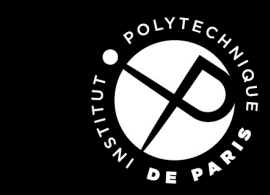


INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2020IPPAS003

Thèse de doctorat

Supporting Management and Orchestration of Cloud Resources in Multi-cloud environment

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à TELECOM SudParis en cotutelle avec la Faculté des Sciences Économiques et de Gestion de Sfax

École doctorale n°ED 626 de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Evry, le 13/02/2020, par

HAYET BRABRA

Composition du Jury :

Philippe Merle Professeur, Inria Lille, France	Président
Fabio Casati Professeur, Université de Trento, Italy	Rapporteur
Carine Souveyet Professeur, Université Paris 1 Panthéon Sorbonne, France	Rapporteur
Boualem Benatallah Professeur, Université de New South Wales, Australie	Examinateur
Amel Bouzeghoub Professeur, TELECOM SudParis, France	Examinateur
Lotfi Chaari Associate professor HDR, INP Toulouse, France	Examinateur
Walid Gaaloul Professeur, TELECOM SudParis, France	Directeur de thèse
Faiez Gargouri Professeur, Institut Supérieur d'Informatique et de Multimédia, Tunisie	Directeur de thèse
Achraf Mtibaa Associate professor, École Nationale d'Electronique et des Télécommunications, Tunisie	Invité
Layth Sliman Enseignant-chercheur, École d'Ingénieurs Informatique - Efrei Paris	Invité

Titre : Supporter la gestion et l'orchestration des ressources cloud dans un environnement multi-nuages

Mots clés : Elasticité, Orchestration, Gestion, Multi-nuages, APIs, DevOps

Résumé : Face à la croissance de l'adoption des ressources cloud dans les tâches informatiques quotidiennes, la demande pour des techniques d'orchestration et de gestion efficaces a considérablement augmenté. Cependant, comme le cloud souffre d'un manque évident d'interopérabilité, l'orchestration et la gestion des ressources cloud réparties entre des fournisseurs hétérogènes deviennent des tâches très complexes et coûteuses. En outre, l'élasticité, caractéristique distinctive de l'informatique en nuage, est devenue primordiale pour préserver la qualité de service souhaitée tout en optimisant les coûts impliqués. Néanmoins, la dynamicité du cloud et son hétérogénéité inhérente rendent la gestion de l'élasticité une tâche très fastidieuse.
Afin de surmonter ces difficultés, nous visons dans cette thèse à (i) fournir de l'orientation et de l'assistance à la conception d'APIs de gestion interopérables, (ii) rationaliser l'orchestration des ressources cloud et (iii) supporter la gestion de haut niveau de l'élasticité multi-nuages. Pour atteindre le premier objectif, nous adoptons des patrons et des antipatrons comme moyen de représenter les bonnes et les mauvaises pratiques des principes OCCI (Open Cloud Computing Interface Standard) et REST (Representational State Transfer) que les développeurs doivent prendre en compte lors de la conception de leurs APIs. Nous proposons ensuite une approche sémantique permettant la détection automatisée de ces (anti) patrons tout en fournissant un support de recommandation afin de guider les développeurs à la révision de leurs APIs. Pour atteindre le deuxième objectif, nous appuyons l'idée d'intégrer la norme TOSCA (Topology and Orchestration Specification for Cloud Applications) avec les solutions DevOps en tant qu'une étape essentielle pour atténuer la complexité liée au processus d'orchestration tout en maintenant le niveau d'interopérabilité souhaité. Pour soutenir cette intégration, nous proposons une approche dirigée par les modèles.
Pour assurer le troisième objectif, nous reconnaîtrons que les caractéristiques d'élasticité devraient être fournies au niveau de la description de la ressource au lieu de reposer sur des mécanismes dépendant de la technologie. Pour ce faire, nous proposons un nouveau modèle de description de l'élasticité basé sur le formalisme de la machine à états. Enfin, nous développons trois preuves de concept et réalisons des expériences approfondies pour valider nos approches.

Title : Supporting Management and Orchestration of Cloud Resources in Multi-cloud environment

Keywords : Elasticity, Orchestration, Management, Multi-cloud, APIs, DevOps

Abstract : With the rising adoption of cloud resources in everyday computing tasks, the demand for effective orchestration and management techniques has considerably increased. However, as there has been a clear lack of cloud interoperability, orchestrating and managing elastic cloud resources distributed across heterogeneous providers become very complex and costly missions. Moreover, elasticity, the key distinguishing feature in cloud computing, has become primordial to preserve the desired quality of service while optimizing the involved costs. Nevertheless, the rapid dynamicity of the cloud and its inherent heterogeneity make supporting elasticity a very tedious task.
To resolve these issues, in this thesis, we aim to (i) provide guidance and assistance in the design of interoperable management APIs ; to (ii) streamline the orchestration of cloud resources and to (iii) support high-level management of multi-cloud elasticity. To achieve the first objective, we adopt patterns and antipatterns as means to represent the good and poor practices of both OCCI (Open Cloud Computing Interface Standard) and REST (Representational State Transfer) principles that API developers should be taking into account when designing their APIs. We then propose a semantic-based approach that allows automated detection of these (anti) patterns as well as provides recommendation support to guide cloud developers in revising their management APIs.
To achieve the second objective, we support the idea of integrating TOSCA (Topology and Orchestration Specification for Cloud Applications) with DevOps solutions as an essential step toward alleviating the complexity related to the orchestration process while maintaining a desired level of interoperability. To support this integration, we propose a model-driven approach following MDE principles. To ensure the third objective, we realize that the elasticity features should be provided at the resource description level instead of relying on low-level and technology-dependent mechanisms. So, we propose a new Cloud resource elasticity description model based on the state-machine formalism.

Institut Polytechnique de Paris
91120 Palaiseau, France

To my mother,
to my father,
to my loving family
for their unconditional
love and support.

Acknowledgment

First and foremost, I thank god for giving me strength, knowledge, ability and opportunity to undertake this research work. A thesis journey is not always easy, but with blessings of God, the supervision, support, and encouragement of many people, I have never regretted that I started it.

I would like to thank all members of the jury. I thank Professor Carine Souveyet and Professor Fabio Casati for accepting being my thesis reviewers and for their attention and thoughtful comments. I also thank Professor Boualem Benatallah, Professor Philippe Merle, Professor Amel Bouzeghoub and Dr. Lotfi Chaari for accepting being my thesis examiners.

I would like to express my deepest thanks and gratitude to my supervisor Walid Gaaloul. His valuable advice, enthusiasm and constant support during this thesis allowed me to acquire new understandings and extend my experiences. He was not only an advisor but also a good listener who showed me just what I was able to achieve even when I did not see it myself. Thank you for taking me on this journey and I deeply hope that we can continue our collaboration.

I am also grateful to my supervisor Faiez Gargouri for allowing me to join his team. His wide knowledge and logical way of thinking have been of great value for me. I am also grateful to my supervisor Achraf Mtibaa for his great advice and his guidance. I am thankful for the opportunities he provided, and for having faith in me.

I am also thankful to Layth Sliman, Fabio Petrillo, Yann Gaël Guéhéneuc, Naouel Moha, Mohamed Sellami, and Kais Klai for the beneficial collaboration we have had. A special thanks go to the anonymous reviewers, evaluation participants and to my master students Marwa and Sarra for their beneficial collaboration in the implementation part.

I owe my deepest gratitude and warmest affection to the members of the computer science department of Telecom SudParis and Faculty of Economics and Management. I would like to thank Brigitte Houassine, Veronique Guy, Nizar Nasri, and Emna Hamza, for their kind help and assistance. I also thank my office mates Emna, Rami, Kunal, Wassim, Rania, Souha, Nabila, Hela for the lovely moments we spent.

I am forever thankful to my mother Zina who keep showering me with her love and kindness, your love and prayers made me keep moving forward, and to my father who kept asking me about the end of my research although I keep giving him wrong

answers. I try my best to make you both proud. I also thank my sisters Rachida and Halima, my sisters-in-laws Jamila and Monia, my brothers Najeh, Fawzi, Bassem, and my brother in law Kamel, who were always there for me with encouraging words whenever I started doubting myself. I cannot forget their beautiful kids Mohamed-Yasin, Amen, Ahmed, Youssef, Yassine, Fatma. I dedicate this thesis to all of you, my wonderful family.

I am forever thankful to my friends, Ikram who did the second reading of most parts of this thesis, Ahlem, Nesrin, Aicha, Leila, Halima, Farah, Emna, Wafa, Sabra and my cousin Noura for continuous support and encouragement.

Abstract

With the increased adoption of cloud computing, multiple and heterogeneous configuration and management APIs/tools and platforms have been proposed to enable end-to-end management and orchestration tasks. However, this proliferation is one of the fundamental reasons that has intensified the heterogeneity issue in multiple respects, making the cloud interoperability very difficult to achieve. With the lack of interoperability, orchestrating and managing elastic cloud resources distributed across heterogeneous providers become very complex and costly missions for the cloud consumers.

Towards fostering cloud interoperability, standardization is a definitive method according to many professionals and researchers from both academic and industrial sectors. In this respect, we are interested in two relevant standards, namely OCCI and TOSCA because of their broad adoption and holistic approaches in addressing cloud interoperability from management and orchestration perspectives. OCCI is essentially introduced to create remote management REST APIs for supporting the management of any kind of cloud resources while preserving a high level of interoperability. On the other hand, TOSCA is introduced to empower cloud interoperability by modeling cloud applications in a technology-independent manner. With this specification, it ultimately aims at automating the whole application orchestration process, which includes the selection, deployment, monitoring, and runtime controlling of cloud resources.

Despite the growing interest in TOSCA and OCCI, their adoption is still not prevalent in modern solutions. More specifically, there is a lack of holistic orchestration engines that supports TOSCA, while there is no assistance for developers to create interoperable management APIs according to OCCI. In this thesis, we believe that any innovative design and orchestration approaches adopting such standards would be beneficial in order to cope with the heterogeneity and interoperability issues. In addition to avoiding the above issues, preserving the desired quality of service (QoS) while optimizing the involved cost is of paramount importance for both cloud users and providers. Elasticity is known as a key factor to ensure this cost-QoS tradeoff. However, the rapid dynamicity of the cloud and its inherent heterogeneity make supporting elasticity a very tedious task.

In this thesis, we aim to (i) provide guidance and assistance in the design of interoperable management APIs; to (ii) streamline and improve the orchestration of cloud resources and to (iii) support high-level management of multi-cloud elasticity. To achieve the first objective, we adopt patterns and anti-patterns as means to represent respectively the good and poor practices of both OCCI and REST best principles that API developers should be carefully taking on when designing their APIs. We then propose a semantic-based approach that allows automated detection of these (anti) patterns as well as provides recommendation support to guide cloud develop-

ers in revising their management APIs. To achieve the second objective, we support the idea of integrating TOSCA with DevOps solutions as an essential step toward alleviating the complexity related to the orchestration process while maintaining a desired level of interoperability. To support this integration, we propose a model-driven approach following MDE principles. To ensure the third objective, we realize that the elasticity features should be provided at the resource description level instead of relying on low-level and technology-dependent mechanisms. So, we propose a new Cloud resource elasticity description model based on the state-machine formalism. Finally, to validate our approaches, we develop three proofs of concepts and conduct a set of extensive experiments, some of which are with academics and professionals to demonstrate their effectiveness and feasibility.

Table of contents

List of Tables

List of Figures

Chapter 1

General Introduction

Contents

1.1 Research context

Since its launch, Cloud Computing has revolutionized the whole IT field shaking up both industry and academia. It is recognized as an effective technology allowing easy and on-demand access to a shared set of configurable computing resources delivered as services. Attracted by its economic pay-as you-go model and reduced maintenance cost, many enterprises are competitively adopting cloud computing in order to enhance their business outcomes as well as achieve cost-to-performance tradeoffs. A Gartner survey conducted in April 2019 mentioned that more than a third of enterprises consider the investment in the cloud as a top-three investing priority. The same survey reported that the global market related to cloud services is estimated to grow by 17.5% between 2018 and 2019 to go from $182.4 billion in 2018 to $214.3 billion in 2019.

With the noteworthy growth of cloud computing, a vast variety of cloud providers have emerged. They deliver computing resources with competitive and reduced costs. To accommodate this momentum, multiple and heterogeneous configurations and management APIs, tools and platforms have been suitably proposed to enable the end-to-end management tasks. However, this proliferation is one of the fundamental reasons that has intensified the heterogeneity issue in multiple respects, making the cloud interoperability very difficult to achieve. Cloud Interoperability refers to

the ability of the system to execute software programs across multiple clouds and interact with their underlying systems regardless of their physical architectures and technical specifications [159]. In fact, with the lack of interoperability, managing a potentially large number of cloud resources can only be done at a high cost. Unfortunately, enterprise applications are inherently complex and consist of multiple individual components that potentially involve a large number of cloud resources going beyond the capacity of one provider. Thus, the development of such applications would imperatively become distributed across multiple and heterogeneous providers, which renders their orchestration during different phases of their life-cycle a very complex and costly task. Indeed, cloud orchestration includes the creation, management, and manipulation of cloud resources in order to fulfill the business objectives of cloud applications [98]. Consequently, with the observed above issues, designing effective solutions for the orchestration and management tasks spanning across multiple cloud environments remains a deep and open challenging problem [47, 135, 155].

In this regard, many authors and professionals from both academic and industrial sectors [57, 69, 75, 79, 84, 92, 135, 155, 159] noted that open standards are the key factors towards avoiding heterogeneity issues and fostering cloud interoperability. As a consequence, many projects have been carried out as a joint effort between industry and academia to develop open standards. In this thesis, we are interested in using two relevant standards, namely OCCI and TOSCA because of their broad adoption and holistic approaches in addressing cloud interoperability from both management and orchestration perspectives. OCCI [118] is the open cloud computing interface proposed by the Open Grid Forum (OF), to essentially address heterogeneity, interoperability, integration, and portability in cloud computing. In essence, OCCI was introduced to create a remote management API to support the management of any kind of cloud resources while preserving a high level of interoperability. To achieve this goal, OCCI provided a set of guidelines in its different specifications, forming a minimal set of practices to achieve interoperability by providing a uniform way to discover and to manage cloud resources across various providers. Arguably, following these guidelines is a definitive requirement for developers to enforce the interoperation of cloud resources. TOSCA [116], the OASIS Topology and Orchestration Specification for Cloud Applications, was introduced to empower cloud interoperability from an orchestration perspective. This has been achieved thanks to TOSCA's support for the modeling of cloud applications in a technology-independent manner without referring to any specific cloud provider. With such modeling, it ultimately aims at automating the whole orchestration process of applications across multiple clouds. In simple terms, the orchestration process (OP) includes the set of operations that cloud users take on for selecting, deploying, monitoring, and dynamically controlling the configuration of hardware and software resources.

Despite the growing interest in TOSCA and OCCI, their adoption is still not prevalent in modern solutions [135, 155]. More specifically, there is a lack of holistic orchestration engines that supports TOSCA, while there is no assistance for develop-

ers to create interoperable management APIs according to OCCI. In this thesis, we believe that any innovative design and orchestration approaches adopting such standards would be beneficial in order to cope with the heterogeneity and interoperability issues.

In addition to avoiding the above issues, preserving the desired quality of service (QoS) while optimizing the involved cost has paramount importance to both cloud users and providers. Elasticity, the key distinguishing feature of cloud resources, is known as a key factor to ensure this cost-QoS trade-off [11, 53]. In simple terms, it refers to the capability of dynamically reconfiguring cloud resources to adapt to varying resource requirements. However, the rapid dynamicity of the cloud and its inherent heterogeneity make supporting elasticity a very tedious task. Certainly, this issue is getting worse with the insistence on the adoption of federated cloud resources. Therefore, in this thesis, we argue that the more effective elasticity solution should shield resource users from cloud heterogeneity and protect cloud applications with guaranteed QoS despite variations and dynamicity in the underlying environment.

Summing up, the ultimate goal of this thesis is providing solutions that would aid cloud designers in creating interoperable management APIs and in supporting orchestration and elasticity of cloud resources in heterogeneous and multi-cloud environments.

1.2 Motivation and problem statement

We intend to facilitate the management and orchestration of cloud resources in multi-cloud environments while focusing on their elasticity nature. We, therefore, separate research issues into three areas: 1) Interoperable management of cloud resources, 2) Interoperable orchestration of cloud resources, 3) Multi-cloud elasticity of cloud resources.

1.2.1 Interoperable management of cloud resources

Nowadays, exploiting resources from multiple clouds has become a natural choice for enterprises that seek to provide applications with high availability and rapid recovery [47]. Achieving this requires seamless management of the underlying cloud resources that could be involved during the full application life-cycle. Nevertheless, this is far from trivial mainly due to the inherent heterogeneity and incompatibility issues that make cloud interoperability hard to achieve. Indeed, the use of diverse cloud providers at the same time poses a great complexity because of the lack of standardized APIs as each provider insists on its own APIs and methods to manage cloud resources. As mentioned above, using open standards is a definitive requirement to avoid such issues.

To this end, there are several notable standardization bodies that have been initiated with the aim of addressing cloud interoperability from a management perspective.

Prominent and widely adopted examples include: Open Cloud Computing Interface (OCCI), Cloud Infrastructure Management Interface (CIMI), Cloud Data Management Interface (CDMI), and Open Virtualization Format (OVF). Because of its ability to support all management tasks that can be related to any kind of cloud resources (IaaS, PaaS, SaaS), OCCI has been considered as the most comprehensive and important standard compared with the other ones that target limited management aspects and resources [109]. More specifically, OVF is devoted for packaging and distribution of virtual appliance (VA) that is composed of one or more virtual machines in a vendor-independent format. Whereas, CIMI concerns the management of resources within the IaaS layer. Finally, CDMI addresses interoperable management of cloud storage. However, it should be noted that OCCI is known by its compatibility with some of these standards especially OVF and CDMI [159].

As a matter of fact, OCCI introduces a RESTful protocol and a unified API. The main purpose is to provide a set of interoperable tools for common management tasks including creation, deployment, autonomic scaling, and monitoring. To achieve this, the management API has to be well-designed and consistently implemented so that it can hide the heterogeneity and evolution of the managed cloud resources across various providers while providing unified and efficient access to them. Therefore, OCCI in its different specifications insists on a set of guidelines for a compliant creation of these APIs. These guidelines represent the recommended best principles that have to be carefully followed by the API designers to provide a uniform way to manage cloud resources. Nonetheless, since they are specified in natural language, these principles are imprecise, ambiguous and may be interpreted differently. Therefore, this can lead to the non-compliance or poor adoption of such principles in current cloud resource management APIs and thus to interoperability issues. In addition, OCCI basically relies on the REST architectural style (Representational State Transfer), the de facto standard adopted by the most modern web applications for the interaction and manipulation of the underlying software components [129]. Moreover, in a thorough and systematic study [129] on OCCI and other cloud APIs, it is demostrated that OCCI fails to support some good practices related to the REST APIs design (follows only 56% (41/73) of the REST best practices). Consequently, this often worsens the quality of REST APIs and makes them hard to understand and use by developers, especially within a complex domain like cloud computing. Interestingly, both understandability and reusability are among the main quality characteristics whose reachability requires following the common REST best principles in the design and development of REST APIs.

To this end, OCCI and REST best principles have to be followed together in the design of cloud standard-based management APIs in order to ensure the desired interoperability on one hand and increase their reusability and understandability on the other hand. Most importantly, cloud APIs developers have to follow the good practice of these principles and avoid as much as possible their poor practice. In software engineering, the good practice of a given design principle is referred to as

Pattern, while its poor practice is known as an ***Anti-pattern*** [17]. In this thesis, we will often use the term ***(anti)pattern*** when denoting both pattern and anti-pattern. In fact, with the increasing number of operations and resources in cloud management APIs, their design tends to be very complex. In addition, these APIs are always subject to change due to the continuous evolution of target cloud resources. This can mislead cloud designers during the REST APIs design, which in most cases leads to introducing deviations or what is known as anti-patterns. Nevertheless, the presence of anti-patterns in the designed cloud APIs is a sign of non-respecting one or more best design principles intended to ensure interoperable and high-quality APIs. Thus, the detection of these anti-patterns in cloud REST management APIs is crucial to facilitate their correction and to promote their evolution while being aligned with the best REST and OCCI best design principles. Furthermore, detecting patterns, i.e. the good practices of best principles, is needed as well as it allows cloud designers to know to what extent their APIs are compliant with these best design principles. However, these patterns and antipatterns have to be carefully identified and formally specified to promote their automated detection as the manual interpretation of their presences is undoubtedly tedious and laborious.

So far, several studies have been proposed to advocate the use of (anti)patterns in the assistance of the design and development of software systems compliant with certain design principles. However, most of them have focused on service-oriented architectures and object-oriented systems. Although (anti) patterns specification and detection approaches have been proposed in the context of REST, they all target general-purpose APIs like Facebook and Twitter or are concerned with mobile and networking applications. Therefore, they cannot be applied to RESTful APIs developped for cloud services or resources. In addition to REST best principles, these APIs are characterized by other principles that relate to the structure, definition, and management of cloud resources. As mentioned above, we intend to identify these principles from OCCI to ensure the management interoperability between different providers.

In view of these limitations, our first objective is to assist the design of interoperable management APIs. We also aim to explore whether the current management APIs follow the (i) REST best principles devoted to improving their reusability and understandability and (ii) OCCI best principles devoted to ensuring interoperable management of cloud resources. In doing so, we adopt (anti) patterns as means to reflect the poor and bad practice of these principles. By detecting these (anti) patterns in Cloud managment Restful APIs, we intend to provide cloud API designers with an evaluation method that quantifies the compliance degree of their APIs with these best principles. To have a better compliance degree, we aim to help API designers in revising their APIs by suggesting a set of correction recommendations that serve to avoid anti-patterns. With the aim of addressing this research need, we have to answer the following questions:

- How to adopt (anti)patterns to assist the design of interoperable management

APIs?

- What are the (anti) patterns that have to be considered to reflect these best design principles?
- How to formally specify these (anti)patterns for supporting their automated detection?
- How to exploit the (anti)patterns detection in the compliance evaluation of Cloud management APIs with respect to the best design principles?
- How to provide the correction recommendations to avoid anti-patterns?
- How efficient our approach is in terms of results quality, accuracy and usefulness?
- Do current APIs follow these best design principles and to what extent?

1.2.2 Interoperable orchestration of cloud resources

According to several important surveys [155], [135], [71] [134], cloud orchestration plays an important role in allowing enterprises to meet their changing business requirements. This is due to the fact that orchestration encompasses all phases in the typical lifecycle of cloud applications as well as their underlying resources. These phases are the description, selection, configuration, deployment, monitoring, and runtime control of cloud resources in order to ensure a successful hosting and seamless delivery of applications. Unfortunately, today's enterprise applications have become increasingly more complex, involving the composition of multiple and heterogeneous components. These components are often powered with different virtualization technologies and distributed across different layers (SaaS, PaaS, IaaS) and multiple clouds. Because of this, supporting the entire orchestration process of these applications is becoming a too cumbersome and time-consuming task.

Moreover, with the growing importance of orchestration, several DevOps solutions (Tools and platforms) have been proposed. Prominent examples that are already adopted in the cloud community include: Docker, Kubernetees, Terraform, Puppet, Juju, Ansible, Chef [155]. While they are powerful in terms of providing promising configuration, management and monitoring facilities that would strengthen orchestration automation to a certain extent, there are still some open issues to be addressed. In particular, one DevOps solution is typically not able to provide all the management features that can be involved during the lifespan of cloud resources. In this manner, orchestrating the whole life cycle requires DevOps users to use multiple solutions. However, each one of these solutions relies on its own and proprietary approaches, this includes resource description models, management capabilities, access tools and interfaces. Therefore, using best-of-breed DevOps solutions in an integrated way while maintaining an acceptable level of interoperability for orchestrating applications is a

difficult and costly task if not an impossible one. This is because it implies extensive expert-driven manual efforts and requires sophisticated programming skills. Moreover, considering that DevOps users want to exploit cloud resources from multiple clouds, the issue only increases several-fold.

As we clearly stated above, open standards are the best candidates to avoid most of these issues. In the context of orchestration, the notable standard is the OASIS TOSCA [116], which is estimated to be more adopted by the cloud community in the next-generation IT solutions. The strength of TOSCA lies in providing portable, reusable and higher-level descriptions of cloud applications along with the required management features. This is playing an important role to facilitate the end-to-end orchestration tasks of cloud applications while maintaining a high level of interoperability between multiple clouds. However, TOSCA has focused only on the modeling perspective of applications and their orchestration aspects, without proposing any implementation language or appropriate mechanisms to automate their creation on top of cloud infrastructures. Instead, it leaves all implementation concerns to interested providers and users. Moreover, there is a clear lack of software solutions that support holistic orchestration automation driven by TOSCA. Proposing from-scratch solutions is tedious and unnecessary, especially with the presence of a huge variety of powerful DevOps approaches and tools that are in continuous evolution. In this thesis, we support the idea of integrating TOSCA with DevOps solutions as an essential step toward alleviating the above issues from both sides. In fact, considering the benefits of each of them, this would streamline the orchestration process of cloud resources while maintaining both the desired interoperability and efficiency.

So far, there has been a handful of approaches that address this integration need. Some of them adopt a low-level and ad-hoc programmatic approach. However, this may be time-consuming as it requires a considerable development effort and perpetuates continuous patching. Few others have provided concepts mapping between TOSCA and some DevOps solutions as a step to support the automated generation of DevOps-specific artifacts that are needed at the orchestration stage. This is useful as it allows taking advantage of the Everything as code, an innovative IT paradigm endorsed by most DevOps solutions, which insists on capturing as much as possible all aspects-related to orchestration such as configuration, environment, and infrastructure descriptions via human-readable artifacts written in YAML, JSON or any domain-specific notation. Advantages include greatly improved automation and repeatability, reduction of errors, reusability, acceleration of the delivery pipelines and more. Despite that, under these solutions, the mapping is provided at a low level of abstraction by writing transformations using programming languages. However, as the programming languages are inherently general-purpose, they lack appropriate high-level constructs (or abstractions) to easily encode transformations [141]. This makes these transformations hard to write, comprehend, maintain and extend when needed [141].

In view of these issues, our second objective is providing mechanisms to support

the integration between TOSCA and DevOps solutions. Unlike the previous solutions, we advocate providing the concepts mapping between TOSCA and DevOps at a high level of abstraction as an essential step to support a seamless integration at an acceptable cost. Indeed, this is required to automate the generation of DevOps-specific artifacts, which allows benefiting from the Everything-as code paradigm and its related practices. Furthermore, to support the orchestration process, the generated artifacts require to be executed by diverse DevOps access APIs/Tools, which are unfortunately heterogeneous and require extra effort to deal with them. To address this research problem, we need to answer the following questions:

1. How to support the mapping between TOSCA and DevOps solutions at a high level of abstraction?

2. How to exploit this mapping toward the automated generation of the underlying DevOps-specific artifacts required at the orchestration stage?

3. How to avoid the heavy lifting involved when interacting with diverse DevOps API/tools to execute these artifacts.

1.2.3 Multi-cloud Elasticity of cloud resources

To ensure the final enterprise's business objectives, the orchestration of their cloud applications must be completely aligned with the desired QoS objectives while ensuring an optimized operational cost. Maintaining a high-level elasticity of the underlying cloud resources at runtime control is the key enabler for ensuring this QoS-to-cost trade-off. The principle of elasticity lies on ensuring on-demand (de-)provisioning or reconfiguration of the cloud resources in such a way that the over-utilization issues that may cause unexpected cost and under-utilization issues that degrade the desired QoS, are avoided as much as possible. Basically, elasticity is achieved through the invocation of actions (e.g., add storage capacity, increase number of servers) that run as a result of events (e.g., service usage increases beyond a certain threshold), allowing a controller to automatically configure or re-configure cloud resources. However, handling elasticity poses a great complexity and is still in the early stages. In particular, for the powerful support of elasticity, special attention must be paid to the new requirements emerged as a consequence of the widespread adoption and continuous evolution of cloud computing.

Firstly, at design time, when defining elasticity policies it has to consider that elasticity can concern diverse type of resources (infrastructure, platform, and application), can span across multiple clouds (as the underlying resources can be provided by multiple providers), can have diverse reconfiguration mechanisms and can be applied in diverse virtualization technologies (especially: container and virtual machine-based). Secondly, at runtime, controlling this kind of elasticity imposes both efficient and extensive monitoring and powerful controlling. Monitoring must be able to gather the

operative data despite the variations of sources and technologies. Whereas, controlling should have the ability to execute required actions as fast as possible avoiding human-based interactions and without introducing unwanted overhead or leading to unexpected system oscillations. Hence, having a holistic solution that supports all these requirements is essential as each one has a big importance for ensuring the intrinsic elasticity goals.

However, despite its critical contribution in ensuring QoS-to-Cost trade-off, managing cloud elasticity while supporting the above key requirements is still not yet mature enough. Mainly, most solutions proposed by market-leading providers, DevOps community and researchers are dedicated only to one provider. Therefore, to dynamically support multi-cloud elasticity, cloud users have to develop their custom programs based on the existing procedural programming solutions of DevOps and providers. In addition to that, they have to deal with multiple re-configurations and monitoring APIs to enforce the elasticity over their cloud resources. This proves to be an increasingly too challenging and time-consuming task as it requires a considerable development effort and multiple and continuous patches. Moreover, this problem is made even worse as the variety of cloud resources and the variations of application resource requirements and constraints increase [93, 134]. Furthermore, under cloud standards like TOSCA and OCCI, elasticity features are not supported. Both of them lack suitable concepts for defining how cloud resources can be configured at runtime as a result of workload variations. So, it is obvious that managing multi-cloud elasticity remains a deeply challenging problem.

In view of these issues, our third objective is supporting the management of multi-cloud elasticity. A more effective solution should allow users to specify elasticity regardless of the technical specifications of any cloud provider or DevOps solution. Instead of relying on low-level and procedural mechanisms or provider-specific rule engines, we argue that models and languages for describing cloud resources should be endowed with intuitive and automation-friendly abstractions that can be used to specify a range of enforceable and flexible elasticity mechanisms in accordance with high-level policies specified by users. To address this research problem, we need to answer the following questions:

1. What is the set of abstractions that can reflect the different elasticity-related features?

2. How can these abstractions be exploited to effectively manage elasticity despite the heterogeneity of the involved providers, DevOps enforcement mechanisms and dynamicity imposed by the cloud environments?

1.3 Objectives and contributions

In light of the previously described challenges and shortcomings, the main objectives of this thesis are summarized as follow:

- provide guidance and assistance in the design of interoperable management APIs
- streamline and improve the orchestration of cloud resources
- support a high-level management of multi-cloud elasticity

To ensure the first objective, we leverage the notions of patterns and anti-patterns to drive respectively the good and poor practices of OCCI and REST best principles that API developers or cloud providers should be taking carefully on when designing their APIs. Patterns are good solutions to recurring design problems while Anti-patterns are the poor ones. Both of them are chosen due to their important contribution in improving the design quality and easing the evolution and maintenance of software systems. Then, we propose to provide the automated detection of these (anti) patterns as well as recommendation support to guide cloud developers in revising their management APIs. This would assist in the design of cloud management APIs that are compliant with both OCCI and REST, which renders them interoperable, understandable, and reusable.

To ensure the second objective, we propose to seamlessly integrate TOSCA with the best-of-breed DevOps technologies. In fact, we realize that the key difference between them is caused by using diverse models, entities, and languages to describe cloud resources and implement the underlying orchestration operations. Therefore, we propose to provide the required conceptual mapping between these two approaches, which would be the foundation to enable their integration in an automated and seamless way. Furthermore, we advocate support based on model-driven and transformation languages, the key principles of Model-Driven Engineering (MDE), to ease and manipulate this mapping at a high level of abstraction.

To ensure the third objective, we realize that the elasticity features should be provided at the resource description level instead of relying on low-level, scripting and technology-dependent mechanisms. More precisely, we propose a set of abstractions with which cloud elasticity features can be described in an intuitive and easy way, without referring to any provider/DevOps specific enforcement mechanisms. Besides, for a broader application of these abstractions, we propose to describe them independently from the cloud resource representations (i.e. the ones provided within the current resource description languages) and without being tied to any specific implementation languages. This will ease the integration of our elasticity solution with any orchestration solutions that follow any standard like TOSCA or domain-specific notations like Docker, Kubernetes, Terraform and so one.

This thesis work intends to meet the above objectives by proposing three contributions.

1. The first contribution proposes a semantic-based approach for defining and detecting REST and OCCI (anti)patterns and providing a set of correction recommendations to comply with both REST and OCCI best principles. We analyze both the literature and the OCCI standard with the aim of identifying the

set of REST and OCCI (anti)patterns. We provide a formal specification of OCCI and REST patterns and anti-patterns using an ontological model. The proposed ontologies include the most important and relevant concepts needed for the detection and recommendation purposes. Moreover, we introduce four detection algorithms acting on this specification to detect OCCI (anti)patterns and REST (anti)patterns respectively. Also, in case of any anti-pattern detection, they provide developers with a set of correction recommendations to help them revise and correct their APIs. To validate our work, we developed a proof of concept implementation called *ORAP-Detector* providing the detection and recommendation support. Finally, using this tool, we evaluated our approach by analyzing both OCCI and REST (anti) patterns on a real dataset that includes five cloud RESTful management APIs: OOi, COAPS, OpenNebula Amazon S3, and Rackspace, and assessing its feasibility in terms of accuracy and usefulness.

2. The second contribution introduces a model-driven approach for streamlining and improving the orchestration of cloud resources. We adopt the TOSCA standard for blueprinting all modeling artifacts related to cloud resources and their orchestration in a technology-independent manner. Moreover, to ease the integration of TOSCA with DevOps technologies, we propose a methodology that enables the automated translation of high-level TOSCA artifacts to underlying DevOps-specific artifacts. Further, we propose a set of high-level connectors acting as a DevOps abstraction layer to automate the end-to-end orchestration tasks while exploiting the obtained specific artifacts. For assessment purposes, we developed a proof-of-concept ***ToDev***, an integrated and standards-driven orchestration framework based on **TO**SCA and **Dev**Ops technologies. The framework includes open-source DevOps tools like Docker, Terraform, and Kubernetes and is empowered with MDE facilities to manage the involved models and transformation algorithms. Finally, to ensure the feasibility of our approach, our framework was used in diverse cloud use cases and was evaluated using two experiments to show in particular its transformation performance as well as its gained productivity in comparison to existing DevOps solutions.

3. The third contribution proposes a new **c**loud **E**lasticity **D**escription **M**odel (called cEDM) to describe cloud elasticity at a higher level of abstraction. Our model is based on a set of new abstractions, among them, the concept of resource requirement State Machines (C-SM). We devote this novel abstraction for capturing the elasticity-related behavior by characterizing the resource requirement variations over time and different phases of the application life-cycle. In the proposed model, states characterise application-specific resource requirements (e.g., CPU and storage usages), when they are needed. Transitions between states are triggered when certain conditions are satisfied (e.g., a temporal event, application workload increases beyond a certain threshold). Transitions automatically trigger controller actions to perform the desired resource (re-)configurations to

satisfy the requirements and constraints of target states.

Once defined, The C-SM can be used to manage and control the related elasticity behavior. For achieving this, we propose to translate C-SM into Event-condition action rules. These ECA rules could be then executed by any rule engine. Motivated by its benefits in terms of expressivity and performance, we opted for the Drools rule engine. Furthermore, we proposed an integrated monitoring system by taking advantage of diverse and powerful open-source DevOps systems for effectively collecting and analyzing events across different layers and heterogeneous clouds. The proposed monitoring system will work in tandem with the Drool engine to provide an appropriate runtime environment to manage all elasticity aspects while mitigating its related challenges. All these solutions have been prototypically implemented and integrated into a proof-of-concept called cEDMCore, an integrated system for high-level management of multi-cloud elasticity. Using the developed system, two experiments, one of which is with academics and professionals, are conducted to demonstrate the effectiveness of our approach in comparison to existing solutions.

It is noteworthy that the proposed solutions and mechanisms in contributions 2 and 3 could be used separately or combined together toward providing a holistic and integrated system for an elasticity-aware orchestration of cloud resources in a multi-cloud environment. For instance, an interesting part can leverage our ToDeV framework to support the basic orchestration operations of cloud applications that include the selection, initialization/configuration, Deployment/ Undeployment. Moreover, using the cEDMCore system, it can support the advanced runtime orchestration operations, in particular, monitoring and elasticity controlling.

1.4 Publications

Journal Articles

1. Hayet Brabra, Achraf Mtibaa, Walid Gaaloul, Boualem Benatallah: Toward higher-level abstractions based on state machine for cloud resources elasticity. Information Systems Journal, 2019 .

2. Hayet Brabra,Achraf Mtibaa, Fabio Petrillo, Philippe Merle, Layth Sliman, Naouel Moha, Walid Gaaloul, Yann-Ga el Gueheneuc, Boualem Benatallah, Faiez Gargouri, On Semantic Detection of Cloud API (Anti)Patterns, Information and Software Technology 107 : 65-82 (2019)

Conference Proceedings

1. Hayet Brabra, Achraf Mtibaa, Walid Gaaloul, Boualem Benatallah, Faiez Gargouri, Model- Driven Orchestration for Cloud Resources, 12th IEEE International Conference on Cloud Computing, CLOUD 2019 , Milan, Italy, July 8-13.

2. Hayet Brabra, Achraf Mtibaa, Walid Gaaloul, Boualem Benatallah, Model-Driven Elasticity for Cloud Resources, In 30th International Conference on Advanced Information Systems Engineering , CAiSE 2018:187-202, Tallinn, Estonia, June 11-15. **Distinguished paper award**

3. Farah Bellaaj Elloumi, Hayet Brabra, Mohamed Sellami, Walid Gaaloul and Sami Bhiri: Transactional Approach for Reliable Elastic Cloud Resources. In: 2019 IEEE International Conference on Services Computing, SCC 2019, Milan, Italy, July 8-13.

4. Hayet Brabra, Achraf Mtibaa, Layth Sliman, Walid Gaaloul, Boualem Benatallah, Faiez Gargouri, Detecting Cloud (Anti)Patterns : OCCI Perspective, In 14th International Conference on Service Oriented Computing ICSOC 2016 : 202-218,Banff, AB, Canada, October 10-13

5. Hayet Brabra, Achraf Mtibaa, Layth Sliman, Walid Gaaloul, Faiez Gargouri, Semantic Web Technologies in Cloud Computing : A Systematic Literature Review, In 13th International Conference on Services Computing, SCC 2016 : 744-751,San Francisco, CA, USA, June 27 - July 2

1.5 Thesis outline

This doctoral thesis is organized in seven chapters:

- **Chapter 2: Background** introduces the basic concepts related to our research and needed to understand the rest of the work. In this chapter, we first present cloud computing, its main service layers and its different deployment models. Then, we give definitions of elasticity and orchestration, followed by an overview of OCCI and TOSCA standards. Finally, we overview the main solutions that we adopted for supporting the main contributions of this thesis. These solutions include semantic Web technologies and model-driven engineering.

- **Chapter 3: State of The Art** provides an exploration and a thorough analysis of the state of the art around the three problematic of our research work. First, we present the work related to the use of (anti) patterns for software design assistance. Next, we explore the different attempts to facilitate Cloud resource orchestration. Finally, solutions that support cloud resource elasticity are also investigated.

- Chapter 4: **Assisting interoperable management APIs design using patterns and anti-patterns** presents our approach to assist cloud developers in the design of interoperable management APIs that are compliant to OCCI and REST best design principles.

- **Chapter 5: Streamlining interoperable orchestration of TOSCA with DevOps technologies using model-driven integration** introduces our approach for streamlining and improving the orchestration process of cloud resources.
- **Chapter 6: Managing multi-cloud elasticity using higher-level abstractions based on state machine** introduces our approach towards supporting the high-level management of multi-cloud elasticity
- **Chapter 7: Conclusion and Future Work** summarizes the proposed contributions and presents an outlook on the potential perspectives that we intend to tackle in the short-medium term.

Chapter 2

Background

Contents

2.1 Introduction

This chapter introduces the main concepts and background required for the understanding of the contributions described in the remaining chapters.

2.2 Cloud Computing

The adoption of cloud computing is now being considered an essential step by many enterprises for improving the quality of their services and maximizing their business outcomes. According to the National Institute of Standards and Technology (NIST) [112], one of the cloud leading references, Cloud computing is *"a model for enabling ubiquitous, convenient, and on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction"*.

Basically, cloud services are classified into three layers, namely infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) [112].

- SaaS layer includes software applications (customer relationship management, google apps, web conference) designed to be used by the end-users via web interfaces.
- PaaS layer provides a set of development tools and runtime environments facilitating the creation and deployment of new SaaS applications.
- IaaS represents the base layer providing a virtualized, distributed and automated IT infrastructure through a set of resources (servers, networks, storage) to meet the variable needs of the PaaS and SaaS layers.

Additionally, cloud services can be deployed according to three deployment models: public cloud, private cloud, hybrid or federated cloud:

- Public cloud: cloud services are deployed for open use by the general public.
- Private cloud: Cloud services are deployed for exclusive use by a single or a group of organizations.
- Hybrid or federated cloud: is an emerging deployment model, where computing resources can be obtained from one or more public clouds and one or more private clouds, combined at the behest of its users.

2.2.1 Elasticity

Elasticity is nowadays considered as one of the most essential features in cloud computing. In this section, we start by defining the notion of elasticity and then describe its main related features.

Elasticity definition There have been many definitions for elasticity in the literature. However, from our point of view, we define elasticity as *"the power of a system to dynamically reconfigure resources to adapt to varying resource requirements and constraints"*. Usually, it is achieved through the invocation of reconfiguration actions that run as a result of events, allowing a controller to automatically configure or re-configure cloud resources.

Elasticity features In order to provide a precise understanding of elasticity, its related core features have to be clarified. By examining the related academic papers and surveys on the cloud resource elasticity, we identified the following features: Scope, Purpose, Method or Action, Mode and Provider.

- **Scope:** specifies the target of elasticity, which can be the infrastructure (Virtual machine, Server, Storage, Network), platform (Container, Database, Runtime environment) or application (service, task or unit, etc.) resources.

- **Purpose:** Purposes for supporting elasticity are multiple. They include reducing the operational cost, maintaining the desired performance, ensuring availability, and saving the energy footprint.

- **Elastic methods or actions:** refers to the way the reconfiguration (or modification) takes to support the elasticity of cloud resources. We distinguish four main methods: horizontal scaling, vertical scaling, migration, and application reconfiguration.

  - Horizontal Scaling (HS): represents the possibility to scale out and in by adding or removing instances (e.g. virtual machine (VM), applications, containers, or Database).
  - Vertical Scaling (VS): in contrast to horizontal scaling, vertical scaling aims at scaling up and down fine-grained resources of the instance such as processing, memory, and storage rather than instances.
  - Migration: refers to moving a component from its initial location to another one. Migration can be applied at the VM level or application level. The former is referred to as VM migration which allows transforming a VM that is running on a physical server to another one. The latter is referred to as application migration which allows migrating only one of the application-specific components instead of the full VM, such as the database.
  - Application Reconfiguration (AR): consists in changing specific application aspects such DB cache size, DB recovery policy, etc. This is complementary to vertical scaling that only concerns attributes related to the capacity of the instance.

- **Mode:** indicates the way to execute elasticity actions. We distinguish two modes, which are: Manual and Automatic. The manual mode requires the intervention of the user who is in charge of the observation, decision and reaction phases in order to perform elasticity actions. On the other hand, the automatic mode means that the elasticity management process is automated, that is to say, that the system supports elasticity control without any external intervention. The implementation of the automatic mode is usually based on one of the following strategies:

  - Reactive: means that elasticity actions are triggered based on rules that indicate the system state in terms of workload or resource utilization. If one of these rules is satisfied, the elasticity controller triggers actions to accommodate the observed changes accordingly.
  - Proactive: triggering elasticity actions is based either on predefined schedules or on predictions that are defined using predictive techniques anticipating the future needs of the system.

  - Hybrid: combines both reactive and proactive approaches to execute elasticity actions.

- **Provider:** elasticity actions can be applied to resources from single or multiple cloud providers. A single cloud provider can be either public or private. Multiple clouds mean that cloud resources are obtained from more than one cloud provider.

2.2.2 Orchestration

Cloud resource Orchestration (CRO) can be seen as an automatic process of organization and coordination between different cloud resources (server, database, network, application, container, etc.). This process allows connecting cloud resources and managing them reliably and consistently with the aim of ensuring a successful hosting of cloud applications with the desired QoS.

Orchestration operations. The orchestration process encompasses four main operations [135], which are: selection, deployment, monitoring and runtime control of cloud resources. These operations are detailed in the following.

- ***Selecting resources (at design and runtime).***

  This operation allows application owners to select infrastructure and software resources. Software resources provide the functionality required while meeting resource requirements and constraints (interoperability with other resources, compatibility with material resources, cost, availability, etc.). Compatible infrastructure resources are then selected and allocated to these software resources.

- ***Deploying resources (both design time and runtime).*** It encompasses the creation and execution of different software resources (services or components) of the application, on a specific cloud platform. This is made possible by instantiating software resources on target cloud services and configuring them to allow communication and interoperability with other software resources. Connecting an application server with a database server is a representative example of this orchestration operation.

- ***Monitoring resources (runtime).*** This operation ensures the real-time monitoring of QoS attributes (known also as metrics). More specifically, it involves the detection of events (such as load peak) from the operation data generated by deployed resources (for example, application usage statistics). For this purpose, collecting metrics related to both PaaS (such as the container) and IaaS (such as the virtual machine) has to be ensured. The collected metrics will be exploited in the next operation which is the runtime control. In fact, metrics represent measures of resource utilization or behavior that can be aggregated and analyzed such as processor load, memory usage, and so on.

- ***Controlling resources (runtime)***. It relies on detected events to react and initiate corrective actions based on rules in order to ensure the proper functioning of the system. This operation is essentially based on a key mechanism which is the elasticity management. An illustrative example of a controlling operation is a vertical scaling of a database by changing its initial CPU resource configuration from small to large in order to enhance the throughput.

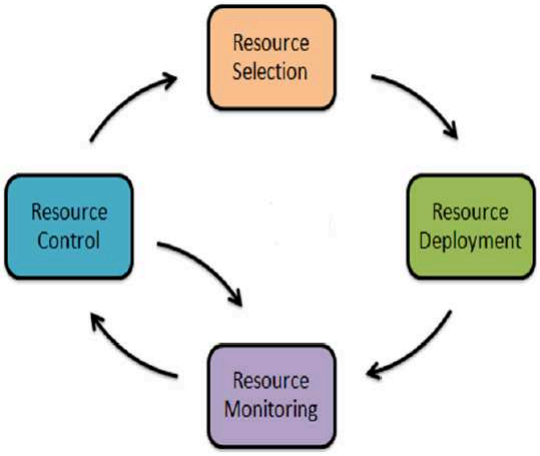


Figure 2.1: Cloud resource orchestration (CRO) operations in the life-cycle of an enterprise application (Source: [135])

Figure 2.1 depicts the flow of different orchestration operations and the coordination between them in the lifecycle of an enterprise application. The process including these operations begins with the selection, passing by the deployment and monitoring to end with the dynamic control of cloud resources. Based on the monitoring, if certain conditions are verified, certain elasticity actions could be triggered to maintain the proper functioning of the deployed application.

2.2.3 Cloud Standards

2.2.3.1 OCCI

In this section, we start by giving an overview of OCCI and then describe the main concepts underlying the REST architecture style.

OCCI is an open cloud standard [59] addressing heterogeneity, interoperability, integration, and portability in Cloud computing. More specifically, OCCI offers a RESTful Protocol and API that can act as a service front-end in the internal management framework of a cloud provider. Figure 2.2 demonstrates the OCCI's position

in this architecture. Here, a service consumer can be either an end-user or other system instance. The strength of OCCI lies in the fact that it is able to manage all kinds of resources which include IaaS, PaaS, SaaS, and potentially XaaS (Everything as a Service) from hardware resources to business applications while maintaining a high level of interoperability. With the aim of being modular and extensible, OCCI

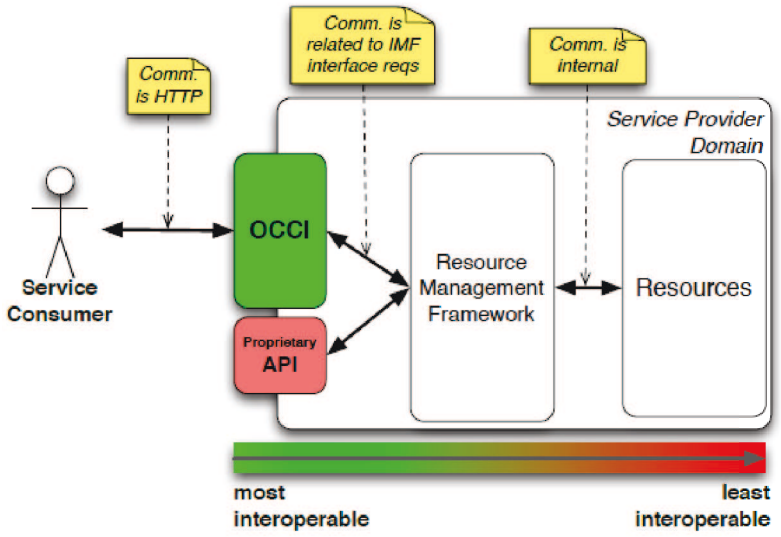


Figure 2.2: OCCI's position in a provider's architecture. (Source: [114])

provides a set of specification documents [118] describing four main layers: Protocols, Renderings, Core, and Extensions as illustrated in Figure 2.3. The OCCI Core

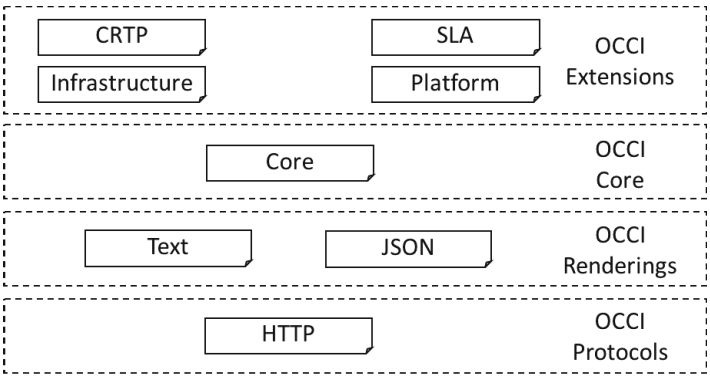


Figure 2.3: OCCI Specifications (Source: [102])

specification [114] defines a general-purpose resource-oriented model providing an abstraction of cloud resources, including the means to identify,classify, associate and

extend those resources. The UML class diagram illustrated in Figure 2.4 provides an overview of the OCCI Core Model. Accordingly, cloud resources are represented as

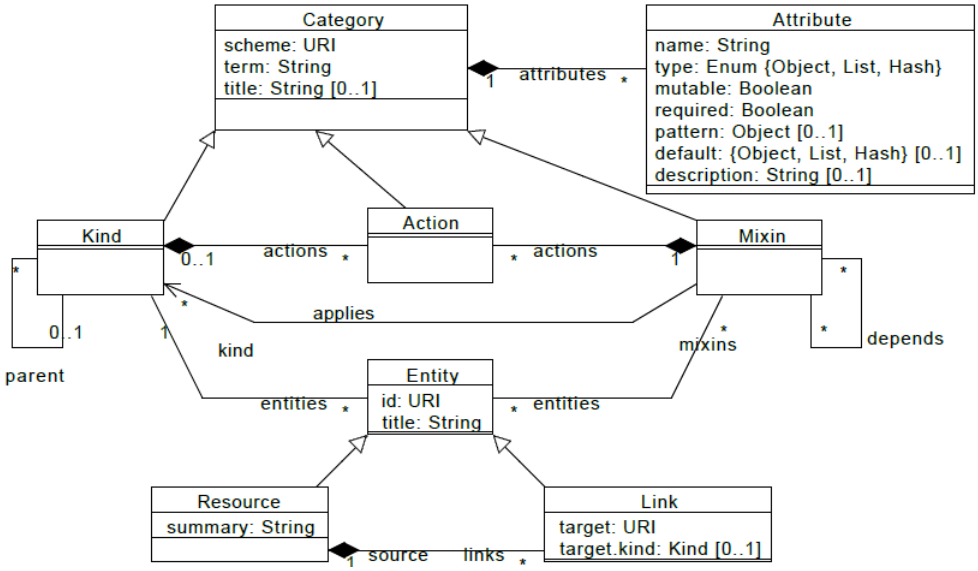


Figure 2.4: UML class diagram of the OCCI Core Model. (Source: [114])

instances of *Resource*. These resources can be linked to each other using instances of *Link*. Both *Link* and *Resource* are sub-types of the generic *Entity* concept. Each *Entity* can be typed using an instance of the *Kind* concept and can be extended by means of *Mixin* instances. *Kind* introduces additional resource capabilities in terms of *Actions*, which representing invocable management operations (e.g. create, stop, start, scale, etc.) that can be applied to resource instances. *Mixin* allows extending the OCCI entity by plugging in/out a set of attributes and actions. An instance of *Mixin* can be attached to any entity instance, which may provide additional capabilities both at creation time and/or run-time [114]. *Mixin* and *Kind* represent sub-types of *Category* which represents the basis of the type identification mechanism. Furthermore, the OCCI Core can be easily expanded using extensions, where each one provides a particular extension of the OCCI Core model describing a specific application domain. For instance, the OCCI Infrastructure [104] aims at abstracting the IaaS network, storage and compute resources. Figure 2.5 depicts the class diagram of these of OCCI Infrastructure types. Moreover, the OCCI Compute Resource Templates Profile (CRTP) [50] specifies a set of well-known instances of compute resources, such as `large`, `small` and `medium` computes. The OCCI Platform [105] defines PaaS application and component resources. The OCCI Service Level Agreements (SLA) [82] defines how SLA can be applied to OCCI resources.

The other OCCI specification includes renderings and Protocols representing together the way to interact with the OCCI core. Rendering specifications consist of

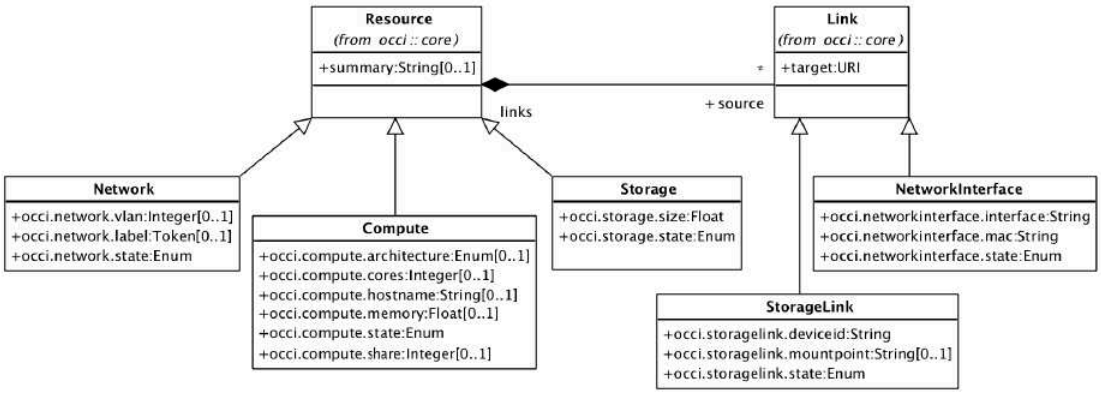


Figure 2.5: Overview Diagram of OCCI Infrastructure Types. (Source: [104])

multiple documents, each one provides a specific rendering of the OCCI Core Model. Currently, there are two rendering formats are provided, which are Text [58] and JavaScript object notation (JSON) [115]. Similarly, the OCCI Protocol specifications consist of multiple documents, each one describing how the model can be interacted with over a particular protocol (e.g. HTTP, AMQP, etc.). The current protocol that was provided by OCCI is HTTP [113], which describes a set of recommended best principles to create unified RESTful APIs for managing cloud resources. These principles form a minimal set to ensure interoperability. For instance, Figure 2.6 represents the textual description of the principle related to the creation of network resources according to OCCI.

Representational State Transfer (REST) REpresentation State Transfer (REST) was defined in the PhD thesis of Roy Thomas Fielding [65]. REST is an architectural style defining a set of rules for the design of distributed systems that assists the design and development of web applications. It is commonly used in the design of APIs for modern web services. Web services supporting properly the REST architectural style are called RESTful Web services and the application programmatic interfaces of these services are called REST APIs. Indeed, the basic constraints driving the REST APIs design are originally the result of architectural choices of the Web to reinforce the scalability and robustness of networked and resource-oriented systems that are based on HTTP. These constraints specifically include the following [65, 138]:

- *Resource addressability.* APIs manage and manipulate resources, which represent any information that can be named. Each resource is uniquely recognized through an appropriate Uniform Resource Identifier (URI).

- *Resource representations.* REST components apply actions on a resource through a representation that defines the intended or current state of a resource. Generally, the representations are associated with metadata such as content-types in the

Creation of Link resource instances: To directly create a Link between two existing resource instances the **Kind** as well as a **occi.core.source** and **occi.core.target** attribute MUST be provided during creation of the Link instance:

Example of compliant creation of Link
Operation request:
> **POST** /link/networkinterface/ HTTP/1.1
> [...]
>
> **Category:** networkinterface;
scheme="http://schemas.ogf.org/occi/infrastructure#";
class="kind";
> X-OCCI-Attribute: **occi.core.source**="http://example.com/vms/foo/vm1"
> X-OCCI-Attribute: **occi.core.target**="http://example.com/network/123"
>[...]
Operation answer:
< HTTP/1.1 200 OK
< [...]
< Location: http://example.com/link/networkinterface/456

Figure 2.6: Textual description about the principle that has to be respected when creating a link between resources, which is extracted from [103, 113]

headers of HTTP messages. This is done in order to allow clients and servers to correctly handle these representations.

- *Representation caching.* Caching is the capability to hide the resource representations with the aim of reducing network traffic between servers and clients, which may thus enhance the performance. Caching can be achieved at the client side or at any intermediate between clients and servers.

- *Uniform interface.* Resources are accessed and manipulated using the set of standard methods provided by the HTTP protocol, e.g., Get, Post, Put, Delete, Head, etc. Each method is conceived with its own expected, standard behaviour and standard status codes.

- *Statelessness.* The interactions between a client and a server should be stateless. In other words, each request must have all the required information to be understandable without the use of any stored information from the server.

- *Hypermedia as the engine of state.* Resources can be interrelated together using links. Links between resources are embodied in their representations. This enables clients to discover and navigate relationships and to maintain a consistent interaction state.

2.2.3.2 TOSCA

TOSCA stands for Topology and Orchestration Specification for Cloud Applications, which represents an OASIS standard. Its ultimate objective is to provide an interoperable and portable description of cloud applications with the aim of automating their deployment and management. TOSCA offers a structured XML-based/YAML language and a technology-independent metamodel for describing composite applications. We report here the main features of the TOSCA modeling language adopted in this thesis.

TOSCA describes a cloud application as a *service template*, which is in turn specified by means of a *topology template*. The latter is represented as a graph consisting of node templates and relationship templates. Node templates define the set of components that the application consists of. Virtual machines, Databases, and TOMCAT servers represent salient examples of application components. Relationship templates define dependencies between application components. A web application connected to a Mongo database could be an example of such a relationship.

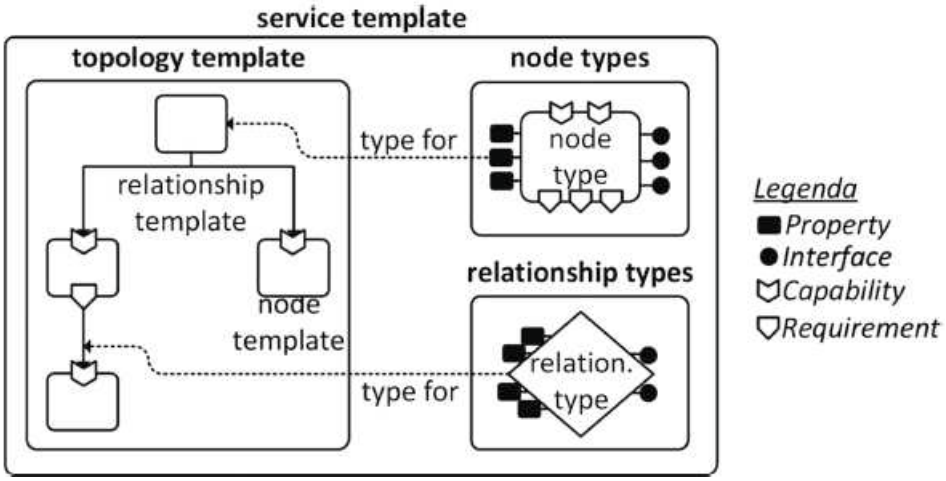


Figure 2.7: The TOSCA meta-model (source: [116])

Furthermore, node templates and relationship templates are assigned to certain types defining their semantics. A node template is typed with a *Node type* defining its main properties, provided capabilities, requirements, and interfaces. *Properties* are a set of attributes used to define the related node, e.g., a MySQL node has root_password and port as *properties*. *Capability* denotes certain functionalities that the node template provides. *Requirement* indicates that the node template may need a certain capability to be fulfilled by another node template. For instance, a Java application node may expose a Java Servlet Runtime requirement, whereas the Apache Tomcat node provides a matching Java Servlet Runtime capability. Concerning the *Interface*, it defines the required management *Operations* (e.g., install, configure, etc.)

that may be applied on a node or relationship tempale to manage its life cycles. Here, it should be noted that requirements, capabilities, interfaces, and operations are also typed with reusable types allowing the specification of core properties characterizing these entities. Moreover, a *relationship type* describes the basic properties of a relationship. In addition, node templates and relationship templates are also associated with artifacts that are required for their deployment and the implementation of their interface management operations. *Artifacts* represent the types of packages and files that can be used by the orchestrator for deployment and implementation purposes. Executable scripts, images, and configuration files are frequent examples of artifacts.

Listing 2.1 demonstrates the typology template for installing WordPress application using the TOSCA YAML-based language. The described application is composed of two software nodes, namely *wordpress* and *mysql* and one compute node named as *app-server*.

```
1
2  tosca_definitions_version: tosca_simple_yaml_1.2
3  description: TOSCA simple profile with wordpress and mysql.
4  node_templates:
5      wordpress:
6        type: tosca.nodes.WebApplication.WordPress
7        properties:
8           port: 8080:80
9        requirements:
10          - database_endpoint: mysql
11          -Host:app-server1
12       interfaces:
13           Standard:
14              Create:
15                  implementation:
16                      primary: Scripts/WordPress/install.sh
17     mysql:
18         type: tosca.nodes.Container.Application.Docker
19         properties:
20          port: 3306
21         mysqlimage:
22              file: mysql:8.0
23              type: tosca.artifacts.Deployment.Image.Container.Docker
24              repository: docker_hub
25        requirements:
26          - host: app-server2
27
28       interfaces:
29           Standard:
30           configure:
31              implementation:
32                    primary: mysqlimage
33
34    app-server:
35       type: tosca.nodes.Compute
36       capabilities:
37         Host:
38              properties:
39                  disk_size: 10 GB
40                  num_cpus: 2
```

```
                mem_size: 8 GB

        OS:
            properties:
                architecture: x86_64
                type: Linux
                distribution: ubuntu
                version: 12.s
```

Listing 2.1: A TOSCA YML Template for a Wordpress application installation

2.3 Adopted solutions

2.3.1 Semantic Web technologies

The Semantic Web coined by Tim Berners-Lee has experienced growing attention from both academic and industrial areas. The main objective is to elevate the meaning of Web information resources and make them easily readable by machines through certain key technologies [127], among which, ontologies represent the cornerstone technology. According to Gruber et al., an ontology is defined as *"An explicit specification of a conceptualization of a given domain'* [74]. This definition is one of the most quoted definitions in the literature, where the term *"conceptualization"* refers to an abstract model of a certain domain and which identifies the relevant concepts of this domain. The term *"explicit"* means that the type of concepts used and the constraints on their use are defined in a clear and precise manner.

Generally, the use of an ontology is justified by the objectives that it ensures. More precisely, an ontology provides a rigorous organization of the knowledge necessary to achieve a comprehensive understanding of all the relevant elements in a given domain. Thus, it facilitates the sharing of knowledge and its reuse. In addition to that, an ontology promotes the possibility of inferring and reasoning on knowledge through reasoning languages and techniques. Indeed, in this thesis, we define a set of ontologies providing together a semantically-enriched description of (anti)patterns that target the design characteristics of RESTful APIs devoted to cloud resource management. The proposed ontologies provide knowledge to facilitate the detection of (anti)patterns in REST APIs as well as the recommendation of corrections to assist developers in revising their APIs.

For an ontology to be understandable and exploitable by software programs, it has to be described by an ontology representation language. In our thesis work, we opted for the Ontology Web Language (OWL) language to formalize and implement our ontologies. This choice is due to the fact that OWL currently represents the most complete and adopted language in the literature. Besides, to enable the reasoning capability which is required for discovering new knowledge, the obtained OWL representation has to be enhanced with the so-called inference rules, whose definition relies mainly on rule languages. Among the important rule languages, we cite the semantic Web rule language (SWRL) and its query-enhanced rule language (SQWRL). The

modeled ontology and defined rules build together a knowledge base that can be later interrogated by means of query languages to perform the desired objectives. Simple Protocol and RDF Query Language (SPARQL) represents the notable and most used language enabling knowledge querying. In the following, we first start by introducing the main components used for building ontology knowledge. Next, we present the languages cited above for ontology modeling, reasoning, and interrogation tasks.

Basic ontology components Knowledge in an ontology is formalized using five main components namely: class, property, relation, instance, and axiom.

- *Class:* is known also as *Concept*, which represents a set of objects and their common properties. It is the basic unit for the vocabulary description used in ontologies.
- *Property:* is commonly known as *Data Type Property*, which represents a characteristic or attribute of a class.
- *Relation:* is commonly known as *Object Property*, which represents an association between two classes named domain and range of the relation.
- *Instance:* called also *Individual*, which designates a concrete object of a domain. It represents an instantiation of a concept. An instance is described by its membership in a given class and by a set of values assigned to the properties of this class.
- *Axiom:* represents a logical assertion that must always be true. Axioms allow combining concepts, relationships, and logic functions with the aim of defining rules that can be used either for checking the ontology consistency or inferring new knowledge.

OWL modeling language. OWL is introduced to enrich the resource description framework schema language by providing a more complete vocabulary able to describe complex ontologies. It is inspired by the description logic, the syntax of RDF triples, and some elements of RDFS. The description logic is a formal language allowing both the representation and inference of knowledge. RDF (Resource Description Framework) is a Semantic Web language recommended by the World Wide Web Consortium (W3C) with simplified semantics. RDF allows to formally represent Web resources and the relationships between them by means of triplets of the following form: *<subject, predicate,object>* where:

- The subject represents the resource to describe;
- The predicate represents a property type applicable to this resource;
- The object represents the value of the property.

RDFS is an RDF extension allowing the expression of semantic relations called predicates at the level of classes and properties by adding new constructors, namely:

- *rdfs: class* is used to define classes;
- *rdfs: property* allows to name relations between classes;
- *rdfs: domain* and *rdfs: range* allow to specify the source class and the target class for an object property;
- *rdfs: subClassOf* is used to define a hierarchy of classes;
- *rdfs: subPropertyOf* is used to define a property hierarchy;
- *rdfs: type* binds an instance to a class.

OWL inherits all the capabilities of all the above languages, making it able to create all the components of an ontology and have a great ability to represent web contents. Also, OWL integrates new constructors for comparing properties and classes, namely disjunction between classes (owl: disjointWith), equivalence (owl: equivalentOf), restrictions (owl: Restriction), cardinalities (owl: cardinality, owl: minCardinality, owl: maxCardinality), etc. In 2012, the second version OWL2 has been recommended by the W3C, which is the same version that we adopt in our thesis work.

SWRL and SQWRL rules languages. SWRL is a W3C standardized language that provides deductive reasoning capabilities by allowing users to define rules overs OWL constructs (class, relation, property, etc.). It is built on the same logical basis as OWL and can be extended through methods called built-ins. These methods are defined in the SWRL specification and can be basic mathematical operators and functions for string and date manipulations. One of the strong points in the SWRL language is its extensibility. In fact, the SWRL language allows users to create their personalized built-ins functions that will be integrated straightaway in SWRL rules. We already exploited such features in order to create our custom built-ins that are used in our semantic-detection approach.
$a_1 \wedge a_2 \wedge ...a_n \rightarrow b\ a_1\ ,\ a_2\ ,\ a_n$, *where*
Antecedent; b : *Consequent* ; *a and b are atoms*
$Atom \leftarrow C(i)|D(v)|R(i,j)|U(i,v)|built-In\ (p,\ v_1,\ ...,\ v_n)|i = j|i \neq j$
C =*Class*; D=*DataType*; R = *ObjectProperty*; U = *Data Type Property*
i, j = *Object variable names or Object individual names*
v_1,..., v_n = *Data type variable names or Data type value names*;
$p = Built-In$ *names*

The syntax shown above demonstrates how to express SWRL rules, which is based on two parties: *Antecedent* and *Consequent*. The *Antecedent* consists of one or more atoms with the aim of specifying the conditions that should be fulfilled. On the other

hand, the *Consequent* allows the definition of the impacts that can be produced once the conditions defined in the *Antecedent* are fulfilled. It often results in one or more atoms. An atom can be a class, an Object Property, a Data type Property or a Built-in. The latter can be SWRL built-ins (e.g., swrlb:matches) or query built-ins (e.g., sqwrl:select) provided by the SQWRL language which is known as an SWRL-based query language. The SQWRL language offers a set of SQL-like operators for retrieving data from OWL ontologies that are required for the definition of the SWRL rules. These operators (called also as SQWRL built-ins) are integrated into SWRL rules to make use of the inferred knowledge. The example below demonstrates a simple SWRL rule that classifies persons older than 17 as adults and lists them.

```
1
2 Person(?p) ^ hasAge(?p, ?age) ^ swrlb:greaterThan(?age, 17) -> Adult(?p) ^sqwrl
    :select(?p)
```

Listing 2.2: SWRL rule example

SPARQL language. SPARQL is a query language recommended by the W3C. It allows to express queries across diverse data models on the condition that the data is stored following the native RDF triplets. The example below demonstrates a simple query that exploits the ontology definition named *foaf* which stands for *friend of a friend*. The query returns the names and emails of every person in the related dataset:

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT ?name
3        ?email
4 WHERE
5   {
6     ?person  a          foaf:Person .
7    ?person  foaf:name  ?name .
8     ?person  foaf:mbox  ?email .
9  }
```

Listing 2.3: SPARQL query example

Here, the *PREFIX* is used to indicate the address (URI) of the data source model to be queried. *SELECT* extracts from the RDF graph related to the data source model a sub-graph corresponding to a set of resources that satisfy the conditions defined in the clause *WHERE*.

2.3.2 Model-driven Engineering

Model-driven Engineering (MDE) [144] is known as a software development methodology, where the first-class entities of the development process are models. In MDE, models represent an essential part towards a solution for a given problem, unlike programming methodologies in which algorithmic constructs are used to solve problems

and models are taken only as illustrations. From a theoretical point of view, a model is an abstract representation of knowledge related to a system from a particular domain, while from a practical point of view, it consists of a set of formal elements in order to describe system entities that are being developed for a given objective.

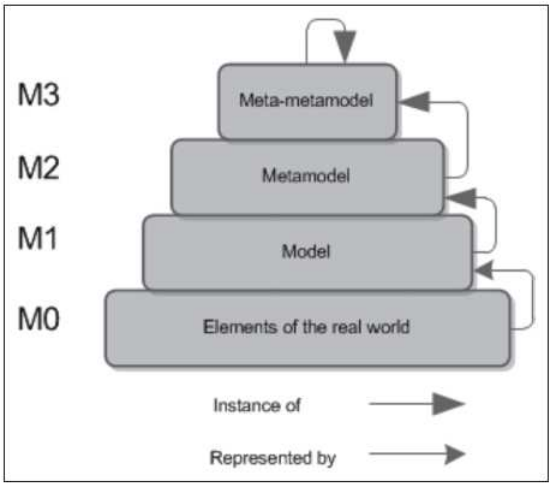


Figure 2.8: Four-layers architecture of MDE (source: [9])

To build models, it is required to define a modeling language. The latter is represented with other elements at a higher level of abstraction that would be the basis to create models. Thus, the modeling language can be seen as a model of the model, which raises the notion of *meta-modeling* [9]. In fact, MDE is based on a meta-modeling architecture with four levels. These levels are defined by the Object Management Group (OMG) as follows:

- M0 level (Reality). It specifies the real-world system. In this level, the created elements are the instances of elements at level M1.
- M1 level (Model). It represents the model of the system. Its main concepts provide the classifications of the elements at level M0. The model, in turn, is an instance of the meta-model defined at M2.
- M2 level (Meta-model). It defines the modeling languages of M1 elements
- M3 level (Meta-meta-model). It represents the most abstract level, where elements of the modeling languages can be defined. In this level, we find languages such as MOF [119] and Ecore [54]. These languages provide the constructors and mechanisms for describing metamodels for modeling languages, such as UML.

2.3.2.1 Model Transformations

Model Transformation is considered as the heart and soul of MDE as it offers a mechanism to enable the automation manipulation of models and generation of their corresponding implementation codes. There have been many definitions in the literature for model transformation. According to the OMG standardization, model transformation is defined as *"the process of converting a model into another model of the same system".* A more broad definition is proposed by Mens et al [6], in which the model transformation is considered as *" the automatic generation of one or multiple target models from one or multiple source models, according to a transformation description".* Here, a transformation description indicates how the source model elements are transformed into target model elements.

In literature, there are two main categories of model transformations according to the target type, which can be a text or model.

- ***Model to text (M2T) transformation.*** It translates a source model into a text file. M2T transformation is commonly used to enable the eventual generation of code or textual documentation. To enable M2T transformation, we distinguish two approaches [81], which are the following:
  - *Visitor-Based Approach.* is considered as a very basic code generation approach that provides a set of visitor mechanisms that is devoted to scan the internal representation of a model and write the corresponding code to a text stream.
  - *Template-Based Approach.* is a synthesis approach generating code from high-level specifications, called templates. Templates are abstract and reusable representations of text fragments embedded with meta-code. It comprises two parts, a static one representing the text fragments that will appear in the output, and a dynamic one representing the code that encodes the underlying generation logic.

  Compared to the Visitor-based approach, the template-Based approach requires less programming effort from the programmers to encode code generators. Because of its focus on abstraction and automation, it is a widely used technique in MDE and is supported by multiple languages, including xTend [5], Acceleo [2], Xpand [4], Jet [148], etc. In our work, we opted for xTend2, which represents a complete programming language relying on DSL providing a syntactical simplification of Java and integrated with Xpand. This choice is justified by the fact that Xtend2 is acknowledged by providing a good compromise between the expressiveness and the performance [95], [148].
- ***Model to model (M2M) transformation.*** It translates a source model into a target model, which can share the same or different meta-models.

This translation is enabled by mapping entities of the source model into entities in the target model. In general, three main approaches have been followed to support M2M transformation [141].

* *Direct model manipulation.* enables the manipulation of the source model using a set of procedural APIs to obtain the target model. One advantage of this approach is that the used language to manipulate the exposed APIs is a common general-purpose language like Java. Thus, developers do not require extra training to write transformations. However, the major issue of this approach is that it lacks of suitable high-level abstractions to specify transformations due to its general-purpose nature. This makes these transformations hard to write, comprehend, maintain and extend when needed [141].
* *Intermediate representation.* promotes the idea of exporting a source model into a standard form such as XML, XMI, etc, then transforming the exported model into the target model. This approach provides less expressive power and requires considerable effort to encode even simple model transformations [141].
* *Transformation language support.* provides a domain-specific language that offers a set of rich constructs to explicitly specify and compose the transformations. In our work, we rely on the transformation language support as it offers the most potential compared to other strategies [141]. In this respect, there exist many languages that can be leveraged to define and execute the model transformation. In our work, we selected the Query/View/Transformation Operational language (QVTo) [120], which is an imperative language, specified by the OMG standard. QVT allows model transformation by explicitly specifying the required steps and mapping rules in order to produce target models. QVTo is mainly designed to support a transformation that deals with target models of complex structures. This is the case with the description models of different DevOps solutions, which range from average (e.g., Docker, Juju) to complex (e.g., AWS Cloud Formation). Moreover, it offers a good compromise between expressiveness and scalability and has a reasonable performance.

2.4 Conclusion

In this chapter, we first introduced the basics of cloud computing. Then, we detailed the elasticity by studying its core features and orchestration by illustrating its basic operations. We also introduced the two cloud standards of our interest, namely: OCCI and TOSCA. We then presented different techniques that we will use in our contributions, namely: semantic Web technologies and model-driven engi-

neering. Throughout this manuscript, we use semantic web technologies to formalize required knowledge about (anti) patterns in cloud RESTful APIs. This knowledge will facilitate the automated detection of these (anti) patterns as well as the recommendation of appropriate corrections when needed. This approach is elaborated in chapter 4. We also use the key principles of model-driven engineering to manage the models and transformation algorithms that are involved in our second and third contributions presented in chapters 5 and 6, respectively.

Chapter 3

State of The Art

Contents

3.1 Introduction

In this thesis, we aim at providing solutions to support the design of interoperable management APIs, orchestration and elasticity of cloud resources in heterogeneous and multi-cloud environments. Therefore we examine, in this chapter, the existing work in the literature relevant to these topics. The goal is to position our work versus these solutions while highlighting the current challenges that justify our thesis directions. In doing so, we classify them according to three main research areas: (i) On using (anti) patterns for software design assistance, (ii) On facilitating Cloud resources orchestration, (iii) On supporting Cloud resources elasticity. For each one, we start by presenting the related approaches. Then, we provide a thorough analysis of the presented approaches to compare our work to these initiatives.

3.2 On using (anti) patterns for software design assistance

Over the last years, patterns and anti-patterns have been widely adopted by various researchers with the aim of expressing architectural solutions and concerns in diverse domains including Service-Oriented Architectures (SOAs), Object-Oriented Systems and more recently in cloud computing and RESTful applications. Many studies have been made to address different (anti) pattern concerns. Some efforts are concentrated on identifying and documenting (anti) patterns that relate to different software domains. Others proposed new models or adopted formalisms for providing a precise specification of (anti) patterns. (Anti) patterns detection and analysis approaches are also proposed either for assisting the design and development of diverse systems or evaluating their quality. Hence, in this section, we investigate the existing work on assisting the design of software systems using (anti) patterns and classify them into three categories: (i) (Anti) patterns identification, (ii) (Anti) patterns modeling and formalization, and (iii) (Anti) patterns analysis and detection. Then, Section 3.2.4 positions our work versus these solutions.

3.2.1 (Anti) patterns identification

(Anti) patterns identification consists of analyzing a real world system with respect to different dimensions to derive what is good that the system users have to follow and what is not that they must avoid. Dimensions including architecture, code and even how the system was documented are among the relevant factors that have been widely considered by numerous researchers. The goal is to define a catalog of (anti) patterns that will raise the software practitioners' awareness of good and poor practices on their system of interest, which therefore contributes to the improvement of system quality and comprehension. As there is a huge number of (anti) pattern catalogs, we will only consider the most relevant ones and those that were subject to diverse research analyses.

Initial identification efforts of (anti)patterns have begun with the object-oriented systems. Indeed, there are various books that introduce a number of OO (anti)patterns. For instance, Martin Fowler et al. in their highly-acclaimed [67] book introduced a set of 22 code smells representing the most low-level anti-patterns that exist in the source code. Another work [51] is related to J2EE-based applications suggesting 53 anti-patterns that address the biggest java issues with respect to their architecture, design, and implementation using technologies such as JSP, EJB, Servlet, and Web services. On the other hand, Arnaudova et al. [18] have dealt with the linguistic aspect of OO programming and suggested 17 linguistic anti-patterns.

Moreover, SOA is also as important as the OO paradigm. Thus, numerous online catalogs and books have been proposed introducing new patterns and anti-patterns that characterize "good" and "poor" service-oriented designs. For example, Rotem-

GalOz et al. [8] introduced in his book 23 SOA patterns and four SOA anti-patterns while explaining their causes, consequences and suggesting their corrections as well. Another book proposed by Erl [61] presents more than 80 SOA patterns that address design, implementation, security, and governance aspects. Another interesting work made by Rodriguez [139] in the context of web services, aimed at analyzing a set of WSDL descriptions in order to define the most common Web service anti-patterns. The proposed anti-patterns are mainly related to linguistic properties such as element names, types, and comments used for specifying the data models in WSDL documents. The aim is to enhance the descriptions of Web services toward ensuring an effective service discovery.

In the REST context, some identification studies have been conducted, introducing new (anti)patterns devoted to enhancing the quality of RESTful APIs as they are being increasingly adopted. An interesting study conducted by Petrillo et al. [129] compiled a catalog containing more than 73 best practices related to the RESTfull APIs design. The proposed best practices are gathered with the aim of improving API quality properties such as understandability and reusability. Thanks to their importance, these practices can be exploited to derive the most common REST patterns and anti-patterns. In fact, a number of online documentations [128, 150] have been already proposed by the REST specialists and practitioners to identify a list of REST (anti)patterns while discussing how developers might cause them at design-time. Also, Some books [46, 52] have introduced a catalog of design patterns related to Web service that follow the REST architectural style. The aim is to design services that comply with the REST constraints elaborated by Roy Thomas Fielding [65] in his thesis.

Recently, the emergence of cloud computing has led to identifying new (anti) patterns, most of which addressed the most common patterns in the cloud computing system design. More specifically, some online catalogs and books have been introduced both by cloud providers and academic researchers. In [63], the authors provide a pattern-oriented view on cloud computing by introducing five categories of patterns, namely Cloud computing fundamentals, Cloud offerings, Cloud application architectures, Cloud application management, and Composite cloud applications. All the patterns have been defined in an abstract form without referring to any concrete provider's technologies, programming languages, middleware or products. AWS [25], the leader cloud provider, proposed a catalog that includes more than forty patterns, representing a set of design ideas and solutions toward a successful use of AWS technologies in order to solve recurring design problems. Microsoft provider [106], on the other hand, introduced a set of twenty-four design patterns addressing diverse architectural concerns related to cloud-hosted applications. Finally, an interesting book in a catalog-like format proposed by Erl et al. [60], introduced a set of vendor-agnostic design patterns. These patterns are devoted to enhancing operational resiliency, runtime reliability, and automated recovery when interruptions take place.

3.2.2 (Anti) patterns modeling and formalization

Most of the identified (anti) patterns have been informally and textually described. This hampers the wide adoption of these (anti) patterns as well as their automated integration in the different lifecycle stages of software design and development. To overcome this limitation, various models and formalisms have been proposed (or adopted) to ensure a more structured, expressive and formal (anti)patterns representation. Thereby, this laid the first foundation stone for tool support. The first attempt goes back to Gamma et al [68], who proposed to use UML diagrams to describe each pattern. Others, including [91, 153] extended UML and used the OCL language to encode structural and behavioral properties related to patterns. In addition, different modeling langauges have been introduced, including: RBML (role-based metamodeling language) [86], DPML (design pattern modeling language) [99], and LePUS3 [70]. RBML relies on the notion of roles to represent design patterns at the metamodel level, where the role can be played by any model elements such as classes, methods, or associations. Using roles, RBML has the ability to represent various pattern perspectives, namely static structure, interactions, and state-based behavior. DPML offers a set of modeling constructs allowing the incorporation of design patterns within UML models. LePUS3 is a formal and visual language expressing design patterns based on a Z formula extended with graphical representations. It allows the specification of design patterns as well as the design of JavaTM programs at any level of abstraction. Although the proposed abstraction is interesting, its degree of expressiveness is too restrictive, for instance, there is no means to encode the relationship between the pattern and its instantiation.

None of the attempts mentioned above did provide any means to represent anti-patterns, the opposite of patterns. In addition, they are tailored to be used only for the specification of design patterns related to OO systems. So they can not support the new considerations that emerged with technologies such as SOA, REST and cloud computing. Consequently, Moha et al. [108], introduced a domain-specific language allowing the representation of SOA anti-patterns in service-based systems. The proposed language is based on new domain-specific vocabularies expressed using Backus-Naur Form (BNF) grammars. The introduced grammars allow specifying SOA anti-patterns in terms of metrics capturing their design static properties such as cohesion and coupling and also dynamic properties, such as QoS criteria.

In addition to previous modeling attempts, notable formalisms including temporal logic [151], Bayesian networks [142], and semantic web ontologies [143], have been adopted toward a more precise and formal (anti)pattern description. In [151], authors used the temporal logic CTL and its subclass LTL to formalize data flow-related anti-patterns in the workflow systems specified using a workflow net with Data (WFD-net), which represents a particular kind of Petri-net. The major advantage of this formalization is enabling the use of standard and powerful model-checking techniques during the discovery step of anti-patterns in WFD-nets. Moreover, [142] suggested the

adoption of bayesian networks to model software project management anti-patterns. The model was dedicated to model the cause-effect relationships that stem from anti-patterns while considering the uncertainties involved in software projects. The major benefit of using a Bayesian network is its inherent powerful analysis for managing uncertainty.

Other initiatives [13, 87, 143] have endorsed the use of semantic models, in particular, ontologies with the aim of providing a semantically enriched representation of (anti)patterns. Settas et al. [143] propose to formalize software management anti-patterns using an ontology model. The proposed ontology relies on the description logic (DL) formalism allowing the semantic formalization of anti-patterns and the possible relationships that can exist between them in terms of three main concepts namely symptoms, causes, and consequences. Kirasic et al. in [87] proposed two ontologies to create the required knowledge base for pattern recognition. The first ontology describes code patterns that can be found, while the second provides a taxonomy of involved OO programming concepts, including Variable, Statement, Constructor, Method, Type, etc. Thus, knowledge about patterns is separated from the programing concepts, thereby from the procedures of their recognition.

Generally, ontologies have been widely used to describe design patterns related to OO systems. Most of these solutions followed two conceptualization approaches. The first approach allows introducing one ontology for all design patterns, with special focus on their classifications and documentations. This degrades the full expressiveness of design patterns since it does not often consider the specificity of each one as well as its structure. Contrary to the first approach, the second approach devotes an ontology for each pattern. Thus, the knowledge base about all patterns is the result of importing all defined ontologies. Despite its importance compared to the first one, this approach may introduce significant redundancy and complexity in the obtained knowledge base. Moreover, contrary to the widespread consideration of patterns by the semantic approaches, anti-patterns are only rarely supported or not at all. By contrast, in our work, we pay the same attention to patterns as well as anti-patterns. Additionally. most of these approaches focus only on design patterns related to OO systems. To the best of our knowledge, there is only one effort [100] that proposes an ontology-based representation for agnostic and vendor-specific patterns. The proposed representation is used to perform the automatic discovery of cloud services and appliances. Finally, there is no semantic approach that addresses REST (anti)patterns.

3.2.3 (Anti) patterns analysis and detection

Regardless of which software domains they approach, analysis and detection of (anti) patterns are playing a crucial role in improving the quality of design and development. In the following, we present the different detection approaches from diverse domains (OO, SOA, REST and cloud computing systems) as they formed a sound basis of

expertise and technical knowledge for our detection method. However, with respect to analysis studies, we only focus on RESTful APIs.

Several works have been proposed to support the automated detection of (anti) patterns. In the context of OO systems, Kessentini et al [85] proposed a two-steps approach that intends to detect the potential design defects in the software code. The first step allows the generation of detectors that capture how the code can deviate from good practices. More precisely, a collection of code fragments originated from a set of the system considered as well-designed were leveraged to generate these detectors. The code fragments are normal code that is considered as reference. The detectors generation step has been considered as a search problem that its resolution follows a genetic algorithm. The latter seeks to optimize two main objectives, namely (i) maximizing the generality of the detector so that it able to covers as much as possible the deviations from the reference code (ii) and minimizing its similarity with the other detectors. Afterward, the second step can be applied which aims at assessing the risk by comparing the generated detectors with the selected code for the evaluation. A code fragment is evaluated as a risky element if it exhibits a high similarity with one of the detectors.

Another important effort has been made by [143]. It proposed a knowledge system based on OWL ontology called SPARSE which provides detection support for software project managers to detect anti-patterns during the software project management. Based on the proposed ontology, a set of inference rules specified using the Semantic Web Rule Language(SWRL) is defined to drive the implicit knowledge related to anti-patterns based on their correlations. This knowledge is structured in terms of new causes, symptoms, and consequences. Further, using the Pellet DL reasoner [145], the ontological knowledge is mapped into an object-oriented (OO) model that is specified in classroom object-oriented language (COOL) of the CLIPS production rule engine [111]. Thus, a set of OO production rules are applied to the obtained model in order to derive the required conclusions assisting project managers in the anti-patterns discovery and detection. While relying on these rules, two matching algorithms are proposed to support the anti-patterns detection. The first one allows the evaluation of real data coming from software projects against the defined symptoms in order to detect the possible anti-patterns. The second takes the matched anti-patterns as input to retrieve other anti-patterns that share with them the same causes and/or consequences.

In [108], Moha et al. revealed the lack of techniques and methods for detecting SOA anti-patterns in service-based systems (SBSs). As a remedy, they proposed an approach named SODA supported by the SOFA framework with the aim of supporting the automated detection of SOA anti-patterns. The SODA'goal is assessing the design of SBS as well as their QoS aspects. The authors relied on the SOA anti-patterns specified using DSL to automatically generate their detection algorithms. Using these algorithms, the authors intended to support both the static and dynamic analysis of SBSs and their combination as well. The static analysis concerns the structural

properties of SBSs at design time, while the dynamic analysis considers mainly the runtime properties related to their QoS.

However, we noted that both OO and SOA detection methods cannot be directly applied to RESTful APIs because OO focuses only on classes and SOA focuses on services and WSDL descriptions. In consequence, few detection approaches [124,125] have been proposed to improve the design quality of REST APIs, in particular, their understandability, maintenance, and evolution. Palma et al. [124] proposed a heuristics-based approach for detecting five REST patterns and eight REST anti-patterns in RESTful systems. The proposed approach relies on a set of heuristics indicating the design issues that are considered as symptoms for the selected (anti)patterns. Based on the defined heuristics, a set of algorithms are implemented to support the automated detection of REST anti-patterns. However, the authors did not mention which formalism or languages are used to implement these algorithms. This work is extended in [125]. The extended approach focused on the linguistic aspects related to the uniform resource identifiers (URIs) in the REST APIs. Similar to their previous approach, the authors identified a set of linguistic patterns and anti-patterns as well as defined their corresponding heuristics to aid their detections. However, both approaches have supported the detection of few REST (anti) patterns and addressed only the non-cloud RESTful APIs, including Facebook, Twitter, Dropbox, and BestBuy. In contrast, in our work, we target the cloud RESTful APIs that are devoted to managing cloud resources. Besides, we intend to provide a unified method for detecting both cloud and REST (anti) patterns.

Moreover, several studies have been conducted with the aim of analysis and evaluating the RESTful APIs. The aim is to know to what extent those APIs advanced with respect to common REST architectural principles. Accordingly, in [138], a huge number of data logs were collected from HTTP traffic of mobile applications with the aim of analyzing these logs, identifying the emerged patterns and comparing them with REST guidelines and principles. In doing so, the authors rely on a set of five best principles that concerns the main design aspects related to REST APIs, namely: (i) resources modeling, (ii) resources identification and resource identifiers (URIs) design, (iii) resources representation, (iv) HTTP operations definition, and (v) resources interlinking ou hypermedia. Further, the authors define a set of heuristics based on request metadata available on the dataset in order to determine the compliance of the identified API with these best practices. In addition, some of these heuristics have been leveraged again to analyze the compliance of these APIs with the levels that are introduced in Richardson's maturity model [137].

The proposed heuristics are all implemented in JavaScript which therefore limits their application on REST APIs from other domains like cloud computing. On the whole, according to the obtained findings, the highest level of maturity is only reached by very few APIs from the analyzed ones. Also, it is shown that there is a clear gap between theory and practice as most of the best practices have been not followed by API developers.

In [97], Maleshkova et al. provided a deep and comprehensive analysis of the current state of Web APIs. More than 220 available Web APIs that include REST were analyzed. The analysis has been manually performed and relates to the technical aspects of the evaluated APIs. More precisely, the authors investigated six characteristics of Web APIs, including specifically, their types, output formats, input parameter, general information, invocation details and finally their complementary documentations. According to the authors, the analyzed Web APIs are not naturally REST and sustained from under-specification as they omit most of the indispensable information like HTTP methods and data-type. Therefore, this illustrates the need to study REST APIs design in cloud services, which support the contribution of our thesis.

In [76, 77], a framework for the structural analysis of REST APIs is proposed. Different from all the previous approaches, the authors focused on API description documents as the first source for the analysis of the structure of REST APIs. In addition, they introduced a canonical meta-model that is used to describe a REST API in terms of its core concepts, including, resources, methods, representations, and URIs. REST API description languages such as RAML [133] and Swagger [146] are used to build this metamodel. However, at the validation step, the authors concentrated only on Swagger since it is commonly and widely used compared to RAML. In particular, a set of Swagger documents are transformed into the canonical metamodel. The transformation serves firstly to identify all resources alongside with supported methods, representations and transform them into the corresponding canonical entities. Relationships between the resources are also considered. However, their identification is critical and not easy as they are not explicitly described in the Swagger documents. This makes their transformation crisp and error-prone. After this transformation, the obtained model is stored in a repository that can be treated by the analysis component. The latter is composed of a set of algorithms dedicated to compute the required metrics for the REST API analysis. Examples of these metrics include the number of resources, used methods (Get, Post, Delete, etc.), links. Although these metrics can provide an overview of structural properties that characterize REST APIs, they do not reflect their compliance with the REST principles. Instead, this feature has been considered as one of the authors' future work.

In [129], the authors conducted a systematic study of REST best practices on three well-known cloud APIs including, Google Cloud Platform, OpenStack, and the OCCI standard. In doing so, a catalog including REST 73 best practices is employed in assessing the REST features offered by selected REST APIs and their compliance with these practices. The analysis is relied on the available documentation related to these APIs and demonstrated that they have reached an acceptable level of maturity. However, the observed values are not high enough, more particularly, it has been demonstrated that Google Cloud follows 66% (48/73), OpenStack follows 62% (45/73), and OCCI follows 56% (41/73) of the adopted best practices. This often worsens the quality of these. Most importantly, the observed lack in supporting some

of the best REST practices by OCCI may negatively impact the understandability and reusability of management APIs that adopt this standard.

3.2.4 Synthesis

With regard to (anti) patterns identification efforts, several domains have been explored, including OO, SOA, REST and recently Cloud computing. While OO and SOA systems have been received a lot of attention and reached an acceptable level of maturity with regard to the definition of the most prominent (anti)patterns, REST and cloud computing still at early stages in this respect. Besides, all identification (anti) patterns efforts, including books, catalogs, and online documentation has made a considerable advance in the design quality of multiple software systems and significantly increased the designers' awareness and understanding about their systems. However, to the best of our knowledge, none of these efforts address the design aspects of cloud RESTful APIs for ensuring or assisting the interoperable management of cloud resources. These design aspects are mainly related to REST operations for ensuring common management tasks including creation, deployment, autonomic scaling, and monitoring. Therefore, in our work, we are particularly interested in identifying a set of new cloud patterns and anti-patterns to assist the creation of interoperable management APIs by analyzing the set of guidelines introduced in the OCCI standard. Besides, for more understandable and reusable APIs, we will explore the current REST literature and practices to identify other related (anti)patterns as the current catalogs are too limited and only focus on patterns. In this vein, our work can be seen as complementary to the previous efforts by proposing another (anti)patterns catalog that addresses a new design aspect, i.e interoperable management of cloud resources.

Besides, with respect to (anti)patterns modeling and formalization, the proposed approaches have followed two directions in order to support the description of (anti) patterns. These directions include: (i) introducing new models based on the new domain-specific grammars or using classical UML diagrams; (ii) adopting already-defined formalisms. Although approaches following the first direction are known by their expressivity power and easiness of use, they still suffer from the lack of formal semantics. This prevents the possibility of formal reasoning about (anti) patterns and driving new knowledge. In most cases, to support the new defined models, it is required to develop from-scratch solutions. However, this is tedious and unnecessary especially with the presence of software tools that can be leveraged to perform any task related to (anti) patterns such as their detection and integration in software design. In response to these issues, various formalisms have been adopted providing formal and precise descriptions of (anti) patterns. In this vein, ontologies have proven their potentials in providing a semantically enriched representation of (anti)patterns. Most importantly, they are endowed with rigorous reasoning engines and tools that can be exploited to infer new knowledge about (anti) patterns as well as systems supporting them, with the assurance that the provided new knowledge is sound. This

knowledge can be used to support any activities related to (anti)patterns, namely their detection and integration in software design and development stages. Nevertheless, previous researches that use ontologies, have focused only on design patterns and anti-patterns in the OO systems. Thus, there is no initiative for providing a semantic definition of (anti)patterns in RESTful cloud APIs. Therefore, to overcome this limitation, we aim at using ontologies toward a semantically-enriched description of (anti)patterns that target the design characteristics of RESTful APIs devoted to cloud resource management.

In regards to (anti) patterns detection and analysis, various initiatives have been proposed with the aim of leveraging patterns, anti-patterns or both in the evaluation of design and development quality of several kinds of systems. Many automated approaches have been proposed to assist the detections of (anti) patterns. Most of them have dealt with OO and SOA (anti)patterns. Whereas few approaches targeted the detection of (anti)patterns in RESTful APIs. In addition, the detected REST (anti)patterns do not reflect all best REST principles (or at least the most common ones) and related only to no-cloud environments. In our thesis, we aim at proposing a detection method that specifically targets cloud REST APIs while being flexible and extensible enough to support a large number of REST (anti)patterns. Different from the previous approaches, we intend to detect new kinds of (anti)patterns toward assisting for an OCCI-compliant creation of REST management APIs, as this is highly required for ensuring interoperability. In addition, most of the current detection methods have relied either on ad-hoc programming or domain-specific rule languages to defined the anti-patterns detectors. Other approaches used the semantic reasoning processes to infer new knowledge needed to detect (anti)patterns and adopted semantic rules languages to defined their detectors. Motivated by the great potentials of semantic solutions in the representation as well as reasoning, we seek likewise to use them as underlying techniques especially in the definition of (anti)patterns detection rules and algorithms.

Finally, with regard to RESTful APIs analysis, except this study [138], all the other studies have been conducted manually. However, this is not practical for APIs that involve a huge number of resources and operations and restricts the application of these analyses on REST APIs in other domains. Also, it is obvious that such kind of analysis requires more attention and effort from specialists. Moreover, as key observations, the authors indicated that few of REST APIs that were selected from general domains (Facebook, Twitter, Network, and mobile), have not reached an acceptable maturity as most of REST-related best practices have been not followed. This is also valid for REST APIs from three cloud computing providers: Google Cloud Platform, OpenStack, and OCCI standard. Hence, this illustrates the need for tools and methodologies that supports developers toward more mature APIs that are compliant to principles and guidelines that REST architectural style has. Our work falls within this context. However, in contrast to previous works, we aim at supporting the automated analysis that targets specifically cloud REST management

APIs with respect to REST and Cloud (anti)patterns that we intend to define from the OCCI standard.

Last but not least, none of the analysis and detection approaches have considered refactoring solutions to avoid the occurring of anti-patterns. To address this lack, we seek to provide APIs developers with recommendation support. Summing up, In our thesis, to address the observed shortcomings, we aim at providing a holistic approach based on semantic technologies to allow the definition and detection of (anti)patterns on one hand and provide recommendation support to correcting the occurring anti-patterns on another hand. Our approach targets cloud REST management APIs and their related (anti)patterns.

3.3 On facilitating Cloud resources orchestration

Cloud resource Orchestration is a key factor to exploit the potential of cloud computing [135]. Faced with its growing importance, various initiatives have been proposed. Some efforts led to the creation of cloud application orchestration and management standards, namely TOSCA [116] and CAMP [117]. Others from academic and DevOps communities proposed new models based on domain-specific languages to represent cloud resources and address some of their orchestration-related aspects. The idea of exploiting solutions proposed by the DevOps community is also explored. Hence, in this section, we position our work versus these initiatives and classify them into three categories: (i) Standard-based approaches, (ii) Non-standard based approaches, (iii) DevOps approaches. In the following, DevOps approaches are firstly explored as some of the standard and non-standard approaches integrated them into their proposed systems.

3.3.1 DevOps approaches

DevOps [96, 160] represents a new emerging IT paradigm that aims at unifying the application development (dev) and its administration operations (ops) such as deployment and testing in order to accelerate the development life cycle and offer a continuous delivery with the desired quality. In the context of the DevOps community, several orchestration platforms and tools are rapidly emerging. Since there is a vast number of DevOps solutions, we analyze here the most adopted and acceptable ones in the cloud community, which include: AWS-CloudFormation [23], Openstack-Heat [123], Docker [49], Kubernetes [90], Terraform [149], Juju [152] , Ansible [15] , Puppet [131], Chef [44]. These solutions are either provider-specific or independent.

AWS-CloudFormation [23], proposed by the cloud AWS provider, provides a language for specifying and provisioning infrastructure resources required for applications that can be span on all AWS regions. The provided language aims at structuring cloud resources as well as the relationships between them in a graph-like format written in

JSON. The major limitation of this orchestrator is the fact that it allows deployment only on Amazonâs cloud services, raising a vendor lock-in issue for its users.

OpenStack Heat [123] is an orchestration engine aiming to manage the entire lifecycle of infrastructure and applications based on templates that represent a set of text files manipulated like code. The definition of these templates relies on the so-called HOT, which stands for the Heat Orchestration Template. HOT describes the infrastructure resources for a cloud application using a YAML syntax. In addition to the deployment, Heat supports the runtime controlling of resources by providing auto-scaling features that are enforced by the OpenStack Telemetry service. Despite the fact that Heat is open-source, it only targets OpenStack clouds.

Docker [49] is known as an effective lightweight virtualization technology that enables packaging application code as well as its dependencies into a standard unit of software called *Containers.* This provides an abstraction layer ensuring that whatever the environment in which the application will be deployed, the execution is still valid. Besides, Docker offers orchestration tools facilitating container management and deployment. As shown in Figure 3.1, notable tools include Docker Compose and Docker Swarm. Docker Compose relies on YAML-based language to describe containers composing the applications and their dependencies. Docker Swarm represents a cluster of nodes (e.g. hosts) and allows the configuration of containers and automatic management of load distribution. This is enabled through its master-slave architecture where each cluster consists of at least one manager node that is responsible for managing the cluster and the distribution of tasks and any number of worker nodes that support the execution of application units (e.g. services).

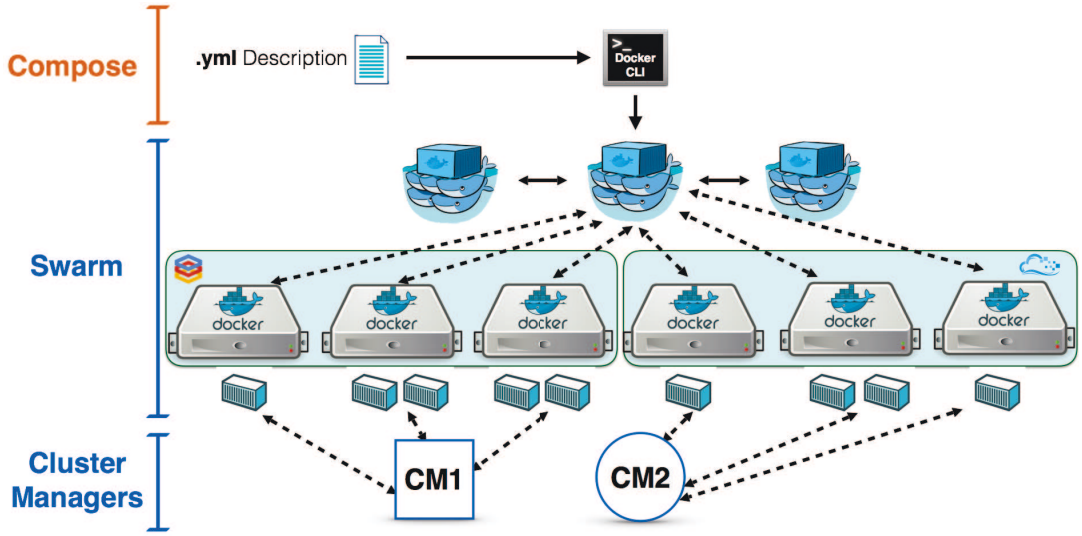


Figure 3.1: Docker Architecture (Source: [1])

Kubernetes [90] is an open-source container orchestration platform proposed by Google. It allows the automated deployment and management of multi-container applications. Besides Dockerfile which is required for each application unit or service, Kubernetes requires two other artifact files namely Deployment file and Service file to deploy applications. Both files are defined using a specific template written in YAML. Deployment file is a manifest file that describes the pods and containers, as well as other properties such as the ports. Here, it should be noted that pods represent instances of application services, where each one consists of one or more containers. The service file is devoted to discovering the proper pod that is running.

Terraform [149] provides a highly configurable platform for building large infrastructure covering multiple cloud providers. Its main goal is the orchestration of resources related to infrastructure. Indeed, it provides plugin interfaces for supporting the specificity of each cloud provider. Terraform relies on a declarative configuration language to build, modify, and versioning infrastructure descriptions. In addition, one of Terraform's strengths is the fact that it adopts the aspect of parallelism when creating resources from different cloud providers. Figure 3.2 shows how Terraform works.

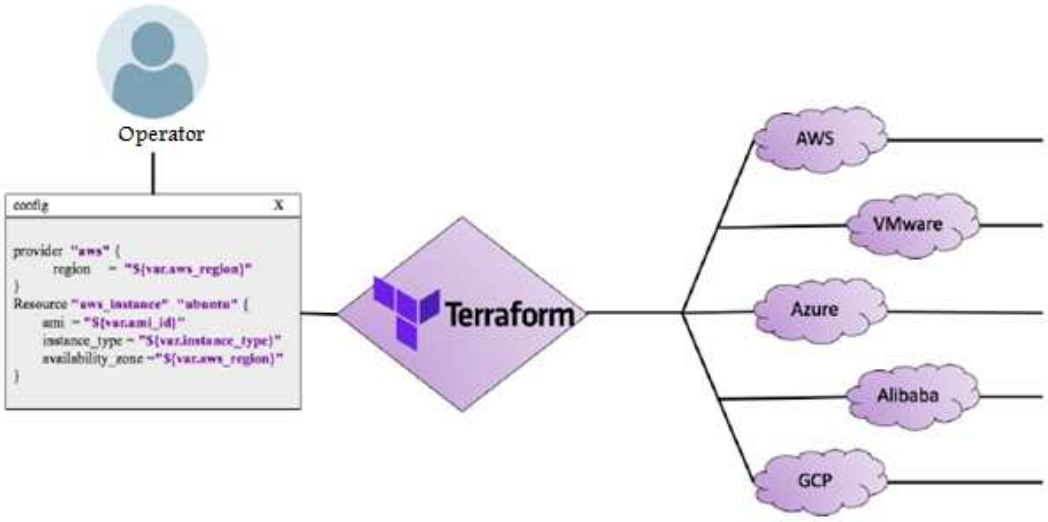


Figure 3.2: The operating principle of Terraform (source: [149])

Juju [152] is an open-source, application and service modeling tool proposed by Ubuntu. It allows the orchestration (i.e. deployment, management, scaling) of applications on private or public clouds. The key concepts of Juju are Charm and Bundle. Charm defines the orchestration logic related to a given service using scripts, which in turn can be implemented in any common scripting language (Python, Ruby, Unix shell, etc.). The bundle represents a compound deployment that comprises multiple charms and describes all their associated relations and configurations. To describe

such bundles, Juju relies on its own DSL following a YAML-based notation.

Ansible [15], Puppet [131], Chef [44] are Configuration management tools that mainly aim at configuring and maintaining systems overs various runtime environments (e.g. Tomcat server, MYSQL server, etc.) that can be hosted on different cloud providers. Configuration and maintenance facilities are both required to ensure the proper deployment of applications. Each one of these configurations tools relies on its own language. For instance, Ansible provides a declarative YAML-based DSL, whereas, Chef uses a Ruby-based DSL.

3.3.2 Standard-based approaches

In this section, we present the standard-based approaches for supporting cloud resource orchestration. All reviewed work is related mainly to TOSCA as it is the only standard that focuses on orchestration aspects of cloud applications. Few works have been proposed to support cloud orchestration through providing runtime platforms for the TOSCA standard. Notable solutions includes OpenTOSCA [43] and CAMF [94]. OpenTOSCA [43] represents the first open-source system providing a runtime deployment and management of TOSCA-based Cloud applications. It relied on imperative processing that is based on defining processes called building plans (e.g. Workflow) to specify the deployment and management logic related to each application component. However, as nowadays most applications involve multiple and larger numbers of components, the design of such plans becomes a very complex and error-prone task. In our proposal, this is avoided by adopting the declarative processing provided by the DevOps solutions, which allow DevOps users to orchestrate their applications without describing the related building plans. Despite the importance of OpenTOSCA, it does not support a runtime orchestration providing monitoring and controlling capabilities.

CAMF [94] stands for Cloud Application Management Framework, which is an open-source solution based on the Eclipse Rich Client Platform. In terms of orchestration facilities, it provided three core operations: Application description, Application deployment, and Application monitoring. The âApplication descriptionâ aims at providing an interoperable specification of application structure through capturing its high-level components, relationships between them as well as their management operations. This enabled by adopting the TOSCA standard. The âApplication deploymentâ supports the submission of the application description to any cloud infrastructure. Finally, âApplication monitoringâ allows collecting operational data (e.g., metric measurements, statistics, resource states) for enabling performance monitoring. Currently, to ensure the deployment operation of the application, CAMF relies on configuration per node using native shell scripts. However, this is complex and not scalable in scenarios that involve the configuration of multiple nodes. Besides, instead of reusing the already proposed standard APIs such as OCCI [118] or open-source ones such as Jcloud [16], CMAF defines its proper connecters to communicate

with cloud providers. Actually, in our proposal, we endorse the using of configuration tools such as Terraform, Chef to avoid direct communication with provider platforms. These tools have already demonstrated their potential in supporting a large number of providers [155].

Other works [12, 40, 42] tried to combine TOSCA with the CAMP standard to support the communication with provider platforms, which is needed to enforce the deployment and management of cloud resources involved in the orchestration life cycle. In essence, this combination aim at exploiting the advantages of this standard in the deployment of applications over multiple and independent cloud providers. Specifically, [12, 40, 42] provide either model-driven and query-based transformation with the aim of converting abstract TOSCA service templates into concrete CAMP deployment plans. Model-driven transformation involves the mapping of TOSCA service template entities into appropriate deployment and management entities in the CAMP plan. Whereas query-based transformation allows interrogating the TOSCA service templates in order to get the required data for supporting the automated generation of deployment plans. The obtained plans can be then deployed through the Jcloud API that allows communication with multiple and independent providers for enforcing the deployment and management operations. However, this proposal provides only an infrastructure-centric deployment as it does not allow the use of external platform resources such as containers, which become now a desirable feature for reducing the management complexity [28]. In our proposal, we support this feature by providing mapping support between TOSCA and Docker, which represent one of the most leading container-centric orchestration solutions.

Few initiatives have been proposed to support the integration between some DevOps solutions and standards in particular TOSCA. Notable approaches include [156], [83], [41]. Wettinger et al. [156] proposed an integrated, standards-based modeling and runtime framework that its major building block consists of transforming Juju charm and Chef cookbooks into TOSCA nodes. The main objective is reusing the existing DevOps artifacts to assist the creation and enrichment of TOSCA topology models for cloud applications. The resulted topology models can be then deployed by the TOSCA runtime environment (i.e. OpenTOSCA) of the proposed framework. This limits the full exploitation of DevOps solutions potentials. Besides, this approach is based on the intermediate representation as to its underlying transformation strategy. However, this strategy proved to be less expressive power and requires considerable effort to encoding complex transformations [141]. Moreover, [83] and [41] are provided mapping support between TOSCA and two DevOps solutions, namely Chef and Docker. Both approaches rely on the direct manipulation of the code to transform a TOSCA topology model into a running application, which could be time-consuming, difficult to maintain and extend in case of changes.

3.3.3 Non-Standard approaches

Other research initiatives, including CloudFM [64], Roboconf [130], CloudCAMP [31], Occopus [88] and Denis et al. [47] provided new models based on domain-specific languages (DSLs) to describe cloud resources for management and orchestration purpose.

Ferry and al. [64] have introduced the CloudFM framework, which comprises a tool-supported DSL called CloudML for describing the intended applications and a models@runtime component for enforcing their provisioning, deployment, and adaptation at runtime. Inspired by the component-based modeling, CloudML allows developers to describe cloud applications with respect to two levels of abstraction: (i) the Cloud Provider-Independent Model (CPIM) and the Cloud Provider Specific Model (CPSM). CPIM intends to describe cloud applications as well as their deployment and provisioning aspects in a cloud provider-agnostic way. Whereas CPIM represents a refinement of CPIM by enriching its instances with cloud provider-specific information. To support the deployment of the obtained CPSM on top of cloud infrastructure, CloudFM relied on Jclouds [16] and the Flexiant APIs [66].

Roboconf [130] is a cloud orchestrator allowing the deployment and runtime control of cloud applications. It is introduced as an autonomic computing system (ACS) kernel implementing the basic administration mechanisms (sensor and effector). To support these mechanisms, Roboconf relies on its ad-hoc components, where each one is devoted to implement an orchestration need such as initialization, deployment, configuration, start-up, stopping and uninstallation. Besides that, it introduces a hierarchical DSL for expressing applications and their execution environments (e.g. cloud platforms). Using this DSL, three file types have to be specified for any application. Specifically, the first file represents a descriptor that specifies the Roboconf configuration along with the application in question. The second one is a cyclic graph devoted to specify the relationships between the application components. Finally, the last file type includes the required configuration scripts and software packages.

CloudCAMP [31] is a platform-agnostic framework allowing the description and deployment of cloud applications. In this framework, a domain-specific modeling language (DSML) is used to abstract the application components and provider specifications. To allow the concrete deployment, CloudCAMP transforms the application specifications into deployable Ansible Infrastructure-as-Code script. This transformation is based on querying an already defined knowledge base and generating infrastructure code.

Occopus [88] is a Multi-Cloud Orchestrator that provides a DSL to describe infrastructure and nodes definitions. Based on these definitions, it enables the automated deployment and maintenance of Scientific Infrastructures in the target clouds. Unlike our solution which adopts TOSCA allowing provider-independent descriptions, Occupus assumes that developers have configuration management knowledge of multiple providers. This is difficult and requires sophisticated skills.

Denis et al. [47] propose the use of domain-specific models (DSMs) for describing

cloud resources as well as their management strategies according to each DevOps solution. To allow that, they rely on a new Entity-Relationship (ER) model to represent these DSMs. Besides, the authors propose a set of connectors to support the automated translation of DSM objects into native resource descriptions and management artifacts specific to the target DevOps tool. This translation is based on direct manipulation of DSMs through a general-purpose programing language. Our solution differs from the above proposal in the two following points. First, instead of the manual DSM creation, we intend to use TOSCA to abstract the heterogeneous entities used in each DevOps solution into high-level ones, which would be then leveraged to generate the DSM for each DevOps solution. Second, we adopt a model-driven transformation as an underlying technology to automate the DevOps artifacts generation.

3.3.4 Synthesis

After detailing the related work while outlining their difference to our work, we give, in this section, a comparative analysis of each investigated category.

Table 3.1 provides a comparative overview of the DevOps approaches with respect to dimensions inspired by [155]. *Resource Representation* indicates how cloud resources are represented and using which notations (JSON, YAML, etc.). *Operations* denote the supported orchestration capabilities, namely, configure, deploy, monitor, control. *Scope* indicates which cloud resources types are considered by the orchestration. *Access Method* denotes how cloud resources are accessed, namely command-line interface (CLI), graphical user interfaces (GUI), application programming interfaces (APIs). The runtime environment is identified in terms of two sub-dimensions: *Virtualization technique* which refers to how physical resources are abstracted to simplify their consumption; *Target environment* which identifies different deployment models such as Single (one provider), multi-cloud (multiples providers). *Interoperability support* verifies if it has been supported by the proposed solution or not. Note that, in some cases, we used " + " to express that the corresponding criterion is fulfilled by the corresponding approach, " − " if it is not fulfilled, " + / − " if it is partially fulfilled, "NR" for *not relevant*.

By analyzing experimentally and theoretically the selected DevOps solutions, we observed that they all hold a lot of potentials for supporting an efficient orchestration of cloud resources. However, as it is demonstrated in the table 3.1, each DevOps solution targets specific resource types and management operations and follow different virtualization techniques. Thus, the orchestration of cloud applications that inherently involve diverse resources and virtualization techniques, would require using multiple DevOps solutions. This is extremely complex with the remarkable lack of interoperability among these solutions as they rely on diverse and heterogeneous resource description models, management capabilities and access methods. Moreover, as most of DevOps solutions assume a low-level and sophisticated programmatic approach, this complexity can only increase several-fold. This is because that seamless

Approaches	Resource representation	Operations	Scope	Access methods	Runtime environment		Intero-perability
					Virtualisation technique	Target environment	
AWS CloudFormation [23]	DSL (JSON)	Deploy	IaaS, PaaS	CLI, APIs	OS	Single	-
Heat [123] (Openstack)	DSL (YAML)	Deploy, Monitor, Control	IaaS, PaaS	CLI/APIs	OS	Mutli (only Openstack based clouds)	+/-
Docker [49]	DSL (YAML)	Deploy,Update, Monitor, Control (+/-)	PaaS	GUI, APIs, CLI	Container	Multi	-
Kubernetes [90]	DSL (YAML)	Deploy,Update, Monitor, Control	PaaS	GUI, APIs, CLI	Container	Multi	-
Terraform [149]	DSL (TF)	Deploy,Update, Monitor, Control	IaaS	GUI, APIs, CLI	OS	Multi	-
Juju [152]	DSL (YML)	Deploy, Control	IaaS, PaaS	GUI, APIs, CLI	OS	Multi	-
Ansible [15]	DSL (YAML)	Configure, Deploy	PaaS	APIs, CLI	NR	Multi	-
Puppet [131]	DSL	Configure, Deploy	PaaS	GUI, APIs, CLI	NR	Single	-
Chef [44]	DSL or Ruby	Configure, Deploy	PaaS	APIs, CLI	NR	Single	-

Table 3.1: Comparative analysis of DevOps solutions

integration between DevOps tools would imply considerable development effort and continuous patching from the DevOps user side. In contrast to DevOps solutions, we adopt the TOSCA standard as a technology-independent metamodel for providing an abstract representation of cloud resources along with orchestration aspects. This would shield the DevOps users from heterogeneity and complexity of underlying DevOps solutions.

Table 3.2 represents a comparative analysis of both standard and non-standard based approaches. We rely on the same criteria used above with the addition of the "Third-party tools" criterion which intends to check whether these solutions reuse the existing solutions or propose their own ones (Ad-hoc). Most of these approaches focused only on providing deployments capabilities for cloud applications. advanced runtime orchestration aspects such as monitoring and runtime controlling have been not handled yet by these approaches. In addition, most of them adopted os-level hypervisors as their underlying virtualization technique. 2 out of 12 selected approaches [47, 88] supports diverse virtualization techniques, i.e. container-level, and os-level virtualizations together. These solutions are not based on standard and assumes that the developer has diverse knowledge about DevOps solutions. As indicated above, with the presence of heterogeneity, the developer mission will be too cumbersome and time-consuming. Besides, regarding non-standard approaches that have been all proposed in an academic context, each one relies upon its own resources description languages with its specific abstraction entities and syntax. However, the heterogeneities imposed by these languages certainly impede interoperability. To avoid such an issue, adopting open standards is highly required, which we did in our

Approaches	Resource representation	Operations	Scope	Third party tools	Run. environment		Interoperability
					Ver. technique	Tar. environment	
OpenTOSCA [43]	TOSCA	Deploy	IaaS, PaaS	shell scripts, Chef, and Ansible	OS	Multi	+
CAMF [94]	TOSCA	Deploy, Monitor, control	IaaS, PaaS	Adhoc	OS	Single	+/-
[12, 40, 42]	TOSCA	Deploy	IaaS, PaaS	CAMP, JCloud	OS	Multi	+
[156]	TOSCA	Deploy	IaaS, PaaS	Juju and Chef	OS	Single	+
[83]	TOSCA	Deploy	IaaS, PaaS	Chef	OS	Single	+
[41]	TOSCA	Deploy	PaaS	Docker	Container	-	+
[64]	DSL	Deploy, Update (+/-)	IaaS, PaaS	Jcloud	OS	Multi	-
[130]	DSL	Deploy	IaaS, PaaS	Adhoc plugins	OS	Multi	-
[31]	DSL	Deploy	IaaS, PaaS	Ansible	OS	Single	-
[88]	DSL	Deploy, Update	IaaS, PaaS	Config tools	OS/Container	Multi	-
[47]	DSM	Configure, Deploy, Monitor, Control	IaaS, PaaS	Docker and Juju	OS/Container	Multi	-
[140]	DSM	Configure, Deploy	IaaS	Ansible	OS	Single	-

Table 3.2: Comparative analysis of standard and non-standard based approaches

Approaches	Hard-programming	Uniform interfaces	Automated Mapping		
			Direct Model Manipulation	Intermediate Representation	Transformation Language Support
Standard-based	[83], [41]	[39]	-	[156]	-
Non-Standard	[88]	[88]	[47], [31]	-	[140]

Table 3.3: Comparative analysis of approaches supporting the integration of DevOps solutions

solution.

A key observation related to both kinds of approaches, with regard to the third-party tools (refer to table 3.2), is the integration of existing cloud open APIs and DevOps solutions to allow the concrete execution of orchestration tasks. Cloud open APIs have provided great potentials for communicating with providers and requesting the execution of certain actions by their underlying management services. Some of these APIs (OCCI, jCloud) have already integrated into DevOps solutions. But lately, DevOps solutions become more powerful comparing to cloud APIs thanks to its everything as code paradigm and its known advantages. Similarly to these approaches, we will use DevOps solutions into our orchestration but in a different way emphasizing their seamless, and easily expandable integration.

Table 3.3 provides a comparative analysis of approaches that provided integration support with DevOps solutions. We align these approaches with respect to different mechanisms adopted to support such integration. In this respect, we distinguish three possible mechanisms that are followed by selected approaches: (1) Hard programming; (2) Uniform interfaces; (3) Automated mapping. As demonstrated in the table 3.3, some of approaches [41, 83, 88] adopted a low-level and hard programming. Hard programming is a software development practice that consists of interpreting the inputs models (i.e. Application typologies in our context) to get required data, which in turn have to be injected directly into the source code. This is opposed to generating these data and put them into stored and machine-readable artifacts that can be processed later by procedural APIs to ensure further management tasks. It has been already proved that this approach is hard to be maintained, extended when needed and keep it synchronized.

Also, uniform interfaces are considered by a few approaches [39,88]. Indeed, they are communication middlewares that aim at abstracting the inherent heterogeneity among the different DevOps APIs and tools that can be invoked to ensure any management operations. By convention between cloud practitioners and adopters, this mechanism is highly recommended. However, the proposed interfaces are limited only to configuration tools specifically chef and ansible. Besides, there is is no assistance for developers to create such interfaces automatically or at least semi-automatically, in a way that most part of code can be generated based on predefined skeletons.

Moreover, few others have supported automated mapping which is either driven by a standard like [156] or based on new DSMs like [140], Weerasiri, [31]. Automated mapping provides solid support to allow the generation of native artifacts (e.g. DevOps-specific artifacts) from ones (TOSCA typologies or the new DSMs) that are often described at a high level of abstraction and independently from technological mechanisms. Nowadays, it became necessary to provide a methodology and techniques to support automated mapping as it is able to avoid the burden of implementing and defining artifacts that are closely dependent on technology-specific solutions. This will ensure the interoperable orchestration of these artifacts while mitigating complexity related to their implementation. Despite that, under the existing solutions, the mapping is provided either based on the intermediate representation [156] or by a direct model manipulation [31,47] using a set of procedural APIs. Both methods are less expressive as they lack appropriate high-level constructs (or abstractions) for easily encoding transformation rules that allow such mapping. This makes these transformations hard to write, comprehend, maintain and extend when needed [141].

Summing up, to address the observed shortcomings, the second thesis contribution is devoted to streamlining and improving the orchestration of cloud resources. Similar to standard-based approaches, we endorse the use of TOSCA as a technology-agnostic resource description model allowing the interoperable and portable representation of cloud applications. To support the orchestration of TOSCA models, we propose a model-driven integration between TOSCA and DevOps solutions. With the aim of en-

abling this integration in a seamless way and acceptable cost, our key mechanisms are providing (i) high-level automated mapping and (i) DevOps abstraction layer. Unlike the previous approaches, automated mapping is provided based on model-driven and transformation language support toward ensuring its manipulation at a high level of abstraction, thereby easing its synchronization and extension when needed. Besides, we propose a set of high-level connectors building together the intended DevOps abstraction layer to hide the heavy lifting involved when interacting with the underlying tools/APIs of target DevOps solutions. Unlike the proposed approaches that interface with Chef and Ansible, our DevOps abstraction layer will target Docker, Kubernetes, and Terraform. Finally, we rely on the model-driven generation principle to allow the semi-automatic creation of connectors while fostering their extensibility.

3.4 On supporting Cloud resources elasticity

Despite the importance of cloud elasticity specifically in ensuring cost-to-performance trade-offs, few approaches have been proposed to tackle with their description and implementation aspects. In this section, we first present an extensive review of cloud resources elasticity solutions both in research and industry. Next, we discuss the positioning of our elasticity solution versus these attempts.

3.4.1 Commercial and open-source solutions

Herein, commercial and open-source resource elasticity solutions proposed both by cloud providers and the DevOps community are explored.

Amazon AutoScaling [26] is an autoscaling service proposed by AWS provider allowing the horizontal scaling by automatically increasing or decreasing Amazon EC2 capacity according to predefined user conditions. Through Autoscaling service, the number of Amazon EC2 instances increases continuously during peak demand to maintain performance and decreases automatically during lower requests to minimize costs. AutoScaling is enabled by the Amazon CloudWatch monitoring system and based on reactive threshold-based rules. In simple terms, it allows the scaling of EC2 instances based on the measurements retrieved by CloudWatch or predictably according to a predefined schedule.

Azure Autoscale [107] is an autoscaling service proposed by Microsoft Azure, almost providing the same capabilities of AWS autoscaling namely a reactive scaling based on threshold-based rules and a proactive scaling based on the schedule. In contrast to AWS that targets only computes instances (i.e.EC2), Azure Autoscale is basically integrated with cloud Services, mobile Services, virtual machines, and websites. Also, it is based on alerting-based monitoring service providing performance measurements for enforcing scaling functionalities.

IBM AutoScaler [78] is an autoscaling service proposed by IBM Cloud, allowing the horizontal scaling of application instance number. The provided scaling is either

scheduled based on time or reactive based on metrics related to application performance. In contrast to the above solutions that propose its own monitoring systems, performance metrics related to the application are collected and analyzed by mean of a mix of open-source tools, namely Prometheus [3] and Grafana [73].

Moreover, some of DevOps solutions support the description and controlling of elasticity. Pod Autoscaler [90] proposed under Kubernetes aimed at providing an automated horizontal scaling of the number of pods based on observed CPU utilization. It comprises a Kubernetes API resource and a controller. The former observes the controller behavior by providing the CPU measurements, whereas the latter is responsible for adjusting the required number of pod replicas. Kubernetes scaling service relies on its own monitoring system called Metrics-server that has to be deployed in the cluster for capturing the desired metrics. Pod Autoscaler allows DevOps users to create their elasticity policies either using CLI or writing their description using YML DSL devoted for this purpose. In contrast to Kubernetes, Docker [49] provides manual horizontal scaling of containers using both Docker CLI or GUI. Thus, DevOps user has to develop their own program based on the procedural programming languages proposed by Docker to allow the missing automated support.

Juju recently adds Juju Autoscaler [152] that allows the horizontal scaling of juju application deployments (units or services) using rule-based algorithms. The current release of the Autoscaler scales the number of applications units based on the CPU usage metrics. To collect these metrics, Juju relies on the telegraph agent which allows pushing their values into an InfluxDB database. Thus, the stored metric values can be consumed by the Charm Autoscaling for further analysis and making scaling decisions. The proposed solution is still in the first stage and does not provide any language to support the description of scaling algorithms. Indeed, the only available method for DevOps users is using Juju CLI which provides only a manual configuration of intended algorithms.

Terraform [149] supports the definition of horizontal scaling policies for enabling the automated adjustment of computing instances (virtual machines or servers). Indeed, Terraform itself does not provide any specification language to define elasticity aspects. Instead, it incorporates into its configuration template the scaling policy specifications that were defined by cloud providers. Basically, in the Terraform configuration template, the scaling policies are injected as configuration resources, where their structural specifications are closely dependent on selected cloud providers. For example, listing 3.1 shows an autoscaling policy that allows increasing the AWS_Compute_group by adding 4 instances whenever the average CPU usage exceeds 80%.

```
1
2 resource "aws_autoscaling_policy" "increasing_policy" {
3   name                   = "test"
4   scaling_adjustment     = 4
5   adjustment_type        = "ChangeInCapacity"
6   cooldown               = 300
7   autoscaling_group_name = "Computeinstance_group"
```

```
}

resource "aws_cloudwatch_metric_alarm" "bat" {
  alarm_name          = "terraform-test-foobar5"
  comparison_operator = "GreaterThanOrEqualToThreshold"
  evaluation_periods  = "2"
  metric_name         = "CPUUtilization"
  namespace           = "AWS/EC2"
  period              = "120"
  statistic           = "Average"
  threshold           = "80"

 resource "aws_autoscaling_group" "Computeinstance_group" {
  availability_zones        = ["us-east-1a"]
  max_size                  = 5
  min_size                  = 2
  // Other configuration options are removed for brevity
}}
```

Listing 3.1: Horizontal scaling policy according to AWS autoscaling specification incorporated into Terraform configuration template adapted from [149]

3.4.2 Research solutions

Some research approaches define languages to describe and to implement the cloud resource elasticity. Two important works are proposed by Copil et al. [45] and et Al-Dhuraibi [10]. [45] provides a domain-specific language, named Simple Yet-Beautiful Language in order to specify elasticity requirements at different levels of cloud applications, including the whole application, the application component, and the component code. The authors distinguish four main directives:

- monitoring directives for specifying which metrics need to be monitored;
- constraints directives for specifying acceptable limits for the monitored metrics;
- strategies directives for specifying actions to be taken and conditions in which those actions need to be taken;
- and finally (4) predefined functions that devoted to obtaining information both on the current environment and the elasticity specifications.

[10] proposed an elasticity management system called MODEMO which allows the vertical and horizontal scaling both of VM and containers across multiple cloud providers simultaneously. MODEMO relies on the OCCI standard to define all elasticity-related features. Another work proposed in [157] aimed at introducing a declarative domain-specific language called SPEEDL. SPEEDL allows the creation of elastic scaling behavior on top of IaaS resources. It aims at simplifying the creation of event-driven policies for resource management (How many resources, and what resource types, are needed?), as well as task mapping (Which tasks should be

handled by which resources?). Another research effort [89] introduced a model-driven language called SRL (Scalability Rule Language) for specifying scalability rules that support complex adaptation scenarios of multi-cloud applications. SRL provides an Eclipse-based tool allowing modelers to specify scalability rules while validating its syntax. It also integrated with CloudML Language to enable associating scalability rules with the involved components and target virtual machines. In contrast to our work that intends to target multiple elasticity actions, SRL support only horizontal scaling and vertical scaling policies. Also, it is not clear whether the authors implement a runtime prototype and monitoring system to enforce the SRL rules defined in the SRL model. Finally, [80] has proposed an elasticity strategy description language, which is tailored to be used only to define the elasticity of business processes as service.

3.4.3 Synthesis

In this section, we provide a comparative analysis with the aim of positioning our work versus the existing attempts according to a set of important dimensions reflecting the description and execution support for elasticity. *Language* is devoted to checking whether the solution provides language support to describe the different features that can be evolved when describing elasticity behavior. *Acess methodes* denotes how cloud elasticity is managed (i.e. edited, updated and controlled). In this respect, we distinguish three common methods, namely command-line interface (CLI), graphical user interfaces (GUI), application programming interfaces (APIs). *Scope* indicates which cloud resources type from the following list (IaaS, PaaS, SaaS) is concerned by the elastic handling. *Elasticity mode* indicates whether the solution provides manual (M) or automatic (A) support for executing elasticity. *Action types* and *Event types* respectively present which elasticity actions (*VS*: *Vertical scaling*; *HS*: *Horizontal scaling*; *AR*: *Application Reconfiguration*, *M*: *Migration*)) and events (*RRE*: *Resource Related Event*; *TE*: *Temporal Events*) that solution support.

Moreover, the *Runtime environment* (R. environment) is identified through the adopted *Virtualization technique* and *Target environment*. *Virtualization technique* indicates whether a solution support OS or Container level virtualization; *Target environment* is identified by two values: *Single* or *Multiple*. *Single* indicates that only one cloud provider is supported, while *Multiple* indicates that the elasticity control is spread across multiple cloud providers simultaneously. *Enforcement capabilities* (E. Capabilities) indicate whether the solution provides the concrete execution of elasticity in terms of supporting its *monitoring* and *controlling* functionalities. Finaly, *User Types* reflect the level of expertise of the involved users, which can be *DevOps* (application developers and system administrators) or *Domain-Experts* (an expert in a particular domain that needs to exploit cloud resources and have very little or no programming expertise).

In Table 3.4, we align the selected approaches according to the dimensions de-

Solutions		Lan guges	A. Methods	Scope	Mode	Met-hods	Events	R. environment		E. capabilities		User Type
								V. Tech-nique	T. Envi-ron-ment	Moni-tor-ing	Exe-cu-tion	
P.S	[26]	CF	GUI, CLI	IaaS	A	HS	RRE, TE	OS	Single	+	+	DevOps
	[107]	-	GUI	SaaS, IaaS	A	HS	RRE, TE	OS	Single	+	+	
	[78]	-	GUI	SaaS	A	HS	RRE, TE	OS	Single	+	+	
D.S	[90]	KDSL	CLI	PaaS	A	HS	RRE	Container	Single	+	+	
	[49]	-	CLI	PaaS	M	HS	RRE	Container	Single	+	+	
	[152]	-	CLI	SaaS	A/M	HS	RRE	OS	Single	+	+	
	[149]	TCL	CLI	IaaS	A	HS	RRE	OS	Multi	+	+	
R.S	[45]	SYBL	-	IaaS/ SaaS	A	HS, VS, AR	RRE	OS	Single	+	+	
	[89]	SRL	GUI	IaaS	A	HS	RRE, TE	OS	Multi	-	-	
	[157]	SPEEDL	-	IaaS	A	HS	RRE	OS	Single	-	-	
	[80]	START	GUI	SaaS	A	HS	RRE	-	Single	-	-	
	[10]	MoD-EMO	GUI	IaaS, PaaS	A	HS, VS	RRE, TE	Container, OS	Multi	+	+	

Table 3.4: Comparative analysis of elasticity solutions

scribed above. Providers leading solutions (refer to **P.S** in the table 3.4) such as Amazon AutoScaling [26], IBM AutoScale [78] and Azure Autoscale [107] rely on graphical user interfaces and command-line syntax to support the definition of elasticity policies that focus often on the horizontal scaling of cloud resources. Except for AWS [14] that provides a set of constructs to define the elasticity features as part of the CloudFormation template [23], the other two solutions [78,107] do not provide any language support. Generally, providers' elasticity solutions are very suitable for cloud users that will exploit cloud resources only from these providers. However, in the case of a multi-provider scenario, cloud users are obliged to follow low-level scripting mechanisms that are not intuitive for them as they do not have configuration management knowledge of multiple providers.

Despite the significance and effectiveness of DevOps solutions (refer to **D.S** in the table 3.4), the provided solutions for supporting elasticity features are still in the early stages [155]. This has been observed through three main issues. Firstly, under

these solutions, in particular, Juju [152] and Docker [49], the execution of elasticity is still done manually. Terraform [149] and Kubernetes [90] provide dynamic support for elasticity, but both are still exclusive to experienced developers as they provide low-level scripting languages. Secondly, all of them [49, 90, 149, 152] only support horizontal scaling and restricted to limited resources types. Regrading the target environment dimension, Terraform [149] is the only solution that has the capability of supporting multi-cloud elasticity. Despite that, such support is still not mature and costly as Terraform relies on the low-level and provider-specific description of elasticity features. This imposes an expert-driven manual effort and high technical expertise. The other DevOps solutions are still restricted to one provider. Thirdly, for holistic support of elasticity (i.e. considering multiple elasticity methods and the multi-cloud environment), DevOps users that rely on these DevOps solutions have to develop their custom program based on the programming languages proposed by these. Also, they have to use multiple reconfigurations and monitoring APIs to enforce their elasticity policies, which proves increasingly too challenging and time-consuming task as it requires a considerable development effort and multiple continuous patches.

In contrast to provider and DevOps solutions, we intend to provide high-level modeling abstractions that facilitate the description of elasticity without referring to any low-level languages, providers-specific formats, and their related technical constraints.

Research approaches (refer to **R.S** in the table 3.4) that handle the modeling and runtime support of elasticity are still limited compared to the immense effort that was devoted to cloud resource description. 3 out of 5 (i.e. [80, 89, 157]) selected approaches support only the modeling of horizontal scaling without providing concrete enforcement support in terms of monitoring and execution facilities. In addition, most of the approaches are tailored only to be used in the context of single provider deployment and os-level virtualization. To best of our knowledge, in terms of elasticity modeling support, MODEMO [10] is the only approach that handles elasticity in a holistic way while providing its runtime support. However, it should be noted that this proposal as well as the implementation of the related runtime system, have been performed recently and have been published after our contribution [35]. Despite that, our approach is still different as it intends to handle elasticity at a higher abstraction level and using intuitive modeling constructs.

Finally, as an observation applied to the reviewed approaches (refer to ***User Type*** criterion in the table 3.4), the proposed solutions can only be handled by professional DevOps, and not by domain-expert users that generally do not have sufficient programming expertise. In this respect, we believe that domain-experts and even end-users will be proportionally interested in exploiting cloud resources if they find suitable and intuitive tools aligned with their programming expertise and skills. This was already observed in the academic context, where researchers and teachers want to exploit cloud resources and specify their elasticity requirements without having to deal with the related modeling and implementation issues.

In light of observed shortcomings, the third thesis contribution is devoted to providing holistic support for managing elasticity in a multi-cloud environment. We provide such support by considering diverse elasticity features, that are multiple modes, multiple methods, multiple events, multiple resource types, and multiple virtualization techniques. In contrast to most of the above approaches, these features will be provided at a high level of abstraction with the intention to be more intuitive and user-friendly. Instead of relying on low-level, scripting and technology-dependent mechanisms, we argue that models and languages for describing cloud resources should be endowed with intuitive constructs that can be used to specify a range of flexible elasticity policies. To this end, we propose a new elasticity description model based on the state-machine formalism. The latter has been broadly used to model the reactive behavior of systems, which makes it very suitable to capture the elasticity behavior.

3.5 Conclusion

In this chapter, we provided an exploration of solutions relevant to our work. We classified them into three main categories: (1) on using (anti) patterns for software design assistance, (2) on facilitating Cloud resources orchestration, and (3) on supporting Cloud resources elasticity. For each category, we briefly introduced the related approaches and provided a thorough analysis in order to position our work versus these attempts. With regard to the first category (1), we showed that most of (anti) patterns-based approaches for the design assistance have focused on SOA and OO systems. Although (anti) patterns specification and detection approaches have been proposed in the context of REST, they all target general-purpose and non-cloud APIs.

With regard to the second category which relates to cloud orchestration, we explored three categories of approaches, namely the DevOps approaches, Standard-based approaches, and Non-standard approaches. We showed that a single DevOps approach is not able to provide all the orchestration capabilities that can be related to diverse cloud resource types. Regarding standard-based approaches, we revealed that most of them focus only on providing deployments capabilities for cloud applications. Advanced runtime orchestration aspects such as monitoring and runtime controlling have been not handled yet. Regarding non-standard-based approaches, they all employ their own resources description languages, which impede cloud interoperability. We also revealed that some of the standard and non-standard approaches tried to integrate DevOps solutions into their systems as third-party tools to ensure the execution of the orchestration operations. However, this integration is still poorly supported. Finally, with respect to the third category (3), we showed that the management of multi-cloud elasticity in existing solutions is still not mature and requires a considerable development effort from the user side.

We start presenting in detail our approaches in the next chapters. In chapter 4, we elaborate on how we assist the design of interoperable management APIs. In chapter 5, we introduce our model-driven approach to streamlining and improving

the orchestration of cloud resources. In chapter 6, we show how we can use a state-machine model to simplify and support the management of multi-cloud elasticity at a high-level of abstraction.

Chapter 4

Assisting interoperable management APIs Design using patterns and anti-patterns

Contents

4.1 Introduction

This chapter introduces our approach for assisting the design of interoperable cloud management APIs that have to be compliant with both OCCI and REST best design principles. Our approach promotes the use of (anti) patterns to guide this assistance. In particular, we define compliance to OCCI best principles as OCCI patterns and

non-compliance to OCCI best principles as OCCI anti-patterns. In like manner, we define compliance to REST best principles as REST patterns and non-compliance to REST best principles as REST anti-patterns. Next, we support the automated detection of the identified (anti) patterns in order to evaluate the compliance of management APIs with the OCCI and REST best principles. In tandem with this detection, recommendation support is also provided by suggesting correction explanations for preventing the occurrence of anti-patterns. Thus, we contribute in designing interoperable, understandable, and reusable cloud management REST APIs.

Toward this end, we concretely made two contributions. We first review the OCCI standard and the literature with the aim of identifying the set of patterns that must be respected and anti-patterns that should be averted to conform with both REST and OCCI best principles. Then, we propose a semantic-based approach for supporting the definition and detection of REST and OCCI (anti)patterns in Cloud RESTful APIs. More specifically, it supports the followings:

1. Proposing semantic modeling of cloud management APIs-related knowledge required for the detection and recommendation purposes

2. Proposing semantic definition of REST and OCCI (anti)patterns and specifying their detection rules in terms of SWRL rules in conjunction with SQWRL queries.

3. Proposing detection algorithms based on SPARQL queries to provide an automated detection of both OCCI and REST (Anti) patterns along with a set of correction recommendations in case of any anti-pattern occurrence.

To evaluate our approach, we developed ORAP-Detector tool as a proof of concept. Our tool aims at providing a compliance evaluation of cloud management APIs with respect to OCCI and REST best principles as well as recommendation support to comply with these principles. To conduct this evaluation, we rely on a validation dataset that includes five real-world Cloud RESTful APIs: OOi [122], COAPS [110], OpenNebula OCCI [121], Amazon S3 [21], and Rackspace [132]. The effectiveness of our approach has been demonstrated by analyzing the accuracy of the detection rules and the usefulness of the provided detection and recommendation support.

In the following, we start by presenting the OCCI and REST patterns and anti-patterns that we have considered after analyzing OCCI standard and existing literature on REST. Then, we introduce our semantic-based approach in Section 4.3. Finally, Section 4.4 presents a validation of our approach and an interpretation of the experiment results.

The work in this chapter was published in conference proceedings [38] and peer-reviewed journal [37]

4.2 Identifying REST/ OCCI (Anti)patterns

In this section, we present the REST and OCCI patterns and anti-patterns that we identified after analyzing both the literature and the OCCI standard. This analysis is done in context of OCCIware project [1]. OCCIware is a scientific research project that aimed at providing a new precise metamodel for OCCI, along with an enhanced tooling environment called OCCIware Studio. Both OCCIware metamodel and Studio are developed for designing, managing and analyzing any kind of cloud resources. In the following, we provide and summarize both REST and OCCI (anti)patterns definitions.

4.2.1 REST (Anti)patterns

REST (anti) patterns represent the good and bad practices in the REST APIs regardless of any cloud standard. To identify them, we conduct a literature review both in research and industry. We performed our research using Google Scholar, Elsevier Scopus, ACM Digital Library, Web of Science, IEEE Xplore and arXiv.org with special focus on the critically-reviewed research conferences, journals and magazines that were relevant to our research context from the year of 2004. Initially, 25 approaches, catalogs and technical reports on REST (anti) patterns were chosen. However, some of these works contain redundant contents. After filtering them, we ended up with 7 studies, including [101], Rodrigues et al. [138], Palma et al. [124, 125], Vinoski [154], Stowe [147], Richardson and Ruby [137]. We analyzed all the above studies to define the REST (anti)patterns and organize them into categories, while inspiring from the work of Masse [101]. Tables 4.1, 4.2, 4.3, 4.4 and 4.5 define (anti)patterns we specified for each category.

- URI (anti)patterns: They represent the poor and good practices in URIs and how they are exposed by services (see Table 4.1).
- HTTP methods (anti)patterns: They represent the poor and good practices in HTTP methods and how they must be used by REST APIs (see Table 4.2).
- Error Handling (anti)patterns: They represent the poor and good practices in HTTP messages and how they must be used as a response of a HTTP request method (see Table 4.3).
- HTTP Header (anti)patterns: They represent the poor and good practices in HTTP headers and how they must be used to complete requests with metadata or complementary data (see Table 4.4).
- Hypermedia (anti)patterns: They represent the poor and good practices in hypermedia representation and how it should be supported to link between resources (see Table 4.5).

[1] Available at www.occiware.org

Table 4.1: URI Design (Anti)Patterns

1. Tidy URIs vs. Amorphous URIs
Description: URIs in the REST resources should be simple to read and tidy. The Tidy URIs pattern appears when URIs use suitable lower resource naming and do not contain trailing slashes, underscores and extensions. While, the Amorphous URI anti-pattern appears when URIs contain symbols, capital letters, underscores, etc. This results in decreased readability and understandability of these URLs [125, 129].
2. Verbless URIs vs. CRUDy URIs
Description: Verbless URIs pattern appears when URIs use one of the Standard HTTP methods, namely POST, GET, DELETE or PUT. While, CRUDy URIs anti-pattern is appeared as consequence of using CRUDy terms such as read, create, delete, update or their equivalent in URIs. Using these terms in actions or resource URIs can be overloading the HTTP methods and prevent API users to employ the appropriate and common HTTP methods [125, 129].
3. Singularized nodes vs. pluralized nodes
Description: URIs should correctly employ singular/ plural nouns for resources naming within an API [125, 129]. Singularized nodes pattern occurs when the last node is provided as a singular noun in the URI of Delete/ Put requests and as a plural noun in POST requests. Contrariwise, the Pluralized Nodes anti-pattern can occur when singular nouns used in POST requests or plural names used in DELETTE/PUT requests. The occurrence of such anti-pattern may have negative impacts in certain cases. For instance, if the last node in Delete (or PUT) request URL is provided as plural, the API clients are not able to create or delete a collection of resources, which leads to 403 Forbidden as a server response.

Table 4.2: HTTP methods (anti)patterns

1. Correct use of POST, GET, PUT, DELETE, HEAD vs. Tunneling every things through GET and POST
Description: The correct use of POST, GET, PUT, DELETE, or HEAD pattern is occurred, whether the following principles are correctly considered by the API developer: [129]: - GET must be used to retrieve a representation of a resource - POST must be used to create a new resource in a collection or to execute controllers - HEAD should be used to retrieve response headers - PUT must be used to both insert and update a stored resource - DELETE must be used to remove a resource In contrast, the Tunneling every things through GET and POST anti-pattern can occur if the API developer relies only on GET or POST methods to execute any kind of actions or operations including deleting, updating or creating a resource. In general, the occurrence of this anti-pattern may lead to several problems: violation of the semantic purpose of each HTTP method, the crawlers from search engines can cause inappropriate side effects [126].

Table 4.3: Error handling (Anti)Patterns

1. Supporting Status Code vs. Ignoring Status Code
Description: The status codes in REST APIs from the classes 2xx, 3xx, 4xx, and 5xx allow servers and clients to communicate in a semantic way. Supporting status code pattern can occur when the provided status code in the response is correct. In general, the correct use of status codes should be as follows [129]: 200 (OK) should be used to indicate non-specific success 200 (OK) must not be used to communicate errors in the response body 201 (Created) must be used to indicate successful resource creation 202 (Accepted) must be used to indicate successful start of an asynchronous action 204 (No Content) should be used when the response body is intentionally empty 302 (Found) should not be used 304 (Not Modified) should be used to preserve bandwidth 400 (Bad Request) may be used to indicate non specific failure 401 (Unauthorized) must be used when there is a problem with the clients credentials 403 (Forbidden) should be used to forbid access regardless of authorization state 404 (Not Found) must be used when a client's URI cannot be mapped to a resource 405 (Method Not Allowed) must be used when the HTTP method is not supported 406 (Not Acceptable) must be used when the requested media type cannot be served 409 (Conflict) should be used to indicate a violation of resource state 500 (Internal Server Error) should be used to indicate API malfunction In contrast, the wrong or unsupported status codes in REST APIs lead to Ignoring Status Code anti-pattern. Consequently, this would decrease the reusability, and hinder the loose coupling and good interoperability of these APIs.

Table 4.4: HTTP Header (Anti)Patterns

1. Supporting Caching vs. Ignoring Caching
Description: REST developers and clients often prefer to not use the caching capability as its implementation is complex. Nevertheless, caching capability is considered as one of the fundamental REST constraints [124,129]. Supporting Caching pattern appears when the API developer does not indicate no-cache or no-store for Cache-Control parameter or specifies an ETag in the response header. Otherwise, the Ignoring Caching anti-pattern can take place. As a result of this anti-pattern, throughput and scalability in requests-per-second would be decreased, which degrades the overall performance.
2. Supporting MIME Types vs. Ignoring MIME Types
Description: The server should allow defining resources in different format, including xml, json, pdf, etc., which in turn may enable clients to develop a more adaptable service consumption using diverse languages. Supporting MIME Type pattern appears when the server supports multiple resource representation formats. In contrast, the Ignoring MIME Types anti-pattern can occur when the server relies on a unique representation or uses personalized formats. Consequently, this restricts the accessibility, reusability as well as the readability of the resources [124,129].

Table 4.5: Hypermedia (Anti)Patterns

1. Supporting Hypermedia vs. Forgetting Hypermedia
Description: Hypermedia provides the way of linking resources together. Supporting Hypermedia pattern occurs when the API developer includes consistent links within the resource representations. In contrast, the absence of links within these representations leads to Forgetting Hypermedia anti-pattern. Consequently, the dynamic communication between clients and servers would be decreased because the servers do not provide for clients any link to follow.

4.2.2 OCCI (Anti)patterns

OCCI (anti)patterns represent the good and bad practices of the presented guidelines in the OCCI RESTful Protocol [113]. To identify them, we conducted an analysis study with 6 participants (1 computer science Professor, 2 computer science PHD students, 1 master student, 1 computer science post-doc, 1 engineer) that are familiar with OCCI standard. We devised the 6 participants into two groups of 3 participants each. We asked each group to manually analyse each textual description of each guideline provided in the OCCI specification and identify the appropriate patterns and anti-patterns. After this task, we have organized a meeting between two groups to share and to discuss the finding results. Finally, as a result of this study, we have considered all OCCI patterns and anti-patterns that have been commonly identified by both groups and extracted from each OCCI guideline that must (or should) be followed.
Ultimately, three categories of OCCI (anti)patterns have been distinguished: Cloud OCCI REST Related (Anti)Patterns, Structure Related (Anti)Patterns, Management Related (Anti)Patterns.

- OCCI REST Related (Anti)Patterns: They represent the poor and good practices related to REST API components. In contrast to general REST (Anti) patterns defined previously (section 4.2.1), OCCI REST (Anti) Patterns are defined according to OCCI standard. We identify 3 OCCI REST (anti)patterns that relate to the URL, response header and request header (see Table 4.6).

- Cloud Structure Related (Anti)Patterns: They represent the poor and good practices to link cloud resources between each other as well as to create a collection of resources using a Mixin, with respect to OCCI perspective (see Table 4.7).

- Management Related (Anti)Patterns: They represent the poor and good practices in the main management operations applied on cloud resources and services, with respect to OCCI perspective (see Table 4.8). We define 6 patterns and their corresponding anti-patterns respectively in Query interface, Create, Retrieve, Update, Delete operations and in Trigger actions.

Here, it should be noted that, in contrast to REST anti-patterns, where the occurrence of each one may lead to a specific impact, the occurrence of each OCCI anti-pattern would restrict the discovery and the interoperability of cloud resources [113].

Table 4.6: OCCI REST Related (Anti)Patterns

1. Compliant URL vs. Non-Compliant URL
Description: A URL path should be compliant, i.e. whenever the URL path is rendered it must be either a string or as defined in RFC6570 [113]. The non-Compliant URL anti-pattern occurs when one of these guidelines is ignored.
2. Compliant Request Header vs. Non-Compliant Request Header
Description: A Request Header can be considered compliant, i.e. client (e.g. OCCI client) : - should specify the media types its implementation data formats (e.g. OCCI Data formats) support in the Accept header, - must specify the implementation (e.g. OCCI version) version number in the User-Agent header, - must specify the media type its implementation data format (e.g. OCCI data format) support in the Content-type header [113]. The *Non-Compliant Request Header anti-pattern* occurs when one of these guidelines is ignored.
3. Compliant Response Header vs. Non- Compliant Response Header
Description: A Response Header can be considered compliant, i.e. a server (e.g. OCCI server): - should specify the media types its implementation data formats (e.g. OCCI Data formats) support in the Accept header, - must specify the media type its implementation data format (e.g. OCCI data format) used in an HTTP response in Content-type header, - must specify the implementation (e.g. OCCI version) version number in the Server header [113]. The Non-Compliant Response Header anti-pattern occurs when one of these guidelines is ignored.

Table 4.7: Cloud Structure (Anti)Patterns

1. Compliant Link between Resources vs. Non-Compliant Link between Resources
Description: To create a Link between two resources, HTTP POST must be used and its kind as well as a "source" and "target" attributes must be provided. The Non-Compliant Link Anti-pattern may occur when one of these attributes is omitted.
2. Compliant Association of Resource(s) with Mixin vs. Non-compliant Association of Resource(s) with Mixin
Description: To associate a Resource with a Mixin in accordance with OCCI, the HTTP POST must be used and the URIs that uniquely identify the resources must be introduced within the request. The Non-compliant Association anti-pattern may occur when one of these guidelines is ignored.
3. Compliant Dissociation of Resource(s) From Mixin vs. Non-compliant dissociation of resource(s) From Mixin
Description: To dissociate a resource from a Mixin in accordance with OCCI, the HTTP DELETE must be used and the URIs that uniquely identify the resources must be introduced within the request. The Non-compliant Dissociation of Resource(s) anti-pattern may occur when one of these guidelines is ignored.

Table 4.8: Management Related (Anti)Patterns

1.Query Interface Support vs Missing Query Interface
Description: To be compliant with OCCI, Query interface must be implemented, which enables the client to discover all provider capabilities [113]. It defines three operations applied on Mixins, Actions and Kind, including requesting of all available Kinds, Actions and Mixins, and adding or removing a Mixin. Query interface must be found at the path /-/ on the implementation root and carried out through the HTTP method GET, POST and DELETE. The no support of query interface on all requests engenders the Missing Query Interface anti-pattern.
2. Compliant Create vs. Non-Compliant Create
Description: The creation of any OCCI entity (i.e., Resource, Mixin) should be compliant with OCCI guidelines defined in OCCI RESTful Protocol specification [113], which include the following constraints: - To create a Mixin, the HTTP POST must be used and HTTP Category term, scheme and location must be introduced in the request. - To create a Resource instance within Mixin or collections, the HTTP POST must be used, otherwise HTTP PUT must be used. Additionally, the HTTP Category rendering that uniquely defines a particular Kind instance must be provided to define the kind of a resource instance. The Non-Compliant Create anti-pattern may occur when one of these guidelines is not supported.
3. Compliant Update vs. Non-Compliant Update
Description: The update of any OCCI entity should be compliant with OCCI guidelines defined in OCCI RESTful Protocol specification [113], which includes specifically the following : - To fully update a Mixin, the HTTP PUT must be used and all URIs defined in the collection of Mixin must be specified within the update request. - To partially update a Resource or Link, the HTTP POST must be used and the new information that will be updated must be specified within the update request. - To fully update a Resource or Link, the HTTP PUT must be used in the request. The Non-Compliant Update anti-pattern may occur when one of these guidelines is poorly adopted.
4. Compliant Delete vs. Non-Compliant Delete
Description: The Delete of a Resource, Mixin or Link should be compliant with OCCI guidelines defined in OCCI RESTful Protocol specification [113]. These guidelines specifically include the following: - To remove a Mixin, HTTP DELETE must be used and the Mixin URI defining the Mixin that will be deleted, must be defined in the request. - To delete a Resource or Link below a given path or only one, the HTTP DELETE must be used and only the URI identifying the entity that will be removed must be provided. The Non-Compliant Delete anti-pattern may occur when one of these guidelines is poorly adopted.
5. Compliant Retrieve vs. Non-Compliant Retrieve
Description: The retrieve of a Resource or Link should be compliant with the OCCI guidelines defined in the OCCI RESTful Protocol specification [113]. These guidelines specifically include the following:

- To retrieve a Resource or Link instance, the HTTP GET must be used and the server must provide as a response the HTTP Category associated with a set of attributes that identify the resource or link kind.
- To retrieve all Resources belonging to kind or mixin, the HTTP GET must be used and a list comprising all instances of a resource that belonging to the requested mixin (or kind) must be returned in the response.
The Non-Compliant Retrieve anti-pattern may occur when one of these guidelines is poorly adopted.

6. Compliant Trigger Action vs. Non-Compliant Trigger Action
Description: To trigger an action on a Resource or Link while following the OCCI guidelines, the HTTP POST must be provided and the URI must contain a query with the action term. Additionally, the specific HTTP Category that identifies the applied action, must be also introduced in the request. The *Non-Compliant Trigger Action anti-pattern* may occur when one of these guidelines is poorly adopted.

4.3 Approach Overview

After defining OCCI and REST patterns and anti-patterns, in this section, we describe our approach to detect their occurrences. The proposed approach relies on ontologies with the aim of formally specifying OCCI and REST (anti)patterns, ensuring their automatic detection and providing a set of correction recommendations in case of any anti-pattern detection. As shown in Fig. 4.1, the proposed approach proceeds in four steps as follows:

Step 1. Definition of OCCI/REST (Anti)Patterns: This step consists in defining the basic ontology *(Anti)Patterns Ontology* allowing a semantic specification of OCCI and REST patterns and anti-patterns. The proposed ontology embodies the most important and relevant concepts needed for the detection and the recommendation purposes.
Step 2. Analysis and Definition of Detection Rules: This step aims to analyze the textual definitions of OCCI and REST (anti) patterns detailed in Section 4.2 for eliciting their pertinent properties. These pertinent properties will be then exploited to specify the semantic rules to detect patterns and anti-patterns as well as the explanations of suggested recommendations. Both (Anti)Patterns Ontology and detection rules form the knowledge base (KB), which will be later interrogated through SPARQL queries for the detection and recommendation purposes.
Step 3. Detection of (Anti) Patterns: This step aims at checking the compliance of the selected Cloud REST API with both REST and OCCI best principles while suggesting appropriate recommendations in case of the anti-pattern detection. As shown in Fig. 4.1, this step includes the following two phases:

- *Check compliance for REST principles:* This step aims at applying on Cloud REST API the REST detection rules specified in Step 2 to detect REST (anti)patterns. If

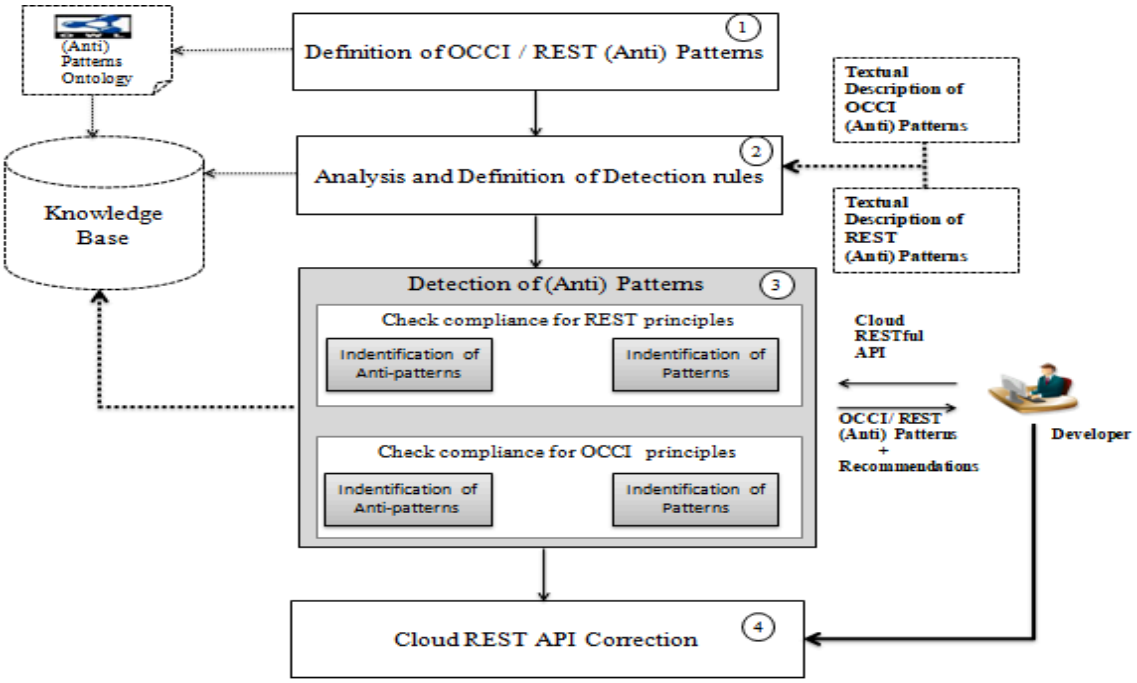


Figure 4.1: Approach overview

no REST anti-pattern is detected, the evaluated REST API is considered as REST-compliant and details related to REST patterns will be provided to the Cloud API developer for analysis or understanding purposes. Otherwise, details related to both REST patterns and anti-patterns will be displayed. While regarding REST anti-patterns, a set of recommendations is also returned to avoid their occurrence.

- *Check compliance for OCCI principles:* This step aims at applying, on cloud REST API, the OCCI detection rules specified in Step 2 to detect OCCI (anti)patterns. If no OCCI anti-pattern is detected, the evaluated REST API is considered as OCCI-compliant and details related to OCCI patterns will be given to the Cloud API developer for analysis or understanding purposes. Otherwise, details related to both OCCI patterns and OCCI anti-patterns along with a set of suggested recommendations to avoid the OCCI anti-patterns occurrence, will be displayed. It is worth mentioning that the assessed REST API can be considered as OCCI and REST compliant when it does not contain any REST and OCCI anti-pattern.

Step 4. Cloud REST API Correction: This step allows developers to manually revise their API by following the obtained recommendations on REST and OCCI anti-patterns in order to avoid their occurrences observed in Step 3. Here, it should be noted that the developer can take into consideration or ignore the suggested recommendations basing on the relevance of the detected anti-patterns.

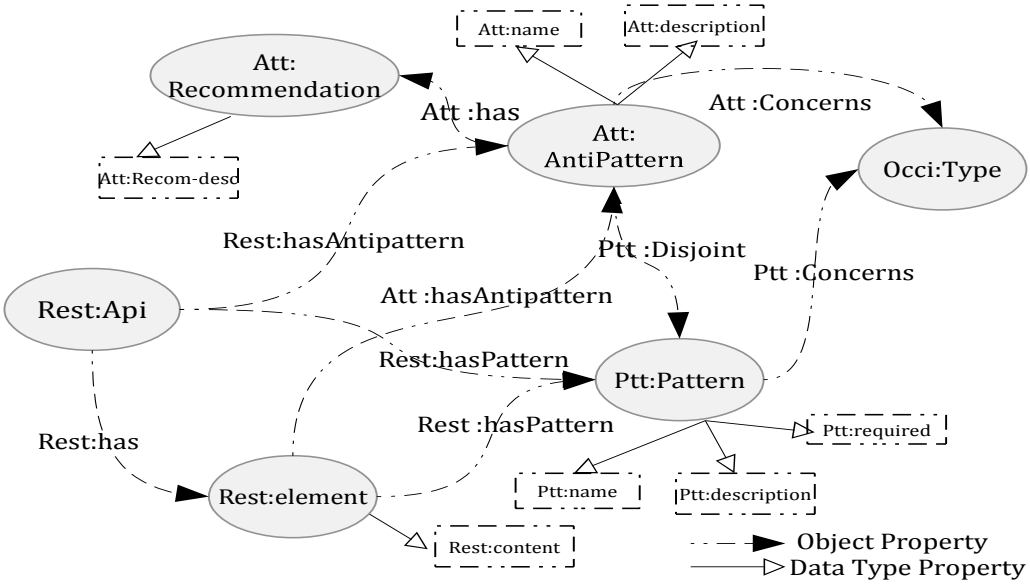


Figure 4.2: (Anti) Patterns Ontology

In the following sections, we detail the first three steps of our approach namely, definition of OCCI/REST (Anti)Patterns, analysis and definition of detection rules, and detection of (Anti)Patterns.

4.3.1 Definition of OCCI/REST (Anti)Patterns

In this step, a domain analysis on RESTful API documentations and the OCCI specification for cloud resources is performed with the aim of building the *(Anti) Patterns Ontology*. The *(Anti) Patterns Ontology* aims at providing a semantic specification of OCCI and REST (anti) patterns using the common Web Ontology Language (OWL) [7]. It is defined as a collection of four ontologies, which are mainly: Anti-Pattern Ontology, Pattern Ontology, and OCCI Ontology and REST API Ontology.

Pattern Ontology: The Pattern Ontology, as depicted in Fig. 4.2, defines the relevant information describing OCCI and REST pattern through the attributes that are associated to its core concept *Ptt:Pattern*. Those attributes (i.e. correspond to data type properties in OWL language) are *Ptt:name*, *Ptt:description* and *Ptt:required*, which represents a boolean value indicating that the given pattern is required or no. In addition, the *Ptt:Pattern* concept holds two relationships: *Ptt:Disjoint* and *Ptt:Concerns*. The *Ptt:Disjoint* relationship defines the opposite anti-pattern for a given pattern. The *Ptt:Concerns* relationship depicts that a pattern relate to a given OCCI Type (i.e. Resource, Link, etc.). Finally, the *Ptt:Pattern* concept also

represents the range of *Rest:hasPattern*, indicating that a given API or its elements can have this pattern.

Anti-Pattern Ontology: As depicted in Fig. 4.2, the definition of Anti-Pattern Ontology is similar to the Pattern Ontology. However, as opposed to OCCI or REST patterns, we use OCCI or REST anti-patterns to capture a bad practice of such OCCI or REST principles. Each anti-pattern denoted by Att: AntiPattern , is defined by Att:name and Att:description, and associated with the concept Att:Recommendation through the Att:has relationship. Att:Recommendation defines the suggested recommendation to avoid the anti-pattern that is associated with once it occurs. More precisely, it contains Att:Recomm-desc that provides a textual explanation explaining which best principle is not respected, that leads to the occurrence of the associated anti-pattern and suggesting as a result an advice to avoid it.

REST API Ontology: The REST API ontology allows providing a seman-

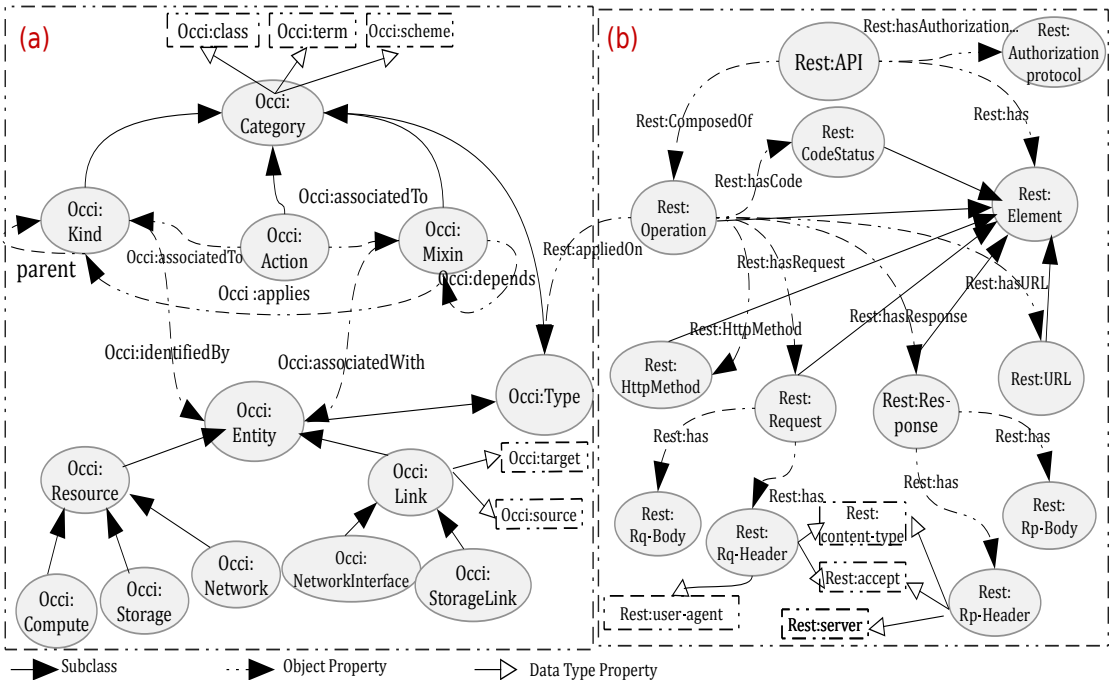


Figure 4.3: (a) OCCI Ontology; (b) REST API Ontology

tic definition of the functional and structural features of the REST APIs. To build this ontology, we examine several documentation on REST APIs while considering OCCI RESTful API into account. The principal concept is *Rest:API* representing a REST API, which is linked, as shown in Fig. 4.3 (b), to the following concepts: *Rest:AuthorizationProtocol*, *Rest:Element*, *Rest:Operation*. *Rest:AuthorizationProtocol* concept defines the authorization protocol used to access the REST API. The *Rest:Element* describes through its subclasses the most

important components that we can find in a REST API, including, response header, request header, code status, URL, operation, response, request, response body, request body etc. Finally, we use the *Rest:Operation* concept to define the possible operations that can be applied on cloud resources e.g., *Create a Server*.

OCCI Ontology: All RESTful API operations are applied on OCCI types (i.e. Resource, Link, etc.) that are already specified both in OCCI Infrastructure [104] and OCCI Core [114]. To allow such capability, we specify the OCCI ontology that presents a semantic specification of the cloud resources abstraction provided in these two specifications while following the OCCI Rendering syntax of these resources [58, 115]. As shown in Fig. 4.3 (a), the heart of this ontology is the *Occi:Resource* concept, which in turn associated with three concepts: *Occi: Compute*, *Occi: Network* and *Occi: Storage*. Accordingly, a cloud resource may be a virtual switch, a virtual storage, a virtual sever, etc. Additionally, *Occi:Resource* is associated to the *Occi:Link* concept, which used to link one resource instance with another. We distinguish two type of link *Storage Link* and *Network Interface*. Each type is described through two attributes (e.g. *Occi:source*, *Occi:target*). Both *Occi:Resource* and *Occi:Link* inherit the *Occi:Entity* concept. The *Occi:Kind* is the core of the classification type system built into the OCCI Core Model. *Occi:Kind* is a specialization of *Occi:Category* and introduces additional capabilities in terms of actions. *Occi:Action* describes a set of operations applicable to an entity instance. The last type specified by the OCCI Core Model is the Occi:Mixin, which allows extending the OCCI entity by plugging in/out a set of attributes and actions. An instance of Mixin can be attached to an entity instance, which may provide additional capabilities at run-time [114].

4.3.2 Analysis and Definition of Detection rules

In this step, the textual definitions of the (anti)patterns listed in Tables 4.1 to 4.8 were analyzed with the aim of eliciting their pertinent properties. We then exploit these properties to specify the semantic rules needed to detect (anti)patterns. To do so, we adopt SWRL language to express these rules. The syntax used to express SWRL rules has been already explained in Chapter 2 Section 2.3.1 (refer to the paragraph on SWRL and SQWRL rules languages) .
Listings 4.1 and 4.2 illustrate, respectively, the SWRL rules for Verbless URI pattern and their anti-pattern CRUDy URIs.

Listing 4.1: SWRL rule for Verbless URIs Pattern

```
1.Rest:Operation(?operation) ^ Rest:hasHttpMethod(?operation, ?httpmethod) ^
    Rest:verb(?httpmethod, ?verb) ^ detection: matchesOne(?verb, "POST", "PUT",
     "GET", "DELETE") ^ Rest:hasURL(?operation, ?url)^ Rest:value(?url,?urlval)
    ^ detection:contain(?return,?urlval, "create", "update", "read", "delete")
    ^ swrlb:matches(?return, "False") → Rest:hasPattern(?operation, Ptt:
```

```
    Verbless_URIs)
```

As shown in Listing 4.1, the SWRL rule for the Verbless URIs pattern aims at evaluating an operation of each request in the API (i.e., Rest:Operation(?operation)). This evaluation consists of checking whether the verb of HTTP method (i.e., Rest:verb(?httpmethod, ?verb)) included in the operation contains one of the HTTP common verb i.e., POST, PUT, GET, DELETE. This is accomplished through our custom built-in "detection: matchesOne(?verb, "POST", "PUT", "GET", "DELETE")". Additionally, the occurrence of this pattern also requires that the URL value (i.e., Rest:value(?url, ?urlval) does not contain one of the common CRUDy terms i.e., create, update, read, delete. The detection:contain built-in consists of checking whether the URL value contains one of the CRUDy terms and returns a boolean value as a result. This value will then be checked through swrlb:matches to make sure that its value corresponds to the "False" term.

Listing 4.2: SWRL rules for CRUDy URIs Anti-pattern

```
1 1.Rest:Operation(?operation) ^ Rest:hasHttpMethod(?operation, ?httpmethod) ^
      Rest:verb(?httpmethod, ?verb) ^ detection:matchesOne (?verb, "create", "
      update", "read", "delete") → Rest:hasAntipattern (?operation, Att:
      CRUDy_URIs)
2 2.Rest:Operation(?operation) ^  Rest:hasURL(?operation, ?url) ^ Rest:value(?url
      ,?urlval) ^ detection:contain(?return,?urlval, "create", "update", "read",
      "delete")  ^ swrlb:matches(?return, "True")  → Rest:hasAntipattern (?
      operation, Att:CRUDy_URIs)
```

In contrast, as shown in Listing 4.2, the CRUDy URIs anti-pattern can be detected through two SWRL rules. The first rule consists of checking for each operation, through the built-in detection:matchesOne (?verb, "create", "update", "read", "delete"), whether the used verb of HTTP method contains one of the common CRUDy terms. The positive satisfaction of this condition indicates the occurrence of CRUDy URIs anti-pattern. The second rule consist of evaluating the URL of each operation defined in an API in order to detect whether its value contains one of the common CRUDy terms. Both detection:contain and swrlb:matches are exploited to carry out this detection. The existence of one of the CRUDy terms in the URL value results in the occurrence of CRUDy URIs anti-pattern in the analyzed operation.

Listing 4.3: SWRL rule for Compliant Link between Resources Pattern

```
1 1. Rest:Operation(?op) ^ Rest:hasHttpMethod(?op, ?httpmd) ^Rest:verb(?httpmd, ?
      verb) ^ swrlb:matches(?verb, "POST") ^ Rest:hasRequest(?op,?req) ^ Rest:has
      (?req, reqbody)^ Rest:hasParameterDefinition(?reqbody, ?pradef) ^ Occi:Link
      (?link) ^Rest:isComposedOf(?pradef, ?link)^ Occi:identifiedBy(?link, ?kind)
       ^ Occi:term(?kind, ?term) ^ detection:matches(?term, "Storagelink", "
      Network Interface")^ Occi:scheme(?kind, ?schee) ^ Occi:class(?kind, "kind")
       ^ Occi:source(?link, ?source) ^ Occi:target(?link, ?target)  →  Rest:
      hasPattern(?op, Ptt:Compliant_Link)
```

In the same way, as shown in Listing 4.3, we define a SWRL rule to detect Compliant Link between Resources Pattern, which aims at checking for each link operation the verb of HTTP Method , the Kind of link and whether its source and target attributes do exist or not. A given link operation is reported as it has Compliant Link between Resources Pattern if we ensure that the used HTTP verb is "POST", the Kind term *?term* has either "Network Interface" or "Storage Link", the Kind scheme (?kind, ?scheme) is not empty, the Kind class ?class has as value "kind". Finally, the link source and target attributes of the concerned link should not contain empty values. Fig. 4.4 shows a partial instantiation of the (Anti) Pattern Ontology with information that we have extracted from a REST operation in the OOi RESTful API[2] in order to add a storage link between a volume and an instance of a VM. Once the detection rule for Compliant Link pattern was executed, the relationship "Rest:hasPattern" illustrated with red color, was instantiated between the Ptt:Compliant_ Link (Ptt:Pattern instance) and the Rest:Link Volume to Instance (REST:Operation instance).

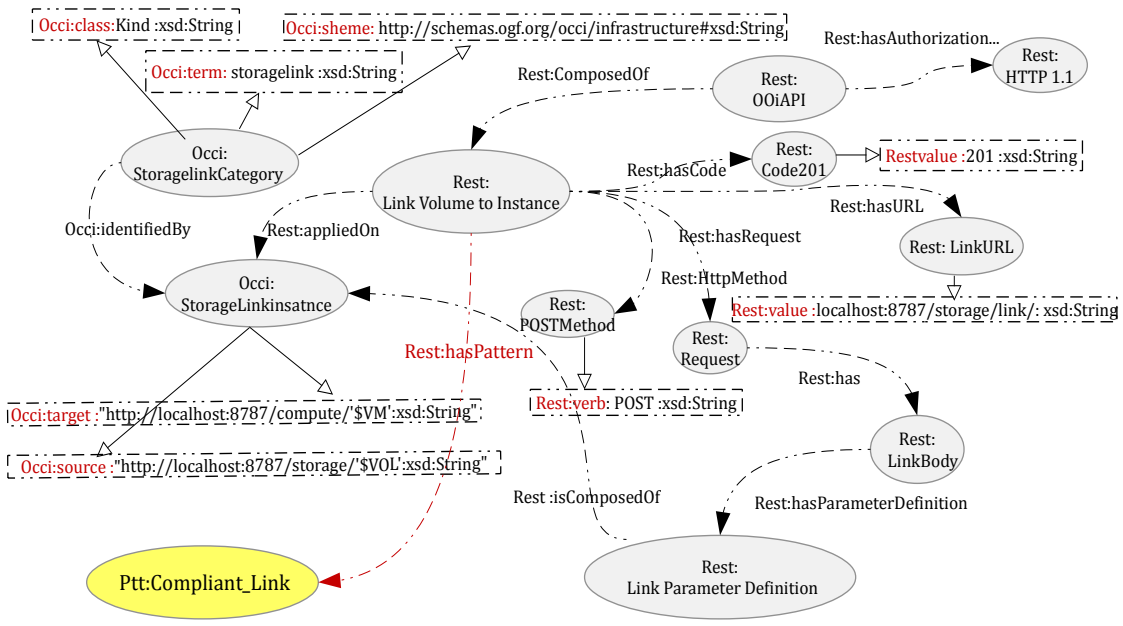


Figure 4.4: A partial instantiation of the (Anti) Pattern Ontology with information extracted from a REST operation in the OOi API, shows the impact after executing the SWRL rule for the *Compliant Link between Resources Pattern*

[2]Available at http://ooi.readthedocs.io/en/stable/user/usage.html

Listing 4.4: SWRL rules for Non-Compliant Link between Resources Anti-pattern

```
1.Rest:Operation(?op) ^Rest:type (?op, "linkResource")^Rest:hasHttpMethod(?op,
    ?httpmd) ^Rest:verb(?httpmd, ?verb)^swrlb:notEqual(?verb, "POST") → Rest:
    hasAntiPattern(?op, Att:Non-Compliant_Link)

2. Rest:Operation(?op) ^Rest:type (?op, "linkResource")^ Rest:hasRequest(?op,?
    req)^Rest:has(?req,?reqbody) ^Rest:hasParameterDefinition(?reqbody, ?pradef
    ) ^Occi:Link(?link) ^Rest:isComposedOf(?pradef, ?link) ^ Occi:identifiedBy
    (?link, ?kind) ^ Occi:term(?kind, "") → Rest:hasAntiPattern(?op, Att:Non-
    Compliant_ Link)

3.Rest:Operation(?op) ^ Rest:type (?op, "linkResource")^ Rest:hasRequest(?op,?
    req)^ Rest:has(?req,?reqbody) ^ Rest:hasParameterDefinition(?reqbody, ?
    pradef) ^ Occi:Link(?link) ^Rest:isComposedOf(?pradef, ?link) ^ Occi:source
    (?link, "") → Rest:hasAntiPattern(?op, Att:Non-Compliant_ Link)

4.Rest:Operation(?op) ^ Rest:type (?op, "linkResource") ^ Rest:hasRequest(?op,?
    req) ^ Rest:has(?req, ?reqbody) ^ Rest:hasParameterDefinition(?reqbody, ?
    pradef) ^ Occi:Link(?link) ^ Rest:isComposedOf(?pradef, ?link) ^ Occi:
    target(?link, "") → Rest:hasAntiPattern(?op, Att:Non-Compliant_ Link)
```

Contrariwise, Non-Compliant Link between Resources Anti-pattern is reported if at least one of the above listed practices was not respected. As shown in Listing 4.4, we specify four SWRL rules defining the different symptoms needed to detect this anti-pattern.

4.3.3 Detection of OCCI/REST (Anti)Patterns

In this section, we aim at evaluating the compliance of the selected Cloud REST API with both REST and OCCI best principles by detecting both patterns and anti-patterns. Also, in case of any anti pattern detection, we intend to provide developers with a set of correction recommendations to help them revise and correct their APIs. As mentioned above, this step involves two phases: Check compliance for REST principles and Check compliance for OCCI principles.
The first phase allows applying, on the Cloud REST API, the SWRL detection rules defined above to detect both REST patterns and anti-patterns, along with a set of SPARQL queries. These SPARQL queries are used to obtain the related details on each detected REST pattern and anti-pattern, as well as the suggested recommendation to avoid the detected anti-pattern. To do that, we firstly propose a REST pattern detection algorithm with the aim of performing the REST pattern detection.

Listing 4.5: The REST Pattern Detection Algorithm

```
Input:(Anti)patternOntology.owl, SWRL_Detection_Rules_List_For_RESTPatterns,
    SPARQL_queries_RESTpatterns_list
Output:REST Patterns_List with relevant Details
begin
  For (each SWRL_rule in SWRL_Detection_Rules_List_For_RESTPatterns) do
  {
    run(SWRL_rule)
```

```
  }
  For (each query in SPARQL_queries_RESTpatterns_list) do
  {
    result=execute(query)
    if( Exist(result) ) then
      Display("the pattern detected and related details")
    else
      Display ("no pattern is detected")
  }
end
```

As shown in Listing 4.5, the provided algorithm takes as input (Anti) Pattern Ontology, the list of SWRL rules to detect REST patterns and the list of SPARQL queries to return relevant details related to each REST pattern. As output, it returns the list of patterns with relevant details, which may help developers in the analysis or understanding. During the execution, the proposed algorithm proceeds as follows: apply the inference process that allows executing all SWRL rules using the Drools engine[3] to detect REST patterns (lines 4-7), execute all SPARQL queries defined in the list using the Pellet reasoner[4], which return for each pattern the relevant details once it is detected or "no pattern detected" text otherwise (lines 8-15). All the SPARQL queries were defined in the same way, but they differ only in respect to the name of the pattern that would be interrogated. In Listing 4.6, we present a SPARQL query that returns the Verbless URIs pattern, each REST element that contains this pattern and the related contents.

Listing 4.6: SPARQL Query to retrieve the relevant details for Verbless URIs pattern

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX base: <http://www.semanticweb.org/asus/ontologies/2017/5/APatterns-
    ontology#>

SELECT distinct ?pattern ?restelement ?content
WHERE {?restelement base:Rest:content ?content. ?restelemnt base:Rest:
    hasPattern ?pattern. ?pattern base:Ptt:name ?name.
FILTER(?name="Ptt:Verbless_URIs"^^xsd:string)
}
```

Like the REST pattern detection, we propose a REST anti-pattern detection algorithm based on a set of SWRL rules and SPARQL queries. As shown in Listing 4.7, our algorithm takes as input (Anti)Pattern Ontology, the list of SWRL rules to detect REST anti-patterns and the list of SPARQL queries to return relevant details related to each REST anti-pattern along with suggested recommendations to avoid its occurrence. As output, it provides the detected anti-patterns associated with relevant

[3]Drool: is a business rule management system that is based on forward and backward chaining inference to produce and execute rules. More details can be found at http://www.drools.org/

[4]https://www.w3.org/2001/sw/wiki/Pellet

details and suggested recommendations which can be further exploited by cloud API developers to correct these anti-patterns.
Similar to the execution process within the above algorithm, SWRL rules would be firstly executed to infer the existence of anti-patterns on the selected API (lines 4-7). Then, the SPARQL queries would be executed with the aim of providing the detected anti-patterns associated with relevant details and suggested recommendations that are useful for the correction purpose (lines 8-15). For example, Listing 4.8 shows the SPARQL query that we defined for the CRUDy URI anti-pattern. It returns each REST element that has this anti-pattern and its related contents as well as the following textual recommendation: "*Method name or resource URL contains one of the following terms âreadâ, âcreateâ, âdeleteâ, âupdateâ. Please change the given value into one from the following terms: âPostâ, âGetâ, âDeleteâ or âPutâ*". Thus, the cloud API developer may use these details and the suggested recommendations to revise her API. It is worth mentioning that the developer can take into consideration the detected anti-patterns and therefore the correction step can take place. Otherwise, the developer can ignore the detected anti-patterns because she thought that they are not relevant.

Listing 4.7: The REST Anti-pattern Detection Algorithm

```
Input:(Anti)patternOntology.owl, SWRL_Detection_Rules_List_For_RESTAntiPatterns
    , SPARQL_queries_REST_Antipatterns_list
Output:REST AntiPatterns_List with relevant Details
begin
  For (each SWRL_rule in SWRL_Detection-Rules-list for antipattern) do
  {
    run(SWRL_rule)
  }
  For (each query in SPARQL_query_antipattern_list) do
  {
    result=execute(query)
    if( Exist(result) ) then
      Display("Detected anti-pattern with Relevant Details and Recommendation
    Text)
    else
      Display("no Anti-pattern is detected")
  }
end
```

Listing 4.8: SPARQL Query to retrieve the relevant details and suggested recommendations for CRYDy URI Anti-pattern

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX base: <http://www.semanticweb.org/asus/ontologies/2017/5/APatterns-
    ontology#>

```

```
SELECT distinct ?antipattern ?restelement ?content ?recommendation
WHERE {?restelement base:Rest:content ?content. ?restelemnt base:Rest:
    hasAntipattern ?antipattern. ?antipattern base:Att:
    hascorrectionrecommendation ?recommendation. ?antipattern base:Att:name ?
    name.
FILTER(?name="Att:CRYDy_URI"^^xsd:string)
}
```

After checking the compliance with REST principles via detecting both REST patterns and anti-patterns, checking the compliance with OCCI principles can take place. Like REST (anti) patterns, we conduct the detection of OCCI (anti) patterns using two algorithms while relying on both SWRL rules and SPARQL queries.

4.4 Experiments and Validation

In this section, we discuss the evaluation of our proposed approach. We conduct this evaluation through a proof of concept implementation and a validation study on a dataset that we built based on a set of cloud management APIs. In fact, our evaluation objective is twofold. Firstly, we assess the compliance of Cloud REST APIs with both OCCI and REST principles. Then, we show the effectiveness of our approach by analyzing the accuracy of the detection rules and the usefulness of the provided detection and recommendation support. In the following, we firstly present the proof of concept ORAP-Detector. Secondly, we describe the hypotheses and the experimental setup followed to conduct the validation of our approach. Finally, we analyze and interpret the experiment results.

4.4.1 Proof of Concept: ORAP-Detector

ORAP-Detector [34] is provided as a web application based on J2EE solutions to support the (anti) patterns detection step described in Section 4.3.3. It provides to API developers two main functionalities, which are mainly: Check compliance for REST principles and Check compliance for OCCI principles. Moreover, we relied on three semantic Web APIs namely Pellet API[5], Apache Jana API[6] and SWRL API[7] to implement the different algorithms described above for the detection of (anti) patterns. These APIs are exploited to handle SPARQL/SQWRL queries and SWRL rules. Moreover, for the evaluation purpose, our ORAP-Detector was deployed on top of the AWS EC2 service [14].

4.4.2 Hypotheses and Experimental Setup

Hypotheses. To evaluate the effectiveness of the proposed approach, we formulate the following hypotheses:

[5] https://github.com/stardog-union/pellet
[6] https://jena.apache.org/
[7] https://github.com/protegeproject/swrlapi

Table 4.9: List of the 5 analyzed Cloud APIs and their on-line documentations

Cloud RESTful APIs	On-line documentations
OOi RESTful API	http://ooi.readthedocs.io/en/stable/user/usage.html
COAPS RESTful API	http://www-inf.it-sudparis.eu/SIMBAD/tools/COAPS/
OpenNebula OCCI RESTful API	http://archives.opennebula.org/documentation:archives:rel4.0:occiddoverview
Amazon S3 RESTful API	http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
Rackspace RESTful API	https://developer.rackspace.com/docs/cloud-servers/v2/api-reference/

- *H1 (Accuracy).* The set of all defined rules have an average precision, recall and F-measure of more than 85%. This value is chosen based on the threshold base used in the previous detection methods [85, 108, 124–126], which have been considered as effective.
- *H2 (Usefulness).* The key detection rules and provided recommendations are useful and relevant.

Experimental Setup. We perform an analysis in the Cloud RESTful APIs of cloud services to build the experimental dataset. As shown in Table 4.9, 5 candidates including OOi, COAPS, OpenNebula, Amazon S3 and Rackspace were selected. This selection has been done because their associated REST operations are well illustrated. Using the operations details existing in each API, we have collected all the requests and responses that build the required knowledge for semantically describing each API. Regarding the accuracy evaluation, we have to build our truth knowledge. Suitably, we manually evaluated the REST operations in order to identify the true positives and false negatives required to compute precision, recall and F1-measure values. Precision is the ratio between the true detected (anti) patterns and detected (anti) patterns reported as positive. Recall is the ratio between the true detected (anti) patterns and all existing true (anti) patterns. Finally, the F1-measure defines the weighted harmonic mean of the precision and recall values.
Moreover, the usefulness is determined via a questionnaire that provides the participants feedbacks about our detection rules and suggested recommendations in case of any anti-pattern detection. Participants were recruited from software engineering experts who have sophisticated understanding of REST and Cloud APIs. The questionnaire consists of two main parts: Usefulness evaluation and Insights/Improvements. The usefulness evaluation questions aim at evaluating whether the detection rules and suggested recommendations are useful and relevant. Whereas, the Insights/Improvements provide some suggestions for improving our detection and recommendation support. The detailed definition of the list of questions is provided in Appendix A.

Table 4.10: Detection results of the REST (Anti) Patterns

Cloud REST API	**OOi (28)**	**COAPS (18)**	**Open-Nebula OCCI (20)**	**Amazon S3 (71)**	**Rack-space (72)**	**O.P.**	**Total (209)**
URI Design (Anti) Patterns							
Tidy URIs (156/209)	6	13	17	70	50	74%	156 (74%)
Amorphous URIs (52/209)	21	5	3	1	22	52%	52 (24%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Verbless URIs (205/209)	28	16	20	69	72	98%	205 (98%)
CRUDy URIs (4/209)	0	2	0	2	0	1%	4 (1%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Singularized nodes (73/96)	6	3	6	33	25	76%	73 (34%)
Pluralized nodes (23/96)	8	3	3	6	4	23%	23 (11%)
No Detection (0/96)	0	0	0	0	0	0%	0 (0%)
Request methods (Anti) Patterns							
Correct use of POST (49/53)	8	7	3	4	27	92%	49 (23%)
Correct use of GET (74/74)	0	6	11	25	32	100%	74 (35%)
Correct use of PUT (29/29)	0	0	3	22	4	100%	29 (14%)
Correct use of DELETE (34/34)	6	3	3	13	9	34%	34 (16%)
Correct use of HEAD (5/5)	0	0	0	5	0	100%	5 (2%)
Tunneling every things through POST (4/53)	0	2	0	2	0	7%	4 (2%)
Tunneling every things through GET (0/47)	0	0	0	0	0	0%	0 (0%)
No Detection (0/199)	0	0	0	0	0	0%	0 (0%)
Error handling (Anti) Patterns							
Supporting Status Code (191/209)	28	18	20	70	55	91%	191 (91%)
Ignoring Status Code (18/209)	0	0	0	1	17	8%	18 (8%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
HTTP Header (Anti) Patterns							
Supporting Caching (71/209)	0	0	0	71	0	33%	71 (33%)
Ignoring Caching (138/209)	28	18	20	0	72	66%	138 (66%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Supporting MIME Types (188/209)	8	18	20	71	71	89%	188 (89%)
Ignoring MIME Types (21/209)	20	0	0	0	1	10%	21 (10%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Hypermedia (Anti) Patterns							
Supporting hypermedia (36/133)	8	13	0	0	15	27%	40 (17%)
Forgetting hypermedia (87/133)	0	0	14	31	42	63%	87 (41%)
No Detection (10/133)	10	0	0	0	0	7%	10 (4%)

4.4.3 Results Analysis and Findings

In this section, we present respectively the detection results of REST (anti) patterns and OCCI (anti) patterns in all selected Cloud REST APIs. Then, a compliance evaluation of the selected API with OCCI and REST best principles is described. Finally, we discuss the validation of our approach in terms of accuracy and usefulness.

4.4.3.1 Detection of REST of (anti) patterns

We present in Table 4.10 the detection results of 21 REST patterns and anti-patterns for the five Cloud RESTful APIs. The first column reports the patterns and anti-patterns, while the analyzed Cloud RESTful APIs are presented in the following columns. For each Cloud RESTful API, we show the occurrence number of each REST pattern and anti-pattern. The last three columns report respectively: the occurrence percentage (OP) of each (anti) pattern compared to the total number of

Table 4.11: Detection results of the OCCI (Anti)patterns

Cloud REST API	**OOi (28)**	**COAPS (18)**	**Open-Nebula (20)**	**Amazon S3 (71)**	**Rack-space (72)**	**O.P.**	**Total (209)**
Management Related (Anti)patterns							
Query Interface Support (0/161)	0	0	0	0	0	0%	0 (0%)
Missing Query Interface (5/161)	1	1	1	1	1	3%	5 (2%)
No Detection (0/161)	0	0	0	0	0	0%	0 (0%)
Compliant Create (6/31)	4	1	0	1	0	19%	6 (2%)
Non-Compliant Create (24/31)	0	1	3	11	9	77%	24 (11%)
No Detection (1/31)	1	0	0	0	0	3%	1 (0.4%)
Compliant Update (15/24)	3	2	3	0	7	62%	15 (7%)
Non-Compliant Update (8/24)	0	0	0	8	0	8%	8 (3%)
No Detection (1/24)	0	0	0	1	0	4%	1 (0%)
Compliant Delete (32/32)	3	2	3	14	10	100%	32 (15%)
Non-Compliant Delete (0/32)	0	0	0	0	0	0%	0 (0%)
No Detection (0/32)	0	0	0	0	0	0%	0 (0%)
Compliant Retrieve (26/90)	10	5	11	0	0	28%	26 (12%)
Non-Compliant Retrieve (64/90)	0	0	0	31	33	71%	70 (30%)
No Detection (0/90)	0	0	0	0	0	0%	0 (0%)
Compliant Trigger Action (0/21)	0	0	0	0	0	0%	0 (0%)
Non-Compliant Trigger Action (21/21)	0	7	0	3	11	100%	21 (10%)
No Detection (0/21)	0	0	0	0	0	0%	0 (0%)
Cloud Structure (Anti)patterns							
Compliant Link between Resources (4/5)	2	0	2	0	0	80%	4 (1%)
Non-Compliant Link between Resources (1/5)	0	0	0	0	1	20%	1 (0.4%)
No Detection (0/5)	0	0	0	0	0	0%	0 (0%)
Compliant Association of Resource with Mixin	-	-	-	-	-	-	-
Non-Compliant Association of Resource with Mixin	-	-	-	-	-	-	-
No Detection	-	-	-	-	-	-	-
Compliant Dissociation of Resource from Mixin	-	-	-	-	-	-	-
Non-Compliant Dissociation of Resource from Mixin	-	-	-	-	-	-	-
No Detection	-	-	-	-	-	-	-
REST Related (Anti) Patterns							
Compliant URL (209/209)	28	18	20	71	72	100%	209 (100%)
Non-Compliant URL (0/209)	0	0	0	0	0	0%	0 (0%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Compliant Req.H (0/209)	0	0	0	71	72	68%	143 (68%)
Non-Compliant Req.H (209/209)	28	18	20	0	0	66%	66 (31%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Compliant R.H (143/209)	0	0	0	71	72	68%	143 (68%)
Non-Compliant R.H (66/209)	28	18	20	0	0	100%	66 (31%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)

operations that may contain such (anti) pattern (i.e. the percentage of Correct use of POST is computed compared only with the existing POST operations) and finally the total number of occurrences of each (anti) pattern compared to the total number of all existing operations.

As specified in Table 4.10, the most commonly followed REST pattern is Verbless URIs. It's quite evident that Cloud API designers are aware of avoiding the use of any of common CRUDy terms or their equivalent. This represents a good practice to obviate confusing API client developers. Additionally, more than 91% of the analyzed operations (191 out of 209) support compliant status codes. This enhances the understandability for clients APIs. Also, it is very reassuring to observe that more than 89% (188 out of 209) of the analyzed operations follows MIME type pattern, which increases the resource or service accessibility and re-usability. In contrast, it is

frustrating to observe that the most selected Cloud APIs except Amazon S3 failed to support the caching capability, although it is one of the principle REST constraints.

4.4.3.2 Detection of OCCI (anti) patterns

Table 4.11 summarizes the detection results of 24 OCCI patterns and anti-patterns on the five Cloud RESTful APIs. As shown in Table 4.11, we observe that the most frequent patterns (*Compliant Delete* and *Compliant Update* patterns) dominate the management related (anti) patterns category. It seems that Cloud API designers follow either explicitly or implicitly the OCCI guidelines to delete and update a cloud resource. On the contrary, *Non-Compliant Trigger Action* and *Non-Compliant Create* represent the most recurrent anti-patterns in this category. In fact, most of the Cloud RESTful APIs do not specify the resource category needed to define a specific type of the resource to be created. In addition, it seems that the cloud API designers ignore both the query exposing the term of the action and its associated HTTP category defining its functionality in the REST operation to trigger an action on a resource. Regarding the Cloud Structure (Anti)Patterns category, *Compliant Link between Resources pattern* is the most commonly followed pattern. It should be noted that there is not a large number of operations we find to test this pattern as the link resource is rarely considered in the selected APIs. Moreover, we find no occurrence of (anti) patterns related to the association and dissociation of resources from Mixin. Regarding the REST anti-patterns and patterns according to OCCI perspective, 100% of the analyzed URIs are compliant with OCCI principles that are related to URI format. In addition, all analyzed request and response headers both in Amazon S3 and Rackspace are compliant with OCCI. In contrast, even though OOi, OpenNebula OCCI and COAPS are OCCI-based APIs, they failed to support compliant headers in any of their operations.

4.4.3.3 Compliance Evaluation

Herein, we aim at assessing whether the selected cloud APIs are compliant with REST and OCCI best principles by computing for each one its compliance degree. The compliance degree shows the percentage of patterns (OCCI or REST) that each API has over all its operations, which is defined as follows:

$$\textbf{Compliance degree} = \frac{1}{N} * \sum_{i=1}^{N} \left(\frac{\sum P_i}{\sum OP_Pi}\right)$$

where N is the number of patterns (e.g. 12 patterns for REST), P_i is a pattern (for instance P_1 denotes the Tidy URIs pattern), $\sum Pi$ the number of operations that really contain the pattern P_i (e.g. only 6 operations contain the Tidy URIs pattern in OOi RESTful API), $\sum OP_Pi$ is the total number of operations that may contain the pattern P_i (e.g. 28 operations that may contain the Tidy URIs pattern in OOi RESTful API).

Now, we discuss the compliance of the selected Cloud REST APIs with REST good principles before assessing their compliances with the OCCI ones. As described in Fig. 4.5, we observe that all selected API have reached acceptable REST compliance degrees as mean value for all selected APIs is greater than 50% with reasonable difference. This shows the maturity of these APIs regarding the REST best principles. Indeed, Amazon S3 API represents the most compliant API with REST best principles with 78% of compliance degree, which means that 78% of its operations properly follow the REST best principles. This is not surprising as Amazon S3 is one of important cloud leaders aiming to attract API developers to increase the use of their API. Moreover, we report that OCCI OpenNebula API reaches 70%, Rackspace reaches 69%, OOi reaches 66% and finally COAPS reach 63% as compliance degree, which are also good values.
Regarding the OCCI best principles, as illustrated in Fig. 4.6, OOi and OpenNebula OCCI APIs represent the most compliant APIs with OCCI best principles. This is not surprising because both APIs are already based on OCCI standard. However, the reached compliance degree is still not convenient enough. It would be more reassuring if future releases of OCCI plan to improve this as the OCCI REST API will be automatically generated from a meta-model instead of designed by hand. Additionally, Rackspace as well as Amazon S3 have reached 48% and 42% of compliance degree respectively. This means that over 40% of operations in those APIs implicitly follow the OCCI standard, although they have based on own model to describe cloud resources. In contrast, even though COAPS API is an OCCI-based API, it reaches only 45%, which shows that its developers did not carefully follow all OCCI best principles.

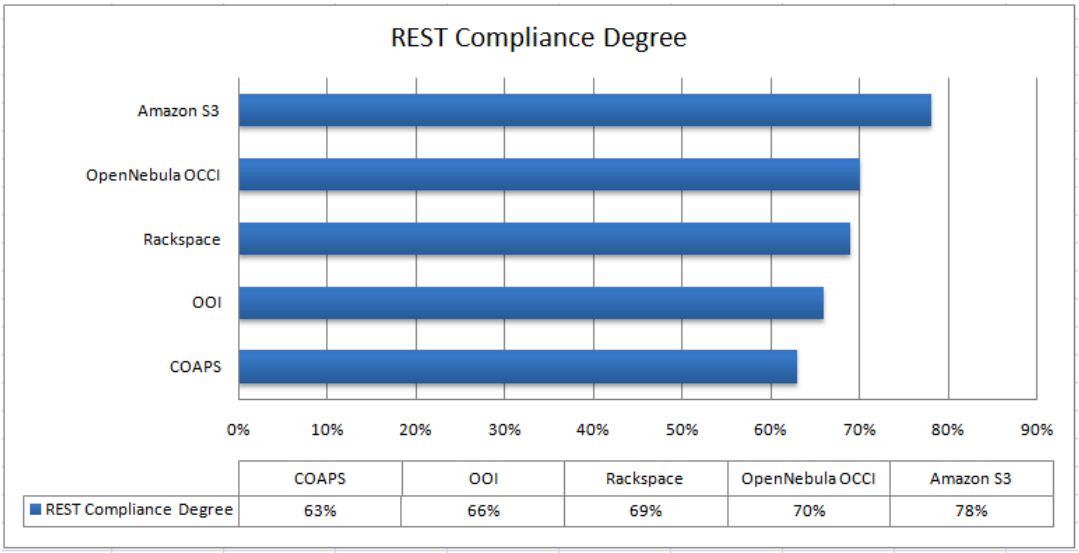


Figure 4.5: REST Compliance Degrees of Cloud RESTful APIs

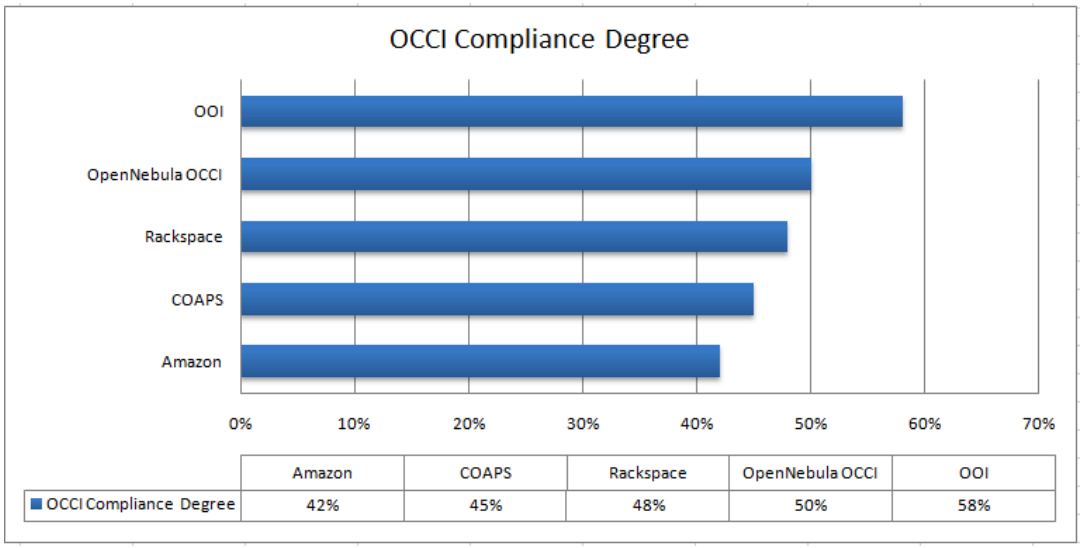


Figure 4.6: OCCI Compliance Degrees of Cloud RESTful APIs

4.4.3.4 Discussion of validation results

In this section, we aim at discussing the validation hypotheses mentioned in Section 4.4.2.

Evaluation of H1 (Accuracy). The hypotheses H1 is evaluated according to three parameters: Precision, Recall and F-measure. Two validations were conducted to evaluate this hypothesis. Table 4.12 illustrates the results of the first validation that aim at evaluating our approach validation in detecting REST patterns and anti-patterns on OOi and Rackspace RESTful APIs. The first column presents the identified (anti)patterns. The remaining columns list the two selected APIs for the validation, including Validated (i.e., the number of validated pattern (or anti-pattern) considered as true which is ensured manually), P (i.e., the number of pattern occurrences reported as positives by our detection algorithms), TP (i.e., the number of pattern occurrences reported as true positives), Precision, Average Precision, Recall and Average Recall. Finally, we report the total average of Precision, Recall and F-measure on the last two rows respectively. Regarding OOi RESTful API, it was pleasantly surprising that our REST detection algorithms allow detecting of REST patterns and anti-patterns on average with a precision of 100% and Recall of 95,1 %, signifying that all the detected (anti) patterns are in the list that we determined manually. Also, we obtain on average almost similar values for Rackspace RESTful API viz. a precision of 100 % and a recall of 91,2. On the whole, we obtain on average F-measure of 97,4 % for OOi and 95,3% for Rackspace.

Table 4.13 illustrates the results of the second validation of our approach in detecting OCCI (anti) patterns on OOi and Rackspace RESTful APIs. Similarly, as for the

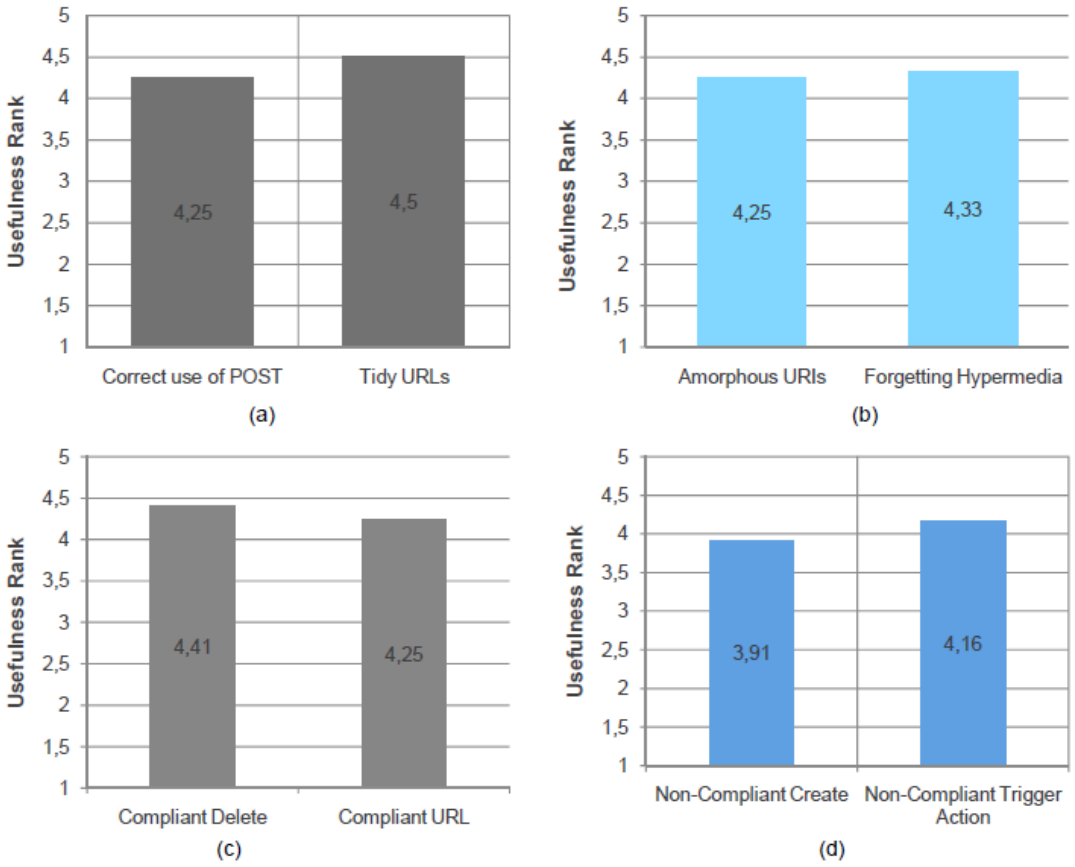


Figure 4.7: Usefulness Detection Rate for (a) REST patterns: Correct use of POST, Tidy URLs; (b) REST anti-patterns: Amorphous URIs, Forgetting Hypermedia; (c) OCCI patterns: Compliant Delete, Compliant URL; (d) OCCI anti-patterns: Non-Compliant Create, Non-Compliant Trigger Action

REST (anti) pattern detection, our approach has reached good results. More precisely, we obtain, on average, precision of 100%, recall of 97,9%, F-measure of 98,9% for OOi API and, precision of 100%, recall of 97%, F-measure of 98% for Rackspace. Accordingly, giving this observation and the above one, we confirm the truth of the first hypothesis H1 with significant difference.

Evaluation of H2 (Usefulness). In this evaluation, we ask participants to rate the usefulness of the detection of a set of OCCI/REST patterns and anti-patterns using a 0-5 scale. We examined 2 REST patterns (Correct use of POST, Tidy URLs), 2 REST anti-patterns (Amorphous URIs, Forgetting Hypermedia), 2 OCCI patterns (Compliant Delete, Compliant URL), and 2 OCCI anti-pattern (Non-Compliant Create, Non-Compliant Trigger Action). Moreover, to evaluate the detection of these (anti) patterns, we employ a set of REST operations from three different management APIs

mainly, Rackspace, COAPS and OOI.
As shown in Fig. 4.7 (a) (b), regarding the detection of selected REST patterns/anti-patterns (i.e., Correct use of POST and Tidy URLs patterns, Amorphous URIs and Forgetting Hypermedia anti-patterns), we observed that the mean usefulness rate is greater than 4, which represents a good and acceptable rate. Similarly, the most of participants have positively rated the detection usefulness of the selected OCCI patterns and anti-patterns (i.e., Compliant Delete and Compliant URL patterns, Non-Compliant Create and Non-Compliant Trigger Action anti-patterns) as the observed score is greater than 3.9 (refer to Fig. 4.7 ((c) (d)). Overall, participants reported that the provided detection and recommendation support are considerably useful. Given this observation, we confirm that the key detection rules and the provided recommendations are useful and relevant. Moreover, as feedback for improvement, most participants highlight the need of sophisticated way to visualize both the detected patterns and anti-patterns. Regarding the detected patterns, it has been suggested to provide a short text explaining it as well as its importance. While regarding anti-patterns, it is better to provide instead of a just an explanation a link that present this explanation in more details and to represent recommendations using markers (i.e., colors, formatting, etc.).

4.5 Conclusion

In this chapter, we achieved the two objectives mentioned in the thesis problematic (section 1.2.1) which are:(1) assisting the design of interoperable management APIs and (2) exploring whether the current management APIs follow the REST and OCCI best principles. In doing so, we answered the different questions raised in this same section (section 1.2.1), which were all around the way of exploiting (anti) patterns towards ensuing these two objectives.
Concretely, we adopted patterns and anti-patterns as means to represent respectively the good and poor practices of both OCCI and REST best principles that API developers or cloud providers should be carefully taken on when designing their APIs. Moreover, we analyzed both the literature and the OCCI standard to identify these (anti)patterns. Furthermore, semantic models, in particular ontologies, have been adopted as an appropriate means relying on SWRL, SQWRL and SPARQL languages to specify and to detect both patterns and anti-patterns and to provide correction recommendations in case of any anti-pattern detection. We proposed a semantic definition of 21 common REST (anti) patterns and 24 OCCI (anti) patterns for Cloud RESTful APIs and four detection algorithms acting on these specifications to detect OCCI (anti)patterns and REST (anti)patterns respectively.
We validated our approach by analyzing both OCCI and REST (anti) patterns on real world Cloud RESTful APIs, and assessing its feasibility in term of accuracy and usefulness. The observed accuracy shows that our approach is an effective technique for detecting OCCI and REST (anti) patterns in Cloud RESTful APIs, which may

assist cloud API developers and providers when assessing the quality of their APIs and correcting them to avoid the anti-patterns occurrences. In addition, through the usefulness evaluation, we proved that the key detection rules and provided recommendations are useful and relevant.
Besides, through the compliance analysis, regarding REST best principles, we observed that the most of the analyzed Cloud RESTful APIs have reached an acceptable level of maturity by considering the most of good REST principles. In opposition, we observed also through the obtained OCCI compliance degrees, that there is no correct and adequate adoption of the OCCI best principles in the selected Cloud RESTful APIs. Hopefully, this inspires the developers of these APIs and other practitioners to include REST and OCCI good principles as much as possible. Thus, we will contribute to the improvement of OCCI specifications as supporting of REST best principles in OCCI-based APIs would make them more visible and easy to understand.
Finally, by assisting in creating interoperable management APIs that is compliant with an open standard like OCCI, our approach presents an important contribution toward facilitating cloud interoperability from a management perspective. However, cloud interoperability to be a concrete truth, it demands to be addressed from other perspectives. Within this context, the following chapter falls. In particular, it aims at supporting the seamless integration between TOSCA standard and current DevOps approaches toward interoperable orchestration of cloud resources.

Table 4.12: Complete validation of REST patterns and anti-patterns on OOi and Rackspace

	OOi							**Rackspace**						
(anti)Patterns	Validated	P	TP	Precision	Avg.Precision	Recall	Avg.Recall	Validated	P	TP	Precision	Avg.Precision	Recall	Avg.Recall
Tidy URIs	6	5	5	100%		83%		20	19	19	100%		95%	
Amorphous URIs	8	8	8	100%	100%	100%	91.7%	20	20	20	100%	100%	100%	97.5%
No Detection	0	0	0	-		-		0	0	0	-		-	
Verbless URIs	10	10	10	100%		100%		20	20	20	100%		100%	
CRUDy URIs	0	0	0	-	100%	-	100%	0	0	0	-	100%	-	100%
No Detection	0	0	0	-		-		0	0	0	-			
Singularized nodes	6	5	5	100%		83%		20	18	18	100%		90%	
Pluralized nodes	8	8	8	100%	100%	100%	91.7%	4	2	2	100%	100%	50%	70%
No Detection	0	0	0	-		-		0	0	0	-		-	
Correct use of POST	8	6	6	100%		75%		10	10	10	100%		100%	
Correct use of GET	0	0	0	-		-		10	10	10	100%		100%	
Correct use of PUT	0	0	-	-		-		4	4	4	100%		100%	
Correct use of DELETE	6	6	6	100%	100%	100%	87.5%	9	9	8	100%	100%	0.88%	97.2%
Correct use of HEAD	0	0	0	-		-		0	0	0	-		-	
Tunneling E.T.T POST	0	0	0	-		-		0	0	0	-		-	
Tunneling E.T.T GET	0	0	0	-		-		0	0	0	-		-	
No Detection	0	0	0	-		-		0	0	0	-			
Supporting Status Code	10	10	10	100%		100%		20	20	20	100%		100%	
Ignoring Status Code	0	0	0	-	100%	-	100%	10	10	10	100%	100%	100%	100%
No Detection	0	0	0	-		-		0	0	0	-			
Supporting Caching Code	0	0	0	-		-		0	0	0	-		-	
Ignoring Caching	10	10	10	100%	100%	100%	100%	10	10	100%		100%		100%
No Detection	0	0	0	-		-		0	0	0	-			
Supporting MIME Types	8	8	8	100%		100%		10	9	9	100%		90%	
Ignoring MIME Types	10	10	10	100%	100%	100%	100%	0	0	0	-	100%	-	90%
No Detection	0	0	0	-		-		0	0	0	-			
Supporting hypermedia	5	4	4	100%		80%		10	7	7	100%		70%	
Forgetting hypermedia	0	0	0	-	100%	-	90%	10	8	8	100%	100%	80%	75%
No Detection	4	4	4	100%		100%		0	0	0	-		-	
Average	**Precision**				100%	**Recall**	95,1%	**Precision**				100%	**Recall**	91.2%
	F-measure					97.4%		**F-measure**					95.3%	

Table 4.13: Complete validation results of OCCI patterns and anti-patterns on OOi and Rackspace REST APIs

	OOi							**Rackspace**						
(anti)Patterns	Validated	P	TP	Precision	Avg.Precision	Recall	Avg.Recall	Validated	P	TP	Precision	Avg.Precision	Recall	Avg.Recall
Query interface support	0	0	0	-		-		0	0	0	-		-	
Missing query interface	1	1	1	100%	100%	100%	100%	1	1	1	100%	100%	100%	100%
No Detection	0	0	0	-		-		0	0	0	-		-	
Compliant Create	4	3	3	100%		75%		0	0	0	-		-	
Non-Compliant Create	0	0	0	-	100%	-	75%	9	9	9	100%	100%	100%	100%
No Detection	0	0	0	-		-		10	10	10	100%		100%	
Compliant Update	3	3	3	100%		100%		7	7	7	100%		100%	
Non-Compliant Update	0	0	0	-	100%	-	100%	0	0	0	-	100%	-	100%
No Detection	0	0	0	-		-		6	6	6	100%		100%	
Compliant Delete	3	3	3	100%		100%		10	10	10	100%		100%	
Non-Compliant Delete	0	0	0	-	100%	-	100%	0	0	0	-	100%	-	100%
No Detection	0	0	0	-		-		0	0	0	-			
Compliant Retrieve	10	10	10	100%		100%		0	0	0	-		-	
Non-Compliant Retrieve	10	0	0	-	100%	-	100%	10	9	9	100%	100%	90%	90%
No Detection	0	0	0	-		-		0	0	0	-			
Compliant Trigger Action	0	0	0	-		-		0	0	0	-		-	
Non-Compliant Trigger Action	0	0	0	-	100%	-	100%	10	8	8	100%	100%	80%	80%
No Detection	8	8	8	100%		100%		0	0	0	-			
Compliant Link between Resources	2	2	2	100%	100%	100%	100%	0	0	0	-%	100%	-	100%
Non-Compliant Link between Resources	0	0	0	-		-		1	1	1	100%		100%	
No Detection	0	0	0	-		-		5	5	5	100%		100%	
Compliant Association of resource with Mixin	0	0	0	-		-%		-	-	-	-		-	
Non-Compliant Association of resource with Mixin	0	0	0	-	100%	-	100%	-	-	-	-	-	-	-
No Detection	5	5	5	100%		100%		-	-	-	-			
Compliant Dissociation of resource from Mixin	0	0	0	-		-		-	-	-	-		-	
Non-Compliant Dissociation of resource from Mixin	0	0	0	-	100%	-	100%	-	-	-	-	-	-	-
No Detection	6	6	6	100%		100%		-	-	-	-			
Compliant URL	10	10	10	100%		100%		10	10	10	100%		100%	
Non-Compliant URL	0	0	0	-	100%	-	100%	0	0	0	-	100%	-	100%
No Detection	0	0	0	-		-		0	0	0	-			
Compliant Request Header	0	0	0	-		-		10	10	10	100%		100%	
Non-Compliant Compliant Request Header	10	10	10	100%	100%	100%	100%	0	0	0	-	100%	-	100%
No Detection	0	0	0	-		-		0	0	0	-			
Compliant Response Header	0	0	0	-		-		10	10	10	100%		100%	
Non-Compliant Compliant Response Header	10	10	10	100%	100%	100%	100%	0	0	0	-	100%	-	100%
No Detection	0	0	0	-		-		0	0	0	-			
Average	**Precision**				100%	**Recall**	97,9%	**Precision**				100%	**Recall**	97%
	F-measure					98.9%		**F-measure**					98%	

Chapter 5

Streamlining interoperable orchestration of TOSCA with DevOps technologies using model-driven integration

Contents

5.1 Introduction

This chapter introduces our approach to streamline and improve the orchestration process of cloud resources. The intrinsic goal is providing an automated integration between TOSCA, the de facto cloud orchestration standard, and the open-source leading DevOps technologies. In doing so, cloud users would benefit from the good usability of TOSCA and its agnostic nature for convenient modeling of their applications on one hand, and the efficiency of DevOps technologies in enabling the end-to-end orchestration tasks on the other hand.
Toward a seamless integration, as we desired, we propose to automate the mapping between TOSCA and DevOps technologies. This will close the gap existing between them which is mainly due to the use of diverse models, entities, and languages when describing cloud resources as well as orchestration aspects. In addition, since DevOps technologies employ several different types of heterogeneous tools/APIs to orchestrate cloud resources, we propose a DevOps abstraction layer that hides their related complex low-level details and avoids dealing with their diverse technical specifications. This alleviates users from the heavy lifting involved when interacting with these DevOps tools/APIs.
A key part of our approach involves leveraging MDE principles namely: the use of models as first-class entities and high-level model transformations. Indeed, in addition to TOSCA that is naturally model-driven, we use models to represent DevOps underlying specification and languages, where their specific low-level details are abstracted and organized into high level, meaningful and machine-readable constructs. This will support their manipulations at a high level of abstraction. Besides, we rely on the MDE transformation principle as an underlying technique for the automated mapping between TOSCA and DevOps solutions.
More specifically, the contribution of this chapter aim at ensuring the following objectives:

1. Adopting the TOSCA standard for describing cloud resource-related artifacts in a technology-independent way. These artifacts form among themselves the cloud application to be orchestrated.

2. Proposing a Model-Driven Transformation Technique allowing an automated translation of cloud resource artifacts (designed using TOSCA) into DevOps-specific artifacts ready to be executed by the DevOps tools and APIs.

3. Providing Connectors acting as a DevOps abstraction layer that serves to establish the bridge between DevOps-specific artifacts (e.g., Docker compose model) and the underlying DevOps tools/APIs (e.g., Docker compose tool and remote management API).

4. Demonstrating the applicability of our integration approach using two case studies based on Docker and Terraform solutions.

5. Proposing a proof-of-concept called *ToDev* , an integrated and standards-driven orchestration framework based on TOSCA and DevOps technologies.

6. Validating our approach by conducting three experiments using *ToDev* framework in order to particularly demonstrate the gained productivity, introduced overhead and the transformation performance.

The rest of this chapter is organized as follows: Section 5.2 articulates our motivations and work fundamentals by providing the classification of the DevOps artifacts. In Section 5.3, we detail our approach. Section 5.4 and 5.5 demonstrate our approach applicability using the selected DevOps solutions. In section 5.6, we describe the *ToDev* framework and its underlying architecture. Section 5.7 presents the approach validation using the implemented framework. Finally, Section 5.8 concludes the chapter.
The work in this chapter was published in a conference proceeding [32].

5.2 Motivations and fundamentals

In this section, we first investigate through a motivating example, specific challenges among existing DevOps solutions to orchestrate cloud resources that are involved in the typical cloud application's lifecycle. Afterward, we give an overview of the available artifacts provided by the DevOps solutions.

5.2.1 Motivating scenario

Our motivating scenario relies on a composite cloud application that is to be deployed on two cloud platforms. This application is composed of two parts. A frontend part developed by Python and Node.js services that respectively allow implementing a graphical voting interface of multiple options and a visualization of the voting results. A backend part composed by the Redis service that collects the votes that are transmitted, via a NET worker service, to a Postgres database. The application will be hosted on two cloud provider platforms namely AWS and Google Compute Platform (GCP). The topology of the intended application is shown in Figure 5.1.
Typically, to support the orchestration life cycle of this application, DevOps users first need to describe the involved services, requirements in terms of cloud resources and

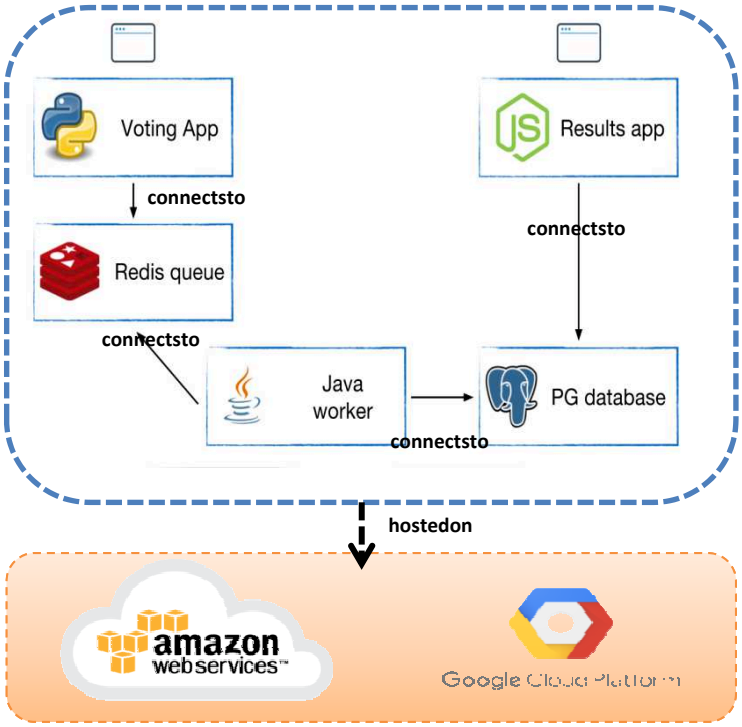


Figure 5.1: Voting application topology

the dependency relationships that may exist between these services, e.g. connected-to and hosted-on. To realize this task, DevOps users may use one of the available DevOps solutions such as Docker [49]. Thus, the DevOps user has to specify some of the Docker-specific artifacts, essentially the so-called *Compose file* and *Dockerfiles*. Then, by using Docker APIs/tools, the DevOps users rely on these artifacts in order to orchestrate necessary resources known as Containers on a given cluster (a set of computing nodes such as VMs or servers). Currently, in Docker, the configuration of this cluster is still performed manually using low-level commands [49]. So, another DevOps tool would be needed such as Terraform [149] or Ansible [15] to provide an easy way to orchestrate the entire cluster. This results in adding a new kind of artifacts to be managed by DevOps users. Therefore, DevOps users are forced to understand and analyze different DevOps tools (Docker, Kubernetes, Ansible, Terraform, etc.) to specify the related artifacts. Moreover, most of these solutions inherently rely on heterogeneous, low-level and ad-hoc script-based languages. Certainly, these issues are getting worse with the desire of users to exploit resources from multiple clouds.
The contribution of this chapter is mainly devoted to resolving such challenges by providing integration support between the TOSCA standard and the underlying DevOps solutions. Combining TOSCA with DevOps solutions would significantly streamline

DevOps Tools	Node-centric Artifacts	Environment-centric Artifacts
Docker	Dockerfile	Compose file
Kubernetes	Dockerfile	Service file, Deployment file
Terraform	-	*.tf files (e.g. instances.tf)
Juju	shell scripts	Charms, Bundles
Chef	Cookbooks	-
OpenStack Heat	-Puppet modules, Cookbooks, Shell scripts	Heat Template
CloudFormation	-Puppet modules, Cookbooks, Shell scripts	CF Template

Table 5.1: Examples of DevOps solutions and their related artifacts

the orchestration process of cloud resources while maintaining the desired efficiency and interoperability. In the following, we provide some required details on the DevOps artifacts.

5.2.2 DevOps artifacts

Recently there is a continuous proliferation of DevOps solutions. Each solution relies on its individual and proprietary artifacts to effectively orchestrate cloud resources. Basically, these artifacts vary in how they are designed and how they are used. Wettinger et al. [156] provided a classification of DevOps artifacts depicting their conceptual and technical variability. Accordingly, two major categories can be distinguished:

- Node-centric artifacts (NCAs). include scripts, images, and low-level configuration definitions, that are designed to be executed on a single node such as a container or a virtual machine. Technically, NCAs can not be used to deploy the entire application topology since the dependencies between nodes are not explicitly specified.

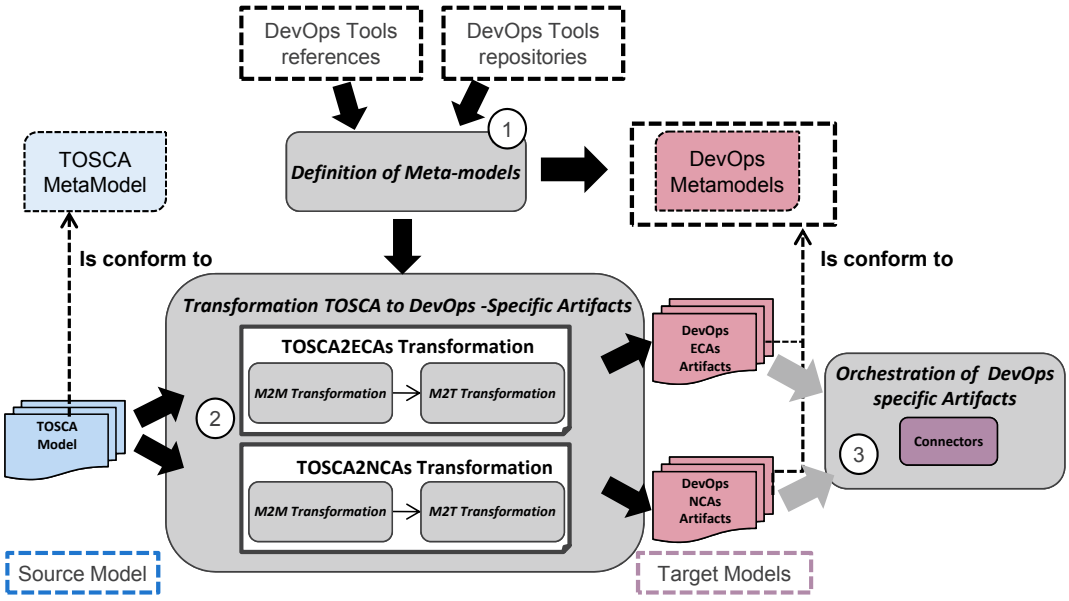


Figure 5.2: Model-driven integration approach overview

- Environment-centric artifacts (ECAs). are designed to be executed in an environment and can be used to deploy the entire application topology. Unlike NCAs, they include scripts, bundles, and, templates, in which multiple nodes can exist and their dependencies can be explicitly expressed.

Most of the DevOps solutions rely on both artifacts kinds as ECAs natively use and orchestrate NCAs. For instance, Docker uses Compose file and Dockerfiles. Compose file represents an ECA specified in a YAML-like format, which describes a collection of services and their dependencies. Whereas, Dockerfile is an NCA used to configure required containers related to each service. Technically, a Compose file needs to employ Dockerfile to configure and orchestrate the involved resource containers. Table 5.1 depicts for each DevOps solution the NCAs and ECAs that it uses. Similar to Docker, most of these solutions employ both artifacts such as Juju Charms [152] employs Unix shell scripts, and so on. Both artifact kinds and the way how they are designed, represent the foundation of our transformation logic.

5.3 Model-driven integration approach

In this section, we detail our model-driven integration toward streamlining and improving the orchestration of cloud applications. Figure. 5.2 provides a high-level overview of our approach, depicting its main three steps, namely: *Definition of Meta-models*, *Transformation of TOSCA into DevOps-specific Artifacts* and *Orchestration of DevOps-specific Artifacts*. The first step defines both TOSCA and DevOps related knowledge for the transformation purpose. The second step defines the underlying logic and techniques for transforming any TOSCA topology model into DevOps-specific artifacts. The third step consumes the generated DevOps artifacts to manage orchestration tasks. At present, we applied our approach to a diverse range of solutions including Docker, Kubernetes, Terraform. It should be noted that our approach is generic enough to be applied to other tools. In fact, six solutions among the most important ones (e.g., Docker, Kubernetes, Terraform, Juju, OpenStack Heat, and CloudFormation), have been considered when defining this approach. In the following, our approach steps are discussed in detail.

5.3.1 Definition of Meta-models

Here, knowledge including TOSCA metamodel and DevOps metamodels have to be defined. As we mentioned above in chapter 2, TOSCA offers a structured language and a metamodel for describing service templates. Since DevOps users have to model TOSCA-compliant applications typologies, TOSCA meta-model represents the first foundation stone for our integration approach. The basics concepts of the TOSCA meta-model relevant for our approach are illustrated in Figure 5.3. We extracted these concepts by analyzing the YAML-based TOSCA specification in [116] and all previous modeling attempts [30]. The definition of each concept is previously given in chapter 2 (refer to Section 2.3.2 about the TOSCA standard). In the remainder of the chapter, we use TOSCAMM to denote the TOSCA meta-model.
In addition to TOSCAMM, DevOps meta-models have to be defined. Indeed, despite the importance of DevOps solutions, most of them lack of formal meta-models to describe cloud resource artifacts (ECA and NCA). As shown in Figure 5.2 (refer to step 1) , during this step, we perform an extensive domain analysis on DevOps related references and repositories with the aim of identifying the corresponding meta-models as well as their underlying representations. More specifically, we identify key entities as well as possible relationships between them, used for describing artifacts according to each DevOps solution.

5.3.2 Transformation of TOSCA artifacts into DevOps artifacts

As shown in Figure.5.2 (refer to step 2) , the entry point of the transformation step, is a TOSCA topology model capturing the required cloud resources as well as their related artifacts. This step relies on a novel model-driven translation technique that

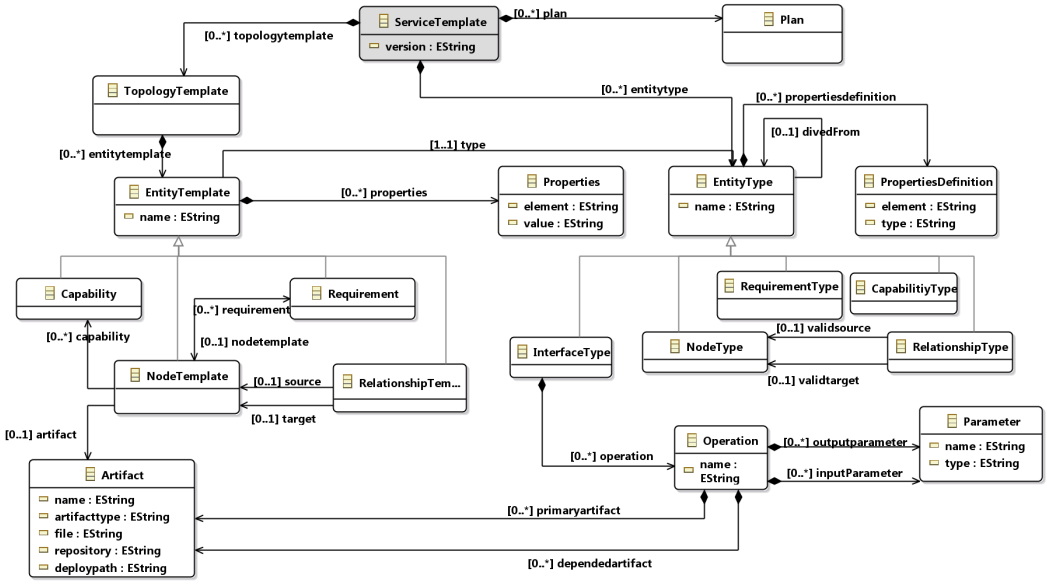


Figure 5.3: TOSCA Metamodel (TOSCAMM)

serves to translate any TOSCA topology into DevOps tool-specific artifacts compliant to DevOps models and their underlying representations. Instead of directly manipulating code which is a time-consuming and error-prone task, we adopted MDE transformation languages as underlying techniques for the proposed transformation technique. By considering the DevOps artifact categories explained in Section. 5.2.2, the proposed translation technique includes two transformation sub-steps:

- *Transformation of TOSCA Topology to ECA (TTopology2ECA)):* This step aims to generate an ECA for the whole TOSCA topology while considering all its nodes and possible dependencies between them. This involves two phases: M2M Transformation and M2T Transformation. Firstly, an M2M transformation is conducted, transforming TOSCA topology into DevOps-specific target ECA model. Then, an M2T transformation is used to produce the textual definition files (template, scripts, etc.) from the generated ECA model. Ideally, this fit with MDE transformation best principles, that is, an intermediate structural model for representing target ECA is firstly generated rather than generating directly plain text for these artifacts. Inevitably, we need such intermediate model to capture, in the generated ECA model, all complex dependencies that may exist between nodes in the TOSCA topology model.

- *Transformation of TOSCA Node to NCA (TNode2NCA):* as stated earlier, ECA basically uses and orchestrate NCAs. So this transformation can be viewed as complementary to the above transformation step. More precisely, this step aims at gen-

erating for each node specified in the TOSCA topology, the corresponding NCA if needed. Similar to the TTopology2ECA transformation, the TNode2NCA transformation proceeds in two main transformation steps, namely: M2M transformation and M2T transformation. M2M transformation is firstly performed, transforming each TOSCA node into a DevOps-specific NCA model that defines this artifact. Then, an M2T transformation is followed in order to produce the textual definition files from the obtained NCA model.

5.3.3 Orchestration of DevOps-specific artifacts

Orchestration is the responsibility of the Orchestrator service (also called orchestration engine) that is exposed to DevOps users. This Orchestrator receives a TOSCA template in order to generate the required DevOps-specific artifacts and to execute the end-to-end orchestration operations related to the described application. The latter task requires communicating with the underlying DevOps solutions and requesting the execution of certain management actions. However, DevOps solutions inherently employ multiple and heterogeneous tools/APIs to execute these actions, which complicates the orchestrator task. To avoid that, we propose a DevOps abstraction layer that is empowered with a set of Connectors for alleviating the orchestrator from the heavy lifting involved when interacting with different DevOps tools/APIs.
Connectors seek to establish the bridge between the high-level description artifacts generated and the underlying cloud orchestration DevOps solutions. Fundamentally, Connectors are application programming interfaces (APIs) devoted to ensure communication with target DevOps tools/APIs. Each connector defines a set of high-level management operations initiating the corresponding API calls and low-level actions specific to the target DevOps solution. To ensure the well-foundation of our Connectors, we defined a set of requirements by considering an important set of software development qualities:

- Connectors have to be enough extensible becuase DevOps curators often require to add new functionalities (i.e. operations).
- The development of Connectors has to promote the reusability since the from-scratch development of code is a tedious task. In addition, this would speed up the development time.
- The development of Connectors has to promote the ease of maintenance of code since Connectors are continuously subject to frequent updates to be able to accommodate the changes in target DevOps tools.

To meet these requirements, we introduce the Essential Connecter Metamodel (ECMM) describing the common and basic concepts that may be involved when developing the native code of every desired connector. By doing so, we are able to

Interface Identity	**Definition**
Init ()	**Syntax:** The signature of this operation is defined as follows: *Init (String DSA DirectoryPath): String.* **Semantics:** This operation enables the analysis of generated artifacts to interrupt whatever necessary in their next deployment. Based on the selected DevOps solution and cloud providers, the initialization may result in the generation of other necessary configuration files and download of the dependencies. **Data Types and Constants:** The operation requires the *directory-path*, where the artifact files are located. The operation returns a *message* indicating the operation state (success/ failure) along with details summarizing the taken treatments.
Deploy ()	**Syntax:** The signature of this operation is defined as follows: *Deploy (String FilePath): Object.* **Semantics:** This operation enables the deployment of one or more cloud resources defined in the inputted artifact. Indeed, it allows executing the low-level deployment actions related to the target DevOps solution. **Data Types and Constants:** The operation requires the *file path* of the concerned artifact and returns an *object* consisting of a *deployment-id* and a *message* that indicate whether the deployment has been successfully completed, completed with errors or has been failed.
Configure()	**Syntax:** The signature of this operation is defined as follows: *Control (String Action actiondef): Message.* **Semantics:** This operation enables the execution of requested action on the target resource defined by the *deployment-id*, which is stated in the action definition *actiondef*. These actions specify how a cloud resource should behave when certain events occur, which include horizontal scaling, vertical scaling, migration, application reconfiguration, and basic actions. The precise definition of each action is given in the next chapter. **Data Types and Constants:** The operation requires the *Action object* that indicates the definition of the action to be executed along with the concerned cloud resource which is often indicated by a precise logic name or a *deployment-id*. As a result, it returns a *message* indicating the operation state (success/ failure) along with details describing the new state of configured cloud resource.
Undeploy()	**Syntax:** The signature of this operation is defined as follows: *Undeploy (String Deployment-id): Message.* **Semantics:** This operation allows canceling the already deployed cloud resources. It can be applied to a single cloud resource or the entire deployed model. **Data Types and Constants:** The operation requires the *deployment-id* of the deployed cloud resource or model and returns a *message* indicating the success/failure of the operation.
List()	**Syntax:** The signature of this operation is defined as follows: *List (String Deployment-id):ResourcesList .* **Semantics:** This operation allows listing all already deployed cloud resources, services and some details describing their states. **Data Types and Constants:** The operation requires the *deployment-id* of the deployed model and returns a list of resources (i.e *ResourcesList*) along with their state details.

Table 5.2: Connector basic Interface Operations

apply MDE transformation to automatically generate most code parts. The generated code may require to be completed by the DevOps developers. Moreover, the core ECMM concepts are extracted by analyzing the orchestration capabilities and operations of the selected DevOps Tools/APIs. The resulting ECMM is illustrated in Figure 5.4. Accordingly, the core connector interface has to support five management operations: *Init, Deploy, Control, Undeploy* and *List* as they are supported by every DevOp tool/API. The definition of each of these operations is detailed in table 5.2 while following the Software Engineering Institute template [27] used for the interface documentation.

In our ECMM (see Figure 5.4), each operation is conceptually composed of one or more *Input* and one *Output* parameters and one or more *Scripts*. The concrete implementation of these scripts is organized into patterns to facilitate later their injection in the whole connector code when proceeding with its generation. Indeed, these scripts are closely related to the target DevOps solution and are parameterized with the input parameters that have to be passed during the execution time. Besides, the

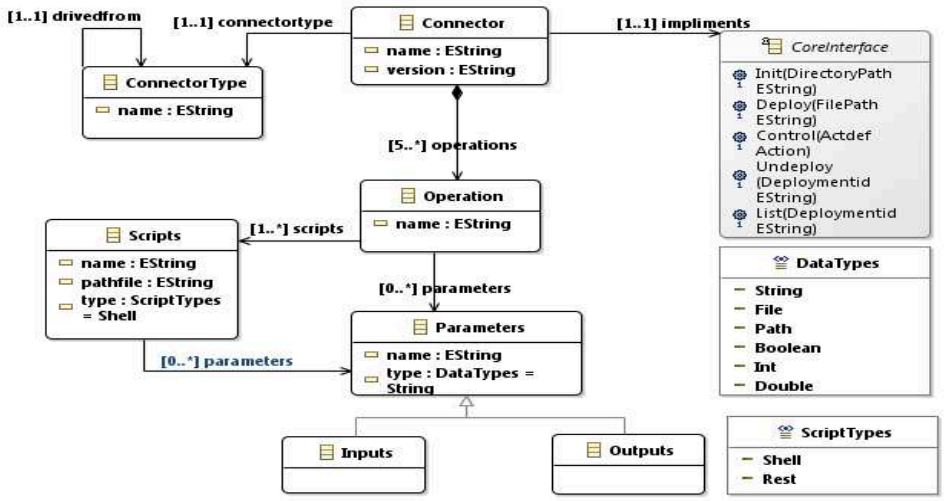


Figure 5.4: The Essential Connector Meta-model (ECMM)

execution of scripts can involve the invocation of specific third-party APIs. For this reason, the type of script (e.g. Shell, REST, etc.) has to be specified. We defined the *Shell utility* pattern that encapsulate the common code to execute a shell scripts using command-line interface. Finally, *ConneteorType* is introduced to promote the reusability of each created connector. The default type is Root Connector denoted by *"ECMM.Connector.Root"* that all other connector types derived from, which represent the basic definition of each connector. Up to now, we defined two normative types: *ECMM.Connector.Docker* and *ECMM. Connector. Terraform* as a result of the concrete implementation of Docker and Terraform Connectors. Figure 5.5 depicts a partial view of the Terraform Connector model, from which a corresponding java code is automatically generated.

5.4 Docker case-study

5.4.1 Building DevOps metamodels

Since both Docker-Compose file (as ECA) and Dockerfile (as NCA) are the main Docker-specific artifacts, a metamodel to represent Compose file (referred as ComposeMM) and a metamodel to represent Dockerfile (referred as DockerfileMM) have to be specified.
Regarding the Docker-Compose, Figure 5.6 presents the resulting ComposeMM after analyzing Docker-specific sources. Compose is the root concept defining the full application stack. Service represents a software piece, by which the application is composed. Each Service can depend on another service and may be connected to

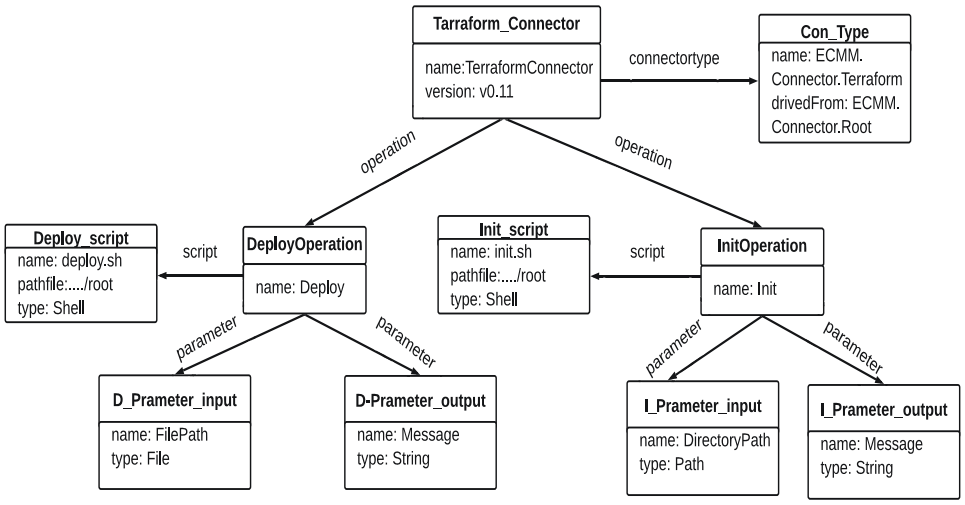


Figure 5.5: A partial view of the Terraform Connector model, which represents an instance of ECMM

one or more Networks and be attached with one or more storage Volumes. It can be deployed in one or more Containers. Image contains required instructions to install and to run each Container and stored in a private or public Registry. Resources related to each container specify the needed requirements and constraints that should be respected at the runtime. Finally, Properties include name-value pairs for describing service, volume, network or container-specific properties (e.g., port). Furthermore, by analyzing the Dockerfile reference and recommended best practices, we build the corresponding eDockerfile as in Figure 5.7. Dockerfile is the root concept which specifies all required instructions, whose executions result in creating the corresponding Docker image. Each Dockerfile consists of one or more Stages defined through one or more Instructions. Each Instruction has an execution order and is defined over a set of Commands, Arguments and Environment Variables (EnvVars). We distinguish 8 instruction types. For instance, FromInstruction defines the base image from which a new Docker image builds and CMDinstruction provides the default command to execute a container resource.

5.4.2 Transformation of TOSCA to Docker-specific artifacts

Illustration example. We present in Figure 5.8 a partial TOSCA topology model for our motivation scenario. It consists of 2 TOSCA nodes, where vote has "Web Application" type and redis has "Container.Application.Docker" type. This topology model will act as a source model for the transformation of TOSCA to Docker-specific artifacts.

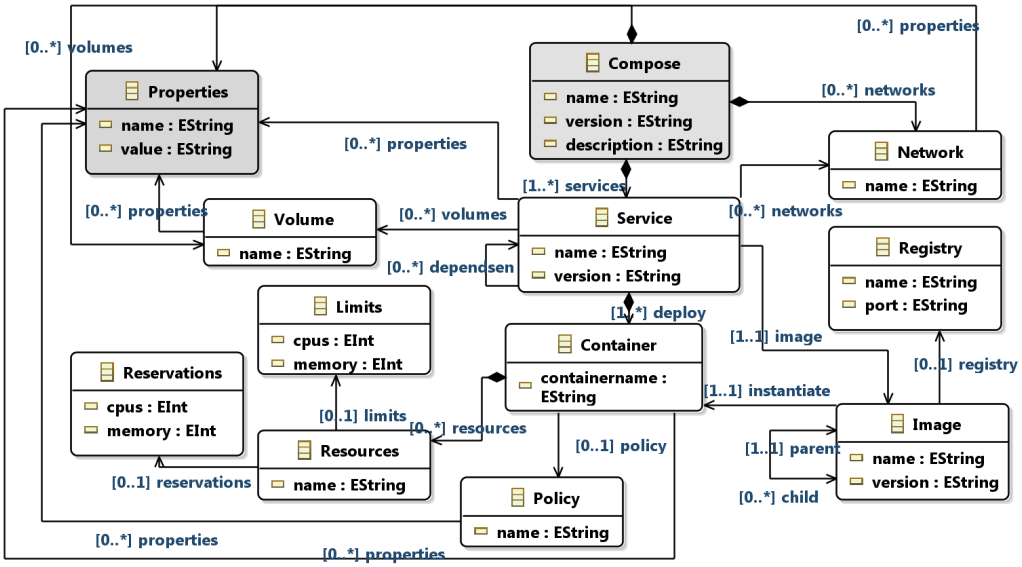


Figure 5.6: Docker Compose Metamodel (ComposeMM)

5.4.2.1 Transformation of TOSCATopology2Compose.

This step involves two phases: M2M Transformation and M2T Transformation. Firstly, an M2M transformation is conducted, transforming any TOSCA topology model (an instance of TOSCAMM) into the Compose model (an instance of ComposeMM). Then, an M2T transformation is followed in order to produce the YAML definition file from this Compose model. To enable M2M transformation, it is essential to clarify the conceptual mapping between TOSCAMM and ComposeMM. We summarize in Table 5.3 both possible and no possible mappings between these two metamodels. As depicted in the top part of the table, some of TOSCAMM concepts do not have any mapping in ComposeMM. This is expected due to the generic nature of TOSCAMM. These concepts are the concepts devoted for the reusability, namely EntityType including all its inherent types (NodeType, RelationshipType, RequirementType, Capabilitytype, InterfaceType) and for the orchestration, including Interface, Operation and Parameter.
Despite that, the mapping of most of the concepts between TOSCAMM and ComposeMM is still possible. This mapping acts as a method to translate any TOSCA topology model to the Compose model. In fact, as indicated in the rest of the Table 5.3 , NodeTemplate can be mapped to one of the key ComposeMM entities namely Service, Network or Volume. This is the case of One-to-Many mapping which demands special treatment for inferring the exact ComposeMM entity to be mapped with NodeTemplate. Basically, the EntityType associated with the NodeTemplate by

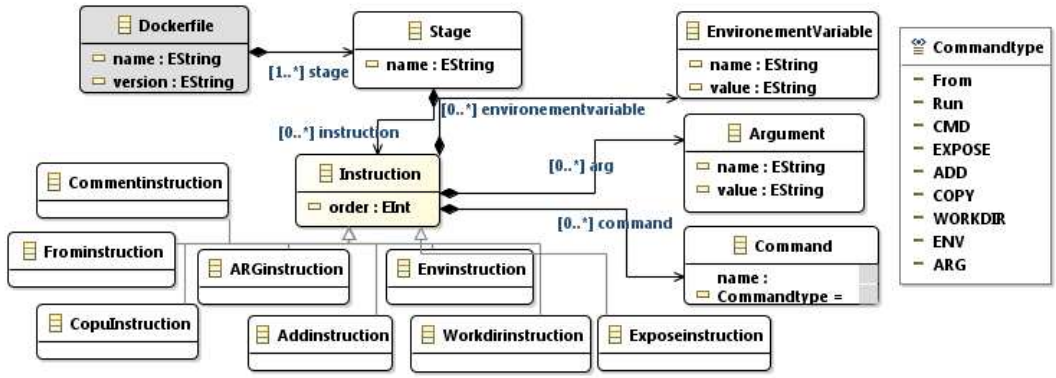


Figure 5.7: Dockerfile Metamodel (DockerfileMM)

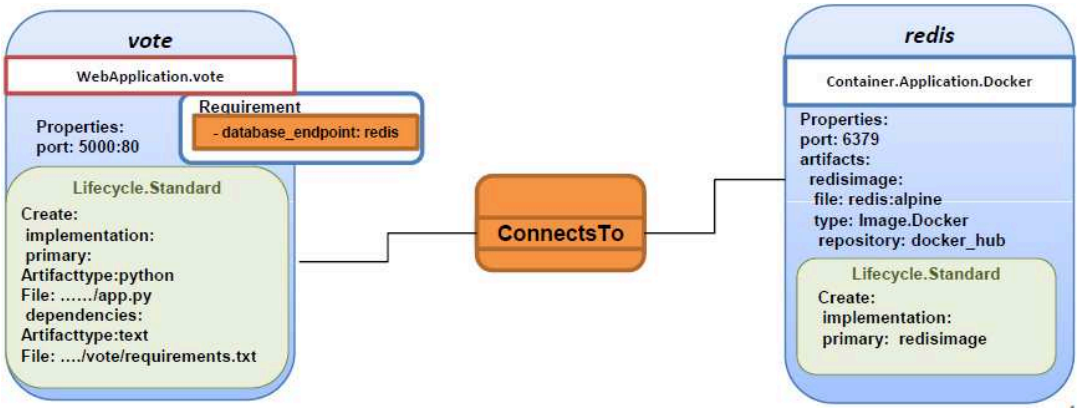


Figure 5.8: Partial part of the TOSCA Topology for the motivating example in Figure 5.1

means of type reference, can straightforwardly resolve this mapping. Accordingly, a NodeTemplate in TOSCAMM can be mapped to Service in ComposeMM iff it is typed with one of EntityTypes devoted for software solutions (e.g., Application, Web application, Database, to name a few).
Furthermore, the Properties associated with each NodeTemplate, can be easily mapped to Properties associated with each Compose entities based on a syntax matching. Moreover, TOSCA RelationshipTemplate and Requirement can be mapped to depend-son, a reference-related to the **Service**, which captures the possible dependencies between services. For instance, in case of RelationshipTemplate, the first member of depend-on reference corresponds to the source of RelationshipTemplate and second to its target. Additionally, NodeTemplate Name or its associated Artifact if exist can be matched with the Image associated with each **Service** in ComposeMM.
Based on this mapping, we define M2M transformation outlined in Algorithm 1 to

TOSCA concept	**Compose concept**
EntityType	–
Interface, Operation	–
NodeTemplate	Service/ Volume/ Network
Properties	Properties
Requirement	dependson/PlacementConstraint
NodeTemplate Name/Artifact	Image
Capability	Resources

Table 5.3: Mapping of TOSCA concepts to ComposeMM concepts

translate any TOSCA topology model to the Compose model. For instance, lines 4-8 allows to transform any NodeTemplate, typed with one of the common software types such as "Application", "Database" to Service. Figure 5.9 (a) depicts the produced Compose model for the used source TOSCA topology, as a result of applying the M2M transformation.

Algorithm 1 M2M transformation Algorithm for transforming TOSCA typology into Docker Compose model

1: **input**: TOSCAMM, ComposeMM, TOSCA application typology TO_{at}, Software types Sof_{type}, platform types Pla_{type}, Volume types Vol_{type}, Network types Net_{type}
2: **output**: Docker Compose Model CO_{mo}
3: **for** $node \in TO_{at}$ **do**
4: **if** $node.type \in Sof_{types} \cap Pla_{types}$ **then**
5: $CO_{mo}.service$=**map** $Node2Service()$
6: $CoO_{mo}.service.properties$=**map** $NodeProperties2ServiceProperties()$
7: **if** $node.artifact.name =$ "my_image" **then**
8: $CO_{mo}.service.image$=map $NodeArtifact2ServiceImage()$
9: **end if**
10: $CO_{mo}.service.dependson$=**resolve** $NodeRequirement2Servicedependson()$
11: **else if** $node.type \in Vol_{types}$ **then**
12: $CO_{mo}.volume$=**map** $Node2Volume()$
13: $CoO_{mo}.volume.properties$=**map** $NodeProperties2VolumeProperties()$
14: **else**
15: $CO_{mo}.network$=**map** $Node2Network()$
16: $CO_{mo}.Network.properties$=**map** $NodeProperties2NetworkProperties()$
17: **end if**
18: **end for**

Once the M2M transformation step is achieved, the M2T transformation can take place. Algorithm 2 illustrates the main steps of the M2T transformation for generating the underlying YAML representation of the generated Compose model. Figure 5.9

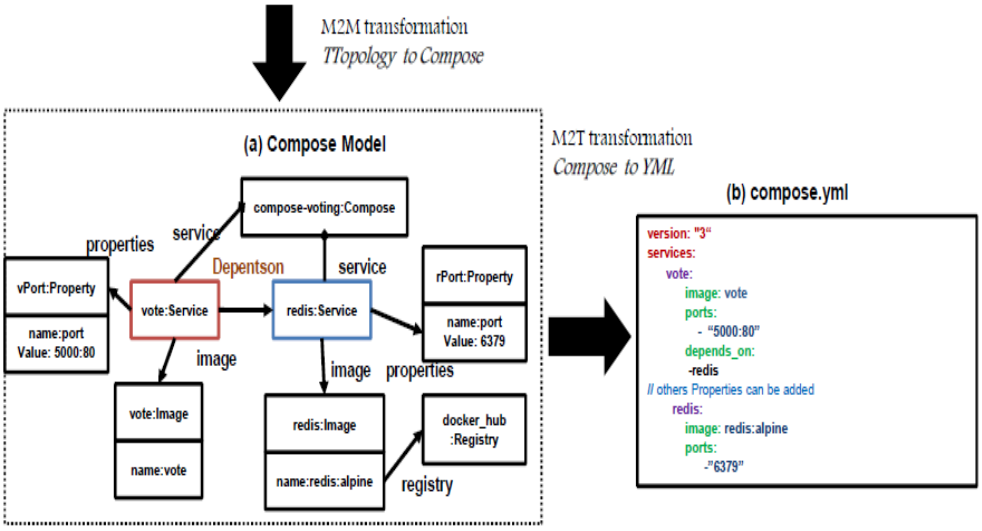


Figure 5.9: Transformation of TOSCA topology depicted in Figure 5.8 to Compose-specific artifacts: (a) Compose model; (b)compose.yml

(b) depicts a partial fragment of the YAML representation for the Compose model, after applying the M2T transformation.

Algorithm 2 M2T transformation Algorithm for generating YAML scripts from the Compose Model

1: **input**: Docker Compose Model CO_{mo}
2: **output**: Compose yaml scripts *compose.yml*
3: **for** $service \in CO_{mo}$ **do**
4: Generate service-specific Scripts
5: **end for**
6: **for** $volume \in CO_{mo}$ **do**
7: Generate volume-specific Scripts
8: **end for**
9: **for** $network \in CO_{mo}$ **do**
10: Generate network-specific Scripts
11: **end for**

5.4.2.2 Transformation of TOSCANode2Dockerfile

In this step, each TOSCA NodeTemplate is transformed into the corresponding Dockerfile. To enable this, a mapping between TOSCA NodeTemplate and Dockerfile metamodel (eDockerfile), has to be explained. Table 5.4 summarizes the possible mapping between TOSCA NodeTemplate and eDockerfile, that is useful to generate a

Dockerfile for any NodeTemplate if needed.

NodeTemplate	**eDockerfile**
name, type	Dockerfile
Properties(port)	Exposeinstruction, port
Operation	Stage
Primary Artifact	From/Copy (or Add)/CMD instructions
Depedency Artifact	From/Copy (or Add)/Run instructions
Input	Environment Variable

Table 5.4: Mapping of TOSCA NodeTemplate to Dockerfile

As specified, type and name in NodeTemplate are used to extract dockerfile concept. More specifically, the type is firstly checked to decide whether we need to create a Dockerfile or not. This is done by verifying whether this type is different to the Nodetypes used to denote Docker, Compute, Volume and Network types while it should belong to one of the Nodetypes devoted for software and platform services. If so, the name in NodeTemplate will be matched to name in eDockerfile.
Furthermore, port property will be matched to the Envvar port in eDockerfile. Thus, an Exposeinstruction can be built based on this port and Expose command. Similarly, all Operations defined in the standard Interface in the NodeTemplate need to be exploited to infer other Dockerfile Instructions. In particular, each Artifact within each Interface Operation will be mapped to a set of equivalent Instructions. The primary Artifact is directly mapped to Frominstruction, Copyinstruction (or Addinstruction), and CMDinstruction. While dependency Artifact is directly mapped to FromInstruction (Iff its artifacttype is different to one in primary Artifact), Copyinstruction (or Addinstruction), and RUNinstruction. Both Artifact Properties and Inputs Operation are respectively mapped to Arguments and EnvVars that will be used in each above Instruction. For instance, artifacttype property within each artifact is mapped to the Argument Baseimage in the FromInstruction.
Finally, the order property of each Dockerfile instruction is a vital requirement that has to be considered in order to properly generate Dockerfile. This last is defined while considering artifact kinds (primary or dependency) and the instruction orders defined in the Dockerfile reference [49]. As a result, in each Stage for each Interface Operation, FromInstruction is firstly specified as well as From (if needed), Add and Run instructions for each dependency artifact and Expose instruction should be specified respectively before specifying all related instructions for the primary artifact.
After defining the relevant mapping between TOSCA NodeTemplate and eDockerfile, the transformation steps can be defined. Similar to TOSCATopology2Compose transformation, we proceed in two transformation steps, mainly M2M transformation outlined in Algorithm 3 and M2T transformation illustrated in the generation algorithm 4. Consequently, after applying the Algorithm 3, Figure 5.10 (a) depicts a partial Dockerfile Model for the NodeTemplate vote. Here, Create operation and primary artifact in the vote node are respectively matched to Create:Stage and Fromins:Frominstruction, Addins: ADDinstruction and Cmdins:CMDinstruction in the

Algorithm 3 M2M transformation Algorithm for transforming TOSCA Node into Dockerfiles model

input: TOSCAMM, DockerfilesMM, TOSCA application typology TO_{at}, Software types Sof_{type}, platform types Pla_{type}, Docker types Dok_{types}
output: Dockerfiles Model DF_{mo}
{**Verifying the need for generating a Dockerfiles for the node**}
for $node \in TO_{at}$ **do**
  if $node.type \in (Sof_{types} \cap Pla_{types})\ Dok_{types}$ **then**
    $DF_{mo}.name$= $node.name$
    for $operation \in node.interface$ **do**
      $DF_{mo}.stage$= map $Operation2Stategе()$
      {**Mapping Primary artifacts into the corresponding instructions**}
      $DF_{mo}.stage.FromIns$=map $PrimaryArtifact2FomIns()$
      $resolveOrder()$
      for $file \in operation.primaryartifact$ **do**
        $DF_{mo}.stage.AddIns$=map $File2AddIns()$
        $resolveOrder()$
      end for
      $DF_{mo}.stage.CMDIns$=map $PrimaryArtifact2CMDIns()$
      $resolveOrder()$
      {**Mapping Dependency artifacts into the corresponding instructions**}
      if $operation.dependencyartifact.type! = operation.primaryartifact.type$ **then**
        $DF_{mo}.stage.FromIns2$=map $DependencyArtifact2FomIns()$
        $resolveOrder()$
      end if
      for $file \in operation.dependencyartifact$ **do**
        $DF_{mo}.stage.AddIns2$=map $File2AddIns()$
        $resolveOrder()$
      end for
      $DF_{mo}.stage.RUNIns$=map $DependencArtifact2RUNIns()$
      $resolveOrder()$
    end for
    {**Mapping Port properties into the Expose instruction**}
    if $Exist(node.port)$ **then**
      $stage_{exp}$=create $newStage()$
      $stage_{exp}.ExposeIns$=map $Port2EXPOSEIns()$
      $resolveOrder()$
    end if
  end if
end for

resulted vote dockerfile model. Additionally, Figure 5.10 (b) depicts the generated dockerfile scripts for the vote Dockerfile model, after applying the generation algorithm.

Algorithm 4 M2T Transformation Algorithm for generating Dockerfiles scripts from the Dockerfiles Model

1: **input**: Dockerfiles Model DF_{mo}
2: **output**: Dockerfiles scripts $Dockerfiles$
{**Sorting the instructions in DF_{mo} according to their execution orders**}
3: $OrderedListOfins$= Sort_Executionorders(DF_{mo});
4: **for** $instruction \in OrderedListOfins$ **do**
5: Generate Instruction Scripts
6: **end for**

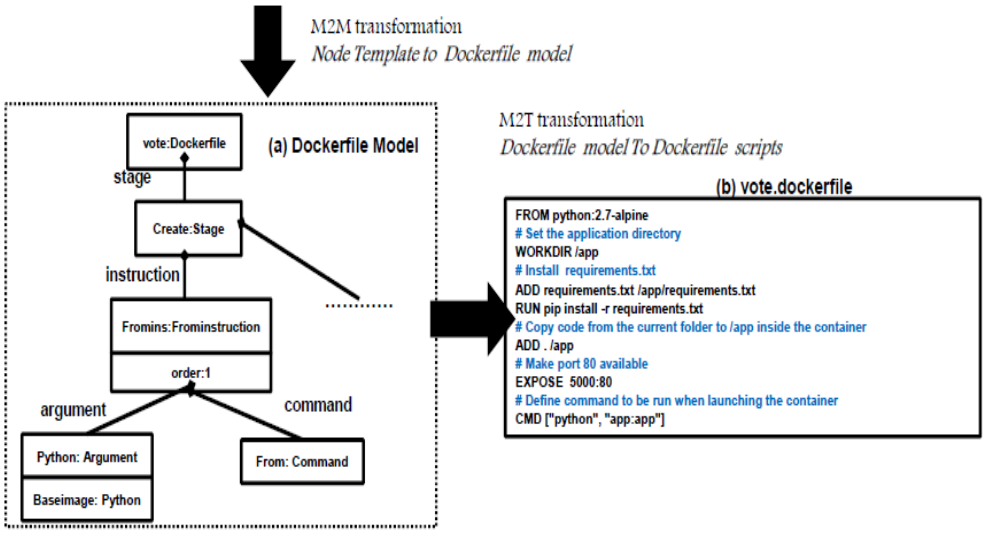


Figure 5.10: Transformation of the node template "vote" depicted in Figure 5.8 to Dockerfile-specific artifacts: (a) Dockerfile model; (b)Dockerfile script for installing "vote" service

5.5 Terraform case-study

5.5.1 Building Terraform metamodel

Terraform [149] relies on a declarative configuration language to build, modify, and versioning infrastructure descriptions. The aim is providing facilities to seamlessly manage large infrastructures that can span across multiple cloud providers. By an-

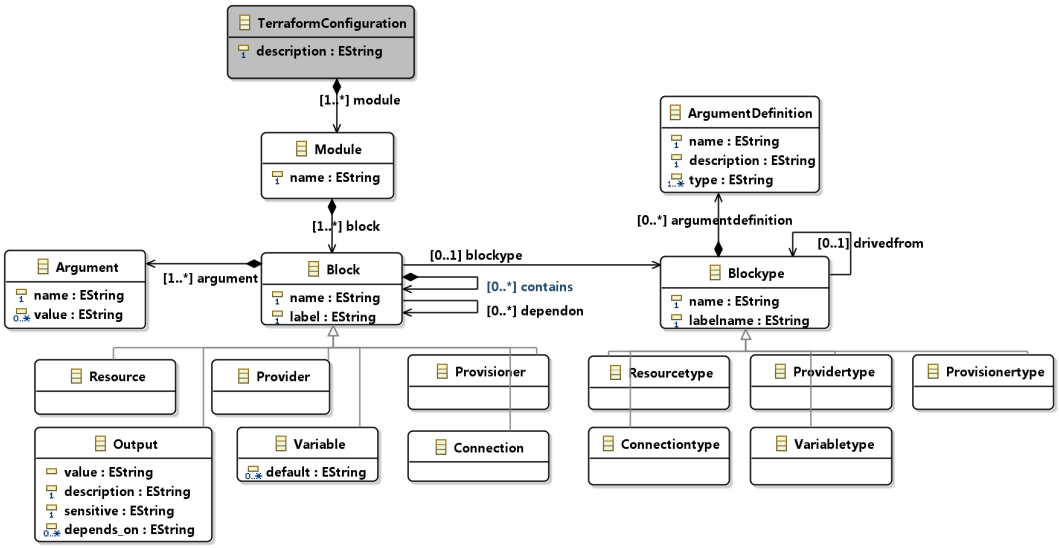


Figure 5.11: Terraform Metamodel (TerraformMM)

alyzing multiple Terraform references, we build the corresponding Terraform metamodel (refer to Figure 5.11) that we named as TerraformMM. Terraform is the root concept defining an encapsulated set of Modules. Modules are used to organize a set of Terraform configurations with the aim of creating reusable groups of resource configurations that can be easily shared between open-source developers. A Configuration represents the core Terraform entity, which enables the definition of any infrastructure object by assigning values to its specific attributes. Any configuration definition involves two key constructs, namely Arguments and Blocks. An Argument represents a simple assignment of value to a particular attribute. Whereas, Block encompasses a collection of Arguments needed to build a Configuration. There exist multiple kinds of block as follows:

- *Resource:* A resource block defines a particular infrastructure object with determined settings, such as compute instances, virtual networks, storage objects, etc.
- *Provider:* A provider block aims at configuring the cloud provider, from which the resources will be provisioned. This configuration requires specifying some provider settings such as authentication credentials, endpoint URLs, regions, and so one.
- *Provisioner:* A provisioner block defines specific operations that have to be executed on the remote or local machine for preparing the use of infrastructure objects by services. For instance, an operation can be the installation of an Apache Tomcat server on a given compute instance. Typically, the provisioner

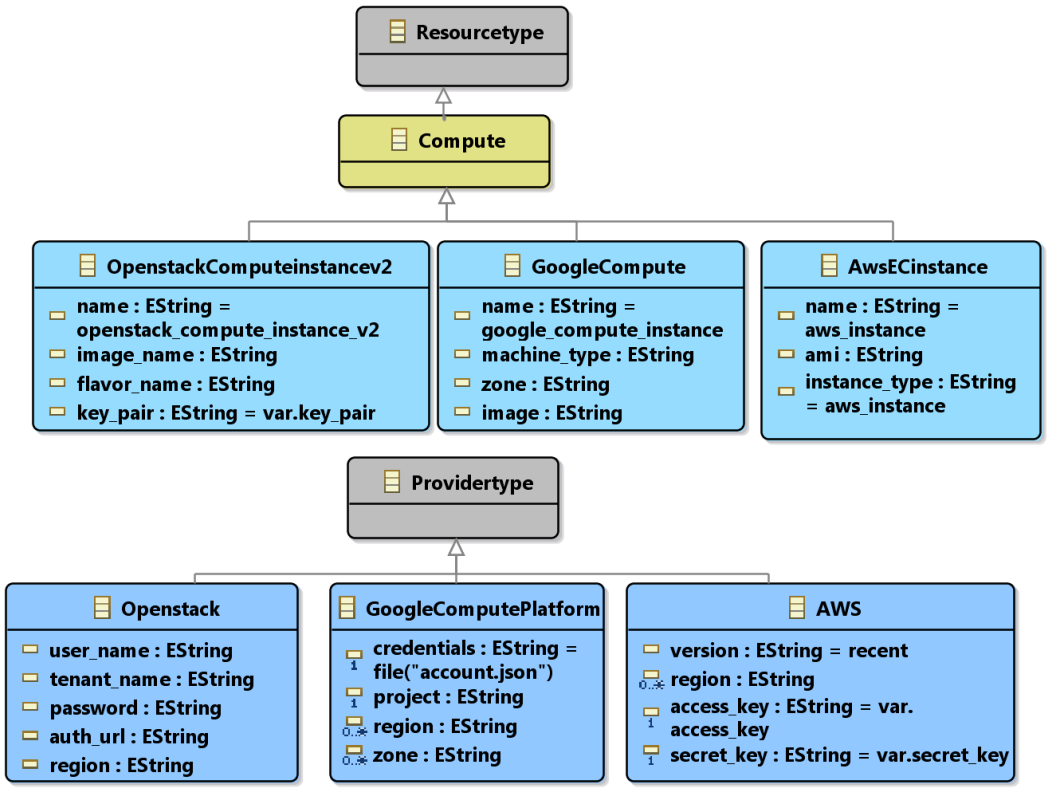


Figure 5.12: Concrete block types: (a) Compute resource types; (b) Cloud provider types

block is defined inside the resource block and whose execution can be at the creation or destruction of the related resource.

- *Connection:* The connection block defines how a provisioner can access to given resources. It typically defined inside resource or provisioner blocks.
- Variable: Variable block describes the input parameters for any Terraform configuration module. Typically, the defined variables are assigned to the argument values in the Terraform blocks.
- *Output:* Output block is used to define the return parameters after the execution of a given Terraform configuration. Typically, these parameters are passed to other Terraform configurations for subsequent management or communicated to the interested users.

Each one of the above blocks may have any number of specific arguments and require considering particular principles recommended by the Terraform curators. To support

this, we introduced the concept named as Blocktype, while its structure is inspired by the Entitytype which was introduced in the TOSCA standard. Blocktype is used to define the semantics of a block that has assigned to this type. As a result, multiple block types inherit the Blocktype are defined, including Resourcetype, Providertype, Provisionertype, Variabletype, Connectiontype, etc. For instance, the resource block is typed with Resourcetype, similarly, Providertype is used to define the type of a provider block, and so one. This classification is a natural result of the existence of multiple kinds of cloud resources, providers, middleware platforms that in turn dramatically contributed to introducing multiples resource and service configurations. Generally, through the ArgumentDefinitions and Constraints, each block type specifies the allowed properties and considerations that the related block may have. In addition, block types may inherit from each other, for instance, the Computetype which is specific to computing resources may inherit from the generic Resourcetype which is used to capture the general resources.
Furthermore, besides the abstract Terraform meta-model, we defined a set of normative block types by analyzing Terraform open-source projects and available configuration modules from the Terraform Registry [48]. These normative types represent the most recurring terraform configuration patterns, that can be either integrated as are or customized in other configurations. As depicted in Figure 5.12, these types include the following: *Compute types* define resource configuration blocks specific to AWS, OpenStack and Google engine compute; *Provider types* define provider configuration blocks for the same providers; And *Provisioner types* define provisioner configuration blocks that are devoted for installing Docker and ApacheTomacat server on a remote machine. It should be noted that other types can be defined in the same way. Finally, the defined types would be reused by our transformation method to produce configuration blocks compliant to these types.

5.5.2 Transformation of TOSCA to Terraform

Illustration example. Basically, Terraform is intended to support two main functionalities, namely the creation and provisioning of infrastructure and the configuration of appropriate middleware environments that allow the successful deployment of services on this infrastructure. In order to illustrate how our transformation could support these two functionalities, we rely on the TOSCA topology illustrated in Figure 5.13, partially reflecting our motivating example. The topology involves tow TOSCA nodes. The first node named vote representing a voting service that has to be hosted on the second node named as app-server1 representing the compute resource that would be provisioned by AWS provider with specific capabilities. This topology model will act as a source model for the transformation of TOSCA into Terraform-specific Artifacts.
Assumptions
As terraform is closely related to cloud providers and middleware platforms regarding

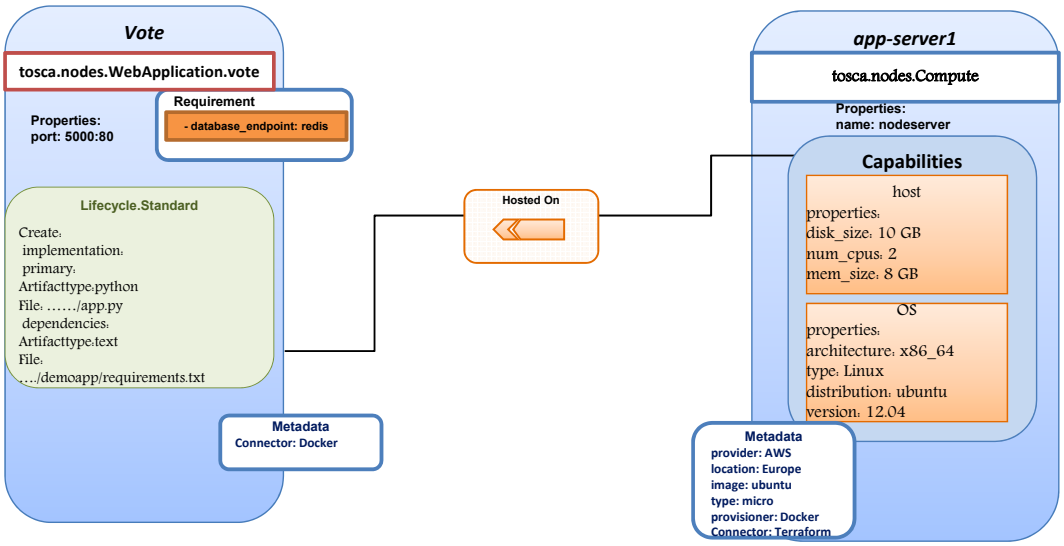


Figure 5.13: TOSCA topology

the way, in which resource, provider and provisioner configurations are described, we assume the following assumptions:

- *Assumption 1:* The provider-specific information has to be known before launching the transformation. For instance, in the case where a user selects AWS as a target provider, information includes the type of resource (e.g., Large, Micro, Medium, etc. [20]) and image (e.g. Ubuntu 64) could be computed from the properties related to the computing node capabilities (CPU, RAM, OS, etc) using one of the available selection algorithms [158]. Notice that the provider itself can be selected automatically if the user wants this choice. Furthermore, because they are irrelevant to the TOSCA topology and may contradict with its technology-agnostic nature, these data are added in the metadata section (refer to app-server1's metadata in Figure 5.13) related to any compute node and their computing is totally transparent to the user.

- *Assumption 2:* The used middleware platform has to be added in the metadata section of any compute node for being able to determine which provisioner has to be generated later. Similarly, the inferring of this data is done transparently to the user. Indeed, the Host-On relationship, type of hosted service and target

connector (Docker, or Kubernetes), if they are fixed by the user, represent the main indicators to pick up which middleware platform has to be installed and configured in the related compute node.

Transformation. Similarly to Docker Compose or any other DevOps solution, the transformation of TOSCA into Terraform-specific artifacts involves two phases: M2M Transformation and M2T Transformation. M2M Transformation is first performed allowing the generation of the Terraform configuration model from the pre-defined nodes for the infrastructure and middleware components. M2T Transformation is then required to produce the underlying Terraform definition files from the generated model at the first step. In addition to the mentioned assumptions, it is necessary to clarify the possible mapping between TOSCA and Terraform meta-models to enable both steps of transformation. The resulting mapping is summarized in Table 5.5.

Table 5.5: Mapping of TOSCA concepts to Terraform concepts

TOSCA concepts	**Terraform concepts**
TopologyTemplate	TerraformConfiguration
NodeTemplate	Module with Resource, provider, connection and provisioner blocks
Inputs	Module with Input blocks
Outputs Properties	Module with Output blocks Arguments
RelationshipTemplate/Requirement	dependson

As demonstrated the root concept TopologyTemplate can be mapped to TerraformConfiguration, which is a root concept in the target TeraformMM. Moreover, all Nodes template building the intended infrastructure can be represented by the same Module that comprises multiple configurations blocks: Resource, Provider, Connection, and Provisioner block. More precisely, while considering the provider-dependent data specified in its metadata section, each node typed with one of the common infrastructure types (e.g. Compute) will be represented by a configuration with a single resource block. Likewise, the provider and provisioner configurations are interpreted from the related node's metadata section that is obtained according to the Assumptions 1 and 2. Furthermore, each TOSCA node that represents a connection between two nodes, will be mapped to one connection configuration block. Indeed, current version of TOSCA does not include a node type that reflects such a connection object. Thus, the connection configuration block will be generated per default in the target Terraform configuration model. Moreover, the TOSCA inputs are directly mapped to a separate module, whose configuration is specified by means of input variable blocks. Here, each input will be represented by a separate variable block. In a similar way, TOSCA outputs will be mapped to a separate module containing one or more output configuration blocks. In addition, the set of block-related arguments can be deduced

Algorithm 5 M2M transformation Algorithm for transforming a TOSCA typology into a Terraform configuration model

input: TOSCAMM, TerraformMM, TOSCA application typology TO_{at}, Compute Types Com_{types}
output: Terraform configuration model TC_{mo}
Module *main*, *output*, *variable*; Configuration C_{pro}, C_{re}, C_{con}, C_{prov}; TargetProvider T_r; TargetProvisioner T_p;
{**Checking whether the node type is one of the compute types**}
for $node \in TO_{at}$ **do**
  if $node \in Com_{types}$ **then**
    $TC_{mo}.main$=map $Node2Module()$;
    {**Determining what is the target provider AWS, OpenStack, etc.**}
    T_r=Resolve ($node.metadata.provider$);
    $TC_{mo}.main.C_{pro}$= map Node2ProviderConfiguration (T_r);
    $TC_{mo}.main.C_{re}$= map Node2ResourceConfiguration (T_r);
    for $pro \in node.metadata.provisioner$ **do**
      T_p=Resolve(pro); {**Determining what is the target provisioner, e.g. Docker install**}
      $TC_{mo}.main.C_{pro}$= map Node2ProvisionerConfiguration (T_p);
      {**Adding the provisioner Configuration to the resource configuration**}
      Add($TC_{mo}.main.C_{re}$, $TC_{mo}.main.C_{pro}$);
    end for
    {**Creating a default Connection Configuration, whose type is SSH**}
    $TC_{mo}.main.C_{con}$=Create ConnectionConfiguration (SSH, u_r, p_{wr});
    {**Adding the connection Configuration to the resource configuration**}
    Add($TC_{mo}.main.C_{re}$, $TC_{mo}.main.C_{con}$);
  end if
end for
{*Mapping of relationship, inputs, and outputs are removed for brevity*}

from the properties related to each TOSCA entity (Node, Input, Output etc). Regarding relationships and requirements which capture the links between nodes, they are mapped to *dependson* links, which in turn reflect the potential relations between the resource configuration blocks.
After defining the relevant mapping between TOSCA and TerraformMM, the M2M and M2T transformation steps can take place. Algorithm 5 represents the M2M transformation logic. As a result of applying this algorithm to the source model depicted in Figure 5.13, a target Terraform model is automatically produced (see Figure 5.14). Accordingly, both *demoGlobalconfig:Terraform* and *main:Module* instances in the target model are introduced for the well-structuring of the target model. Besides, the source compute node *"app-server1"* is transformed into a target resource block named as *nodeserverconfi*, whose arguments are obtained from the metadata related to this latter.

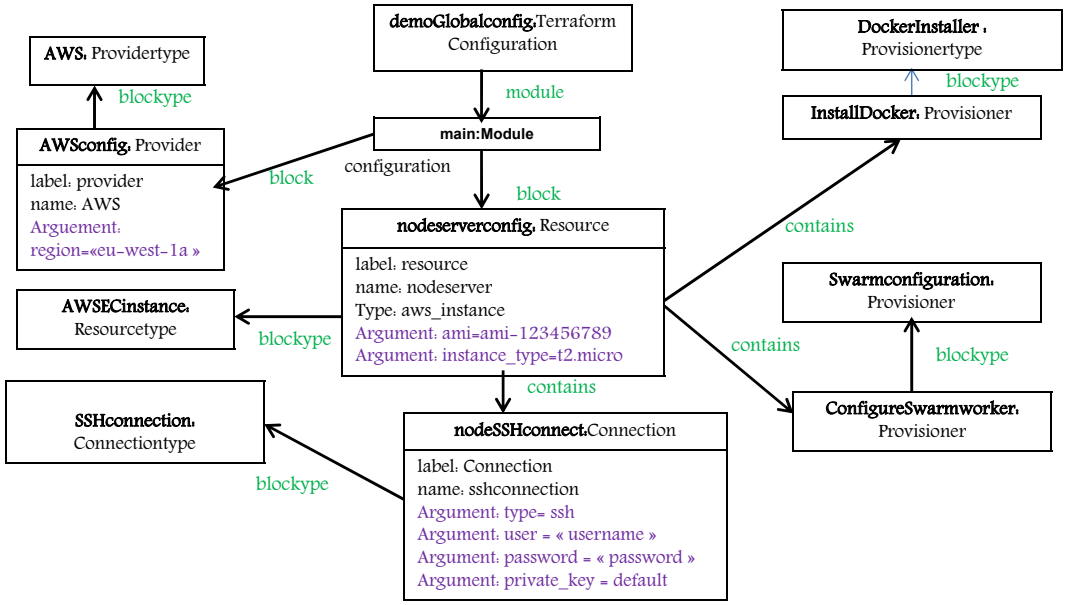


Figure 5.14: Target Terraform configuration model

In addition, since the source compute node is supposed to be provisioned by AWS provider, the target *nodeserverconfig* is assigned to AWS_instance type. Furthermore, the provider metadata related to the source node is used to create the provider block named as *AWSconfig*, whose *blocktype* refers to *AWS provider type* and *region* argument assigned to *"eu-west-1a"*. Similarly, the provisioner metadata filled with

"Docker" in the source node is used to create two provisioner blocks named *"InstallDocker"* and *"ConfigureSwarmworker"*, that are contained in the target resource block. These latter blocks are of type *"DockerInstaller"* and *"Swarmconfiguration"*, respectively. Finally, the connection block named *nodeSSHconnect* is generated by default.
Finally, the obtained target Terraform model will be the main input of the M2T transformation defined by a corresponding algorithm. As a result, Figure 5.15 presents ready-to-execute Terraform scripts for a concrete deployment of the expected cloud infrastructure.

```
# AWS # Setup the cloud provider "Amazon Web Services" (AWS)
provider "aws" {
access_key = "${var.access_key}"
secret_key = "${var.secret_key}"
region = "${var.region}"
}
# Setup of docker_manager resource
# removed for brevity as it is generated regardless of TOSCA Topology
# Configuration of nodeserver" resource mapped from the TOSCA compute node: nodeserver
resource "aws_instance" "nodeserver" {
ami = "${var.ami}"
instance_type= "${var.instance_type}"
#Define how to connect to this resource via ssh protocol
connection {
# Connection type
type = "ssh"
user = "username"
private_key = "${file("InstanceAccessKeys/privateKey.pem")}"
password = "password"}
#Provisioner block is used to execute the required scripts to deal with docker services deployed on this resource
provisioner "remote-exec" {
script = "install-docker.sh"
}
provisioner "remote-exec" {
inline = [
"docker swarm join --token ${data.external.swarm_join_token.result.worker}
${aws_instance.docker_swarm_manager.private_ip}:2377"
]}}
```

Figure 5.15: Terraform configuration scripts

5.6 Implementation

In this section, we discuss the implementation of our proposed approach. We developed a proof-of-concept called *ToDev*, an integrated and standards-driven orchestration framework based on TOSCA and DevOps technologies. The framework includes open-source DevOps solutions, namely Docker, Terraform, and Kubernetes (Partially) and empowered with MDE facilities to manage the involved models and transformation algorithms.

5.6.1 Proof of Concept: ToDev

Our *ToDev* proof-of-concept aims at supporting DevOps users in the orchestration of their cloud applications without a significant learning curve. As shown in Figure 5.16, the internal architecture of ToDev is organized into diverse layers communicating with each other.

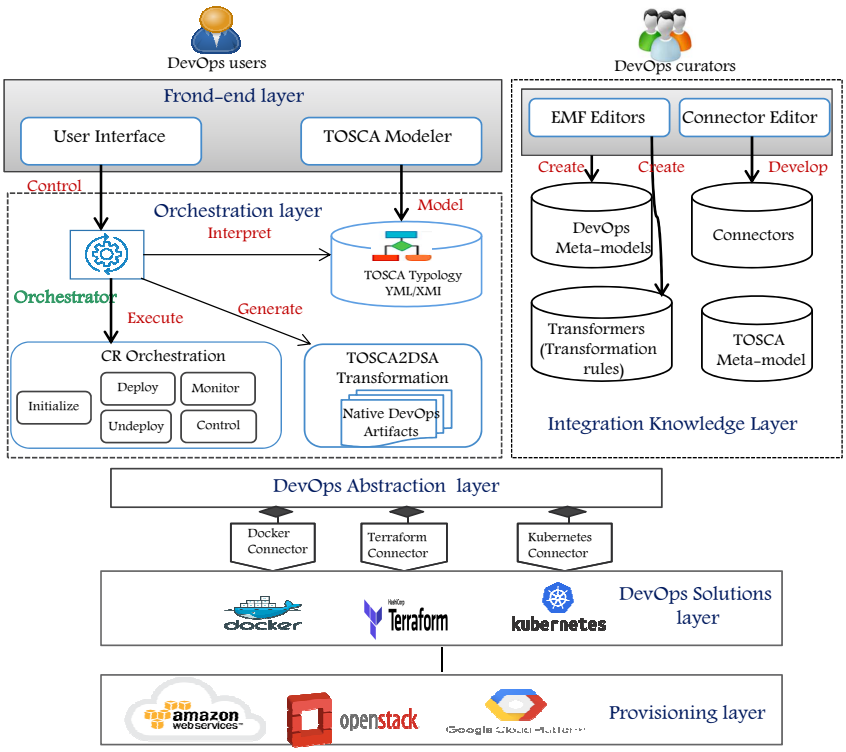


Figure 5.16: ToDev Architecture

- **Frond-end layer.** It is consists of two components: TOSCA modeler and User interface. TOSCA modeler allows DevOps users to graphically model a TOSCA-compliant application typology, representing the required services and resources as nodes while capturing the dependencies between them by means of relationships. The obtained application instance is then simply serialized as YAML/XML file and communicated to the **Orchestration layer** for ensuring the subsequent orchestration operations of the described application. A key part of our TOSCA modeler implementation involves integrating xtext [56] and Sirius [55] technologies. xText is acknowledged as an Eclipse-based development framework for creating textual programming languages. Whereas, Sirius is an open-source Eclipse project allowing the creation of graphical workbenches for the domain-specific models based on four main dialects: diagrams, trees, tables,

and matrices. The second important component in the Front-end layer is the User interface which is devoted to act as the mediator between DevOps users and the underlying orchestrator.

- **Integration layer.** It is in charge of providing the desired integration between TOSCA and DevOps solutions. Precisely, this layer implements the core part of our model-driven integration approach. Herein, appropriate meta-models, transformations, and connectors are defined and registered. We assume that one DevOps curator (e.g. domain expert) for a particular DevOps solution can contribute to this task. With the aim of implementing this layer, we adopted the Eclipse Modeling Framework (EMF) [54] for building all required meta-models and generating the corresponding manipulation and testing plugins to support their utilization later in the transformation phase. EMF is chosen as it is the most widely used modeling framework today, providing rich constructors and mechanisms for describing meta-models and automating their implementation code as well.

```
1
2  modeltype TOSCA uses "http://www.example.org/TOSCA";
3  modeltype compose uses "http://www.example.org/compose";
4
5  transformation TOSCAToComposeTransformation(in Source: TOSCA, out Target:
       compose);
6  main() {
7          var a :=Source.rootObjects()[ServiceTemplate];
8           a.map TOSCAToCompose();
9           }
10
11 mapping ServiceTemplate::TOSCAToCompose(): Compose {
12 description:=self.description;
13 services:=self.topologytemplate.entitytemplate->map NodeMaptoService();
14 networks:=self.topologytemplate.entitytemplate->map NodeMaptoNetwork();
15 volumes:=self.topologytemplate.entitytemplate->map NodeMaptoVolume();}
16
17 mapping EntityTemplate::NodeMaptoService() : Service
18 when {self.type="Application" and self.Resourcetype="DataBase" and ...} {
19 name:=self.name;
20 dependon:=self.oclAsType(NodeTemplate).requirements.nodetemplate.late
       resolve(Service);
21  properties:=self.properties->map ToProperties();
22   image:=self.oclAsType(NodeTemplate).artifacts-> map toImage();
23 // mapping of other attributes
24       }
25
26    mapping EntityTemplate::NodeMaptoNetwork() : Network
27          when {self.oclIsTypeOf(NodeTemplate) and self.type="tosca.nodes.
       Network"} {
28        name:=self.name;
29 // mapping of other attributes
30  }
31
32  mapping EntityTemplate::NodeMaptoVolume() : Volume
33          when {self.oclIsTypeOf(NodeTemplate) and self.type="tosca.nodes.
       Storage"} {
```

```
        name:=self.name;
// mapping of other attributes
        }
```

Listing 5.1: QVTo implementation of the Algorithm 1 for the TOSCA2Compose M2M transformation

In addition, we implement our model-driven transformation technique as a set of automated transformers, each one represents a concrete implementation of a particular transformation algorithm presented above. More specifically, we selected QVTo language to implement the M2M transformation and Xtend2 language to implement M2T transformation. For instance, in the context of the TOSCA to Docker transformation, listing 5.1 presents a QVTo implementation of Algorithm 1 for TOSCA2Compose M2M transformation, whose main steps are specified using the so-called mapping rules. Whereas, listing 5.2 represents the implementation of Compse2Yaml transformation (See Algorithm 2) using the Xtend2 template features.

```
def ComposeModel2Composefile(Compose composeinstance, IFileSystemAccess
   fsa) {
 fsa.generateFile(composeinstance.name +".yml", '''
    version: <<composeinstance.version>>
    services:
     <<FOR service: composeinstance.services>>
         <<service.name>> :
       image: <<service.image.name>>:<<service.image.version>>
        ports:
             "-<<service.ports>>"
         depondon:
                 <<FOR element: service.dependon>>
                 -<<element.name>>
                 <<ENDFOR>>
          networks:
                <<FOR network: service.networks>>
                  -<<network.name>>
                  <<ENDFOR>>
          volumes:
             <<FOR volume: service.volumes>>
                  -<<volumes.name>>
                 <<ENDFOR>>
          deploy:
            replicas: -<<service.replica>>
                /*Adding other service properties is possible*/
    <<ENDFOR>>
networks:
      <<FOR network: composeinstance.networks>>
           <<network.name>> :
       <<ENDFOR>>
  volumes:
       <<FOR volume: composeinstance.volumes>>
                <<volume.name>> :
         <<ENDFOR>>
         ''')}
```

Listing 5.2: xTend2 template implementation of Compose2Yaml transformation

Besides, we provided a Connector editor that enables DevOps curators to define the desired Connector as an instance of the ECMM meta-model. Once the connector instance is defined, an M2T transformation algorithm implemented using xTend2 is applied to generate the native java code related to this connector. If needed, this connector code must be completed by DevOps curators.

- **Orchestration layer.** Orchestration layer is governed by the so-called **Orchestrator component** which is in charge of managing the typical application life cycle at the behest of DevOps users. This orchestrator firstly interprets the TOSCA application typology defined by the DevOps user in order to generate all DevOps-specific artifacts required for the orchestration purpose. This generation is done by executing the appropriate transformers that exist in the **Integration layer**, on the source application typology. Once the required artifacts are available, the DevOps user can initiate the orchestration process of the provided application. For doing so, the orchestrator collaborates with the **DevOps abstraction layer** that is empowered by a set of Connectors to execute the end-to-end orchestration operations. In the scope of this chapter, four main operations are supported, namely: Initialisation, Deployment, Undeployment and Controlling (Start, Stop, Restart). Other operations including monitoring and advanced runtime controlling, in particular, scaling and dynamic reconfiguration are the subject of the next chapter as they are especially devoted to support the multi-management of elasticity.

- **DevOps and Provisioning layers.** Both layers include the external components of our ToDev framework. DevOps solutions layer consists of the DevOps solutions that offer the concrete execution of orchestration operations by communicating with cloud providers' platforms and requesting the enforcement of the appropriate management actions. Whereas, Provisioning layer includes the different cloud providers infrastructures providing resource capabilities to ensure the successful hosting and management of user applications.

5.7 Experiments and Validation

This section discusses the validation of our approach by conducting three experiments using the ToDev framework described above. In the following, we present our evaluation objectives, describe the diverse use cases that we used, and finally detail and analyze the conducted experiments.

5.7.1 Objectives

In order to evaluate the effectiveness of our approach, we fixed the following objectives while outlining the utility of each one:

1. Since we want to demonstrate that our model-driven integration between TOSCA and DevOps solutions can streamline the orchestration process, we will evaluate the gained productivity in comparison to a specific DevOps solution that implies the manual configuration of Docker and Terraform.

2. Since our underlying solutions (i.e. model-driven transformation and Connectors) for supporting the desired integration may introduce a counter-effect, we will evaluate the overhead introduced by our approach in comparison to the manual configuration of Docker and Terraform.

3. Because of our model-driven transformation technique represents a key factor for offering the desired integration, we will evaluate the performance (in terms of execution time) of transforming of TOSCA into DevOps-specific artifacts.

5.7.2 Use cases

With the aim of evaluating the above objectives: we adopted the following application use cases.

1. **Use case 1:** represents an open-source Wordpress application that comprises a Web application front-end installed onto a local Apache Tomcat server.

2. **Use case 2:** represents the application of the motivating example, with the only difference that it has to be deployed only on the Google Compute Platform (GCP). It consists of five services and 2 GCP virtual servers.

3. **Use case 3:** represents the application of the motivating example, with the only difference that it has to be deployed only on the amazon web service platform. It consists of five services and 2 AWS virtual servers.

4. **Use case 4:** represents the application of the motivating example that will be deployed both on AWS and GCP platforms. It consists of five services and 2 AWS virtual servers and 2 GCP virtual servers.

5. **Use case 5:** represents a pizza store application consists of Nodejs, MongoDB, Elasticsearch, Logstash, and Kibana services. Each one of these services has to be deployed on a separate server from AWS cloud. In addition, a demo pizza application has to be hosted on the Nodejs server which is enabled by some monitoring facilities using the Syslog and Collectd services that are devoted to collect the logs. In total, there are seven services and five AWS servers, which are created with this deployment.

5.7.3 Testbed environment

Through this evaluation, we have used the testbed environment shown in Table 5.6.

Characteristics	**Local DELL machine**	**AWS Servers**	**GCP Servers**
Description	A physical machine in which ToDev is deployed	2 to 6 elastic compute cloud (EC2) instances of *a1.large* type, that are used to deploy services in the use cases 2, 4, and 5	3 compute instances of *n1-standard-1* type that are used to deploy services in the use cases 3 and 4
Memory	11 GB	4 GB	4 GB
Processor	Intel(R) Xeon (R)CPU W3530 @2.80GHz	2 vCPU	2 vCPU
Disk	233.3 GB	8 GB	64 GB
Operating system (OS)	Ubuntu 16.04 LTS amd64		

Table 5.6: Characteristics of the used Testbed environment

5.7.4 Experiment 1: Productivity evaluation

We evaluate the productivity in terms of (i) the time taken to complete the modeling task and (ii) the total number of lines-of-code (LOC) of the DevOps-specific artifacts generated by our transformation technique for three uses cases 2, 3 and 4. For instance, Figure 5.17 provides a tree structure view on the generated artifacts regarding the third use case. More precisely, our approach generated compose.yml file describing all involved services, 2 Dockerfiles for Vote and Result nodes and a set of Terraform scripts files devoted for configuring the virtual servers provisioned from the AWS provider. We checked the correctness of all these artifacts by executing their full deployment on AWS. For quantitative comparison purposes, we performed the same task with Docker and Terraform without any support from our approach, that is all DevOps-specific artifacts are manually created based on the technical requirements of each involved DevOps solution. The results of the experiment in Table 6.5 shows the

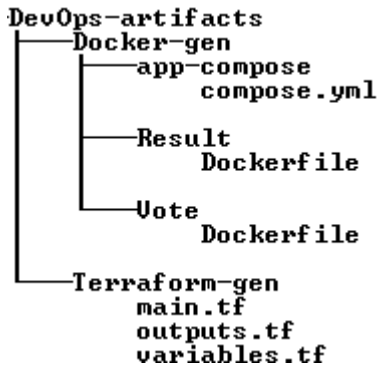


Figure 5.17: Generated artifacts for Docker and Terraform from the TOSCA typology of use case 3

time taken and LOC using our ToDev framework and the combination of Docker &

	Parameters	Our Approach	Docker & Terraform
Case 2	time for the modeling task (min)	31.6	120
	lines-of-code	381 (TOSCA YML file+ generated artifacts)	247
Case 3	time for the modeling task (min)	31.6	120
	lines-of-code	382 (TOSCA YML file+ generated artifacts)	246
Case 4	time for the modeling task (min)	45.9	140
	lines-of-code	544 (TOSCA YML file+ generated artifacts)	360

Table 5.7: Evaluation of the overall productivity

Terraform, for each selected use case. As shown, the time taken to complete the task was significantly reduced using our approach in comparison to the other solution (i.e., Docker & Terraform). For instance, regarding Case 2, it was reduced by 70%, where the task took, on average 31,6 min using our approach and 120 min using Docker & Terraform. We argue that the adopting of a standard like TOSCA improves the time-to-modeling. Moreover, considering the mapping and translation support offered by our approach to generate the required artifacts, the time-to-modeling can only improve several-fold. Moreover, the number of LOC was increased using our approach. For instance, regarding Case 2, Docker & Terraform required 247 lines-of-code while our approach produces 381 lines-of-code, where 135 lines are related to the TOSCA model that is designed by the user and 246 lines are generated from this model. Given these observations, we confirm the improved DevOps productivity offered by our approach in terms of reducing modeling time and generating the corresponding artifacts without any loss.

5.7.5 Experiment 2: Overhead evaluation

We evaluate the overhead introduced by our approach regarding both the single and multi-cloud of cloud applications. Accordingly, we deployed the applications introduced in use cases 1, 2, 3, 4 according to two scenarios; (i) using our approach and (ii) using deployment scripts executed directly by the Terraform and Docker command-line interfaces. For better estimation of the introduced overhead, we repeat the deployment of each use case according to the two scenarios ten times.
Table 5.8 shows the results in terms of variance, average, minimum and maximum time values, as well as the overhead introduced by our approach for deploying each use case. There is no stark difference between the obtained overhead for the different uses cases. Even though the use case 4 involves multi-cloud deployment across two providers, which requires more configurations comparing to the use cases 2 and 3, it does not provide the highest overhead. Indeed, from a technical point of view, we can say that the coordination between the Orchestrator and the underlying DevOps

Use cases	Scenario	variance	minimum (sec)	maximum (sec)	average (sec)	Overhead
Use case 1: Baseline Local deployment	deployment scripts	0.9	9	12	10,5	-
	Our approach	0.9	10	13	11,5	**1,09 %**
Use case 2: Single deployment on GCP	deployment scripts	2,6	162	167	164,4	-
	Our approach	8,9	165	176	171,4	**1,04%**
Use case 3: Single deployment on AWS	deployment scripts	2,6	152	157	154,7	-
	Our approach	4,7	160	166	162,4	**1,04%**
Use case 4: Multi-deployment on AWS and GCP	deployment scripts	4,2	325	330	327,4	-
	Our approach	2,9	340	345	341,7	**1,04%**

Table 5.8: Deployment Overhead

connectors is the main reason for introducing the observed overhead **(**$\sim 1\%$**).** Despite that, we can confirm that the introduced overhead by our approach is very small and still negligible compared to the provided benefits.

5.7.6 Experiment 3: Transformation evaluation

With the aim of evaluating the transformation performance, we perform a series of transformations using all of the selected use cases. Accordingly, we first use our ToDev framework to model the corresponding TOSCA topology model for each use case. Afterward, we execute the transformation for each TOSCA topology model to generate the equivalent DevOps-specific artifacts ten times. Here we note that we use the first use case to establish our baseline transformation as it only contains a WordPress node hosted on a server node.

Parameters	**Case 1**	**Case 2**	**Case 3**	**Case 4**	**Case 5**
Complexity (LoC)	45	135	136	184	227
Average transformation Time (sec)	5	10,4	10,1	11,5	13

Table 5.9: Evaluation of the transformation performance.

Table 5.9 shows both the complexity of the TOSCA topology model for each use case and the average time of its transformation toward DevOps-specific artifacts. The complexity of each TOSCA topology is measured in terms of number of lines-of-code. For example, Case 2 has 135 lines of code, which in turn corresponds to 41 instances of TOSCA elements computed as follows: 8 nodes+8 types+8 properties+5 requirements +5 operations+7 artifacts. By looking at the obtained transformation times, we observed that it remained consistently small for each use case. Although the complexity is growing, the time for each use case remained between 5 and 13 ms. Therefore, we confirm the reasonable performance offered by our transformation technique.

5.8 Conclusions

In this chapter, we achieved our objective mentioned in the thesis problematic which is streamlining and improving the orchestration of cloud resources. For doing so, we provided two key mechanisms to support a seamless integration between TOSCA and DevOps solutions. Concretely, we provided a model-driven translation technique to translate TOSCA applications into native DevOps-specific artifacts that are ready to be executed by the underlying DevOps tools and APIs. Our translation technique follows the key MDE principles, i.e. the use of models as first-class entities and transformation language support. This has been advocated in order to ensure an automated mapping between TOSCA and DevOps solutions at a high level of abstraction, which eases its synchronization and extension when needed. Besides, the second mechanism is the DevOps abstraction layer that is based on a set of high-level connectors in order to avoid the heavy lifting involved when interacting with the underlying tools/APIs of target DevOps solutions.
Our model-driven approach is concretized by creating a proof-of-concept called ***ToDev*** which leverages open-source DevOps solutions and MDE facilities. We validated our approach by conducting a set of experiments especially devoted for demonstrating its gained productivity, introduced overhead and its transformation performance. Experiments demonstrate that our approach provides a powerful enhancement to DevOps productivity, introduces negligible overhead and has a reasonable transformation performance.

Chapter 6

Managing multi-cloud elasticity using higher-level abstractions based on state machine

Contents

6.1 Introduction

This chapter introduces our approach towards supporting the high-level management of multi-cloud elasticity. Specifically, we focus on providing high-level abstractions with which cloud elasticity features can be intuitively and easily described, as a step toward coping with the actual cloud heterogeneity and the inherent technological challenges. With this in mind, we advocate the support of elasticity features at the resource description level by decoupling its specifications from low-level and technology-dependent resource reconfigurations.
Concretely, we propose a novel **c**loud **E**lasticity **D**escription Model (cEDM) to specify elasticity features related to cloud resources regardless of the technical specifications of any resource provider platform or any enforcement mechanisms. Our model is endowed with a novel abstraction known as Cloud resource requirement and constraint State Machines (C-SM) that allow capturing elasticity behavior through a flexible characterization of resource requirement variations during different phases of the application life-cycle. More precisely, in this abstraction, we use the notion of states to characterize application-specific resource requirements (e.g., CPU and storage usages), and constraints in terms of costs and SLA objectives (such response time should be less 3 seconds, etc.). Whereas, we use the notion of transitions to automatically trigger controller actions (scale-out VM, migrate VM, etc.) when certain conditions (i.e. resource CPU usage increases beyond a certain threshold, etc.) are satisfied in order to perform the desired resource (re-) configurations to satisfy the requirements and constraints of target states.
To support the execution of our C-SM, we generate the appropriate low-level artifacts required for the online monitoring, execution and controlling of all elasticity policies described under C-SM. With the aim of avoiding the complexity incurred when manipulating and maintaining these kinds of artifacts which are often heterogeneous, we propose to structure them as Event-Condition Action (ECA) rules. These ECA rules would then exploited by a rule engine in order to support the enforcement and runtime control of elasticity. We choose to deal with the Drools rule engine because it is known by providing a good compromise between the expressivity and performance. Furthermore, for making decisions about which elasticity actions have to be executed, it is required to provide the related operation data, including resource metrics, states, events, etc. To provide this need, we propose an integrated monitoring system that is generic and technology-independent enough for being able to collect and analyze data across different layers and heterogeneous clouds. The proposed monitoring system will work in tandem with the Drools engine to provide an appropriate runtime environment to execute elasticity policies defined in our model.
Finally, all these solutions have been prototypicality implemented toward providing a holistic and integrated system for high-level management of multi-cloud elasticity, that we named as cEDMCore. With the aim of validating the feasibility of our system, we conduct two extensive experiments that specially devoted to evaluate two

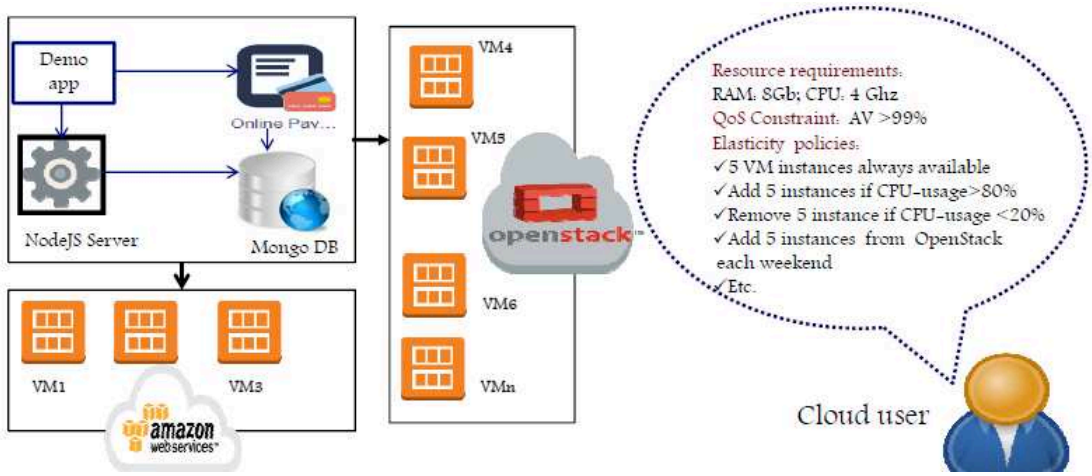


Figure 6.1: Cloud resources and their elasticity requirements for deploying an E-commerce application

assessment criteria: productivity and performance. The productivity evaluation is performed using a case-study with academic and professional users and their results compared to existing solutions like the IBM Softlayer Auto Scale [78] and AWS Auto Scaling [26] and Terraform [149].
The remainder of this chapter is structured as follows: In section 6.2, we articulate the motivations of our novel model through a real cloud scenario. In section 6.3, we introduce our model, which consists of a set of high-level abstractions to capture the elasticity behavior related to cloud resources. Section 6.4 describes our proposed system and the underlying components. Section 6.5 presents the evaluation of our work and the interpretation of the experimental results. Finally, an overview of our chapter conclusions is given in Section 6.6.
The work in this chapter was published in conference proceedings [35] and peer-reviewed journal [36].

6.2 Motivation

In this section, we investigate through a motivating example, specific limitations among existing cloud resources elasticity solutions.

Motivating Scenario. Consider a cloud user wants to specify resource requirements, constraints and elasticity policies for deploying an e-commerce application (refer to Figure 6.1), which consists of a MongoDB service, a JS application based on a NodeJS server and online-payment service . The cloud user selects Amazon web services (AWS), where each application service could be hosted on a separate Virtual machine (VM). For instance, he/she needs 5 instances to be always available for a specified period regarding the VM that will host the JS application. Each VM in-

stance has 8 GB RAM and 4 GHz CPU. As QoS constraints, the user would like that the availability for each instance must be at least 99%. Moreover, when the average CPU usage for 5 minutes is greater than or equal 80%, she wants 5 more instances for each VM to be added from AWS. In contrast, these 5 instances should be removed whenever the average CPU usage is less than 20%. However, during the business spikes every weekend, whenever the application reaches 10 instances in AWS, she wants to horizontally scale out into another cloud like Openstack or Google compute platform (GCP) by adding 6 instances. In the same way, the requirement, constraints and elasticity policies for the other application resources can be defined.
To realize this scenario, the cloud user has two possible solutions to follow: using proprietary solutions of the selected cloud provider or developing from scratch a customized program. While the first solution requests less development effort than the second, it limits the cloud user to one cloud provider. Precisely, proprietary solutions of a particular cloud provider cannot be applied to manage and control elasticity of cloud resources acquired from other providers. Therefore, cloud user has to follow the second solution to exploit cloud resources from multiple providers. This implies that cloud user has to employ multiple monitoring and enforcement tools either from cloud providers (e.g. AWS, Openstack, GCP) or existing DevOps solutions (e.g. Docker, Terraform, Juju, etc.) to support different management aspects related to elasticity. However, with the expanding complexity of cloud, this proves as too cumbersome and time consuming task. Moreover, the fact that each cloud solution employs its individual resource description models, management interfaces and capabilities and relies on low-level script-based APIs, can only complicate this task several-fold .
Based on these observations, we concluded that existing cloud resources description models (RDMs) and elasticity solutions (1) are rarely transparent and adaptive to support the management of resources across various providers; (2) oblige users to acquire new expertise in multiple RDMs and different elasticity implementation mechanisms; and (3) lead to a costly environments and potential vendor lock-in as exploiting resources from a new provider demands an extensive programming effort [47]. To resolve the challenges related to cloud resources description, previous research follows two possible approaches: proposing a new unified model or using standards like TOSCA [116] and OCCI [118]. However, less attention has been paid to elasticity features, especially the issue of their modeling. Therefore, in our work, we are specifically interested in providing high-level modeling abstractions, with which cloud resource elasticity features can be intuitively described without referring to any Provider/DevOps specific elasticity mechanisms (e.g . specific monitoring and reconfiguration APIs)

6.3 An embryonic elasticity description model

In this section, we identify a set of key high-level modeling abstractions that allow users to specify their required cloud resources and associate them with the intended

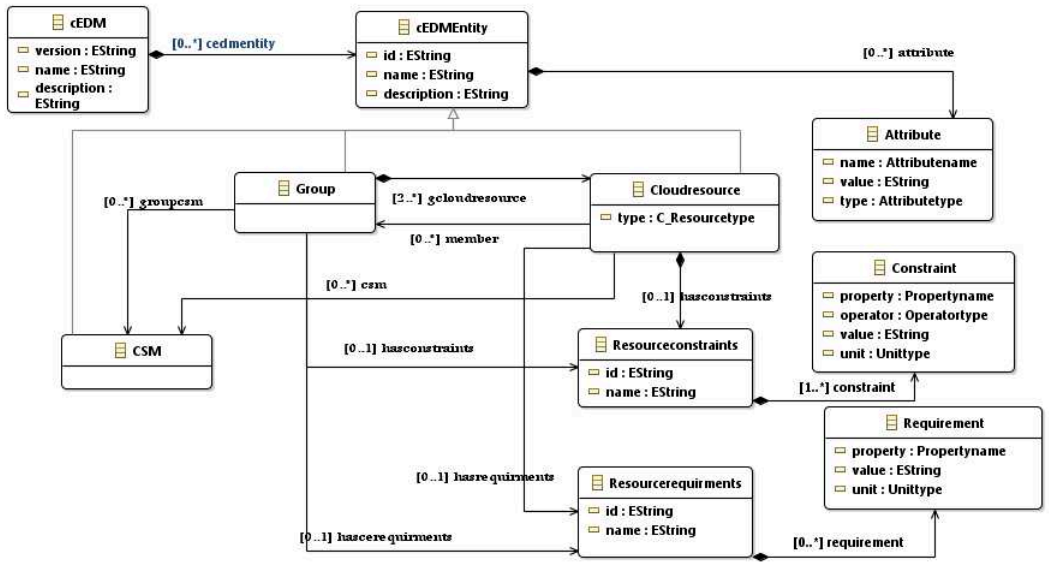


Figure 6.2: UML class diagram for the cloud elasticity Description Model (cEDM)

elasticity polices without referring to any low-level or specific technologies. To do so, our original intent orients toward understanding the common concepts and characteristics used when describing cloud resources and their elasticity features from several providers, including AWS [26], Azure [107], GCP [72] , IBM [78], Openstack [123] and from diverse DevOps solutions, including, kubernetes [90], Docker [49], Juju [152], and Terraform [149]. We selected these solutions as they are widely adopted, represent the range of different types of available cloud solutions (commercial offers and open source implementations) and target diverse cloud resource types (IaaS, PaaS, SaaS). Our analysis strategy is enough rational, and is based on two main principles:

- determine a set of abstractions that are very simple, so we can start from a minimal base and progressively extend it as needed.
- avoid the complex abstractions that we could have thought useful, and that may even be needed in some cases, but that are rarely used in practice.

Accordingly, in Section 6.3.1, we focus on the basic abstractions to be included when a cloud resource is acquired from a single provider. Whereas we are showing in Section 6.3.2 how to extend these abstractions in order to support the multi-provider scenario.

6.3.1 Abstractions overview

Fig. 6.2 represents the conceptual UML model illustrating the main abstractions of our cloud elasticity Description Model (cEDM). To illustrate it, we rely on the moti-

vating example previously described in section 6.2, which consists in deploying an E-commerce application both on AWS and OpenStack. As shown in Fig.6.2, *cRDMEntity* is the top level entity in our model, which can be specialized into *CloudResource*, *Group*, *C-SM*, .

- **CloudResource**: indicates the target of elasticity (i.e. scope), which can infrastructure, platform, or a software object, and potentially everything as a service from hardware resources to business applications. It is described by a reference id and name inheriting from the cEDM entity. *id* is assigned automatically by the system, *name* indicates the used reference name for this resource in the resource description model. Here it should be noted that the entire description of the concerned resource is given by the used resource description model, which is TOSCA [116] in our context, but it can any other model. Typically, Cloud users have diverse requirements that need to be satisfied by the provider when selecting the desired cloud resources. In addition to the requirements, cloud users naturally may define constraints over cloud resources, which must be respected at the selection and runtime. Motivated by these needs, we associate the cloud resource entity with the following two concepts: Resource Requirements, Resource Constraints

  - **Resource Requirements:** include the set of requirements regarding certain cloud resource properties. The set of *Resource Requirements* is defined by a logic *name* and a reference *id* and consists of one or more *requirements*, where each one is defined by *property-value* pairs for describing the desired requirement, such as CPU, RAM, provider, region, etc.
  - **Resource Constraints**: include the set of constraints regarding certain cloud resource properties that need to be respected at the selection phase and runtime. Resource Constraints are defined through a *reference id* and a *logic name* and is composed of one or more *constraints*. Each constraint is expressed through a set of attributes including *property* indicates the resource metric; *operator* indicates the comparison operator such =, >=, etc.; *value* indicates a value for this property ; and unit indicates the used *unit* if needed.

- **Group**: is introduced with the aim of simplifying for cloud users the manipulation and the configuration of complex systems that involve exploiting a big number of cloud resources. It allows grouping cloud resources that share the same elasticity behavior and managing them as one group. It is defined through a *reference id*, a *name*, a *description* and is composed of at least two cloud resources.

- **CSM**: represents an extended version of the traditional state machine, allowing the definition of the elasticity behavior of a cloud resource. Indeed, state-based

models have been broadly used to model the reactive behavior of systems. This inspires us to use this model in order to capture the elasticity behavior by characterizing the resource requirement variations over time and different phases of the application life-cycle. For instance, a conference submission site will require more resources one week before each deadline, but less at other times. Conceptually, the C-SM machine is defined through a logic name, a reference id and is composed of a set of states and a set of transitions. We describe state and transition abstractions in the subsections 6.3.1.1 and 6.3.1.2.

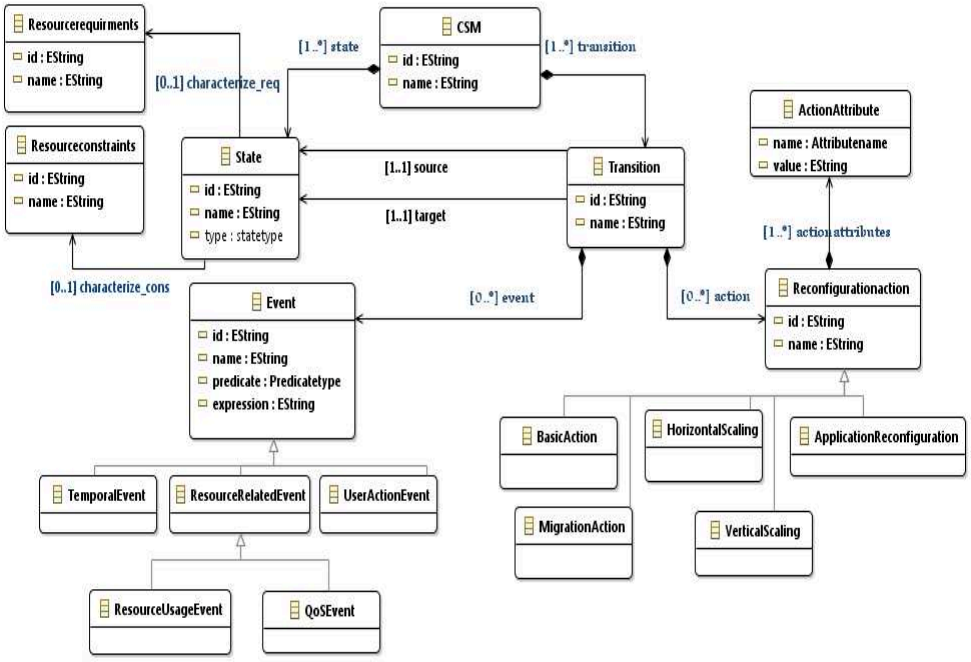


Figure 6.3: C-SM: **C**loud resource requirement and constraint **S**tate **M**achines

6.3.1.1 States

We use states to characterize application specific resource requirements and constraints, therefore we associate it with the Resource requirements and Resource constraints abstraction as shown in Fig.6.3. As well, each state has an id, a *name* and a *type* indicates whether the state is initial, intermediate or final.

Example. As illustrated in the figure 6.4, CSM instance consists of four states S1, S2, S3 and S4 that VM1,VM2 in VM-group may go through during the application life cycle where S1 is the initial state and S4 represents the final state. Additionally, each state is annotated with resource requirements that should be satisfied under that

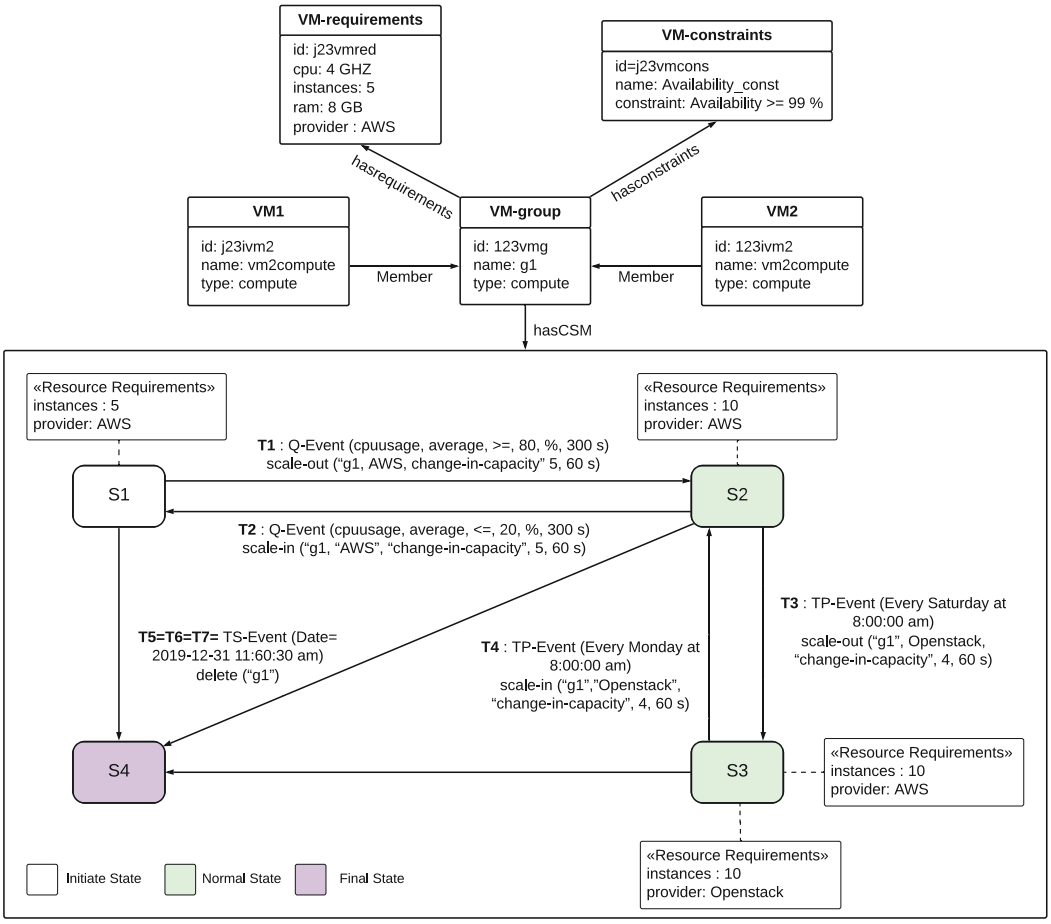


Figure 6.4: cEDM instance provided by the cloud user for the motivating scenario

state. For example, in case of VM1, the state S1 can only be reached if 5 instances of VM1 from AWS have been started.

6.3.1.2 Transitions

We use transitions to express the possible elasticity policies (also called re-configuration policies) that may be occurred during the whole life-cycle of a cloud resource. In particular, these transitions will be automatically triggered when certain events are detected. This will ultimately triggers actions that perform the desired resource re-configurations to satisfy the requirements and constraints of target states. Conceptually, as shown in Fig. 6.2, each transition is specified by *id* that indicates the identifier to refer this transition, a logic name, *source* that indicates the source state and *target* that indicates the target state. It consists of zero or one event and zero or

one re-configuration action that must be triggered once the related event occurs.

6.3.1.3 Events.

As in software engineering, Event in the context of cloud resources represents the occurrence of any change that results in triggering of certain actions on a cloud resource to adapt to that change. Typically, events can be triggered by the user, the cloud resource itself in a consequence of the satisfaction of some conditions overs its resource-related metrics, or out of the cloud resource context like time, environment, market, and so forth. Accordingly, we distinguish three event types to trigger a reconfiguration action: Temporal Events, Resource Related Events, and User Action Events (refer to Fig. 6.2). Each event is specified through a logic name, a reference id, a predefined predicate type and an expression that encompasses the body of the event according to the predefined predicate type using EBNF common grammars.
Temporal Events. Actions may require that certain temporal events must be occurred to be executed. We identify two patterns that these events can take: specified date and periodicity patterns as they are the most used in practice [26,78]. The specified date pattern specifies that an action needs to be executed at a specified date. We define it through a predicate called TS-Event(c) with c defined as follows:

```
<c>::= 'Date=' <D> | 'Date=' '[' <D> ',' { <D> ',' } ']'
<D>:: <yy>'-'<mm>'-'<dd> ' '  <hh>'-'<mm>'-'<ss> <format>
<format>:: 'am'| 'pm'
```

with Date is a clock, and D is an absolute date which can be expressed as yy-mm-dd hh-mm-ss am/pm format. For example, as shown in Fig.6.4, TS-Event (Date=2019-12-31 11:00:00 am) within the transition T5, T6 and T7 represents a temporal event expressed using this pattern, which will be triggered on 2019-12-31 at 11:00:00 am.
Whereas periodicity pattern specifies that a certain action needs to be executed following a certain periodicity rule over time, which is defined using TP-Event (p) predicate with p is defined as follows:

```
<D>::= 'Every' <weakday> '-' { <weakday> '-' } 'at' <Time>
|  'Everyday' 'at'  <Time> 'Except'  ' <weakday> '-' { <weakday> '-'}
<weakday>::= Mo | Tu | Wed | Th | Fr | Sa | Su
<Time>::= <Hour>':'<Minute>':'<Second>'am'|'pm'
<Hour>::= 01 | 02 | 03 | 04 |.... | 12
<Minute>::= 00 | 01 | 02 | 03 |.... | 60
<Second>::= 00 | 01 | 02 | 03 |.... | 60
```

For instance, as shown in Fig.6.4, TP-Event (Every Saturday at 8:00:00 am) within T3 represents a temporal event, expressed using the above notations, which shall be triggered every Saturday at 08:00:00 am.

Resource Related Events. An action needs to be executed once certain resource metric meets a predefined reference value (i.e., threshold) . That type of events includes two sub-types of event that we call *Resource Usage Events* and *QoS Events*. The former is defined using metrics related to the usage percentage of a resource such CPU_ usage, RAM_usage, etc. Whereas the latter is expressed through QoS (i.e., Quality of Service) metrics such availability, response time, throughput, etc. Both event sub-types are defined using the predicate type Q-event(m, k, fc, op, rv, un, w, tm, tg), where:

- m is the metric; k precises the metric kind whether measurable or abstract;
- fc precises the metric evaluation way (e.g. none, average, maximum, minimum, etc...);
- $op \in \{==, \neq, <, >, <=, >=\}$; rv is the reference value; un is the used unit;
- w defines a time window during which the metric could be evaluated;
- tm defines the number of consecutive times that an event has to occur to trigger the action;
- tr represents which target resource this metric is related to, which by default represents the resource associated with the state machine and it is assigned automatically by the system.

Example. In our motivation example, as shown in Fig.6.4, the cloud user defines their resource-related event in terms of CPU_ usage, such as *Q-Event (CPUusage, average, >=, 80%, 120s, 3, g1)* within T1, which checks whether the CPU_ usage average is greater than or equal 80% during a window of 2 minutes for 3 consecutive times across all VM1 (in case of VM1) or VM2 (in case of VM2) instances.

User Action Events. Actions are executed at the behest of a cloud user. For instance, a cloud user can demand to set manually the capacity of his/her VM instances. These events are defined through a predicate called U-Event (u) over a set of messages M, with u defined as follows:

u::= message=e (T_r)

with message is an incoming action message from a user and e $\in$ M and the T_r is the name of the target resource. For example, U-Event (message=Stop (VM1)) will be triggered when we receive from user a stop message related to the VM1 resource. Other messages include delete, restart and start. Advanced messages related to migrate, scale, and update actions will be considered in the future.

6.3.1.4 Reconfiguration actions.

Reconfiguration actions specify how a cloud resource should behave when certain events occur. In our model, we have considered five reconfiguration action categories: horizontal scaling, vertical scaling, migration, application reconfiguration and basic actions. Each reconfiguration action is defined by a name and a set of action attributes that define required inputs to execute this action. Both name and action attributes of each action are defined depending the category of action (refer to Fig. 6.2). For the sake of clarity, in the following, we choose JSON schema to show the required attributes for each action.

Horizontal Scaling (HS): As shown in listing 6.1, an horizontal scaling action has as name Scale-in or Scale-out and it should contain the following action attributes: resource-target, adjustment-type, adjust, cooldown. The resource-Target represents the name of the resource that will be adjusted. The adjustment-type specifies the way of change which can be change in capacity (i.e. Add/Remove the given number of resource instances), exact-capacity (Set the current number of resource instances to the given number) or percent-change-in-capacity (Add/Remove the given percentage to the number of resource instances). The adjust specifies a value, its meaning depends on adjustment-type and finally the cooldown indicates the time period during which no other reconfiguration actions for the same resource will be taken.

```
{ "HorizontalScalingAction" : {"id" :{"type": "string"},
    "name " :{"type": "enum['scale-in', 'scale-out']"},
  "actionattributes":{
         "resource-target":{"type":"string"},
         "adjustment-type":{"type":"enum['exact-capacity', 'change-in-capacity
    ',  'percent-change-in-capacity']"},
         "adjust":{"type": "string"},
        "cooldown":{"type": "number"}
} }}
```

Listing 6.1: Horizontal Scaling specification

Vertical Scaling (VS): As shown in listing 6.2, a vertical scaling action has as name Scale-up or Scale-down while it has the same action attributes of horizontal scaling type. Moreover, we add along with the provided attributes the "attribute-target" to indicate the attribute name (e.g. CPU, RAM) of a resource to be modified.

```
 { "VerticalScalingAction" : {"id" :{"type": "string"},
     {"name": {"type"  : "enum ['scale-up', 'scale-down']"},
    "actionattributes":{
        "resource-target":{"type":"string"},
        "attribute-target":{"type": "string"},
```

```
        "adjustment-type":   {"type":"enum['exact-capacity', 'change-in
    capacity', 'percent-change-incapacity']"},  "adjust":{"type": "string"},
         "cooldown":{"type": "number"}
}}}
```

Listing 6.2: Vertical Scaling specification

Migration: As mentioned above, we distinguish two types of migration: *VM Migration* and *Application migration*. In this category, we find only one action which is a *migrate* action. As shown in listing 6.3, a *migrate* action has as name "*migrate*" and contains a set of attributes: The *Target* represents the name of the VM or the application component that will be migrated. The "*host*" which can be filled by the host name (e.g. the name of a node in case of a VM migration or the name of VM in case of an application component of a cloud user). In some cases, it is filled by "None", so the controller must choose the appropriate host automatically. The migration *type* indicates the type of migration: Cold (requires to turn off a VM or application component before transferring it to another host) or Hot (allows to move a VM or application component without stopping it).

```
 { "MigrationAction" : {"id" :{"type": "string"},
        "name": "migrate",
        "actionattributes":{
        "target":{"type": "string"},
        "host":{"type": "string", "default": "None"},
        "type":{"type": "enum ['Cold', 'Hot']"},
        "cooldown":{"type": "number"}
}}}
```

Listing 6.3: Migration action specification

Application Reconfiguration (AR): In that category, we find only one action which is an update action. As shown in listing 6.4, an update action has as name "update" and contains a set of attributes: the Target represents the name of the concerned application component; the "attribute-target" indicates the attribute name (e.g. cache-size) to be modified and finally the "attribute-value " indicates the new value to be assigned.

```
{ "ApplicationReconfigurationAction" : {"id ":{"type": "string"},
      "name": "update",
      "actionattributes":{
           "resource-target":{"type": "string"},
           "attribute-target":{"type": "string"},
           "attribute-value":{"type": "string"}
       }}}
```

Listing 6.4: Application Reconfiguration specification

Basic Actions: regroup the basic actions applied on a cloud resource including: Start (start the running of a resource), Stop (stop the running of a resource), Restart (restart the running of a resource) and Delete (delete definitely a resource). As shown in listing 6.5 each basic action should contain a name that should be Start, restart, Stop, or Delete and resource-target as attribute that indicates the name of the resource to be manipulated.

```
1  { "BasicAction" : { {"id" :{"type": "string"},
2       "name": {"type" : "enum ['start', 'stop',  'delete', 'restart']"},
     "actionattributes":{
3       "resource-target": {"type": "string"}
4  }}}
```

Listing 6.5: Basic action specification

Example. In our motivation example, as shown in Fig.6.4, the cloud user used both the horizontal scaling and basic actions. For example, *Scale-out ("VM1_ compute", "AWS", "change-in-capacity", 5, 60s)* within T1 represent an instance from horizontal scaling action, which allows to add for *VM1_ compute* resource 5 *instances* from *AWS provider*, where *60s* represent a clowdown period that must be respected after triggering this action. While Delete (VM1_ compute) within T5, T6, T7 represents a basic action that aims at removing definitely the resource VM1_ compute.

6.3.2 Supporting Multi-Providers Abstractions

The previous section has outlined the key aspects to be included when specifying elasticity features for cloud resources from a single provider. However, our proposed model is also able to support several providers. To do so, we need to consider that a cloud resource can be acquired from several providers during a cloud user'service life-cycle. As we have defined the resource requirements in a more general way, they are enough to support multi-cloud scenarios. For example, regarding the requirement attribute, we only need to instantiate an attribute "Provider" to indicate the provider that offers a given resource. Moreover, we need to identify all events or even required that lead a cloud user to acquire cloud resources from a new provider or to definitely migrate its services to be deployed on other cloud resources. As well, re-configurations actions need to be extended in order to support these new situations. Indeed, in practice, once a service has been deployed in a cloud resource from a specific provider, different situation can occur at runtime, which are mainly:
(1) Service can be scaled manually (i.e., User Action Events) or dynamically (i.e. Resource Related or Temporal Events) by adding or removing cloud resource from a new provider,
(2) Service can be migrated to another provider at the behest of its cloud user (i.e., User Action Events),

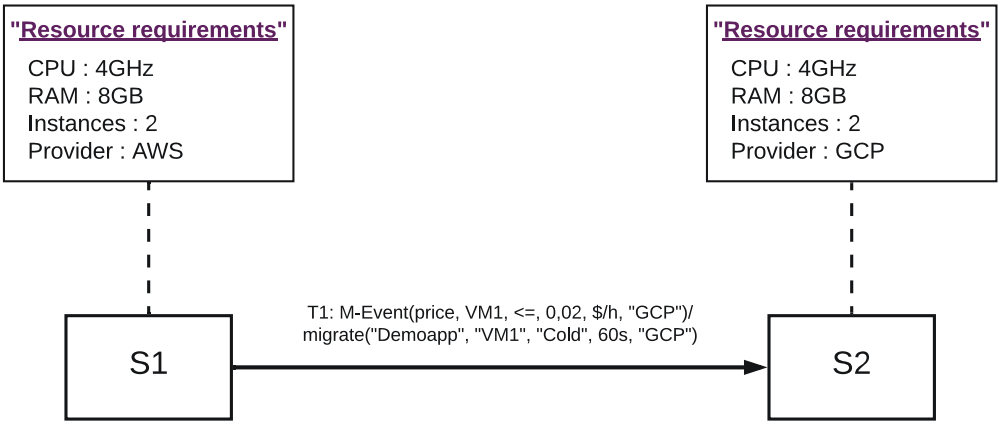


Figure 6.5: Migration of Demo app that is deployed on the VM1 to other VM from GCP as it provides the same resource with a better price

(3) Service can be migrated in case that the provider does not respect the QoS constraints (i.e. Resource Related Events),
(4) finally, service can be migrated to another provider when this new provider offers better utility to a cloud user than the previous provider.
Consequently, our basic resource model should be simply extended to support these new considerations. To support (1), (2) and (3) we need only to extend the reconfiguration actions (i.e. horizontal and vertical scaling, migration, etc.) by adding the property "provider" as an attribute to these actions. For instance, the *Scale-out ("VM1_ compute", "Openstack", "change-in-capacity", 10, "60s")* within T3 shows a horizontal scaling from a new provider than AWS, which is OpenStak. Furthermore, to support (4), a new type of events should be defined along with the above defined events, that we call Market related events. Market related events depend on QoS and resource properties too, we define it as one of the types of Resource Related Events.

Market Related Events. Events occur as a result of changes within cloud market. Such events can be triggered whenever there is a cloud resource offer providing QoS or any other resource property (e.g., price) better than the one is currently using by the user. They are expressed through the predicate *M-Event(m, r, op, tr, u, p)*, with q is the QoS or resource property, r is the concerned resource, and *op*, *tr* and u have the same definition within Q-event. While p indicates the new provider which can be filled by the provider name such "GCP" or by "any", so, the runtime controller must automatically choose the appropriate provider that satisfies the user needs. For example, Fig.5 illustrates a market related event that consists to trigger the migration of the Demo app that was hosted on the AWS VM1 in other VM from GCP provider

as it offers the same VM with a price less than or equal 0.02 $/h.

6.4 Monitoring and execution of Elasticity

In this section, we present cEDMCore system [33] which represents the execution environment of our cEDM model. More specifically, it supports the modeling, monitoring, and execution of elasticity features that were defined using the above presented abstractions. Figure 6.6 provides a high-level overview of the underlying parts that our system architecture relies on. Firstly, using cEDMcore designer, the cloud user can graphically instantiate from cEDM model the cEDM instance that corresponds to their elasticity requirements. This instance is then simply serialized as JSON/XMI file, which represents a list of required cloud resources, each one is associated with the state machine C-SM that describes the related elasticity behavior.

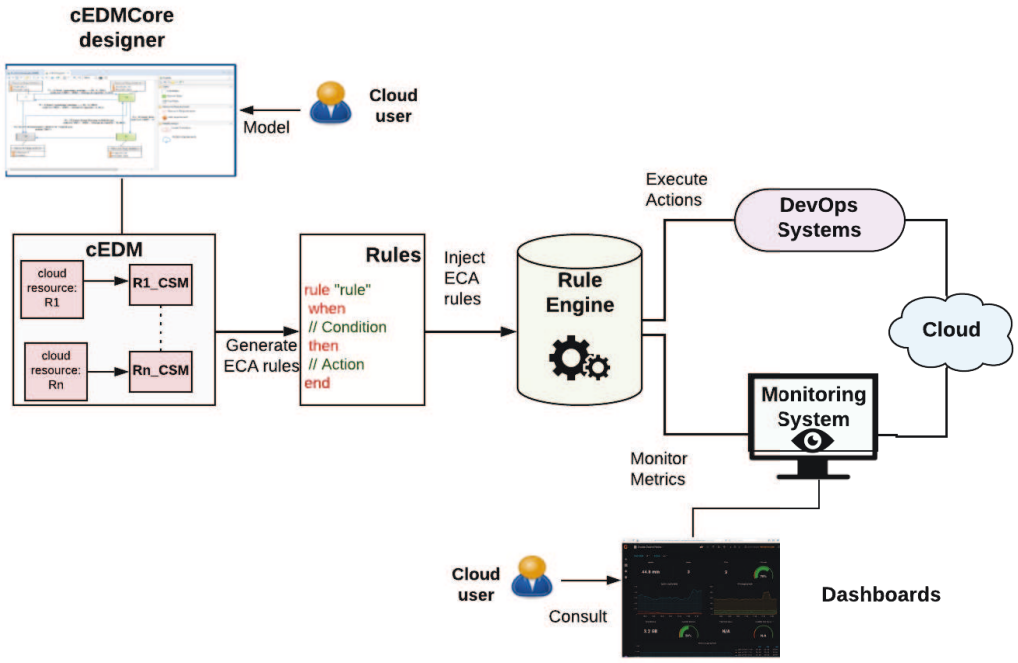


Figure 6.6: Architecture Overview

After that, our runtime system exploits each defined C-SM to generate the ECA execution rules using a rules generator component, whose underlying logic is explained later in Section 6.4.2. These rules would be then exploited by the rule engine in order to support the enforcement and runtime control of elasticity. To choose the suitable rule engine, we performed a comparative analysis between Drools [136] and Espers [62] as they represent the most known and used rule engines. According to this analysis,

we opted for the Drools rule engine because it provides a good compromise between expressivity and performance, has good documentation and all its components are open-source.
Furthermore, our runtime system deploys a monitoring service on each cloud resource defined in the cEDM with the aim of collecting the related operational data that are needed later to execute ECA rules. The management of all involved monitoring services and the persistence of collected data are handled by a monitoring system explained in Section 6.4.3.
Finally, at the behest of the Drools engine, the concrete enforcement of elasticity actions defined in ECA rules, is made by the DevOps systems. Three DevOps systems are integrated into our system, namely Docker, Kubernetes, and Terraform. The communication between Drools engine, Monitoring system and DevOps systems is ensured by a set of connectors.
In the following, we detail the main components of our cEDMCore system, namely *cEDMcore designer*, *ECA rule generator*, and the *Monitoring system*.

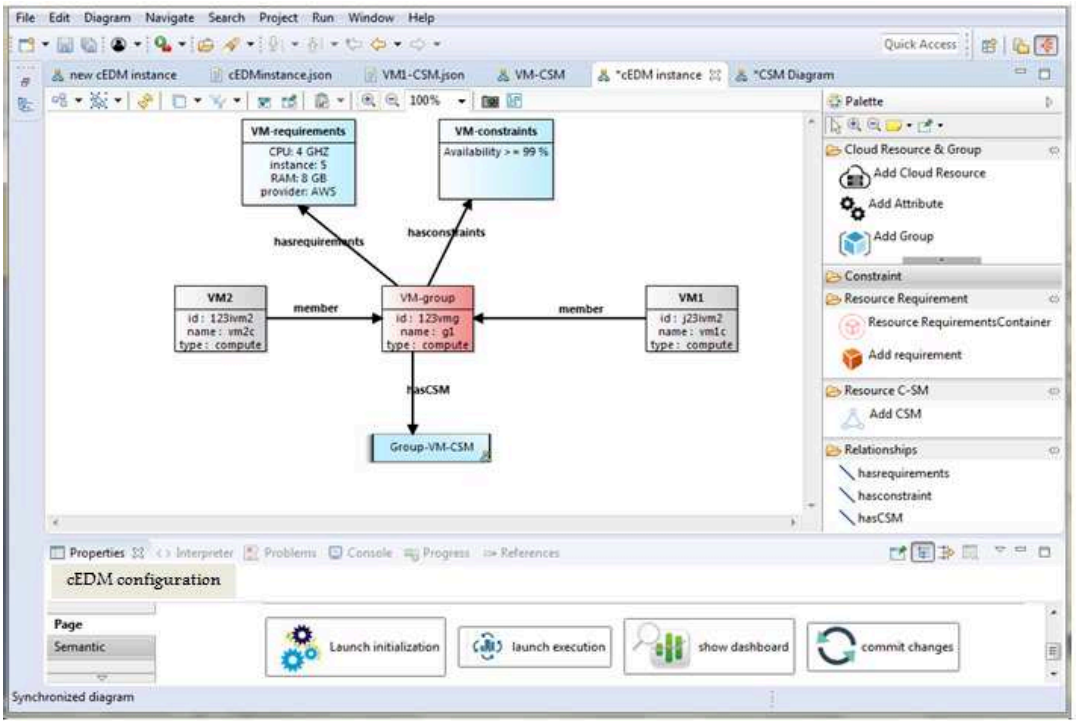


Figure 6.7: The visual features of the cEDM editor for specifying the concerned cloud resources, the desired resource requirements and constraints

6.4.1 cEDM design tool

cEDMcore designer enables cloud users to describe the elasticity behavior related to their cloud resources. It is composed of two editors: cEDM editor and CSM editor. *cEDM editor* (refer to Figure 6.7) enables a cloud user to instantiate from cEDM model his/her corresponding cEDM instance. It also allows cloud users to attach each cloud resource or group of multiple cloud resources to one or more CSM machines for defining the desired elasticity behavior. Here, cloud users have also to supply the required access credentials of cloud providers of their choices. Whereas, CSM editor (refer to Figure 6.8) is devoted to design the involved CSM machines. Both editors are provided with (i) textual (JSON) and visual notations to allow editing models, both graphically and textually, and (ii) model views to represent the model hierarchical tree. In addition, these editors have been provided with automatic completion and quick fixes functionalities enabling cloud users to evaluate their models and to fix design errors when occurred. A key part of our editors implementation involves integrating MDE frameworks such EMF [54], xtext [56] and Sirius [55]. These MDE frameworks are used to facilitate the design of cEDM model and the code generation of its manipulation plugins.

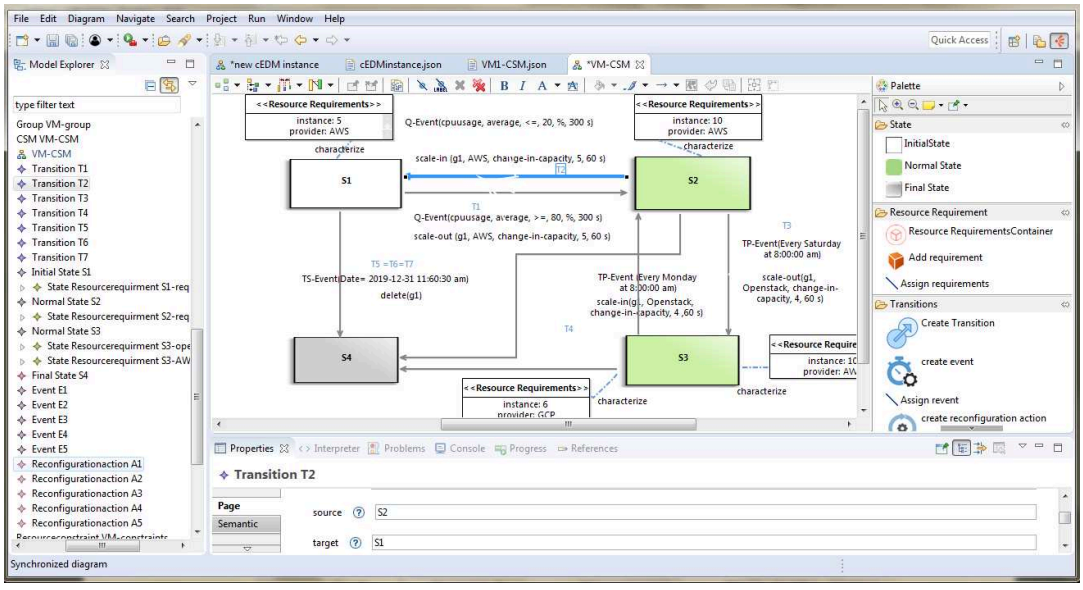


Figure 6.8: The visual features of the CSM editor that describes cloud resource elasticity through a state machine

6.4.2 ECA rules Generator

In this section, we present our ECA rule generator that serves to generate ECA rules from a given C-SM. For the sake of clarity, we use the JSON notation when exemplifying the C-SM machine elements (i.e. State, transition, action, event).
Indeed, under the Drools engine, the ECA rules are defined in a text file called *DRL* with .drl extension. DRL stands for Drools Rule Language. Listing 6.6 gives an overview of the rule syntax in the DRL file.

```
1
2 rule "name"
3     attributes
4     when
5         LHS: <conditional element>*
6     then
7         RHS: <action>*
8 end
```

Listing 6.6: Drools' ECA rule syntax

Accordingly, each rule must have a unique name, and is composed of two parts: LHS and RHS. The Left Hand Side (LHS) denotes the conditional part of the rule, which includes zero or more *conditional elements*. The LHS always follows the *when* keyword. The Right Hand Side (RHS) indicates the consequence of the rule, which may contain a list of actions to be executed. An action defines a specific code to be executed whenever the LHS part is satisfied. The RHS follows the *then* keyword. Furthermore, a rule may contain a set of *Attributes* and *Timers* defining how the rule should behave. For instance, a timer associated within a rule indicates when this rule can fire. It is a property defined by Drools engine to reason about the time dimension. In our work, we will use the timer functionality to specify any temporal event. In addition to the rules, any DRL file may include some global variables. Generally, these variables represent the means (i) to provide data, services or APIs that rules need to use and (ii) return data, logs or values added as a consequence of the rules execution.
Back to our ECA rule generator shown in Figure 6.9, it relies on M2T transformation that takes as input a C-SM model (e.g. CSM.csm) which is also represented as JSON file and generates the corresponding DRL file. The M2T transformation serves to create for each transition description a DRL-compliant rule while taking its source and target state. Our generator follows a straightforward procedure, whose mains steps are depicted at the bottom part of Figure 6.9 while using a concrete example of a Transition that we pick up from the C-SM related to our motivating example. Accordingly, the first step (1) consists of generating the *rule name* from the given *transition name* concatenated with the *transition id*. Secondly, (2) the *transition's source state* will be used to generate a *conditional element* in the LHS part of the rule that aims to verify whether the concerned current state (represented by a global variable named *"currentState"*) has already reached or no. Moreover, (3) the *transition's*

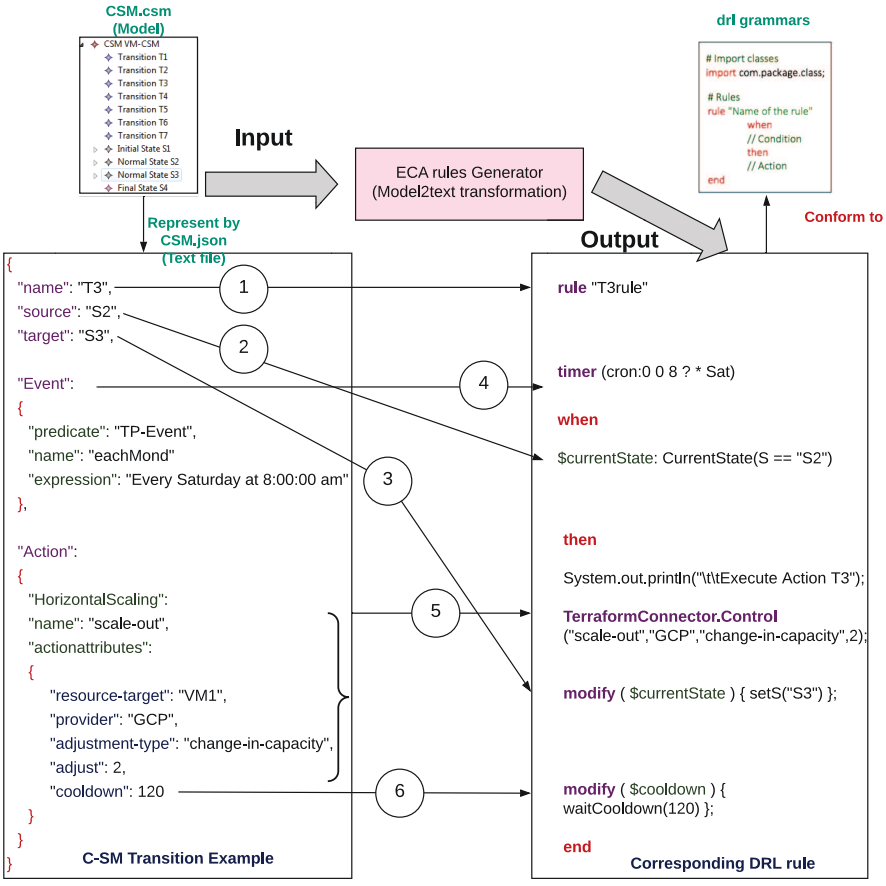


Figure 6.9: Automated generation of native execution DRL rules from C-SM

target state will be used to generate the *modify action* in the RHS part of the rule, which allows updating the current state of the C-SM as a consequence of executing the desired reconfiguration action. As a result of this modification, the runtime of C-SM moves from the source state *"S2"* to the new state *"S3"*. This feature helps the cloud users to track the behavior of their C-SM machines at runtime.
Furthermore, the fourth step (4) concerns the generation of DRL expression that corresponds to the transition event. Here, it should be noted that this generation depends on the event type. As mentioned above, we supported three event types to trigger a reconfiguration action, mainly temporal events including specified date/periodicity rule-based events, a resource-related event including QoS event and User-action events. Currently, we propose generation support for the two temporal events (i.e., TS-Event, TP-Event) and the resource-related event (i.e., Q-Event). After exploring

the DRL specification for events, we reveal that each resource-related event has to be mapped into a conditional element in the LHS part of the rule which consists of verifying the resource metric value captured from the monitoring system with the predefined threshold. Whereas, each temporal event has to be mapped into a timer Cron expression in the *attributes part* of the rule. The Cron expression is a string of seven fields separated by a white space, which is commonly used to specify the schedule details in a machine-readable way. The syntax of a Cron expression is depicted in Figure 6.10.

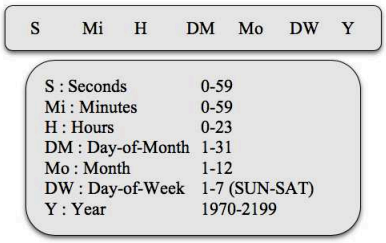


Figure 6.10: *Syntax of Cron expression*

According to the Cron syntax, Figure 6.11 shows an example of transforming the specific date event, which is based on the TS-Event predicate, into the corresponding Cron expression ready to be triggered by the runtime system once it is reached.

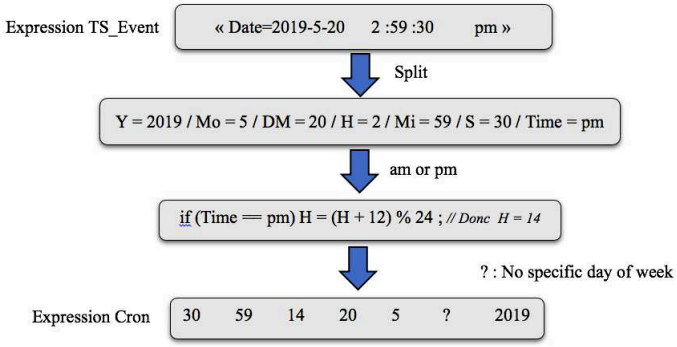


Figure 6.11: *Transforming TS-Event expression into a Cron expression*

In a similar way, we transform the following TP-Event expression "Every Saturday at 8:00:00 am" into the following Cron expression "timer (cron: 0 0 8 * * 7)", where 7 means Saturday and the * symbol means any value (i.e., any day of the month and any month). Finally, the last step (5) allows the generation of DRL expression that corresponds to the transition action. More specifically, it maps the high-level reconfig-

uration action into its corresponding low-level action that has to be executed whether the LHS part of the rule is satisfied. In our solution, all low-level actions are executed by means of suitable connectors through their *Control* operations. For instance, in the given example, *TerraformConnector.Control* operation takes the imputed data related to the horizontal scaling action of VM1 and initiates the corresponding API calls and low-level scripts specific to the Terraform tool in order to increase the VM1 instance number by adding six instances from GCP provider.

6.4.3 Monitoring system

The monitoring system enables the monitoring of cloud resources and the collecting of the data that are required for making decisions about the execution of elasticity actions. Currently, we support the monitoring of containers, virtual machines, and servers. Our monitoring system has to be able to gather any data despite the variations of sources and technologies due to the use of multiple and heterogeneous cloud providers, DevOps enforcement systems and virtualization mechanisms. To satisfy this critical requirement, we performed an exploration and comparative analysis of the existing monitoring tools and plugins. The examined solutions include AWS CloudWatch [19], Docker Stats [49], Nagios [24] , Prometheus [3] and Grafana [73]. Briefly, any solution that dependent on cloud providers such as AWS CloudWatch or DevOps systems such as Docker Stats has been excluded because it contradicts the technology-independence requirement that our monitoring solution should satisfy. In addition, traditional monitoring systems such as Nagios do not fit our needs as they do not take into consideration the monitoring of containers.
As a result, we propose to use the Prometheus system. Prometheus allows Data analysis, storage, and visualization. It supports the monitoring of containers and servers by integrating the corresponding plugins. It also allows the capture of several types of metrics and facilitates their extraction and manipulation via the query language (PromQL). Despite these obvious advantages, Prometheus has some disadvantages that can be summarized in the following two points: it does not offer ergonomic visualization interfaces and does not visualize data in real-time. To remedy these shortcomings, we propose to integrate the Grafana dash-boarding tool with Prometheus. Indeed, Grafana has very advanced presentation capabilities via customized dashboards. It also allows for real-time tracking of metrics.
With all this in mind, we propose an integrated monitoring system that takes advantage of diverse and powerful open-source systems for effectively collecting, analyzing and visualization of data that can be obtained from diverse resources and heterogeneous clouds. Figure 6.12 provides a high-level overview of the proposed system. Our system provides three tasks, namely *Retrieval of monitoring Data*, *Storage of monitoring Data*, and *Visualization of monitoring Data*. To support them, it relies on the following components:

- *Rest Connection API:* is used to connect our system to the authentication server.

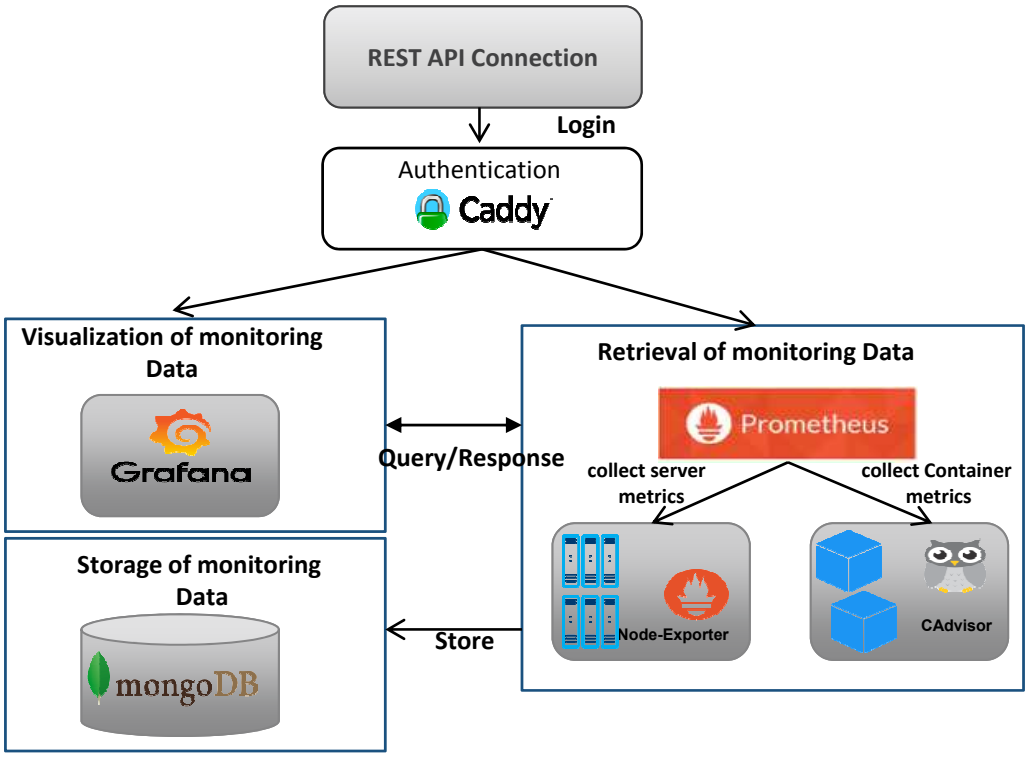


Figure 6.12: Architecture Overview

- *Caddy web server*: represents an authentication server that is necessary to control the access to monitoring data and launching queries to retrieve metrics data.

- *CAdavisor*: is a real-time monitoring tool for containers. It allows for exposing a set of container metrics.

- *Node-Exporter:* is a plugin that allows the monitoring of virtual machines and servers in a given cloud host.

- *MongoDB:* is a database used to save the monitoring data retrieved from Prometheus.

- *Grafana:* is used to create supervision dashboards on data saved in the MongoDB.

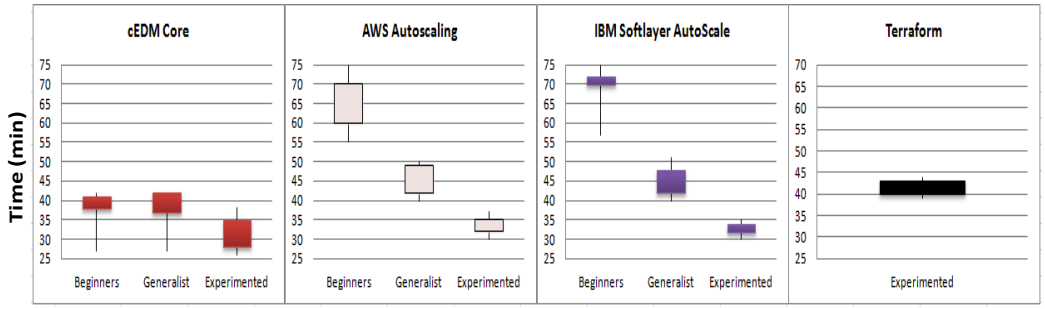
6.5 Evaluation

For evaluation purposes, we conducted two experiments using our cEDMCore system. The first experiment intends to assess the overall productivity of cEDMCore in comparison to the existing solutions. Whereas, in the second experiment, we aim at assessing the overhead that is introduced by our approach in comparison to manual and provider-dependent configuration regarding the multi-cloud deployment.

6.5.1 Experimentation 1

6.5.1.1 Experimental setup

We evaluated the productivity of our cEDM by conducting a deeper user-study with 32 participants: 25 Data Scale master students from the university of Paris-Saclay and 7 professional users (1 PHD student working on cloud orchestration, 1 master student in cloud computing, 5 DevOps engineers). Indeed, our ultimate goal from this evaluation is to essentially check whether the proposed modeling abstractions are enough intuitive and do not require an extensive programming effort from the user side for being used. Therefore, we intentionally selected our participants from 3 groups having diverse levels of technical expertise: (1) Beginners (12 participants out of 25 master students): who do not have any knowledge of cloud solutions ; (2) Generalists: who have a little knowledge of cloud solutions (13 participants out of 25 master students) and (3) Experts (7 participants from the professional users): who have a sophisticated understanding of cloud solutions. Such diversity in participants is quite relevant for our analysis as it aids us to have a global view about the productivity of our abstractions from different user categories.
Moreover, the productivity is tested in terms of the efficiency and the usefulness of cEDM in describing cloud resources elasticity behaviour. The *efficiency* is measured in terms of the time taken to complete the modeling and configuration tasks. The *usefulness* is determined via a questionnaire that asses the participants feed-backs about our cEDM. The questionnaire devised into four main parts: Background, Functionality, Usability, Insights/Improvements. We provide the complete list of questions in Appendix B. The background questions aim to confirm whether the participants are familiar or no with existing cloud resource elasticity tools. The functionality questions verify whether the participants correctly understand the main functionalities of cEDM. The usability questions sought to discover whether the key modeling abstraction offered are easy and intuitive.
We asked all participants to model all the elalsticity requirements described in our motivation scenario using our cEDM Core while considring only AWS provider. For quantitative comparison purpose, we asked them to do the same scenario with two provider-specific solutions viz. the IBM Softlayer Auto Scale [78] and AWS Auto Scaling [26] and by using Terraform [149]. A total of 32 out of 32 participants participated in AWS Auto Scaling and IBM Softlayer Auto Scale based experiments respectively.



(e)

	cEDM Core	AWS Autoscaling	IBM Softlayer AutoScale	Terraform
Average (min)	37,7	51,6	51	41,42
Observations	32: 12(B)+13(G)+7(E)	32: 12(B)+13(G)+7(E)	32: 12(B)+13(G)+7(E)	7: 0(B)+0(G)+7(E)
Variance	32	175,7	219	4.2

Figure 6.13: Time to complete the task; (a) (b) (c) (d) Time grouped by level of expertise; (e) Average time for all the participants.

However, only a total of 7 out of 32 participants participated in Terraform based experiment. This is due to the fact that only expert participants have expertise and confidence in using this tool.

6.5.1.2 Evaluation Results

Results of the experiment in Fig. 6.13 (a) (b) (c) (d) show the time taken using cEDMcore, AWS Auto Scaling, IBM Softlayer Auto Scale, and Terraform, for completing the task. As shown, it is surprising that there is no big difference in times between the different user categories, i.e. Beginners, Generalists, Experts when using the cEDM Core. In contrast, we notice a stark difference in times between these categories when using the other tools, i.e. AWS Auto Scaling, IBM Softlayer Auto Scale , and Terraform. This clearly shows that our cEDM Core does not require extensive development skills and high technical expertise in specific cloud tools and programming compared to the other tool that insists on particular technical expertise. Overall, the time taken to complete the task was reduced in comparison to other solutions. On the whole, as shown in Fig.6.13 (e), the participants took on average 37,7 min using our cEDMcore, 51 min using IBM Softlayer Auto Scale, 51,6 min using AWS Auto Scaling and 41,42 min using Terraform. This demonstrates the efficiency of our cEDMcore system. In fact, we argue that technology-independent elasticity abstractions and declarative model like state machine to describe the elasticity behaviour significantly reduce the time-to-modeling.
Moreover, to evaluate the cEDM usability, we use the Usability section of the questionnaire by asking participants to rate the usability for each abstraction (scale 0-5).

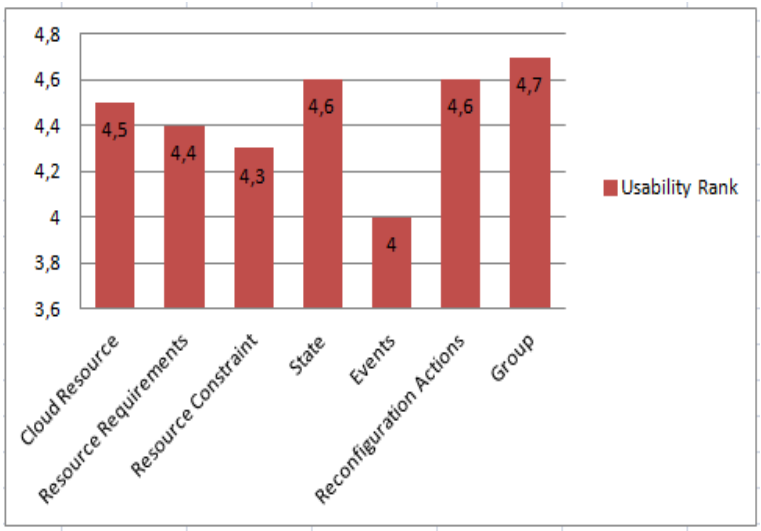


Figure 6.14: Usability Rate of the main cEDM abstractions

We examined the basic cEDM abstractions: Cloud Resource, Group, Resource Requirement, Resource Constraint, State, Event, Reconfiguration action. We observed that the mean score for all abstractions in Figure. 6.14 is greater than the neutral value of 3 with a noticeable difference. Overall participants reported that our model is a familiar and intuitive, especially C-SM is not far from natural language and allows defining cloud resources elasticity behaviour in a very simple and easy way. Accordingly, giving these observations, we confirm that the key modeling abstractions offered are useful and comprehensible. Moreover, as feedbacks for improvement, some of the participants highlighted the need for mechanisms that help users to choose the best thresholds when defining elasticity policies based on resource-related events. Some others suggested adding a cost computing method that allows users to have an idea about the expected cost of their policies. We will take all those points and add the requested features in the future.

6.5.2 Experimentation 2

We evaluated the overhead introduced by our approach regarding the execution of elasticity related to containers and virtual machines acquired from two cloud providers: AWS and Google compute platform (GCP). More specifically, two reconfigurations policies were considered. The first consists of scaling Node-bookshop service by adding 4 containers whenever the cpu_usage exceeds 80% to two containers already exist. The second one allows scaling the virtual machine VM1 that was already deployed on AWS, by adding 4 instances from GCP whenever the cpu_usage exceeds 80%.
Moreover, to determine the overhead introduced by our cEDM Core regarding the

elasticity, we execute these policies according to two cases: (1) using cEDM core by defining the corresponding C-SM machine for each elasticity policy and (2) using native scripts that directly executed on the infrastructure provider in case of VM elasticity and on the Docker platform in case of container elasticity. For the evaluation purpose, we simulated an increase in CPU usage on a VM and a container using the Stress tool. Stress is a workload generator, imposing on the system a configurable amount of CPU, memory, I / O and disk. In addition, for a better estimation of overhead, we repeated the execution of each policy ten times.
As a result of this experiment, we report the variance, average, minimum and maximum times values as well the introduced overhead for executing each elasticity policy. Table 6.2 shows the observed values regarding VM horizontal scaling. Accordingly, the introduced overhead by cEDM Core is 1.01%, which is too low value. Finally, Table 7 presents the time values as well as the introduced overhead for the horizontal scaling related to container resources. As demonstrated, the obtained overhead is 1.1 %.

Table 6.1: Horizontal elasticity of virtual machines

solution	variance	minimum (sec)	maximum (sec)	average (sec)	Overhead
Manual configuration	92,9	253	287	266,5	-
cEDM Core	91,8	256	288	270,4	1.01%

Table 6.2: Horizontal elasticity of containers

solution	variance	minimum (sec)	maximum (sec)	average (sec)	Overhead
Manual configuration	2,04	29	33	30,4	-
cEDM Core	8,8	31	38	33,8	1.1 %

Based on the above observations, we confirm that the cEDM Core does not provide a significant overhead, which yields good performance. The introduced overhead is negligible compared to the benefits provided by our solution. This is thanks to the use of the Drools engine which is known by its speed and scalability. Indeed, the Drools engine relies on an enhanced implementation of the Rete algorithm which has been proved that it performs efficiently, accurately and quickly.

6.6 Conclusions

The work presented in this chapter meets our third thesis objective, namely: supporting high-level management of multi-cloud elasticity. In doing so, we proposed a novel Cloud Elasticity Description Model (cEDM) based on a state machine (C-SM), that provides high-level abstractions to describe elasticity features related to cloud

resources. Instead of directly manipulating low-level interfaces and scripting elasticity rules over complex cloud services/APIs, C-SM reason about resource requirement states, a recurring and intuitive abstraction in modern IT resources management processes.
Our cEDM model has been implemented and executed using a software system called cEDMcore. It relies on the Drools engine and an integrated monitoring system, coordinating together to govern the elasticity behavior encapsulated in the different state machines. Our system enables cloud users to intuitively describe their elasticity policies and to support their execution without referring to any resource provider or DevOps enforcement mechanism.
For assessment purposes, we (i) demonstrated the gained productivity of our model using a user-case study with academics and professionals and (ii) evaluated the introduced overhead of our system. Through the productivity evaluation, we revealed that the adoption of state machine formalism greatly hides the actual complex implementation within cloud DevOps and provider solutions. Moreover, the overhead-related experiment demonstrates that our approach supports the execution of elasticity policies without introducing significant overhead.

Chapter 7

Conclusion and Future Work

In this chapter, we first summarize our contributions in this thesis to support the interoperable management and orchestration of cloud resources in multi-cloud environments while focusing on their elasticity nature. Next, we discuss our future research directions.

7.1 Fulfillment of Objectives

With the rising adoption of cloud resources in everyday computing tasks, the demand for effective orchestration and management techniques that promote interoperability has considerably increased. In addition, elasticity, the key distinguishing feature in cloud computing, has become primordial to preserve the desired quality of service while optimizing the involved costs. In this thesis, we identified three main objectives in order to accommodate these needs. We aimed specifically to (1) provide guidance and assistance in the design of interoperable management APIs; to (2) streamline and improve the orchestration of cloud resources and to (3) support high-level management of multi-cloud elasticity. Consequently, we proposed three major contributions in order to cover the different objectives.
In the first contribution, we tackled the first objective by providing a compliance evaluation of OCCI and REST best principles and recommendation support to comply with these principles. First, we leveraged patterns and anti-patterns to derive respectively the good and poor practices of OCCI and REST best principles. Then, we proposed a semantic-based approach for defining and detecting REST and OCCI (anti) patterns and providing a set of correction recommendations to comply with both REST and OCCI best principles. We validated this approach by applying it on cloud REST APIs and evaluating its accuracy and usefulness. We found that our approach accurately detects OCCI and REST(anti)patterns and provides useful recommendations. According to the compliance results, we reveal that there is no widespread adoption of OCCI principles in existing APIs. In contrast, these APIs have reached an acceptable level of maturity regarding REST principles. Cloud API developers can benefit from our approach and defined principles to accurately evaluate their APIs from OCCI and REST perspectives. This can contribute to the design of interoperable, understandable, and reusable Cloud management APIs.

The second contribution aims at streamlining and improving the orchestration process of cloud resources. To reach this goal, we provided a model-driven approach that integrates TOSCA with the open-source DevOps solutions in a seamless way and at an acceptable cost. TOSCA is adopted to provide a convenient and technology-independent description of cloud applications, whereas DevOps solutions are leveraged to ensure the runtime of these applications by executing the required orchestration operations.
Our model-driven approach provides two mechanisms to close the existing gap between TOSCA and DevOps solutions. The first mechanism is a model-driven translation technique that serves to transform TOSCA applications into native DevOps-specific artifacts ready to be executed by the underlying DevOps tools/APIs. Our translation technique follows the key MDE principles. We used models to represent DevOps specifications and languages at a high level of abstraction using meaningful and machine-readable constructs. In addition, we relied on transformation language support to encode transformations between the DevOps models and TOSCA that are required to generate the native DevOps-specific artifacts. Besides, the second mechanism is the DevOps abstraction layer that is based on a set of high-level connectors to automate the end-to-end orchestration tasks while exploiting the generated DevOps artifacts. The proposed layer is especially devoted to simplifying the orchestrator task by avoiding the heavy lifting involved when interacting with the underlying DevOps tools/APIs.
To validate our approach, we developed a proof of concept *ToDev,* an integrated and standards-driven orchestration framework. *TODev* includes three open-source DevOps solutions, namely Docker, Terraform, and Kubernetes, and features MDE capabilities. We performed experiments with diverse cloud use cases considering both the single and multi-deployment. Experimental results showed that our approach (i) provides a powerful enhancement to DevOps productivity, (ii) introduces negligible overhead and (iii) has a reasonable transformation performance. Our approach allows DevOps users to benefit from the strengths of both TOSCA and DevOps approaches. The earned benefit is simplifying the orchestration process while maintaining the desired level of interoperability and efficiency.
In the third contribution, we are interested in supporting high-level management of multi-cloud elasticity. In this work, we proposed a novel Cloud Resource Elasticity Description Model based on a state machine, that provides high-level abstractions to describe elasticity features related to cloud resources. These abstractions have been designed with the intention to be more intuitive and user-friendly. The proposed model is dedicated to manage and control the elastic behavior of resources in a multi-cloud environment. In doing so, we relied on a model-driven generation technique that exploits the proposed model to generate appropriate low-level artifacts required for online monitoring, execution and controlling elasticity. These artifacts are structured into ECA rules which are then executed using the Drools engine. Furthermore, we proposed a monitoring system to effectively collect and analyze events related to

resources and services. The monitoring system has been integrated with the Drools engine in order to provide an appropriate runtime environment to manage and control elasticity. All these solutions have been prototypicality implemented into a proof of concept named cEDMCore.
To validate our approach, we relied on two experiments. The first experiment was conducted with academics and professionals to evaluate the productivity of our approach compared to existing solutions like the IBM Softlayer AutoScale [78], AWS Autoscaling [22], and Terraform [149]. The evaluation results showed that our approach does not require extensive development skills and high technical expertise in specific cloud tools and programming compared to the other solutions that insist on particular technical expertise. On the other hand, the second experiment was devoted to evaluate the approach performance by testing its introduced overhead. The obtained results demonstrated that our approach does not introduce a significant overhead, which yields a good performance. The introduced overhead is negligible compared to the benefits provided by our solution.

7.2 Future work

Our work opens several research perspectives to accomplish in short and middle terms. More precisely, we will focus on extending our two approaches presented in Chapters 5 and 6 and considering other related research issues as well. In the following, we start by presenting how to enhance the transformation mechanism related to our model-driven approach (Section 7.2.1). Next, specifically in Sections 7.2.2, 7.2.3, 7.2.4, we explain how our elasticity approach can be extended to cover other elasticity aspects and optimize the configuration of elasticity policies.

7.2.1 High Order Transformations

Our model-driven integration approach presented in Chapter 5 demonstrated that the integration between TOSCA and DevOps solutions is possible and can be automated by (i) generating all DevOps-specific artifacts that are necessary to perform the related orchestration tasks and (ii) providing a set of Connectors that establish the bridge between both solutions. However, any change in the target DevOps solutions or resource providers pushes the need for continuously adapting the provided connectors and transformations. Currently, this adaptation is done manually. Even though our approach was designed with the intention to be easily extended by following MDE principles, the manual adaptation may be worrisome for DevOps curators that are often interested in agile automation. To resolve this, we are currently investigating the potentials of Higher-Order Transformations (HOT) to automatically create (or update) the corresponding transformation rules. HOT is considered as an advanced model-driven technique that aims to automatically generate transformations. A HOT takes as input a correspondence model representing the possible conceptual

mapping between source and target meta-models and transforms it into a transformation model. Then, a transformation engine uses the resulting transformation model to translate the source model into the target model.
Furthermore, we will also investigate how semantic web and machine learning techniques may help to automatically identify the model mappings that are required for HOT. Here, it should be noted that the conceptual mappings that we have already specified within our approach represent the starting point for the definition of the possible correspondence patterns that may exist between different resource models. In addition, they can serve as a reference data-set that can be used later to check the accuracy of the obtained correspondence models.

7.2.2 More expressiveness and automation

We are also working on the extension of the third contribution presented in Chapter 6. Firstly, we are considering more sophisticated elasticity policies, whose definition involves the notion of composite events and composite actions. For instance, a cloud user wants to vertically scale the current resources of CPU, RAM, Storage when there is a remarkable exceeding over CPU, RAM, and Storage usages. Such an elasticity policy may be defined through (1) a composite action which involves the simultaneous execution of three actions, mainly CPU scaling, RAM scaling and Storage scaling; (2) a composite event as well, which is in turn defined through three resource-related events that shall occur at the same time.
Secondly, we want to enhance our C-SM with autonomic features by considering (i) the proactive mode of elasticity and (ii) the obstacles that could impede the successful execution of elasticity actions [29]. In doing so, we are working on augmenting the C-SM abstraction with (i) the concept of *Nested state machine* that may exist in each basic state and (ii) *Means* that help to perform prediction and repairing tasks. The aim is to provide the C-SM machine with a method allowing it to manage itself in case of unexpected conditions that can be related to cloud users or cloud environments. Learning-based approaches represent a suitable area to be investigated for supporting this objective.
Furthermore, we envision to define more event patterns such as service level agreement (SLA) violation Event, whose consequence may often lead to disconnecting from the actual provider and migrating to a new one. We also plan to enhance our model with context information that allows providing personalized editors aligned with a given context (i.e. expertise level, application domain, etc.).

7.2.3 Verification

According to users' feedback, we are confident that our elasticity model (i.e CEDM) provides an intuitive and user-friendly mean to describe their elasticity policies. However, in practice, cloud users are also interested in possible assurances that any new model can provide in order to meet their requirements and QoS expectations. There-

fore, we intend to provide cloud users with verification support that allows checking the correctness of the elasticity policies both at design time and runtime. Currently, we are working on the design-time verification of elasticity. Given that flexible elasticity policies require time-related constraints (e.g., latency and response time, and variable application workload in different time windows), we propose to translate C-SM state-machine into timed-automata. Timed automata are well suited for formal verification as they provide a mathematically well-founded analysis and are endowed with powerful checking techniques.
Using the obtained automata, we specifically interested in verifying whether there is no conflict between the defined elasticity policies in C-SM. This can be ensured by verifying the absence of deadlock in target timed automata. Furthermore, we propose to use the properties of liveness, safety, and reachability in order to verify whether the defined policies will lead to the expected elasticity behavior. We also aim to verify whether the defined C-SM can give any guarantees to reach an expected designer cost. This is required to avoid situations that can lead to unexpected costs.

7.2.4 Optimization and Learning from previous elasticity executions

Acting on user feedback given in the conducted case study, we envision to provide guidance and assistance to the cloud users in the design of their elasticity state machine by recommending optimized configurations. These configurations may be related to elements such as threshold conditions in the transitions, or pricing strategies that allow satisfying some cost constraints, to name a few. In this respect, optimization-based methods could be the best candidate to ensure such recommendations. Furthermore, we intend to help the cloud users to enhance and evolve their state machines by learning from previous elasticity executions. This may also be useful for diagnostic and repair tasks.

Appendices

Appendix A

Evaluation Questionnaire in Chapter 4

This questionnaire aims at evaluating the usefulness of the detection and recommendation support provided by our approach. It focuses on 6 (anti)patterns as follows: 2 REST patterns (Correct use of POST, Tidy URIs), 2 REST anti-patterns (Amorphous URIs, Forgetting Hypermedia), 2 OCCI patterns (Compliant Delete, Compliant URL), and 2 OCCI anti-pattern (Non-Compliant Create, Non-Compliant Trigger Action).
The questionnaire includes 2 sections:

- The section named "Usefulness Questions" should be completed during the experiment.
- The section "Insights/Improvements" should be completed after the experiment.

A.1 Usefulness Questions

The answer format to each usefulness question is as follow:

A.1.1 REST Patterns

Correct use of POST pattern: POST must be used to create a new resource or to execute an action.
Tidy URLs: appears when URIs use lower resource naming and does not contain trailing slashes and underscores.

1. How useful do you think the detection of the "Correct use of POST" pattern?
2. How useful do you think the detection of the "Tidy URLs" pattern?

A.1.2 REST Anti-Patterns

Amorphous URIs anti-pattern: appears when URIs contain symbols, capital letters, underscores, etc., making them hard to read and employ.
Forgetting Hypermedia anti-pattern: appears when links (i.e., hrefs and rels) within the resource representations are absent.

1. How useful do you think the detection of the "Amorphous URIs" anti-pattern and the suggested recommendations to avoid it?
2. How useful do you think the detection of the "Forgetting Hypermedia" anti-pattern and the suggested recommendations to avoid it?

A.1.3 OCCI Patterns

Compliant Delete pattern: HTTP DELETE must be used and only the URI identifying the resource that will be removed must be provided.
Compliant URL pattern: URL must be either a string or as defined in RFC6570.

1. How useful do you think the detection of the "Compliant Delete pattern"?
2. How useful do you think the detection of the "Compliant URL" pattern?

A.1.4 OCC Anti-Patterns

Non-Compliant Create anti-pattern: appears when other HTTP method instead of POST or PUT is used, or the resource definition is not complete (for instance, the Category defining a particular resource Kind is missing) .
Non-Compliant Trigger Action anti-pattern: appears when other HTTP method instead of POST is used, or the action definition is not complete (for instance, the Category defining a particular action is missing) .

1. How useful do you think the detection of the "Non-Compliant Create" anti-pattern and the suggested recommendations to avoid it?
2. How useful do you think the detection of the "Non-Compliant Trigger Action" anti-pattern and the suggested recommendations to avoid it?

A.2 Insights/Improvements

Please provide some suggestions for improving the detection and recommendation support and feel free to provide your criticism about it.

1. What the points you don't like in the detection and the recommendation
2. How would you improve the detection and the recommendation?
3. Do you have any other comments?

Appendix B

Evaluation Questionnaire in Chapter 6

This questionnaire includes 4 sections.

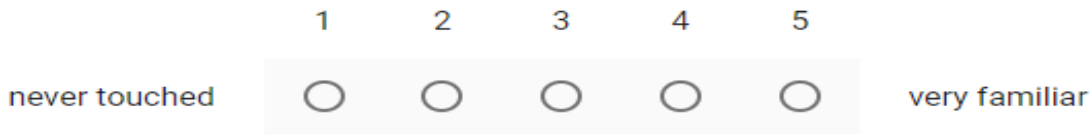
- The section named "Background Questions" should be completed before the experiment.
- The section named "Functionality Questions" should be completed during the experiment.
- The sections named "Usability Questions" and "Insights/Improvements" should be completed after the experiment.

7.1 Background Questions

Please answer all the questions before starting the experiment.

1. How familiar are you with Cloud providers tools?
2. How familiar are you with the AWS Auto Scale/AWS CLI?
3. How familiar are you with the IBM softlayer Auto Scale?
4. How familiar are you with Docker?
5. How familiar are you with Terraform?

The answer format to each question is as follow:



7.2 Functionality Questions

1. How many States you need to describe the provided case study?
2. How many Transitions needed between the states?
3. What are the types of events you need to describe in the case study?
4. How many VM instances needed to handle the application workload during the high peak phase?
5. How many transitions led to the final state?

7.3 Usability Questions

Please answer all the questions. note: Make sure you finish the experiment before answering these questions.

1. How intuitive do you think the Cloud Resource concept?
2. How intuitive do you think the Group concept?
3. How intuitive do you think the Resource Requirements concept?
4. How intuitive do you think the Constraint concept?
5. How intuitive do you think the State concept
6. How intuitive do you think the Events concept (e.g Temporal events, Resource usage events, QoS events, Market related events, User action events?
7. How intuitive do you think the Reconfiguration Action concept?
8. Do you think the cEDMCore system is useful for your day-to-day tasks?

The answer format to each question is as follow:

7.4 Insights/Improvements

1. What the points you don't like in the cEDMCore system?
2. How would you improve the cEDM Core system?
3. Do you have any other comments?

Bibliography

[1] Create cluster using docker swarm. Tech. rep., available at https://medium.com/tech-tajawal/create-cluster-using-docker-swarm-94d7b2a10c43

[2] Generate Anything from any EMF model, available at https://www.eclipse.org/acceleo/

[3] Prometheus - Monitoring system time series database, available at https://prometheus.io/

[4] Xpand, available at https://wiki.eclipse.org/Xpand

[5] xTend, available at http://www.eclipse.org/xtend/

[6] A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152, 125 – 142 (2006), proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)

[7] OWL 2 Web Ontology Language Document Overview (Second Edition) (2012), https://www.w3.org/TR/owl2-overview/

[8] A., R.G., E., B., U., D.: Soa patterns. Manning Publications (2012)

[9] Akdur, D., Garousi, V., Demirörs, O.: A survey on modeling and model-driven engineering practices in the embedded software industry. Journal of Systems Architecture 91, 62 – 82 (2018)

[10] Al-Dhuraibi, Y.: Flexible Framework for Elasticity in Cloud Computing. Theses, Université lille1 (Dec 2018), https://tel.archives-ouvertes.fr/tel-02011337

[11] Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Elasticity in cloud computing: State of the art and research challenges. IEEE Trans. Services Computing 11(2), 430–447 (2018)

[12] Alexander, K., Lee, C., Kim, E., Helal, S.: Enabling end-to-end orchestration of multi-cloud applications. IEEE Access 5, 18862–18875 (2017)

[13] Alnusair, A., Zhao, T.: Towards a model-driven approach for reverse engineering design patterns. In: In Proc. 2nd International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE'09 (2009)

[14] Amazon: Amazon Web Services, available at https://aws.amazon.com/fr/

[15] Ansible: Ansible-in-depth. online article (jun 2018), available at https://www.ansible.com/resources/whitepapers/ansible-in-depth

[16] Apache-Foundation: Apache jclouds: An open source multi-cloud toolki, https://jclouds.apache.org/

[17] Appleton, B., Patterns, I.: Patterns and software: Essential concepts and terminology (1998)

[18] Arnaoudova, V., Penta, M.D., Antoniol, G.: Linguistic antipatterns: what they are and how developers perceive them. Empirical Software Engineering 21(1), 104–158 (2016)

[19] AWS: Amazon CloudWatch, available at https://aws.amazon.com/fr/cloudwatch/

[20] AWS: Amazon EC2 Instance Types, available at https://aws.amazon.com/fr/ec2/instance-types/

[21] AWS: Amazon S3 RESTful API, available at http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html

[22] AWS: Aws Autoscaling, available at https://aws.amazon.com/autoscaling/

[23] AWS: AWS CLI; AWS Cloud Formation, available at https://aws.amazon.com/documentation/

[24] AWS: Nagios: Network, Server and Log Monitoring Software, available at https://www.nagios.com/

[25] AWS: Aws cloud design patterns and practice. Tech. rep. (2014), https://docs.huihoo.com/infoq/qclubshanghai-aws-cloud-design-pattern-in-autodesk-20140607.pdf

[26] AWS: Amazon Auto Scaling. Online article (jun 2019), available at https://aws.amazon.com/fr/autoscaling/

[27] Bachmann, F., Bass, L., Clements, P., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting software architecture: Documenting interfaces. Tech. Rep. CMU/SEI-2003-TR-016, Software Engineering Institute, Carnegie Mellon University (2003)

[28] Baur, D., Seybold, D., Griesinger, F., Tsitsipas, A., Hauser, C.B., Domaschka, J.: Cloud orchestration features: Are tools fit for purpose? In: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC). pp. 95–101 (Dec 2015)

[29] Bellaaj, F., Brabra, H., Sellami, M., Gaaloul, W., Bhiri, S.: A transactional approach for reliable elastic cloud resources. In: 2019 IEEE International Conference on Services Computing, SCC 2019, Milan, Italy, July 8-13, 2019. pp. 154–161 (2019)

[30] Bergmayr, A., al.: From architecture modeling to application provisioning for the cloud by combining UML and TOSCA. In: Proceedings of the 6th International Conference on Cloud Computing and Services Science(CLOSER). pp. 97–108 (2016)

[31] Bhattacharjee, A., Barve, Y., Gokhale, A., Kuroda, T.: A model-driven approach to automate the deployment and management of cloud services. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). pp. 109–114 (Dec 2018)

[32] Brabra, H., Mtibaa, A., Gaaloul, W., Benatallah, B., Gargouri, F.: Model-driven orchestration for cloud resources. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). pp. 422–429 (July 2019)

[33] Brabra, H.: cEDMCore: an integrated system for high-level management of multi-cloud elasticity, https://github.com/hayo03/cEDMcore.elastcity

[34] Brabra, H.: ORAP-Detector, https://github.com/hayo03/ORAP-Detector

[35] Brabra, H., Mtibaa, A., Gaaloul, W., Benatallah, B.: Model-driven elasticity for cloud resources. In: Advanced Information Systems Engineering - 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings. pp. 187–202 (2018)

[36] Brabra, H., Mtibaa, A., Gaaloul, W., Benatallah, B.: Toward higher-level abstractions based on state machine for cloud resources elasticity. Information Systems (2019)

[37] Brabra, H., Mtibaa, A., Petrillo, F., Merle, P., Sliman, L., Moha, N., Gaaloul, W., Guéhéneuc, Y., Benatallah, B., Gargouri, F.: On semantic detection of cloud API (anti)patterns. Information and Software Technology 107, 65–82 (2019)

[38] Brabra, H., Mtibaa, A., Sliman, L., Gaaloul, W., Benatallah, B., Gargouri, F.: Detecting Cloud (Anti)Patterns: OCCI Perspective. In: Proceedings of 14th International Conference on Service-Oriented Computing, ICSOC'16. pp. 202–218. Springer International Publishing (2016)

[39] Breitenbücher, U., Endres, C., Kepes, K., Kopp, O., Leymann, F., Wagner, S., Wettinger, J., Zimmermann, M.: The opentosca ecosystem - concepts and tools. In: APPIS 2019 (2019)

[40] Brogi, A., Carrasco, J., Cubo, J., D'Andria, F., Di Nitto, E., Guerriero, M., Pérez, D., Pimentel, E., Soldani, J.: Seaclouds: An open reference architecture for multi-cloud governance. In: Software Architecture. pp. 334–338. Springer International Publishing (2016)

[41] Brogi, A., Rinaldi, L., Soldani, J.: Tosker: Orchestrating applications with tosca and docker. In: ESOCC. pp. 130–144 (2017)

[42] Carrasco, J., Cubo, J., Pimentel, Ernesto", e.G., Tran, C.: Towards a flexible deployment of multi-cloud applications based on tosca and camp. In: Advances in Service-Oriented and Cloud Computing. pp. 278–286. Springer International Publishing, Cham (2015)

[43] Carrasco, J., Cubo, J., Pimentel, E.: Towards a flexible deployment of multi-cloud applications based on tosca and camp. In: Advances in Service-Oriented and Cloud Computing. pp. 278–286. Springer International Publishing, Cham (2015)

[44] Chef: Get started with chef automate. online article (jun 2018), available at https://automate.chef.io/

[45] Copil, G., Moldovan, D., Truong, H.L., Dustdar, S.: Sybl: An extensible language for controlling elasticity in cloud applications. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. pp. 112–119 (2013)

[46] Daigneau, R.: Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Addison-Wesley (2011)

[47] Denis, W., Chai, B.M., Boualem, B., Jian, C.: A model-driven framework for interoperable cloud resources management. In: 14th International Conference Service-Oriented Computing, ICSOC 2016 (2016)

[48] DevOps-community: Terraform registry: Discover providers for any service, and modules for common infrastructure configurations. Tech. rep., available at https://registry.terraform.io/

[49] Docker: Docker documentation. online article (jun 2018), available at https://docs.docker.com/

[50] Drescher, M., Parák, B., Wallom, D.: OCCI Compute Resource Templates Profile. Recommandation GFD-R-P.222, Open Grid Forum (Oct 2016), http://www.ogf.org/documents/GFD.222.pdf

[51] Dudney, B., Asbury, S., Krozak, J.K., Wittkopf, K.: J2EE Anti-Patterns. John Wiley Sons Inc (2003)

[52] Dullmann, T.F.: Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In: ICPE (2017)

[53] Dustdar, S., Guo, Y., Satzger, B., Truong, H.L.: Principles of elastic processes. Internet Computing, IEEE 15, 66 – 71 (11 2011)

[54] Eclipse-Foundation: EMF: Eclipse Modeling Framework, available at https://www.eclipse.org/modeling/emf/

[55] Eclipse-Foundation: Siruis, available at https://www.eclipse.org/sirius/

[56] Eclipse-Foundation: xText: Framework for development of programming and domain-specific languages, available at https://www.eclipse.org/Xtext/

[57] Edmonds, A., Metsch, T., Papaspyrou, A., Richardson, A.: Toward an open cloud standard. IEEE Internet Computing 16(4), 15–25 (July 2012)

[58] Edmonds, A., Metsch, T.: Open Cloud Computing Interface – Text Rendering. Recommandation GFD-R-P.229, Open Grid Forum (Oct 2016), http://www.ogf.org/documents/GFD.229.pdf

[59] Edmonds, A., Metsch, T., Papaspyrou, A., Richardson, A.: Toward an Open Cloud Standard. IEEE Internet Computing 16(4), 15–25 (2012)

[60] homas Erl, Robert Cope, A.N.: Cloud Computing Design Patterns. Prentice Hall (2015)

[61] Erl., T.: Soa design patterns. Prentice Hall PTR (2009)

[62] EsperTech-Inc.: Esper: Language, compiler and runtime for complex event processing (CEP) and streaming analytics, available at http://www.espertech.com/esper

[63] Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer (2014)

[64] Ferry, N., Song, H., Rossini, A., Chauvel, F., Solberg, A.: Cloudmf: Applying mde to tame the complexity of managing multi-cloud applications. In: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing. pp. 269–277 (Dec 2014)

[65] Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)

[66] Flexiant-Corporation: Flexiant Cloud Orchestrator, available at https://www.flexiant.com/

[67] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)

[68] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)

[69] García, Á.L., Fernández-del-Castillo, E., Fernández, P.O.: Standards for enabling heterogeneous iaas cloud federations. CoRR abs/1711.08045 (2017), http://arxiv.org/abs/1711.08045

[70] Gasparis, E., Nicholson, J., Eden, Amnon H.", e.G., Howse, J., Lee, J.: Lepus3: An object-oriented design description language. In: Diagrammatic Representation and Inference. pp. 364–367. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

[71] Global, F.: Service orchestration: Increasing the efficiency of hybrid it. Tech. rep., A Forrester Consulting Thought Leadership Paper Commissioned By Fujitsu (2017)

[72] Google: Google Cloud Platform, https://cloud.google.com/compute/

[73] Grafana-Labs: The open-source platform for monitoring and observability., available at https://github.com/grafana/grafana

[74] Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition 5(2), 199 – 220 (1993)

[75] Harsh, P., Dudouet, F., Cascella, R.G., Jegou, Y., Morin, C.: Using open standards for interoperability issues, solutions, and challenges facing cloud computing. In: 2012 8th international conference on network and service management (cnsm) and 2012 workshop on systems virtualiztion management (svm). pp. 435–440 (2012)

[76] Haupt, F., Leymann, F., Scherer, A., Vukojevic-Haupt, K.: A framework for the structural analysis of rest apis. In: 2017 IEEE International Conference on Software Architecture (ICSA). pp. 55–58 (2017)

[77] Haupt, F., Leymann, F., Vukojevic-Haupt, K.: Api governance support through the structural analysis of rest apis. Computer Science 33(3-4), 291–303 (2018)

[78] IBM: Softlayer Auto scale (jun 2018), available at http://www.softlayer.com/fr/autoscale

[79] James, A., Chung, J.Y.: Business and industry specific cloud: Challenges and opportunities. Future Generation Computer Systems 48, 39 – 45 (2015), http://www.sciencedirect.com/science/article/pii/S0167739X14002593

[80] Jrad, A.B., Bhiri, S., Tata, S.: Description and evaluation of elasticity strategies for business processes in the cloud. In: 2016 IEEE International Conference on Services Computing (SCC). pp. 203–210 (2016)

[81] Kahani, N., Bagherzadeh, M., Cordy, J.R., Dingel, J., Varró, D.: Survey and classification of model transformation tools. Softw. Syst. Model. 18(4), 2361–2397 (Aug 2019)

[82] Katsaros, G.: Open Cloud Computing Interface – Service Level Agreements. Recommandation GFD-R-P.228, Open Grid Forum (Oct 2016), http://www.ogf.org/documents/GFD.228.pdf

[83] Katsaros, G., Menzel, M., Lenk, A., Rake-Revelant, J., Skipp, R., Eberhardt, J.: Cloud application portability with tosca, chef and openstack. In: 2014 IEEE International Conference on Cloud Engineering. pp. 295–302 (2014)

[84] Kaur, K., Sharma, D.S., Kahlon, D.K.S.: Interoperability and portability approaches in inter-connected clouds: A review. ACM Comput. Surv. 50(4), 49:1–49:40 (Oct 2017)

[85] Kessentini, M., Vaucher, S., Sahraoui, H.: Deviance from Perfection is a Better Criterion Than Closeness to Evil when Identifying Risky Code. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. pp. 113–122. ASE'10 (2010)

[86] Kim, D.K., France, R., Ghosh, S., Song, E.: A role-based metamodeling approach to specifying design patterns. In: Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003. pp. 452–457 (Nov 2003)

[87] Kirasić, D., Basch, Danko", e.I., Howlett, R.J., Jain, L.C.: Ontology-based design pattern recognition. In: Knowledge-Based Intelligent Information and Engineering Systems. pp. 384–393. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

[88] Kovács, J., Kacsuk, P.: Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures", journal="journal of grid computing 16(1), 19–37 (Mar 2018)

[89] Kritikos, K., Domaschka, J., Rossini, A.: Srl: A scalability rule language for multi-cloud environments. In: 2014 IEEE 6th International Conference on Cloud Computing Technology and Science. pp. 1–9 (2014)

[90] Kubernetes: Kubernetes documentation. online article (jun 2018), available at https://kubernetes.io/docs/

[91] LeGuennec, A., Sunye, G., Jezequel, J.M.: Precise modeling of design patterns. In: Proceedings of the Unified Modeling Language: Advancing the Standard Third International Conference. pp. 4 82–496 (2000)

[92] Lewis, G.A.: Role of standards in cloud-computing interoperability. In: 2013 46th Hawaii International Conference on System Sciences. pp. 1652–1661 (2013)

[93] Liu, C., Loo, B.T., Mao, Y.: Declarative automated cloud resource orchestration. In: Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC) (2011)

[94] Loulloudes, N., Sofokleous, C., Trihinas, D., Dikaiakos, M.D., Pallis, G.: Enabling interoperable cloud application management through an open source ecosystem. IEEE Internet Computing 19(3), 54–59 (May 2015)

[95] Luhunu, L., Syriani, E.: Survey on template-based code generation. In: Proceedings of MODELS 2017. pp. 468–471 (2017)

[96] Lwakatare, L.E., Kuvaja, P., Oivo, M.: Dimensions of devops. In: Lassenius, C., Dingsøyr, T., Paasivaara, M. (eds.) Agile Processes in Software Engineering and Extreme Programming. pp. 212–217. Springer International Publishing, Cham (2015)

[97] Maleshkova, M., Pedrinaci, C., Domingue, J.: Investigating Web APIs on the World Wide Web. In: 2010 Eighth IEEE European Conference on Web Services. pp. 107–114. IEEE (dec 2010)

[98] Mao, Y., Liu, C., van der Merwe, J.E., Fernández, M.F.: Cloud resource orchestration: A data-centric approach. In: CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings. pp. 241–248 (2011)

[99] Mapelsden, D., Hosking, J., Grundy, J.: Design pattern modelling and instantiation using dpml. In: Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications. pp. 3–11. CRPIT '02, Australian Computer Society, Inc. (2002)

[100] Martino, B.D., Esposito, A., Cretella, G.: Semantic Representation of Cloud Patterns and Services with Automated Reasoning to support Cloud Application Portability. IEEE Transactions on Cloud Computing (2015)

[101] Massé, M.: REST API Design Rulebook. Tech. rep., O'Reilly Media, Sebastopol (2011)

[102] Merle, P., Gourdin, C., Mitton, N.: Mobile Cloud Robotics as a Service with OCCIware. In: Proceedings of the 2nd IEEE International Congress on Internet of Things, IEEE ICIOT 2017 (Jun 2017), best Paper Award

[103] Metsch, T., Edmonds, A.: Open Cloud Computing Interface - RESTful HTTP Rendering. Recommandation GFD-R-P.185, Open Grid Forum (2011), https://www.ogf.org/documents/GFD.185.pdf

[104] Metsch, T., Edmonds, A., Parák, B.: Open Cloud Computing Interface – Infrastructure. Recommandation GFD-R-P.224, Open Grid Forum (Oct 2016), http://www.ogf.org/documents/GFD.224.pdf

[105] Metsch, T., Mohamed, M.: Open Cloud Computing Interface – Platform. Recommandation GFD-R-P.227, Open Grid Forum (Oct 2016), http://www.ogf.org/documents/GFD.227.pdf

[106] Microsoft: Cloud design patterns: Prescriptive architecture guidance for cloud applications. Tech. rep. (2015), https://docs.microsoft.com/en-us/previous-versions/msp-n-p/dn568099(v=pandp.10)

[107] Microsoft-Azure: Azure Autoscale . Online article (jun 2019), available at https://azure.microsoft.com/fr-fr/features/autoscale/

[108] Moha, N., al: Service-Oriented Computing: 10th International Conference, ICSOC'12, chap. Specification and Detection of SOA Antipatterns, pp. 1–16. Springer Berlin Heidelberg (2012)

[109] Mohamed, M., Belaïd, D., Tata, S.: Extending occi for autonomic management in the cloud. Journal of Systems and Software 122, 416 – 429 (2016)

[110] Nacer, M.A., Mohamed, M., Sellami, M., Yangui, S., Tata, S.: A Generic Multi-PaaS API for Cloud Application Provisioning, available at http://www-inf.it-sudparis.eu/SIMBAD/tools/COAPS/

[111] NASA: CLIPS: A Tool for Building Expert Systems, available at http://clipsrules.sourceforge.net/

[112] NIST: The nist definition of cloud computing. Tech. rep., National Institute of Standards and Technology (2011)

[113] Nyrén, R., Edmonds, A., Metsch, T., Parák, B.: Open Cloud Computing Interface – HTTP Protocol. Recommandation GFD-R-P.223, Open Grid Forum (Oct 2016), http://www.ogf.org/documents/GFD.223.pdf

[114] Nyrén, R., Edmonds, A., Papaspyrou, A., Metsch, T., Parák, B.: Open Cloud Computing Interface – Core. Recommendation GFD-R-P.221, Open Grid Forum (Oct 2016), http://www.ogf.org/documents/GFD.221.pdf

[115] Nyrén, R., Feldhaus, F., Parák, B., Sustr, Z.: Open Cloud Computing Interface – JSON Rendering. Recommandation GFD-R-P.226, Open Grid Forum (Oct 2016), http://www.ogf.org/documents/GFD.226.pdf

[116] OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA), available at https://www.oasis-open.org/committees/tosca/

[117] OASIS: Oasis cloud application management for platforms (camp) . online article (jun 2018), from https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=camp

[118] OGF: Open Cloud Computing Interface, http://occi-wg.org/

[119] OMG: Meta Object Facility (MOF) 2.0 Core Specification, oMG Documentformal/2006-01-01, available at : https://www.omg.org/spec/MOF/2.4.1/PDF

[120] OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, available at https://www.omg.org/spec/QVT/1.1/PDF

[121] OpenNebula: OpenNebula OCCI RESTful API, available at http://archives.opennebula.org/documentation:archives:rel4.0:occiddoverview

[122] OpenStack: OpenStack OCCI interface, available at https://ooi.readthedocs.io/en/stable/user/usage.html

[123] OpenStack: OpenStack HEAT Documentation (2018)

[124] Palma, F., Dubois, J., Moha, N., Guéhéneuc, Y.: Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach. In: Proceedings of 12th International Conference on Service-Oriented Computing, ICSOC'14. pp. 230–244 (2014)

[125] Palma, F., Gonzalez-Huerta, J., Moha, N., Guéhéneuc, Y.G., Guy, T.: Are RESTful APIs Well-Designed? Detection of their Linguistic (Anti)Patterns. In: Proceedings of 13th International Conference on Service-Oriented Computing, ICSOC'15. pp. 171–187 (2015)

[126] Palma, F., Moha, N., Tremblay, G., Guéhéneuc, Y.G.: Specification and Detection of SOA Antipatterns in Web Services. In: Proceedings of 8th European Conference on Software Architecture, ECSA'14. pp. 58–73 (2014)

[127] Pascal, H., Markus, K., Sebastian, R.: Foundations of Semantic Web Technologies. Chapman and Hall/CRC, 1st edn. (2009)

[128] Pautasso, C.: Some rest design patterns (and anti-patterns). Tech. rep. (2009), http://www.jopera.org/node/442

[129] Petrillo, F., Merle, P., Moha, N., Guéhéneuc, Y.G.: Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study. In: Proceedings of 14th International Conference on Service-Oriented Computing, ICSOC'16. pp. 171–187. Springer International Publishing (2016)

[130] Pham, L.M., Tchana, A., Donsez, D., de Palma, N., Zurczak, V., Gibello, P.: Roboconf: A hybrid cloud orchestrator to deploy complex applications. In: 2015 IEEE 8th International Conference on Cloud Computing. pp. 365–372 (June 2015)

[131] Puppet-Labs: Puppet enterprise (jun 2018), available at https://puppet.com/products/puppet-enterprise

[132] Rackspace: Rackspace RESTful API, available at https://developer.rackspace.com/docs/cloud-servers/v2/api-reference/

[133] RAML-Workgroup: RAML: RESTful API Modeling Languagel, available at https://raml.org/about-raml

[134] Ranjan, R., Benatallah, B.: Programming cloud resource orchestration framework: Operations and research challenges. CoRR abs/1204.2204 (2012)

[135] Ranjan, R., Benatallah, B., Dustdar, S., Papazoglou, M.P.: Cloud resource orchestration programming: Overview, issues, and directions. IEEE Internet Computing 19(5), 46–56 (2015)

[136] RedHat: Drools - Business Rules Management System, available at https://www.drools.org/

[137] Richardson, L., Ruby, S.: RESTful Web Services. Tech. rep., O'Reilly Media Inc., Sebastopol (2007)

[138] Rodríguez, C., Baez, M., Daniel, F., Casati, F., Carlos, J., Canali, L., Percannella, G.: Rest apis: A large-scale analysis of compliance with principles and best practices. In: Proceedings of 16th International Conference on Web Engineering (ICWE2016). pp. 21–39 (2016)

[139] Rodriguez, J.M., Crasso, M., Zunino, A., Campo, M.: Improving web service descriptions for effective service discovery. Science of Computer Programming 75(11), 1001 – 1021 (2010)

[140] Sandobalin, J., Insfrán, E., Abrahão, S.: An infrastructure modelling tool for cloud provisioning. In: IEEE SCC. pp. 354–361 (2017)

[141] Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Software 20(5), 42–45 (2003)

[142] Settas, D., Bibi, S., Sfetsos, P., Stamelos, I., Gerogiannis, V.: Using bayesian belief networks to model software project management antipatterns. In: Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06). pp. 117–124 (Aug 2006)

[143] Settas, D.L., Meditskos, G., Stamelos, I.G., Bassiliades, N.: SPARSE: A symptom-based antipattern retrieval knowledge-based system using Semantic Web technologies. Expert Systems with Applications 38(6), 7633 – 7646 (2011)

[144] da Silva, A.R.: Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems Structures 43, 139 – 155 (2015)

[145] Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical owl-dl reasoner. Journal of Web Semantics 5(2), 51 – 53 (2007), software Engineering and the Semantic Web

[146] SmartBear: Swagger, available at https://swagger.io/

[147] Stowe, M.: Undisturbed REST: A Guide to Designing the Perfect API. Tech. rep., MuleSoft, San Francisco, USA (2015)

[148] Syriani, E., Luhunu, L., Sahraoui, H.: Systematic mapping study of template-based code generation. Computer Languages, Systems and Structures 52, 43 – 62 (2018)

[149] Terraform: Terraform documentation. online article (jun 2018), available at https://www.terraform.io/docs/index.html

[150] Tilkov, S.: Rest anti-patterns (2008)

[151] Trcka, N., van der Aalst, W.M.P., Sidorova, N.: Data-flow anti-patterns: Discovering data-flow errors in workflows. In: Advanced Information Systems Engineering, 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June 8-12, 2009. Proceedings. pp. 425–439 (2009)

[152] Ubuntu-Juju: Juju documentation. online article (jun 2018), available at https://docs.jujucharms.com/2.4/en/about-juju

[153] Uwe, Z., Carsten, H., Schahram, D.: Modeling process-driven and service-oriented architectures using patterns and pattern primitives. ACM Trans. Web 1(3) (Sep 2007)

[154] Vinoski, S.: RESTful Web Services Development Checklist. IEEE Internet Computing 12(6), 96–95 (2008)

[155] Weerasiri, D., Barukh, M.C., Benatallah, B., Sheng, Q.Z., Ranjan, R.: A taxonomy and survey of cloud resource orchestration techniques. ACM Comput. Surv. 50(2), 1–41

[156] Wettinger, J., al: Streamlining devops automation for cloud applications using TOSCA as standardized metamodel. Future Generation Comp. Syst. 56 (2016)

[157] Zabolotnyi, R., Leitner, P., Schulte, S., Dustdar, S.: Speedl-a declarative event-based language to define the scaling behavior of cloud applications. In: 2015 IEEE World Congress on Services. pp. 71–78 (2015)

[158] Zhan, Z.H., Liu, X.F., Gong, Y.J., Zhang, J., Chung, H., Li, Y.: Cloud computing resource scheduling and a survey of its evolutionary approaches. ACM Computing Surveys 47, 1–33 (07 2015)

[159] Zhang, Z., Wu, C., Cheung, D.W.: A survey on cloud interoperability: Taxonomies, standards, and practice. SIGMETRICS Perform. Eval. Rev. 40(4), 13–22 (Apr 2013)

[160] Zhu, L., Bass, L., Champlin-Scharff, G.: Devops and its practices. IEEE Software 33(3), 32–34 (May 2016)