieva, and Agnese Pinto. A multiplatform reasoning engine for the Semantic Web of Everything. *Journal of Web Semantics*, 73:100709, 2022.

[165] Michele Ruta, Floriano Scioscia, and Eugenio Di Sciascio. Enabling the Semantic Web of Things: framework and architecture. In *Sixth IEEE International Conference on Semantic Computing (ICSC 2012)*, pages 345–347, 2012.

[166] Michele Ruta, Floriano Scioscia, Eugenio Di Sciascio, and Ivano Bilenchi. OWL API for iOS: early implementation and results. In *13th OWL: Experiences and Directions Workshop and 5th OWL reasoner evaluation*

workshop (OWLED - ORE 2016), volume 10161 of *Lecture Notes in Computer Science*, pages 141–152. W3C, Springer, Nov 2016.

[167] Michele Ruta, Floriano Scioscia, Filippo Gramegna, Ivano Bilenchi, and Eugenio Di Sciascio. Mini-ME Swift: the first OWL reasoner for iOS. In *16th Extended Semantic Web Conference (ESWC 2019)*, pages 298–313. Springer, 2019.

[168] Michele Ruta, Floriano Scioscia, Filippo Gramegna, and Eugenio Di Sciascio. A Mobile Knowledge-based System for On-Board Diagnostics and Car Driving Assistance. In *4th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM)*, pages 91–96, Wilmington, USA, 2010. ThinkMind.

[169] Michele Ruta, Floriano Scioscia, Filippo Gramegna, Saverio Ieva, Eugenio Di Sciascio, and Raffaello Perez De Vera. A knowledge fusion approach for context awareness in vehicular networks. *IEEE Internet of Things Journal*, 5(4):2407–2419, 2018.

[170] Michele Ruta, Floriano Scioscia, Giuseppe Loseto, and Eugenio Di Sciascio. A semantic-enabled social network of devices for building automation. *IEEE Transactions on Industrial Informatics*, 13(6):3379–3388, Dec 2017.

[171] Michele Ruta, Floriano Scioscia, Giuseppe Loseto, and Eugenio Di Sciascio. Knowledge-based sensing/acting in mobile autonomous robots. In *2017 First IEEE International Conference on Robotic Computing (IRC)*, pages 422–427, 2017.

[172] Michele Ruta, Floriano Scioscia, Giuseppe Loseto, Filippo Gramegna, Saverio Ieva, Agnese Pinto, and Eugenio Di Sciascio. Social Internet of Things for Domotics: a Knowledge-based Approach over LDP-CoAP. *Semantic Web Journal*, 9(6):781–802, 2018.

[173] Michele Ruta, Floriano Scioscia, Giuseppe Loseto, Agnese Pinto, and Eugenio Di Sciascio. Machine learning in the Internet of Things: A semantic-enhanced approach. *Semantic Web*, 10(1):183–204, 2019.

[174] Michele Ruta, Floriano Scioscia, Agnese Pinto, Filippo Gramegna, Saverio Ieva, Giuseppe Loseto, and Eugenio Di Sciascio. CoAP-based collaborative sensor networks in the Semantic Web of Things. *Journal of Ambient Intelligence and Humanized Computing*, 10(7):2545–2562, jul 2019.

[175] F Sandu, C Costache, and T Balan. Semantic data aggregation in heterogeneous learning environments. In *2015 IEEE 21st International Symposium for Design and Technology in Electronic Packaging (SIITME)*, pages 409–412. IEEE, 2015.

[176] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[177] Guus Schreiber and Fabien Gandon. RDF 1.1 XML syntax. Recommendation, W3C, February 2014. `http://www.w3.org/TR/rdf-syntax-grammar/`.

[178] Floriano Scioscia, Ivano Bilenchi, Michele Ruta, Filippo Gramegna, and Davide Loconte. A multiplatform energy-aware OWL reasoner benchmarking framework. *Journal of Web Semantics*, 72:100694, 2022.

[179] Floriano Scioscia, Giuseppe Loseto, Davide Loconte, Corrado Fasciano, Filippo Gramegna, Saverio Ieva, Agnese Pinto, and Michele Ruta. Osmotic Computing Platform for Smart City Applications in the Cloud-Edge Continuum. In *9th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2023)*, sep 2023.

[180] Floriano Scioscia, Giuseppe Loseto, Arnaldo Tomasino, Ivano Bilenchi, Filippo Gramegna, Saverio Ieva, Agnese Pinto, Eugenio Di Sciascio, and Michele Ruta. Embedded reasoning for UAV operations: towards real-time efficiency and trustworthy autonomy. In *9th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2023)*, sep 2023.

[181] Floriano Scioscia and Michele Ruta. Building a Semantic Web of Things: issues and perspectives in information compression. In *Proceedings of the 3rd IEEE International Conference on Semantic Computing*, pages 589–594. IEEE Computer Society, 2009.

[182] Floriano Scioscia, Michele Ruta, Giuseppe Loseto, Filippo Gramegna, Saverio Ieva, Agnese Pinto, and Eugenio Di Sciascio. A mobile matchmaker for the Ubiquitous Semantic Web. *International Journal on Semantic Web and Information Systems*, 10(4):77–100, dec 2014.

[183] Floriano Scioscia, Michele Ruta, Giuseppe Loseto, Filippo Gramegna, Saverio Ieva, Agnese Pinto, and Eugenio Di Sciascio. Mini-ME matchmaker and reasoner for the Semantic Web of Things. In *Innovations, Developments, and Applications of Semantic Web and Information Systems*, pages 262–294. IGI Global, 2018.

[184] Christian Seitz and René Schönfelder. Rule-based OWL reasoning for specific embedded devices. In *International Semantic Web Conference*, pages 237–252, Berlin, Germany, 2011. Springer.

[185] Kewei Sha, T. Andrew Yang, Wei Wei, and Sadegh Davari. A survey of edge computing-based designs for IoT security. *Digital Communications and Networks*, 6(2):195–202, 2020.

[186] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Computing Surveys*, 54(11s), nov 2022.

[187] Vishal Sharma, Dushantha Nalin K Jayakody, and Marwa Qaraqe. Osmotic computing-based service migration and resource scheduling in Mobile Augmented Reality Networks (MARN). *Future Generation Computer Systems*, 102:723–737, 2020.

[188] Vishal Sharma, Ilsun You, Ravinder Kumar, and Pankoo Kim. Computational offloading for efficient trust management in pervasive online social networks using osmotic computing. *IEEE Access*, 5:5084–5103, 2017.

[189] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3:637–646, 2016.

[190] Alex Sinner and Thomas Kleemann. Krhyper–in your pocket. In *International Conference on Automated Deduction*, pages 452–457. Springer, 2005.

[191] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[192] Lea Skorin-Kapov and Martin Varela. A multi-dimensional view of QoE: the ARCU model. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 662–666, Piscataway, NJ, USA, 2012. IEEE.

[193] Alexander Smirnov and Tatiana Levashova. Knowledge Fusion Patterns: A Survey. *Information Fusion*, 52:31–40, 2019.

[194] Iraj Sodagar. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE MultiMedia*, 18(4):62–67, 2011.

[195] Steffen Staab and Rudi Studer. *Handbook on ontologies*. Springer Science & Business Media, 2010.

[196] Andreas Steigmiller, Thorsten Liebig, and Birte Glimm. Konclude: system description. *Journal of Web Semantics*, 27:78–85, 2014.

[197] Luke Steller and Shonali Krishnaswamy. Pervasive Service Discovery: mTableaux Mobile Reasoning. In *International Conference on Semantic Systems (I-Semantics)*, pages 93–101, Graz, Austria, 2008. TU Graz.

[198] Vitalijs Stepanovs. BrandT: Browser Hosted OWL Reasoner. Technical report, University of Mancherster, May 2011.

[199] W. Tai, R. Brennan, J. Keeney, and D. O'Sullivan. An Automatically Composable OWL Reasoner for Resource Constrained Devices. In *2009 IEEE International Conference on Semantic Computing*, pages 495–502. IEEE, 2009.

[200] Wei Tai, John Keeney, and Declan O'Sullivan. Resource-constrained reasoning using a reasoner composition approach. *Semantic Web*, 6(1):35–59, 2015.

[201] Antero Taivalsaari and Tommi Mikkonen. The Web as a Software Platform: Ten Years Later. In *13th International Conference on Web Systems and Technologies (WEBIST'17)*, pages 41–50, 2017.

[202] Gunnar Teege. Making the Difference: A Subtraction Operation for Description Logics. In *Proceedings of the Fourth International Conference on the Principles of Knowledge Representation and Reasoning (KR'94)*, pages 540–550. ACM, 1994.

[203] Mehdi Terdjimi, Lionel Médini, and Michael Mrissa. HyLAR+ improving hybrid location-agnostic reasoning with incremental rule-based update. In *Proceedings of the 25th International Conference Companion on World Wide Web*, pages 259–262, 2016.

[204] Edward Thomas, Jeff Z. Pan, and Yuan Ren. TrOWL: Tractable OWL 2 reasoning infrastructure. In *Extended Semantic Web Conference (ESWC)*, pages 431–435, Berlin, Germany, 2010. Springer.

[205] Daniele Tovazzi, Francescomaria Faticanti, Domenico Siracusa, Claudio Peroni, Silvio Cretti, and Tommaso Gazzini. GEM-Analytics: Cloud-to-Edge AI-Powered Energy Management. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*, pages 57–66. Springer, 2020.

[206] Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 292–297, Berlin, Germany, 2006. Springer.

[207] Edgaras Valincius, Hai H Nguyen, and Jeff Z Pan. A Power Consumption Benchmark Framework for Ontology Reasoning on Android Devices. In *OWL Reasoner Evaluation (ORE) Workshop*, pages 80–86, 2015.

[208] William Van Woensel and Syed Sibte Raza Abidi. Optimizing Semantic Reasoning on Memory-Constrained Platforms Using the RETE Algorithm. In *Extended Semantic Web Conference (ESWC)*, pages 682–696. Springer, 2018.

[209] William Van Woensel and Syed Sibte Raza Abidi. Benchmarking semantic reasoning on mobile platforms: Towards optimization using OWL2 RL. *Semantic Web*, 10(4):637–663, 2019.

[210] William Van Woensel, Newres Al Haider, Ahmad Ahmad, and Syed SR Abidi. A cross-platform benchmark framework for mobile Semantic Web reasoning engines. In *International Semantic Web Conference*, pages 389–408. Springer, 2014.

[211] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S. Nikolopoulos. Challenges and opportunities in edge computing. In *2016 IEEE International conference on Smart Cloud*, pages 20–26. IEEE, 2016.

[212] Juan Ignacio Vazquez, Diego López De Ipiña, and Iñigo Sedano. Soam: an environment adaptation model for the pervasive semantic web. In *Computational Science and Its Applications-ICCSA 2006*, pages 108–117. Springer, 2006.

[213] Ruben Verborgh and Jos De Roo. Drawing conclusions from Linked Data on the Web: the EYE reasoner. *IEEE Software*, 32(3):23–27, 2015.

[214] Silviu Vert, Bogdan Dragulescu, and Radu Vasiu. LOD4AR: Exploring Linked Open Data with a Mobile Augmented Reality Web Application. In *ISWC (Posters & Demos)*, pages 185–188, 2014.

[215] Seoane Victor, Garcia-Rubio Carlos, Almenares Florina, and Campo Celeste. Performance evaluation of CoAP and MQTT with security support for IoT environments. *Computer Networks*, 197:108338, 2021.

[216] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, Devki Nandan Jha, and Rajiv Ranjan. Osmosis: The osmotic computing platform for microelements in the cloud, edge, and internet of things. *Computer*, 52(8):14–26, 2019.

[217] Zekun Wang, Juan Cui, Tingshan Liu, Shanming Bai, Congcong Hao, Yongqiu Zheng, and Chenyang Xue. Composited pressure-velocity

sensor based on sandwich-like triboelectric nanogenerator for smart traffic monitoring. *IEEE Sensors Journal*, 2023.

[218] Charlotte Weil, Simon Elias Bibri, Régis Longchamp, François Golay, and Alexandre Alahi. A Systemic Review of Urban Digital Twin Challenges, and Perspectives for Sustainable Smart Cities. *Sustainable Cities and Society*, page 104862, 2023.

[219] Patricia Whetzel, Natasha Noy, Nigam Shah, Paul Alexander, Csongor Nyulas, Tania Tudorache, and Mark Musen. Bioportal: Enhanced functionality via new web services from the national center for biomedical ontology to access and use ontologies in software applications. *Nucleic acids research*, 39:W541–5, 06 2011.

[220] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[221] Marilyn Wolf. Machine learning + distributed IoT = Edge Intelligence. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1715–1719. IEEE, 2019.

[222] Jiangchao Yao, Shengyu Zhang, Yang Yao, Feng Wang, Jianxin Ma, Jianwei Zhang, Yunfei Chu, Luo Ji, Kunyang Jia, Tao Shen, et al. Edge-cloud polarization and collaboration: A comprehensive survey for ai. *IEEE Transactions on Knowledge and Data Engineering*, 35(7):6866–6886, 2022.

[223] Zhoujing Ye, Guannan Yan, Ya Wei, Bin Zhou, Ning Li, Shihui Shen, and Linbing Wang. Real-Time and Efficient Traffic Information Acquisition via Pavement Vibration IoT Monitoring System. *Sensors*, 21(8), 2021.

[224] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A survey on the edge computing for the Internet of Things. *IEEE Access*, 6:6900–6919, 2017.

[225] Roberto Yus, Carlos Bobed, Guillermo Esteban, Fernando Bobillo, and Eduardo Mena. Android goes semantic: DL reasoners on smartphones. In *Proceedings of 2nd International Workshop on OWL Reasoner Evaluation (ORE'13), Ulm (Germany)*, pages 46–52, 2013.

[226] Roberto Yus and Primal Pappachan. Are Apps Going Semantic? A Systematic Review of Semantic Mobile Applications. In *1st International Workshop on Mobile Deployment of Semantic Technologies*, volume 1506 of *CEUR Workshop Proceedings*, pages 2–13, 2015.

[227] A. A. Zaidan, B. B. Zaidan, Muzammil Hussain, Ahmed Haiqi, M. L. Mat Kiah, and Mohamed Abdulnabi. Multi-criteria analysis for OS-EMR software selection problem: A comparative study. *Decision Support Systems*, 78:15–27, 2015.

[228] Ke Zhang, Yuming Mao, Supeng Leng, Quanxin Zhao, Longjiang Li, Xin Peng, Li Pan, Sabita Maharjan, and Yan Zhang. Energy-efficient offloading for mobile edge computing in 5G heterogeneous networks. *IEEE Access*, 4:5896–5907, 2016.

[229] Lixia Zhang, Deborah Estrin, Jeffrey Burke, Van Jacobson, James D. Thornton, Diana K. Smetters, Beichuan Zhang, Gene Tsudik, Dan Massey, Christos Papadopoulos, et al. Named Data Networking (NDN) project. *Technical Report NDN-0001, Xerox Palo Alto Research Center-PARC*, 157:1–158, 2010.

[230] Xingzhou Zhang, Yifan Wang, Sidi Lu, Liangkai Liu, Lanyu Xu, and Weisong Shi. OpenEI: An open framework for edge intelligence. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1840–1851. IEEE, 2019.

[231] Mengchu Zhou, Giancarlo Fortino, Weiming Shen, Jin Mitsugi, James Jobin, and Rahul Bhattacharyya. Guest Editorial: Special Section on Advances and Applications of Internet of Things for Smart Automated Systems. *IEEE Transactions on Automation Science and Engineering*, 13(3):1225–1229, 2016.

[232] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.

List of publications

Journal articles

1. Ivano Bilenchi, Filippo Gramegna, Giuseppe Loseto, Saverio Ieva, Floriano Scioscia, and Michele Ruta. Cowl: Pushing OWL 2 over the Edge. *Internet of Things*, 29:101439, 2025
2. Davide Loconte, Saverio Ieva, Filippo Gramegna, Ivano Bilenchi, Corrado Fasciano, Agnese Pinto, Giuseppe Loseto, Floriano Scioscia, Michele Ruta, and Eugenio Di Sciascio. Serverless Microservice Architecture for Cloud-Edge Intelligence in Sensor Networks. *IEEE Sensors Journal*, 2024
3. Floriano Scioscia, Ivano Bilenchi, Michele Ruta, Filippo Gramegna, and Davide Loconte. A multiplatform energy-aware OWL reasoner benchmarking framework. *Journal of Web Semantics*, 72:100694, 2022
4. Michele Ruta, Floriano Scioscia, Ivano Bilenchi, Filippo Gramegna, Giuseppe Loseto, Saverio Ieva, and Agnese Pinto. A multiplatform reasoning engine for the Semantic Web of Everything. *Journal of Web Semantics*, 73:100709, 2022
5. Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, Saverio Ieva, Corrado Fasciano, Ivano Bilenchi, and Davide Loconte. Osmotic Cloud-Edge Intelligence for IoT-based Cyber-Physical Systems. *Sensors*, 22(6):2166, 2022

Peer-reviewed conference papers

1. Saverio Ieva, Ivano Bilenchi, Filippo Gramegna, Floriano Scioscia, Michele Ruta, and Giuseppe Loseto. Integrating AI-based fleet optimization with mixed reality to enhance last-mile delivery. In *10th Italian Conference on ICT for Smart Cities and Communities (I-CiTies 2024)*, 2024

2. Giuseppe Loseto, Ivano Bilenchi, Filippo Gramegna, Davide Loconte, Floriano Scioscia, and Michele Ruta. Tiny-ME Wasm: Description Logics Reasoning in Your Browser. In Casteleyn *et al.*, editor, *Current Trends in Web Engineering. ICWE 2023.*, pages 114–126, Cham, 2024. Springer Nature Switzerland

3. Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, Saverio Ieva, Corrado Fasciano, Ivano Bilenchi, Davide Loconte, and Eugenio Di Sciascio. A Cloud-Edge Artificial Intelligence Framework for Sensor Networks. In *9th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI 2023)*, pages 149–154, 2023

4. Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, and Ivano Bilenchi. Semantic-based adaptation of quality of experience in web multimedia streams. In *38th ACM/SIGAPP Symposium On Applied Computing (SAC 2023)*, pages 1821–1830. ACM, ACM Press, March 2023

5. Floriano Scioscia, Giuseppe Loseto, Arnaldo Tomasino, Ivano Bilenchi, Filippo Gramegna, Saverio Ieva, Agnese Pinto, Eugenio Di Sciascio, and Michele Ruta. Embedded reasoning for UAV operations: towards real-time efficiency and trustworthy autonomy. In *9th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2023)*, sep 2023

6. Floriano Scioscia, Giuseppe Loseto, Davide Loconte, Corrado Fasciano, Filippo Gramegna, Saverio Ieva, Agnese Pinto, and Michele Ruta. Osmotic Computing Platform for Smart City Applications in the Cloud-Edge Continuum. In *9th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2023)*, sep 2023

7. Filippo Gramegna, Arnaldo Tomasino, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. Features and capabilities of a blockchain-based ridesharing enhanced with semantics. In Agapito *et al.*, editor, *Current Trends in Web Engineering. ICWE 2022.*, pages 113–124, Cham, 2023. Springer

8. Ivano Bilenchi, Arnaldo Tomasino, Filippo Gramegna, Saverio Ieva, Agnese Pinto, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. Knowledge Representation and Reasoning for Unmanned Aerial Vehicle Intelligence. In *7th Italian Workshop on Embedded Systems (IWES 2022)*, 2022

9. Filippo Gramegna, Arnaldo Tomasino, Saverio Ieva, Ivano Bilenchi, Agnese Pinto, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. RideMATCHain: a Semantic-enhanced Blockchain Marketplace for Ridesharing. In *8th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2022)*, 2022

10. Saverio Ieva, Davide Loconte, Agnese Pinto, Filippo Gramegna, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. Semantic-enhanced Network Orchestration to Support Smart City Services. In *8th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2022)*, 2022