Container Description Ontology for CaaS

Khouloud Boukadi

MIRACL, Sfax University, Tunisia
E-mail: khouloud.boukadi@fsegs.usf.tn

Molka Rekik*

[1]Department of Information Systems, Prince Sattam Bin Abdulaziz University, AlKharj, KSA
[2]MIRACL, Sfax University, Tunisia
m.rekik@psau.edu.sa
*Corresponding author

Jorge Bernal Bernabe

Department of Information and Communications Engineering,
University of Murcia, Spain

Jaime Lloret

Integrated Management Coastal Research Institute, Universidad
Politecnica de Valencia, Valencia, Spain

Abstract: Besides its classical three service models (IaaS, PaaS, and SaaS), Container as a Service (CaaS) has gained significant acceptance. It offers without the difficulty of high-performance challenges of traditional hypervisors deployable applications. As the adoption of containers is increasingly wide spreading, the use of tools to manage them across the infrastructure becomes a vital necessity. In this paper, we propose a conceptualization of a domain ontology for the container description called CDO. CDO presents, in a detailed and equal manner, the functional and non-functional capabilities of containers, dockers and container orchestration systems. In addition, we provide a framework that aims at simplifying the container management not only for the users but also for the cloud providers. In fact, this framework serves to populate CDO, help the users to deploy their application on a container orchestration system, and enhance interoperability between the cloud providers by providing migration service for deploying applications among different host platforms. Finally, the CDO effectiveness is demonstrated relying on a real case study on the deployment of a micro-service application over a containerized environment under a set of functional and non-functional requirements.

Keywords: Container as a Service; Docker; Container Orchestration System; Ontology; Container Discovery.

1 Introduction

More and more the enterprises are turning to the cloud computing thanks to their low costs and availability of services and data from any location, on any type of media at any time [12, 15, 3]. Besides its classical three service models (IaaS, PaaS, and SaaS), cloud computing has been recently empowered by a new service offering called Containers-as-a-Service (CaaS) [18]. Container-based virtualization is gaining significant acceptance because it offers a lightweight solution that allows bundling applications and data in a simpler and more performance-oriented manner, making them runnable on different cloud infrastructures. In a recent study, Cloud Foundry has reported that the adoption of container technologies is on fire, with 53% of the enterprises either investigating or using containers in development or in production [5].

As the use and the wide adoption of containers rises, the need for tools to manage them across the infrastructure becomes a vital necessity. The container management and automation tools represent a hot area for development as organizations race to fill the growing need to manage highly distributed cloud-native applications. Major cloud providers, including Amazon Web Services (AWS), Azure and Google, offer container services and orchestration tools to manage the container creation and deployment. Orchestrating a cluster of containers is a competitive and rapidly evolving area where many tools are proposed offering various feature sets. Container orchestration tools can be broadly defined as providing a dedicated framework for the integration and management of containers at scale. These tools aim to simplify the container management by providing a framework not only for defining container deployment but also for managing multiple containers as one entity for availability, scaling, and networking.

Not all orchestration systems are equally created, and some of them have particular strengths and functionalities that are worth considering. In fact, some of these orchestration systems provide framework(s) to deploy multiple containers, provide container clusters using cloud VMs, and/or compact multiple apps onto a premise or public cloud infrastructure, etc. Deploying an application on a container orchestration system can be a challenging task especially when relying on public cloud providers for a particular application workload. The container orchestration system often have similar services described differently or provide different capabilities. For instance, Amazon EC2 Container Service (ECS) offers a container management service that supports Docker containers for running applications on a managed cluster of Amazon EC2 instances. While ECS uses its proprietary orchestrator, Microsoft Azure orchestrates using DC/OS, Docker Swarm, or Kubernetes. Some of the orchestration tools can be installed on-premise or in most public clouds, while others can be offered only as a hosted solution [21]. Such heterogeneous usage mode complicates the container orchestration system discovery. It is worsened by the absence of a standard language for describing the container tools and services. Therefore, the application of ontology in container orchestration systems needs to be emphasized and a systematic approach for the application needs to be developed.

We propose in this paper to conceptualize ontology for container description. The proposed ontology, which is called Container Description Ontology (CDO), semantically and structurally represents container specification along with the orchestration systems. The CDO is linked to an existing ontology [25] to point to some classes. To recapitulate, this paper makes a three-fold contribution: (i) the conceptualization and population of a domain ontology dealing with container, Docker and container orchestration system related concepts and properties; (ii) a framework dedicated for managing the ontology, which is composed of a set of modules that provides more flexibility and plain basis for further system enhancement. This framework can be used by both the final users and CaaS providers to help them with the discovery of container orchestration system and the migration of the containers instances into different host platforms achieving interoperability, and (iii) the proposition of capabilities-based inference rules for a pertinent container orchestrator discovery.

The remainder of the paper is organized as follows: Sect. 2 summarizes the existing works on semantic descriptions of virtualization principles and management. Sect. 3 presents a detailed design and construction of the ontology. Sect. 4 depicts the proposed framework for the CDO population, the container migration and the reasoning upon the CDO ontology. In Sect. 5, we evaluate the ontology based on its application on a real case study. Finally, we conclude the paper and give some directions for future work.

2 Related work

Cloud-related standards like Open Cloud Computing Interface (OCCI), Cloud Infrastructure Management Interface (CIMI), and Open Virtual Format (OVF) already cover the management of virtual infrastructure level elements. Precisely, they manage the naming, addressing, identity, presenting, messaging, as well as identifying the objects involved in the means to transfer, store, and process information.

In this sense, in [2], the authors propose a capability container software based on an ontology, which offers a wider view qualification to improve the discovery and reuse of REST-based services. The proposed EACP ontology is based on an Enterprise Architecture Capability Profile offering a qualification covering business, operational and technical aspects for services. The qualification profile is based on a meta-model that helps to retrieve and gather initial requirements used to guide the development of existing REST-based Web Applications. Furthermore, a Framework is proposed to exploit the designed container in order to respond to users requirements for developing future business process and efficiently reuse the qualified services.

Rekik et al. [20] propose a Cloud Service description Ontology (CSO) based on cloud standards. CSO covers the functional and non-functional capabilities of the three main cloud provision models (IaaS, PaaS, SaaS). To ensure the quality of the CSO ontology, they define an evaluation approach that detects and corrects consistency, redundancy and incompleteness errors.

Similarly, the mOSAIC ontology [16] models the cloud concepts and definitions according to National Institue of Standards and Technologies (NIST). The ontology allows modeling the proper cloud resources at the IaaS level, including virtual machines. The mOSAIC infrastructure makes use of a semantic engine in order to handle the ontology and to assist customers with the selection of API components and functionalities needed for building new cloud applications. The mOSAIC does not properly cover security aspects, and SOFIC [4] extended mOSAIC to enable security assessments in the Intercloud.

Although the aforementioned ontologies deal with virtualization principles and management, they do not cover container-based virtualization and management, as it is done in the paper presented herein. In this sense, Ayed et al. [1] present an ontology that describes docker images reusing concepts from SIOC and PROV vocabularies. Besides, they automatically extract data from the Docker Hub Registry to populate such ontology.

In [11], the ontology called Smart Container (SC) conceptualizes Docker software objects. SC ontology is aligned with other existing ontologies, such a PROV-O and Core Software Ontology. It provides a mechanism that can capture the provenance of Docker containers as artifacts themselves and potential enable sharing Docker information on the Semantic Web via Linked Data principles.

In [6], the authors provide a simple framework to address the preservation of the docker containers and their environment. They create a domain specific language to ensure that docker files be reused at a later stage, recreating the original environment. It reuses PROV and Functional Requirements for Bibliographic Record (FRBR) ontologies.

Recently, the Big Data Analysis Community Group of the W3C populate new Docker Ontology [25] that models the main semantic vocabularies of the docker ecosystem. The current ontology version is 'alpha' quality and it basically ports to OWL classes and properties of the JSON structure taken from docker instances. Our ontology also models the main classes of the docker instance in OWL, such as *HostConfig*, *State*, *Mounts*, and *NetworkSettings*. Nonetheless, our ontology aims at covering a broader spectrum, since it also includes semantic vocabularies for container orchestration systems and inference rules for a pertinent container orchestrator discovery.

3 Container Description Ontology (CDO)

3.1 Construction Methodology

We proceed to conceptualize our ontology according to NIST and by analyzing TOSCA, CMN, CNI and OCI standards as well as research papers from the literature. To create the ontology, we use NOY and McGuinness methodology [17] since it is the most cited and used methodology. This includes the following steps: (1) Ontology domain and goal identification, (2) Existing ontology examination, (3) Concepts and hierarchy definition, (4) Data and object property definition, (5) Property facet definition as well as domains, and ranges and cardinality definition, and finally (6) Class instantiation from the real world. On the other hand, we follow the design principles, presented on [10], which are objective criteria for guiding and evaluating ontology designs. In addition to these principles, we followed the notion of Ontology as a Service (OaaS) introduced in [7]. In fact, the authors proposed sub-ontology extraction and merging, whereby multiple sub-ontologies are extracted from various source ontologies, and then merged to form a complete ontology to be used by the end-user. Such an extraction method helps to facilitate the elaboration of the ontology. By following the presented methodology, we propose the *Thing* class which covers all the CDO's concepts (see Figure 1). In addition, we adopt the Protégé editor [19] to create, visualize and manipulate our container description ontology. In what follows, we detail the main classes.

3.2 CDO Main Concepts

3.2.1 Container Class

A container is an OS-level virtualization for running multiple isolated systems on a single host. Referring to [25], the Container class has a host configuration, amounts of CPU, memory, and block IO, a state, and network settings. In addition, a container has a set of libraries and isolation features (*Resource_Isolation*, *Process_Isolation*, *Network_Isolation*,

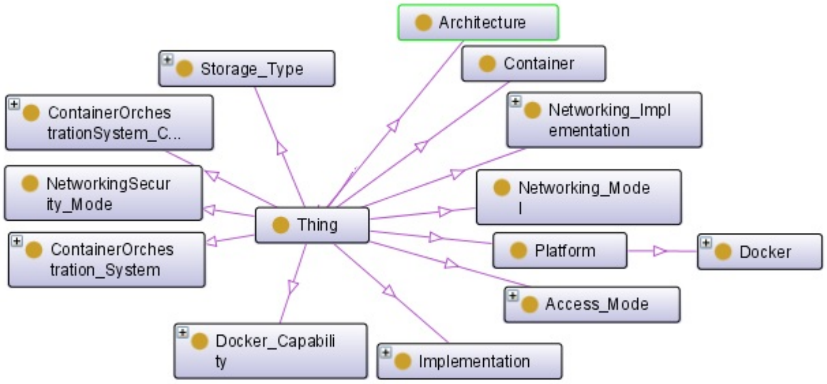


Figure 1 Thing Class: CDO Main Concepts

and *FileSystem_Isolation*) provided respectively through Linux *Cgroups*, *Pid*, *Namespaces* and *Chroot*. Obviously, the container uses a Docker and in reverse the Docker manages a container (see Figure 2).

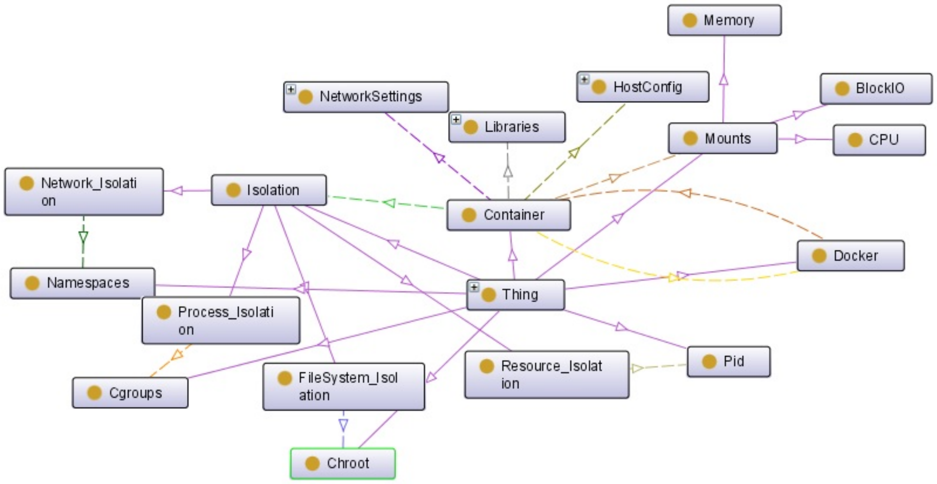


Figure 2 Container Class

3.2.2 Docker Class

Docker is an open source platform that enables the development, shipping, and running of applications as containers. Docker containers can run in multiple infrastructures including VMs, bare-metal servers, and public cloud instances. The major cloud providers, such as AWS, Azure and Google, support the docker containers. Docker manages container by managing namespaces for each container, which are: Process ID (PID), Networking, InterProcess Communication (IPC), Mount (MNT) namespace and UNIX Timesharing

System (UTS). Docker uses control group (cgroups) to manage available hardware resources among containers and a lightweight file system called (UnionFS) to prove the building blocks for containers. Then Docker combines these components into a container format called libcontainer [14].

The Docker container is created using an image which is a read-only template containing everything required by the application and from which the container is launched. The description of a Docker container image is presented in a configuration file named Image_File. As shown in Figure 3, the description could contain:

- URL: this class includes Description_URL, ImageBug_URL, Company_URL, MetaDataDetails_URL, and HelpPage_URL sub-classes to specify, respectively, URLs for the pages (i) that provide information of the image like installation parameters , (ii) in which issues and bugs are reported, (iii) of companies supporting this image, (iv) of repository information describing image metadata used to enrich the description of the docker image , and (v) providing help about this image.

- Supported architecture: it defines the supported computer architectures (e.g. amd64, arm32).

- Supported Docker version: it presents the supported docker versions (e.g. v17.09.0-ce).

- Local: it specifies a link indicating the exact location of the docker image files.

- Command: this class serves to automatically perform actions on a base image. It is divided into three sub-classes: (i) Pull_Command: command needed to download the image (e.g. docker pull Ubuntu), (ii) Run_Command: command needed to run the image (e.g. docker run ubuntu:16.04 grep -v ’#̂’ /etc/apt/sources.list), and (iii) Default_Command: command used to build a Docker on the host.

- Manifest MIME: it provides information about the image in a JSON text file. It serves to install a docker to the home-screen of a host, providing users with quicker access along with other capabilities such as being available offline and receiving push notifications.

- Supported tag: the user can group his images together using tags, and then upload them to share images via repositories.

- Image platform: it specifies the environment in which a docker is executed (e.g. windows amd64).

- Package: this class provides information about all the packages used in the docker file.

Once a Docker image is built, it can be stored in a *Public* or *Private Registry* and then be searched, removed, updated, to serve as the basis of other images. Indeed, a Docker registry is setup to store Docker images. The Docker Daemon will able to search and pull the Docker Image from the Docker registry. The Docker includes a huge Docker repository called *Docker_Hub*. The Docker hub provides both public and private storage for images. The public storage is searchable and can be downloaded by anyone. Private storage is excluded from the search results and only authorized users can pull images down and use them to run containers.

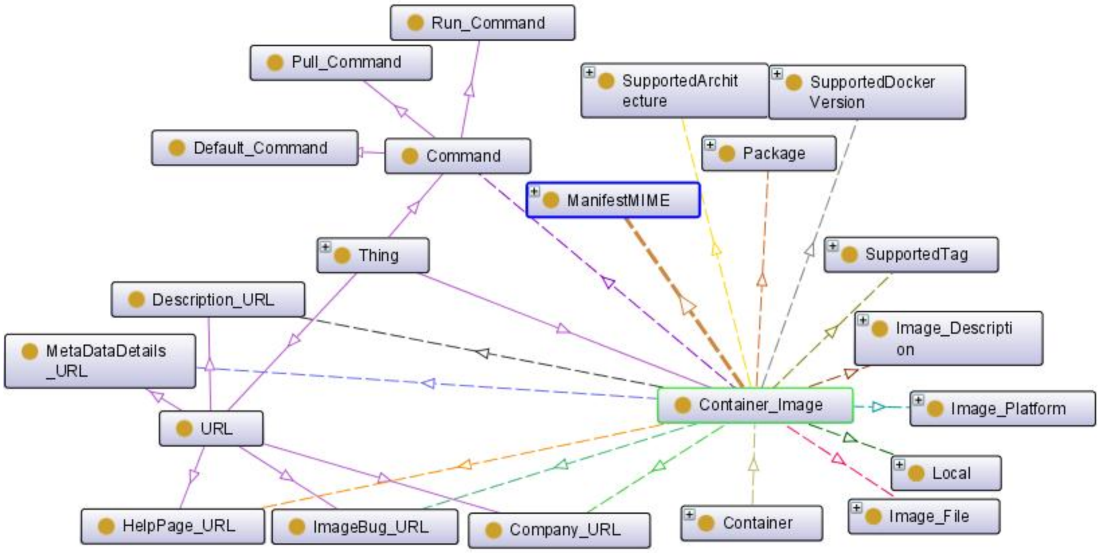


Figure 3 Docker_Image Class

3.2.3 Docker_Capability Class

Based on [21], the Docker has a set of functional and non-functional capabilities (see Figure 4). The functional capabilities cover:

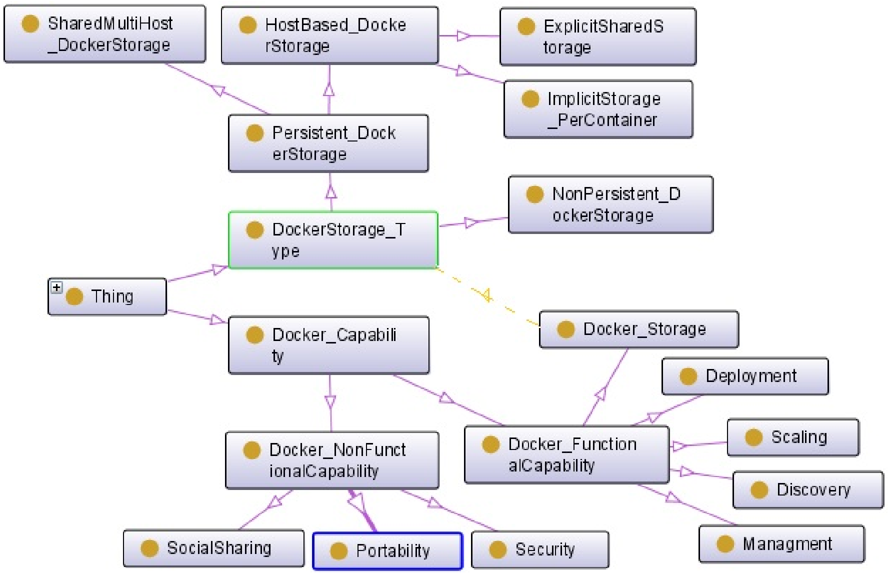


Figure 4 Docker_Capability Class

- Discovery: it is enabled thanks to the Docker registries.

- Deployment: since they do not had boot up process, the Docker containers can be deployed on desktops, physical servers, virtual machines, data centers, and up to public and private clouds in a very short time. By consequently, the application process within a container is able to start instantaneously.

- Docker_Storage: it has a type (*DockerStorage_Type*) that can be persistent or non-persistent Docker storage. Persistent can be host-based or multi-host shared Docker storage. The *HostBased_DockerStorage* is one of the early implementations of data persistence in containers. This kind of storage supposes that containers depend on the underlying host. It can be *ImplicitStorage_PerContainer* or *ExplicitSharedStorage*. The first creates an implicit storage sandbox for the container that requests host-based persistence. The second exposes an explicit location on the host file-system as a mount within one or more containers. It is needed to share data across multiple containers running on the same host. The *SharedMultiHost_DockerStorage* addresses the container non-portability issue caused by *HostBased_DockerStorage*. It requires a distributed storage, which is made available to all the hosts and is then exposed to the containers through a consistent namespace. In addition, caching data via shared and replicated volumes offers decoupling data from services.

- Scaling: Docker's lightweight containers make scaling up and down easier. More containers can be launched when needed and then shut them down easily when they are no longer used.

- Management and migration: a Docker container is created and maintained differently than hypervisor-based virtual machines. The entire environment, except the host OS, is created and recreated on each update or a change made inside the container. In addition, after configure an application in a Docker container, the Docker container is easier to shift from one location to another.

The non-functional capabilities of the Docker_Capability class include:

- Security: the standard security features of the Linux kernel are implemented into docker, and added as a layer of configurable amendments to the containers executed by the docker engine. This layer considers the risk management needed when containers are scaled upward with accelerating release velocity. Indeed, it enables an easier management of some security configurations and limits containers to interact with the appropriate resources.

- Portability: docker container is a simple directory which can be compressed and copied anywhere.

- Social sharing: the developers can share container images in a public or private way, something that enables and encourages the development of the platform. The images of the new containers can be based on other ones built by other developers.

3.2.4 Container_OrchestrationSystem Class

The container orchestration system also named container scheduler or container cluster is a tool for integrating and managing containers at scale. The main task of the container orchestration system is to launch containers on the adequate host and connect them together. Usually, the container orchestration system follows a master/slave *Architecture*.

ContainerOrchestrationSystem_Capability Class

The container orchestration system has a set of functional and non-functional capabilities. As shown in Figure 5, the functional ones cover:

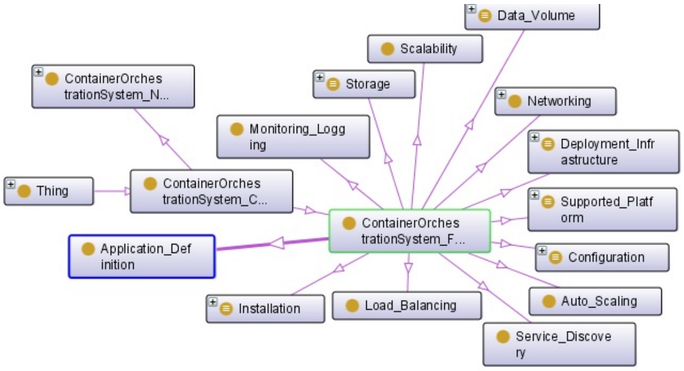


Figure 5 ContainerOrchestrationSystem_FunctionalCapability Class

- Application_Definition: it specifies the blueprint for an application workload and its configuration in a standard schema, using languages such as YAML and JSON. It can also include the container image repositories, ports, storage, etc.
- Deployment_Infrastructure: which is also called Supported_Platform, consists in running containers either on a physical infrastructure, or outside on a virtual infrastructure of private or public clouds.
- Storage: which is also called Data_Volume, consists in persisting data in a container writable layer.
- Service_Discovery: container orchestration systems provide a distributed key-value store, a lightweight DNS or some other mechanism to enable the discovery of containers. Some container orchestration systems, such as the Nomad, lack this capability.
- Scalability: it is the ability to schedule any number of container replicas across a group of node instances to meet an applications scale.
- Networking: it connects containers running on the cluster nodes.
- Monitoring: it consists in tracking the health of the containers and hosts in the cluster. If a container crashes, a new one can be spun up. When a host fails, the container orchestration system will restart the failed containers on another host. It will also run specified health checks at the appropriate frequency and update the list of available nodes based on the results.
- Load_Balancing: it is the capability to load balance requests across any of the containers on any of the hosts in the cluster.

- Provisioning: it determines the right placement for the containers by selecting an appropriate host based on the specified constraints such as resource requirements, location affinity, etc.
- Configuration: also called installation, is the ability to make the orchestration system ready for execution.

However, as shown in Figure 6, the non-functional capabilities are:

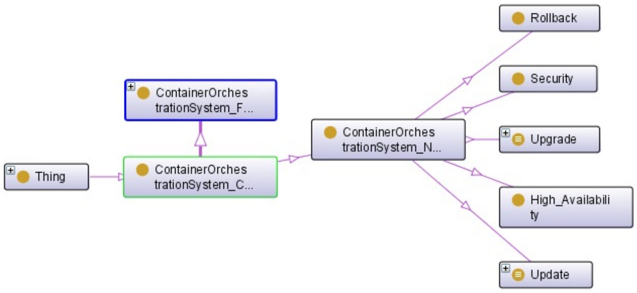

Thing
ContainerOrchestrationSystem_C...
ContainerOrchestrationSystem_F...
ContainerOrchestrationSystem_N...
Rollback
Security
Upgrade
High_Availability
Update

Figure 6 ContainerOrchestrationSystem_NonFunctionalCapability Class

- Availability: it is provided through container replication and service redundancy. The same container is deployed on multiple nodes to provide redundancy and redeployed again if a host running the service goes down making the service self-healing.
- Rolling upgrades: once a new version of an application is available, rolling upgrades, incrementally across the cluster, should be performed.
- Rollback: if the new version of an updated application does not perform as expected, then the container cluster system can roll back the applied change.
- Security: is a critical element of the container orchestration system. It deals with the used security mechanisms, such as role-based access control, transport layer security (TLS), X.509 certificate or token-based, etc. For example, the container orchestration tool should ensure that the container images be regularly scanned for vulnerabilities, and that the images are digitally signed.

Storage Class

Container orchestration system offers *NonPersistent_Storage* and/or *Persistent_Storage* types. The latter has an *Access_Mode* which may be: Read Write Once (*RWO*), Read Only Many (*ROX*), and/or Read Write Many (*RWX*) describing how the storage volume can be mounted by nodes. *SharedMultiHost_Storage*, which is a persistent storage used by almost all the container orchestration systems, takes advantage of a distributed filesystem combined with the explicit storage technique. Examples of shared filesystems, such as Ceph, GlusterFS, Network File System (NFS) and others, can be used to configure a distributed filesystem on each host running the containers. In addition, some container orchestration systems can propose *NonPersistant_Storage* that can be used to read and write files with a container. To extend the capabilities of the containers to a variety of storage backends, container orchestration system employs *Storage_Plugin*. For instance, Flocker, which is a storage plugin that works with Docker Swarm, Kubernetes and Mesos, enables the storage support ranging from Amazon Elastic Block Store (EBS), GCE persistent disk, OpenStack Cinder, vSphere, vSAN and more. The implementation of an orchestration system storage follows *NFS*, *ISCSI* and/or *cloudProvider_StorageSpecificSystem*. Indeed, NFS enables files to be shared among multiple client machines. In contrast, a block protocol, such as iSCSI supports a single client for each volume on the block server, while in the case of cloud provider specific system, the data sharing spans multiple servers.

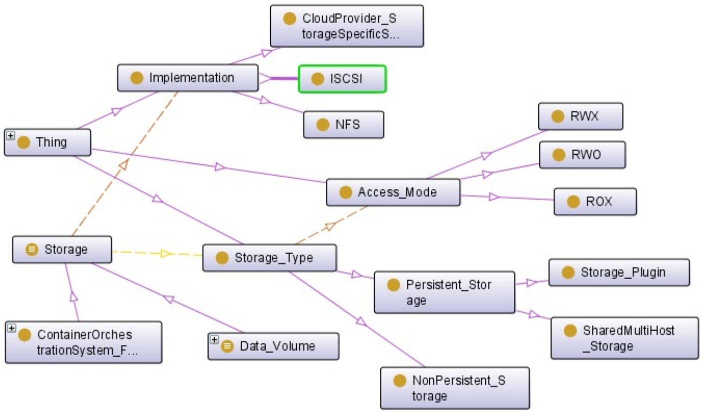


Figure 7 Storage Class

Networking Class

Since containers reside on either a local or a virtual host, they require an adequate networking approach to provide communication to other containers or components (other nodes). Without efficient communication, containers are almost useless. Container networking is similar to VM networking with two main differences [22]. The first deals with the fact that containers are short-lived compared to VMs and hence users are running many more containers than VMs, so the address space should be a lot bigger. The second is that VMs emulate the hardware and encompass virtual network interface cards (NIC) used to connect to the physical NIC, in contrast with containers that are just as processes sharing the same host kernel. Therefore, containers can be connected to either the same network interface and namespace as the host, or to an internal virtual network interface and their own network namespace and then connected to the external world in various ways.
Based on [13], the Network class has a set of object properties, such as the *Networking_Model*, *NetworkingSecurity_Mode* and *Networking_Implementation* (see Figure 8).

- Networking_Model: it describes the network model respected by a container orchestration system. According to the two container network standards, there are two main models used to configure interfaces in Linux containers: the Container Networking Model (CNM) and the Container Network Interface (CNI). The CNM is a standard proposed by Docker, which enables all the containers on the same network to

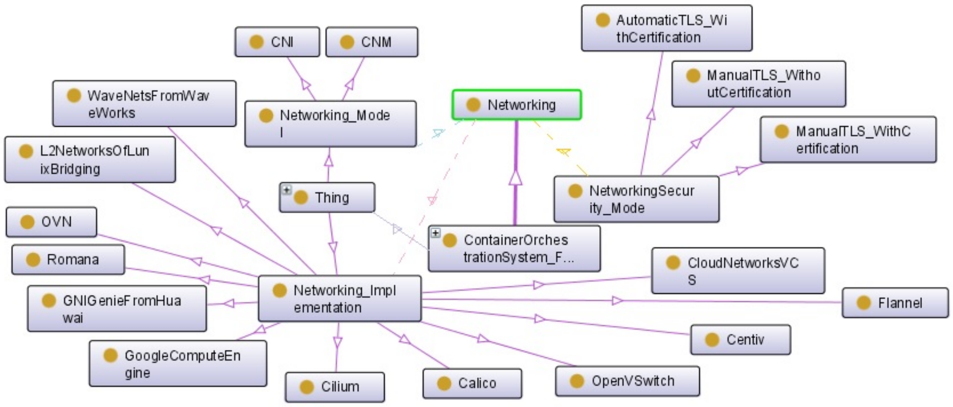


Figure 8 Networking Class

freely communicate with other. On the other hand, the CNI is a container networking standard proposed by CoreOS, which provides a simple set of interfaces for adding and removing a container from a network. Both standards are driver-based or plugin-based model, in order to create and manage network stacks for containers.

- NetworkSecurity_Mode: it may be automatic or manual. It depicts the manner of securing the connections between nodes. Usually, TLS authentication is used by container orchestration systems. While the connections are automatically secured through TLS authentication with certificates with Docker Swarm, it should be manually configured with Kubernetes.

- Networking_Implementation: there are a large number of network implementation technologies used by container orchestration systems [23], among which we cite: Contrail, Flannel, Contiv and Cilium.

Configuration Class

The configuration differs from an orchestration system to another. In fact, it may be considered as a serious advantage for a user to easily view, edit, and update the configuration file definitions of services and containers. This is introduced throughout the user-guide, getting-started documentation, and/or examples and via a *Dashboard*; either by using a *Dashboard_CLI* or by accessing via a *Dashboard_UI* (see Figure 9). Obviously, the configuration specifies a stable API version; which may be *Automatic* or *Manual* by writing the necessary configuration files. In both configuration modes, the user should follow a set of installation instructions which depends mainly on the supported OS and/or cloud provider platforms. Moreover, an orchestration system configuration needs launchable *DiscoveryService_Component*, *Networking_Component*, and *ContainerRuntime_Component*.

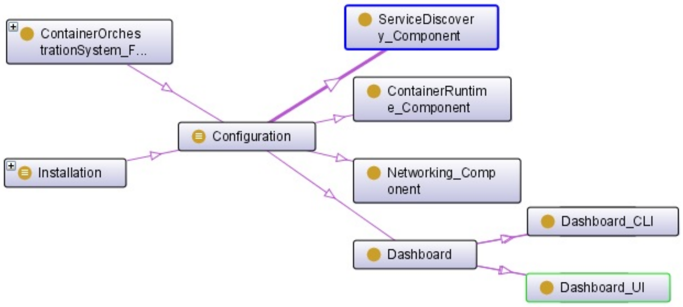

ContainerOrchestrationSystem_F...
ServiceDiscovery_Component
ContainerRuntime_Component
Configuration
Installation
Networking_Component
Dashboard_CLI
Dashboard
Dashboard_UI

Figure 9 Configuration Class

4 Proposed Framework and Ontology Population

4.1 Framework Overview

As illustrated by the proposed framework shown in Figure 10, different Cloud Service Providers (CSPs) running different Container-as-a-Service (CaaS) would incorporate our *Container Description Ontology Service* in charge of dealing with the management of the CDO ontology. The *CDO Service* features different modules and interfaces. The main module is the semantic rule engine that holds the Knowledge Base of the ontology, that is, the terminological part (TBox) explained in Sect. 3, the assertion part (ABox) with the instantiated individuals of the TBox (see Sect. 4), as well as the SWRL rules. In addition, our *CDO Service* is endowed with different parsers that allow instantiating the ontology, as explained in the following subsection.

The *CDO Service* interfaces with the *Container Orchestration System* to maintain up-to-date the instantiated ontology according to the status of the users' containers. In addition, the *CDO service* is endowed with a *Container Migration Service* that can assist the user on porting their containers across different CaaS. It allows contacting other peer modules deployed in other CaaS to share the instantiated CDO and the container images in order to port a container from one cloud provider to another. Additionally, final users that run their own containers orchestration system in their premises, can install our proposed *CDO App*. It is a similar application as the *CDO service* running on the Cloud. In addition, the *CDO App* also features a *Container selector Helper* to aid users on discovering the container orchestration system instantiated in the ontology. This particular module will be shown in Sect. 5.

4.2 Ontology Population

Our proposed framework supports three main ways to populate the ontology:

1. Instantiation from Docker Hub: in this case, the *Docker hub ontology parser* acts as a crawler that collects the set of Docker URLs and Docker file instructions from the official Docker registry hub. This module can parse the hub and instantiate different classes, such as the *Docker Image* class, thereby adding the ABox statements to the knowledge base. The instantiated CDO ontology can be shared among other users for interoperability, or can be just used to launch a new container. Thus, the module can then

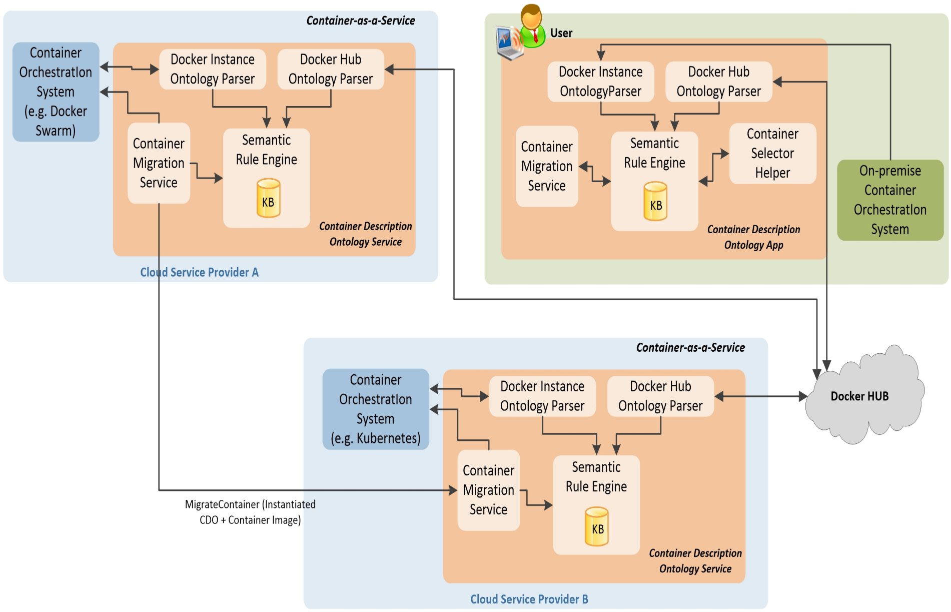

Container-as-a-Service
Container Orchestration System (e.g. Docker Swarm)
Docker Instance Ontology Parser
Docker Hub Ontology Parser
Container Migration Service
Semantic Rule Engine
KB
Container Description Ontology Service
Cloud Service Provider A
User
Docker Instance OntologyParser
Docker Hub Ontology Parser
Container Migration Service
Semantic Rule Engine
KB
Container Selector Helper
Container Description Ontology App
On-premise Container Orchestration System
Container-as-a-Service
Container Orchestration System (e.g. Kubernetes)
Docker Instance Ontology Parser
Docker Hub Ontology Parser
Docker HUB
Container Migration Service
Semantic Rule Engine
KB
Container Description Ontology Service
MigrateContainer (Instantiated CDO + Container Image)
Cloud Service Provider B

Figure 10 Overview of the proposed Framework

communicate using the CDO ontology with the on-premise container orchestration system to launch the container.

2. Instantiation from Docker tool: both final users and CSPs can make use of the *Docker instance ontology parser* component of the framework to populate CDO ontology with the actual containers running in the user's premises or in the Cloud (*i.e.*, in the CaaS). This module relies on the *container orchestration system* to obtain the ontology individuals and properties. The module parses the JSON generated as outcome of the command *Docker inspect $instance*, which allows obtaining the configuration of the particular running Docker instance. Among other classes and properties, it allows instantiating the *HostConfig* and *NetworkSettings* classes of the ontology. Then, the instantiated ontology can be shared along with the image in order to migrate the Docker instances among different host platforms achieving interoperability.

3. Manual instantiation: advanced users and administrators can also instantiate the ontology based on their own expertise. To this aim, they can rely on tools, such as Protégé, to create the individuals and properties of the ontology.

For instance, as orchestration systems, CDO is instantiated with *Docker Swarm*, *Amazon EC2 Container Service* (ECS), *Azure Container Service* (ACS), *Google Container Engine*, *Mesosphere Marathon* and *Nomad*.

Once CDO is populated, one can apply the inference rules to generate knowledge to help final users during their discovery for an appropriate container orchestration system.

4.3 Capabilities-based Inference Generation

By reusing a common vocabulary, semantic descriptions of the container orchestration systems can be shared and understood by the cloud user. Currently, container orchestration system discovery requires a significant amount of expertise. However, for a wide acceptance of the CDO, such task should become easier and more intuitive. So, it should introduce the inference rules capable of reasoning with service descriptions used to ease the discovery task. To define such rules, the majority of inference engines support Semantic Web Rule Language (SWRL). In fact, through a set of inference rules, we try to quantify the orchestration system capabilities in order to facilitate their discovery while satisfying the user's requirements. In other words, for an adequate container orchestration system discovery, we propose to introduce inferences based on functional and non-functional orchestration system capabilities.

4.3.1 Storage Capability-based Inference Generation

The container orchestration systems are classified according to their offered storage type into two levels of inductive inferences: LongTerm_StorageCapability and ShortTerm_StorageCapability. Indeed, an orchestration system is classified as LongTerm_StorageCapability if it offers the persistent data storage type (see Rule 1). Otherwise, an orchestration system has a ShortTerm_StorageCapability (i.e., it offers only the non-persistent storage type).

Rule 1 LongTerm_StorageCapability Rule

Container_OrchestrationSytsem(?p) ∧ ContainerOrchestrationSystem_Capability(?p, ?c) ∧ ContainerOrchestrationSytsem_FunctionalCapability(?c) ∧ Storage (?c, ?v) ∧ Storage_Type(?s) ∧ sqwrl:makeSet(?s, ?v) ∧ sqwrl:groupBy(?s, ?p) ∧ sqwrl:size(?l, ?s) ∧ swrlb:equal(?l, 1) ∧ sqwrl:element("Persistent_Storage", ?s) ∧ Persistent_Storage (?v) ⟶ LongTerm_StorageCapability(?p)

4.3.2 Configuration Capability-based Inference Generation

We introduce two inference reasoning rules (Hard_ToInstall and Easy_ToInstall) serving to assist users in their choices. An orchestration system is considered as Hard_ToInstall, if the configuration should be done manually to support the multi-OS and multi-cloud providers and if it covers the container runtime service discovery and networking components. Otherwise, the orchestration system is considered as Easy_ToInstall. Rule 2 defines the Hard_ToInstall inference rule.

Rule 2 Hard_ToInstall Rule

Container_OrchestrationSytsem(?p) ∧ ContainerOrchestrationSystem_Capability(?p, ?c) ∧ ContainerOrchestrationSytsem_FunctionalCapability(?c) ∧ Configuration(?c, ?v) ∧ sqwrl:makeSet(?s, ?v) ∧ sqwrl:groupBy(?s, ?p) ∧ sqwrl:size(?l, ?s) ∧ swrlb:equal(?l, 4) ∧ sqwrl:element("ContainerRuntime_Component", ?s) ∧ sqwrl:element("ServiceDiscovery_Component”, ?s) ∧ sqwrl:element("Networking_Component”, ?s) ∧ Dashboard(?v) ∧ sqwrl:element("Dashboard_CLI", ?v) ∧ Configuration_Mode(?v) ∧ sqwrl:element("Manual", ?v) ∧ Installation_Instruction (?v) ∧ OS_Support (?s) ∧ sqwrl:element("Multi", ?s) ∧ sqwrl:element("Manual", ?v) ∧ Installation_Instruction (?v) ∧ Provider_Support (?s) ∧ sqwrl:element("Multi", ?s) ⟶ Hard_ToInstall(?p)

4.3.3 Networking Capability-based Inference Generation

Each container orchestration system exposes its networking security mode as proof that ensures the adoption of the best security strategies. Besides, each one owns a network design and implementation technology. The network design and implementation are crucial because a poorly designed environment can cause security and performance issues, so it is better to get it with diverse technologies. These technologies cover the WAN data communication, SDN, mobile network, and enterprise transmission and access network services. So, we classify orchestration systems into three classes: Low_NetworkingCapability, Medium_NetworkingCapability, and High_NetworkingCapability. An orchestration system is considered as (1) High_NetworkingCapability, if its network implementation technologies cover greater than two services and its security mode is automatic Transport Layer Security (TLS) with certifications (see Rule 3). It is worth mentioning that: (i) TLS is a cryptographic protocol that provides authentication and data encryption between servers, machines, and applications operating over a network. The TLS protocol standards include certificates, keys, symmetric keys, and passwords, (ii) a TLS automatic key management is better than a manual one due to that a manual key management is sensitive to scale. Obviously, if we have a big, large or heterogeneous infrastructure, there will be a complex and error-prone manual key management. (2) Medium_NetworkingCapability, if its network implementation technologies cover only two services and its security mode is manual TLS with certifications. And (3) Low_NetworkingCapability, if the orchestration system enables end-users to use a network with implementation technologies involving less than two services and with a manual TLS without certification as security mode.

Rule 3 High_NetworkingCapability Rule

Container_OrchestrationSystem(?p) ∧ ContainerOrchestrationSystem_Capability(?p, ?c) ∧ ContainerOrchestrationSytem_FunctionalCapability(?c) ∧ Networking(?c, ?v) ∧ NetworkingSecurity_Mode(?v, ?s) ∧ sqwrl:element("AutomaticTLS_WithCertification”, ?s) ∧ Networking_Implementation(?c, ?s) ∧ sqwrl:makeSet(?s, ?v) ∧ sqwrl:groupBy(?s, ?p) ∧ sqwrl:size(?l, ?s) ∧ swrlb:greaterThan(?l, 2) ⟶ High_NetworkingCapability(?p)

5 CDO Evaluation

This section demonstrates the CDO effectiveness and the performance evaluation of the proposed framework. Performance measures the checking time of the ontology consistency, the time spent on the SPARQL discovery query and the reasoning time expended to infer knowledge.

5.1 CDO Effectiveness

Firstly, according to [8], we evaluate the ontology structure (concepts, axioms and taxonomy) while verifying its consistency. After that, to ensure a large and a best CDO usage, we propose to apply it on a real case study. For this purpose, we consider an online retail application used by an offshore petroleum logistics company (INA-Group [9]) along with its subsidiaries. This application is implemented as a fully independent set, fine-grained, and self-contained (micro) services (see Figure 11). These services are implemented on top of different technology stacks where each of which addresses a specific business scope. For instance, the purchase requisition service follows the process by which the company's employees may request to purchase goods or services. The company's IT experts

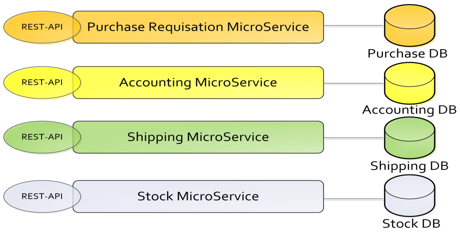


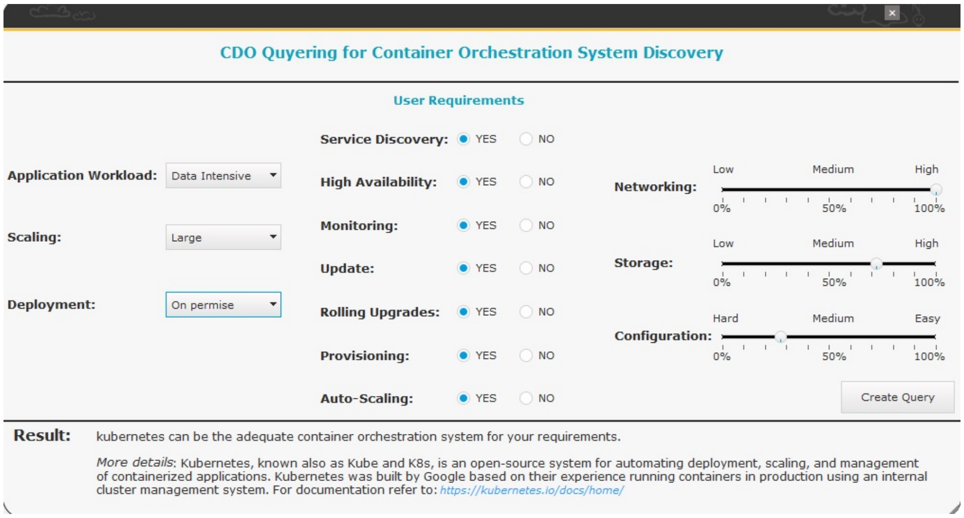
Figure 11 The Online Retail Application

are willing to run the micro-services application on containers. Using the *CDO App*, they request to deploy the micro-services on-permise and/or on an orchestration system from the available ones, such as Docker Swarm, Kubernetes, Mesos with Marathon, etc. The IT experts formulate a set of requirements using the *Container Selector Helper* interface (see Figure 12). Besides, the monitoring, the high-availability and the update of the running instances, expect an orchestration system with a large scale capability, a fault tolerance cluster and a high network capability. However, they do not pay a lot of attention to the system configuration.

Based on the inference rules already defined in Sect. 5, the CDO querying result is presented (See bottom part of Figure 12). The Kubernetes system is recognized to be an adequate orchestration system. Such result can be argued by the fact that Kubernetes is suited for large scale deployment. Moreover, it provides a high term storage (i.e., a long term storage), since it offers a persistent data storage services. Besides, as the IT experts do not pay attention to the system configuration, the kubernetes is the best alternative, otherwise Docker swarm can be selected, as it requires less configuration efforts. Compared to Swarm, Kubernetes offers the automatic scaling and can deliver updates to the running instances, which are missing in the Swarm. On the contrary, Mesos offers these capabilities and adds cluster health check which enhances the cluster fault tolerance, however, it fails to propose a high or a medium network capability. In the second querying scenario shown on Figure 13, the IT experts choose to deploy the application on the cloud infrastructure while preserving the same requested capabilities. Reasoning over the CDO reveals that Kubernetes over the Google cloud provider or Microsoft Azure can be adequate deployment choices. These results seem to convince the IT experts, which proves that the CDO can properly meet the user's requirements.

5.2 Performance Evaluation

An important part of our framework, which widely affects the performance, is the reasoner in charge of making the container orchestration discovery. Indeed, the discovery is taken based on ontology instances present on the KB. A way of measuring the performance of our Container Selector Helper is making different complexity executions based on different sets of workloads in the KB (individuals and statements). This serves to make the test



CDO Quyering for Container Orchestration System Discovery

User Requirements

Application Workload: Data Intensive

Scaling: Large

Deployment: On permise

Service Discovery: ● YES ○ NO

High Availability: ● YES ○ NO

Monitoring: ● YES ○ NO

Update: ● YES ○ NO

Rolling Upgrades: ● YES ○ NO

Provisioning: ● YES ○ NO

Auto-Scaling: ● YES ○ NO

Networking:

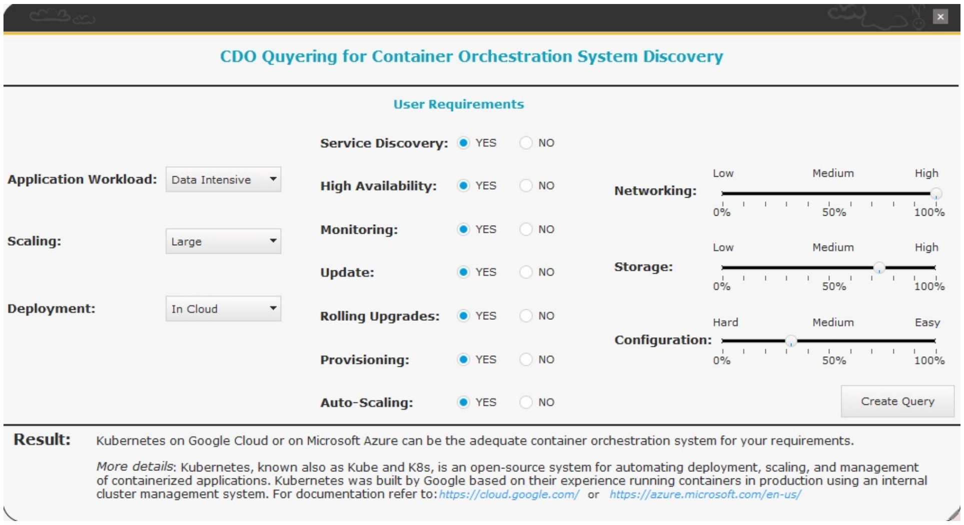
Storage:

Configuration:

Result: kubernetes can be the adequate container orchestration system for your requirements.

More details: Kubernetes, known also as Kube and K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. Kubernetes was built by Google based on their experience running containers in production using an internal cluster management system. For documentation refer to: *https://kubernetes.io/docs/home/*

Figure 12 Container Selector Helper: First Querying Scenario



CDO Quyering for Container Orchestration System Discovery

User Requirements

Application Workload: Data Intensive

Scaling: Large

Deployment: In Cloud

Service Discovery: ● YES ○ NO

High Availability: ● YES ○ NO

Monitoring: ● YES ○ NO

Update: ● YES ○ NO

Rolling Upgrades: ● YES ○ NO

Provisioning: ● YES ○ NO

Auto-Scaling: ● YES ○ NO

Networking:

Storage:

Configuration:

Result: Kubernetes on Google Cloud or on Microsoft Azure can be the adequate container orchestration system for your requirements.

More details: Kubernetes, known also as Kube and K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. Kubernetes was built by Google based on their experience running containers in production using an internal cluster management system. For documentation refer to: *https://cloud.google.com/* or *https://azure.microsoft.com/en-us/*

Figure 13 Container Selector Helper: Second Querying Scenario

becoming more and more complex. Thus, the complexity of executions is incremented by increasing the number of individuals present in the ABox component of the ontology which, in turn, increases the number of statements held in the knowledge base. It should be noticed that the number of axioms or statements that are held in the ABox differs from that of individuals within the ontology. Since more than one axiom is usually necessary to represent one individual. The addition of new instances leads the KB to reach the next population state, whereby it will necessary to perform the KB consistency checking again.

As shown in Table 1, the testbed makes use of 10 incremental populations and the query proposed in Sect. 5.1. Each population is composed of individuals representing instances of different OWL classes defined in our ontology. These populations will be used to quantify the time that our Container Selector Helper takes to check the KB consistency, infer knowledge in order to derive information explicitly specified in the Sect. 4.3, and carry out discovery query. It is worth mentioning that *(i)* we have checked the ontology consistency and derived the capability-based inferences by using the FaCT++ reasoner [24] and *(ii)* the selection time is the one taken by the Container Selector Helper to interpret the user's query and present the target outputs (*i.e.*, the adequate container orchestration system) to the user. The proposed framework has also been tested based on CPU speed. In fact, the experiments have been conducted on a set of workstations with different hardware configurations (6 GB RAM with 2.4 GHz, 2.5 GHz, and 2.7 GHz). For the results depicted in Figure 14, it can be concluded that the consistency checking and the selection time depend on three major factors: the testbed size, the CPU speed, and the reasoning complexity (i.e, the rules to infer knowledge, such illustrated in Figure 15). Overall, the proposed framework provides a reasonable checking consistency and selection time. For instance, in the case of CPU with 2.4 GHz and for the biggest population in the testbed (1050 individuals and 12200 statements), the consistency checking time takes 5.5 seconds; the selection time takes 5.53 seconds, which is usually considered an acceptable time. For this querying, the three inference rules are activated. The selection time can be reduced once one or two rules are used to infer over the CDO (see reasoning time per rule in Figure 15 for the three CPU speed).

Table 1 Performance Evaluation

Population	Individual	Statement
1	270	1200
2	390	3600
3	450	5200
4	630	6800
5	710	8600
6	770	9800
7	950	11000
8	990	11400
9	1030	11800
10	1050	12200

6 Conclusion and Future Work

Inspired by the existing container standards as well as by referring to research papers, we proposed in this paper a conceptualization of domain ontology for container description called CDO. CDO covers the functional and non-functional capabilities for containers, Dockers and container orchestration systems. It is instantiated and interrogated using a dedicated framework. This framework can be used by both the final users and CaaS providers. In fact, it helps users to make the discovery of an adequate container orchestration system easier through a set of capability-based inference rules and enhances interoperability between CSPs by providing migration service for deploying applications among different host platforms. The CDO effectiveness is demonstrated relying on a real case study where the company's IT experts are willing to deploy a micro-service application on a containerized environment under a set of functional and non-functional requirements. In addition, the

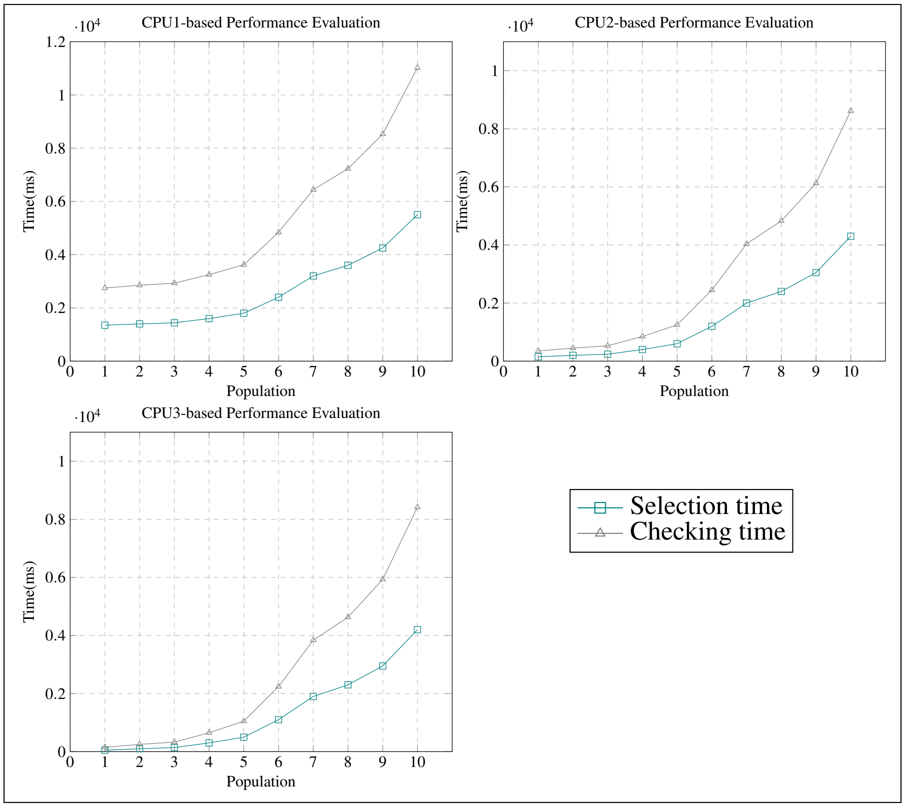


Figure 14 Consistency Checking and Selection Time Evaluation

performance evaluation over the system shows reasonable times, thereby demonstrating the feasibility of the proposal.

As a future endeavor, we plan to further investigate the linking of CDO with other ontologies to achieve the interoperability and usability. In addition, we plan to take advantage of the Docker image information to recommend adequate deployment. Moreover, our aim is to extend the CDO evaluation while treating the interoperability between CSPs.

References

[1] A. B. Ayed, J. Subercaze, F. Laforest, T. Chaari, W. Louati, and A. H. Kacem. Docker2rdf: Lifting the docker registry hub into rdf. In *2017 IEEE World Congress on Services (SERVICES)*, pages 36–39, June 2017.

[2] A. Belfadel, E. Amdouni, J. Laval, C. Cherifi, and N. Moalla. Ontology-based software capability container for restful apis. In *International Conference on Intelligent Systems (IS)*, pages 466–473, 2018.

[3] E. Ben-Abdallah, K. Boukadi, J. Lloret, and M. Hammami. Crosa: Context-aware cloud service ranking approach using online reviews based on sentiment analysis. *Concurrency and Computation: Practice and Experience*, page e5358, 2019.

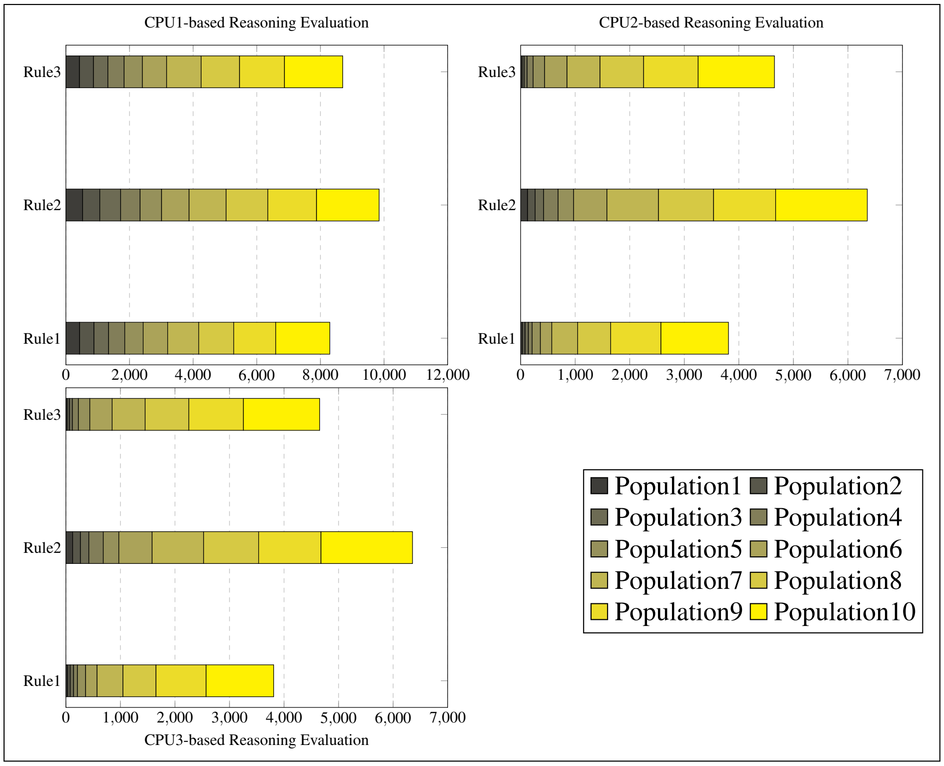


Figure 15 Reasoning Time Evaluation

[4] J. B. Bernabe, G. M. Perez, and A. F. S. Gomez. Intercloud trust and security decision support system: an ontology-based approach. *Journal of Grid Computing*, 13(3):425–456, Sep 2015.

[5] Cloud Foundry Foundation C. O'Connell. Latest cloud foundry survey reveals container production deployment remains sluggish, contrary to industry hype. `https://www.cloudfoundry.org/container-report-2017-press-release/`, September 2017. [Online; accessed March 2020].

[6] Iain Emsley and David De Roure. A framework for the preservation of a docker container. In *Proceedings of the 12th International Digital Curation Conference (IDCC17)*, 2017.

[7] A. Flahive, D. Taniar, and W. Rahayu. Ontology as a service (oaas): a case for sub-ontology merging on the cloud. *The Journal of Supercomputing*, 65(1):185–216, 2013.

[8] A. Gomez-Perez, M. Fernandez, and A. deVicente. Towards a method to conceptualize domain ontologies. In *Proceedings of the Workshop on Ontological Engineering*, 1996.

[9] ABID GROUP. Abid group. `http://groupe-abid.com.tn/`, 2015.

[10] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum. Comput. Stud.*, 43(5-6):907–928, December 1995.

[11] D. Huo, J. Nabrzyski, and C. Vardeman. Smart container: an ontology towards conceptualizing docker. In *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, PA, USA, October 11, 2015.*, 2015.

[12] J. Höller, V. Tsiatsis, C. Mulligan, S. Karnouskos, S. Avesand, and D. Boyle. *From Machine-To-Machine to the Internet of Things: Introduction to a New Age of Intelligence*. Academic Press, Inc., 1st edition, 2014.

[13] Kubernetes. Cluster networking. The Linux Foundation, 2017.

[14] D. Liu and L. Zhao. The research and implementation of cloud computing platform based on docker. In *11th International Computer Conference on Wavelet Actiev Media Technology and Information Processing*, pages 475–478, Dec 2014.

[15] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi. Cloud computing - the business perspective. *Decis. Support Syst.*, 51(1):176–189, April 2011.

[16] F. Moscato, R. Aversa, B. Di Martino, T. Fortis, and V. Munteanu. An analysis of mosaic ontology for cloud resources annotation. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 973–980, 2011.

[17] N. F. Noy and D. L. McGuinness. Ontology development 101: A guide to creating your first ontology. Technical report, March 2001.

[18] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya. Containercloudsim: An environment for modeling and simulation of containers in cloud data centers. *Software: Practice and Experience*, 47(4):505–521, 2017.

[19] Protégé. Protégé. `https://protege.stanford.edu/`, 2010. [Online; accessed March 2020].

[20] M. Rekik, K. Boukadi, W. Gaaloul, and H. Ben-Abdallah. Anti-pattern specification and correction recommendations for semantic cloud services. In *Proceedings of Hawaii International Conference on System Sciences*, HICSS-50, 2016.

[21] M. A. Rodriguez and R. Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 49(5):698–719, 2019.

[22] R. J. Sanchez. *Dynamic Deployment of Specialized ESB instances in the Cloud*. PhD thesis, Institute of Architecture of Application Systems, University of Stuttgart, 2014.

[23] Kublr Team. Choosing the right containerization and cluster management tool. Kublr, 2016.

[24] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *Proceedings of the International Joint Confererence on Automated Reasoning*, pages 292–297, 2006.

[25] W3C. Docker ontology | vocabularies for big data analysis community group. `https://www.w3.org/community/bigdata-tools/2017/10/30/docker-ontology/`, 2017. [Online; accessed March 2020].