A Methodology and a Tool for QoS-Oriented Design of Multi-Cloud Applications

Giovanni Paolo Gibilisco

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB)

Politecnico di Milano

Supervisor: Prof. Danilo Ardagna

Co-supervisor: PhD. Michele Ciavotta

Tutor: Prof. Carlo Ghezzi

A thesis submitted for the degree of

Doctor of Philosophy

05 Feb. 2015

Abstract

This work focuses on the support of the development of multi-cloud enabled applications with Quality of Service (QoS) guarantees. It embraces the model driven engineering principles and aims at providing development teams with methodologies and tools to assess the expected QoS of their applications early in the design stages. To do so we adopt and enrich different component based and UML-like modeling technologies like the Palladio Component Model and MODACloudML extending them in order to determine an optimized deployment in multi-cloud environments by introducing a new cloud specific meta-model. The integration of the new meta-model into state of the art modeling tools like Palladio Bench or Modelio allows software architects to use well known modeling approaches and specify a cloud specific deployment for their applications. In order to ease the portability of both the model and the application the meta-model uses three abstraction levels. The Cloud enabled Computation Independent Model (CCIM) allows to describe the application without any reference to specific cloud technologies or providers; the Cloud Provider Independent Model (CPIM) adds the specificity of some cloud technologies introducing concepts like Infrastructure and Platform as a Service (IaaS/PaaS) but still abstracts away the specificity of each particular provider; the Cloud Provider Specific Model (CPSM) adds all the details related to a particular cloud provider and the services offered allowing to automatize the deployment of the application and generate performance models that can be analyzed to assess the expected QoS of the application. High level architectural models of the application are then transformed into a Layered Queuing Network performance model that is analyzed with state of the art solvers like LQNS or LINE in order to derive performance metrics. The result of the evaluation can be used by software architects to refine their design choices. Furthermore, the approach automates the exploration of deployment configurations in order to minimize operational costs of the cloud infrastructure and guarantee application QoS, in terms of availability and response time. In the IaaS context, as an example, the deployment choices analyzed by the tool are the size of instances (e.g. Amazon EC2 m3.xlarge) used to host each application tier and the number of replicas for each hour of the day. The problem of finding the optimal deployment configuration has been analyzed from a mathematical point of view and has been shown to be NP-hard. For this reason a heuristic approach has been proposed to effectively explore the space of possible deployment configurations. The heuristic approach uses a relaxed formulation based on M/G/1 queuing models to derive a promising initial solution that is then refined by means of a two level hybrid heuristic and validated against the LQN performance model. The proposed methodology has been validated by two industrial case study in the context of the MODAClouds project. A scalability and robustness analysis has also been performed and shows that our heuristic approach allows reductions in the cost of the solution ranging from 39% to 78% with respect to current best practice policies implemented by cloud vendors. The scalability analysis shows that the approach is applicable also to complex scenarios, with the optimized solution of the most complex instance analyzed being found in 16 minutes for a single cloud deployment and in 40 minutes for a multi-cloud scenario.

Contents

List of Figures

Chapter 1

Introduction

One of the most pervasive changes happened in recent years in the IT world is the appearance on the scene of *cloud* computing. The main feature of this new computing paradigm is its ability to offer IT resources and services in the same way fresh water or electric power is offered, as a utility. In a traditional environment in order to make use of a software system to address some business needs a company would need to acquire and manage the hardware infrastructure, different software stacks and, in many situations, develop their own software. Cloud computing changes this paradigm by offering all these elements as services that the user can acquire and release with high flexibility.

Cloud providers offer a variety of IT resources using the "*as a Service*" paradigm. Complex software systems that require multiple application stacks and different hardware resources can now be acquired, entirely or in parts, in matter of minutes. Hardware resources like computation nodes, storage space or network capacity are offered as *Infrastructure as a Service* (IaaS), software stacks that allows application developers to run their own code are offered as *Platform as a Service* (PaaS), finally, entire software system that can be directly used to provide some business value without the need of developing a new system are offered as *Software as a Service* (SaaS).

Since the early appearance of this technology in the market, many companies have decided to evolve their own infrastructure in order to embrace this new paradigm and the offering o cloud services, as well as the number of cloud providers, has grown quickly [38].

There are many advantages introduced by the adoption of the cloud technology, among all one of the most important is the flexibility in the acquisition and decommission of software systems and the underlying infrastructure. This advantage is due to the ability of cloud providers to offer an almost unlimited amount of resources and a pricing policy that allow application developers to avoid long term commitments and pay only for resources they actually use. Another key advantage brought by the adoption of the cloud paradigm is the shift of responsibility in the management of the portion of the software system that is acquired from the cloud provider. If, for example, a company decides to decommission its own physical infrastructure in favor of a new infrastructure offered by a cloud provider all the maintenance operations required by the hardware and some software systems, like OS acquisition and update, are delegated to the cloud provider allowing the internal IT team of the company to focus on tasks that provide more value for the company.

Delegating management responsibility of part of the infrastructure to a third party, in this case a cloud provider, comes inevitably with a loss of control on the entire infrastructure. This change creates some new challenges for teams that were used to build entire software systems from the ground up. When faced with the selection of a cloud service the application developer has to take into consideration many new characteristics that he/she was probably not considering before. The wide variety of similar services, the lack of interoperability between APIs of services offered by different cloud providers and the lack of specific training for developers are just a few of the new challenges that the IT staff of a company has to face when considering a migration to a cloud infrastructure. Furthermore, the loss of control on the infrastructure exposes users to the variability of QoS offered by cloud providers. Usually providers address this issue by providing generic Service Level Agreements (SLAs) specifying their expected QoS level and providing discounts on future usage of their services in case the specified QoS is not met. Amazon EC2 SLA, for instances provides an availability of 99.95% of up-time in a month for VMs and in case this availability level is not met users are granted a discount on 10% on service cost. In many situations such a discount is not comparable to the possible loss generated by the downtime of the application. Many examples of cloud outages like the ones happened recently to Google Cloud [1] or Microsoft Azure [2] shows that availability concerns play a major factors in

[1]https://status.cloud.google.com/incident/compute/15045
[2]https://azure.microsoft.com/en-us/status/

moving a critical application to the cloud environment.

A solution to this problem comes from the wide variety of similar cloud services offered by other providers. If, as an example, the software architect thinks that the application under development is critical and requires an availability of 99.999% (also called 5-nines availability) he/she could replicate the application on two cloud providers, say Amazon AWS and Microsoft Azure, obtaining the required availability. Moreover, the use of multiple providers might allow the application developer also to redistribute the incoming workload in order to exploit differences in pricing in order to reduce the operational costs.

The contributions of this thesis are a methodology and a tool, to help software developers to build applications capable of exploiting the cloud infrastructure. In particular the work presented in this thesis tries to simplify the development process by providing software architects with a *meta-model* in order to describe possible deployments of their application over the cloud infrastructure. We then automate the QoS and cost analysis of such deployments in order to provide software architects with feedback on their deployment choices. Finally, we automate the generation of possible deployment configurations, possibly on multiple cloud providers, in order to minimize infrastructural costs and guarantee a desired QoS level.

Our work embraces the Model Driven Engineering (MDE) paradigm and makes use of models of the application specified at different levels of abstractions. Allowing the software architect to start by building simple component based models in a UML-like notation and then refine them adding information related to the desired cloud environment allows the development team to keep the focus on the functionality of the system their are building and delegate some of the architectural choices to the tool we have developed and integrated in the modeling environment.

Our approach targets the development team and in particular software architects, application developers and domain experts. The modeling paradigm that we embraced allow separation of concerns between these figures involved in the software development process. In the reminder of this thesis we will refer to this actors as *users* of our tool. In contrast the users of the cloud applications developed by using our approach and deployed on a cloud environment are mentioned as *final users* or *end users*. When the use of these terms might generate ambiguity we will speak directly of software architects or application developers.

The use of state of the art performance models and tools allows to provide to the development team estimation on the expected QoS of the system under development, in order to take informed decisions and adapt the design of the system early in the design stages avoiding complex and expensive re-factoring of the application.

The proposed approach allows designers to annotate theirs models with requirements related to the QoS of the application, like the expected response time of certain functionality or the minimum availability of the system and delegates to the tool the task of finding a deployment plan capable of fulfilling such constraints.

The deployment optimization strategy proposed in this thesis explores a huge and complex space of possible configurations by assigning to each component described in the model of the application a particular cloud service and analyze the behavior of the entire application in order to see if a particular choice of cloud services is capable of fulfilling the constraints. This search process takes also in to consideration the cost of the deployment solution and tries to minimize it.

The scientific literature shows some similar approaches that try to automate deployment decisions on component based systems but, to the best of our knowledge, this is the first approach that targets directly multi-cloud environments. In [49], Koziolek shows that due to the increasing size and complexity of software systems, architects have to choose from a combinatorially growing number of design and deployment alternatives. Different approaches have been developed to help architects explore this space with the help of automated tools like [10, 25] or [23, 49]. These approaches are presented in Chapter 2 help developers to analyze different design choices but do not address directly the cloud environment.

We argue that the problem becomes significantly more complex when considering the cloud services that can be employed for the execution of the application components. Traditionally the allocation problem has been considered independently from the design of the application but the possibility of exploiting different cloud services for different parts of the application has an impact on how the entire system works and makes the deployment problem even more relevant in the design phase.

If we consider even the simple example of a web application deployed on a single tier, we need to decide if we want to use a PaaS service to host our application code or to directly manage the platform used to run our code, say a Tomcat server. This choice directly affects the design and the development of the application. If we choose to manage directly a Tomcat instance and deploy it on a Virtual Machine (VM) offered by Amazon, we still need to decide which type of VM we need to use and how many number of replicas of this machine according to the expected number of end user of our system. Using a high number of cheap VMs, like the m3.large in order to cope with a variable workload might seem a good strategy but the software stack needed to run our application might required more resources and, in this case using a smaller number of more powerful instances, like the c4.3xlarge, might be more convenient.

In a multi-cloud scenario this problem becomes even more complex because, beside making these decisions for both providers, we also have to define how the incoming workload is split among the providers. Since the performance and the price of resources offered by cloud providers might change with the time of the day, this problem is very dynamic and requires some automation to help the designer to generate possible configurations.

When we deal with a more realistic and complex application the development team is faced with deployment decisions and analyzing all the possible alternatives is a daunting tasks that calls for automation. The tool developed during this work, called SPACE4Cloud (System Performance and Cost Evaluation for Cloud), allows to automate the exploration of these design alternatives. SPACE4Cloud has been developed in the context of the MODAClouds FP7 IP [1] European project and constitutes one of the core tools of the MODAClouds IDE design time environment. Our approach takes into consideration QoS constraints that predicate on the response time, both on average and percentiles, constraints on the availability of the application and *service allocation constraints*. By service allocation constraints we mean those constraints that are related to the type of technology chosen to build the application and include minimum requirements on some characteristics of the cloud services required to host specific components, e.g., minimum amount of memory or cores, or limitations on the scalability of some service. Our approach differs from those already available in the literature, since it targets directly the cloud environment taking into consideration some peculiar features. Cloud environments are naturally shared by multiple users, the use of a shared infrastructure might lead to contention problems. To address this kind of behaviors we make use of a performance analysis tool called LINE that take into consideration variability in the characteristics of the processing resources by using

[1]www.modaclouds.eu

a statistical characterization (via Random Environment [29]). Web applications, like those developed in a cloud environment are also dynamic and the number of end users and the price of the cloud resources changes during the day. In many applications, the incoming workload shows a daily pattern, for this reason we introduce a time-dependent workload profile over a 24-hour time horizon, which leads to the solutions of 24 intertwined capacity allocation problems.

We first introduce the modeling paradigm proposed to apply the MDE principles in the context of cloud application development in Chapter 3. We also present an industrial case study that is used later on in the evaluation of the approach and throughout the thesis to clarify both modeling concepts and the optimization approach.

We then introduce the general design methodology and optimization strategy used to tackle the problem along with the architecture of the tool in Chapter 4. We then formalize the problem in Chapter 5 and we show it is equivalent to a class of NP-hard problems. These initial part of the work has been submitted for publication in IEEE Transactions on Software Engineering. In Chapter 6 we use a simplified performance model and a relaxed formulation of the problem in order to quickly derive a promising initial solution for the heuristic algorithm and in Chapter 7 we describe in details the main optimization algorithm used to explore the design space and derive the optimized deployment configuration. Details on the impact of the initial solution on the entire optimization procedure have been published in [17].

To validate our approach we have used two industrial case studies that show how software architects can benefit from using the early QoS analysis and deployment optimization provided by our work. We have also inspected how the complexity of the application under development affects the cost of the solutions obtained and the time required to execute the optimization with a scalability analysis, the results of this study is reported in Chapter 8. We compared our heuristic approach against common used threshold based heuristic that keep the utilization of the system below 60% or 80%. Using our heuristic we found optimized solutions with cost reductions ranging from 39% to 78%. The analysis also shows that the algorithm is both scalable and robust, the optimized solution for the most complex case was found in 36 minutes in a single cloud scenario and 42 minutes in multi-clouds. Robustness has been analyzed by repeating several times the optimization procedure during the scalability analysis and in the worst case the standard deviation of the time spent in the optimization is 18%

of the average execution time and the standard deviation of the cost of the solution is within 6% of the average solution cost. For what concerns the correctness of the QoS estimation with respect to the real model we relay on the extended literary work on performance prediction based on LQN starting from [36] that analyzes the accuracy of the QoS prediction for LQN models. With respect to the characterization of the parameters of the performance model used to evaluate application QoS we relay again on previous works like the one by Casale et al. [68] that presents several techniques to estimate application demands.

A discussion of the results achieved and an outline of future work are drawn in Chapter 9.

Chapter 2

State of the art

This chapter provides a critical view on previous works. We first introduce some approaches that deal with the problem of modeling applications with a particular focus on QoS characteristics, in Section 2.1; then we analyze some works that try to solve the problem of deploying component based applications under QoS constraints. Few approaches directly target the cloud environment but most of the works presented in this section focus on similar domains or use techniques that are similar to the one we adopted; finally we briefly present a classification of meta-heuristics approaches in general and describe our approach in terms of the classification, in Section 2.3.

2.1 Modeling Approaches

As we have described in Chapter 1, the contribution of this thesis lays in the area of Model-Driven Quality Prediction (MDQP). MDQP is a special phase of Model Driven Engineering (MDE) that focuses on the identification of performance characteristics of a system. The MDQP process starts with the specification of the system under study, both in terms of the application and the running infrastructure, by using a high level modeling language that allows the characterization of functional and non aspects, like the QoS.

The application of MDE principles in the context of software development is called Model Driven Development (MDD). The most known language for MDD is the Unified Modeling language (UML), the main feature of UML is its ability to represent a software system under different aspects. This ability comes from the extensibility of the language by the specification of *profiles*. UML allows software architects to specify separate diagrams for different views of the application. Object and Class diagrams, for example, can be used to specify the architecture of the system, activity diagrams and state machines can be used to represent behavioral aspects. A UML profile is a meta-model that can be used in conjunction with the UML meta-model to tailor the language to describe particular contexts. This mechanism have been extensively used to create Domain Specific Languages (DSL) as extensions to UML.

To support QoS analysis, for example, the Object Management Group (OMG) presented two UML profiles: the profile for Schedulability, Performance and Time (SPT) [63] and the Modeling and Analysis of Real-Time and Embedded system (MARTE) profile [64]. These extensions allow to model resource consumption, application components allocation and non-functional properties. MARTE introduces a Generic Quantitative Analysis and Modeling profile (GQAM) that includes concepts shared among many kind of analyses like resources, behaviors and workloads. The GQAM profile has been further expanded into three separate sub-profiles focused on different quality aspects: SAM is focused on Schedulability aspects, DAM is focused on Dependability and PAM is focused on Performance.

MARTE also specifies a language to define Non Functional Properties (NFP). This language allows to define properties of qualitative and quantitative nature by the use of predefined or user defined types. Variables, Expressions and Qualifiers can also be specified to characterize the precision of a property and allows derivations of other properties.

Another extension to the UML language is the Object Constraint Language (OCL) that can be used to specify restrictions on the values that certain elements of the UML diagram can assume.

Many modeling tools have been developed to support designers in the use of UML and its wide variety of profiles. In particular Papyrus[1] is an open source tool based on the popular Eclipse IDE. It implements the complete UML2[2] specification and provides support for many UML profiles like MARTE.

Modelio [71] is an open source UML modeling tool that supports UML2 and Busi-

[1]https://eclipse.org/papyrus/
[2]http://www.omg.org/spec/UML/2.0/

ness Process Model and Notation (BPMN) specifications. It also provides support for the MARTE profile and has been extended with the support for the specification of cloud resources and QoS characteristics by the MODAClouds European project. This tool has been used in our work as main interface for modeling activities.

The fUML[1] and Profiles for Performance Analysis (UPUPA) [8] tool provides a model-based analysis framework to support non-functional properties early in the development process. It carries out performance analysis using traces obtained by running Executable UML Models.

Other tools that support the UML or UML2 specification and various profiles are: MOdeling Sofware KIT (MOSKitt)— [3], ArgoUML [1], StarUML [6], UML Designer [7], MagicDraw [2] and IBM Rational Software Architect [5].

These tools allow architects to consider QoS characteristics in the modeling phase, but most of them lack the support of automated techniques to derive accurate performance models, needed to make analysis of the expected QoS, from these high level specification. Furthermore, recent studies [69] show that UML is not considered adequate for wide adoption by the industry and in many cases it is not used, or it is used in a way that does not comply with the MDD principles.

The lack of adoption of UML as a standard modeling language led to the development of alternative languages; with respect to the cloud environment, the *Organization for the Advancement of Structured Information Standards* (OASIS) is developing the *Topology and Orchestration Specification for Cloud Applications* (TOSCA)[2] specification, with the goal of enhance portability of applications and services among providers. They do so allowing the specification of applications, cloud infrastructures and operational behaviors (e.g., the application deployment) in a cloud provider independent way. This specification is still focused on functional aspects of the development and is mainly interested in supporting the portability of models and software artifacts across different cloud vendors.

A similar language, capable of describing functional aspects of the application and its management in a cloud vendor agnostic way, is CloudML [34] developed by Sintef[3]. The proposed domain specific language is supported by a run-time environment

[1]http://www.omg.org/spec/FUML/
[2]https://www.oasis-open.org/committees/tosca/
[3]http://cloudml.org/

capable of provisioning and managing the cloud resources and services described in the application model. This language is evolving in order to support the specification of different infrastructures but still lack the support for QoS characteristics.

Other approaches that exploit specifically designed languages, such as KLAPER [42] and the Palladio Component Model (PCM) [23], exist but they all lack specific support for the cloud environment. Recent surveys of modeling approaches that take into consideration performance prediction aspects can be found in [20], [51], [10] and [21].

In particular [51] analyzes different performance prediction approaches for component based software systems. The author describes how the MDQP approach can be applied to component based systems by attaching non functional information, such as resource demands, to the component specification and performance related information to the middleware or infrastructure used to host those components. The paper presents many performance evaluation techniques that are based on UML or on proprietary meta-models, focus on the impact of the middleware or use formal performance specifications.

In the context of self-adaptive systems [21] presents the SimuLizar tool that extends the Palladio modeling environment with the ability to model and evaluate transient phases that occur during a self-adaptation process. This tool has been further expanded in [22] with the support of the RELAX requirement language to specify possible ranges of requirements fulfillment and has been used to evaluate the compliance of user defined adaptation strategies with respect to requirements expressed on the adaptation phase itself, e.g., the speed of the adaptation after a violation of a QoS requirement is discovered.

To further support the adoption of MDE principles the OMG proposed the Model Driven Architecture (MDA) [4] to guide software engineers in building systems using a model-centric perspective. MDA propose three abstraction layers: the *Computational Independent* layer is used to express business scenarios and system goals; the *Platform independent* layer comprises multiple architectural views of the system to specify design and quality issue; the *Platform Specific* layer enriches the design with technological details. The use of separate abstraction layers allows architects to focus on different aspects of the system and separate concerns related to the business aspect, architectural choices and the technology.

After the specification of architectural models by the software development team, the MDQP approach uses automatic transformations of these models into performance models like LQNs or Markov Chains (see, e.g., [77]), more suited to be analyzed, or simulated in order to derive QoS characteristics.

An analysis on the derivation of different performance models from the higher level PCM is presented in [25]. The authors analyze the transformation process used to derive a simulation model (SimuCom), a Queuing Petri Net and a LQN model and compare the results of the analysis of the resulting models in order to quantify the trade-off between accuracy and time required for the analysis. The authors show the semantic gaps between the presented performance models and compare the results of the evaluation on four case studies. This comparison can be used as a guideline in the selection of a performance model. In our approach, as an example, a high number of solutions has to be evaluated so the use of a fast performance model is preferred.

Most of the approaches aims at providing the user some feedback on the expected QoS of the system under design and rely on the expertise of users to make adjustments to the application or its deployment in order to solve QoS problems identified by the analysis. Our approach aims at ease this process by suggesting to the user possible deployment choices generated by optimizing a cost function and guaranteeing QoS constraint satisfaction.

2.2 Other approaches for Designing Applications with QoS Guarantees

In order to present how similar optimization problems have been approached in the literature we use a classification partially derived by the one presented in [59]. This classification is less complex that the one proposed in [74] that we will use to describe our approach in Section 2.3 but is better suited to describe a broad range of similar approaches. We will divide the presented approaches in three main categories: rule-based, meta-heuristic, and Generic Design Space Exploration (GDSE).

Rule-based approaches. This category groups together approaches that embed performance knowledge into feedback rules. The general flow of optimization approaches that fall in this category is to evaluate a candidate solution to derive performance metrics and then apply rules, by means of model to model transformations, according to the result of the analysis in order to improve the quality of the solution.

A common way to define model to model transformations has been proposed by the OMG in the Query/View/Transformation (QVT) language. Drago et al. proposed an extension of QVT in [32] in order to support feedback rules defined on non-functional requirements. They also provide a framework, called QVTR^2 to support software engineers in making architectural choices. In their work a set of feedback rules can be defined by the user in order to guide the system in the generation of possible alternative solutions, starting from the one designed by the user. Alternative solutions are evaluated from a performance point of view and presented to the user that is in charge of selecting the best one according to its expertise. This approach requires a good deal of knowledge of performance optimization from the user since it depends on it both for the specification of the rules used to generate candidate solutions and to select the best solution among those generate by the framework.

[78] introduce Performance Booster (PB), a tool capable of deriving performance models from annotated UML diagrams in order to analyze and optimize the design of an application. The system proceeds iteratively by analyzing the performance model and modifying it by triggering feedback rules. Some feedback rules are already included in the framework and more can be specified by the user with the JESS [44] scripting language, when a rule is activated it can modify the performance model in order to produce a new solution. When no more rules are activated the optimized performance model is proposed to the user who has to trace back the changes in order to apply them in the UML model. The main drawback of this approach is the lack of an automated mechanism to trace the changes from the performance model into the UML domain, furthermore the approach procedure does not allow the user to specify constraints on the final solution, so changes in the optimized performance model might lead to solution that can not be implemented in the real system.

The framework proposed by Parsons et al. in [66] monitors a running system in order to retrieve performance information, it then uses data mining techniques to detect general, well known, design flaws called anti-patterns. The detection of anti-patterns from the monitored performance data is performed by means of a rule engine. The main limitation of this approach is its dependency on run-time data collection, moreover the need of a prototypical implementation of the system make this approach not applicable in early stages of the design.

Another approach that aims at identifying performance anti-patterns is presented in [31]. In this work the authors propose a feedback system capable of identifying well known anti-patterns and provide to the development team alternatives derived by applying known solutions. Their approach is based on model to model transformation to derive performance metrics and detect performance anti-patterns. The approach is general enough to be applied to many modeling languages but extensions to these languages have to be developed before being able to map the defined anti-pattern into the specific application model. Our works does not aim at finding performance anti-pattern at the architecture level and could be integrated with these kind of approaches but is focused on the identification of an optimized deployment of the application. As shown in Chapter 8 our approach can lead to the discovery of architectural anti-pattern as side effect.

[57] propose an approach based on Model-Driven Development in order to analyze component based software systems from a non functional point of view. Their approach comprises a trade-off analysis of competing Non-Functional Requirements (NFR) in order to find which components of the architecture represent critical points for one or more NFR, related for example to application response time or memory footprint. Component developers are then required to specify transformation rules that change the component in order to modify its non-functional properties, a set of rules is then selected according to the results of the tradeoff analysis and applied to the model. The resulting model is finally evaluated in order to check its compliance with the specified NFR. Their work is focused on trade-off analysis of competing NFR and involves manual intervention for the specification and selection of transformation rules. Finally the goal of their approach is not to derive an architecture optimized with respect to some aspect, like the cost, but to find an architecture that satisfy NFR by further refinements of an initial one.

Meta-heuristics. Meta-heurisic approaches aim at effectively exploring the space of possible solutions using high-level algorithms, often inspired by biology or other fields of study. These algorithms might operate by modifying a single solution in order to improve some characteristics or generate multiple solutions and then evaluate and compare all of them. In general all this algorithms require one or more objective functions that are used to drive the goodness of the solutions and might support constraints on the characteristics of the solution that are used to define their feasibility.

An approach that uses multiple solutions generation is presented in [55]. The authors present a toolkit, called AQOSA (Automated Quality-driven Optimization of Software Architecture), that implements evolutionary algorithms (i.e., NSGA-II and SPEA2) to solve a multi-objective optimization problem. The toolkit integrates modeling capabilities and performance analysis techniques in order to build and evaluate solutions from a QoS perspective. The optimization problem presented in their work is tailored to the context of embedded systems and aims at minimizing processor utilization, cost and data flow latency. The main drawback of this approach, and in general of many approaches based on genetic algorithms, is the fact that in order to properly converge to optimal solutions they need to evaluate a huge number of candidate solutions, called individuals. The time needed to evaluate a single solution is very critical, so only simple performance models can be adopted. A key difference of multi-objective approaches, with respect to a single solution heuristic like the one proposed in this thesis, is that they usually provide a set of non-dominated solution called Pareto front. Some techniques has been developed in order to explore the Pareto front but ultimately the choice of the final solution has to be delegated to the user. In general a Pareto front, usually generated by multiple objectives optimization approaches, are useful to explore trade-offs between multiple quality criteria. Our work, on the other hand, performs a single objective optimization reducing the cost of the application guaranteeing the required QoS level. This approach is more suited for situations in which the user has to come up with a single deployment configuration that guarantee multiple SLA constraints.

Another approach for the optimization of software architecture of embedded system is presented in [11]. The proposed framework uses the Architecture Analysis & Design Language (AADL) to model the software system. The framework is composed by a set of components offering different functionality needed for any generic optimization approach like constraint evaluation or quality evaluation. An optimization interface has been proposed in order to plug in different optimization engines. The framework supports different quality criteria. The authors used data transmission reliability and communication overhead as quality criteria and component deployment into a fixed set of containers as decision variables to validate their approach using a genetic algorithm.

A hybrid approach in which feedback rules, called tactics, used to identify performance issues and propose alternative solutions, are applied within an heuristic optimization algorithm is presented in [47]. The proposed framework, called PerOpteryx, is focused on component based architectures that are analyzed by means of a LQN performance model and optimized using a multi-objective evolutionary algorithm (i.e., NSGA-II). The algorithm has been extended in order to include the evaluation of tactics during the reproduction step in order to drive the generation of the population of candidate solutions. PerOpteryx uses the Palladio Component Model (PCM) [23] language to describe the application and applies transformations directly at this modeling level. The main variables considered by the tool to generate candidate solutions are the allocation of components to servers and the choice of the optimal number of replicas and processing power for each server. There are two main differences of our work with respect to the one implemented in PerOpteryx: the search space of our approach is shaped by the cloud environment, for this reason the speed of resource containers has to be selected according to the cloud offering. Furthermore, we focus on the deployment of the application and do not take into consideration the re-allocation of components into tiers; another difference is in the optimization approach itself. Our approach use a single solution heuristic that is improved after each iteration. The use of a single solution usually requires a less number of evaluations of the performance model that translates in a lower execution time.

In [50], the authors generalize the approach presented in PerOpteryx providing a meta-model for the definition of Generic Degree of Freedom (GDoF). This meta-model can be used to specify Degree of Freedom Instances (DOFIs) for specific model elements. The use of this approach allows a generic definition of the design space of possible configurations for the system that can be explored by means of different optimization techniques.

In [62] PerOpteryx is further evolved in order to take into consideration quality constraints in the generation of candidates solutions. This extension integrates the Quality of service Modeling Language (QML) with the PCM and allows software architects to specify at the same time objective functions and constraints on non-functional characteristics of the application. The use of quality constraints in the generation of candidate solutions by the genetic algorithm implemented by PerOpteryx allows to focus the search on promising areas of the design space and reduces the overall time required for the optimization of about 35%.

Koziolek et al. [48] proceed in the direction of [47] and address the problem of deriving deployment decisions using an approach similar to the one presented in this work. The authors make use of an analytic optimization problem derived by a relaxation of the entire problem to derive a promising initial population for an evolutionary algorithm. In this work they deal with a three objective optimization problem trying to minimize cost and response time, while maximizing application availability. This work has many points in common with our approach, especially in the use of the same modelling tool, the PCM and in the use of a hybrid analytic and heuristic model. The main differences are in the adoption of a genetic algorithm as heuristic optimization approach and in the constraints posed by the cloud infrastructure to our optimization.

[28] apply genetic algorithms to perform service composition. Their approach allow to evaluate a composition of services in terms of QoS characteristics like response time, reliability or cost and then optimize one or more of these dimensions while fulfilling constraints on others. In the example reported in their work the system was used to optimize a function of application response time and cost. Their work is focused on the realm of service composition and suppose that services expose information about their QoS.

The authors in [65] developed an efficient tabu search heuristic to solve the redundancy allocation problem for embedded system design. Their approach is focused on the solution of a Redundancy Allocation Problem (RAP) of series-parallel multistate systems. In such systems application components are connected in series and contain multiple elements connected in parallel. Each element is characterized by a cost and an availability value. Replicating components can increase system availability but also increases the total cost. The heuristic algorithm search for the solution that guarantees some user defined availability level and minimizes the cost. This paper presents an heuristic approach similar to the one we propose in this thesis and applies it to a different domain. The main differences from the heuristic search point of view is that the problem analyzed in this work can be separated into a set of disjoint subsets in which tabu search can be applied independently.

[37] presented a combined meta-heuristic-simulation approach to solve the prob-

lem of migrating existing enterprise software to cloud platforms considering a combination of a specific cloud environment, deployment architecture, and runtime reconfiguration rules. The design space is explored by means of a genetic algorithm while a simulator is charged with the solution performance evaluation. This approach aims at easing the migration of legacy enterprise systems to the cloud, while our objective is to help QoS engineers to design new cloud-ready applications. Another key difference is that we explicitly consider both architectural and QoS constraints during the search process, giving the possibility to express QoS constraints for many criteria in terms of both average values and percentiles. Finally, our approach takes into consideration deployment scenarios of one day divided into 24 hour periods leading to multiple final solutions each one tailored to one hour of the day.

In [26] a multi start hill climbing approach is used to optimize component deployments on a set of resources. This work takes into consideration both communication and processing delays to evaluate the response time of the application. The authors use the UML SPT profile to model the application from a performance point of view, Finite State Processes (FSPs) are then built from the initial model and analyzed via simulation to derive performance related metrics, like object utilization or thread instances. The results of the simulation are then used together with a model of resources to build a Software Execution Model that is used to evaluate solutions generated by the heuristic search. During the search evaluation of performance characteristics of the candidate deployment is done by means of a performance function proposed by the authors that takes into consideration networking and processing delays caused by co-location of components and synchronous and asynchronous calls among components. The main difference between this work and the one presented in this thesis is that in our work the co-location of components in application tiers is delegated to the software architects and the optimization heuristic focus on the assignment of computational resources, or PaaS services, to the application tiers. The approach presented in this paper, on the other hand, assumes a fixed set of resources to host application tiers and optimizes the assignment of component to tiers.

In [60] the authors propose an automatic optimization process for the exploration of the adaptation space of a service-oriented application. The goal of the exploration is to find the cheapest sequence of adaptation actions that allow a service-oriented application to adapt to changes in the runtime environment or new requirements. Adaptation actions include adding or removing components, changing their implementation, the interaction style (sync or async) or component re-deployment. The proposed optimization strategy is an iterative process composed by two main phases: the *new candidate generation* and the *functional and quality analysis*. In the first phase a tabu search heuristic is used to generate a population of new candidate adaptations, in the same phase another method based on the application of best practice design pattern is used to generate more candidates. In the latter phase both functional and non-functional requirements are evaluated using a variety of approaches, like scenario-based simulation, model-based testing and queuing network simulation, to analyze the result of the adaptation strategy. Since the formulation of the problem deals with multiple objectives, the outcome of the optimization process is a set of Pareto-optimal solutions. This approach has been particularly tailored for runtime adaptation of service-oriented application when new requirements are specified.

Generic Design Space Exploration (GSDE) Generic Design Space Exploration approaches apply feedback rules in a Constraint Satisfaction Problem (CSP) to shape the space of possible configurations.

An interesting approach is presented in the DESERT framework [33, 61]. DESERT explores the design space alternatives by organizing them in a tree of system variants, boolean constraints are aplied to the tree in order to prune non-feasible solutions; the framework does not specifically target QoS but might be configured to do so. In the original formulation [61], the approach required the intervention of the human in the exploration loop in order to prune non-viable solutions, this operation was performed by specifying additional constraints that the solutions had to verify. The most recent version of the framework, DESERT-FD [33] automates the generation of these constraints removing the need for human intervention.

A more general framework is the one presented by Saxena et al. in [70], called Generic Design Space Exploration. GDSE allows the definition and the solution of domain specific design space exploration problems, it provide a language to express constraints and enable the support for multiple solvers to generate solutions.

A different approach is Formula, presented in [45] by Jackson et al. Their approach consist in specifying the problem as a satisfiability problem and use the Z3 Satisfiability Modulo Theory solver to derive solutions compliant with the design specification.

To do so, Formula makes use of logic programs to specify non-functional requirements and transform these constraints along with the application models and meta-models into first-order logic relations. This work does not use performance prediction models but assume the performance data to be embedded in the model.

2.3 Deployment Selection Approach Classification

In this section we position our optimization approach, presented in Chapter 7, according to the classification of meta-heuristic optimization algorithms presented in [74].

Our optimization procedure can be briefly described as a Tabu Search (TS), in which the intensification action is implemented by the *ScaleLS* function, followed by the *OptimizeWorkload* in the multicloud scenario, and the diversification action is implemented by the *TSMove* procedure (see Chapter 7).

According to the classification of hybrid meta-heuristics presented in [74], the optimization approach proposed in this thesis follows in the *Low-level Relay Hybrid* (LRH) category. It is a *Hybrid* methauristic since the *TSMove* procedure implements a Multi-start Local Search (MLS) that uses a *roulette wheel* (or a fitness proportionate) selection criteria [56], while the *ScaleLS* implements an Iterated Local Search (ILS) [72] that is applied independently on all the 24 hours slots of the solution. An extensive survey on hybrid meta-heuristics in combinatorial optimization can be found in [74] and more recently in [24], whilst a complete taxonomy is presented in [73]. It can be classified as a *Low-level* since combines two single solution based meta-heuristics (or S-meta-heuristics) in such a way that the lowest level local search constitute the intensification action of the upper level search procedure. Finally, it falls in the *Relay* category since the low level heuristic (the ILS) uses as starting point the outcome of the upper level procedure (the MLS), the execution of which is driven by the outcome of the low level procedure.

The algorithm is also bi-level because it tackles separately each optimization level of the problem (see Chapter 6), the upper level, the selection of the VM type, is addressed by the *TSMove* heuristic and the lower level is tackled by the *ScaleLS* procedure both presented in Chapter 7.

Chapter 3

The Model Driven Approach

"*The mere formulation of a problem is far more often essential than its solution, which may be merely a matter of mathematical or experimental skill. To raise new questions, new possibilities, to regard old problems from a new angle requires creative imagination and marks real advances in science*"

A. Einstein

3.1 Introduction to Model Driven Engineering

This quote from Albert Einstein reflects the same basic principle of Model Driven Engineering (MDE). The main idea behind MDE is to focus on the representation of the problem, called *Model*, and use this representation as main driver for all the processes needed to analyze and eventually solve it. Focusing on the representation of the problem from the very beginning allows us to gather more insights on the nature of the problem itself and enriches our knowledge. Building different representations, or different Models, of the object we are studying allows us "*to regard old problems from a new angle*."

A very important tool of MDE is the use of Domain-Specific Modeling Languages (DSMLs). These languages offers the flexibility required to address problems in very different domains by providing a limited set of concepts with well defined relationships and semantic. Adopting the use of a DSML, and in general of different modeling languages, eases communications between different members of the development team and domain experts. It also allows to use well established tools and techniques to analyze the problem from different point of views.

In this thesis we adopted the MDE approach in several aspects. First we relay on existing modeling technologies like the Palladio Component Model [23] or Creator4Cloud [13] for the modeling interactions with the software architect. Relying on well established modeling techniques and tools removes the burden of learning a new modeling language and allows the architect to make use of other functionality offered by these frameworks. We then provided a meta-model, describing the cloud infrastructure in Section 3.4, to extend the modeling capabilities of the PCM and allow the software architect to specify deployment configurations.

In Section 5.2 we change the point of view on the problem by formalizing it in mathematical terms. The architecture of the application and the constraints introduced by the architect have been mapped to equations that define the shape of the search space, while possible deployment alternatives have been represented with variables that have to be assigned to provide a complete solution. The use of this formalism allowed us to better comprehend the nature of the problem and design an effective optimization strategy.

Another view on the model is given by its formulation as a Layered Queuing Network (LQN) [35]. This formulation is focused on performance estimation, the main entities of this modeling paradigm are processors that represent resources used to process incoming requests. Tools like LINE [67] or LQNS [35] can be used to analyze this model and derive estimate of performance indicators, like application response time or resource utilization.

In order to build a LQN model representing the application, we used another key feature of MDE, the *Model Transformation*. As we have seen using different modeling languages and tools allows us to see a problem under different lights, unfortunately working simultaneously with multiple models representing the same object is challenging. In the MDE approach, models evolve very rapidly to adapt to changes in the requirements or, as in our case, to inspect possible variations of the system. Keeping track of these changes and updating all the models is a daunting task that requires automation. Model to model transformation techniques have been widely adopted to ease this task. In our work, performance models are derived from the higher level models, designed by the architect, by means of a transformation technique offered by Palladio

Bench. The mathematical formulation of the problem can also be updated by deriving values for many parameters. Our tool then operates directly on the performance model and updates automatically the PCM representation.

The integration of our approach for performance prediction and deployment optimization into state of the art modeling platforms allows the software architect to exploits the MDE approach in the development of cloud applications. It allows to take into consideration QoS aspects in early stages of the development keeping the focus on the model of the application.

In the following sections we will introduce the modeling paradigm used to model an application with our approach. We will do so using the Creator4Cloud IDE since it better supports the modeling extensions developed to address the cloud environment. We will use an industrial case study called *Constellation* developed by SOFTEAM[1] to clarify the modeling concepts here and in the rest of this thesis.

One of the products of SOFTEAM is a modeling environment called Modelio [71]. Modelio supports many modeling standards and provides a central IDE that allows to combine different modeling languages in the same model. Modelio uses the UML2.x standard as main modeling technology and represents different languages as UML profiles, it provides a comprehensive tool for MDE. One of the main feature of Modelio is its ability to separate concerns among stakeholder providing support for each of them, it offers high level modeling capabilities as well as low level support of complex code bases. Modelio is distributed as a desktop tool that developers install on their own machines and uses a back-end to share models among multiple user. Both Modelio clients and the back-end have to be managed by SOFTEAM clients.

In order to ease the setup and maintenance of Modelio from the customer perspective and to include more functionalities, SOFTEAM decided to move to a Software as a Service (SaaS) solution, offering Modelio as a web based tool. SOFTEAM aims to ease the burden of their clients from managing their systems by migrating the tool and its backend to a public cloud infrastructure. The new cloud based extension of Modelio has been named *Constellation*.

During the development of *Constellation* and initial SVN based architecture has been proposed. This architecture has been modeled and analyzed by means of SPACE-4Cloud in order to evaluate its expected quality. Section 3.2 describes this architecture

[1]http://softeam.com/

and introduces the modeling notation used in our work.

We will see in the remainder of this Chapter how the *Constellation* architecture has been modeled, in Chapter 8 we will see the results of the QoS evaluation of this architecture and how the discoveries of this analysis based on the tool developed in this thesis lead to a change in the design of the application.

SOFTEAM had to deal with many challenges in their adoption of a cloud infrastructure. The main requirements for a supporting tool that help SOFTEAM move their product to the cloud were: The ability to study alternative selections of cloud services taking into account QoS requirements and specific needs of customers; The ability to keep under control the cost of the cloud infrastructure to achieve return on investment; The ability to exploit the elasticity of the cloud environment to cope with changes in the number of users maintaining the required QoS level specified by a set of constraints. We will see how the work developed in this thesis meets these needs.

The remainder of this chapter is structured as follows. Section 3.2 introduces the modeling paradigm used by our tool by means of an industrial case study called *Constellation*. Section 3.3 expands the models introduced by adding elements that are specific to the cloud environment. Section 3.4 provides some details on the meta-model developed in this thesis to describe the characteristics of the cloud environment.

3.2 Modeling the Application

The modeling approach used in the following adopts the guidelines of the Model Driven Architecture (MDA)[1] specified by the Object Management Group (OMG). It is composed by three levels of abstraction, the highest one being the *Cloud Computation Independent Model* (CCIM). Models built at this level are used to represent the application architecture, its behavior and the interactions with the appication end user. These models carry no information about the platform used to host application components and keep the focus on business related aspects keeping the model of the application free from details related to the infrastructure. We then add to these models some information related to the execution platform and the technology used to host some application components by introducing some elements that are peculiar of the cloud

[1]http://www.omg.org/mda/

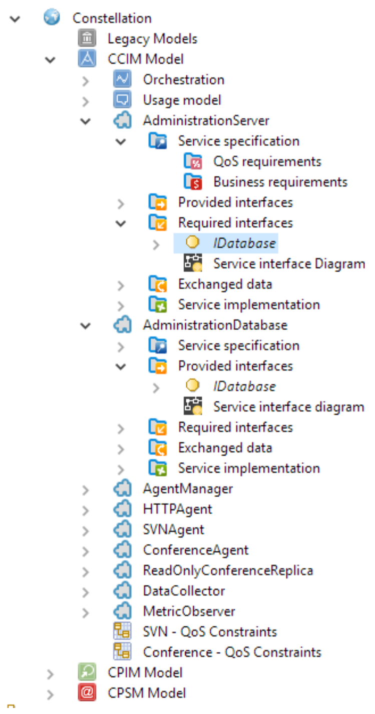
environment but still not tied to a specific cloud provider, this modeling level is called *Cloud Provider Independent Model* (CPIM). Finally, we add all the details related to the services offerd by a particular provider binding the general platform selection to the concrete service offering deriving a *Cloud Provider Specific Model* (CPSM).

As introduced in Section 3.1, we use Creator4Clouds to build the model of the application and the deployment. Creator4Clouds allows architects to build multiple views of the application in order to specify different aspects of the application. It has been built in order to keep the distinction among the cloud abstraction levels (CCIM, CPIM and CPSM) clear and allow refinements of views according to the specific abstraction level. The specification of a new application starts with the definition of CCIM level views. In particular the software architect specifies a list of components composing the application, as shown in Figure 3.1. A component is an element of the application that provides a set of functionalities. Each component allows the specification of Provided Interfaces to define which functionality is offered to other components or to the application end user. Required Interfaces allow to specify functional dependencies of a component. In the example of Figure 3.1, we see that the Administration Service required the *IDatabase* interface to provide its functionality. Such an interface is offered by the Administration Database component.

This view allows the architect to specify functional offerings and requirements of each component via interfaces and non functional requirements via the Service Specification objects. Such objects are divided into QoS and Business requirements. These requirements predicate of quality aspects on functionalities offered by the component, like the average time required to execute such a functionality. More details on requirement specification are provided in Section 3.5.

The CCIM abstraction level provides two main views of the model. The Service Assembly view allows to define interconnection among components in order to solve functional requirements introduced by means or Required Interfaces in the component specification. The Orchestration view allows the specification of the component internal behavior by means of a call chain of functionalists implemented internally by the component or by other components of the application. More information on this modeling notation can be found in [13].

Figure 3.2 shows a possible architecture for the *Constellation* application. This architecture is characterized by five components, enclosed in the blue square that rep-

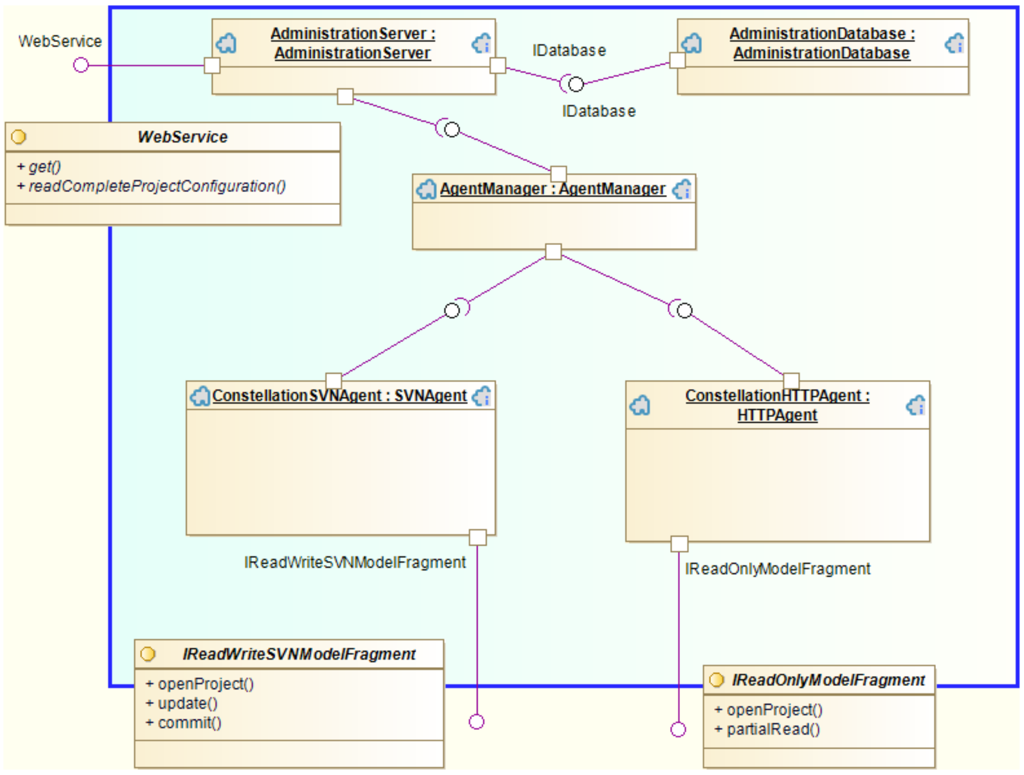


```
Constellation
    Legacy Models
    CCIM Model
        Orchestration
        Usage model
        AdministrationServer
            Service specification
                QoS requirements
                Business requirements
            Provided interfaces
            Required interfaces
                IDatabase
                Service interface Diagram
            Exchanged data
            Service implementation
        AdministrationDatabase
            Service specification
            Provided interfaces
                IDatabase
                Service interface diagram
            Required interfaces
            Exchanged data
            Service implementation
        AgentManager
        HTTPAgent
        SVNAgent
        ConferenceAgent
        ReadOnlyConferenceReplica
        DataCollector
        MetricObserver
        SVN - QoS Constraints
        Conference - QoS Constraints
    CPIM Model
    CPSM Model
```

Figure 3.1: An instance of a CCIM model

resent the system boundaries. Some of these components expose interfaces directly to the final user while other components are used only internally. In particular the *AdministrationServer* exposes the *WebService* interface that provides a way to retrieve the available projects and read their configuration. This component uses the *AdministrationDatabase*, internal to the system and not exposed to the end user, to store the access permission policy and define which projects a user can see. Interactions between the *AdministrationServer* and the *AdministrationDatabase* are defined by the *IDatabase* interface. The main interface that allows model manipulation is *IReadWriteSVNModelFragment* which is implemented by the *ConstellationSVNAgent*. This interface allows users to open, update and modify a model. As suggested by the component name, the *ConstellationSVNAgent* uses SVN[1] to provide versioning, sharing and conflict detection in order to allow multiple users to work simultaneously on the same model. To offload the previous component from some of the load the *ConstellationHTTPAgent* provides read only access to the models. Users can open their projects

[1]http://subversion.apache.org/



WebService
AdministrationServer : AdministrationServer
IDatabase
AdministrationDatabase : AdministrationDatabase
IDatabase
WebService
+ get()
+ readCompleteProjectConfiguration()
AgentManager : AgentManager
ConstellationSVNAgent : SVNAgent
ConstellationHTTPAgent : HTTPAgent
IReadWriteSVNModelFragment
IReadOnlyModelFragment
IReadWriteSVNModelFragment
+ openProject()
+ update()
+ commit()
IReadOnlyModelFragment
+ openProject()
+ partialRead()

Figure 3.2: Service Assembly diagram of the SVN Architecture of Constellation

and check for updates by interacting with the *IReadOnlyModelFragment* interface. Finally, the *AgentManager* orchestrates the *SVNAgent* and the *ConstellationHTTPAgent*.

This figure represents the core of the application, defines the system boundaries and provides information about possible interactions between the user and the system as well as some of the interactions internal to the system itself. All the functionality depicted in this figure can be annotated with non functional properties like the expected demand that the execution of a functionality will generate on the system. It is important to notice that, this model is agnostic with respect to the technology used to host each component and can be re-used across many different platforms. We will see later how we can add details to this model to specify, for example, which technology is used for the *AdministrationDatabase* and if this service is managed by the cloud provider, in a PaaS fashion, or installed on a VM managed by the system administrator.

In order to further specify the internal behavior of the system we use *Orchestration*

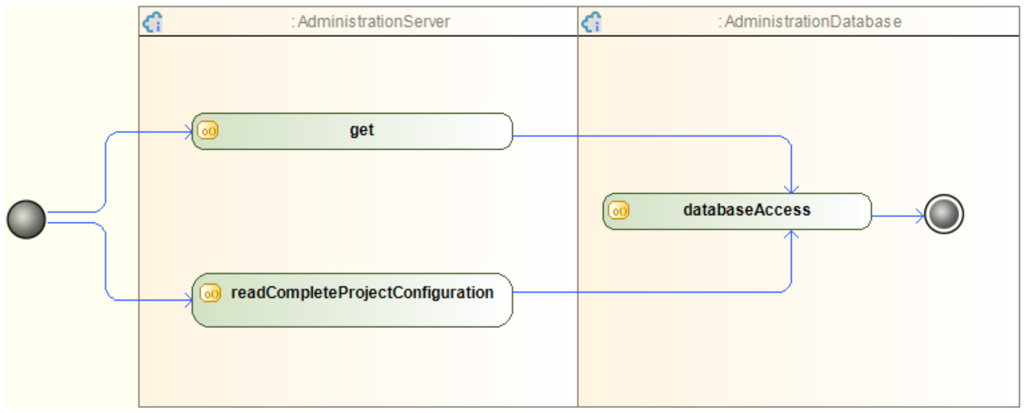


Figure 3.3: Orchestration model

Models. An example of an Orchestration diagram is shown in Figure 3.3, it shows the interactions between the *Administration Server*, whose interface is directly exposed to the final user, and the *Administration Database*. This diagram shows that both the *get* functionality, used to provide a list of projects, and the *readCompleteProjectConfiguration*, used to load the configuration of a project, require a *databaseAccess*. The details of the access, namely the query, is hidden from the model and is embedded in the *Administration Server* implementation. The use of separate Orchestration diagrams allows the software architect to focus on modeling the interactions between a subset of all of the components of the application and helps maintaining a logical separation between different parts of the model. The diagram in Figure 3.3 shows only the subset of the application related to the administration aspect and hides all the interactions with other, logically separated, parts of the application. In order to get an overall view of the interactions between the components the architect can refer to Figure 3.2. This modeling approach helps to achieve loose coupling between parts of the application architecture that are related to different logical aspects and high cohesion among closely related components.

A very important factor in the analysis of the application performance is represented by the amount of work the system is subject to. In the context of our application the workload is composed by requests generated by users that exploit Constellation for their modelling activities. As shown in Figure 3.2, users can interact with many different interfaces exposed by the system in order to retrieve the list of available projects, open a new project or update a project on which they are already working and committing changes to the shared repository. To better understand how the system will be used we make use of the model reported in Figure 3.4.

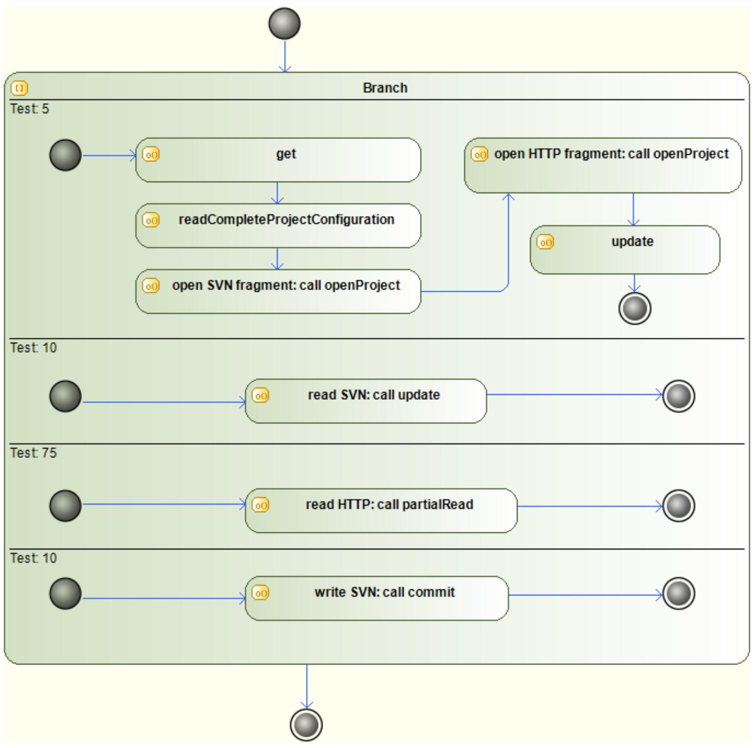
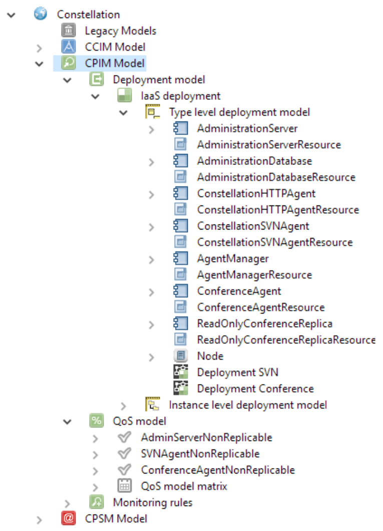

Branch
Test: 5
get
readCompleteProjectConfiguration
open SVN fragment: call openProject
open HTTP fragment: call openProject
update
Test: 10
read SVN: call update
Test: 75
read HTTP: call partialRead
Test: 10
write SVN: call commit

Figure 3.4: Usage Model

This model shows the typical behavior of a user that interacts with the system. In this picture we can see that four main interactions, represented as branches, are performed. The first interaction is a sequence of actions that the user performs to get the list of available projects, load the configuration of a specific project, open it from the SVN server and from the HTTP replica and finally update the local copy of the project. This sequence of interactions might be hidden in a single automated function from the client or directly performed by the user. This interaction models a new user that decides to work on a project, this behavior is expected to happen only 5% of the times as shown by the value of the *Test* attribute in the top left part of this interaction branch.

Other interactions are much simpler and represent the common behavior of users that already have a local copy of the model on which they are working. In this scenario users retrieve frequent updates of the model from the HTTP server and only rarely perform an update of the entire model directly from the SVN. Finally users commit their work to the SVN server. The interaction scenarios and the associated probabilities shown in Figure 3.4 have been gathered by mining the logs of the current version of Modelio.

The architecture model of Figure 3.2 can be enriched by adding details about the technology used to host application component in order to derive a deployment model at the *Cloud Provider Independent Model* (CPIM) level. As shown by Figure 3.5, this abstraction level allows the initial specification of deployment diagrams. These diagrams are used to enrich the model with information required to deploy the system such as dependencies on particular technology. The CPIM level allows also to refine the QoS model specified at the CCIM level by adding infrastructural requirements like the minimum or maximum number ore replicas of the resources used to host a component.



```
Constellation
  Legacy Models
  CCIM Model
  CPIM Model
    Deployment model
      IaaS deployment
        Type level deployment model
          AdministrationServer
          AdministrationServerResource
          AdministrationDatabase
          AdministrationDatabaseResource
          ConstellationHTTPAgent
          ConstellationHTTPAgentResource
          ConstellationSVNAgent
          ConstellationSVNAgentResource
          AgentManager
          AgentManagerResource
          ConferenceAgent
          ConferenceAgentResource
          ReadOnlyConferenceReplica
          ReadOnlyConferenceReplicaResource
          Node
          Deployment SVN
          Deployment Conference
        Instance level deployment model
    QoS model
      AdminServerNonReplicable
      SVNAgentNonReplicable
      ConferenceAgentNonReplicable
      QoS model matrix
    Monitoring rules
  CPSM Model
```

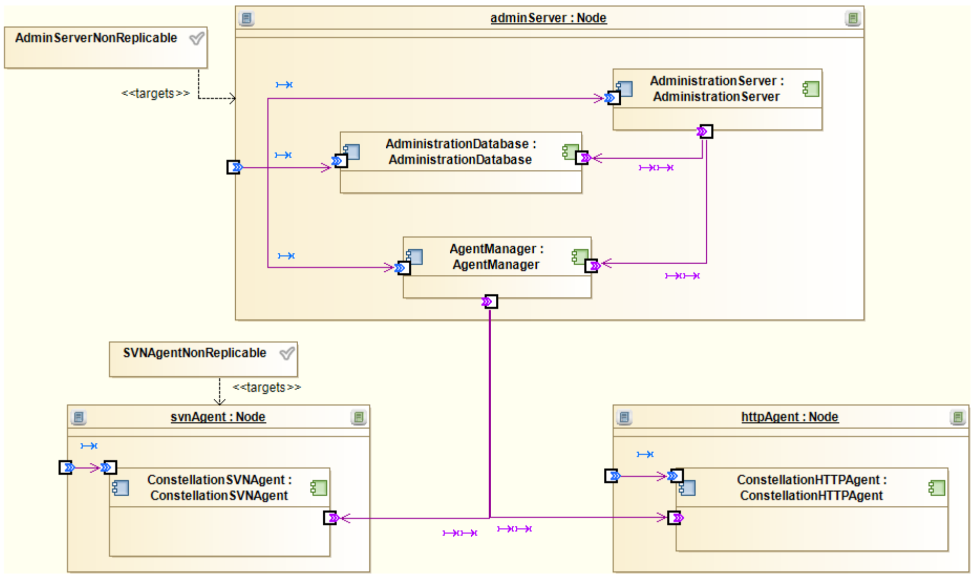
Figure 3.5: An instance of a CPIM model

An example of such a model for the deployment of the SVN architecture of our case study is shown in Figure 3.6. Here components introduced in the architectural model are grouped into tiers that represent resources used to host the components. In the deployment scenario of Figure 3.6 all the components related to the administration and management aspects of the application are hosted on the same *adminServer* tier while the SVN and HTTP agents are hosted each one on a dedicated tier. The diagram also contains two constraints that target the *adminServer* and *svnAgent* tiers. These constraints predicate on the maximum number of replicas that these tiers can support. In this case the requirements constraint both tiers to have only a single replica. These kind of constraints are due to the technology chosen to implement some of the components. SVN, for example, is a centralized monolithic technology that assumes the use of a single central server. We call this type of limitations *Architectural* constraints. These constraints do not predicate on quality aspects of the application as seen by the final user, as opposite to the response time constraint described above, but define characteristics that a deployment solution must fulfill that are derived by architectural choices. Some of the concepts presented in this meta-models have been integrated in MODACloudML [13] .

This diagram adds some more information on how the application will be deployed on the target infrastructure, as an example Figure 3.6 specifies that all the three tiers composing the application are hosted on groups of VMs, called *Nodes*, but still does not make any commitment to a choice of services of a particular cloud provider. Models at this level of abstraction are called *Cloud Provider Independent Models* (CPIMs).

3.3 Modeling the Cloud

In the previous section we have modeled aspects of the application that do not relate to any particular cloud technology or provider. In order to derive a final design of the application that can actually be deployed in a cloud environment, either manually or by means of some automated tool, we need to enrich the previous models with cloud specific information. The approach followed up to now is a common practice in the context of Model Driven Engineering and many tools support modeling applications in similar ways. In the next Section we introduce some modeling extensions that we have developed in order to describe the cloud environment. These extensions have been built in the form of a meta-model whose elements can be used along with the models previously presented to enrich them with cloud specific concepts. The main goal of this meta-model is to describe the core relations among cloud services from the functional and non-functional point of view, allowing the construction of a model repository from



AdminServerNonReplicable
<<targets>>
adminServer : Node
AdministrationServer : AdministrationServer
AdministrationDatabase : AdministrationDatabase
AgentManager : AgentManager
SVNAgentNonReplicable
<<targets>>
svnAgent : Node
ConstellationSVNAgent : ConstellationSVNAgent
httpAgent : Node
ConstellationHTTPAgent : ConstellationHTTPAgent

Figure 3.6: Deployment Model at CPIM level

which the software architect can pick the required service. The repository is also used during the deployment optimization process to evaluate alternative services selections. The meta-model has been used to create a model repository that has been integrated in the tool used to develop this case study allowing the software architect to specify the cloud specific entities as UML elements.

Figure 3.7 shows a deployment configuration in which each tier, modeled in Figure 3.6, has been assigned to a VM hosted on Amazon EC2. In particular for the *adminServer* an instance of size *c3.large* has been selected; for the *svnAgent*, the component that hosts most of the computation and whose number of replicas is limited to 1, an instance of size *c3.4xlarge* has been chosen; finally the lightweight and scalable *httpAgent* has been assigned to instances of size *m3.xlarge*.

The selection of resources provided in this diagram along with the performance characteristics of these resources contained in the resource model allows us to derive a complete performance model that can be used to analyze the QoS of the application.

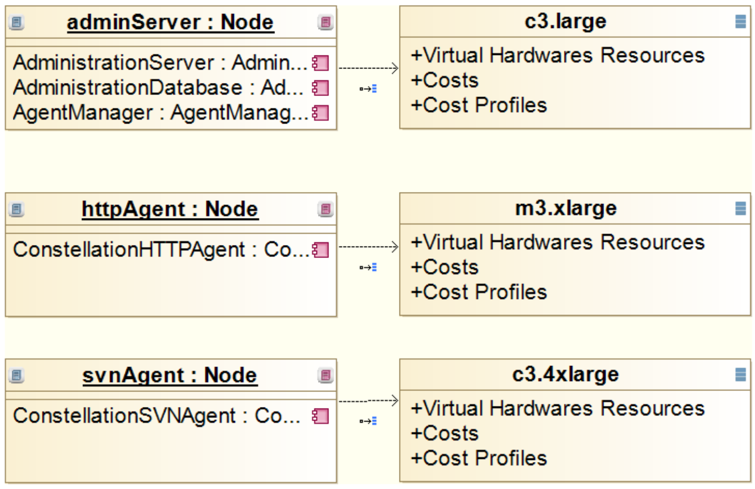

adminServer : Node
AdministrationServer : Admin...
AdministrationDatabase : Ad...
AgentManager : AgentManag...
c3.large
+Virtual Hardwares Resources
+Costs
+Cost Profiles
httpAgent : Node
ConstellationHTTPAgent : Co...
m3.xlarge
+Virtual Hardwares Resources
+Costs
+Cost Profiles
svnAgent : Node
ConstellationSVNAgent : Co...
c3.4xlarge
+Virtual Hardwares Resources
+Costs
+Cost Profiles

Figure 3.7: Deployment Model at CPSM level

3.4 Modelling Cloud Specific Concepts

The models introduced in previous sections of this chapter allow to describe the architecture of an application by focusing on different aspects. Here we present a meta-model that has been introduced to further enrich application models, in particular the deployment model, by including cloud specific concepts. This meta-model is intended to bridge the gap between the application specification and the cloud infrastructure by allowing users to select generic cloud services or specific cloud resources to host their application tiers. A generic selection of a service, e.g., specifying a database technology, can be used to guide further refining of the model and drive the implementation of the application. A specific selection of a cloud resource, e.g., using a certain type of VM offered by a particular provider, allows to estimate the performance of the application and its cost. This model is used to identify the relations between the different cloud services offered by providers and provide information about the performance and cost characterization of such services.

Following the same methodology used in previous sections, we designed the model in order to provide different levels of abstraction, the highest one being the *Cloud Provider Independent Model* or CPIM. This level of abstraction is more specific than the CCIM used in Section 3.2 since it allows to specify concepts specific to the cloud domain like VMs or PaaS services but still hides the details of a specific cloud provider.

3.4.1 Cloud Provider Independent Model

Figure 3.8 shows the main elements of the meta-model and their relations. This model has three main components: the *CloudProvider*, the *CloudService* and the *CloudElement*. The *CloudProvider* represents the entity, usually a company, that provides some *CloudService*. Many modern cloud providers, like Amazon, Google or Microsoft offer many different kind of cloud services, a common way to group together these kind of servicing is by distinguishing services that offers computational or storage infrastructure, *IaaS-Service*s; managed execution platforms for user code, *PaaS-Service*s and complete software solutions, *SaaS-Service*s. These services are usually composed by may different elements that cooperate together in the cloud provider environment, this scenario has been modeled by means of a general *CloudElement* component that is further specialized in *CloudIaaSResource*, *CloudPlatformResource* or *CloudSoftwareResource*. A set of *runsOn* relations between the CloudSoftwareResoruce, the CloudPlatformResource and the CloudInfrastructureResource models internal dependencies between the services offered by cloud providers. This dependency is not always visible to the final user but some providers give details about the infrastructure used internally to provide platform or software services for billing purposes.

CloudElements represent resources that the user can acquire from cloud providers at a cost. In order to model different billing strategies introduced by many cloud providers we introduced a *CostProfile* element related to each CloudElement. Each CostProfile is composed by a set of *Cost* elements that represent the basic billing unit applied to the specific CloudElement. Different CloudElements can be priced according to the type of service they provide, VMs for example are priced per hour of usage, Databases are priced by number of transactions and Network resources are priced by GB of data transferred. Using multiple Cost elements each with its own *unit* type associated to the same *CostProfile* allows to take into consideration this variety of billing mechanisms. The Cost might have a validity period in order to model resources whose cost changes with the time of the day, like Amazon Spot Instances[1]. Upper and lower bounds can be used to define specific ranges over the specified unit in order to rep-

[1]https://aws.amazon.com/it/ec2/spot/

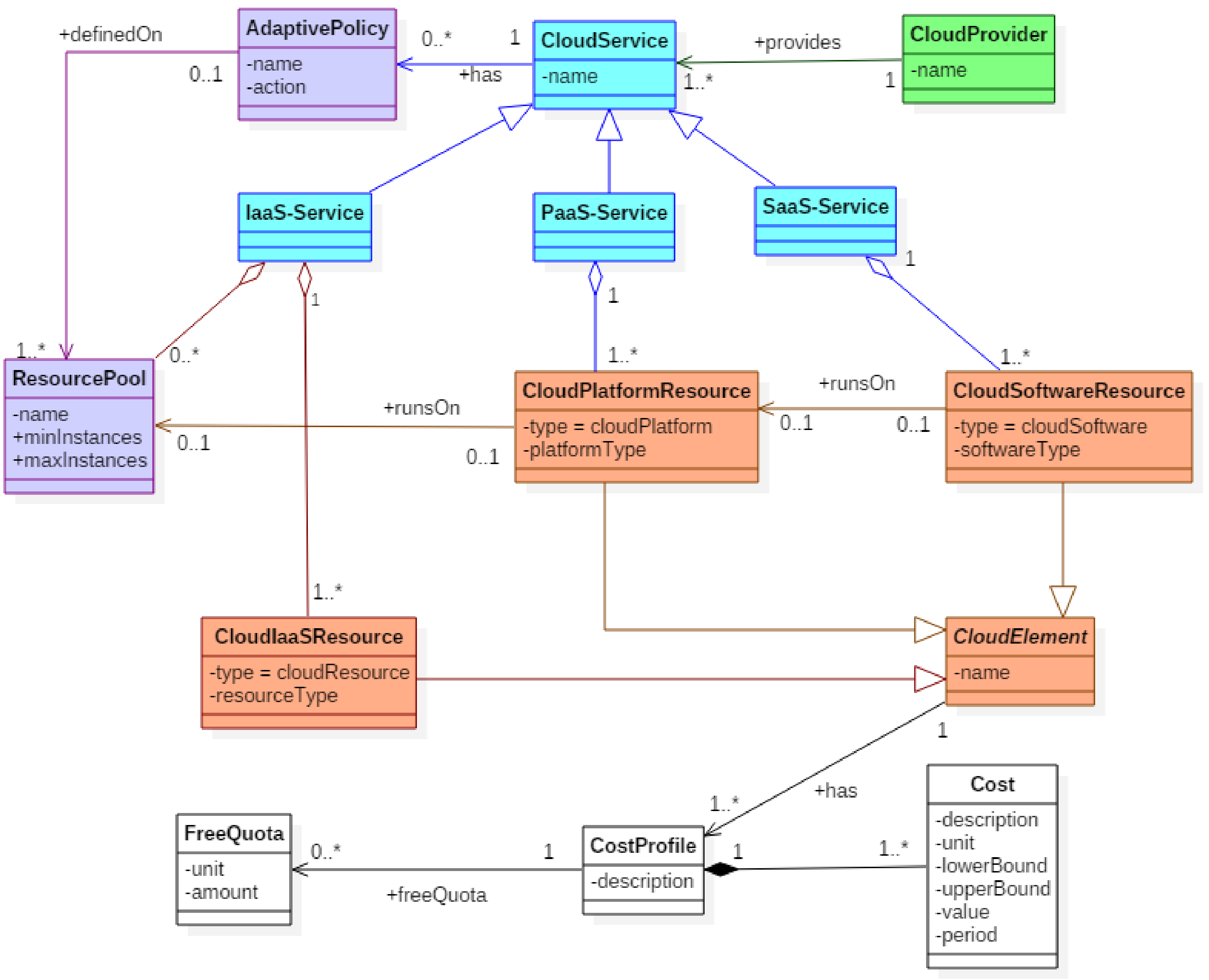


Figure 3.8: Cloud Meta-Model - General Concepts.

lowerBound	**upperBound**	**Cost Type For X Units**
null	null	Fixed cost, no range
A	null	Cost relative to the range $X \geq A$
null	B	Cost relative to the range $X < B$
A	B	Cost relative to the range $A \leq X < B$

Table 3.1: Cloud Meta-Model - Cost Ranges

resent discounts applied when utilizing a large amount of resources (see Table 3.1). The FreeQuota element models a common practice of cloud providers that provide a limited amount of resources free of charges to their users.

PaaS-Services provide complete execution platforms for many different languages and technologies. The management of such platforms and of the QoS provided by them is usually delegated to the cloud provider which implements some *AdaptivePolicy* to react to dynamic behavior of the environment like variable number of user. IaaS-Services allow users to write their own AdaptivePolicies and delegate to the cloud provider their actuation. These policies are defined on sets of IaaS-Services used to host the application code. Common actions implemented in AdaptivePolicies operate either on the type of resources used to host the application or on their type. A common scenario related to IaaS-Services is the scaling of VMs. In an homogeneous ResourcePool of VMs a scaling action might increase (or decrease) the amount of instances performing *horizontal scaling*. Another common adaptation action is to change the type of VMs used by adopting more (or less) powerful instance types, this action is called *vertical scaling*.

Figure 3.9 further expands the CloudElement concepts by adding some information about the physical location of the resources offered by the CloudProvider. In particular each CloudElement is hosted in a particular *Location*. Many cloud providers group resources in a particular area of the world into a *Region*, inside the region multiple datacenters, or multiple areas of a single datacenter are managed as *SubRegion*. Being aware of the location in which key components have been deployed is critical for many applications. In sensitive domains like health care regulations do not permit sensitive data to leave a certain geographical zone so the user must be able to take this constraint

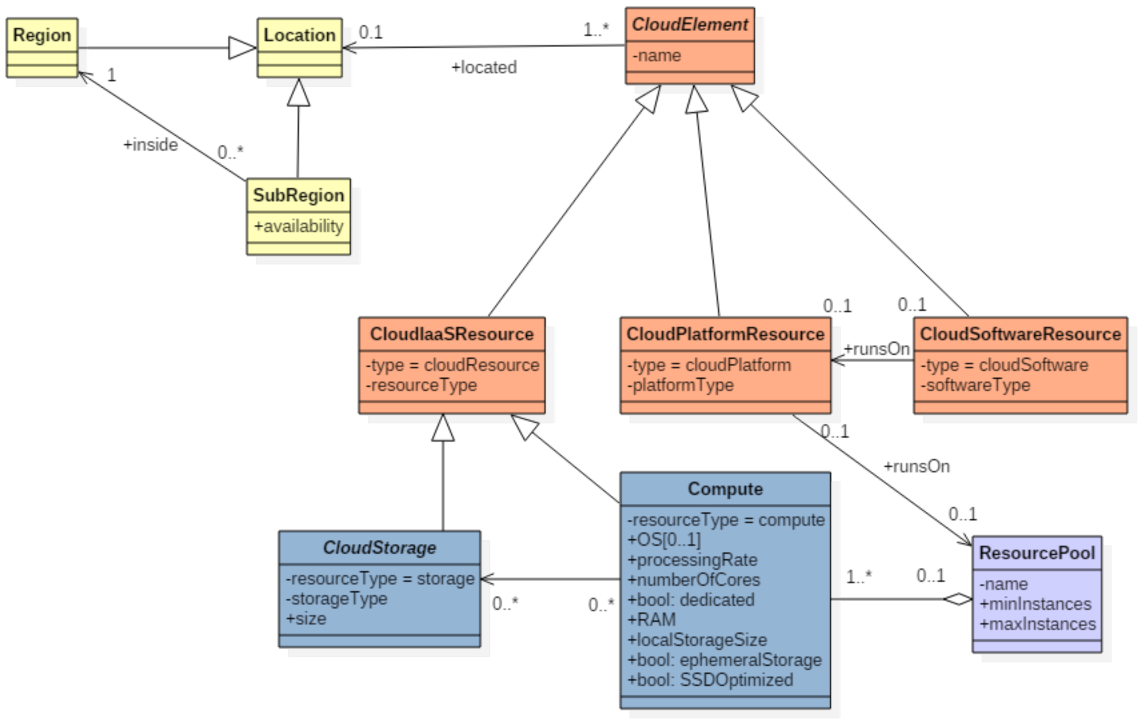


Figure 3.9: Cloud Meta-Model - Locations

into consideration when choosing which provider to use. In the context in which a low latency is a key factor, like online gaming, being aware of the proximity of the infrastructure used to host an application and the final user is also of critical importance. Finally, if availability is of critical importance replicating the application on multiple SubRegion or Regions provide an effective way to avoid main outages that could affect datacenters.

Figure 3.9 also expands the CloudIaaSResource showing two resource types. The *Compute* element represent VMs while the *CloudStorage* element represent a data storage service. A group of homogeneous VMs constitute the ResourcePool of Figure 3.8. Each VM has an internal storage used for its operating system and other CloudStorages can be attached to provide additional space for user or application data.

Figure 3.10 expands the meta-model by adding details related to the IaaS offering of cloud providers. The two basic infrastructure services offered by all the cloud providers are a *Compute* engine, composed of different type of VMs and a *Storage*

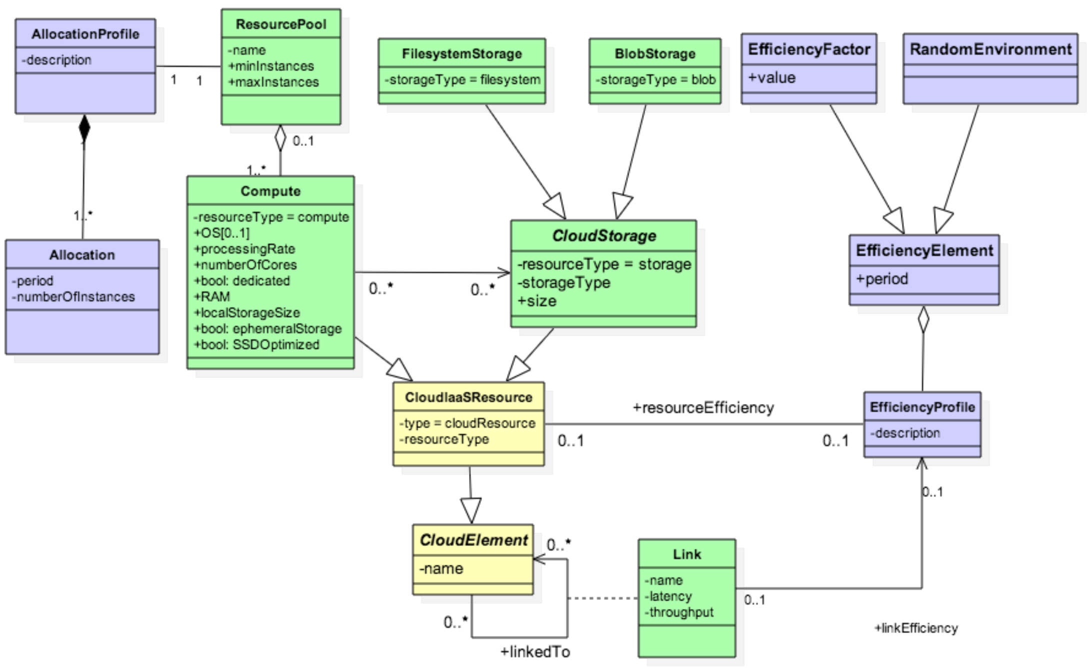


Figure 3.10: Cloud Meta-Model - IaaS Cloud Resource

service that allows to save application and user data. Storage services are usually offered as file systems attached to some VM or as Binary Large Object (Blob) storage. Different Cloud Elements communicate through links, usually *Links* that represent the networking infrastructure internal to the cloud provider. VMs are usually grouped in *ResourcePool*s, sometimes referred to as autoscaling groups. Resource Pools are used to host application tiers and provide services like load balancing across the VMs of the pool or monitoring. As shown in Figure 3.8 the actual number of instances in a pool can be controlled by applying adaptive policies. Another way to specify the amount of resources used in each pool is to perform a static partitioning of the time frame, usually one day, into discrete periods called *Allocation*s. We used 24 allocation periods of 1 hour each, and define the exact number of replicas for each of these periods. A collection of non overlapping allocations constitute an *Allocation Profile*.

Cloud Elements, like VMs or Blob Storages, represent virtual resources that are hosted on the cloud provider premise on a variety of different hardware. The internals of the cloud provider datacenter used to host the virtual infrastructure is usually not known to the user, nevertheless it is important to be aware of possible performance issues due to shared nature of the cloud infrastructure. The *Efficiency Element* captures this behavior by providing two ways of considering performance variability. The most simple way is to use an *Efficiency Factor* that is used during analysis to reduce the performance of the affected cloud resource from their nominal value. The more advanced way is to use a *Random Environment*, described in [29], that allows to take into consideration more complex scenarios like VM failure and recovery.

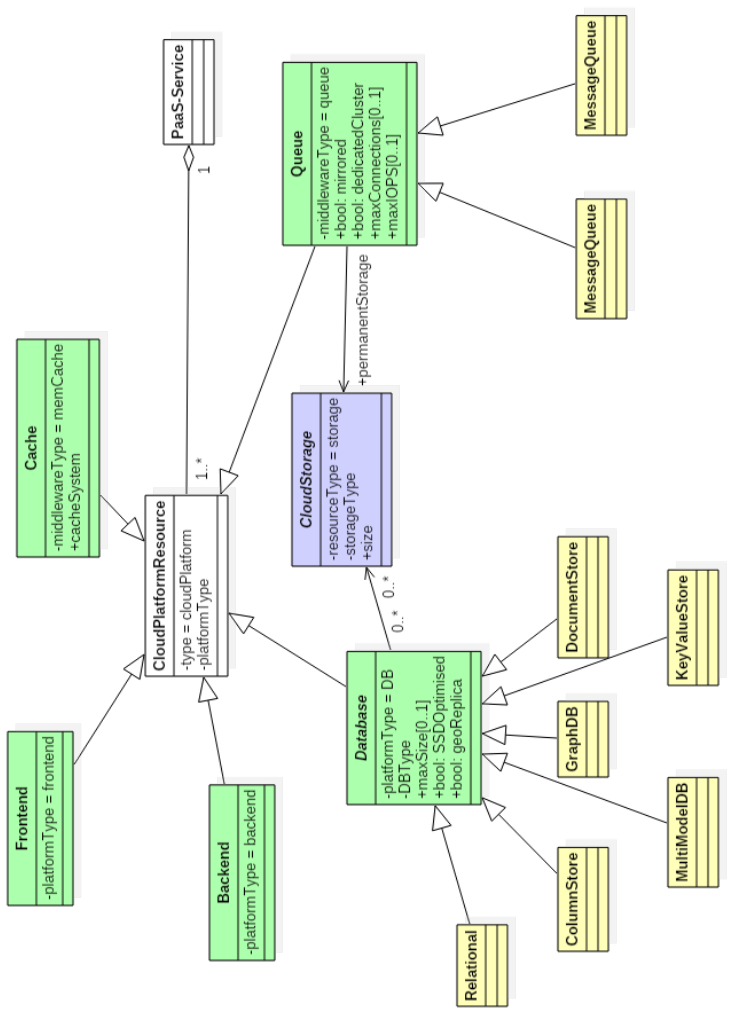


Figure 3.11: Cloud Meta-Model - Cloud Platform Service

Figure 3.11 expands the Platform as a service offering of cloud providers by introducing five main type of PaaS services. *Frontend* and *Backend* services are similar to compute engines in the IaaS environment, in fact they offer a computation resource on which to run user code. These services are usually backed by a pool of VMs managed by the cloud provider itself. The *Cache* service is used to store frequent access data in order to speed up the execution of some functionality of the application, many different caching technologies like Memcached [1] or Redis [2] are supported by cloud providers.

Many different type of *Database*s are supported by modern cloud provider, beside traditional *Relational* databases that support SQL access many NoSQL technologies are provided as platform services. Leveraging database PaaS services users have an interface to upload their data and submit queries but do not need to deal with the infrastructural complexity underlying the chosen database technology. They can also leverage functionality like data replication or backups managed by the cloud provider. Among the NoSQL databases we can identify some very popular technologies that are widely supported by cloud providers. These include *Key-Value Stores*, that allow to store a piece of information with a given key and retrieve it very efficiently; *ColumnStores* that organize data in columns, instead of rows as happens in relational databases, this technology if particularly efficient for analytic applications that calculate aggregates of values in the same column; *Document Stores* are used to manage semi structured data; *Graph* databases are used to host object oriented entities; finally, *MultiModel* databases are hybrid technologies that provides characteristics of multiple of these technologies.

Queue services are used to decouple application components. Queues can use to host *Message*s, either containing user requests or internal to the system or *Task*s that have to be scheduled for execution.

In order to provide a persistent storage Queues and Database might rely on the IaaS *CloudStorage* service offered by the provider.

3.4.2 CPIM for Private Cloud specification

When a company decides to move to the cloud it rarely abandons completely its previous infrastructure but usually adopt a gradually the new technology. This leads to hybrid situations in which both the private infrastructure and the public one coexist.

[1]http://memcached.org/
[2]http://redis.io/

To allow users to specify these kind of scenarios we have enriched the cloud meta-model presented in the previous section introducing elements that allow to specify a *Private Cloud* infrastructure. The main difference between the public and the private cloud is that in the latter one the user is aware of the physical infrastructure that can be specified in terms of *Hosts*. Each host is characterized by specific performance parameters. In order to hide the heterogeneity of different hosts usually some virtualization technology is used. The virtualization platform allows the user to manage virtual resources, like VMs, in the same way he/she manages them in the public cloud. A common strategy adopted by virtualization platforms to achieve high utilization of the physical infrastructure is to expose to the user more resources than those that are actually available in the physical servers, a *density* factor associated to each host can be used to specify how many resources are allowed for overbooking. This strategy is usually implemented also by public cloud providers, the main difference is that in a private cloud the density factor is known and has to be take into consideration when calculating the amount of virtual resources the private infrastructure can host. Another key difference of the private environment, with respect to the public one is that the cost of the infrastructure is mostly related to the energy consumption of each host, taking into consideration also the energy used for cooling purposes.

3.4.3 Cloud Provider Specific Model: The Windows Azure use case

Cloud Provider Independent Models can be used to specify some characteristics to an application that will be deployed in a cloud environment. These characteristics include many technological choices that can be used to drive the implementation and to have a better understanding of the management efforts of the operations team. Still these models do not provide all the information needed to deploy the application, the main deployment decision that has been postponed to the very end of the design process is the choice of the actual provider onto which deploy the application. This decision is complex and needs to take into consideration not only technical and economical aspects but might involve also business constraints and limitations imposed by the law. In order to specify a complete deployment the user needs the ability to select a specific technology offered by a cloud provider. In the Constellation case study, for example, we saw on Figure 3.7 that the user choose to deploy the svnAgent tier on a VM of size

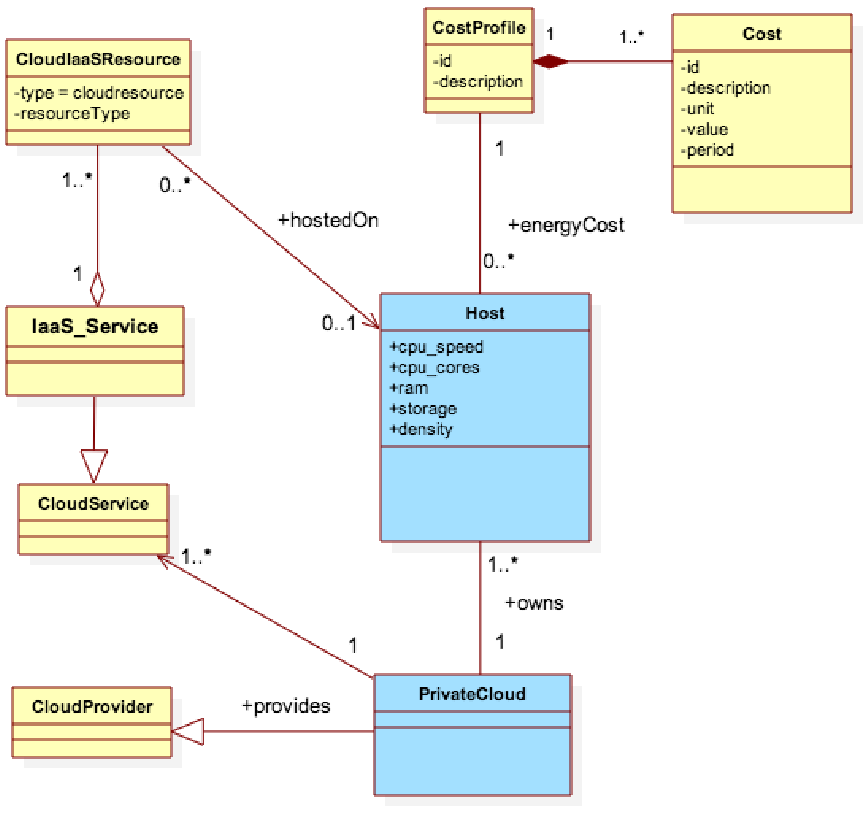


Figure 3.12: meta-model for the specification of private clouds

c3.4xlarge offered by Amazon.

The models presented in this section are a specializations of the previous model built to represent a particular cloud provider, in this case Microsoft Azure, and are called *Cloud Provider Specific Models* (CPSM). By specifying objects taken from this meta-model users provide all the information needed to deploy the application. This meta-model also stores performance characteristics of each resource that can be used to build a performance model and provide an estimation of the application performance, and cost, at design time.

Figure 3.13 shows specific PaaS and IaaS services offered, as an example, by **Windows Azure**[1]. **Web Role** and **Worker Role** are examples of *PaaS-Services* as they both provide platforms that can host *Frontend* and *Backend* services, respectively. **Virtual**

[1]The providers analyzed in this thesis and supported by the tool are: Amazon, Google, Microsoft Azure, Flexiscale, ProfitBricks, CloudSigma, Heroku and Pivotal

Machine, **Azure Blob**, **Azure Drive** are *IaaS-Services*, while **Azure Table** and **SQL Database** are *PaaS-Services*. They provide virtual machines, blob storage, volume storage, NoSQL and SQL databases, respectively.

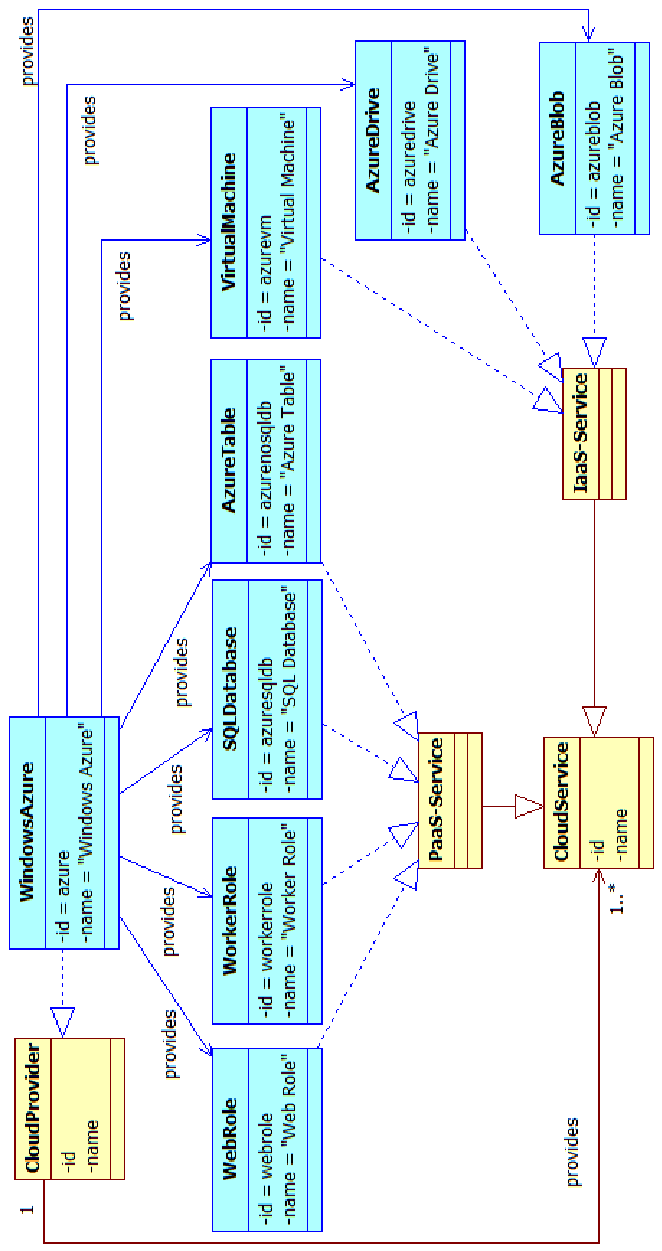


Figure 3.13: Azure CPSM - General overview.

Figure 3.14 shows the derivation of the CPSM associated with the Azure Blob and Azure Drive from the CPIM. **Azure Blob** and **Azure Drive** are IaaS services offered by Azure, they both relate to the generic *Cloud Storage* service of the CPIM model. In particular *Azure Blob Instances* are concrete realizations of a *Blob Storage Service* while **Azure Drive Instances** are an implementation of the *File System Storage* service.

Figure 3.15 shows a similar expansions of CPIM models into CPSM models with a focus on PaaS services, in particular on Database services that are mapped to *SQL Database* and *Azure Table* services. **SQL Database** and **Azure Table** are two *PaaS-Service* realisations and composed of **SQL Database Instances** and **Azure Table Instances**, respectively. SQL Database instances offer *Relational DBs*, while Azure Table instances offer *NoSQL DBs*, so both of these instance types are *Database Cloud Resources*.

Another specification of PaaS services is offered by Figure 3.16. **Web Role** and **Worker Role** are two *PaaS-Service* realisations composed of platforms running on one or more instances. **Worker Role Platforms** run on **Worker Role Instances** and **Web Role Platforms** run on **Web Role Instances**. Worker Role Platforms relate to *Backend Cloud Platforms* at CPIM level, while Web Role Platforms are related to *Frontend Cloud Platforms*.

Figure 3.17 expands the IaaS offering of Azure for what concerns the *Compute* service. The **Virtual Machine** service is composed of several **Virtual Machine Instances**, which are *Compute* resources. Many types of instance are offered by Azure, Figure 3.17 only shows some of them, like **Extra Small Instance**, the **Small Instance** and the **Medium Instance**.

Figures 3.18, 3.19 and 3.20 shows more details related to the virtual machine offering of Microsoft Azure. In particular, Figure 3.18 shows that the Virtual Machine has a **Virtual Machine Pricing** model, derived from the CPIM *Cost Profile*, which might be composed by multiple cost elements with different validity periods (e.g. different hours of the day). It is also possible to define **Azure Scaling Policies** on **Azure Scaling Groups**, which are realisations of CPIM *Adaptive Policies* and *Resource Pools*.

Figure 3.19 details the Virtual Machine Medium Instance type. A pool (**Azure VM Pool**) of Virtual Machine Instance can be associated to *Allocation Profiles* (**Azure Allocation Profile**), which keeps track of how many instances are allocated in a given time period. Information about the *Location* of the instances like the **Azure SubRe-**

gion specification are also available. For the performance point of view the virtual resources are characterised by their own *Efficiency Profiles*, which specify how their efficiencies vary in a given time period.

Finally, Figure 3.20, shows cost and performance details related to the *Azure Example Medium Instance*. This instance is characterised by a *Cost* (**Azure Medium VM Windows Cost**), an Operating System, CPU, Memory and Storage. This diagram also shows that the instance is located in the North Europe SubRegion (**North Europe** in the figure) within the Europe Region (**Europe** in the figure).

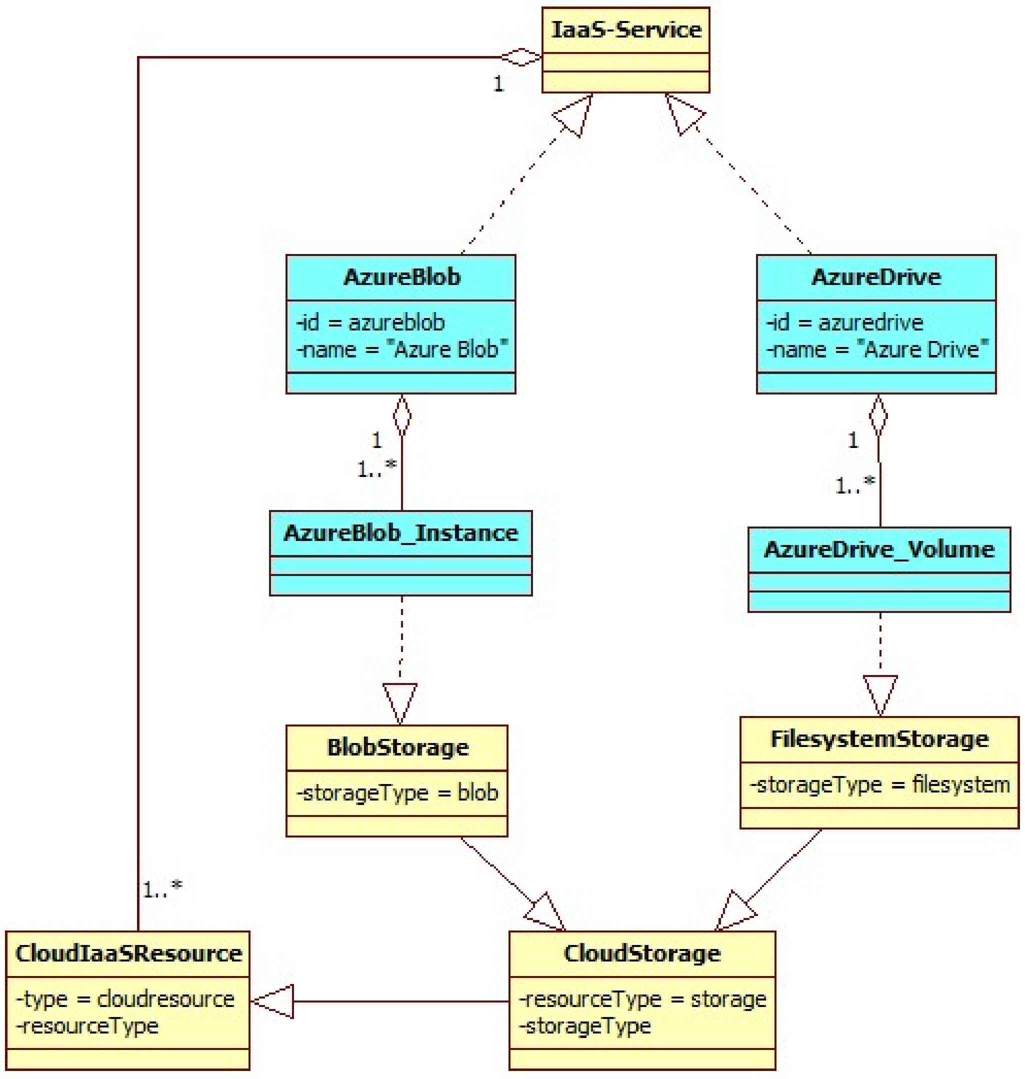


Figure 3.14: Azure CPSM - Blob and Drive Storage overview.

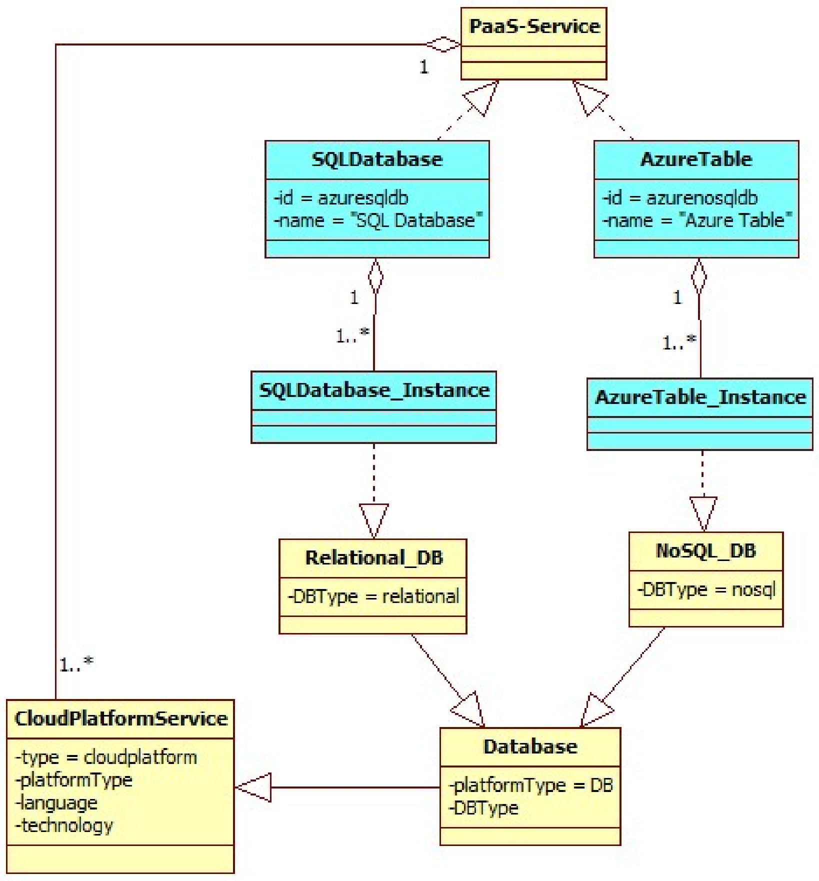


Figure 3.15: Azure CPSM - SQL Database and Table overview.

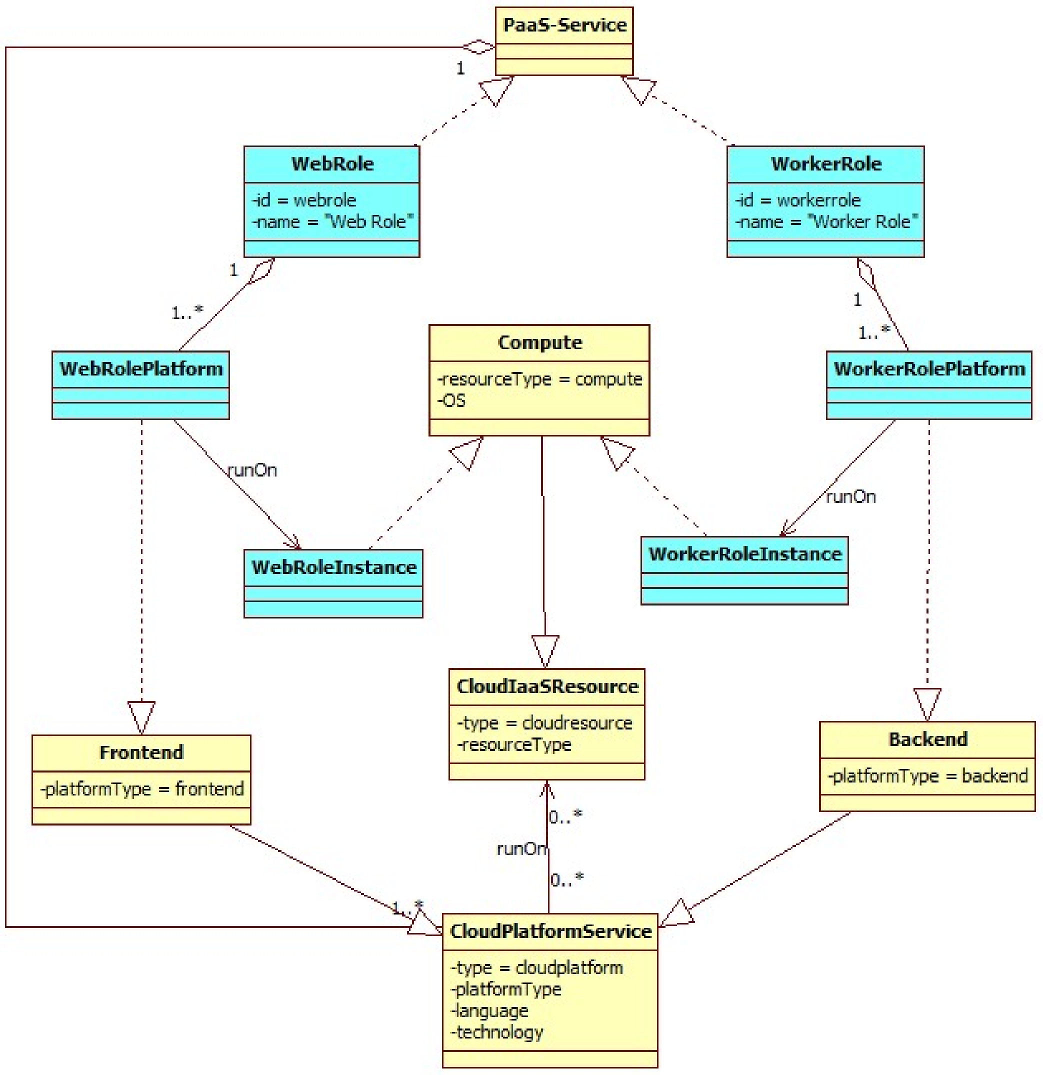


Figure 3.16: Azure CPSM - Web and Worker Roles overview.

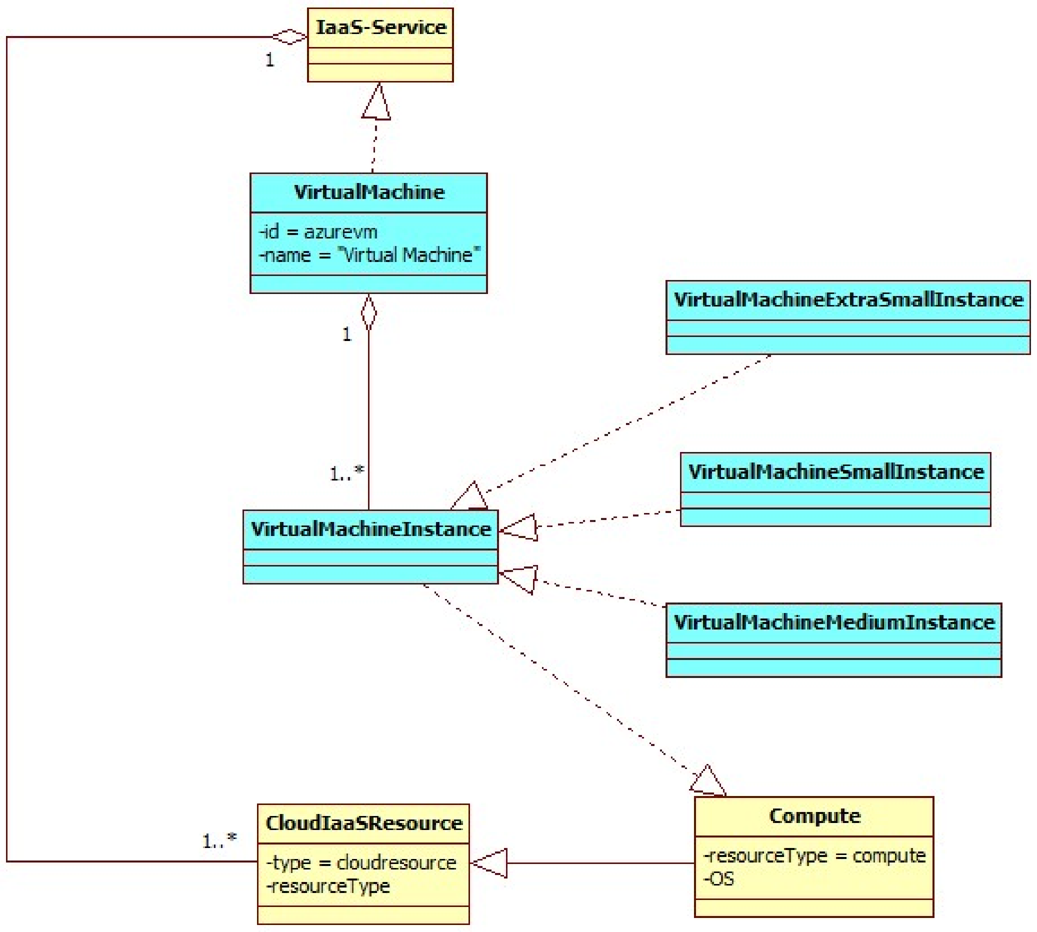


Figure 3.17: Azure CPSM - Virtual Machine overview.

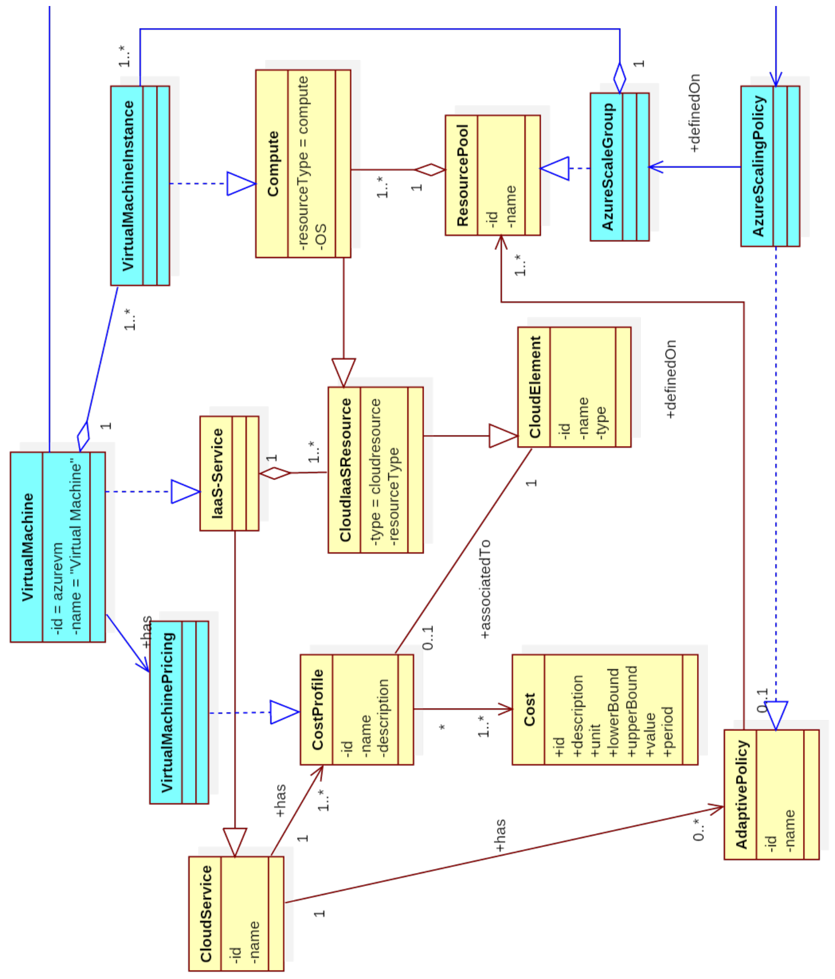


Figure 3.18: Azure CPSM - Virtual Machine details.

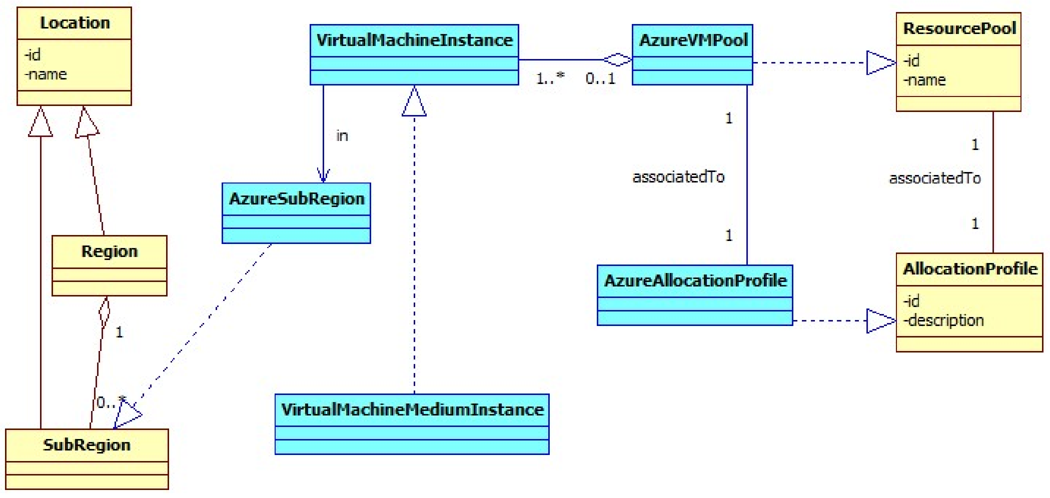


Figure 3.19: Azure CPSM - Virtual Machine Instance details.

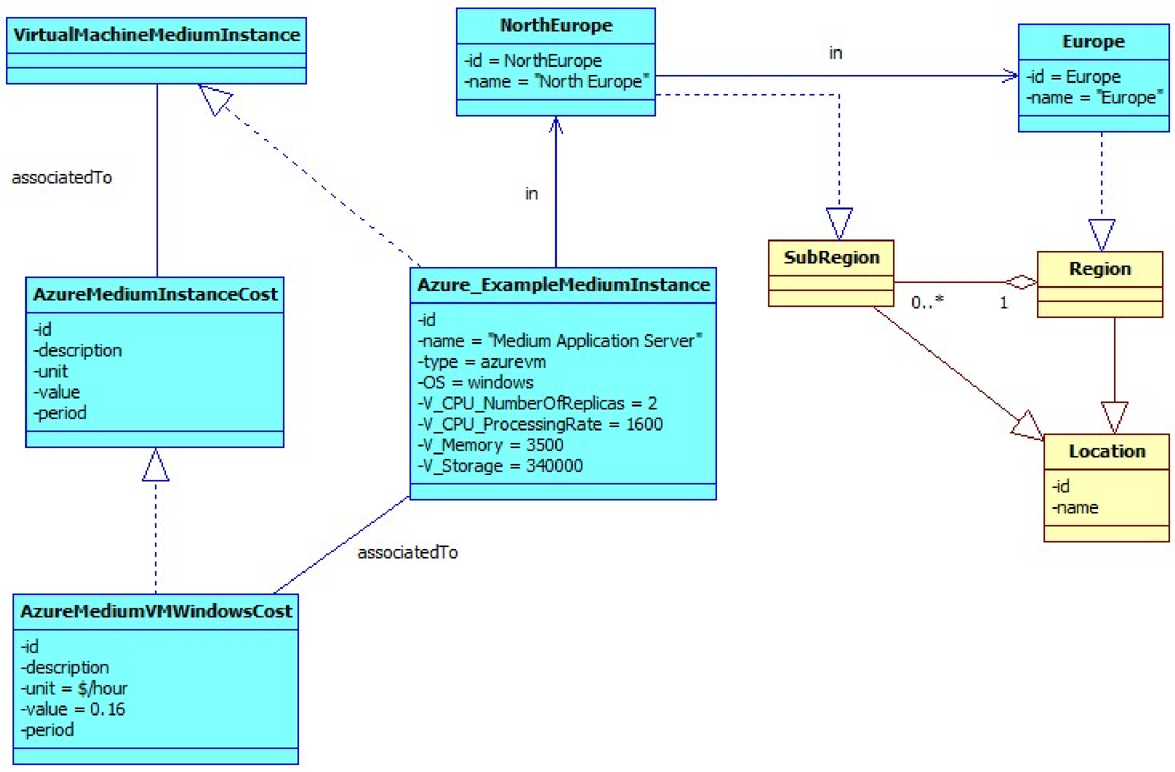


Figure 3.20: Azure CPSM - Virtual Machine Medium Instance example.

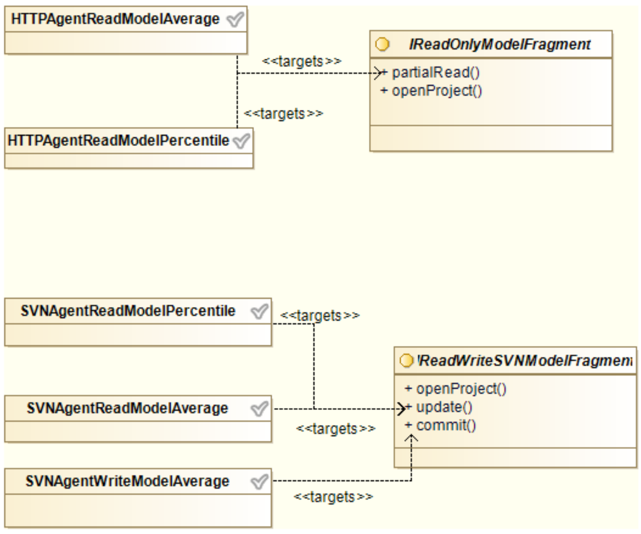
3.5 Modeling the Quality

As introduced in Chapter 1, the core of our approach is an optimization procedure that automate the selection of cloud services in order to minimize cost and guarantee QoS. For this reason, we included in the modeling phase some elements that allow the application architect to specify quality goals that the application should achieve. These requirements can be specified at all levels of abstractions by the use of QoS Models.

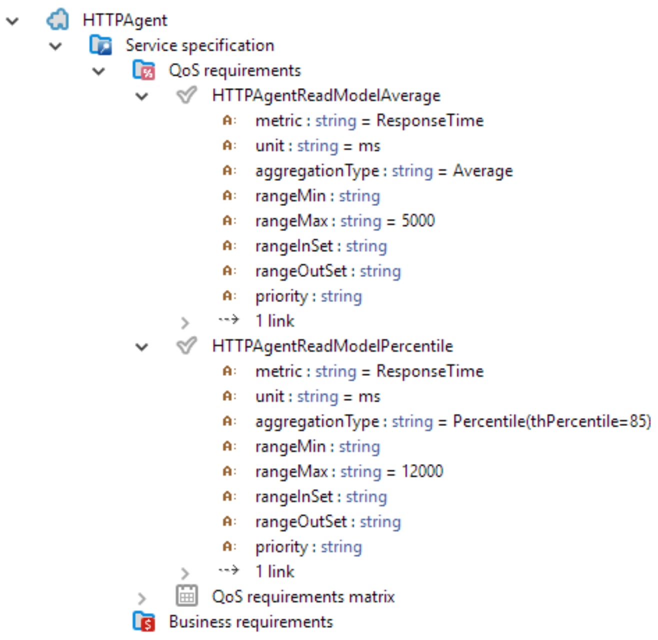
At CCIM level, interfaces exposed to the final user, as well as internal to the system itself, can be annotated with QoS properties. In particular the user can specify constraints on the QoS as shown in Figure 3.21. This figure shows five constraints related to the response time of the application. Constraints *HTTPAgentReadModelAverage* and *HTTPAgentReadModelPercentile*, whose details are shown in Figure 3.21b, target the *partialRead* functionality offered by the *IReadOnlyModelFragment* interface. These constraints specify that a model partial update has to be executed on average within 5 seconds and that the probability of such a request to be executed within 12 seconds is at least 85%. Figure 3.21a shows three similar constraints defined over the *update* and *commit* functionality of the *IReadWriteSVNModelFragment* interface. The main constraints considered to define the quality of the application are related to the response time of the functionalities offered to the user and the availability of the application.

At CPIM level, other type of constraints related to the architecture of the application can be specified. These constraints include the minimum and maximum amount of memory and number of cores required by a tier, the required operating system and the location of the service. Also constraints on the maximum CPU utilization level, for VM services, can be specified.

At CPSM level, the application architect can explicitly avoid the use of certain type of services (e.g. VMs of size m3.small) or prohibit the use of some cloud providers.



(a) QoS Constraints Association



```
HTTPAgent
  Service specification
    QoS requirements
      HTTPAgentReadModelAverage
        metric : string = ResponseTime
        unit : string = ms
        aggregationType : string = Average
        rangeMin : string
        rangeMax : string = 5000
        rangeInSet : string
        rangeOutSet : string
        priority : string
        1 link
      HTTPAgentReadModelPercentile
        metric : string = ResponseTime
        unit : string = ms
        aggregationType : string = Percentile(thPercentile=85)
        rangeMin : string
        rangeMax : string = 12000
        rangeInSet : string
        rangeOutSet : string
        priority : string
        1 link
      QoS requirements matrix
    Business requirements
```

(b) QoS Constraints Values

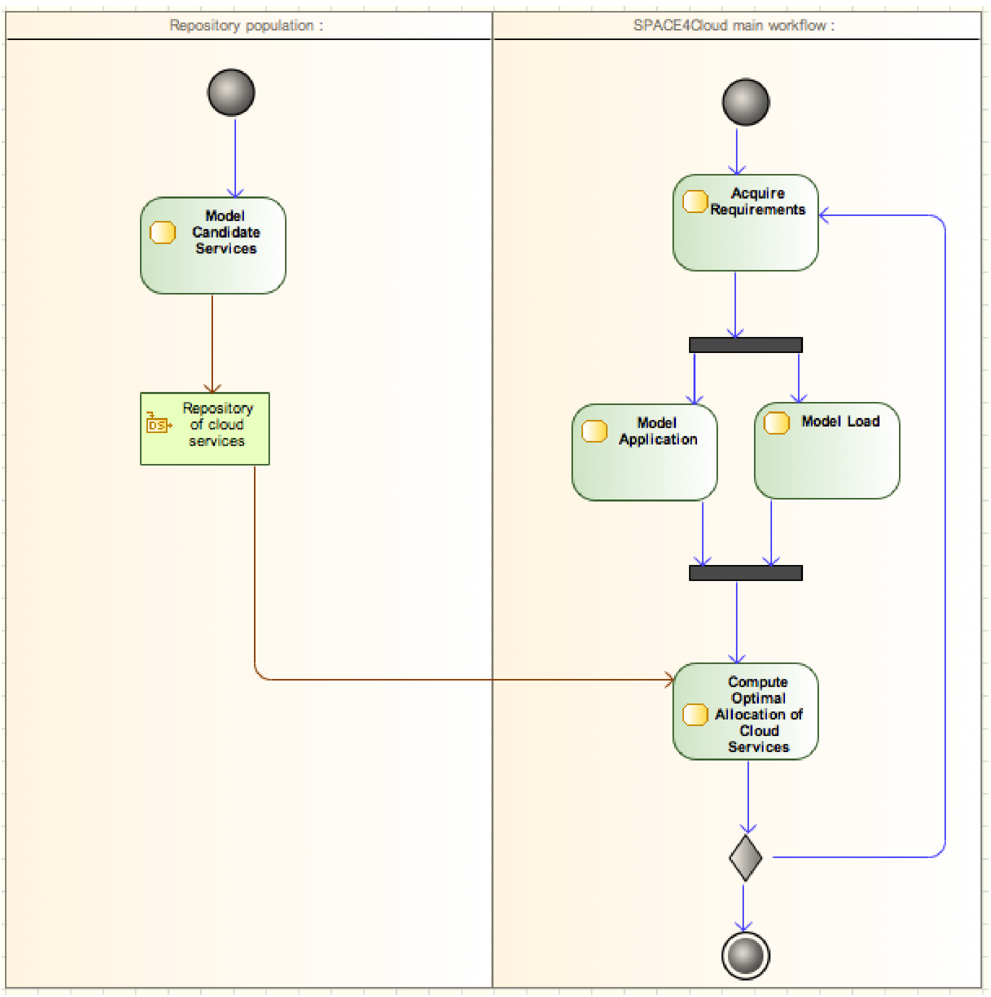
Figure 3.21: QoS Constraints

Chapter 4

Design Methodology and Optimization Approach Overview

Previous chapters have introduced the problem we want to solve (Chapters 1 and 3) and the modelling technology we use to tackle it (Chapter 3). This chapter introduces our approach in more details. Embracing the model driven methodology we make use of specific types of models in different stages of the process. We begin with the specification of high level design models as shown in Sections 3.2 and 3.3 to ease the modelling effort of the user. We then derive a performance model from the high level specification in order to analyze the QoS of the proposed design and deployment. This separation of concerns between the two different modelling technologies allows to simplify the interaction with the two models and is reflected in the architecture of the tool. Hiding the complexity of the performance model to the user allows him/her to focus on the design of the architecture leaving the synchronization of the design model to the performance model and the interpretation of its results to the tool.

In the remainder of this Chapter we overview the proposed modeling workflow in Section 4.1 and then we introduce the optimization approach that we developed to solve the minimum cost deployment problem, in Section 4.2.



Repository population :
SPACE4Cloud main workflow :
Model Candidate Services
Repository of cloud services
Acquire Requirements
Model Application
Model Load
Compute Optimal Allocation of Cloud Services

Figure 4.1: Main modeling steps in the SPACE4Cloud approach.

4.1 QoS-Oriented Design Methodology

Figure 4.1 shows the main steps of the modeling approach adopted by SPACE4Cloud. The left side of the picture shows the steps performed in the selection of the cloud offering to consider for the development of the application. These steps can be performed by cloud service provider in order to add their services to the public offering or by an independent quality assurance analyst. This activity is independent of the development of a particular application and its results can be re-used for the development of many applications. The outcome of these activities is a set of models that describe the available cloud offerings and can be used in the definition of the application deployment. The SPACE4Cloud tools comes with some pre-built models of many popular cloud provider that come as result of such an analysis.

The right side of the of Figure 4.1 shows the main modeling cycle performed in the design of a cloud application. These steps include requirement elicitation, application and workload modeling and deployment optimization. All these pieces of information are required by SPACE4Cloud to perform architectural analysis and the optimization process.

4.1.1 Definition of the characteristics of the candidate cloud services

To support the analysis an optimization of deployment choices performed by SPACE-4Cloud all the services that could be used by the application have to be modelled and details related to their cost and performance need to be available. Collecting such characteristics is usually a complex task due to the complexity of the cloud offerings, in terms of different cost models of similar services and to the lack of information provided directly by the cloud providers. For the performance point of view, for example, many cloud providers give some information on the number of cores or amount of memory of the VMs they are offering but they usually lack a description of the processor used to host the machine and the effects of the shared and virtual environment. To overcome this limitation many approaches, like the ARTIST project [19], involve benchmarking of different cloud services. In SPACE4Cloud we assume that this information is available and provided by a third party that might not be involved in the application modelling activity. The meta-models presented in Figure 3.8 to 3.20 of Chapter 3 allow developers to specify performance information of many cloud services. This information can be shared in a common repository by multiple users of SPACE4Cloud and updated periodically to add new cloud services or to increase the accuracy of the information of those already available. SPACE4Cloud comes with a Resource Model database, that stores information in the meta-model presented in Chapter 3. This database has been populated with the data that comes from the ARTIST project and with some manual gathering of information from coud providers websites. Currently the supported cloud providers are: Amazon AWS[1], Microsoft Azure[2], Flexiscale[3], Google[4], Heroku[5], Pivotal[6], ProfitBricks[7], and CloudSigma[8].

4.1.2 Modelling

In order to analyze the behavior of the application and derive a deployment solution SPACE4Cloud needs some knowledge of the application as well as of the incoming workload. The modeling formalism used to provide these information has been introduced in Chapter 3, we will give here a brief overview of the steps needed to provide the required information without entering again in the details of the modeling language. The first task in our modeling approach is to *Acquire Requirements* (see Figure 3.21a) for the application, that can be functional or non-functional requirements arising from the business level. Some of these requirements can be input directly in the model and will be taken into consideration by the modeling tool, others are only used by the development team to build a meaningful model of the application. An example of a requirement that can be gathered in this phase is a QoS requirement on the average expected execution time of a critical functionality. A functional requirement, for example, might derive by a particular technological choice performed by the software architects, e.g., the use of a centralized technology like SVN that impose some limitations on the maximum number of replicas of a particular component. In the Constellation case study introduced in Chapter 3 we see an example of a QoS requirement depicted in Figure 3.21, in this case the critical functionality is *parialRead* that is used to provide a quick update of a model and the required average execution time has to be lower than 5 seconds. Other constraints gathered in this phase might require a minimum level of availability for the entire system. These constrains are used directly by the optimization approach to drive the search for feasible solutions and can be used also by the development team to make some further considerations on the evolution of

[1]http://aws.amazon.com/
[2]http://azure.microsoft.com/it-it/
[3]http://www.flexiscale.com/
[4]https://cloud.google.com/
[5]https://www.heroku.com/
[6]http://pivotal.io/
[7]http://www.profitbricks.com/
[8]http://www.cloudsigma.com/

the application architecture.

After requirement elicitation the development team proceeds to *Model the Application* and *Model the Load*. These phases can be performed in parallel by different figures of development team. The first phase is usually performed by software architects that derive an initial architecture for the system by considering the requirements and their knowledge of the business environment, possibly interacting with domain experts. During this phase software architects might add other constraints related to the choice of a particular technology or design pattern building the architectural model shown in Figure 3.2 and the orchestration model of Figure 3.3. Recalling again the Constellation case study of Chapter 3, the use of the SVN technology to implement one of the application components comes with the definition of a new requirement on the scalability of the tier hosting that component. Since SVN is a centralized technology only a single replica of the SVN server component is allowed to exist. The second phase is focused on providing an estimation on how the system will be used by the customers by building a Usage model similar to the one presented in Figure 3.4. This phase requires a deep knowledge of the business domain and might require inspection of legacy systems or other systems in a similar domain to understand their typical usage pattern and report it to the current application model under development.

The third phase combines the information provided in the application and load model with the performance information stored in the repository of cloud services in order to find the optimal allocation of application components to cloud services. In our approach this step is completely automated by SPACE4Cloud and takes into consideration both the QoS requirements gathered in the first phase and those added during application modelling. The outcome is a more refined model of the application that could be directly used to drive the deployment of the system along with a performance characterization of the expected behavior of the application.

4.1.3 Reiteration

The models derived by the optimization of the allocation of cloud services are complete enough to initiate a deployment of the application, either by hand or by using some automated tool, but in many cases the development team would prefer to further inspect those models in order to refine the application, the requirements or inspect some other workload conditions. By reiterating the modeling phases, development teams can increase their knowledge of the system and come up with better alternative architecture or more informed deployment decisions. In Chapter 8 we will see how the reiteration of the application modeling phase on the Constellation case study brought to a better architecture that is capable of supporting a much higher workload and make a better use of the scalability offered by the cloud environment.

4.2 Hybrid optimization architecture

SPACE4Cloud delegates to other tools, namely Palladio Bench and Creator4Clouds, most of the modelling interactions with the user. Palladio Bench[1] is a state of the art modeling tool focused on early performance, reliability, maintainability and cost prediction of software artifacts developed by Karlsruhe Institute of Technology (KIT), FZI Research Center for Information Technology and Paderborn University. In the case of Palladio Bench, in particular, SPACE4Cloud has been seamlessly integrated in order to obtain performance evaluation results.

Creator4Clouds is the main component of the MODAClouds IDE, build during the MODAClouds European project as an extension of Modelio[2].

Leaving the modelling aspects to other tools SPACE4Cloud can reduce its interaction with the user mostly to the specification of analysis and optimization configurations.

As shown in Figure 4.2, SPACE4Cloud takes as input an *Enriched PCM*, described in Section 3.2. Such a model can be build using either Palladio Bench or Creator4Clouds, the latter tool offers more support to the definition of cloud related characteristics of the model.

SPACE4Cloud has been developed as a plugin for the popular Eclipse IDE, it is coupled with Palladio since it exploits its transformation engine to derive the LQN performance model from the design model. Once the performance model has been derived, the execution of SPACE4Cloud proceeds independently of Palladio acting directly on this model and maintaining an internal representation of the application in order to perform analysis and optimization. The final solution generated by the

[1]http://www.palladio-simulator.com/
[2]https://www.modeliosoft.com/

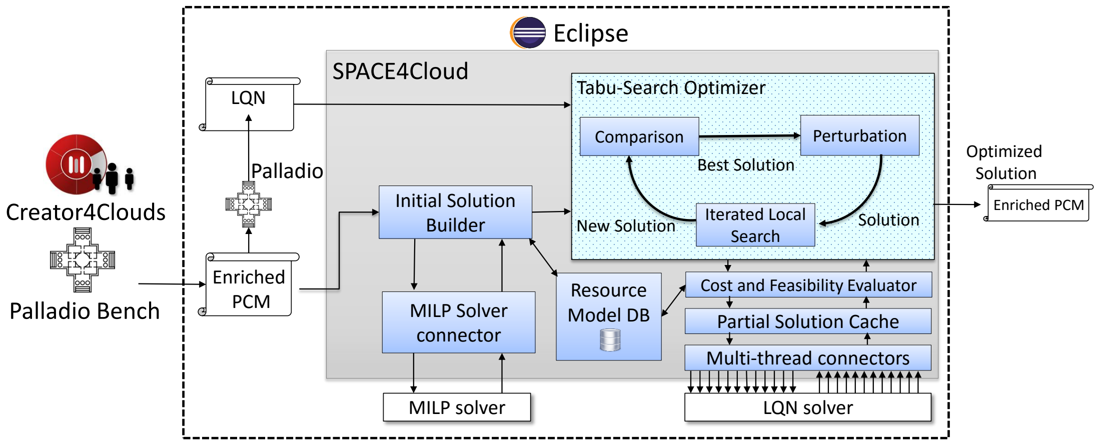


Figure 4.2: SPACE4Cloud architecture.

optimization is then exported back in the Enriched PCM model format in order to be inspected and eventually modified by the user.

The architecture of SPACE4Cloud is closely related to the design of its optimization strategy. The main component, shown in Figure 4.2 is the *Tabu-Search Optimizer*. This component implements all of the optimization algorithms described in Chapter 7. In order to support an efficient execution of the heuristic search and to provide other auxiliary functionalities to the user, many other components have been developed. In particular the *Initial Solution Builder* hosts the relaxed formulation of the optimization problem shown in Chapter 6. This component creates an instance of the optimization problem by considering a simplified performance model based on M/G/1 queuing networks and the QoS constraints. This instance is then solved by using an external *MILP Solver*. In order to support commercial as well as open source solvers, all the interactions with this service have been decoupled from the Initial Solution Builder introducing the *MILP Solver Connector* component.

The goal of the *Initial Solution Builder* is to perform a quick analysis of the entire optimization space and derive an initial solution for the heuristic problem.

The *Tabu-Search Optimizer* operates by modifying the initial solution to assess its feasibility against the more accurate LQN performance model and then optimizes it to reduce its operational cost. This method draws its inspiration from both Tabu [39] and Iterated local search [58]. More details on the algorithms implemented in this component can be found in Chapter 7. A similar approach, in which a relaxation of the problem is used to generate a good initial solution for an heuristic search, has been successfully applied in [48]. An analysis on the impact on the initial solution on the optimization has been reported in [17] and more details can be found in Chapter 8.

During the optimization process the *Tabu-Search Optimizer* generates many solutions exploring different deployment alternatives. This component generates several LQN performance models that represent a candidate solutions and relays on the *LQN Solver* to derive performance metrics. In order to effectively evaluate these solutions we have added few layers that decouple the interaction with the *LQN Solver*. Solving a single LQN model involves a computational intensive process and in fact represents the main bottleneck of the optimization, the main goal of the layers that mediate the interaction of the *Tabu-Search Optimizer* with the *LQN Solver* is to reduce as much as possible the actual number of evaluations performed.

The first layer, the one that interacts directly with the *Tabu-Search Optimizer*, is the *Cost and Feasibility Evaluator*. It has the responsibility of comparing the performance metrics derived by the LQN analysis against the constraints defined by the user and asses the feasibility of the solution. Another important feature of this component is related to the evaluation of the cost of the application. To this end the *Cost and Feasibility Evaluator* inspects the types and replicas of the solutions used to host the application and interacts with the *Resource Model Database* in order to retrieve the cost of each resource and calculate the cost of the entire solution.

The heuristic search described in Chapter 7 implements some well established techniques to avoid cycling into already explored solutions and avoid as much as possible to re-evaluate solutions already explored. Nevertheless in some situations the *Tabu-Search Optimizer* might require the solution of a performance model that has already been evaluated in earlier stages of the optimization. A simple situation in which such a behavior might occur is the case of two models of different hours that share the same workload. When evaluating deployment choices for these hours the heuristic algorithms will try to solve the two problems independently, generating similar solutions.

Recognizing these situations and re-using results of previous evaluations can greatly reduce the number of actual interactions with the *LQN Solver* and reduce the optimization time. The *Partial Solution Cache* component has been developed with this objective. When the evaluation of a performance model is requested this component looks at its internal memory to see if an equivalent model has been previously evaluated and, in that case it forwards the results of the previous evaluation to the *Cost and Feasibility Evaluator* component. If, on the other hand, there is no equivalent instance of the performance model previously evaluated, the request for the evaluation is forwarded to the *LQN Solver*. When the evaluation has been completed the result of the analysis is added to the memory. Since a complete optimization might involve the solution of a large number of performance models, from several hundreds to a few thousands, storing all the results of the evaluation in memory is not a viable option. To this end the *Partial Solution Cache* builds an hash of the models that it receives as input and relies on a memory structure to hold pointers to files containing the result of the evaluations. In such a way the memory footprint of this component is greatly reduced.

Finally, in order to exploit the parallel architecture of modern computers a *Multi-Threaded Connector* is used to manage different *LQN Solver* at once. SPACE4Cloud supports LQNS [35] and LINE [67] as *LQN Solver*. Beside their functional differences, the two solvers also differ in the way they communicate with SPACE4Cloud. LQNS is an executable that receives as an input the path to a LQN model to analyze and produce a file with the result of the evaluation. LINE, on the other hand, acts as a server receiving evaluation requests using socket connections. From the point of view of the parallel evaluation of different models, a single instance of LINE can be used while for LQNS a separate instance has to be launched for each model. The *Multi-Threaded Connector* takes care of all of the interactions with the solver and synchronizes the evaluation of all the models composing a solution.

In all our experiments we choose to use LINE as performance engine because of its ability to take into consideration variability in the performance of processing resources by means of Random Environments [29]. Random environments are Markov chain-based descriptions of time-varying operational conditions. The evolution of such conditions is independent of the system state and model the exogenous variability in a cloud deployment. Each processing resource of the LQN model generated by the application architecture is associated to a Continuous Time Markov Chain (CTMC) in which each state represent a certain performance level. A simple two state CTMC, for example, could model situations in which the shared physical resource is under heavy or low contention. This ability is quite important in our view since the cloud infrastructure is usually shared among multiple users and complete independence from the actions of other users, from the performance point of view is not always possible. Another important feature provided by LINE is the ability to derive not only averages but also percentiles of the distribution of the response time of the application. This ability allowed us to include this metric in the definition of QoS constraints giving the user a more powerful way to control the quality of its application.

Another key component of the architecture is the *Resource Model Database*. This component hosts performance and cost information of cloud resources supported by the tool. Many components interact with this database, in particular Creator4Clouds integrates it in order to provide the user a list of supported resources to use when modeling the application. The database includes benchmark results of the ARTIST[1] European project in order to assess the performance of VMs offered by the supported cloud providers. The Initial Solution Builder uses it to retrieve performance and cost information of the available resources to be used as parameters of the relaxed problem. The Tabu-Search Optimizer uses it to retrieve performance information needed to select which type of resource to use for the deployment and update the LQN performance model. Finally, the Cost and Feasibility Evaluator uses the resource cost in order to evaluate the cost of a given solution.

[1]http://www.artist-project.eu/

Chapter 5

Optimization Problem Formulation

As discussed in Chapter 1, when it comes to deploy an application in cloud environments the development team is faced with many possible deployment alternatives. If the team is also concerned about the availability of its application and decides to replicate it on multiple providers the problem becomes even more complex.

In the remainder of this chapter, as introduced in Chapter 1, we will use the term *user* to identify the development team involved in the use of our approach to build a multi-cloud application.

In the example of Section 3.3, the user chooses to deploy the application on a single cloud provider, Amazon, and to deploy the adminServer tier in a VM of size c3.large, the httpAgent on a set of VMs of size m3.large and the svnAgent on a VM of size c3.4xlarge.

We have seen that a complete description of a deployment configuration requires mostly three choices: which type of cloud resource (e.g. size of VM) assign to each tier of the application; how many replicas of such a resource to use for each hour of the day; how to forward the incoming requests on the available providers, if we are considering a multi-cloud scenario.

An analysis of the deployment choices we made in this example can give us some information on the expected QoS of our application. If we find out that some of the constraints introduced in the model are not met by our deployment we might reconsider some deployment choices (e.g., we could decide to use more expensive and powerful VMs or increase the number of replicas). Refining our deployment configuration in an iterative way by analyzing several alternatives is a very tedious and long process.

Moreover, performing a manual exploration, we will only be able to analyze a very limited subset of the possible deployment configurations so we are likely to miss some more effective deployments.

We can try to automate the generation of the deployment configuration by formalizing the problem and solving it. A rigorous mathematical problem formulation can help us in two ways. From a very practical point of view, if we are able to express the problem in a closed form, we can try to solve it using state of the art tools and analyze manually only the deployment configurations derived by the solution of the problem.

Even if we are not able to provide a formulation of the problem that can be effectively solved with standard methods or tools we might learn some insights of the structure of the problem to drive our search for good solutions.

According to the Model Driven Approach introduced in Section 3.1, such a formalization of the problem can be seen as a specification of our application along with its constraints in a different modeling formalism. As we build an LQN model for the application and use established tools and techniques to perform QoS analysis, we describe our problem in the mathematical domain and use tools developed in that domain to analyze our problem. This analysis is the main focus of this chapter.

The remainder of this Chapter introduces the problem, in Section 5.1, in order to provide some context and relate the main variables and parameters to those defined in the Constellation case study of Chapter 3. We then provide the analytic formulation of the deployment problem, in Section 5.2 and an equivalent formulation that shows our problem to be NP-hard in Section 5.3.

5.1 Problem definition

As shown in Chapter 3, applications are modeled with a component based approach where each application component contains a closely coupled set of functionality. The definition of the scope of a components is a complex task performed by the software architect in the modeling phase. Components provide a logical separation of concerns of the responsibility of different parts of the application and usually require some interconnection in order to provide their functionality. Components are grouped into application tiers, like the Administration Server in our case study, that represent a single deployment unit. Tiers are allocated on a cloud service, considering the Constellation case study the Administration Server tier is allocated to a pool of VMs. In general tiers represent dynamic elements with characteristics that can vary in order to handle dynamic changes in the application environment, such as a sudden increase or decrease of the incoming workload.

The goal of our problem is to find the cheapest configuration of cloud services (in case of IaaS services the number and type of VMs for each application tier) capable to fulfill QoS requirements and service allocation constraints for each hour of the reference day. As introduced in Chapter 3, QoS constrains predicate on the performance of the application as experienced by the user, while service allocation constraints predicate on the technological requirements e.g., minimum amount of memory required by a component, provided by application architects. These constraints are specified at the CPIM and CPSM level as shown in Section 3.5.

5.2 Analytic Formulation

As previously introduced, an application is hosted on a set of cloud providers $\mathcal{P}$ and divided into tiers, denoted by $\mathcal{I}$, that support the execution of application components. Each tier consists of multiple homogeneous resources, like VMs in the IaaS scenario, that shares evenly the incoming workload. Let $\mathcal{V}$ be the set of available types of resources (e.g., Amazon EC2 m3.medium, c3.large, etc.[1]) offered by provider p and $\mathcal{T}$ a set defined by the N time intervals in which the reference day has been split (i.e., 24). Each resource type $v \in \mathcal{V}$ is characterized by the corresponding model stored in the *Repository of cloud services*. This model stores cost and performance information like the price of the resource $C^p_{v,t}$, which might change over the time horizon, the memory size M^p_v and some other performance information like the number of cores or the speed of the processing resource that will be used in Chapter 6. Each user interacts with the application executing a chain of requests according to the defined end users' behavior model, like the one in Figure 3.4; the set of possible requests is referred to as $\mathcal{K}$. Each request $k \in \mathcal{K}$, in turn, is characterized by a probability to be executed, derived from the users' behavior model, and by a set of components supporting its execution (i.e., its execution path [15]). The incoming requests that the application has to process in

[1]http://aws.amazon.com/ec2/instance-types/

a particular time interval is described by N_t, with $t \in \mathcal{T}$. This workload is split non uniformly among all the available providers, the amount of workload processed by each provider is N_t^p. If the user is concerned with the availability of the application and decides to use a multicloud deployment the minimum number of provider that the optimization should select is given by $\overline{H}$ and the minimum amount of workload that each of the selected cloud provider receives at any time interval is given by $\overline{\gamma}$. The availability of a particular provider p is specified by parameter $avail_p$.

As shown in Figure 3.21, in Chapter 3, SPACE4Cloud supports the definition of performance requirements in terms of thresholds on the average or expected response time $\overline{R}_k^E$ and on the percentiles $\overline{R}_k^{Perc}$. Constraints on the average execution time can be specified for a set of functionalities represented by $\mathcal{K}_{Avg} \subset \mathcal{K}$ and constraints on the value of the α_k percentile of the response time can be specified for a set of functionalities represented by $\mathcal{K}_{Perc} \subset \mathcal{K}$. In the most general case both constraints can be specified for any functionality, leading to $\mathcal{K}_{Avg} \cap \mathcal{K}_{Perc} \neq \emptyset$. Architectural constraints introduced by software architects can predicate on the minimum amount of memory required by a resource to host a particular tier, represented by $\overline{M_i}$ or the maximum replicas of resources used to host the tier $\overline{Z_i}$. In the Constellation case study of Chapter 3 this limit is set to 1 for the tier hosting the SVN server.

The decision variables of our problem are the following:

- x_p is a binary variable that assumes value of 1 if provider p is selected to host the application, 0 otherwise;
- $z_{i,v,t}^p$ is an integer variable that specifies the number of replicas of resource of v type (either a IaaS VM or a PaaS Container), assigned to the i-th tier at time t;
- N_t^p is an integer variable that specifies the number of requests entering to provider p at time t;
- $w_{i,v}^p$ is an accessory binary variable equal to 1 if the resource of type v is assigned to the i-th tier of the application, 0 otherwise.

It is worth to notice that variables $z_{i,v,t}^p$ and N_t^p are the main decision variables while $w_{i,v}^p$ and x_p are an accessory variables used to simplify the definition of constraints and make the problem easier to understand.

Table 5.1 and Table 5.2 summarizes all the parameters and variables described in this section and used in the rest of the chapter.

For what concerns the incoming workload, the heuristic optimization process performed in Chapter 7 supports both open and closed workloads. In most of our analysis we used LINE to analyze the LQN models derived by the application design because of its ability to derive percentiles of response time. At the time of writing LINE supports only closed workloads. When addressing an open workload we can make use of the Little's law, $N = \Lambda \times (R + Z)$ where Z is the user think time. Since in the considered scenarios the maximum response time allowed by the application, $\overline{R}$ is significantly lower than Z we can estimate N by using $N = \Lambda \times Z)$ as in [14]. Under this assumption we can move from open to closed workloads. For simplicity the MILP optimization model presented in this section is presented as an open model.

Decision variables.

x_p	Binary variable that is equal to 1 if provider p is selected, 0 otherwise
$w^p_{i,v}$	binary variable that is equal to 1 if a resource of type v, of provider p, is assigned to the i-th tier and equal to 0 otherwise
$z^p_{i,v,t}$	Number of virtual machines of type v assigned to the i-th resource pool at time t
N^p_t	Workload entering to provider p at time t

Table 5.1: Optimization model decision variables.

The optimization model presented in this section describes a multi-cloud deployment problem:

$$\min_{\mathcal{Z},\mathcal{V},\mathcal{P}} \sum_{t\in\mathcal{T}} \sum_{p\in\mathcal{P}} \sum_{v\in\mathcal{V}_p} \sum_{i\in\mathcal{I}} C^p_{v,t} z^p_{i,v,t} \tag{5.1}$$

System parameters

Index	
$t \in T$	Time Interval
$i \in \mathcal{I}$	Set of application tiers
$k \in \mathcal{K}$	Set of class of requests
$p \in \mathcal{P}$	Set of cloud providers
$v \in \mathcal{V}_p$	Type of resource offered by provider p
Parameters	
N_t	Number of incoming requests (workload) at time t
$C^p_{v,t}$	Cost of a resource of type v at hosted by provider p at time t
M^p_v	Memory of a resource of type v in provider p
$\overline{M}_i$	Minimum amount of memory required to host tier i
$\overline{Z}_i$	Maximum number of replicas used to host tier i
$\overline{R}^E_k$	Maximum average response time for requests of class k
$\overline{R}^{Perc}_k$	Value of the α_k-th percentile of the response time of class k
α_k	Percentile level for class k
$\overline{\gamma}$	Minimum percentage of workload processed by a provider
$\overline{H}$	Minimum number of cloud providers
$avail_p$	Availability of provider p
$MaxUnavail$	the maximum unavailability specified by the user

Table 5.2: Optimization model parameters.

Subject to:

$$\sum_{p\in\mathcal{P}} x_p \geq \overline{H} \tag{5.2}$$

$$\sum_{v\in\mathcal{V}_p} w^p_{i,v} = x_p \qquad \forall p \in \mathcal{P}, \forall i \in \mathcal{I} \tag{5.3}$$

$$x_p\overline{\gamma}N_t \leq N^p_t \qquad \forall p \in \mathcal{P}, \forall t \in \mathcal{T} \tag{5.4}$$

$$N^p_t \leq x_p N_t \qquad \forall p \in \mathcal{P}, \forall t \in \mathcal{T} \tag{5.5}$$

$$\sum_{p\in\mathcal{P}} N^p_t = N_t \qquad \forall t \in \mathcal{T} \tag{5.6}$$

$$w^p_{i,v} \leq z^p_{i,v,t} \qquad \forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall v \in \mathcal{V}_p, \forall i \in \mathcal{I} \tag{5.7}$$

$$z^p_{i,v,t} \leq \overline{Z}_i w^p_{i,v} \qquad \forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall v \in \mathcal{V}_p, \forall i \in \mathcal{I} \tag{5.8}$$

$$x^p \in \{0,1\} \qquad \forall p \in \mathcal{P} \tag{5.9}$$

$$w^p_{i,v} \in \{0,1\} \qquad \forall p \in \mathcal{P}, \forall v \in \mathcal{V}_p, \forall i \in \mathcal{I} \tag{5.10}$$

$$z^p_{i,v,t}\ Integer \qquad \forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall v \in \mathcal{V}_p, \forall i \in \mathcal{I} \tag{5.11}$$

$$\sum_{v\in\mathcal{V}_p} w^p_{i,v} M^p_v \geq \overline{M_i} \qquad \forall p \in \mathcal{P}, \forall i \in \mathcal{I} \tag{5.12}$$

$$\sum_{p\in\mathcal{P}} (log(1 - avail_p) \cdot x_p) \leq log(MaxUnavail) \tag{5.13}$$

$$x_p E[R^{tp}_k(N^p_t, \mathcal{Z}^p_t)] \leq \overline{R}^E_k \qquad \forall p \in \mathcal{P}, \forall t \in \mathcal{T}, \forall k \in \mathcal{K}_{Avg} \tag{5.14}$$

$$P(R^{tp}_k(N^p_t, \mathcal{Z}^p_t) \leq \overline{R}^{Perc}_k) \geq \alpha_k x_p \qquad \forall p \in \mathcal{P}, \forall t \in \mathcal{T}, \forall k \in \mathcal{K}_{Perc} \tag{5.15}$$

Where $\mathcal{Z}_t^p = \{z_{i,v,t}^p | (i,v) \in \mathcal{I} \times \mathcal{V}_p\}$ represents the assignments of VM types and replicas to application tiers for provider p at time t.

The main objective of this problem is to minimize the cost of using the cloud infrastructure. This cost is represented by Formula 5.1 and can be derived by the sum of all the costs related to the utilization of cloud resources over all the application tiers i, the time intervals t, the selected providers p and corresponding selection of resource types v.

In the most general case our application might be replicated over multiple providers to provide guarantees on its availability. The minimum number of providers on which the application has to be replicated is estabilished by Constraints 5.2. Each cloud provider offers different type of resources, we use the set $\mathcal{V}_p$ to identify all the resources offered by provider p. The binary variable, $w_{i,v}^p$, which denotes the assignment of a certain type v of resource, among those offered by provider p, to host tier i of the application. The two binary variables just introduced are used in Constraints 5.3 to guarantee that only a single type of resource is selected for each active provider and application tier. These constraints also guarantee that if a provider is not selected (i.e., $x_p = 0$), no type of resource offered by that provider is assigned to host the application.

Constraints 5.4, 5.5 and 5.6 are related to the the incoming flow of requests and how it is split among the selected providers. The incoming workload is represented by N_t, while the amount of requests that reach a provider p is represented by N_t^p. Recalling the modeling approach presented in Section 3.2 we see that different values of the workload can be specified for each time interval t. Constraints 5.4 reflects the minimum partition of workload $\overline{\gamma}$ that has to be served by each provider, and Constraints 5.5 guarantees that requests are only directed to selected providers. Finally, Constraints 5.6 makes sure that all incoming requests are served.

As previously introduced, variable w_i^p, v represents the binding between a resource type and an application tier. This variable alone does not convey the information about the number of replicas of such resource. For what concerns the number of replicas of each resource used to host application tiers we have to make a distinction between the IaaS and PaaS scenarios. If the resource selected to host a tier is an IaaS resource, namely a group of VMs, we can directly control the number of replicas, but if the resource is a PaaS Container this control is not always available. Some cloud services, like Microsoft Azure Web App[1] allow to control how many IaaS instances are used to provide a PaaS service and we can exploit this information to control this resource in the same way we operate with IaaS resources. Other cloud services, like Amazon Dynamo DB[2], allow only to specify a threshold on the response time or the throughput of the

[1]https://azure.microsoft.com/en-us/documentation/articles/web-sites-scale/

[2]http://aws.amazon.com/it/dynamodb/details/

service and autonomously adjust the number of replicas used to meet the defined goal, in this scenario we can use the prediction of the response time of the component generated by the assessment of the solution at design time and use it as threshold for the autoscaling. To this end we introduced the integer variable $z^p_{i,v,t}$. The two Constraints 5.7 and 5.8 represent upper and lower bounds to the number of replicas of resources of type v, assigned to tier i within each provider. Another important effect of this constraints is to restrict resources assigned to a certain tier to be of the same type. Using homogeneous resources eases the process of load balancing at runtime and is widely used in cloud environments [12]. The management of heterogeneous VMs is relevant in a private cloud, but it is usually not considered in public clouds [27][41].

Constraints 5.12 represents a constraint family that can be provided by the user when selecting computational resources, namely VMs. These constraints restricts the type of machines that can be assigned to host a particular tier by removing those machines that do not provide enough memory. The constraint on the minimum amount of memory specified by the user is represented by parameter $\overline{M_i}$ while the amount of memory offered by the VM is represented by M^p_v.

All the constraints presented so far define requirements that shape the structure of the solution but do not address directly the QoS of the application; we call these *Architectural* requirements. On the other hand the last two family of constraints of the model are related to the QoS of the application and are called *QoS* requirements. Both these constraints provide some restrictions on the response time of the application. To calculate this metric we need to rely on some performance model. In this formulation we focus on modeling constraints that the user can express with our modeling approach postponing the discussion on how to derive this metric to Chapter 6. In general we can say that the response time of a functionality k offered by the application replica in provider p at time t, identified by R^{tp}_k is a, possibly non linear, function of its incoming workload N^p_t and our deployment choice $\mathcal{Z}^p_t$.

Constraints 5.14 predicates on the average response time of the application, only on selected providers, to be lower than the threshold defined by the user $\overline{R}^E_k$. Constraints 5.15 operates on the distribution of the response time. In particular it constraints the value of the α_k percentile of the response time to be lower than a threshold defined by the user $\overline{R}^{Perc}_k$. As an example if α_k is 95% and R^{Perc}_k is 10 seconds this constraint says that 95% of requests of class k has to be served within 10 seconds by their arrival in the system. In both Constraints 5.14 and 5.15 variable x_p is used as a logical value in order to enable the constraint only for those providers that are selected to be used.

Finally Constraint 5.13 is used to ensure a minimum level of availability for the system.

It is derived by operating on the unavailability, defined ad $1 - avail_p$ which represents the probability of provider p to be unavailable. Since our application can be deployed on multiple clouds we consider the entire application unavailable is all the selected providers, those for which $x_p = 1$, are unavailable. Since the failure of different cloud providers are independent events the probability that all of the selected providers fail is given by:

$$\prod_{p \in \mathcal{P}} (1 - avail_p)^{x_p} \leq MaxUnavail \tag{5.16}$$

This value has been bounded by the maximum unavailability of the system defined by the user. By applying the logarithm to both sides of Formula 5.16 we get constraint 5.13.

5.3 Bi-level formulation and NP-Hardness

As said in the introduction of this chapter, using the mathematical notation to express our model allows us to exploit some tools and techniques to analyze different aspects of the problem at hand. In particular, we can analyze the complexity of the problem in order to gain hints on what are the main factors that affect the time required to solve it. To do so, we can re-write the optimization problem of the previous section by making more evident the dependency among the variables that we need to optimize. This work lead to the following formulation of the problem.

$$\min_{\mathcal{V},\mathcal{P}} \sum_{t \in \mathcal{T}} \sum_{p \in \mathcal{P}} \sum_{v \in \mathcal{V}_p} \sum_{i \in \mathcal{I}} C^p_{v,t} z^p_{i,v,t} \tag{5.17}$$

Subject to:

$$\sum_{p \in \mathcal{P}} x_p \geq \overline{H} \tag{5.18}$$

$$\sum_{v \in \mathcal{V}_p} w_{i,v}^p = x_p \qquad \forall p \in \mathcal{P}, \forall i \in \mathcal{I} \tag{5.19}$$

$$x_p \overline{\gamma} N_t \leq N_t^p \qquad \forall p \in \mathcal{P}, \forall t \in \mathcal{T} \tag{5.20}$$

$$N_t^p \leq x_p N_t \qquad \forall p \in \mathcal{P}, \forall t \in \mathcal{T} \tag{5.21}$$

$$\sum_{p \in \mathcal{P}} N_t^p = N_t \qquad \forall t \in \mathcal{T} \tag{5.22}$$

$$x^p \quad \in \{0,1\} \qquad \forall p \in \mathcal{P} \tag{5.23}$$

$$w_{i,v}^p \quad \in \{0,1\} \qquad \forall p \in \mathcal{P}, \forall v \in \mathcal{V}_p, \forall i \in \mathcal{I} \tag{5.24}$$

$$\sum_{v \in \mathcal{V}_p} w_{i,v}^p M_v^p \geq \overline{M_i} \qquad \forall p \in \mathcal{P}, \forall i \in \mathcal{I} \tag{5.25}$$

$$\sum_{p \in \mathcal{P}} (log(1 - avail_p) \cdot x_p) \leq log(MaxUnavail) \tag{5.26}$$

$$z_{i,v,t}^p = \arg\min_{\mathcal{Z}} \sum_{t \in \mathcal{T}} \sum_{p \in \mathcal{P}} \sum_{v \in \mathcal{V}_p} \sum_{i \in \mathcal{I}} C_{v,t}^p z_{i,v,t}^p \tag{5.27}$$

Subject to:

$$z_{i,v,t}^p \leq \overline{Z_i} w_{i,v}^p \qquad \forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall v \in \mathcal{V}_p, \forall i \in \mathcal{I} \tag{5.28}$$

$$z_{i,v,t}^p \ Integer \qquad \forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall v \in \mathcal{V}_p, \forall i \in \mathcal{I} \tag{5.29}$$

$$x_p E[R_k^{tp}(N_t^p, \mathcal{Z}_t^p)] \leq \overline{R}_k^E \qquad \forall p \in \mathcal{P}, \forall t \in \mathcal{T}, \forall k \in \mathcal{K}_{Avg} \tag{5.30}$$

$$P(R_k^{tp}(N_t^p, \mathcal{Z}_t^p) \leq \overline{R}_k^{Perc}) \geq \alpha_k x_p \qquad \forall p \in \mathcal{P}, \forall t \in \mathcal{T}, \forall k \in \mathcal{K}_{Perc} \tag{5.31}$$

Where $\mathcal{Z}_t^p = \{z_{i,v,t}^p | (i,v) \in \mathcal{I} \times \mathcal{V}_p\}$ represents the assignments of VM types and replicas to application tiers for provider p at time t.

This formulation is equivalent to the one presented in Section 5.2, in fact most of the constraint can be easily traced back, but provides a more clear view on the interaction of the decision variables. The objective function in Formula 5.17 operates on sets $\mathcal{P}$ and $\mathcal{V}$ and represent two deployment decisions: the splitting of the workload among the providers and the selection of which type of resource to use. Constraints 5.18 to 5.25 are related to the assignment of these variables. The assignment of the number of replicas for the resources of each tier, that we represent with $z_{i,v,t}^p$, has been separated into another problem whose objective function 5.27 is the same as the previous one but operates only over the set $\mathcal{Z}$.

This formulation is particularly important because it falls into a class of problems called bi-level programming problems that is known to be $\mathcal{NP}$-hard. Even the simpler case of linear-linear bi-level programming problem (BLPP) [46] has been demonstrated to be $\mathcal{NP}$-hard. A recent survey on the bi-level optimization problems can be found in [30]. Chapter 7 shows how these considerations have been used in the design of our heuristic algorithm.

The fact that the problem we are addressing is $\mathcal{NP}$-hard means, from a practical point of view, that there exist no algorithm that can solve it exactly (i.e., determine the global optimum solution) in polynomial time. These kind of problems are among the hardest one to solve and historically have been addressed with the use of heuristic algorithms in order to find an approximate solution in a reasonable amount of time. This consideration drove our choices of using an heuristic optimization approach, in particular the choice of a Tabu search with an embedded ILS. The efficiency of the ILS in exploring the neighborhood of the solution identified by the lower level problem allowed us to use a Tabu search instead of a more simple LS to address the upper level problem and escape local optima. More details on the heuristic algorithm are given in Chapter 7. Considering the problem at hand, in many real world scenarios the exact solution is not needed because the formulation of the problem, performed at design time estimating many parameters, might be subject to errors itself. Furthermore, applications deployed in cloud environments often use auto-scaling mechanism that adapt the results of the offline analysis to runtime conditions of the environment.

Chapter 6

Determining an Initial solution

The focus of this chapter is to detail the role of the *Initial Solution Builder* component presented in Section 4.2. As previously introduced, its main goal is to find a promising initial point for the heuristic optimization by using a MILP formulation that can be solved efficiently. To accomplish this goal we go back to the formulation of the problem introduced in Section 5.2 and we provide a closed form formula for performance evaluation constraints. A detailed study of this work has been published in [17].

6.1 Generation of an initial solution

In Chapter 5 we have identified that the problem is *NP*-hard. Furthermore in the formulation of the problem we did not introduce an explicit formula to derive the response time of the application. This omission has been done in order to simplify the formulation and be able to concentrate our attention on the analysis of the structure of the problem itself. To provide a MILP formulation we need to decide which performance model to adopt and derive a closed form formula to apply to Equations 5.14 and 5.15.

As introduced in Chapter 3, we decided to use LQN as performance model due to its ability to model complex software systems and derive accurate performance estimations. One of the main drawbacks of this model is that it is not possible to derive a closed form formula for the average or the distribution of the response time, even for open models. For this reason we have to choose another, more simple performance model to derive an initial solution.

One of the most studied performance model is the *M/G/1*. This model constitutes the base of more complex models, like the LQN and, more importantly, allows us to write in a closed form the expected response time of the application under the hypothesis of open workloads.

Unfortunately the same can not be said for the distribution of the response time, for this reason we can not calculate values for the percentiles (which leads to non-linear models, especially for multi-tier systems).

Since our goal in this stage is to derive a promising initial solution, not to solve the entire problem, we relax constraints on percentiles leaving their evaluation ad satisfaction to the heuristic part.

The more complex LQN model, used in the heuristic optimization of Chapter 4, allows to define a wider ranges of behavior like fork and joins or shared software resources. The main problem of using this model in the formulation of the MILP problem is that it is not possible to derive a closed formula for the response time of the functionalists offered by the application and the utilization of the resources. A theoretical analysis of the differences in accuracy between the two performance models used is not possible because of the lack of this closed formula. It might be possible to perform a comparison by deriving minimum and maximum boundaries of response time from the models but such an analysis would be complex and the closeness of such boundaries to average values, which are used in the formulation of this problem is not known. To overcome this limitation we performed an empirical analysis reported in Chapter 8.

In order to simplify the following discussion, we will not report here the entire problem but only the changes with respect to the formulation of Section 5.2. The objective function of Formula 5.1 and all the constraints from 5.2 to 5.13 are left unchanged, constraint 5.15 is relaxed since the performance model does not allow to write a closed form formula for the derivation of percentiles of the response time. We can then expand constraints 5.14 by using the information derived by the *M/G/1* performance model.

This model is characterized by some parameters that we have to specify in relation with the application in order to derive the required performance metrics. The parameters that have been introduced, in addition to those already presented in Section 5.2 are shown in Table 6.1.

System parameters

Index	
β_k	Proportion of requests of class k in the workload
$\mu_{k,v}^{p}$	Maximum service rate of requests of class k when executed on a resource of type v hosted by provider p
$\mathcal{K}_i$	Set of classes of requests co-located in tier i

Table 6.1: M/G/1 Performance Model Parameters.

The first parameter, β_k is used to characterize the mixture of the incoming workload and represent the proportion of requests of class k. In the general formulation of the problem of Chapter 5 this information was hidden inside the workload Λ_t^p but in order to use the M/G/1 performance model to estimate the average execution time of each functionality k we need to explicitly express this information here. Parameter $\mu^p_{k,v}$ is the maximum service rate of a request of class k when executed of a machine of type v hosted by provider p. The other parameter is related to the application modeled by the user. Recalling the component based modeling approach of Section 3.2, the user can group similar functionalities together in aggregates called components, specify interactions among components and define in which tier each component is installed. Running different components in the same tier generates contention on the physical, or virtual, resource used to host that tier. Parameter $\mathcal{K}_i$ represents the set of all the functionalities that are located in the same tier i and is used to take this contention into consideration.

Recalling the case study application presented in Section 3.2, we can see in Figure 3.6 that the adminServer tier is used to host the Administration Server, the Administration Database and the Agent Manager component. All the load generated by these components will be served by the same processing resource generating contention.

To derive the constraint in Equation 6.19 for the M/G/1 queuing model, we start by partitioning the workload entering a provider, namely Λ^p_t, into the k classes, or functionalities, offered by the application according to probability β_k.

The β_k parameter can be derived by the specification of the Usage Model diagram. Recalling Figure 3.4 of Chapter 3, the Usage Model diagram shows that, as an example, 75% of users perform a call to the *partialRead* functionality offered by the HTTP component.

$$\Lambda^p_{k,t} = \Lambda^p_t * \beta_k. \tag{6.1}$$

The maximum service rate of the system, when processing a request of class k, depends on the speed of the resource used to process it, in order to ease the setup of the problem we can explicitly show this dependency, according to equation 6.2, and use a machine independent maximum service rate μ_k and a scaling factor that depends on the machine S^p_v.

$$\mu^p_{k,v} = \mu_k S^p_v \tag{6.2}$$

In the previous equation S^p_v represents the ratio between the speed of VMs of type v, hosted by provider p, and a reference machine of type e with service rate μ_k. The maximum service rate of a functionality is defined as the inverse of the demand D_k of that functionality, that can be specified by the user in the model of the application. The use of a reference machine eases the definition of the demands and allows to remove the reference to the type of machine on which the demand has been measured. The underlying assumption is that the demand of a request on a machine is proportional to the speed on the machine. Under these considerations S_v^p is given by Formula 6.3.

$$S_v^p = \frac{Speed_v^p}{Speed_e} \tag{6.3}$$

The processing of a request of class k might involve the execution of other functionalities in the system, so the average response time of requests of a certain class is given by summing up the response time of all the sub classes called by that functionality. Looking back at Figure 3.3 of Section 3.2 we see that, as an example, the functionality offered by the administration server to read the configuration of a project generates an internal database access request to the administration database component.

The probability that a request of class k calls a request of class k' is given by $\beta_{k,k'}$ and can be derived by analyzing the model of the application.

$$R_k^p = \sum_{k' \in \mathcal{U}_k} \beta_{k,k'} R_{k'}^p \tag{6.4}$$

Using M/G/1 queues to model tiers, we can write a formulation for the average response time of requests of a certain class as in Equation 6.5:

$$R_{k,v,t}^p = \frac{\frac{1}{\mu_{k,v}^p}}{1 - \sum_{k' \in \mathcal{K}_i} \frac{\Lambda_{k',t}^p}{\mu_{k',v}^p N_{i,t}^p}}, \tag{6.5}$$

where $N_{i,t}^p$ is the number of replicas of the resource hosting the i-th tier, at provider p. The link between $N_{i,t}^p$ and variable $z_{i,v,t}^p$ of the optimization model can be defined as in Equation 6.6. In practice the use of $N_{i,t}^p$ is a shortcut to hide the dependency on the type of resource selected, since only one type of resource is selected for each tier.

$$N_{i,t}^p = N_{i,t}^p \sum_{v \in \mathcal{V}_p} w_{i,v}^p = \sum_{v \in \mathcal{V}_p} w_{i,v}^p N_{i,t}^p = \sum_{v \in \mathcal{V}_p} z_{i,v,t}^p \tag{6.6}$$

The M/G/1 queuing model allows to derive a formulation of the expected average response time only under the condition that the system is not overloaded with requests. This condition has to be added to our model, as Equation 6.7, in order for it to be valid.

$$1 - \sum_{k \in \mathcal{K}_i} \frac{\Lambda_{k,t}^p}{\mu_{k,v}^p N_{i,t}^p} < 1 \tag{6.7}$$

By using 6.1 and 6.2 we can get

$$R^p_{k,v,t} = \frac{\frac{1}{\mu_k S^p_v}}{1 - \frac{\Lambda^p_t}{N^p_{i,t} S^p_v} \sum_{\tilde{k} \in \mathcal{K}_i} \frac{\beta_{\tilde{k}}}{\mu_{\tilde{k}}}} \tag{6.8}$$

As we have seen in Section 3.2, an application is composed by a set of functionalities that cooperate to provide some features. When the user imposed a constraint on a functionality offered by the system it is actually adding a constraint on the execution of a subset of the functionalities internal to the system.

Recalling the Constellation case study of Chapter 3, Figure 3.3 shows that when a request of the readCompleteProjectConfiguration reaches the Administration Server component from an external user, an internal request for a databaseAccess to the Administration Database component is generated. Any constraint specified on the execution of the original request would have to take into consideration both the time spent in the Administration Server and in the Administration Database.

The constraint on the response time of a given functionality exposed by the system, provided as input to the model is represented by $\overline{R}_k$. We first replicate this constraint by applying it on all available cloud providers by means of Equation 6.9.

$$R^p_{k,t} \leq \overline{R}_k \tag{6.9}$$

Unfortunately applying this constraint as is leads to a non linear system (see [53] for further details). To remove this non linearity from the model we split the constraint $\overline{R}_k$ over the functionality exposed by the system into a set of stricter constraints $\overline{R'}_k$ on all the sub-functionalities internal to the system involved in the computation of k, leading to:

$$R^p_{k,t} \leq \overline{R'}_k \tag{6.10}$$

Recalling the example of the case study a constraint on the average execution time of the readCompleteProjectConfiguration of at most one second could be expanded into two constraints of, say 0.5 seconds on the part of the functionality processed in the Administration Server and 0.5 second on the portion executed in the Administration Database. Splitting the time specified in the constraint into the functionalities leads to a more strict problem that might not admit feasible solutions. In order to better generate this splitting, and increase the chance of getting a better initial solution, we decided to make the proportion of the constraint time assigned to each sub functionality dependent with the demand of that sub functionality and the overall chain of calls generated by the user request. We describe in the following how we generate this splitting.

To derive these stricter constraints we analyzed the chain of calls between a functionality, exposed by the system, and all the other functionalities internal to the system. By looking at the model of the application we can construct a Discrete Time Markov Chain (DTMC) using functionalities as nodes and calls between functionalities as arcs. From a given functionality k we can then look at all possible paths that the request can follow within the system, an example of such a chain can be found in Figure 6.1.

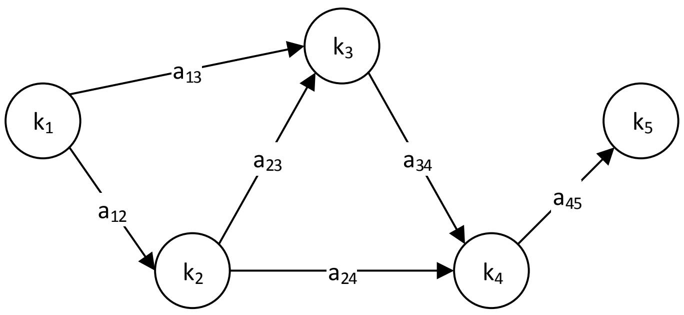


Figure 6.1: Call chain of functionalities

Let U_k be a set of all possible execution paths for requests of class k and designate through $U_{k,k'}$ a subset of all execution paths for requests of class k calling functionality k'. For each edge l, connecting two functionalities k_a and k_b, and for each execution path $u \in U_k$ we will deliver in compliance $p_l(u)$, the probability that functionality k_a calls functionality k_b. Given this chain, the probability that a request of class k flows through path u is $p_k(u)$ and is:

$$p_k(u) = \prod_{l \in u} p_l(u) \tag{6.11}$$

The probability that, given a request of class k a functionality k' is executed can be calculated by summing up the probability that the functionality k' is in any of the execution paths of class k multiplied by the probability of the path to be executed. I.e.

$$p_{k,k'} = \sum_{u:k' \in u} p_k(u) = \sum_{u:c \in u} \prod_{l \in u} p_l(u) \tag{6.12}$$

Looking at Figure 6.1 we can calculate the probability that functionality k_3 is called when a request of class k_1 enters the system, namely p_{k_1,k_3} as $a_{13} + a_{12} * a_{23}$.

Recalling the case study introduced in Chapter 3 we can derive a simple example of such a DTMC from Figure 3.3, such a DTMC is depicted in Figure 6.2. In this example functionalities k_1 and k_2 represent respectively the *get* functionality and the *readCompleteProjectConfiguration* of the *AdministrationServer* component. These functionality allow users to retrieve a list of projects and open a specific one. Both these functionality require an interaction with *AdministrationDatabase* by means of the *databaseAccess* functionality. In this case the interaction is required so the probability that, for example the *get* functionality generates a *databaseAccess* call, represented by $a_1 3$ in the figure, is equal to 1.

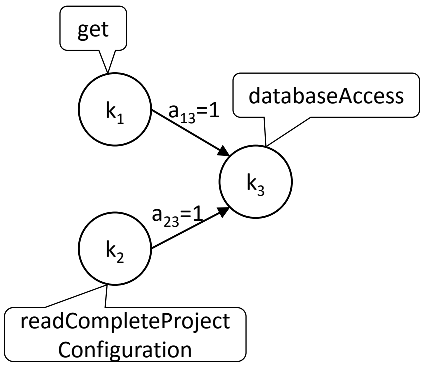


Figure 6.2: DTMC derived by Figure 3.3

In order to derive the response time constraint to apply to the optimization model we decided to split the constraint $\overline{R}_k$ defined by the user into a set of $\overline{R'}_k$ constraints on the functionalities included in the computation of k proportionally to the demand of each sub functionality, taking into consideration all possible execution paths. To this end we first derive the overall demand of an execution path by summing all the demands of the functionalities on that path, I.e. $D_{k,u} = \sum_{k' \in u} D_{k'}$ then we derive the proportion of the demand for all the functionalities by dividing the given demand by the demand of the entire execution path, as in Formula 6.13:

$$r_k(u) = \frac{D_k}{D_{k,u}} = \frac{D_k}{\sum_{k' \in u} D_{k'}} \tag{6.13}$$

Recalling formula 6.3 we can express the maximum service rate of a class μ_k according to the speed of the reference machine e as in Formula 6.14:

$$\mu_k = \frac{Speed_e}{D_k} \tag{6.14}$$

.

By using this proportion to split the user defined response time constraint $\overline{R}_k$ across functionalities in the call chain we get a set of new constraints on the response time of each functionality k' when executed within the execution path u is:

$$\overline{R'}_{k'}(u) = r_k \overline{R}_k \tag{6.15}$$

since a request of class k can have multiple execution paths, for each functionality involved in its execution chain we use the most stringent constraint to remove the dependency of the constraint to the specific execution path of class k.

$$\overline{R'}_k = \min_u \overline{R'}_k(u) \tag{6.16}$$

The model obtained with this approach is linear and can be efficiently solved by state of the art tools.

Re-writing equation 6.10 with 6.5 we get the following formulation for the response time constraint:

$$\frac{\frac{1}{\mu_k S_v^p}}{1 - \frac{\Lambda_t^p}{N_{i,t}^p S_v^p} \sum_{\tilde{k} \in \mathcal{K}_i} \frac{\beta_{\tilde{k}}}{\mu_{\tilde{k}}}} \leq \overline{R'}_k \tag{6.17}$$

After some algebra we get:

$$\Lambda_t^p \mu_k \overline{R'}_k \sum_{\tilde{k} \in \mathcal{K}_i} \frac{\beta_{\tilde{k}}}{\mu_{\tilde{k}}} \leq \mu_k \overline{R'}_k S_v^p N_{i,t}^p - N_{i,t}^p \tag{6.18}$$

Recalling 6.6, we can express explicitly the dependency with variable $z_{i,v,t}^p$ obtaining the formulation of the constraint reported as Equation 6.19:

$$\sum_{v \in \mathcal{V}_p} (1 - \mu_k \overline{R'_k} S_v^p) z_{i,v,t}^p \leq \Lambda_t^p \mu_k \overline{R'_k} \sum_{\tilde{k} \in \mathcal{K}_i} \frac{\alpha_{\tilde{k}}}{\mu_{\tilde{k}}} \qquad \forall t \in \mathcal{T}, \forall k \in \mathcal{K}, \forall p \in \mathcal{P} \tag{6.19}$$

To evaluate the quality of the initial solution derived by this approach we have performed several experiments. We have seen that using this approach over defining an initial solution using current best practices allows to reduce significantly the time required for the entire optimization. More details on the effectiveness of using the solution derived by this approach as starting point for the heuristic search algorithm are presented in Chapter 8.

Chapter 7

Meta-heuristic Approach

In this Chapter we present the meta-heuristic algorithm expressly designed to solve the optimization problem presented in Section 6.1. We choose to use a meta-heuristic algorithm as a consequence of the analysis of the mathematical formulation of the problem presented in Chapter 5. The outcome of that analysis showed that the problem we are dealing with is NP-hard; meta-heuristics algorithms have been shown to be very effective in tackling these type of problems. The analysis presented in Section 5.3, also showed that this problem has a particular bi-level structure. For this class of problem a solution to the lower level problem can be found for each assignment of the variables of the higher level problem. We decided to exploit this characteristic of the problem to design an heuristic algorithm divided into two main phases each one tackling a particular level of the problem. The development of the heuristic approach presented in this chapter has been an iterative process during which the algorithm has evolved and new features have been added to better guide the exploration of the search space. During the development we tried many alternative procedures and policies in different parts of the algorithm in order to evaluate their effectiveness in this particular context. In the remainder of this chapter we will illustrate the choice we made and briefly explain other policies that we tried and were not included in the final version.

The final outcome of this process is a heuristic algorithm that has been tailored in most of its parts to effectively tackle this kind of bi-level optimization problems and has been shown to be effective in exploring the space of possible deployment configurations of cloud applications.

Our approach is composed by a main optimization procedure shown in Algorithm 1 and described in Section 7.1. This algorithm make use of three main procedures: *MakeFeasible*, shown in Algorithm 2 and described in Section 7.2, to assess the feasibility of a solution; *ScaleLS*, shown in Algorithm 3 and described in Section 7.3, to find the optimal number of replicas of resources used to host each tier; *TSMove*, shown in Algorithm 4 and described in Section 7.4, to change the type of resource assigned to each tier. Two other procedures are used to tackle specific problems. The *OptimizeWorkload* procedure, shown in Algorithm 6 and described in Section 7.6, is used to derive load balancing decisions in a multi-cloud scenario. The *Restart* procedure, shown in Algorithm 5 and described in Section 7.5, is used to build a new solution when the optimization of the higher level problem is trapped in a local optimum.

7.1 Main Algorithm

This section presents the main structure of the heuristic algorithm proposed to solve the cloud deployment optimization problem. The pseudo-code for the main procedure can be found in Algorithm 1. This algorithm implements a *Tabu Search* (TS) [40] heuristic in order to explore the space of possible solutions. The TS technique is very effective in solving minimization or maximization problems in which the objective function to minimize has some valleys of local optima, as shown in Figure 7.1. As shown by the bi-level formulation of the problem of Section 5.3, the lower level problem presents at least one optimum for each assignment of variables of the upper level problem. Each of the optima of the lower level problem are local optima for the entire problem, for this reason the adoption of these kind of algorithms, able to escape local optima is required.

Traditional local search approaches [74] used to find a minimum value of a function, the configuration of cloud services with the minimum cost in our context, start from an initial solution, depicted in the upper left corner of Figure 7.1, generate new solutions by means of some *move* or action that modify the initial solution and then replace the initial solution with the newly created one if its value, the cost in our problem, is smaller. This simple approach has been modified and implemented in several techniques that focus on finding very effective moves to generate candidate solutions and reduce the number of iterations, and consequently the time, required to reach the optimum. The main limitation of local search is that when a local optimum is reached the set of moves used to modify the solution might not be able to generate solutions with a lower cost and the optimization ends without reaching the global optimum. To escape these situations many techniques have been studied. Some techniques, like those presented in [75], restart the entire optimization process from a different initial solution, on the extreme right side of the picture for example, and using as final solution the best solution found by both the optimization runs. Other approaches try to change the way neighboring solutions are generated either by applying different moves in the hope of generating a solution capable of escaping the attractive basin of the local optimum.

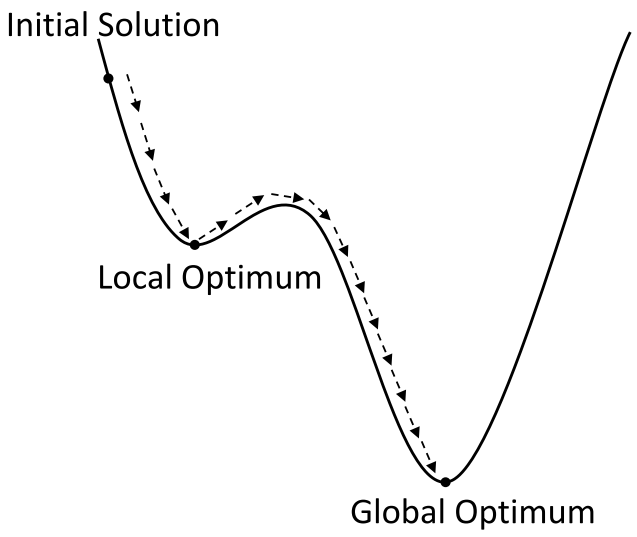


Figure 7.1: Tabu Search behavior in presence of local optima

The TS algorithm solves the local optimum problem by accepting solutions with a higher cost in case none of the solutions found in the neighborhood of the current solution has a lower cost. Referring again to Figure 7.1 a traditional local search would only descend the leftmost part of the slope and stop into the local optimum while the TS approach would accept solutions with higher costs climbing the slope on the right side of the local optimum and reaching the valley in which the global optimum sits.

Accepting solutions with a higher cost leads to a series of issues that have to be addressed in order to make the TS an effective algorithm. First of all, we need to specify an exit criteria that define when the search of new solutions should stop. Then, we need to save the best solution found during the entire optimization, a process called *elitism*, in order to provide it as result of the optimization process. Finally, we need to avoid to generate twice solutions that have already been evaluated. Intuitively if we accept a solution with a higher cost near a strong local optimum, our optimization process might be led back to the local optimum already visited. The most effective way to avoid this kind of behavior is to introduce a short term memory that stores the most recent visited solutions, or some equivalent information of the last steps of the exploration, and inhibits the generation of solutions already visited.

The effectiveness of this technique in exploring complex solution spaces motivated us to adopt it in the context of our problem and tailor the moves that generate neighborhood solutions by considering the two levels structure identified by the analysis of Chapter 5. In particular we have delegated the exploration of the lower level problem into one of the moves of the TS. This move implements another heuristic algorithm called *Iterated Local Search* (ILS) to solve the particular instance of the lower level problem defined starting from the solution identified by the upper level.

Since our approach embeds an ILS into a TS it can be classified as an hybrid heuristic algorithm according to the classification in [74]. More information on the classification of our approach and the comparison with other heuristic approaches can be found in Chapter 2.

The main element of this algorithm, and of most of the algorithms in this chapter, is an object called $Solution$. This object represents a deployment configuration, either in a single or multi cloud scenario. As a matter of fact, in the more general case of an application to be executed on multiple clouds the $Solution$ object contains the decisions about *workload partitioning*, *resource type selection* and *allocation*, and *resource replicas* needed to define a complete deployment over 24 hours. During the optimization process, three main instances of the solution are always kept alive: the *Current* solution, which is the one that changes more frequently as different moves are applied to this instance in order to modify it and explore different regions of the search space.

The *Best* solution, instead, is the finest solution found by the optimization algorithm. During each iteration of the main optimization loop the *Current* solution is compared against the *Best* solution (on line 9). If the comparison reveals that the *Current* solution shows some improvement with respect to the *Best* one we promote that solution to *Best* and save it. This process is called *Elitism*, and allows the algorithm to work on sub-optimal solution while keeping track of the the best solution.

The third main solution used in the optimization process is called *LocalBest*. Its role is similar to the one of the *Best* solution in the sense that it stores the best solution found in a subset of the optimization space. The main difference with the *Best* solution is that this solution is reset when the optimization moves to a different area of the search space, when the algorithm is exploring a particular subset of the search space this solution is used to restart many procedures and focus the search process toward promising solutions in that particular region. On the contrary, the *Best* solution keeps track of the cheapest solution generated during the entire optimization process and is never reset but only updated when a better solution is found.

Algorithm 1: Optimization Algorithm

```
Input  : $MaxIter$                          // Maximum number of iterations
Output: $Best$                                        // Optimized solution
1  $Current \leftarrow$ MILP()
2  $Best, LocalBest \leftarrow Current$
3  $BestUpdated \longleftarrow$ True
4  $Iter \leftarrow 0$
5  $MemST, MemLT \leftarrow ()$      // Initialization of memory structures
6  while $Iter < MaxIter$ do
7      if $Current$ is not Feasible then
8          $Current \leftarrow$ MakeFeasible($Current$)              // Repair action
9      $Current \leftarrow$ ScaleLS($Current$)          // Lower level local search
10     $LocalBestUpdated \longleftarrow$ False
11     if $Current < LocalBest$ then   // Update the local best solution
12         $LocalBest \leftarrow Current$
13         $LocalBestUpdated \longleftarrow$ True
14     if $\#Providers > 1 \wedge LocalBestUpdated$ then   // Multiple clouds are used
15         $Current \leftarrow$OptimizeWorkload($Current$) // Redistribute workload
16         $Current \leftarrow$ ScaleLS($Current$,$LocalBest$)
17         $LocalBest \longleftarrow Current$
18     if $Current < Best$ then                  // Update the best solution
19         $Best \leftarrow Current$
20     $Current \leftarrow LocalBest$
21     $Candidate \leftarrow$ TSMove($Current$, $MemST$)             // New candidate solution
22     if $Candidate = Current$ then
23         $Candidate \leftarrow$ Restart($Current$, $MemLT$)            // Aspiration mechanism
24         $LocalBest \leftarrow Current$
25     $Current \leftarrow Candidate$
26     UpdateMem($MemST$,$Current$) // Updating the short term memory
27     UpdateMem($MemLT$,$Current$)  // Updating the long term memory
28     $Iter \leftarrow Iter + 1$
```

More details on the use of this solution are provided in the description of the *TSMove* in Section 7.4, for the moment it is sufficient to notice that this solution, unlike the *Best* one, is reset when the *Restart* procedure is applied (on line 23).

The comparison between two solutions, represented in the algorithm via the operator $<$, takes into account different factors. Given two solutions A and B, we say that $A < B$ if any of the following occurs:

- A is feasible and B is not
- Both A and B are not feasible and the number of constraints violated by A is smaller than those violated by B
- Both A and B are feasible and the cost of A is smaller than the cost of B

In order to simplify the representation of the algorithm we avoided to explicitly address the multi-cloud scenario, with the exception of Algorithm 6. This simplification is due to the fact that all the procedures described in this chapter are applied independently to all of the providers. In order to avoid the complexity of adding a third level to the optimization problem and explorer we decided to tackle the problem of splitting the workload in a slightly different way. We first relay on the splitting of incoming workload for each hour of the reference day specified in the initial solution derived by the problem presented in Chapter 6. We solve the lower level problem using the workload split defined in the initial solution and then redistribute the workload towards more utilized providers with the goal of removing some resource from less loaded cloud providers and reduce the overall cost. More details on this phase are provided in Section 7.6.

Finally, two other main structures used our approach are the *Short Term* and *Long Term* memory. As previously said, memory structures are often adopted in a TS algorithm in order to avoid the generation of solution that have already been evaluated. We decided to use the same approach by introducing the *Short* term memory that stores the 10 most recent visited solutions. We have performed several test run of the algorithm before choosing the size of the short term memory and even if storing a single solution is not expensive in terms of memory consumption, given the very small footprint of the solution, values greater than 10 did not produce any significant change in the behavior of the algorithm. As said in the introduction to this chapter, the development of the heuristic algorithm has been performed in an iterative way constantly evaluating our choices by optimizing some synthetic models. When we started performing evaluations on multi-cloud deployments on providers that offer a large number of resource types we found out that the use of a short term memory was not always sufficient to avoid cycles in the search path. For this reason we introduced a second layer of memory called *Long Term* memory. This layer stores statistics about all the solutions that have been evaluated during the search process and is used to reconstruct a new solution that has never been visited when the *Short Term* memory becomes ineffective. More details on the use of these memory structures are provided in Sections 7.4 and 7.5.

The main optimization algorithm starts with the generation of the initial solution from the MILP formulation of the relaxed problem (on line 1). This solution is used to initialize both the $Best$ and $LocalBest$ solutions. The two memories structures called short term memory, or *MemST*, long term memory, or *MemLT*, are then initialized.

The TS is an iterative algorithm, and the main optimization loop, shown from line 6 to 28, is executed up to a maximum number of iterations.

The number of iterations of the TS algorithm can be specified by the user in order to control the duration of the optimization process. As a general rule, applications with a higher number of tiers require more iterations since they require a broader exploration of the space of the higher level problem.

If the number of iterations is set to 1, the optimization procedure will not take into consideration the effect of the *TSMove* and, as a result, the optimization process will be limited only to seek for the optimal number of replicas and, in a multi-cloud scenario, the workload balancing.

The first part of the optimization loop (lines 6-25) tackles the lower level problem. The process starts by assessing the feasibility of the *Current* solution and if necessary repairing it by means of the *MakeFeasible* procedure, on line 8, described in Section 7.2. This operation is needed because the *ScaleLS* (applied next) only operates on feasible solutions.

After the feasibility of the solution has been assessed the *ScaleLS* procedure, presented in Section 7.3, is applied on line 9. This procedure operates on the lower level problem by reducing the number of replicas in order to minimize the operational cost of the solution preserving, however, its feasibility. The solution derived by this procedure is optimal in the context of the lower level problem.

We then proceed by comparing the updating the $LocalBest$ solution with the $Current$ one and, if necessary updating it.

At this point the algorithm makes a distinction between solutions defined on a single or on multiple clouds. If the user specified deployment model makes use of a single cloud, then no further action is needed and the optimization proceeds. If, on the other hand, the deployment refers to more than one cloud provider, the algorithm needs to address the additional problem of distributing the workload. To avoid dealing with the problem of workload distribution for every solution considered, we decided to focus the effort of this procedure by applying it only to locally optimal solutions. We decided to perform this operation only after the solution of the lower level problem because it is a time consuming process that analyzes several times the lower level problem for different workload configurations.

This operation is performed in the *OptWorkload* procedure, described in Section 7.6 and performed at line 15. This procedure operates on the *Current* solution without affecting the number of instances derived in the previous step, rather it tries to bias the distribution of the workload towards underutilized providers, still preserving the feasibility of the solution, removing load from providers that are already under utilized. The reduction of the load on these providers give the possibility to the *ScaleLS* procedure, applied again on line 16, to further reduce the number of instances used increasing their utilization but reducing the overall cost of the solution. The *OptWorkload* and the *ScaleLS* procedures generate only feasible solutions, even if they work internally also with unfeasible solutions, for this reason the $Current$ solution derived after the optimization of the workload is feasible and contains at most the same number of replicas of resources of the $LocalBest$ solution so we can safely update it.

At this point the $Best$ solution is compared to the $Current$ one and eventually updated.

The second part of the optimization loop (lines 25-28) deals with the upper level problem. The search process starts from the *LocalBest* solution, that is copied over the *Current* solution, this action drives the search to new types of services to explore sooner regions that are close to the *LocalBest* solution. Focusing the search in the proximity of the *LocalBest* solution allows to limit the diversification effect of the restart action and better explore regions of the search space surrounding the *LocalBest* solution.

The *TSMove* procedure, described in Section 7.4, is applied in order to generate a *Candidate* solution containing a new service choice for one of the tiers. This procedure makes use of *MemST* in order to avoid re-generating solutions that have already been analyzed.

If the space surrounding a *LocalBest* solution has been fully explored the effect of the *TSMove* is no more sufficient to generate a new solution and escape the local optimum. If this situation occurs, it is necessary to restart the search from a different region of the space, or in a more extreme way to restart the entire optimization process. In a more general way the criteria applied to allow the optimization process to escape from the space already explored is called *aspiration criteria*. This operation is performed by applying the *Restart* procedure applied on line 23 and presented in Section 7.5. The procedure constructs heuristically a new solution by using the frequency of assignments of services to tiers, such information is contained in the *MemLT*. If the *Restart* procedure is applied then the *LocalBest* solution has to be invalidated, i.e., overwritten by the new *Candidate* solution, since the search has been moved into a different region. Before proceeding to the next iteration of the optimization loop both *MemST* and *MemLT* are updated in order to keep track of the search path and avoid generating again solutions that have already been visited. More details on the use of these memory structures are provided in Section 7.4.

7.1.1 Solution Evaluation

One of the core operations required by all the optimization procedure presented in this chapter is the evaluation of a candidate solution. The evaluation is a complex process that requires interactions with many components of the architecture presented in Chapter 4. It is composted by two main phases: the cost evaluation and the feasibility evaluation. The cost evaluation part looks at the type and number of resources specified in the solution and interacts with the resource model database to retrieve the relative cost, which might also be dependent on the time of the day, and calculate the final cost of the deployment configuration. The feasibility evaluation is the more complex and time consuming, in fact most of the time required for the optimization is spend in this phase. When a solution is evaluated the LQN model that represents the application is updated to reflect the deployment information of that solution, i.e., resource type and number of replicas. The LQN model is then evaluated by LINE, the results of the evaluation are parsed and finally the constraints defined by the user are checked. Since the evaluation of a LQN model is a compute intensive and time consuming task, many optimizations, described in Chapter 4, have been implemented in order to speed up the evaluation process. In particular a caching layer has been introduced in order to re-use partial evaluations of previous solutions.

7.2 MakeFeasible

The *MakeFeasible* procedure, detailed in Algorithm 2, is used by the main optimization algorithm to restore the feasibility of a solution in the context of the lower level problem as described in Section 7.1. The objective of this procedure is not to solve the lower level problem, since this task is delegated to the *ScaleLS*, but rather to guarantee the feasibility of a solution, paving the way for the execution of the *ScaleLS* procedure. This operation is performed in two different phases of the optimization process. It is used first to analyze the initial solution that is derived, as shown in Chapter 6, using a different performance model and might be unfeasible. It is then used after the *TSmove* changes the assignment of a resource type to a tier. This action might lead to unfeasible solutions since the processing power of newly selected resource might be lower than the original one and require a larger number of replicas to fulfill QoS constraints.

In essence, this procedure embodies an iterative process that gradually increase the number of resource replicas used to host each tier of the application by multiplying the original number for a given factor. Such factor is initialized at the beginning of the procedure to a default value and adapted during the optimization. The default values of 5 for the *MaxFactor* and 1.2 for the *MinFactor* have been chosen by performing some preliminary experiments which have proven to be effective for most of reasonable size instances.

Algorithm 2: MakeFeasible

```
Input  : $Solution$;                          // Candidate Solution
         $Constraints$;                       // Set of Constraints
         $MaxIter$                // Maximum number of iterations
Output: $FeasibleSolution$;                    // Feasible Solution
1  $Iter \leftarrow 0$
2  $MaxFactor \leftarrow 5$
3  $MinFactor \leftarrow 1.2$
4  $T = [1..24]$
5  $Feasibile \leftarrow$ Evaluate($Solution$)
6  if ViolatedConstraints($Solution$) contains RAM or Type constraints then
   return;
7  while $\neg Feasible \wedge Iter < MaxIter$ do
8      $$Factor \leftarrow MinFactor + \frac{(MaxFactor - MinFactor)}{MaxIter} \times Iter \quad (7.1)$$
9      foreach $time \in T$ |$Solution(time)$ is not feasible do
           // Select the tiers according to the policy
10         $Tiers \leftarrow$ getCriticalTiers($FailingConstraints$)
           // Check the constraints on the maximum
              replicas
11         $Tiers \leftarrow$ filterTiersByMaxReplicaConstraints($Tiers$,$factor$)
12         foreach $tier \in Tiers$ do
13             ScaleOut($tier$,$factor$)
14     $Feasible \leftarrow$ Evaluate($Solution$)
15     $iteration++$
16 $FeasibleSolution \leftarrow Solution$
```

At the beginning of the procedure, the input solution is evaluated in order to assess its feasibility (on line 5). If the solution is already feasible then no adjustment is needed and the procedure can terminate immediately. If, on the other hand, the solution is not feasible an initial check is performed to inspect which constraints are violated. As said, this procedure operates in the context of the lower level problem, since it does not change the type of services used to host each application tier. For this reason, this procedure should be applied only if the violated constraints are related to CPU utilization, Response Time or Minimum and Maximum number of replicas; any other type of constraint violated by the solution, like the constraint of the minimum amount of memory, will not be affected by this procedure and their resolution is delegated to the *TSMove* procedure. This condition is checked by line 6, if an architectural constraint (e.g., a constraint on the minimum amount of memory of VMs used to host a tier) is not met the procedure is aborted. The procedure delegated to the solution of this type of problems is the *TSMove* which is applied before beginning the next iteration of the main optimization algorithm.

The first operation performed during each iteration of the main optimization loop, is an update of the factor used to control the replicas increase on selected tiers. Multiplying the number of replicas of a resource by a constant factor during each iteration leads to an exponential growth of the replicas. This behavior is desirable since allows the algorithm to find a feasible solution in a small number of iterations but at the same time it moves the actual number of replicas far from the optimal one, implying more work for the *ScaleLS*, which is appointed to optimize the number of replicas of the solution which requires many interaction with the LQN solver. By starting the process using a small factor in early iterations and increasing it after each iteration in which the solution is not yet feasible, we reduce the impact of the exponential growth and the amount of work required by the *ScaleLS*. The factor is then increased linearly as shown in Equation 8 in order to address solutions that require a higher number of replicas.

Once the new value of the multiplication factor has been updated, the algorithm proceeds by selecting which tier to scale; the scaling action is then applied independently for each hour of the reference day for which a feasible solution has not been found. In order to select the tiers to scale we make use of the *getCriticalTiers* procedure (on line 10). This procedure looks at each constraint violated by the solution and selects all the tiers that affect the metric specified in the constraint. As an example, if the violated constraint is related to the CPU utilization of the resource assigned to a particular tier then the selection of the tier to scale is straightforward. If, on the other hand, such constraint concerns the average response time of a functionality depending on two or more tiers, choosing the one to scale is not a trivial task. In order to deal with this kind of situations we have implemented and evaluated different policies:

a *Random* selection of one among the affected tiers is the simplest choice and it has been used as a reference for the evaluation of other policies; a more advanced policy consists in using the information on the breakdown of the execution time, the time used to process a request, on all tiers involved in the computation and choosing the one in which the request requires on average the *longest* time; finally, we considered the *utilization* of each of the affected tiers in order to take into consideration not only the single functionality but the entire system and selected the tier with the highest utilization.

Testing these policies in several experiments, we have found out that the policy considering the *utilization* of each tier allows to better identify the bottleneck of the system, leading to a reduction of the number of iterations needed to find a feasible solution. Intuitively this can be explained by the fact that, in many situations, the demand of the entire application is not evenly distributed across all the tiers. In such a scenario, if the tier in which most of the computation is performed is lightly loaded, scaling it will not have a great impact on the overall execution time. The final outcome of the selection of tiers is a list of all the tiers that affect some unfulfilled constraints.

Before increasing the number of replicas of these tiers we have to make sure that our decision on the new number of replicas does not violates constraints on the maximum number of replicas. For this reason we remove from the list of tiers those that are already using the maximum number of replicas by means of the *filterTiersByMaxReplicaConstraints* procedure.

Finally, each remaining tier is scaled (on line 13) and the new solution is evaluated.

7.3 ScaleLS

The *ScaleLS* (Scale Local Search) procedure implements one of the most important functionalities of the entire algorithm. It focus on the solution of the lower level problem defined in Chapter 5. The main goal of each iteration is to reduce the number of replicas of the resources assigned to a single application tier in order to reduce its cost while maintaining the feasibility of the solution. Since this procedure operates on the lower level problem, we can safely solve independently the problem of finding the exact number of replicas for each of the 24 hours. The only dependency on the structure of the solution that spans all the 24 hours is the assignment of the type of resource to a tier but this decision has already been taken in the upper level problem and, in the context of this procedure, is fixed.

Algorithm 3: ScaleLS

```
Input  : $Solution$;                          // Candidate Feasible Solution
         $LocalBestSolution$;                 // The local best solution
         $DefaultFactor$;                     // Initial ScaleLS factor
         $NonImprovementLimit$;   // Maximum number of iterations without
improvements
         $OOBLimit$; // Maximum number of iteration in which the solution
is far from the local best
         $InitialBoundFactor$;                // Initial bound factor
Output: $OptimizedSolution$;                  // The Optimized Solution
1  $Iterations \longleftarrow 0$;
2  $NoImprovementIterations \longleftarrow 0$;
3  $OOBIterations \longleftarrow 0$;
4  $BoundFactor \longleftarrow InitialBoundFactor$;
5  $AdjNonImpLimit = NonImprovementLimit \times NumberOfTiers$;
6  set $T = [1..24]$;
7  for $t \in T$ do
8      $Factors[t] = DefaultFactor$;
9      $UnfeasibleIterations[t] = 0$;
10 while
   $NoImprovementIterations < AdjNonImpLimit \vee OOBIterations < OOBLimit$ do
11     $Iterations++$
12     $Scaled \longleftarrow false$
13     $PreviousSolution \longleftarrow Solution$
14     for $t \in T$ do
15         $Tiers \longleftarrow$ FindScalableTiers($Solution$,$t$)
16         if $Tiers$ is empty then continue;
17         ScaleIn($Tiers(random)$,$Factors(t)$)
18         $Scaled \longleftarrow true$
19     EvaluateSolution($Solution$)
20     for $t \in T|$ GetHour($Solution$,$t$) is not feasible do
21         restoreHourSolution($Solution$,$t$,$OriginalSolution$)
22         $UnfeasibleIterations[t]++$
               $$Factor[t] \longleftarrow 1 + \frac{(DefaultFactor - 1)}{UnfeasibleIterations[t]} \quad (7.2)$$
23     if $Current < LocalBest$ then
24         $LocalBest \leftarrow Current$
25     if $\neg Scaled \vee$ cost($PreviousSolution$) <= cost($Solution$) then
26         $NoImprovementIterations++$
27     if cost($Solution$) - cost($LocalBestSolution$) >
       $BoundFactor \times$cost($LocalBestSolution$) then
28         $OOBIterations++$
30
29     else
               $$BoundFactor \longleftarrow \frac{BoundFactor}{Iterations - OOBIterations} \quad (7.3)$$
```

This function implements an Iterated Local Search (ILS) paradigm that operates starting from a feasible solution reducing the number of replicas of all the application tiers in order to produce a lower cost solution that still fulfills all the constraints. Operating on the lower level problem only this procedure does not modify the type of resources chosen by the upper level problem, since this task is delegated to the TSMove. The ILS heuristic is a specification of the LS heuristic. The main difference between the TS presented in Section 7.1 and the ILS is how the two procedures deal with local optima. As previously introduced, the TS tries to avoid local optima by changing the way the solution is modified and temporary inhibiting some changes by using a memory structure in order to escape local optima. On the other hand, ILS tries to escape local optima by restarting several times the optimization procedure from different initial solutions. In order to drive the execution of the procedure toward more promising regions of the search space the restart action might use some information derived by solutions generated by previous iterations of the optimization.

Algorithm 3 shows the structure of the procedure. The main input parameter is a candidate feasible solution identified by the variable *Solution*. In the context of our hybrid optimization approach this solution is generated by the *MakeFeasible* procedure described in Section 7.2. Another important input is the *DefaultFactor*, this parameter is used to define the intensity of scaling actions. The procedure operates by removing a certain number of replicas of resources in order to reduce operational costs but if too many replicas are removed the derived solution might violate some QoS constraint. If this situation occurs the procedure is restarted and a smaller number of replicas are removed. The *DefaultFactor* controls how many replicas are removed during the first iteration of the procedure. All other parameters shown in Algorithm 3 are used to define the termination condition of the search procedure.

The main optimization loop has two exiting conditions, shown on line 10: the first one estimates the distance between the current solution and the (unknown) optimal one by taking into consideration how many iterations of the optimization procedure have been performed without generating a better solution; the second one considers the distance between the current solution and the *LocalBestSolution* identified by the main optimization procedure in order to terminate the optimization if the current solution is too far from the best one.

When an optimal solution is found, no further reduction is possible. Intuitively the closer the solution gets to the optimal configuration, the harder becomes to find the right number of resources to remove without violating any constraint. The *NoImprovementIterations* variable keeps track of this information by counting the number of iterations of the main optimization loop that have been executed without affecting the solution.

It is a common practice in many local search approaches, in which the distance of a candidate solution to the optimal one can not directly be computed, to use as a stopping criteria the time spent on the optimization or, as in our case, the number of consecutive iterations that did not lead to any progress. To this end, we use the *NonImprovementLimit* as an exit condition. Its value is not directly used to upper bound the number of iterations but it is multiplied by the number of tiers in the application, as shown on line 2. The dependency of the maximum number of iterations with the number of tiers of the application can be intuitively explained by the fact that the *ScaleLS* procedure operates on a single tier during each iteration. Since the tier selected for the scaling action is chosen randomly, we decided to multiply the number of maximum iterations by the number of tiers to give the procedure a better chance to fully explore solutions with a complex structure. When the *NoImprovementIterations* reaches the *NonImprovementLimit* threshold the algorithm has reached a local optimum and needs to be restarted.

If we look more closely to the structure of our entire optimization problem, we will see that the result of the optimization process on the lower level problem strongly depends on the decisions taken at the upper level, as discussed in Chapter 5. Indeed, if the Solution provided to the *ScaleLS* procedure contains a poor choice on the type of resources to use, relying only on the exit condition described above might lead the optimization to spend a considerable amount of time to find the exact amount of replicas for a solution that will be discarded later on as too expensive. It is easy to see that any exit condition that only operates on the current solution, and its own progress towards the optimum, can not detect such a situation. For this specific reason we introduced a second exit condition from the main optimization loop. This condition operates as a shortcut by allowing the optimization to end before reaching the maximum number of iterations without improvements if the solution we are considering is far enough from the *LocalBestSolution*.

The *OOBLimit* (Out Of Bound Limit) and *InitialBoundFactor* are used to control how the second exiting condition operates. In order to identify whether a solution is far from the *LocalBestSolution* we look at the difference of the cost of the two solutions. If this difference is higher than a certain threshold we can speculate that successive reductions of the number of replicas are not likely to bring the solution close enough to the *LocalBestSolution*. This bound is identified in line 27 by considering the cost of the *LocalBestSolution* and multiplying it by a *BoundFactor*. Furthermore, the factor we are using is not static but evolves during the optimization process. After each iteration in which the candidate solution was considered close to the *LocalBestSolution*, the dimension of the bound is reduced. If, after the next iteration, the cost of the solution is too high and falls out of the new bound then the BoundFactor is not changed, allowing the solution to further evolve and reduce its cost. Reducing the $BoundFactor$ according to Formula 30 makes it approach zero as the number of iterations, in which the solution was close to the *LocalBestSolution*, increases. This behavior allows to focus the search to solutions that are likely to perform better than the *LocalBestSolution* and avoid solutions that are very far apart. A simpler approach that does not take into consideration the distance of the current solution from the *LocalBestSolution* has also been considered, we have discarded that approach because of its inability of considering the effect of the lower level optimization with respect to the entire problem. Ignoring this distance the *ScaleLS* procedure would spend a lot of time in finding the global optimum of the lower level problem for solutions that are very far from the *LocalBestSolution* because of a bad choice performed in the upper level problem and are be discarded later on in the search process. Using the adaptive behavior described in this paragraph allowed us to terminate early the *ScaleLS* procedure and reduce the time required for the optimization.

In order to identify which tiers can be scaled we implemented a procedure called *FindScalableTiers*, applied in line 14. In principle, each tier can have a minimum of one replica, this limitation can be altered by a user specified constraint, as shown in Chapter 3. The *FindScalableTiers* procedure filters out all the tiers for which the minimum number of replicas has already been reached. A random tiers is selected among those identified by the previous operation and its number of replicas is divided by a scaling *Factor*.

Dividing the number of replicas during each iterations by a constant factor leads to an exponential reduction on the number of replicas in the solution. This behavior leads to a quick convergence to solutions that have a high number of replicas but when we get close to the optimal such a strong reduction is likely to lead to unfeasible solutions. For this reason we adapt the scaling factor during each iteration as shown in Formula 7.4 reported in line 22.

$$Factor[t] = 1 + \frac{(DefaultFactor - 1)}{UnfeasibleIterations[t]} \tag{7.4}$$

The factor is reduced when the number of iterations that led to unfeasible solutions increases, up to a point in which every successive iteration will remove just a single replica from the selected tier. It is worth to notice that if an iteration did not produce an unfeasible solution then the factor is not updated. Using the same factor for several reductions results in exponential decrease of the number of replicas which make solutions with a large number of replicas approach the optimum quickly.

Since this operation is applied to all of the 24 hourly problems in parallel, we use 24 scaling factor in order to make the magnitude of the scaling actions independent.

To better exploit the ability of the performance solver to analyze multiple performance models in parallel, we evaluate the entire solution only after all the 24 hours have been updated. The reduction of the number of replicas might lead to a solution in which some of the hours are feasible but others are not. In order to save the progress of the feasible hours we restore the feasibility of the entire solution by reverting the failing hours to their original state, using a copy of the solution saved at the beginning of each iteration (see line 13). This operation is performed by the *restoreHourSolution* function in line 21.

When the feasibility of the solution has been restored, we compare it with the *LocalBest-solution* and, if the cost of the new solution is lower, we update it.

7.4 TSMove

The *TSMove* (Tabu Search Move) procedure, shown in Algorithm 4, constitutes the main action of the global tabu search implemented by Algorithm 1. It allows to change the type of resources used to host the tiers of the application and explore the space of the upper level problem. The procedure starts by building a list of all the tiers composing the application and randomizing its order (steps at lines 1- 2)

The randomization is performed in order to select randomly a tier to which apply the changing action. In some situations, that will be highlighted in a few paragraphs, the only type of resources that can be used to host a certain tier is the one already present in the solution. If there is no other choice for the selected tier the procedure will skip this tier and use the next one on the list. If none of the tiers can be changed then the procedure is not able to modify the solution and the *NoChanged* flag is raised, at line 20, to inform the main optimization procedure that a stronger action is needed.

When the tier has been selected the procedure continues by building a list of all the possible resources that could be used to host that tier, on line 5. To build this list the *getCandidateResources* procedure interacts with the Resource Model Database, as shown in Chapter 4, to retrieve a list of resources among which to choose the new service. Such list is built by analyzing the currently selected resource and querying the database for services that have the same provider, e.g. Amazon, and the same type, e.g. VMs. For example, if the resource assigned to the selected tier in the *Solution* is a VM of the provider Amazon of size m3.large, this functionality will retrieve all the other sizes of VMs offered by Amazon. This list is then filtered, on line 6, in order to meet architectural constrains; these constraints, introduced in Chapter 3, includes for example constraints on the minimum amount of RAM, if the selected resource is a VM. Application developers can also explicitly state which type of resources are allowed, or excluded, in the search process. Finally, the list of candidate resources is filtered by the

Algorithm 4: Tabu Search Move *TSMove*

```
Input  : $Solution$;                          // Candidate Solution
         $Constraints$;                       // Set of Constraints
Output : $NewSolution$;                       // The Changed Solution
         $NoChange$;                          // Boolean Flag
1  $Tiers \longleftarrow$ getTiers($Solution$)
2  $Tiers \longleftarrow$ randomize($Tiers$)
3  while $Tiers$ is not empty do
4      $Tier \longleftarrow$ removeFirst($Tiers$)
       // Get resources to change
5      $CloudResourceList \longleftarrow$ getCandidateResources($Tier$)
       // Filter resources according to constraints
6      $CloudResourceList \longleftarrow$
       filterResourcesByConstraints($CloudResourceList$,$Constraints$)
       // Filter resources according to short term memory
7      $CloudResourceList \longleftarrow$ filterVisitedSolutions($CloudResourceList$)
8      if $CloudResourceList$ is not empty then
           // Calculate the fitness of each resource
9          for $i \leftarrow 0$ ; $i < CloudResourceListSize$ do
10             $Resource \longleftarrow CloudResourceList(i)$
11             $$Fitness(i) \longleftarrow \frac{Resource.cores \times Resource.speed}{Resource.cost} \quad (7.5)$$
12             $Fitness(i) \longleftarrow Fitness(i-1) + Fitness(i)$
13         $RandomFitness \longleftarrow$
           $random[0,1] * Fitness(CloudResourceListSize - 1)$
14         $NewResource \longleftarrow$
           getResourceFromFitness($CloudResourceList$,$RandomFitness$)
15         ChangeResource($Tier$,$NewResource$)
16         EvaluateSolution($Solution$)
17         UpdateMemory($Solution$)
18         $NoChange \longleftarrow false$
19         return
20 $NoChange \longleftarrow True$
```

filterVisitedsolutions procedure on line 7 to avoid the generation of solutions already visited. To perform this filtering we maintain a short term memory of the configurations generated by the *TSMove* procedure, this memory is updated on line 17.

The structure of this memory is quite simple and reflects the purpose of the procedure itself. Since this procedure is only interested in exploring the upper level problem, the number of replicas used in each tier can be ignored, relaying on the fact that the *ScaleLS* procedure will take care of optimizing this aspect of the problem. The remaining configuration can be easily summarized by looking at the binding of resource types to each tier of the application. The short term memory uses a simple hash function that concatenates the name of the tier with the type of the resource used to host that tier. When filtering the available resource types, the *filterVisitedsolutions* can check if a solution has already been visited by build its hash and look into the memory.

After all the filtering has been performed the *CloudResourceList* will only contain resource types that would lead to new solutions that satisfy the architectural constrains. In order to select the resource to use we exploit a well known selection mechanism used in many heuristic approaches, called Roulette Wheel (or Fitness Proportional) Selection [74]. The principle behind this mechanism recalls the game of the roulette, in our scenario the bins of the roulette are the resource types. When we throw the ball in the roulette it will fall in one of the bins, the resource corresponding to that bin will be the selected one. In a regular roulette all the bins have the same size meaning that the probability of a resource to be chosen is uniformly distributed. In order to bias the selection towards more promising configurations change the sizes of the bins of the roulette making them dependent on a function of the characteristics of the resource they are representing, this function is called *Fitness Function*. In such a way the probability of a resource to be selected is proportional to the value of the fitness function. The core of the Roulette Wheel algorithm is shown in lines 10- 12. In particular, the calculation of the fitness function is shown by Equation 20 applied on line 11.

$$Fitness(i) \longleftarrow \frac{Resource.cores \times Resource.speed}{Resource.cost} \tag{7.6}$$

Heuristically, the fitness of a resource type is given by the number of cores offered by the resource multiplied by its speed, over the cost of using that resource. This value identifies the efficiency of the resource allowing to drive the selection process toward resources that have a better trade-off between cost and performance.

This kind of formulation is straightforward for resources like VMs but not so intuitive for

PaaS services, which might not have the information on the speed or the number of cores. In some situations the PaaS resource is backed by a IaaS resource, if this information is available in the Resource Model Database then we will use the corresponding IaaS resource to retrieve the information needed for the fitness calculation. If none of this information is available, e.g. for PaaS services whose autoscaling policy is completely managed by the cloud provider, the fitness is set to a fixed value leading to a uniformly random selection of the resource.

A common way to choose a fitness function is to use directly the optimization objective. In our case this is not possible because the optimization objective, the total cost of the solution, depends heavily on the number of replicas of the resources which has been optimized for the current selection of resources.

When the resource type has been selected the procedure modifies the solution, proceeds with its evaluation and updates the short term memory. Finally, the *NoChange* flag is set to false to signal the general optimization procedure that the solution has been modified and it is ready to be optimized in the context of the lower level problem.

7.5 Restart

The *Restart* procedure, shown in Algorithm 5, has a goal similar to the one of the *TSMove* since both procedures operates on the higher level problem in order to increase the diversification of the solutions explored by the optimization algorithm. This procedure in particular applies a stronger diversification action when the one performed by the *TSMove* is not able to generate solutions that have not been previously examined. This situation is quite rare if the *MaxIter* parameter of line 6 of Algorithm 1, that identifies how many times the TSMove is applied, is small. Nevertheless if the user wants to explore a wider range of possible deployments and sets a high value for *MaxIter* this situation might arises. This situation is identified by the fact that the *LocalBestSolution* has reached a strong local optimum and all of the solutions in its neighborhood (the one considered by the *TSMove*) have already been explored.

As shown in Algorithm 1, on line 23, we can identify this situation by looking at the effect of the *TSMove* on the *Candidate* solution. In order to escape this local optimum the *Restart* procedure implements a more disruptive action with respect to the *TSMove*. It is more disruptive in two ways: first, it changes at the same time the type of resources used in each tier, while the *TSMove* operates on a single tier; second, it chooses the new type of machines by considering how many times a solution with the specified assignment has already been evaluated and favoring solutions that have been explored less.

The procedure is composed by two operations that are repeated for each tier of the solution.

First we use the long term memory, briefly introduced in Section 7.1, in order to find the new type of resource to assign to the tier, on line 3. Then we apply the change by using the new type of resource, on line 4.

The main contribution of this procedure to the overall algorithm is given by the way the long term memory is used to select a new type of resource. The long term memory implements a frequency memory that stores information about how often a certain assignment of a resource type to a particular tier appears during the evaluation process. The main difference between this memory and the short term memory used by the *TSMove* is that this memory does not store information about the entire configuration of the application but operates at the level of each tier. Every time a solution is evaluated this memory gets updated by looking at what type of resource was assigned to each tier for that particular solution and the appropriate counter is incremented.

When the *Restart* procedure is invoked it generates a new solution in a region of the solution space that has been explored less, in order to escape from attractive regions that have already been extensively explored. To do so, the procedure uses information related to the frequencies of evaluations stored in this memory. It operates by changing at the same time the selection of cloud services associated to each of the tiers of the application, in contrast with the *TSMove* procedure that changes the service selection of just one tier. For each tier of the application the procedure looks into the long term memory to find which service has been less frequently assigned to that tier. By doing so it generates a solution that is composed by resource types that have already been chosen by the *TSMove* and fulfill architectural constraints, as explained in Section 7.4. The outcome of this procedure is used by the general algorithm to generate a novel starting point before restarting the main optimization loop.

Algorithm 5: Restart

```
Input  : $Solution$;                              // Candidate Solution
Output: $ChangedSolution$;                         // Changed Solution
1 $ChangedSolution \longleftarrow Solution$
2 for $tier \in$ getTiers($ChangedSolution$) do
3     $NewResource =$ getLeastFrequentlyUsedResource($tier$)
4     ChangeResource($Tier$,$NewResource$)
```

7.6 OptimizeWorkload

As introduced in Section 7.1, most of the procedures described in this chapter are applied independently to all of the providers involved in the optimization. This approach of tackling the deployment problem on each cloud separately greatly simplifies the optimization process and improves its performance. However, as shown in Chapter 5, the solution of the entire problem depends also on the distribution of the traffic among the available cloud providers. This dependency makes the problem much more complex since it introduces another level to the problem. Adding a third level to the optimization algorithm described so far in this chapter would increase notably its complexity and would have a great impact also on the overall optimization time. For this reason we decided to deal with this problem is a different way.

The main idea behind our approach is to postpone the optimization of the workload distribution until a new *BestSolution* is found, as shown in Algorithm 1. When a new *BestSolution* is identified, it has already been optimized in the context of both the upper and lower level problems using a fixed choice of workload distributions that comes from the initial solution, derived by the MILP relaxation of the problem presented in Chapter 6.

By applying the *OptimizeWorkload* procedure to that solution we change the way the workload is distributed. The new solution goes then through a new *ScaleLS* phase in order to tailor the number of replicas of resources to the newly found distribution of workloads. The goal of this procedure is to change slightly the distribution of workloads from an initial solution by increasing the workload of the provider that is less utilized without changing the number of resources used. This action allows to offload other providers so that the *ScaleLS* procedure can remove replicas of their resources. Similarly, to the *ScaleLS* this procedure operates independently on all of the 24 hours evaluating the entire solution only when all of the hours have been modified. Finding the maximum amount of workload a provider can support without incurring in a scaling action is a complex problem. For this reason we decided to implement a local search similar to the one implemented by the general algorithm. This initial workload distribution contained in the input solution is refined iteratively by identifying the provider that can accommodate the most workload, on line 8 and increasing its workload by 5% in each iteration.

Given the context in which this optimization takes place, namely during early stages of the development, using a finer grain on the splitting of the workload is not advisable. At runtime the workload balancing is usually performed by dedicated components that implement more sophisticated control mechanism that operate at a lower granularity taking into consideration runtime characteristics of the infrastructure, e.g. machine failure[1].

[1]https://status.cloud.google.com/incident/compute/15045

In order to choose which provider has more room to accommodate additional workload the *getLeastUtilizedProvider* looks at all the available providers and inspects the utilization of all the tiers. Increasing the workload on a provider has an impact on all the tiers of the application, so we consider the tier that has the highest utilization as an indicator of whether that provider can host more requests or not. The procedure selects the provider on which the most utilized tier is farther from its maximum, that is either 100%, or a threshold defined by the user.

The workload share on the selected provider is then increased and the workload on other providers is reduced uniformly. Before evaluating the obtained solution, the procedure checks that the reduced amount of workload entering other providers does not violate any architectural constraint (presented in Chapter 3); this check is performed by the *checkSplitting* procedure on line 12. In a multi-cloud scenario the application architect, concerned with the availability of the application, might have specified a constraint saying that all of the providers must receive at least a minimum share of the workload, say 20%. If a provider reached this minimum amount then we can not move further away the workload, in this case we keep the minimum level of workload share and mark the optimization for that particular hour as complete.

Algorithm 6: Optimize Workload

```
Input  : $Solution$;                              // Multicloud Solution
         $Constraints$;                           // Set of Constraints
Output: $FinalSolution$;                          // Multicloud Solution

1  $Increment \longleftarrow 0.05$
2  $minWL \longleftarrow$ getMinimumWorkload($Constraints$)
3  for $t \in [1..24]$ do
4      $Working(t) \longleftarrow$ True
5  while $Solution$ is feasible $\wedge$ ($\exists t \in [1..24] | Working(t)$) do
6      $FinalSolution \longleftarrow Solution$
7      foreach $t \in [1..24] | Working(t)$ do
8          $Provider \longleftarrow$ getLeastUtilizedProvider($Solution$)
9          $Workload \longleftarrow$ getWorkload($Solution$,$Provider$)
10         $NewWorkload \longleftarrow Workload + Increment$
11         $Solution \longleftarrow$ changeWorkload($Solution$,$Provider$,$NewWorkload$)
12         $MiniumReached \longleftarrow$ checkSplitting($Solution$,$Constraints$)
13         if $MinimumReached$ then
14             $Solution \longleftarrow$ changeWorkload($Solution$,$Provider$,$Workload$)
15             $Working(t) \longleftarrow$ False
16     evaluate($Solution$)
```

Finally, when all of the hours have been modifies we evaluate the entire solution, on line 16. The optimization procedure continues until all of the 24 hours can not be changed or until the solution derived is not feasible. In that case the solution of the previous iteration is used as final solution.

Chapter 8

Approach Evaluation

To evaluate our approach we identified a set of evaluation questions aimed at showing the ability of the approach to deal with real world scenarios and to help software architects in better understanding how to build applications that can effectively exploit the cloud environment.

Q.1 *Is the approach useful?*

The main evaluation question is related to the utility of using the proposed approach in the development of new cloud applications. This question has been analyzed by using the Constellation case study introduced in Chapter 3 and detailed in Section 8.1.1 and by another industry relevant application developed by BOC[1] described in Section 8.1.2. With the help of these case studies we expanded the evaluation question **Q.1** into three sub-questions:

Q.1.1 *Does the proposed methodology help in understanding how the system could leverage the cloud platform?*

Question **Q.1.1** is related to the ability of the tool in understanding if the application design proposed by software architects can exploit the characteristics of the cloud environment. This question is analyzed in Section 8.1.1 when the initial design proposed by Softeam is analyzed by SPACE4Cloud under different workload conditions and a bottleneck is identified. The information derived by the analysis has been used by Softeam to successfully change their design in order to avoid the bottleneck and make a better use of the elasticity offered by the cloud environment.

Q.1.2 *Does the approach helps in understanding the cost of providing a certain quality level (cost of quality)?*

Question **Q.1.2** consider the quality aspect of the analysis. In this case the question has been raised by the case study developed by BOC in which developers asked how the level of

[1]boc-eu.com

quality provided to their users affected the cost of the deployment. In this sense they performed a small sensitivity analysis by repeating the optimization process performed by SPACE4Cloud reducing the constraints on the application response time. A secondary feature inspected by this question is related to the cost of multicloud deployments. Again in the case study developed by BOC, the cost of deploying the whole application on Profit Bricks or Cloud Sigma, two cloud providers already in use at BOC, has been analyzed. Finally, we have studied the cost of distributing replicas of the application on both providers in order to avoid failures due to the cloud provider.

Q.1.3 *Does the approach support the analysis of different mixture of workloads?*

Question **Q.1.3**, is related to the ability of the approach to evaluate changes in the application specifications and their impact on the quality of the application. In particular it addresses the fact that workload estimations used at design time to build user behavior specifications are often inaccurate. We have analyze the effect of this inaccuracy by changing the specification of the mixture of users in the workload of the BOC case study and analyzed how the cost of providing the same quality level changes.

Q.2 *Does the optimization approach scale to complex scenarios?*

Question **Q.2** consider the applicability of the optimization heuristic performed by SPACE-4Cloud in the real world by analyzing how the time required to find the final solution changes with the complexity of the application. To answer this questions we have performed a scalability analysis that shows how the optimization time changes with the increasing number of components and tiers. This analysis show that the approach is capable of handling complex scenarios in a reasonable time. The longest optimization of an application deployed on three cloud providers took 40 minutes. To better understand the efficiency of the optimization we inspected the time required by the different phases of the optimization process and showed that almost 93% of the time is spent in the execution of the heuristic optimization. Finally most of this time is spent in solving LQN performance model by means of the LINE tool.

Q.3 *Does the approach provide benefits with respect to current best practices?*

Finally, question **Q.3** consider the performance of the proposed approach with respect to current best practices proposed in the literature and used in real systems. To answer this question we repeated the scalability analysis by using two best practices based on thresholds on the utilization of resources. We have then compared the solutions obtained by these approaches with those derived by our and reported the results in Section 8.3. Results show that using the proposed approach allows an average reduction in the deployment costs of 37%.

8.1 Industrial Case Studies

This section presents the use of the design methodology we presented in Chapter 3 in two real world industrial case studies. The first one, introduced in Chapter 3 and developed by Softeam, shows how the analysis process implemented by SPACE4Cloud allows the development team to gather insights on some performance characteristics of the architecture under development and evolve it in the early stages of the development in order to meet the desired QoS.

The second case study focus on the migration of an existing application into a multi-cloud environment. The peculiarity of this case study is that the application that BOC decided to move to the cloud is of critical importance for the attraction of new customers, the requirements of high availability and proximity of the infrastructure to the final users motivates the adoption of a multi-cloud deployment.

Together these case studies are used to answer the evaluation question **Q.1**, along with the sub questions, related to the usefulness of the approach.

8.1.1 The Constellation Platform

In order to assess the applicability of our approach in a real world scenario, we have used SPACE4Cloud to study the performance of the *Constellation* architecture presented in Chapter 3. In particular we are interested in evaluating how well this architecture handles different workload conditions. For this reason we started our study with a typical pattern of users derived by an analysis of the current system and then inflated the workload to simulate a growth in the number of users. In this case study we will use Amazon as target cloud provider.

8.1.1.1 Initial analysis

To better focus the analysis Softeam identified the core of their application in the SVN and HTTP services that are used to provide users most of the modeling capabilities that represents the main feature of their application. Administration services are accessory operations required when joining a new project but are more lightweight and are used less often by end users so we decided to neglect them in this initial analysis. Table 8.1 shows the demanding time (i.e., the time required to serve a single request [54]) of the operations provided by the SVN and HTTP components, recalling the role of this components in the overall architecture, it is easy to see that SVN operations are more heavyweight than HTTP since they allow users to update the entire model and commit their modifications checking and resolving eventual conflicts. The **partialRead** functionality offered by the HTTP component is a lightweight version of the update functionality offered by the SVN used to provide users a fast way to update a portion of their models.

Operation	**Average time (ms)**
readProjectFirstPageDescription	597
readCompleteProjectConfiguration	723
databaseAccess	900
openProject (SVN)	15,000
update (SVN)	2,500
commit (SVN)	41,000
partialRead (HTTP)	1,000

Table 8.1: Constellation demanding times.

To evaluate the behavior of *Constellation* against the growth of the number of user, we inflated the base workload, derived by analyzing logs of the current implementation of the application and reported in Figure 8.1, by multiplying it up to 10 times and repeated the optimization performed by SPACE4Cloud for each workload configuration. Each execution provided both a service allocation optimized for that workload condition as well as system performance estimates.

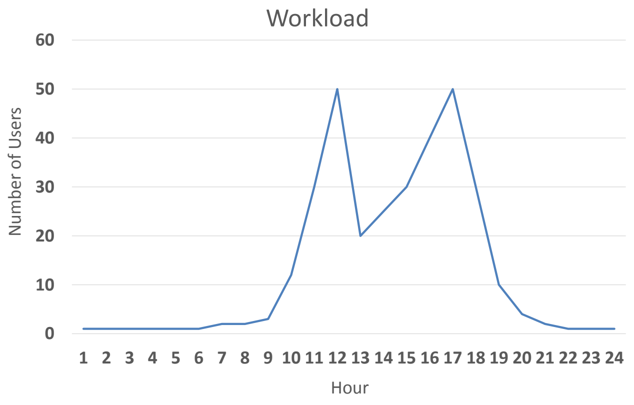


Figure 8.1: Base workload for the Case Study analysis.

The result of the optimization performed by SPACE4Cloud for a peak number of users lower than 200 suggested to use the Amazon c3.large flavor to host the tier dedicated to the HTTP component and the most powerful type of VM available, the c3.2xlarge, to host the tier dedicated to the SVN server component. Using such a configuration the performance analysis shows that all QoS constraints defined by the user, and reported in Chapter 3 are fulfilled.

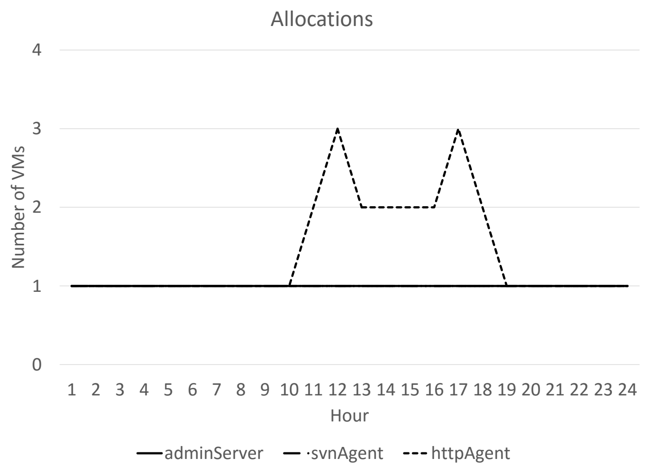
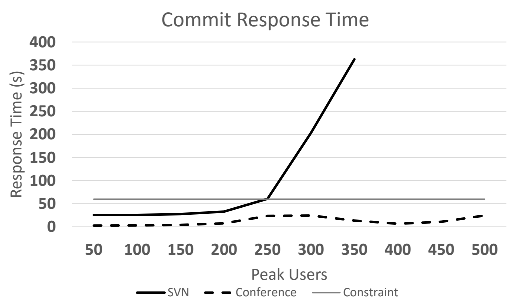


Figure 8.2: Optimization Result.

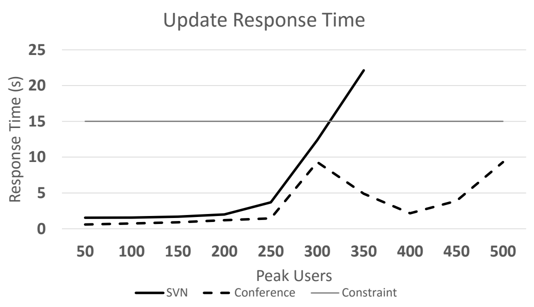
When the number of users is further increased, reaching a maximum of 250 users in the peak hour, the optimization performed by SPACE4Cloud do not produce any feasible solution. As shown in Figure 8.2,the HTTP Agent has to be replicated 3 times during peak hours to provide the required QoS but the single VM assigned to the SVN Agent can only handle a load up to 250 contemporary users. All the solutions explored by the tool whith more than 250 users at peak are not capable of fulfilling constraints associated to the *commit* and *update* operations offered by the SVN component. The result of this analysis allowed the identification of the weakest point of the architecture in the use of the SVN server, it also showed that other components of the architecture, like the HTTP component or the administration server and database, proved to adapt well to increasing workload conditions.

Figure 8.3 summarizes the results of the analysis conducted using SPACE4Cloud. The thick black line shows the expected response time of the *commit* functionality, in Figure 8.3a, and the *update* functionality, in Figure 8.3b.

In the deign models the software architect defined a constraint of 60 seconds as the upper bound of the average response time, shown as a thin grey line, and a constraint of 5 minutes on the 95th percentile for the *commit* functionality. For what concerns the *update* functionality, a constraint of 15 seconds has been specified on the average response time and a constraint of 30 seconds has been specified for the 95th percentile. Figure 8.3 shows that both functionalities do not scale well with the number of users but the *commit* functionality is the most critical.



(a) Expected response time of the *commit* functionality in the *Constellation* and *Conference* architectures



(b) Expected response time of the *update* functionality in the *Constellation* and *Conference* architectures

Figure 8.3: Analysis of the case study *Constellation* and *Conference* architectures

The information provided by this analysis allowed to identify the problem related to the scalability of the SVN architecture and led the development team to reconsider their design decisions.

8.1.1.2 Reiteration

As introduced in Section 4.1.3, application developers can now use the information gathered by the analysis of their candidate architecture in order to refine it. The *Conference* architecture has been created by the Softeam design team in order to overcome the scalability limitation found by the previous analysis. The architecture of the new solution is shown in Figure 8.4, some of the components of the initial architecture have been maintained while the central **SVNAgent** has been replaced by a **ConferenceAgent** used to store the models and apply the modifications performed by users. This component propagates updates on the model to other lightweight components, called **ReadOnlyConferenceReplica**. These components are used to host replicas of the model accessed by users when retrieving an update in order to offload the main **ConferenceAgent** of the reading workload. The decoupling of updates and commits has been performed in order to offload the central SVN component from part of the work and allow it to effectively manage update requests. In the new *Conference* architecture, update operations are delegated to dedicate scalable agents.

From the user point of view the expected behavior will change allowing users to update their local models from **ReadOnlyConferenceReplica** agents, they will also be able to commit more frequently smaller chunks of data, instead of performing few large commits (see Figure 8.5). The model of the user behavior shows just a small change in the way the system is used, in

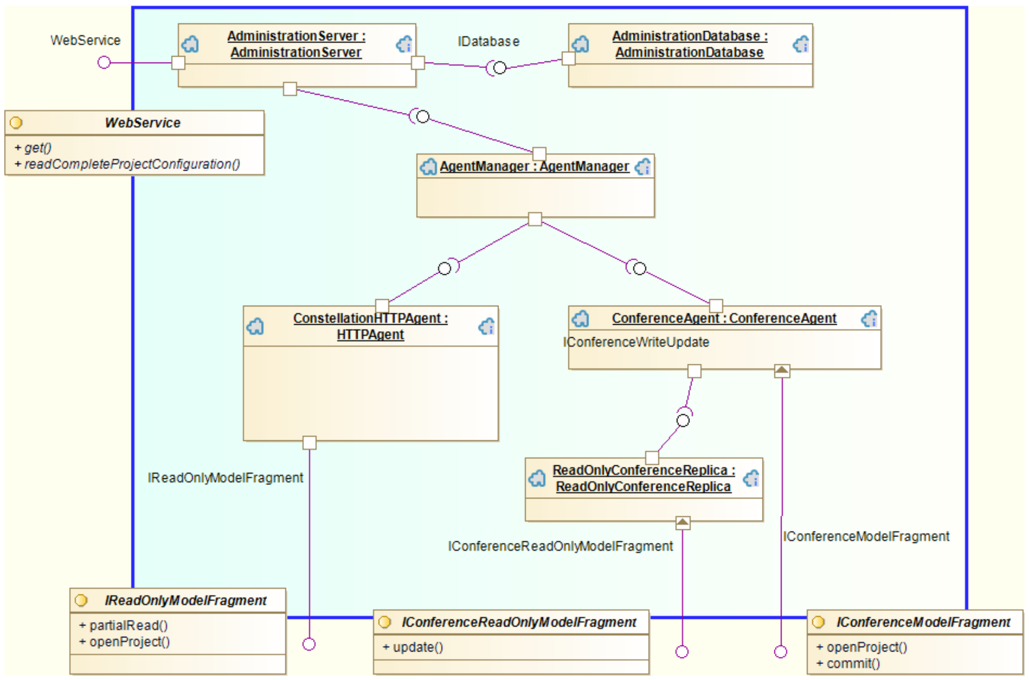
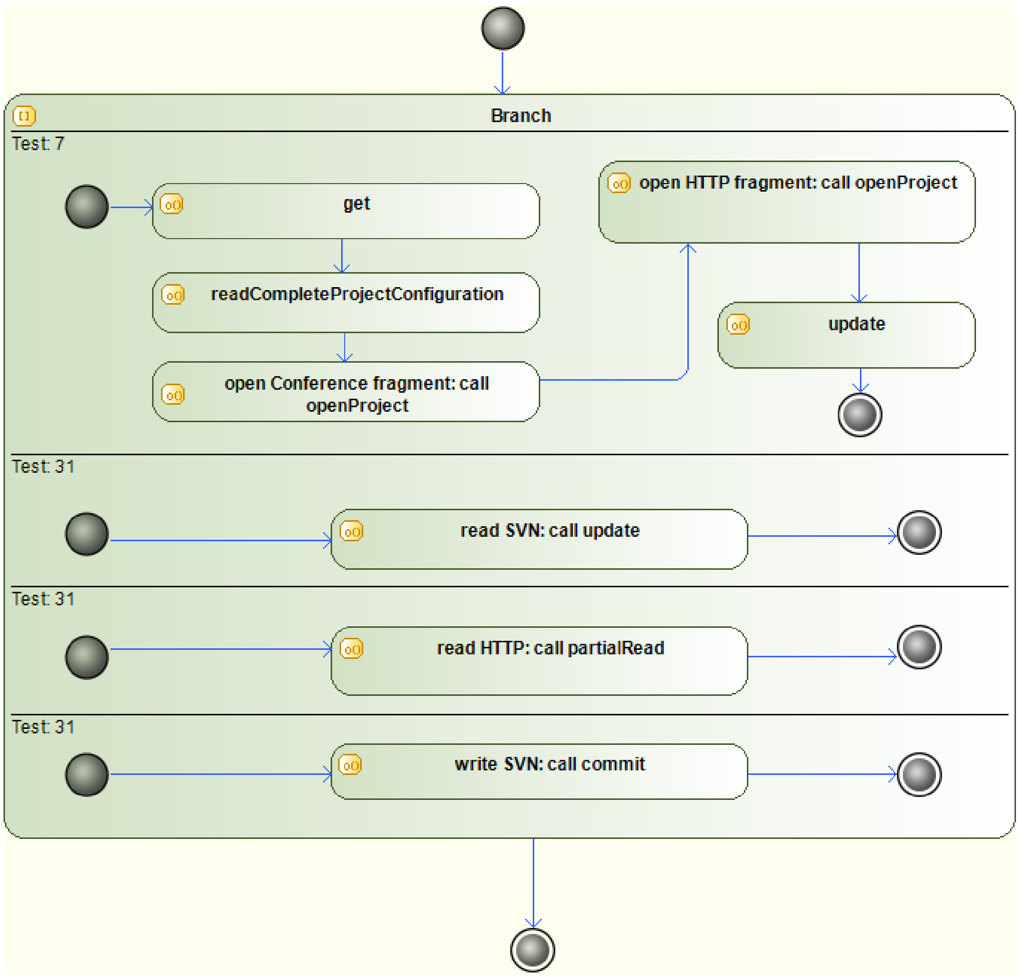

WebService
AdministrationServer : AdministrationServer
IDatabase
AdministrationDatabase : AdministrationDatabase
WebService
+ get()
+ readCompleteProjectConfiguration()
AgentManager : AgentManager
ConstellationHTTPAgent : HTTPAgent
ConferenceAgent : ConferenceAgent
IConferenceWriteUpdate
ReadOnlyConferenceReplica : ReadOnlyConferenceReplica
IReadOnlyModelFragment
IConferenceReadOnlyModelFragment
IConferenceModelFragment
IReadOnlyModelFragment
+ partialRead()
+ openProject()
IConferenceReadOnlyModelFragment
+ update()
IConferenceModelFragment
+ openProject()
+ commit()

Figure 8.4: Conference Architecture of Constellation

fact most of the elements of this figure are identical to those in Figure 3.4 just the probabilities of invoking the functionalities to update and commit models are changed.

In order to derive the demanding times of the involved operations, we performed some preliminary experiments also with this new configuration. Given that this new architecture has not been implemented, since Softeam is still in the design phase, we used an instance of the *SVN* architecture hosted on Amazon EC2 to reproduce the new user behavior and gather demands in a scenario with small and frequent commits and updates. The demanding times derived by this analysis are reported in Table 8.2, demands on the Administration Server, Administration database and HTTP server are the same as those reported in Table 8.1, since that part of the architecture has not been modified but commit times are much lower than the ones for the original architecture thanks to reduced size of the commits. These initial estimations can be used to analyze the general behavior of the new architecture and can then be refined once a prototype version of the system has been implemented.

Going back to Figure 8.3 we can now see a comparison between the expected response times for both the *commit* and *update* operations in the two considered architectures, the *SVN*, shown as a black thick line, and the *Conference*, shown as a dashed black line. Results show



Branch
Test: 7
get
readCompleteProjectConfiguration
open Conference fragment: call openProject
open HTTP fragment: call openProject
update
Test: 31
read SVN: call update
Test: 31
read HTTP: call partialRead
Test: 31
write SVN: call commit

Figure 8.5: Conference Architecture usage model

that the new *Conference* architecture is capable of supporting a higher number of users and keep response time below the threshold even when the maximum number of users grows up to 500.

Figure 8.6 shows a comparison of the cost of the solutions derived by the two architectures and demonstrates a similar situation as the *Conference* system make use of smaller-sized machines and scales the number of replicas according to actual needs of capacity.

The case study presented in this section shows how SPACE4Cloud can be used to analyze a candidate application architecture under different workload conditions and identify design choices, in this case the use of a central SVN server, that can not make an effective use of

Operation	Average time (ms)
openProject (Conference Model Fragment)	15,000
commit (Conference Model Fragment)	1,500
partialRead (ReadOnlyConferenceReplica)	1,000
commit (ReadOnlyConferenceReplica)	2,500

Table 8.2: Conference service specific operations demanding times.

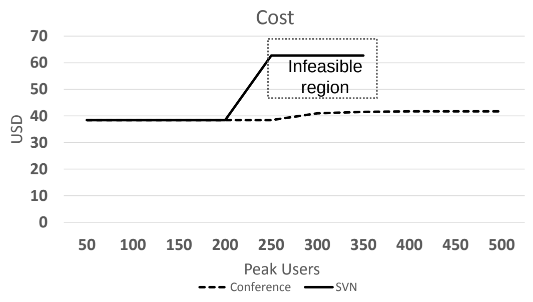


Figure 8.6: Cost analysis of the two architectures under evaluation (daily costs for VMs usage).

the scalability provided by the cloud environment. The new architecture presented in Section 8.1.1.2 shows how this information can be used to derive a better application architecture capable of exploiting the scalability offered by the cloud environment in order to reduce costs and support a higher number of users. We argue that this example shows how the proposed approach helps in understanding how the system under development could leverage the cloud platform, addressing question **Q.1.1**.

8.1.2 The ADOxx platform

The second industrial case study we used to evaluate the proposed approach has been developed by BOC[1]. BOC is a medium-sized software development and consulting company located in different countries across Europe. BOC offers a set of tools to support management approaches in different areas like Strategy, Performance Management, Business Process Management and others. BOC developed the ADOxx meta-modeling platform to build and customize their tools. Typically BOC installs the ADOxx platform and the required components on the premises of

[1]boc-eu.com

its customers leaving to the customer organization the provisioning and maintenance of the required infrastructure and environment.

In order to provide users with a test environment in which they can try out the products, BOC hosts on its own premise a small datacenter in which a minimal infrastructure setup of ADOxx is built inside a virtual machine. BOC realized that using a standardized environment, like the one offered for product testing, allows to minimize infrastructure related issued during the installation of the ADOxx platform. Managing directly the infrastructure used to provide ADOxx to its users, BOC can better monitor and maintain the ADOxx platform providing a more satisfying customer support.

Given these benefits, BOC decided to migrate their service offering adopting a SaaS approach in which they maintain the environment needed to run the ADOxx platform and give to the final users access to their own copies of ADOxx through a web interface. Adopting this solutions allows BOC also to exploit economy of scale by allowing different instances of the ADOxx platform to share some components. In order to avoid large infrastructural investments, BOC decided to relay on the public cloud offering to outsource the infrastructure management and focus their development and administration efforts to the provisioning and configuration of ADOxx.

The case study presented in this section is the first step of a more complex migration approach presented in [43], in which BOC moves their product trial infrastructure to a public cloud provider. This case study has been used by BOC to test different migration approaches in order to understand how their application could be evolved to better exploit the characteristics of the cloud environment and inspect the cost of providing different quality levels.

The portion of the system that the company is moving to the cloud is the one dedicated to the attraction of new potential customers, for this reason providing a reliable and responsive environment for new potential user is critical to maintain company brand reputation. To increase the responsiveness of the system they decided to replicate their infrastructure on different regions, to reduce distance from the data-center to the user increases the responsiveness of the system making it less susceptible to the unpredictable network performance of the internet. Recalling the fact that in a traditional setup users have a dedicated infrastructure, usually hosted inside the company LAN, it is clear that exploiting the proximity of data-centers to final users is a critical aspect that BOC need to take into consideration in their migration to the cloud.

The choice of using of multiple cloud providers has also been motivated by availability concerns. Since this environment is used freely available to users, any failure of the system could lead to potential customer loss. BOC decided to use services offered by different cloud providers in order to build a system that is robust against failure of the infrastructure managed by the cloud provider.

Figure 8.7 shows the architecture of ADOxx. The platform is composed in a classical 3-layers structure in which the presentation layer, here implemented by the *ADOxx Presentation* component, manages interactions with end users; the back-end layer, implemented by the *ADOxx Business*, hosts all the application logic; finally, the database tier, implemented by the *ADOxx Database* component, stores user data.

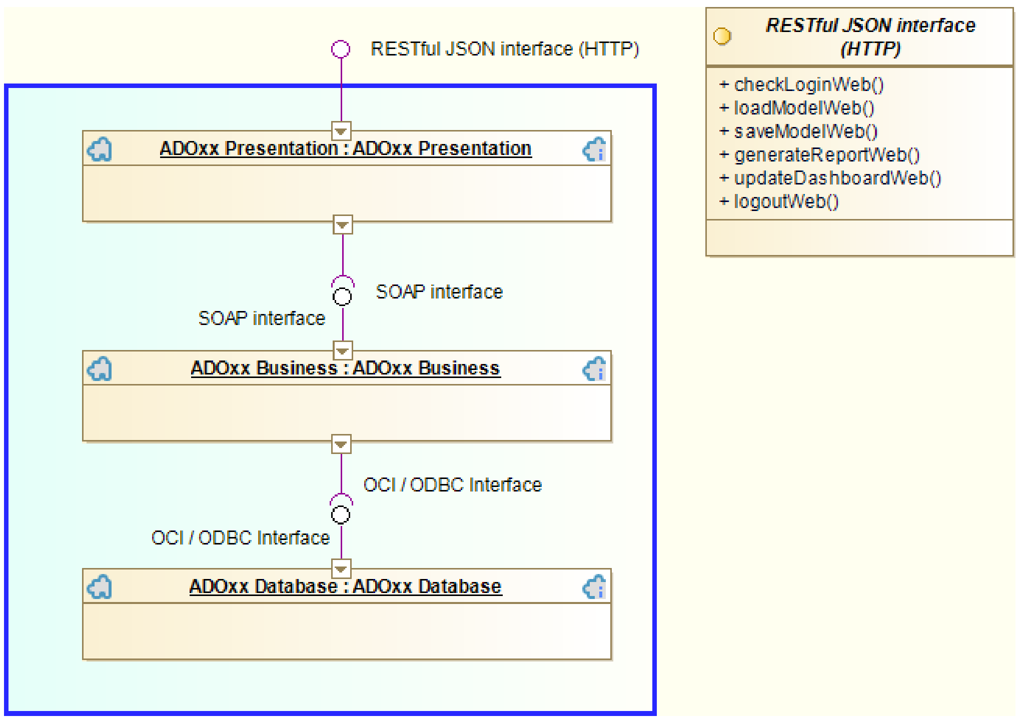
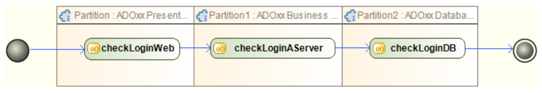
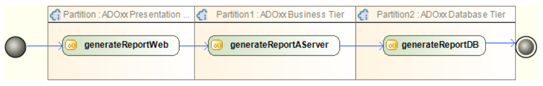


Figure 8.7: ADOxx Architecture

Each tier i interacts with $i+1$ in order to provide to the user a particular functionality. Each functionality of the presentation tier has been isolated by others by implementing dedicated APIs using a Simple Object Access Protocol (SOAP) interface. To retrieve the information needed to build the objects required by the presentation layer and to store the modifications of these objects in a persistent way, the business layer makes use of the database layer by means of an Open DataBase Connectivity (ODBC) interface. Each functionality implemented by this interface represents a particular query used by the business layer. Figure 8.8 shows two example of orchestration diagrams for the login functionality and for the report generation functionality.


Partition : ADOxx Present...
Partition1 : ADOxx Business ...
Partition2 : ADOxx Databa...
checkLoginWeb
checkLoginAServer
checkLoginDB

(a) Orchestration model for the login functionality


Partition : ADOxx Presentation ...
Partition1 : ADOxx Business Tier
Partition2 : ADOxx Database Tier
generateReportWeb
generateReportAServer
generateReportDB

(b) Orchestration model for the report generation functionality

Figure 8.8: Orchestration models

In traditional production environments installed by BOC on customer premises each of these components is deployed into a dedicated tier but in the test infrastructure, that offers a limited set of functionality, modeled here all the layers have been installed into a single tier deployed on a VM in a public cloud provider. Using a single VM to host the entire application stack allows BOC to easily move their current application from one cloud environment to another without modifications.

Figure 8.9 shows the typical behavior of users that interacts with the ADOxx environment specified by domain expert that analyzed the behavior of similar systems. The figure shows two main classes of users, 20% of them are modelers who logs into the system, explore several models loading them, make some modifications to some of the models saving them on the system and, finally, log off the system. Most of the users, on the other hand, interact with the dashboard in order to gather many information on the different models and then log into the system to generate a report.

Figure 8.10 shows a deployment of the ADOxx architecture at the CPIM level. This diagram shows that all the three main layers have been packed into a single VM called *BPMSCaseStudyVM*. This VM contains all the components and some additional tools used to host the application and interconnect the three layers. This diagram shows that the database used to store users' data is implemented using SQL server, while the presentation layer is hosted on a Tomcat Java server.

Finally, Figure 8.11 shows a CPSM level deployment of the application in which different replicas of the VM are hosted on two cloud providers in this case using instances of size *c3.large* on Amazon EC2 and instances with 4GB of memory anf 2CPUs on Flexiant. These instances have been used for prototype development to profile the application and derive the demands specified in the model. Each tier can host several replicas of the VM in order to manage traffic coming from multiple users and guarantee QoS.

BOC decided to manage their client engagement environment by providing three QoS levels. Users that occasionally reach on their website and try out the tool are marked are *Bronze*. These users might be interested in the products offered by BOC but are not likely to make any

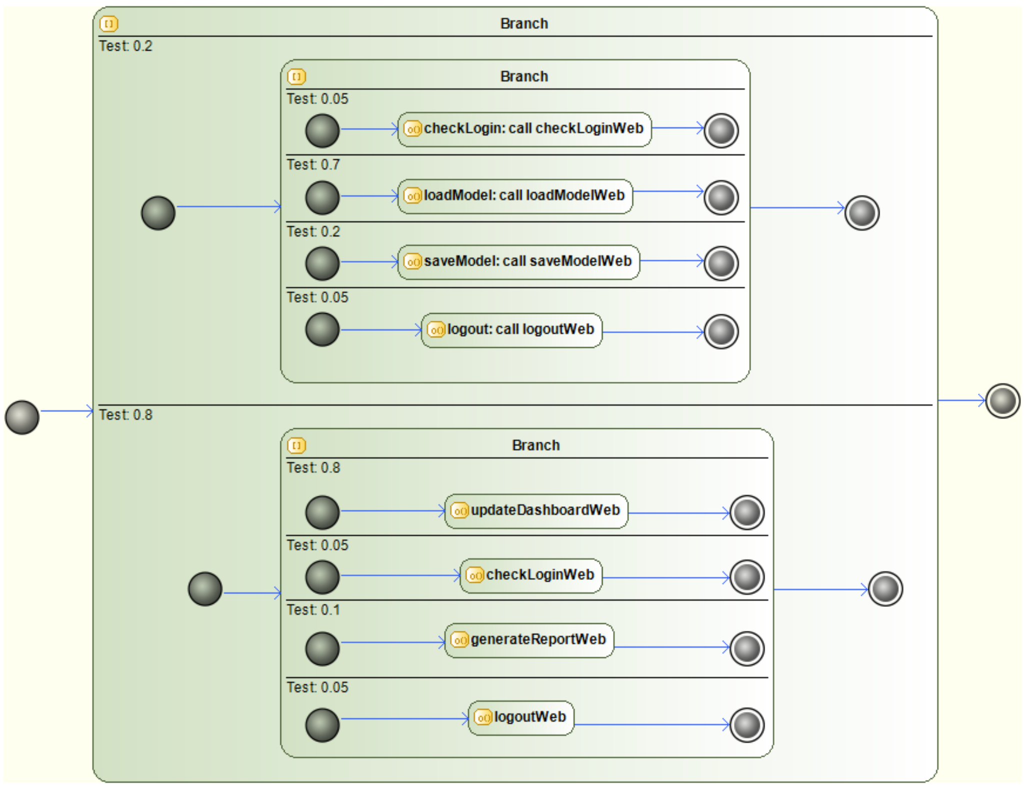


Figure 8.9: ADOxx Usage Model

commitment. Users that reach the trial platform after an initial contact with BOC are more likely to be seriously interested in acquiring their product and are labelled as *Silver*. Finally, users that have already initiated the negotiation of a contract with BOC and want to evaluate more seriously the tools are marked as *Gold*, these users are very likely to acquire the product and BOC wants to make sure that they have the best experience possible working with the evaluation infrastructure. Application administrators provided a different set of constraints for each class of users and repeated three times the analysis in order to evaluate the impact on costs of the different QoS levels. By repeating the optimization process separately for each quality level BOC performed a What-If analysis aimed at understanding the cost of providing to their users certain quality levels.

As previously introduced, BOC choose to replicate their infrastructure on two cloud providers in order to increase the availability of their system and exploit the different locations of cloud providers datacenters to increase proximity to the user. BOC already identified two European

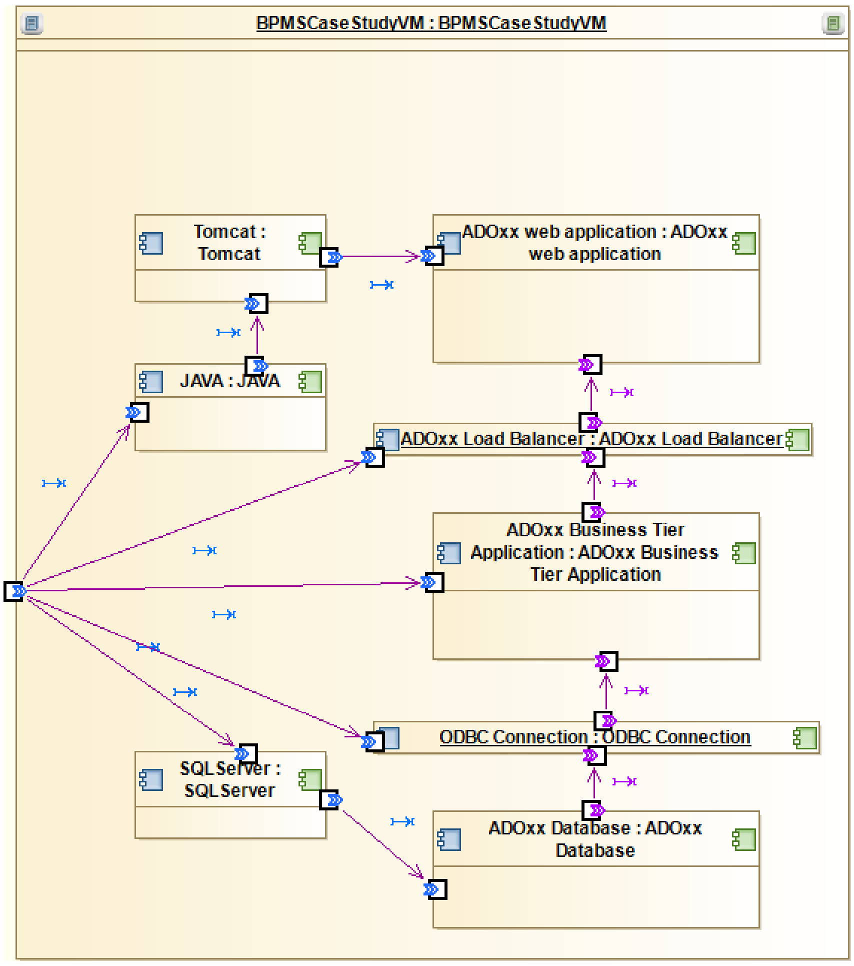

BPMSCaseStudyVM : BPMSCaseStudyVM
Tomcat : Tomcat
ADOxx web application : ADOxx web application
JAVA : JAVA
ADOxx Load Balancer : ADOxx Load Balancer
ADOxx Business Tier Application : ADOxx Business Tier Application
ODBC Connection : ODBC Connection
SQL Server : SQL Server
ADOxx Database : ADOxx Database

Figure 8.10: ADOxx Deployment CPIM

providers, ProfitBricks[1] and CloudSigma[2], as candidates to host their infrastructure and tried to evaluate a solution with a single cloud deployment in order to have an initial estimate of the daily cost that they would sustain choosing either one of these cloud providers.

Table 8.3 shows the constraints on the functionality specified for each level and Table 8.4 shows the cost of the associated deployment configuration Bronze users, for example have a maximum average response time of the login functionality of 4 seconds while Gold users have a maximum of 2 seconds. For the **saveModel** functionality the limit is of 4 seconds for all the categories but for Bronze users the limit is applied to the 80^{th} percentile but for the Gold

[1]https://www.profitbricks.com/
[2]https://www.cloudsigma.com/

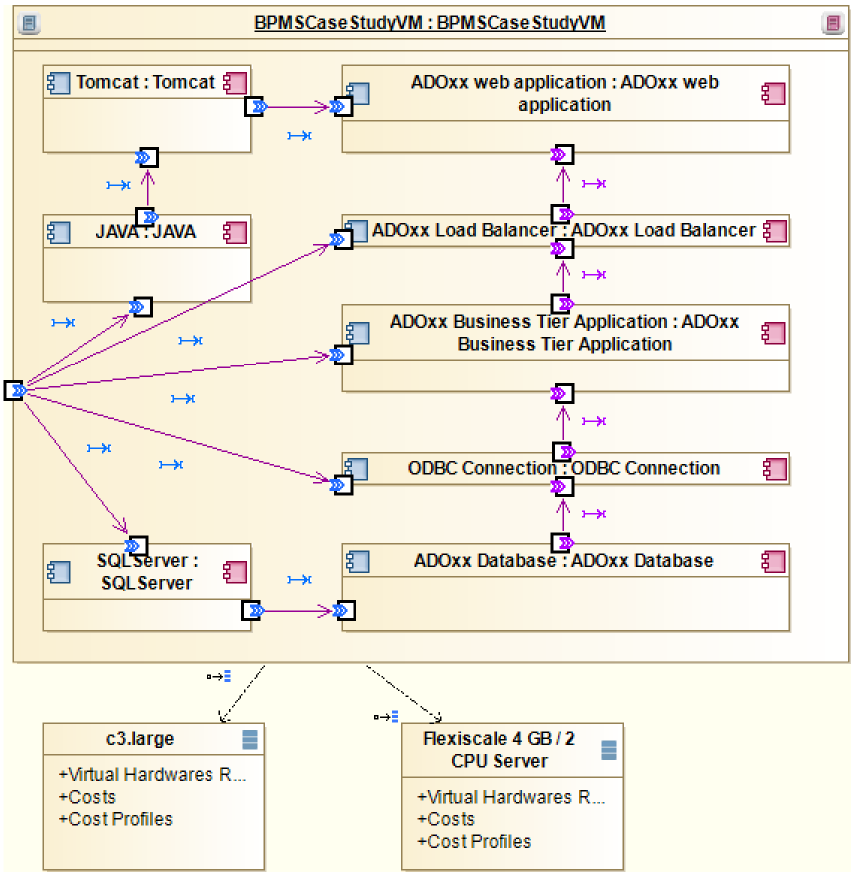

BPMSCaseStudyVM : BPMSCaseStudyVM
Tomcat : Tomcat
ADOxx web application : ADOxx web application
JAVA : JAVA
ADOxx Load Balancer : ADOxx Load Balancer
ADOxx Business Tier Application : ADOxx Business Tier Application
ODBC Connection : ODBC Connection
SQLServer : SQLServer
ADOxx Database : ADOxx Database
c3.large
+Virtual Hardwares R...
+Costs
+Cost Profiles
Flexiscale 4 GB / 2 CPU Server
+Virtual Hardwares R...
+Costs
+Cost Profiles

Figure 8.11: ADOxx Deployment CPSM

Class	Login	Generate Report	Save Model	Update DashBoard	Load Model
Bronze	4s	20s	80% <4s	80% <15s	80% <8s
Silver	3s	15s	90% <4s	90% <15s	90% <8s
Gold	2s	10s	95% <4s	95% <15s	95% <8s

Table 8.3: User classes constraints

users the limit is applied to the 95^{th} percentile. The optimization process has been repeated by considering deployments using Cloud Sigma or Profit Bricks, finally we have analyzed the cost of deploying the application on both cloud providers in order to better exploit the locality of the data-centers of both providers and reduce the risk of unavailability.

To better characterize the behavior of the application with respect to uncertainty in the modeling of user behavior BOC decided also to perform a sensitivity analysis on the mix of

Class	$ Cloud Sigma	$ Profit Bricks	$ Multi-Cloud
Bronze	294	232.98	365.68
Silver	304.5	184.41	297.24
Gold	429.25	142.91	245.27

Table 8.4: User classes costs

users in the workload changing the distributions of *readers* and *writers*. These analysis have been performed using the constraints of the Gold class. Results, reported in Table 8.5, show the daily cost of deploying the infrastructure in the scenario in which the percentage of users that perform report generation grows from 75% to 85%. Report generation is a compute intensive task, for this reason as the number of readers grows so the cost of the deployment does.

Writers, Readers	$ Multi-Cloud
25%, 75%	358.62
20%, 80%	365.68
15%, 85%	372.74

Table 8.5: User behavior analysis

The case study presented in this Section address evaluation question **Q.1.2** showing how SPACE4Cloud can be used to understand the cost of quality when dealing with different quality levels as in the What-If analysis performed by BOC. The case study also address question **Q.1.3**, since it shows a sensitivity analysis on the workload mix in order to address uncertainty in the estimations of the workload performed at design time.

8.2 Scalability Analysis

The two industrial case studies presented in this chapter show how our approach can provide useful insights on QoS characteristics and cost of the application. This information empowers software architects to revise some architectural choices in order better exploit some characteristics of the cloud environment, with the ultimate goal of providing a better QoS to end users or reduce infrastructural costs. In order to understand if our approach is capable of finding optimal allocations for complex architecture models withing reasonable time boundaries, and answer the evaluation question **Q.2**, we performed a scalability analysis.

8.2.1 Design of Experiment

As introduced in Chapter 5, the problem of finding the optimal allocation of cloud services to application components presents several dimensions. Using the experience we gathered during the analysis of the case studies we identified the main factors that affect the time needed to derive a quality solution and designed the experiments of this section in order to explore their effects.

The factors that have a greater impact on the time required to obtain a good solution are the *number of tiers* of the application under analysis and *number of components*. The *number of tiers* have a direct effect on the size of the upper level optimization problem discussed in Section 5.2, since the optimization procedure needs to evaluate many assignments of cloud services to each of them. The *number of components* do not affect directly the optimization procedure, since the assignment of services is made at the tier level, but has a big impact on the complexity of the LQN performance model. Solving LQN performance models is a very time consuming task and the optimization procedure has to evaluate many performance model, in the worst case one model per hour of the day, for each candidate solution considered.

To test the performance of SPACE4Cloud we build a large set of randomly generated optimization problem instances. Since most of real world applications are composed by two or three tiers [9] [14] we have focused on such a dimension, we then varied the number of components from a minimum of 4 to a maximum of 10. Applications expose to users from a single entry point component, that represents a web server or an application proxy, three classes of functionality. The execution of each functionality invoked by the end user involves some computation on several components hosted on all of the tiers.

Figure 8.12 shows the largest model we have used for the analysis and highlights interactions between the various components required to provide the three classes of functionalities.

The three functionalities offered to the users are exposed by component 0 which acts as

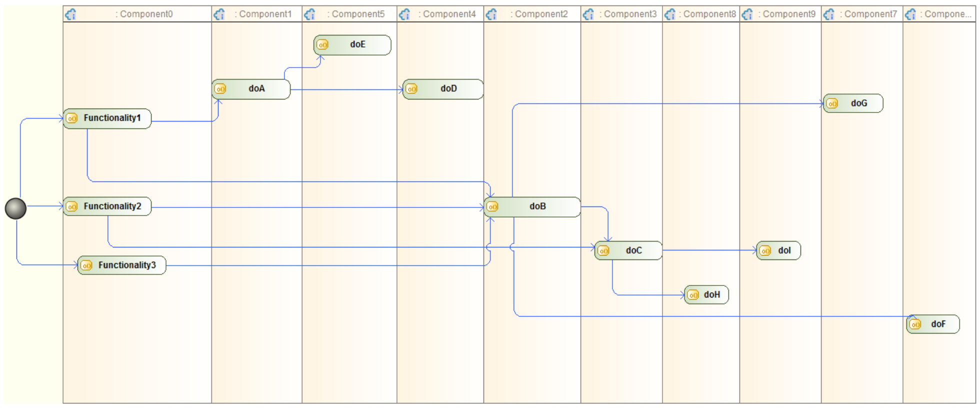


Figure 8.12: Scalability Analysis Case Study, distribution of calls within the system components.

a proxy and forwards the requests to internal components required for the execution. As an example, we see that the execution of *Functionality 1* involves the forwarding of the user request from component 0 to components 1 and 2. Component 1, in turn, requires functionalities offered by components 4 and 5. Functionalities 2 and 3 have similar behavior and involve in the computation all 10 components of the application. The allocation of component into tiers changes with the number of components in the model and the number of tiers in the system. In the most complex scenario with 10 components and 3 tiers we designated the first tier to host components 0, 1, 4 and 9; the second to components 2, 6 and 5; and the third to the remaining ones 3, 8 and 7. This allocation allows to split the load generated by the components in a balanced way across all the tiers of the system. This principle of splitting the load among all the tiers has been followed also for the generation of models with two tiers, we removed the very last tier and moved the components allocated there on the first and second tier in order maintain as much as possible an even distribution of demands.

We decided to split the demands evenly between the components. This choice has been made in order to stress the optimization algorithm, in particular in the solution of the lower level problem that has to focus the local search on all the application tiers since none of them appear to be critical. If, on the other hand, the system has a clear bottleneck, as shown in the *Constellation* case study of Chapter 3, the search procedure will find easily the best number of replicas for non critical tiers and focus most of the search effort for the critical one.

To derive the constraints on the execution time of the three functionalities offered by the system we summed up the demands of the execution of each functionality across all the involved components and multiplied this value 10 times as in [15]. We have then introduced an architectural constraint specifying that the first and third tier of the application have to be hosted on VMs with at least 2GB of memory. We did not specify any constrain on the second tier in order to allow the algorithm to explore a wider space of possible configuration, including machines with any memory size. We performed the analysis both in a single and multicloud scenario with 2 and 3 providers. In the two multicloud scenario we addressed, we introduced a constraint requiring that, if a provider is selected, it has to serve at least 20% of the incoming workload.

8.2.2 Scalability Results

The results of the scalability reported in the following try to give more insights on the effectiveness of the optimization approach we implemented in solving complex models. We first report a breakdown of the time spent during the optimization in all the phases and then discuss how the time required for each of this phase changes with the complexity of the application model. All the analysis reported in this section have been performed on a development machine equipped with an Intel Core i7 Q740 processor and 8GB of RAM. Each model has been optimized 10 times changing the seed of the optimization procedure in order to take into consideration the effect of non determinism present in many parts of the optimization procedure. Results reported in this section represent average values over these runs.

Figure 8.13 shows how the time spent during an optimization is split across the main phases of its execution. Initialization of the connections between the optimization engine, the LQN solver, the resource model database and the MILP solver as well as the load of the models have been omitted since they account for less then 0.1% of the entire optimization time. The first step of the optimization is the generation of an initial solution to the deployment problem derived by the MILP formulation introduced in Chapter 6. In all our experiments this process has been executed by an external CMPL solver[1] hosted on a virtual machine with four CPUs hosted on a Xeon E5530 and 6GB of memory. The average time required to generate the initial solution across all our experiments accounts for 4.24% of the entire time spent in the optimization. We will see in Section 8.4 a more detailed analysis of the impact of this initial solution on the optimization process. The second step in the solution of the problem is the transformation of the design time model and the initial solution into a LQN performance model. This process is performed in the *LQN Generation* phase by PalladioBench and accounts for less than 1% of the optimization time. The resulting model is then evaluated, for the very first time, in the *Initial Evaluation* phase, in order to derive an initial estimation of the QoS and the cost of the

[1]ILOG CPLEX 12.2.0.0

solution derived by the MILP formulation. This evaluation requires the solution of one performance model for each hour of the day, further evaluations of the solutions generated during the optimization process will make use of the partial solution caching mechanism illustrated in Chapter 4 and will require a smaller number of interactions with the LQN solver. The time required for the evaluation of the initial solution accounts for 2.49% of the optimization time.

The remaining, which includes also the time required to analyze the LQN model, account for 92.97% and is spent in the execution of the *Optimization* phase which represents the execution of Algorithm 1 of Chapter 7. Most of this time is spent in solving performance models generated by the different procedures.

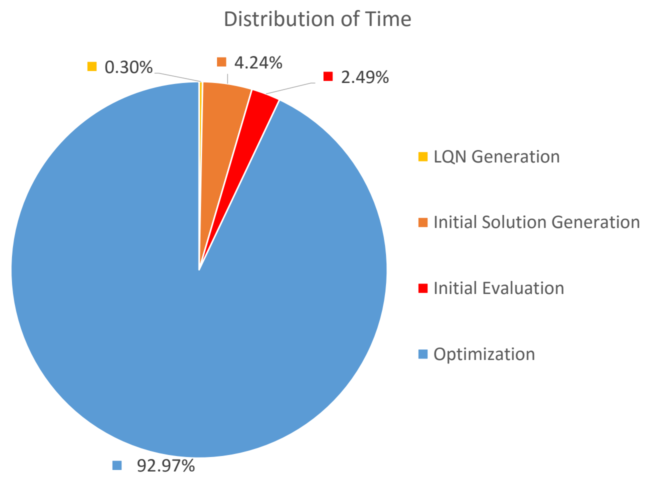


Figure 8.13: Distribution of time spent during the optimization in the mian phases

In the *Optimization* phase many solutions are generated and evaluated by LINE. Recalling the description of the optimization algorithm of Chapter 7, the main optimization procedure terminates when a number of iterations of the optimization loop that explores the higher level problem has been reached. In all of the optimization runs we have performed the best solution was not the last solution generated by the search procedure, but was created earlier in the optimization process. Exploiting the elitism principle the optimization procedure continues its exploration of the search space until the exit condition is met keeping track of the best solution. The average time spent in the optimization to find the best solution, over all the evaluated models, accounts for 42% of the time spent in the optimization. This behavior shows that the optimization algorithm is very effective in generating good candidate solutions, for this reason results obtained by running small scale optimization can be used during the development process to quickly analyze different architectural solutions, delegating the optimization of the final deployment in a later stage of the design process in which the developer team can dedicate more time to the analysis of the final deployment configuration.

Figure 8.14 shows the detailed results of the scalability analysis for deployments on a single cloud provider. The time spent for each of the optimization phases presented in the beginning of this section have been reported in order to show how the impact of each phase affect the entire optimization time. Figure 8.14a shows that the time spent in solving the MILP formulation is not heavily affected by the increased number of tiers and component, the simplest model has been solved in 39 seconds and the most difficult one in 90 seconds. A similar behavior is found in Figure 8.14b, which shows the time spent by Palladio to transform the application model into the LQN performance model. In this case the dependency of the time spent in transforming the model with the number of components is not significant, considering also that overall time required for the transformation accounts for less than 1% of the entire optimization time. All other figures, on the other hand, show a linear growth in the time spent when the number of components or the number of tiers of the application is increased. This is due to the fact that the first two phases are not involved in the solution of the LQN model, while the initial evaluation time, shown in Figure 8.14c, and the overall optimization time, shown in Figure 8.14e heavily depend on the time required to solve the LQN performance model. Both these phases show a linear growth in their execution time when the number of components grows. This growth is purely due to the increased complexity of the underlying LQN model. The time required to find the best solution, shown in Figure 8.14d grows as well with the increasing number of components but also grows with the number of tiers. This is due to the fact that adding a component does not change the size of the search space but only increases the complexity of the underlying LQN model; adding a tier, on the other hand, increases the size of the upper level problem making it harder to find the best combination of cloud services.

Figure 8.15 shows the details of the results of the scalability analysis on deployments performed on two cloud providers. We can identify here a behavior similar to the one shown in the analysis of the single cloud scenario. The time required to solve the relaxed problem, shown in Figure 8.15a, increases when the application under analysis has three tiers but does not seem to have a strong dependency with the number of components, the initial solution for the simplest model was found in 82 seconds and the most complex model has been solved in 210 seconds. The transformation phase from the application design model to the LQN performance model represents again less than 1% of the entire optimization time. The dependency of the process with the number of components, shown in Figure 8.15b, is more evident than in the single cloud case and reflects the increased complexity of the LQN model. Both the initial solution evaluation phase and the overall optimization time, shown in Figure 8.15c and Figure 8.15e, show a linear dependency with the number of components in the application that is due again to the increased time required to solve the LQN model. Finally, the time required to find the best solution, shown in Figure 8.15d, shows a similar linear dependency with the number of tiers and application components as in the single cloud scenario.

Figure 8.16 shows the same behavior in the scenario of a deployment on three cloud providers. We can recognize here the same dynamics described in Figure 8.14 and 8.15. Figure 8.16a shows two spikes in the time required to solve the relaxed problem problem to generate the initial solution for models with 3 tiers and 7 and 10 components. These spikes are due to the nature of those particular problems for which an integer solution was hard to find. For all other instances evaluated the CPLEX solver was able to find an integer solution with the same cost of the solution of the continuous relaxation of the problem, in the cases of these two particular instances CPLEX was stopped when the cost of the integer solution was 0.1% grater of the optimal cost of the continuous relaxation.

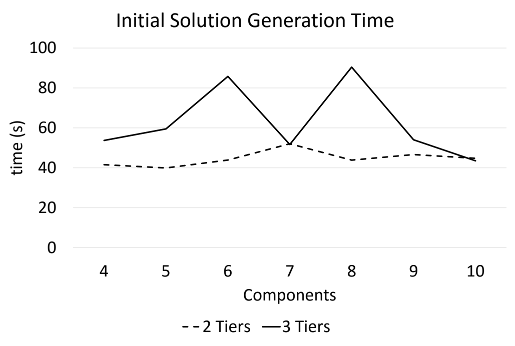
As shown in Chapther 7, SPACE4Cloud implements a non deterministic algorithm. Randomness comes into play in particular in the *TSMove* described in Section 7.4 for the selection of the tier to consider for changing the type of service to use; it also affects the selection of the new service performed by the roulette wheel mechanism. Introducing some randomness in the search for new configuration has many advantages as it allows the algorithm to better explore wide regions of the search space. When non determinism is involved in a search heuristic repeating the same optimization several times might lead to the different results. An optimization algorithm is said to be robust with respect to randomness if, repeating the same optimization process several times, all the solutions found are very close to each other. In order to test the robustness of our approach we performed several executions of the optimization procedure changing the seed used to generate randomness in the algorithm. Each of the 42 models of the scalability analysis have been optimized 10 times for a total of 420 optimization runs. Results of the optimization have been analyzed and reported in Table 8.6. For each model the table reports the number of cloud providers considered for the deployment, the number of tiers, the number of components of the application, the average and the standard deviation of the cost of the best solution and the time required to find that solution. In all the experiments the algorithm proved to be stable with respect to both the cost of the solution and the time required to find it.

The scalability analysis reported in this section show that the proposed optimization approach is able to scale well with the increasing complexity of the application under analysis,

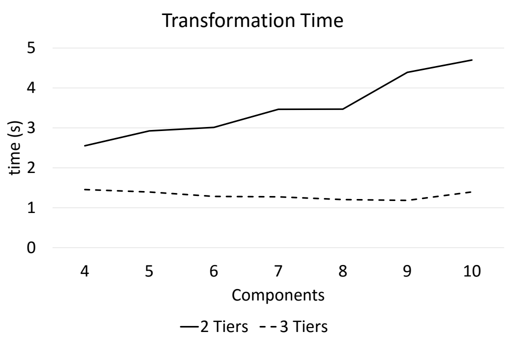
Cloud	Tier	Components	Cost ($)	σ_{Cost}($)	Time (s)	$\sigma_{Time}(s)$
1	2	4	21.53	3.69	264.82	86.05
		5	22.32	3.12	350.83	177.12
		6	30.88	6.69	322.15	127.40
		7	31.82	7.15	378.11	100.49
		8	37.81	8.18	561.63	382.46
		9	39.77	9.01	648.79	341.21
		10	44.38	8.62	725.31	377.52
	3	4	24.18	2.32	229.51	136.19
		5	26.57	3.13	396.62	139.88
		6	31.53	4.79	408.32	184.60
		7	38.02	3.49	419.14	112.24
		8	40.55	4.61	673.71	21.61
		9	45.88	10.02	657.90	147.51
		10	45.43	10.54	968.11	560.09
2	2	4	67.00	4.21	561.74	4.53
		5	80.44	4.61	602.92	11.89
		6	97.87	5.64	639.34	12.18
		7	96.37	6.43	689.86	20.80
		8	117.50	6.79	742.65	15.37
		9	122.92	8.11	967.29	25.18
		10	146.70	8.25	916.73	22.87
	3	4	78.72	1.74	489.69	167.98
		5	89.43	0.68	758.97	4.79
		6	97.99	7.32	1050.47	342.27
		7	109.54	3.04	1173.48	159.80
		8	123.44	3.51	1200.68	13.48
		9	135.42	8.52	1113.44	23.21
		10	151.93	2.76	1191.64	70.44
3	2	4	77.27	0.00	535.34	8.86
		5	93.30	4.53	807.40	9.55
		6	101.38	2.83	862.27	39.04
		7	101.85	5.05	1076.26	161.63
		8	114.17	1.04	1018.75	50.29
		9	136.80	4.78	1980.88	502.78
		10	144.21	1.30	1297.42	22.66
	3	4	79.22	0.08	663.04	11.72
		5	90.05	3.88	973.52	208.57
		6	103.87	0.07	1006.44	44.90
		7	120.62	0.76	1460.91	78.33
		8	130.89	1.00	1642.25	205.27
		9	161.50	1.95	1986.03	128.81
		10	162.60	1.35	2370.96	92.25

Table 8.6: Average, standard deviation and variance of the final cost and execution time with different random seeds

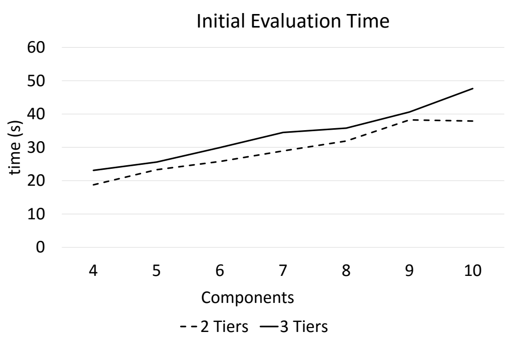
answering to the evaluation question **Q.2**. The time required to solve the most complex application design in a scenario with 3 cloud providers is of 40 minutes. Given the complexity of the problem at hand such an optimization time is acceptable for a design time analysis. The breakdown of the total optimization time into different phases and the analysis of the growth of the time required for each phase shows that all the phases scale well with the complexity of the problem.



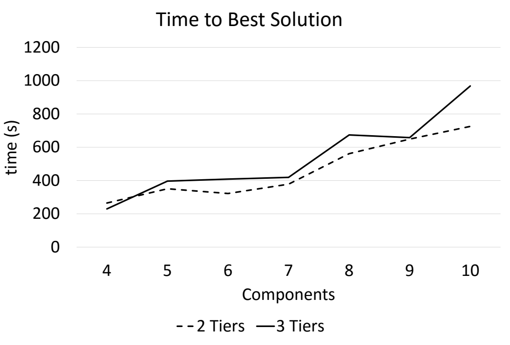
(a)



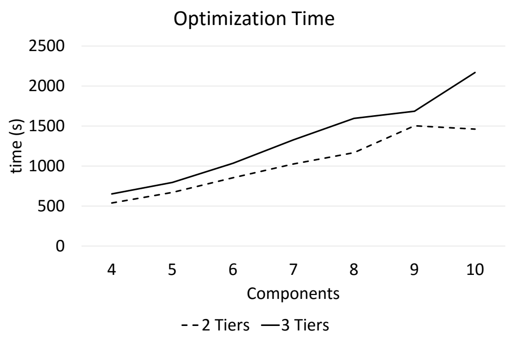
(b)



(c)

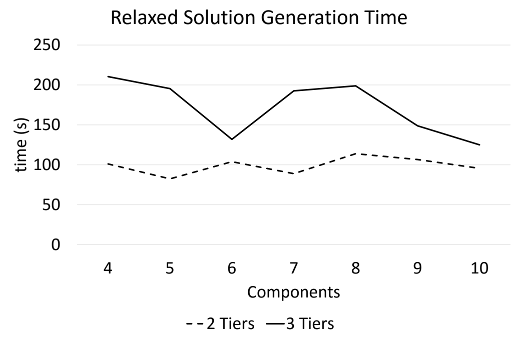


(d)

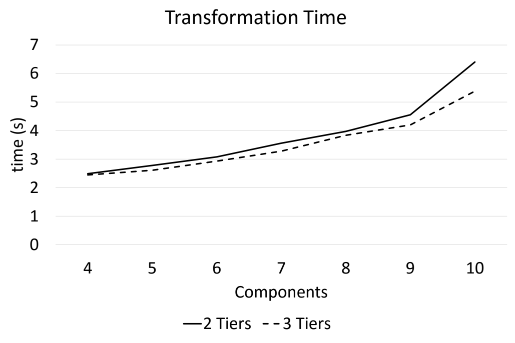


(e)

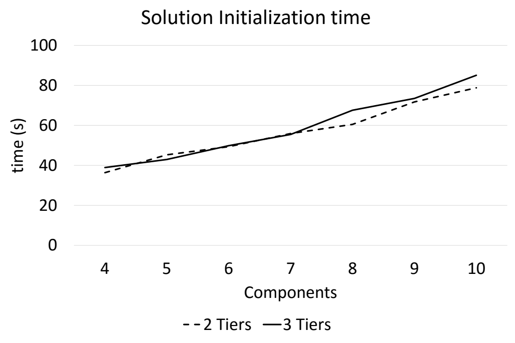
Figure 8.14: Scalability Analysis with a single candidate provider, time breakdown



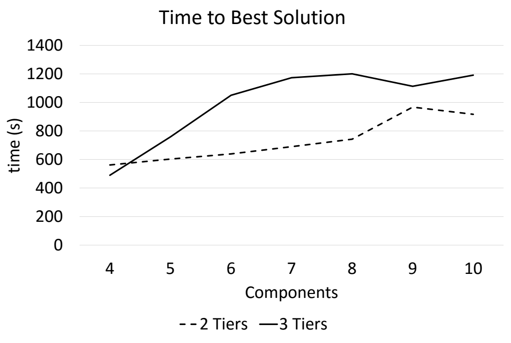
(a)



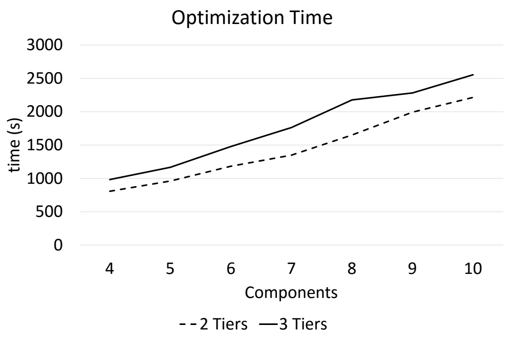
(b)



(c)

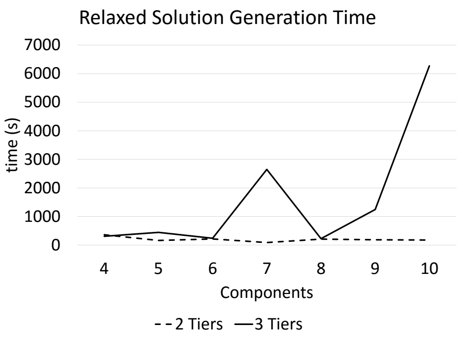


(d)

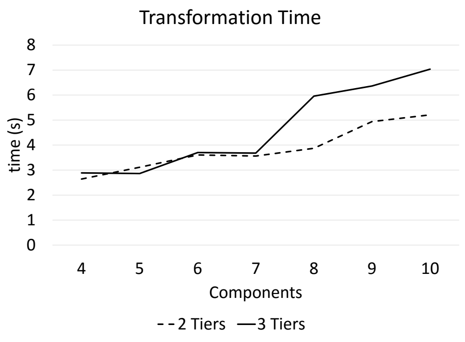


(e)

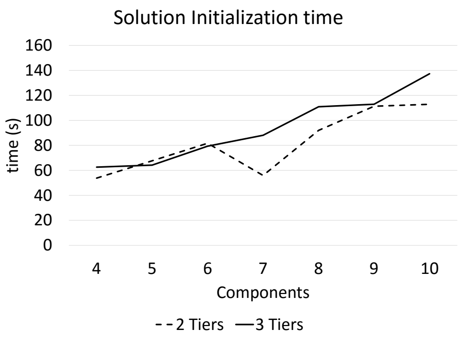
Figure 8.15: Scalability Analysis with a two candidate providers, time breakdown



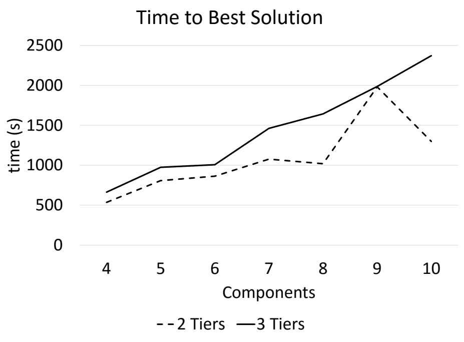
(a)



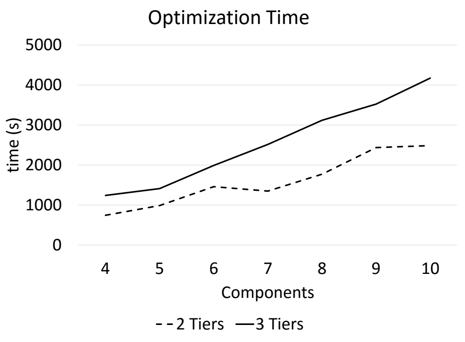
(b)



(c)



(d)



(e)

Figure 8.16: Scalability Analysis with a three candidate providers, time breakdown

8.3 Comparison with best practice heuristics

In order to evaluate the quality of the solutions derived by SPACE4Cloud we compared it with a best practice approach widely adopted by cloud providers and practitioners based on CPU utilization. As reported in [76, 79], a very common policy to decide how many replicas of resources to allocate for the infrastructure hosting an application is to fix a threshold on the maximum utilization of the resources and calculate the number of replicas so so that the threshold is never exceeded. This policy is also used by many cloud providers[1] that allow users to define such a threshold and manage automatically scaling actions in case the threshold is exceeded. For what concerns the selection of cloud services, i.e., VM types in the context of a IaaS deployment, we decided to choose the cheapest type of resource available at the cloud provider that satisfy all the architectural constraints (e.g., constraints on the minimum amount of memory).

To run the optimization reported in this section we first selected manually the type of resource to assign to each tier of the application looking in the resource model database. We then run SPACE4Cloud using a single constraint on the utilization of each tier, we also specified in the configuration of SPACE4Cloud to avoid performing the *TSMove*, so that the optimization only focuses on the lower level problem and do not change the resource type assignment.

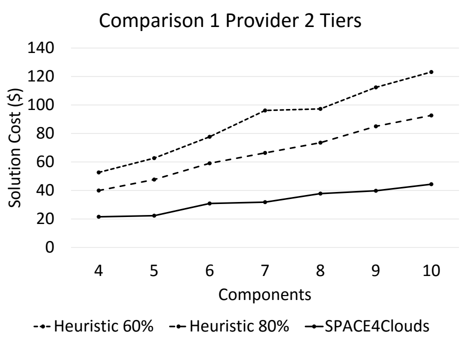
The idea of using a threshold on the utilization of the resource to indirectly control the response time of the application is given by the fact that the response time is quite smooth when the utilization of the system is low and grows quickly when the utilization approaches 100%. For this reason many approaches use a fixed threshold on the utilization to decide when the system requires more replicas. Deciding which utilization threshold to use is not an easy task, since the sudden growth in response time due to high utilization depends on many factors, like the demand of the application and the number of replicas of the resource. Our approach, on the other hand, allow application developers to define constraints directly on the response time of the application. We also allow the possibility to define constraints on the utilization of resources. In practice if a user defines a both a constraint on the utilization of a resource and on the response time of a functionality provided by that resource the more stringent constraint will be used.

We experimented two utilization thresholds at 60% and 80%, leading to two heuristics called $Heur_{60}$ and $Heur_{60}$ against which we compared the solution derived by SPACE4Cloud. Figure 8.17 compares the solution obtained by SPACE4Cloud and those obtained by the Heur_{60} and Heur_{80} heuristics for each of the models in the scalability analysis. As we can see the

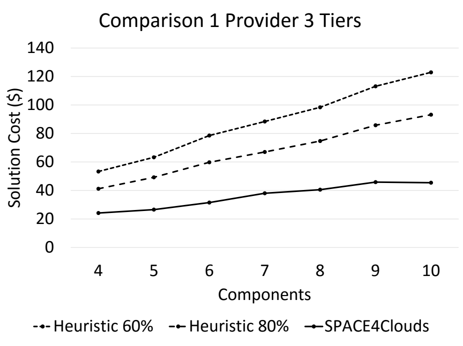
[1]http://aws.amazon.com/elasticbeanstalk/

solution derived by SPACE4Cloud always shows a significant cost reduction with respect to both heuristics. In particular with respect to Heur_{80} the average gain is of 55.9%, meaning that the cost of the solution found by SPACE4Cloud is 55.9% cheaper than the one identified by Heur_{80}, the minimum and maximum gains are 38.1% and 75% respectively. The benefit of using SPACE4Cloud over the more conservative Heur_{60} policy is even greater with an average cost reduction of 65.5%.

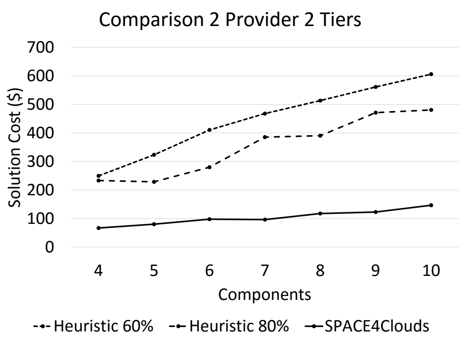
The results reported in this section shows that approach proposed in this work outperforms current best practices based on thresholds on resource utilization, as required by the evaluation question **Q.3**. Since the approach also allows the possibility to define resource utilization constraints, it is also capable of emulating the behavior of these policies.



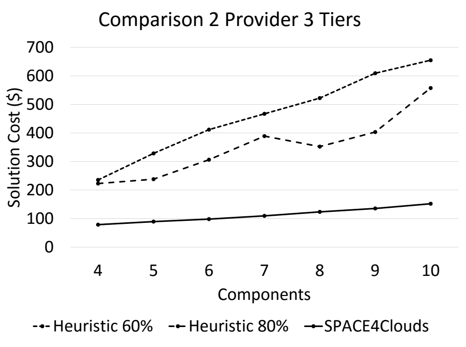
(a)



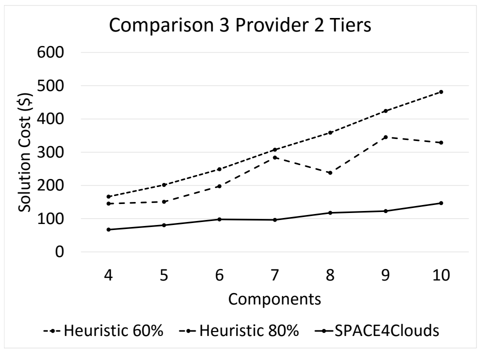
(b)



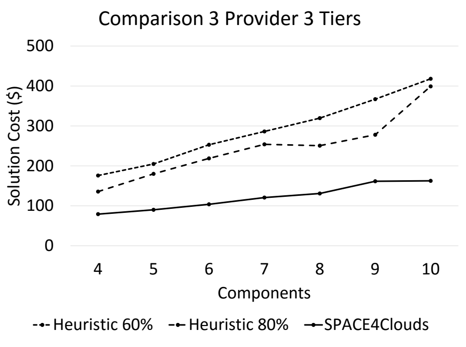
(c)



(d)



(e)



(f)

Figure 8.17: Comparison with heuristics Heur_{60} and Heur_{80}

8.4 Initial solution Evaluation

Section 8.2.2 analyzed how the time required to find the optimal solution by our hybrid heuristic algorithm varies with the complexity of the application in terms of number of components and tiers and the complexity of the deployment, in terms of number of cloud providers. In this Section we want to focus on the analysis of the generation of the initial deployment configuration, that is the solution of the problem presented in Chapter 6 and on its impact over the entire optimization procedure. Section 8.2.2 reports in Figures 8.14b, 8.15a, and 8.16a how the time required to solve the MILP formulation of the problem changes when dealing with multiple cloud providers, here we focused the analysis on a single cloud provider deployment.

Experiments have been performed on a much larger number of instance synthetically built to evaluate the sensitivity of the solution of the MILP problem with respect to many parameters. The results of this scalability analysis tailored to the MILP relaxation of the problem have been published in [17].

Results have been gathered from an Ubuntu based VirtualBox virtual machine running on an Intel Xeon Nehalem dual socket quad-core system with 32 GB of RAM. The commercial CPLEX 12.2.0.0 [1] has been used as MILP solver. To guarantee statistical independence of the results, we have considered ten different instances of each optimization run with the same configuration of parameters and averaged the results. Results reported here have been obtained by considering more than 10,000 runs.

We have considered incoming workload generated by considering the trace of a real world large Web system including almost 100 servers. The trace follows a bimodal distribution with two peaks around 11.00 and 16.00, we have used the workload derived by this trace as a reference and added random white noise to generate multiple workloads as in [16] and [52]. As we did for the sensitivity analysis of Section 8.2.2, and reported in [16, 18], the constraint on the service time of each class of requests has been set equal to 10 times the sum of the demands involved in the computation of that class. Details on the cardinality of the sets of the MILP optimization model are reported Table 8.7 and details on the models are reported in Table 8.8.

For the cases considered in this analysis CPLEX was able to find a solution on average in 3 seconds, in the very worst case, considering a system with 5 tier, 9 classes of requests and 24 time intervals the time required for the optimization was 8.72 seconds. Figure 8.18 reports a representative example and shows how the optimization time varies by changing the number of tiers and request classes.

[1] http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

Parameter	Range
β_k	$[0.1; 1]$ %
$p_{k,k'}$	$[0.01; 0.5]$ %
$\mu_{k,v}$	$[50; 2800]$ req/sec
$C_{v,t}$	$[0.06; 1.06]$ \$ per hour
$\overline{M}_i$	[1;4] GB
$\overline{N}_i$	5000 VMs
$\overline{R}_k$	$[0.005; 0.01]$ sec

Table 8.7: Ranges of model parameters

Description	Variation range
Number of tiers $\|\mathcal{I}\|$	$[1; 9]$
Time Intervals $\|\mathcal{T}\|$	$[4; 24]$
Number of Requests Classes $\|\mathcal{K}\|$	$[1; 10]$
Number of VM types $\|\mathcal{V}\|$	$[1; 12]$

Table 8.8: MILP sets cardinalities

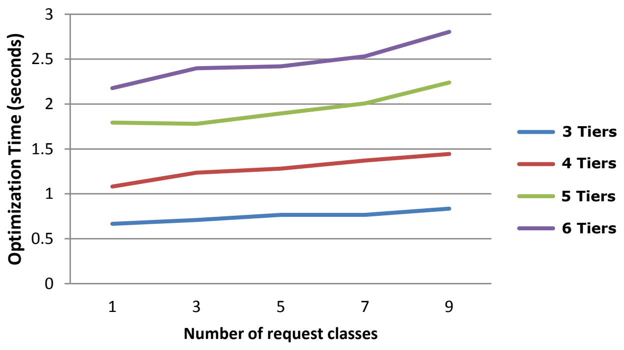


Figure 8.18: MILP optimization time varying the number of tiers and classes of requests

8.4.1 Quality Evaluation

Section 8.4 showed that the time required to solve the relaxation of the problem, presented in Chapter 6, does not grows excessively with the complexity of the application; furthermore, Section 8.2.2 shows that the time spent in generating the initial solution is a very small fraction of the total time spent by the hybrid heuristic approach to obtain the final solution. In this section we want to see how the use of this initial solution, with respect to solutions derived by the best practice heuristic called Heur$_{60}$, as reported in [79], presented in Section 8.3, impacts the overall optimization approach. We evaluated the gains of using the MILP formulation to find the initial solution in terms of reduction of the optimization time and increased quality of the final solution, meaning a reduction in its cost.

Figure 8.19 shows the execution trace of the optimization performed by SPACE4Cloud using as initial point the solution obtained by the Heur$_{60}$ heuristic, in gray, and the one derived by the MILP formulation, in black. On the x axis the time spent in the optimization process is

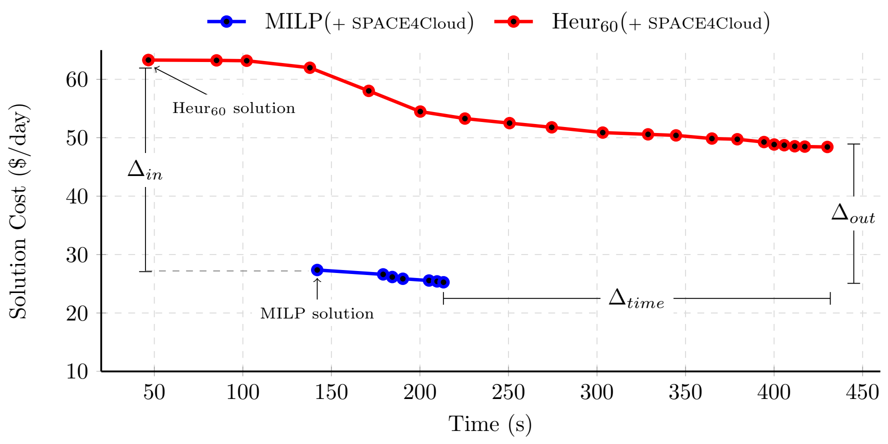


Figure 8.19: Comparison of solution costs found by SPACE4Cloud in the optimization process starting from the MILP initial solution and the Heur$_{60}$ solution.

reported (including the time required by CPLEX to gin the initial solution), the y axis reports the daily resource cost of cloud services.

We have instrumented SPACE4Cloud to export the trace and analyze how the cost of the solution changes during the optimization process. We have performed over 200 runs considering all the models used for the scalability analysis and reported the detailed results in this Table 8.9. The curves shown in Figure 8.19 comes from one of these runs but are representative of most of the behaviors we have observed. In our analysis we have reported the difference in cost between the first feasible solution found by SPACE4Cloud starting from the heuristic solution and by the MILP approach, marked as Δ_{in}. This value shows the distance between the staring points of the two optimization runs. A positive value for Δ_{in} shows that the solution derived by the MILP approach is cheaper. The initial gap of the black line represents the time required to generate the initial solution by the MILP approach. The difference between the cost of the final solutions obtained by SPACE4Cloud is represented by Δ_{out}. A positive value indicates savings obtained by the use of the MILP formulation, with respect to Heur$_{60}$, to generate the starting point of the optimization. Finally, Δ_{time} represents the time difference in the time required to find the best solution. A positive value of Δ_{time} indicate a reduction in the time required by the optimization process when using as initial solution the one derived by the MILP formulation of the problem. A negative value indicate an increase in the time required for the optimization.

As shown in Figure 8.19, the initial delay introduces by the solution of the MILP formulation is compensated by the reduced time required to find the final solution. In the comparison shown in the figure, the final solution obtained with the used of the MILP initial solution is

55% cheaper than the one obtained by the use of Heur$_6$0 and the time required to identify such a solution is reduced by half.

Table 8.9 summarizes the results achieved in the columns on the right side, under the *Quality Evaluation* header. These results have been obtained by considering 210 total runs of the entire optimization procedure performed by SPACE4Cloud starting from an initial solution generated either by using our MILP approach, the Heur$_{60}$ or the Heur$_{80}$ heuristics. Columns Δ_{in}, Δ_{out} and Δ_{time} report the relative gain/loss of using the MILP approach to generate the initial solution.

The final solution obtained by SPACE4Cloud when using the MILP approach to generate the initial solution is, on average, 37% cheaper than the one obtained using the Heur$_{60}$ heuristic and 22% cheaper than the one that uses the Heur$_{80}$ heuristic. The time required to find the final solution, again using the MILP approach, is reduced by half with respect to the one obtained using the Heur$_6$0 heuristic and by 22% with respect to the Heur$_{80}$ heuristic.

Table 8.9 shows that when three Cloud providers are considered, using the initial solution generated by MILP might increase the optimization time. Nevertheless, the final solution is, on average, 27% cheaper.

			Heur60			Heur80		
Cloud	Tier	Components	Δ_{In} %	Δ_{Out} %	Δ_{Time} %	Δ_{In} %	Δ_{Out} %	Δ_{Time} %
1	2	4	76.9%	32.5%	57.8%	69.6%	35.2%	25.9%
		5	82.5%	49.8%	47.1%	77.0%	45.2%	9.4%
		6	83.8%	47.8%	22.7%	78.7%	28.1%	68.3%
		7	90.5%	36.0%	67.5%	82.4%	33.7%	66.6%
		8	87.1%	34.0%	60.5%	82.9%	29.8%	52.3%
		9	89.6%	42.8%	57.1%	86.2%	39.6%	49.9%
		10	89.8%	44.0%	46.6%	86.4%	33.3%	56.2%
	3	4	72.1%	37.3%	57.7%	63.9%	38.1%	43.1%
		5	75.4%	39.3%	49.0%	68.4%	43.7%	18.3%
		6	80.6%	42.9%	58.4%	74.5%	39.6%	54.6%
		7	83.3%	36.0%	74.1%	78.0%	36.3%	52.0%
		8	85.4%	45.4%	21.3%	80.8%	39.5%	43.8%
		9	87.2%	44.2%	50.0%	83.1%	42.3%	41.9%
		10	89.2%	46.9%	46.2%	85.7%	47.6%	19.0%
2	2	4	86.0%	39.1%	68.7%	85.1%	44.5%	43.6%
		5	89.0%	36.2%	68.1%	84.4%	43.9%	43.0%
		6	91.4%	36.3%	71.5%	87.3%	43.3%	62.7%
		7	92.5%	42.4%	76.7%	90.9%	53.1%	67.5%
		8	92.8%	40.4%	73.9%	90.6%	48.4%	69.2%
		9	93.5%	44.5%	74.9%	92.3%	53.3%	56.2%
		10	93.9%	40.8%	78.4%	92.3%	46.5%	60.9%
	3	4	81.3%	35.7%	68.5%	80.3%	40.1%	52.6%
		5	86.2%	38.7%	59.2%	80.9%	43.1%	22.1%
		6	89.7%	38.2%	64.0%	86.2%	45.9%	43.3%
		7	90.1%	41.1%	65.0%	88.1%	46.8%	48.6%
		8	91.2%	40.5%	67.3%	86.9%	45.3%	57.9%
		9	92.5%	44.5%	77.9%	88.7%	47.7%	67.3%
		10	93.0%	40.2%	80.4%	91.8%	48.0%	68.1%
3	2	4	76.6%	22.6%	37.7%	73.2%	17.9%	-27.2%
		5	81.0%	19.6%	25.3%	74.6%	17.3%	-59.5%
		6	84.3%	30.3%	31.5%	80.2%	28.2%	-33.4%
		7	87.8%	37.8%	26.0%	86.8%	35.6%	-20.4%
		8	89.0%	37.5%	38.9%	83.4%	34.7%	-13.7%
		9	90.5%	33.3%	-2.5%	88.3%	31.8%	-79.1%
		10	91.8%	35.1%	41.3%	88.0%	33.6%	-40.1%
	3	4	74.2%	25.8%	39.2%	66.5%	22.5%	-68.2%
		5	76.9%	28.2%	25.5%	73.7%	25.3%	-71.9%
		6	82.0%	31.2%	40.8%	79.2%	28.8%	36.9%
		7	83.0%	28.7%	33.6%	80.8%	25.8%	-2.0%
		8	84.6%	29.8%	40.4%	80.4%	27.7%	-7.3%
		9	86.9%	23.9%	26.0%	82.7%	21.6%	-28.7%
		10	88.3%	30.2%	26.9%	87.8%	27.5%	-34.9%

Table 8.9: Results of the MILP quality evaluation analysis.

Chapter 9

Conclusions

"*That's all Folks!*"

Porky Pig

In this work we presented an approach that tries to simplify the process of migrating an application to the cloud by providing a methodology and a tool to support development teams in building new applications capable of running in a multi-cloud environment. We proposed a meta-model that describes cloud services and integrated it with well established modeling tools like Palladio and Modelio in order to allow application architects to specify configurations in cloud environments. We then automated the process of evaluating the QoS of the deployment configuration specified by the software architect allowing her/him to gain valuable insights on how the design reacts to different working conditions (e.g., variable incoming workload). This ability empowers the application architects to follow MDE principles and perform QoS and cost analyses early in the design stage allowing prompt modification of the architecture to tailor it better to the runtime environment. This ability has been shown by the first industrial case study by Softeam in which an early analysis of an initial architecture model revealed its inability to gracefully scale and support higher workloads. This discovery led to a re-design of part of the architecture leading to a system that could exploit better the scalability feature offered by cloud environments. A second industrial case study used the tool to evaluate different application deployments in a multi-cloud scenario.

We then focused on helping the application architect not only in the discovery of potential issues in the architecture or in a particular deployment configuration, but also in deriving an optimized deployment that minimize the cost of using cloud services and provide QoS guarantees at the same time. To derive this configuration we designed an optimization heuristic that effectively explores a wide space of possible configurations. We first formalized the problem from a mathematical point of view and showed it to be NP-hard. We then used M/G/1 queuing network models to derive a closed form formulation of the application response time and used this model to solve a relaxation of the original problem and derive a promising initial solution. This solution is then modified by our hybrid heuristic that makes use of a more accurate performance model, i.d. LQN models, to evaluate the feasibility of the application against user defined constraints.

We evaluated the applicability of the proposed approach to complex models by means of a scalability analysis that showed how the solution derived by means of our heuristic algorithm outperform those derive by policies currently used by practitioners providing an average reduction in the cost of the deployment around 55%. The scalability analysis also showed that the approach can be effectively applied to complex problems, since the optimized solution was obtained in around 40 minutes in for the most complex model considered.

The main threat to the validity of our approach is the lack of accurate data on the performance of cloud services. The optimization approaches uses a Resource Database to update the performance model with the characteristics of the cloud resource under analysis, as show in Section 4.2. While some of the information stored in this database are provided publicly by cloud providers (e.g. the cost of using such a resource or the number of cores of a particular VM type), other parameters are unknown and have to be estimated by benchmarking such resources. Furthermore the service offer of main cloud providers change very frequently both in terms of performance upgrades or cost reductions. Since a complete benchmark campaign was not feasible, we decided to integrate in our approach the results of the ARTIST[1] European project which provide benchmarking information of many cloud resources.

From the optimization point of view, we allowed users to define constraint on the response time of the application. Current best practices use constraints on resource utilization since as long as the utilization is low the response time does not change significantly. Using directly the response time might be effective only if the constraint is so high that high utilization can be tolerated. To overcome this limitation we added the possibility to take into account resource utilization constraints as well so that if both type of constraints are defined, one will dominate the other. Finally the user can specify response time constraint on a functionality offered to the end user of the application and utilization constraints on part of the resources used to provide such a functionality. In such a scenario high utilization might be tolerated on some resourced involved in the processing of the user request while key components might be kept under a utilization constraint.

[1]http://www.artist-project.eu/collaboration

Our approach has been particularly tailored to support the analysis and optimization of common web based applications but the modeling approach and the optimization techniques presented in this thesis can be extended to address different deployment problem.

An extension that we have considered takes into account the use of a mixed cloud environment in which a private infrastructure is used to process most of the workload and, when the private infrastructure is not capable of accepting more users, a public cloud is used to replicate the entire system.

In our work we considered application deployment on multiple clouds in which the entire system has been replicated on all the available providers. Another possible research line can consider a more flexible deployment solution where application tiers are deployed on different cloud providers. The analysis of such a deployment configuration should take into consideration the peculiarity of the technology used to interconnect the application tiers, which we expect will be enabled by progress in the area of network design. The deployment configuration we considered in this thesis allows to exploit high performance networks available in cloud providers datacenter and avoid the increased complexity in the analysis.

Recent years showed the appearance of Big Data frameworks and applications. These applications perform complex analysis of high volumes of data in order to extract insights with business value. The complexity of the frameworks used to process this kind of data and the size of the infrastructure required to perform such analysis make the cloud environment the default choice for many companies that want to exploit this new technology. The approach proposed in this thesis could be adapted to take into consideration the peculiarity of these applications and frameworks by the use of a more tailored performance model.

If we consider current technology trends, the raise of container based hosting services like Amazon EC2 Container Service (ECS)[1] or Google Container Engine[2] eases the use of component based approaches in the development of cloud applications. In this context an optimization technique that derive container characteristics like the container size could greatly help architects to speed up the development and testing of their application architectures.

Finally, adding the link between the runtime application execution and the models built at design time by the use of automated tools and an appropriate monitoring technology will enable a full DevOps approach. The main idea is to use real monitoring data to refine application performance parameters (e.g., service demands, number of users at peak, switch probabilities in end user behaviour, etc.). In this vision development decisions and operational aspects are considered together and the barrier between design time and runtime can be dissolved.

[1]https://aws.amazon.com/it/blogs/aws/cloud-container-management/
[2]https://cloud.google.com/container-engine/

Bibliography

[1] Argouml - http://argouml.tigris.org/. 10

[2] Magicdraw - http://www.nomagic.com/products/magicdraw.html. 10

[3] Modeling software kitt - https://moskitt.gva.es. 10

[4] Omg model-driven architecture. 11

[5] Rational software architect - http://www-03.ibm.com/software/products/it/ratisoftarch. 10

[6] Staruml - http://staruml.io. 10

[7] Uml designer - http://www.umldesigner.org/. 10

[8] Upupa - http://www.modelexecution.org/. 10

[9] Bernardetta Addis, Danilo Ardagna, Barbara Panicucci, Mark S. Squillante, and Li Zhang. A hierarchical approach for the resource management of very large cloud platforms. *IEEE Trans. Dependable Sec. Comput.*, 10(5):253–272, 2013. 123

[10] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *Software Engineering, IEEE Transactions on*, PP(99):1–1, 2013. 4, 11

[11] Aldeida Aleti, Stefan Stefan Björnander, Lars Grunske, and Indika Meedeniya. Archeopterix: An extendable tool for architecture optimization of aadl models. In *Proc. of Workshop MOMPES 2009*, 2009. 15

[12] Amazon Inc. AWS Elastic Beanstalk. http://aws.amazon.com/elasticbeanstalk/. 71

[13] Nicolas Ferry Stepan Seycek Elisabetta Di Nitto Antonin Abherve, Marcos Almeida. Modacloudml ide final version. Public deliverable, 2015. 22, 25, 31

[14] D. Ardagna, B. Panicucci, M. Trubian, and L. Zhang. Energy-aware autonomic resource allocation in multitier virtualized environments. *Services Computing, IEEE Transactions on*, 5(1):2–19, 2012. 68, 123

[15] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *Software Engineering, IEEE Transactions on*, 33(6):369–384, June 2007. 66, 124

[16] Danilo Ardagna, Sara Casolari, Michele Colajanni, and Barbara Panicucci. Dual time-scale distributed capacity allocation and load redirect algorithms for cloud systems. *Journal of Parallel and Distributed Computing*, 72(6):796 – 808, 2012. 137

[17] Danilo Ardagna, GiovanniPaolo Gibilisco, Michele Ciavotta, and Alexander Lavrentev. A multi-model optimization framework for the model driven design of cloud applications. In *SBSE 2014 Proc.* 2014. 6, 61, 75, 137

[18] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Trans. Software Eng.*, 33(6):369–384, 2007. 137

[19] George Kousiouris Danilo Ardagna Athanasia Evangelinou, Michele Ciavotta. A joint benchmark-analytic approach for design-time assessment of multi-cloud applications. In *Proceedings of the 1st International Conference on Cloud Computing, Information Technology, Big Data and Big Data Management*, 2015. 56

[20] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *Software Engineering, IEEE Transactions on*, 30(5):295–310, 2004. 11

[21] Matthias Becker, Steffen Becker, and Joachim Meyer. SimuLizar: Design-Time Modelling and Performance Analysis of Self-Adaptive Systems. In *Proceedings of Software Engineering 2013 (SE2013), Aachen*, 2013. 11

[22] Matthias Becker, Markus Luckey, and Steffen Becker. Performance analysis of self-adaptive systems for requirements validation at design-time. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA ’13, pages 43–52, New York, NY, USA, 2013. ACM. 11

[23] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009. 4, 11, 16, 22

[24] Christian Blum, Jakob Puchinger, Günther R Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135–4151, 2011. 20

[25] F. Brosig, P. Meier, S. Becker, A. Koziolek, H. Koziolek, and S. Kounev. Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures. *Software Engineering, IEEE Transactions on*, 41(2):157–175, Feb 2015. 4, 12

[26] Omid Bushehrian. The application of fsp models in automatic optimization of software deployment. In Khalid Al-Begain, Simonetta Balsamo, Dieter Fiems, and Andrea Marin, editors, *Analytical and Stochastic Modeling Techniques and Applications*, volume 6751 of *Lecture Notes in Computer Science*, pages 43–54. Springer Berlin Heidelberg, 2011. 18

[27] Claudia Canali and Riccardo Lancellotti. Exploiting ensemble techniques for automatic virtual machine clustering in cloud systems. *Automated Software Engineering*, 21(3):319–344, 2014. 71

[28] G. Canfora, M. Di Penta, R. Esposito, and M.L. Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1069–1075. ACM, 2005. 17

[29] Giuliano Casale and Mirco Tribastone. Modelling exogenous variability in cloud deployments. *SIGMETRICS Perform. Eval. Rev.*, 40(4):73–82, April 2013. 6, 39, 62

[30] Benoît Colson, Patrice Marcotte, and Gilles Savard. An overview of bilevel optimization. *Annals of operations research*, 153(1):235–256, 2007. 74

[31] Vittorio Cortellessa, Antinisca Di Marco, Romina Eramo, Alfonso Pierantonio, and Catia Trubiani. Approaching the model-driven generation of feedback to remove software performance flaws. In *EUROMICRO-SEAA*, pages 162–169. IEEE Computer Society, 2009. 14

[32] MauroLuigi Drago, Carlo Ghezzi, and Raffaela Mirandola. Qvtr2: A rational and performance-aware extension to the relations language. In Juergen Dingel and Arnor Solberg, editors, *Models in Software Engineering*, volume 6627 of *Lecture Notes in Computer Science*, pages 328–328. Springer Berlin Heidelberg, 2011. 13

[33] B.K. Eames, S.K. Neema, and R. Saraswat. Desertfd: a finite-domain constraint based tool for design space exploration. *Design Automation for Embedded Systems*, 14(1):43–74, 2010. 19

[34] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In Lisa O'Conner, editor, *IEEE CLOUD 2013 Proc.*, pages 887–894. IEEE Computer Society, 2013. 10

[35] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *Software Engineering, IEEE Transactions on*, 35(2):148–161, March 2009. 22, 62

[36] Greg Franks and Murray Woodside. Performance of multi-level client-server systems with parallel service operations. In *Proceedings of the 1st International Workshop on Software and Performance*, WOSP '98, pages 120–130, New York, NY, USA, 1998. ACM. 7

[37] Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 512–521, Piscataway, NJ, USA, 2013. IEEE Press. 17

[38] Gartner Group. Hype Cycle for Cloud Computing, 2014. https://www.gartner.com/doc/2807621/hype-cycle-cloud-computing-, 2014. 1

[39] Fred Glover. Tabu search: part i. *ORSA Journal on computing*, 1(3):190–206, 1989. 60

[40] Fred Glover and Manuel Laguna. Tabu search. In *Handbook of Combinatorial Optimization*, pages 2093–2229. Springer, 1999. 84

[41] Daniel Gmach, Jerry Rolia, and Ludmila Cherkasova. Selling t-shirts and time shares in the cloud. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on*

Cluster, Cloud and Grid Computing (Ccgrid 2012), CCGRID '12, pages 539–546, Washington, DC, USA, 2012. IEEE Computer Society. 71

[42] V. Grassi, R. Mirandola, and A. Sabetta. From design to analysis models: a kernel language for performance and reliability analysis of component-based systems. In *Proc. of the Workshop WOSP 2005*, 2005. 11

[43] A. Gunka, S. Seycek, and Kuhn H. Moving an application to the cloud an evolutionary approach. In *MultiCloud*, 2013. 116

[44] Ernest Friedman Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA, 2003. 13

[45] E.K. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen. Components, platforms and possibilities: towards generic automation for mda. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 39–48. ACM, 2010. 19

[46] Robert G Jeroslow. The polynomial hierarchy and a simple model for competitive analysis. *Mathematical programming*, 32(2):146–164, 1985. 74

[47] Anne Koziolek. *Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes*. PhD thesis, Institut für Programmstrukturen und Datenorganisation (IPD), Karlsruher Institut für Technologie, Karlsruhe, Germany, July 2011. 16, 17

[48] Anne Koziolek, Danilo Ardagna, and Raffaela Mirandola. Hybrid multi-attribute qos optimization in component based software systems. *Journal of Systems and Software*, 86(10):2542 – 2558, 2013. 17, 61

[49] Anne Koziolek, Heiko Koziolek, and Ralf Reussner. PerOpteryx: Automated Application of Tactics in Multi-objective Software Architecture Optimization. In *QoSA 2011 Proc.*, QoSA-ISARCS '11, pages 33–42, New York, NY, USA, 2011. ACM. 4

[50] Anne Koziolek and Ralf Reussner. Towards a generic quality optimisation framework for component-based system models. In *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, CBSE '11, pages 103–108, New York, NY, USA, 2011. ACM. 16

[51] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010. 11

[52] Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy, and Guofei Jiang. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing*, 12(1):1–15, 2009. 137

[53] Alexander Lavrentev. An optimization approach for cloud providers selection and capacity allocation for multi-iaas systems. Master's thesis, Politecnico di Milano, 2013. 79

[54] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984. 109

[55] Rui Li, Ramin Etemaadi, Michael T. M. Emmerich, and Michel R. V. Chaudron. An evolutionary multiobjective optimization approach to component-based software architecture design. In *Proc. of Congress, CEC 2011*, 2011. 15

[56] Adam Lipowski and Dorota Lipowska. Roulette-wheel selection via stochastic acceptance. *CoRR*, abs/1109.3627, 2011. 20

[57] Grzegorz Loniewski, Etienne Borde, and Emilio Insfran. Towards a model driven refinement process through architecture evaluation. In *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*, NFPinDSML 2012, pages 4:1–4:6, New York, NY, USA, 2012. ACM. 14

[58] HelenaR. Loureno, OlivierC. Martin, and Thomas Sttzle. Iterated local search: Framework and applications. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 363–397. Springer US, 2010. 60

[59] Anne Martens, Heiko Koziolek, Steffen Becker, and Ralf Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *Proc. of Conference WOSP/SIPEW 2010*, 2010. 12

[60] Raffaela Mirandola, Pasqualina Potena, and Patrizia Scandurra. Adaptation space exploration for service-oriented applications. *Science of Computer Programming*, 80, Part B:356 – 384, 2014. 18

[61] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts. Constraint-based design-space exploration and model synthesis. In *Embedded Software*, pages 290–305, 2003. 19

[62] Qais Noorshams, Anne Martens, and Ralf Reussner. Using quality of service bounds for effective multi-objective software architecture optimization. In *Proceedings of the 2Nd International Workshop on the Quality of Service-Oriented Software Systems*, QUASOSS '10, pages 1:1–1:6, New York, NY, USA, 2010. ACM. 16

[63] OMG. *UML Profile for Schedulability, Performance, and Time Specification*, 2005. 9

[64] OMG. A uml profile for marte: Modeling and analysis of real-time embedded systems, 2008. 9

[65] Mohamed Ouzineb, Mustapha Nourelfath, and Michel Gendreau. Tabu search for the redundancy allocation problem of homogenous series-parallel multi-state systems. *Rel. Eng. & Sys. Safety*, 93(8):1257–1272, 2008. 17

[66] Trevor Parsons and John Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–91, 2008. 13

[67] J.F. Perez and G. Casale. Assessing sla compliance from palladio component models. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 409–416, Sept 2013. 22, 62

[68] Juan F. Pérez, Giuliano Casale, and Sergio Pacheco-Sanchez. Estimating computational requirements in multi-threaded applications. *IEEE Trans. Software Eng.*, 41(3):264–278, 2015. 7

[69] Marian Petre. Uml in practice. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 722–731, Piscataway, NJ, USA, 2013. IEEE Press. 10

[70] T. Saxena and G. Karsai. Mde-based approach for generalizing design space exploration. *Model Driven Engineering Languages and Systems*, pages 46–60, 2010. 19

[71] SOFTEAM. Modelio. The open source modeling environment. https://www.modelio.org, 2015. 9, 23

[72] T Stützle. Local search algorithms for combinatorial problems. *Darmstadt University of Technology PhD Thesis*, 1998. 20

[73] E-G Talbi. A taxonomy of hybrid metaheuristics. *Journal of heuristics*, 8(5):541–564, 2002. 20

[74] El-Ghazali Talbi. *Metaheuristics - From Design to Implementation*. Wiley, 2009. 12, 20, 84, 86, 101

[75] A.P.A. van Moorsel and K. Wolter. Analysis and algorithms for restart. In *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, pages 195–204, Sept 2004. 84

[76] Andreas Wolke and Gerhard Meixner. Twospot: A cloud platform for scaling out web applications dynamically. In *ServiceWave 2010 Proc.*, 2010. 134

[77] M. Woodside, D.C. Petriu, D.B. Petriu, H. Shen, T. Israr, and J. Merseguer. Performance by unified model analysis (puma). In *Proc. of Workshop WOSP 2005*, 2005. 12

[78] Jing Xu. Rule-based automatic software performance diagnosis and improvement. In *Proc. of Workshop WOSP 2008*, 2008. 13

[79] Xiaoyun Zhu, Donald Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret Mckee, Chris Hyser, Daniel Gmach, Robert Gardner, Tom Christian, and Ludmila Cherkasova. 1000 islands: An integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12(1):45–57, March 2009. 134, 138