Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Aleksandra Simić

Speaking the Same Language Through Logic and Ontologies

Master's thesis in Communication Technology
Supervisor: David Palma
February 2024

NTNU
Norwegian University of Science and Technology

Aleksandra Simić

Speaking the Same Language Through Logic and Ontologies

Master's thesis in Communication Technology
Supervisor: David Palma
February 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Title: Speaking the Same Language Through Logic and Ontologies

Student: Simić Aleksandra

Problem description:

Over the past few years, the requirements and expectations of end-users have left an impact on the design and implementation of the computer networks infrastructure. As a result, technologies such as Network Function Virtualization (NFV) and Anything as a Service (XaaS) have been developed to accommodate the growing demand for cutting-edge technologies. Network softwarization, automation and programmability are some of the features added to improve the flexibility of computer networks. However, an increase in the number of functionalities offered by computer networks, and their expansion in size, has resulted in the growth in their complexity.

Furthermore, the operation of computer networks has become challenging because different collaborators in the system frequently interact at varying levels of understanding. In addition, numerous actors typically do not speak the same language which can lead to difficulties in knowledge management. In order for the information to be arranged in a structured way and improve the interoperability, there is a need to develop a common language. For that, ontologies can be used to describe the knowledge of a specific domain and create a knowledge base. Additionally, this approach might enable computer agents to verify specific policies using discrete logic.

This thesis will examine ontologies to establish a formal model that reflects the processes of computer networks in order to handle their complexity. In particular, the context of Docker containers and their networks will be analysed. Although Docker provides a flexible networking model, the complexity of managing networks can be high having in mind the scope of the applications running on containers. To ensure that particular rules are upheld, a formal model can provide a structured way of representing network functions and configurations. This will be done by exploring the World Wide Web Consortium (W3C) specifications for formalising ontologies and tools for representing knowledge that can be both machine and human comprehensible.

Approved on: 2023-02-23

Main supervisor: Palma David, NTNU

Abstract

In recent years, with the development of technologies such as Software-Defined Networking, Network Function Virtualization and Infrastructure as Code (IaC), modern Information and Communication Technology infrastructures have become more software-based, providing faster deployment, scaling and simplified network management. Moreover, the number of organizations working with IaC is growing, typically consisting of highly skilled professionals with different educational and cultural backgrounds. In such organizations, effective collaboration between various human actors is crucial, and it often relies on efficient knowledge management.

Various human participants, including application developers, policy officers and network engineers, possess varying levels of understanding regarding the system they collaborate around. To develop a common language between them, we propose a knowledge-based approach, allowing formal representation of different concepts and their relationships. By adding semantics to the pre-deployment task of defining an infrastructure and representing concepts based on Description Logics, we develop a prototype interpretable by both humans and software agents.

In this thesis, we utilize the Semantic Web Technologies and follow the ontology development process to define an ontology to represent knowledge about Docker services and networks. We demonstrate the possible uses of our knowledge-based approach by creating knowledge graphs with data from several Docker Compose-based scenarios. Moreover, we apply the reasoning mechanism to check the consistency of each knowledge base and verify whether the definitions of web applications and two 5G Core Network solutions comply with a set of pre-defined logic rules. With our analysis of realistic IaC deployments, we confirm the potential of drawing new conclusions based on formal knowledge management and well-defined rules, which may assist various actors in making more informed decisions.

Preface

Together with the specialization project, this thesis concludes my 2-year international master's program in Communication Technology and the first phase of the Integrated PhD studies in Information Security and Communication Technology at the Faculty of Information Technology and Electrical Engineering at the Norwegian University of Science and Technology (NTNU). The supervisor of the thesis has been Associate Professor David Palma at the Department of Information Security and Communication Technology.

The research conducted for this thesis has been carried out alongside the duty and coursework involved in the PhD program. Moreover, the background knowledge in Web Semantics has been acquired through participation in the TM8110 — PhD Topics in Information Security and Communication Technology course. Consequently, this work lays the foundation for the upcoming second phase of the PhD studies.

Acknowledgements

This thesis is a result of my biggest academic challenge so far, and it would not be possible without the people who supported me throughout the past two years. To all of them, I must express my sincere gratitude.

To begin with, I would like to thank my supervisor, David Palma, for his guidance, support, and valuable feedback. I am very grateful for getting the opportunity to extend my academic path by enrolling in the Integrated PhD program.

I would also like to acknowledge my friends back home for encouraging me to pursue my dreams. To my friends in Norway, thank you for providing comfort during moments of loneliness.

Lastly, heartfelt thanks go to my family. To my mother, Biserka, and my father, Branko, thank you for having faith in me and teaching me never to give up. To my sisters, Katarina and Sofija, thank you for being my role models and unique sources of motivation. To my boyfriend, Valentin, thank you for your boundless understanding and support.

Contents

List of Figures

List of Tables

List of Acronyms

3GPP 3rd Generation Partnership Project.

5G the fifth generation of cellular networks.

6G the sixth generation of cellular networks.

AI Artificial Intelligence.

API Application Programming Interface.

CaaS Container as a Service.

CI/CD Continuous Integration and Deployment.

CLI Command-Line Interface.

CNF Container Network Function.

DevOps Development Operations.

DLs Description Logics.

DNS Domain Name System.

FAIR Findable Accessible Interoperable and Reusable.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

IaaS Infrastructure as a Service.

IaC Infrastructure as Code.

ICT Information and Communication Technology.

IoT Internet of Things.

IP Internet Protocol.

IPv4 Internet Protocol version 4.

IRI Internationalized Resource Identifier.

IT Information Technology.

JSON-LD JavaScript Object Notation for Linked Data.

LTE Long-Term Evolution.

NF Network Function.

NFV Network Function Virtualization.

NTNU Norwegian University of Science and Technology.

OAI OpenAirInterface Software Alliance.

OS Operating System.

OWA Open World Assumption.

OWL Web Ontology Language.

RDF Resource Description Framework.

RDFa Resource Description Framework in Attributes.

RDFS Resource Description Framework Schema.

REST Representational State Transfer.

SDN Software-Defined Networking.

SLSA Supply-chain Levels for Software Artifacts.

SPARQL SPARQL Protocol and RDF Query Language.

SWRL Semantic Web Rule Language.

TCP Transmission Control Protocol.

Turtle Terse RDF Triple Language.

URI Uniform Resource Identifier.

VM Virtual Machine.

VNF Virtual Network Function.

W3C World Wide Web Consortium.

WWW World Wide Web.

XaaS Anything as a Service.

XML Extensible Markup Language.

XSD XML Schema Definition.

YAML Yet Another Markup Language.

Chapter 1
Introduction

With the expansion of various applications and services provided over the Internet, underlying computer network infrastructures have become more complex than before. Millions of devices, such as switches and servers, are connected to the Internet, and billions of users connect daily through their smartphones or laptops [1]. As a result, the networking infrastructure must be flexible to meet this increasing demand.

Different requirements, such as low latency, high reliability and the capacity for quick recoveries in case of failures, have forced changes in networking infrastructure [2]. Therefore, Information and Communication Technology (ICT) infrastructures are shifting from a legacy environment to a software-based approach that improves flexibility and scalability. Technologies such as Software-Defined Networking (SDN) and Network Function Virtualization are widely used in today's computer networks, with virtualization and containerization as the main enablers of independence from dedicated hardware. In the realm of mobile networking, the underlying networking infrastructure has changed with recent advancements from a Virtual Network Function (VNF) based architecture to Container Network Function (CNF) Cloud architecture [3].

To enable network automation and programmability, network administrators need scripting to optimize the configuration and deployment of various network elements. With that, human actors reduce the time needed for deploying and provisioning different nodes of the ICT infrastructure.

Running nodes and network functions as virtual instances on the cloud led to the great adoption of Infrastructure as Code. Some of the advantages are improved consistency, transparency and reusability [4]. Different human actors can track code modifications, deploy identical instances and suggest changes for improvement. It is also easier to implement changes in the existing topology composed out of virtualized resources compared to manual reconfiguration of physical hardware.

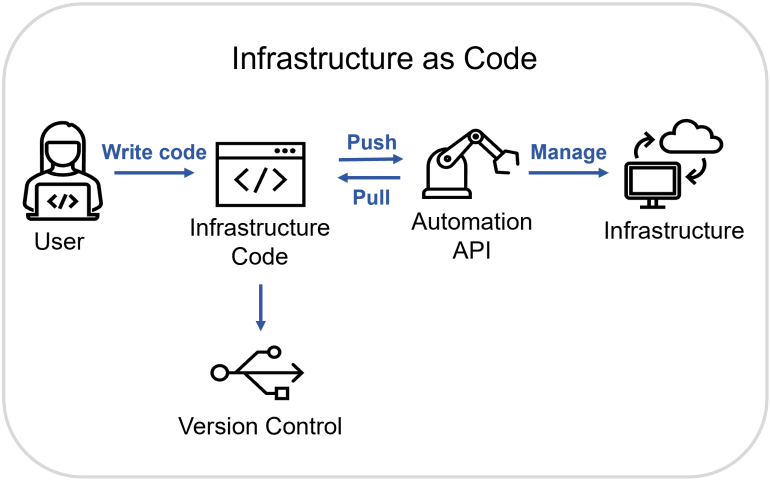


Figure 1.1: A high-level view of IaC components. Adapted from [5].

Figure 1.1 illustrates the elements of IaC, showing the flow of processes from the user (e.g. developer, network engineer, or Development Operations (DevOps) engineer) to the deployment of infrastructure. The user initiates the process by defining the infrastructure as code, which should capture the desired requirements and configurations. With version control, it is easier to monitor updates in the code. The code, then, interacts with the specific Application Programming Interface (API) that deploys and manages the infrastructure according to the user's specification.

Together with IaC, DevOps practices are changing infrastructure management. They satisfy the need for continuous software updates in cloud-based environments [6]. As mentioned in [7], besides providing automation, another goal of DevOps is to improve the collaboration between developers and Information Technology (IT) operations professionals by combining both management and engineering perspectives. However, organizations working around the IaC are typically composed of different departments in which development, operations, product and DevOps teams cooperate around the same product. They are usually highly skilled professionals who need to collaborate in a cloud-based environment.

1.1 Motivation

IaC and Continuous Integration and Deployment (CI/CD) principles and containerized architecture are important aspects of the fifth generation of cellular networks (5G) [3]. Containers are a more lightweight approach compared to Virtual Machines

(VMs) solution which reduces costs and improves performance. Nevertheless, the 5G networks coexist with legacy networks that are mainly deployed via VMs, so the network operators have to handle hybrid VNF and CNF networking systems.

One of the leading vendors in the telecommunications industry, Ericsson, uses Kubernetes as a container management system as part of the cloud-native infrastructure [8]. However, according to Attaoui, Sabir, Elbiaze, *et al.* [3], Kubernetes has some limitations, such as dealing with 5G services in various places while meeting strict demands for speed and performance.

Wikström, Persson, Parkvall, *et al.* [9] explain the possibilities and future directions of the sixth generation of cellular networks (6G). They discuss the increase of Artificial Intelligence (AI) and the possibility of having even more automated networks. Some of the keywords mentioned are *logic* and *intelligence*. However, the authors also point out that it will be essential to comprehend abstract knowledge and make conclusions from existing information and data sets. Moreover, as future mobile networks are growing in size, capabilities, services and importance, it is crucial to highlight the collaboration between various actors, from business managers, network and cloud providers to developers.

As discussed in the project preceding this thesis [10], the further adoption of IaC made ICT infrastructure more operationally complex. Technologies such as SDN and NFV enhanced the flexibility of networking infrastructure but also introduced a level of abstraction that must be appropriately managed. With DevOps principles, many different teams collaborate around the ICT infrastructure, and they do not necessarily share a common understanding of a system's behaviour, different components, and the connections between them. In such cases, human communication and knowledge sharing can be an issue.

In this thesis we propose the use of knowledge management tools and specifications for complex knowledge formalization and representation. We apply an ontology-based approach for conceptualization and formal representation of domain knowledge, and create knowledge graphs to represent elements of IaC networking domain. Some of the benefits of this approach are interoperability and extensibility, which can assist the process of managing networking systems [11].

1.2 Knowledge-based Approach

Formal knowledge representation is a sub-field of AI, and it can help us to express abstract knowledge uniquely. By enriching the information with semantics, we can improve the interoperability among multidisciplinary human actors, who typically have different educational and cultural backgrounds, especially in large cloud-based

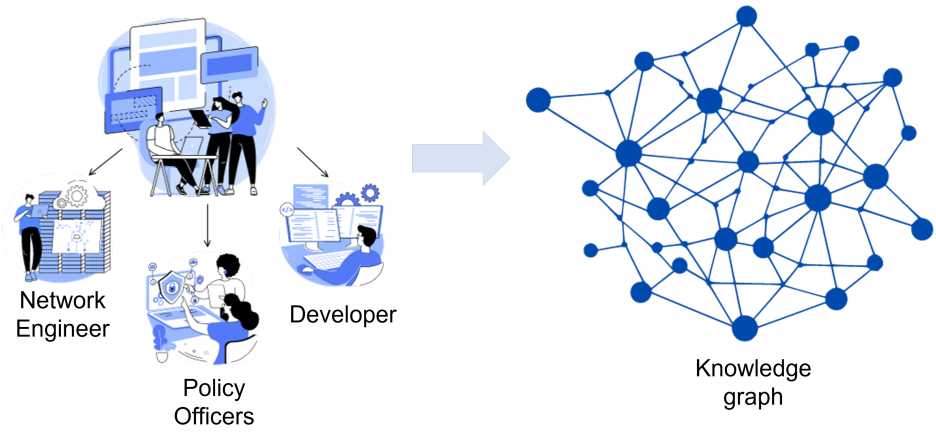


Figure 1.2: Motivational scenario of a knowledge-based approach in a multidisciplinary organization.

organizations.

Figure 1.2 illustrates an example of several actors working around the IaC, who have different expectations of the system, thus do not necessarily speak the same language. Since developers are typically not network specialists, they can struggle with understanding specific network policies. Network engineers, on the other hand, can find it challenging to grasp different definitions of particular software resources and their relationships to the networking domain. Furthermore, policy officers may have established distinct policies that the system must comply to.

To improve overall knowledge management, we can create an ontology that represents entities of an infrastructure declared as code. The knowledge-based approach should allow a formal representation of a domain of interest and enable both human and machine comprehension. Moreover, with formal logic and added semantics, software reasoners can deduct new information and make conclusions based on explicitly defined knowledge. Additionally, by applying and verifying rules, the policy officers, developers and network engineers can better understand the system they collaborate around and share their expertise.

1.3 Research Questions

The high-level objective of this master's thesis is to create a semantically enriched model for checking the networking rules of containerized services. We aim to extend

existing ontologies to create knowledge graphs to automate the process of checking networking rules based on formal logic.

Within this thesis, we expect to find answers to the following research questions:

RQ1: What existing ontologies are available for representing knowledge about container networking?

RQ2: How can these ontologies be adjusted to align with the requirements of specific network policies?

RQ3: How can we integrate generic rules to query knowledge graphs in order to validate network policies?

To answer these research questions, we look into the Semantic Web Technologies and the Docker Compose specification. Moreover, we explore various Compose-based scenarios to validate our approach and perform reasoning to infer new knowledge.

1.4 Sustainability Aspects of the Thesis

The semantically enriched model for checking networking rules of containerized services can facilitate better collaboration between multidisciplinary actors and improve knowledge sharing. Our approach can help improve interoperability and collaboration and, therefore, can be used to tackle social sustainability issues [12]. Moreover, the work presented in the rest of the thesis contributes to the following *Sustainable Development Goals* and their targets [13]:

- **Goal 8: Decent work and economic growth.** Our approach is based on the use of ontologies, which can assist people of various educational and cultural backgrounds to improve the decision-making process. Moreover, through a well-defined system representation, different stakeholders and software agents can increase their level of understanding, which can contribute to a more efficient working environment.
- **Goal 9: Industry, innovation and infrastructure.** We aim to foster trust among various human actors by implementing policies in the form of logic rules within the deployed infrastructure. Our approach can improve the trustworthiness of an infrastructure developed as code since application developers would also gain more control over their design decisions. Ontologies (and vocabularies) contribute to the interoperability aspect of the Findable Accessible Interoperable and Reusable (FAIR) principles [14]. Moreover, our approach is applicable for detecting issues and understanding the reasons behind

certain deviations from the specified logic-based rules in a container-based ICT infrastructure.

– **Goal 17: Partnership for the goals.** In our motivational scenario (cf. Section 1.2), we illustrate diverse participants: network engineers, developers and policy officers who do not typically speak the same language. With the knowledge-based approach, we intend to improve collaboration and minimize interoperability issues related to technical cooperation between both human and machine processes.

To conclude, the model created in this thesis is intended for different developers who aim to improve their understanding of the networking aspects of their container-based applications. Moreover, the knowledge-based approach fosters better collaboration within organizations working around the infrastructure defined as code.

1.5 Thesis Structure

The remaining chapters of this thesis are organized as follows:

– **Chapter 2** introduces the definition of an ontology, presents an overview of Description Logics (DLs) and the Semantic Web Technologies used in the rest of the thesis. Additionally, it introduces the concepts of Docker and Docker Compose.

– **Chapter 3** presents a literature review with the focus on ontologies representing software-based networks and container-based infrastructures.

– **Chapter 4** covers the followed research methodology and the ontology design process.

– **Chapter 5** focuses on the ontology created to represent core networking aspects of Docker containers. In addition, it presents the module designed to facilitate the creation of knowledge graphs, querying them, and the process of verifying logic-based rules.

– **Chapter 6** provides an analysis of several knowledge graphs representing Docker Compose-based deployments. Moreover, it presents the definition of logic rules and queries equally applied to each knowledge base.

– **Chapter 7** discusses the possible implications and avenues of the research presented and summarizes main contributions.

Chapter 2
Background

This chapter provides an overview of different terms and specifications used in this thesis. It begins with the ontology definition and a brief overview of DLs. After that, the review of some Semantic Web Technologies that allow software agents to interpret the knowledge will be presented. Since we identified that ICT infrastructures widely employ containerized platforms, this chapter concludes with the summary of Docker, the IaC platform for running containerized applications.

2.1 Ontologies

The term *ontology* is rooted in philosophy. It is a philosophical discipline that studies the existence of entities and their relationship [15].

In Computer Science, one of the first definitions, widely adopted by knowledge engineers, states that an ontology is *"an explicit specification of a conceptualization"* [16]. This definition went under later revisions, and the widely accepted interpretation of an ontology was introduced by Studer, Benjamins, and Fensel [17]:

Definition 2.1. "An ontology is a formal, explicit specification of a shared conceptualization".

The term *formal* indicates that an ontology should be interpreted by machines [17]. *Explicit* suggests that the types of concepts and their properties are clearly defined. *Shared* means that an ontology should capture general knowledge. The word *conceptualization*, implies that an ontology is an abstract model representing a phenomenon within a domain through concepts and their relations.

In the following sections of this thesis we will refer to Definition 2.1, when using the term *ontology*.

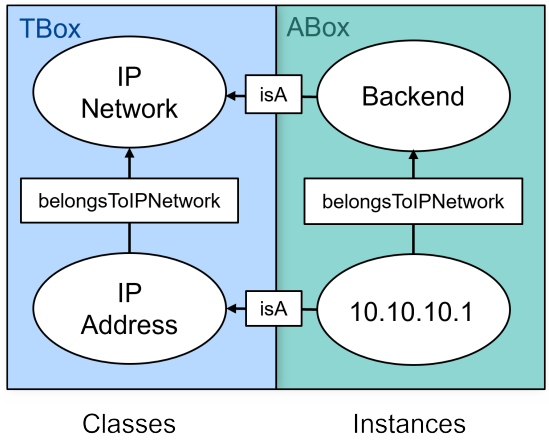


Figure 2.1: Example of a knowledge base with TBox and ABox.

2.2 Description Logics

Description Logics are a family of knowledge representation logics that allow concept definition. DLs support the modelling of ontologies, and they serve as a foundation logic for the Web Ontology Language (OWL) [18].

As outlined by Rudolph [18], DLs model concepts (classes), roles (binary relationships), and individuals (instances of classes). Within DLs, we can define a knowledge base as $\mathcal{KB} = (\mathcal{A}, \mathcal{T})$, where:

- $\mathcal{A}$ is a set of assertions, descriptions about named individuals (ABox).
- $\mathcal{T}$ is a set of terminologies, concepts' descriptions(TBox).

More advanced DLs have as their integral part an RBox or $\mathcal{R}$, representing role-centric knowledge.

Figure 2.1 shows a sample knowledge base, where we have defined two classes, namely `IP Address` and `IP Network`. Along with their relationship, they constitute the TBox. The knowledge base is populated with the real-world instances (`Backend` and `10.10.10.1`) which form the ABox.

The core concepts of DLs are negation ($\neg B$), intersection ($B \sqcap C$), and union ($B \sqcup C$), with B and C representing concepts. Additionally, DLs enable the representation of implicit relationships between concepts [18]. For example, the *isA* relationship allows inheritance from concepts to sub-concepts, as well as the definition of individuals (like in Figure 2.1).

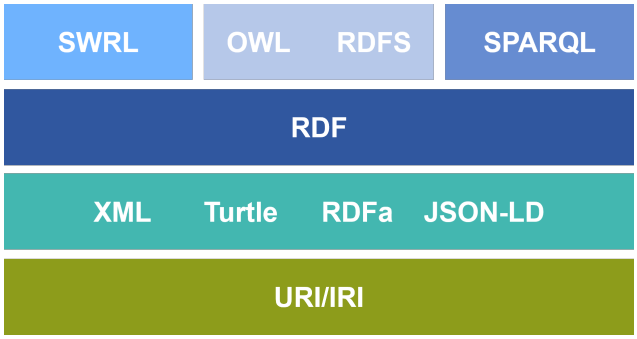


Figure 2.2: Semantic Web Technology stack, specifications and solutions. Adapted from [21].

DLs support restrictions such as $\forall R.B$, meaning that, for a given concept, all individuals with a role R belong to the concept B. Basic inferences in DLs involve checking subsumption $B \sqsubseteq C$ or equivalence $B \equiv C$. More expressive constructors include number restrictions or role inversion [19].

2.3 Semantic Web Technologies

To enable machine interpretation of an information within a domain, we need languages and specifications that allow ontology modelling. Semantic Web Technologies provide tools for creating ontologies and specifications for managing, storing and querying data.

In [20], Tim Berners-Lee, the inventor of the World Wide Web (WWW), proposed an extension of the traditional web to make data on the web machine-readable by adding semantics. With the Semantic Web, it is possible to reuse information specified by one individual in various contexts. Additionally, we can connect diverse systems handling large amounts of data, reason over it, and make logical inferences.

Figure 2.2 illustrates the four bottom layers of the Semantic Web Technology stack:

1. Uniform Resource Identifier (URI) [22] and Internationalized Resource Identifier (IRI) [23] identify resources on the web (first layer).

2. Formats for the creation of structured documents such as Extensible Markup Language (XML) [24], Terse RDF Triple Language (Turtle) [25], Resource Description Framework in Attributes (RDFa) [26] and JavaScript Object Notation for Linked Data (JSON-LD) [27] serialization formats (second layer).

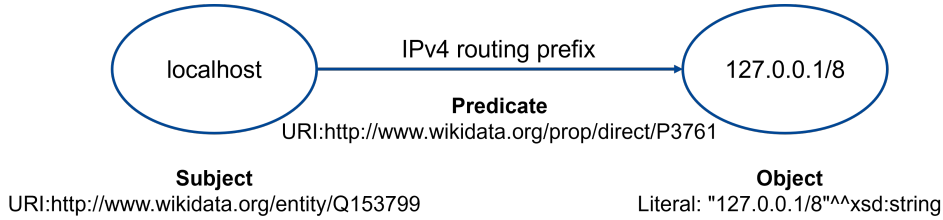


Figure 2.3: Example of an RDF triple, sourced from Wikidata [36].

3. Information exchange with RDF [28], used for simple representation of facts (third layer).
4. Models for more descriptions of facts, OWL [29] and Resource Description Framework Schema (RDFS) [30], query language SPARQL [31] and rule languages, such as Semantic Web Rule Language (SWRL) [32] (fourth layer).

Semantic Web Technologies allow us to integrate data from various sources while adding meaning and representing them in a standardized way. These specifications have been widely used in sharing and organizing information, especially in fields such as digital humanities [33] and biomedicine [34]. In the computer networking field, Semantic Web Technologies offer modelling capabilities to improve monitoring, management and overall operations of the ICT services [35].

2.3.1 Resource Description Framework

The Resource Description Framework [28] is a simple data model for expressing facts in a <*subject*> <*predicate*> <*object*> format, in the same manner as we construct simple sentences. A block composed of these three elements is called a **triple**. We can represent triples as directed graphs of two nodes connected via an edge. In RDF:

- A **subject** is a resource that must have a URI. In a graph structure, the subject represents a node.
- A **predicate** is a property which must be uniquely identified with URI. It can be represented as an edge in a graph structure. Edges are also known as directed relationships or links between nodes.
- Besides being a resource and identified as a URI, an **object** can refer to a *literal* that has a data value. XML Schema Definition (XSD) data types are prevalent in the use specification of literals.

Figure 2.3 demonstrates an intuitive way of representing RDF triples, in the form of a directed, labelled, graph. With RDF, we can store sentences, created using a

natural language, in a structured format and enable their machine interpretations. The triple, stored as a graph, can be mapped to the following human-understandable statement:

$$\underbrace{\texttt{localhost}}_{\text{subject}} \quad \underbrace{\text{has an IPv4 routing prefix}}_{\text{predicate}} \quad \underbrace{\text{127.0.0.1/8 .}}_{\text{object}}$$

Here, both subject and predicate are identified via the URIs, while the object is a literal of a type *xsd:string*. This example is retrieved from one of the largest knowledge graphs, Wikidata [37]. In Wikidata, data is labeled with identifiers, such that resources are denoted as Q, while properties and lexemes have prefixes P and L, respectively [38]. As a result, in Wikidata, the `localhost` is denoted as *Q153799* and the predicate `IPv4 routing prefix` as *P3761*.

2.3.2 Resource Description Framework Schema

RDF provides the structure for representing statements, but, it does not allow us to add constraints and meaning to data. For instance, in Figure 2.3, `127.0.0.1/8` is an object whose meaning is not explicitly specified. It might be obvious to people knowledgeable about networking that it is an Internet Protocol version 4 (IPv4) address with a subnet mask, yet it might have different interpretations.

The Resource Description Framework Schema provides more semantic expressivity and descriptions for the data stored as triples [39]. It allows the definition of classes to categorize resources that share similar attributes with *rdfs:Class*[1]. The individuals or instances of a class can be defined with *rdf:type*[2]. With RDFS, it is possible to describe the hierarchical relationship between classes. We can define *rdfs:subClassOf* relationship between classes, which means that any individual belonging to a subclass is also a member of its superclass.

It is also possible to describe properties and their hierarchy with *rdf:Property* and *rdfs:subPropertyOf*, respectively. We can use *rdfs:domain* and *rdfs:range* to put more restrictions on a property, to specify that a subject or object of a triple should belong to a specific class [30].

The example illustrated in Figure 2.3, can be extended using RDFS. For instance, we can define a class `IPv4 address range` and put a constraint on the value of the `IPv4 routing prefix` property:

- `IPv4 address range` *rdf:type rdfs:Class* .
- `IPv4 routing prefix` *rdfs:range* `IPv4 address range` .

[1]The URI for `rdfs` is http://www.w3.org/2000/01/rdf-shema#.
[2]The URI for `rdf` is http://w3.org/1999/02/22-rdf-syntax-ns#.

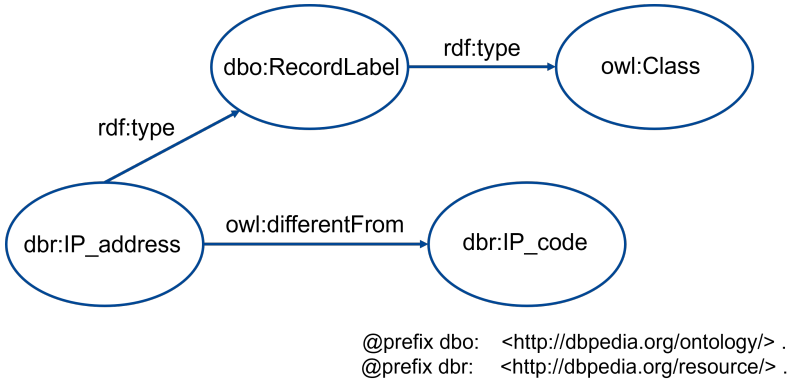


Figure 2.4: Visualization of *owl:Class* and *owl:differentFrom* relationships, sourced from DBpedia [40].

This addition would help us to state that the object of the triple, `127.0.0.1/8`, is of a type `IPv4 address range`[3]. Then, this conclusion would not be an assumption because every individual should understand what the object represents. Moreover, in RDFS we can add human-readable annotations with *rdfs:label*, *rdfs:comment* and *rdfs:seeAlso* [30].

2.3.3 Web Ontology Language

The Web Ontology Language [29] builds on RDFS adding more semantic expressivity and allowing the representation of complex relationships. It is created for knowledge representation, and grounded on DLs. OWL axioms consist of *classes*, *individuals* (instances in RDFS) and *properties*. Pre-existing classes always defined in OWL are: *owl:Thing* to which all individuals belong; and its opposite *owl:Nothing*[4]. Custom-specific classes can be defined with *owl:Class*, and there are two possible types of properties: *owl:ObjectProperty* which relates two individuals; and *owl:DatatypeProperty* whose range is a data value [39].

OWL follows the Open World Assumption (OWA), and therefore all the individuals may be identical. The OWA implies that we should not consider something untrue just because it has not been explicitly mentioned [41]. For example, with *owl:disjointWith* we can specify that two classes cannot possibly share the same individuals. Conversely, if we want to define that two classes are identical, we can use the *owl:equivalentClass*. Similarly, to denote identical or different individuals *owl:sameAs* and *owl:differentFrom* can be used [29].

[3]Following the naming convention, the class and property could be written as `:IPv4AddressRange` and `:ipv4RoutingPrefix`, respectively. The colon (`:`) is associated with a default namespace.

[4]The URI for `owl` is http://www.w3.org/2002/07/owl#.

Figure 2.4 illustrates an example from a publicly available knowledge graph, DBpedia [40]. It shows the subset of the `IP_address` resource's description, which is of a type *dbo:RecordLabel*. In the DBpedia Ontology, *dbo:RecordLabel* is an *owl:Class*. *owl:differentFrom* is used to distinguish `IP_address`, a label used in computer networking, from `IP_code`, which is used for measuring if a device is dust and waterproof [40].

Properties in OWL can be further characterised if we describe them, as *reflexive*, *transitive*, or *symmetric*. Logical relationships between classes (logical `and`, `or` and `negation`) can be defined with *owl:intersectionOf*, *owl:unionOf* and *owl:complementOf*. For even higher expressivity, it is also possible to add cardinality and value restrictions on properties [29].

2.3.4 Semantic Web Rule Language

The Semantic Web Rule Language [32] is built on top of OWL aiming to create statements that cannot be expressed in OWL. It assists in addressing specific constraints of OWL, such as the intersection of properties and certain arithmetic operations.

According to the W3C specification [32], SWRL allows us to apply DATALOG [42] logic rules to the OWL ontologies. Those rules are in the form of implications of body (antecedent) and head (consequent). In the human-readable syntax, a rule can be written as:

$$\underbrace{\text{antecedent}}_{\text{body}} \quad \Rightarrow \quad \underbrace{\text{consequent}}_{\text{head}}$$

Both antecedent and consequent are conjunctions of atoms in the form of $C(x)$, $P(x, y)$, $sameAs(x, y)$ or $differentFrom(x, y)$, where $C(x)$ is an OWL class axiom and $P(x, y)$ is a property description. To enable even more complex rules, SWRL is extended to support arithmetic operations, manipulation with string values, and more built-in functions, such as *swrlb:equal* for comparison [32].

To show the possibilities of SWRL, we can extend the knowledge base illustrated in 2.1, and define the following:

- `IPNetwork` and `IPAddress` as an *owl:Class*.
- `belongsToIPNetwork` as an *owl:ObjectProperty*.
- `hasLabel` as an *owl:DatatypePoperty*, where:
  - *rdfs:domain* of `hasLabel` is `IPAddress`.
  - *rdfs:range* of `hasLabel` is *xsd:string*.

We can also assume that if the label, which is a string value of an `IPAddress`, contains `/`, then it includes a subnet mask (i.e., the number of bits used to define a network). Therefore, we can further extend our ontology with `hasSubnetMask` as an *owl:DatatypeProperty*, as such:

- *rdfs:domain* of `hasSubnetMask` is `IPAddress`.
- *rdfs:range* of `hasSubnetMask` is *xsd:boolean*.

This restriction can be made by defining the following SWRL rule:

$$\texttt{IPAddress(?x)} \wedge \texttt{hasLabel(?x, ?y)} \wedge \texttt{swrlb:contains(?y, "/")} \rightarrow \texttt{hasSubnetMask(?x, true)}$$

Based on SWRL rules, by employing various reasoning algorithms, we can draw additional conclusions and further expand the knowledge base.

2.3.5 SPARQL Protocol and RDF Query Language

SPARQL Protocol and RDF Query Language [31] is a query language designed for both accessing and modifying RDF data. It provides different ways to create flexible queries using pattern matching. Within a graph pattern, wildcards represent entities to be fetched. Variables or question words can be associated with any element of an RDF triple [39].

Queries in SPARQL are written in Turtle. `SELECT` and `WHERE` keywords are needed for constructing a basic query. As described in [31], we can write more complex queries by combining logical operators. SPARQL supports different aggregation functions such as `COUNT()`, `SUM()`, `AVG()`, `MIN()`. Besides querying, with SPARQL we can assign a value to a variable with `BIND()` and use the `CONSTRUCT` keyword to create a new RDF graph based on a desired pattern.

In Listing 2.1, we can see a SPARQL query obtained using the Wikidata Query Service [43]. The query aims to determine whether the object of a triple illustrated in Figure 2.3 is a *literal*, and what is its data type. With the `SELECT` keyword, the `?ip` variable is bound to the graph pattern from `WHERE` statement. We use `BIND` in combination with two built-in functions: 1) `isLiteral()` checks whether the selected RDF term is a *literal*, returning true or false, while 2) `dataType()` returns its datatype value. The `SERVICE` keyword is used in Wikidata for retrieving labels.

The result of this query is shown in Table 2.1, while it is visually represented in Figure 2.3, as the object of the RDF triple. Here, we have one match to the graph pattern specified inside the `WHERE` block, and it is returned as a row in the table.

```
SELECT ?ip ?isLiteral ?dataType WHERE {
  wd:Q153799 wdt:P3761 ?ip.
  BIND(isLiteral(?ip) AS ?isLiteral)
  BIND(dataType(?ip) AS ?dataType)
  SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
}
```

Listing 2.1: SPARQL query through Wikidata Query Service.

Table 2.1: Response to the SPARQL query in Wikidata [43].

ip	isLiteral	dataType
127.0.0.1/8	true	xsd:string

2.4 Virtualization and Containerization

Virtualization and containerization refer to the creation of virtual computing environments that are abstract from the physical hardware. They enable easier scaling and better utilization of computing resources [44]. As illustrated in Figure 2.5, in a virtualized architecture, VMs are computing instances that share the physical resources but can have different a Operating System (OS) than the host.

Containers provide a more lightweight approach compared to VMs [44]. Containers use the host's OS while only needing storage space for binaries, libraries and applications, which reduces the hardware costs. They are suitable for running multiple applications sharing the same OS.

2.4.1 Docker

Docker [45] serves as a tool for container configuration and provisioning. By encapsulating applications within containers, it is easier to share the application code, regardless of the underlying platform. According to Guerriero, Garriga, Tamburri, *et al.* [6], Docker is the most used tool for building containers among IaC developers. Moreover, Docker Swarm[5], the orchestration engine for clusters of containers, is one of the most frequently employed containerization orchestration tools within Container as a Service (CaaS) platforms [3].

[5]The Docker Swarm documentation is available at https://docs.docker.com/engine/swarm/key-concepts/.

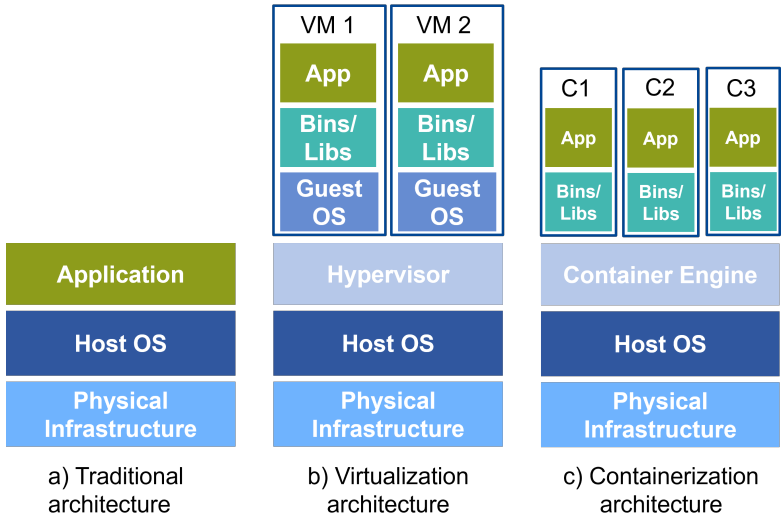


Figure 2.5: An overview of a traditional, virtualized and containerized architecture. Adapted from [44].

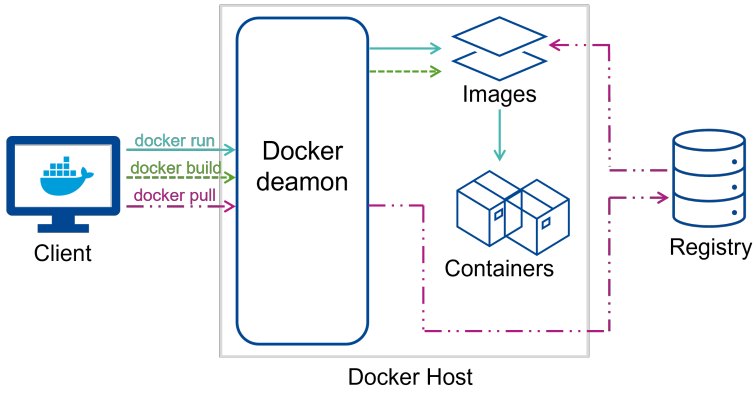


Figure 2.6: A high-level view of Docker components. Adapted from [45].

Docker is one of the key technologies of DevOps as it provides the environment for fast deployment of containers [46]. Additionally, several developers can easily share their applications packed in an isolated environment. Therefore, the delay that usually occurs between developing the application code and the testing phase is reduced. The components of Docker, illustrated in Figure 2.6, are [45]:

– **Docker deamon**, also known as the Docker engine, responsible for building, running and distributing Docker objects. It communicates with Docker clients

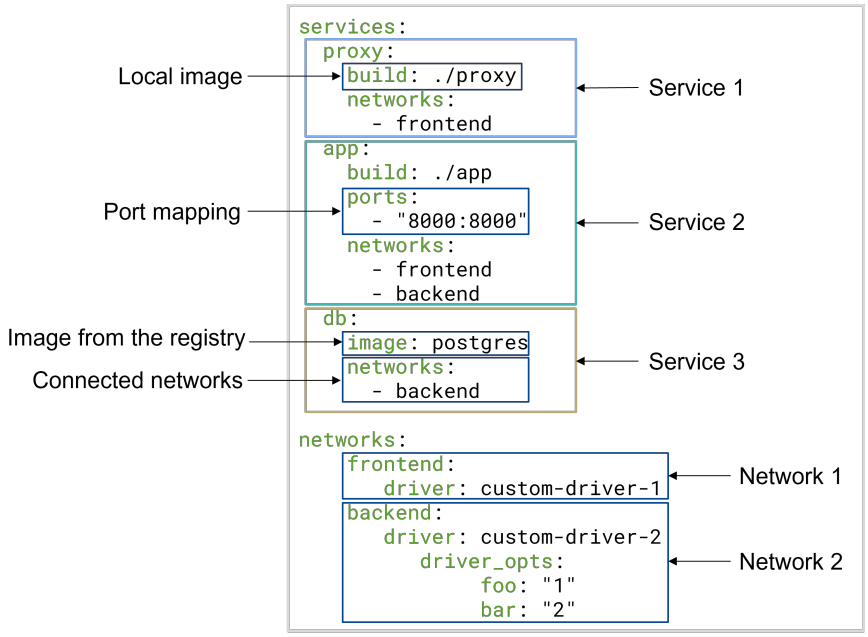


Figure 2.7: An example of services and networks definitions in a *compose.yaml* file. Adapted from [47], [48].

using a Representational State Transfer (REST) API.

- **Docker registry**, responsible for storing Docker images. The largest public database that stores Docker images is called Docker Hub[6]. Images can be stored in private repositories, too.
- **Docker objects**:
  - **Images** are read-only templates that contain instructions for building Docker containers. We can use pre-existing images stored on platforms like Docker Hub or create a Dockerfile that contains instructions for generating and executing images.
  - **Containers** are based on Images, and they are used to encapsulate the applications and their dependencies.

2.4.2 Docker Compose

Docker Compose [49] is a tool that allows the definition of multi-container applications through an orchestration file, based on Yet Another Markup Language (YAML).

[6]The Docker Hub is available at https://hub.docker.com/.

Then, using the *docker compose up* command, we can build Docker containers based on the description of the application's services from the YAML file [50].

Figure 2.7 illustrates the example of a *compose.yaml* file, which specifies a multi-container application. We constructed this example based on the Docker Networking documentation [47] and the example shown in [48]. The shown `proxy` and `app` services are built from a local image, while the `db` service uses `postgres` Docker image from the Docker Hub registry that corresponds to a *PostgreSQL*[7] setup. Each service connects to its own custom network(s). Services that do not share the same network, `db` and `proxy` in this case, are isolated from each other. For the `app` service, port mapping is explicitly stated with the *ports* key. In this example, port 8000 on the Docker Host is mapped to the port 8000 of the `app` container. Additionally, the other ways to specify ports exposed to the host are *IPADDRESS*:*HOST_PORT*:*CONTAINER_PORT* and *CONTAINER_PORT*.

2.5 Discussion

At first glance, it might not be obvious what is the connection between the various technologies presented in this chapter. Although Docker provides flexibility for IaC and DevOps, containerized tools are missing certain aspects that could be complemented by the features native to the Semantic Web Technologies. Despite its well-known syntax, Docker and Docker Compose lack a way of providing a higher-level representation of their services and their relations. On the other hand, ontologies capture the general knowledge while providing information share and reuse. Through the integration of these technologies, we aim to provide a more structured approach to working with IaC, enhanced by the validation of defined concepts and principles. Therefore, this chapter contains a description of Semantic Web Technologies and Docker as a containerization tool, that serve as a basis for the rest of this thesis.

[7]The PostgresSQL Image is available at https://hub.docker.com/_/postgres.

Chapter 3
State of the art

This chapter provides a short overview of previous studies and findings which are relevant to our research questions. Ontologies that represent different elements of communication networks have been proposed, and there are several attempts to build a knowledge graph with the aim to ease network management. Since our point of interest is the networking domain of containerization platforms, we focused our search on relevant papers addressing the application of knowledge graphs and ontologies in communication networks and Docker. Within this chapter, we delve into the relevant literature concerning knowledge representation in Docker, as it has been employed as a tool within cloud-native architectures [51], and open-source 5G deployments [52]. Moreover, the related work on validating Docker Compose files will be presented.

3.1 Ontologies in Communication Networks

According to the recent study by Javier Zorzano Mier and Iglesias [11], ontologies have been used in the past to represent hybrid telecommunication systems, which are composed of various access networks such as wired, cellular and optical. Since networks greatly expanded in size, with many connected devices made by various manufacturers, the configuration and management of such networks can be difficult. Moreover, as demonstrated by Zhou, Gray, and McLaughlin [53], we can build and query a knowledge graph to get specific information about the state of the network and ease network management.

An existing ontology known as the Toucan ontology (ToCo) offers a comprehensive representation of entities within hybrid telecommunications systems [54]. It is composed of physical components, users, services and metrics for channel quality assessment. As showed in the recent analysis by Tesolin, Demori, Moura, *et al.* [55], ToCo ontology can be, to some extent, utilized while assessing the mobility management in Long-Term Evolution (LTE), the fourth generation of wireless cellular technology. The core of ToCo ontology is an ontology-design pattern named “Device-

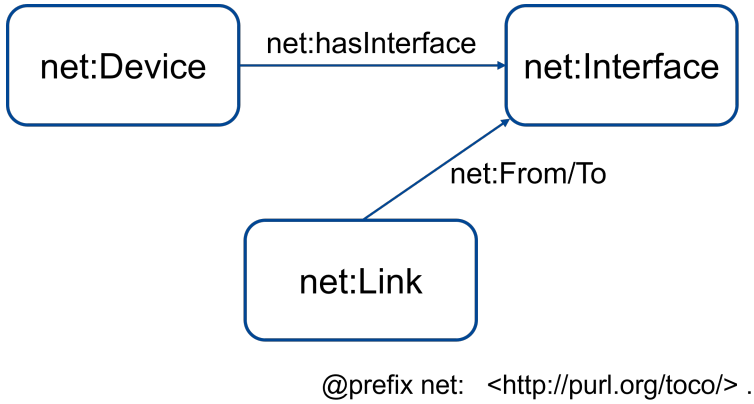


Figure 3.1: Example ontology-design pattern of network elements. Adapted from [53].

Interface-Link". Zhou, Gray, and McLaughlin [53] identify those three elements as the most important and repetitive in all telecommunication systems. Figure 3.1, provides a simplified illustration of the core of the ToCo ontology outlining its essential elements.

With the deployment of 5G networks, the operations and monitoring of heterogeneous mobile networks are complex [56]. Therefore, those processes are becoming more automated and controlled by machines. One of the goals of automation is to make both the management of different network functions easier for human actors. However, the link between human and machine must be formally defined in order to guarantee a seamless co-existence. Saraiva de Sousa, Lachos Perez, Esteve Rothenberg, *et al.* [56] propose a module for defining metrics for service monitoring in zero-touch networks. Service requirements are transformed into the RDF format, mapped to the monitoring ontology and queried with SPARQL. As a result, the module creates a service monitoring template that is agnostic from the underlying network infrastructure.

The introduction of different paradigms, such as Infrastructure as a Service (IaaS) and NFV, as well as the adaptation of DevOps principles within ICT infrastructures brought an additional level of complexity. Modern underlying infrastructures of ICT systems are more software-based, with many highly interdependent components. In the case of failures, it is important to follow the failure chain, track changes in the system and have an overview of the dependencies between the system's components. The Supply-chain Levels for Software Artifacts (SLSA) framework aims to prevent the unauthorized modification of a source code [57]. It represents the checklists for standards addressing the issue of software reliability, though does not explicitly cover

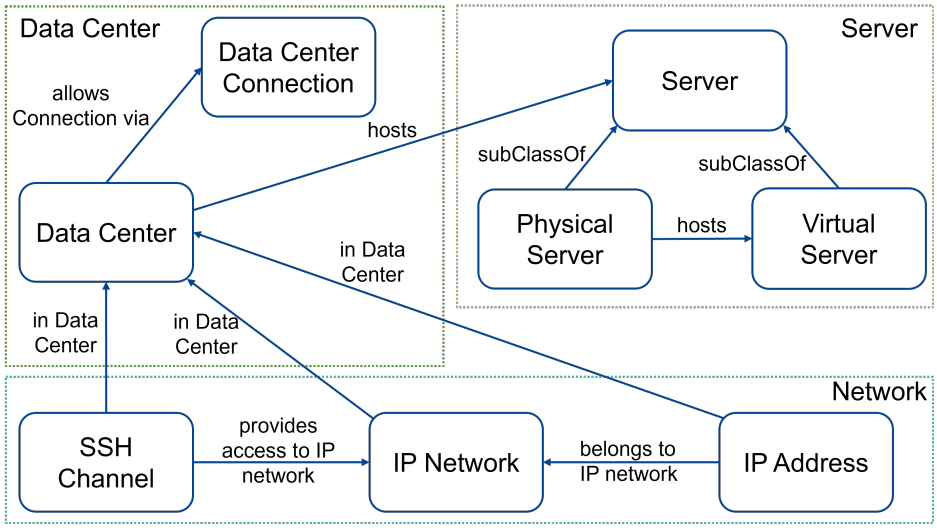


Figure 3.2: High-level overview of Network, Data Center and Server ontologies. Adapted from [58].

the networking aspects. Nevertheless, modern computer networks are being deployed as a code, which implies that there is a potential to adapt SLSA principles on the SDN and container-based orchestration systems.

Corcho, Chaves-Fraga, Toledo, *et al.* [58] propose an ontology network to solve the issue of heterogeneity in ICT systems. The ontology is composed of 10 ontologies, each of them representing different aspects of an ICT system, from network infrastructure to organisational entities and business products and services. The core entities and their relationships are commonly described in configuration and IT service management databases. This ontology is properly documented and allows the interoperability that is one of the requirements of the FAIR data principles [14]. In Figure 3.2, a high-level perspective of Data Center, Server, and Network ontologies are presented, with specific relevance to the networking aspect of the ICT infrastructure.

3.2 Ontologies and Knowledge Graphs in Software-Defined Networking

In the field of SDN, knowledge graphs and ontologies emerged as a paradigm for enhancing network management, decision-making, and efficiency. Researchers combined knowledge-based approach and SDN to solve tasks related to configuration,

resource allocation and to enable autonomic management of SDN networks. Some of the issues such as complex configuration of devices in hybrid networks, where SDN coexists with traditional networking models have been explored in [59]. In traditional networks, the configuration of devices depends on the specifications from different vendors, which is a challenge for network managers. Martinez, Yannuzzi, Vergara, *et al.* [59] demonstrate the possibility of automatically deducing Command-Line Interface (CLI) semantics using an ontology-based information extraction system.

Enabling autonomic management of SDN networks is explored by Tran, Tran, Nguyen, *et al.* [60] and Zhou, Gray, and McLaughlin [61], each offering different perspective. The former presents an ontology of Internet of Things (IoT) devices, locations, applications, and their relationships to optimize device allocation. An SDN controller uses the ontology to decide which devices will be used for a specific user requirement, and based on that manages flows. The latter utilizes knowledge graphs to ease network information extraction and querying. They create a system that extracts information about the SDN-enabled network and stores it in the knowledge graph. Through an API, network managers can query the knowledge graph and perform tasks related to configuration and management, without the need to understand the syntax of SPARQL.

3.3 Ontologies in Docker

The focus of the research community in the area of Docker infrastructures and their formalization has mostly been directed towards understanding Dockerfiles, and the process involved in constructing Docker images. Dockerfile instructions are not easy to understand without additional comments added by creators, and the information about packages or versions installed is missing.

Tommasini, Meester, Heyvaert, *et al.* [62] have created a vocabulary for representing Dockerfile by adding annotations and using instructions that are not strictly related to Docker. With this, the specific commands used for building Docker container are more understandable. Another ontology pattern described by Huo, Nabrzyski, and Ii [63] represents high-level Docker concepts with the focus on Docker infrastructure that hosts computational experiments. The main focus of their work is to represent concepts related to building the Docker images.

Osorio, Buil-Aranda, Santana-Perez, *et al.* [64] present an ontology called DockerPedia, which is composed of software images, package versions of the software installed on the Docker container. They analyze Docker images and their content, available in Docker Hub, and create a knowledge graph. The authors demonstrate querying the knowledge graph with the goal to obtain more information about images, packages and their dependencies. Instead of checking them manually, the task of

comparing images or finding ones with specific software packages can be done faster with SPARQL queries. Zhou, Chen, Liu, *et al.* [65] propose a model that parse the information about instructions in Dockerfiles, which is then combined with the information extracted from images found in Docker Hub. They use the DockerPedia ontology to construct a knowledge graph.

In addition to the ontologies depicting Docker Images, there have been efforts to develop ontologies specifically for representing Docker containers. Within the W3C, a community group focusing on vocabularies for Big Data analysis has published draft versions of ontologies for Docker [66]. The Docker Ontology defines classes such as Image, Container and Network, incorporating datatype properties to represent the relationships within the Docker environment [67]. This ontology serves as the basis for the Container Description Ontology (CDO) that has been extended with the entities related to the container orchestration systems [68]. The definition of classes is visible in this ontology, but it is impossible to determine the exact relationship between them. Boukadi, Rekik, Bernabe, *et al.* [68] present only the high-level view of the networking concepts of containerized platforms, meaning the Classes are visible, however, the description of properties is missing, which leaves room for improvements.

3.4 Validation of Compose Files

Docker Compose checks the syntax of YAML files with the *docker compose up* command [50], by default. However, it might not detect some of the unintentional configurations made by the IaC developers. Depending on their background and expertise, the developers may face challenges in identifying misconfiguration issues in the Compose file. To address this, the research community has developed tools such as the Label Checker[1] and Docker Compose Validator [69], which can be employed to validate the *compose.yaml* file before launching the intended containers. This additional validation tool can help the developers to detect:

- Duplicate images, ports, key, service and container names.
- Labels and typing mistakes, such as version, services, networks and volumes.
- `Expose`, `depends_on` and Domain Name System (DNS) directives.
- Invalid volume directory.

Another approach that aims at helping the developers to reduce mistakes while creating *compose.yaml* files signifies the need for the visualization of a given configuration. Tools, such as DockStation [70], have been developed to graphically assist developers in deploying the desired infrastructure. A study presented by Piedade,

[1]The script is available at https://github.com/serviceprototypinglab/label-consistency.

Dias, and Correia [71] shows that a similar prototype (named Docker Composer) can significantly improve the experience of developing Docker containers. It provides a visual overview of services, their dependencies, the exposed ports, links and networks. This study demonstrates that using the visualization tool makes is simpler to understand the simple configuration of containers within the Docker environment. Nevertheless, as the authors highlight, their empirical study has been done on a limited group with similar cultural and educational backgrounds.

3.5 Summary

The literature review presented in this chapter shows us that prior research has been made to understand the role of ontologies, in both communication networks and Docker. While these studies have provided valuable insights, some of the ontologies presented lack proper documentation, which does not allow their reuse. The ontology created by Corcho, Chaves-Fraga, Toledo, *et al.* [58] has been annotated following the FAIR principles, and it covers our domain of interest. However, a detailed examination, of the *ontology network* presented, suggests that its subset can be reused to cover the networking aspects of Docker containers, while the concepts of a Docker service or a container cannot be mapped fully. Additionally, based on the literature survey related to Docker, we can observe that specific aspects of container networking are not being covered. As a result, the DevOps Infrastructure Ontology [58] represents the basis for our ontology development.

Chapter 4
Methodology

This chapter provides an overview of the research methodology implemented throughout this thesis. It contains the adaptation of a design science methodology with a description of the iterative steps of the design cycle. Additionally, the outline of the methodology that guided our ontology development process will be presented.

4.1 Research Design

We start our research design process by examining the relevant literature in the field of IaC and cloud-based networking infrastructure. This phase assists us in determining the scope of our work, by identifying Docker, as the IaC tool widely used to develop different applications as a code. Concurrently, we delve into the domains of Web Semantics and formal knowledge representation. In the starting phase of the thesis, we examine the context of the thesis and the potential issues to be solved, which led to the definition of the research questions, presented in Chapter 1. To be able to establish a research design methodology and find the answers to the research questions, we define the following design tasks:

1. Create the ontology to represent the knowledge about core networking aspects between Docker services.
2. Design a module to automatically parse the Docker compose files and populate the previously defined ontology.
3. Extend the prototype to check whether the definition of Docker services complies with a specific set of policies.

These tasks present the action points to be executed through the modified version of the design cycle proposed by Wieringa [72]. The author states that the problem-solving process in research projects usually includes several iterations over the steps within the design cycle. Figure 4.1 depicts the iterative steps within our research design process, including the following stages:

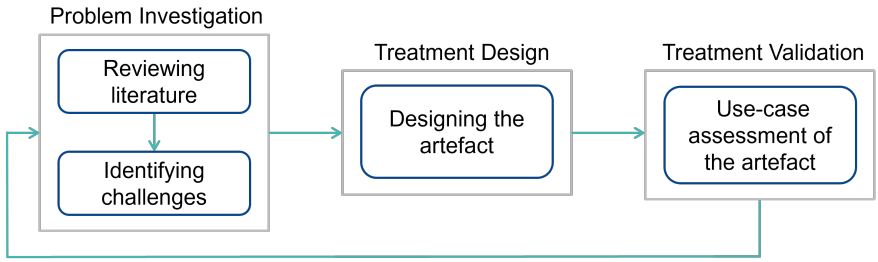


Figure 4.1: The research design cycle. Adapted from [72].

- **Problem investigation**, also a starting phase. This stage aims to evaluate the problem framework within the context[1] of IaC and the possible effects. In our adapted version of the design cycle we perform a *literature review* and, based on the state of the art, we *identify possible challenges*. During this phase of the research design, we assess the possibilities of the ontology-based approach and analyze our target containerization platform, Docker.
- **Treatment[2] design**. According to Wieringa, we should study the domain, requirements, and available treatments and design the artefacts throughout this stage. However, in our design cycle, the main action in this step is *designing the artefact*[3]. In the first iteration of the design cycle, we create a simple ontology to represent the knowledge about some of the core networking concepts of Docker infrastructure and a module to automatically populate the ontology. These two components are a part of the artefact that expands in the following iteration of the design cycle. Towards the end of the design process, we extend the artefact to check whether the definition of Docker services complies with a specific set of policies. To solve the third design task, we include logic rules to broaden the functionality of the artefact.
- **Treatment validation** is a phase in which the investigation of the interaction between the artefact and the problem context takes place. During our design cycle, we *assess how the artefact behaves in a different use-case*, meaning that we validate the consistency of the populated ontology.

Treatment implementation and evaluation are not part of our design cycle as we do not study how the artefact interacts in a real-world environment. As suggested by Wieringa [72], a potential way of executing these two tasks might include artefacts' interaction with the stakeholders (human evaluators) through surveys. Although we

[1]The context can be any element (for example, people, norms, methods) that interacts with the artefact [72].

[2]Treatment refers to the solution that can potentially solve the research problem.

[3]Different artefacts can be created in the research project, such as service, method, software or hardware components of the system and conceptual structure.

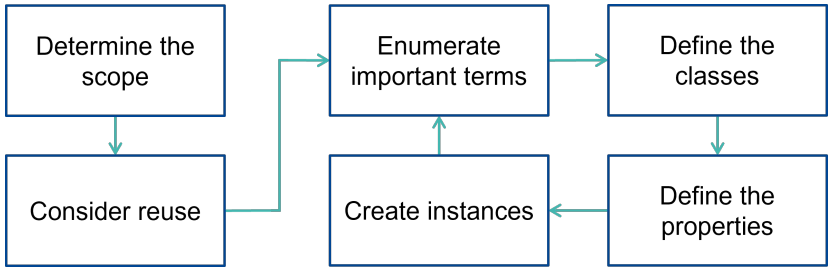


Figure 4.2: The ontology development process. Adapted from [74].

do not implement our treatment in the production environment, we do use publicly available data to test the behaviour of the artefact in different scenarios throughout the treatment validation phase.

To conclude, as illustrated in Figure 4.1, we repeat the three steps of the design cycle to find the answers to our research questions. We follow an agile development process and start by designing a small-scale prototype, progressively testing and adding more functionality. Moreover, additional details about solving the design tasks are presented in the following chapter.

4.2 Ontology Development

The first design task, presented in the previous section, involves creating an ontology to better understand the networking aspects of Docker services. Several studies suggest different methodologies for ontology development. For instance, Poveda-Villalón, Fernández-Izquierdo, Fernández-López, *et al.* [73] emphasize the need for collaboration between ontology developers, users and domain experts. The authors propose a methodology that aligns with industrial projects, and they suggest that the interaction between different actors should last throughout the entire life cycle of the ontology.

To create an ontology that captures our domain of interest, we follow the *simple knowledge-engineering methodology*, widely adopted among ontology developers [74]. Throughout the design cycle, we do not interact with other human actors as we do not place the treatment in the industrial environment. Noy and McGuinness [74] describe many approaches to model a domain. Despite the specific steps followed during the ontology development process, the ontology should be created in several iterations with continuous updates. The authors also emphasize that the representation of concepts and roles, within the determined scope, should be closely aligned with the physical (real-world) objects and their relationships.

Figure 4.2 illustrates adapted version of the *simple knowledge-engineering methodology* suggested by Noy and McGuinness [74]. Based on this methodology, our ontology development process considers the following steps:

1. **Determine the domain and scope of the ontology:** This step aims to examine the domain of interest, construct *competency questions*[4], and determine the possible users of the ontology. We look into the Docker Compose specification and its networking domain, analyze who can use our ontology, and define technical questions that limit the scope of the ontology.
2. **Consider reusing an existing ontology.** One of the main principles of ontology development is *"... enabling reuse of domain knowledge..."* which implies that one ontology should potentially be reused in different contexts. For instance, Dublin Core (DC) ontology is widely used to add annotation properties such as `creator` and applied in various ontologies with different scopes[5]. In our research design process, we conduct a literature review to find existing ontologies and look into the knowledge representation of Docker containers and the networking aspects. As presented in Chapter 3, we detected prior studies that aim to organize knowledge within the Docker and cloud-based architecture. As a result, we identified the properly documented *DevOps Infrastructure Ontology* [58], which covers our domain of interest.
3. **Enumerate important terms in the ontology:** This step aims to detect all the significant terms within the determined scope and evaluate their attributes and relationships. Moreover, Noy and McGuinness assume that the reusable ontology might not be available, which is not the case in our development process. We examine the *DevOps Infrastructure Ontology* to identify terms that can be reused and select those that need to be added to answer the *competency questions*.
4. **Define the classes and the class hierarchy:** In this step, we define classes and their taxonomy. The creation of class hierarchy can start from the broadest concepts (top-down approach), specific concepts (bottom-up approach), or concepts that are neither general nor specific. Our ontology development process employs the top-down approach since we follow the class hierarchy from the *DevOps Infrastructure Ontology*.
5. **Define the properties of classes:** This step aims to specify object and datatype properties. In our ontology development process, cardinality, value, domain and range restrictions on properties are also part of this step. In

[4]According to Grüninger and Fox [75], competency questions are the set of problems that should be solved through an ontology. They assist in determining what information the ontology should capture.

[5]The list of properly documented and interoperable ontologies, published by W3C, is available at https://www.w3.org/wiki/Good_Ontologies.

opposition to the *simple knowledge-engineering methodology*, we consider the definition of *"the facets of the slots"* as part of properties' definitions and not as a separate stage.

6. **Create instances:** Finally, we populate our ontology with individuals.

In conclusion, the *simple knowledge-engineering methodology* provides the foundation for the steps we follow during the ontology development process. After making a decision to reuse a part of the existing ontology (step 2), we begin our process by extending the ontology with classes and properties that cover the identified scope. Then, we iterate over the next four steps to answer the competency questions. Finally, our ontology serves as an input for a knowledge graph creation. It represents the proof of concept that demonstrates the feasibility of the knowledge-based approach within IaC definitions.

Chapter 5
Container Networking Ontology

In this chapter, we present the ontology created to meet the high-level objective of this thesis. We design a model that aims to represent the knowledge within the domain of container networking, and to make it comprehensible by both human actors and software agents. Our model consists of the following two components:

1. The **Container Networking Ontology**, through which some of the core networking concepts between Docker containers are represented.
2. The **Data Parsing and Populating Module**, that automates the parsing of *compose.yaml* files and populates the ontology.

The Container Networking Ontology in Turtle and XML formats, the script for parsing and populating the knowledge base and the script for creating the knowledge graph for each use case in HyperText Markup Language (HTML) format, can be found in the public repository on GitHub[1].

5.1 Designing the Ontology

As discussed in Chapter 4, we follow the main principles of the simple knowledge-engineering methodology during the ontology designing process. In the first phase, we determine the domain of the ontology by examining the networking concepts of the Docker Compose documentation [47]. We observe concepts that can be defined through the Compose files, such as Internet Protocol (IP) address, network and subnet. In addition, we look into the definition of an exposed port mapping between a container and Docker Host.

Following the motivational example presented in Chapter 1, the potential users of our ontology can be anyone interested in understanding the networking between

[1]The GitHub project is available at https://github.com/aleksandra-simic/TTM4905.git.

services in a container-based infrastructure. Those can be application developers, network and DevOps engineers. The scope of the ontology is determined through *competency questions* that are oriented towards understanding the network connectivity between services:

- How can two services (and their respective containers) be connected?
- What type of port mapping is possible in service definition?
- What type of IP Address allocation can a service have?
- How can we ensure that two services are connected through the specific IP network?

Faithful to the FAIR principles (cf. Chapter 3), and considering ontology reuse, we detect that the *ontology network*, also known as the *DevOps Infrastructure Ontology* [58], enables data interoperability, and it is available for modifications and extensions. This ontology contains entities that can potentially map to the Docker infrastructure and its networking definition. By representing the network-related parts of the DevOps infrastructure [2], this *ontology network* provides definitions of the `IP Address` and `IP Network` class with a set of properties that can be useful to address our *competency questions*.

The reflection of the real-world relationships of Docker Containers, or the definition of services in a Docker Compose file, might potentially be expressed through the `Virtual Server` class, available in the *Server Ontology* [3]. The `IP Network`, `IP Address` and `Virtual Server` classes are defined in different ontologies, and although they are connected through the *Data Center Ontology*, we could not observe a direct connection between them. Therefore, we extend the *Network Infrastructure Ontology* by defining classes and properties within the determined scope to assess the service connectivity questions.

The class hierarchy in the defined Container Networking Ontology is illustrated by Figure 5.1. The classes `Configuration Item` and `Resource` are defined within the Core ontology of the *DevOps Infrastructure Ontology*. The class `Configuration Item` is any item, `Resource`, group of items, or `Resource Group`, that can be configured in a DevOps infrastructure [4]. The Core ontology is the top-level ontology of the *ontology network* presented by Corcho, Chaves-Fraga, Toledo, *et al.* [58], as it connects the other nine ontologies. The classes `IP Address` and `IP Network` are imported from the Network Infrastructure Ontology, while we define a new class `Service` which is an

[2]The Network Infrastructure Ontology is available at https://oeg-upm.github.io/devops-infra/ontology/network/index-en.html.

[3]The Server Infrastructure Ontology can be found at https://oeg-upm.github.io/devops-infra/ontology/server/index-en.html.

[4]The Core ontology is available at https://oeg-upm.github.io/devops-infra/ontology/core/index-en.html#classes-headline.

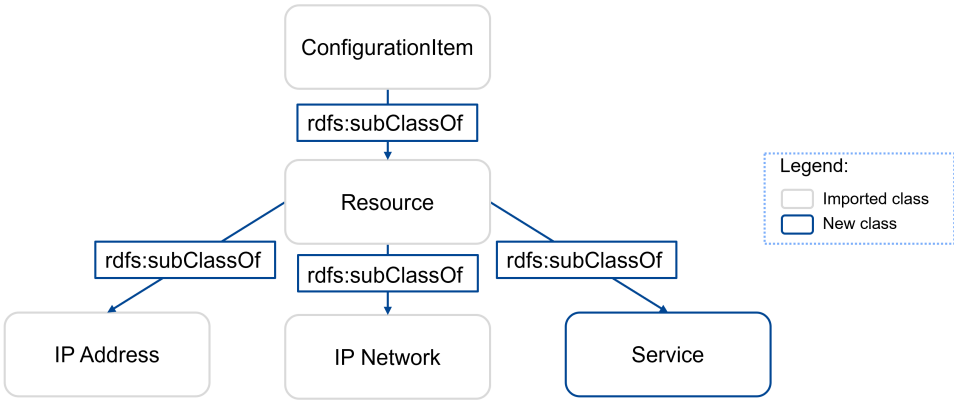


Figure 5.1: The class hierarchy in the Container Networking Ontology.

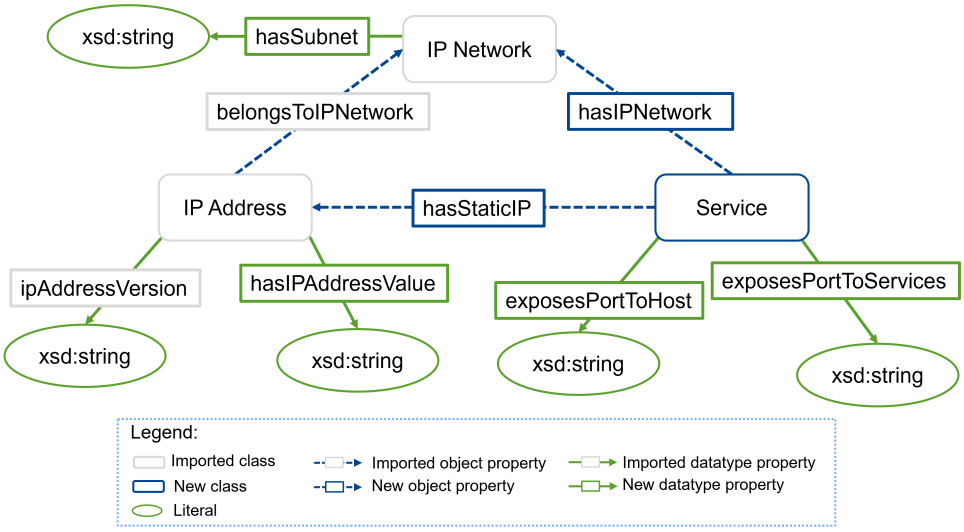


Figure 5.2: Object and datatype properties in the Container Networking Ontology.

abstract definition of a Docker container in a *compose.yaml* file. The class `Service` is defined as an *owl:Class*, while the subclass relationship is achieved through the *rdfs:subClassOf* property. Additionally, all the subclasses of the class `Resource` are mutually disjoint.

When it comes to the definition of properties, we begin with the simple relation-

ships between classes (object properties), which we further extend with datatype properties. Figure 5.2 shows the resulting object and datatype properties of the Container Networking Ontology. The properties `belongs To IP Network` and `ip Address Version` are imported from the Network Infrastructure Ontology. To further describe each of the properties, we define their domain and range. Moreover, all entities within the Container Networking Ontology are further described with *rdfs:label* and *rdfs:comment* to improve human readability.

Finally, the tasks carried out in this section were accomplished using *Protégé*, a widely adopted open-source ontology editor [76].

5.2 Data Parsing and Populating Module

The Container Networking Ontology is intended for different deployments of Docker containers and can be used for complex applications defined as services and networks through a *compose.yaml* configuration. With this in mind, we design the Data Parsing and Populating Module, which automatically creates a knowledge base for a given use case. It is implemented in Python, specifically, using the *owlready2* package [77]. Besides loading the ontology with the *get_ontology* function, we use this package to define different individuals. Since OWL follows the OWA (cf. Section 2.3.3), we use the *AllDifferent* function, ensuring that each knowledge base contains different individuals of the `Service` and the `IP Network` classes.

Figure 5.3 presents a high-level view of this module, where the Container Networking Ontology and Docker Compose files serve as inputs. As output and based on the rules defined in the ontology, the parsing and populating process produces a file that can be represented as a knowledge graph with real-world instances.

To visualize the produced knowledge base, we use OntoGraf within *Protégé* for the representation of classes, instances of classes and their relationships. For the knowledge graph creation, we use the *kglab* package [78] in Python. The input of the script for knowledge graph creation is the knowledge base file in the Turtle format. It generates the graph of nodes and edges based on all the RDF triples, and OWL axioms found in the input file. The output of this script is a file in HTML format, which we can open in any browser to interactively navigate the knowledge graph. For readability, the knowledge graph in each assessed use case is accessible within the public project on GitHub[5].

[5]The knowledge graphs can be found at https://github.com/aleksandra-simic/TTM4905.git.

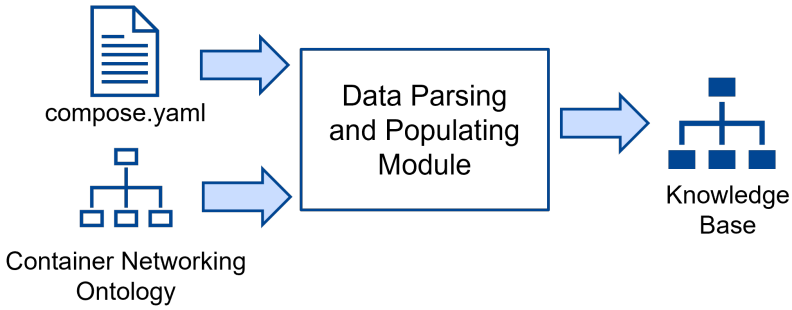


Figure 5.3: The input and output of the Data Parsing and Populating Module.

```
1    INFO   13:45:12  ----------- Running Reasoner ------------
2    INFO   13:45:12  Pre-computing inferences:
3    INFO   13:45:12      - class hierarchy
4    INFO   13:45:12      - object property hierarchy
5    INFO   13:45:12      - data property hierarchy
6    INFO   13:45:12      - class assertions
7    INFO   13:45:12      - object property assertions
8    INFO   13:45:12      - same individuals
9    INFO   13:45:12  Ontologies processed in 123 ms by Pellet
```

Listing 5.1: A snapshot of the Log, showing the result of the consistency check with Pellet.

5.3 Consistency Checks and Verifying Rules

To check the consistency of the knowledge base, we run the Pellet reasoner [79], available as a plugin in *Protégé*. Reasoners help us to determine whether a knowledge base is created based on the definition of concepts and their relationships, as defined by our ontology. Listing 5.1 shows the output of the consistency check in *Protégé*. In this example, we can see which inferences the reasoner makes and how fast it processes the populated ontology, in this case, in 123 milliseconds.

After creating a knowledge base and checking its consistency, we add more expressivity to the Container Networking Ontology by exploring DLs through the creation of Semantic Web Rule Language rules (cf. Section 2.3.4). This can be done with *SWRLTab* (available in *Protégé*), following steps below:

1. Define the object or datatype property that will be a property description in the consequent.

2. If needed, define the domain and range for a given property.
3. Construct the SWRL rule in the antecedent → consequent format.
4. Start Pellet to make inferences over the knowledge base and extend it.

With different rules, we can check if specific network policies are maintained across the knowledge base and infer new knowledge. When executing Pellet in *Protégé*, we can observe newly drawn conclusions as highlighted property assertions. Finally, with SPARQL, we can check which triples have been added to the knowledge graph by asking questions to access the RDF data. The use of a reasoner such as Pellet is a prerequisite for getting the results from a SPARQL query within the *SPARQL Query Tab* in *Protégé*. However, the created knowledge graph may also be imported into several programming languages (e.g., Python) or into a *Triple/QuadStore* [80] allowing SPARQL to be used directly.

5.4 Summary

The Container Networking Ontology is designed based on a properly annotated ontology network that represents different concepts within the cloud-based organization. Since it could not assist us in fully representing our determined scope, we extend it to create a *domain ontology* that captures core concepts in the domain of networking in Docker Compose.

The Data Parsing and Populating module presents the script used to populate the ontology with individuals from different container-based applications. For each use case, the output is a knowledge base on which we use reasoners to formally verify if it complies with the rules defined in the ontology. For rules that go beyond the definitions provided by OWL, we extend the knowledge base by defining SWRL rules that allow further reasoning and inferring new knowledge. Finally, this chapter lays the foundation for a better comprehension of Docker-based IaC deployments that we analyse in the next chapter.

Chapter 6
Semantically Enriched Infrastructure as Code

In this chapter, we provide the results of populating the Container Networking Ontology with data acquired from Docker-based applications. The goal is to analyse these applications based on the rules within the Ontology, to check the consistency of the generated knowledge base and to explore the potential of DLs through SWRL rules to infer new knowledge.

Docker is widely used for deploying web-based applications, which take advantage of the IaC features of Docker Compose to interconnect multiple services (e.g., a web frontend, a backend and a database). However, these features are also exploited by larger projects aimed at complex infrastructures such as a 5G Core Network comprising multiple VNF. The following sections present two use cases, preceded by the definition of queries and logic rules that can help us to analyse each one.

In our first use case, we examine *compose.yaml* files used for applications that rely on various technologies. To conduct our analysis, we pool data from Docker Compose files publicly available on GitHub[1]. The first analysis provides a macro view of 21 applications as a whole. Afterwards, we choose two multi-container web-based applications, reasoning over each knowledge base for a more extensive analysis. To better illustrate the capabilities of our approach, we present a second use-case for deploying a container-based 5G Core Network infrastructure and look into the *compose.yaml* files within the Free5GC project [81] and the Open Air Interface 5G Core Network solution [82].

6.1 Extending Knowledge with Description Logics

As discussed in Chapter 2, the formal definition of a knowledge base through ontologies allows extending that knowledge base through DLs. In this section, we provide an

[1]The used reference repository is available at https://github.com/docker/awesome-compose, contains different *compose.yaml* files with multiple containers. The hash of the version used is e6b1d2755f2f72a363fc346e52dce10cace846c8.

overview of how a set of logic rules can be used universally to achieve this for each of the use cases (i.e., the same principles can be applied to all use cases without requiring changes in the inference mechanisms).

We used SWRL to define the rules presented in the following subsections.

6.1.1 Default Compose Network rule

The definition of services within *compose.yaml* file does not always include a custom network which is, as a concept, represented as the class `IPNetwork` in the Container Networking Ontology. When the connection between containers is not specified, Docker Compose creates a default network that connects all the deployed containers defined through the same file.

To extend our ontology with additional information about the used networking solution, we create a new SWRL rule. This rule, based on the definition of the `hasIPNetwork` property, inspects whether the instances of the class `Service` are connected to the default network, such that:

$$\texttt{Service(?x)} \land \texttt{hasIPNetwork(?x,?y)} \rightarrow \texttt{hasDefaultIPNetwork(?x, false)}$$

6.1.2 Connectivity between Services rule

Two Docker containers running on the same host and belonging to the same custom network should communicate with each other. However, a misconfiguration of the IP segment of a pair of containers may break connectivity between them. Therefore, we create a rule to check which instances of the class `Service` within the knowledge base have the same `IP Network` defined.

Besides other atoms, the antecedent contains `differentFrom(?x,?y)` meaning that it must be satisfied for different individuals. The resulting SWRL rule is defined as follows:

$$\texttt{Service(?x)} \land \texttt{Service(?y)} \land \texttt{hasIPNetwork(?x,?s)} \land \texttt{hasIPNetwork(?y,?s)} \land \texttt{differentFrom(?x,?y)} \rightarrow \texttt{hasConnectivityWith(?x,?y)}$$

6.1.3 Exposed HTTP Port rule

Hypertext Transfer Protocol (HTTP) is one of the core protocols used in web applications. Moreover, it has been defined as a signalling protocol in 5G Core Networks [83]. The standard port for plain text HTTP traffic is 80, which can be expected to be exposed internally in a private network domain. However, if this port is exposed to the Docker Host (i.e., externally), it may indicate a potential vulnerability.

Bearing in mind the prevalence of HTTP traffic in modern applications and in 5G Core Networks, we create rules to check whether this port is exposed. In addition, port 80 can be denoted as 80/tcp, which indicates Transmission Control Protocol (TCP) as a transport protocol. Based on the definition of the class `Service` and its properties, the following two logic rules inspect if this particular port is exposed to the Docker Host:

```
Service(?x) ∧ exposesPortToHost(?x,?p) ∧ swrlb:equal(?p,"80")
            → exposesHTTPPortExternally(?x,true)

Service(?x) ∧ exposesPortToHost(?x,?p) ∧ swrlb:equal(?p,"80/tcp")
            → exposesHTTPPortExternally(?x,true)
```

Similarly, due the importance of HTTP traffic between services and VNFs, the logic rules below check if the default port is exposed internally to the other containers within the same network:

```
Service(?x) ∧ exposesPortToServices(?x,?p) ∧ swrlb:equal(?p,"80")
            → exposesHTTPPortInternally(?x,true)

Service(?x) ∧ exposesPortToServices(?x,?p) ∧ swrlb:equal(?p,"80/tcp")
            → exposesHTTPPortInternally(?x,true)
```

6.1.4 Compliance Check rule

Network security officers may implement policies that container-based applications must adhere to. For instance, to prevent exposing port 80 to the Docker Host, they might establish a rule allowing only port 443 (and 443/tcp), used by Hypertext Transfer Protocol Secure (HTTPS) protocol, encouraging external HTTP traffic to be encrypted. Additionally, the security officers may restrict internal host traffic (i.e., the ports exposed by containers) to port 80 (and 80/tcp).

The following two logic rules illustrate how the previously described network security compliance check could be defined:

```
Service(?x) ∧ exposesPortToServices(?x,?p) ∧ swrlb:notequal(?p, "80")
            ∧ swrlb:notequal(?p,"80/tcp") → nonCompliance(?x,true)

Service(?x) ∧ exposesPortToHost(?x,?p) ∧ swrlb:notequal(?p, "443")
            ∧ swrlb:notequal(?p,"443/tcp") → nonCompliance(?x,true)
```

The prerequisite to verify whether these network-related rules are maintained is to extend the Container Networking Ontology with the properties that appear as a consequent of SWRL rules. With this in mind, we extend the Ontology with the `hasConnectivityWith` object property. Additionally, we create

```
SELECT DISTINCT (COUNT (?s) AS ?x) (COUNT(?n) AS ?y) (COUNT(?ip) AS ?z)
WHERE {
#instance of the class Service (Total Services)
?s rdf:type :Service .
#instance of the class IPNetwork (# IP Networks)
?n rdf:type netw:IPNetwork .
#instance of the class IPAddress (# IP Addresses)
?ip rdf:type netw:IPAddress .
}
```

Listing 6.1: SPARQL query for retreiving the total number of services, IP addresses and custom networks.

datatype properties in which the domain is restricted to the class `Service` and the range is `xsd:boolean`. These datatype properties are `hasDefaultIPNetwork`, `exposesHTTPPortExternally`, `exposesHTTPPortInternally` and `nonCompliance`.

6.2 Exploring Knowledge with SPARQL

With SPARQL, we can retrieve knowledge stored as RDF triples in each knowledge graph. Therefore, we utilize SPARQL to analyze how well these graphs comply with the rules within the Ontology as well as with the previously presented logic rules, defined with SWRL.

The knowledge bases we analyze are denoted as:

- **All (21)** — large knowledge base created based on 21 *compose.yaml* files with different services.
- **App 1 and App 2** — the knowledge base of two representative web-based applications is presented in Section 6.3.
- **Free5GC** — 5G Core network knowledge base, created by Free5GC [81], presented in Section 6.4.1.
- **OAI5GC** — Basic 5G Core network knowledge base, created by OpenAirInterface Software Alliance (OAI) [82], presented in Section 6.4.2.

For example, after parsing a *compose.yaml* file into a knowledge graph, we may want to determine how many services were defined. Listing 6.1 shows the combination of example three queries used to obtain the total number of instances of each class

Table 6.1: Summary of responses to SPARQL queries for all use cases.

	Awesome Compose			5G Core	
	All (21)	**App 1**	**App 2**	**Free5GC**	**OAI5GC**
Total Services	55	3	3	15	9
Host Ports	43	4	1	3	0
Internal Ports	11	0	1	12	16
HTTP Host Ports	13	1	1	0	0
HTTP Internal Ports	0	0	0	0	6
Custom IP Network	20	3	0	15	9
# IP Networks	12	2	0	1	1
# IP Addresses	0	0	0	1	9
Non-compliances	40	2	2	11	7

defined by our Container Networking Ontology[2]. In more detail, with the `COUNT()` function the query engine produces the total number of all the matches for the graph pattern defined. The response to this query is recorded in Table 6.1, as the number of *Total Services*, *# IP Network* and *# IP Addresses*.

Another example is the analysis of the total number of exposed ports and HTTP ports to both the Docker Host and internally between containers, as seen in the query of Listing 6.2. The obtained results are documented as *Host Ports*, *Internal Ports*, *HTTP Host Ports* and *HTTP Internal Ports* also in Table 6.1.

In Listing 6.2, we also inspect the "Default Compose Network" rule (see Section 6.1.1), where we search for the number of services that connect to their own network and instead of a default Docker network. The number of services connected to the custom network is shown as *Custom IP Network* in Table 6.1.

The final graph pattern in Listing 6.2 focuses on the "Compliance Check" rule (see Section 6.1.4) for each created knowledge base. Table 6.1 includes the resulting number of services that are not compliant with this rule as *Non-compliances*.

[2]The prefixes must be defined in order to execute queries in *Protégé*, however, we exclude them from the Listings. The URI for `netw:` and `:` are http://w3id.org/devops-infra/network# and http://www.semanticweb.org/ontologies/2024/ContainerNetworking#, respectively.

```
SELECT DISTINCT (COUNT (?service) AS ?x)
WHERE {
?service rdf:type :Service .

#service with exposed ports to the Host  (Host Ports)
?service :exposesPortToHost ?port .

#service with exposed ports internally (Internal Ports)
?service :exposesPortToServices ?port .

#service with HTTP port exposed to the Host (HTTP Host Ports)
?service :exposesHTTPPortExternally true .

#service with HTTP port exposed internally (HTTP Internal Ports)
?service :exposesHTTPPortInternally true .

#service with custom network (Custom IP Network)
?service :hasDefaultIPNetwork false .

#service that is non-compliant (Non-compliances)
?service :nonCompliance true .
}
```

Listing 6.2: SPARQL query for analyzing the impact of SWRL rules in each knowledge graph.

In addition to counting the occurrence of certain graph patterns, SPARQL allows retrieving specific information from the edges and vertices of the knowledge graph. For example, to further examine which two individuals have connectivity between themselves, based on the "Connectivity between Services" rule (see Section 6.1.2), we can perform the query shown in Listing 6.3. By using the `FILTER` keyword, we set a condition that limits the output to all the matching instances from the class `Service` without repetitions. For example, when the reasoning engine infers that `backend hasConnectivityWith database` and `database hasConnectivityWith backend`, the output of this query would include only the `backend database` pair, meaning that these two instances of the class `Service` have connectivity between each other.

The goal of the SPARQL query presented in Listing 6.4 is to combine the knowledge resulting directly from the parsing of the *compose.yaml* files and the defined SWRL rules. In the analysis of each of our use cases, we look for non-compliant services and list all the ports they expose. In particular, with the `UNION` keyword, we seek all the non-compliant services and their exposed ports, determining whether these are exposed to the Docker Host, and therefore potentially to public

```
SELECT DISTINCT ?service1 ?service2
WHERE {
?service1 rdf:type :Service .
?service2 rdf:type :Service .
?service1 :hasConnectivityWith ?service2 .
FILTER (STR(IRI(?service1)) < (STR(IRI(?service2))))
}
```

Listing 6.3: SPARQL query for retreiving distinct pairs of services that have connectivity.

```
SELECT DISTINCT  ?service ?hostPort ?internalPort
WHERE {
?service rdf:type :Service .
?service :nonCompliance true.
{?service :exposesPortToHost ?hostPort} UNION
{?service :exposesPortToServices ?internalPort} .
}
```

Listing 6.4: SPARQL query for obtaining non-compliant services and their exposed ports.

networks, or internally only to other Docker containers.

6.3 Application Deployment

In this section, we present how the semantically enriched IaC approach can be used for understanding the definition of various services within a *compose.yaml* file.

First, we populate the Ontology with all of the 21 *compose.yaml* files used for deploying different applications composed of multiple defined services. The resulting knowledge base is represented as **All (21)** in Table 6.1. It is built based on the definition of networks and services within the Container Networking Ontology, and it contains 55 instances of the class `Service` and 12 instances of the class `IP Network`. The first striking result is that no instance of the class `IP Address` can be found.

While it is possible to have an overview of a Docker Compose repository by combining multiple *compose.yaml* files into one knowledge base, we also lose granularity. Given the complexity of such a large knowledge graph, we choose to focus on a more detailed examination of these applications. Therefore, we select two representative

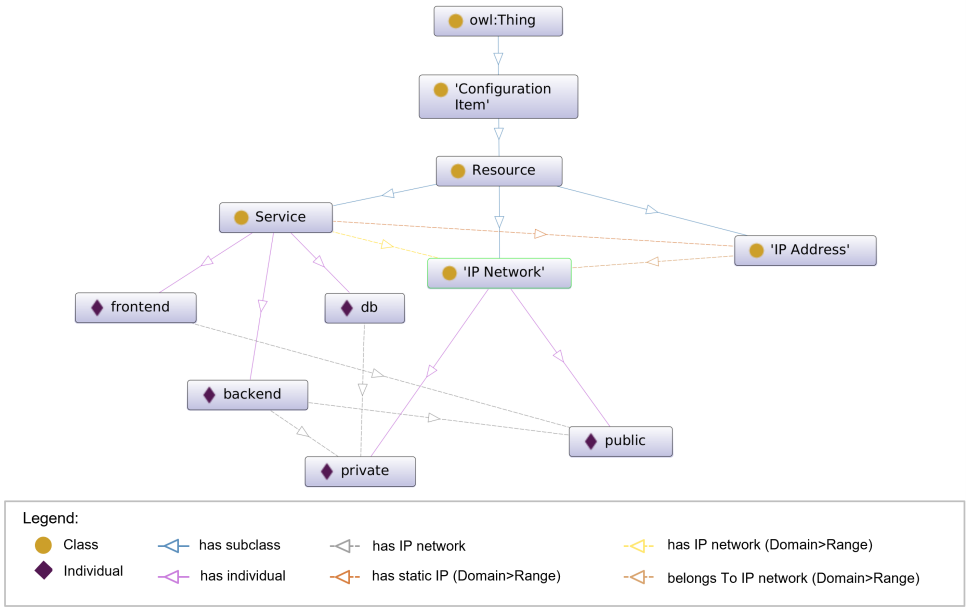


Figure 6.1: Visual representation of classes, their instances and object properties of the App 1 knowledge base.

application deployments with their respective *compose.yaml* files (out of the total 21) and create two corresponding knowledge bases.

Figure 6.1 illustrates the visual representation of the first knowledge base, which we named **App 1**. It is created based on an application that has three services, `backend`, `frontend` and `db`[3]. To assess the connectivity between these services, we can examine the visual representation of instances and the `has IP network` property. Alternatively, we can also execute the SPARQL query shown in Listing 6.3. When the "Connectivity between Services" rule is incorporated, we can obtain the direct relationship between services that have connectivity. Table 6.2 illustrates the result of the SPARQL query which reveals that the `backend` service has connectivity with both the `frontend` and `db` services.

By referring to the results presented in Table 6.1, we can verify the existence of the two non-compliant services within the **App 1** knowledge base. We can determine the cause of the conflict by querying the corresponding knowledge graph, which returns

[3]This application has a *React* frontend, a *NodeJS* backend and a *MySQL* database. The description of this application is available at https://github.com/docker/awesome-compose/tree/master/react-express-mysql. The hash of the version used is c2f8036fd353dae457eba7b9b436bf3a1c85d937.

Table 6.2: The result of the query presented in Listing 6.3 (i.e., connectivity between services), based on App 1.

App1	
?service1	**?service2**
:backend	:frontend
:backend	:db

Table 6.3: The result of the query presented in Listing 6.4 (i.e., compliance check), based on App 1.

App 1		
?service1	**?hostPort**	**?internalPort**
:backend	80^^xsd:string	
:backend	9230^^xsd:string	
:backend	9229^^xsd:string	
:frontend	3000^^xsd:string	

the contents observed in Table 6.3. Here, we verify that the `backend` service exposes port 80 to the Docker Host, which goes against the defined policy to encourage only encrypted HTTP traffic (cf. Section 6.1.4). In addition to this port, `backend` service exposes ports 9230 and 9229, while `frontend` exposes port 3000, which they should not, by definition.

The second knowledge base, denoted as **App 2**, results from another application deployment that also has three services, named `backend`, `db` and `proxy`[4]. As shown in Table 6.1 and illustrated by Figure 6.2, this knowledge base does not contain any instances of the `IP Network` or `IP Address` classes. This implies that a default Docker network is used to interconnect all services, and therefore, there is no need to verify the "Connectivity between Services" rule. However, we can observe that one HTTP port is exposed to the Docker Host and that two services do not adhere to the "Compliance check" rule. Looking in more detail by referring to the response obtained through the query in Listing 6.4, and presented in Table 6.4, we can determine the reason for this discrepancy. The `proxy` service exposes port 80 to the Docker Host, while the `db` service exposes port 5432 internally[5].

[4]This application has a *Go* backend, an *Nginx* proxy and a *PostgreSQL* database. The description of this application is available at https://github.com/docker/awesome-compose/tree/master/nginx-golang-postgres. The hash of the version used is c2f8036fd353dae457eba7b9b436bf3a1c85d937.

[5]It is expectable for the `db` service to expose PostgreSQL's default port to the *backend*, but not to all other services. This verification could potentially be added to our rule policy too.

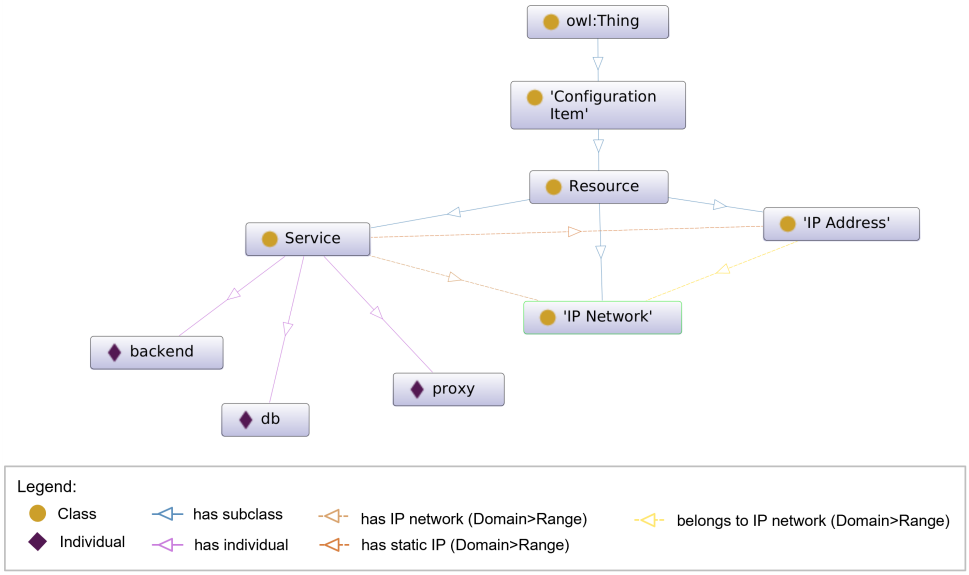


Figure 6.2: Visual representation of classes, their instances and object properties of the App 2 knowledge base.

Table 6.4: The result of the query presented in Listing 6.4 (i.e., compliance check), based on App 2.

App 2		
?service1	**?hostPort**	**?internalPort**
:proxy	80^^xsd:string	
:db		5432^^xsd:string

6.4 Docker-based 5G Core Network Deployment

The 5G Core Network definition is cloud-native and composed of VNFs that can be realized as micro services [84]. In 5G networks, Network Functions (NFs) have different functionalities, with a clear decoupling between control and data plane NFs. These properties make the deployment of 5G network elements as Docker containers, as well as the verification of certain rules, an interesting exercise.

This section presents automated deployment scenarios for VNFs, whose definition is provided through *compose.yaml* files. It explores the possibilities of the Container Networking Ontology to improve the comprehension of Docker-based 5G Core Network

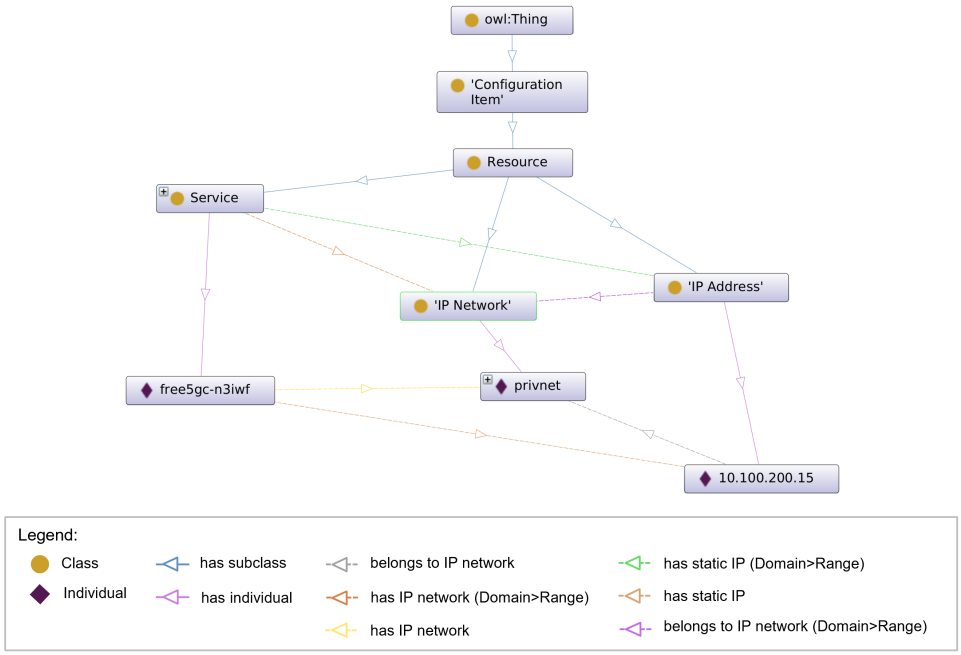


Figure 6.3: Visual representation of classes and object properties of the Free5GC knowledge base, emphasizing the `privnet` custom network.

deployments. Moreover, it contains an overview of two knowledge bases populated using publicly available *compose.yaml* files, commonly used for deploying open-source 5G Core Network infrastructures.

6.4.1 Free 5G Core Solution

Free5GC [81] is a project established for developing an open-source 5G Core Network. This Docker-based 5G Core Network solution is aligned with the 3rd Generation Partnership Project (3GPP) Release 15 [85].

Based on the publicly available Docker Compose file[6] created as part of the Free5GC project, we build a knowledge base named **Free5GC**, included already in Table 6.1. The resulting knowledge base has 15 instances of the class `Service`, one instance of the class `IP Network` and another of the class `IP Address`. For enhanced clarity and visibility, only one instance of the class `Service` and its neighbouring

[6]The *docker-compose.yaml* file is available at https://github.com/free5gc/free5gc-compose/tree/master. The hash of the version used is f37df73be36da7d0752e9a7075e588bcec8e0815.

nodes are presented in Figure 6.3. However, the entire graph structure can be found in the Appendix A.

In this scenario, all services share the network definition since they all connect to the `privnet IP Network`, as presented in Table 6.1 for the **Free5GC** knowledge base. Consequently, the "Connectivity between Services" rule is maintained between all the services since they have connectivity among themselves. Furthermore, there are no specific HTTP ports defined, implying that the 11 *Non-compliances* arise from ports other than HTTP and HTTPS.

6.4.2 Open Air Interface 5G Core Network Solution

Another 5G Core Network deployment, also compliant with the 3GPP *standalone* 5G deployment scenario, specifically Release 16, is provided as part of the OAI project [82]. This project presents a Docker Compose-based solution that is regularly updated. To build the **OAI5GC** knowledge base, we refer to the *Basic 5GC* deployment mode[7].

As presented in Table 6.1, there is a total of nine instances for each of the classes `Service` and `IP Address`, with a single `IP Network` defined. Additionally, the result for the *Custom IP Network* suggests that all the services connect to the same custom network. Therefore, the reasoning engine creates the `hasConnectivityWith` property between each pair of services, based on the "Connectivity between Services" rule. Moreover, with the visual representation of classes, properties and instances, we can observe the networking definitions for each service. In Figure 6.4, we highlight the `oai-amf` instance of the class `Service` to allow a more focused examination of a service and its relationships with instances of the other two classes. However, the knowledge graph corresponding to this case, which includes all the instances of the class `Service`, is available in the Appendix A.

Table 6.1 provides more information about the **OAI5GC** knowledge base. Within this knowledge base, there are 16 *Internal Ports* defined, out of which six are detected as *HTTP Internal Ports*. To investigate the cause behind the seven identified *Non-compliances*, or seven services with forbidden ports defined, we execute the query presented in Listing 6.4. In Table 6.5, we can observe the services which, besides port 80, expose other non-allowed ports internally. As a result, we can see the discrepancy between the service definition and the "Compliance Check" rule.

[7]The *docker-compose-basic-nrf.yaml* file is available at https://gitlab.eurecom.fr/oai/cn5g/oai-cn5g-fed/-/tree/master/docker-compose?ref_type=heads. The commit hash for the version used is 8848fde594081b07e44abd777321a6c29b993363.

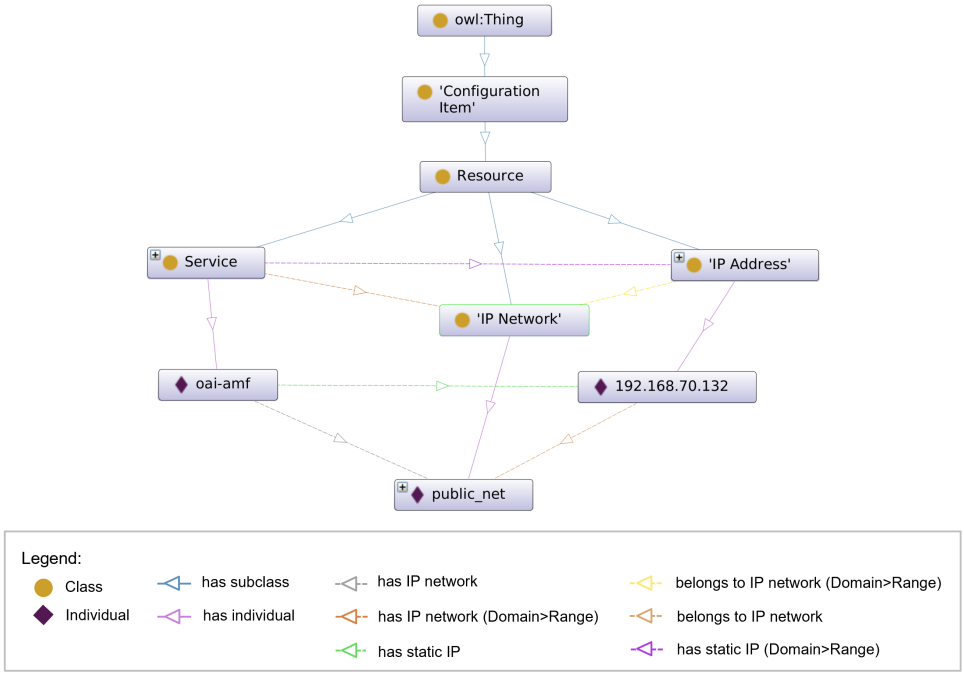


Figure 6.4: Visual representation of classes and object properties of the OAI5GC knowledge base, emphasizing the `oai-amf` service.

6.5 Summary

Based on the survey of different Docker Compose-based applications, we utilize the Container Networking Ontology to capture the implicit knowledge of various IaC solutions that rely solely on the syntax of *compose.yaml* files. Accordingly, we enrich each IaC deployment with explicit semantics and further explore and reason with additional knowledge by resorting to formal logic and graph querying. Moreover, we present scenarios that rely on different technologies, from web applications to the 5G Core network Docker-based deployments.

Our results reveal that the analysis of IaC deployments can be conducted by defining rules and principles that services must comply with. In particular, we demonstrated that these principles can be represented universally (i.e., mapped to well-defined concepts) and be transparently applied to a wide variety of diverse IaC applications without requiring any modification. Besides drawing of new conclusions about these applications, this allows the development of a common understanding between different stakeholders. For instance, based on the cumulative analysis of the

Table 6.5: The result of the query presented in Listing 6.4 (i.e., compliance check), based on OAI5GC.

OAI5GC		
?service1	**?hostPort**	**?internalPort**
:oai-amf		8080/tcp^^xsd:string
:oai-amf		38412/sctp^^xsd:string
:oai-amf		**80/tcp^^xsd:string**
:oai-ausf		8080/tcp^^xsd:string
:oai-ausf		**80/tcp^^xsd:string**
:oai-nrf		8080/tcp^^xsd:string
:oai-nrf		**80/tcp^^xsd:string**
:oai-smf		8080/tcp^^xsd:string
:oai-smf		8805/udp^^xsd:string
:oai-smf		**80/tcp^^xsd:string**
:oai-udm		8080/tcp^^xsd:string
:oai-udm		**80/tcp^^xsd:string**
:oai-udr		8080/tcp^^xsd:string
:oai-udr		**80/tcp^^xsd:string**
:oai-upf		8805/udp^^xsd:string
:oai-upf		2152/udp^^xsd:string

Awesome Compose samples, we can observe that `IP Address` definitions are excluded. This suggests that application developers rely on the default assignment of private IP addresses provided by Docker Compose, disregarding typical network security conventions. Conversely, the developers of the presented 5G Core Networks, likely to be more familiar with network management, opt for static IP address allocations. By having a collective view of different approaches to IaC deployments enabled by our proposed ontology, we believe that sharing best practices and common security policies will be facilitated.

Chapter 7
Concluding Remarks

Throughout this thesis, we capture the definitions of networking concepts within a container-based ICT infrastructure by creating an ontology. By its very nature (cf. Section 2.1), an ontology allows machine interpretation and intelligible definitions of concepts by abstracting their representation. Through our semantically-enriched IaC approach, we explored ontology extension and reuse, the integration of network policies as logic-based rules, the inference of new knowledge, and applied these concepts to assess different applications defined as code.

In the following two sections of this concluding chapter, we discuss our approaches in more detail and provide an overview of contributions through the recapitulation of our research questions.

7.1 Discussion

In this section, we reflect on the possible implications and constraints of the research presented in this thesis. Furthermore, we examine potential avenues for improving our findings.

The scope of the thesis

The high-level objective of this thesis is to demonstrate the potential of a knowledge-based approach to represent IaC deployments and enhance the understanding of the networking domain. To achieve this goal, our scope converged towards Docker, even though we could have referred to other container orchestration platforms such as Kubernetes [86] to validate our approach. However, by narrowing the scope to the Docker and Docker Compose, we have been able to delve deeper into key concepts and achieve progress in adding explicit semantics to the applications defined through *compose.yaml* files.

Besides the ability to formalize the definition of container-based applications, we also demonstrated the practice of inferring new knowledge, based on well-defined concepts within the Container Networking Ontology. Moreover, the analysis of various Compose-based applications provides suggestions for improving the knowledge sharing between stakeholders, such as application developers, network managers and policy officers.

Ontology design

As presented in Chapter 5, the Container Networking Ontology represents the extension of the *Network Infrastructure Ontology* created by Corcho, Chaves-Fraga, Toledo, *et al.* [58]. In our literature search, in which we focused on the ontologies for software-based networks and containers, we identified that prior studies often do not fully comply with the FAIR principles. Furthermore, we observed that an ontology must be publicly accessible to use the definition of concepts within a context which aligns with our determined scope.

The *ontology network* presented in [58] provides definitions of concepts which match our motivational example and follow the standards for ontology reuse. To create the *ontology network*, the authors follow the human-centric ontology design process by considering collaboration with multiple stakeholders. Furthermore, it supports our aspiration to use an ontology as a tool for enhancing the comprehension of network-related concepts of IaC systems and for assisting different human actors in making informed decisions. We acknowledge this effort and extend it to demonstrate its applicability in container-based IaC.

The potential for expanding the ontology

The Container Networking Ontology contains `IP Address`, `IP Network` and `Service` classes, which are mutually disjoint. Based on the definition of concepts within our target platform, Docker, we designed relationships between these three classes using object properties. In addition, we defined the datatype properties used to provide additional details to each class. In particular, the `IP Network` class has an attribute representing the subnet, specified by the datatype property `hasSubnet`. In Docker terms, this class represents a custom or *user-defined network*.

The concept of a network can be further characterized with additional keywords such as `host`, `none`, `ipvlan` or `macvlan` [87]. Other container-specific concepts, such as *link* which allows two containers to communicate using aliases, may also be interesting to consider in the future. Therefore, the Container Networking Ontology has the potential for extension, enabling it to encompass more networking options available in different IaC specifications. This is indicated in the ontology development methodology (cf. Section 4.2), which suggests updates during the ontology life-cycle.

Annotations for a more robust parser

The Container Networking Ontology was designed to enable knowledge representation of containerized platforms beyond Docker. This was the rationale that motivated the exclusion of Docker-specific attributes. However, the developed Data Parsing and Populating Module (cf. Section 5.2) is a prototype and is solely based on the Docker Compose syntax. Nevertheless, this module can be adapted to be agnostic to the underlying containerization platform. For instance, we could resort to annotations in the *compose.yaml* file, or equivalent in other tools, in order to allow more explicit mapping of concepts and properties as well as to add explanations for networks and services.

A similar principle of annotations already exists for HTML documents in the field of Web Semantics. It is achieved with RDFa, which embeds attributes like `typeof` or `property` to make human-readable content in web pages machine-readable too (i.e., hints on the semantics of specific data such as a name or a phone number) [26]. Similarly, we could utilize the comment symbol (#) or YAML's reserved character (@) to explicitly include information about services, such as their connectivity requirements, or even specific network policies across the deployment. This approach intends application developers, network engineers and others to have an active role in including annotations and enhancing the definition of infrastructure with meaning for both humans and machines. Moreover, such an extension would directly map the definition of container-based infrastructure to our ontology, and the parser would not be limited to the Docker Compose. With this in mind, the Data Parsing and Populating Module would be compatible with other container-based platforms using YAML, such as Kubernetes.

Post-deployment knowledge

We utilized the Container Networking Ontology and the Data Parsing and Populating Module to create knowledge graphs of different sizes, representing the content of the Docker Compose files. This would commonly be considered a pre-deployment task, which does not cover operation, management and maintenance of a typical infrastructure. Nevertheless, our model has the potential to be extended beyond adding meaning to static definitions of services and networking configuration. For example, we could resort to a monitoring agent to track changes and populate the knowledge graph. With the up-to-date knowledge base, compliance checks against defined logic rules can be verified in real time while the application is running. By tracking changes, and being assisted by a reasoner, we could detect misconfigurations and gain a deeper understanding of sources for non-compliance. Potentially, this approach could even be used to prevent changes that would cause an issue or even a network security risk.

Increased automation

Handling post-deployment knowledge would require creating an automated process for reasoning over the populated knowledge base. Since our solution presents the proof of the ontology-based concept in IaC, we executed actions of rule creation and consistency checks in *Protégé*. This is a well-known tool among ontology engineers which is fully compatible with OWL, SWRL and other Semantic Web Technologies of interest [76]. However, it has certain limitations, such as the need for manual interaction with the interface and reasoner compatibility for Windows OS. In particular, after significant effort, we realised that the reasoning with Pellet over SWRL rules did not work in this OS, while the same process immediately worked in Linux and Mac.

To understand the potential of automation, we explored the use of general-purpose programming languages like Python. We utilized the *owlready2* package in Python, compatible with Semantic Web Technologies, to create and query the knowledge base. Moreover, we identified the potential of running the reasoner and developing a monitoring agent in one tool (e.g., a Python-based application), which could potentially increase control and efficiency. Nevertheless, we did not manage to reason over logic rules written in SWRL using *owlready*. Therefore, future work would require a detailed investigation of software compatible with Semantic Web Technologies to create a more automated solution.

Human-centric evaluation

The aim of the semantically-enriched IaC is to assist human actors by creating a common understanding of the system. The results presented in Chapter 6 suggest that the Container Networking Ontology can **assist multidisciplinary actors in making informed conclusions**, based on the rules written through DLs. Bearing in mind the results of various deployment scenarios that we assessed, we can verify that *traditional* application developers tend to overlook network management best practices, in contrast with the developers of 5G Core Networks, which include a more explicit network definition in their deployment. By arranging knowledge about the services and network-related definitions in a structured way, we argue that best practices and coding patterns in IaC can be more easily identified, and expertise can be shared.

In addition to knowledge sharing, developers can utilize the Container Networking Ontology to understand better if their defined infrastructure as code is in accordance with the rules specified by experts in varied domains, such as network security officers. For that reason, we could expand our methodology to include an assessment of our model in a real-world environment. With the assistance of stakeholders, especially

experts in the IaC field, we would be able to better understand how IaC practices could be improved by our approach.

7.2 Summary of Findings

In this section, we highlight the key contributions achieved while creating a semantically enriched model for checking networking rules in a container-based infrastructure. Moreover, we provide an overview of the main findings obtained by answering the research questions.

RQ1: What existing ontologies are available for representing knowledge about container networking?

To answer this research question, we performed a literature review, which we organized into three subcategories: (1) ontologies in communication networks, (2) ontologies in Docker and (3) validation of Compose files. We examined these ontologies, focusing on the software-based networks and network management of Information and Communication Technology infrastructures. Moreover, we explored the ontologies related to Docker infrastructure and the different tools used for the Docker Compose files validation. Based on our literature search, we identified that prior research studies have been made in our domain of interest, contributing to the improvement of knowledge management and interoperability. Specifically, we discovered the *DevOps Infrastructure Ontology* that follows the FAIR guidelines and a human-centric ontology design while it also aligns with our high-level objective.

RQ2: How can these ontologies be adjusted to align with the requirements of specific network policies?

Upon reviewing the relevant literature, we observed that the existing ontology representing ICT infrastructure did not incorporate all the core concepts associated with the chosen container-based definitions. Following ontology design best practices, we developed the Container Networking Ontology. For this, we extended the *Network Infrastructure Ontology* by introducing the class `Service`, representing the definition of a Docker container. To validate network-related policies within the containerized infrastructure, we adopted the definition of IP address and network from the *Network Infrastructure Ontology* and created new properties that capture additional networking aspects.

RQ3: How can we integrate generic rules to query knowledge graphs in order to validate network policies?

To address this research question, we designed the Data Parsing and Populating Module, responsible for the construction of a knowledge base for a given IaC definition.

Moreover, we explored the capabilities of Description Logics and Semantic Web Rule Language to establish four groups of rules related to the connectivity between services and to detect possible port exposure vulnerabilities. To examine various Compose-based scenarios, we applied these rules and used a consistent reasoning mechanism to query each knowledge graph. The obtained results confirm our approach's capability of inferring new knowledge using logic-based rules, which can be readily used in assessing IaC deployments.

In conclusion, the research presented in this thesis highlights a knowledge-based approach that can assist humans in understanding the deployment of the container-based infrastructure as code. This is achieved through a human-readable definition of concepts represented in a flexible ontology. Moreover, with a formal knowledge representation and the utilization of Semantic Web Technologies, we demonstrated machine-based interpretation and reasoning of the parsed IaC data. Based on this reasoning, newly inferred insights, such as the detection of non-compliance, may serve as a basis for human actors to make more informed decisions, which can become an integral part of the life cycle of an infrastructure. By enriching IaC with semantics, we enable machines and humans with different expertise to speak the same language.

References

[1] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed. United Kingdom: Pearson Education Limited, 2022.

[2] T. Wood, K. K. Ramakrishnan, *et al.*, "Toward a software-based network: Integrating software defined networking and network function virtualization", *IEEE Network*, vol. 29, no. 3, pp. 36–41, 2015.

[3] W. Attaoui, E. Sabir, *et al.*, "VNF and CNF Placement in 5G: Recent Advances and Future Trends", en, *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, pp. 4698–4733, Dec. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10090468/ (last visited: Dec. 18, 2023).

[4] K. Morris, *Infrastructure as Code*, en. O'Reilly Media, Inc., Dec. 2020, Google-Books-ID: UW4NEAAAQBAJ.

[5] A. Valdes, The Best Infrastructure as Code Tools for 2024. [Online]. Available: https://www.clickittech.com/devops/infrastructure-as-code-tools/ (last visited: Dec. 18, 2023).

[6] M. Guerriero, M. Garriga, *et al.*, "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry", en, in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA: IEEE, Sep. 2019, pp. 580–589. [Online]. Available: https://ieeexplore.ieee.org/document/8919181/ (last visited: Dec. 18, 2023).

[7] L. Leite, C. Rocha, *et al.*, "A Survey of DevOps Concepts and Challenges", en, *ACM Computing Surveys*, vol. 52, no. 6, pp. 1–35, Nov. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3359981 (last visited: Apr. 21, 2023).

[8] Cloud Container Distribution. [Online]. Available: https://www.ericsson.com/en/portfolio/cloud-software-and-services/cloud-core/cloud-infrastructure/cloud-native-infrastructure/cloud-container-distribution (last visited: Dec. 19, 2023).

[9] G. Wikström, P. Persson, *et al.*, 6G – Connecting a cyber-physical world. [Online]. Available: https://www.ericsson.com/en/reports-and-papers/white-papers/a-research-outlook-towards-6g (last visited: Dec. 19, 2023).

[10] A. Simic, "Speaking the same language through logic and ontologies", Department of Information Security, Communication Technology, NTNU – Norwegian University of Science, and Technology, Project report in TTM4502, Nov. 2022.

[11] F. Javier Zorzano Mier and C. Á. Iglesias, "Applications of Knowledge Graphs in Telecommunication Systems Management", *IEEE Internet Computing*, vol. 27, no. 3, pp. 29–34, May 2023, Conference Name: IEEE Internet Computing.

[12] Social Sustainability. [Online]. Available: https://unglobalcompact.org/what-is-gc/our-work/social#:~:text=Social%20sustainability%20is%20about%20identifying,with%20its%20stakeholders%20is%20critical. (last visited: Feb. 19, 2024).

[13] THE 17 GOALS. [Online]. Available: https://sdgs.un.org/goals (last visited: Feb. 19, 2024).

[14] M. D. Wilkinson, M. Dumontier, *et al.*, "The FAIR Guiding Principles for scientific data management and stewardship", en, *Scientific Data*, vol. 3, no. 1, p. 160 018, Mar. 2016, Number: 1 Publisher: Nature Publishing Group. [Online]. Available: https://www.nature.com/articles/sdata201618 (last visited: Sep. 6, 2023).

[15] Logic and Ontology. [Online]. Available: https://plato.stanford.edu/entries/logic-ontology/#Onto (last visited: Dec. 22, 2023).

[16] T. R. Gruber, "A translation approach to portable ontology specifications", *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220, Jun. 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1042814383710083 (last visited: Aug. 30, 2023).

[17] R. Studer, V. R. Benjamins, and D. Fensel, "Knowledge engineering: Principles and methods", *Data & Knowledge Engineering*, vol. 25, no. 1, pp. 161–197, Mar. 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169023X97000566 (last visited: Dec. 22, 2023).

[18] S. Rudolph, "Foundations of Description Logics", en, in *Reasoning Web. Semantic Technologies for the Web of Data*, A. Polleres, C. d'Amato, *et al.*, Eds., vol. 6848, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 76–136. [Online]. Available: http://link.springer.com/10.1007/978-3-642-23032-5_2 (last visited: Sep. 5, 2023).

[19] F. Baader, D. L. McGuinness, *et al.*, *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[20] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web: A New Form of Web Content that is Meaningful to Computers will Unleash a Revolution of New Possibilities", en, in *Linking the World's Information*, O. Seneviratne and J. Hendler, Eds., 1st ed., New York, NY, USA: ACM, Jul. 2023, pp. 91–103. [Online]. Available: https://dl.acm.org/doi/10.1145/3591366.3591376 (last visited: Dec. 23, 2023).

[21] The Semantic Web - Not a piece of cake... [Online]. Available: https://web.archive.org/web/20220628120341/http://bnode.org/blog/2009/07/08/the-semantic-web-not-a-piece-of-cake (last visited: Dec. 23, 2023).

[22] T. Berners-Lee, R. Fielding, and L. Masinter, Uniform Resource Identifier (URI): Generic Syntax, 2005. [Online]. Available: https://www.ietf.org/rfc/rfc3986.txt (last visited: Jan. 8, 2024).

[23] M. Duerst and M. Suignard, Internationalized Resource Identifiers (IRIs), 2005. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3987 (last visited: Jan. 8, 2024).

[24] T. Bray, J. Paoli, *et al.*, Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008. [Online]. Available: https://www.w3.org/TR/xml/ (last visited: Jan. 8, 2024).

[25] D. Beckett, T. Berners-Lee, *et al.*, RDF 1.1 Turtle, 2014. [Online]. Available: https://www.w3.org/TR/turtle/ (last visited: Jan. 8, 2024).

[26] I. Herman, B. Adida, *et al.*, RDFa 1.1 Primer - Third Edition, 2015. [Online]. Available: https://www.w3.org/TR/rdfa-primer/ (last visited: Jan. 8, 2024).

[27] M. Sporny, D. Longley, *et al.*, JSON-LD 1.1, 2020. [Online]. Available: https://www.w3.org/TR/json-ld11/ (last visited: Jan. 8, 2024).

[28] F. Manola, E. Miller, *et al.*, RDF 1.1 Primer, 2014. [Online]. Available: https://www.w3.org/TR/rdf11-primer/#section-data-model (last visited: Jan. 1, 2024).

[29] S. Bechhofer, F. van Harmelen, *et al.*, OWL Web Ontology Language Reference, 2004. [Online]. Available: https://www.w3.org/TR/owl-ref/ (last visited: Jan. 4, 2024).

[30] D. Brickley, R. Guha, and B. McBride, RDF Schema 1.1, 2014. [Online]. Available: https://www.w3.org/TR/rdf-schema/ (last visited: Feb. 1, 2024).

[31] S. Harris, A. Seaborne, and E. Prud'hommeaux, SPARQL 1.1 Query Language, 2013. [Online]. Available: https://www.w3.org/TR/sparql11-query/ (last visited: Jan. 4, 2024).

[32] I. Horrocks, P. F. Patel-Schneider, *et al.*, SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004. [Online]. Available: https://www.w3.org/submissions/SWRL/ (last visited: Jan. 5, 2024).

[33] P. Hitzler, K. Janowicz, and E. Hyvönen, "Using the Semantic Web in digital humanities: Shift from data publishing to data-analysis and serendipitous knowledge discovery", *Semantic Web*, vol. 11, no. 1, pp. 187–193, Jan. 2020. [Online]. Available: https://doi.org/10.3233/SW-190386 (last visited: Jan. 21, 2024).

[34] T. Tudorache, C. I. Nyulas, *et al.*, "Using Semantic Web in ICD-11: Three Years Down the Road", en, in *The Semantic Web – ISWC 2013*, H. Alani, L. Kagal, *et al.*, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2013, pp. 195–211.

[35] J. M. Serrano Orozco, *Applied Ontology Engineering in Cloud Services, Networks and Management Systems*, en. Boston, MA: Springer US, 2012. [Online]. Available: https://link.springer.com/10.1007/978-1-4614-2236-5 (last visited: Sep. 10, 2023).

[36] localhost (Q153799). [Online]. Available: https://www.wikidata.org/wiki/Q153799 (last visited: Jan. 1, 2024).

[37] Wikidata:Introduction. [Online]. Available: https://www.wikidata.org/wiki/Wikidata:Introduction (last visited: Jan. 1, 2024).

[38] Wikidata:Identifiers. [Online]. Available: https://www.wikidata.org/wiki/Wikidata:Identifiers (last visited: Jan. 1, 2024).

[39] D. Allemang, J. Hendler, and F. Gandon, *Semantic Web for the Working Ontologist: Effective Modeling for Linked Data, RDFS, and OWL*, 3rd ed. New York, NY, USA: Association for Computing Machinery, 2020, vol. 33.

[40] About: IP address. [Online]. Available: https://dbpedia.org/page/IP_address (last visited: Jan. 3, 2024).

[41] C. M. Keet, "Open World Assumption", en, in *Encyclopedia of Systems Biology*, W. Dubitzky, O. Wolkenhauer, *et al.*, Eds., New York, NY: Springer, 2013, pp. 1567–1567. [Online]. Available: https://doi.org/10.1007/978-1-4419-9863-7_734 (last visited: Jan. 3, 2024).

[42] Datalog for Data Analysis: A Beginner's Guide. [Online]. Available: https://datalog.dev/article/Datalog_for_data_analysis_A_beginners_guide.html (last visited: Feb. 25, 2024).

[43] Wikidata Query Service. [Online]. Available: https://w.wiki/8hGP (last visited: Jan. 1, 2024).

[44] A. Bhardwaj and C. R. Krishna, "Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey", en, *Arabian Journal for Science and Engineering*, vol. 46, no. 9, pp. 8585–8601, Sep. 2021. [Online]. Available: https://doi.org/10.1007/s13369-021-05553-3 (last visited: Jan. 5, 2024).

[45] Docker overview. [Online]. Available: https://docs.docker.com/get-started/overview/#docker-architecture (last visited: Jan. 8, 2024).

[46] J. Turnbull, The Docker Book, 2016. [Online]. Available: https://dockerbook.com/ (last visited: Jan. 8, 2024).

[47] Networking in Compose. [Online]. Available: https://docs.docker.com/compose/networking/ (last visited: Jan. 8, 2024).

[48] M. H. Ibrahim, M. Sayagh, and A. E. Hassan, "A study of how Docker Compose is used to compose multi-component systems", en, *Empirical Software Engineering*, vol. 26, no. 6, p. 128, Sep. 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-10025-1 (last visited: Jan. 8, 2024).

[49] Docker Compose overview. [Online]. Available: https://docs.docker.com/compose/ (last visited: Jan. 8, 2024).

[50] Why use Compose? [Online]. Available: https://docs.docker.com/compose/intro/features-uses/ (last visited: Jan. 8, 2024).

[51] Docker/Kubernetes Projects. [Online]. Available: https://www.ericsson.com/en/careers/global-locations/poland/dockerkubernetes-projects (last visited: Jan. 26, 2024).

[52] 4G/5G Networks Software. [Online]. Available: https://www.northeastern.edu/coloseum/cellular-software/ (last visited: Jan. 26, 2024).

[53] Q. Zhou, A. J. G. Gray, and S. McLaughlin, "ToCo: An Ontology for Representing Hybrid Telecommunication Networks", en, in *The Semantic Web*, vol. 11503, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019, pp. 507–522. [Online]. Available: http://link.springer.com/10.1007/978-3-030-21348-0_33 (last visited: Jan. 31, 2023).

[54] Z. Qianru, TOUCAN Ontology (ToCo). [Online]. Available: https://github.com/QianruZhou333/toco_ontology (last visited: Aug. 4, 2023).

[55] J. C. C. Tesolin, A. M. Demori, *et al.*, "Enhancing heterogeneous mobile network management based on a well-founded reference ontology", *Future Generation Computer Systems*, vol. 149, pp. 577–593, Dec. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X23003084 (last visited: Jan. 22, 2024).

[56] N. F. Saraiva de Sousa, D. Lachos Perez, *et al.*, "End-to-End Service Monitoring for Zero-Touch Networks", en, *Journal of ICT Standardization*, May 2021. [Online]. Available: https://journals.riverpublishers.com/index.php/JICTS/article/view/5789 (last visited: May 31, 2023).

[57] About SLSA. [Online]. Available: https://slsa.dev/spec/v1.0/about (last visited: Jan. 26, 2024).

[58] O. Corcho, D. Chaves-Fraga, *et al.*, "A High-Level Ontology Network for ICT Infrastructures", en, in *The Semantic Web – ISWC 2021*, A. Hotho, E. Blomqvist, *et al.*, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2021, pp. 446–462.

[59] A. Martinez, M. Yannuzzi, *et al.*, "An Ontology-Based Information Extraction System for bridging the configuration gap in hybrid SDN environments", en, in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Ottawa, ON, Canada: IEEE, May 2015, pp. 441–449. [Online]. Available: http://ieeexplore.ieee.org/document/7140321/ (last visited: Jan. 31, 2023).

[60] H. A. Tran, D. Tran, *et al.*, "A novel SDN controller based on Ontology and Global Optimization for heterogeneous IoT architecture", en, in *Proceedings of the Eighth International Symposium on Information and Communication Technology*, Nha Trang City Viet Nam: ACM, Dec. 2017, pp. 293–300. [Online]. Available: https://dl.acm.org/doi/10.1145/3155133.3155143 (last visited: Jan. 31, 2023).

[61] Q. Zhou, A. J. G. Gray, and S. McLaughlin, *SeaNet – Towards A Knowledge Graph Based Autonomic Management of Software Defined Networks*, arXiv:2106.13367 [cs], May 2022. [Online]. Available: http://arxiv.org/abs/2106.13367 (last visited: Aug. 4, 2023).

[62] R. Tommasini, B. D. Meester, *et al.*, "Representing Dockerfiles in RDF", en, [Online]. Available: https://ceur-ws.org/Vol-1963/paper528.pdf (last visited: Aug. 15, 2023).

[63] D. Huo, J. Nabrzyski, and C. F. V. Ii, "Smart Container: An ontology towards conceptualizing Docker", en, [Online]. Available: https://ceur-ws.org/Vol-1486/paper_89.pdf (last visited: Aug. 15, 2023).

[64] M. Osorio, C. Buil-Aranda, *et al.*, "DockerPedia: A Knowledge Graph of Software Images and Their Metadata", en, *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, no. 01, pp. 71–89, Jan. 2022. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0218194022500036 (last visited: Feb. 15, 2023).

[65] J. Zhou, W. Chen, *et al.*, "DockerKG: A Knowledge Graph of Docker Artifacts", en, in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, Seoul Republic of Korea: ACM, Jun. 2020, pp. 367–372. [Online]. Available: https://dl.acm.org/doi/10.1145/3387940.3392161 (last visited: Aug. 1, 2023).

[66] L. Jonathan, Docker Ontology, 2017. [Online]. Available: https://www.w3.org/community/bigdata-tools/2017/10/30/docker-ontology/ (last visited: Jan. 8, 2024).

[67] Docker Ontology, 2017. [Online]. Available: https://github.com/langens-jonathan/docker-vocab/blob/master/docker.md (last visited: Jan. 8, 2024).

[68] K. Boukadi, M. Rekik, *et al.*, "Container description ontology for CaaS", *International Journal of Web and Grid Services*, vol. 16, no. 4, pp. 341–363, Jan. 2020. [Online]. Available: https://doi.org/10.1504/ijwgs.2020.110944 (last visited: Nov. 20, 2023).

[69] Introducing the Docker Compose Validator, 2019. [Online]. Available: https://blog.zhaw.ch/splab/2019/10/04/introducing-the-docker-compose-validator/ (last visited: Jan. 8, 2024).

[70] DockStation, 2017. [Online]. Available: https://dockstation.io/ (last visited: Jan. 8, 2024).

[71] B. Piedade, J. P. Dias, and F. F. Correia, "Visual notations in container orchestrations: An empirical study with Docker Compose", en, *Software and Systems Modeling*, vol. 21, no. 5, pp. 1983–2005, Oct. 2022. [Online]. Available: https://doi.org/10.1007/s10270-022-01027-8 (last visited: Jan. 16, 2024).

[72] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*, en. Berlin, Heidelberg: Springer, 2014. [Online]. Available: https://link.springer.com/10.1007/978-3-662-43839-8 (last visited: Oct. 13, 2023).

[73] M. Poveda-Villalón, A. Fernández-Izquierdo, *et al.*, "LOT: An industrial oriented ontology engineering framework", *Engineering Applications of Artificial Intelligence*, vol. 111, p. 104 755, May 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197622000525 (last visited: Sep. 7, 2023).

[74] N. F. Noy and D. L. McGuinness, Ontology Development 101: A Guide to Creating Your First Ontology. [Online]. Available: https://protege.stanford.edu/publications/ontology_development/ontology101.pdf (last visited: Jan. 24, 2023).

[75] M. Grüninger and M. S. Fox, "The role of competency questions in enterprise engineering", in *Benchmarking — Theory and Practice*, A. Rolstadås, Ed. Boston, MA: Springer US, 1995, pp. 22–31. [Online]. Available: https://doi.org/10.1007/978-0-387-34847-6_3.

[76] M. A. Musen, "The Protégé Project: A Look Back and a Look Forward", *AI matters*, vol. 1, no. 4, pp. 4–12, Jun. 2015. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4883684/ (last visited: Jan. 28, 2024).

[77] J.-B. Lamy, "Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies", *Artificial Intelligence in Medicine*, vol. 80, pp. 11–28, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0933365717300271.

[78] P. Nathan, *kglab: a simple abstraction layer in Python for building knowledge graphs*, 2020. [Online]. Available: https://github.com/DerwenAI/kglab.

[79] Pellet. [Online]. Available: https://www.w3.org/2001/sw/wiki/Pellet (last visited: Feb. 1, 2024).

[80] LargeTripleStores - W3C Wiki. [Online]. Available: https://www.w3.org/wiki/LargeTripleStores (last visited: Feb. 16, 2024).

[81] What is free5GC? [Online]. Available: https://free5gc.org/ (last visited: Feb. 1, 2024).

[82] 5G CORE NETWORK. [Online]. Available: https://openairinterface.org/oai-5g-core-network-project/ (last visited: Feb. 1, 2024).

[83] N. Wehbe, H. A. Alameddine, *et al.*, "A Security Assessment of HTTP/2 Usage in 5G Service-Based Architecture", *IEEE Communications Magazine*, vol. 61, no. 1, pp. 48–54, Jan. 2023, Conference Name: IEEE Communications Magazine. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9952199 (last visited: Feb. 8, 2024).

[84] J. J. Mats, Your guide to building a cloud native infrastructure for 5G. [Online]. Available: https://www.ericsson.com/en/blog/2020/10/guide-to-building-cloudnative-infrastructure (last visited: Feb. 21, 2024).

[85] Releases. [Online]. Available: https://www.3gpp.org/specifications-technologies/releases (last visited: Feb. 12, 2024).

[86] Why you need Kubernetes and what it can do. [Online]. Available: https://kubernetes.io/docs/concepts/overview/ (last visited: Feb. 17, 2024).

[87] Network drivers overview. [Online]. Available: https://docs.docker.com/network/drivers/ (last visited: Feb. 17, 2024).

Appendix

In-depth Knowledge Base Visualization

This Appendix provides a more comprehensive view of Docker-based 5G Core Network deployments, presented in Chapter 6. The following two sections present the **Free5GC** [81] and the **OAI5GC** [82] knowledge base, visualized with Ontograf, in *Protégé* [76].

A.1 Free5GC Knowledge Graph

As presented in Section 6.4.1, the **Free5GC** knowledge base consists of 15 instances of the class `Service`, which is also shown in Figure A.1 (extended view of Figure 6.3). In this knowledge graph, we can observe one instance of the class `IP Network` and verify that all instances of the class `Service` are connected to this custom network via the `has IP Network` object property. As a result, we can confirm that all pairs of instances of the class `Service` have the connectivity between each other, even without querying data with SPARQL. Moreover, we can see that static IP allocation has been defined only for the `free5gc-n3iwf` service.

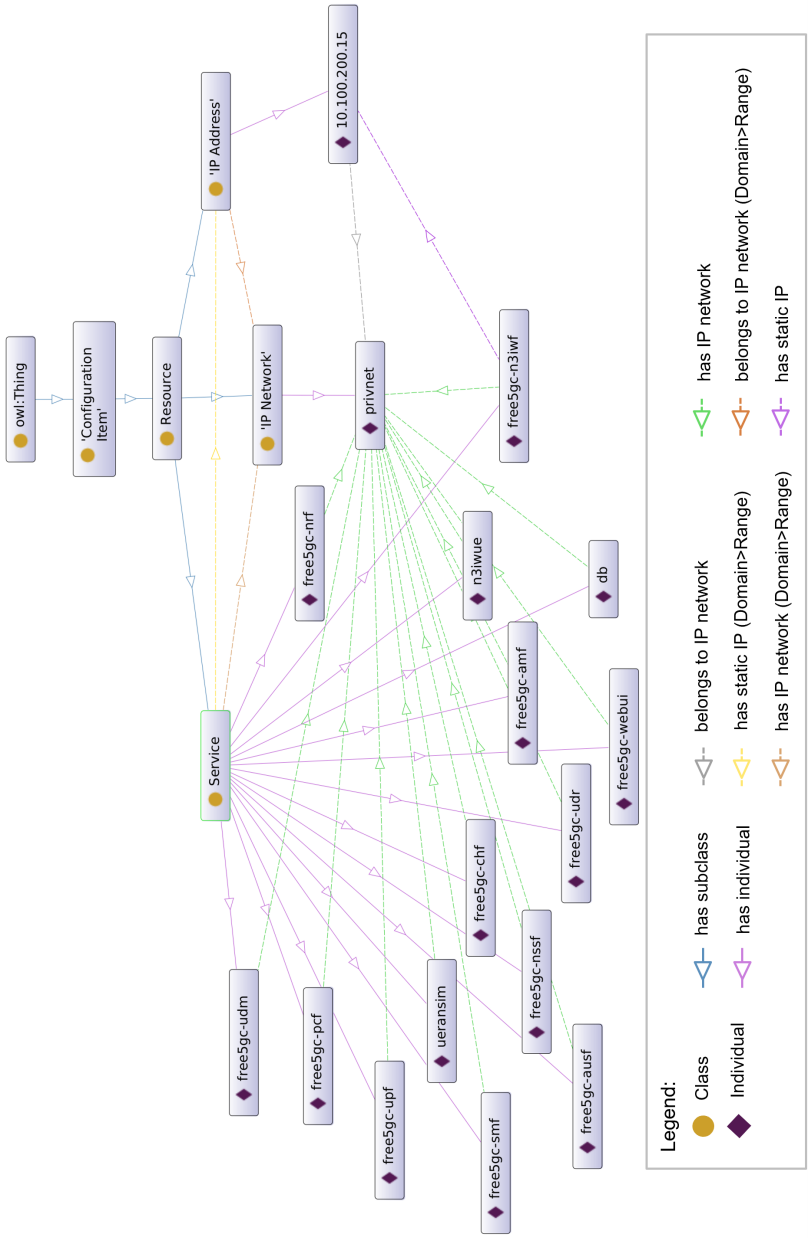


Figure A.1: Visual representation of classes, instances and object properties of the Free5GC knowledge base.

A.2 OAI5GC Knowledge Graph

Figure A.2 shows a more detailed knowledge graph compared to the visual representation presented in Section 6.4.2. Here, we can observe all the instances of the class `Service`, their relationship with the `public_net` instance of the class `IP Network` and all the instances of the class `IP Address`. The knowledge graph presented in Figure A.2 does not include the `has static IP` object property between all instances of the class `Service` and `public_net` network. To reduce the number of edges in the graph, similarly as in Figure 6.4, we emphasize the `oai-amf` service.

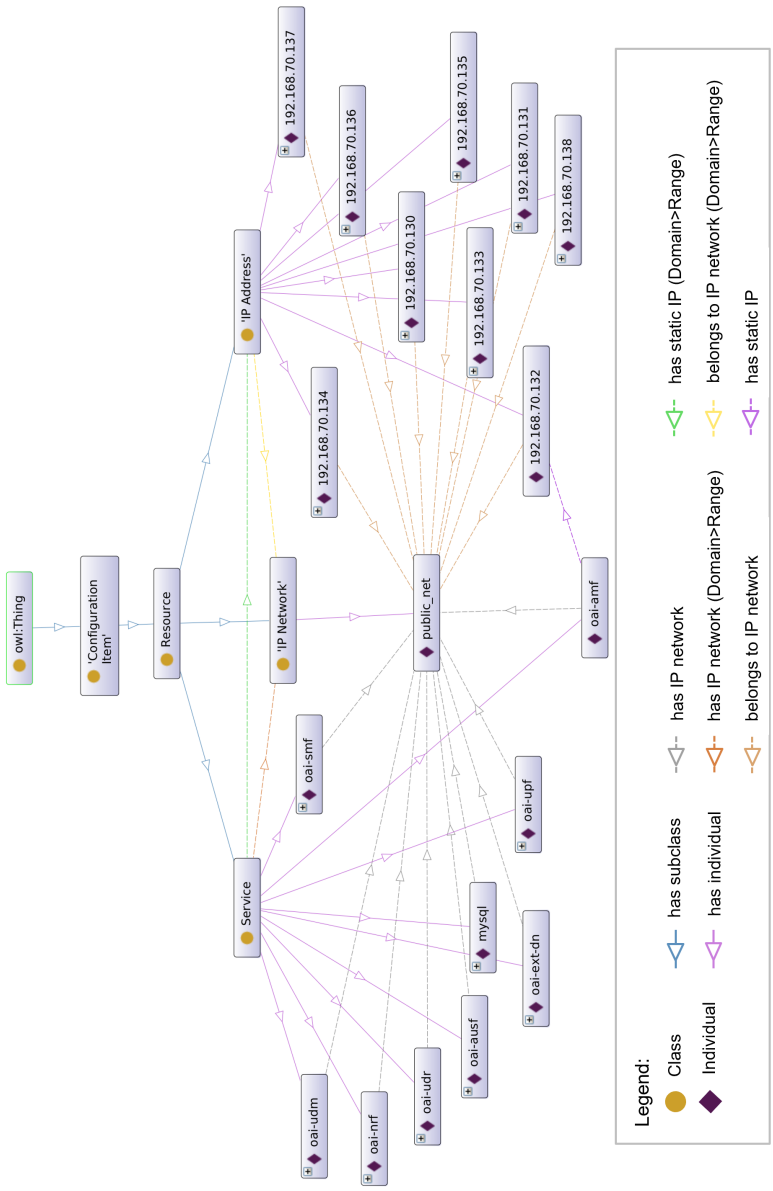
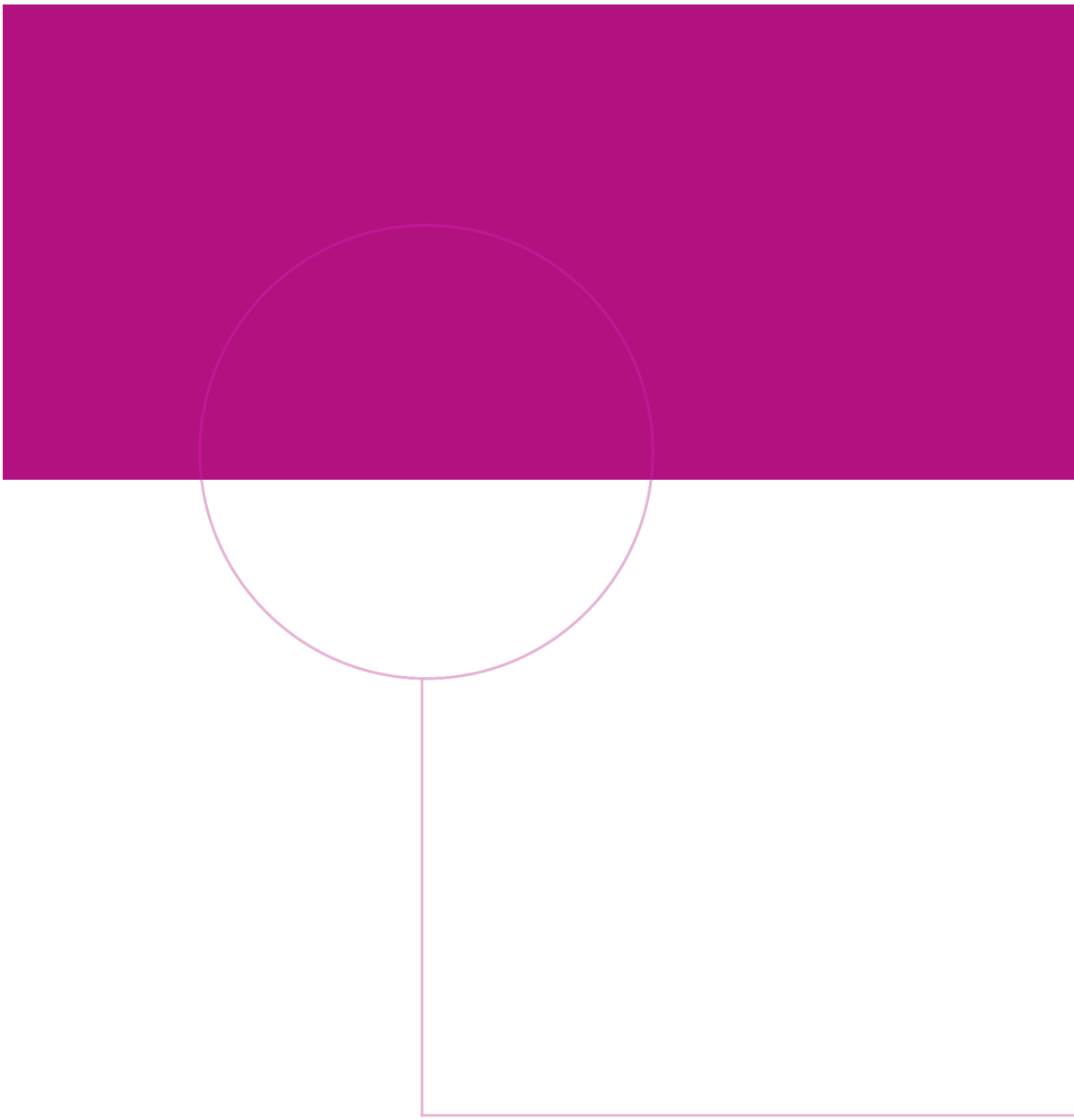


Figure A.2: Visual representation of classes, instances and object properties of the OAI5GC knowledge base.



NTNU

Norwegian University of
Science and Technology