Alex Palesandro

Deployment and Security supervision for multi-cloud architectures


Disponible sur : http://www.theses.fr/2018LYSE3071

Document diffusé sous le contrat Creative Commons « Attribution – Pas d’utilisation commerciale - Pas de modification » Vous êtes libre de le reproduire, de le distribuer et de le communiquer au public à condition d’en mentionner le nom de l’auteur et de ne pas le modifier, le transformer, l’adapter ni l’utiliser à des fins commerciales.
Deployment and Security supervision for multi-cloud architectures
A Daniele e Angela,
che non mi hanno mai lasciato
ABSTRACT

Cloud Computing represents one of the most important changes in information and communications technology (ICT) of the latest ten years. The elastic, on-demand, and pay-per-use service model that cloud offers is able to not only provide as many large-scale computation and storage resources as needed, but also to do so as long as requested and at a pay-per-use price.

However, after a decade since its commercial debut, there are still several applications that cloud computing is not able to fully serve. These are the applications that, due to their particularly stringent requirements, must rely simultaneously on multiple Cloud Service Providers (CSPs), rather than only one. Multiple CSPs can in fact offer a better availability, improve QoS, and break the business dependence w.r.t. a single CSP. A cloud infrastructure based on multiple CSPs is called multi-cloud.

Despite the benefits of multi-clouds, organisations (i.e developers and operators of IT services) seldom accept the challenge of building applications and crossing multiple CSP domains [42]. In fact, multi-CSP architectures come at the cost of more complex applications and the logic in terms of architecture and performance optimization. Recently, Multi-cloud client-oriented architectures emerged as important approach to construct multi-cloud applications. It provides cloud consumers a mechanism to allocate resources over multiple CSPs without requiring any cooperation among the CSPs themselves. In particular, Infrastructure as Code-based (IaC-based) [59] represent the reference paradigm when building multi-cloud applications. Consumers describe infrastructure resources as specific code introducing easily reproducible and flexible infrastructures, that can be easily integrated in the software lifecycle.

However, the adoption of IaC in the multi-cloud context is limited by the fact that the cloud consumer cannot easily reuse the infrastructure code across different applications. This is due to two major problems, which we investigate in this manuscript.

First, infrastructure are composed of functional (e.g. resources for applications) and non-functional services (e.g. monitoring). Non-functional related code should be shared at most across different applications and cloud consumers. However, this separation between functional and non-functional code is often blurred and, therefore, non-functional code is hard to be shared across them. Consequently, the codebase and the effort required to maintain services becomes rapidly intractable. In this work, we introduce the AOP-based weaving to allow injection/eviction of non-functional services in multi-clouds. This enables the possibility of code re-using across different cloud consumers (e.g., their different multi-cloud infrastructures) and static analysis of infrastructure templates. Furthermore, we present a TML (TOSCA Manipulation Language) aspect specification language to dynamically inject "non-functional" services to the virtual multi-cloud infrastructure. The TML expresses flexible connections among the application code and the reusable non-functional components.

Second, the multi-cloud paradigm is limited by the "least common denominator" barrier. The cloud consumer can hardly obtain an optimized usage of resources and services through existing IaC frameworks. Despite compatible with different CSPs, those frameworks do not specialize the output according to deployment context. To tackle the
"under-specialization" of multi-cloud templates, we introduce a "context-based matching" scheduling algorithm to select the most compelling set of CSPs according to the cloud consumer needs. Such algorithm takes as input the list of available CSPs and their data-centers together with a set of consumer-provided parameters (e.g. the amount of distinct CSPs, the service template, the optimization criteria). The algorithm outputs the list of the compelling data-centers (possibly located across different CSPs) that satisfy the user's requirements. The context is managed by extending the TOSCA formalism. Such extension consists in encapsulating inside the TOSCA format the matching parameters between CSP features, cloud consumer preferences and implementation limitations.

Third, since the beginning of the cloud era, the adoption of public CSPs had important consequences not only in terms of security and control over data but also in terms of loss of control over the infrastructure. Compared to private infrastructures, the loss of control may not only concern the physical possession of data but also the loss of control on "lower" system layers of the software stack, notably the virtualization layer or hypervisor. To regain control over remote public infrastructure, we introduce a virtualization architecture which provides more control to the user as mentioned, still providing a compatibility layer but being capable of leveraging enhanced features. Based on microkernels and nested virtualization, this architecture enables the user to have more control on lower layers of the infrastructure (e.g. the hypervisor), assuring strong isolation w.r.t other tenants and the compatibility with the rest of multi-cloud.

To validate such contributions, we defined an end-to-end workflow to optimize a multi-cloud infrastructure definition. More precisely, in our model, the cloud consumer initially models the IaC code as an high-level graph of services, leveraging the combination of TML and context-based matching adoption. The output of this workflow is the instantiation of such optimized and fully-featured multi-cloud on most suitable CSPs. We implemented MANTUS, a multi-cloud compiler, which encapsulates this workflow and we benchmarked this implementation according to different perspectives as scalability and performance.

Leveraging Aspect-Oriented weaving and Context-based matching, a cloud consumer can accelerate at scale multi-cloud adoption by reducing the code base and obtaining a usage optimized infrastructure code, without having to deal with specific CSP technologies as required by currently available solutions.
I have so many people to thank, who supported me during this journey. First and foremost, I am grateful to Prof. Chirine Ghedira and Dr. Nadia Bennani for their supervision and for teaching me the right approach to research. Also, I would like to thank Dr. Marc Lacoste, my industry advisor, for his precious advice during the writing process of papers and discussions.

I am really grateful to all the member of the jury which have decided to spend their time and efforts to read and evaluate this work.

Giulia, I will never forget your incredible support during this tough and final step of my studies. Thank you.

I want to thank my parents Piera e Gino, my sister Erica and her boyfriend Paolo for their support. also supported me with good words e "vettovaglie". We do not live together anymore, yet they were always there for me, to cheer me up and lead by example (and replenish my pantry with Italian food!).

I am grateful to Prof. Fulvio Risso, who gave me priceless lessons. Thanks to all the members of the Netgroup research team I worked with: Ivano, Matteo, Fabio, Roberto, Matteo, Marco, Serena, Francesco. Our time working together was short, but nevertheless you made it valuable and fun.

Thank you also to my extended family: my aunts, uncles, cousins. In particular, I want to thank Mirella, whose handmade scarf made my Winters in Paris warmer.

I can not forget my wonderful "collegues" at Politecnico di Torino. We shared so much time together learning about operating systems and algorithm, preparing exams and also chilling out. Thank you to all members of the "Rosso's Society"!

I would like to thank all the friends I met in Paris, who made this city my home: Sandro, Giulio, Fabrizio, Francesco, Francesco, Matilde, Nicolas, Bruna, Luca, Luca, Aurelie, Angelo, Antonia, Hannah, Cecilia, Umberto, Michele, Mathieu.

I thank my colleagues at Orange (Pascal, Kahina, Micael, Christophe, Gilles, Xavier, Xavier, Todor, Said, Aymeric, Isabelle, Loic), my manager Sébastien and, in particular, the other PhD students: Alicia, Sandra, Maxime, Maxime, Paul, Xiao, Benjamin, Pierre. It was great to work with you and I really learnt so much from our discussions (not only about computer science). In particular, I will always be grateful to Paul, who explained me what really is open source and plenty of other stuff. I would also thank my former colleague Aurelien Wailly, which supervised me during the master internship. Thank you for your precious lessons!

I would like to thank all my good friends in my hometown Pinerolo. So thank you Alessandro, Elisa, Fabio, Francesco, Paride, Marco, Davide, Alain, Francesco, Francesco, Andrea, Serena.
A special mention to the friends with I have started this adventure in France during the Erasmus exchange in Grenoble and all the wonderful people that I met there. Thank you, Andrea, Federica, Luca, Silvia, Chiara, Alex, Thuat.

Thank you also to all people I met during the last steps of this long but wonderful journey! Thank you Loïc, Alexandre, Vincent, Kahina, Yassine, Jihad, Erkan, Smael, Mahmoud, Xavier, Vasyl! Thank you guys for your incredible support!

Last but not least, thank you also to my current colleagues at D2-SI: it is great to work with you guys!
CONTENTS

List of Figures ............................................. 13
List of Tables ............................................. 15
1 CONTEXT AND PROBLEM STATEMENT ......................... 3
  1.1 Context .................................................. 3
    1.1.1 The Health-Care Use-Case .......................... 3
      1.1.1.1 Tele-medicine .................................. 4
      1.1.1.2 EHR analysis .................................. 4
      1.1.1.3 Organizations and Actors ......................... 4
    1.1.2 Limitations of single-provider model ................. 5
    1.1.3 The way of multi-cloud ............................. 6
  1.2 Multi-cloud Scenario ...................................... 8
  1.3 Problem Statement ....................................... 9
    1.3.1 Lack of extensibility ............................... 9
    1.3.2 Lack of Specialization ............................. 10
    1.3.3 Lack of control on public CSPs ...................... 11
  1.4 Research objectives and contributions .................... 12
    1.4.1 Multi-cloud infrastructure extensibility ............ 13
    1.4.2 Multi-cloud infrastructure specialization .......... 14
    1.4.3 Building and deploying a multi-cloud architecture . 14
  1.5 Multi-cloud deployment step by step ..................... 15
    1.5.1 Multi-cloud deployment workflow .................... 16
  1.6 Thesis Plan ............................................. 19
2 CLOUD COMPUTING: STATE OF THE ART ....................... 21
  2.1 Introduction ........................................... 21
  2.2 Background ............................................. 22
    2.2.1 Taxonomy ............................................ 22
      2.2.1.1 Service/Deployment models ....................... 22
      2.2.1.2 IaaS Cloud Enabling Technologies ................ 23
      2.2.1.3 System Virtualization ........................... 24
    2.2.2 Current limitations of Cloud Computing ............... 26
    2.2.3 CSP Lock-in ......................................... 27
      2.2.3.1 Lack of control ................................ 27
      2.2.3.2 QoS and geo-location issues ...................... 28
  2.3 State of the Art: Inter-cloud architectures ............. 28
    2.3.1 Benefits and roadblocks of CSP interconnection ....... 29
    2.3.2 Provider-centric architectures .......................... 30
      2.3.2.1 CSP Federation .................................. 30
      2.3.2.2 Hybrid Cloud ................................... 32
    2.3.3 Client-oriented architectures ........................ 33
      2.3.3.1 Brokering ....................................... 33
      2.3.3.2 Multi-cloud Libraries ............................ 34
      2.3.3.3 Overlay Approaches ............................... 35
    2.3.4 Conclusion ........................................... 35
A.1 Introduction ................................................. 163
A.2 Multi-Cloud Scenario ...................................... 165
A.3 Enoncé du problème ........................................ 167
   A.3.1 Manque d’extensibilité ............................... 167
   A.3.2 Spécialisation ......................................... 168
   A.3.3 Manque de contrôle .................................. 169
A.4 Objectives des recherche et contributions ............ 170
   A.4.1 Extensibilité de l’infrastructure multi-cloud ....... 170
      A.4.1.1 TML ............................................. 171
      A.4.1.2 MANTUS 1.0 ..................................... 171
   A.4.2 Spécialisation des infrastructure Multi-cloud .... 172
      A.4.2.1 Context-based matching ......................... 173
      A.4.2.2 Implémentation: MANTUS 2.0 ................. 174
      A.4.2.3 Construction et déploiement d’une architecture multi-cloud/75
   A.4.3 Perte de contrôle ..................................... 178
A.5 Conclusion .................................................. 180
A.6 Limitations .................................................. 183
A.7 Future Work ................................................ 183
   A.7.1 Améliorations dans le tissage d’aspect non fonctionnel .... 185
   A.7.2 Specialization and predictions ....................... 185
   A.7.3 Reconfiguration Autonomique ....................... 185
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Multi-cloud health-care services</td>
<td>5</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Multi-cloud MPC architecture. Different organisations (e.g. CDOs, laboratories) can securely share their data through MPC, using MPC without relying on an omniscient 3rd party.</td>
<td>7</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Multi-cloud workflow and manuscript contribution mapping</td>
<td>17</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>Manuscript plan and contribution mapping</td>
<td>18</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Full virtualization example architecture</td>
<td>25</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>OS containers</td>
<td>26</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>OS containers</td>
<td>26</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Different levels of control in private and public cloud settings.</td>
<td>28</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>CSP Federation</td>
<td>30</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Example of Cloud Bursting through Hybrid Cloud. The consumer accesses private cloud resources which are spawned transparently on a public CSP.</td>
<td>32</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>Multi-CSP Broker</td>
<td>33</td>
</tr>
<tr>
<td>Figure 2.8</td>
<td>Library-based multi-cloud</td>
<td>34</td>
</tr>
<tr>
<td>Figure 2.9</td>
<td>Overlay-based multi-cloud</td>
<td>35</td>
</tr>
<tr>
<td>Figure 2.10</td>
<td>IaC schema for service configuration on a specified CSP</td>
<td>38</td>
</tr>
<tr>
<td>Figure 2.11</td>
<td>IaC-based MCL architecture: the IaC principle is now extended to allocate and configure resources on multiple CSPs</td>
<td>39</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>IaC extensibility contribution in the context of multi-cloud deployment</td>
<td>48</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>TOSCA ServiceTemplate example [82]. A service template on the left is composed of NodeTypes and RelationshipTypes described on the right composing the TOSCA graph.</td>
<td>50</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>TOSCA Matching. The ServiceTemplate presented matches the NodeType and, therefore, it can replace the NodeType in the TOSCA matching process.</td>
<td>52</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>AOP schematic representation</td>
<td>53</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Graphic Representation of template resources</td>
<td>55</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>The template sample in figure 3.5 is weaved with Floating_ip script, which adds to a specified VM an external Floating IP addresses to provide external internet access.</td>
<td>59</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>IaC Specialization contribution in the context of multi-cloud deployment</td>
<td>62</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>TOSCA Exact Matching</td>
<td>66</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>TOSCA Plugin Matching</td>
<td>68</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Multi-Cloud Construction Workflow</td>
<td>77</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>End-to-end deployment of a multi-cloud</td>
<td>85</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Implementation contribution in the context of multi-cloud deployment</td>
<td>88</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>The ORBITS architecture</td>
<td>89</td>
</tr>
</tbody>
</table>
Figure 5.3 MANTUS autonomic behavior model based on FORMS [117] ........ 95
Figure 5.4 MANTUS autonomic behavior model ................................ 96
Figure 5.5 The ORBITS architecture weaved with a basic monitor aspect .. 97
Figure 5.6 MANTUS autonomic behavior model .............................. 99
Figure 6.1 U-Cloud contribution in the context of multi-cloud deployment .. 102
Figure 6.2 Multi-Cloud Architecture Proposal ............................... 106
Figure 6.3 U-Cloud Node Architecture Proposal ............................ 108
Figure 6.4 Instantiation of a U-Cloud ........................................ 111
Figure 6.5 Instantiation of a U-Cloud ........................................ 112
Figure 6.6 Reconfiguration of a U-Cloud .................................... 113
Figure 6.7 MANTUS workflow integration with a U-Cloud Node ........... 115
Figure 7.1 Openstack-based ORBITS prototype mapping .................... 119
Figure 7.2 Mesos Two-level scheduling mechanism .......................... 120
Figure 7.3 ORBITS OpenStack Prototype .................................... 121
Figure 7.4 TCP Latency benchmark on ORBITS ............................. 122
Figure 7.5 TCP Throughput benchmark on ORBITS .......................... 123
Figure 7.6 Wordpress - Requests Served, measured on ORBITS ............ 123
Figure 7.7 Wordpress - Throughput, measured on ORBITS .................. 124
Figure 7.8 U-Cloud Node Architecture Proposal ............................ 125
Figure 7.9 TCP Bandwidth - First Configuration ............................. 126
Figure 7.10 TCP Bandwidth - Second Configuration ........................ 126
Figure 7.11 TCP Bandwidth - Third Configuration .......................... 127
Figure 7.12 Weaving Efficiency - Marginal Cost ............................ 131
Figure 7.13 Compositional MANTUS Efficacy ................................ 131
Figure 7.14 Compositional Weaver Efficacy ................................. 133
Figure 7.15 Weaver Efficiency vs. Aspect Complexity ...................... 134
Figure 7.16 The MANTUS 2.0 Architecture ................................ 136
Figure 7.17 Memory-intensive MANTUS broker-dictionary benchmarks ... 138
Figure 7.18 Compute-intensive MANTUS broker-dictionary benchmarks ... 139
Figure 7.19 Disk-intensive MANTUS broker-dictionary benchmarks ....... 139
Figure 7.20 MANTUS flavor selection benchmark .......................... 140
Figure 7.21 Context-based execution time .................................... 141
Figure 7.22 MANTUS Solver Execution Time ............................... 142
Figure 7.23 Compositional Matcher/Weaver Efficacy Comparison .......... 143
Figure A.1 Multi-cloud health-care services .................................. 164
Figure A.2 Efficacité des différentes étapes de MANTUS ................. 172
Figure A.3 Construction de bout en bout d’une architecture Multi-Cloud à travers MANTUS et ORBITS .............................. 172
Figure A.4 Evaluation des performances de MANTUS context-based matching 175
Figure A.5 Architecture multi-cloud ORBITS ................................ 177
Figure A.6 Proposition d’architecture multi-cloud avec contrôle amélioré de la part de l’utilisateur .......................... 179
Figure A.7 Architecture du prototype U-Cloud Node ....................... 180
Figure A.8 Résumé du déploiement de bout en bout à travers ORBITS et MANTUS .......................... 181
Figure A.9 Résumé des contributions de thèse .............................. 184
LIST OF TABLES

Table 2.1 Provider Federation Architectectures ...................................... 31
Table 2.2 Broker-based Architectures .................................................. 33
Table 2.3 Comparison among different tools that can be adopted as Multi-
Cloud Libraries .................................................................................. 42
Table 2.4 Comparison among different Multi-Cloud Libraries considering
extensibility and specialization .......................................................... 44
Table 3.1 TML Filter operators ............................................................... 57
Table 3.2 TML advice actions ................................................................. 57
Table 4.1 Context Matching Model ........................................................ 70
Table 4.2 Translation mappings between TOSCA and provider-specific or-
chestration language ........................................................................ 83
Table 4.3 Multi-cloud input parameters and how they are tackled by MAN-
TUS workflow .................................................................................. 84
Table 5.1 Mapping context mutations to MANTUS reactions ................. 95
Table 7.1 Benchmark Variables .............................................................. 130
LIST OF ACRONYMS

AOP Aspect-Oriented Programming
CBH Component-based hypervisor
CDO Care Delivery Organization
CM Configuration Management Tool
CSP Cloud Service Provider
DSL Domain-specific Language
EHR Elastic Health Records
IaaS Infrastructure as a Service
IaC Infrastructure as Code
ICT Information and Communications Technology
MCL Multi-cloud Library
MH Micro-hypervisor
MPC Multi-party Computation
NFV Network Function Virtualization
NIST National Institute of Standards and Technology
PaaS Platform as a Service
PHR Personal Health Records
QoP Quality of Protection
QoS Quality of Service
SaaS Application as a Service
SLA Service Level Agreement
ST Service Template
TCB Trusted Compute Base
TML Tosca Manipulation Language
VM Virtual Machine
VMM Virtual Machine Monitor
1.1 CONTEXT

The cloud computing paradigm represents one of the most important trends in information and communications technology (ICT) of the last decade. The elastic, on-demand, and pay-per-use service model that cloud offers is able to not only provide as many large-scale computation and storage resources as needed, but also to do so as long as requested and at a pay-per-use price.

Many application domains are moving to the cloud. Prominent examples are Network function virtualization (NFV), a new approach to network function deployment which benefits of scalability and elasticity advantages of the cloud, and Big Data workloads, which benefit of the pay-per-use model of cloud computing.

There are several other applications that would benefit from cloud computing. However, these applications are subjected to particularly stringent requirements that can be fulfilled only when multiple Cloud Service Providers (CSPs), rather than one, are leveraged simultaneously. One of those domains is represented health-care systems, where the potential benefits obtained by cloud adoption are impacted by stringent constraints in terms of security and data geo-location, as we describe in the next section.

1.1.1 The Health-Care Use-Case

Let us consider a generic cloud-bursting [2] use-case, shown in Figure 1.1. A Care-Delivery Organization (CDO) leverages multiple public cloud providers to enhance the resources available on its private cloud. The deployment of multiple CSPs enables the CDO to manage at a lower cost different classes of workloads, ranging from telemedicine (e.g. remote healing) to data mining on electronic health records (EHRs). EHRs and, thus, the organizations dealing with EHRs, are subjected to strict constraints in terms of Quality of Protection (QoP) [53]. For instance, in the context of health care, we identify two main services that would benefit from a cloud infrastructure based on multiple CSPs. These are tele-medicine and data analysis over EHRs discussed, respectively, in Section 1.1.1.1 and Section 1.1.1.2. The actors involved in the design of these healthcare applications are presented in Section 1.1.1.3.
1.1.1.1 Tele-medicine

A first interesting use-case for multi-provider cloud includes several health-care activities concerning EHRs: consultation for patients, prescription management application for doctors or other institutions, assisted surgery [38]. Patients affected by chronic diseases or receiving daily treatments (e.g. drug therapy self-assessment questionnaire, periodic self-treatments, epidemiological studies) require a constant monitoring and produce Personal Health Records (PHR).

Services such as remote healing or remote access and consultation of EHRs are important applications that may dramatically benefit from cloud cost-reduction [127]. Cloud-based home healthcare systems are notably a widely investigated research area [63]. Such systems usually involve three main players: patients, practitioners and CDOs, such as hospitals. Current home-based scenarios remain limited to patients who usually may leverage the service on premise, also relying on the same practitioner or CDO. Besides, follow-me scenarios are not supported where travelling patients may require healing away from their principal residence, potentially relying on new practitioners and CDOs (see Figure 1.1). Such applications have, among others, three main types of requirements [63], which concern the underlying cloud infrastructures: (1) geographical constraints regarding where applications and data are hosted, usually imposed by laws (becoming more stringent recently [44]); (2) severe high-availability constraints [127]; and (3) severe constraints in terms of integrity and confidentiality of application and data treated [127].

1.1.1.2 EHR analysis

Analysis and aggregation on big data from different sources is gaining momentum in industrial and public institutions to help decision support [23]. This also includes healthcare organizations, which may gain an insight from analytics on EHRs. These, for instance, may lead to the discovery of associations and the understanding of patterns that potentially improve care. Also, this is achieved while lowering the costs [95].

However, such analysis have strict requirements in terms of security and privacy, in particular when dealing with remote healing and potentially with different CDOs. The EHRs provided by each CDO have to remain confidential to protect the patients’ privacy. Yet, the result of the statistics over these data have to be publicly known to all the CDOs that contributed with their inputs.

A prominent solution to do this is provided by secure multi-party computation (MPC) [123], a cryptographic primitive that enables computations among multiple parties in distributed fashion, where the input from each party is kept private. Low performance is a critical bottleneck of MPC systems [22], which are still considerably slower than approaches performing computations on data in clear. Therefore, increasing computing and storage capacity at a reasonable price using the cloud may be a viable way to address performance issues. However, the presence of an omniscient third party, the CSP, capable to reconstruct the whole secret poses limitations to the adoption of cloud computing.

1.1.1.3 Organizations and Actors

Many organizations cooperate in health-care domain. This includes CDOs (e.g. clinics, hospitals), scientific laboratories, insurance companies and pharmaceutical companies.
Figure 1.1: Multi-cloud health-care services

On the one hand, the health-care use-cases normally embrace a wide range of different actors involved (patients, pharmacists, CDOs administrative employees, doctors) using different classes of devices and having different responsibilities. Those can be considered as the end-user of health-care services: they have no specific technical training to deal with IT, and cloud computing usage should be transparent to them as much as possible.

On the other hand, those entities normally have to consider service developers and IT operators, which are responsible for developing, maintaining and operating applications. Developers and operators teams are normally coordinated by an IT services manager. Such actors represent the real consumers of the cloud computing services. They have to select the most suitable cloud computing providers, build and operate applications over services offered by CSPs. In the rest of the document, we refer to developers and operators as cloud consumers, as they are in charge of building healthcare applications, while fulfilling the requirements of this use-case.

1.1.2 Limitations of single-provider model

When analyzing the suitability of a single-provider model, we may observe that it is not completely satisfying to embrace the above mentioned use-cases.

On the one hand, Tele-medicine has strict requirements that cannot easily be met by a single provider. First, data processing has strict requirements in terms of location-awareness. Sensitive EHR data location has to be carefully monitored and may not cross nation boundaries enforcing country-specific jurisdictions, moreover incompatible with per-continent granularity of single-provider data centers. Second, single-provider availability guarantees may not be sufficient in medical environments [63]. QoS is also
impacted by latency, increasing with distance between service end-user (e.g. patient, doctor) and the data-center. Third, cloud providers have to be trusted entirely, e.g., raising privacy issues for medical data from a security but also from a commercial perspective, introducing a strong dependency with CDO.

On the other hand, a single-provider cloud approach may also not be applicable for MPC. Privacy of some MPC-based algorithms relies on secret-sharing techniques [22]. Computation among mutually non trusted parties have been addressed extensively in literature from the cryptographic perspective [123], leveraging secret sharing techniques to produce answers to a common question without having to share secret input from parties. A single cloud provider solution violates de facto the secret-sharing scheme [103], as the provider has access to all fragments of the secret. In Figure 1.2, we propose a reference MPC architecture, derived from the one presented in [23]. In our scenario, different CDOs put their data, leveraging secret-sharing techniques [104] on several uncorrelated cloud providers. Those data are then treated by some MPC framework designed to answer a medical/statistical request. In this architecture, any CDO participating to the computation cannot access the entire data of another CDO, but it is possible to extract aggregated information by the whole set leveraging MPC. In a nutshell, healthcare data analytics may look with interest to the exploitation of multiple underlying CSP [43, 45].

To sum up, outsourcing health records in distributed fashion may be done by actively selecting the countries and continents where the data-centers (DCs) of the deployed CSPs are located [63]. This enables the possibility to address at the same time two opposite types of constraints that CDOs have to fulfill. To do so, the cloud consumer should be capable to leverage transparently simultaneously multiple CSPs. In the next section, we analyze the benefits of multi-cloud and the obstacles that prevent an easy adoption of such an approach.

1.1.3 The way of multi-cloud

The deployment of multiple CSPs and their DCs is referred to as a multi-cloud (or inter-cloud) model [2]. The benefits of the multi-cloud model with respect to the single-CSP model and the way such architectures may be instantiated were investigated and formalized. Among these, there is notably a finer-grained distribution of resources obtained by leveraging multiple providers, which ultimately leads to an enhanced QoS of the applications, or QoP (Quality of Protection) if talking about CDOs applications. This does not only concern CDOs and EHRs management, but also all those applications having stringent requirements in terms of QoS. Furthermore, the optimized expenditures through dynamic price comparisons between CSPs may lead to a significant cost reduction for the organization that demands cloud resources.

However, organisations (i.e developers and operators of IT services) seldom accept the challenge of building applications and crossing multiple CSP domains [42]. In fact, despite the aforementioned opportunities, multi-CSP architectures come at the cost of three main limitations.

First, multi-CSP applications are not trivial to be built and executed. Indeed, together with the actual applications offered to clients, referred to as functional services, multi-cloud developers and operators are also responsible for ensuring so called non-functional services. These are, for instance, monitoring, security and availability. Non-functional
services are key for the well-functioning of the application provided. Developers and operators have to carefully craft their applications to correctly work with each CSP-specific set of demanded non-functional services.

Second, multi-cloud architectures require an extra effort to be adopted. In fact, the Application Programming Interface (API) of a new cloud framework potentially breaks legacy compatibility with existing services when they integrate within the cloud ecosystem. Consequently, it is not only difficult to cope with interoperability but also to have an optimized use of multiple CSPs. This means that the cloud consumer has to identify the optimized set of most compelling CSPs for a specific use-case (e.g. tele-medicine) and then propose the most suited implementation of application components using the CSPs facilities.

Third, the problem of loss of control observable in public CSPs compared to private setting has to be handled in a multi-CSP environment. In particular, the cloud consumer may want the same security tools in public CSPs than those provided in the private settings, in order to protect the usage of the end-user application. Due to technical constraints that we detail in the next sections, it is not straightforward to do so.

To sum up, despite healthcare organization can really benefit from the cloud, its adoption requires carefully designed infrastructures and applications to satisfy their constraints in terms of security and business dependency. It is worth noting that this class of problem does not impact only the healthcare domain. Many other domains have similar constraints but those are normally less strict since they may not involve sensitive data.

In the next section, we present and compare different architectures studied to realize a multi-cloud, highlighting their advantages and their drawbacks.
1.2 Multi-Cloud Scenario

Interoperability is the capacity for the cloud consumer to interact seamlessly with different CSPs, using the same method and semantic to provision resources, being “understood” by the different CSP orchestration engines. This represents the first main obstacle that prevents a straightforward interconnection among different CSPs. Interoperability is not given since providers use different and incompatible technologies and application programming interfaces (API) to instantiate and deliver resources to their cloud customers (e.g. IT department of a given organization).

To cope with interoperability issues, two main architectures are proposed so far: provider-oriented architectures and client-oriented architectures. First, provider-oriented architectures allow a set of cooperating CSPs to present themselves to cloud consumers as a single CSP instance. CSPs may achieve this by federating their resources and by leveraging common standards in resource virtualization and APIs. The main advantage of provider-oriented architectures is the seamless and effortless integration of resources w.r.t. the cloud consumer. However, the adoption of provider-oriented approaches faces several obstacles. CSPs are normally competitors and often are not interested in cooperating [85]. In addition, the technological choices to build their own infrastructure made by each CSP (e.g. the choice of the hypervisor) may impact negatively interoperability. Finally, CSPs do not allow consumers to have a deep infrastructure control similar to what they experience on their premises [5].

Second, client-oriented architectures provide cloud consumers a mechanism to allocate resources over multiple CSPs without requiring any cooperation among the CSPs themselves. In this scenario, the cloud consumers have the burden to overcome CSP interoperability limitations. Proposed architectures introduce dedicated multi-cloud libraries [49, 62] or rely on third parties, such as brokers [18, 102]. More precisely, multi-cloud libraries (MCLs) [24, 57, 62, 82] have gained popularity since they allow developers to build and maintain their multi-cloud alone, without a third-party. Such tools are capable of dealing, respectively, with resource provisioning and service configuration on multiple CSPs.

We consider that MCL frameworks open the door to a maintainable and reproducible client-oriented multi-cloud, despite those tools present several important limitations.

In fact, most of the available MCLs leverage a paradigm called Infrastructure as Code (IaC) [59] to describe hardware resources as code, similarly as commonly done with software. This enables the possibility to adopt the techniques normally used for software development (e.g. versioning, automatic testing, continuous delivery), for the infrastructure description. The IaC approach enables the possibility to have the development environment the most similar to the production one, since the infrastructure resources are allocated in the overall pipeline. IaC applied to MCL reduces the time needed to redeploy a service and the maintenance costs.

The model of infrastructure that should be built by such "IaC-oriented MCLs" is an homogeneous view of different CSPs consisting in an "overlay" virtual infrastructure. Proposed by several works [6, 48, 74, 96], an overlay virtual infrastructure introduces an always applicable homogeneity layer that decouples CSP API from the cloud consumer by partially re-implementing the services offered by the underlying CSP. By doing so, the multi-cloud may be reproduced in equivalent instances on each CSP, exposing to the developers exactly the same configuration on every different provider.
Such overlay virtual infrastructure properly fits the above-mentioned use-case. CDOs developers can simply take a cartography of their "private cloud" settings and then construct an overlay "virtual infrastructure" instantiating exactly the same services on each different CSP, in order to not disrupt compatibility with their applications.

However, writing the code for this layer is not a straightforward task and has a non-negligible cost to be afforded by the cloud consumer. As mentioned above, overlay virtual infrastructures are composed of a set of functional and non-functional services. Non-functional services are generally the same on every different implementation of the overlay virtual infrastructure. The cloud consumer would like to share non-functional service code across different applications but a reality checks shows that in practice it is a complex task. The separation between functional and non-functional code is often blurred and, therefore, non-functional code is hard to be shared across them.

Moreover, the cloud consumer would like to reuse the CSP-agnostic IaC code service description among different applications, obtaining the most suitable implementation of that code depending on the workload peculiarities. In a nutshell, cloud consumer does not want to tailor IaC code for one or another CSP feature. He wants to change the pool of CSPs parts of the multi-cloud without having to carefully rewrite services code. Currently used IaC-MCL paradigms are not capable to flexibly handle that. Consequently, the consumer is de facto locked-in the group of CSPs he adopted at the beginning of the development. Even if he was able to go over single provider limitations, he is now stuck in a similar situation with a group of providers.

To sum up, after a decade of cloud computing existence, provider-centric architectures remains viable only in limited scenarios. The huge assumption of cooperating CSPs does not completely hold [42, 85] because CSPs are natural competitors, not willing to cooperate. Software houses building cloud management systems and public CSPs have shown interest to cooperate in order to create a seamless environment w.r.t. the cloud consumer point of view [15]. However, hybrid clouds have a limited scope, since they do not involve more than one public CSP and only one private cloud.

We consider the combination of IaC-based MCL and overlay virtual infrastructure as a promising solution for multi-cloud. However, if the initial "interoperability" issues are currently addressed, the limitations in terms of extensibility and specialization in infrastructure definition and the lack of control still stand. Those problems are analysed in depth in the following sections.

1.3 Problem Statement

As aforementioned, three key limitations of the MCL-based client-oriented multi-cloud architectures exist. First, the lack of service extensibility when dealing with the IaC paradigm for a multi-cloud deployment, discussed in Section 1.3.1. Second, the lack of specialization when utilizing CSP resources and services, discussed in Section 1.3.2. Third, the loss of control in the context of multiple providers Section 1.3.3.

1.3.1 Lack of extensibility

IaC-based on overlay multi-cloud, introduced in Section 1.2, provides a solution for the aforementioned interoperability problem. In fact, the declarative and CSP-agnostic code,
proper to most IaC frameworks, simplifies a lot the integration and the deployment of functional and non-functional services.

However, the IaC adoption is still not an ideal solution for the creation of a virtual infrastructure since this involves a huge amount of code to be developed and maintained. More precisely, enlarging the code-base introduces non trivial extra-costs, because each CSP represents a different environment where to debug, test and deploy the CSP-agnostic code.

As already presented before, we can consider that each customer has to rely on its applications and a set of non-functional services, which cover auxiliary features (e.g; monitoring, backup, debug). The code of such non-functional services should be reused as much as possible across different customers. This would also benefit to our use-case for a more agile and scalable outsource of EHRs to a multi-cloud based storage system. However, the capacity of handling the lifecycle of such multi-cloud application is not trivial. In fact, non-functional services are normally scattered across several infrastructure elements (e.g. Network topology, OS configuration). For example in case of monitoring, we may have multi-layer agents inside user VMs and alerts managers as standalone [116]. This blurred "border" violates the traditional "module" or "package" driven service composition, requiring a more sophisticated integration of components. Consequently, it is not possible to easily reuse non-functional code across different applications. For instance, a monitoring framework [92, 125] may necessitate not only to deploy a standalone central monitoring service but also to distribute agents on each monitored VM. Thus, assuming that IaC is used as mentioned above, the code of a monitoring service cannot be easily inserted or removed from the infrastructure template.

Therefore, each possible combination/configuration of non-functional services has to be manually realized and tested. In the scenario of multi-clouds, this process has to be repeated for each CSP involved in the infrastructure, without any scaling in terms of investment for the cloud consumer.

Second, after deployment, infrastructure services have to be maintained by continuously checking integrity and availability. Run-time updates of the infrastructure in case of an unexpected event cannot be easily handled dynamically (for instance, reacting to the suspicious rise of incoming network traffic, deploying security services to protect a victim service). In the absence of an extensibility support from the IaC framework, such run-time manipulations have to be manually performed on the IaC code, impacting the capacity for the cloud consumer to quickly respond to that specific event.

To sum up, in a multi-cloud, the codebase and the effort required to maintain services becomes rapidly intractable and prevents a multi-cloud consumer to experience the same flexibility as in a single-cloud ecosystem.

1.3.2 **Lack of Specialization**

Multi-cloud infrastructures come to the field at the price of the loss of specialization in the cloud resources definition. In fact, most popular approaches, such as MCLs [49, 57, 62], enable multi-CSPs resources allocation in a provider-agnostic fashion. This means that the resources are deployed in the same generic manner without taking into account the specificities of each CSP. Indeed, MCLs rely on a single formalism to express "least common denominator" [42] resource definition. This leads to the two next limitations.
First, the most straightforward limitation is about specialization. The cloud consumers cannot express their requirements in terms of services and resources from the CSP and, consequently, cannot obtain the most suitable implementation for the given applications. For instance, some users may not be able to dynamically benefit from specific needed services such as on “hardware accelerators” (e.g. GPU, FPGA) or CSP-managed services (i.e. DBMS as a service). The only way to get rid of this limitation is to manually set up a specific implementation available on a certain CSP [57] that the consumers wants to adopt. This would result in a different infrastructure code specification for each underlying CSP and would not scale the amount of work to be done to set up a multi-cloud infrastructure.

Second, the cloud consumers cannot describe their desired IaC infrastructure without an in-depth knowledge of CSP-specific ecosystems while building their multi-cloud architecture. In fact, CSPs ecosystems are growing in complexity and in number of exposed services. Every year, public CSPs and even open-source cloud management systems introduce new services to intercept new use-cases. Furthermore, CSPs are today proposing multiple region/data-centers where to deploy customer workloads, which may differ a lot in terms of hardware configurations and the available services. Therefore, the cloud consumers have to know each specific region of each provider in order to correctly choose the one that best meets their requirements.

Consequently, this pushes the necessity of dedicated investment in order to embrace what each ecosystem propose, requiring dedicated training from the consumer side. Therefore, the cost of adopting multi-cloud requires to limit the targeted CSP. Such approach makes multi-cloud not affordable for many organizations and represents an important barrier to its initial adoption.

To sum up, building functioning intercloud infrastructures through MCL requires not only an extra effort in terms of CSP insight but also, even when that is achieved, it is difficult to optimize service w.r.t. the variety of features in different cloud ecosystems and the desired user usage.

1.3.3 Lack of control on public CSPs

Despite being the most promising solution for multi-cloud, client-centric architecture-based overlaying virtual infrastructures come with unsolved problems. In the following, we individuate and define for the first time such roadblocks w.r.t. the above mentioned technology.

At first, we consider the problem of "loss of control" in public that exists both in traditional clouds and multi-cloud architectures.

Indeed, since the beginning of the cloud era, the adoption of public CSPs had important consequences not only in terms of security and control over data but also in terms of loss of control over the infrastructure. Compared to private infrastructures, the loss of control may not only concern the physical possession of data but also the loss of control on "lower" system layers of the software stack, notably the virtualization layer or hypervisor\(^1\). Such "loss of control" is a straightforward consequence of multiplexing the same physical hardware among multiple untrusted customers. In traditional public cloud architectures, such loss of control is the price to pay to ensure isolation among tenants in

\(^1\) The hypervisor is a component responsible to create and manage virtual machines. It is normally implemented in software and executed at the highest level of privilege.
monolithic virtualization layers. More precisely, as reported in [121], the possibility to control those layers give developers and operators the possibility to use an equivalent mechanism (e.g. introspection) to protect their applications. Many approaches were conceived and prototyped in literature [7, 108, 114] to re-enable cloud consumers to control the infrastructure without impacting security and cross-tenant isolation. However, none of them was conceived for a multi-cloud environment. The major issue is the inability of the virtualization layer to preserve cross-provider interoperability in the presence of an enhanced control.

However, such architectures did not consider at all the possibility of multi-CSP interconnection similarly as traditional virtualization layers, as we briefly discussed in Section 1.2. However, if we consider that the consumer now controls the virtualization infrastructures, in a multi-cloud, he cannot (1) flexibly control and program such virtualization layer through IaC seamlessly (e.g. to add a security appliance in the virtualization layer) and (2) preserve legacy compatibility with existing tools and applications (e.g. avoid to redefine virtual infrastructure template to match different virtualization architectures).

To sum up, analyzing related works, it remains still unclear how multi-cloud architecture can deal with the problems pinpointed above. New modular architectures breaks the compatibility with legacy tools and the consumer cannot use IaC to model a customized distributed provider-agnostic overlay virtualization layer. Furthermore, it remains difficult for the consumer to have an equivalent level of control when moving to a cloud remote infrastructure.

1.4 RESEARCH OBJECTIVES AND CONTRIBUTIONS

In this section, we present the research objectives of this manuscript, whose aim is to mitigate of the limitations of client-centric approaches for the multi-cloud architecture presented in Section 1.3.

In a nutshell, the research objectives of this manuscript ultimately aim at improving flexibility of the building process of multi-clouds and the supervision of their life-cycle, with a particular focus on security perspective. In particular, we investigated the possibility to rely on an consumer-oriented optimized construction capable to guarantee an equivalent level of control of private infrastructure and to scale in terms of number of supported providers and applications.

Despite the fact that overlay "virtual infrastructure" [6, 48, 74, 96] provides a first answer to the interoperability problem, such approach requires a lot of efforts in terms of code writing and maintenance and scales very hardly to multiple CSPs.

Starting from this generic yet tailored description, the multi-cloud builder should be then capable of constructing the most suitable multi-cloud to the targeted user through a considered CSPs selection and a compelling IaC infrastructure description. We envision a multi-cloud definition completely transparent w.r.t. the CSP selection. In such scenario, the cloud consumer can specify a generic virtual infrastructure template with a proposed usage optimization criteria and a list of non-functional services to be integrated into its functional applications.

The multi-cloud construction requires the realization of a builder capable to extend the capabilities of currently IaC paradigm to express an agnostic and completely decoupled service definition. As a multi-cloud "compiler" leveraging a context-aware approach,
the builder has to conciliate all research objectives presented so far: (1) non-functional extensibility of basic templates, (2) capacity to adapt to provider features, and (3) improved level of control over in a multi-CSP scenario. Following the definition of context given by Dey [39], we consider part of the context all pieces of information that "can be used to characterise the situation of a participant in an interaction". In the multi-cloud scenario, those pieces are represented by (1) the consumer requirements or the consumer context, (2) the CSP available services or Provider context and (3) the knowledge on how a particular service can meet a consumer requirement, the "broker" context.

In the next sections, we detail the three research objectives of this manuscript and highlight the respective contribution for each objective.

1.4.1 Multi-cloud infrastructure extensibility

The first research objective of this manuscript is the introduction of the extensibility property in IaC-based service modeling. As already discussed in Section 1.3.1, the lack of extensibility prevents IaC frameworks from completely addressing multi-cloud challenges in terms of (1) integration and deployment of non-functional services and (2) runtime modification on deployed multi-cloud infrastructures.

Extensibility facilitates the injection and eviction of non-functional services. In fact, this would enable the possibility of code re-using across different consumers (e.g., their different multi-cloud infrastructure mentioned in Section 1.4.3). In fact, a mechanism to dynamically plug the non-functional services to the rest of the infrastructure would avoid an ad-hoc and manual configuration of those for each consumer. The consumer would only select the required non-functional additions to trigger the "enrichment" of its base infrastructure.

So far, realizable solutions for an easy injection and eviction of non-functional services in the multi-cloud scenario are not present to our knowledge and only few basic static analysis tools have been proposed to early detect issues on infrastructure templates.

The first contribution of this manuscript is to propose a mechanism that enables injection/eviction of non-functional services in the multi-cloud scenario. This enables the possibility of code re-using across different consumers (e.g., their different multi-cloud infrastructures) and static analysis of infrastructure templates and mitigate the lack of extensibility of code for non-functional services across different CSPs. We propose to extend the IaC-based multi-cloud deployment and management with aspect-oriented approach [66] concepts to obtain an extensible virtual infrastructure definition. More precisely, we first identify an Aspect-oriented approach to cope with this problem. We define a TML (TOSCA Manipulation Language) aspect specification language to dynamically inject "non-functional" services to the virtual multi-cloud infrastructure. TML leverages the OASIS TOSCA [82] template language. The OASIS TOSCA [82] format provides interoperability between templates of cloud applications by leveraging an object-oriented resources modelling and the "Infrastructure as Code" paradigm. TML analyzes TOSCA graph of resources looking for pattern which can be trigger graph modifications.

Second, we introduce the possibility for the cloud consumer to describe its requirements in terms of non-functional services developing TML scripts to describe reusable non-functional services. Such TML code describes how extra TOSCA services have to be added to the functional IaC TOSCA code of her application through aspect-oriented
programming. In a nutshell, the TML represents the "glue" among the application code and the reusable non-functional components. The TML-based weaver represents the first functional component of the multi-cloud compiler/builder.

1.4.2 Multi-cloud infrastructure specialization

As discussed in Section 1.3.2, building well functioning multi-cloud infrastructures through MCLs requires not only an extra effort in terms of IaC extensibility to gain scalability but also in terms of optimization w.r.t. to reach the best level of suitability and efficacy. In fact, “least-common” denominator in multi-cloud architectures always implies less specialization and a consequent less efficient allocation of the resources. This finally leads to a loss in the overall performance and cost optimization.

The research objective is to introduce a mechanism that allows to select the set of CSP that best fits the consumer needs and that exploits CSP peculiarities. We achieve this through the following two steps.

First, we introduce a “context-based matching” scheduling algorithm to select the most compelling set of CSPs according to the consumer needs. Such algorithm takes as input the list of available CSPs and their data-centers and a set of consumer-provided parameters (e.g. the amount of distinct CSPs, the service template, the optimization criteria), producing as output the list of compelling data-centers belonging to different CSPs. The context is managed by extending the TOSCA formalism. Such extension consists of encapsulating inside the TOSCA format the matching parameters between CSP features, consumer preferences and implementation limitations. The algorithm, considering the features of each single data-centers w.r.t. the service desired, is capable to select the most interesting data-centers by matching their feature capabilities and hardware configuration to existing IaC service implementations.

Second, we define an end-to-end workflow to optimize a multi-cloud infrastructure definition. To do so, the consumer initially models the IaC code as an high-level graph of services, leveraging the combination of TML and context-based matching adoption. The output of this workflow is the instantiation of such optimized and fully-featured multi-cloud on most adapted CSPs. Nevertheless, such workflow can also be used for a dynamic reconfiguration of the multi-cloud, upfront the occurrence of an unexpected event (e.g. a CSP service is not available).

1.4.3 Building and deploying a multi-cloud architecture

The third research objective we envision is to build and deploy a complete multi-cloud leveraging an overlay virtual infrastructure. Starting from a generic overlay virtual infrastructure and several parameters to characterize the workload, the multi-cloud builder should be capable to build a specialized implementation for selected CSPs with the integration of the desired non-functional services. To do so, we design and implement the MANTUS multi-cloud compiler, which combines the extensibility and specialization features introduced above.

Complementary to components described and presented in the previous chapter, we introduce the following three elements.

First, we define and implement an overlay virtual infrastructure template, ORBITS. ORBITS definition groups together services to (1) orchestrate workloads and conse-
quent resource allocation across the multi-cloud and (2) provide an adapted virtualization architecture, decoupled from the technological choice of the underlying CSPs. More precisely, ORBITS proposes a multi-cloud orchestration logic capable to scale over multi-cloud.

Second, we design and implement a virtualization architecture which provide more control to the user as mentioned in Section 1.3.3, still providing a compatibility layer but being capable of leveraging enhanced features. Our prototype, based on micro-kernels and nested virtualization, enables the user to have more control on lower layers of the infrastructure (e.g. the hypervisor), assuring strong isolation w.r.t other tenants and the compatibility with the rest of multi-cloud.

Third, we present how dynamic update to the multi-cloud infrastructure can be tackled by the framework by the combination of MANTUS and ORBITS, respectively the multi-cloud compiler and the overlay infrastructure. In particular, we show how those updates can be inserted in autonomic loops in order to lower the effort required to maintain a multi-cloud architecture.

1.5 Multi-cloud Deployment Step by Step

The multi-cloud brings important advantages to cloud consumers by breaking their business dependency to a single CSP and introducing more flexibility to geographic distributions of workloads. We have already identified that solutions based on MCLs are interesting to overcome the interoperability issues introduced by using multiple CSPs at the same time. However, MCLs have still several limitations in terms of extensibility, specialization and lack of control. In Section A.4, we highlighted the contribution of this manuscript to tackle the problems identified in Section 1.3.2.

In this section, we map those contributions in the multi-cloud construction process. Our discussion focuses on how the contributions listed in Section A.4 may improve crucial steps of this process. Moreover, this discussion takes the perspective of a cloud consumer who is familiar with the process of deploying a single CSP, but who has never dealt with a multi-cloud. In particular, we address the concerns and doubts that a cloud consumer might have when approaching for the first time a multi-cloud architecture.

The first question a cloud consumer may ask is how to choose the right CSPs for a multi-cloud infrastructure, given the final application to be built. CSPs are international companies whose physical presence is not bounded by the geographical borders of a single nation. CSPs span across multiple countries and continents through their data-centers, which are in turn subjected to the laws of the hosting country. This is mainly due to the legal and availability constraints that we presented above (Section 1.1.1). In particular, if we consider health-care and other use-cases which process sensitive data, consumers want to filter out locations that cannot legally be used (e.g. data-centers outside a country) and rely on a sufficient number of distinct providers.

The second question is directly related to the first one and concerns the selection of the compelling services offered by the targeted CSPs. In fact, inside a single CSP ecosystem, there are multiple possible approaches that may be suitable to execute a specific workload. In other words, different options have different costs and technological trade-off in terms of maintenance efforts and customization, and such choice is simply up to the cloud consumer. For example, if we consider the data analytics, several CSPs offer managed versions of popular analysis frameworks (e.g. Hadoop) which requires minor
effort to the customer to be deployed. However, if the customers want to maintain the complete control on their software stack, they may leverage the traditional IaaS-based approach, where it is up to them to install, maintain and scale its cluster. This step is critical because it impacts the way the multi-cloud is built but it is not easy to be addressed since CSPs ecosystems are evolving continuously.

The third and last question that a cloud consumer may ask regards the day-by-day life-cycle management of a multi-cloud architecture. In fact, the adoption of multi-cloud increases a lot the different infrastructures to monitor and manage. Obviously, this can require a very important extra effort for cloud consumers. In fact, each single CSP provides the possibility to have a single-point of management and orchestration, for example through web dashboards or software libraries. This is very useful for consumers since they can rely on a privileged point of policy observation and enforcement. However, multi-cloud infrastructure currently do not provide this single point of orchestration, as each CSP involved is independent of each other. Therefore, the adoption of multi-cloud should not imply the loss of such unique point of orchestration which is a key to have a smooth transition from a single provider to a multiple configuration. The consumer should be capable to flexibly manages services and resources over multiple CSPs and should be handled trying to automate at most those processes. In a nutshell, the multi-cloud administration should not be more complicated than the single-cloud administration. This question can be tackled by proposing an overlay infrastructure, which introduces the same API over all different CSPs masking to the consumer the complexity and the heterogeneity of having different underlying technologies.

In the next section, we present the template of a multi-cloud deployment workflow, analyzing how it can answer to those consumer questions, and how the thesis contributions helps in those directions.

1.5.1 Multi-cloud deployment workflow

Inspired by the TOSCA mode of functioning [36, 82], we sketch the main steps of a multi-cloud deployment relying on IaC-based MCL.

In Figure 1.3, we present on the left the three main steps of a multi-cloud deployment. For the moment, we do not focus on which component is responsible to perform the requested steps but only on what they do and which transformation they apply to the input.

First, the cloud consumer introduces its infrastructure template as input to the provider selection step. Those steps considers not only what infrastructure services the consumer is willing to perform but also which are the technological (e.g. non-functional services) and non-technological constraints (e.g. geo-location regulations) are mandatory. For example, if we take the data analytics example presented in Section 1.1.1, the consumer template can specify that he wants to deploy a MapReduce cluster of a certain size. In these early steps, it has not to specify as input the technology, but only an high-level abstraction of the desired output (e.g. Hadoop).

Taking into account the consumer inputs, the list of CSPs and the list of CSPs information has to be leveraged to obtain a refined definition of the service the consumer want to deploy. This step is presented in Figure 1.3 as "Provider Section". The key objective of this step is not to obtain as output an "infrastructure code" that can be straightforward deployed over different providers, but to match the abstract service definition introduced
Figure 1.3: Multi-cloud workflow and manuscript contribution mapping
by the consumer with compatible implementations available on selected CSPs. Continuing the simple example of data analytics we presented before, the expected output of this step is for example a "concrete" template where the abstract "MapReduce" framework is implemented through CSPs managed services or, leveraging the least-common denominator, with several VMs and the compelling software configuration.

Third, this service implementation has to be translated and pushed on different CSPs to be instantiated. In this step, the translation should benefit from the information about a specific CSP API and the modelization we designed for the provider context. An example of what information can be added is represented by the OS images or flavors available to instantiate VMs in a specific CSP zone. Each CSP treats such information in a different way, and, moreover, the choice of the flavor can impact a lot on final performance and should keep into account consumer requirements.

The resulting multi-cloud is constructed through a virtual overlay infrastructure which is presented to consumer applications with homogeneous APIs, which hides the complexity of the different underlying providers.

In next section, we present how those steps are tackled by the manuscript contributions and the thesis plan.
In the remaining, we present the structure of the manuscript with a precision of the contents of next chapters, mapping presented contributions as showed in Figure 1.4:

**Chapter 2** introduces cloud computing and multi-cloud fundamentals. It presents different approaches to interconnect multiple CSP, by focusing in particular IaC-based client-oriented approaches. In this context, we detail in particular the problem of extensibility, specialization and loss of control.

**Chapter 3** present the AOP-based weaving to address the problem of IaC-based multi-cloud extensibility. First, we detail the context, highlighting existing approaches and current limitations. Second, we present Aspect-Oriented Programming and the roadblocks to adopt it in IaC infrastructures. Third, we introduce the Tosca Manipulation Language (TML) showing how it can introduce extensibility to multi-cloud infrastructures.

**Chapter 4** provides the contribution of "context-based matching" to mitigate the specialization problem in IaC based infrastructures. We propose to leverage such technology to make an optimized provider selection w.r.t. user constraints and to obtain a specialized implementation of multi-cloud services on selected CSPs.

**Chapter 5** presents the deployment of an end-to-end deployment of a IaC-based multi-cloud, considering the lack of control in CSPs. We detail in particular the model of multi-cloud services deployed, ORBITS, and the possibility to introduce a dynamic reconfiguration in the multi-cloud life-cycle.

**Chapter 6** present the architecture of U-Cloud which addresses the lack of control in CSPs. We detail the architecture model and its concrete implementation in a prototype. In addition, we present an example of the behavior of the U-cloud showing several workflows.

**Chapter 7** present an implementation of the proposed contributions, and corresponding experimental validations. First, we detail the implementation of a multi-cloud builder MANTUS. In particular, we present performance and scalability evaluation to show the effectiveness of each different step. Furthermore, we analyze the multi-cloud infrastructure prototype, ORBITS and the U-Cloud node.

**Chapter 8** details conclusions and future work. We recap key contributions, highlighting limitations and envisioned next steps to extensible and specialized IaC-based multi-cloud.
The right understanding of any matter and a misunderstanding of the same matter do not wholly exclude each other.

Franz Kafka, The Trial

In this chapter, we introduce Cloud Computing and discuss its core properties and features (Sec 2.1). Afterwards, we focus on the multi-cloud infrastructure and, in particular, on client-centric architectures.

2.1 INTRODUCTION

Cloud Computing is a distributed computation paradigm that gained a lot of popularity in the last decade. The key concept behind it is to decouple physical location of resources from their usage location [13]. Firstly proposed by Amazon in 2006 [9] as a commercial offer, Cloud Computing represents the most adopted approach to deploy IT services. In 2011, NIST (National Institute of Standards and Technology) proposed the following definition of Cloud Computing [78]:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

NIST identified five “essential characteristics” describing the Cloud Computing use model:

1. **On-demand self-service.** The capacity of the consumer to “unilaterally provision computing capabilities, such as server time and network storage”.

2. **Broad network access.** Resources availability has to be provided over the network.

3. **Resource pooling.** Multiple consumers multiplex hardware and software resources in a multi-tenant model.

4. **Rapid elasticity.** Resources can be elastically provisioned and released.

5. **Measured service.** User may leverage a "pay-for-use" metering model without an up-front commitment.
2.2 BACKGROUND

In this section, we describe what are the main components of the Cloud Computing and how they are organized. First, we provide a taxonomy of cloud computing services in Section 2.2.1. Second, we present the limitations of cloud computing when it comes to applications with stringent requirements both in terms of availability and privacy in Section 2.2.2.

2.2.1 Taxonomy

In the following, we provide an overview of the different models for cloud deployment and services (Section 2.2.1.1), discuss the key enabling technologies which backs those services (Section 2.2.1.2), and present how virtualization let untrusted users leverage the same physical machine (Section 2.2.1.3).

2.2.1.1 Service/Deployment models

Cloud computing services can be easily classified through the type of interface exposed to customers or through the type of targeted consumers. First, NIST identified three main service models, in which consumers can leverage cloud computing.

The first service model is “Infrastructure as a Service” (IaaS). IaaS consists in providing remote access to “hardware”-level resources, exposed through low-level abstractions (e.g.; Virtual Machines (VMs), Object Storage). The first service proposing such class of service was Amazon EC2 in 2006, followed by many other providers. Instead of buying physical machines, the consumers purchase time “virtual” machines, with specific resource capacity. Furthermore, consumers are charged for network traffic on a pay-per-use basis.

The second service model is “Platform as a Service” (PaaS). The consumers leverage a development platform which provides access to resources with high-level programming interfaces (e.g. DBMS, messaging queue, data analytics, DNS). PaaS consumers are typically developers and operators. The logic behind those service is to enable fast provisioning for common components of applications such as DBMS API. In particular, security, scaling, availability and disaster recovery are major concern when dealing with applications. Using PaaS, with low adaption work, cloud consumers can reduce the time and the effort to have an application in production. Despite its positive aspects, PaaS normally offers lower customization than IaaS and has therefore see lower adoption in the first decade of cloud computing [120]. Popular PaaS are Google App Engine, Salesforce.com and OpenShift, an open-source platform.

The third service model is “Software as a Service” (SaaS). Such model provides end-users with the possibility to consume application remote services through an API or a lightweight/mobile application. SaaS is normally the one intended by end-users when referring to the “cloud” and it is by far the most popular. The key principle is that the application is remotely executed and operated. The end-user can consume the service unaware of which provider is executing the underlying application or how many servers may be involved in providing it. In fact, the SaaS consumer does not have to have any technical expertises in IT. One of the most popular applications is file-based storage,
where the end-user can have a folder of its disk transparently synchronized through different devices and remotely backed up.

In addition, the aforementioned models do not exclude each other and can, instead, be combined together to build the same application (e.g. have a PaaS offering leveraging its own IaaS). Traditionally, consumers like developers and operators are the commercial target of IaaS and PaaS offerings, while SaaS may interest also end-users.

Another important distinction to be made is in terms of the deployment models, considering the property of the resource provisioned to cloud consumers and end-users. Public Cloud [13] consists in a deployment model where any cloud consumer who subscribed to the CSP realm may provision its desired amount of resources. CSP shares the same hardware resources among untrusted cloud consumers. The CSP leverages several virtualization techniques to multiplex hardware resources and isolate user execution. There is no restriction in this model on who can subscribe to the service and provision resources. A mitigated form of Public Cloud is represented by "Community Clouds", where users are members of a selected group of people (e.g. a consortium of organizations).

Private Cloud [78] is a completely different deployment model. Cloud computing has become popular also among single organizations to centralize the resources provisioning within the organizations themselves. Reconfiguring the internal resources to a cloud-oriented architecture means that organizations can reduce resource management costs by introducing flexible and elastic provisioning, while still keeping all their applications inside their data-centers.

However, private cloud traditionally suffers for the limited amount of resources that are available and requires to be seamlessly extended through integration with one or more public clouds outside the organization.

In the context of this manuscript, we will consider the IaaS model when not explicitly specified. In fact, on the one hand, the SaaS targets mainly end-users because it provides already ready-to-use applications. On the other hand, PaaS does not attract cloud consumers because of these above-mentioned defaults [52, 120]. In contrast, IaaS is a generic and extremely popular approach among cloud consumers because it enables the maximum level of customization in terms of software stack. Furthermore, it can represent a smooth transition between legacy on-premise and CSP infrastructures.

### 2.2.1.2 IaaS Cloud Enabling Technologies

Secure multiplexing of infrastructure among untrusted cloud consumers represents a mandatory feature of Cloud Computing, especially for public CSPs. The key point which makes cloud computing economically interesting for providers is to share the same hardware resources among different untrusted customers. Since normally every customer is not requiring all the allocated resources at the same time (e.g. by not completely using a virtual processor), the provider can overcommit hardware resources presenting to different customer more "virtual resources" than the physical one actually available. In fact, the problem of sharing hardware resources between different entities (e.g. Users, Operating Systems, Applications) has interested researchers for several decades, where computers were rare and expensive. Focusing on IaaS, three main resources should be shared across different consumers: computation, storage and networking. For each
of them, we analyze the key technologies that allow user to securely share the same hardware resources.

Concerning networking, a vast amount of works were proposed in literature to allow the transparent multiplexing of network resources [67]. In particular, Software Defined Networks (SDN) introduced the possibility to "program" the network configuration by leveraging (1) a centralized control plane and (2) programmable network components (e.g. Hardware Switches/Routers). Leveraging SDN principles, network Virtualization is "by the ability to create logical, virtual networks that are decoupled from the underlying network hardware" [119]. Network virtualization leverages Software-Defined networking (SDN) and Network Function Virtualization (NFV) to define "virtual networks" and other network elements (e.g. Virtual Routers, Firewalls). More precisely, network virtualization enables for example the possibility to create a completely customized network topology (i.e. virtual network) modeled through virtual network elements (e.g. virtual routers).

Storage virtualization is a technique to abstract information about storage hardware in order to integrate hardware resources in a single view. Two main classes of abstraction, block and object, are normally leveraged to propose cloud storage services. In particular, object storage really represents a disruptive technology in the cloud computing era. The generic abstraction of an "object" fits perfectly well the requirements in terms of horizontal scalability of cloud providers which is not satisfied by block storage. Instead of having a single block (e.g a disk partition) which can not scale very flexibly (e.g. concurrent access from multiple instances), the generic abstraction of "object" storage can provide scalability and flexibility. Application can simply push and pull "objects" with simple API calls.

Concerning computation, system virtualization [90] allows the user to completely configure virtual machines independently from the underlying configuration of the system. In the next section, we provide an in-depth analysis of system virtualization which has a key role in assuring customer isolation and security on a remote cloud infrastructure since it is in charge of let untrusted consumer share the same hardware.

2.2.1.3 System Virtualization

System virtualization is the key enabling technology of cloud computing. More precisely, system virtualization isolates as much as possible the execution of processes and applications run by different and untrusted cloud consumers. This is achieved by limiting their resource consumption as well as the access to other application data or code. More concretely, an application is only able to access its own memory space, have a limited number of CPU cycles or transfer only a certain quota of network traffic in a certain amount of time. There exist three main approaches, based on software and hardware, to achieve system virtualization, which are discussed in the following.

First, the so called "full-virtualization" is the most adopted technique to obtain system virtualization. This system model is derived from the formal definition of virtualization provided by Popek&Golberg in [90]. They introduced the concept of virtual machine monitor (VMM), also known as hypervisor. The VMM concretely provides the abstraction of a "virtual machine", which is considered by the operating system exactly as bare metal as shown in Figure 2.1. The authors specified three fundamental properties of the abstraction layer that has to be provided by VMM to virtual machines:
1. **Equivalence/Fidelity**: a program running under the VMM should exhibit a behaviour essentially identical to the one demonstrated when running directly on an equivalent machine;

2. **Resource control / Safety**: the VMM must be in complete control of the virtualized resources;

3. **Efficiency / Performance**: a statistically dominant fraction of machine instructions must be executed without the VMM intervention.

Considering hypervisors and full-virtualization, we observe that they expose a low-level interface equivalent to bare metal, enabling the possibility to transparently multiplex hardware across different hypervisors. The protection proposed by full virtualization is based on hardware support (i.e. on x86, Intel VT-x or AMD SVM), but requires the implementation of resource multiplexing features (e.g. scheduling). Therefore, the architecture of most common hypervisor is similar to the one of monolithic OS, suffering from the same weaknesses. In particular, their Trusted Compute Base (TCB) is huge and may introduce many security issues [88].

Second, Operating System-level virtualization, also known as containerization, emerged as a paradigm to enforce system isolation by leveraging the most privileged software on a computer system: the operating system. As shown in Figure 2.2, instead of relying on hardware support like the full virtualization, separation mechanisms are implemented in the OS kernel. For instance, Linux implements resource confinement and quotas through specific functionalities called respectively namespaces and cgroups. This form of virtualization is normally considered less effective than full virtualization since it relies on a more wide attack surface (i.e. system calls), compared to the one exposed by the hypervisors (i.e. ISA sensitive instructions). However, this form of virtualization is becoming really popular since "containers" do not require to boot a full OS enabling faster application scaling. Starting from 2013, Docker [41] introduced the possibility to construct portable application containers which become a very popular approach to distribute and deploy software.

Third, to solve the TCB size issue, the idea of micro-kernels (MK) architecture was introduced, drawing inspiration from evolution in OS architecture. Such designs were widely explored in academia [73], but adopted mainly by the mobile device industry [83]. The aim of micro-kernels, as shown in 2.3, is to expel as much code as possible...
from the TCB, making the hypervisor ultra-thin: typically, around 10KLoC for the TCB, an order of magnitude smaller than traditional hypervisors [108, 114]. The whole MK architecture exposes a minor attack surface, since a significant part of services provided by the hypervisor in kernel space are now provided in user space, leaving to the MK core only the burden to correctly implement IPCs. Following this approach, the OS-/hypervisor users can retain more control over the systems, more flexibly customizing components which are normally embedded in the kernel, while not compromising the applications executed by other users. This property of MK may be useful in the context of providing to the user an additional but still limited control over a remote server. We discuss more about this property in the next section. Furthermore, several works [108, 114] proposed to implement a hybrid approach, micro-hypervisor, in order to reduce the TCB of traditional hypervisors while offering the same low level abstraction. This is achieved by a non-privileged user-space component which implements the feature of a VMM.

To sum up, in this section we provided a general presentation of cloud computing, presenting taxonomies in terms of different service, deployment models and key implementation technologies. For example, a cloud consumer can build its application components on the CSP infrastructure leveraging the abstraction provided by virtualization-provider indirections, through the IaaS offering of a public CSP. On the one hand, the big advantage is the possibility for the consumer is to rely on flexible, elastic and pay-per-use resource allocation. On the other hand, the CSP can optimize and multiplex hardware resources among different consumers, leveraging the isolation given by virtualization technologies.

2.2.2 Current limitations of Cloud Computing

Despite the positives features provided by such technologies, the centralised nature of cloud computing comes to the field with important limitations in terms of vendor lock-ins, lack of control and QoS/Geo-replication. In fact, outsourcing applications and data
to a remote CSP imposes important concerns. This section focuses on underlining and
detailing the actual limitations of cloud computing in terms of security and interoper-
ability.

2.2.3 CSP Lock-in

When outsourcing their IT division to CSP, cloud consumers become really dependent
on the chosen CSP. Moreover, technological lock-ins prevent the possibility to easily
switch from a CSP to another one. Traditionally, the presence of such lock-ins and the
consequent lack of interoperability prevent the consumer to easily migrate its VMs and
workloads from a CSP to another. Such situation becomes even worse when consumer
does not use only IaaS resources (e.g. VMs) which can rely on a "semantical" least
common denominator but leverages CSP-managed higher level services, such as DBMS
as a Service, Firewall as a service. This leads the cloud consumer to a difficult tradeoff.

On the one hand, such services may represent a proper added value for the cloud
consumers which can offload an important part of the complexity of the aforementioned
services (e.g. scaling, deployment, upgrade) to the CSP. A "managed" service exposes
dedicated APIs and relies on an ad-hoc pricing (e.g. not based on time as VM, but on
effective usage). On the other hand, the adoption of those services introduces "de facto"
lock-ins to the consumer. The APIs and the logic of "managed" services differs from
one CSP to another, increasing the potential cost of a migration to a different CSP. In
addition, similar services usually motivate users to remain inside the provider realm,
since data transfer inside the same provider region is either free or cheap.

Such situation pushes big cloud consumers to retain a part of their IT infrastructures
to remain on their premises in a private cloud, interconnected to a specific public CSP.
Such situation is infeasible for small cloud consumers (e.g. Small Companies) which
have to rely on a single specific CSP, facing important costs when they want to migrate.

2.2.3.1 Lack of control

Indeed, since the beginning of the cloud era, the adoption of public CSPs had important
consequences not only in terms of security and control over data but also in terms of
loss of control over the infrastructure. Compared to private infrastructures, the loss of
control may not only concern the physical possession of data but also the loss of control
on "lower" system layers of the software stack, notably the virtualization layer or hyper-
visor. Such "loss of control" is a straightforward consequence of multiplexing the same
physical hardware among multiple untrusted customers. In traditional public cloud ar-
chitectures, such loss of control is the price to afford to ensure isolation among tenants
in monolithic virtualization layers. In Figure 2.4, we schematize the components in a
private and public cloud architecture. In a private cloud, customers control the infras-
tructure and can decide to customize to use security services at different levels in the
software stack. Those customization can be added as security modules to the hyper-
visor [121] or as network middle-boxes implemented physically or as virtual network
functions [31]. In a public CSP, tenants only have to rely on execution environment
abstractions for computation and on a “big switch” abstraction for networking.

More precisely, as reported in [121], the possibility to control those layers give devel-
opers and operators the possibility to use an equivalent mechanism (e.g. introspection)
to protect their applications. Many approaches were conceived and prototyped in literature [7, 108, 114] to re-enable cloud consumers to control the infrastructure without impacting security and cross-tenant isolation. However, none of them was conceived for a multi-cloud environment. The major issue is the inability of the virtualization layer to preserve cross-provider interoperability in the presence of an enhanced control.

2.2.3.2 QoS and geo-location issues

According to major web applications providers [110], high QoS (low service latencies, high scalability and availability) are tightly related to the consumer satisfaction for web services. A multi-provider design enables lower latencies for service usage, adapting distribution of applications to the geographical location of the user. Moreover, the possibility of cost optimization coming from an open market of resources may be strongly reduced by data transfer costs among different providers, normally much lower or absent inside a single provider domain [8, 54].

CSP are structured in several Data centers (DCs). As previously mentioned in Chapter 1, the availability guaranteed in the CSP Service Level Agreement (SLA) is not sufficient for several application domains such as Healthcare. Consumer have therefore to rely on a multi-DCs setting. Despite the fact that CSP have multiple data-centers, application data cannot be replicated everywhere but may be subject to law restrictions in terms of allowed countries.

Such geographic placement may influence the QoS experienced by users across the globe. A multi-DC setting may be necessary if a service has to be accessible with a constrained latency from many geographical positions. Despite CSPs are continuing to open new DCs across the world, a single DC is normally not sufficient to deliver fine-grained and highly-available services.

2.3 STATE OF THE ART: INTER-CLOUD ARCHITECTURES

As we mentioned above, the cloud computing model strongly centralizes applications and data on a single entity where the consumer is locked-in, geographically and technologically, and that he/she has to completely trust. As presented in Section 1.1, cross-CSP cloud computing is necessary for some application which cannot rely on a single CSP. Aiming at this goal, many works considered the possibility to federate resources across multiple CSPs in order to overcome some of the limitations presented above.

In this section we analyze "Intercloud" approaches. First, we introduce CSP interconnection, showing the promises coming from its adoption and hurdles which undermine...
its effective exploitation. Second, we present different Intercloud architectures, comparing and contrasting different approaches. Third, we identify multi-cloud libraries (MCL) as the most interesting approach to achieve Intercloud. We detail fundamental principle behind such solution and we highlight advantages and drawbacks of different families of MCL. Finally, we detail existing limitations of currently adopted works.

2.3.1 Benefits and roadblocks of CSP interconnection

The possibility to rely on multiple CSPs may represent an interesting approach to overcome single provider limitations mentioned above. Indeed, interconnection of resources from multiple providers promises important benefits [42] compared to the single-CSP model: (1) finer-grained distribution of resources across multiple countries, which improves QoS; (2) business independence from a single CSP, optimizing expenditures through dynamic price comparisons between providers (e.g., for Spot instances). However, a reality check shows that the benefits of multi-CSP interconnections can be mitigated.

First, the QoS argument for the multi-provider cloud has been hampered by two trends. Regarding scalability, the majority of cloud providers already has multiple datacenters or distributed infrastructures, even on the same continent. Regarding high availability, most common commercial providers guarantee generally no more than “3-nines” availabilities [10, 54], which seems to be acceptable for their use-cases, as shown by recent outages. For low latency, many efforts are spent today to enable resources at the edge of the WAN inside cloud infrastructures, offering finer-grained solutions in terms of locality [24, 32, 51].

Second, multi-CSP can solve the problem of vendor lock-in, but at the price of important efforts from the consumer. In fact, multi-CSP solutions require often an increased cost to be built, deployed and maintained. Such costs encourage consumers not to rely on multi-CSP, accepting the risk of losing their business independence w.r.t. a single CSP.

Third, multi-CSP does not provide by design an effective answer for the lack of control on a public cloud platform.

Thus, the choice of some real-world applications not to invest in a multi-CSP approach means that single-cloud infrastructures may be enough for user needs. For instance, until now, single provider availability has empirically proven to be sufficient for most real-world use-cases. Despite the negatives, as we present in Section 1.1 for health-care applications, several classes of applications cannot rely on the single-cloud model. As we stated before, this is the case for privacy and resilience requirements, but also from a mere "commercial" perspective. Single CSP model creates an enormous dependency for the consumer w.r.t. CSP. This is the case for IaaS service but become more and more the case for managed services, where applications become to be conceived for a specific CSP. This strong bond can have two effects: (1) it makes harder for customer to negotiate attractive prices or discounts and (2) CSP decision to disrupt or update his services may oblige customers to adapt their application to them. Therefore, multi-cloud is really interesting for organizations which cannot trade-off above those points, such as healthcare institution as well as big companies, which cannot passively accept to introduce a strong business dependency w.r.t. another company.

In next sections, we present a detailed analysis of proposed multi-CSP architectures highlighting advantages and drawbacks for each of them. More precisely, we analyze
the different architectures proposed in literature classifying existing approaches according to the capacity to satisfy the multi-cloud interoperability, the effort required from the customer to be adopted and the effective feasibility w.r.t. technological and non-technological aspects.

2.3.2 Provider-centric architectures

In this section, we present provider-oriented inter-cloud architectures. Many works based on this approach were proposed at the end of the 2000's, before losing in popularity [2], with the exception of hybrid clouds.

From the perspective of user-control, provider-oriented approaches are normally exporting an homogeneous set of APIs to the user, abstracting provider differences and therefore making the necessity of a user-defined infrastructure less interesting. We resumed Provider-oriented approaches in table 2.1.

2.3.2.1 CSP Federation

In federation-oriented approach [28], providers mutualize their resources agreeing on a common standard to cooperate.

The resources federation enable single providers to better support peak demands or maintenance operations. This approach presents two limitations [85]: (1) providers are normally competitors, and often are not interested in cooperating; and (2) different technological choices on their infrastructure reduces dramatically interoperability among them [5].

In addition, one key reason for "CSP-federation" is represented by the possibility for a small public CSP in shortage of resources to scale up through another bigger CSP. However, if we look at market polarization (9/10 of market is controlled by the first four biggest players [52]) we can observe that even small providers are far from being overwhelmed by customer resource requirements.

Provider Federation proposition can be structured in three main classes. First, centralised-federation [55] introduce a centralised entity as the federation entry-point. For example, the RESERVOIR project [98] defines a multi-level federation where participants manage different execution environments across multiple CSPs. In particular, a central-
<table>
<thead>
<tr>
<th>Name</th>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCM[75]</td>
<td>Centralised</td>
<td>FCM introduces seamless integration of multiple federated CSPs. FCM transforms CSP-independent user requests to provider-specific resource allocations.</td>
</tr>
<tr>
<td>Reservoir [98]</td>
<td>Centralised</td>
<td>Relies on a centralised and hierarchical federation to expose users different execution environments across multiple CSP.</td>
</tr>
<tr>
<td>Contrail [30]</td>
<td>Centralised</td>
<td>Contrail establishes a multi-provider environment, enabling for customer the possibility to transparently leverage resources from multiple CSP through a provider-agnostic interface.</td>
</tr>
<tr>
<td>Intercloud [20]</td>
<td>Peer</td>
<td>Intercloud consists in a blueprint which describes concepts and architectures behind federated clouds.</td>
</tr>
<tr>
<td>OpenCirrus [14]</td>
<td>Peer</td>
<td>OpenCirrus is a federation-based cloud computing test-bed built around a partnership among companies and universities. Researchers are exposed transparently to multiple data-centers.</td>
</tr>
<tr>
<td>OpenGrid OCCIware [61]</td>
<td>CSP Standard</td>
<td>OCCI was originally designed to implement a unified remote management API for multiple IaaS cloud management systems, including autonomic scaling and monitoring. Today, the works are more focused on integration, portability, interoperability of cloud applications across CSPs.</td>
</tr>
<tr>
<td>DMTF CIMI [35]</td>
<td>CSP Standard</td>
<td>CIMI enables the users to standardize interactions between cloud environments and users to achieve interoperable cloud infrastructure management.</td>
</tr>
</tbody>
</table>
Cloud Computing: State of the Art

Figure 2.6: Example of Cloud Bursting through Hybrid Cloud. The consumer accesses private cloud resources which are spawned transparently on a public CSP.

ized "service provider" component analyze the applications requirements, identifying a set of cooperating CSPs which implements the same APIs.

Second, peer-based federations consists in the cooperation of different CSPs to expose to the user a unified view of their resources deployed over multiple providers. OpenCirrus [14] represents a company-academic partnership to federate multiple Data-centers in order to execute scientific workloads. OpenCirrus is architected with a distributed approach. After a unified authentication, a user can schedule workloads to the federated CSPs. Each DC part of the federation may decide to accept or not a specific user task.

As a further attempt, cross-CSP resource management standards (OGF OCCI and DTMF CIMI for IaaS [61], OASIS Camp for PaaS [81]) pushes CSP to adopt a common interface to access and provision their services. Even without a direct cooperation as seen in peer-based federation, consumers can allocate resources on multiple CSPs without having to deal with interoperability issues.

After more than ten years, we may observe that provider-centric federation-based architecture are struggling of being adopted. We can identify mainly two reasons. First, similarly as stated for CSP-federations architectures assuming provider cooperations, providers are not interested in adopting a common standard API to look similar to customers. Second, each CSP tries to differentiate itself from the concurrency by launching as much new cloud services as it can to intercept new trends and anticipate customers, making therefore complex to establish a "stable" inter-CSP standard.

2.3.2.2 Hybrid Cloud

A second provider-oriented approach, hybrid cloud [79, 115], normally provide a seamless extension of a private cloud, supporting bidirectional workload migration. Hybrid cloud proved to be very popular according to recent surveys in cloud utilizations. However, such multi-provider extension is limited to a single public provider. This is normally owned by the same enterprise behind the private cloud infrastructure (e.g. Microsoft Azure [79]) or a commercial partner (e.g. AWS-VMware [15]).

Initially, hybrid clouds were very interesting from the consumer perspective. Those solutions proposed a seamless burst of private resources over the public CSP premises, without requiring any further effort from the customer. However, when ecosystems of CSPs were enlarged with more auxiliary services (e.g. DBMSaaS), such seamless extension lost momentum since consumers were not capable to find the same services on private and public cloud.
2.3 STATE OF THE ART: INTER-CLOUD ARCHITECTURES

Figure 2.7: Multi-CSP Broker

Table 2.2: Broker-based Architectures

<table>
<thead>
<tr>
<th>Name</th>
<th>Architecture</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>mOSAIC [40]</td>
<td>Application Broker</td>
<td>mOSAIC provides an unified CSP-agnostic API for managing different cloud providers. It leverages a semantic engine and a common representation of resources to transparently deploy applications across multiple CSP.</td>
</tr>
<tr>
<td>RightScale [97]</td>
<td>Application Broker</td>
<td>RightScale provides multi-CSP application management, handling all different life-cycle steps.</td>
</tr>
<tr>
<td>STRATOS [87]</td>
<td>Application Broker</td>
<td>STRATOS deploys application topologies considering deployer constraints and objectives objectives</td>
</tr>
</tbody>
</table>

Moreover, hybrid cloud architectures would not fulfill completely the possibility of extending the single CSP model. Considering hybrid clouds we are limited to leverage the “CSP-Software Vendor” coupled with a specific CSP without having the flexibility to leverage more than two CSPs or change on-the-fly.

2.3.3 Client-oriented architectures

In this section, we consider client-oriented approaches for multi-cloud. The client-oriented model breaks the general limitation of absence of a standard, since the customer or a third-party is in charge of resource federation. Two main architectures are available: (1) brokering; and (2) multi-cloud libraries.

2.3.3.1 Brokering

Brokering [18, 55, 111] allows to completely off-load multi-provider orchestration, agreeing with the broker on a desired SLA and associated costs. Brokers normally introduce a semantic engine and a common representation of resources to manage transparently applications across multiple providers as showed in Figure 2.7. For example, mOSAIC [40]
provides an unified vendor-agnostic API for managing different cloud providers to form a multi-cloud.

However, as highlighted in recent surveys, brokers are struggling to be adopted. According to [42], consumers do not want to involve a third-party in their multi-cloud construction. Brokering normally brings to the field some opaque resource allocation, which is not appreciated by big consumers. As we mentioned before, they consider data and applications as important assets and they consequently want to know exactly where applications are deployed (e.g. property of used DC)

Therefore, consumers want to select their underlying CSPs, affording the technical challenge of the integration inside their organization adopting solutions that we analyze in the next sections.

2.3.3.2 Multi-cloud Libraries

Consumer-controlled approaches were proposed in literature to allow the user to completely control its "multi-cloud" as shown in Figure 2.8. Library-based (e.g.; jclouds, libcloud) approaches provide the best flexibility, supporting many different provider APIs. Those libraries are normally called "multi-cloud libraries" (MCLs) to specify the fact that they are capable of abstracting more different CSPs eco-systems. Libraries are useful to express resource-oriented workloads requirements, communicating with CSP APIs with a driver mechanism. Among the others, different approaches may implement such interoperability with different peculiarities. Those libraries represent the most adopted approach for inter-cloud interconnection and have seen a continuous evolution during recent years.

Multi-cloud libraries [49, 62, 71] build a least common denominator among a lot of different CSP APIs. Library-based approaches provides the best flexibility, supporting a lot of different providers API. Libraries are useful to express resource-oriented workloads requirements, communicating with different APIs with a driver mechanism, but after several evolutions two main limitations may still be identified. MCLs are uncapable to provide an optimized multi-cloud deployment over least-common denominator. In addition, they cannot be easily extended since they leverage a pure modular design, which prevents the possibility to share sparse code portions as we will detail in next sections.

Despite the fact that they pushes all the complexity on the consumer, multi-cloud libraries showed a wide adoption since they implemented a software delivering paradigm called "Infrastructure as Code". Such approach makes the construction of multi-cloud in-
2.3 STATE OF THE ART: INTER-CLOUD ARCHITECTURES

In order to simplify the access to different CSPs, several works [5, 48, 86, 96] introduce a virtual IaaS infrastructure layers to expose to the customer an homogeneous set of API. The cloud consumer can therefore access this “layer” to allocate resources on underlying CSPs without have to deal with peculiarities of each CSP. In a nutshell, such client-controlled infrastructure layer allow to escape vendor lock-in without introducing more complexity, in applications, as showed in Figure 2.9.

More precisely, a cross-CSP interoperability is built as an infrastructure template to instantiates on different CSPs. For instance, as proposed in [48, 96], introduced a consumer-controlled OpenStack [84] cloud management systems deployed over CSPs. Another example is represented by open-source container orchestrator (e.g. Kubernetes) which have gained popularity and are more and more deployed over existing CSPs by consumers representing a compatible overlay layer. In fact, those are becoming more and more popular pushing major CSPs to provide “managed” versions to their consumer (e.g. for Kubernetes: Azure AKS, AWS EKS, Google GKE). Those container orchestration proposes an hybrid IaaS approach since they provide OS-abstraction which follow the life-cycle of the contained application.

Overlay layers can be combined with aforementioned client-oriented approaches, multi-cloud libraries and brokers, to construct a customized infrastructure. Obviously, the IaaS interoperability, introduced by overlay services, may not solve incompatibility occurring with "managed services", but allows to hide underlying heterogeneity to applications. Such techniques showed acceptable performance and consolidation improvements compared to traditional cloud deployment [5].

2.3.4 Conclusion

In the previous sections, we analyzed different classes of approaches for multi-CSP interconnection.

First, we observed that approaches requiring the involvement of multiple CSPs (federations) were not adopted in general. The reason is mainly twofold. Provider-oriented approaches have important limitations. CSPs are competitors, tend to "tie" at most their framework closer to traditional software development, traditionally properly handled by consumers.
customers to their eco-system and they are not willing to cooperate. Therefore, provider-centric approaches do not represent a really viable approach from the consumer perspective, despite those represent an interesting solution for the consumer since it provides the best score in terms of user-involvement. In addition, hybrid cloud provide a seamless solution in terms of user involvement in constructing the multi-CSP integrations, but those are normally limited in number of CSPs and does not provide a generic and always-applicable answer to this problem.

Second, client-oriented approaches introduce extra costs, either in terms of a new actor, the broker, which is in charge of constructing the interconnection, or in terms of important technical efforts on the consumer side, which has to gain an insight on MCLs. More precisely, on the one hand, broker-based approaches are capable to construct the multi-cloud by interpreting the requirements of a customer and allocating resources consequently. However, a reality check shows that consumer seldom accept to introduce a third-party and pay the associated costs. On the other hand, MCLs provide the consumer a mechanism to deploy and spawn resources across multiple providers, without any particular constraints. Several approaches enables a least-common denominator to consumers, abstracting them from the CSP technical details. Such approaches showed to be flexible, but requires the user to become familiar with the target CSP cloud ecosystems.

To sum up, we consider MCLs as the most interesting solution to overcome CSP interconnection limitations. Despite the effort required on the consumer's side, those approaches showed an important popularity, since they does not involve any further actor involved in cloud consumption and cannot be blocked by CSPs federation.

2.4 STATE OF THE ART: MULTI-CLOUD LIBRARIES (MCL)

MCLs are software libraries capable of provisioning resources and consuming services on multiple APIs, by supporting multiple CSP. Multi-cloud libraries partially overlap the traditional tools for software delivery, in the sense that it extends the "Infrastructure as Code" (IaC) paradigm to embrace resource allocation on different CSPs. Since IaC simplified a lot the process of software delivery, it is worth observing which effects is in a multi-cloud scenario and how MCL tools based on IaC can simplify the multi-cloud life-cycle management.

This section is organized as follows. First, in Section 2.4.1, we introduce the IaC and describe its functionalities and design approaches. Second, in Section 2.4.2, we introduce the MCLs and describe their functionalities and how they are designed. In particular, IaC is the fundamental paradigm over which the construction of MCLs is based on. Thus, we detail what are the design properties that IaC introduces in MCL, which are stressed and emphasized in a multi-cloud context. In particular, we focus on how multi-cloud architectures actually benefit from the novelties brought by MCLs, which enable functionalities that were not possible by merely using non-dedicated IaC tools. Third, in Section 2.4.3, we discuss the limitations that MCLs present with respect to the multi-cloud scenario. Despite being the most prominent candidate for resource delivery across multiple and diverse CSPs, we have identified the two following limitations: lack of extensibility and lack of specialization. These are presented and motivated as they are the open problems that this manuscript ultimately aims to solve.
2.4.1 **Infrastructure as Code**

IaC is a software delivery paradigm that has grown in popularity during the past years [60], changing the way cloud infrastructures and applications are delivered. Conceived in the context of the DevOps software engineering culture, IaC benefits from cloud elasticity, self-service and on-demand nature to automate the software delivery life-cycle. In the context of multi-cloud, as we will present later in this chapter, IaC properties represent an important basis to lower the complexity of multi-CSP deployments compared to pre-IaC tools.

Before introducing in depth IaC, we recall the details of "software delivery" process. Software delivery is normally organized in five major steps: Develop, Build, Test, Deploy, Run. These steps are traditionally tied to different roles in IT organizations. On the one hand, developers are responsible for developing, building and testing the service. On the other hand, "operators" are in charge of deploying and running the artifacts produced by the developers.

The DevOps [118] software engineering culture blurs the separation between developers and operators and breaks the conflicts among demand for new delivery and conservation of the "status quo" of the IT system (e.g. a stable version of a certain application). DevOps principles are practically instantiated by the usage of tools and approaches that allow to effectively transform traditional developer and operator roles into the DevOps. The most important are (1) IaC and (2) continuous integration/delivery (CI/CD). First, CI/CD enable the possibility to automatically build, integrate, test and deliver a software. The usage of such tools may boost the delivery project since it automates many steps through the definition of "pipelines", which describes how the software should be built, tested and deployed.

Second, the IaC paradigm can be resumed as "write and execute code to define, deploy and update your infrastructure" [25]. Before the introduction of IaC, the traditional approach was to use *ad hoc scripts* where installation, configuration and provisioning could be automated through scripting languages (e.g Bash) or imperative programming (e.g: Python, Ruby). Such scripts performed a certain amount of actions on target resources without expliciting which was the expected status. On the contrary, as described in Figure 2.10 the IaC approach is based on describing the desired state of the new resources (e.g. package installed on a virtual machine) instead of simply performing actions on them (e.g. the command to install the package). Such approach has the key advantage to improve automation compared to the script baseline. More precisely, IaC artifacts normally completely encapsulate all the information necessary to construct the infrastructure (e.g. allocating resources, connect and configure them) and can be seen as an holistic representation of it. In other words, from the Devops perspective, resource allocations and software configurations are simply a state to achieve without caring about the previous state, and, depending of the tools adopted, also of the technical details concerning the action performed.

Combined with CI/CD, IaC can be used to automatically test and deploy different environments with low manual intervention. Therefore, the IaC holistic representation reduces the presence of differences between testing and production environment, which normally prevents the possibility to deliver new version of a service. The final result is the possibility to completely reconstruct the infrastructure on multiple independent instances, which is useful in many steps of CI/CD (e.g. Testing).
In addition, IaC introduces two major possibilities compared to traditional ad-hoc scripts. First, IaC code can be versioned and configured through the same tools used for actual software development (e.g. versioning, continuous integration, testing).

Second, IaC introduces the possibility of static analysis of the code responsible for resource allocation and configuration. This enables early detection of errors and prevents erroneous configurations (e.g. wrong privileges, network topology errors) before deployment.

In the following, we present the three main techniques in which IaC is nowadays implemented.

**Configuration Management Tools (CM)** \[11, 33, 94, 100\] are tools designed to cope specifically with configuration tasks on already deployed resources. The straightforward scenario is the configuration of a pool of hosts (e.g. VMs, physical servers) belonging to different groups and environment (e.g. development, production). Those tools were massively adopted in cloud computing to configure blank Virtual Machines after boot-strap but can be adopted in legacy environment to configure physical servers. Moreover, such tools normally exports high-level "functions" to obtain a result on a target VM (e.g. have a package installed). Compared to traditional scripts, CMs normally provide a rich feedback about the fact that the state coincides with the desired one.

Those tools may rely on a dedicated declarative/imperative syntax and present a centralized agent-less architecture or rely on active agents, executed on each target node, that are pulled for configuration updates.

**Server Templating Tools**, as Docker \[41\] or Packer \[56\], enables the possibility to define the configuration of a target system allowing to construct a ready to deploy image of such configuration and then use it multiple times, shortening the amount of time required to spawn new instances. Such class of tools are normally really good for scalability since they do not require any "live" configuration and are fully operational after spawning. However, server templating is suitable for immutable infrastructures (e.g stateless applications), where the execution environment has not to preserve a status. This become more complex for execution environments that have to be persistent and change across the time (e.g. system upgrades). This brings the problem of configuration drift since the original image has now a modified configuration and differs by the expected state.

**Server Provisioning Tools**, as Heat \[84\], CloudFormation [] or Terraform \[57\], consist of a declarative language focused on the description of allocated resources. In contrast to Server Templating, this class of tools is designed to describe "mutable infrastructures", whose evolution should be tracked and leaded across the time. Such tools stresses the capacity of a DevOps to analyze the structure of deployed services and to enforce such structure respecting idempotency. Server provisioning tools represent a complementary approach more than an alternative to CM. Compared to other tools, those normally rely
2.4 State of the Art: Multi-Cloud Libraries (MCL)

Figure 2.11: IaC-based MCL architecture: the IaC principle is now extended to allocate and configure resources on multiple CSPs

on a further "state" registry which helps tracking deployments across the time. When a template is deployed, this produces a variation of state in the CSP allocating resources. If the deployment is successful, this state will converge to the one specified in the code. However, the state of the deployment is recorded aside the developer code and can allow to detect configuration drift in deployed resources (e.g.; a deleted VM) or evaluate the impact of a code change over the infrastructure without having to contact the actual CSP, with a significant speed up in performance. Such feature is interesting particularly in multi-cloud context as described in the next section.

To sum up, we can observe that many different and complementary tools exists to adopt IaC. All those three classes can be combined together and used at the same time, and have overlapping features. For example, we can decide to use Server Provisioning to spawn some pre-packaged server templates and then do the post-configuration through a CM.

The great part of those tools is that they support the API of multiple CSPs and represent therefore a candidate to be adopted as MCL to build a client-centric multi-cloud. In next section, we analyze if and how such IaC approaches can be used as MCLs, how they compare to other MCLs and which represents the most suitable approach.

2.4.2 Infrastructure as code and multi-cloud libraries

As we discussed in the previous section, software delivery in cloud-computing strongly benefits from IaC. The different classes of tools for IaC lead to different compromises to
automate resource provisioning and configuration in different environments (e.g. Dev, Testing, Production).

Furthermore, pushing forward the analysis of the scenario we presented in Section 2.3, we observe that the multi-cloud introduces two further complexities which can massively benefit from IaC features. First, multi-cloud has by design many different target "production" environments. In fact, a multi-cloud application should be executed on multiple CSPs at the same time. This mean that the application has to be developed, tested and deployed independently on each CSP. Second, the distributed "multi-provider" architecture of the multi-cloud benefits from the "state" oriented approach of IaC, since it is more difficult to track the state of allocated resources in each CSP.

Based on IaC, we can identify two major requirements that a multi-cloud library should satisfy:

1. **CSP-Agnosticism**: the MCL should not be tied to a specific CSP service specification. The MCL should be able to create resources in different providers. There exists two classes of CSP-agnosticism:
   - First, the MCL proposes a unified domain specific language (DSL) to define resources with the same syntax but a different semantic (e.g. VMs for different CSPs have different primitives).
   - Second, the MCL supports a semantic decoupled from the underlying CSP: a generic resource allocation to be translated to the appropriate resource in the selected CSP ecosystem. MCL should reduce at most the quantity of CSP-specific code, having the possibility to reuse the code on different providers and assure the maintainability across the time. MCL modules should be replaceable, and clearly defined in terms of the required/offered services.

2. **Declarative DSL**: a declarative syntax may allow to easily analyze the topology of deployed services, rather than present the actions performed. This is due to the fact that declarative syntax describes the expected state to be enforced on CSP and therefore, may be considered as an "image" of the expected results. For example, resources declared in a IaC file can be analyzed as a graph.

In Table 2.3, we analyze available MCL tools classes. In this analysis, we decided not to consider server templating technique as a possible alternative since they are conceived for immutable infrastructures, which does not fully cover all scenario in our use-case. On the contrary, we analyze the other classes highlighting how they can fulfill our requirements.

First of all, it is worth noting that MCL are intrinsically connected to IaC principles. Before the introduction of IaC, existing MCL (e.g. fog.io [49], jclouds [62]) represents multi-cloud extensions of traditional ad-hoc scripts in different programming languages, as presented in the previous paragraph as term of comparison for IaC. Those MCLs are imperative and their semantic does not focus to converge to a desired state but targets only the fact of performing an action of a remote CSP. This prevents the possibility to have a simple management over the time of allocated resources, since idempotent allocations is up to developers and not provided by the tool. Moreover, it should be considered that those approaches embed library inside applications, requiring them to remain synchronized with CSP APIs evolution. This poses also some maintainability
issues [42] since the library and relative application has to be kept updated to work with the more recent version of CSP’s API. To sum up, those solutions do not represent a suitable approach for our use-case, where we want to transparently build and manage a complex infrastructure across multiple CSPs.

Analyzing MCL tools based on IaC, we can observe that they extend the state-based enforcing approach over multiple CSP, as shown in Figure 2.11. Presented in previous section as one important class of IaC tools, Configuration Managers (CMs) are not specially conceived for cloud-computing purposes and focuses more on configuration perspective, namely action-oriented perspective, rather than on resource allocation. CM normally supports multiple CSP resource allocation [11, 94], but integrates such resources allocations in their configuration-driven syntax. As we previously described, CMs normally rely on a declarative/imperative DSL to enforce a particular state. Such DSL encapsulates functions to allocate resources on most popular CSP and therefore, de facto supporting multi-cloud. However, such support is not completely satisfying since it does not focus on allocated resources and their interconnection. This prevents the consumer from the possibility to easily analyze the state of a deployment since the enforced state has to be extracted from the actions. On the contrary, several server provisioning tools (or orchestration languages) [57, 82] proposes multi-cloud integration with a resource oriented approach. Such resource oriented approach enables the possibility to construct a graph of allocated resources, on one or multiple CSPs, and to easily inspect the deployment and track the state of the resources.

OASIS TOSCA [82] introduces the "matching" mechanism which enables the possibility to decouple the "abstract" initial service description of service from concrete implementations. In other words, compared to other "Server Provisioning" tools or CMs, TOSCA decouples the specification in presence of a different CSP, without having to modify the base code. This pushes the CSP-agnosticism beyond the mere capacity of allocating resources on a remote CSP but defines a common representation of those resources which can be instantiated on multiple CSP. For example, Terraform [57] supports a driver-based mechanism to support a wide variety of different CSP but the code that has to be written to allocate those resources is different from a CSP to another. This does not fulfill the requirement of having the size of code to maintain as tiny as possible. In addition, the Cloud Modelling Framework (CloudMF) [46] uses model-driven development to generate the specifications of resource provisioning and deployment of multi-cloud applications. Despite the fact that it does not follow the above mentioned IaC principles, it proposes a model-based approach which can deal with multi-provider semantic. Starting from a cloud independent model expressed through a language called CloudML, the framework is capable to generate a corresponding implementation on different supported CSPs. CloudMF proposes a semantic multi-CSP support through a cross-provider meta-model, but do not propose a declarative syntax which can simplify a lot resource management in a multi-cloud context.

To sum up, server provisioning tools provide the most interesting toolkit to handle multi-cloud deployments. The resource-oriented declarative model and support for encapsulation and multiple-CSP makes such tools the most interesting choice. However, as presented in the next section, several important limitations still exist considering their adoption in a multi-cloud setting.
Table 2.3: Comparison among different tools that can be adopted as Multi-Cloud Libraries

<table>
<thead>
<tr>
<th>Approach</th>
<th>Library/Toolkit</th>
<th>Type</th>
<th>Multi-CSP</th>
<th>Paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad-hoc scripts</td>
<td>JClouds [62]</td>
<td>Ad-hoc</td>
<td>Syntactic</td>
<td>Imperative</td>
</tr>
<tr>
<td></td>
<td>Fog.io [49]</td>
<td>Ad-hoc</td>
<td>Syntactic</td>
<td>Imperative</td>
</tr>
<tr>
<td></td>
<td>Libcloud [71]</td>
<td>Ad-hoc</td>
<td>Syntactic</td>
<td>Imperative</td>
</tr>
<tr>
<td>Configuration Managers (CMs)</td>
<td>Chef [33]</td>
<td>CM</td>
<td>Syntactic</td>
<td>Imperative</td>
</tr>
<tr>
<td></td>
<td>Puppet [94]</td>
<td>CM</td>
<td>Syntactic</td>
<td>Declarative</td>
</tr>
<tr>
<td>Server Provisioning Tools</td>
<td>TOSCA [36, 82]</td>
<td>SP</td>
<td>Semantic</td>
<td>Declarative</td>
</tr>
<tr>
<td></td>
<td>Terraform [57]</td>
<td>SP</td>
<td>Syntactic</td>
<td>Declarative</td>
</tr>
<tr>
<td></td>
<td>CloudMF [46]</td>
<td>SP</td>
<td>Semantic</td>
<td>Imperative</td>
</tr>
</tbody>
</table>

2.4.3 Limitations

Server provisioning tools can provide an effective solution to multi-cloud issues. The possibility to describe in a provider-agnostic and declarative way deployed resources allows consumers to orchestrate their multi-CSP workloads, while reducing the efforts of development and maintainance. However, such efforts are still not negligible and they still concern (1) the extensibility of infrastructure code and (2) its specialization w.r.t. implemented CSP services.

2.4.3.1 Extensibility

In this section, we analyze the different MCL approaches presented above. First of all, we analyze how the different classes of MCLs can be extended and which properties they should provide. Second, we analyze how this is achieved in imperative programming, with a particular focus on Aspect-Oriented Programming (AOP).

The declarative and CSP-agnostic code, proper to several IaC server provisioning frameworks, simplifies a lot the integration and the deployment of functional and non-functional services.

However, such adoption still does not represent an ideal solution. As we mentioned above, The multi-cloud requires the definition of a virtual infrastructure capable of working on different CSP and this represents a huge amount of code to be developed and maintained. More precisely, the code-base to be maintained introduces non trivial extra-costs, where each CSP represents a different environment where to debug, test and deploy the CSP-agnostic code.

As already presented before, we can consider that each consumer has to rely on its applications and a set of non-functional services, which cover auxiliary features (e.g; monitoring, backup, debug). The code of such non-functional services should be re-used as much as possible across different customers.

However, the capacity of handling the lifecycle of a similar multi-cloud application is not trivial. In fact, non-functional services are normally scattered across several infrastructure elements (e.g. Network topology, OS configuration). For example in case of monitoring, new trends [92] have multiple agents inside user VMs and alerts man-
agers as standalone. This blurred "border" violates the traditional "module" or "package" driven service composition, requiring a more sophisticated integration of components.

Moreover, after deployment, infrastructure services have to be maintained by continuously checking integrity and availability. Run-time updates of the infrastructure in case of an unexpected event cannot be easily handled dynamically (for instance, reacting to the suspicious rise of incoming network traffic, deploying security services to protect a victim service). In the absence of an extensibility support from the IaC framework, such run-time manipulations have to be manually performed on the IaC code, impacting the capacity for the cloud consumer to quickly respond to that specific event.

Consequently, it is not possible to easily reuse non-functional code across different applications. Continuing with the previous example, a monitoring framework may necessitate to distribute agents on each monitored VM. Thus, assuming that IaC approaches to multi-cloud presented above, the code of a non-functional service cannot be easily inserted in or removed from the infrastructure template. Therefore, each possible combination/configuration of non-functional services has to be manually realized and tested. In fact, in the scenario of multi-clouds, this process has to be repeated for each CSP involved in the infrastructure, without any support from the IaC framework.

In the previous section, we discussed different IaC-based MCL that can be used to deploy a multi-cloud infrastructure. In particular, we identified that the server provisioning tools are more suitable due to their declarative syntax and their semantic support of multiple CSP. However, they do not introduce any support to simplify the re-usage of code.

This feature is necessary but not sufficient to support extensibility. Declarative syntax is necessary since it enables the possibility to write generic "extension modules" without having to specify them for each different CSP. On the contrary, it is not sufficient since declarative syntax has to be coupled with a resource-oriented semantic which allows to construct a topology graph. As we mentioned earlier, CM does not normally meet this requirements since their capacity to allocate resources can be seen as an extension of their initial configuration objective. Despite the fact that Terraform supports dynamic updates and manages them through the definition of a desired "state" of the deployment, the framework does not meet this criteria and therefore TF files cannot be easily extended in a multi-provider fashion. Each resources declared in a TF file is specifically crafted for a CSP and therefore any extension code should respect the same semantic.

CloudMF and TOSCA supports multi-CSP through a semantic abstraction and templates can therefore be extended. In particular, they are capable to do so since it provides a mechanism to define ontology of resources, which can then be translated to CSP-specific languages. However, no projects have targeted a similar objective we consider key to simplify the deployment of multi-cloud applications.

If we analyze how a similar extensibility is achieved in imperative programming, we observe that Aspect Oriented Programming (AOP) propose a paradigm which is capable to do so. AOP [66] enables to separate cross-cutting concerns to enhance modularity. In other words, AOP consists of separating the program logic in different parts, called concerns, in standalone modules called aspects. More precisely, concerns (e.g. logging, security, transaction) may traverse the whole logic of the program, being spread in different uncorrelated modules. This can lead to the well-known problem of code replication. The latter is the result of reinserting often the same code to satisfy the desired concern
Table 2.4: Comparison among different Multi-Cloud Libraries considering extensibility and specialization

<table>
<thead>
<tr>
<th>Library/Toolkit</th>
<th>Type</th>
<th>Multi-CSP</th>
<th>Paradigm</th>
<th>Extensibility Support</th>
<th>Specialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>JClouds [62]</td>
<td>Ad-hoc</td>
<td>Syntactic</td>
<td>Imperative</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Fog.io [49]</td>
<td>Ad-hoc</td>
<td>Syntactic</td>
<td>Imperative</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Libcloud [71]</td>
<td>Ad-hoc</td>
<td>Syntactic</td>
<td>Imperative</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Chef [33]</td>
<td>CM</td>
<td>Syntactic</td>
<td>Imperative</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Puppet [94]</td>
<td>CM</td>
<td>Syntactic</td>
<td>Declarative</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>TOSCA [82]</td>
<td>SP</td>
<td>Semantic</td>
<td>Declarative</td>
<td>Modules</td>
<td>Partial</td>
</tr>
<tr>
<td>Terraform [57]</td>
<td>SP</td>
<td>Syntactic</td>
<td>Declarative</td>
<td>Modules</td>
<td>×</td>
</tr>
<tr>
<td>CloudMF [46]</td>
<td>SP</td>
<td>Semantic</td>
<td>Imperative</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

in a specific module. This has the side effect to potentially introduce more faults and degrade maintainability, since an update to this code has to be replicated several times.

Leveraging aspects, AOP splits the "functional" code of the applications and "non-functional" aspects. Code composition, or weaving, is behavior-oriented. More precisely, it relies on the concepts of point-cut and advice, which are respectively the extra-code and the code location where it should be applied. Additional “aspects’ to the base code are described without having to modify the original code itself.

A further advantage versus traditional modularization in imperative programming is represented by the possibility for a developer to change the state of the entire system easily, by modifying a single aspect and reflecting the modification on the whole program. Support for AOP is implemented in many popular languages [109].

To sum up, in a multi-cloud, the codebase and the efforts required to maintain services becomes rapidly intractable and prevents a multi-cloud consumer from enjoying the same flexibility as in a single-cloud ecosystem. Available solutions cannot fully cope with the requirements of dynamic extensibility of IaC templates in order to overcome the lifecycle handling issues and continous integration of a multi-cloud setting. Existing solutions outside the IaC can provide a compelling approach to handle it, but they still require to be adapted to the IaC context.

2.4.3.2 Specialization

Multi-cloud infrastructures come to the field at the price of the loss of specialization in the cloud resources definition. In fact, most popular approaches, such as MCLs [82], enable multi-CSPs resources allocation in a provider-agnostic fashion. This means that the resources are deployed in the same generic manner without taking into account specificities of each CSP. Indeed, MCLs rely on a single formalism to express "least common denominator" [42] resource definition. In a nutshell, the cloud consumer cannot express his requirements in terms of services and resources from the CSP and, consequently, cannot obtain the most suitable implementation for the given applications. For instance, some users may not be able to dynamically benefit from specific needed services such as on “hardware accelerators” (e.g. GPU, FPGA) or CSP-managed services (i.e. DBMS as
a service). Among the existing approaches, server provisioning tools as Terraform [57] and Configuration Management tools (CM) as Ansible [11] are not designed to handle code transformation capable to deal with such specialization, since it does not support semantically CSP-agnosticism but they only provide a multi-CSP compatible syntax.

TOSCA primer [82] introduces a mechanism to recursively adapt service definition, making concrete NodeTypes through a substitution with equivalent ServiceTemplates (STs). A ST can substitute a selected Node if and only if the exported "boundaries" (Operations, features, Requirements, Properties) are the same of targeted NodeType. This "matching" process introduces some form of specialization since it is designed to select existing implementations and components on a specific CSP starting from an abstract definition.

However, in our scenario this is probably not sufficient. Another limitation we observe is related to the capacity of the cloud consumer to describe its desired IaC infrastructure as he needs an in-depth knowledge of CSP-specific ecosystems while building multi-cloud architectures. In fact, CSPs ecosystems are growing in complexity and in number of exposed services. Every year, public CSPs and even open-source cloud management systems introduce new services to intercept new use-cases.

If we analyze the TOSCA matching, we observe that the consumer cannot customize on-the-fly the matching process to obtain an optimized usage of newly introduced CSP servers. More precisely, in [26], Brogi and Soldani formally introduced several different types of matching (exact, plug-in, flexible, white-box), extending the strict and abstract concept of having the same boundary to looser definitions introduced in the TOSCA primer [82]. However, the different TOSCA matching algorithms presented are only capable to select implementations which respect at different level a "structure" defined by the NodeType to replace. In a nutshell, the idea is to match the TOSCA Requirements and Capabilities of a NodeType with a compelling ServiceTemplate, obtaining a refined and equivalent definition of the service. Modern CSPs offer normally multiple ways to perform certain tasks (e.g. containers or VMs) which may suit different use-cases and optimization criteria. Managing TOSCA, the matching algorithm only evaluates the "compatibility" of a certain component to replace it in an abstract STs but it is not designed to choose the most suitable one for the specific deployment context.

Consequently, this calls for the necessity of dedicated investment in order to embrace what each ecosystem proposes, requiring dedicated training from the consumer side. Therefore, the cost of adopting multi-cloud requires to reduce the number of target CSP. Such approach makes multi-cloud not affordable for many organizations and represents an important barrier to initial adoption.

To sum up, the only way to get rid of this limitation is to manually set up a specific implementation available on a certain CSP that the consumer wants to adopt. This would result in a different infrastructure code specification for each underlying CSP and would not scale the amount of work to be done to set up a multi-cloud infrastructure.

### 2.5 Conclusion

In this chapter, we presented background and related works on multi-cloud architectures. Despite being interesting for many applications, multi-cloud architectures require a lot of efforts to be adopted and be maintained. IaC is the most prominent candidate for simplifying software delivery and can provide a compelling solution for making multi-cloud
deployment easier. This, like other approaches, provide an interesting but not optimal solution to handle multi-cloud as we may observe in 2.4. In particular, we identified the lack of extensibility and the lack of specialization as the key limitations of currently available solutions. Such lack prevent the possibility to have a simple and flexible supervision of multicloud and security management. In particular, the lack of extensibility prevents to reuse the same non-functional modules across different customers and deployment imposing important efforts to re-implement the same features. Furthermore, the lack of specialization prevents the capacity in a multi-cloud to use the most suitable implementation of a service to perform a specific tasks, imposing overhead and extra-costs to customers.

In the next chapter, we discuss the contribution proposed in this manuscript to overcome one of those problem, the extensibility.
This chapter focuses on how to overcome the problem of extensibility (specialization is left to Chapter 4). Lack of extensibility of currently available MCLs is witnessed when non-functional services have to be introduced to applications to ensure a continuous life-cycle reality check. In particular, the code describing these non-functional services cannot be automatically used for each of the CSP ecosystems deployed and need to be rewritten and inserted manually for each and every CSP of the underlying multi-cloud architecture. This is due to the fact that the injection of such code is not dynamically supported by the available IaC templating languages responsible for translating an abstract model of resources deployment to a concrete and ready-to-be-used code that actually does that. Addressing lack of IaC extensibility represent an important contribution, since it represents one of the building blocks of the multi-cloud orchestration, as presented in Figure 3.1.

The chapter is organized as follows. First, we recall the problem of extensibility showing why it is interesting in the context of multi-cloud. Second, we introduce the preliminaries which supports the Aspect-oriented Programming and TOSCA. Third, we present the design of an Aspect-Oriented approach based on weaving on Infrastructure as Code capable to address the problem of extensibility. More precisely, we present a language we designed, the TML, Tosca Manipulation Language. We present how it can rely on aspect programming to modify an infrastructure template, then we show how it can be used in a simple example. Fourth, we discuss this contribution, suggesting how AOP weaving can effectively alleviate the software delivery complexities introduced by multi-cloud.

3.1 INTRODUCTION

As discussed in Chapter 1, the multi-cloud architecture is a solution for the design of applications related to use-cases with stringent requirements in terms of QoS. These requirements ultimaly enforce the usage of multiple CSPs at the same time rather than only one. However, leveraging multiple CSPs introduces complexity in terms of both construction and long-term management of the applications built on multi-clouds, which cloud consumers have to face. In Chapter 1.2, we individuated two major limitations, lack of extensibility and of specializations, and pointed out IaC overlay multi-clouds, and in particular, MCLs, as a promising candidate to overcome them.

We consider here lack of extensibility in TOSCA templating language. As we mentioned in Section 2.4.3.1, extensibility prevents easy IaC-based code evolution during the life-cycle and sharing across different consumers. Among different languages, we
adopted TOSCA which embeds important features in terms of interoperability and primitive support to specialization, despite lacking in extensibility support. In this chapter, this lack is overcome by the definition of what we have called the TOSCA manipulation language (TML). The TML is the language obtained by introducing aspect oriented programming (AOP) to the IaC framework. The feature of AOP that we leverage is that it separates non-functional code from functional code. This way, it is possible to abstract the non-functional code needed for multi-cloud applications so that it can fit the ecosystems of all CSPs. For example, on the one hand, different CSPs can propose a managed service to handle monitoring of resources inside its ecosystem. On the other hand, the consumer is interested by the fact that a precise amount and type of functional services are correctly monitored.

Therefore, this abstract code is then specialized and transformed in a concrete and ready-to-be-used piece of code for each specific CSP through TOSCA. In summary, lack of extensibility is mitigated because TML allows at the same to re-use the same abstract code (e.g. monitoring agents and their configuration) across all CSPs and relieves the cloud consumers from manually adapting it to each CSP.

3.1.1 The impact for the multi-cloud

First of all, we recall the fundamental elements of a multi-cloud environment described in Section 1.1. Above all, a multi-cloud scenario implies many heterogeneous production environments. As we already discussed in Section 2.4.1, IaC is a great tool to make different life-cycle environments similar to production by encapsulating all resources involved in deployment. However, CSPs are intrinsically different and this does not only
concern how they implement a service (e.g. the virtualization technology of preference), but also their network configuration and security prescriptions.

In order to simplify multi-cloud delivery of applications, it is key to automate it as much as possible. This automation should not only concern the "proper" construction of the application infrastructure, but also the insertion/eviction of services proper of different life-cycle steps. Therefore, cloud consumers should easily manipulate coherently the infrastructure on different CSPs in order to (1) speed up process and (2) easily troubleshoot effects in spite of the different production environments.

As aforementioned, this is partially achieved by the adoption of TOSCA. TOSCA is capable to obtain different instances of the same infrastructure template, as presented in Section 2.4.2. However, the consumer has to manually modify the code template and relative implementation through each different step. This can represent a lot of work and requires an in-depth knowledge of a CSP-ecosystem.

Let’s analyze the aforementioned problem with an example. A cloud consumer has designed an application for a multi-cloud infrastructure and wants to make this infrastructure evolve during the delivery life-cycle.

The consumers has different non-functional services that he wants to add on specific steps. During the development phase, the cloud consumer would like to have services for troubleshooting. This might mean for instance simple middleboxes capable to sniff network traffic and selective log collecting. CSPs handle network traffic with different policies and what is allowed on one CSP may not be allowed on another one (e.g. multicast traffic).

During the test phase, there are different types of services that a cloud consumer would like to leverage, e.g. performance analyzers (e.g. charge injectors) and security tester (e.g. OS hardening). It is not easy to find equivalent performance among different CSPs. For example, VMs with the same amount of CPU and RAM resources may not perform in the same way since the underlying infrastructure uses different hardware.

During the production phase, monitoring is key to ensure the correct functionality of the platform. Therefore, all resources useful to detect the functionality of the application should be observed and information (e.g. logs; metrics) to detect technical events (e.g.) should be collected in a single point of orchestration. The resulting TOSCA implementations may be different since each CSP may provide or not managed services to integrate information scraped by inside the consumer resources (e.g. application logs) with infrastructure information (e.g. resource provisioning status or budget consumption). Moreover, during the production phase, an event occurrence (e.g. a security alert) may impose a certain modification to the infrastructure, considering the different implementation on each CSP.

To sum up, the presence of different CSP ecosystems introduces significant complexities and pushes the need for the cloud consumer to easily instrument her infrastructure. This is necessary for two main reasons. First, the cloud consumer wants to simplify the set up of its applications (e.g. debugging) or to make easier the proper deployment of those in production.

To do so, in the next section, we propose an Aspect-Oriented Approach (AoA) that is based on the weaving paradigm. While the easiness of manipulation does not represent a strictly necessary requirement for multi-cloud, it must not be underestimated how the complexity and the increased amount of work discourages cloud consumers. Conse-
currently, we do not foresee an important adoption of a generic and flexible multi-cloud without the extensibility feature properly adopted in multi-cloud tools.

3.2 Preliminaries

In this section, we present a preliminar introduction of TOSCA template language and aspect-oriented programming, that will extensively reference across this chapter. Briefly, TOSCA emerged as one of the most promising solution for IaC resource provisioning in a multi-CSP context, supporting a declarative syntax, a resource oriented representation and supporting a semantic yet syntatic Provider-agnostic resource definition. Base of our AoA [66] introduced in traditional imperative programming the possibility to overcome modular-based encapsulation of code in order to confine different "aspects" of the software.

3.2.1 TOSCA

As discussed in Section 2.4.2, MCL-based on IaC offers an interesting approach to provision resources over a multitude of Cloud Providers. In particular, IaC brings to the field a "state-oriented" paradigm, which focuses of enforcing a configuration (i.e. state ) on a target system (e.g. multiple CSPs). MCL-based on IaC provides abstraction to construct idempotent provisioning and configurations. Among the others, we identified TOSCA [82] as a suitable candidate for customer-centric multi-cloud due to its properties in terms of CSP agnosticism, support for CSP-specialization and declarative paradigm (as discussed in 2.4.3).
The OASIS TOSCA represent the base format to represent any resources involved in the multi-cloud deployment process. TOSCA format is designed for providing interoperability between templates of cloud applications by leveraging an object-oriented resources modelling. TOSCA template can model the whole life-cycle of an application (e.g., deploy, patch, shutdown) and provides a definition which is completely decoupled from underlying CSPs. In fact, unlike other solutions like Terraform, TOSCA templates are completely generic and does not reference directly a specific CSP feature.

As presented in Figure 3.2, TOSCA templates are composed of a graph of by nodes_template instances, which "reifies" abstract node_types. Those "equivalent" classes not only describe interactions between different parts of the service but also their life-cycle.

TOSCA nodes, presented on the right in Figure 3.2, are described by a set of 4 classes of attributes: (1) capabilities that they are able to provide to other nodes (Caps), (2) requirements that they need to run correctly (Reqs), (3) Properties (Props) and (4) interfaces (Ints). More precisely, in TOSCA, properties model attributes of resources and interfaces models ways to modify the resource dynamically at run-time, supporting the mapping of (5) operations. For example, for a resource as a VM, interface and operations can map the fundamental actions like start and stop of the VM.

Such attributes model the life-cycle of the components and may be provided through concrete artifacts. An artifact is a named and typed file used to implement deployment and other interface operations (e.g., build or configuration scripts).

A TOSCA ServiceTemplate, as shown in Figure 3.2, is composed of: (1) a TopologyTemplate, which a directed graph-oriented definition of services; (2) NodeTypes, which is a list of definitions of nodes composing the TopologyTemplate, and (3) RelationshipTypes, which is a list of edge types for the TopologyTemplate, modelling custom links between resources. In addition, a ServiceTemplate usually includes: (4) a set of BoundaryDefinitions that specify which capabilities and requirements have to be exported outside of the ServiceTemplate; and (5) Plans, which specify how the node operations should be executed to manage the service life-cycle.

TOSCA Orchestrators match concrete implementations through a matching process [26, 82]. In particular, TOSCA fosters reuse defined components and interoperability through a mechanism of matching between the NodeTypes abstract and equivalent ServiceTemplate implementations, as shown in Figure 3.3.

The TOSCA templates get interpreted and instantiated in a set of distinct resources called a topology instance. Table 3.1 presents a sample ServiceTemplate, which is graphically presented in Listing 3.5 composed of one VM (ComputeBox) with two NICs, two virtual networks (31-33, and 21-23) and a virtual router (35-37) and a router interface (38-42), with annex interface connected to control network (lines 45-48). Resources and their requirements compose the set of vertex and edges of the TOSCA graph.

3.2.1.1 TOSCA Matching

TOSCA primer [82] introduces a mechanism to recursively adapt service definition, making concrete NodeTypes through a substitution with equivalent STs. A ST can substitute a selected Node if and only if the exported "boundary" (Operations, features, Requirements, Properties) are the same as targeted NodeType. An example of Matching is proposed in Figure 3.3. a ServiceTemplate can be matched with a NodeType VMBox. It
is considered equivalent since it exposes a set of properties, interfaces and capabilities completely compatible with the selected NodeTypes.

In [26], Brogi et Soldani formally introduced several different types of matching (exact, plug-in, flexible, white-box), extending the strict and abstract concept of having the same boundary to looser definitions introduced in the primer. All those kind of matching if the matching can still be performed in presence of structural differences. This is more suitable when using TOSCA to model pure software components and therefore they do not really target the multi-cloud infrastructure scenario we are focusing. In particular, Brogi et Soldani proposed the plug-in match, which extends the exact matching to the case where a NodeType N can be matched if the ST "requires less and offers more" than the substituting Node.

It is worth noting that the matching process may be recursive and the NodeTypes of the node inside the ST can be replaced by other lower-layer implementations w.r.t. the matching rules. The matching process ends when no further refinement is possible and the granularity of NodeTypes can be understood by the software component responsible of concrete provisioning (e.g a CSP resource orchestration as Heat).

3.2.2 Aspect-Oriented Programming

AOP [66] is a programming paradigm that takes parts of the program, called concerns, and separates them into different standalone modules, called aspects. Figure 3.4 provides a schematic representation of AOP.

As we mentioned in Section 2.4.3.1, leveraging aspects, AOP splits the "functional" code of the applications and "non-functional" aspects. Parts of code concerning a complementary "aspect" which is repeated many times in the code, is written once in an external module and then added at compile or run time in the right position. This addition or code composition, called weaving relies on the concepts of advice and point-cut,
Figure 3.4: AOP schematic representation

as shown in Figure 3.4. Those concepts models, respectively, the extra-code and the code location where it should be applied.

We can identify two main advantages of AOP across other programming frameworks as Object-oriented programming. First, AOP reduces the code-base, the replicated code and, during the development process, and let the cloud consumers focus on the functional code only. The second advantage w.r.t. traditional modularization (e.g object-oriented classes) is that AOP easily propagates a modification from a single aspect to the whole program.

The main disadvantage of AOP is its complexity. A straightforward effect of AOP in imperative programming is that the execution control flow becomes more obscured and this makes code more complex to be understood and debugged.

Despite the negatives, support for AOP is implemented in many popular languages, for example Java extensions [109] and Python (i.e. annotations).

3.3 ASPECT-ORIENTED WEAVING FOR IAC

In this section, we introduce the principles of aspect-oriented approach for IaC. We start with some background on the weaving specification for TOSCA. We then introduce TOSCA Manipulation Language (TML) for TOSCA templates.

3.3.1 AOP-Based Weaving

Mainstream cloud infrastructures do not always rely on identical functional services to provide on-demand resources. In fact, two classes of extensibility challenges may be identified in a multi-cloud setting with a continuously evolving infrastructure. The first extensibility classes are non-functional services (e.g., security middleboxes, monitoring services, network applications, OS hardening profiles), which are not present in the description of the basic infrastructure, are often required by users when extending their private clouds to public CSPs. Those services offer a better integration with respect to legacy
and enrich the set of features available for the user. This might introduce an interesting added-value for cloud brokers – beyond basic resource brokering, thus enlarging the so far narrow market share of the cloud brokering model [42]. The second extensibility class is represented by the life-cycle management of functional/non-functional services. In particular, infrastructure templates are composed by a set of functional services (e.g., cloud management system) which must be debugged, tested and monitored through additional non-functional services. These might change across the entire life-cycle (development, auditing, production). The ability to inject auxiliary services for different life-cycle steps can thus represent a straightforward approach to simplify deployment and operations. Part of this challenge are notably: (1) the easiness of non-functional service injection/eviction (e.g., to isolate the cause of a misbehavior); (2) possible code re-use across different base templates (e.g., required by different customers).

As we mentioned above, AOP [66] enables the separation of cross-cutting concerns to enhance modularity. Additional "behaviors" to the base code are described without having to modify the original code itself. Code composition, or weaving, is behavior-oriented. More precisely, it relies on the concepts of point-cut and advice, which are respectively the extra-code and the code location where it should be applied. We apply the “aspect” definition to “Infrastructure as Code” by considering non-functional properties that are normally scattered across several resources (e.g. network topology, OS configuration). We also implement a common non-functional service (e.g. monitoring, auditing). For example, in case of monitoring, there can be multi-layer agents inside user VMs and alerts managers as standalone [116]. Decoupling non-functional and functional services results in (1) an enhanced re-usability on different base TOSCA Templates and (2) a dynamic injection upon events.

To address such extensibility challenges, we design TML. TML is an AOP-oriented Domain Specific Language (DSL) to manipulate TOSCA templates. Code weaving does not operate on imperative code, but on the declarative graph in TOSCA files that are parsed, analyzed and modified according to external TML scripts.

The objectives of TML are the following:

- **Service weaving / un-weaving.** This means to inject / evict flexibly non-functional services in / from a TOSCA template. This can satisfy the request of weaving/unweaving services dedicated to a specific life-cycle step.

- **Semantic checking.** This means exploring the graph of NodeTemplates in a ServiceTemplate for early detection (i.e., before deployment) of semantic errors (connection to the wrong resource, security constraint violation), beyond simple syntactic checking (missing connections, wrong references). For example, VMs executing sensitive applications which are directly attached to virtual network directly connected to the Internet, without any security middlebox (e.g. transparent firewall).

### 3.3.2 The TOSCA Manipulation Language (TML)

In this section, we present the TML language to weave non-functional resources to TOSCA Template. We design the TML language to be capable of (1) filter TOSCA Graph properties and (2) manipulate the TOSCA graph injecting/removing nodes and edges.

More precisely, TML scripts enable to modify a TOSCA graph, composed by the resources specified in the `node_templates` fields. This graph is composed of a set $V$ of
tosca_definitions_version: tosca_simple_yaml_1_0
description: Main demo file
Connection Up Time:
topology_template:
  inputs: # [...] Omitted for brevity
  node_templates:
    controller_node:
      type: orbits.nodes.ComputeBox
      properties:
        name: Controller
        flavor: m1.xl
        # [...] Omitted for brevity
    control_port:
      type: orbits.nodes.network.Port
      properties:
        order: 0
        requirements:
          - binding: controller_node
          - link: control_network
    control_net:
      type: orbits.nodes.network.Network
      # [...] Omitted for brevity
    internal_port:
      type: orbits.nodes.network.Port
      properties:
        order: 1
        requirements:
          - binding: controller_node
          - link: internal_network
    internal_net:
      type: orbits.nodes.network.Network
      # [...] Omitted for brevity
    router:
      type: orbits.nodes.network.Router
      properties:
        external_network_name: ext-net
        router_interface:
        type: orbits.nodes.network.RouterInterface
        requirements:
          - routable: control_network
          - router: router

substitution_mappings:
# [...] Omitted for brevity

Listing 3.1: TOSCA template with sample TopologyTemplate (extract)

Figure 3.5: Graphic Representation of template resources
vertices, capturing the set of resources with their attributes (e.g., properties, interfaces with additional artifacts), and of a set \( E \) of edges, capturing the set of requirements for interconnection. We define the structure of a TML script as follows:

- **Filters.** They enable to navigate within the graph, and to specify “anchor” resources (i.e., point-cut) from which rules will be triggered. A filter is composed of a name, a root field that identifies the type of resource to be used as anchor, and a rule that defines the conditions to be verified to select the node when exploring the graph. The operators used to build a rule are presented in Table 3.1.

- **Actions.** They represent the core of the advice and define the list of modifications to apply to the graph, leveraging filtered resources as arguments. An action is composed of an action type and of arguments. Following a declarative approach, their execution order does not correspond to their specification order on how actions are specified. A priority graph allows to create resources before they have to be manipulated or connected with the others. The actions used to build a rule are presented in Table 3.2.

- **Checks.** They are performed after executing the actions. In particular, a list of statements is run to validate the obtained template. In case of an empty actions section, the advice simply attempts to verify some properties on the template graph.

Weaving of a filter advice is straightforward. That is, starting from resources of a specified type, the filters’ rules select resources in alignment with specific requirements. The mandatory rules in the filter act as trigger conditions for the advice: when all mandatory filter conditions are able to retrieve a resource, the actions section is executed. Advices are atomic. Therefore, the TOSCA template should be in a valid state after the application of all the actions. If check statements are specified, they should be verified to commit the template and accept the weaving. Developers may also define an optional inputs section to customize TML script inputs.

As we mentioned above, developers define actions inside the TML scripts. Actions may modify the graph according to TOSCA type specifications, but cannot violate imposed constraints (e.g., introducing new requirements or capabilities). We implement the weaver logic to prevent Actions from using filters on temporary states of the graph as filters are computed before execution – similarly for filters “embedded” inside actions (e.g., for the select operator), and for checks after weaving of any action.

### 3.3.3 An example for weaving from a TML advice: Injecting a Floating IP within a VM

During the debugging of an infrastructure, it can be useful to have external IP addresses to inspect the status of all VMs – which on the contrary should not be accessible in a production deployment. Listing 3.2 shows a simple TML advice to selectively inject a floating IP (also called elastic IP), or public Internet-accessible IP into a VM such that each network port is connected to Internet. However, not all ports should receive a floating IP, as most are not connected to the Internet, but only to internal virtual networks. This script can be used in the Continuous Integration (CI) chain to easily collect information about errors while testing.

The filters rule starts looking-up and selecting Ports in the graph, with the following point-cut conditions (line 4 of Listing 3.2): (1) selecting all ports that have links connected to networks, which are again connected to one or many virtual routers that are connected to Internet; and (2) avoiding interfaces already having a floating IP.
### Table 3.1: TML Filter operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&gt;</td>
<td>Requirements</td>
</tr>
<tr>
<td>[]</td>
<td>Capabilities</td>
</tr>
<tr>
<td>()</td>
<td>Interfaces</td>
</tr>
<tr>
<td>all()</td>
<td>Select all elements</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Not Existence</td>
</tr>
<tr>
<td>output : x</td>
<td>Select action created element</td>
</tr>
<tr>
<td>input : x</td>
<td>Select input element</td>
</tr>
<tr>
<td>select : [S,&quot;R&quot;]</td>
<td>Select elements R in set S of elements previously filtered</td>
</tr>
</tbody>
</table>

### Table 3.2: TML advice actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Arguments</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>create / update / delete</td>
<td>resource specifications</td>
<td>Source resource, offering a capability</td>
</tr>
<tr>
<td>connect</td>
<td>source, destination, capability</td>
<td>Target resource, Capability/requirement pair</td>
</tr>
<tr>
<td>create_and_connect</td>
<td>for_each, create arguments, connect arguments</td>
<td>Number of iterations</td>
</tr>
<tr>
<td>create_artifact</td>
<td>artifact specifications</td>
<td>Source resource</td>
</tr>
<tr>
<td>connect_artifact</td>
<td>source, destination, interface</td>
<td>Target artifact, Interface to attach the artifact</td>
</tr>
<tr>
<td>delete</td>
<td>Resource</td>
<td></td>
</tr>
<tr>
<td>update</td>
<td>Resource, Property</td>
<td></td>
</tr>
</tbody>
</table>
The actions section define which modification can be applied on the infrastructure template. In the actions section of the sample example (lines 8-17 of Listing 3.2), the `create_fip` item will create the new resources for each selected port. It defines an iterator \((R)\) (line 12) to specify the set of new resources and to complete the other resource parameters (\(net\_name\)). For every matching resource, it creates and connects a new floating IP resource, through the "floatable" capability defined in the type definition of `orbits.nodes.network.Port`.

Weaving the TML script to the template shown in Listing 3.1 of Section 3.2 yields the result shown in Figure 3.6. More precisely, when run, the TML advice creates and connects a new floating IP resource for the unique port connected to the Internet, the `control_port`.

It is worth noticing that the TML aspect is completely uncorrelated from the base template and that it can be applied to any TOSCA ServiceTemplate. Moreover, if the ServiceTemplate has more than one port connected to an external network, multiple resources are injected without requiring more generated code.

In this work, for the sake of simplicity, we assume the absence of overlap between TML scripts in case of multiple weaving, not focusing yet on idempotency and conflicts issues. However, more sophisticated conflict resolution strategies have already been explored to satisfy an idempotency property [58].

3.4 DISCUSSION

The contribution presented in this chapter is not restricted to the lack of extensibility of the template enrichment/refinement step and the service instantiation step of the multi-cloud deployment (we refer to Figure 1.3 as a remainder).

TML can be used to inject resources or to validate properties in the graph of TOSCA resources. This can achieve automatic injection of services across different steps of the life-cycle. By simply deciding to weave a certain group of scripts, the TOSCA template will be enriched with the desired resources and check that the graph respect certain validation criteria (e.g. for security).

In fact, what we have achieved goes beyond the very initial multi-cloud deployment phase. The TML addresses the modifications that will have to be implemented to the multi-cloud throughout the entire life-cycle of the infrastructure. More precisely, TML allows cloud consumers to write code during the first deployment, that prototypes future evolutions of the infrastructure.

For example, in case these foreseen events occur, the TML, which is built on TOSCA, will let the consumer to weave an update version of the template. This way, equivalent evolutions and modifications are implemented to all CSPs, making the whole system resilient to such events without the necessity for cloud consumers to manually modify the TOSCA code. This feature can ultimately be seen as a single point of orchestration for the multi-cloud infrastructure, where issues, modifications, and countermeasures are solved or applied at once. This prototyping code can be shared among different users, applications and CSPs.

Without the TML, the cloud consumer would have to apply the desired modifications manually as soon as such an error occurs, as it cannot be easily prepared beforehand. In Chapter 5, we will discuss the benefits of TML and TOSCA matching when the multi-cloud infrastructure evolves, and how it can trigger these evolutions.
Listing 3.2: TML Floating IP injection Script

```
inputs: [...] filters:
ports_internet_without_fip:
  root: orbits.nodes.network.Port
  rule: "R.link[].routable<>.rou_int[] && !R.floatable[]"
  mandatory: true

actions:
create_fip:
  type: create_and_connect
  args:
    for_each: [ports_internet_without_fip,"R"]
    name: compute_fip
tosca_type: orbits.nodes.network.FloatingIP
capability: floatable
  properties:
    net_name:{sel_cur:"R.link[].routable<>.router[].net_name"}
  checks:
    no_ports_internet_without_fip:
      root: orbits.nodes.network.Port
      rule: "R.link[].routable<>.rou_int[] && !R.floatable<>"
      mandatory: true
```

Figure 3.6: The template sample in figure 3.5 is weaved with Floating_ip script, which adds to a specified VM an external Floating IP addresses to provide external internet access. The filter action present in the TML script parses the graph and identifies a instance of resource of Type Port which satisfies the constraints of being connected to a router (R.link[],routable <> .rou_int[]) and has not already a floating IP (!R.floatable <>) since the capability "Floatable" is not used.
3.5 CONCLUSION

In this chapter, we presented our contribution about introducing extensibility in "Infrastructure as a Service". Leveraging AOP, we introduced the possibility to inject/extract extra resources to a base IaC template. This fosters code reuse, since non-functional services can be dynamically added to different base infrastructure without modifying the "functional" code-base. More concretely, we introduced AOP-based weaving to introduce extensibility in TOSCA, a IaC framework for Multi-cloud.

To do so, we defined a language, TML, to formalize TOSCA manipulations and how those have to be performed. As we will describe in next chapter, the TML weaver is one of the fundamental behind MANTUS, an aspect-oriented multi-cloud builder, designed to generate an "enriched" multi-cloud application template (e.g. overlay infrastructure). The objective of MANTUS is to flexibly weave non-functional services, translating the outcome to a provider-specific language. As we present in Chapter 7, at a cost of an acceptable overhead, aspect-oriented weaving separates cross-cutting concerns, fostering reuse of extra code and infrastructure life-cycle automation.

In next chapter, we analyze the second contribution which focuses IaC specialization. More precisely, we propose to enhance the TOSCA matching process through the usage of the deployment context. Such contribution, combined with the TML weaving, completes the multi-cloud deployment pipeline.
In this chapter, we tackle the IaC specialization problem. As already introduced in Section 2.4, the currently available IaC approaches for multi-cloud infrastructures do not fully exploit the properties and services that each of the CSPs offers. IaC approaches [57, 82] use the so-called matching mechanism for the concrete provider resource allocation that only takes into account a "least common denominator" of the resources available across all CSPs (Section 3.2.1.1). The least common denominator introduces interoperability across resources on different CSPs, but is unable to handle the specificity of each CSP (e.g., managed services, hardware accelerators). Thus, the resulting multi-cloud architecture might not be as effective and efficient as it could with a proper deployment in a single CSP fashion of the resources. We overcome this problem by introducing a context-based matching mechanism for the allocation of the resources. More precisely, the allocation of the resources is carried out by taking into account not only the structure of the CSPs features and services, but also the concrete context and application that the multi-cloud infrastructure will serve.

As mentioned before, we consider part of the context all pieces of information that "can be used to characterize the situation of a participant in an interaction", following the definition given by Dey [39]. In the multi-cloud, the context is represented by (1) the consumer requirements or the consumer context, (2) the CSP available services or Provider context and (3) the knowledge on how a particular service can meet a consumer requirement, the "broker" context.

Context-based matching is a fundamental building block that can be used to (1) select the most suitable "implementation" for a specific resource or component in a specific CSP context, (2) select the most compelling set of CSPs in multi-cloud construction and (3) translate the abstract resource definition in a concrete one. This allows for the design of a specialized IaC, which fulfills the requirements of the underlying use-case, and, thus, for an optimal and more effective usage of CSP resources. The contribution to IaC specialization problem presented in this chapter concerns all the steps of multi-cloud construction as presented in Figure 4.1. The Context-based matching represent the basic mechanism which is capable to collect consumer parameters leveraging them to influence the construction of the multi-cloud infrastructure.

The chapter is structured as follows. First, we introduce the "least-common denominator" problem. Second, in Section 4.2 we present our approach, the "context-based" matching, describing the concept, the mapping to TOSCA of context matching and how it can be leveraged in concretely multi-cloud construction steps. Third, in Section 4.3, we present the MANTUS multi-cloud builder design, detailing its workflow and how it can integrate the context-matching in its workflow.
4.1 LIMITATIONS OF TOSCA MATCHING

Multi-cloud infrastructures come to the field at the price of the loss of specialization in the cloud resources definition. In fact, most popular approaches, such as MCLs [49, 57, 62], enable multi-CSPs resources allocation in a provider-agnostic fashion. As we introduced in Section 2.4, the great advantage of MCL is represented by the possibility to build infrastructure resources on different CSPs without requiring the cooperation of underlying cloud. Such approach is quite effective to obtain an interoperability layer but it struggles to optimize resources allocation beyond services which are available on every CSP, which represent de facto the "common denominator". In a nutshell, this means that the resources are deployed in the same generic manner without taking into account specificities of each CSP. For example, certain CSP offers managed service to handle a specific task (e.g. firewalling) which are more interesting in terms of pricing and performance than constructing the same appliance using the always-available IaaS resources.

MCLs rely on a single formalism to express "least common denominator" [42] resource definition. For example, as we already mentioned in 3.2, the TOSCA framework fosters reuse of defined components and the interoperability leveraging a mechanism of "matching" abstract "NodeTypes", components of the applications with "concrete" ServiceTemplate implementations.

However this matching process which allows to map equivalent implementations of the same component on different CSPs does not take benefit of the inputs and the parameters expressed by the consumer in order to select the most compliant implementation. This leads to the two following limitations.
First, the most straightforward limitation is about specialization. The cloud consumer cannot express her requirements in terms of services and resources from the CSP and, consequently, cannot obtain the most suitable implementation for the given applications. For instance, some consumers may not be able to dynamically benefit from specific needed services such as on “hardware accelerators” (e.g. GPU, FPGA) or CSP-managed services (i.e. DBMS as a service).

The only way to get rid of this limitation is to manually set up a specific implementation available on a certain CSP [57] that the consumer wants to adopt. This would result in a different infrastructure code specification for each underlying CSP and would not scale the amount of work to be done to set up a multi-cloud infrastructure.

Second, the cloud consumer cannot directly customize the “matching” process. TOSCA does not support the possibility to optimize the “matching” process w.r.t. a specific criteria (e.g. Cost, Performance, Latency). As analyzed by [26], the TOSCA matching is only capable to select compelling implementations which respect at different level a "structure" defined by the NodeType to replace.

For instance, the consumer wants to select which class of storage to use. Cloud Providers have different offers which are different in terms of geo-replication, performance and pricing. For the development phase, the consumer may want to rely on cheapest offers, while she may want to change it to premium when moving towards the production phase. From the TOSCA perspective, the implementations to be matched are equivalent, since they functionally implement the same service. The simple approach consisting of using NodeType attributes to filter out which implementation to match (e.g, storage type variable equal to Standard) would require the consumer to know in advance which CSP features are available. This is in conflicts with the requirement which states that the consumers have not to know in advance the CSP they will use, as presented in 4.1.1.

Furthermore, the consumer would need an in-depth knowledge of CSP-specific ecosystems while building her multi-cloud architecture. CSPs are today proposing multiple region/data-centers where to deploy customer workloads, which may differ a lot in terms of hardware configurations and the available services. Therefore, the consumer has to know each specific region of each provider in order to correctly choose the best one matching its requirements. We may classify those enhanced services/accelerators in two classes:

- **Hardware accelerators**: Dedicated hardware which can be exposed to the consumer to better perform specific workloads (FPGAs [29] GPUs, CPU features)

- **Enhanced services**: Common applications components (e.g.; DBMS) may be offered as managed services, since they offload the consumer of the administration burdens and reduce costs. Another important example is represented by "serverless computing" [16] and managed resource monitoring.

Such approach makes multi-cloud not affordable for many organizations and represents an important barrier to initial adoption. Going to multi-cloud would mean to be stick to the least common denominator maximum, losing a part of specific functionalities.

In next section, in order to concretize the multi-cloud construction scenario, we characterize different parameters the consumer specifies up front a multi-cloud deployment by leveraging the use-case (Section 1.1).
4.1.1 Use-Case Requirements

To better characterize the consumer requirements w.r.t. the multi-cloud construction we refer to the CDO cloud-bursting use-case, presented in Section 1.1. As we mentioned, CDO have to tackle two major shortcomings while moving to the cloud. First, Quality of Protection (QoP) [53] and geolocation constraints may not be achieved at the same time by one public cloud provider. For instance, the management of health records may require to perform computations over data stored within different public cloud providers (e.g. to draw statistics on certain diseases and patients). These computations need to be performed in such a way that only the result of the operation is known, while the inputs from each cloud providers are kept private. This is necessary in order to maintain confidentiality of the patients involved. Consequently, according to her needs, the consumer should provide the following architecture parameters:

- **DC**: minimal number of data-centers to use for the construction of the multi-cloud, which belong to any provider.

- **P**: minimal number of distinct providers. For instance, MPC Secret-sharing compliance requires a multi-CSP infrastructure to validate the secret sharing security model.

- **C**: minimal number of distinct continents/country where to deploy its multi-cloud. This parameter may be used to assure that requirements in terms of geo-replication (e.g. access for all consumers worldwide).

- **Forbidden Entries**: Blacklist of countries, regions, providers that should not be considered eligible for the multi-cloud.

- **SLA filters**: SLA conditions that should be at least guaranteed by underlying CSPs (e.g; Minimal Guaranteed Availability).

Second, a CDO has normally strict requirements w.r.t. infrastructure protection. In particular, a CDO may require to meet a given security standard (e.g. PCI, HIPAA) from underlying providers. Furthermore, it would require to optimize its infrastructure w.r.t the class of tasks it has to execute. For example, in the case of MPC performance boost may impact directly the acceptability of execution time. The use would express several parameters concerning multi-cloud infrastructure:

- **Base Infrastructure**: The basic infrastructure type that the consumer want to be deployed (e.g.; OpenStack, Kubernetes, Mesos/Marathon). Implicitly, in a certain percentage, the choice of infrastructure services would influence the preferred workloads, since infrastructure service performance can benefit from different CSP-specific features.

- **Additional Services**: Specific services to add to the base infrastructure to assure specific requirements from the consumer (i.e. have the same infrastructure as in its private cloud). As example of consumer-defined extra services, we can consider services for security/performance monitoring or auditing.
• **Optimization Criteria**: Depending on the budget allocated or the criticality of the deployment, the consumer may want to let adaptation be driven by the maximization/minimization of some criteria (e.g.; maximize performance, minimize cost, maximize performance/cost ratio, minimize deployment time) to satisfy different requirements (e.g. minimize deployment time for fast recovery after a fault).

• **Preferred Workloads**: The consumer may discretize the set of workloads it would run indicating in percentage their distribution. For example, a consumer would like to select best features of selected CSPs in order to perform common workloads (e.g.; Machine-learning, Data Mining, Web application, DBMS) which can prefer a specific service implementation. Those specifications combined with the optimization criteria will be used by the broker to optimize the virtual infrastructure adaption and provider choice.

To sum up, consumer is capable to expose to the IaC framework a wide set of details about its workload, the way it want to optimize it and the required filters for the CSPs. In next section, we present the context-based matching, which leverages those input to generate a specialized IaC code, analyzing available CSP and service implementations.

4.2 **CONTEXT-BASED MATCHING**

The objective of this section is to introduce the code specialization in "Infrastructure as code" on a multi-cloud. To introduce the context-based matching, we first present the fundamental elements of TOSCA matching, formalizing the qualitative description presented above. Second, we model the scenario of multi-cloud construction, identifying the actors involved and their "inputs" to the multi-cloud construction. Third, we introduce the Context-based matching algorithm and the TOSCA formalism extension to support it. Fourth, we model the different contexts that leverage the TOSCA extension. The outcome of the context-based matching is a consumer-driven choice of available implementations. Leveraging the consumer inputs defined in thr above mentioned section, we define the characteristics of a consumer workload and CSPs configuration that have to be considered during the code specialization process.

4.2.1 **Traditional TOSCA matching and extensions**

In Section 3.2, we presented the fundamentals of TOSCA templating language, introducing also the Matching mechanism. In this section, we highlight two formal definitions of the matching mechanism, exact matching and plug-in matching. We focus in particular on how those can perform the matching process and which limitations exist in a multi-cloud scenario. Simplifying the model, this allows to make concrete the definition of application NodeTypes by replacing them to an optimized compatible ServiceTemplate. We recall that such substitution process is what makes the TOSCA code really agnostic to CSPs. It also makes it a common platform to identify cloud resources on different providers, like other popular languages such as Terraform.

The "exact" matching was firstly introduced in the TOSCA primer [112], and formalized in [26].

**Definition 1 ([26]).** A ServiceTemplate $S$ exactly matches a NodeType $N$ ($S \equiv N$) iff:
In Figure 4.2, we show an example of exact matching. In a nutshell, the candidate ServiceTemplate presents exactly the same Properties, Operations, Requirements and Capabilities as specified in the NodeType. We briefly recall that Requirements and Capabilities are the most important features of NodeTypes since they describe the relationships between different NodeTypes. They model, respectively, the requirements that a certain NodeType/ServiceTemplate necessitates to see satisfied (e.g.; for instance, a VM have to rely on a disk) and the capabilities that a NodeType/ServiceTemplate has to offer (e.g. a disk can provide storage to a VM).

However, "exact" matching does not accept as viable candidates the ServiceTemplates which are "semantically" compliant with the NodeType, but do not expose exactly the same properties, operations, capabilities and requirements. This implies that compatible but more "rich" candidates, which have less Requirements and offer more Capabilities will not be considered as eligible. To handle this, in [26], the authors introduced a new definition of the matching, which remains compliant with the matching TOSCA principles while relaxing the acceptance criterias:

**Definition 2** ([26]). A ServiceTemplate $S$ plug-in matches a NodeType $N$ ($S \simeq N$) iff:

1. $\text{Reqs}(S) \equiv_R \text{Reqs}(N)$ and
2. $\text{Caps}(S) \equiv_C \text{Caps}(N)$ and
3. $\text{Pols}(S) \equiv_{P_0} \text{Pols}(N)$ and
4. $\text{Props}(S) \equiv_{P_R} \text{Props}(N)$ and
5. $\text{Ints}(S) \equiv_I \text{Ints}(N)$.

In Figure 4.2, we show an example of exact matching. In a nutshell, the candidate ServiceTemplate presents exactly the same Properties, Operations, Requirements and Capabilities as specified in the NodeType. We briefly recall that Requirements and Capabilities are the most important features of NodeTypes since they describe the relationships between different NodeTypes. They model, respectively, the requirements that a certain NodeType/ServiceTemplate necessitates to see satisfied (e.g.; for instance, a VM have to rely on a disk) and the capabilities that a NodeType/ServiceTemplate has to offer (e.g. a disk can provide storage to a VM).

However, "exact" matching does not accept as viable candidates the ServiceTemplates which are "semantically" compliant with the NodeType, but do not expose exactly the same properties, operations, capabilities and requirements. This implies that compatible but more "rich" candidates, which have less Requirements and offer more Capabilities will not be considered as eligible. To handle this, in [26], the authors introduced a new definition of the matching, which remains compliant with the matching TOSCA principles while relaxing the acceptance criterias:

**Definition 2** ([26]). A ServiceTemplate $S$ plug-in matches a NodeType $N$ ($S \simeq N$) iff:

1. $\text{Reqs}(S) \equiv_R \text{Reqs}(N)$ and
In the plug-in matching, the candidate ServiceTemplate is selected if it "offers more" TOSCA capabilities/properties and "requires less" in terms of Requirements. The Plug-in matches may be defined as follows for Requirements and Capabilities:

**Definition 3 ([26]).** Let \( N \) be a NodeType and let \( S \) be a ServiceTemplate. Then: \( \text{Reqs}(S) \simeq_{\text{R}} \text{Reqs}(N) \) iff

\[
\forall r_S \in \text{Reqs}(S) \exists r_N \in \text{Reqs}(N) : \text{name}(r_N) = \text{name}(r_S) \land \text{type}(r_N) \geq \text{type}(r_S).
\]

**Definition 4 ([26]).** Let \( N \) be a NodeType and let \( S \) be a ServiceTemplate. Then: \( \text{Caps}(S) \simeq_{\text{C}} \text{Caps}(N) \) iff

\[
\forall c_N \in \text{Caps}(N) \exists c_S \in \text{Caps}(S) : \text{name}(c_S) = \text{name}(c_N) \land \text{type}(c_S) \geq \text{type}(c_N).
\]

\( \text{Propos}(S) \simeq_{\text{PR}} \text{Propos}(N) \) iff \( \text{XMLtype}(\text{Propos}(S)) \geq \text{XMLtype}(\text{Propos}(N)) \).

This two definitions introduces the concept of "offer more" and "require less", which allows to match a not exactly candidate. In Figure 4.3, we show an example of plug-in matching. In this scenario, the candidate ServiceTemplate presents an acceptable configuration in terms of Properties, Operations, Requirements and Capabilities as specified in the NodeType. In particular, it offers more capabilities that the NodeType and this does not prevent the ST to functionally replace the NodeType.

However, as we explained in Section 4.1, both of the above mentioned matching mechanisms do not allow to distinguish the ServiceTemplates according to "non functional" features. In other words, for instance, the plug-in matching does not include a mechanism to select the best ServiceTemplate between two compliant from the "offer more, require less" principle. This prevents the matching mechanism to obtain the most suitable result according to consumer requirements and, globally, a tailored multi-cloud infrastructure.

In other words, the TOSCA matching mechanism cannot be used as it is to construct the consumer-centric multi-cloud infrastructure. The reason is that it cannot be guided to have an optimized usage of the CSP services. In next sections, we better define the multi-cloud scenario and we propose an extension of the matching algorithm to introduce the optimization possibility. Such extension represents the basic block of our contribution since it allows to take advantage of the multi-cloud scenario modelization.

### 4.2.2 The Multi-cloud scenario

In this section, we characterize the multi-cloud scenario where we extend the TOSCA matching mechanism. In particular, we detail the different parties involved in the multi-cloud construction and discuss the need for a third-party, a broker, to fill the gap between consumer and CSP contexts. The aim is to present in details which "inputs" the TOSCA
matching extension should take into account to adapt different steps of multi-cloud construction to the consumer, as we show later in next sections.

To do so, we first analyze the context as all "parameters" which may influence a multi-cloud deployment but which are subject to change across deployments, coming from the provider analysis and consumer. Basically, we can recall two different "contexts" concerning a multi-cloud deployment:

- **Data-center Context.** Encapsulates a provider region, composed of the available features, "virtualizing" providers data-center configuration. Each DC specifies its company, its geo-localization and the list of services and hardware configuration available.

- **Consumer Context.** As specified in the use-case description, consumer context describes desired base-template and desired "aspects", non-functional features, to be included. For example it may want to deploy a Kubernetes infrastructure with monitoring and operating systems hardening aspects. In addition, the consumer context is also composed of hints about which workloads will run (e.g. data analytics, ) and an objective about the multi-cloud construction (cost, performance, etc).

Consequently, we observe that exists a clear semantic "gap" between the consumer context expressed and CSP context. In a nutshell, we need a sort of "knowledge" capable to "match" and transform the abstract high-level consumer "context" in a compelling IaC code capable to select and use desired features of CSP. To this end, we can recall the concept of "broker-based" multi-cloud, where a third-party has the knowledge of features of different CSPs. The idea of such "broker" differs from the traditional one, which is normally involved in the deployment process. The broker here is primarily important to construct a cartography of CSP features and available and match them w.r.t. consumer requirements. We therefore require a further "context":

![Figure 4.3: TOSCA Plugin Matching](image)
• **Broker Context.** Represents the glue between provider offers and consumer requirements. It provides a "dictionary" which is capable of mapping workloads with added values proposed by providers, and a multi-dimensional vector which maps each extra "feature" w.r.t. a set of optimization criteria (cost, performance, etc.).

In other words, the broker-context bridges the gap between CSP features and their interest for customer workloads. The construction of this "dictionary" does not strictly require the presence of a "third party" since a static broker-context may be embedded within the software responsible of building the multi-cloud. In fact, outsourcing the broker context to a third party is not always desirable for consumers, as discussed in Section 2.4.1.

However, the broker-context construction may require frequent updates to remain effective since it is influenced by a wide number of constantly changing factors (e.g. the load on CSPs due to the day of the week or the day of the year). As presented in [101], this adds complexity to the maintenance of the broker-context due to continuous data analysis to be performed to estimate the correct values for any given point in time. Therefore, if a customer wants better accuracy, the broker context should be outsourced to a third-party, which propose a dynamic and up to dated broker context as a service.

In Table 4.1, we provide a formal definitions of those contexts, which will be leveraged by the matching algorithm detailed in the next section. Such algorithm is designed to combine the three different contexts formalized here to indicate which ServiceTemplate is the most suitable. Such algorithm represents the basic block to construct the multi-cloud pipeline which is used in different steps as presented later in this chapter.

### 4.2.3 Matching Algorithm

As aforementioned, we define our context-based matching algorithm. In particular, we show how we can obtain a specialized IaC code starting from a generic service definition and the set of contexts defined above. We refer to the TOSCA matching, by extending it through the introduction of a "context" inside the TOSCA code.

Looking at the TOSCA documentation, we can define the matcher algorithm as responsible to find a concrete and acceptable ServiceTemplate for a given TOSCA Node [112]. Recalling the TOSCA matching mechanism and its extensions presented in Section 3.2, this mechanism allows to specialize an abstract service definition with compelling concrete ServiceTemplate leveraging the "broker-context" defined above.

The objective we want to achieve is represented by the substitution of a generic resource or service abstraction, expressed as NodeType, with a compelling ServiceTemplate of it. We introduce a formalization of this concept as follows:

**Definition 5.** Given a provider data-center $P$ with a context provider $CP$ defined as the ensemble $CP = \{nf_1, \ldots, nf_t\}$ of the non functional features corresponding to provider $P$, a ServiceTemplate $S$ contextually matches a NodeType $N$ ($S \sim N$) iff:

1. $Reqs(S) \simeq_p Reqs(N)$ and
2. $Caps(S) \simeq_c Caps(N)$ and
3. $Pols(S) \equiv_{PC} Pols(N)$ and
4. $Props(S) \simeq_{PR} Props(N)$ and
## Table 4.1: Context Matching Model

<table>
<thead>
<tr>
<th>Component Source Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of all available provider features</td>
<td>Broker Set</td>
</tr>
<tr>
<td>( P = p_1 \ldots p_l ), ( l &gt; 2 )</td>
<td></td>
</tr>
<tr>
<td>List of all optimization criteria</td>
<td>Broker Set</td>
</tr>
<tr>
<td>( C = c_1 \ldots c_l ), ( m &gt; 2 )</td>
<td></td>
</tr>
<tr>
<td>List of all workloads and aspects to weave</td>
<td>Broker Set</td>
</tr>
<tr>
<td>( W = w_1 \ldots w_n ), ( n &gt; 2 )</td>
<td></td>
</tr>
<tr>
<td>Provider Context</td>
<td>Consumer Vector</td>
</tr>
<tr>
<td>Provider implements features ( i ) ( \in {0,1} ) if the selected provider supports feature ( i )</td>
<td></td>
</tr>
<tr>
<td>Provider Context</td>
<td>Consumer Vector</td>
</tr>
<tr>
<td>List of all available provider features</td>
<td>Broker Set</td>
</tr>
<tr>
<td>( P = (p_1 \ p_2 \ldots p_l) ), ( p_i \in {0,1} ) if ( P_i ) supports feature ( i )</td>
<td></td>
</tr>
<tr>
<td>Consumer Context</td>
<td>Consumer Vector</td>
</tr>
<tr>
<td>Consumer desires workloads and aspects weaved ( \in {0,1} )</td>
<td></td>
</tr>
<tr>
<td>Consumer Context</td>
<td>Consumer Vector</td>
</tr>
<tr>
<td>Provider implements features ( i ) ( \in {0,1} ) if the selected provider supports feature ( i )</td>
<td></td>
</tr>
<tr>
<td>Available features ( d ) ( \in {0,1} ) if ( \exists j ) ( d_j = 1 )</td>
<td></td>
</tr>
<tr>
<td>Consumer Context</td>
<td>Consumer Vector</td>
</tr>
<tr>
<td>Provider implements features ( i ) ( \in {0,1} ) if the selected provider supports feature ( i )</td>
<td></td>
</tr>
<tr>
<td>List of all available provider features</td>
<td>Broker Set</td>
</tr>
<tr>
<td>( \exists i \in {0,1} ) ( i \in {0,1} ) if ( \exists j ) ( i_j = 1 )</td>
<td></td>
</tr>
<tr>
<td>List of all available provider features</td>
<td>Broker Set</td>
</tr>
<tr>
<td>( \exists i \in {0,1} ) ( i \in {0,1} ) if ( \exists j ) ( i_j = 1 )</td>
<td></td>
</tr>
<tr>
<td>Consumer Context</td>
<td>Consumer Vector</td>
</tr>
<tr>
<td>Provider implements features ( i ) ( \in {0,1} ) if the selected provider supports feature ( i )</td>
<td></td>
</tr>
<tr>
<td>List of all available provider features</td>
<td>Broker Set</td>
</tr>
<tr>
<td>( \exists i \in {0,1} ) ( i \in {0,1} ) if ( \exists j ) ( i_j = 1 )</td>
<td></td>
</tr>
<tr>
<td>List of all available provider features</td>
<td>Broker Set</td>
</tr>
<tr>
<td>( \exists i \in {0,1} ) ( i \in {0,1} ) if ( \exists j ) ( i_j = 1 )</td>
<td></td>
</tr>
</tbody>
</table>
4.2 CONTEXT-BASED MATCHING

\( (5) \) \( \text{Ints}(S) \sim_i \text{Ints}(N) \) and

\( (6) \) \( \text{nf}_i(S) \geq_i \text{nf}_i(CP), \) for \( i = 1, \ldots, t. \)

To simplify the understanding, this matching process may be seen as a "job market" where the abstract service definition, the TOSCA Node to be matched, opens a new "position". The "candidates", different alternative ServiceTemplate implementations, would be chosen for the "job", replacing the Node code, if and only if they fit the open-position "criteria" (e.g.; same set of exported TOSCA capabilities and requirements).

In other words, to continue the "job market" analysis, we introduce a further characterization about the implementations of the service, the ServiceTemplates, which is represented by the selection of the "candidates". On the one hand, the traditional "matching" considers only if a specific implementation is capable to match requirements in terms of TOSCA features (e.g. capabilities, requirements, properties). On the other hand, we enrich the scope introducing the context of the deployment and through the "broker" context, the mandatory and optional features to be satisfied in the context of a specific implementation. Overall, the new matching algorithm evaluates compatible implementations in order to maximize the rating of a chosen TOSCA implementation according to a consumer-defined non-functional criteria (cost, latency, performance).

We formalize the Context-based matching algorithm in Listing 1. Driven by to the consumer specified workloads, the context-base matching algorithm leverages the consumer workload specifications to choose a certain implementation and fill the vacancy. More concretely, the comparison among all different implementations are performed by maximizing/minimizing score after having weighted the features available w.r.t the consumer-defined workloads. Each candidate ST which is acceptable according to the PluginMatching, is then evaluated w.r.t. the score it can obtain with the Data-center context. If the non functional features nf of ST are not fulfilled by the data-center context, the ST is discarded. Otherwise, the score of the compatible ST is computed accessing to the broker context BC and the most suitable according to the criteria C is taken.

In other words, to continue the comparison with the job market, not all skills have the same importance and the employer may orient its choice through different criteria, have excellence/performance or cost constraints.

In next section, we focus on the evaluation function which is included in the algorithm to rate different ServiceTemplate implementations in order to find the best one w.r.t. consumer context.

4.2.3.1 ServiceTemplate evaluation

In this section, we discuss how to evaluate a specific implementation quantitatively in order to compare different ServiceTemplates w.r.t. different consumer criteria.

As we presented in Section 4.1.1, the consumers define a set of criteria (e.g. efficiency, cost, flexibility) they are interested in maximizing or minimizing for its multi-cloud. Given a consumer’s criteria \( C \), and a node \( N \), a score \( S_C^N \) for node \( N \) w.r.t. criteria \( C \) has a value between 0 and 1 describing how much (or how little) a node fulfills that criteria when implementing the non-functional features proper of the aforementioned workloads. Clearly, when a criteria has to be maximized, if \( S_C^N \) is 1 or close to 1, then this means that node \( N \) implements the desired workloads maximizing criteria \( C \) and should be selected by the consumer to be part of the multi-cloud infrastructure.
**Data:** WLs, nf(CP), BCi, List of STs candidates, NodeType to match, C

**Result:** Selected ST

```plaintext
if C.fun == "Max" then
    max_score = 0;
else
    min_score = MAXINT;
end
selected_candidate = ∅;
while ListST not empty do
    l = get_ST(ListST);
    valid = 1;
    isValidST = PluginMatching(l, N);
    if PluginMatching(l, N) == True then
        foreach i in nf(CP): do
            if nf(CP) < nf(l) then
                value = 0;
                break;
            end
        end
    end
    if valid == 1 then
        score = computeResult(WLs, BCi, l);
        if C.fun == "Max" then
            if score > max_score then
                max_score = score;
                selected_candidate = l;
            end
        else
            if score < min_score then
                min_score = score;
                selected_candidate = l
            end
        end
    end
return(selected_candidate);
end
```
Vice versa, if $S^N_C$ is 0 or close to 0, then this means node $N$ implements the desired workloads minimizing criteria $C$ and should not be selected to be part of the multi-cloud infrastructure. The other way round holds when a criteria has instead to be minimized.

It is the broker context that computes the score $S^N_C$ and that discards the nodes that do not fit the desired criteria. Denoted as $WL$ the amount of workloads that the consumer wants the yet-to-be-built multi-cloud infrastructure to perform and denoted $L$ as the amount of non-functional features provided by a node $N$, we define the score $S^N_C$ of node $N$ w.r.t. criteria $C$ as follows:

$$S^N_C = \sum_{i=0}^{WL} \sum_{j=0}^{L} \frac{BC(i, j, C) \cdot nf_j(S) \cdot nf_j(N)}{WL},$$

(4.1)

where $BC(i, j, C)$ is referred to as the broker-context weight of workload $i$ to perform the non-functional feature $j$ w.r.t. the criteria $C$. In other words, $BC(i, j, C)$ is a value between 0 and 1 that the broker-context assigns to a certain node $N$ to quantify how well it provides the non-functional feature $j$ w.r.t. criteria $C$ when carrying out workload $i$. The terms $nf_j(S)$ and $nf_j(N)$ can be 0 or 1 depending on whether a certain non-functional feature is, respectively, desirable for the ServiceTemplate and provided by the node $N$ in question.

In the following, we discuss the complexity time to implement the TOSCA context-based matching algorithm presented above. This is based on the complexity analysis done for the TOSCA matching algorithm presented in [106]. In fact, their algorithm is similar to the one we propose except for the fact that the context is not considered. Denoted as $TM$, the TOSCA matching algorithm has the following upperbounded complexity:

$$T_{TM} = O(2^r),$$

(4.2)

where:

- $r$ is the number of the available ServiceTemplate (ST) candidates;
- $t$ is the number of all potential features proper of the TOSCA algorithm (i.e. capabilities, operation, requirements, properties) to be matched to a Node $N$.

Note that they should not be confused with the functional and non-functional features of cloud applications. Equation 4.2 is the complexity of implementing the TOSCA matching algorithm taking into account the worst-case scenario, where all or almost all of the available features must be implemented and there is plenty of ST candidates that are suitable for a given node. However, as the authors of [106] pointed out, what happens in practice is that (1) the number of effectively required features $n$ is negligible compared to the set of all available features $t$ and (2) the number of ST candidates which cannot be matched is a lot greater than the matching ones. They proved that, under these two assumptions, the time complexity of the TOSCA matching algorithm is on average linear in the amount of features:

$$T_{TM} = O(rt).$$

(4.3)
As pointed out in Definition 5, the TOSCA context-based matching algorithm that we propose in this paper is substantially equivalent to the TOSCA matching algorithm. The only difference is that the set of available features \( t \) is enlarged because also the non-functional features corresponding to a given context are added. Thus, the introduction of the context-based analysis does not modify the effective linear complexity of the matching mechanism. This means that Equation 4.2 provides a description of the complexity of the TOSCA context-matching algorithm for the worst-case scenario. Furthermore, assumptions (1) and (2) hold also for our case and thus we can conclude that Equation 4.3 displays the complexity of the TOSCA context-based matching algorithm for the average-case scenario.

4.2.3.2 TOSCA Mapping

The TOSCA mapping of the contexts above is done as follows. The Matching Process involves two different objects: first, the ServiceTemplate, i.e. the "candidate" resource sub-graph, and second, the NodeType, i.e. the abstract resource specification we want to replace.

For the NodeType, the context is encapsulated as an extra property of the \texttt{node\_filter} section (Listing 4.1). TOSCA NodeTypes specifies inside the \texttt{node\_filter} section the list of features that a selected data-center. Such embedding allows to preserve all the information related to the state of the deployment inside the TOSCA NodeType code, avoiding to disperse the state outside the code.

It is worth noting that TOSCA context is used to encapsulate non-functional features only.

```
remote_provisioner:
  type: orbs.boxes.Provider
  properties:
    # [...] Omitted for brevity
node_templates:
  [ ... ]
  overcloud_1:
    type: orbs.boxes.Overcloud
    properties:
      # [...] Omitted for brevity
node_filter:
  properties:
    context:
      contains:
        # Provider Region Context
        HighIO: False
        ServiceFunctionChaining: True
```

Listing 4.1: TOSCA NodeType context embedding example

For the candidate ServiceTemplates, the context-related information are introduced in the TOSCA substitution mapping template section (for instance, as presented in listing 4.2). In other words, every "candidate" ServiceTemplate specifies which features they require to be effectively deployed in the substitution mappings section.

The matcher retains only implementations which features requirements are fulfilled by the CSP context, presented in \texttt{node\_filter} section.

```
node_templates:
  [ ... ]
compute_node:
  type: orbs.nodes.Compute
  properties:
    # [...] Omitted for brevity
```
Mapping the above principles to TOSCA can be done straightforwardly. The context property is part of the root resource of TOSCA resource hierarchy and it is inherited by all TOSCA implementations. In Listing 4.1, we show how the "candidate" context is injected in TOSCA files at the first step (Overcloud NodeType) in node filter section. During the matching iterations, the context is further refined by different selected implementations as showed in 4.2, where "requested features" are demanded in substitution_mapping section.

It is worth mentioning that this approach enables the possibility to automatically build the "list" of required features to be showed in substitution_mapping section for a specific infrastructure template, following a bottom-up approach where each layer has to require at least features wanted by every candidate to replace a node of its node_template subgraph.

In Listing 4.3, we provide a sample description of the DC candidate profile, where all available services and features are listed.
Listing 4.3: Provider Context example (CSP1 US East 2 Region)

Their presence/absence is determined by context-matching leveraging the broker context (example in Listing 4.4). The context example presents simple evaluations for each valuable features according to various criteria w.r.t. different workloads.

Listing 4.4: Broker Context example

### Workloads:
- **Compute optimized**
  - Cost: 0.3
  - Performance: 0.15
- **Storage optimized**
  - Cost: 0.1
  - Performance: 0.15
- **Memory optimized**
  - Cost: 0.1
  - Performance: 0.15
- **'Fast-NIC'**
  - Cost: 0.2
  - Performance: 0.1
- **'Compute optimized'**
  - Cost: 0.1
  - Performance: 0.15
- **Fast-NIC**
  - Cost: 0.1
  - Performance: 0.1
- **SSD**
  - Cost: 0.1
  - Performance: 0.2

4.3 **FROM ABSTRACT TEMPLATE TO MULTI-CLOUD DEPLOYMENT**

In this section, we introduce a multi-cloud builder workflow and present how previously introduced contributions, i.e. AOP-based weaving and Context-based matching, are integrated. We call **MANTUS** the multi-cloud builder. As we detailed in Section 4.1, the idea behind the **MANTUS** multi-cloud builder is to introduce the possibility to build an optimized overlay and consumer-customized multi-cloud by completely decoupling consumer specifications from provider technologies (e.g. CSP Orchestration language) and the available features.

To illustrate how **MANTUS** builds a multi-cloud, we show here the fundamental steps required from consumer parameters definition to the multi-cloud deployments (Figure 4.4). In other words, we focus on how **MANTUS** adapts the generic service definition to features that are actually present on a specified provider data-center configuration. The fundamental steps are as follows:

1. **Context-based Provider Selection** During this first step, **MANTUS** simply filters out provider or single region which cannot be acceptable to be part of the multi-
Figure 4.4: Multi-Cloud Construction Workflow
cloud. This early selection prevents to execute context-based matching and AOP-based weaving on unacceptable candidates. For instance, a consumer may avoid to have its data in certain countries or cloud provider companies of a certain country due to law prescriptions (e.g. healthcare legislation). The outcome of pre-filtering step is represented by a list of provider regions eligible to be part of the multi-cloud which have to be tested by context-based provider selection. Furthermore, leveraging the ILP program described above, \textsc{MANTUS} sorts acceptable set of CSPs in order to have an optimized selection of CSP. In a nutshell, each matched TOSCA Node has received an evaluation during the matching process which measures its adequacy to satisfy consumer-defined workloads w.r.t. one or more optimization criteria. Therefore, we may define a normalized per-region score which will indicate the suitability of a certain candidate to be part of the multi-cloud. At the end of this step, \( C \) data-centers are considered as target to build the multi-cloud.

2. **Matching & Weaving** The following step recursively refines the sub-graph of TOSCA resource definitions until concrete TOSCA resources are obtained, which can be translated to CSP processable code. At the same time, this step is responsible for the integration of resources belonging to the base infrastructure with the ones of non-functional services required by the consumer. More precisely, each CSP candidate to be part of the multi-cloud provides its context to match the set of NodeTypes composing the consumer defined service. In parallel to the context-based matching, non-functional services have also to be weaved at the same time on the template. The output of the Matching & Weaving workload results are per-DC adapted TOSCA resources, composed of TOSCA concrete nodes, including non-functional services.

3. **Context-based Translation** The matched-TOSCA template of chosen DCs will then be translated to provider-specific language, leveraging hints given by matched TOSCA to select the correct context parameters (e.g.; VM Flavor). When the translation is performed, the resulting code is deployed on different CSP.

In a nutshell, as a multi-cloud “compiler”, \textsc{MANTUS} conciliates: (1) non-functional extensibility of basic templates through AOP (see Chapter 3), (2) capacity to adapt to CSP features through context-based matching, and (3) capacity to deploy services over a set of CSPs. The core step of the \textsc{MANTUS} workflow is represented by the "Matching & Weaving" step. This step integrates AOP-weaving and context-based specialization. This step can be designed through multiple different approaches, which may influence the final \textsc{MANTUS} performance and execution time, as we discuss in Chapter 7. In the next section, we provide more details about each step of the workflow.

4.3.1 **Context-based Provider Selection**

In the last section, we defined the "context-based" matching algorithm and showed how it can be used to select a specific implementation of a service by extending the "matching" TOSCA formalism. Context-matching provide an approach to select a compelling implementation in a certain CSP configuration w.r.t a consumer-specified workload. Context-matching can not only be used for multi-cloud specialization but also to select which set of data-centers of a certain CSP offer the most interesting features for a customer.
As we presented in Section 1.1, consumers often express criteria to filter out and select CSPs for building a multi-cloud architecture. Those have to be taken into account when deploying the multi-cloud and have an important impact on the service specialization obtained through the matching algorithm. In this section, we design through Integer Linear-Programming (ILP) a model to choose the best candidates while satisfying consumer constraints in terms of multi-cloud architecture.

For a single data-center, as we mentioned in Section 4.2, the score is defined as the normalized weight of matching results nodes.

For a candidate data-center \( i \) composed by \( T \) nodes, the score \( S_i \) of data-center \( i \) is defined as follows:

\[
S_i = \frac{\sum_{N=1}^{T} S_i^N}{T} \tag{4.4}
\]

Therefore, we may define an initial ILP program for \( V \) available data-center candidates as:

maximize \( F(x) = \sum_{i=0}^{V} S_i \cdot x_i \)

subject to \( \sum_{i=0}^{V} x_i = DC \quad x_i \in \{0, 1\} \) \tag{4.5}

where DC is the number of data-centers that the consumer indicated as the number of DC results. \( F(x) \) is the "objective function" of the program and models the selection of the best possible choice among available data-centers. Such solution identified for \( F(x) \) have to be compliant with the lest of constraints which compose the ILP program. The basic constraint is represented by the number of data-centers to select, taken as input from the customer. Considering the objective function, the variables \( x_i \) model the fact of having selected or not a specific data-center to be part of the multi-cloud and are booleans. Those boolean variables models that the specific data-center \( i \) is selected or not to be part of the solution (i.e. the list of chosen data-centers).

To model all different multi-cloud deployment scenarios, the ILP program is further refined adding specific set of constraints or coefficient manipulations to map extra consumer-defined requirements, which apply in many customers use-case:

- **Diversity constraints**: map the necessity for a consumer to accept a multi-cloud only if it spans at least a certain number of providers or geographical locations. In other words, those constraints impose that among all subsets \( G \) of data-centers, at least \( P \) from different providers/subsets are taken. Each subset \( j \) is composed of \( D_j \) data-centers and if one of them is chosen, than the whole subset \( j \) is automatically chosen. The presence of a data-center in a subset may be represented by a 1 or a 0 coefficient expressing the constraint. Therefore: Given:

\[
G_j = g_0 x_0 \lor g_1 x_1 \lor ... \lor g_{DC} x_{DC}, g_j \in \{0, 1\} \tag{4.6}
\]
The constraint may be modeled as:

$$\sum_{j=0}^{G} \left( g_{0j}x_0 \lor g_{1j}x_1 \lor \ldots \lor g_{DCj}x_{DC} \right) \geq P \quad (4.7)$$

Leveraging “inclusive or” constraints translation [27], we would leverage the “truth table” of the expression to generate the set of constraints to add in the problem.

- **Private Clouds constraints** models on premise resource where the consumers have full control and they have already spent in advance for their deployment. Consequently, consumers want to use them in any multi-cloud infrastructure. In order to accept only solutions which adopt private cloud $x_i$, it is sufficient to add one constraint for each private data-center to be mandatory used $x_i = 1$, with coefficient 0 for any other data-center.

- **Correction coefficient** a broker which knows a specific advantage in terms of the selecting criteria (e.g. commercial partnership with a specific CSP, performance advantage for a certain data-center) may simply manipulate the coefficients of data center $i$ multiplying $S_i x_i$ with a correction factor $c_i$ in the objective function. This has the effect to raise or low the $x_i$ weight, increasing the possibility of the associated DC to be selected as part of the solution.

Those constraint generation may be applied similarly to geographical diversity (i.e. the necessity for a customer to have an instance deployed in many different continents), generating simply different coefficients of ”participation” to other groups of variables, but modeling constraints in a similar way.

The outcome of the optimization process is represented by an array of $x_i$, where $x_i = 1$ if the corresponding data-center $i$ is selected and $x_i = 0$ if the corresponding data-center $i$ is not selected to build the multi-cloud infrastructure. The array of $x_i$ which maximize at most the objective function $F(x)$ is retained as the output of the selection, and all enabled data-centers (i.e. which have their variable equal to 1) will be part of the multi-cloud construction. As we show in Figure 4.4, the solution taken as outcome for this step is then "splitted" and processed by the following steps of the MANTUS pipeline. More precisely, as we describe in next section, the context of each selected provider is "plugged" to the TOSCA ServiceTemplate representing the abstract infrastructure of the multi-cloud to perform the ”matching & weaving” step.

### 4.3.2 Matching & Weaving

In this section, we propose a more detailed analysis of Context-based matching and weaving, which encapsulates the two major contributions for extensibility, AOP-based weaving and specialization. In particular, we analyze how those two contributions can be leveraged together to obtain an extensible but specialized multi-cloud architecture.

In fact, how to combine together AOP and MDE may impose important changes to the way the multi-cloud is constructed. In particular, the key problem is represented by how we can design non-functional aspects (i.e. TML scripts) which can be meaningful for the types of TOSCA resources we are dealing with. On one hand, TOSCA matching transforms abstract NodeTypes in refined ServiceGraph with fine-grained NodeTypes.
On the other hand, TML scripts search into TOSCA graph to meet "triggering conditions" which are represented by properties, connection and other structural properties of the graph. The question to answer is how to express meaningful TML scripts which verifies properties on a graph which is designed to evolve during matching.

More precisely, a straightforward answer that we adopted in early prototypes, that we present in the Chapter dedicated to implementation, is that TML scripts are written to verify properties only on very low-level resources (i.e. results of TOSCA matching). This would imply that consumers have to build their TML scripts on low level resources and this may introduce a further complexity to the multi-cloud construction.

First of all, to analyze possible solutions, we observe that TOSCA matching represent a form of Model-Driven Engineering (MDE) code generation. MDE consists in generating code through well-documented process starting from standardized models. Considering TOSCA, we can consider the matching as a MDE process that relies on NodeType models to generate, through automatic transformation, concrete implementations of the same abstract types. This can be useful since the combination of MDE and Aspect-Oriented programming was widely studied in other works [77].

As a reminder, in Section 3.3, we introduced Aspect-Oriented weaving in TOSCA to extend the base infrastructure with non-functional services. More precisely, we model those aspects as TOSCA Manipulation Language (TML) scripts.

In particular, those approaches can be classified in three main classes. We briefly introduce them in the next paragraph, detailing potential adoption consequences in the case of MANTUS workflow and their consequent impact in terms of performance and development efforts:

- **Weave-then-Generate**: performs the weaving process before the code generation. In MANTUS, this approach consists of injecting non-functional features before doing the generation, just after the dispatching of selected infrastructure services over multiple CSP. Then, injected abstract services are matched following the context of the CSP region. This approach is complex to be adopted before matching and TOSCA resource graph does not expose details about the TOSCA graph before the weaving topology. This prevents the possibility for the weaver to have visibility on resources properties and interconnection. Adopting this methodology would result in a more complex TOSCA matching, since all acceptable combinations of aspect weaved should be covered by a different TOSCA matching choice.

- **Generate-then-Weave**: consists in performing the generation of code of aspects and infrastructure code separately, based on the same context. Consequently, in our case, context-matching should be performed not only on TOSCA template resources but also on TML scripts. At the end of the matching phase, the results would be weaved together, leveraging the action field of TML scripts. Traditionally, this approach has the advantage of simplifying error tracing during code generation, since the delimitation of code responsibility between aspects and base model is clear when weaving is done late [77]. In addition, the late injection prevents early detection of errors in connection and the possibility of injection of high-level of abstraction. However, from an implementation perspective, this method requires the development of two distinct matchers (one for TOSCA, one for TML) increasing significantly the effort required in engineering and maintenance.
• **An Hybrid Approach:** which may have an acceptable compromise in terms of visibility of all different layers of abstractions and with an acceptable implementation effort. In this approach, MANTUS implements an iterative “Match & Weave” loop. When NodeTypes are matched with ServiceTemplates, MANTUS tests the resulting code “against” any available aspect. If the mandatory "filtering" conditions are met, the condition requested to apply the injection. The possibility of having an iterative application of matching allows to inject new sub-STs across the entire template, at any level of abstraction. However, multiple injections may represent a risk if a certain condition is still valid even after the "injection", introducing unwanted multiple injections. Therefore, such approach would require a more complex TML rules definitions, offloading the burden on the weaver and the developer. To limit this we only accept rules which are idempotent.

To sum up, two main designs, Generate-then-Weave and Hybrid, are interesting for this step, having both advantages and drawbacks. We will discuss performance implications of those design in the follow up of this chapter: in Section 7.4, we discuss the implementation details of both alternatives, analyzing performance evaluations in Section 7.4.1

### 4.3.3 Context-based IaC translation

In this section, we present the per-provider translation process, detailing different steps and the expected outcome. More precisely, we focus on the role of "context" as defining a compelling translation of TOSCA resources on a specific CSP.

After having obtained a compelling TOSCA implementation and filtered the most adapted CSP for a consumer-defined workload, the IaC code has to be deployed on different CSPs in order to perform effectively the deployment.

The usage of a CSP-agnostic formalism poses severe concerns about how such "abstract" code has to be used to trigger deployments. Resource implementations on different CSPs (e.g. Virtual Machines) may vary a lot according to the parameters and attributes associated to its declaration. For instance, a specific tag which defines the "flavor" of the VM may define the class of storage and CPU used, dramatically changing performance w.r.t. a specific use case.

In a similar scenario, the "context-based" translation works as follows. The above mentioned semantic continuum between different CSPs enables the possibility to build the well-known set of base resources that are considerable always available and that may allow to define an "always-working" implementation. Matched "implementations" are translated according to the CSP capabilities, which is guaranteed to be feasible from the matching selection step. However, it is important that the translation logic is aware of existing CSP services to perform the correct "translation". In fact, leveraging "context" hints expressed in TOSCA nodes, the translation logic may infer the correct provider-specific property value (e.g. VM instance type, Size) which fits at best the selected implementation. In other words, context-based translation ensures that matched TOSCA types are translated to effective provider-specific language types.

Discussing the feasibility of such approach, IaaS resources offered by different providers have specifications but often recover similar semantics, at least for fundamental low-abstraction resource types (e.g.; VM; Block Storage, Object Storage, Virtual Networks). From our analysis, no semantic differences are present for computing and storage resources. Conversely, all providers offer primitives to traditional networking, allowing
<table>
<thead>
<tr>
<th>ORBITS</th>
<th>AWS CloudFormation</th>
<th>OpenStack Heat</th>
<th>MS Azure Resource Manager</th>
<th>Google Compute Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported Format</td>
<td>JSON/YAML</td>
<td>JSON/YAML</td>
<td>JSON</td>
<td>YAML</td>
</tr>
<tr>
<td>Resource basename</td>
<td>AWS::*</td>
<td>OS::*</td>
<td>Microsoft.*</td>
<td></td>
</tr>
</tbody>
</table>

**Tosca Types**

<table>
<thead>
<tr>
<th>Compute</th>
<th>EC2::Instance</th>
<th>Nova::Server</th>
<th>Compute/virtualMachines</th>
<th>compute.v1.instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>AMI</td>
<td>Glance::Image</td>
<td>Storage/images</td>
<td>compute.v1.image</td>
</tr>
<tr>
<td>Storage</td>
<td>EC2::Volume</td>
<td>Cinder::Volume</td>
<td>Storage/</td>
<td>compute.v1.disk</td>
</tr>
<tr>
<td>Networking</td>
<td>EC2::EIP</td>
<td>Neutron::FloatingIP</td>
<td>Network/publicIPAddresses</td>
<td>compute.v1.globalAddress</td>
</tr>
<tr>
<td>Networking</td>
<td>EC2::VPC</td>
<td>Neutron::Net</td>
<td>Network/virtualNetworks</td>
<td>compute.v1.network</td>
</tr>
<tr>
<td>Networking</td>
<td>EC2::Subnet</td>
<td>Neutron::Subnet</td>
<td>✓</td>
<td>compute.v1.subnetwork</td>
</tr>
<tr>
<td>Networking</td>
<td>EC2::Route</td>
<td>Neutron::Router</td>
<td>Network/routeTables</td>
<td>compute.v1.router</td>
</tr>
<tr>
<td>Networking</td>
<td>EC2::VPCGatewayAttachment</td>
<td>Neutron::RouterInterface</td>
<td>Network/virtualNetworkGateways</td>
<td>compute.v1.globalForwardingRule</td>
</tr>
<tr>
<td>Networking</td>
<td>EC2::NetworkInterface</td>
<td>Neutron::Port</td>
<td>Network/networkInterfaces</td>
<td>compute.v1.address</td>
</tr>
<tr>
<td>Networking</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Networking</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Networking</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
to deploy virtual networks. However, most of them do not offer primitives for traffic steering and service chaining. Those abstraction are particularly useful for private clouds which may rely on middleboxes, appliances that normally perform transparently a specific task on processed traffic (e.g.; Intrusion Detection System). In table 4.2, we provide a list of basic low-level TOSCA resources mapping in provider-specific orchestration language, and we mapped them with an equivalent object in several popular CSP orchestration language.

### 4.4 Conclusion

In this chapter, we presented the "context-based’ matching. Context-based matching enables the possibility to reify an abstract TOSCA-based generic resources in an optimized concrete implementation, taking in account consumer parameters, CSP data-center configuration and broker context. We extended the TOSCA matching principle to deal with context respecting the TOSCA formalism, extending the meaning of existing TOSCA template sections.

To present this, we provided a formal definition and we showed its adoption in several steps of multi-cloud construction. We presented MANTUS multi-cloud builder which implements context-based design principles and how they tackle use-case requirements. In table 4.3, we summarized how MANTUS is capable of considering consumer parameters to produce a consumer-centric multi-cloud built on multiple CSPs. To do so, we presented the step-by-step workflow of MANTUS, presenting how each group of consumer-parameters is considered and leveraged to build an adapted multi-cloud. We highlighted how best set of DC can be selected w.r.t. consumer preferences, data-centers features and broker context. We introduced an ILP-based programming to model this selection, introducing the possibility to add further constraints specified by the consumer (e.g. Degree of CSP diversity). Moreover, we showed how AOP-Weaving and Context-based matching
in MANTUS to make the multi-cloud flexible and optimized, combining contributions of this manuscript.

At the top of Figure 4.5, we resume the three main steps of a multi-cloud deployment performed through MANTUS. The first step is the context-based provider selection and the second step is the TOSCA matching. These two steps output, respectively, the list of the selected providers and the corresponding concrete TOSCA code ready to be deployed. The third step is the service implementation to be translated and deployed on different CSPs. MANTUS calls the API of each involved CSPs implementing different specifications and deployment behavior. The result of the MANTUS pipeline is represented by an overlay multi-cloud infrastructure deployed by the consumer through the MANTUS pipeline.

Together with what introduced in this chapter, the architecture of such infrastructure should also satisfy several properties to guarantee interoperability and usability (i.e.;
single point of control). This will be addressed in the next chapter, where we introduce ORBITS. ORBITS is a multi-cloud architecture responsible for orchestrating and executing consumers tasks on the multi-cloud. MANTUS takes in input a generic IaC TOSCA description of ORBITS to generate and deploys a concrete deployment of the multi-cloud.
Building an Overlay Multi-Cloud

In previous Chapter 3 and Chapter 4, we detailed how the IaC paradigm can be enhanced to obtain a multi-cloud capable of an optimized usage of CSP resources and that is easier to maintain. In a nutshell, we focus on how a consumer-centric multi-cloud infrastructure mitigating the issues of lack of specialization and extensibility can be built to optimize code and resources.

Regardless the precise context that should guide the way the multi-cloud building, the proposed multi-cloud infrastructure should at least address the problem of interoperability among different CSP-ecosystems. This feature will introduce an advantage for the cloud consumer who will be able to manage the workload w.r.t. different CSP in a unique and standardized way, decoupled from the specific CSP he is interacting with. In this chapter, we discuss an end-to-end deployment of a multi-cloud infrastructure and focus on which service has to be part of it. We recall that the result of MANTUS pipeline is represented by an overlay multi-cloud infrastructure deployed by the consumer through the MANTUS pipeline. The architecture of such infrastructure, ORBITS, satisfies several properties to guarantee interoperability and manageability (i.e.; single point of orchestration). Receiving as input a precise context, MANTUS is capable to derive a compelling implementation of ORBITS to realize the multi-cloud.

More precisely, as we will discuss in this chapter, the ORBITS infrastructure template is taken as input by the MANTUS pipeline, as shown in Figure 5.1. MANTUS uses IaC extensibility and specialization presented respectively in Chapter 3 and 4 to manipulate the ORBITS code and providing as output a context-adapted ORBITS instance. Such output, as we detailed in Section 4.3.3, corresponds to IaC code ready to be deployed on selected CSPs. In particular, the output of MANTUS is represented by code translated from TOSCA to a format which can be processed by CSPs and used to initiate the multi-cloud deployment.

So far, we focused the MANTUS pipeline and how IaC services can be manipulated to be extended or adapted. In this chapter, we focus more on the how the multi-cloud overlay services have to designed and integrated. We firstly discuss how the ORBITS template architecture should be designed and show how the resulting IaC-based overlay multi-cloud built through ORBITS and MANTUS can be deployed by a customer and how she may handle unexpected events (e.g. automatically rebuild a new version of the multi-cloud optimized to the mutated conditions). We present flexibility points (e.g. specialization, extensibility) can be declined to simplify the building process. The objective is to introduce the possibility to automatically trigger multi-cloud infrastructure reconfigurations with low human intervention. We present this by presenting a possible integration of MANTUS to extend a policy-based Anti-DDoS system leveraging context-
based remediation. Considering the positioning of this chapter with the proposed contributions, the elements here presented are (1) the ORBITS template architecture which will manipulated by MANTUS to obtain the specialized multi-cloud deployment, (2) the end-to-end multi-cloud construction process spent to set up the infrastructure layer to set up the multi-cloud (ORBITS) and (3) the MANTUS multi-cloud life-cycle management performed on ORBITS instances.

The chapter is structured as follows. In Section 5.1, we introduces the ORBITS architecture, detailing its functional components. In Section 5.2, we present a complete multi-cloud dynamic deployment and reconfiguration. We analyze the role of ORBITS and MANTUS and how to combine them to adapt the infrastructure to a mutated context. We detail their behavior in a concrete scenario by tackling an anti-DDoS attack.

5.1 OVERLAY INFRASTRUCTURES FOR MULTI-CLOUD

In this section, we pinpoint the main functional components of an overlay infrastructure, by detailing features of each group of components. Traditional overlay-based approaches [48, 96] provide the user with an important degree of control (e.g. virtualization layer, security appliances), but they are neither designed for a multi-provider purpose nor to tackle with infrastructure homogeneity. The idea of ORBITS is to completely abstract the CSP APIs, from an orchestration perspective to a virtualization perspective (e.g. not have to deal with different hypervisor technologies).

Such overlay infrastructure represents the base "infrastructure" template which is taken as output by MANTUS in order to build the customer-tailored multi-cloud.
5.1 Overlay Infrastructures for Multi-Cloud

The ORBITS architecture satisfies the requirement of interoperability proper to the multi-cloud infrastructure because it leverages an overlay-based approach. The key ORBITS idea is to provide simultaneously flexible applications provisioning across multiple providers, and a homogeneous service abstraction across multiple clouds. In other words, ORBITS is designed to integrate a set of infrastructure services capable to implement IaaS cloud. The consumers can use ORBITS instantiations remaining completely decoupled from underlying deployment and, by using the ORBITS API, they have not to deal with different CSP APIs. The ORBITS architecture definition, manipulated by MANTUS (Section 4.3), is then translated into the most compelling implementation in a given context.

In this section, we first present ORBITS abstract architecture, introducing its different layers and their functions. In particular, the contribution we propose is to identify the functional components necessary to build an infrastructure that also fulfills the multi-cloud constraints in terms of interoperability. ORBITS abstract architecture is then supposed to feed the MANTUS workflow, where each component will be replaced by a concrete implementation according to the context.

As shown in Figure 5.2, ORBITS adopts a three layered-design. Each layer implements a different part of a multi-cloud architecture and is described in the following.

- **The orchestration layer** ensures the flexible provisioning across multiple providers required by the use-cases. It has an overall view of available providers and coordinates applications orchestration among provider instances. This layer also implements application orchestration, by leveraging different cloud contexts it is aware of. For example, it may be deployed on multiple providers, driving the selected management layer on a specific CSP, the needed number of replicas to guarantee high availability of ser-
services (for EHR), or to achieve effective provider isolation (e.g. for privacy-preserving multi-party computations on healthcare data for example).

- The **management layer** is in charge of the resource provisioning on each overlay provider, managing the virtualization layer and the creation of new execution environments (EEs). This layer also meets the Quality of Protection (QoP) requirement, focusing not only on the execution of the applications, but also on the access to the resources.

- The **virtualization layer** executes scheduled jobs, with trade-offs between performance and isolation among different workloads. It provides a homogeneous view of security services to consumers to meet the QoP requirement (e.g., for EHR systems).

Management and virtualization layer services are deployed on each provider selected to be inside the multi-cloud. We refer to those instances as **overclouds**, as they are overlay instances that provide an homogeneous view of resources to the orchestration layer.

In the following section, we will extensively describe the three layers listed above, detailing features they have to implement and technologies they rely on. More precisely, we describe in details tasks and components that have to be included and realized in a concrete implementation of ORBITS.

### 5.1.2 The Orchestration Layer

The orchestration layer is in charge of the service provisioning and the coordination of different Overcloud instances, concerning infrastructure components (e.g. network connectivity) and applications. Both types of services have to be deployed over multiple CSPs, implementing de facto the multi-provider coordination among the different overclouds.

**Infrastructure Orchestration** Following the “Infrastructure as Code” paradigm, a cloud template text-description for the overlay infrastructure defines which services are deployed and where. Orchestration covers:

1. Providing on-demand interconnection between providers;
2. Managing identity and access across overlay instances.

Therefore, virtual networks are created inside each overlay cloud by hosting cloud providers. To create multi-provider connections, a **Network Fabric Builder** component extends local virtual networks across provider barriers.

Finally, an overall **Authentication & Authorization Service** transparently manages identity and access across deployed overclouds, e.g., coordinating different authentication services.

Finally, we can consider as part of the orchestration layer, the MANTUS builder since it communicates with orchestration providers API (e.g. OpenStack Heat, Amazon CloudFormation) deploying on selected providers the overclouds template.
5.1 Overlay Infrastructures for Multi-Cloud

While the role of infrastructure services is building and maintaining the ORBITS multi-cloud, flexible provisioning across clouds is the role of the application orchestration logic – typically for placing application microservices across providers.

Orchestration frameworks are usually composed of application frameworks and of a resource multiplexer (e.g., Apache Mesos). Indeed, Application frameworks are responsible for application deployment on available resources, following developer/operator specifications. The resource multiplexer guarantees fair sharing between frameworks on a pool of resources.

In ORBITS, we enhance the placement logic of application frameworks, introducing multi-provider awareness of overclouds deployed by MANTUS. The overcloud-aware placement leverages a component deployed in the management layer, STRATOPAUSE. STRATOPAUSE instances: (1) receive updates about overcloud instance availability; and (2) dispatch on a given provider selected jobs.

In a nutshell, the application-level orchestration designs the entry-point for application deployment on the multi-cloud. In particular, considering the reference use-case, essential requirements of healthcare applications [1] as confidentiality, data integrity, anonymity may leverage the single-point of orchestration to effectively decide where to deploy different instances of services, relying on the homogeneity of the infrastructure. This run-time control may allow also operators of the service to easily respect legislation in terms of data protection and geo-localization.

5.1.3 The Management Layer

For infrastructure homogeneity, ORBITS aims not only at virtualization interoperability but at homogeneous resource management across multiple clouds. This implies uniform APIs across providers. Indeed, complete interoperability issues coming from the multi-provider nature of the infrastructure may be prevented by security services provided “as-a-Service” by specific cloud providers (e.g., anti-DDoS, firewall). Different APIs may require per-provider adaptation work and, thus, homogeneous resource management is the key to guarantee QoP in our use-case.

Two classes of management services are distinguished for ORBITS overclouds:

- **Local resource provisioning** The Local Cloud OS and SDN controller components are normally in charge of compute, storage, and networking management.

- **Relation with orchestration logic** The Local Orchestrator, named from now on STRATOPAUSE component, is the link between local resource provisioning and application dispatching. It informs regularly the application orchestration framework about available overclouds, e.g., resources, cloud attributes (provider, region, virtualization technologies). When the application orchestration logic schedules a job on a certain STRATOPAUSE instance, STRATOPAUSE communicates with the Cloud OS service to trigger resource allocation to satisfy the allocation requirements demanded by the orchestration layer. The global orchestration logic collects updates from STRATOPAUSE instances to reach placement decisions. This instance also collects microservices dispatching commands to local overlays, which are transmitted to the local Cloud OS to provision resources according to expressed requirements.
The management layer enables using equivalent security services on different providers, e.g., to fulfill EHR systems security requirements. However, this layer does not have the overall vision of all deployed overclouds.

5.1.4 The Virtualization Layer

The virtualization layer runs microservices with a provider-agnostic approach. Virtualization is a widely-adopted approach to obtain isolated and transparent hardware resource sharing between competing software or systems. Several technologies may be adopted to deploy and run execution environments (EEs) that are generally not interoperable [5].

In ORBITS, the virtualization layer should realize interoperability among isolated EEs across different providers, hiding provider heterogeneity. This is not possible at underlay level because of technological heterogeneity. The virtualization layer should also: (1) be customizable, allowing each operator to deploy its chosen security services; and (2) impose minimal performance overheads.

Thus, developers and/or operators may adapt the virtualization technique to workloads, isolating homogeneously across providers components of an application selectively. This may be achieved through the management layer API.

Therefore, ORBITS provides a secure and description-based overlay infrastructure, with an abstract view of an homogeneous multi-cloud, tightly coupled with application-driven orchestration logic. So far, we presented the ORBITS architecture and components, highlighting the major design challenges and how they are addressed by the architecture. We present in the following section a first architecture prototype, describing implementation choices and flexibility points.

5.2 END-TO-END MULTI-CLOUD CONSTRUCTION

In the previous section, we detailed the ORBITS architecture showing how they can build a multi-cloud infrastructure. This allows to show how the contributions of Chapter 3 and Chapter 4 are integrated in the overall proposed architecture. In particular, we highlight how MANTUS can manipulate an abstract service definition to obtain a compelling concrete implementation, inserting on-the-fly non-functional features and optimizing the usage of CSPs. However, so far, it remains unclear how MANTUS and ORBITS can handle sudden context mutations. More precisely, we focused on the deployment of the infrastructure and on the flexible provisioning of the multi-cloud. In this section, after having detailed the key steps of MANTUS workflow, we analyze how MANTUS and ORBITS may handle at the same time unexpected mutations in the CSP/Consumer context and reflect them on the deployed multi-cloud.

5.2.1 Multi-cloud provisioning workflow

In this section, we recall fundamental steps of the end-to-end deployment of a multi-cloud, by leveraging the MANTUS multi-cloud builder and the ORBITS infrastructure. We describe a practical deployment of a multi-cloud through the MANTUS multi-cloud compiler. First, we assume that MANTUS software is packaged and available as a traditional web-application which is downloaded and deployed by the customer on its infras-
structure. Each consumer can have access to MANTUS dashboard to modify all aspects that the "compiler" logic would leverage. In particular, consumer can modify the TOSCA infrastructure code, configure extensions to be weaved to this and trigger a deployment on her development machine.

We assume that the customer has already valid accounts on different cloud providers, and fills to MANTUS access keys to deploy and manage resources. MANTUS has to be completed with the list of available CSPs and their context to obtain a refined definition of the service the consumer wants to deploy. Such information can be provided statically (e.g. embedded in the MANTUS download package) or downloaded at compile time leveraging 3rd-parties APIs to benefit from variations in time of CSP contexts.

Moreover, MANTUS is provided with a base set of TOSCA NodeTypes and ServicesTemplates which represent the basic building blocks of the multi-cloud infrastructure. For example, those building blocks, which models the VMs, storage, network and managed services (e.g. PaaS-like) will be used to instantiate the abstract ORBITS architecture definition. The customer can modify and change parameters of ORBITS abstract to create its own infrastructure template. For instance, it has to characterize its overcloud, the set of services she wants to deploy on every CSP being part of the selection.

Due to the ORBITS and the recursive TOSCA modeling presented in Section 4.2.3.2, the consumer deployment remains at an abstract layer and provision resources through MANTUS on the underlying CSPs. The ORBITS infrastructure layer guarantees portability of Execution Environments (EEs) where service components are executed on any CSP of the multi-cloud. Deployed services work similarly and expose the same APIs on all different CSPs, and will require small adaptations from existing applications.

In addition, as presented in Section 4.2.3.2, MANTUS lets the consumer be able to describe the further multi-cloud parameters (non-functional, SLA) without knowing in advance on which CSP the multi-cloud will be deployed, and how it will be actually implemented. In case the consumer desires to be also capable to have improved control in the deployed multi-cloud infrastructure, she can specify this in the customer-context provided.

In the next section, we present how MANTUS is capable to deal with the whole lifecycle of a ORBITS multi-cloud.

5.2.2 Multi-cloud Dynamic Reconfiguration

So far, we have considered the deployment of the ORBITS infrastructure layer for multi-cloud and we manipulated it through MANTUS for flexibility and optimization without considering the whole life-cycle of the multi-cloud. In fact, after achieving the building process, unexpected events (e.g.; intrusion, provider malfunctioning) can occur through the whole ORBITS life-cycle. In particular, these events can increase the complexity of managing a multi-cloud because human intervention is required in order to handle them. Simplifying multi-cloud management, by limiting the necessity of human intervention as depicted in [42], would imply lower management costs.

In this section, we make a step further w.r.t. the initial construction of the multi-cloud. In particular, we shows how the MANTUS workflow can be used to update a deployed multi-cloud infrastructure in case of unexpected events. More precisely, we investigate the possibility to create an autonomous detection-reaction loop [65] to perform dynamic reconfiguration of the multi-cloud without human intervention. This can be done by fore-
seeing automatic context-adapted reaction in the case of certain triggering events. This section is structured as follows. First, we define an autonomic MANTUS-based architecture. We also analyze which sources of events may affect the life-cycle of a multi-cloud and what it takes to properly handle them. We then present the self-protection example, where we configure MANTUS to detect and react against a DDoS attack.

5.2.3 Mantus-based autonomic architecture

Following the ORBITS architecture model presented above, we can apply the autonomic detection-reaction model [65] by firstly defining resources to monitor in an overlay multi-cloud environment and dividing them into three categories:

- **CSP software stack** represents all components which are not under the responsibility of CSP. In case on unavailability or security accidents, the consumer may be notified of those events without having any control on them.

- **ORBITS software components** are deployed and described inside the ORBITS template and resources on which MANTUS is aware of and can manage them. It is executed over the CSP software stack and leverages its services.

- **End-consumer services** are final services deployed by end-consumer and not described inside the ORBITS template.

Such resources represent the sources of events which can trigger a change in the context and therefore trigger a reaction from our environment. To make this mapping more explicit we present in Figure 5.3 previously mentioned resources in a scheme inspired by autonomic architecture FORMS [117]. In such scenario, MANTUS represents a single multi-cloud point of enforcement, which can control any aspect of the of the multi-cloud change.

MANTUS has complete control over ORBITS software since, it is charged to generate and deploy the effective ORBITS instance of a multi-cloud. Moreover, it can configure the CSP software stack to notify the MANTUS builder about the status of the CSP infrastructure (e.g.; hooks on verified condition to MANTUS API), extending the visibility on events outside the consumer resources. However, MANTUS is unaware of consumer-deployed applications since applications are deployed after MANTUS completes its infrastructure deployment.

Leveraging context-based matching and AOP-based weaving, MANTUS may perform a flexible and incremental service reconfiguration to cope with different events. For instance, in case of self-protection, a detected threat can be treated through TML aspects to be matched & weaved without modifying already deployed services. To configure the self-protection loop, the consumer would program the Autonomic Manager through policies, which will be used, proactively or reactively, to generate adapted TML scripts. Those scripts would be agnostic to the underlying CSP but will implement a certain self-protection pattern [124]. Consequently, MANTUS has to first “contextually match” those TML scripts, adapting them to a certain CSP context and then weave them to the template, reusing the traditional workflow triggering an update of current deployment.
Figure 5.3: MANTUS autonomic behavior model based on FORMS [117].

Table 5.1: Mapping context mutations to MANTUS reactions.

<table>
<thead>
<tr>
<th>Mutation</th>
<th>MANTUS Feature</th>
<th>Action Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC Unavailable</td>
<td>ILP-based Selection</td>
<td>Disable corresponding variable</td>
</tr>
<tr>
<td>DC Service Not-available or Degraded</td>
<td>Context-Based Matching</td>
<td>Re-perform Context-matching with the new context</td>
</tr>
<tr>
<td>ORBITS malfunctioning - intrusion detected</td>
<td>AOP-based Weaving</td>
<td>Weave new aspects and redeploy differentially the template</td>
</tr>
</tbody>
</table>
In Table 5.1, we provide a short list of possible events treated by MANTUS, presenting how those can be tackled and how they are affected by the change.

In next section, we present a self-protection example. We use the architecture introduced in Figure 5.3, where an autonomic manager (AM) receives information of mutated context and instructs MANTUS about how to handle context changes.

### 5.2.4 Self-Protection Example

In this section, we provide a concrete example of MANTUS integration in a self-protection scenario. We propose in Figure 5.4 a concrete instance of abstract architecture presented in Figure 5.3.

Following the presented scenario, MANTUS is integrated with a policy-based SDN-based Anti-DDoS architecture, extending what proposed in [99]. In this work, the authors proposed a policy-based management framework to allow network administrators to define global network and security policies. Policies are enforced through SDN-based technologies (e.g. OpenFlow) to network devices. In addition to this former proposition, we propose to extend such approach to the multi-cloud and use MANTUS to complete the anti-DDoS policy-based mechanism.
Aside redirecting traffic flows through traffic steering primitives [76], referred as "in-band" Point of Enforcement (PEP), MANTUS introduces the possibility of spawning ad-hoc middleboxes or other infrastructure elements to filter out malicious traffic patterns, acting as a "Out-of-band" PEP. In fact, several CSP and Cloud Management systems enables the possibility for the consumer to not only define traditional virtual network primitives (e.g. Network Interface, Subnet, Network, Router) but also traffic steering ones [31, 105] providing a direct control to consumer on the path of incoming/outgoing traffic. Traffic steering primitives may allow the consumer to redirect network flows. In such approach, the idea is to steer traffic to appliance dedicated to analyze malicious traffic before reaching the ORBITS VM. On the one hand, the MANTUS AOP-based weaving, discussed in Section 4.3.2, enables the possibility to dynamically add or remove new services to the ORBITS infrastructure, changing the service topology and leveraging the incremental approach supported by most providers. On the other hand, the MANTUS context-based matching, presented in Section 4.2, allows to implement optimized services (e.g. middleboxes) adapted to provider features and consumer-requirements.

In the following sections, we analyze the different possible configurations of ORBITS in detection and reaction phase of the self-protecting loop.

5.2.4.1 Detection

We consider the ORBITS infrastructure based on Open-Stack, previously presented in this chapter, as shown in Figure 5.5. We observe in the bottom part of the this figure the introduction of a Log Aggregator and agents on each VM. This can be obtained through MANTUS by weaving "Monitoring/Logging services" to ORBITS base infrastructure. This
can be triggered for example by a consumer selection at instantiation time. A simple monitoring architecture may be designed as proposed in Figure 5.5. The base infrastructure is enriched with on-board agents which collect metrics and logs inside each VM and then forward them an aggregator. Obviously, the results of this weaving may depend on features offered by selected data-center candidates to be part of the multi-cloud.

A key design point is represented by the mean of communication between agents, local aggregators and autonomic managers. In Figure 5.5, MANTUS uses the network to do this by introducing a new virtual Network Interface Controller (NIC) for each VM and new virtual networks to isolate autonomic communication.

It is worth noting that the flexibility of AOP-based weaving and context-based matching allows to obtain different implementations on different CSP. For instance, the matching mechanism is flexible enough to leverage transparently different implementations of the agents. For example, a CSP may propose, in order to achieve a better isolation, that the agent deployed inside the VMs (e.g. as a system service) may be capable to communicate through specific devices inside the VM for recording logs [116] and not through the network. Another possibility, given by the adoption of advanced security technologies (e.g Intel SGX [37]), executes those agents inside a CPU-provided execution domain, protected from corrupted VM operating systems.

To sum up, according to consumer-defined optimization criteria, the most suitable implementation of each service is selected on each different CSP to enrich the base ORBITS infrastructure.

5.2.4.2 Reaction

When it is triggered, the reaction step modifies the state of the system. The AM is contacted upon an event occurred, based on its policies it derives a set of MANTUS TML scripts to be weaved dynamically on ORBITS services. Those scripts are designed to match target services and performing actions on them (e.g., VM Configuration, Orchestration policies) [116] leveraging the approach discussed in Section 3.3. Therefore, recalling the scope of TML scripts, the AM can not only change the configuration of the services but also modify their topology, by introducing new virtual devices (e.g. virtual NIC hotplug), middleboxes and other required infrastructure elements.

Obviously, features proposed by the target CSP data-center has to impact which implementation of countermeasure has to be adopted. MANTUS collects them in the so-called "Provider context". In figure 5.6, we show the result of two equivalent deployments of the same "aspect" weaved on a OpenStack-based ORBITS overcloud, showed respectively in the top and the bottom of the figure. The protection aspect weaves a IDS to analyse traffic of management layer services (OpenStack controller and SDN Controller) looking for traffic anomalies and anti-DDoS middle-boxes to prevent attacks on deployed servers. In this example, when a CSP supports traffic steering as in the upper part of Figure 5.6, the traffic is steered to a "middlebox" before reaching the VM. On the contrary, as presented in the lower part of the figure, when a CSP does not support "Traffic Steering" it deploy an agent inside each VM, In addition, traditional providers normally offer managed protection mechanisms against such classes of attacks which can be introduced dynamically if it is possible to subscribe them through IaC.
Figure 5.6: MANTUS leverages differently distinct CSP context and adapts the matching to find a compatible solution: (1) middleboxes and traffic steering in the top part and (2) in-VM application otherwise.

5.3 Conclusion

In this chapter we completed the multi-cloud framework presented in Chapter 3 and Chapter 4.

First, we presented the ORBITS architecture, which provides multi-provider IaaS interoperability through a user-controlled virtualization layer and flexible application provisioning over a set of clouds. Therefore, MANTUS is capable of manipulating an abstract ORBITS TOSCA definition to deploy a multi-cloud spanning over multiple providers, in a flexible way.

Second, we show how MANTUS and ORBITS can be used to tackle the flexible orchestration requirement in terms of infrastructure life-cycle. Leveraging MANTUS AOP-based weaving and context-based matching, introduced respectively in Chapter 3 and 4, over an ORBITS template it is possible to deploy a multicloud and make it evolve according to a mutated context (e.g. unexpected event occurred). Since CSP and consumer context embeds completely the occurred mutation, MANTUS pipeline can be triggered with low or any manual intervention in order to react fast.

In next chapter, we present how we can tackle the lack of control in cloud and multi-architectures by introducing a re-designed virtualization architecture, the U-cloud. In addition, we present how this architecture may be integrated in the MANTUS-ORBITS workflow.
5.3.1 Conclusion

Presenting ORBITS, we satisfied the design requirement of infrastructure homogeneity building an overlay multi-cloud architecture.

However, if we observe the ORBITS model, the "infrastructure control" requirement is still not completely achieved. Underlying CSP-controlled layers, (e.g. hypervisor-layer) provided by the CSP cannot be customized by a customer as in a private cloud. In next section, we will focus on the infrastructure control requirement and how a customer can obtain this on a remote infrastructure. In particular, we propose an extension of ORBITS to enable the possibility for the user to introduce its custom modules into the virtualization layer. More precisely, we would like to introduce the possibility for a consumer to customize the provider infrastructure while deploying its desired services (e.g. security features). This requires the faculty to configure security services at hypervisor level, as shown in [122]. The key principle is to the introduction of this enhanced control over remote hypervisor without breaking compatibility with existing applications or changing the way the customer consumes CSP resources.
In the previous chapter, we completed the presentation of ORBITS and MANTUS multi-cloud framework and showed how the entire life-cycle of a multi-cloud can be handled from the early deployment phase to the reconfiguration phase. In this chapter, as presented in Figure 6.1, we tackle the problem of the lack of control in virtualization architecture of modern cloud providers. In a nutshell, when cloud consumers use public clouds, they have to accept to eventually reduce their standards in terms of (1) security (regarded here as isolation of the virtual machine) and (2) customization of the whole infrastructure. In particular, there are two reasons why the security standards might not be met. First, the same hardware is shared among several untrusted consumers and this by design decreases the isolation among users. Second, the software responsible for isolation within the same hardware, i.e. the hypervisor, relies on a huge Trusted Computing Base (TCB), which does not prevent an administrator from snooping the consumers’ content. The reason why the infrastructure cannot be fully customized is due to the impossibility to customize the shared layer of the hypervisor. This prevents cloud consumers from installing in this layer their add-ons (e.g.; introspection, intrusion detection). The loss of control limitation might even prevent developers from adopting the cloud to design their applications. This holds not only for single CSP applications, but also for multi-clouds where the control of hypervisor may introduce the same customizations on each selected provider.

The solution that we propose to mitigate the loss of control of cloud-base applications is a consumer-defined architecture named U-cloud. The U-cloud architecture enables consumers to define the services part of the virtualization stack (where their applications are executed) without loosing the compatibility with existing applications. In particular, the U-cloud architecture is based on a combination between nested virtualization and the micro-kernels, which were recently reproposed as an interesting technology for cloud commodity hardware. Micro-kernels allow to minimize the TCB through operating system modularization, opening the door to consumer customization of the hypervisor. Nested virtualization enables a fully de-privileged hypervisor to be executed and controlled by the consumer on a underlying hypervisor.

The combination of this two entails a reduction of the TCB and a consequent control by the user on which services and hypervisor to adopt to, respectively, build their virtualization layer and execute their applications. Furthermore, this leads to the possibility to customize the cloud infrastructure while at the same time enhancing isolation among users and mitigates, in this sense, the problem of loss of control.

The chapter is structured as follows. In Section 6.1, we discuss in detail the problem of loss of control w.r.t. the virtualization architecture (mentioned briefly in Section 1.1). Section 6.2 presents a solution architecture to this problem, i.e. the U-cloud architecture, given by the combination of micro-kernel and nested virtualization. In Section 6.3 we show a U-cloud architecture prototype and in Section 6.4 we present the U-cloud provisioning and reconfiguration workflows. Finally, we discuss the integration with ORBITS and MANTUS.
6.1 BACKGROUND

State-of-art virtualization architectures do not allow the user to customize the virtualization layer, despite multitude of different works that targets hypervisor-level tools [6, 7, 48]. This is mainly due to isolation and security reasons. In fact, traditional hypervisors inherit from monolithic kernels a wide Trusted Compute Base (TCB) which may be prone to failures and vulnerabilities [88]. Moreover, the possibility to introduce consumer-defined modules in the hypervisor may be used as presented in recent works [121]. In fact, the hypervisor represents a privileged point of observation to monitor applications and less privileged operating systems components and can be extended through add-ons enriching its basic "virtualization tasks" (e.g., Intrusion and malware detection).

Those add-ons, introduced in the hypervisor, where they are usable are impossible to be deployed in a shared hardware as in cloud environment. In fact, multi-tenancy at hypervisor level is not possible with a traditional monolithic architecture in a multi-consumer node as for a public cloud provider. This implies that consumers cannot configure their virtualization layers as they wish without having enough privilege to break the system or isolation (e.g., inspect other tenants executions).

6.1.1 Design Requirements

The U-Cloud architecture aims to introduce the capacity for the consumer to control the entire software layer responsible for executing cloud applications to achieve full control over allocated resources. To this end, we identified four main design requirements:
1. **Minimal Trusted Computing Base (TCB)** Reducing TCB is key for system reliability and availability. The TCB of an hypervisor is the set of all hardware, network and software components that are critical, or in other words, where occurring faults or vulnerabilities can disrupt the security properties of the entire system. For example, malicious consumer sharing the same hypervisor may try to exploit flaws in the huge TCB of modern virtualization stacks to escape execution environment isolation. In this analysis, since hardware represents a common basis for every architecture, we limit to consider part of the TCB of an hypervisor all the software components which are executed within VMs on behalf of the consumer.

2. **User Black-box Resource Administration** Consumer services should be allocated by administrators in a black-box fashion, similarly as proposed by [128]. Malicious administrators may leverage their extended privileges to snoop the private execution state of applications, without any possible user control. Instead, system component instances should be controlled directly by consumers in order to configure their own resources without the possibility for the administrator to observe inside.

3. **Consumer Customization** Virtualization stack should support consumers customization allowing them to deploy security appliances at hypervisor level [6, 7, 48].

4. **Legacy Support** To encourage its adoption, the virtualization layer should require the least porting effort in application and control logic adaptation.

6.1.2 **Virtualization Background**

6.1.2.1 **General-Purpose Hypervisors (GPH)**

The GPH is the key cloud-enabling technology and represents the traditional hypervisor used in cloud computing. A GPH exports a hardware abstraction for concurrent execution of different OSes in isolated VM instances. GPHs normally leverage full virtualization with hardware assistance that provides hardened resource isolation [4].

A GPH is considered as monolithic and poses security concerns: GPHs have a relatively small attack surface, but non-negligible TCBs, as traditional OSes. This architecture does not enable a customer to partially personalize services or control the hypervisor. Finally, due to different lock-ins or implementation designs, cross-provider interoperability is often limited [6].

6.1.2.2 **Minimal and Modular Hypervisors**

To solve the TCB size issue of GPH solutions, the idea of Micro-Hypervisors (MH) architecture was introduced, drawing inspiration from evolution in OS architecture. Such designs were widely explored in academia [64, 108, 114], but adopted only by the mobile device industry [83]. The aim is to expel as much code as possible from the TCB, contributes to make the hypervisor ultra-thin. Typically, around 10KLoC for the TCB, an order of magnitude smaller than a GPH [108, 114]. The whole MH architecture exposes a minor attack surface, since a significant part of services provided by the hypervisor in kernel space are now provided in user space, leaving to the MH core only the burden to correctly implement Inter Process Communications (IPCs).
Some hypervisor components (e.g., device drivers, VM address space management) may be executed outside the MH, isolated, and restarted in case of compromise, improving resilience and isolation. In such increasingly modular designs, each component is provided with the privilege level required to perform its specific task, enforcing the principle of least privilege. The utmost MH removes the virtualization layer altogether [64].

However, the MH quest towards ever increasing minimalism could represent a serious limitation to preserving functions of existing platforms: the constraint of privileged code size may force MH developers to drop some basic features considered as non-essential, like multi-guest support [114].

Other works [7] proposed to divide the monolithic management plane of GPHs to a multitude of components which may be controlled by different consumers, e.g., the Self-Service Cloud (SSC) architecture modularizes the Xen Dom0 VM, introducing components directly controlled by the user (*user domains*) to manage resources, adopting a black-box system administration.

To sum up, such family of hypervisors offer a high level of user control, also enabling trade-offs with provider control: to let the provider monitor user resources, inspection of user domains may be performed through mutually-trusted service domains, with known a priori and fine-grained permissions.

However, all those approaches introduces strong changes in control logic and APIs, impacting support of legacy cloud management platforms. In next section, we present nested virtualization which represent an interesting approach to maintain legacy compatibility while allowing the consumer to control a customizable depriviledged hypervisor.

#### 6.1.2.3 Nested Virtualization (NV)

As we briefly introduced when presenting ORBITS, NV is a system architecture with two layers of virtualization: the guest OS virtualizes a nested guest [19]. The concept may be generalized to an arbitrary number of nested guest layers, leading to recursive virtualization [3, 47]. An NV architecture is composed of: (1) the hypervisor running above the hardware (*L0 hypervisor*); and (2) the nested hypervisor (*L1 hypervisor*) which executes applicative domains. Nested VMs are generally called *L2 guests*.

At the beginning of the 10’s, the NV growing maturity opened plenty of new possibilities and avenues for research. The trend is to diversify hypervisor functionalities, with new features but also keeping existing ones. Each layer clearly addresses different sets of issues:

- **L0** is the last line of defense of the platform and a privileged point for monitoring its status.
- **L1** provides the widest possible support for different customer requirements and virtualization techniques. This layer could be steered by the user for complete control over the infrastructure. It could also be a layer of interoperability across different providers, extending towards an integrated control plane as presented in ORBITS.

Today, NV presents still limitations related to performance, which can be reduced by recent advanced hardware support [80], and TCB size inherited from the use of a GPH at the L0 layer (DR2). Full virtualization with hardware assistance is the most popular technique, but requires specific code in the hypervisor.
TCB size issues are the following: each feature directly introduced within the hypervisor such as security enhancements enlarges the amount of code running in highly privileged mode and required to be trusted.

6.1.3 Conclusion

To sum up, despite promising in terms of the possibility to consumers to control their own virtualization layers, NV does not address the TCB requirements. In addition, NV does not meet by design a "black-box" resource administration property. Due to the monolithic architecture of GPH, which represent the building blocks of existing NV architectures, a curious administrator which controls the "lower" hypervisor would be capable to inspect the L1 VM execution.

Related works [126], leveraged NV is using it as simple technique to make traditional hypervisor less privileged. However, this prevents the possibility to leave the control of such layer to the consumer, as discussed in [7].

However, we consider that Nested Virtualization (NV) is the best solution to enable legacy support, which represents the least but not less important requirement. We consider that it is up to consumers to handle the life-cycle not only of their VMs but also of their hypervisor/virtualization layer.

Consequently, the approach which "flattens" the nested virtualization layers proposed by [7] would require a change in how consumers and their applications interacts with the hypervisor.

As we will detail in the next section, we propose U-cloud, a new hypervisor-based architecture using NV, where, on the one hand, the L1 hypervisor is "lifted and shifted" without modification as L1 to respect the control and the legacy support requirement. On the other hand, the underlying hypervisor, the L0, satisfies the black-box resource administration model through a micro-hypervisors architecture. This can satisfy all three requirements and the resulting complexity may be overwhelmed by the consumer through the usage of the MANTUS-ORBITS pipeline to compose their own virtualization stack.

6.2 U-CLOUD ARCHITECTURE

In this section, we provide a high-level representation of the U-Cloud architecture we propose in this manuscript. In Figure 6.2, we may observe the concept of U-Cloud, where the consumer has control to lower layers of architecture. In fact, using Nested Virtualization we can rely on two system layers, one, the "Upper Layer" which is completely controlled by the user (e.g. through the ORBITS architecture and MANTUS orchestration) and one, the "Lower Layer" securely shared by user and provider leveraging micro-kernel modularity, as we analyse in next sections.

In this section we reuse some well-known and adopted components designing an inclusive architecture. The novelty of this architecture is represented by the possibility to satisfy all three design requirements, remaining also compatible with an adoption in a multi-cloud context.

Figure 6.3 illustrates in an example the concrete design of the U-Cloud node. We divide the architecture in two layers: the Low layer, which embraces the non-virtualized layer or L0, and the Upper Layer (UL) which embraces virtualized L1 and, where are present, L2 layers.
6.2.1 Lower Layer

The lower layer represents the lowest software layers in a node virtualization stack, which is responsible to multiplex hardware resources to multiple users. We consider the virtualization layer of a cloud provider based on a MH-based. This design choice may provide a solid foundation to meet all isolation features, notably smaller TCB size and VM security. The LL is composed of the MH under provider control and of a set of user-land infrastructure services for increased user control over U-Clouds. Leveraging the strict modular architecture of MH, the TCB is now reduced and some of most failure-prone components (e.g. Device Drivers) are now outside the architecture core.

The Micro-Hypervisor core layer provides a tiny set of kernel features, e.g., scheduling, Memory Management Unit (MMU) management, IPCs, handling Virtual Machine eXtensions (VMX)-related events and interrupts. Following different "personalities" principles proper to micro-kernel/hypervisor [50], each user controls a Virtual Machine Monitor and a set of user-space services. The cloud system administrator directly controls resource management policies, system-wide resource multiplexing, and device drivers, but without control over the MH core. Similar to Zhang [128], we consider the "MH core", the micro-kernel, under control of CSP organization. As discussed by [126], we consider that administrators would not require to have the highest amount of privilege to do their routine tasks. Despite in control of hardware resources, the CSP organization does not want to trust every single administrator. The MH core represents the only container which runs with the highest level of privilege (i.e. ring 0 in root mode in x86) and cannot be accessed by the system administrator. Other tools which are responsible for other tasks (e.g. resource management, device drivers) is expelled in user-space.
From the user standpoint, the LL architecture increases control over the infrastructure. In fact, the LL introduces the possibility to add new services "under the hood" of the virtualization layer not only inside the UL hypervisor but even in LL user-space without disrupting legacy applications. From the provider standpoint, security is strongly enhanced due to two-level isolation by the MH core and L0 hypervisor virtualization. While MH isolates processes with a very tiny attack surface, the traditional HW-based full virtualization isolation is responsible to isolate the VM execution environment with the other LL processes.

### 6.2.2 Upper Layer

The Upper Layer (UL) represents the nested virtualization stack which is under the control of the consumer. The consumer, leveraging its virtualization technique of choice, has total control on L1 virtualization and can customize its own hypervisor as mentioned above. Moreover, the consumer can deploy here its L2 execution environments (e.g. VMs, containers) which encapsulates its workloads and which can be moved easily across different CSP.

Comparing the U-Cloud architecture with the ORBITS template presented in Chapter 5, the UL design coincides de facto with the ORBITS virtualization layer, as reported in Figure 6.2. Leveraging IaaS low-level abstractions, ORBITS provides the L1 virtualization layer and executes the L2 execution environments. At the end of this chapter, we discuss in detail how the U-Cloud can be integrated to ORBITS through the MANTUS pipeline.

It is worth noting that a key limitation to the UL customizability is represented by the virtualization technology that can be used by ORBITS for the "virtualization layer". If the LL hypervisor is crafted to enable NV through support for running nested VMX instructions, as presented in [19], hardware-based full virtualization can be leveraged for ORBITS. Otherwise, similarly to the most part of public cloud providers, the UL should rely on containerization or para-virtualization. In order to cope with nested virtualization hardware limitations, we adopted Xen as UL virtualization layer. Xen supports a wide range of virtualization techniques, from Para-virtualization to hardware-assisted virtualization, with several hybrid configurations [113]. Similarly, lighter virtualization mechanisms as Docker, LXC or other container-based technologies can be adopted without requiring any adaptation work.

### 6.3 U-Cloud Implementation

In this section we present the implementation work done to implement the U-Cloud architecture presented above. We realized a prototype of the U-Cloud Node leveraging respectively the Nova and Genode components. NOVA [108] represents an MH which integrates the design architecture presented in Section 6.1.2.2, which shows very good performance and it has made available as open-source software. Due to its tiny core and its native virtualization support, it is an ideal candidate to be used as LL core hypervisor. Genode [91] is a micro-kernel open-source ecosystem, which provides a set of operating system components (e.g. device drivers) and a support for virtualization. As we will present in the next sections, we adopted and extended several components of those projects to obtain a prototype of a virtualization node capable to realize the improved
control architecture we discussed above. This prototyping work, as later presented in Section 7.1.2, was performed in order to evaluate the feasibility of the approach and to quantify the overhead introduced by the "extra" infrastructure layer. In this section, as illustrated in Figure 6.3, we map functional components highlighted in Section 6.2 to existing or new software components.

6.3.0.1 Reused components

We provide a list of components we adopted from open-source projects, mainly GENODE to construct the U-Cloud architecture. We discuss in particular the base micro-hypervisor component, NOVA, which is responsible to ensure isolation between different customers.

MICRO-HYPERVISOR NOVA OS Virtualization Architecture (NOVA) [108] is an open-source research MH that combines high-performance hardware-assisted virtualization for the x86 architecture with a minimal TCB (around 10 KLOC) and the capability-based security design of the L4 micro-kernels [73]. These characteristics makes it an ideal candidate for the LL hypervisor of our architecture.

Genode [91] is an open-source ecosystem integrated with NOVA which provides ready-to-use drivers for the most common devices, and several programming primitives making the link between the kernel API and user-level applications.

Genode has two interesting features for our purpose, which are not present in all micro-kernels/micro-hypervisors:

- **Recursive protection mechanism** introduces complete encapsulation of child process, having responsibility and control for the resources allocated to it and its capabilities. The child process is unaware of any other process or resource exists in the system, enforcing a complete isolation between different consumers.
**Text-based configuration** Genode leverages a component, *Init* to configure the execution of all services on the node. *Init* has a native "Infrastructure as Code" handler which describes the "tree of processes" of a sub-system to execute at runtime.

**Command Center** is responsible of resource multiplexing among different U-Cloud allocated on the node. Even if does not own the resources allocated to U-Clouds (e.g. it cannot inspect the content of U-cloud memory), it has to authorizes each request for resource quotas update and can unilaterally revoke the assigned resources.

**Drivers and VMM** The drivers are the interface between the physical devices and the system. They allow Genode components to interact with the devices through dedicated sessions. Because a device driver only allows one simultaneous session to be opened, in systems where several components must share a physical resource, multiplexers must be added to seamlessly handle it. Micro-kernel approach pushes device handling in user-space, reducing the dramatically TCB but complexing the architecture of device drivers.

We adopted Virtualbox port to NOVA as L1 hypervisor. "Upstream" Xen and LXC were adopted as L2 virtualization layer.

### 6.3.1 Newly Introduced components

Based on this initial subsystem, we developed several system services to address (1) complete functionality of virtualization system (2) isolate user and system administrator domains and (3) enable a secure remote control over deployed resources.

We implemented a **Network Switch** as a network software switch which was missing in Genode framework which introduces network connectivity through the "backward learning" algorithm over multiple VLANs. A different instance is created in each U-Cloud to enable connectivity for user nested VMs and hypervisors. A privileged instance of the switch has to be executed among the administration services to multiplex Network Cards (NIC) resources among different users.

Due to the requirements in terms of dynamic configuration of service that we detail in next section, we introduced the **Init User** as a modified version of the standard *Init* service developed by Genode. Similarly as the traditional *Init* does for the entire system, *Init-U* instances acts as "root" of every user domain and for the provider domain, thus representing the single interface for their interactions with the rest of the system. We introduced the possibility to dynamically reconfigure the consumer subsystem defined in *Init* without having to trigger a redeployment of the entire set of applications, which can be useful in order to reconfigure a U-Cloud or, as we detail more precisely in next section, with custom services a bare U-Cloud.

The **Domain Loader** deals with the delegated instantiation of the consumer U-Clouds. Based on an existing Genode component, it was extended to be a common launcher to create all consumer domains and to act as interface between them and the system. More precisely, it acts as the enforcement point to create U-Cloud and admin their resource allocations. In particular, it acts as a black-box handler from the perspective of the node administrator, since provider software stack and domain loader does not belong to the same hierarchy of processes and therefore provider software stack has not the privilege...
to snoop and inspect U-Cloud execution domains. In a nutshell, while protecting the
launched domain from interference by the provider, the Domain Loader represent a
trusted component outside of the provider domain that consumers have to trust.

The **Command Center** implements the CSP policy in terms of U-Cloud creation and
it acts on the Domain Loader interface to create U-Cloud domain and reserve hardware
resources (e.g. memory) for the new U-Cloud.

The **Control WebService** represents the remote entry-point to the U-Cloud Node,
which validates requests and passes them to the Command Center. It represents the
only control service which is exposed for remote resource allocations, and it is the de-
coupled by the decision logic implemented by the **Command Center**. Similarly to the
control web service, each U-Cloud is composed of an **Admin Web Service** which is re-
ponsible to expose an API to let the consumer customize the U-Cloud domain, without
having to rely on Provider stack components.

### 6.4 **U-Cloud Node Workflows**

In previous section we detailed the U-Cloud architecture, detailing the necessary com-
ponents to build it. In this section, we details how the node implementing the above
presented architecture is capable of constructing the "U-Cloud" abstraction for different
customers, guaranteeing a "black-box" control from the Provider.

In order to illustrate the behaviour of the U-Cloud, we present the base functioning of
the U-Cloud architecture presenting as example how it is capable to handle the construc-
tion or the reconfiguration triggered by a user. First, we present the steps that lead to the
construction of a U-Cloud. Second, we show how the architecture is capable to handle
a dynamic service reconfiguration without having to redeplo virtualised environments.

#### 6.4.1 **U-Cloud deployment**

In this section we describe the U-Cloud deployment process. The first steps leading
to U-Cloud domain creation are depicted in 6.4 and are followed by those described
in Figure 6.5, which concerns the configuration of an already allocated U-Cloud. We
recall that the virtualization architecture we present is characterized by two different
stack of services deployed in the IL: (1) one defined and controlled by the provider/ad-
ministrator which comprehends hardware devices and resource multiplexing and a (2)
the U-Cloud "stack" which comprehends the list of services deployed by the consumer.
In the following we assume that the consumer is responsible of the initiation of the
U-cloud construction. However, the construction of the U-Cloud can be triggered indi-
directly by the driver of cloud management systems (e.g. OpenStack), performing some
pre-configuration on the target U-Cloud.

In order to configure a virtualization node, we propose the following protocol (Fig-
ure 6.4):

1. The client performs a request to the control web service. The request should con-
tain a header specifying an authentication token for the client. The control web-
  service parses it and checks its conformity with the expected format and should
  validate the token with an external identity entity (e.g. Keystone).
2. If the request is appropriate, the web service opens a temporary session on the Command Center and uses it to notify him of the request. The Command Center, responsible of resource sharing across different users, then validates the request.

3. It instructs the Domain Loader component by RPC to build an initial U-Cloud instance, sending a minimal configuration to deploy and setting an initial amount of resources. The Init User is created by the Domain Loader, and receives the minimal configuration, a "bare" U-Cloud. It deploys the admin web service with the correct IP address.

4. Once the Domain Loader has deployed the initial U-Cloud, the Command Center informs the control web service that the request has been processed and the instantiation is ongoing. The HTTP endpoint coordinates of the allocated U-Cloud admin web-service is sent back to the consumer by the Command Center.

It is worth noting that the step 4 ensures that the consumer’s client receives the endpoint of the admin web service and, therefore, it will be able to contact the admin web-service that was created inside its U-Cloud.

So far, the client has only told the "Provider Stack" (the orange stack in Figure 6.4) that he necessitates to instantiate a U-cloud. The Command Center, which controls resource allocations, have acknowledged to create a "bare" U-Cloud. A "bare" U-Cloud represents an instance of the Init and the "Admin Web Service", without any other software instance. This represent the minimum to bootstrap a user-subsystem. Once a U-Cloud is instantiated, the client is able to deploy desired services, as described in Figure 6.5:
1. After obtaining the IP address of the admin web service of its stack, the client can contact it. The client can specify the desired configuration for the U-Cloud (e.g. leveraging the Genode Init XML format). The admin web-service only parses the HTTP request, leaving the processing of the configuration to the Init User.

2. The admin web service sends the XML configuration to the Init User (Init User). The Init-U parses it and validates its correctness. For that reason, U-Cloud policies for the architecture are defined through the request processing infrastructure of the Init User: the available services, the necessary and the possible configuration informations, etc.

3. If the topology of services described in the XML is valid, the Init User starts deploying it, negotiating with the Domain Loader the amount of resources. Every resource upgrade request is relayed in turn to the Command Center, which decides whether it should be accepted. In case of invalid or deliberately improper configuration, only the Init User (and the corresponding U-Cloud) will be affected.

4. If the configuration contains a VirtualBox instance, the Init User expects it to specify the type of VM that should be launched. It can thus perform a VM setup request to the Disk Manager for the aforementioned VM type.

5. Once the configuration has been deployed completely, the Init User notifies the admin web-service, which in turn replies to the client.
6.4 U-Cloud Node Workflows

6.4.2 U-Cloud Reconfiguration

In this section, we analyze the possibility for a consumer to reconfigure a U-Cloud without major service outage. The usage of nested virtualization enables the possibility of decoupling the Level 2 VM from the state of UL Hypervisor and LL User Level Services which are part of the U-Cloud. In particular, the consumer may be capable to reconfigure LL U-Cloud services transparently leveraging live migration [34]. Such approach introduces a further degree of flexibility and may be interesting when the user detects a misbehaviour inside the U-Cloud services, which cannot be monitored by the administrator by default or the user may want to change its service topology (e.g. for example to cipher disk accesses).

We recall that L2 VMs are virtualized over a full-fledged hypervisor and therefore supports all advanced features supported by commodity hypervisors (e.g. Xen, KVM). One of these, live migration allows to move without major interruptions a VM. Such L2 live migration is handled by the Xen hypervisor and such hypervisor supports live migration [34]. In our architecture, those migration can be performed between "in-host" (i.e. on another Xen instance in another U-Cloud) on different consumer stack or "cross-host" (i.e. another host where a Xen instance is running). The sole constraints are represented by low latency network connection (e.g. such as a Local Area Network) and a common networked file storage location (e.g. NFS) to store L2 virtual machines disks to move them across multiple hosts. We identify "in-host" migration when the L2 VM is simply moved from a U-Cloud subsystem to another one on the same physical machine. On the contrary, "cross-host" implies a live-migration on a different physical host.

In any situation, L2 VMs is affected by reconfiguration of the virtualization stack since it happens outside the VM domain. In fact, it is worth noting that the proposed architec-
ture completely decouples between U-Clouds and the rest of the system. Consequently, this prevents the necessity to treat differently intra-host or cross-host or eventually cross-provider live migration, even if such migrations are handled with a different timing.

Figure 6.6 illustrates a workflow of an in-host live-migration a L2 VM. To perform such operations, the U-Cloud node implements the following:

1. The client requests a new U-Cloud as previously done for the U-Cloud creation (See Section 6.4.1).
2. The request is transmitted to the Command Center, which processes it.
3. The Command Center then transmits its instructions to the Domain Loader, which deploys the U-Cloud.
4. The client can then deploy in the U-Cloud the new configuration. Modifications could range from simply modifying the L1 hypervisor (for example updating it to a more recent version) to deploying user services in the L0 (for example making all network access in the U-Cloud pass through a firewall component).
5. Supposing both U-Clouds are now deployed with interoperable L1 hypervisors, the client can use their control interfaces to migrate the L2 VMs from the outdated one to the updated version. Most hypervisors support migration without considerably affecting the guests run time.
6. When the migration is finished, the client can use the administration interface of the outdated U-Cloud to exit it. Only the updated version, now running the L2 guests, remains.

To sum up, the U-Cloud node introduces an important degree of flexibility w.r.t. the consumer which is capable not only to build its own stack of system services but also to rely on features already present today in common hypervisors to reconfigure it.

In next section, we present how the U-Cloud node can be integrated with ORBITS and integrated in the MANTUS pipeline.

6.5 INTEGRATION WITH THE ORBITS ARCHITECTURE

The U-Cloud architecture can be easily integrated with the ORBITS architecture leveraging TOSCA flexibility, as we discussed previously in Section 4.3. In fact, TOSCA enables complete transparency to underlying technology adopted. The U-cloud components mentioned above can be packaged through TOSCA similarly to the ORBITS architecture components. If a specific CSP supports the U-Cloud architecture and the consumer selects this feature in its context, MANTUS will match the abstract service definition to a compelling set of U-Cloud TOSCA-packaged implementations. In other words, MANTUS context-matching enables transparently the selection of those modules if the candidate data-centers supports them in its context.

Consequently, this enables to customize the virtualization layer by modeling the composition of "U-Cloud" through TOSCA as described in Section 6. For instance, the matched TOSCA NodeTypes would eventually be translated to the GENODE XML format by the context-based translator. The translator driver implements the deployment and management protocols described in 6.4, transparently leveraging the higher control of
Figure 6.7: The MANTUS workflow integration with a U-Cloud Node. When a U-Cloud Node Provider is selected as candidate, the resulting Infrastructure code is matched and weaved using its provider context and then translated into compatible Genode and Openstack resources.
remote user over the infrastructure. In particular, MANTUS classifies providers by type: multiple DCs with their own context belong to the same provider type. The provider type determines which orchestration technology (e.g. CloudFormation, ARM, Heat) is used and consequently the same translation component in MANTUS.

For Genode, we can combine its node specification inside a modified MANTUS Openstack translation driver. When the context-based provider matching selects the "Genode" Provider, it will leverage the traditional translation for services which are in the ORBITS layer, and the "Genode" code for the ones which are in the hypervisor layer. We sketched the concept in Figure 6.7, applying the MANTUS algorithm to a Genode-based CSP. When the algorithm obtains concrete node, the driver logic maps the final Tosca NodeType to a specific resource. We may observe that MANTUS leverages the specific U-Cloud-based Provider context and it provides as output "hybrid" NodeTypes (dark blues and light blue boxes), which describes resources respectively for the LL (Genode Specific NodeTypes) and UL (Openstack Heat NodeTypes).

Such resource-oriented specification is completely compatible with TOSCA and the extensibility of TOSCA nodeTypes ontology can be easily translated in Genode components similar as done for other CSPs, as presented in Figure 6.7, before triggering the deployment.

6.6 CONCLUSION

In this chapter, we presented the U-Cloud node architecture to introduce for the consumer enhanced control over remote infrastructure. The U-Cloud architecture leverages an user-defined virtualization architecture. Adopting an hybrid approach among microhypervisors and nested virtualization, the U-Cloud architecture presents a secure but legacy-compatible stack, which separates services deployed by the user and the ones under the provider responsibility. The consumers can define their "U-Cloud" by cherry-picking the components to insert in their own TCB, without legacy compatibility disruptions.

In next chapter, we present several implementation of MANTUS, ORBITS and U-Cloud architectures presented so far. We analyze evaluation results performed on those prototypes to asses the feasibility, performance and scalability.
In this chapter, we evaluate the three main contributions of this thesis by implementing prototypes for the provisioning of multi-cloud that makes use of what we have introduced. We recall the three main contributions in the following. The first contribution, presented in Chapter 3, was to overcome the lack of extensibility of the code describing non-functional services across different CSP ecosystems. More precisely, we defined a new language, called TML, to inject non-functional features into infrastructure templates. The second contribution, presented in Chapter 4, was to mitigate the lack of specialization of state-of-the-art IaC matching mechanisms when integrating the different CSP resources into the multi-cloud architecture. We did so by introducing the “context-based matching” mechanism to the existing TOSCA matching process. To complete the definition of the multi-cloud deployment process, in chapter 5, we introduced an overlay architecture to deal with multi-cloud interoperability. We showed how it can be used to construct and manage the lifecycle of a multi-cloud by leveraging extensibility and specialization introduced above. The third contribution, presented in Chapter 6, was to tackle the loss of control of the currently deployment methods of multi-cloud infrastructures. More precisely, we proposed ORBITS, an overlay infrastructure as a common layer for the multi-cloud, and a user-controlled virtualization stack that we named U-Cloud node, where the consumer has an improved control over system resources.

The first and the second contributions are evaluated through the performance of MAN- TUS, the multi-cloud builder that we have designed. MANTUS uses the TML as the language to describe the extensions of TOSCA templates and leverages the context-based matching for the selection of the CSPs to be deployed. More precisely, the first contribution on extensibility is evaluated by testing the efficiency of MANTUS for allocating a multi-cloud against builders without this feature. In fact, extensibility introduces a further step to be run when building a multi-cloud and possibly slows down the entire process. Our objective with this evaluation is to analyze how longer it takes for MANTUS to allocate a multi-cloud. The second contribution on specialization is evaluated by testing how optimized are w.r.t a certain aspect the multi-clouds built by MANTUS compared to ones built by the other multi-cloud builders. The introduction of the context in the matching process allows a user to specify what aspects of the multi-cloud infrastructure should be optimized or minimized. For example, he might specify to allocate a multi-cloud infrastructure that minimizes the cost or that maximizes the reliability of CSPs. This leads to a more fine-grained selection of the CSPs that takes into account these aspects. Given a certain aspect to be minimized or maximized, our objective with this evaluation is to test whether the multi-clouds built with MANTUS are actually optimal in this aspect compared to the multi-clouds built otherwise.

The third contribution is evaluated through the performance of ORBITS and of the U-cloud node. ORBITS is an overlay architecture that provides a common layer above
multiple providers to fulfill the interoperability issues among different CSPs systems. U-cloud node is a virtualization architecture that overcomes the loss of control of cloud consumers. Our objective is to evaluate the performance overhead of ORBITS and U-cloud node compared to traditional overlay and virtualization architectures, respectively.

The outline of this chapter is as follows. First, we discuss infrastructure related aspects, discussing performance of ORBITS and U-Cloud (Section 7.1). In Section 7.3, we present the first MANTUS prototype and we analyze benchmarks executed to evaluate the scalability and performance of the extensibility. In Section 7.4, we present an enhanced MANTUS prototype and we assess the added value of context-based matching and its impact on the overall MANTUS workflow. We do this (1) by constructing the MANTUS broker context, analyzing its impact on a simple deployed infrastructure and (2) analyzing context-based matching impact in the MANTUS workflow.

7.1 ORBITS OVERLAY INFRASTRUCTURE

In this section, we discuss a possible ORBITS implementation, focusing in particular on which class of services can be used to compose an ORBITS architecture implementation (Section 7.1.1). Afterwards, we validate the feasibility of this ORBITS architecture implementation in terms of performance and scalability (Section 7.1.2).

In Section 5.1.1, we discussed the ORBITS architecture, showing how orchestration, management and virtualization layer was responsible to respectively orchestrate the entire U-Cloud execution, manage local resources and provide isolated execution environments to applications. Focusing on implementation, we wanted to validate such approach in a context for applications which were not specially designed to be executed in a multi-cloud environment.

In particular, if we look at software engineering nowadays, we can consider existing applications living a major transformation in the way they are conceived and implemented. More precisely, we can identify a first new class corresponding to applications which are implementing micro-service paradigm. Several important advantages compared to traditional monolithic pattern: scalability, high availability and simplified life-cycle. Those applications normally require a container orchestrator to be properly delivered. As mentioned in Section 2.1, "containerization" [107] is a model of virtualization where the isolation is handled by the Operating systems. The flexibility of this solution pushed to couple the life-cycle of the execution environment with the life-cycle of one single process executed inside [41]. This last paradigm can improve application portability and boost application delivery process.

Container orchestrators, as Kubernetes [68] or Docker Swarm [70], propose to leverage containers [41] as portable virtualization layer for micro-service applications. Container orchestrators are normally already designed to fit the ORBITS specification and implement the local ORBITS "management" layer and provides a support for multiple orchestrators federations. For instance, in the case of Kubernetes, the Kubernetes Federations [69] allow to spawn applications based on multiple instance of the orchestrator on different CSPs.

On the other hand, traditional legacy applications does not easily fit the Kubernetes model and are difficult to be integrated to this new model anytime soon. Those are not by default portable to a multi-cloud environment, mapping existing services to the ORBITS architectures become more complex. Consequently, we designed a "legacy ap-
applications” ORBITS architecture in order to let them be executed in the context of multi-cloud without losing important features w.r.t. container orchestration. We present this in the next section. Use the “Docker-like” containers may represent a problem for legacy applications. In fact, those applications are often monolithic and makes their management when dealing with ephemeral docker containers more complex. First, they may be composed by several processes, not respecting the pattern “one process, one container” typical of Docker. Second, the presence of multiple processes inside a single container tends to make the detection of anomalies more difficult (e.g. unexpected memory/CPU consumption). Third, they are not designed to be managed as application containers are designed to be (e.g. initialization scripts) but often require manual actions to be executed inside the application domain (e.g; through a remote connection), which imply that a dedicated daemon (e.g.; SSH) has to be executed inside, violating again the pattern.

7.1.1 OpenStack-based ORBITS implementation

![Diagram of OpenStack-based ORBITS implementation](image)

Figure 7.1: We maps a OpenStack-based ORBITS prototype w.r.t the abstract architecture. The *overcloud*, which comprehends the virtualization and management layer, is executed on each CSP while the orchestration layer is distributed across multiple CSP.

Therefore, to cope with those "legacy" applications which struggles to fulfill the micro-service architecture requirements, we oriented the ORBITS principles to a traditional cloud infrastructure which provide server-oriented execution environments. The local resource provisioning is delivered by an overlay OpenStack instance for each selected provider, leveraged as a management layer. The entry-point of the multi-cloud is a
set of cooperating instances of Mesos Masters which provides a multi-cloud-aware and application level service orchestration. The STRATOPAUSE role is played by a Mesos slave used as a proxy, which advertises resources to the set of Masters and deploys jobs accordingly to Master dispatch, connecting the local provisioning to application orchestration. Considering virtualization layer, we experimented a wide spectrum of system-virtualization technology, considering KVM, Xen and LXC.

In order to develop an ORBITS template, we firstly introduced a hierarchy of the TOSCA resource types to model infrastructure services, focusing on computing and networking resources. Infrastructure components were described as TOSCA templates using our custom *NodeTypes*. We also defined an encapsulating ORBITS *overcloud* type to foster template reuse for multi-cloud environments.

We mapped those three layers with the following software components:

**Orchestration Layer**  
Apache Mesos [12] is used for application provisioning. Mesos implements a two-layer architecture, based on Master and Slaves. The whole framework is represented in Figure 7.2. Mesos Masters implements two layer of scheduling. First, they split available resources across different workloads. Second, each "applications" rely on a framework, workload-specific logic, which is responsible to select the corresponding slaves. Slaves advertises their offers to masters, specifying the quantity of resources available and a custom set of parameters (e.g. specific hardware capabilities) that only certain framework are able to understand. Each framework can accept one of those "offers".

![Figure 7.2: Mesos Two-level scheduling mechanism](image)

In ORBITS, we leveraged this logic as follows. Each ORBITS template is deployed with an active slave, in the configuration file of this slave. This context is advertised to masters by the slave instance. The master server will therefore reconstruct the topology of multi-cloud, being aware of the selection done by MANTUS. It is worth mentioning that Mesos Master and Slave code is not modified, disrupting legacy components. The only modification required to benefit from MANTUS-based multi-cloud is to write Mesos frameworks capable to understand context flags advertised to the master.
7.1 ORBITS OVERLAY INFRASTRUCTURE

Figure 7.3: ORBITS OpenStack Prototype. We selected a subset of OpenStack components to compose the Virtualization and the Management layer and Mesos to be part as Application Orchestration logic.

MANAGEMENT LAYER To implement the ORBITS management layer, we leveraged OpenStack, which represents the most adopted IaaS cloud management system. We included common services part of the OpenStack ecosystem to handle compute and images (Nova and Glance), networking (Neutron), access management (Keystone).

VIRTUALIZATION LAYER We adopted several virtualization technologies to validate their performance in a nested configuration. We focused in particular on Xen and KVM, leveraging the VM template provided by OpenStack in terms of hardware description. We also tested LXC as an OS-based system virtualization alternative to Full Virtualization. The OpenStack Nova “filter & weight” with its drivers-based mechanism allows to transparently handle heterogeneous compute nodes in the same OpenStack instance.

7.1.2 Performance Evaluation

We evaluate the overhead and the performance loss introduced by the overlay infrastructure layer. In fact, we introduce a further consumer-controlled layer of virtualization to provide the consumer more flexibility. However, as many other works reported, nested virtualization may introduce severe performance penalties that can penalize the effective usability of ORBITS architecture.

First of all, we evaluate whether the approach of having a "nested" infrastructure versus a CSP-controlled ones is acceptable. To assess the overhead due to the nested virtualization, we evaluated our system both in terms of performance and scalability. To this
end, network latency and bandwidth represent important parameters to influence the execution performance of health-care applications, as analysed in Section 1.1. The tests were performed on a Intel Xeon E5-2650 Haswell at 2.60GHz with 64 GB of RAM and Centos 7 as bare-metal operating system. The base software platform is an OpenStack over Linux KVM virtualizing Ubuntu 16.04 guests, with paravirtualised VirtIO drivers network card and disk.

In Figure 7.4-7.5, we present a comparison between nested virtualized EEs (VM plus containers), single-layer VMs and bare-metal. Degradations are in particular concentrated in the nested KVM configuration, which have often overheads over 50 %, compared to the baseline. In particular, LXC performs quite well and may be considered as a viable solution to introduce a ‘user-controlled’ layer of virtualization.

In Figure 7.6-7.7, the scalability of nested EEs was tested towards an increasing load in a Wordpress application. A wordpress application, similarly to a great part of healthcare applications [38], relies on a web front-end, a server-side application logic and access to a DB. It could be used as a generic and representative benchmark. Both from the perspective of throughput and elapsed time, Xen and LXC perform well remaining below 20% of overhead.

In addition, it is worth noting that, the control of the nested layer of virtualization on public cloud enables the possibility to have physical co-localization [5, 96] which may enable better performance independently from the underlying provider, in the context of multi-EE applications.

To sum up, the results show that only OS-level technology can properly be nested over traditional full virtualization platforms without a major performance loss. LXC would allow to create “lightweight VMs” without having to integrate the Docker application-container model, while using the same OS-provided virtualization as presented in Section 2.2.1.3. Using Xen and KVM, consumer have to face an important performance drop.
Figure 7.5: In this Figure, we show the TCP throughput of different configuration tested with "qperf" utility. We observed that all virtualized configuration suffer minor throughput when dealing with lower payloads and more packets to process. When the size of TCP fragments grows, performance tends to converge to line rate.

Figure 7.6: We tested the different ORBITS configurations in an application benchmark, counting the latency in request served by an Apache Server running a dynamic site, a Wordpress instance, with an increasing number of parallel connections. Xen and LXC configurations introduces an overhead which increases with the number of parallel connections, while KVM present a significant higher overhead.
7.1.3 U-Cloud Node

After having tested the feasibility of ORBITS approach and its acceptable cost, we performed the similar kind of tests on the U-Cloud prototype detailed in Section 6.3. In this section, we detail some performance benchmark discussing experimental results.

We firstly evaluated the effectiveness of reconfiguration and the lack of impacts on L2 VMs, as showed in Figure 7.8. We allocated a first U-Cloud, spawning a L2 VM over a L1 Xen Hypervisor. We then spawned a second U-Cloud on the same host with another L1 Xen hypervisor and a different L0 configuration. We were able to move a Xen L2 paravirtualized VM across two different U-Cloud instances using Xen live-migration feature. We validated it in two different scenarios. Moreover, we tested also the live migration between a traditional Xen domain and the U-Cloud node. We opened a remote control ssh connection to the L2 VM and we remained connected to the remote terminal through the whole experiment.

Despite the functional validation of the benchmark, we experienced several important performance issues related to CPU scheduling. First, we experimented several performance problems due in particular to the absence of a fine-grained scheduling, similarly as the one provided by the Linux kernel. Nova [108] does not rely on a complex scheduler algorithm as Linux but implements a Round Robin and fixes a certain process to a dedicated core at start-up. The degradation of performance we incurred in was present in particular during the execution of L1 VMs with multiple vCPUs on the same physical cores.
Second, we observed many network performance problems. Using a TCP bandwidth test (qperf), we benchmarked the traffic produced by L1 VM running Linux being on the receiving end, progressively increasing the size of the transmitted messages. In micro-kernels, network packets are transferred as common IPC messages among processes and may represent a bottleneck when dealing with a lot of IPC messages on different CPUs. To quantify the IPC bottleneck, we were able to partially evaluate the networking performances of the switch. In Figure 7.9 and 7.10 we show the obtained results.

Figure 7.9 shows evaluations obtained using only one vCPU for the VM, executed in parallel with the "CSP" services on the first physical core. The CSP services included the network interface driver which was responsible to send the data to communicate to the switch. Figure 7.10 executed the vCPU on a separate core. The red line represents the mean value obtained over several tests. Executing the VM on a dedicated core understandably greatly improved performances: the NIC driver and the switch not operating in concurrence with the receiving network stack may be beneficial. However, the switch do not work at full line rate: the maximum bandwidth obtained was around 50 MB/sec.

Furthermore, we repeated the same test observing how a multiple VCPUs L1 VM can perform. Nova allows to "pin" VCPUs on physical CPUs and we leveraged this feature to test the effect of different vCPU configurations. Figure 7.11 has been obtained with two vCPUs for the L1 VM which executed Xen, placed on the two different physical cores. The results were very unstable, and performances were actually severely inferior to the VM based on a single vCPU from figure 7.10. Moreover, adding vCPUs can, of course, negatively impact performances: complex synchronization mechanisms are put in place, so the VM is only able to run if it can simultaneously schedule its vCPUs on different physical cores. The low availability of the physical core 0 could explain partially the results, due to the concurrence with "Provider" components (device drivers, resource multiplexers).
Figure 7.9: TCP Bandwidth test deploying L1 VMs composed of a single vCPU and the Provider-subsystem on the same physical core

Figure 7.10: TCP Bandwidth test deploying L1 VMs composed of a single vCPU and the Provider-subsystem on different physical cores
7.2 CONCLUSION

To sum up, experimental results show that the performance and scalability loss of ORBITS architecture due to the adoption of an extra layer of virtualization may be affordable. Nested VMs in all configuration seem to introduce a very important overhead, which remains acceptable in case of nested containers. In particular, the hypothesis of leveraging nested upstream "Xen" as L2 hypervisor, similarly as indicated in [6], may not always provide very good performance.

Despite more rich in terms of proposed features, it is worth mentioning two major drawbacks that ORBITS introduces. First, infrastructure-oriented multi-cloud is heavier in terms of maintenance and resources deployed compared to the bare allocation of application resources, resulting in an higher cost in terms of deployment and management. Nevertheless, overlay-based approaches provide the user with an important level of control (e.g., virtualization layer, security appliances), but requires effective way to dynamically modify the infrastructure templates.

Moreover, we functionally validated the U-Cloud prototype, being capable to configure for different consumers dedicated sub-systems and then reconstruct them transparently for L2 VMs. However, we encountered multiple performance issues which may impact a straightforward utilization of the prototype and require careful operating system tuning. Performance tuning appeared to be a tough task to realize. This customization of virtualization layer is similar to the one proposed by [7] but without requiring careful engineering from the consumer thanks to MANTUS context-based matching. IPC performance are strongly dependent on which CPU is scheduled. After having demonstrated the feasibility of the approach, we preferred to focus on dynamic reconfiguration of OR-
BITS template performed by MANTUS, in order to show how the multi-cloud compiler is capable to handle user parameters and provider configurations.

7.3 MANTUS 1.0: INITIAL PROTOTYPE

In this section, we present the first MANTUS prototype which was developed with the focus of (1) have an end-to-end complete multi-cloud deployment and (2) introduce extensibility in the TOSCA workflow leveraging TML. The first Mantus prototype includes the following phases:

- **Matching** A ServiceTemplate is created, composed of concrete nodes starting from an abstract specification. We adopted the standard TOSCA plug-in matching algorithm similarly as presented by Brogi et al. [26].

- **Fusion** The matched file is generated, reconnecting different branches of matching.

- **Weaving** Non-functional services are injected by applying TML scripts against the TOSCA ServiceTemplate as described in Section 3.3.1.

- **Translation** Finally, the resulting TOSCA templates are translated to the Heat Orchestration Language (HOT) to be deployed on an OpenStack CSP as a target example of CSP.

We implemented the MANTUS TML weaver in Python. MANTUS first builds the expression abstract syntax tree for resources in the TOSCA graph using the pyPlus LR-parser. We used the TOSCA-parser library to manipulate TOSCA ServiceTemplates for a fully compliant TOSCA generation. We also developed a driver-based small TOSCA translator supporting HOT and CloudFormation back-ends. The translator was developed from scratch to keep it minimal and to put an emphasis on polymorphism for multi-CSP support.

We completed this prototype by implementing a hierarchy of TOSCA NodeTypes to model ORBITS infrastructure services. We described requirements, capabilities, operations and properties of computing and networking resources, such as Virtual Machines and virtual networks, which are part of the ORBITS prototype mentioned in Section 7.1.2. Such NodeTypes were then applied to a full-fledged cloud infrastructure based on OpenStack and Mesos, using Xen, KVM and LXC as core virtualization technologies: infrastructure components were described as TOSCA templates using our custom node-Types.

In the next section, we introduce a prototype of the MANTUS multi-cloud builder, focusing on the TML weaver component. We introduce here only the key elements of its workflow, in order to provide a first benchmark of AOP-based weaving.

7.3.1 Mantus Extensibility Evaluation

To recall briefly the AOP-weaving, in Section 3.3.1, we presented an AOP-based weaving to extend a multi-cloud infrastructure through TOSCA graph manipulation. To do so, we designed a DSL called TML to model the logic which is able to express graph modifications. In a nutshell, when the AOP-weaving is triggered, MANTUS analyzes the

1 https://github.com/erezsh/plyplus
2 https://github.com/openstack/tosca-parser
TOSCA graph trying to check properties expressed in TML scripts, precisely in the "filter" section. If the filter condition is met, the "action" section is triggered and the TOSCA graph is modified. Actions may concern add/remove/update resources and links, which are respectively the nodes and edges of the infrastructure graph. Furthermore, as we detailed in Section 5.2.2, the AOP-weaving can be triggered not only at deployment, but also incrementally, to react to a specific event (e.g.: intrusion detection), acting on the multi-cloud infrastructure at runtime.

We now validate our approach through several benchmarks. We selected three main type of experiments:

- **Performance**: Infrastructure extension requires service injection to be effective, supporting timely weaving. To do so, we evaluated weaving overhead in overall deployment, considering different steps of the pipeline, we compared different aspects design to evaluate the best strategy to model services. More precisely, to assess performance, we analyzed incremental weaving efficiency, when a given TML script is executed over a TOSCA infrastructure, which can provide us the behavior of weaving with a TOSCA template of increasing size. In addition, we weighted the cost of weaving in the whole MANTUS workflow, analyzing the time required to perform each step while increasing the complexity of templates.

- **Scalability**: We analyzed the scalability of the system in terms of number of injections and size of scripts. More precisely, we analyzed the compositional efficiency (multiple weavings) to evaluate how the time elapsed for weaving evolve w.r.t. aspect complexity. Such scalability concerns not only the size of a TOSCA infrastructure i.e. the size of the graph, but also the size and the number of TML scripts.

- **Dynamicity**: The support of dynamic reconfiguration of service, in particular "incremental injections" to be able to add or drop services during execution. We analyzed the cost of incremental weaving during infrastructure life-cycle.

In Table 7.1, we provide a reference for different variables used in the following benchmarks. Those variables capture different dimensions that can vary during the matching and weaving process. First, $T$ models the initial number of TOSCA Template nodes, the inputs of the MANTUS deployment. $S$ corresponds to the number of scripts, which contain TML Filters, Resources and Edges to be added to the graph. In fact, $F$ models the number of "Filters field" used to analyze the graph of TOSCA resources as described in 3.3.1, to find properties that can trigger the injections of new resources $R$ and edges $E$.

We used, as input template, the basic OpenStack-based ORBITS template, as the testing input for all the benchmarks. In a nutshell, this template is an example of "overlay cloud infrastructure layer", composed of a "controller" node and 3 different instances of "compute nodes" (Xen, KVM, LXC), with two virtual networks (internal and external access (Internet)) and relative network resources (ports, routers). After the Matching and Fusion phases, the resulting template used for running benchmarks is a graph of 33 nodes in the MANTUS workflow, starting from a single Overcloud component.

The following tests were performed on a Intel Xeon E5-2650 Haswell at 2.60GHz with 64 GB of RAM with Ubuntu 16.04 LTS with a SSD Samsung EVO 740.
Table 7.1: Benchmark Variables

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Number of initial Template Nodes</td>
</tr>
<tr>
<td>F</td>
<td>Number of filters per TML Script</td>
</tr>
<tr>
<td>R</td>
<td>Number of injected resources per TML script</td>
</tr>
<tr>
<td>E</td>
<td>Number of injected &quot;edges&quot; per TML script</td>
</tr>
<tr>
<td>S</td>
<td>TML script characterized by S[R;E;F]</td>
</tr>
<tr>
<td>W</td>
<td>Weaving operation W(T,S)</td>
</tr>
<tr>
<td>A</td>
<td>Coefficient of replication</td>
</tr>
</tbody>
</table>

7.3.1.1 Performance

**Incremental Weaving Efficiency** In a nutshell, we want to evaluate the cost for the weaver to inject more TML scripts to a base template. First, in Figure 7.12, we analyze the marginal cost (i.e. weaving efficiency) in terms of execution time, also showing standard deviation of samples. The analysis of marginal cost can provide a better insight of how the weaving phase is capable to handle weaving in a Template T which increases in size (Section 3.3.2). Starting from a base template (T = 1), we performed incremental weaving (W-i), iteratively adding one additional TML script to a template \( T_i = W(T_{i-1}, S) \) with the following profile: \( A = 1; R = 3; E = 3; S = 1 \). More precisely, each script S is composed of one action A which injects three new resources in the template and three new edges.

We can observe that the standard deviation from average is very low and linear w.r.t. the numbers of iterations. In other words, this benchmark suggests that incrementally weaving new scripts is interesting even in presence of bigger "previously-weaved" templates. This enables the possibility to weave new resources as reaction to specific events (e.g.; intrusion detection) to an already deployed multi-cloud infrastructure.

**Relative Overhead of Weaving in Workflow** To assess performance, we analyzed the impact of the weaving phase in the overall MANTUS workflow, to validate the performance of AOP-weaved approach. Therefore, we can consider that the weaving operation remains acceptable even with bigger templates. In Figure 7.13, one can find the compositional MANTUS efficiency with a logarithmic scale tested against the 49 regions of AWS, Azure and GCE. We performed a full deployment workflow, weaving 10 independent TML scripts of size \( (A = 1, E = 3, R = 3, S = 10) \), for an increasing number of starting overclouds nodes \( (T \rightarrow [0 - 49]) \). The TML script simply injects one resource per compute node. We modeled the filter section of the scripts in order to have conditions which are always verified in the TOSCA graph. Therefore, 3 resources R and 3 edges are weaved per-script iteration. We observe that the largest part of the execution time is spent doing the matching & weaving step. From the ILP program perspective (Section 4.3.1), its cost is calculated in the presence of two sets of diversity constraints (Continent, Provider). The execution time is linear w.r.t the set \( D \) of eligible data-centers. The translation step (Section 4.3.3) has also an important cost. However, this latter is proportional to the number of selected data-centers (i.e. DC).
Figure 7.12: This graph shows the execution time of weaving when a new script \( S \) is iteratively executed on a Template \( T \). We can observe that the execution time of weaving step scales linearly with the number of scripts weaved iteratively on the template.

Figure 7.13: We present the execution time of \textsc{MANTUS} with an increasing number of candidate data-centers and templates, to compare the weaving overhead with other steps of \textsc{MANTUS}. Results are shown in (log scale): for all phases, the execution time is linear (logarithmic in log scale) w.r.t. the graph size and the weaving phase overhead being one order of magnitude less expensive in time than matching.
The overall MANTUS workflow execution on all 49 data-centers composing $D$ takes 37 seconds to output the best feasible solution, according to user-requirements. This execution time is compatible with the time that the deployment takes on public CSPs which is decoupled from MANTUS. The cost of the matching&weaving step can be reduced by decreasing the initial number of candidates with some approximation filters. Those filters may guess the affinity of a certain candidate to user-constraints executing the ILP optimization on a subset of $D$ eligible data-center, as we described in Section 7.4.

7.3.1.2 Scalability

**Compositional weaving efficiency** After analyzing the behavior of MANTUS w.r.t. the number of TOSCA resources, we analyzed scalability in terms of TML script complexity. A first complexity metric is the number of injected resources ($R$) and relationships ($E$) in the action section (resp. vertices and edges of the graph). We thus evaluated weaver efficiency w.r.t. the number of actions ($A \rightarrow [1−100]$) in weaved script, adding different amounts of resource nodes ($A \ast R$ at each iteration) and edges per script ($A \ast E$). The number of TML actions inside each scripts represent a modifications on the graph, which add/remove resources.

In addition, complexity may also be captured by increasing the number of rules in the filters and checks sections ($FilterRules curve$, $F \ast A$). The benchmarks were performed with a single template and with a single script of increasing complexity $T = 1, S = 1$. Each curve is characterized by a different configuration of scripts $S$. We considered scripts which inject a different number of resources $R$, edges $E$ or filters $F$, in order to observe how execution time can be influenced by an increasing complexity of each part of the TML script. As shown in Figure 7.14, weaving time increases but remains linear with the amount of actions ($A$) and resources injected, which gives a good indication of scalability avoiding combinatorial explosion. Such results, combined with the ones obtained in Figure 7.12 are interesting when dealing with dynamic multi-cloud reconfiguration as shown in Section 5.2.3, where scripts should be added as quick as possible to an existing graph. Those results indicate that look-up operations in graph scale well and should encourage TML scripts developers to add extra-rules condition (e.g.; to verify semantic properties after the weaving process).

7.3.1.3 Dynamicity

**Weaver efficiency vs. Aspect complexity** In Figure 7.23, to identify a valuable strategy to conceive and develop TML scripts, we compared weaving overheads of multiple weaving ($W(T,S \ast A)$ with $S[R; E; F]$) injecting a single resource and weaving overhead a single script injecting multiple resources($W(T, A)$ with $S[R \ast A, A \ast E, F]$), according to the number of resources. Due to the cost of YAML parsing and graph reconstruction, the single-script approach shows more efficiency compared to using multiple scripts. This trend may lead to an interesting direction for further works. Scripts should be "bulked" as much as possible in single weaving operation in order to keep the elapsed time required for the weaving phase as low as possible. However, as we discussed in Section 3.3.2, this gives more and more importance to conflict detection among actions of different resources. In a nutshell, introducing in MANTUS the capacity of detecting and avoiding conflicts between TML scripts opens the door to weaving performance optimizations, resulting in script size increase, but reducing the number to be weaved.
Figure 7.14: This graph represents the behavior in terms of execution time w.r.t. a template which is weaved with script with increasing complexity. We can observe that the execution time of weaving step remains linear with the "complexity" of scripts weaved on a base Template. The "red" curve which increases the number of filters have a flat slope, compared to steeper one of those which make the number of weaved resources increase linearly. This suggests that the graph analysis phase is less expensive than resource/edge injection in the graph.
Figure 7.15: The slope of multiple script curve approach shows a steeper slope of single script, implying that the cost needed to analyze the graph and inject resources is fixed and does not depend from the number of resources that have to be weaved to the TOSCA Template.

7.3.2 Conclusion

In this section, we evaluated Performance, Scalability and dynamicity of MANTUS 1.0, which encapsulates the first contribution of this paper. The prototype scales linearly with the increase of different inputs given to MANTUS i.e. the size of TOSCA template, the number and the complexity of TML scripts). Furthermore, AOP-weaving does not add a significant overhead to the MANTUS workflow and can be integrated with the other steps of the pipeline without major overheads (i.e. one order of magnitude faster than TOSCA matching).

7.4 MANTUS 2.0

In this section, we evaluate a second prototype of MANTUS, which focuses on the second contribution of this manuscript, the context-based matching. After having tested the performance of weaving and its capacity to scale according to the size and number of TML scripts, we focused on inserting and benchmarking the context-based matching in MANTUS, as presented in Chapter 3. For those benchmarks, we enhanced the initial MANTUS prototype presented in Section 7.3, in order to (1) better integrate context-based matching and aspect-oriented weaving and (2) systematically construct the CSP context and (3) have an architecture suitable for a consumer.
From the weaver perspective, we kept the MANTUS TML weaver and matcher in Python. Notably, we used the TOSCA-parser\(^3\) library to manipulate TOSCA ServiceTemplates for fully compliant TOSCA generation as we described in the previous section.

From the context-based matcher perspective, as we presented in Section 4.2.3.2, we extended the TOSCA Standard definitions by introducing an always available context property. Leveraging the TOSCA object-oriented inheritance, the introduction of this property is propagated to each TOSCA NodeType. To match context-based matching requirements, we rewrote several parts of the TOSCA-parser library in order to better adapt this library to MANTUS. First, we reduced at most I/O operations replacing them with in-memory operations in order to optimize the pipeline of weaving and matching. Second, we introduced an ID to identify different TOSCA NodeTypes instances in a template, which was useful w.r.t. the iterative algorithm. Third, we introduced some class-wide memory caching, avoiding to repeat the same I/O operations across different MANTUS "threads".

To combine Context-based matching and Aspect-oriented weaving, as discussed in Section 4.3.2, we implemented two different prototypes of possible workflows, which are displayed in Figure 1.3. We implemented one "iterative" Matcher-Weaver, which reapplies the selected aspects. This entails the entire re-weaving of the template at each iteration, or in other words, the graph obtained after a "matching" step is tested by AOP to look if it meets TML filters property, by adopting the hybrid approach presented in Section 4.3.2. The other viable approach for building the matcher-weaver component, inspired by the "Generate-Than-Weave" approach, performed the AOP only on "concrete" TOSCA resources after having weaved them correctly.

When the weaving phase is finished, it is still necessary to select the best CSP according to consumer-parameters. We kept the ILP program to select providers and implementations leveraging the Pulp [93] LP modeler, which did show acceptable performances.

From the translation perspective, we developed a driver-based small TOSCA translator supporting OpenStack HOT, AWS CloudFormation and Microsoft Azure Resource Manager back-ends. The translator was developed from scratch to keep it minimal and to put emphasis on polymorphism for multi-CSP support. We also introduced a support for the U-Cloud Node, allowing TOSCA templates to be translated to Genode XML configuration files.

Finally, MANTUS leverages a "deployer" component which has an architecture similar to the "translator" where platform-independent commands (e.g. create deployment) are mapped to platform-dependent implementation APIs (e.g. OpenStack Heat, Amazon CloudFormation). In particular, the two possible design choices covers the usage of CSP-specific IaC languages (e.g. CloudFormation, HEAT, ARM) or to manipulate resources through traditional APIs as performed by Terraform [57].

Aside the core MANTUS compiler, we developed several auxiliary components, which helped to build provider-context or to provide the service to a consumer as presented in Figure 7.16.

---

\(^3\) https://github.com/openstack/tosca-parser
discovery APIs to easily inspect, for each different region, the available services or other parameters (e.g. VM flavours) in order to construct the context.

**RESULTING ARCHITECTURE** We developed a MANTUS prototype composed of several services. The MANTUS builder is separated in five main components, each of them with a specific task and the core MANTUS compiler:

- **Template Registry** is responsible to store and trace Template files.

- **Aspects Registry** is responsible to store the TML scripts that have to be weaved to TOSCA, according to user selections.

- **Provider-Context Builder** builds the "candidate" data-centers context, leveraging APIs of each different CSPs.

- **MANTUS compiler** is a service which implements the MANTUS workflow. Upon a trigger request coming from the front-end, it retrieves the complementary information from auxiliary services and then process the request until the deployment leveraging the workflow presented in Section 4.3.

- **Front-end Portal** provides the entry-point to the MANTUS multi-cloud builder. It consumes micro-services, providing to the consumer an unified view of its multi-cloud.

- **MANTUS CSP-context Builder** explores CSP API and builds CSP-contexts as specified in Section 4.2, regrouping services implementing the same tasks as equivalent from the point of view of context dictionary entry.
Upon reception of a new deployment request, MANTUS compiler service retrieves all complementary information from auxiliary services and then perform the workflow manipulation until the deployment.

7.4.1 Context-matching evaluation

In Section 7.3.1, we focused in particular on aspect-oriented weaving and we considered completely separated the steps of context-based matching and Aspect-Oriented. However, as we discuss in the next section, the combination of context-based matching and aspect-oriented weaving may represent a key design choice in the MANTUS multi-cloud construction process.

In this section, we evaluate the benefits coming from the adoption of MANTUS for multi-cloud deployment. We construct the whole MANTUS broker context creation and how a multi-cloud generation benefits from it. To do so, we executed several benchmarks on two common public clouds (Amazon EC2 and Microsoft Azure) and, locally, MANTUS was executed on a Intel Xeon E5-2650 Haswell at 2.60GHz with 64 GB of RAM with Ubuntu 16.04 LTS with a SSD Samsung EVO 740. As explanatory example, we decided to use the MANTUS context-based matching optimization to select the most-suitable VM flavor (named also type on certain providers) for a consumer generic workload.

We recall that public CSPs normally provide workload-specific instances, which can be used to boost the performances of a certain task. This comes at the price of an extra-cost for an optimized VM flavor compared to the cost of the general-purpose flavors. Those flavors offer a performance boost to applications, but requires to be tested to ensure the performance boost. Moreover, when dealing with multiple providers, the consumer has more work to do in order to construct the infrastructure on a different ecosystem to leverage the same optimization.

MANTUS provides a mechanism to abstract the instantiation of those VMs across multiple providers through its context-based matching optimization and through the consequent TOSCA to CSP-specific language translation. We developed several TOSCA implementations of a sample compute node. More precisely, we designed each of them differently according to the features demanded, which are listed in substitution_mappings TOSCA section (Section 4.2.3.2). These features distinguish each implementation, by making them suitable only for provider regions that offer those features.

As described in Section 4.1.1, the consumer inputs specify a list of different “workloads” and a list of optimization criteria. The consumer context is a guideline for the selection of the most suitable implementation through the context-matching optimization.

We tested the capacity of MANTUS to select a compelling implementation by executing common workloads which should benefit from a precise VM flavor. More precisely, we leveraged the Phoronix Test Suite [89] to evaluate the performance of different workloads (CPU-oriented, Disk-oriented, Memory-oriented) on different VM types with similar sizes. We adopted them to qualify different patterns of CPU/Disk/Memory resource utilization. We used instances on AWS and Azure disposing of 2 VCPUs and 8 GB of memory presenting different optimizations.

First of all, we constructed the MANTUS broker context. We shows in Figures 7.17-7.19 the data collected for a simple “learning phase” of MANTUS broker context construction. Considering price and performance, we compiled the MANTUS broker context. For each
flavor, we displayed the gain/loss in performance w.r.t. the general-purpose flavor and the average cost of each instance type on AWS/Azure DCs. From our results, we observe that compute-optimized flavors perform well on CPU-intensive and memory-intensive workloads, but they perform badly for asynchronous-IO disk-oriented workloads. Surprisingly, memory-optimized micro-benchmarks did not perform so well w.r.t. memory intensive VM flavors.

The results of those phases were that compute-optimized flavors are really suitable for all workload types but disk-intensive oriented. Due to the small difference in price compared to the general purpose flavors, it is an interesting candidate at least for CPU-intensive and memory-intensive workloads. Storage optimized flavors perform very well in all different classes of workloads, but its price is considerably higher than general-purpose flavors. Memory-oriented flavors did not show any remarkable performance.

Once the learning phase is over, we leveraged the broker context to evaluate MANTUS optimization benefits. More precisely, we developed a sample infrastructure leveraging the Mesos framework (Section 5.1.1. Adopting the Mesos two-layered scheduling mechanism, it is possible to notice the Master about a certain number of “tags” (e.g. the instance type of a certain slave). We executed three “applicative” benchmarks to evaluate the MANTUS optimization impact: the first is the CPU-intensive benchmark, the second is the data-intensive PostgreSQL-based pgbench, and the third is the memory intensive Hadoop sample Map/Reduce workload. In these experiments, we considered two optimization criteria: performance and performance/cost ratio.

In Figure A.4, we present MANTUS context-based evaluation comparing randomly selected VM flavors, general-purpose flavors, and the optimized selections made through context-matching. The consumer context specifies that the deployed workloads are CPU-intensive, Disk-intensive and Memory-intensive workloads in the same portion. The best-performing TOSCA implementation matched by context-matching was one tem-
Figure 7.18: Compute intensive micro-benchmarks performed to build the MANTUS “broker dictionary”

Figure 7.19: We present here the execution of micro-benchmarks focused on disk intensive utilization to build the MANTUS “broker dictionary”. Those benchmarks executed sequential and not-sequential access to disk in order to assess VM flavors throughput.
Figure 7.20: We present MANTUS context-based evaluation comparing randomly selected VM flavors, general-purpose flavors, and the optimized selections made through context-matching. We observe that selection done through context-based performs better than the baseline (i.e. between 5% and 10%).

The performance criteria context-matching selection shows an average improvement of 5-6% when using the random selection and 10-20% when using the general purpose. The performance/cost criteria context-matching selection results target only TOSCA templates which instantiate a single VM to be shared by all different workloads. Due to the cost/performance analysis we injected in the broker context, MANTUS picked the "compute optimized" flavors that provides the above discussed improvement for CPU-intensive and memory-intensive workloads, while introducing an overhead for storage-intensive workload.

It is worth noticing that results obtained during this learning phase show a huge variability depending on the provider region selected, on the day and the time. Due to the difficulty to predict the better performing implementation and the wide set of influencing parameters, a more sophisticated logic based on prediction can be proposed by a broker as added-value for resource optimization. A more sophisticated broker context building logic may be plugged on MANTUS without any modification of the code-base. This approach was proposed in [101] in a similar manner for a single provider.

7.4.1.1 Micro-benchmarks

In this section, we show our results concerning the evaluation of MANTUS different steps. Similarly to what we presented for MANTUS 1.0 in Section 7.3.1, we evaluate the MANTUS workflow execution time to assess its effective usability in a day-by-day multi-cloud
construction. To do so, we firstly evaluated the (b) overhead introduced by context-matching, (c) the optimization solver problem performance, and (d) the performance of different implementations of matching & weaving as discussed in Section 4.3.2.

In Figure 7.21, we observe the cost of mere "context-based matching" (Section 4.2) compared to a generic matching, which simply randomly chooses an acceptable candidate. We performed the test with two TOSCA templates: one simply representing a VM connected to the Internet and another implementing the ORBITS Openstack-based architecture [86]. Despite the slightly more higher execution time than the random matching, the context-based approach does not impose any important further overhead to MANTUS execution. This is still valid for the more complex Openstack-based template and confirms the analysis of complexity we did in Section 4.2.3, where we considered that the introduction of context does not influence the overall complexity.

Furthermore, we present in Figure 7.22 the execution of CSP selection step in the presence of an increasing number of constraints. More precisely, we tested its behavior w.r.t multiple diversity constraint generation. As presented in Section 4.3.1, each logic “diversity constraints” may result in a quadratic number of inequalities added to the set of problem constraints. Moreover, the number of inequalities introduced is proportional to the number of eligible candidates, \( D \). In those benchmarks, given the size of the program we described so far, the hypothesis that we want to validate is that the problem remains resolvable in an acceptable amount of time. We present the required execution time in three different scenarios. First, we evaluate the behavior without diversity constraints, second we analyze an execution when the consumer provides one "provider diversity" constraint, and third we introduce a further diversity constraints, adding to provider
c) Mantus ILP performance

Figure 7.22: In this graph, we show the elapsed time of context-based provider selection. We can observe that each set of constraints increases the execution time but the latter scales linearly with the number of constraint sets.

diversity also a further "geographical diversity" constraint. Clearly each set of constraints increases the execution time but the latter scales linearly with the number of set of constraints and seems to be feasible with a number of DCs around 50 if compared with the elapsed time of other steps of MANTUS pipeline. We also have to consider that each CSP increases every year the number of proposed "data-center" but we do not envision that this number will overcome 200, even considering new supported providers. To sum up, we think that the elapsed time to solve the ILP model is completely acceptable.

Finally, in Figure 7.23 we compare the different performances of two matching & weaving implementations in MANTUS, as discussed in Section 4.3.2. The two implementations implement two different approaches. First, we present the hybrid-based approach, which enforces AOP-weaving at each iteration of the matching process. Second, we test the "Generate-then-weave", which only performs weaving at the end of the context matching process. The second implementation is not complete since to really deal with AOP-weaving it would require a context-based matcher also for Aspect-Oriented injection. In other words, non-functional features modeled as TOSCA and TML scripts should be concretely matched to obtain TOSCA code leveraging the same NodeTypes which are used by the "matched" TOSCA code of the rest of infrastructure (e.g. ORBITS services). Whilst this latter implementation is not implementing a complete "Generate-then-weave" algorithm, it performs better than the hybrid one (2x). This overhead is mainly due to the TOSCA reconstruction performed at the end of each iteration. Therefore, this huge overhead suggests us to complete the "Generate-then-Weave" implementation with TML matching even if it requires an important development effort.
Figure 7.23: In this figure, we compare the different performance of two matching & weaving implementations in MANTUS. The two implementations compares an implementation of the hybrid approach and the "Generate-then-weave". For each implementation, we tested the case where there are (e.e. the red and the violet curve) or not TML scripts to weave at each iteration. We may observe that Generate-than-weave obtains the TOSCA matched template faster than hybrid approach (2x).
7.4.2 Conclusion

In this section, we showed several MANTUS benchmarks to assess the added value of context-based matching and its feasibility. We showed a simple example of how the "broker-context" inside MANTUS can be built and leveraged to make an infrastructure that can adapt to consumer requirements. Furthermore, we benchmarked the steps introduced in MANTUS 2.0 to show their acceptability in terms of performance. We compared performance of Context-based matching with a traditional Plug-in TOSCA matching and showed the feasibility of ILP model to solve. In addition, we analyze the comparison between two different implementation of MANTUS context-based matcher and AOP-based weaver. This benchmark showed us the most promising approach to improve MANTUS performance while conciliating context-based matching and aspect-oriented weaving.

The key take-away message that we can obtain from those benchmarks concern the construction of the broker context. Due to the variability of CSP performance, the variations of prices for certain resource and the number of factors that can impact on those latter (e.g. date, time), the 'broker context' should provide suggestion to evaluate different implementations with a more dynamic approach that the one implemented in MANTUS 2.0. As we proposed in Section 4.2, such knowledge may be provided at MANTUS "compile time" by an external API which may be proposed by a third-party service as a broker. In this case, the role of the broker may become more acceptable to final customers. Through MANTUS, they do not introduce any mandatory commercial dependency with the broker, but may flexibly leverage the broker knowledge to take better decision when instantiating or reconfiguring their multi-cloud infrastructure.

7.5 Conclusion

In this chapter, we evaluated the three main contributions of this thesis by implementing and benchmarking prototypes of each of them.

First, we showed the feasibility of the architecture we proposed to tackle the loss of control of the currently deployment methods of multi-cloud infrastructures (Chapter 5). More precisely, we benchmarked ORBITS, an overlay infrastructure as common layer for the multi-cloud, and a user-controlled virtualization stack that we named U-Cloud node, where the consumer has an improved control over system resources. The benchmarks shows that ORBITS implementation can be effectively leveraged by multi-cloud applications with acceptable overhead, in particular adopting the configuration using the LXC virtualization. Furthermore, we functionally validated the U-Cloud architecture performing migration and reconfiguration of the entire stack transparently from the perspective of the consumer. However, we encountered important performance problems which showed the careful tuning that should be done to obtain acceptable performance.

Second, we validated the "AOP-Weaving" contribution which address the lack of extensibility of the code, as shown in Chapter 3. More precisely, we defined a new language, called TML, to inject non-functional features into infrastructure templates. We presented a first MANTUS prototype which uses the TML as the language to weave the extensions of TOSCA templates. More precisely, this was evaluated by testing the performance, scalability and dynamicity of the implementation. Results showed that the MANTUS implementation was capable to scale linearly with the amount of different data-center
candidates, TOSCA infrastructure template size, TML scripts size and complexity. We showed also that the weaving step does not introduce a considerable extra time during the construction/reconfiguration of TOSCA infrastructure since matching is considerably longer (10x). To sum up, the introduction of extensibility fosters code reutilisation and simplify the introduction of non-functional requirements and the handle the lifecycle events of the multi-cloud. To the best of our knowledge, there is no IaC framework which proposes a similar feature for single or multi-cloud.

Third, we presented an extended MANTUS prototype which tackled the lack of specialization of state-of-the-art IaC matching mechanisms when integrating the different CSP resources into the multi-cloud architecture. The prototype did so by introducing the "context-based matching" mechanism to the existing TOSCA matching process (Chapter 4). We presented an example of the MANTUS added-value while building an infrastructure in terms of performance. Such added-value may be enhanced by a dynamic construction of broker-context which can provide more accurate inputs during the context-based matching steps. In addition, we showed how the context-based matching scales linearly w.r.t. the number of candidate data-centers and the number of TOSCA resources. Compared to other popular solutions, such as Terraform[57], the specialization feature proper of MANTUS enhances the TOSCA capacity to create provider-agnostic infrastructures by introducing the possibility to specialize them according to the context. Finally, the MANTUS design solves the contraposition between the customer-centric and the broker-based multi-cloud models. In fact, the context-based matching is compatible with the presence or not of a cloud broker party, leaving up to the consumers the choice to adopt their use model of preference.
CONCLUSION AND FUTURE WORK

8.1 MAIN CONTRIBUTIONS

In this thesis, we presented three major contributions which addresses respectively the problem of extensibility, specialization and lack of control on multi-cloud platform. For each of them we implemented and evaluated a prototype to assess acceptability of our approaches.

First, we designed and implemented MANTUS. MANTUS objectives are to: (1) customize the cloud template according to tenant requested security services and features offered by the underlying CSPs; and (2) select a subset of cloud providers, compatible with policies expressed by the tenant needs.

MANTUS leverages two contributions introduced in this thesis. First, extensibility, leveraging AOP, we introduced the possibility to inject/extract extra resources to a base IaC template taking TOSCA as a target IaC framework. At the cost of an acceptable overhead, aspect-oriented weaving separates cross-cutting concerns. To do so, we defined a language, TML, to formalize TOSCA manipulations and how those have to be performed. We validated the suitability of the aspect-oriented approach for non-functional service injection evaluating scalability, dynamicity and performance. This fosters code reuse, since non-functional services can be dynamically added to different base infrastructure without modifying the "functional" code-base. As a result, this enables the possibility to multiplex the same portion of code among different customers.

Second, the context-based matching enables the possibility to reify an abstract TOSCA-based generic resources in an optimized concrete implementation, taking into account consumer parameters and CSP data-center configuration. We provided a formal definition of context-based matching and we showed its adoption in several steps of multi-cloud construction. We extended the traditional TOSCA matching principle to deal with CSP and consumer context. The result enables the possibility to optimize the "compilation" of a generic template, obtaining an optimized result of an abstract template.

The third contribution, presented in Chapter 5, tackles the loss of control of the currently proposed methods for cloud and multi-cloud infrastructure deployments. More precisely, the U-Cloud architecture introduces an user-defined virtualization architecture. Adopting an hybrid approach among micro-hypervisors and nested virtualization, the U-Cloud architecture presents a secure but legacy-compatible stack, which separates services deployed by the user and the ones under the provider responsibility.
To concretely instantiate the multi-cloud, we implemented the overlay infrastructure ORBITS. ORBITS is an overlay architecture that provides a common layer above multiple providers to fulfill the interoperability issues among different CSPs systems. We evaluated the acceptability of performance overhead of ORBITS and U-cloud node compared to traditional overlay and virtualization architectures, respectively.

Leveraging the U-Cloud and ORBITS building blocks, MANTUS defines and deploys a well-adapted version of ORBITS on a selected group of providers. Furthermore, MANTUS is capable to make the deployed multi-cloud evolve according to a mutated context (e.g., unexpected event occurred) leveraging autonomic reconciliations loops. Since CSP and consumer context embeds completely what is required to treat the occurred mutation, MANTUS pipeline can be triggered with low or any manual intervention in order to react fast.

8.2 LIMITATIONS AND FUTURE WORK

Future directions of work will focus on three main directions, addressing three major limitations existing today.

8.2.1 Container as a Service-based multi-cloud

We can observe a definitive consideration of the importance to break vendor lock-in by consumer. This has been effectively obtained by the adoption of containerization and the arise of open-source container orchestration frameworks, as Kubernetes and Swarm. Such frameworks are normally open-source project which are designed to run on multiple CSPs. The huge popularity of this frameworks pushed any huge provider or IaaS cloud manager to support them, in order to integrate their services (e.g. Load Balancer as a Service, Identity Management) inside those container orchestration frameworks. In addition, CSPs have started to propose to their customer the possibility to rely on a completely managed instance of those frameworks. Furthermore, several container orchestrator, such as Kubernetes, are based on IaC and integrates a state-based logic which simplifies as we described in Chapter 2 deployment and operations.

This trend introduces a de facto provider-centric multi-cloud since providers are finally proposing services which implements exactly the same abstractions and API. Moreover, it breaks the existing gap between client-centric and provider-centric analyzed in many works [2, 42, 55] since any consumer can deploy the same framework on its premises or on any IaaS service.

However, container as a services solutions still struggles when dealing with multiple CSPs. First, a single instance of such orchestrator cannot span geographic networks, requiring low-latency connection to correctly work. Second, many orchestrator federation across different geographic regions pose severe challenges in terms of deployment, configuration and reliability, which have seen many attempts but remains still not consolidated at scale so far [kubernetes:federation].

Future work will then investigate those trends, analyzing in depth if the presented model can completely completely satisfy the multi-cloud requirements presented in Section 1.1.

MANTUS approach still remains interesting in a similar scenario. This is due to mainly three reasons: (1) MANTUS relies on TOSCA which can easily deal with equivalent im-
implementations of the same service (managed, un-managed); (2) MANTUS extensibility feature can be easily extended to those orchestrators due to their utilization of IaC. For instance, Kubernetes resources may easily translate in TOSCA resources making MANTUS able to enforce desired condition with them through its extensibility properties; (3) MANTUS is capable to guarantee that the same non-functional properties (e.g. security) are enforced on instances on many providers through extensibility. Such enforcement remains interesting for consumers, like CDOs, which have specific constraints in term of infrastructure they are deploying w.r.t. security and integration with their existing services.

8.2.2 Specialization and predictions

In this work, we present several benchmarks of MANTUS pipeline and how they can be used to build the MANTUS context. This has two major limitations: (1) the construction was done manually and (2) the scope was limited to IaaS resources and not considered managed services (e.g. DBMSaaS, serverless computing).

Related works, showed that in-depth analysis of cloud benchmark may lead to accurate performance predictions [101]. However, those works presented analysis limited so far to a single provider. Leveraging MANTUS multi-cloud capabilities, we will investigate the possibility of leveraging such techniques on multiple providers on a wider scope than traditional IaaS computing. In fact, the work presented in this manuscript did not investigated in depth the potentials and possible difficulties of integrating the multi-cloud pipeline paradigm such a "Serverless computing". Those managed services are challenging to be adopted since they are not part of the common denominator as IaaS resources, but they promises important benefit in terms of application performance, scalability and costs. Part of the challenge is represented by the serious impact that those can have on portability of code, introducing technological CSP lock-ins.

An important axe of future work regarding MANTUS would essentially concern how integrate managed services into the MANTUS context-matching in order to select the most adapted implementation. In particular, MANTUS has to be able to: (1) select the right IaaS flavor, also considering managed services which can be provisioned through IaC; (2) integrate in the consumer context the grade of desired portability, in order to construct an adapted multi-cloud w.r.t. the escape of vendor lock-ins.

8.2.3 Non-functional Service injections improvements

As we presented in the evaluation part, incremental deployments may be used to intersynchronization between overclouds instantiated over different data-centers. We would extend this capacity to introduce fast autonomic multi-cloud reconfiguration. To do so, we will explore the possibility to rely on some idempotence property when dealing with extensibility. Moreover, MANTUS should integrate one or more techniques capable of detecting/solving conflicts when weaving multiple scripts used by AOP-based weaving. For instance, we would evaluate the complexity of the filter rules language, elaborating more complex rules and evaluating the performance w.r.t. the rule complexity and not only their number.

This axe of work is particularly interesting due the potential benefit about optimization, as we presented in the result of 7.23 in chapter 7, where the weaver showed better
performance with few huge scripts compared to a multitude of tiny scripts. Therefore, future work will interest also the identification of a method to reduce in a single-script non conflicting multiple-scripts would speed up significantly the weaving process.


[15] *AWS has made VMware the Airbnb of the cloud*. URL: http://www.theregister.co.uk/2016/10/14/vmware_aws_analysis/ (cit. on pp. 9, 32).


[17] P. Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. “Xen and the Art of Virtualization”. In: *SOSP’03* (cit. on p. 178).


[41] Docker. URL: https://www.docker.com/ (cit. on pp. 25, 38, 118).


[51] FP-7 Unify. URL: https://www.fp7-unify.eu/ (cit. on p. 29).


[71] Libcloud. URL: https://libcloud.apache.org/ (cit. on pp. 34, 42, 44).


[93] *PuLP* (cit. on p. 135).


[100] Salt. URL: https://saltstack.com/ (cit. on p. 38).


Appendices
A.1 INTRODUCTION

Le paradigme « Cloud Computing » représente un des plus importantes tendances in ICT des derniers dix ans. Le modèle de service offert par le cloud computing, basé sur une allocation des ressources élastique et à-la-demande et une facturation ponctuelle, est capable pas seulement de satisfaire les demandes en termes de ressources mais aussi de pouvoir relâcher ces dernières à tous les moments.

Plusieurs domaines applicatifs sont en train de migrer au cloud computing. Un exemple important est représenté par le Network Function Virtualization (NFV), une nouvelle approche pour le déploiement de fonctions réseau capable de bénéficier par les avantages en termes d’élasticité et passage à l’échelle du cloud. Un autre exemple très significatif est représenté par le Big Data, qui exploite le passage à l’échelle du cloud pour faire de l’analyse de données en grand volumétrie.

Cependant, il restent encore plusieurs applications qui pourraient obtenir des avantages par l’adoption du cloud computing. Ces applications sont soumises à une législation assez stricte qui peuvent être satisfaite seulement en présence de plusieurs fournisseurs de cloud computing (CSPs).

Par exemple, nous considérons un cas d’usage classique d’extension de cloud (cloud bursting) présenté dans la figure A.1, concernant une organisation qui fournit de prestations de services de santé (CDO). L’utilisation de plusieurs fournisseurs de cloud public pour étendre le cloud privé de l’organisations peut avoir plusieurs avantages sur les différents types de « workload » exécuté dans le cadre des activités du CDO. Ces dernières peuvent concerner des activités de télémédecine (remote healing) pour aller à l’analyse poussée de données médicales (EHR), à travers du data mining ou autre technologies. La sensibilité des données analysée impose aux CDO beaucoup d’efforts pour faire en sorte que les données en entrée restent confidentielles.

En conséquence, le traitement de ces données ne peut pas être délégué à une partie tierce omniscient autre que le patient et le médecin. Au contraire, la donnée peut être divisé entre plusieurs CSPs d’une telle façon qu’aucun des fournisseurs puisse retrouver les données originelles pendant il exécute les analyses. Cela peut être obtenu par la computation à plusieurs parties (Multi-party computation ou MPC) [123]. MPC est primitive cryptographique que rend possible une computation distribuées parmi plusieurs acteurs en gardant privée les inputs initiaux de chaque partie.

Dans ce scenario, nous pouvons identifier deux types d’acteurs. D’un côté, nous pouvons identifier les « end-users » comme bénéficiaires du cloud bursting aux quels l’infrastructure sur quelle reposent les applications et l’architecture des services cloud est complètement transparente. D’autre coté, nous pouvons aussi identifier les consommateurs de cloud, ou « cloud consumers », qui sont chargés de définir l’infrastructure du système d’information du CDO et qui doivent faire les choix technologiques et de fournisseurs concernant le cloud computing. Ces derniers peuvent être considérés comme des utilisateurs avertis, qui connaissent les bases du cloud computing, comme le tech-
L’utilisation au même temps sur plusieurs CSPs et data-centres est appelé multi-cloud (or intercloud) [2]. Les avantages dans l’utilisation de ce model rapporté au model avec un seul CSP ont été investigués et formalisés. Parmi ces derniers, nous pouvons retrouver notamment la distribution fine des ressources à travers plusieurs CSPs, qui peut améliorer la qualité de service (QoS) des applications, et aussi la Qualité de protections (QoP) pour toutes les applications ayant des contraintes strictes en terme de sécurité comme les CDOs.
De plus, la possibilité de s’appuyer sur plusieurs CSPs permet de casser le lien commercial de forte dépendance entre consommateurs et fournisseurs de services. Plusieurs CSPs peuvent avoir plusieurs différents data-centres dans le territoire d’un seul pays, simplifiant en conséquence les problèmes réglementaires en terme de positions de données mais sans imposer au client de devoir s’appuyer sur un seul Fournisseur de service.

Enfin, utiliser plusieurs CSPs introduit la possibilité d’optimiser les dépenses en comparant les prix des ressources cloud disponible sur les différents fournisseurs.

Cependant, les organisations (i.e. les développeurs et opérateurs des services IT) relèvent rarement le défi de construire des applications qui traversent plusieurs CSP [42]. En réalité, malgré les avantages mentionnés en précédence, le fait de s’appuyer sur plusieurs fournisseurs de cloud doit faire face à trois limitations majeures.

En premier lieu, les applications multi-CSP sont plus compliquées à être construites par rapport aux applications cloud traditionnels. En effet, à côté des pures applications, qui peuvent être appelées services « fonctionnels », les développeurs et les opérateurs de cloud doivent faire face aux services non fonctionnelles, qui s’occupent d’aspects auxiliaires dans l’application. Comme services non fonctionnels nous pouvons considérer le monitoring où la protection des applications. Dans le cadre d’une application multi-cloud, la gestion de ces services sur plusieurs CSP hétérogènes rajoute une complexité majeure.

En deuxième lieu, les architectures multi-cloud demandent un effort supplémentaire pour être adoptées. Les différents API et écosystèmes de services posent des difficultés techniques et de compétences assez importantes. De plus, ces dernières ne concernent pas que l’interopérabilité de services avec plusieurs différents fournisseurs mais aussi l'impossibilité d'obtenir un usage optimisé des ressources sur les différents CSPs. Cela signifie que le consommateur de cloud doit tout seul capable de sélectionner les CSPs et les services les plus adaptées à son cas d’usage pour construire ses applications.

En troisième lieu, le problème de la « perte de contrôle » observé dans le cadre d’un seul provider de cloud public, doit aussi être géré dans le cadre de plusieurs providers. De plus, le consommateur de cloud normalement souhaite réutiliser les mêmes outils qu’il utilise dans le cadre de son cloud privé pour protéger les applications end-user. En raison de contraintes techniques, cela n’est pas du tout simple à mettre en œuvre.

Pour être capable à dépasser ces limitations, dans la section suivante, nous allons analyser les différentes approches disponibles pour construire des applications multi-CSPs, en mettant en évidence technologies clefs et les outils les plus populaires.

A.2 MULTI-CLOUD SCENARIO

L’interopérabilité, dans le cadre du multi-cloud, est la capacité d’interagir de façon transparente avec plusieurs CSPs, en adoptant le même façonnage et sémantique pour provisionner ressources. Cela représente le premier obstacle qui empêche une interconnexion directe à travers différents CSPs. En particulier, public CSPs utilisent tous des technologies différentes et parfois incompatibles, accédées à travers API propriétaires.

Pour dépasser les limitations en termes d’interopérabilité, deux architectures ont été proposé jusqu’ici : provider-oriented architectures et client-oriented architectures.

Les architectures provider-oriented architectures permet à un groupe de CSP qui collabore de se « présenter » aux consommateurs de ressources cloud comme un seul CSP. Les
CSP peuvent obtenir cette configuration à travers la fédération des ressources, obtenue en s'appuyant sur des standards communs, en termes de ressource virtualisation et APIs. L'avantage principale de cette configuration est la complète transparence dans l'adoption de la part du cloud consumer, qui ne lui demande pas des efforts supplémentaires en termes de développement. Par contre, cette approche présente plusieurs difficultés dans son effective mise en œuvre [85]. D'abord, il existe une concurrence assez forte parmi les différents CSPs qui amène chacun à proposer des API et services différents et justifier sa position dans le marché. De Plus, le choix technologique faites par chaque CSP peut rendre très compliqué l'interopérabilité à travers deux différents providers [5].

La seconde approche, client-oriented architectures, introduit la possibilité pour les consommateurs de cloud d'allouer ressources à travers plusieurs providers sans devoir s'adapter à chaque technologie où devoir dépendre de la volonté du provider d'adhérier à une fédération. Les différentes architectures proposées dans le cadre de cette approche relient sur des outils spécifiques, bibliothèques multi-cloud (MCL) [49, 62], ou sur des parie tierces, comme des brokers [18, 102]. En particulier, MCL sont devenus assez populaires pour la capacité de permettre aux développeurs/operateurs de construire, maintenir leur multi-cloud en autonomie. A travers différentes techniques, ces outils sont capables de simplifier l'allocation de ressources et la configuration des services sur plusieurs CSPs.

Nous considérons que les « framework » MCL introduisent la possibilité pour le client de s'appuyer sur une infrastructure multi-cloud reproducible, bien qu'ils présentent encore des limitations importantes. La reproducibilité est garantie par le paradigme "Infrastructures as Code" qui modèle les ressources cloud à travers une description basé sur du code declaratif.

Le modèle d'infrastructure qu'il devrait être mis en place par ce « IaC-oriented MCLs » consiste dans la présentation d'une vue homogène des ressources sur les différents CSPs à travers une infrastructure virtuelle « overlay ». Proposée par plusieurs travaux [6, 48, 74, 96], l'approche overlay introduit une couche de services toujours applicable. Cette couche découpe les API des CSPs et le consommateur de cloud à travers une partiel ré-implémentation des services offerts par le CSP sous-jacent. En faisant cela, le multi-cloud résultant est capable d'être reproduit de façon équivalente sur plusieurs CSPs, en exposant aux développeurs toujours la même configuration.

Par exemple, cette infrastructure virtuelle overlay est capable de satisfaire les contraintes présenté dans le cas d'usage exposé en précédence. Développeurs et operateurs de la part des CDOs peuvent simplement reproduire les services disponibles dans leur cloud privé et après construire une infrastructure virtuelle « overlay » en introduisant les mêmes services, pour réduire au minimum les problèmes de compatibilité des applications.

Cependant, construire une infrastructure virtuelle de tel façon a un cout non négligeable à être assumé par les consommateurs de cloud. Comme mentionné en précédence, les infrastructures sont composées par services fonctionnel et non-fonctionnels. En particulier, ces derniers sont généralement les mêmes parmi plusieurs implémentations de l'infrastructure overlay. En effet, le consommateur de cloud voudrait réutiliser les mêmes services non fonctionnels à travers différents infrastructure virtuelles mais en réalité cela reste un pratique assez compliqué à mettre en œuvre. En effet, la séparation entre services fonctionnel et non fonctionnels est flou et, en conséquence, le code des services non-fonctionnelles est compliqué à être partagé.
De plus, le consommateur de cloud voudrait aussi réutiliser le code IaC, à travers différents CSP, en pouvant obtenir de cette définition générique une implémentation efficace qui utilise aux mieux les capacités des providers. En résumé, le consommateur de cloud ne veut pas adapter son code IaC pour supporter une nouvelle fonctionnalité pour un ou un autre CSP, n'ayant non plus à devoir changer le code si le groupe de providers choisi change. Les paradigme IaC-MCL utilisés aujourd'hui ne sont pas capable d'adresser en flexibilité ces points. En conséquence, le consommateur est de facto verrouillé dans le choix des CSPs, en étant bloqué non plus par les limitations d'un seul provider mais par un groupe de providers.

Pour résumer, nous considérons la combinaison de MCL basé sur IaC et infrastruc- ture virtuelle overlay une solution prometteuse pour le multi-cloud. Cependant, si les problèmes initiales d'interopérabilité sont maintenant adressée, les limitations en terme d'extensibilité, spécialisation de l'infrastructure et manque de control doivent encore être résolues. Ces problèmes sont analysés en détail dans les sections suivantes.

A.3 enoncé du problème

Cette section traite les trois limitations clés des architectures multi-cloud orienté client basé sur MCL : la manque d'extensibilité du IaC dans le cadre des déploiement multi-cloud ; Second, l'impossibilité d'une utilisation optimisée des ressources et services, quand le multi-cloud est initié à partir. Troisième, la perte de control dans le contexte de plusieurs providers.

A.3.1 Manque d'extensibilité

Overlay multi-cloud basé sur IaC, introduit en section 1.2, fournit une solution pour le problème mentionné en précédence de l'interopérabilité parmi différents CSPs. En effet, le code déclarative et agnostique au CSP, propre au « framework » IaC, simplifie beaucoup l'intégration et le déploiement de services fonctionnel et non-fonctionnel. Cependant, l'adoption de l'IaC ne représente pas encore une solution idéale pour la création d'une infrastructure virtuelle puisque cela représente une grande quantité de code à développer et maintenir. Plus précisément, le choix d'agrandir la base de code avec l'IaC introduit des couts non trivial en considérant le cas du multi-cloud, où chaque CSP représente environnement où développer, tester et déployer le code agnostique au CSP.

Comme il a été présenté en précédence, nous pouvons considérer que chaque consom- mateur doit reposer sur un groupe d’application ou service fonctionnels et des services non-fonctionnels, qui couvrent des services auxiliaires (ex.; monitoring, backup, debug). Le code de ces services non-fonctionnels devrait être réutilisés le plus possible entre plusieurs utilisateurs. Cela faciliterait aussi notre cas d’usage pour une traitement plus agile des données EHRs sur un stockage multi-cloud.

En effet, les services « non fonctionnels » sont normalement épars parmi plusieurs éléments d’infrastructure (ex. configuration d’OS, topologie réseau). Par exemple, en considérant le monitoring, service clef pour déterminer des mal fonctionnements, plusieurs couches d’agents peuvent être utilisés. Les agents peuvent collecter des données sur l’exécution à partir de l’intérieur de l’environnement d’exécution de l’application (ex. VM) ou à partir des couches d’infrastructure (ex. hyperviseur). Ces « bordures flous » entre le code applicatif et le code non fonctionnel ne respecte pas la composition de
services traditionnelle faite en « module » et « package », en demandant une intégration de composants plus sophistiquées. En conséquence, ce n'est pas possible réutiliser simplement le même code non-fonctionnel à travers différents applications. En considérant l'exemple du « framework » du monitoring [92, 125], cela peut nécessiter pas seulement de devoir déployer une unité centrale de collecte de métriques mais de distribuer des agents sur toutes les VMs pour exporter les mesures d'intérêt. Ainsi, en assumant que l'IaC soit utilisé, le code du monitoring est compliqué à être ajouté efficacement dans le code d'infrastructure.

Donc, dans cette scenario, toutes les combinaisons possibles de configurations des services non-fonctionnels doivent être testés. Si nous considérons le scenario des multi-cloud où il y a par défaut plusieurs environnements de production, le procès de test doit être effectué à la main sans pouvoir s'appuyer sur aucune économie d'échelle pour le consommateur.

En deuxième lieu, après le déploiement, les services doivent être maintenu et leur disponibilité monitorée. Mise à jour de l'infrastructure dans le cas d'un événement pas attendu sont très compliquées à être géré sans la présence d'un support à l'extensibilité. Les événements possibles couvrent des pannes interne à l'infrastructure pour arriver à des actions malveillantes détectés dans l'infrastructure ou une panne dans un des CSPs utilisés. Sans un support à l'extensibilité, une modification du code d'infrastructure à la main est nécessaire et cela limite la possibilité de répondre rapidement à un événement spécifique.

En conclusion, dans un multi-cloud, la base de code et l'effort demandé pour maintenir les applications et l'infrastructure applicative deviennent rapidement très élevés et cela prévient le consommateur de multi-cloud d'avoir le même niveau de flexibilité qu'il a dans l'écosystème d'un seul CSP.

A.3.2 Spécialisation

Les infrastructures multi-cloud impliquent aujourd'hui de renoncer aux fonctionnalités spécifiques dans l'allocation et la définition des ressources cloud. En effet, les approches les plus populaires, comme les bibliothèques multi-cloud (MCL) [49, 57, 62], référencent les ressources avec une définition agnostique aux CSPS. Cela signifie que les ressources sont déployées de la même façon générique sans tenir en compte les spécificités de chaque CSP. Plus précisément, les MCLs s'appuient sur un seul formalisme en utilisant une définition des ressources « à dénominateur commun minimal » [42]. Cela cause les deux limitations suivantes.

En premier lieu, la limitation directe concerne la spécialisation. Le consommateur de cloud ne peut pas exprimer ses besoins en termes de services ou ressources of a CSP et, par conséquence, il ne peut pas obtenir l'implémentation la plus adaptée pour une application donnée. Par exemple, certains consommateurs pourraient améliorer leur performance d'une accélération matérielle (ex. GPU, FPGA) ou en utilisant des services managés par le CSP (ex. DBMS as a service). En deuxième lieu, le consommateur de cloud ne peut pas construire son infrastructure applicative en facilité sur plusieurs CSPs vu qu'il nécessite pour faire ça une très grande connaissance des écosystèmes spécifiques de chaque CSP. En effet, les écosystèmes de chaque CSP sont en train de grandir en complexité et en nombre de services exposés. Tous les ans, les CSPs publics et les cloud managers open-source introduisent nouveaux services pour inter-
cepter nouveaux cases d'utilisations. De plus, les CSPs aujourd'hui sont en train de proposer plusieurs régions/data-centres où il est possible déployer les workloads des consommateurs. Ces data-centres sont construit en différentes années et ils ont donc une configuration matérielle qui peut varier largement. En conséquence, l'utilisateur doit connaitre chaque région spécifique de chaque provider pour correctement choisir ceux qui ont plus d'intérêt pour lui.

Donc, cela pousse une nécessité d’investissements dédiés pour pouvoir utiliser ce que chaque écosystème propose, en demandant formations spécifiques pour le consommateur. En effet, cette approche rend le multi-cloud pas soutenable pour la plupart des organisations de tailles moyennes et cela représente une barrière d’entrée très importante.

Pour résumer, construire une infrastructure multi-cloud à travers MCL demande un effort n’important pas seulement en termes de connaissance de CSP mais aussi, quand cela est dépassé, une difficulté d’optimisation des infrastructures par rapport aux besoin utilisateurs et les ressources offertes par les différents providers.

A.3.3 Manque de control

Malgré les architecture cloud client-centrique représentent un des approches plus intéressants pour le multi-cloud, les solutions basées sur architectures « overlay » d’infrastructure virtuelle présentent des problèmes pas résolus. Dans les paragraphes qui suivent, nous identifions et caractérisons ces obstacles.

En premier lieu, nous considérons le problème de « manque de control » pour le consommateur dans le contexte d’un cloud provider public, qui reste toujours valide dans le contexte d’un multi-cloud.

En effet, depuis le début de l’époque du cloud, l’adoption des CSP publics n’a pas que soulevé des problèmes des sécurité et control sur les données hébergées chez les providers mais elle a aussi représenté une perte de control sur l’infrastructure du côté des consommateurs Comparée avec les infrastructures privées, la perte de control ne concerne pas que la possession physique des data mais aussi la perte de control sur les couches basses de la pile logicielle, notamment la couche de virtualisation ou hyperviseur. Cette “perte de control” est une conséquence directe du partage des mêmes ressources matérielles par plusieurs consommateurs qui ne se font pas confiance. Dans un provider traditionnel, cette perte de control est le prix à payer pour pouvoir isoler les tenants en présence d’une couche de virtualisation monolithique.

Plus précisément, comme reporté par [121], la possibilité de contrôler la couche de virtualisation permet aux développeurs et aux opérateurs d’utiliser les mêmes mécanismes de protection (ex. introspection) pour protéger leurs applications.


Ces architectures n’avaient pas considéré la possibilité et le conséquence d’une connexion multi-provider, comme nous avons discuté précédemment en Section 1.2. Plus précisément, si nous considérons que le consommateur contrôle désormais les couches d’infrastructures, il ne peut pas dans un domaine multi-cloud (1) control de façon flex-
ible et le programmer à travers IaC de façon transparente et (2) préserver la compatibilité avec des outils et applications existants (ex. éviter de devoir redéfinir le template d'infrastructure pour s'adapter aux différentes architectures de virtualisation).

Pour résumer, en analysant les travaux existants, il reste encore à traiter comment le IaC peut résoudre les problèmes présentés dans les paragraphes précédents. Les nouvelles architectures modulaires peuvent casser la compatibilité avec les outils existants et, de plus, l'utilisateur ne peut pas englober dans sa définition d'infrastructure multi-cloud à travers l'IaC pour modeler une couche de virtualisation overlay agnostique aux providers, mais capable au même temps de bénéficier des spécificités de chaque CSP.

A.4 OBJECTIFS DES RECHERCHE ET CONTRIBUTIONS

Dans la section 1.3, nous avons souligné les trois limitations clefs observées sur les architectures orientés multi-cloud disponible aujourd'hui. Dans cette section, nous présentons les objectifs de recherche de ce manuscrit, qui ont comme objective la mitigation des limitations des approches client-centrique pour les architectures multi-cloud présenté en Section 1.3.

En un mot, les travaux présentés en ce manuscrit ont comme objective ultime d'améliorer la performance et la flexibilité du procès de construction des multi-clouds. Plus précisément, nous nous sommes concentrés sur la possibilité de construire un multi-cloud optimisé selon des paramètres utilisateur, capable à garantir un niveau un niveau de contrôle équivalent à une infrastructure privée et de passer à l'échelle en termes de cloud service providers supportés.

Malgré le fait que les "infrastructures virtuelles" [6, 48, 74, 96] donnent une première réponse au problème de l'interopérabilité, cette approche demande beaucoup d'effort en terme de lignes de codes pour être adopté et maintenu. De plus, cela passe avec beaucoup de difficulté à l'échelle.

En partant par cette description, le constructeur de multi-cloud devrait être capable de construire le multi-cloud le plus adapté possible au consommateur par rapport au choix du CSP et des implémentations des services que ce dernier veut déployer dans son multi-cloud. Dans cette scenario, le consommateur de cloud fournit comme input une description générique d'infrastructure qu'il veut mettre en place, des critères d'optimisation et une liste de services non-fonctionnels qui veut rajouter à son infrastructure.

Le multi-cloud nécessite donc de la réalisation d'un constructeur capable à étendre les capacités du paradigme IaC courant pour exprimer une définition de services agnostique et complètement découpé des CSP subjacentes. En jouant le rôle de "multi-cloud compiler", en se basant sur une approche "context-aware «, le constructeur doit concilier: (1) l'extensibilité non fonctionnelle, (2) la capacité de s'adapter aux fonctionnalités du provider et (3) niveau amélioré de control dans un scenario multi-cloud.

Dans les prochaines sections, nous détaillerons les trois objectifs de recherche de ce manuscrit et pour chaque objective, nous soulignerons les contributions apportées.

A.4.1 Extensibilité de l'infrastructure multi-cloud

Le premier objectif de ce manuscrit est l'introduction de l'extensibilité comme propriété dans la modélisation de service basé sur IaC. Comme discuté autrement en Section 1.3.1, le manque d'extensibilité prévient le framework IaC d'adresser complètement les chal-
lenges multi-cloud en termes de (1) intégration et déploiement de services non-fonctionnels et (2) modification à temps d'exécution des multi-cloud infrastructures.

L’extensibilité peut simplifier l’insertion et l’enlèvement des services non-fonctionnels. En effet, cela peut permettre la possibilité de réutilisation de code à travers des différents consommateurs (ex., leur différentes infrastructures multi-cloud mentionnées en Section 1.4.3). En effet, un mécanisme pour insérer dynamiquement les services non fonctionnels à l’intérieur du reste de l’infrastructure pourrait permettre d’éviter intervention ad hoc et configurations manuelles de ces dernières. Le consommateur pourrait sélectionner seulement les ajouts non-fonctionnels demandés pour "enrichir" son infrastructure de base.

Jusqu’ici, il n’y a pas, à notre connaissance, solutions proposées pour l’insertion et l’enlèvement des services non-fonctionnels in a scenario multi-cloud et quelques outils d’analyse statique des « template » ont été conçues [57].

La première contribution de ce manuscrit propose un mécanisme pour activer l’insertion/en-lève-ment de services non-fonctionnels in à multi-cloud scenario. Cela permet de réutiliser le code à travers différents consommateurs (ex. multi-cloud infrastructures différentes) et l’analyse statique des modèle ou « template » d’infrastructure. Nous proposons d’étendre le model basé sur IaC avec les concepts du paradigme de la programmation par aspect (AOP) [66] pour obtenir une définition extensible d’infrastructure virtuelle.

A.4.1.1 TML

Dans le cadre de cette contribution, nous avons développé un premier prototype d’un constructeur de multi-cloud appelé MANTUS. Ce dernier a été développé dans le but de (1) disposer d’un déploiement multi-cloud complet de bout en bout et (2) introduire une extensibilité dans le workflow TOSCA en utilisant le langage basé sur la programmation par aspects, Tosca Manipulation Language (TML). Les scripts TML permettent de modifier un graphe TOSCA, composé des ressources spécifiées dans les champs node_templates. Ce graphe est composé d’un ensemble de vertices V, capturant l’ensemble des ressources avec leurs attributs (par exemple, propriétés, interfaces avec artefacts annexes) et d’un ensemble E de lien, qui représentent l’ensemble des exigences pour l’interconnexion. À partir de l’analyse de ce graphe, TML peut injecter/expulser de manière flexible des services non fonctionnels dans/depuis un modèle TOSCA. Cela peut répondre à la demande de services d’insertion de nouveaux service, dans une étape spécifique du cycle de vie de l’infrastructure (ex. ajouter le monitoring pour le run).

A.4.1.2 Mantus 1.0

Du point de vue de l’implémentation, nous avons implémenté le tisseur d’aspect basé sur TML in Python. Nous avons également développé un traducteur TOSCA vers les langages OpenStack HOT et CloudFormation. Le traducteur a été développé à partir de zéro pour rester minimal et mettre l’accent sur le polymorphisme pour la prise en charge multi-CSP.

Pour évaluer les performances, nous avons analysé l’impact de la phase de tissage dans le flux de travail global MANTUS, afin de valider les performances du prototype (présenté en figure A.2). En analysant les résultats, nous pouvons considérer que l’opération de tissage reste acceptable même avec des modèles plus grands.
Figure A.2: Le graph montre le temps d'exécution de MANTUS avec un nombre croissant de centres de données et de gabarits candidats, afin de comparer le temps système de tissage aux autres étapes de MANTUS. Les résultats sont affichés dans (échelle logarithmique): pour toutes les phases, le temps d'exécution est linéaire (logarithmique dans l'échelle du journal) w.r.t. la taille du graphe et le temps d'exécution supplémentaire de la phase de tissage (un ordre de grandeur moins onéreux en temps du matching).

Sur MANTUS 1.0, nous avons évalué la performance, l'évolutivité et la dynamique. Ces aspects permettent d'évaluer la première contribution de ce manuscrit. Le prototype passe à l'échelle linéairement w.r.t. l'augmentation des différentes entrées données à MANTUS, c'est-à-dire la taille du modèle TOSCA, le nombre et la complexité des scripts TML. De plus, le tissage AOP n'ajoute pas une surcharge significative au flux de travail. L'AOP weaving peut être intégré dans MANTUS avec les autres étapes du pipeline sans surcharge majeure, vu que son impact est d'un ordre de grandeur plus important que le « TOSCA matching ».

A.4.2 Spécialisation des infrastructure Multi-cloud

Comme on a présenté dans la Section A.3.2, construire des multi-cloud infrastructure à travers des MCLs basées sur IaC ne demande encore pas d’efforts en termes d’extensibilité IaC pour améliorer le passage à l'échelle mais aussi en termes d'optimisation pour obtenir la meilleure efficacité dans le déploiement. En Effet, le "dénominateur commun" dans les architectures multi-cloud implique toujours une spécialisation de l'infrastructure diminué comparée au scenario à CSP unique. Cela implique aussi une perte en terme de performance et optimisation des couts.

Dans ce manuscrit, le deuxième objective de recherche a été introduire un mécanisme qui permet de sélectionner l'ensemble de CSP qui représentent le meilleur choix pour les besoins et les paramètres du consommateur, en étant capable de profiter des spécificités de ce que chaque CSP est capable d’offrir.

D'abord, nous avons introduit le langage TML pour permettre la possibilité pour le consommateur de cloud de décrire ces rééquipements en terme des services non fonc-
tionnels, en plus du code fonctionnel de son application. Le TML représente "la colle" entre le code applicatif et les composants non-fonctionnels réutilisables.

De suite, nous avons introduit un algorithme d'ordonnancement basée sur le "context-based matching" pour sélectionner l'ensemble de CSPs plus adaptés pour le besoin du consommateur. Cet algorithme prend comme input la liste des CSP disponibles, leur data-centres et un ensemble de paramètres exprimés par le consommateur (ex. nombre minimum de CSP à sélectionner, template d'infrastructure de base, critère d'optimisation). A partir de ces entrées, l'algorithme produit la liste des data-centres plus adaptées appartenant à des différents CSPs. Le contexte est managé à travers une extension du formalisme TOSCA. Cette extension introduit dans la définition de chaque ressource TOSCA les paramètres nécessaires à identifier les fonctionnalités, les préférences et les limitations de chaque CSP et de chaque implémentation de service. L'algorithme, en considérant les fonctionnalités de chaque data-centre comparé au service désiré est capable de sélectionner les data-centres les plus intéressantes en cherchant des correspondances entre les capacités de chaque data-centre et leur configuration matérielle pour des services IaC décrit dans le template du consommateur.

Finalement, nous avons défini un workflow de bout en bout pour optimiser la création du multi-cloud. Plus précisément, le consommateur initialement modélise son code IaC comme un graph haut-niveau de services. En s'appuyant sur le TML et le context-based matching, le compilateur raffine le graph jusqu'à obtenir des ressources spécialisées, qui peuvent être déployés sur les CSP sélectionnés.

A.4.2.1 Context-based matching

Nous proposons d'étendre la matching de base TOSCA avec l'introduction d'un matching basé sur le contexte. En d'autres termes, nous introduisons une caractérisation supplémentaire sur les implémentations du service, la sélection des "candidats" pour remplacer le TOSCA NodeTypes plus abstraits. La "Context-based matching" en TOSCA permet de réifier des ressources abstraites basées dans une implémentation concrète optimisée, en prenant en compte les paramètres du consommateur, la configuration du centre de données CSP et le contexte de "brokering" pour matcher les deux ensemble.

Pour simplifier la compréhension, ce processus d'appariement (i.e. matching) peut être considéré comme un espèce de "marché du travail" où la définition du service abstrait, le nœud TOSCA à mettre en correspondance, ouvre une nouvelle "position" à pourvoir. Les "candidats", différentes implémentations ServiceTemplate alternatives, seraient choisies pour le "job", remplaçant le code Node, si et seulement si elles correspondent aux "critères" de position ouverte (par exemple, même ensemble) des capacités et exigences TOSCA exportées). D'une part, le "matching" traditionnel ne considère que si une implémentation spécifique est capable de répondre aux exigences en termes de fonctionnalités TOSCA (i.e. Properties, Capabilities, Requirements). D'autre part, nous enrichissons la portée en introduisant le contexte du déploiement et à travers le contexte, les fonctionnalités obligatoires et facultatives à satisfaire dans le contexte d'une implémentation spécifique. En d'autres termes, le nouvel algorithme de matching évalue les implémentations compatibles afin de maximiser l'évaluation d'une implémentation TOSCA choisie en fonction de critères non fonctionnels définis par le consommateur (coût, latence, performance).
Figure A.3: Construction de bout en bout d’une architecture Multi-Cloud à travers MANTUS et ORBITS

A.4.2.2 Implémentation: Mantus 2.0

Pour valider la correspondance basée sur le contexte, nous avons amélioré le prototype MANTUS présenté dans la section A.4.1.2. En bref, en tant que « compilateur » multi-nuage, MANTUS concilie: (1) l’extensibilité non fonctionnelle des modèles de base via l’AOP (voir le chapitre 3), (2) la capacité à s’adapter à Fonctionnalités CSP via une correspondance basée sur le contexte et (3) capacité à déployer des services sur un ensemble de CSP différents.

Du point de vue du matching basé sur le contexte (Context-based matching), comme nous l’avons présenté dans la section précédente, nous avons étendu le premier prototype réalisé pour valider l’extensibilité. D’abord, on a étendu définitions TOSCA Standard en introduisant une propriété de contexte toujours présente. En tirant parti de l’héritage orienté objet TOSCA, l’introduction de cette propriété est propagée à chaque type de nœud TOSCA utilisé. Pour Implémenter le “matching” basées sur le contexte,
A.4 objectifs des recherche et contributions

Nous présentons l'évaluation basée sur le contexte MANTUS en comparant les résultats de MANTUS effectuées par correspondance de contexte contre des variantes de machines virtuelles sélectionnées aléatoirement et des "flavors" à usage général. Nous observons que la sélection effectuée via des performances basées sur le contexte est meilleure que la ligne de base (c'est-à-dire entre 5% et 10%).

Enfin, nous avons validé le choix d'implémentation et de conception de MANTUS avec des tests d'efficacité et de performance. Premièrement, MANTUS s'est avéré efficace pour sélectionner des ressources optimisées sur différents CSP. La sélection de contexte correspondant aux critères de performance montre une amélioration moyenne de 5-6% lors de l'utilisation de la sélection aléatoire et de 10% lors de l'utilisation de services de type générale. Deuxièmement, nous avons montré que le temps nécessaire à l'exécution de MANTUS peut être améliorée en adoptant une approche "Generate-Then-Weave" pour l'étape "Matching & Weaving".

A.4.2.3 Construction et déploiement d'une architecture multi-cloud

Le troisième objectif que nous envisageons est de construire et déployer un multi-cloud en utilisant une infrastructure overlay. En utilisant une couche overlay comme infrastructure virtuelle et les paramètres mentionnés avant pour caractériser le workload, le constructeur multi-cloud doit être capable de construire une implémentation spécialisée pour le CSP sélectionné avec l'intégration des services non-fonctionnels désirés.

Pour obtenir cela, nous avons réalisé trois contributions. D'abord, nous avons défini une architecture overlay de référence. Cette architecture overlay regroupe les services pour (1) orchestrer les workloads et la conséquent allocation de ressources et (2) fournir une architecture de virtualisation adaptée, découpé du choix technologique du CSP subjacent.

De plus, nous avons fait le design et l'implémentation une architecture de virtualisation capable à fournir plus de contrôle à l'utilisateur comme mentionné in 1.3.3, toujours en fournissant une couche de compatibilité avec les outils existants. Our prototype,
basé sur micronoyau et la virtualisation imbriquée, ouvre à l'utilisateur les couches plus basses de l'infrastructure (ex. the hyperviseur), en assurant une forte isolation vis-à-vis des autres consommateurs et une compatibilité avec le reste du multi-cloud.

Finalement, nous avons implémenté un constructeur de multi-cloud, capable d'élaborer le code générique et agnostique au CSP reçu en entrée et fournir un déploiement de multi-cloud en sortie. Le constructeur, appelé MANTUS, permet de facto l'interopérabilité multi-CSP et l'optimisation du code vis-à-vis des fonctions du code généré. MANTUS introduit la possibilité de construire une infrastructure virtuelle overlay customisé au consommateur et il support un raffinement dynamique des ressources. MANTUS intègre dans son pipeline de travail ce raffinement avec une extensibilité d'infrastructure introduite avec le langage TML.

En effet, par rapport aux infrastructures privées, le consommateur perd non seulement la possession physique de données mais également la perte de contrôle sur les couches système « inférieures » de la pile logicielle, notamment la couche de virtualisation ou l'hyperviseur. Pour adresser ces points, nous avons d'abord introduit une infrastructure capable de prendre en charge des infrastructures multi-cloud en exploitant l’approche de superposition, ORBITS. Puis, on a réalisé le U-Cloud Node, un prototype d'architecture de virtualisation permettant au consommateur de mieux contrôler les couches de virtualisation en se basant sur la virtualisation imbriquée et les systèmes d'exploitation à micronoyau.

**ORBITS** Les approches traditionnelles basées sur la "superposition" ou "overlay" [48, 96] offrent à l'utilisateur un degré de contrôle important (par exemple, couche de virtualisation, outils de sécurité), mais elles ne sont pas conçues pour des fournisseurs multiples. L'idée de ORBITS est d'abstraire complètement les API des CSP dans une perspective d'orchestration et une perspective de virtualisation (par exemple, ne pas avoir à gérer différentes technologies d'hyperviseur). L'architecture ORBITS répond à l'exigence d'interopérabilité propre à l'infrastructure multi-cloud, car elle s'appuie sur une approche basée sur la superposition. ORBITS introduit un ensemble de services d'infrastructure capables de déployer un cloud IaaS complet, complètement découpé du déploiement sous-jacent. En exploitant ORBITS, le consommateur n'a pas à gérer les différentes technologies CSP, mais exploite directement les APIs fournies par ORBITS. La définition de l'architecture ORBITS, manipulée par MANTUS (Section A.4.2.2), est ensuite traduite dans la mise en œuvre la plus adaptée selon le contexte donné. La figure A.5 donne un aperçu de l'architecture ORBITS, qui adopte un design à trois couches. Chaque couche implémente une partie différente d'une architecture multi-cloud:

- **Le virtualization layer** ou "couche de virtualisation" est chargé d'exécuter les tâches, avec des compromis entre performances et isolation parmi les différents workloads. Il fournit une vue homogène des services de virtualisation aux couches supérieures afin de répondre aux exigences de qualité de service (par exemple, pour les systèmes de DSE).
- **Le management layer** ou "couche de gestion" est chargé de l'approvisionnement des ressources pour chaque fournisseur de superposition, de la gestion de la couche de virtualisation et de la création de nouveaux environnements d'exécution (EE). Cette
La couche répond également à l’exigence de qualité de service, en mettant l’accent non seulement sur l’exécution des applications, mais également sur l’accès aux ressources.

• La couche orchestration layer garantit un provisionnement flexible sur plusieurs fournisseurs requis par les cas d’utilisation. Il a une vue d’ensemble des fournisseurs de cloud ou CSPs disponibles et coordonne l’orchestration des applications entre les instances de fournisseur. Par exemple, il peut être déployé sur plusieurs fournisseurs en s’appuyant sur le nombre requis de répliques nécessaires pour garantir une haute disponibilité des services (pour DSE) ou une isolation efficace du fournisseur (ex. MPC sur les données de santé).

Les services de couche de gestion et de virtualisation sont déployés sur chaque fournisseur sélectionné pour être dans le multi-cloud. Nous faisons référence à ces instances sous la forme de overclouds, car ce sont des occurrences de superposition qui fournissent une vue homogène des ressources à la couche d’orchestration.

Nous avons orienté les principes d’ORBITS vers une infrastructure cloud traditionnelle qui fournit des environnements d’exécution orientés serveur. Le provisionnement des ressources locales est fourni par une instance OpenStack superposée pour chaque fournisseur sélectionné, exploitée comme couche de gestion. Le point d’entrée du multi-cloud est réalisé à travers un ensemble d’instances coopérantes de Mesos Masters qui fourniront une orchestration de services multi-cloud et au niveau de l’application. Le rôle de STRATOPAUSE est joué par un "esclave" Mesos utilisé en tant que proxy, quiannonce les ressources sur l’ensemble des Masters et déplie les tâches correspondant à la répartition principale, en connectant la configuration locale à l’orchestration des applications. En tant que couche de virtualisation, nous avons expérimenté un large spectre de technologies de virtualisation de systèmes, en prenant en compte KVM, Xen et LXC.

Afin de développer un modèle ORBITS, nous avons d’abord introduit une hiérarchie des types de ressources TOSCA pour modéliser les services d’infrastructure, en mettant l’accent sur les ressources informatiques et réseau. Les composants d’infrastructure ont été décrits comme des modèles TOSCA à l’aide de notre NodeType personnalisé. Nous
 avons également défini un type d’encapsulation ORBITS overcloud pour favoriser la réutilisation des modèles pour les environnements multi-cloud des différents consommateurs.

Nous évaluons si l’approche consistant à avoir une infrastructure «imbriquée» au-dessus des infrastructures contrôlées par CSP est acceptable. Pour ce faire, nous évaluons la surcharge due à la virtualisation imbriquée, nous avons évalué notre système à la fois en termes de performances et d’évolutivité.

En résumé, les résultats montrent que seule la technologie au niveau du système d’exploitation peut être correctement imbriquée dans la virtualisation intégrale traditionnelle sans perte de performance majeure. LXC permettrait de créer des "VM légères" sans avoir à intégrer le modèle de conteneur d’application Docker tout en utilisant. En exploitant Xen et KVM, les consommateurs doivent faire face à une baisse de performance importante, bien qu’ils obtiennent une meilleure isolation par rapport à LXC qui implémente l’isolation entre les environnements d’exécution via le système d’exploitation et non pas via le matériel.

A.4.3 Perte de contrôle

Comme nous l’avons mentionné précédemment dans cette section, nous abordons le problème de perte de contrôle dans le infrastructures cloud et multi-cloud. Une telle « perte de contrôle » est une conséquence directe du multiplexage du même matériel physique entre plusieurs clients et des administrateurs système omniscients.

U-CLOUD NODE Comme nous en avons discuté dans le chapitre 2, une quantité importante de travaux de recherche cible l'architecture de virtualisation en se concentrant sur l'amélioration de la fiabilité ou l'amélioration de la sécurité. Le résultat est qu'aucune architecture n'est complètement satisfaisante du point de vue de la sécurité, de la personnalisation de l'utilisateur et de la compatibilité héritée. Le consommateur de cloud peut reprendre le contrôle grâce aux principes suivants:


2. **Administration des ressources en boîte noire** Les administrateurs malveillants peuvent tirer parti de leurs privilèges étendus pour surveiller l’état d’exécution privé des applications, sans aucun contrôle utilisateur possible. Contrairement à l’approche monolithique des hyperviseurs "General purpose", les micro-hyperviseurs ([50, 72]) modulaient le plan de contrôle traditionnel des hyperviseurs à usage général (par exemple Dom0 dans Xen), ouvrant aux utilisateurs Ressources. Cependant, cette approche nécessite que l’utilisateur adapte son infrastructure de contrôle à cette nouvelle logique, en rompant la compatibilité avec les boîtes à outils de gestion existantes.
Support à l’existant

La virtualisation imbriquée hérite de la transparence des hyperviseurs à usage général, permettant de contrôler les ressources avec les mêmes interfaces, alors qu’Hypervisors et les micro-hyperviseurs introduisent une nouvelle logique de contrôle et/ou une interface hyperviseur.

Dans la figure A.6, nous montrons une représentation de haut niveau de l’architecture U-Cloud que nous proposons dans ce manuscrit. Dans cette figure, nous pouvons observer le concept de U-Cloud, dans lequel le consommateur contrôle les couches inférieures de l’architecture. En fait, en utilisant la virtualisation imbriquée, nous pouvons appuyer sur deux couches de système, l’une "Upper Layer" complètement contrôlée par l’utilisateur (i.e. l’architecture ORBITS) et l’autre, la "Lower Layer" dont le contrôle est partagé l’utilisateur et le fournisseur de la modularité du micronoyau.

Nous avons implémenté l’architecture U-Cloud en utilisant le micronoyau Genode / NOVA comme couche inférieure et Xen comme hyperviseur L1.

Malgré la validation fonctionnelle du test, nous avons rencontré plusieurs problèmes de performances importants liés à l’ordonnancement du processeur. Premièrement, Nova [108] ne s’appuie pas sur un algorithme d’ordonnancement flexible et complexe comme Linux, mais implémente un simple Round Robin et fixe une fois pour toute un certain processus sur un CPU core au démarrage de ce processus. La dégradation des performances que nous avons subie était notamment présente lors de l’exécution de VM L1 avec plusieurs vCPU sur les mêmes cœurs physiques.

A.5 CONCLUSION

Dans cette thèse, nous avons présenté trois principales contributions. Les deux premières proposent une amélioration du paradigme IaC à travers l’introduction de l’extensibilité et de la spécialisation. Cela a été obtenu à travers le compilateur-multi-cloud MANTUS, dont il est possible d’observer un schéma de fonctionnement en figure A.8. MANTUS permet à un consommateur de cloud de construire un multi-cloud en répondant aux exigences strictes d’extensibilité et de spécialisation.

Avant MANTUS, les approches actuelles du multi-cloud présentaient deux inconvénients majeurs, qui rendaient difficile l’adoption du multi-cloud. Premièrement, la base de code et l’effort requis pour maintenir les services deviennent rapidement très importants. Cela peut empêcher un consommateur multi-cloud de bénéficier de la même flexibilité que dans un écosystème à un seul cloud provider. Les modèles IaC ne sont pas facilement extensibles. Deuxièmement, les approches traditionnelles aux multi-cloud présentent des problèmes de sous-spécialisation. La mise en place d’une infrastructure intercloud par IaC exige non seulement un effort supplémentaire en termes de connaissance CSP mais, même lorsque cela est réalisé, il était difficile d’optimiser le service. La variété des fonctionnalités dans différents écosystèmes de cloud et l’utilisation utilisateur souhaitée.

MANTUS fournit une solution qui améliore la spécialisation et l’extensibilité en étendant le paradigme IaC. IaC est normalement adopté par les consommateurs pour con-
Multi-cloud Deployment

Figure A.8: Résumé du déploiement de bout en bout à travers ORBITS et MANTUS

Pour résumer, les objectifs de MANTUS sont: (1) personnaliser le modèle de cloud en fonction des services de sécurité demandés par le locataire et des fonctionnalités offertes par les fournisseurs de services de chiffrement sous-jacents; et (2) sélectionner un sous-ensemble de fournisseurs de cloud, compatible avec les politiques exprimées par les besoins du locataire. MANTUS tire parti de deux contributions introduites dans cette thèse.


Deuxièmement, le “matching” basé sur le contexte permet de réifier des ressources abstraites TOSCA dans une implémentation concrète optimisée, prenant en compte les paramètres du consommateur et la configuration du centre de données CSP. Nous avons fourni une définition formelle de ce mécanisme et nous avons montré son adoption en plusieurs étapes de la construction multi-cloud. Nous avons présenté une nouvelle version du MANTUS multi-cloud builder qui implémente des principes de conception basés sur le contexte et comment ils répondent aux exigences des cas d’utilisation. Nous avons étendu le principe de correspondance TOSCA traditionnel pour traiter le CSP et le contexte consommateur. Le résultat permet d’optimiser la "compilation" d’un modèle générique, en obtenant un résultat optimisé à partir d’un modèle abstrait.

En exploitant ces blocs de construction, MANTUS est capable de définir et déployer une infrastructure de cloud sur un groupe de fournisseurs sélectionné. A partir des paramètres CSP et consommateur, MANTUS construit un multi-cloud adapté et personnalisé sélectionnant les meilleurs centres de données fournisseurs en fonction des spécifications de l’utilisateur.

La troisième contribution, présentée au chapitre 5, propose an architecture pour limiter la perte de contrôle des méthodes de déploiement actuelles des infrastructures cloud et multi-cloud. Plus précisément, nous avons proposé ORBITS, une infrastructure de superposition en tant que couche commune pour le multi-cloud, et une pile de virtuali-
A.6 LIMITATIONS

Dans cette thèse, nous avons axé la portée multi-cloud sur la manière dont un modèle de service peut être déployé facilement et efficacement sur plusieurs CSP. Ainsi, MANTUS n’aborde pas plusieurs limitations qui concernent l’ensemble du multi-cloud dans une large mesure et peuvent avoir un impact sur le déploiement réel des multi-clouds.

Premièrement, les fournisseurs de cloud n’exposent pas toujours leurs ressources dans leurs propres langages « Infrastructure as Code ». Par conséquent, la capacité de MANTUS à spécialiser la spécification de modèle ne serait pas capable de couvrir toutes les ressources sur différents fournisseurs de Cloud.

Deuxièmement, MANTUS n’intègre pas de prédicteur qui sélectionne les ressources les plus adaptées au contexte. Dans le chapitre 4, nous avons montré comment un contexte MANTUS est construit à partir des benchmarks CSP. Cependant, nous n’avons pas automatisé ce processus afin de permettre à MANTUS de se familiariser avec la pertinence d’une implémentation spécifique, en créant un prédicteur multi-CSP. Ce manque peut avoir une incidence sur l’évolutive et la facilité d’utilisation de MANTUS puisque l’opérateur doit alimenter manuellement le contexte MANTUS.

Troisièmement, la fédération d’identité sur plusieurs fournisseurs n’a pas été traitée dans ce manuscrit. La gestion des identités est un sujet important dans la construction de multi cloud et nous l’avons adressé simplement en considérant d’obtenir manuellement le bon jeton d’accès pour le compte client pour plusieurs fournisseurs de cloud. Cela rendrait l’exploitation d’une solution similaire inconfortable dans un système réel et devrait être traitée.

A.7 FUTURE WORK

Les orientations futures du travail se concentreront sur trois directions principales.
Figure A.9: Résumé des contributions de thèse
A.7.1 Améliorations dans le tissage d’aspect non fonctionnel

Nous pouvons observer une prise en compte définitive de l’importance qu’il va interrompre le blocage dans le choix des fournisseurs de la part des consommateurs. Cela a été effectivement obtenu par l’adoption de la conteneurisation et l’émergence de cadres d’orchestration de conteneurs, tels que Kubernetes et Swarm. Ces frameworks sont normalement des projets open-source conçus pour s’exécuter sur plusieurs CSP et aussi sur des infrastructures privées. L’énorme popularité de ces frameworks a poussé tout grand fournisseur ou gestionnaire de cloud IaaS à les prendre en charge, afin d’intégrer leurs services (par exemple, Load Balancer en tant que service, gestion des identités) dans ces frameworks d’orchestration de conteneurs. De plus, les CSP ont commencé à proposer à leurs clients la possibilité de s’appuyer sur une instance entièrement gérée de ces frameworks. Cette tendance introduit non seulement un multi-cloud centré sur le fournisseur de facto puisque les fournisseurs proposent enfin des services qui implémentent exactement les mêmes abstractions et API, mais brisent le fossé existant entre les analyses centrées sur le client et centrées sur le fournisseur dans de nombreux ouvrages [2, 42, 55] puisque tout consommateur peut déployer le même framework dans ses locaux ou sur tout service IaaS.

Les travaux futurs étudieront ces tendances en analysant en profondeur si le modèle présenté peut complètement satisfaire les exigences multi-nuages présentées dans la section 1.1. Il est à noter que dans un tel scénario, l’approche de MANTUS reste intéressante. En fait, les consommateurs, tels que les CDO, ont des contraintes spécifiques en termes d’infrastructure qu’ils déploient du point de vue de la sécurité et de l’intégration avec leurs services existants. Ce qui est intéressant, c’est qu’ils peuvent facilement avoir des déploiements d’applications interopérables au-dessus de différents CSP, comme nous l’avions envisagé de proposer des orbites.

A.7.2 Specialization and predictions

Dans ce travail, nous présentons plusieurs repères de disponibilité et comment ils peuvent être utilisés pour construire le contexte MANTUS. Cette construction a été faite manuellement, limitant la portée aux ressources IaaS. Des travaux connexes ont montré qu’une analyse similaire peut conduire à des prévisions de performances précises [101]. Cependant, ces travaux ont présenté une analyse limitée à ce jour à un fournisseur unique. En exploitant les capacités multi-cloud MANTUS, nous étudierons la possibilité de tirer parti de ces techniques sur plusieurs fournisseurs sur une plus grande échelle que l’informatique IaaS traditionnelle. En fait, le travail présenté dans ce manuscrit n’a pas étudié en profondeur les potentiels et les difficultés possibles d’intégrer dans le pipeline multi-cloud un nouveau paradigme tel que l’informatique sans serveur. Ce paradigme peut être difficile à adopter. Il promet d’importants avantages en termes de performances et d’évolutivité des applications, mais peut avoir un impact sérieux sur la portabilité des intrusions de code au niveau des applications.

A.7.3 Reconfiguration Autonomique

Comme nous l’avons présenté dans la partie d’évaluation, les déploiements incrémentiels peuvent être utilisés pour l’inter-synchronisation entre les overclouds instanciés sur
différents centres de données. Nous étendrions cette capacité pour introduire une reconfiguration multi-cloud autonome. Pour ce faire, nous explorerons l'idempotence et les conflits lors du tissage de plusieurs scripts utilisés par le tissage basé sur AOP. Plus précisément, nous évaluerions la complexité du langage des règles de filtrage, en élaborant des règles plus complexes et en évaluant les performances, la complexité de la règle et pas seulement leur nombre.