Cloud Service Description Ontology: Construction, Evaluation and Interrogation

Khouloud Boukadi, Molka Rekik, Hanène Ben-Abdallah, Walid Gaaloul

Abstract

Cloud federation systems have recently emerged as a scalable delivery model that interconnects services from two or more cloud providers for load balancing and accommodating spikes in demand. One challenge in this delivery model is the complexity of service selection due to the heterogeneity of cloud service descriptions among cloud federations. To overcome this complexity, it is crucial to uniform cloud service descriptions. Towards this end, we propose a Cloud Service description Ontology (CSO) that will be modeled based on concepts from cloud standards. The proposed CSO covers functional and non-functional capabilities of infrastructure, platform and software services’ providers. To populate CSO, we defined a set of semantic mapping rules to collect instances from cloud providers’ web pages. In addition, to ensure the high quality of CSO, we propose an evaluation approach that detects and corrects consistency, redundancy and incompleteness errors. Furthermore, to show the correctness and inference ability of CSO, we present an experimental evaluation that measures the precision and recall ratios of querying CSO.

Keywords: Cloud Service Description, Ontology, Evaluation, Consistency, Patterns and Anti-patterns, Errors Detection, Correction Recommendation

1. Introduction

The innovation shift of Cloud computing resides in the way it supplies on-demand computing resources with minimal management effort or service provider interaction. Cloud computing offers its benefits commonly through
three types of services, namely: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS), where the providers offer access and manage, respectively, virtual resources, platforms and applications. Recently, cloud federation systems have emerged as a scalable delivery model that interconnects the cloud computing environments of two or more service providers to balance loads and accommodate spikes in demand.

Because cloud federation systems are created by merging services originally offered independently by cloud providers, they often have similar services described differently (names, quality of services, attributes, etc.). For example, Amazon refers to compute services as "EC2 instance", while Microsoft Azure refers to them as "Virtual machine". Such heterogeneous description complicates service comparison and selection, and may create interoperability problems among multiple providers (especially when migrating from one provider to another). It is worsened by the absence of a standard language for describing cloud services. As a result, the only means that users have to compare and select cloud services is to manually examine the providers’ documentation. This non-trivial means is made more difficult with unannounced updates that providers make as well as their reluctance to give detailed service descriptions. In fact, the main source of service information is the providers’ web sites which are not designed to compare and reason about the relations among the different types of cloud services and their configurations. However, service description models and discovery mechanisms are essential to guide the user in the discovery and selection of the suitable services meeting their functional and non-functional requirements.

In summary, at the current state, there is no cloud service description ontology that covers the three cloud layers (IaaS, PaaS and SaaS) which is automatically and efficiently populated and evaluated. We propose to define, populate and evaluate ontology for cloud service description. The proposed ontology, called Cloud Service description Ontology (CSO), semantically and structurally represents cloud providers’ information. It specifies, in a standard way, the knowledge shared between the cloud providers within a cloud federation. As
such, the herein presented work makes the following contributions to cloud federation domain:

1. A cloud service description ontology (CSO) the definition of which is based on cloud computing standards. The proposed ontology treats the IaaS, PaaS and SaaS layers and accounts for a set of functional and non-functional properties.

2. A set of capabilities-based inference rules capable of reasoning with service descriptions are used to ease the cloud service discovery task.

3. A two-phased evaluation approach for the proposed ontology based on OWL DL reasoner and on a set of proposed anti-patterns to detect inconsistency, incompleteness and redundancy as well as domain-knowledge errors and, consequently, improve the quality (i.e., effectiveness and accuracy) of the proposed ontology.

4. An interface allowing cloud federation users to interrogate CSO in order to select the appropriate service providers that fulfill their functional and non-functional requirements.

The rest of the paper is organized as follows. Section 2 details some related work. Section 3 describes our proposed cloud service description ontology. In Section 4, a set of capability-based Inference rules is presented. We evaluate the proposed ontology in Section 5. In Section 6, we interrogate the CSO with users’ requirements through the CSOIE interface. Finally, we summarize the status of the presented work and outline its extensions.

2. Related work

Our literature review revealed that several proposals have emerged to solve the heterogeneity problem in cloud service descriptions, mainly in the context of services selection for running applications/business processes into multi-cloud systems.

The RESERVOIR project [1], proposed by IBM, federates IaaS providers interacting with each other to create a pool of resources in a transparent way.
To solve the problem of heterogeneity of cloud service descriptions, the authors used the OCCI standard.

The mOSAIC project [2] is another interesting initiative that aims at solving the problem of heterogeneous service description in multi-cloud environments. Based on the OCCI standard, the authors proposed an ontology handling the functional cloud IaaS and PaaS service properties. In fact, a semantic engine along with a discovery service dedicated to customers are defined to help them to deploy their applications.

The CONTRAIL project [3] specifies the resources within a cloud federation using an OWL ontology. The proposed ontology covers the functional and non-functional properties of the IaaS services.

In [4], the authors proposed a Cloud Computing Ontology called CoCoOn to discover the suitable services for the user’s needs. The CoCoOn ontology defines a set of functional and non-functional properties to describe the IaaS services. Then, the authors implemented a recommendation system based on the relational model of the CoCoOn ontology. In order to discover the adequate services, the recommender system applies SQL queries to interrogate the CoCoOn ontology.

The authors, in [5], offer a brokerage system based on the semantic description language for web services (SWRL [6]) to select the adequate cloud services. The brokerage system identifies the correspondence between the required service and the published ones as: an exactly match, a "Subsume" match, a "Reverse subsume" match, a partial match and a wrong match. The brokerage system then chooses the suitable service that respects the user’s budget.

In [7], the authors proposed an architectural framework for federated cloud (FCFA) to ensure the interoperability and facilitate the service selection in federated infrastructure. The FCFA framework is based on a semantic aspect supporting three OWL ontologies. The first ontology is proposed to link both physical and virtual infrastructures. The cloud ontology covers concepts and properties representing the provider, the user and the contract signed between them. The third ontology, the federation ontology, inherits from the two previ-
ous ontologies and highlights a set of concepts, such as those which represent the complete list of providers within the federation and the link between them.

In [8], the authors defined a Cloud Query Manager (CQM) to overcome the heterogeneity problem between different cloud providers. The CQM architecture includes five components: (i) RDF models, which describe services of the three cloud platforms, namely: OpenStack, Eucalyptus and OpenNebula, (ii) SPARQL server, which defines the SPARQL queries, (iii) Query processor, which is responsible for updating the RDF models with data collected from cloud platforms, (iv) Connection manager, which manages HTTP connections to target cloud platforms and drivers for cloud platforms in order to invoke the SPARQL queries and (v) Drivers, which implement the platform interfaces allowing the APIs calls.

In [9], the authors proposed a framework based on an ontology and an integrated multi-agent system for a common cloud service description. Indeed, the provider agent monitors his services. The user agent offers a GUI that helps the users select adequate cloud services. The consumer agent interacts with the discovery agent through ACL (Agent Communication Language) messages. The discovery agent is responsible for the discovery of the services based on SPARQL while interrogating the ontology.

In [10], the authors proposed a service discovery engine including four main layers: (i) Cloud service collection layer: it automatically collects cloud services by accessing Web portals, such as Google, Yahoo and Baidu, (ii) Cloud service Data extraction layer: it filters the contents of cloud providers’ home pages, automatically extract service information and send them to the cloud services categorization layer, (iii) Cloud services categorization layer: it computes the similarity measures between the extracted and the NIST standard concepts to categorize the services and (iv) Cloud service ontology layer that manages the cloud service ontology.

In [11], the authors presented two cloud ontologies CO1 and CO2 designed to semantically define cloud services. CO1 contains only cloud concepts while CO2 includes a set of individuals of these concepts, properties and relationships.
Then, the authors proposed two recommendation approaches (R1 and R2) while calculating the similarity between the providers’ services and the user’s requirements. By applying the R1 approach, the broker chooses the provider that has the highest similar score. By applying the R2 approach, the broker chooses the cloud provider with the highest average similarity score while considering the concepts, the object and the data type properties.

In [12], the authors proposed a set of ontologies to semantically represent cloud resources: (i) standard-specific ontologies, namely, sTOSCA, sOCCI, and sCIMI representing cloud resources described in TOSCA, OCCI, and CIMI, respectively, and (ii) an upper ontology called linked cloud resource (Linked-CR) representing a high-level abstraction of the three standard-specific ontologies. The authors also defined a set of semantic inference rules to automatically translate the cloud resources instantiated by a standard-specific ontology to another. Thus, organizations can query the Linked-CR ontology and customize cloud resources regardless of their representation used by cloud providers.

Our analysis of the above mentioned works highlights the following points:

1. The majority of works consider only the service functional properties and neglect the non-functional properties known as Quality of Service (QoS). For the non-functional properties, ontology can generate some knowledge assessing cloud providers and their services and therefore make the service selection task easier. In [4], the authors studied only sustainability as a QoS property. However, other QoS properties, such as security and availability, are widely required by the cloud users.

2. The majority of the proposed approaches ignore cloud standards when defining the ontology concepts. In [1], [2] and [3], the authors rely on a single standard to define their ontologies. However, relying on several standards and standardization attempts should ensure a wider coverage of the cloud domain and capture the various standards used by cloud providers to describe their services.

3. The majority of current proposals focus only on the IaaS layer, but they
neglect the PaaS and SaaS layers.

4. Expect the ontology presented in [7], all the proposals do not deal with the cloud federation properties.

5. Some approaches treat only the conceptualization phase of their proposals by identifying concepts, relations and axioms, also ignore individuals. Except [10] and [12], the presented works manually populate their proposals [2] [8]. The manual population process is generally considered as an expensive process in terms of execution time and effort.

6. All existing proposals were not evaluated, which has a negative impact on their reusability.

Based on these observations, we propose a cloud service description ontology that relies on several cloud standards; covers IaaS, PaaS and SaaS; accounts for functional and non-functional federation properties; and is automatically populated and systematically evaluated.

3. Standard-based Cloud Service Description Ontology

Referring to Gruber, an ontology is a specification of a conceptualization used to solve the problem of semantic heterogeneity and enable semantic interoperability between different parties [13]. An ontology contains five principal components: concepts or classes, properties, relationships, axioms and instances or individuals.

For the CSO construction, we adopted: (i) the design principles presented in [13]. These principles include clarity, minimal encoding bias, extendability, coherence and minimal ontological commitments; (ii) the standardization name suggested in [14] which eases the understanding of the ontology; and (iii) commonly used cloud standards, such as NIST[1] OCCI[2] CIMI[3]

---

1http://www.nist.gov/itl/cloud
2http://occi-wg.org
3https://www.dmtf.org/standards/cloud
For the CSO construction, we used the ”protégé” tool which is the most widely used, freely available, platform independent for developing and managing terminologies, ontologies and knowledge bases.

As shown in Figure 1 among the main concepts in CSO, we cite: Actor, Service, Essential_Characteristic and CloudFederation.

![Figure 1: CSO Top Level Concepts](image)

**CloudFederation Class**

In a cloud federation, the services are provisioned by a group of cloud providers that collaborate together to share resources. The property *interconnected* represents the different members of a cloud federation (See Figure 2). Each federation has an architecture *hasFederationArchitecture*, a type *hasFederationType* and a network *hasNetwork*, as shown in Table 2. The federation architecture can be installed with broker, FederationArchitecture_Centralized, where there is a central broker that performs and facilitates the resource allocation or without broker, FederationArchitecture_Decentralized. The federation type can be vertical, which spans multiple levels or horizontal, which takes place on one cloud level.

**Actor Class**

The Actor class presents the different actors that participate in the cloud environment. In Figure 3 we distinguish between: the user, the provider and the broker. The Provider class includes three sub-classes, Provider_IaaS, Provider_PaaS and Provider_SaaS. Each cloud provider offers three service types which are, re-
respectively: (i) IaaS including the Storage and the Virtual Machine, (ii) PaaS covering the Platform and (iii) SaaS encompassing the Software or application.

**ProviderCapability Class**

The competitive advantage that a cloud provider could have thanks to his functional and non-functional (QoS) capabilities. This class has two sub-classes, namely FunctionalCapability and NonFunctionalCapability to help users select the suitable provider.
The Functional_Capability concept includes the following sub-concepts:

- **Support**: it refers to a plethora of services by which cloud providers give assistance to cloud users, such as social networks, forums, emails, phone numbers, etc.

- **API Support**: it’s required by the users to access the cloud functionality as well as to ensure the interoperability with different providers.

- **BackUp**: it refers to the copying and archiving of data so it may be used to restore the original data, after a resource failure event.

- **Elasticity**: represents the degree to which an IaaS service can be adapted to workload changes by provisioning resources in an autonomic manner.

- **Load Balancing**: the ability to distribute the workload between different resources.

- **Migration**: it is the ability of migrating activities/applications from a cloud provider to another.

- **Monitoring**: it displays information about the real time use of resource (CPU, memory, disk, and network) to detect overloaded or failed resources.

- **Compute Capability**: it is a computing ability that characterizes a virtual machine (VM), such as Operating System (OS), CPU core number, memory size, use case, storage volume and the essential characteristics.

- **Storage Capability**: it characterizes a storage space, such as the number of Input/Output Operations Per Second (IOPS) and the minimum and the maximum sizes.

- **Network Capability**: it characterizes the network like data transfer size and latency.
• Platform Capability: it characterizes a platform, such as Web server, application server, API, IDE (Integrated Development Environment) and database.

• Software Capability: it characterizes an application, such as Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), Email, social networking, collaboration and content management.

On the other hand, the NonFunctional Capability concept specifies:

• Availability: it is the probability that a cloud provider is functional.

• Security: it is the feature that addresses the minimum risk for the users. To quantify this concept, it is necessary to specialize it into a set of sub-concepts. In this field, several initiatives have been proposed in the literature [15] [16]. Following these initiatives as well as the NIST security recommendations, we distinguish the following properties:

1. Certification: the International Organization for Standardization (ISO) defines a certification as a "written assurance by a third party that a product, service or system meets specific requirements" [17]. The relevant certifications, audit reports and laws in the cloud area are: CSA [18], SSAE 16 (SOC 1 [19], SOC 2 [20], SOC 3 [21]), PCI DSS [22] and ISO 27001 [23].

2. Data protection: each cloud provider must ensure the data traffic safety against all attack types. To achieve this goal, several data protection mechanisms have been proposed to establish a secure network connection between resources using firewall, third-party or VPN (Virtual Private Networking).

3. Data Management: through data management, we distinguish encryption, storage, recording, transfer and data destruction.

Axiom Definitions
Table 1 identifies the set of axioms which cover the specialization, the disjoint and the union relations between the important concepts within CSO.
Table 1: CSO’s important axioms

<table>
<thead>
<tr>
<th>Axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker ⊑ Actor; User ⊑ Actor; Provider ⊑ Actor;</td>
</tr>
<tr>
<td>Broker ⊓ Provider ⊓ User ⊑ ⊥;</td>
</tr>
<tr>
<td>Provider ≡ Provider_{IaaS} ⊓ Provider_{PaaS} ⊓ Provider_{SaaS};</td>
</tr>
<tr>
<td>IaaS ⊑ Service; PaaS ⊑ Service; SaaS ⊑ Service;</td>
</tr>
<tr>
<td>IaaS ⊓ PaaS ⊓ SaaS ⊑ ⊥;</td>
</tr>
<tr>
<td>Storage ⊑ IaaS; Virtual_Machine ⊑ IaaS;</td>
</tr>
<tr>
<td>Platform ⊑ PaaS;</td>
</tr>
<tr>
<td>Software ⊑ SaaS;</td>
</tr>
<tr>
<td>OnDemand_Service ⊑ Essential_Characteristic;</td>
</tr>
<tr>
<td>Reserved_Service ⊑ Essential_Characteristic;</td>
</tr>
<tr>
<td>OnDemand_Service ⊓ Reserved_Service ⊑ ⊥;</td>
</tr>
<tr>
<td>ComputeCapability.ContainsCpu ≡ cpuIsPartOf⁻;</td>
</tr>
<tr>
<td>ComputeCapability.ContainsMemory ≡ memoryIsPartOf⁻;</td>
</tr>
</tbody>
</table>

Each Virtual_Machine (VM) has specific characteristics depending on its type. The object property hasVMTtype links between the two concepts Virtual_Machine and VMTType, as shown in Table 2. The Provider_IaaS offers a wide range of virtual machine types, such as small, medium, large. Each VMTType has an essential characteristic which is either on demand service, which is a prime feature of most cloud offerings where the user pays for use, or on a reserved service where the user gets a discount after a service is used for a specific amount of time. Moreover, each VMTType should have a computing capability which includes two characteristics: CPU and memory which could not both exist without a computing capability. As shown in the two last lines in Table 1 the object properties cpuIsPartOf and memoryIsPartOf are the inverse of the object properties ComputeCapability.ContainsCpu and Compute-Capability.ContainsMemory respectively.

It is worth mentioning that Tables 1 and 2 present the useful axioms for the
<table>
<thead>
<tr>
<th>Object property</th>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasVMType</td>
<td>Virtual_Machine</td>
<td>VMType</td>
</tr>
<tr>
<td>hasEssentialCharacteristic</td>
<td>VMTYPE</td>
<td>Essential_Characteristic</td>
</tr>
<tr>
<td>ComputeCapability.ContainsCpu</td>
<td>Compute_Capability</td>
<td>CPU</td>
</tr>
<tr>
<td>cpuIsPartOf</td>
<td>CPU</td>
<td>Compute_Capability</td>
</tr>
<tr>
<td>ComputeCapability.ContainsMemory</td>
<td>Compute_Capability</td>
<td>Memory</td>
</tr>
<tr>
<td>memoryIsPartOf</td>
<td>Memory</td>
<td>Compute_Capability</td>
</tr>
<tr>
<td>hasFederationArchitecture</td>
<td>CloudFederation</td>
<td>FederationArchitecture</td>
</tr>
<tr>
<td>hasFederationType</td>
<td>CloudFederation</td>
<td>FederationType</td>
</tr>
<tr>
<td>hasNetwork</td>
<td>CloudFederation</td>
<td>Network</td>
</tr>
<tr>
<td>interconnected</td>
<td>CloudFederation</td>
<td>Provider</td>
</tr>
</tbody>
</table>

Ontology Populating

A detailed review of ontology population systems is presented in [24]. Existing approaches, in the literature, have proposed manual, semi-automatic or full-automatic populating processes. The manual ontology populating is often a very hard task especially in the case of a large ontology [25, 26]. For this reason, we propose a semi-automatic populating process implemented by an entity, named Semantic Cloud Broker (SCB). The SCB is responsible for achieving the automatic phases, collecting functional information from the providers’ web pages and the non-functional one from the well-known monitoring parties, such as CloudHarmony [27], CloudOrado [28] and CloudLook [29] and then instantiating the ontology’s concepts and properties based on the proposed mapping rules.

For space limitation, the CSO populating process is out of the scope of this paper. Readers can refer to 4 where we detail the CSO populating approach.

4https://drive.google.com/file/d/0B17XK9qGY-dQazuRtY19fMTFyMDA/view
Moreover, we deal with an interesting aspect of ontology evaluation, which is not well addressed in the literature. This aspect is related to the handling of redundancy, inconsistency, incompleteness, etc. For example, if an ontology is populated with an instance without checking if the real object or event represented by the instance already exists in ontology, then redundant instances will be inserted. Therefore, after populating an ontology, it’s necessary to evaluate it to ensure the consistency maintenance and the redundancy elimination [30].

4. Capabilities-based Inference Generation

According to the World Wide Web Consortium (W3C): "The inference means that automatic procedures can generate new relationships based on data and on some additional information in the form of a vocabulary, for example, a set of rules". Indeed, the generation of new knowledge is based primarily on existing instances and a set of inference rules. To define such rules, the majority of inference engines support Semantic Web Rule Language (SWRL) [6].

By reusing a common vocabulary, semantic descriptions of the cloud provider services can be shared and understood by the cloud user. Currently, cloud service discovery within a cloud federation requires a significant amount of expertise. However, for a wide acceptance of the CSO, service discovery should become easier and more intuitive. Inference rules capable of reasoning with service descriptions used to ease the discovery task should be introduced.

To define the inference rules, we apply SWRL [6]. The generated knowledge is directly related to the cloud service provider. In fact, through a set of inference rules, we try to quantify the providers’ qualitative criteria in order to facilitate the discovery of suitable provider that satisfies the user’s requirements. Then, for an adequate cloud service discovery, we propose to introduce inferences based on functional and non-functional providers’ capabilities.

The proposed inference reasoning rules are presented on three levels which are: low, medium and high. This can be explained by the fact that the user is not supposed to distinguish the various mechanisms proposed to conclude that
a particular provider offers high, medium or low capability.

**Support-based Inference Generation**

The technical assistance differs from one provider to another.

- **Low Support**: to admit that a cloud provider has a low support, means that he provides only a forum as a technical assistance, as shown in the following rule:

  \[
  Provider(?p) \land support(?p, ?v) \land sqwrl : makeSet(?s, ?v) \land \\
  sqwrl : groupBy(?s, ?p) \land sqwrl : element("Forum", ?s) \land \\
  sqwrl : size(?l, ?s) \land swrlb : equal(?l, 1) \rightarrow Low\_Support(?p)
  \]

- **Moderate Support**: in addition to the forum, the cloud provider has another technical assistance, such as Forum, Social Media, Chat, Phone or Email.

- **High Support**: it considers a cloud provider as a high support that requires at least these means of assistance: Forum, Social Media, Chat, Phone and Email.

**API-based Inference Generation**

The second property that characterizes a cloud provider is APIs owning. If a provider has APIs compatible with other providers, he is considered as a provider who supports cooperation with others. Depending on the number of APIs, we try to classify the cloud providers.

- **Low API Support**: a cloud provider who has only his own API and has no compatibility with other cloud service providers.

- **Moderate API Support**: As shown in the following rule, it’s the fact that a cloud provider has in addition to his own API, another API compatible
with another providers.

\[
Provider(?p) \land API(?p, ?v) \land sqwrl : makeSet(?s, ?v) \land \\
\text{sqwrl : } \text{groupBy}(?s, ?p) \land \text{sqwrl : } \text{element}("API", ?s) \land \text{sqwrl : } \text{size}(?l, ?s) \land \\
\text{swrlb : } \text{greaterThan}(?!l, 1) \rightarrow \text{Moderate}_\text{API}_\text{Support}(?p)
\]

- **High/API/Support**: a cloud provider is compatible with more than one provider.

**Security-based Inference Generation**

Security is one of the most important criterion to trust any cloud provider and address the minimum cloud environment risks. As mentioned in Section 3, we choose to measure security considering certifications, protection mechanisms and data management capabilities. Regarding certifications, each provider exposes his certifications as a proof that ensures the adoption of best security strategies.

According to this criterion, we classify providers into three classes:

- **Low/Certification**: a cloud provider who has less than three certifications and has no Cloud Security Alliance (CSA) is considered as a provider with low certifications, as shown in the following rule:

\[
Provider(?p) \land Provider\_ServiceCapabilities(?p, ?c) \land Security(?c) \land \\
certifications(?c, ?v) \land sqwrl : makeSet(?s, ?v) \land sqwrl : \text{groupBy}(?s, ?p) \land \\
sqwrl : \text{size}(?l, ?s) \land \text{swrlb : } \text{equal}(?!l, 1) \land \\
sqwrl : \text{notElement}("CSA", ?s) \rightarrow \text{Low}\_\text{Certification}(?p)
\]

- **Moderate/Certification**: to admit that a provider belongs to Moderate/Certification class, he should have more than three certifications, except CSA.

- **High/Certification**: if a cloud provider publishes detailed information of his security controls, such as the questionnaire provided by Cloud Security Alliance (CSA), then he is classified in the third class named High/Certification.

In addition, we classify cloud service providers according to their protection mechanisms.
• Low_Protection: if a provider offers only a firewall as a protection mechanism, then he is considered as a low protection provider.

• Moderate_Protection: as shown in the following rule, the use of other protection mechanisms other than VPN, in addition to firewall, enables to qualify the cloud provider as moderate protection provider.

\[
Provider(?p) \land Provider\_ServiceCapabilities(?p, ?c) \land Security(?c) \land
protections(?c, ?v) \land sqwrl:makeSet(?s, ?v) \land sqwrl:groupBy(?s, ?p) \land
sqwrl:element("Firewall", ?s) \land sqwrl:notElement("VPN", ?s) \land
sqwrl:size(?l, ?s) \land swrlb:greaterThan(?l, 1) \rightarrow Moderate\_Protection(?p)
\]

• High_Protection: in order to be a cloud provider with high protection, he must, not only own a firewall and a third-party security, but also provide a VPN (Virtual Private Networking) as protection mechanisms.

The last criterion that can influence the security of cloud service providers is the data management. Based on the standards and principles proposed by a cloud provider, we can characterize his level of data control and classify him into one of the three following classes:

• Low_Data_Management: if the cloud provider complies with either the Safe Harbor law or the law of EU Directive 95/46/EC so he has low data control policy.

• Moderate_Data_Management: using at least two laws at the same time, expect of the NIST SP800-88, increases the data level control.

• High_Data_Management: it uses both Safe Harbor and EU Directive 95/46/EC laws as well as the NIST SP 800-88 for the data destruction. The following inference rule identifies cloud providers having a high degree of data
In order to classify security in general and consequently propose a more flexible cloud services discovering approach for users, we define the following classes.

- **Low Security**: as shown in the following inference rule, security is considered low if the three measures (Certification, Protection and Data Management) are low.

\[
\text{Provider}(p) \land \text{Low Certification}(p) \land \text{Low Protection}(p) \land \\
\text{Low Data Management}(p) \rightarrow \text{Low Security}(p)
\]

- **High Security**: whether the three measures used to quantify the security of a cloud service provider are high.

- **Moderate Security**: if security is neither high nor low, then certainly it is Moderate. This level includes all the other possibilities. Indeed, we can cite the following cases: (i) if the level of certifications is low with a moderate or high level of protection and with any level of data management, (ii) if the level of certifications is moderate implies a moderate security, (iii) if the level of certifications is high with low or moderate level of protection and with any level of data management, (iv) if the level of certifications is low with a low level of protection and with a moderate or high level of data management, (vi) if the level of certifications is high with a high level of protection and with a low or moderate level of data management and (vii) if the three proposed security measures are moderate.
Once CSO is populated, we can first apply the predefined inference rules to generate the above presented knowledge to help the users discover appropriate cloud services. However, the manual construction of the ontology as well as its semi-automatic population process can lead to an incomplete, inconsistent or redundant ontology and consequently impact the quality and the effectiveness of the CSO. To avoid these disadvantages, we propose in the following section an effective evaluation approach to detect and correct consistency, redundancy and incompleteness errors to ensure a large and a best CSO utilization.

5. Pattern and Anti pattern-based Ontology Evaluation Approach

Evaluation is considered as a crucial step in measuring the quality of an ontology since it has a direct impact on the ontology understandability and reusability. According to Gómez-Pérez, ontology evaluation is an important phase in the ontology life-cycle. It "is a technical judgment of the content of the ontology" [31]. In this work, we propose an evaluation approach to detect errors within CSO and correct them in order to increase the CSO quality and guarantee its utilization.

As shown in Figure 4, the proposed approach is a two-phase approach: the ontology evaluation phase, that includes consistency verification, anti-pattern detections and recommendations, and the ontology correction phase. The remainder of this section describes the phases, separately and how they find out complement each other.

5.1. Consistency Verification

A set of preliminary ideas and frameworks have been suggested in the literature in order to get rid of errors within an ontology. Several reasoners (inference engine or classifier) have been proposed in order to automatically check the consistency and the redundancy of a concept within an ontology, such as Pellet [32], Fact++ [33], HermiT [34], etc. The main objectives of the reasoners are to check the consistency of an ontology, verify unintended relationships between classes
and classify individuals into classes. According to [31], "a given definition is consistent if and only if the individual definition is consistent and no contradictory knowledge can be inferred from other definitions and axioms". To validate the CSO consistency, we choose to use the DL reasoner FaCT++ [33] which is free and available with the Protégé editor. Indeed, while loading CSO in Protégé and applying the FaCT++, we remark that no error is detected. Thus, in order to investigate the manner that FaCT++ handles the ontology errors, we introduce some errors, such as:

- we define an instance “IaaS_1” and we link it directly to class IaaS instead of IaaS sub-classes.

- we add a disjoint relation between the class Broker and the sub-class User_IaaS, yet, we already have one between the class Broker and the super-class User.

- we omit the disjoint relation between the sub-class Functional_Capability and the sub-class Non_Functional_Capability. By doing so, it’s not obvious that the class Provider_Capability represents the union of these two
classes.

- we define a new object property named `hasArchitectureFederation` that has the same couple (domain, range) of `hasFederationArchitecture` which is (FederationCloud, ArchitectureFederation) and

- we add a disjoint relation between the two sub-classes Platform and Software, yet, we already have one between the PaaS and SaaS super-classes.

Thereafter, while executing the FaCT++, the reasoner considers our definition of the instance "IaaS_1" as a normal one. Furthermore, he does not detect any error.

To the best of our knowledge, there is no reasoner that is able to detect such anomalies and errors.

As shown in Figure 4, if the ontology is inconsistent, its correction phase should be followed by the consistency verification phase, that should be repeated. Otherwise, the anti-pattern detection step should be triggered.

5.2. Anti-Pattern Detection and Recommendations

To optimize the evaluation effort, it is better to reuse and benefit from the existing reasoners and cover their limitations by a set of patterns and anti-patterns. In this section, we intend to detect the errors undiscovered by the FaCT++ reasoner using a set of proposed patterns and anti-patterns. The Ontology Design Patterns (ODP) encode best practices in some fields [35, 36]. Indeed, the introduction of ODP represents a good solution to the recurring modeling problems and creates and maintains ontologies, which help create rich and rigorous ontologies with less effort. In opposition to patterns, anti-patterns produce the side effect of checking errors in an ontology. They might help, guide and train ontology developers to contribute to the quality of produced ontologies. According to Roussey et al., the detection of ontology anti-patterns contributes to ontology quality assessment [37]. In fact, to validate the CSO cloud standard conformity, we define the following patterns and anti-patterns.
5.2.1. Decomposition Pattern Definition

As defined in [35], the ontology design pattern is a modeling solution to a recurrent ontology design problem. In a previous work [38], we have presented OCCI REST patterns and anti-patterns to assess cloud service APIs compliance with OCCI standard. Whilst, in this work, the aim of patterns and anti-patterns is to detect errors and anomalies of cloud service description. The patterns definition is mainly based on a taxonomy that hierarchically organizes classes and instances within an ontology [31]. In the cloud domain, a taxonomy can be identified while referring to standards (e.g., NIST, OCCI, CIMI, TOSCA) that hierarchically present cloud service descriptions. In this context, we define the following decomposition patterns considered as generic and can be applied to any ontology. The hierarchy definition should meet one of the following decomposition manners:

1. Disjoint decomposition: This pattern defines a set of disjoint sub-classes of class C. This classification implies that each instance can belong directly to class C or to one of the sub-classes of C, as shown in Figure 5. This pattern is expressed through DL as follows:

\[
C_1 \sqsubseteq C; C_2 \sqsubseteq C; C_3 \sqsubseteq C; C_1 \sqcap C_2 \sqcap C_3 \sqsubseteq \bot;
\]

2. Exhaustive decomposition: This pattern defines a complete classification of sub-classes of class C which are not necessarily disjoint. Indeed, each instance of class C should be an instance of one or more of the sub-classes as shown in Figure 6. This pattern is expressed through DL as follows:

\[
C_1 \sqsubseteq C; C_2 \sqsubseteq C; C_3 \sqsubseteq C; C_1 \sqcup C_2 \sqcup C_3 \equiv C;
\]
3. Partition decomposition: This pattern defines a set of disjoint sub-classes of class C. This classification is also complete and class C is the union of all the sub-classes. Each instance of class C should be an instance of only one sub-class, hence no shared instances are allowed (shown in Figure 7).

This pattern is expressed through DL as follows:

\[
C_1 \sqsubseteq C; C_2 \sqsubseteq C; C_3 \sqsubseteq C;
\]

\[
C_1 \cap C_2 \cap C_3 \sqsubseteq \bot; C_1 \cup C_2 \cup C_3 \equiv C;
\]

5.2.2. Anti-Pattern Definition

An anti-pattern is similar to a pattern, “except that instead of a solution, it gives something that looks superficially like a solution, but it is not” [39].

In our work, we propose three anti-pattern categories: taxonomic errors, instantiation anomalies and domain errors. Each category includes a set of anti-patterns. It is worth noting that the anti-patterns devoted to detect taxonomic errors and instantiation anomalies are generic, which can be applied on any ontology domain. However, the anti-patterns for detecting the domain errors are specific to a cloud service description ontology. On the other hand, taxonomic and domain errors cover, respectively, the errors related to the concepts as well
as to the instances. Indeed, the consideration of the instantiation anomalies permits to improve the ontology usability [40].

1. **Taxonomic errors**: In this category, we aim at detecting possible violations of the set of patterns already defined in Section 5.2.1. In this context, we identify different error types that occur when modeling taxonomic knowledge within an ontology.

   - **Inconsistency**: occurs when there are external instances in a complete classification or when the disjoint relation is defined between classes with different hierarchies.

     - **External instances in exhaustive decomposition and partition**: these errors occur when there is one or more instances of the class that do not belong to any sub-class. This generic anti-pattern is expressed through DL as follows:

       \[
       \forall \text{Individual}_1; C(\text{Individual}_1) \sqsubseteq C; C_1 \sqsubseteq C; C_2 \sqsubseteq C; C_3 \sqsubseteq C; C_1 \sqcup C_2 \sqcup C_3 \equiv C;
       \]

     - **Partition error**: appears when a class is disjoint with the sibling of its super-class instead of being disjoint with its sibling classes. This anti-pattern can be expressed through DL as follows:

       \[
       C_1 \sqsubseteq C; C_2 \sqsubseteq C; C_3 \sqsubseteq C_1; C_2 \sqcap C_3 \sqsubseteq \bot;
       \]

   - **Incompleteness**: occurs when the ontologists omit the definition of the disjoint or the union relations. In fact, information about the super class or the sub-classes is missing. Thus, we identify an anti-pattern called *Decomposition knowledge Omission*, which covers two types of omission errors: (i) the omission to identify that the sub-classes of concept are disjoint or (ii) the omission to identify that
the concept is the union of all its sub-classes. This anti-pattern is expressed through DL as follows:

\[ C_1 \sqsubseteq C; C_2 \sqsubseteq C; \]

- **Redundancy**: appears when two object properties have the same formal definition or when the disjoint relation is duplicated. The redundancy errors cover:
  
  - **Identical formal definition of object properties**: occurs when different object properties have the same formal definition, i.e. they have the same couple (domain, range), although they have different names. This anti-pattern is expressed through DL as follows:

    \[
    \exists R. \top \sqsubseteq C_1; \top \sqsubseteq \forall R.C_2;
    \exists R_1. \top \sqsubseteq C_1; \top \sqsubseteq \forall R_1.C_2;
    \]

  - **Redundancy of disjoint relation**: it is the fact of defining a concept as disjoint with more than one concepts, i.e. classes that have more than one disjoint relation. The DL expression shows that the disjoint relation between sub-classes C3 and C4 is repeated since they have already inherited it from their base classes C1 and C2.

    \[
    C_3 \sqsubseteq C_1; C_4 \sqsubseteq C_2; C_1 \cap C_2 \sqsubseteq \bot;
    C_3 \cap C_4 \sqsubseteq \bot;
    \]

2. **Instantiation anomalies**: The correction of the instantiation anomalies guarantees the usability of an ontology [40]. Actually, these anomalies include:

- **Lazy concept**: there is an instantiated concept of which all the datatype properties are not defined. The correspondent anti-pattern is expressed through DL as follows:

    \[
    \neg \exists T.d \equiv \forall T.-d;
    \]
• **Weak lazy concept**: is an instantiated concept with undefined datatype properties. The anti-pattern can be expressed through DL as follows:

\[ \neg \forall T.d \equiv \exists T.\neg \top; \]

• **Weak concept definition**: this refers to the situation when an instantiated concept has all or some of its object properties not defined. The anti-pattern is expressed through DL as follows:

\[ \neg \forall R.C; \neg \exists R.C; \]

3. **Domain errors**: As opposed to previously cited anti-patterns which are generic, the following ones are specific and aim at evaluating cloud service description ontology. In others words, the following set of anti-patterns focuses on cloud service description errors. It is worth noting that due to space limitations, we choose only a set of such anti-patterns and we refer the reader to the web site\(^5\) in order to find the complete list illustrated with examples.

• **Invalid VMType characteristic**: each VM type must have at least an essential characteristic and a compute capability. Such anti-pattern is expressed through the DL expression as follows:

\[ \neg \exists \text{hasEssentialCharacteristic}. \]

\[ \text{EssentialCharacteristic}; \]

\[ \neg \exists \text{Compute.CapabilityAttachment}. \]

\[ \text{Compute.Capability}; \]

• **Invalid provider description**: each provider has a deployment model, a hosting type, a pricing model and a set of service capabilities. The omission of one or more pieces of information leads to an

\(^5\)https://sites.google.com/site/csoevaluation/home
invalid provider description.

\[ \neg \exists \text{hasDeploymentModel.DeploymentModel}; \]
\[ \neg \exists \text{hasHostingType.HostingType}; \]
\[ \neg \exists \text{hasPricingModel.PricingModel}; \]
\[ \neg \exists \text{Provider_ServiceCapabilities. Provider_Capability}; \]

- **Invalid Compute_Capability definition**: each virtual machine type must have a compute capability, that includes two characteristics: CPU and memory. We admit that these two characteristics could not exist without a compute capability. The following DL expression shows the verification in both directions.

\[ \neg \exists \text{cpuIsPartOf.Compute_Capability}; \]
\[ \neg \exists \text{Compute_Capability.ContainsCPU.CPU}; \]
\[ \neg \exists \text{memoryIsPartOf.Compute_Capability}; \]
\[ \neg \exists \text{Compute_Capability.ContainsMemory. Memory}; \]

- **Invalid Service_Capability description**: each service must have a set of capabilities. For example, the storage and network capacities must be provided by the IaaS service. The following DL expression shows the capabilities that should be offered by each service type.

\[ \neg \exists \text{PlatformCapabilities.Functional Capability}; \]
\[ \neg \exists \text{SoftwareCapabilities.Functional Capability}; \]
\[ \neg \exists \text{StorageCapabilities.Functional Capability}; \]
\[ \neg \exists \text{NetworkCapabilities.Functional Capability}; \]

5.2.3. Anti-pattern Detection Procedure

In this section, we deal with the anti-pattern detection procedure that includes the SPARQL queries corresponding to the anti-patterns defined in Sec-
tion \[5.2.2\] in order to detect errors and anomalies within CSO. Due to the space limitation, we present only the SPARQL query of the invalid VMType characteristic anti-pattern and its correction recommendation (see Listing \[1\]). Such query returns each type of virtual machine (\(?\text{VMt}\)) that is instantiated, but its definition is missing an essential characteristic (\(?\text{E}_C\)) and a compute capability (\(?\text{compC}\)). Once this anti-pattern is detected, the following correction recommendation is displayed in order to guide ontologists to correct the error, "Please complete the description of the VM type class by defining these two object properties: has\text{EssentialCharacteristic} and Compute\_Capability\_Attachment!".

A complete description of the queries and the recommendations is available on our web site\[^6^]\.

<table>
<thead>
<tr>
<th>Listing 1: SPARQL Query: Invalid VMType Characteristic</th>
</tr>
</thead>
</table>
| \[ \text{SELECT} \ ?\text{VMt} \ ?\text{E}_C \ ?\text{compC} \]
| \[ \text{WHERE} \]
| \[ (?\text{VMt} \ \text{rdf:type ns:VMType} \]
| \[ \text{OPTIONAL} \]
| \[ (?\text{VMt ns:hasEssentialCharacteristic} \ ?\text{E}_C) \]
| \[ \text{OPTIONAL} \]
| \[ (?\text{VMt ns:Compute\_Capability\_Attachment} \ ?\text{compC}) \]
| \[ \text{FILTER} \ ( \!\text{bound}(\!\text{?E}_C)) \| ( \!\text{bound}(\!\text{?compC})) \] \]

The proposed detection procedure has as inputs: CSO and the list of SPARQL queries and produces as output, the detected errors and the recommendations of correction. As shown in Listing \[2\] the procedure executes the three following methods: (i) select a query from the list; (ii) execute the query and (iii) once the anti-pattern is detected, the error appears with a list of correction recommendations.

\[^6^\]https://sites.google.com/site/csoevaluation/home
Listing 2: The anti-pattern detection procedure

```
Input: CSO.owl, SPARQL_query.antipattern_list
Output: Detected Errors and Recommendations

Begin
  for each antipattern
    for (each query in SPARQL_query_antipattern_list) do
      result = Execute(query)
      if (Exist(result)) then
        Display("Recommendation Text!")
    }
End
```

The proposed procedure is implemented using the Apache Jena.

It should be noted here that the CSO is semi-automatically populated with 767 interconnected instances while collecting functional information from the providers’ web pages and non-functional information from the well-known monitoring parties.

As shown in Figure 4, after the consistency verification, the procedure is applied on CSO to detect the existing errors. We notice that the taxonomic errors are not detected in CSO thanks to the fact that its construction follows the patterns defined in Section 5.2.1. Beside the instantiation anomalies, the anti-pattern detection procedure detects some domain errors, as shown in Table 3.

The results prove that the classification of CSO concepts is well-defined due to the absence of taxonomic errors. However, the presence of the instantiation anomalies and the domain errors can be explained by several reasons. The main reason is related to the ontology population process. In fact, the CSO

---

7https://jena.apache.org/
<table>
<thead>
<tr>
<th>Errors</th>
<th>Number of Detected Errors</th>
<th>Percentage of CSO’s Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taxonomic Errors</td>
<td>0/5</td>
<td>0%</td>
</tr>
<tr>
<td>Instantiation Anomalies</td>
<td>3/3</td>
<td>100%</td>
</tr>
<tr>
<td>Domain Errors</td>
<td>2/8</td>
<td>25%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5/16</strong></td>
<td><strong>31.25%</strong></td>
</tr>
</tbody>
</table>

population is based on the providers’ catalogs and the cloud service portals which are not conform with the cloud standard. Therefore, the extracted data (i.e., cloud service description) can include some errors. Moreover, since the extracted data can be incomplete. For this reason, concepts, such as VMTyp and ServiceCapability, have some missing information within CSO.

Figure 8 shows that the anti-pattern detects errors, namely Invalid VMTyp Characteristic, and its correction recommendations.

![Anti-Pattern: Invalid VMTyp Characteristic](image)

Figure 8: Detected Invalid VMTyp Characteristic Error

If no error is detected, ontology is considered as evaluated. Otherwise, for
each detected error or anomaly, a recommendation of correction should be triggered.

In fact, the set of recommendations can be taken into consideration by ontologists within the correction phase. This manual phase consists in correcting the errors detected by either the reasoner or by following the recommendations proposed by the anti-patterns. The ontology evaluation phase will be redone in case recommendations are ignored by the ontologists and consequently the evaluation phase is terminated.

6. CSO interrogation for cloud service discovery

In order to apply the CSO in an adequate cloud services discovery, we implement a Cloud Services Ontology Interrogation and Evaluation (CSOIE) prototype. The CSOIE prototype, which is based on the inference rules already defined in Section 4, can be adopted by users who are generally unfamiliar with the SPARQL language. Indeed, the prototype helps them to express their functional and non-functional requirements through a dedicated interface and then generates the equivalent SPARQL query. The implementation details of the CSOIE are available on 8. Two series of experiments were conducted in order to validate the correctness of the CSO responses: the first one consists in assessing the CSO performance where effectively selecting the appropriate services based on expert interpretations while the second demonstrates a real case study for services discovery. These experiments were conducted on a laptop with a 64-bit Intel Core 2.50 GHz CPU, 6 GB RAM and Windows 8 as an operating system.

6.1. Expert-based Interpretation

We choose to test about 100 cloud service discovery requests in order to evaluate the CSO performance. Thereafter, we ask ontology and cloud teachers from the IT department of the University of Sfax (considered as domain expert)

8https://drive.google.com/file/d/0B17XK9qGY-dQNEI2U29GeGJOems/view
to evaluate the quality of the ontology discovery results. These results are measured using the two most useful metrics: precision and recall. The precision represents the proportion of retrieved services that accurately match the user’s requirements. It is calculated according to Equation 1. The recall depicts the proportion of relevant services that are retrieved. It is calculated according to Equation 2.

\[
\text{Precision} = \frac{|\{\text{relevantservices}\} \cap \{\text{retrievedservices}\}|}{|\{\text{retrievedservices}\}|} \tag{1}
\]

\[
\text{Recall} = \frac{|\{\text{relevantservices}\} \cap \{\text{retrievedservices}\}|}{|\{\text{relevantservices}\}|} \tag{2}
\]

In this context, we propose some requests that can be accurately matched with available services (Exact). Besides, we add some requests that can have a good matching score with the available services (Not Exact). Based on this scenario, as shown in Table 4, we propose five requests with different requirements.

Through the experiments, we obtain:

- a precision of 99% and a recall of 95%, which are considered as very satisfactory rates. Hence, CSO can return all the services related to the requests with a high rate of “false-positive” (indicated by precision) by respecting functional and non-functional end users’ requirements. This proves the CSO correctness, which makes it a relevant and a pertinent means for users to select the appropriate cloud services.

- an average response time of 12.5 seconds, which represents a reasonable processing time for the selection of the most appropriate service.

Overall, these results demonstrate that with the proposed ontology, a cloud user is able to efficiently find all or almost the services fulfilling his requirements, with the lowest rate of ”false positive”.

6.2. Real case study application of CSO

To apply the CSO in a real case scenario, we considered an industrial case study from the French company of telecommunication ”Orange Teleco”.

company decided to outsource a business process called "Quality Supervisory Process" in a cloud federation environment. This process is considered expensive in terms of processing and network. The company requires a medium security level and highly available cloud providers. It also wished a highly technical support cloud provider. Besides, the elasticity, the backup, the recovery, the load balancing and the monitoring capabilities should characterize the cloud providers in the federation. The required virtual machines must have a CPU capacity at least equal to two cores, a capacity memory that exceeds 3 GB and a storage volume exceeding 100 GB. For network capacity, the company needed a minimum capacity of 50 MB. The virtual machines must be available in Europe and have the Windows operating system. Figure 9 shows providers and their service lists which adequately meet the company’s functional and non-functional requirements.

![Interface-based CSO interrogation](image.png)

Figure 9: Interface-based CSO interrogation
Hence, our proposal promotes the creation of a common semantic knowledge base of homogeneous cloud service description which facilitate the discovery of cloud resources when a cloud user desires to optimally allocate cloud resources to execute his business process into a cloud federation [41].

7. Conclusion

In this paper, we proposed a domain ontology named Cloud Service description Ontology (CSO), in order to address the heterogeneity problem of cloud service description within a cloud federation. CSO was manually built while referring to existing cloud standards, standardization initiatives and research proposals. It commonly specifies the knowledge shared between different cloud providers within a cloud federation in terms of functional and non-functional capabilities. In addition, we introduced a set of axioms and inference rules to generate knowledge related to the functional and non-functional providers’ capabilities. To improve the ontology quality, we proposed an ontology evaluation approach including an evaluation and a correction phases. Thus, we applied, in a first step, the FaCT++ reasoner to validate the consistency of the proposed CSO. In a second step, an anti-patterns detection procedure based on SPARQL queries is applied in order to detect errors and anomalies not covered by the reasoner. The proposed procedure offers a set of correction recommendations in order to revise CSO. Thereafter, we proposed an end user interrogation prototype to ease the cloud service discovery. The ontology based discovery is tested by domain experts as well as by applying it on a real case study. The experimental results demonstrate the ability of the CSO to efficiently discover cloud services according to both functional and non-functional user’s requirements.

As future works, we plan to extend the proposed ontology to deal with other non-functional provider capabilities, such as reliability and response-time. In addition, we plan to enhance the evaluation approach by defining semantic patterns and anti-patterns in order to consider semantic redundancies (semantically identical concepts and/or properties), semantic incompleteness (semantically
identical of relationships between concepts and/or properties) and semantic inconsistencies (concepts and/or properties that subsume other concepts and/or properties). Furthermore, we intend to contact other research groups for a wider evaluation of CSO.

References


[25] V. Uren, P. Cimiano, J. Iria, S. Handschuh, M. Vargas-Vera, E. Motta, F. Ciravegna, Semantic annotation for knowledge management: Require-


<table>
<thead>
<tr>
<th>Request</th>
<th>User’s Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req1</td>
<td>CPU=4, RAM=6, Storage=1000, Location=Europe, Use case=General, VM type price (max)=1.5, Network capability=10, Network price (max)=1, Operating system=Windows, Essential characteristic=OnDemand Service, Availability (min)=90, Security=Moderate, API Support=Moderate, Support=High, Elasticity=Vertical, Backup and recovery=Yes, Load Balancing=Yes, Monitoring=Yes.</td>
</tr>
<tr>
<td>Req2</td>
<td>CPU=2, RAM=4, Storage=500, Location=Europe, Use case=General, VM type price (max)=1, Network capability=10, Network price (max)=0.75, Operating system=Lunix, Essential characteristic=OnDemand Service, Availability (min)=95, Security=High, API Support=Low, Support=Moderate, Elasticity=Vertical, Backup and recovery=yes, Load Balancing=yes, Monitoring=yes.</td>
</tr>
<tr>
<td>Req3</td>
<td>CPU=4, RAM=8, Storage=10000, Location=Asia, Use case=Memory optimized, VM type price (max)=1.5, Network capability=100, Network price (max)=1.5, Operating system=Windows, Essential characteristic=OnDemand Service, Availability (min)=90, Security=High, API Support=High, Support=High, Elasticity=Vertical, Backup and recovery=Yes, Load Balancing=No, Monitoring=Yes.</td>
</tr>
<tr>
<td>Req4</td>
<td>CPU=2, RAM=4, Storage=100, Location=Asia, Use case=Compute optimized, VM type price (max)=0.5, Network capability=100, Network price (max)=0.5, Operating system=Windows, Essential characteristic=OnDemand Service, Availability (min)=95, Security=Moderate, API Support=High, Support=Moderate, Elasticity=Vertical, Backup and recovery=Yes, Load Balancing=Yes, Monitoring=Yes.</td>
</tr>
<tr>
<td>Req5</td>
<td>CPU=4, RAM=8, Storage=2000, Location=Europe, Use case=Storage Optimized, VM type price (max)=1, Network capability=100, Network price (max)=1, Operating system=Linux, Essential characteristic=OnDemand Service, Availability (min)=95, Security=High, API Support=Moderate, Support=Moderate, Elasticity=Vertical, Backup and recovery=Yes, Load Balancing=No, Monitoring=Yes.</td>
</tr>
</tbody>
</table>