An Integrated Modeling Framework for Managing the Deployment and Operation of Cloud Applications

by

Mohammad Hamdaqa

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the proposal, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Cloud computing can help Software as a Service (SaaS) providers to take advantage of the sheer number of cloud benefits such as, agility, continuity, cost reduction, autonomy, and easy management of resources. To reap the benefits, SaaS providers should create their applications to utilize the cloud platform capabilities. However, this is a daunting task. First, it requires a full understanding of the service offerings from different providers, and the meta-data artifacts required by each provider to configure the platform to efficiently deploy, run and manage the application. Second, it involves complex decisions that are specified by different stakeholders. Examples include, financial decisions (e.g., selecting a platform to reduces costs), architectural decisions (e.g., partition the application to maximize scalability), and operational decisions (e.g., distributing modules to insure availability and porting the application to other platforms). Finally, while each stakeholder may conduct a certain type of change to address a specific concern, the impact of a change may span multiple models and influence the decisions of several stakeholders.

These factors motivate the need for: (i) a new architectural view model that focuses on service operation and reflects the cloud stakeholder perspectives, and (ii) a novel framework that facilitates providing holistic as well as partial architectural views, and generating the required platform artifacts by fragmenting the model into artifacts that can be easily modified separately.

This PhD research devises a novel architecture framework, "The 5+1 Architectural View Model", for cloud applications, in which each view corresponds to a different perspective on cloud application deployment. The architectural framework is realized as a cloud modeling framework, called "StratusML", which consists of a modeling language that uses layers to specify the cloud configuration space, and a transformation engine to generate the configuration space artifacts. The usefulness and practical applicability of StratusML to model multi-cloud and multi-tenant applications have been demonstrated though a representative domain example. Moreover, to automate the framework evolution as new concerns and cloud platforms emerge, this research work introduces also a novel schema matching technique, called "Liberate". Liberate supports the process of domain model creation, evolution, and transformations. Liberate helps solve the vendor lock-in problem by reducing the manual efforts required to map complex correspondences between cloud schemas whose domain concepts do not share linguistic similarities. The evaluation of Liberate shows its superiority in the cloud domain over existing schema matching approaches.

Acknowledgements

I would like to convey my heartfelt gratitude and sincere appreciation to all the people who made this work possible.

First and foremost, I would like to thank my supervisor Dr. Ladan Tahvildari for giving me the opportunity and freedom to explore my research interests. During my Ph.D., Ladan was a sister and a mentor. Her continuous support, encouragement, and confidence in my success were the reason I am presenting this today. Her advice did not only have an impact on my research, but it will remain invaluable throughout the rest of my career.

I would like to thank the members of my dissertation committee: Professor Eleni Stroulia, my external examiner, for taking the time out of her busy schedule to read my thesis and provide me with her insightful suggestions, Professor Daniel M. Berry my internal-external examiner for his invaluable comments on my thesis, Professor Kostas Kontogiannis, my grand-advisor, for his inspiring remarks throughout my PhD, which broadened my perspectives in research, Professor Derek Rayside for his encouraging comments and indispensable support, and Professor Sue Horton for serving as the thesis examination chair.

I am grateful to have had the opportunity to meet Professor Hausi Muller at the beginning of my Ph.D. I cannot thank Hausi enough for all the valuable feedback, sweet gestures and kind support during the several CSER and CASCON meetings.

I would like to express my sincere appreciation to Dr. Marin Litoiu for exposing me to the SAVI network by inviting me to present my work in cloud computing and schema matching to the SAVI community. Also, I want to express my gratitude to Dr. Patrick Martin for his valuable feedback and remarks during the ORF-RE project on Ultra-Large-Scale Services meeting, and for giving me the opportunity to collaborate with the DSL lab.

I am very lucky to have been given the opportunity to be a part of Sceneverse Inc. family. Through my work with Sceneverse, I had the chance to meet so many wonderful people and professionals who had a significant impact on my professional career. Many thanks to Neil Lachapelle, David de Weerdt, Brian Campbell, Lindsay Stewart, Tanya McGinnity and James Swidersky for providing a pleasant and friendly working and research environment.

I am indebted to my colleagues in the STAR lab and the wider members of the University of Waterloo. Needless to mention, Dr. Amir Keyvan Khandani, Dr. Krzysztof Czarnecki, Dr. Wojciech Golab, Dr. Vijay Ganesh, Dr. Gordon B. Agnew, Dr. Fakhri

Karray, Dr. Mahesh Tripunitara, Dr. Ashraf Aboulnaga and many others for their contribution to my academic life. During my Ph.D. research, I learned many new things from each one of you.

Finally to all my friends in Waterloo, the Williams Group, Habeeb Group, the Waterloo Cool Guys, the Gravitational Waves Group, Alkhateeb Family, and My Fishing Buddies thank you for the happy moments and making this Ph.D. journey an enjoyable and wonderful experience.

Last but not least, I would like to thank my wife Ayesha whom without her not only this thesis will be empty but my life will be obsolete. I would like to thank Ayesha for standing beside me throughout this journey. Thank you for all the love and support and sacrifice you have made through bad times and for being a part of my happiness in the good times. Thank you for being with me, for putting up with me, for taking care of me, for inspiring me, for helping me, for showing me how to be a better me and for understanding me without me having to say a word. Thank you because simply without you there is no I. I love you Ayesha!

This work was financially supported by Ontario Graduate Scholarship (OGS), University of Waterloo Presidents Graduate Scholarship (PGS), The Queen Elizabeth II Graduate Scholarships in Science and Technology, and Mitacs Accelerate.

Dedication

To my best friend and unison
the wind beneath my wings
my wife Ayesha

To the joy of my life
the pure definition of happiness
without whom, I would have finished this thesis two years earlier :)
my beautiful children Lujain, Adam and Dina

To those who witnessed my past
Who believe in my future
accepts me the way I am
and will never let me down
My parents (Adnan and Nabela)

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

"Works of imagination should be written in very plain language; the more purely imaginative they are the more necessary it is to be plain."
– Samuel Taylor Coleridge, 1834

It is no wonder Gartner's expects cloud adoption to hit $250 billion by 2017 [56]. Cloud computing has changed the traditional computing model, in which organizations should invest in building sophisticated infrastructures to support their need for applications that are always available and reachable, and can scale rapidly to meet changes in business demand. Cloud providers offer distributed infrastructures that support fault tolerance and capacity on-demand as a service in an affordable "pay-per-use" basis. Moreover, they offer *software-defined* platforms that abstract the complexity of application delivery to enable their rapid deployment and easy management through configuration.

This new delivery model changes the way development teams need to think about and deal with the underlying resources while building and managing their applications. Today, organizations do not question whether to adopt cloud computing. Instead, they question what type of cloud deployment they should consider to address their performance and security requirements, and how to leverage the cloud capabilities to satisfy their specific business goals. Moreover, the diversity of cloud providers' offers and services creates an obstacle with respect to the application interoperability and portability. This diversity increases the complexity of developing and managing cloud applications to operate on *multi-cloud* platforms.

This thesis investigates the process of building, deploying and managing cloud applications (a.k.a., Cloud DevOps process). The thesis describes a new methodology supported

by a modeling framework to enable organizations that build cloud applications, such as SaaS providers, to unbiasedly exploit the cloud platform building blocks to leverage the flexibility, reliability, and scalability that these platforms provide to the application layer in a cost effective manner.

1.1 Motivations

Cloud computing shifts the control to provider's side. While this shift helps organizations to focus on their business functionalities, it is still the responsibility of the platform consumers and SaaS providers to plan for their service operation. For example, consumers should architect their applications to utilize the platform components to build services that are widely reachable, scalable and available. This section provides examples that emphasize this responsibility in order to guarantee the desired service operation and highlights the challenges of the cloud development and operation process.

1.1.1 Responsibility of the Cloud Consumers

The following is evidence that consumers should not rely solely on providers support.

- On April 21^{st} 2011, a failure in the Amazon Elastic Block Store (EBS) on the US East Region availability zone took down a large number of businesses including many cloud platform providers such as Heroku, Reddit, Quora, and many others. Lydia Leongas; the research vice president at Gartner noted "Customers who rely upon Amazon to run their businesses will, and should, think hard about the resiliency of their architectures"[1]. Since that incident, several outages of cloud services have been reported that cover all cloud providers at all service levels (see Table1.1). The main lessons to learn from such failures are: first, it is the consumers' responsibility to plan for failure and to read their (SLA) contracts, second it is not enough to handle availability and fault recovery at the infrastructure layer; planning for fault recovery should also be addressed and automated at the upper layers.

- Even though elasticity is an essential cloud computing capability, current cloud platforms do not fully support dynamic elasticity. This limited support is due to some technical challenges and to liability concerns. For example the limitation on rapid

[1] http://blogs.gartner.com/lydia_leong/2011/04/21

scale-down capabilities of the cloud applications as well as the way these applications are structured, which does not allow full components controllability pose technical difficulties on providers to support efficient dynamic elasticity. On the other hand, the lack of transparency between the cloud providers and their consumers, which can be represented in the lack of a formal way to convey the consumer intention to scale up and down on-demand make it risky for the providers to support dynamic elasticity to avoid any service level agreement violation. Moreover, cloud consumers cannot fully trust the cloud providers to perform auto-scale on their behalf without their consent. This is because these decisions are reflected on the consumers' bill (i.e., in the case of scale-up) and may disrupt their business stability (i.e., in the case of scale-down). As a result, even with current cloud development platforms, it is still the responsibility of the cloud developers to divide their application into scalable units, keep track of these components at run-time, and define the set rules that specify how each scalable unit will respond to the different environment changes and user demand requests.

Accordingly, while cloud platforms provide services to facilitate deploying and managing applications in the cloud, it is the platform consumer's task to select the provider that offers the cheapest and best offer, then leverage the platform capabilities to ensure agility and continuity and assure the application portability.

1.1.2 Challenges of Architecting Cloud Applications

Architecting cloud applications to leverage the underlying platform and at the same time ensure portability is challenging. It requires a full understanding of the service offerings from different providers and the meta-data artifacts required by each provider to deploy, run and manage an application. Moreover, it involves complex decisions that are specified by various stakeholders; such as, financial decisions (e.g., selecting a platform to reduce costs), architectural decisions (e.g., partitioning the application to maximize availability), and operational decisions (e.g., distributing modules to ensure availability). The following is a list a list of the main challenges that motivates the research in this thesis:

(a) The Vendor Lock-in Problem: The cloud "pay per use" model promotes business agility and promises cost savings. However, for this model to make sense, customers should have the ability to port their applications between providers to maximize business opportunities. Without portability, migrating applications will require partially

Service Outage	**Date**	**Duration**	**Reason of outage**	**Results**
Twitter Service	2/25/2011 3/27/2011	1 to 4 hours per outage	over-capacity	significant delays in delivering messages to SMS and Facebook
Gmail and Google Apps Services	2/27/2011	2 days	a bug in one of Google's updates	login errors and empty mailboxes
Netflix Streaming Service	3/22/2011	4 to 8 hours	rare technical issue	prevented customers from accessing their queues and streaming content
Amazon Web Services	4/21/2011	4 Days	stuck volumes caused read and write problems so the AWS operators had to disable all control APIs for the degraded EBS cluster. There were two services involved in this outage, Amazon EC2 and Amazon RDS Service.	many customers including cloud platform providers were taken offline
Salesforce	7/10/2012	2 days	a power outage caused by faulty equipment upgrades in the company's West Coast data centers	many customers were unable to access the companys services
Microsoft Windows Azure	10/30/2013	$>$ 20 hours	a subcomponent of the system failed worldwide.	every single Azure region was affected (including West US, West Europe, Southeast Asia, South Central US, North Europe, North Central US, East Asia, and East US)
Dropbox	1/10/2014	2 days	a scripting glitch caused OS upgrades to be applied on actively running machines during routine maintenance.	5 percent of users were having trouble syncing files from the desktop client, and about 20 percent were having issues with Dropbox's mobile applications

Table 1.1: A Sample of Cloud Service Outages 2011-2014

modifying or even rewriting the application, which is difficult and costly. Therefore, customers will be locked into a particular vendor. This problem is usually referred to as the "vendor lock-in" problem [125]. This problem is further escalated if an organization decided to adopt a hybrid cloud model for security reasons, as an example.

(b) Deployment Architecture Mismatch: Cloud computing promotes reuse at all levels (i.e., Infrastructure, Platform, and Software). This has been made applicable, *within each provider*, through providing standard service-based interfaces, interaction protocols, and architectural styles (e.g., RESTful). Unfortunately, as noted by Garlan, "as the level of reuse and the complexity of assumption increase, architectural mismatch become more of an issue" [55] that requires advanced software engineering solutions. A mismatch occurs due to hidden assumptions about (i) the application domain, (ii) components at the same level of abstraction, and (iii) the infrastructure. Cloud providers use customizable virtual machines that arguably can deal with a mismatch at the code level by allowing the software components to run on a variety of low-level infrastructures. However, cloud dynamics represented by the multiple providers and continuous updates in the platform, infrastructure and applications create a mismatch at the architectural deployment level. Therefore, an architecture description language that makes these assumptions explicit is needed.

(c) Lack of Domain Concepts and Methodologies: Although they share many concepts with existing development paradigms (e.g., real time, service-oriented and distributed computing); cloud applications have their characteristics, domain requirements, and development process. There is a need for a modeling language that supports the cloud application development process and provides the concepts required to (i) address the cloud specific characteristics, such as malleability (i.e., ability to be re-used, re-configured, re-combined, and re-composed), availability and self-adaptivity, (ii) specify the deployment model for a multi-tenant and web-farm friendly application so that the application can be distributed, parallelized, and hosted in multiple locations [67], and (iii) support patterns that enable loosely coupled asynchronous interactions and late binding of services.

(d) Quality Assurance: Current software architectures lack the elements to address the cloud quality attributes. Moreover, consumers usually overlook these characteristics as they rely on the providers support. However, as explained earlier, providers poorly manage and partially support these characteristics. These factors urge for a language that integrates quality of service and performance models and techniques with the cloud models; to facilitate design time and runtime analysis to deal with the different cloud trade-offs.

(e) The Gap Between Cloud Stakeholders: Existing cloud modeling frameworks address the requirements of each stakeholder individually. An efficient modeling framework should provide a holistic view that facilitates collaboration between the different stakeholders at the various cloud service levels.

Several academic and industrial projects aim at addressing the aforementioned challenges (see [6], [17], [57] to mention few). The results from these projects are necessary to promote interoperability and contribute to prevent vendor lock-in, but they are not sufficient to adequately manage the complexity of development and administration of multi-cloud systems [6].

1.2 Problem Description and Research Focus

The challenges explained in the last section uncovered two research gaps:

1. The first research gap is related to the need for an **architectural framework** to address the complexity of the cloud *operational requirements* and facilitate *collaboration* between the different cloud application stakeholders.

2. The second research gap is related to the need for a **configuration generator** that is able to (i) generate the configuration space artifacts based on the target vendor specifications in order to address the vendor lock-in problem, and (ii) bridge the gap between the architectural models and operational perspectives through model transformation.

This research is investigating the process of building, deploying and managing cloud applications. *The goal is to build a multi-cloud framework that provides the cloud platforms users (SaaS providers) with the tools required to exploit cloud platforms capabilities in order to architect malleable cloud applications that can guarantee the desired service level and minimize operational costs.*

1.3 Approach and Research Questions

To tackle this problem, a Model Driven Engineering (MDE) approach has been adopted. An overview of the research approach is depicted in Figure 1.1. The approach aims to provide answers to four research questions. For each of these questions, we explain its objective and the methodology followed to address it.

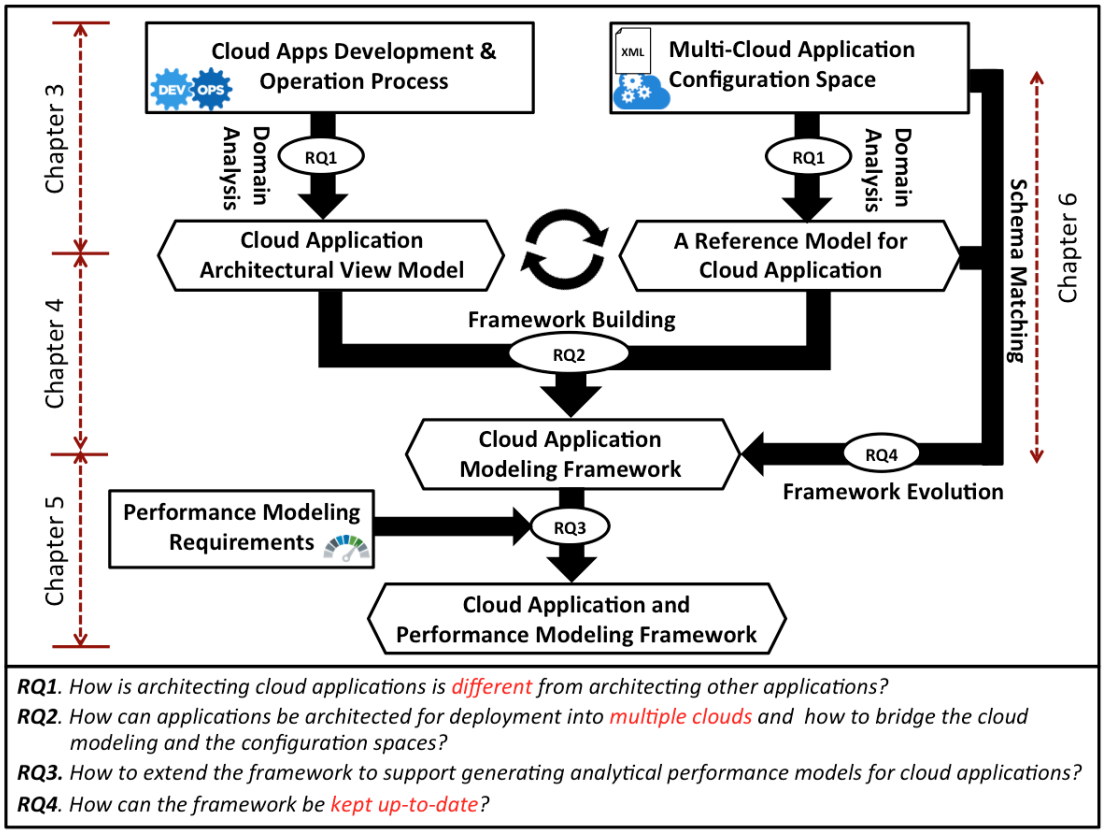


Figure 1.1: Research Approach

1.3.1 RQ1. How is architecting cloud applications different from architecting other applications?

Objective: The objective of the first question is to devise a software architecture framework for cloud applications. The framework will capture the common practices of the different stakeholders of the cloud application in order to enable them to create, interpret, analyze and use architecture descriptions to specify cloud applications deployment and management aspects.

Methodology: To understand the domain and the architectural requirements of cloud applications, we started by a study for two cloud computing reference architectures [67]; namely, the NIST Cloud Computing Reference Architecture [115] and the IBM Cloud Computing Reference Architecture [12]. Consequently, a list of cloud services and stakeholder's

roles have been identified. We considered roles that are directly related to the cloud application development and operation process and studied the different financial, design and operational decisions required to build a cloud application, deploy it, and manage its operations while running in the cloud. The initial observations were that: (i) a cloud application needs to evolve dynamically at runtime to meet performance, availability, and scalability targets, and (ii) the quality of a cloud application depends on its configuration and the architecture of its service model. These are fundamental differences from what existing software architecture frameworks address, where the focus is on the application implementation. In cloud computing, architecture evolves during deployment; therefore, runtime operation needs as much architectural modeling as functional design does. This shift in emphasis from architecting for implementation to architecting for service operation motivated the need for a view model that is constructed around the application service and deployment model. Chapter 3 provides the details of this model.

1.3.2 RQ2. How should an application be architected to be deployed into multiple clouds and how should cloud modeling and configuration spaces be bridged?

Objectives: Deploying an application on a cloud platform requires specifying how the application service model will use the platform resources of that particular provider. Our objectives are the following:

- facilitate the migration of cloud applications between different cloud providers, first, by providing an abstraction layer (reference model) for cloud applications to enable modeling the service structure and configuration independently from the target cloud platform specifications, second, by mapping the platform independent models into the target platform specifications as needed.
- maintain the architectural models in synchrony with the configuration space as applications evolve, through developing a modeling framework that supports transforming the high-level architectural models into platform specific artifacts.

Methodology: To create a reference model for cloud applications, we started by domain analysis for the different artifacts required to successfully deploy and manage an application on three cloud platform providers (i.e., Amazon AWS, Windows Azure and Google App Engine). Particularly, we analyzed the schemas provided by these platforms. These schemas specify the syntax and structure of the information that is required by the

platform. Our methodology includes both a manual and semi-automated (cf. Chapter 6) schema matching process to identify the similar domain concepts and then creates meta-concepts that describe them independently from the target platform.

To build the modeling framework, we created a meta-model for each of the (5+1) view models. Then integrated the meta-models into one meta-model, by extending the top element of each meta-model from a common meta-meta component, and defining associations between the elements of the different meta-models. The outcome meta-model is then realized as a layered Domain Specific Modeling Language (DSML), where we utilized the hierarchical structure of the meta-model to divide the model into layers to facilitate incremental and collaborative modeling. To facilitate model transformation, a model generator has been implemented. We used template-based transformation as it provides the required flexibility and portability, since the only thing that needs to be changed to support a new platform is the template. The details of the meta-models and framework are provided in Chapter 4.

1.3.3 RQ3. How should the framework be extended to support generating analytical performance models for cloud applications?

Objective: The objective here is to allow the developers, system architects and performance engineers to analyze the performance of a cloud system under several configuration and platforms by reusing the existing cloud configuration space artifacts. This should help them make better decisions regarding the system architecture, design, configuration and the target platform.

Methodology:

To extend the cloud application framework to support generating analytical performance models for cloud applications, we started by analyzing the requirements of performance modeling to assess how much information can be reused from the cloud service and deployment modeling artifacts. The initial observation is that there is a significant overlap between the information that is required to analytically analyze the system performance and the information that is required to deploy and run an application on a particular platform. Accordingly, we extend the framework to capture the application runtime parameters. Then we defined a set of transformation rules to automatically transform the StratusML models into Layered Queuing Network (LQN) models. Chapter 5 provides the details of how to generate LQN models from the StrausML models.

1.3.4 RQ4. How should the framework be kept up-to-date?

Objective: The objective of this question is to support the framework evolution as new cloud platforms evolve. In MDE, the process of domain model creation, evolution, and transformation depends on the ability to find correspondences (mappings) between the concepts of the source and the target models. Consequently, this phase will focus on devising a semi-automated approach to facilitate matching domain concepts of different providers' schemas to facilitate creating new concept abstractions and transformation rules.

Methodology: To automate the generation of alignments between providers' schemas, we used schema matching to identify the possible matches between different cloud providers' schemas. We noticed that the shortcomings of traditional schema matching techniques are that they rely on linguistic and structural similarities to identify the possible matches. In the cloud, schema terms diverge so much that such matching is impossible. To address this challenge we incorporated domain knowledge in the schema matching process. The details of the proposed schema matching techniques and the results of applying them on cloud schemas are provided in Chapter 6.

1.4 Research Contributions

This research aims at providing a comprehensive framework that covers the entire Cloud Applications' life-cycle. The contributions of this research:

- an architectural view model for cloud applications that focuses on service operation rather than implementation,
- a modeling framework for cloud applications that
  - fragments models into artifacts that are easy to modify,
  - allows weaving fragments into model views,
  - uses layers and multiple views to facilitate collaboration,
  - provides an intuitive way to specify the dynamic behavior of cloud applications, and
  - uses template-based transformation to enable a model-once deploy-everywhere approach,

- a pivot performance model for cloud applications and an analytical performance model generator that
  - addresses the dynamic nature of cloud applications, where both the platform and the application are fluid and continuously changing after deployment,
  - captures the cloud application runtime performance parameters, and
  - defines transformation rules to automatically transform the cloud models into Layered Queuing Network models, and
- a schema matching approach for the cloud domain that
  - addresses the vendor lock-in problem, and
  - reduces the manual efforts needed to create a meta-model and keep it up-to-date as schemas evolve, and generate transformation templates.

1.5 Document Organization

The rest of this thesis is organized as follows. Chapter 2 briefly reviews the background concepts and some related work. Chapter 3 characterizes cloud native applications and proposes an architectural framework for cloud applications. Chapter 4 shows how the architectural framework has been realized as a cloud modeling framework (StratusML) that consists of a modeling language that uses layers to specify the cloud configuration space, and a transformation engine to generate the configuration space artifacts. Chapter 5 extends the framework to support capturing the cloud application performance. Chapter 6 shows how to automate the framework evolution as new concerns and cloud platforms emerge. In which, a new approach has been devised for schema matching and then evaluated using public schemas of two major cloud providers (i.e., Windows Azure and Google App Engine) and private schemas of two reference cloud models (StratusML and TOSCA). Chapter 7 summarizes the thesis, draws several conclusions, and suggests ideas for potential future work.

Chapter 2

Supporting Concepts and Related Work

"Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning."
– Albert Einstein, 1955

Cloud computing is a big disruption to how information technology (IT) is consumed and managed. The concepts and technologies behind cloud computing are not new. It is believed that the concept of cloud computing is the same as what John McCarthy, in the 1960's, referred to as the ability to provide and organize computation as a "utility". The main characteristics of cloud computing were also discussed by Douglas Parkhill in "The Challenge of the Computer Utility" in 1966 [124]. On the other hand, the term *cloud* and its graphical symbol have been used for decades in computer network literature; first to refer to the large Asynchronous Transfer Mode (ATM) networks in the 1990s, and then to describe the internet (i.e., a large number of distributed computers).

Even though the concept is not new, the past few years have witnessed several attempts to define "Cloud Computing". Vaquero et al. compared 22 different definitions in an attempt to provide a unified definition [156]. Some of these definitions are general: "applications delivered as services over the internet and the hardware and systems software in the datacenters that provide those services" [50]. Others are more specific describing the cloud as "a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established

through negotiation between the service provider and consumers". The fact that cloud computing is not a purely technical term, as well as the large number of interdisciplinary technologies that participate in characterizing cloud computing, are the reasons behind all the different definitions.

The information technology industry has sought to standardize the definition of cloud computing. One of the first standardized definitions is the one by Forrester Research. In which, they defined cloud computing as "a standardized IT capability (services, software, or infrastructure) delivered via internet technologies in a pay-per-use, self-service way" [148]. Forrester's definition focuses on the service and business models of the cloud; however, it ignores the deployment models. The most recent and accepted standardized definition of cloud computing is the one by the National Institute of Standards and Technology (NIST) [102]:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models."

The NIST definition is relatively technical and covers all of the cloud service (IaaS, PaaS, SaaS) and deployment (Public, Private, Hybrid, Community) models. The NIST definition is concise and accurate. It distinguishes between the main and derived characteristics (rapid elasticity vs. massive scalability) of cloud computing. Also, it differentiates between the cloud characteristics and its enabling technologies (i.e. virtualization and autonomic computing). Nevertheless, the definition ignores the "pay-per-use" business model of cloud computing. However, the "pay-per-use" model was not omitted from the definition out of ignorance, but because NIST tried to cover all of the cloud deployment models, including the "private cloud" deployment model, which does not necessarily involve the "pay-per-use" practice. Instead of explaining the business model, NIST identified the main technological characteristics that can result in cost reduction and add constraints (e.g., all services must be measurable) that provide all the requirements for cloud deployment models to adopt any billing or utility model (i.e. "pay-per-use").

The rest of this chapter will be organized as follows: Section 2.1 explains the cloud computing principles and requirements and clarifies the relationship between cloud computing and the service-oriented architecture, grid computing, parallel computing, utility computing, autonomic computing, and virtualization. In Section 2.2 we present the most related research projects to the one presented in this thesis.

2.1 Cloud Computing Principles and Requirements

As shown in Figure 2.1, the NIST definition of cloud computing reveals the main characteristics, delivery models, and service models of cloud computing.

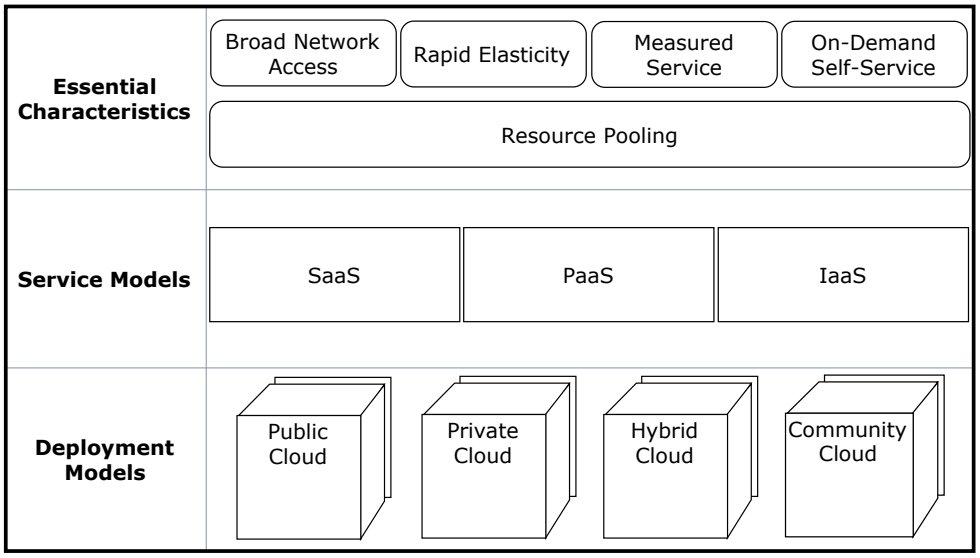


Figure 2.1: NIST Visual Model of Cloud Computing Definition.

This section describes the foundation of cloud computing by listing and explaining these characteristics and models in more detail.

2.1.1 Cloud Computing Characteristics

The following are the five main characteristics of cloud computing that most people agree upon:

(a) **On-demand self-service**. Cloud services are on-demand; that is, service consumers can automatically request the service based on their needs, without human interaction with the service provider.

(b) **Easy to access standardized mechanisms**. NIST refers to this characteristic as broad network access; however, the term "global reach capability" is also used. The idea is that it should be possible to access cloud services through the network using standardized interfaces and access mechanisms. Having global reach capability does

not mean that these services must always be accessible from the internet, because this depends on the deployment model used. However, it should be possible to reach the service globally, when policies allow this.

(c) **Resource pooling and multi-tenancy**. In cloud computing, resources (i.e. storages, processors, memory, network bandwidth, and virtual machines) are shared between multiple tenants, and assigned exclusively at run-time to one consumer at a time. Assigning resources is done dynamically based on the consumers' needs. Sharing resources can help increase utilization, and hence significantly reduce the operation cost. Scheduling algorithms can be used to dynamically assign resources to different tenants based on the type of workload, fairness, locality, and many other factors. [155, 163].

(d) **Rapid elasticity**. Elasticity is the ability to scale in and out by provisioning resources and releasing them, respectively. Cloud computing should provide mechanisms to allow quick and automatic elasticity. The large pool of resources in cloud datacenters gives the illusion of infinite resources to the consumers, and elasticity provides the flexibility to provision these recourses on-demand.

(e) **Measured service**. Providing cloud metrology or mechanisms for measuring service usage as well as to monitor the health of services is crucial in cloud computing. Measuring services enables optimizing resources and provides transparency for both consumers and providers, allowing them to better utilize the service. Measured services can help in building closed-loop cloud systems that are fully automated.

(f) **Auditability and certifiability**. Regulatory compliance requires enforcing rules and regulations. Services should provide logs and trails that allow the traceability of policies, so as to ensure that regulations are correctly enforced.

The list above (except point (f)) is based on the NIST definition. These characteristics are the most adopted in the cloud industry. Some other definitions may have slight variations. For example, we added "auditability and certifiability" to the list above based on the regulatory compliance requirements. Some more specific lists of characteristics can be made for each layer and each type of service provided in the cloud environment. For example, at the application level, a possible cloud service characteristic is that a service must be portable, pre-configured, or adaptable. Chapter 3 provides an example of a list of more specific characteristics and requirements for cloud applications.

2.1.2 Cloud Computing Deployment Models

A cloud computing deployment model is a model that describes the environment where cloud applications and services can be installed, in order to be available to consumers. By the deployment environment, we mean the physical location, the infrastructure facilities, the platform constraints, as well as anything that can affect the access mechanisms of the deployed applications. There are four main cloud computing deployment models: public, private, hybrid, and community cloud.

(a) **Public cloud:** A public cloud or external cloud is an open model, in which the resources and services (e.g., applications, storages) are provided by a third party (the cloud providers). The infrastructure or platform services are provided to the public based on the service-level agreement between the service provider and the consumer. This type of resource sharing between multiple organizations or consumers is referred to as the multi-tenancy model. Public cloud is the least expensive choice for application hosting. However, the lack of a trust model between the cloud providers and consumers is the main obstacle for this model.

(b) **Private cloud**: A private cloud or internal cloud is a model, in which the infrastructure and platform services are hosted and operated entirely by the application provider on premises. This eliminates the need for a trust model and provides more flexibility. Organizations can implement their own policies with regards to privacy, security, and access mechanisms. However, this option is expensive in terms of resources, and the manpower needed to manage the resources.

(c) **Hybrid cloud**: A hybrid cloud model is a combination of the public and private cloud models. It provides a similar trust model for in-house services as in private clouds, yet it is less expensive than a private cloud as it uses public cloud platforms and infrastructure for part of its services. However, having both public and private clouds working together requires interoperability and portability of both applications and data to allow communication between the models.

(d) **Community (cooperative) cloud**: A community cloud is similar to extranets, but with virtualization and on-demand capabilities. In a community cloud, a number of organizations, which usually share some common goals or belong to a specific community, build a shared cloud datacenter that can be used by all of the members. The goals are to alleviate deficiencies in the individual IT infrastructures, reduce the cost of administration, and lower the cost per unit [72]. The community can be created between a professional community (i.e. organizations with business relationship), a

Attribute	Public Cloud	Private Cloud	Hybrid Cloud	Community Cloud
Cost of building the datacenter on service consumer	No initial cost	High initial cost	Medium initial cost	Varies , depends on the number of cooperatives
Operation and maintenance cost on the provider	Lowest cost with respect to the datacenter size	Highest cost with respect to the datacenter size	Weighted average, depending on the percentage of public and private parts.	Similar to private clouds, but the cost divided on the participants
Size of the datacenter	~50,000 Server	~5,000 Server	Less than private Cloud	~15000 More than private cloud but much less than public cloud
Infrastructure Controllability and Flexibility	Limited configuration controllability	Full controllability (HW + SW)	Full controllability over the private part and limited for the public part	High controllability but limited by the community policies
Level of trust	Lowest trust,	Highest	Medium trust	High trust
Infrastructure Location	Off-Premise	On-Premise	Both On- and Off-Premise	Within the Cooperative Facility
Owner of the infrastructure	The IaaS Vendor	The Customer	The IaaS vendor owns the public part, and the consumer owns the in-house part	Shared between the Cooperatives

Table 2.1: A Comparison Between the Different Cloud Deployment Models

geographic community, or some other well-defined community group. A community cloud relies on the trust relation between all the members, which is driven by their mutual benefits [24, 98]. As a result, this model is more trusted than the public cloud, and less expensive on participating members than having a private cloud. This model also provides more controllability over the shared infrastructure resources. However, a community cloud still needs to enforce strong security and privacy policies. Furthermore, regulatory compliance is a main obstacle facing community cloud adoption.

Table 2.1 shows a comparison between the different cloud deployment models, based on the initial cost of building the cloud datacenter or the capital expenses (CapEx) on the consumer, the operating expenses (OpEx) and maintenance cost of the datacenter, the

size of the datacenter, controllability and flexibility, the level of trust, the location of the infrastructure, and who owns the infrastructure.

As shown in Table 2.1 there is no initial cost associated with adopting public cloud by consumers [139]. Consumers need not worry about creating the cloud infrastructure. Instead, they can request the services and resources on-demand and pay just for what they use. Conversely, a private cloud requires a big initial capital investment in order to build the private datacenter[139]. Unlike the private model, the hybrid model builds a relatively small private datacenter for sensitive and important tasks and information, and uses the public cloud for other jobs. For this reason, the cost of adopting the hybrid cloud model is between that of public and private clouds. Finally, the community cloud model shares the cost of building the required datacenter with the cooperatives. For this reason, the initial cost can vary; the larger the community the smaller the share and the lower the cost.

Table 2.1 also shows that the operating cost (i.e. power consumption, manpower expenses, rent, maintenance, upgrades, etc.) of public cloud is lower than the other models. This is due to the economies of scale, as well as the high level of automation and optimization the public cloud. Production costs drop significantly as the number of units produced increase [1] [82]. This allows public cloud providers to enjoy favorable prices for IT equipment and needed resources, since they purchase them in bulk. According to Jon Moore's blog, a private datacenter should have on average 10,000 servers to get an economically feasible marginal cost that is comparable to what current public cloud providers charge [111]. On the other hand, public providers tend to invest more in automation and optimization, which results in fewer administrative staff. For example, while the ratio between IT staff to servers is (1:100) [60] in traditional datacenters, this ratio goes to (1:1000) [60] and even (1:5000) in public cloud datacenters.

It is clear from Table 2.1 that consumers can have full control over a private cloud infrastructure, whereas in a public cloud, controllability is limited to tuning some configuration parameters. On the other hand, while community cloud consumers can have access and control over the infrastructure, this controllability is bounded by the community policies and agreements.

The level of controllability and flexibility can also affect the level of trust. This explains why consumers trust the private cloud model more than the other models. However, it is important to note that the level of trust is not related to the actual security level. Public cloud providers tend to implement best practices and try to ensure security at every level of the security stack. However, the cloud computing paradigm introduces new security

[1] This phenomenon is referred to as the experience curve effect and was first noticed by Bruce Henderson in 1960 at BCG (Boston Consulting group).

threats that did not exist in traditional datacenters, such as threats related to sharing resources through virtualization [32]. Most of these threats are equally applicable to both public and private models. Some of the cloud security myths assume that all clouds are created equally [44], while others assume that public and private cloud providers have the same experience and capabilities to implement security measures [107] for data protection, identity management, compliance, access control rules, and other security capabilities. If these assumptions are true, then a public cloud is the least secure, while a private cloud is the most secure.

Cloud deployment models differ based on the infrastructure's owner, location, or operators and their policies. One model does not fit all business types. The selection of a cloud deployment model depends on the consumers' need and budget, and on whether they favor price reduction and control delegation over flexibility, control, and customization.

2.1.3 Cloud Computing Service Models

This subsection presents and compares the cloud computing service models. Cloud service models are sometimes referred to as the cloud service hierarchical view [154], cloud service offerings, cloud service delivery models [93], or the cloud service layered architecture, in analogy to the network layered architecture. Cloud service models try to classify "*anything*" providers offer *as a service* (XaaS), where X means is an arbitrary service (e.g., infrastructure, software, storage). A cloud service model represents a layered high-level abstraction of the main classes of services provided by the cloud computing model, and how these layers are connected to each other. This separation between layers allows each cloud provider to focus on the core services they provide, while at the same time being able to reuse the services from the lower layers by following the set of standard communication mechanisms and protocols between the different layers. Layers differ based on the management scope covered by the provider [12], which means that a user in the upper layers cannot bypass the interfaces provided by the layer beneath, so as to directly access the resources. This separation does not only help in service integration but also allows having a fully distributed, scalable, and fault-tolerant architecture. By having different layers with different abstraction levels, cloud providers can have better manageability over the resources, as well as higher controllability and security.

As in the network layered architectures (i.e. OSI, TCP/IP), there are different cloud service models. These models vary based on the time they were proposed, relative to the maturity of the cloud computing paradigm at that time; and on the level of detail in these models, as represented in the number of model layers. However, the differences between service models do not contradict each other. Instead, these models are complementary [2].

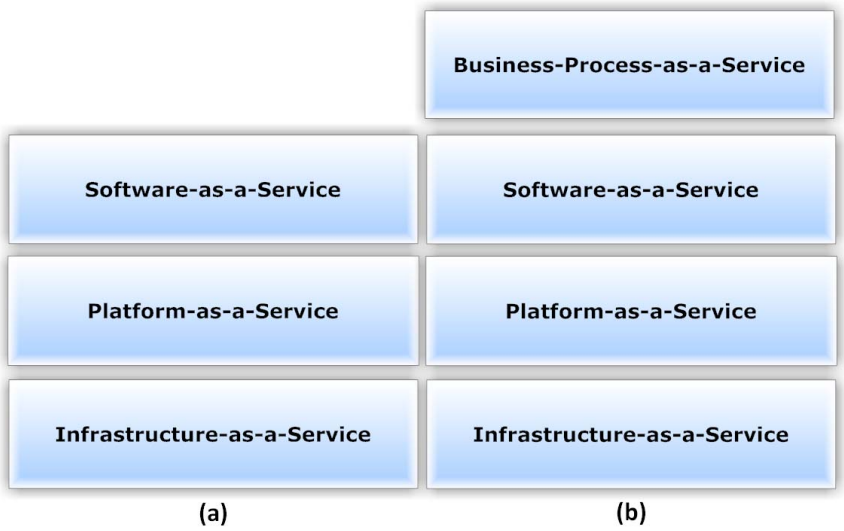


Figure 2.2: Cloud Service Models (a) SPI vs. (b) IBM.

This subsection will briefly discuss the main two service models: The NIST SPI model [102] (aka, the three service-layers model), and the IBM service model [12] (aka, the four service layers model). Figure. 2.2 shows a comparison between these two models.

The NIST SPI model:

The SPI model classifies the services provided by the cloud providers into three main categories (layers): **S**oftware services, **P**latform services, and **I**nfrastructure services. The SPI model is named after these categories, which are described below:

- Software as a Service (SaaS): A service is classified as a software service if it allows the consumer (end user) to access and use a provider software application that is owned (hosted), deployed, and managed by the provider. Consumers normally have limited control over the application, and are restricted in how they can use and interact with the application. The application is usually accessed via a thin client (i.e. web browser), through which consumers can input data and get output [102]. Because most SaaS services are specific applications rather than being generic software services, SaaS is sometimes referred to as Application-as-a-Service. Examples of SaaS are content services such as video-on-demand (i.e Netflix), email services (i.e. gmail), and business applications such as customer relationship management applications (i.e. Salesforce).

- Platform as a Service (PaaS): A service is classified as a platform service if it allows the service consumer (usually a SaaS provider, cloud developer, or administrator) to define, develop, configure, deploy, manage, and monitor cloud applications. While PaaS allows consumers to deploy and control applications and their hosting environment configurations, consumers do not have direct control over the underlying cloud infrastructure [102]. PaaS services abstract the communications with the lower-level infrastructure by providing easy to access and easy to use interfaces. Operating systems and application frameworks are part of the PaaS layer.

- Infrastructure as a Service (IaaS): A service is classified as an infrastructure service if it allows the service consumer (usually PaaS providers) to lease infrastructure capabilities based on demand. The infrastructure capabilities include processing, storage, network or any other basic computing resources that can be used to deploy and run platforms (i.e. operating systems, management tools, development tools, and monitoring tools) and the applications developed on top of the platforms. Again, consumers are not given direct access to resources but have the ability to select and configure resources as required based on their needs [102]. IaaS is sometimes referred to as the virtualization layer, since it utilizes virtualization technologies to partition physical resources, so as to provide the consumers with a pool of storage and computing resources.

IBM service model:

According to the IBM Cloud Computing Reference Architecture [12], a cloud service model consists of four service layers; from top to bottom, these are the Business-Process-as-a-Service, Software-as-a-Service, Platform-as-a-Service, and Infrastructure-as-a-Service layers.

The last three layers are exactly the same as in the SPI model. In fact, IBM used the same definition of SaaS, PaaS and IaaS as defined according to the NIST SPI model. The only difference between the IBM service model and the NIST SPI is in the Business-Process-as-a-Service (BPaaS) layer. Since this is the only difference, we will only explain BPaaS and why it was introduced.

- Business-Process-as-a-Service: A service is classified as a business process service if it allows the consumer (end user, business process manager, or designer) to design, manage and integrate a set of transactional and collaborative activities based on the SaaS provided in the layer beneath, so as to accomplish a specific business organizational goal. Accordingly, IBM classifies any business process service - whether it

focuses on technology and reuse (horizontal), or it is domain specific (vertical) - as BPaaS if and only if the service (i) represents a business process that is delivered through the cloud computing model based on its main characteristics, as defined in the NIST definition (i.e. multi-tenant service, self-service, elastic, metered, and priced); (ii) accessed through a web-centric interface; (iii) and utilizes a web-oriented cloud architecture. Similar to IaaS and PaaS services, the service provider of a BPaaS provides the tools to access and utilize the resources in the BPaaS layer. Consumers do not need to access services in the underlying layers. "a BPaaS provider is responsible for the related business function(s)[12]". Some examples of BPaaS include a process for employee benefits management; and IT-centric processes, such as a process for software testing where the whole process, including the testing staff, is provided as a cloud service.

As mentioned earlier in this subsection, currently there are many cloud service models in the market. This is because different cloud providers use different service models to reflect the types of services they provide. However, the differences between cloud service models are minute. In addition, these models complement each other. The most dominant service models currently in use are the ones discussed in this subsection. The next subsection discusses the relationship between cloud computing and other computing paradigms.

2.1.4 The Relationship Between Cloud Computing and Other Computing Paradigms

Cloud computing is the result of evolution and adoption of existing technologies and paradigms. The goal of cloud computing is to allow users to take benefit from all of these technologies, without the need for deep knowledge about or expertise with each one of them. The cloud aims to cut costs, and help the users focus on their core business instead of being impeded by IT obstacles. Figure. 2.3 summarizes how cloud computing is related to the other technologies that it has emerged from.

The figure shows that the main enabling technologies for cloud computing are virtualization and autonomic computing. Virtualization abstracts the physical infrastructure, which is the most rigged component, and makes it available as a soft component that is easy to use and manage. By doing so, virtualization provides the agility required to speed up IT operations, and reduces cost by increasing infrastructure utilization. On the other hand, Autonomic Computing automates the process through which the user can provision resources on-demand. By minimizing user involvement, automation speeds up the process and reduces the possibility of human errors.

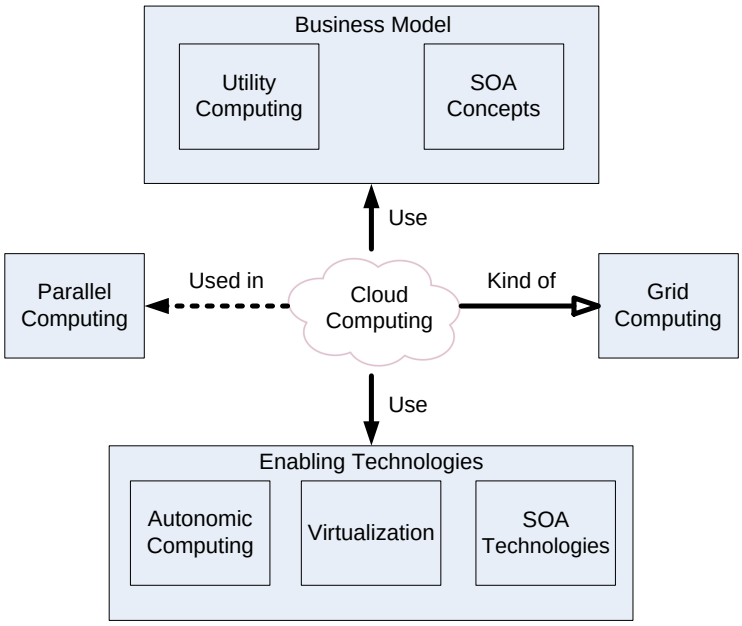


Figure 2.3: Cloud Computing Relationship with Other Computing Paradigms.

Users face difficult business problems every day. Cloud computing adopts concepts from SOA that can help the user break these problems into services that can be integrated to provide a solution. Cloud computing provides all of its resources as services, and makes use of the well established standards and best practices gained in the domain of SOA to allow global and easy access to cloud services in a standardized way. Cloud computing also utilizes concepts from utility computing in order to provide metrics for the used services, based on the benefits gained. These metrics are at the core of the pay-per-use model in public clouds. Having measured services is also an essential part of completing the feedback loop in autonomic computing, which is required for fully automating services so that they can scale on-demand and automatically recover from failures.

Cloud computing is a kind of Grid Computing; it has evolved from Grid by addressing the QoS and reliability problems. Cloud computing provides the tools and technologies to build data/compute intensive parallel applications with much affordable prices compared to traditional parallel computing techniques.

More details of the relationships between cloud computing and other computing paradigms can be found in our survey paper [67].

2.2 Related Research Projects

Our work has taken shape in the context of a rich literature focused on viewpoint architecture frameworks, model-driven engineering and quality prediction, cloud domain specific modeling languages, and lastly schema matching for cloud vendor schemas. Here we relate our work to the most related projects in each of these fields.

2.2.1 Viewpoint Architectural Frameworks

A viewpoint is a set of representations (views and models) of an architecture that covers a stakeholder's issues. In Chapter 3, the (5+1) view model, which is an architectural view model for cloud applications is presented. The model uses different views to describe different perspectives of the cloud application deployment. Using multiple views to explore systems from different angles is a common practice in software engineering. For example, Krutchen [91] famously proposed four views: logical, development, process and physical; plus "use cases" as the fifth view, to completely describe the lifecycle of a software system. Zou and Pavlovski [165] extended the (4+1) view model by adding a *control case view* to address non-functional system requirements.

The (4+1) view model [91] is an example of many other frameworks used in software development. Some of these examples include: the Department of Defense Architecture Framework (DoDAF) [120], the Three-Schema Approach, [84], Zachman Framework [162], the Reference Model of Open Distributed Processing (RM-ODP) [94], and the Open Group Architecture Framework (TOGAF) [157]). These frameworks are not cloud specific; they do not target the cloud applications' development process, or capture the domain specific constraints related to the cloud platforms and providers. The aforementioned frameworks are generic enough to describe large class of software systems. However, they only consider the dynamism and variability of the design model and assume a static deployment infrastructure. Even when deployment models are considered, they are considered as part of the lifecycle and not a driver of architectural evolution.

The (5+1) model was inspired by the (4+1) model *process view* [91]. In fact, if tasks are considered processes, the (5+1) model can be seen as an extension of the *process view*, distinguished by its focus on the cloud application ecosystem (represented by variable deployment models, dynamic infrastructure, and standardized service offerings). The (5+1) model augments its core with elements that enable quality prediction. Its views are limited but expressive enough to cover operational requirements and the application lifecycles in the cloud.

2.2.2 Modeling Frameworks for Cloud Applications

An architecture framework is an encapsulation of a minimum set of practices and requirements for artifacts that describe a system's architecture. Models are representations of how components in a system fit structurally in and behave as part of the system, where views are a partial expression of the system from a particular perspective. This section focuses on the state of the art *modeling frameworks* for cloud applications. We focus on the frameworks that utilize model driven engineering to reduce the complexity of managing the configurations of the cloud applications and their deployment into multi-cloud platforms.

Recently, there have been several projects and proposals that exploit model driven engineering to address the cloud applications' modeling and management concerns. A sheer volume of these projects addresses a single concern at a time; such as portability [62, 125, 138], or security [4]. This thesis devises a multi-cloud and multi-concern framework that addresses the provisioning, deployment, adaptation, orchestration and performance of the cloud services into multiple cloud platforms. The following is a list of the cloud modeling frameworks that are the most relevant to our project (i.e., the StratusML framework). For each of these projects, we provide a brief description.

SINTEF-REMICS [22] REMICS stands for Reuse and Migration of legacy applications to Interoperable Cloud Services. The project was initiated in September 2010 by the Foundation for Scientific and Industrial Research in Norway (SINTEF). The objective of the project is to develop tools based on model-driven engineering to support the migration of legacy applications to the cloud service model. To support the migration process, REMICS offers a set of modelling concepts at the cloud infrastructure as a service (IaaS) level and a domain specific language called PIM4Cloud. PIM4Cloud enables designers to model applications that can be deployed on the cloud. PIM4Cloud is cloud provider independent. It is implemented using Scala as a hosting language. Different from the StratusML language, PIM4Cloud focuses only on modeling the cloud infrastructure concepts.

SINTEF-CloudML [22, 57] CloudML is another project by the Foundation for Scientific and Industrial Research in Norway (SINTEF). The project started in 2012. CloudML aims to model service profiles and developers requirements and to represent the physical and virtual resources in D-Clouds as well as their state. SINTEF defined D-Clouds as small scale distributed datacentres where resources are shared across geographic boundaries. CloudML is an XML-based language, and it is based on three requirements: (i) representation of physical and virtual resources as well as

their state; (ii) representation of services provided; and (iii) representation of developers requirements. While both StartusML and CloudML are multi-cloud, CloudML does not support visual modeling (i.e., CloudML is an XML-based language) and multi-layers. Moreover, CloudML is limited to represent the concerns related computational and network resources, provider services, and developers requirements. CloudML has been extended and adopted by several other European projects including MODAClouds [6], ARTIST[153], and PaaSage [85].

mOSAIC [113, 125, 126] The mOSAIC project is another multi-national European project that started in 2010. mOSAIC aims to address the challenges of application portability and interoperability across multiple clouds. It provides an agnostic API at IaaS level and an open source platform, with adapters to most notable cloud providers APIs, in order to enable developing cloud applications with abstraction of IaaS services and provisioning and deploying the applications on multiple clouds. Different from StratusML, which uses model driven technologies, mOSAIC uses semantic technologies. mOSAIC creates a cloud ontology and semantic engine for service brokerage to help users select the API components and functionalities needed for building new Cloud applications as well as in identifying the proper cloud resources to be consumed.

ARTIST [153] ARTIST stands for Advanced software-based seRvice provisioning and migraTIon of legacy SofTware. It is a European integrated project that started in 2012. ARTIST aims at migrating existing software applications to the cloud as software as a services. It covers both business and technical aspects of the software and uses a model-based modernization approach based on UML profiles to provide a vendor- and platform-independent methodology to enable automating the reverse engineering and forward engineering phases to support the migration, maintenance and evolution of cloud-based applications. ARTIST focuses on the on one concern that is migrating already existing (legacy) applications to the cloud. The objectives of ARTIST are different from those of StratusML. StratusML focuses on enabling the cloud stakeholders to build cloud applications, and addresses multi-concerns (e.g., Portability, Performance, Availability, Adaptation and Cost).

MODAClouds [6] MODAClouds is a European project that started in September 2012. It focuses on challenges that face the application developers and operators, when adopting a cloud solution. The objectives of the MODACLouds are to avoid the vendor lock-in by enabling system developers to design clouds agnostic software applications, and facilitate the management and deployment of these applications across multiple clouds. MODAClouds offers: (i) a Decision Support System (DSS)

to guide the selection of Clouds based on cost and requirements, (ii) an Integrated Development Environment (IDE) to support high-level design, early prototyping, and semi-automatic code generation, (iii) a run-time environment to support monitoring and self-adaptation. To model multicloud application, MODAClouds uses three different levels of abstraction: a Computation Independent Model (CIM) in which non-functional requirements are modeled, a Cloud-Provider Independent Model (CPIM) where cloud concepts are introduced into the model but kept away from any specific platform, and a Cloud-Provider Specific Model (CPSM) in which the artifacts required by each specific platform are introduced into the models.

ModaClouds is the most related project to StratusML. The objectives of both projects are the same. However, ModaClouds and StratusML are different in terms of implementation technologies and the set of features they provide. StratusML uses template-based transformation and layered view model. On the other hand, MODAClouds provides monitoring capabilities. Both StratusML and MODAClouds support performance modeling and analysis. StratusML transforms the StratusML models in Layered Queueing Network (LQN) models, while MODAClouds allows users to describe the architecture of their application by means of Palladio Component Models (PCMs).

PaaSage CloudMF and CAMEL PaaSage is nation-wide European project that started in October 2012. PaaSage aims at leveraging model-driven engineering techniques and methods to facilitate the specification and execution of cloud-based applications. PaaSage adopts the Cloud Application Modelling and Execution Language (CAMEL) that integrates and extends five domain specific languages [14]; namely, CloudML [57], Saloon [135], CERIF [80] and Scalability Rule Language (SRL) [90]. PaaSage realizes CAMEL as a modeling framework called the Cloud Modelling Framework (CloudMF) [48]. CloudMF has been implemented as an extension to the Eclipse Modelling Framework (EMF). CloudMF enables engineers to specify multiple aspects of multi-cloud applications. CloudMF consists of two components: (i) CAMEL, the modelling environment (DSL); and (ii) Models@run-time, which provides an abstract representation of the running system. CloudMF and the PaaSage project are still in the development phase. The project is expected to be completed by September 2016.

Caglar DSML In 2013 [27], Caglar et al. proposed a model driven engineering framework that consist of a domain specific modeling language and generative technologies to address three cloud challenges; namely, (i) programming and deployment heterogeneity, (ii) resource management, and (iii) performance and cost estimation. Caglars

framework targets the cloud customers to enable various cloud service providers agnostic "what-if analyses while automating the deployment and resource management. The main feature that distinguishes StratusML from Caglars framework it its ability to model the self-adaptation behavior.

TOSCA and Winery The Topology and Orchestration Specification for Cloud Applications (TOSCA) [17] is a standard for cloud computing developed by the Organization for the Advancement of Structured Information Standards (OASIS). The first draft of the standard has been approved in January 2014. TOSCA provides an XML based language for specifying the cloud applications components and structure described by a topology (i.e., service templates) along with the processes that describe the applications management plans (e. g., provisioning or migration). Kopp et al. [87] developed the a graph-based modeling language called Winery that supports modeling of TOSCA-based applications. Winery is HTML5-based environment that offers support for the complete TOSCA standard: Most importantly, types can be defined in the Element Manager and composed in the Topology Modeler. Similar to StratusMLs core service model, TOSCA aims at enabling specifying the cloud application deployment and management independently from the target platform. However, the TOSCA currently lacks the components required to specify other concerns such as adaptation and performance as in StratusML.

Blueprint The Blueprint Specification Language (BSL) [117] is a uniform specification language, for specifying cloud services across several cloud vendors. Nguyen et al. developed BSL in 2012 at Tilburg University. The BSL is divided into modular modules to cover all aspects of a cloud service specification. Depending on the cloud layer (i.e., SaaS PaaS, IaaS), a BSL user can select modules to specify his cloud service. Different from StratusML, BSL focuses only on portability aspects. It uses XML-based syntax that is supported by OWL schema model to formalize the semantics.

MULTICLAPP The MULTICLoud migratable and interoperable APPlications (MULTICLAPP) [63] is a framework that aims to enable the cloud developers to model cloud applications and support their adaptation and migration without the need for the developers to be familiar with the specification of any cloud platform. Different from the StratusML approach, where we built a domain specific modeling language from scratch, the MULTICLoud approach is based on creating a UML profile for modeling multicloud applications. The UML profile provides a set of components and stereotypes to model and annotate cloud application with the deployments and run-time parameters.

CloudMIG CloudMIG [53] is a framework to facilitate the migration of legacy software systems to the Cloud. CloudMIG offers a Cloud Environment Model (CEM) and a dedicated tool support (CloudMIG Xpress). The CEM model is realized as an ecore metamodel. The metamodel provides the components required to specify both the cloud application and cloud environment. Particularly, the CEM model provides components to specify the environment; mostly at the infrastructure level, and partially at the platform level. Moreover, CloudMIG enables specifying the application constraints, pricing, and deployments parameters. CloudMIG Xpress is a prototype implementation of the CloudMIG approach. CloudMIG Xpress enables the automatic computation of optimal cloud-based deployments and conformance checking of legacy software with respect to potential cloud providers. Both StratusML and CloudMIG are based on model driven engineering. While our approach starts from current Cloud platforms to extract common vocabulary and elements to create a Cloud meta-model, CloudMIG starts from existing legacy systems, extracts the actual architecture, and then uses the selected target cloud platform meta-model along with a utilization model to generate a target model toward system migration.

SOCCA In 2010, Tsai et al. [154] proposed the "Service-Oriented Cloud Computing Architecture" to enable Cloud Applications to work with each other. The proposed architecture is a three layered architecture that consists of a *Cloud ontology mapping layer*, a *Cloud broker layer*, and a *SOA layer*. The work we are presenting in this research falls under the ontology mapping layer where we define a reference model, providing the main vocabulary of Cloud Applications and the relations between them. The reference model takes into consideration the multi-tenancy pattern (the *Single Application Instance and Multiple Service Instances*) presented by the authors.

Charlton Model In 2009, a model-driven approach for building cloud solutions is also presented by Charlton et al. [30]. Three design goals, which are similar to our goals, are presented. These goals are the following: the separation of applications from infrastructure, the enablement of computer-assisted modeling and control automation, and the explicit collaboration to enact changes. In the same publication, the authors denoted eight characteristics that a Cloud Application should incorporate, in order to achieve the above-mentioned goals. Furthermore, this industrial paper introduced and briefly described three Elastic Modeling Languages (EML) for computing, deployment and management of elastic applications. The reference model we are presenting in this research acts as a meta-model for such languages and conforms with the goals presented in this paper.

Reservoir Manifest Language The Resources and Services Virtualization without Bar-

riers (Reservoir) [140] is a European cloud computing project supported by IBM. The project started in 2009, with the aim is to provide a modular, extensible cloud architecture to support service management and the federation of clouds. Reservoir utilizes virtualization to enables efficient migration of resources, maximizes resource exploitation, and minimizing their utilization costs. In 2012, Reservoir introduced its Manifest Language [29]. The Reservoir Manifest Language is an XML-based language that extends the Open Virtualization Format (OVF) [21] to enable developers to describe the cloud service requirements, deployment constraints and elasticity rules. A service manifest is a contract between the service and the infrastructure. The abstract syntax of this language is defined using the Essential Meta-Object Facility, while the behavior constraints of the underlying infrastructure are expressed in OCL.

Variation Analysis Variation analysis is the technique proposed in [164] as a way of designing cloud applications. The authors defined a set of Architectural Building Blocks and ways to assemble a Cloud-SOA solution from them, using variation analysis. This approach is specifically designed and checked for service oriented architecture software systems built using the service-oriented solution stack, a template layered architecture from IBM.

Maximilien Middleware The need to detach the cloud application development process from specific cloud platforms is addressed in [100]. A platform-agnostic middleware is proposed. This middleware lies on top of the Platform-as-a-Service layer. It provides API and services to be used by cloud users, and transparently deploys the application to a specific (but initially unknown) cloud platform. Interestingly, this approach relieves developers from the cloud vendor lock-in problem, but ties the development of cloud applications to the proposed middleware.

Overcat A bottom-up approach for assembling cloud applications from simpler components, the MacroComponents is presented in [99]. The proposed approach leverages a number of open source technologies, in order to provide a component model for building cloud applications. Open source technologies involved, include the OSGi component model, the P2 provisioning infrastructure for OSGi component model, and the CloudClipse, which is an eclipse plugin for managing the deployment and installation of specific virtualization images used in cloud platforms, such as Amazon EC2 or Eucalyptus.

C3A Framework CA Labs [26] proposed a cloud architecture to facilitate compatibility between ITIL and cloud computing, as well as portability of cloud applications between different cloud vendors. The goal is to maximize the return of IT investment

(ROI) in cloud computing. The approach's foundation is the C3A paradigm. The reference architecture consists of specific components, which enable provisioning of application's agreement on SLAs, and the migration of the application to different Cloud vendors. The proposed architecture is essential for migrating ITIL compatible applications to the Cloud, and managing existing Cloud Applications. Nevertheless, our research presents a reference model rather than a reference architecture. The proposed reference model facilitates the cloud application development, from the design to the implementation, in a transparent way, without depending on specific Platform-as-a-Service or Infrastructure-as-a-Service components.

In addition to the aforementioned projects, there are also distinct technologies and industrial projects that bear similarities to StratusML. These tools are usually described as DevOps tools such as Cloudify, Chef [31], Puppet [133], AWS CloudFormation [8]. These tools facilitate the deployment of applications and services, as well as the configuration management of cloud capabilities. Some of these tools are provider specific (e.g., CloudFormation), thus, they do not address the vendor lock-in problem. Moreover, all these works are at the code-based level; hence, they do not leverage the power of models, such as the ability to verify the correctness and completeness of the models.

2.2.3 Model-Driven Quality Prediction

Using model driven engineering for managing and configuring software systems at runtime to satisfy desired quality attributes is not new [18]. Examples of approaches that address this problem are surveyed in [38, 77, 89], and more recently in [77]. The shortcomings of these approaches are: (i) most of them are limited to performance, (ii) they focus on domains other than cloud computing, and (iii) they normally capture the dynamics of the software components only without considering the dynamics of the underlying resource model. The framework proposed in this thesis addresses these issues. In addition to performance, the framework addresses cloud specific issues such as availability and adaptation (elasticity) for malleable applications and fluid infrastructure.

Model-driven quality prediction approaches can be differentiated based on: (i) the analytical prediction model it uses, (ii) the architecture-level performance models that describes the system under test functionality, (iii) the model transformation process, and (iv) the level of automation involved.

- **Analytical prediction models** are used to analyze the non-functional properties of the system (e.g., system performance). Common analytical models include Petri

nets [3, 10, 110], Queueing Networks (QNs) [104, 147], Stochastic Process Algebra [35, 74], or Layered Queueing Networks (LQNs). There is a significant number of research studies that have been dedicated to evaluate the different performance models [45, 152]. Based on the recommendations of these studies, the StratusPM adopts the Layered Queuing Network Models (LQNM) as a target performance model. In a nutshell, queuing network approaches are more appropriate for performance analysis for systems with non-linear service centers such as in software systems, because of its ability to capture contention in resources and services [104], and because it does not suffer from state space problems as in petri-nets, markov models and other approaches that depends on state representations. Another important factor to select LQN is the availability of solvers that take as input a well formed model.

- **Architecture-level performance models** depict the system architecture and the key performance factors, resources, and usage profile of the system. According to Koziolek [89] classification, architecture-level performance models can be based on:

  (i) UML extensions (e.g., ComponentBased-Software Performance Engineering (CB-SPE) [16]), the UML Profile for Schedulability, Performance and Time (UML-SPT) [143], the UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [123]), or

  (ii) proprietary meta-models (e.g., KLAPER [33], the Component Based Modelling Language (CBML) [161], ROBOCOP [19], Palladio Component Model (PCM) [11]), Descartes Modeling Language (DML) [88]. The approach presented in this thesis is cloud specific and is based on proprietary meta-model (i.e., the StratusML meta-model).

- **Model transformation** in model-driven performance prediction approaches includes converting the source model architecture-level performance model into analytical model. Several transformation approaches are available [105]. Selecting the best approach depends on many factors. Mens and Gorp provided a taxonomy of model transformation approaches and tools [105] that can assist in selecting the best tool and approach that fits each case. Following their taxonomy and based on the source (architecture model) and target model (analytical model) we selected a transformation approach that support vertical, exogenous and refinement transformation to be able to bridge the architectural abstract modeling space with the XML technical space. Particularly, we selected a template-based transformation approach.

- **Level of automation**. Performance models can be created automatically using the information in the runtime model application or using the design specifications [25].

Regarding the latter approach, performance models can be derived from a variety of different design specifications such as the Unified Modeling Language (UML) including sequence, activity, and collaboration diagrams [131], execution graphs and use case maps [127], Specification and Description Language (SDL) [83], or object-oriented specifications of systems like class, interaction, or state transition diagrams based on object-modeling techniques [39]. Automation can also depends on how the mapping rules (transformations) are generated. This thesis provides a unique approach with this regard as shown in Chapter 6.

While several approaches have been proposed in the literature to deal with capturing the performance aspects, and automating the analytical performance model generation from component and service models [38, 43, 129, 160], according to our best knowledge, only few frameworks address the automatic generation of analytical performance models from cloud application artifacts and models. The most related to our project is the work by Franceschelli et al. framework "Space4Cloud" [51, 61]. The System PerformAnce and Cost Evaluation on Cloud (Space4Cloud) is an integrated environment for model-driven design-time QoS assessment and optimization of cloud applications. Space4Cloud has been developed as part of the MODACloud project. Both Space4Cloud and StratusPM frameworks share the same goal; derive analytical performance models from cloud applications. However, there is a number of differences between both frameworks. First, while Space4Cloud is based on PCM, our approach is based on extending LQNM. Second, the focus of Franceschelli is to anticipate performance analysis at early stages in the software development. However, our focus is to integrate the architecture of the application in the decision making process at run-time to enable dynamic adaptation for systems with variable architecture.

2.2.4 Schema Matching for the Cloud Vendors' Schemas

As explained earlier, the most common way to address portability is through abstraction approaches (e.g., meta-modeling, feature modeling, ontologies) [137, 146]. Abstraction approaches highlight the commonalities and differences between the different provider models. Several standardization bodies [92] have created cloud domain reference models and ontologies in an attempt to tackle the portability problem. Unfortunately, it is almost impossible to impose a single standard that covers all providers. In fact, several contributions in the literature have also concluded that having a single standard is not advisable in practice [73, 121, 142]. This makes schema matching a mandatory requirement for portability in order to bridge the gap between the different providers and standards.

Schema matching is the process of discovering correspondences between the elements of two schemas. Figure 2.4 shows a typical schema matching process that is composed of two steps; namely, *similarity analysis* and *elements mapping*.

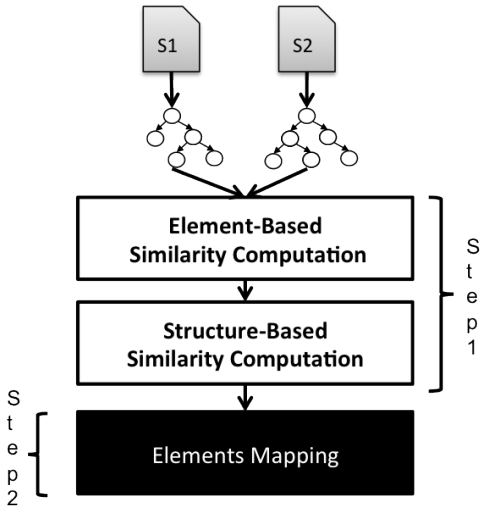


Figure 2.4: The Schema Matching Process [68]

Given two schemas (s1,s2), *similarity analysis* uses *element-based*, or a combination of *element-based* and *structure-based* techniques to measure the similarity between the two schemas. Element-based techniques usually depend on measuring the syntactic or semantic similarity between entities. On the other hand, structure-based techniques use the results obtained from the element-based techniques to measure the similarity between the inter-related elements. Normally, structure-based techniques use graph-matching algorithms to analyze the structural similarity between the schemas [103].

The *element mapping* process compares the similarities and filters the results to establish the final matching. Mapping can be done manually based on the proposed matches or automatically based on threshold filtering.

Matching schemas enables the knowledge and data of the source schema to be expressed with respect to the matched target schema, which facilitate portability [145]. Several solutions for schema matching have been devised in the last decades [9, 103]. Many surveys [136, 144], and books [13] have thoroughly covered the topic. Morover, several schema-matching tools have been developed by the database community in the past two decades; Cupid , COMA++ [7], AgreementMaker [41] and OpenII Harmony [112], are possibly the most noticeable.

Table 2.2 shows a comparative overview of these tools. These tools integrate useful matchers based on linguistic, and structural based matching techniques, they provide comprehensive GUIs, module importers, and facilitate domain knowledge matching based on external dictionaries. Unfortunately, none of these tools currently support semantic matching based on web search results, or incorporate domain knowledge in the way that Liberate does. For this reason the work presented in Chapter 6 not only paves the way for solving the cloud vendor lock-in problem, it also contributes to the schema matching community by making search-based semantic similarity matching, gloss based matching, and the generalized similarity flooding (SF++) available as part of one of the most comprehensive schema matching tools (OpenII).

Table 2.2: Comparison of Selected Match Tools

Comparison Criteria		Cupid	COMA++	Agreement Maker	OpenII Liberate
First time introduced		2001	2002/2005	2007	2008
Comprehensive GUI		×	√	√	√
Matchers	Linguistic	√	√	√	√
	Structural	√	√	√	√
	Web	×	×	×	√
Use of External Dictionary		√	√	√	√
Extensibility Support		×	×	×	√

The role that schema matching can play to address the vendor lock-in problem is unquestionable. To date, the only research initiative we are aware of that plans to adopt schema-matching is mOSAIC [113]. As part of mOSAICs work toward this goal, Cretella and Di Martino [40] described a schema matching prototype system that aims to support mapping of providers functionalities and resources between the different providers APIs. Different from our approach, Cretella and Di Martino approach uses a traditional linguistic schema matching approach based on syntactic analysis and WordNet thesaurus. The study presented in this thesis uncovers the problems in such approaches and provides a solution through incorporating domain knowledge using web-search based semantic matching (Web Similarity Measure).

The relationship between cloud computing and schema matching is mutual. Schema matching contributes to solving the vendor lock-in problem, while cloud computing can be used to develop efficient schema matching systems to address some limitations of existing

mapping processes (i.e., scalability). For example in [118] the authors use cloud computing to support the collaborative reconciliation of schemas.

Using web metrics for semantic disambiguation is not new. Cilibrasi and Vitanyi paper "The Google similarity distance" [34] has been cited over a thousand times since 2007. The Liberate approach proposed in Chapter 6 distinguishes itself by virtue of our application domain and implementation. Chapter 6 provides a case study from the cloud domain, and evaluates the applicability of web-metrics in addressing the vendor lock-in problem.

2.3 Summary

This chapter reviewed briefly the principles and terminologies in the domain of cloud computing and highlighted some of the most related research projects to this thesis. The projects selected are those that are classified as multi-cloud frameworks that address multi-concerns and target the portability and interoperability issues.

Our initial investigation shows that there is no unique solution that takes into account all the aspects required to manage the deployement and operation of cloud applications into multi-clouds. Moreover, there is a need to automate the process of mapping the different providers' artifacts and generating transformation rules from the standardized languages to the platform specific artifacts. In the next chapter we will present our approach to address the muti-cloud concerns that need to be taken into consideration while building cloud native applications.

Chapter 3

An Architecture Framework for Cloud Native Applications

"Being cloud-native is more about the application architecture and design than how you code the thing... cloud native applications are 70% more efficient than traditional applications migrated to a cloud"
–David Linthicum

This chapter presents an architecture framework for cloud applications, here after called the (5+1) view model. The ISO/IEC/IEEE 42010 Conceptual Model of Architecture Description [76] defines the term architecture framework as: "A framework that establishes a common practice for creating, interpreting, analyzing and using architecture descriptions within a particular domain of application or stakeholder community".

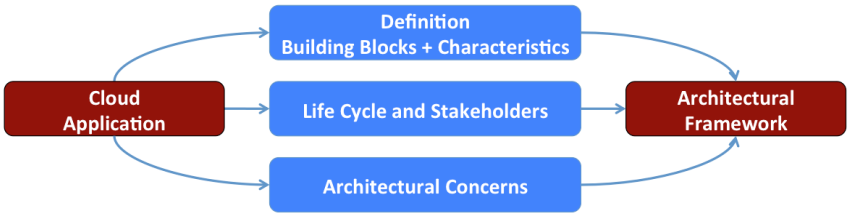


Figure 3.1: Creating an Architectural Framework for Cloud Applications

To devise an architecture framework for cloud applications, we follow the approach depicted in Figure 3.1. We start by characterizing cloud applications in Section 3.1. Then we study the cloud applications life cycle and stakeholders in Section 3.2. Followed by

the set of architecture-related concerns that can affect the application malleability and manageability in Section 3.3. Accordingly, we introduce the (5+1) view model for cloud applications: an architectural framework that uses views to cover the identified cloud concerns and define the set of model correspondence rules between the different views.

3.1 Characterizing Cloud Applications

To build a modeling framework for cloud applications, it is important to have a clear understanding of what is a "cloud application". Unfortunately, so far there is no consensus on such a definition. This section explores the concept from different perspectives, and identifies the cloud applications' building blocks and essential qualities and characteristics.

3.1.1 The Definition of Cloud Applications

Different cloud providers have diverse views of what is a cloud application. IaaS providers call any software application that can be deployed on their infrastructure a cloud application. Software as a Service (SaaS) providers use the term to refer to softwares developed by composing web services. Lastly, PaaS providers refer to the applications developed and managed using the set of tools these platforms provide as cloud applications.

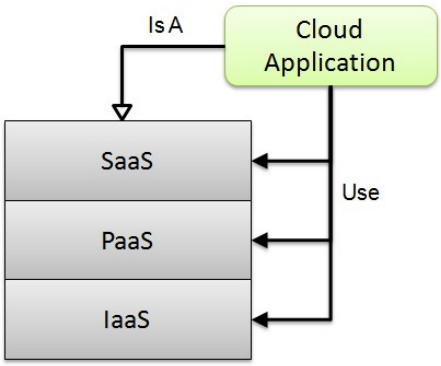


Figure 3.2: The Relationships between Cloud Applications and the Different Cloud Computing Service Deployment Models.

In fact, all of the aforementioned views are correct. In software architecture, one way to define applications is based on the basic units of composition used to create the application (e.g., Object Oriented, Service Oriented, Component Based). Accordingly, depending on where the provider lays on the cloud service deployment model (i.e., IaaS, PaaS, SaaS), and the granularity level of the components used to build the application (i.e., Virtual Machines, Containers (which we refer to as Tasks in this thesis), Components or Services), each service provider will have a different perspective.

Figure 3.2 presents an unbiased view of the cloud application definition, by relating it to all of the cloud computing service deployment models. Based on figure 3.2, a cloud application *is a* software that is provided to consumers as a service. A cloud applicaton utilizes the cloud services provided at the SaaS, PaaS and IaaS service levels to perform its operations. Accourdingly, in order to model cloud applications, the proposed modeling framework shoud support modeling the providers' specification at the (IaaS) level, how they are provided as service models at (PaaS) level, and how the service models templates and services provided by the platforms providers can be composed into higher level services and workflows at (SaaS) level.

In this thesis we argue that to call an application a cloud application, it should be designed to be *cloud ready*. In other words, a *cloud native application* should be designed from the ground up using *modular components* communicating through *lightweight and standardized mechanisms* (e.g., REST). These components should have the ability to be *reused*, *reconfigured*, *recombined*, and *recomposed* independently and at runtime. By designing the application in such modular fashion, the application can be *distributed* and *parallelized* when needed. Morover, it can be *independently deployed* on demand over a pool of virtual machines (VMs) with theoretically unlimited resources via *automated deployment* machinery.

As the reader proceeds through this chapter, the definition of cloud applications should be more clear. Section 3.1.2 covers the selection of a cloud application building blocks, hereafter we call them *Tasks*, and the rationale behind it, while Section 3.1.3 higlightes some of the characteristics of cloud applications that distinguish them from other types of applications.

3.1.2 The Cloud Architecture Building Blocks

A software system can be decomposed into components of different granularity levels. A *component* is "a unit of composition with contractually specified interfaces and explicit

context dependencies" [149]. It can be an object, a service, a microservice [151], a program instance, or a virtual machine image.

Components are usually designated based on the type of input they consume and output they provide. Modern service oriented programming models (e.g., OSGi, SCA, Jini) describe components as functional units of composition that take services as input (i. e., by referencing them) and provide services as output (i.e., by exposing service contracts). Cloud systems follow the same trend. However, in cloud computing, services are classified into layers (i.e. Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS)), where components in upper layers consume services from the same layer or the layer below and provide services to be used in composing other services or applications at the upper layers. The higher the service layer that hosts the component, the courser the granularity of the component.

Selecting the right granularity level of components can have a significant impact on the software architecture. It can affect the complexity of the design, the ability to reuse the architectural elements, and the quality of the architecture. This section, analyzes the cloud applications' components to determine which level is the best in order to describe a cloud application in a way that maximizes controllability and ensures the cloud application scalability, availability and portability.

Figure 3.3 shows the different units of composition of a cloud application as perceived by developers at different service deployment layers. As shown in the figure, it is possible to distinguish three different units of composition.

(i) **Services at the SaaS layer** : A SaaS service is an application provided as a service that consists of at least one *Task*. Services can be orchestrated into business processes using any SOA business process modeling language.

(ii) **Tasks at the PaaS layer**: Each *service* at the SaaS layer is composed of a set of collaborative *Task*s at the platform level (PaaS), in addition to the definitions and relationships between these *Task*s, and their logical configurations.

(iii) **Processes at IaaS layer**: A *Task* with its platform configuration is packaged as a virtual machine image to be deployed as a running *process* on a cloud infrastructure. Several tasks can be packaged in the same virtual machine image.

As shown in Figure 3.3, at each of the service layers (i.e., SaaS, PaaS, IaaS) the application consists of components that consume the layer beneath services to provide higher

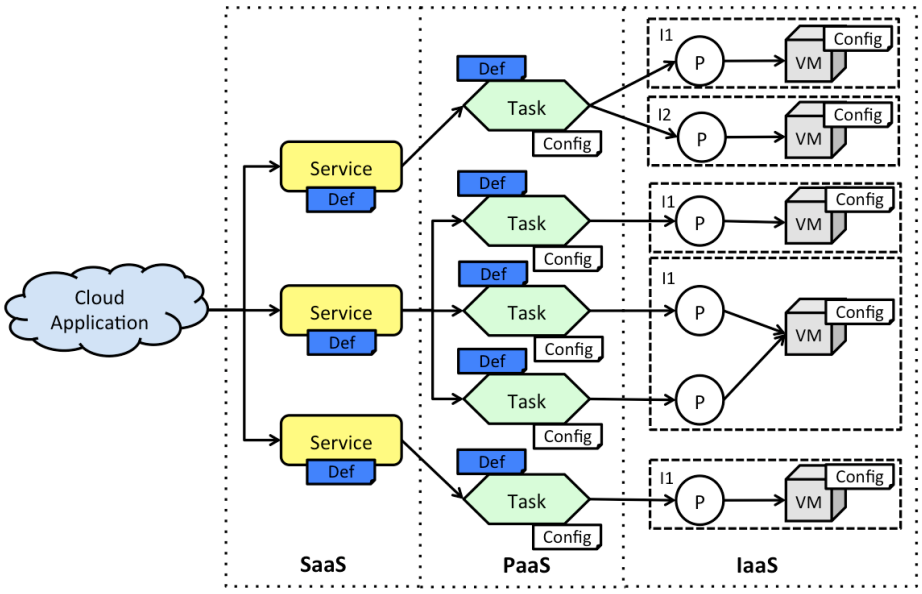


Figure 3.3: Cloud Application Components Granularity.

level services. When the services consumed are middleware services, the developer's focus will be on the component itself (Task) and *How* to exploit the middleware services to build a task that can address both the functional and non-functional requirements of the application (i.e., service). In contrast, when the consumer services represent business functionalities, the developers' attention will be on *What* are the services needed in order to achieve the business goal.

Coarse-grained components have higher contributions towards the solution; hence, they have high reuse efficiency. However, because coarse grained components are tailored to provide solutions for specific problems, they have a lower level of reusability. Moreover, course-grained components abstracts the technical details of the platform; hence they are more useful for modeling business processes. In contrast, fine grained components are more suitable for technical-level design that is why they are more favorable by developers.

Accordingly, the layer of abstraction selected is on the top of the PaaS layer. The selection of PaaS as the layer of abstraction has been made based on the following criteria:

(i) **Level of Detail**: To maximize reuse and bridge the gap between business requirements and the technical implementations, If a component does not have the ability to fully express the requirements through configuration, its granularity level should be decreased. Accordingly, the right granularity of the main unit of composition should

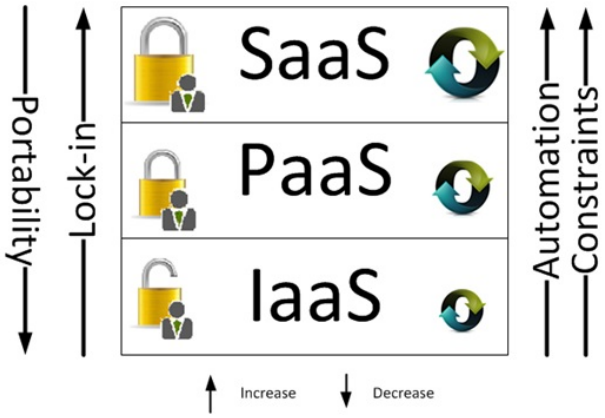


Figure 3.4: The Tradeoffs between Applications' Portability and the Level of Automation.

enable translating the requirements into functionalities that utilize the middleware capabilities of the cloud platforms.

(ii) **Application Manageability**: Granularity has a negative correlation with managerial goals, such as cost effectiveness, customization and maintainability[158], and a positive correlation with component assembly. Selecting the granularity at the PaaS level enables taking advantage of both.

(iii) **Reduce Coupling and Design Complexity**: Raising the granularity of the components can create a level of abstraction that simplifies the communication with the upper layers. This is important for distributed applications (e.g., cloud applications) where remote service invocations may lead to network performance problems. On the other hand, decreasing the level of granularity can decrease design complexity. A balance between fine- and coarse-grain components can help address both issues.

(iv) **Automation vs. Portability**: Portability and automation vary between SaaS, PaaS, and IaaS. In general, the number of constraints increase as we move up to the SaaS layer. The more the constraints the less the portability and the higher the risk of vendor lock-in. On the other hand, while cloud infrastructures enable quality attributes, such as dynamic elasticity, and high availability; these attributes cannot be fully addressed at the infrastructure level. The higher the level of abstraction the easier to automate these characteristics. This makes the PaaS layer the best candidate to support portability and enable automation.

There is a trade-off between selecting a courser or a finer component granularity. Se-

lecting the right granularity level can assist in alleviating many architectural issues. Based on our analysis the best level of abstraction to achieve the highest flexibility to control the cloud application quality is right above the PaaS layer and below the SaaS layer. Hence, a cloud task is the basic unit of granularity and the main composition unit for cloud applications. Tasks can be composed into services and can migrate between platforms without the need to perform any changes to their code. Changes can only be applied to the definition and configuration artifacts. More about cloud Tasks types and relationships will be addressed later in the next chapter.

3.1.3 Cloud Applications Requirements

Characterization should not be limited to building blocks of the application. It should also consider the applications' requirements and feature that can affect the application's behavior (e.g., Distributed, Real-Time, Embedded), and its lifecycle. The following are some of the requirements that should be satisfied in order for an application to be considered a cloud application:

- **Req1:** *Self-aware:* A cloud application should be aware of the following:
  - (i) Its architecture (i.e., Tasks and relationships diagram).
  - (ii) Its resource requirement (i.e., the required resources to successfully deploy the application on any platform).
  - (iii) Its status (the runtime model and the configuration under which it is currently running).
  - (iv) Its service level operational requirements e.g., (desired required availability, scalability and performance).
  - (v) Its cost model constraints and status.
  - (vi) Its location: In order for the cloud application to support high-availability and automatic failure recovery, the cloud application architect should have the ability to decide how the different component instances should be distributed on different fault domains. For example, depending on the availability level required the architect should be able to decide whether the different instances of a certain component should be located in the same datacenter, or distributed across different geographical locations (regions).

- **Req2:** *Malleable:* A cloud application should have the ability to be composed from components that can be reused, re-combined, and re-configured. These units of composition should:

  (i) Follow microservices architecture style [151]. This means that it should be possible to partition the application into fine-grained services. Each service perform one task.

  (ii) Components should use smart endpoints that embed the logic of communication (i.e., naming, discovery, and connectivity) within the endpoint to remove dependency on the infrastructure connections (pipes) (e.g., Enterprise Service Bus (ESB)).

  (iii) Building asynchronous communication wherever possible in the application architecture would allow for loose coupling amongst components. Asynchronous behavior is at the heart of cloud native applications.

  (iv) Components should use standardized interface to facilitate automating the process of establishing communication between the components. This makes it easy to build configurable workflow applications. Today, the most common APIs in the cloud are the REST based or RESTful APIs. This is because REST APIs use the standard HTTP methods (i.e. GET, PUT, POST, DELETE, etc.) to standardize the way services and resources are exposed to the user.

- **Req3:** *Self-adaptive:* A cloud application should have the ability to utilize the resiliency and autonomic capabilities of its underlying platform. Accordingly, it should have the ability to specify policies, constraints, rules and actions to maintain its behavior under various condition, scenarios and environments.

- **Req4:** *Multi-Cloud*: A cloud application should be architected from platform independent components that can run in the same way on different platforms. It should also have the ability to run and migrate at runtime from one platform to another, and between on-premises and off-premises environments.

- **Req5:** *Multi-Tenant*: A cloud application should support a hybrid (single/multi) tenancy model at the component level granularity. Multi-tenancy is the ability of a component to be shared with multiple users by having a single logical instance of that component. On the other hand, single tenancy allows creating several logical instances of the same component, one for each user. Supporting a hybrid tenancy architecture at the component level provides flexibility of choice for the developers and architects, as they can select the tenancy model that fits their application. This

process can be done for each individual component and at any of the software architecture tiers (i.e., presentation, business logic, and data storage). For example, for components that have high-demand and require high-performance each user can have a separate instance of the component; in contrast, the same instance can be shared between multiple users to allow full utilization of the resources when demand is low. Enabling a hybrid tenancy model is a prerequisite feature to support both dynamic elasticity and high availability.

- **Req6:** *Manageable:* Cloud applications should provide management interfaces that allow monitoring the applications and changing their properties at runtime. This is an important prerequisite to automate the applications lifecycle management, and support the applications self-adaptivity.

- **Req7:** *Measurable*: Cloud applications should provide mechanisms to quantify the resources they use. This is important for the cloud application to support the dynamic price models, as well as testing and monitoring.

- **Req8:** *Distributed at the enterprise levels:* A cloud application is a distributed application. It supports distribution at the presentation level, the business logic level, and the data level. At the presentation tier, a load balancer handles the distribution of requests into a number of distributed web tasks, which process the requests, then assign jobs to the appropriate background (business logic) task. Usually a cloud application performs jobs that can be divided into a number of atomic tasks (workloads) that can be processed in different machines and then accumulated. Finally, at the data tier, a cloud application can make use of distributed cloud storage (i.e., blobs, tables, queues) to support fault tolerance and availability, while maintaining eventual consistency. These distribution capabilities are what make a cloud application highly available and elastic.

- **Req9:** *Accessible through web-enabled interface:* Each application should have at least one web-enabled endpoint . Having a web-enabled interface does not mean that the application should be publicly accessible from the internet, but instead it should be enabled to be widely accessible whether this is through intranets, extranets, or the internet.

- **Req10:** *Has the capability to be upgraded modified and updated with minimal service interruption*: Cloud applications should provide mechanisms that allow tracking the location of each component, and group related components within the same update-domain, as to be updated or modified at the same time.

By satisfying the previous requirements the cloud application will comply with the cloud computing's main characteristics previously defined by NIST. The cloud application will be available, support fault tolerance, scalable on-demand, and elastic. A cloud application is an application that supports reuse of software services and components with affordable prices based on a pay-per-use model. On the other hand, if an application does not utilize the cloud computing infrastructure or platform,then this application cannot be called a cloud application.

Note that many of the cloud application characteristics are shared with existing paradigms, such as real-time, service oriented, and distributed computing. What is unique about cloud computing is its ability to combine all these paradigms and provide simple and efficient reuse mechanisms. These mechanisms allow users to get the intended benefits from the existing technologies with minimal efforts and basic knowledge of technicalities.

In a nutshell, a cloud application is a special Service Oriented Architecture (SOA) application that runs under a specific environment (the cloud environment). This environment is characterized by horizontal scalability, rapid provisioning, ease of access, global reach, high-availability, and so on. Cloud applications share lots of concepts with SOA such as: focusing on reusability at the large scale, creating applications by composing pre-existing autonomous components, and focusing on evolution. However, being a specific case of SOA applications mandates addressing the Cloud domain specific issues and variations. The relationship between cloud applications and SOA is similar to the relationship between Web-Services and SOA; both can be considered as SOA implementations. However, cloud applications are more generic than Web-Services. In fact, Web Services are a type of cloud application services. The detail of the relationships between cloud computing and the paradigms it has evolved from is explained in detail in the background chapter.

3.1.4 Term Disambiguation

The term task used in this thesis has been coined in 2010. The term has been defined in our paper that was published in the 1^{st} International Conference on Cloud Computing and Services Science as follows: "A Cloud Task is a composable unit, which consists of a set of actions that utilize services to provide a specific functionality to solve a problem. It is a mutated unit that can be copied to other virtual machines in order to allow horizontal and vertical scalability. When composed, tasks should satisfy the following principals: statelessness, low coupling, modularity, and semantic interoperability. Tasks are semantically connected to other tasks in the cloud through the roles they play in order to satisfy a specific business requirement, which is bounded by obligations or responsibilities. Cloud tasks

are uniquely identified by a global Dynamic Name Service (DDNS) that can be assigned to a dynamic virtual IP address at run time. This makes the task highly available and fault tolerant, and allows the cloud application to be dynamically upgradable without any interrupts" [66].

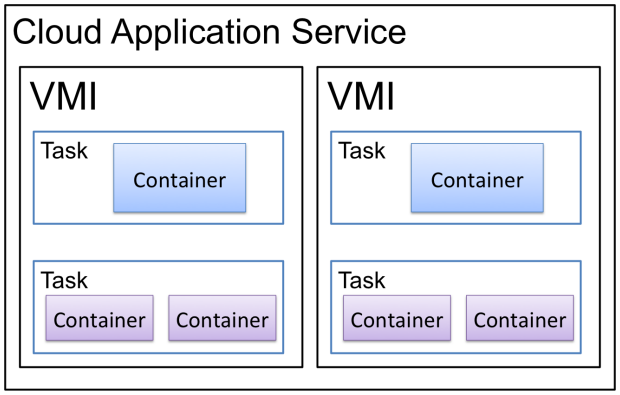


Figure 3.5: Cloud Application Composition

Since that time the cloud domain has evolved rapidly. Different terms and technologies have been emerged to explain similar concepts. In march 2014, Martin Fowler and James Lewis wrote a joint article that describes a new architectural style that explains the composition of small, independent processes communicating with each other using language-agnostic APIs. Those units are called microservices. They are small, highly decoupled services that focus on doing a small task. The term "task" conforms with what Fowler and Lewis refer to as a "microservice". The term "task" also aligns well to another trend in the cloud domain. This is the container-based development. Figure 3.5 shows the relationship between the "task" and the "container" concepts. Container technologies, such as Docker, allow multiple containers to share the same kernel while running in complete isolation from one another. This makes deploying the code of a task or a microservice that is developed using different languages and frameworks very easy. Containers are a better choice to realize microservices or tasks than virtual machines as they offer (i) ability to start-up and shut-down more quickly, (ii) better resource utilization (i.e., computing, memory and other resources can scale independently) and (iii) easier life cycle management.

The following is a clarification of the relationships between the compositional units used to describe the architecture of a cloud application service model.

- **Microservice**: A conceptual modeling concept that refers to the source code of a fine-grain service that does one job, but does it well.

- **Container**: A build image of a micro service with all OS and binary dependencies that allow the microservice to run the same way on different environments.
- **Task**: A container along with its platform independent specifications (resource requirements), end-point interfaces and interaction mechanisms with other containers and the environment.
- **Cloud Service**: A model composed of cloud tasks and follow a microservice architecture.

The next subsection focuses on the development and operation process of a cloud application.

3.2 The Cloud Application Development and Operation Process

According to the Information Technology Infrastructure Library (v3), there are three keys to the success of any application development project: people, process and technology [28].

Prior to the cloud, the bottleneck in IT delivery projects where mainly due to technology; the lack of resources when needed and the long waiting times for resource provisioning. Cloud platforms are Software Defined Environments (SDE) that uses technologies for capturing the whole infrastructure and platform *as a code*. Offering the infrastructure and platforms as code facilitates managing the scale and the speed with which environments need to be provisioned and configured to enable continuous delivery. This reduces the operational complexity of building, deploying and operating applications. It also pushes the bottleneck to the process and people side, calling for tools to automate the process of software delivery pipeline and to improve collaboration between people.

3.2.1 The Cloud Application Life Cycle Management (CALM)

Figure 3.6 shows a typical software *delivery pipeline* for cloud applications from development through production and evolution after production.

The steps in this pipeline may vary based on the organizations needs, the maturity of the software delivery process, and the level of automation involved. The delivery pipeline has the following steps:

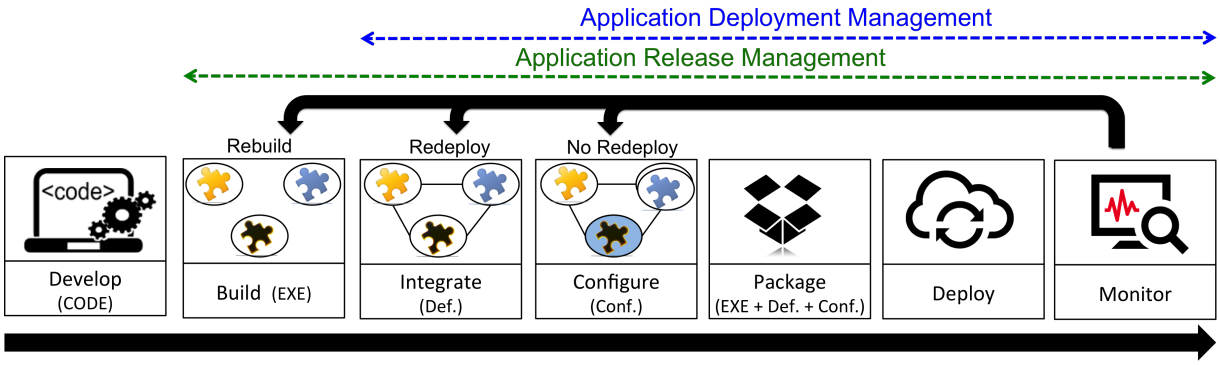


Figure 3.6: The Cloud Application Life Cycle Management Process

(i) **Develop**: This stage focuses on creating (coding) loosely coupled fine-grained services (microservices). Each provides a specific well defined functionality. Several integrated development environments (IDE) can be used in this stage depending on the developers preference and the IDEs support for languages, tools libraries and collaboration. Each microservice can be developed with different IDE using the best and most appropriate tool for the job. However, the rule of thump in this stage is to break the application down into small but high cohesive services.

(ii) **Build**: In this step the code created in the previous step is pulled from the code repository and compiled to create binaries, or if you follow the current trends into fully encapsulated containers that encapsulate the entire state around an application. Encapsulating the binary with all its dependencies as an image facilitates continuous integration and delivery and ensures that the code will run exactly the same in production as in development, testing and staging environments. Again several build tools and image building and management solution may be used in this step, depending on platform technology (e.g., Docker-based build[1]).

(iii) **Integrate**: So far, we have fine-grained services. Each of which preforms one job and encapsulated with all its dependencies as an image. In the integration step, these isolated fine-grained services will be weaved into an application, by defining their endpoints and interaction mechanisms. Establishing connections requires also set of security, authentication, and failure recovery rules. For example patterns such as circuit breaker [119] could be considered to avoid overloading other services.

(iv) **Configure**: In this step each of the tasks (containers and their definitions) will be

[1]https://www.docker.com

configured to be deployed on a target cloud datacenter. Configuration is a challenging task in distributed systems, and even more challenging in the cloud, where both the platform and infrastructure parameters can be tuned using configuration code. The price of this flexibility is a large number of configuration options and scenarios that need to tune for optimal performance and operation.

(v) **Package**: Packaging refers to bundling the application binaries with the artifacts that are required to successfully deploy, instantiate, distribute and manage the application on a particular cloud platform. Examples of these artifacts include: configuration files, infrastructure-as-code files, deployment scripts, policies, scalability or adaptation rules and so on.

(vi) **Deploy**: After packaging the application according to the target platform templates, the rest is to upload the package to the cloud. In the cloud side, deployment automation tools (i.e., environment management and provisioning tools) will use the different artifacts in the package to provision the resources required and distribute the application with the ratios specified.

(vii) **Monitor and Manage:** After deployment, monitoring tools allow organizations to monitor the deployed applications in production, and management tools allow them to manage the application while running in the cloud.

This research does not focus on how to build these individual loosely coupled services and components, but on how these components are connected to each other (which depends on the platform used (e.g., Azure, GAE)), and whether the configuration matches the target server template of the cloud datacenter (which depends on the underlying infrastructure). This is because, the way the components are connected (application definition) can change the static behavior of the program, while the configuration can affect its dynamic behavior. Moreover together definitions and configurations can determine if the application can be deployed to a particular cloud datacenter and hence affect its portability.

3.2.2 Envisioned Users

Common practice shows that framework developers often have in mind a set of established stakeholders within the domain of the framework [46]. The stakeholders motivate the set of concerns which the architecture framework will focus upon

The following list describes four different cloud stakeholders that we envisioned to have maximum benefit from the proposed framework:

(i) **Platform Providers** can specify the resources and services they provide.

(ii) **Service Developers** can define their application services.

(iii) **System Administrators** can configure the services for deployment, specify the application runtime behavior through a set of adaptation rules.

(iv) **Performance Engineers** can evaluate the application performance under different configurations and platforms.

(v) **Financial Managers** can estimate the cost of deployment onto different platforms.

One of the main goals of the cloud architecture framework is to facilitate collaboration between different cloud stakeholders. The next subsection provides more elaboration on the cloud architecture main concerns and how the proposed framework enables collaborate between the aforementioned stakeholders to address the various concerns.

3.3 Architecting for the Cloud

Grady Booch has been quoted as saying that "building a dog house is different from building a high-rise" [21]. This is because the different scale entails different requirements, complexity, and management. Today cloud applications development typically resemble high-rise buildings. In a high-rise architecture, you design the first-floor sketch then replicate it as needed. Similarly, in cloud applications you design a schema that represents your service model (e.g., containers graph) then replicate it as needed on-demand through service configuration. The main difference between a cloud and a typical high-rises architecture is that a cloud application continuously evolves at runtime. In the cloud, the behavior of the cloud application depends on how the service model is designed by connecting several container-unites; however, the specific functionality depends on the loosely coupled fine-grained services that inhabit these units. This is analogical to organizations where departments' relationships are clearly pre-defined, however; the performance of each department depends on the performance of its employees.

Architectural frameworks and view modeling is a well-established practice for describing and analyzing complex software systems. Existing view models and frameworks include the (4+1) view model [91], the Three Schema approach [84], the Zachman framework [162] and the Department of Defense framework [120]. These frameworks all clarify software *design*, because that is when architectural decisions were traditionally finalized. Static deployment

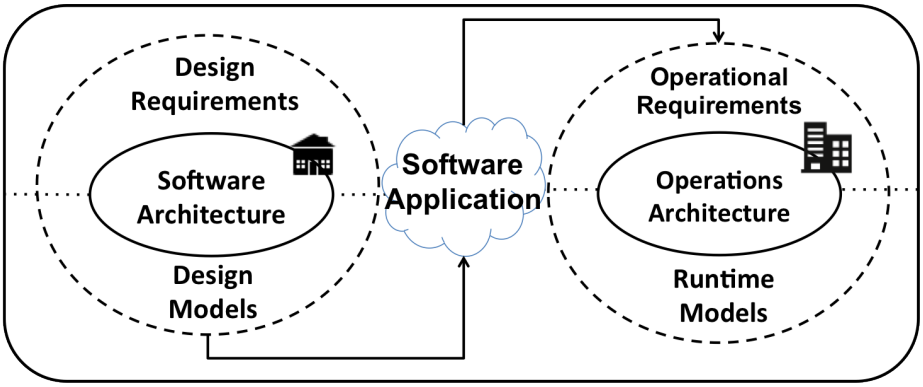


Figure 3.7: Architecting for Application Implementation v.s. Service Operation

infrastructures were the norm. In cloud computing the applicatoin architecture evolves during deployment. *Runtime operation* needs as much architectural modeling as design does [150]. Cloud applications must morph *at runtime* to meet performance, availability, and scalability targets under changing conditions.

This shift in emphasis from *architecting for implementation* to *architecting for operation* is illustrated in Figure 3.7. Models of implementation architecture capture the *design requirements* to create *design models* that reflect the *source code* of the application. Models of service operation architecture capture the *operational requirements* to create *runtime models* that describe the *configuration space* of the application. The practices complement each other. However, the latter is more malleable and so better illuminates the dynamic evolution of cloud applications.

3.3.1 The Malleable Application Architectural Style

Cloud computing improves service availability, minimizes downtime, and scales applications on demand. This means cloud applications must morph during runtime without requiring redeployment or restart [67]. They must scale out by adding new instances to meet demand, and scale up to larger virtual hosting machines as needed. They routinely switch tasks on/off to alter their behavior, and they may need to change the wiring between tasks that communicate through common storage (queue, blob, etc.). To support performance debugging, they must be able to switch between logging fine or coarse grained system dynamics. Figure 3.8 shows that this level of *flexibility* can be achieved by:

(i) *Separating the configuration space concerns*: Separate the executable components,

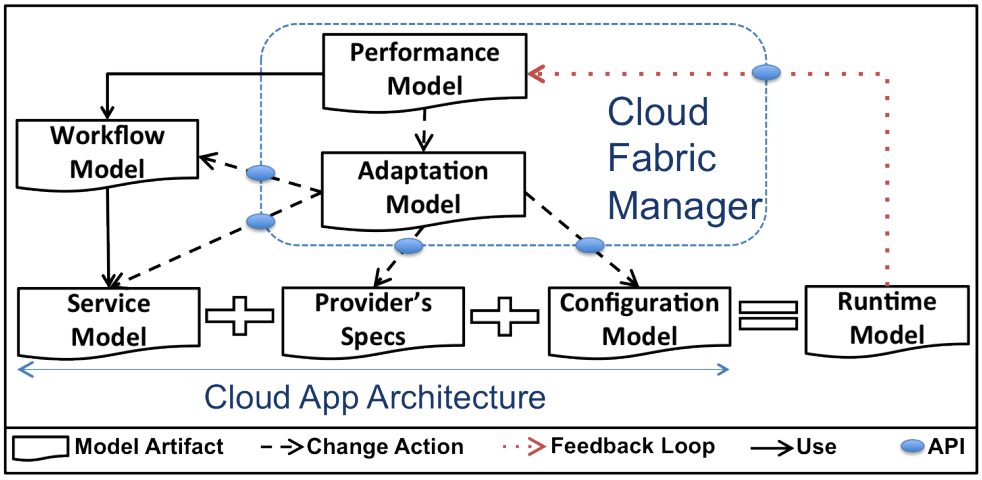


Figure 3.8: The Malleable Application Architectural Style

from the application structure, configuration, the target resource specifications, and execution scenarios and control.

(ii) *Supporting automation*: Use performance and adaptation models and enable continuous monitoring of the environment.

Each concern will be represented in a separate model. The *service model* specifies the tasks provided by the cloud service, their types, and their relationships. The *configuration model* specifies the replication of the tasks, and their concurrency and distribution. The *provider's specifications* specify the resource configuration of the target hosting environment. Lastly, the *workflow model* represents different usage scenarios for the service model tasks. Each of these models can change separately to meet desired operational requirements.

In order for the changes to be applied automatically at run time, an *adaptation model* should be defined to access all the elements and parameters of the models that need to be changed. The *adaptation model* uses key performance indicators to initiate change requests. These indicators can be gathered from the *runtime model* and represented in a *performance model* to enable performance analysis.

We call the afformentioned pattern the *Malleable Application Architectural Style (MAAS)*. MAAS is a common architectural pattern in cloud systems. It relies on separating the configuration space concerns (structure, configuration and control) from the *executable components*. In Figure 3.8, the control model corresponds to the performance and adaptation

models, which are defined to influence the actions of the *Cloud Fabric Manager*. The fabric manager is a platform specific *autonomic manager* [75] that uses a set of adaptation rules and performance indicators to enact change actions on the target model using APIs.

3.3.2 The (5+1) Architectural View Model

Fragmenting models into artifacts based on machine prespectives faciliates automation and increases flexibility. Unfortunately, this fragmentation makes it difficult for humans (cloud stakeholders) to create and comprehend these models. The information in these fragmented models are usually specified by different stockholders. Moreover, this information is often overlaps on variety of ways. Accourdingly, there is a need to weave the fragmented models into views that enable stakeholders to better understand the runtime dynamics and evolution of complex cloud applications. This section analyzes the artifacts in MAAS from architecutral prespective based on dependency and containment relationships, and the stakeholders involved in specifying the different parts of these artifacts. Accordingly, we propose the (5+1) architectural view model, where each view corresponds to a different perspective on cloud application deployment.

Figure 3.9 shows a high level architectural perspective of the relationship between the different artifacts required to deploy an application on a target platform. As show in the figure, the service model plays a centralized role in this architecture. To deploy an application onto a cloud platform; architects need to partition the application into software processes and package them into loosely coupled software modules (i.e., Tasks) with the required software stack and *app* or *web* servers, and determine the virtual resources required to run the modules. Accordingly, a service template with the required specification can be selected and the application can be packaged into a set of Virtual Machine Images (VMI). Administrators then can instantiate and replicate these VMIs into the target cloud datacenters and specify the rules that govern any configuration changes based on monitoring the environment conditions (e.g., load), or by analyzing the system using the performance models that is created by the performance engineers.

Figure 3.10 shows the (5+1) architectural view model. The (5+1) view model consists of five model views that specify the *core/service* model view to address five different, but interleaved cloud concerns related to service deployment and evolution. Each model view conforms to its corresponding meta-model. Moreover, the top element in all the model view meta-models extends the (5+1) component (i.e., a meta-meta model element). The aforementioned modeling hierarchy makes it possible to integrate all the (5+1) meta-models and facilitates organizing the different modeling elements into categories (layers) based on the cloud concern they address.

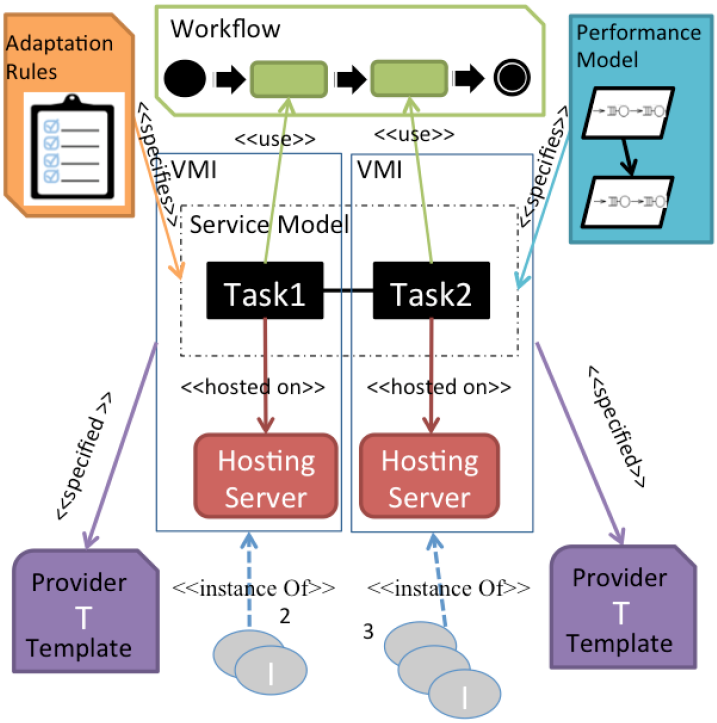


Figure 3.9: The Architectural Perspectives of Cloud Stakeholders

The details of each of the (5+1) meta-models and the methodology of defining them will be explained as part of defining the StratusML modeling language in chapters 4 and 5 respectively. These meta-models capture all essential information for architecting malleable and platform independent cloud applications. The (5+1) approach is based on the assumption that containers can deal with code level mismatches by letting software components run on low level infrastructure. We argue that this assumption is reasonable given cloud dynamics. The constant updating and multiple service providers that characterizes cloud computing drives mismatches between infrastructure and applications that impact deployment architecture. The (5+1) view meta-models make this assumptions explicit.

In a nutshell, the *core* meta-model is a fairly comprehensive pivot model that describes the application's deployment architecture in terms of tasks and interactions. It clarifies the cloud service model and its requirements in terms most vendors would understand. Each of the other five meta-models further enrich the expressiveness of the core model. The *performance* meta-model enables annotating the core model with performance parameters. The *adaptation* meta-model specifies adaptation rules and actions for each task or group of

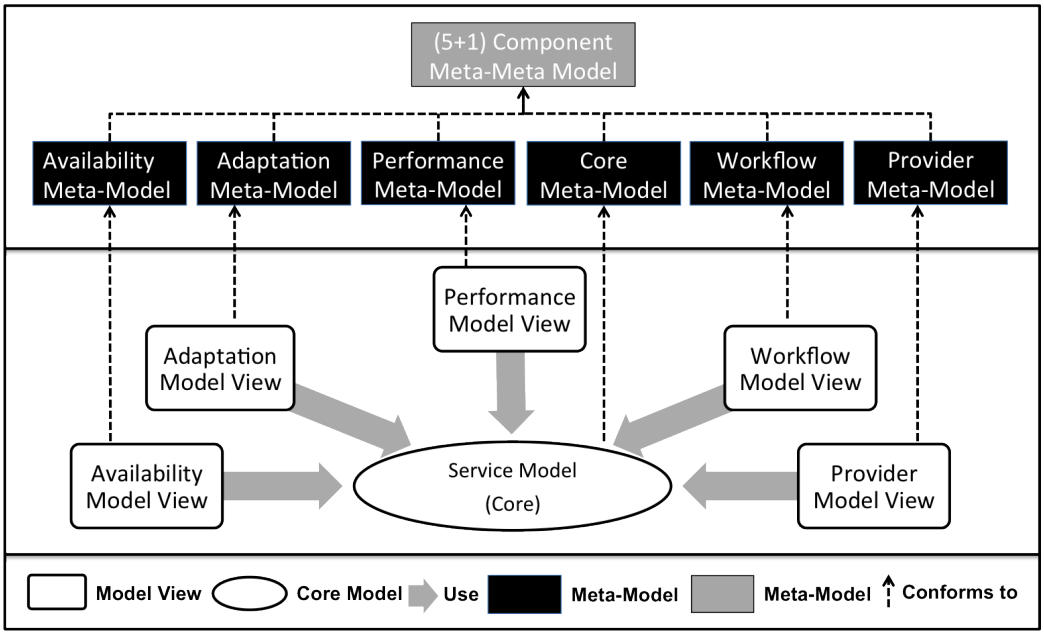


Figure 3.10: The (5+1) Architectural View Model

tasks in the core model. The *availability* meta-model defines the model configurations (i.e., configuration models in MAAS). Its components instantiate the core model and distribute its instances to different geographic locations. The service *workflow* meta-model facilitates creating different service scenarios, by composing and executing core tasks in series to achieve a certain goal. Finally, the *provider* meta-model creates a provider profile, which consists of the provider's service templates and pricing profile.

3.4 Summary

This chapter introduced the (5+1) view model for cloud applications. The (5+1) view model is an architectural framework that captures the essence and the detail of the cloud applications' characteristics that affect their dynamic evolution at runtime. It enables cloud stakeholders (e.g., providers, developers, administrators, performance engineers and financial managers) to leverage cloud platform capabilities to maximize availability, maintain performance levels, minimize costs, and leverage portability and scalability.

At the core of the (5+1) is the cloud application *service model*. This model describes a reconfigurable executable units of composition that we call *Task*s. The service model is

further specified using a set of four *operational model views* that address the application performance, adaptation, availability, and execution scenarios, and one model that address the portability and operational cost of the application though capturing the platform *providers* specifications.

Chapter 4

StratusML: A Domain Specific Modeling Language for Cloud Applications

"I believe in a visual language that should be as strong as the written word."
– David LaChapelle, 1963

It is widely acknowledged within the software engineering community that architectural languages are defined by stakeholder concerns [95, 101], and that a single "universal" notation (e.g., UML) is impractical to address all the concerns of a particular domain [95]. A domain language is usually needed. As explained in chapter 3, in the cloud domain, an application evolves *at runtime* to meet performance, availability, and scalability targets under changing conditions; a routine task that involves continuous changes to the application service models and deployment artifacts. While each stakeholder may conduct a certain type of change to address a specific concern, the impact of a change may span across multiple models and influence the decisions of several stakeholders. This chapter presents StratusML, a multi-cloud (i.e., platform independent) modeling framework and domain specific modeling language for cloud applications. StratusML is a realization of the (5+1) view model. It aims to satisfy the cloud stakeholders need to model and evolve applications that leverage cloud computing for maximum value with minimum conflicts between the stakeholders.

The rest of this chapter is organized as follows. Section 4.1, presents an example of the cloud configuration space artifacts (i.e., what we are modeling). Section 4.2 provides an

overview of the StratusML modeling framework, its features, architecture, the methodology followed to build the framework, and its implementation. Section 4.3 dives deeper into the StratusML language: abstract syntax, well-formdness rules and semantics. Section 4.4 shows how the StratusML model concepts is mapped to the cloud specific platform constructs. Section 4.5 demonstrates the framework capabilities using an example. Finally, Section 4.6 summarizes the chapter.

4.1 An Example of the Cloud Configuration Space Artifacts

Deploying an application on a cloud platform requires specifying how the application service model will use the platform resources of that particular provider. This involves specifying (i) the service model, which defines the structure of the service itself in terms of the software modules that compose the service and how those modules are communicating, (ii) the runtime deployment configurations that specify how the modules of the service model are instantiated and replicated, and (iii) the behavior of the application at runtime under diverse conditions. This behavior is usually specified using an adaptation model, which is a set of rules and actions.

```
<ServiceDefinition>
   <WorkerRole name="ShoppingCartProcessing" vmsize="Small"=>
      <ConfigurationSettings>
         <Setting name.."DataConnection" />
      </ConfigurationSettings>
   </WorkerRole>
</ServiceDefinition>
```

Listing 4.1: Example of Service Definition File

```
<ServiceConfiguration>
   <Role name="ShoppingCartProcessing">
   <Instances count="2" />
      <ConfigurationSettings>
         <Setting name="DataConnection"
            value="UseDevelopmentStorage=true" />
      </ConfigurationSettings>
   </Role>
</ServiceConfiguration>
```

Listing 4.2: Example of Service Configuration File

```
<rules>
   <reactiveRules>
      <rule name="ScaleUp" description="Increases instance count" enabled="true" rank="3
       ">
         <when>
            <greaterOrEqual operand="Avg_CPU" than="75" />
         </when>
         <actions>
            <scale target="ShoppingCartProcessing" by="1" />
         </actions>
      </rule>
   </reactiveRules>
   <operands>
      <performanceCounter alias="Avg_CPU"
         performanceCounterName="\Processor(_Total)\% Processor Time" aggregate="Average
          " source="RoleB" timespan="00:10:00" />
   </operands>
</rules>
```

Listing 4.3: Example of Windows Azure Adaptation Model

Listings 4.1, 4.2 and 4.3 show respectively a service definition (service model), a configuration file (runtime deployment configuration), and an adaptation model that are used to deploy an application on Windows Azure platform. The syntax of these files conforms to the azure platform schemas. The service definition file in Listing 4.1 describes a cloud application that uses one *role*. A *role* in Windows Azure refers to a virtual appliance that is prepared with the required software stack to run a certain family of applications (i.e., web, or back-end). The service configuration file further specifies the service definition by assigning values to the configuration settings defined in the service definition file. For example, the service configuration in Listing 4.2 specifies the number of instances of the worker role. Finally the adaptation model in Listing 4.3 shows a reactive rule "ScaleUp"; which is used to scale the role "ShoppingCartProcessing" when the average CPU utilization exceeds 75%.

While this example is based on Windows Azure application packaging specifications; the information required to specify a cloud application deployment is essentially the same (e.g., the previously described role is equivalent to Amazon AWS beanstalk and GAE Module [68]). It is apparent from this example that managing these related artifacts and maintaining consistency between them requires a holistic view that integrates the artifacts. Moreover, in order to deploy the same application on multiple providers and facilitate its migration, there is a need to provide a layer of abstraction that captures the domain concepts then identifies the mapping between the domain independent concepts and the

deployment-description artifacts concepts of the different providers.

4.2 The Stratus Modeling Language Framework

This section presents the StratusML features, architecture and implementation.

4.2.1 The StratusML Features

StratusML is a modeling framework for cloud applications. It enables the design of high-quality distributed applications that are tailored to be deployed on the cloud. Through layers StratusML empowers the cloud stakeholders to view the models, each from its own perspective. This ability to separate between concerns makes working with complicated models more efficient. A layer can be turned on or off at anytime. This provides a holistic or partial view. StratusML supports visual modeling of adaptation rules and constraints, it also automates the generation of the corresponding artifacts for the target adaptation manager. StratusML supports the generation of complete platform specific artifacts based on template-based transformation.

Using templates and layered modeling, generating complete platform specific artifacts, and the ability to visually model adaptation rules and actions are the main distinctive advantages of StratusML over existing frameworks. However, StratusML provides several other features that make developing and managing cloud applications a seamless experience. StratusML allows users to (i) define a cloud application deployment model, and partition components into groups based on geolocation, scaling factors, and functionality, (ii) specify a cloud application configurations and adaptation rules (e.g., auto-scaling rules). (iii) select a cloud provider or create a custom one (iv) estimate the applications' running cost, and (v) use templates to transform the model into platform specific artifacts (e.g., Azure definition files).

4.2.2 The StratusML Framework

As shown in Figure 4.1, the StratusML framework covers model creation, validation transformation, and adaptation. Its architecture adheres to the Model-View-Controller (MVC) style. This aims at maintaining the consistency of the models, by performing the required model *transformations*, *validation*, and *analysis* whenever the models are updated. Both model *validator* and *editor* use the *StratusML meta-model*.

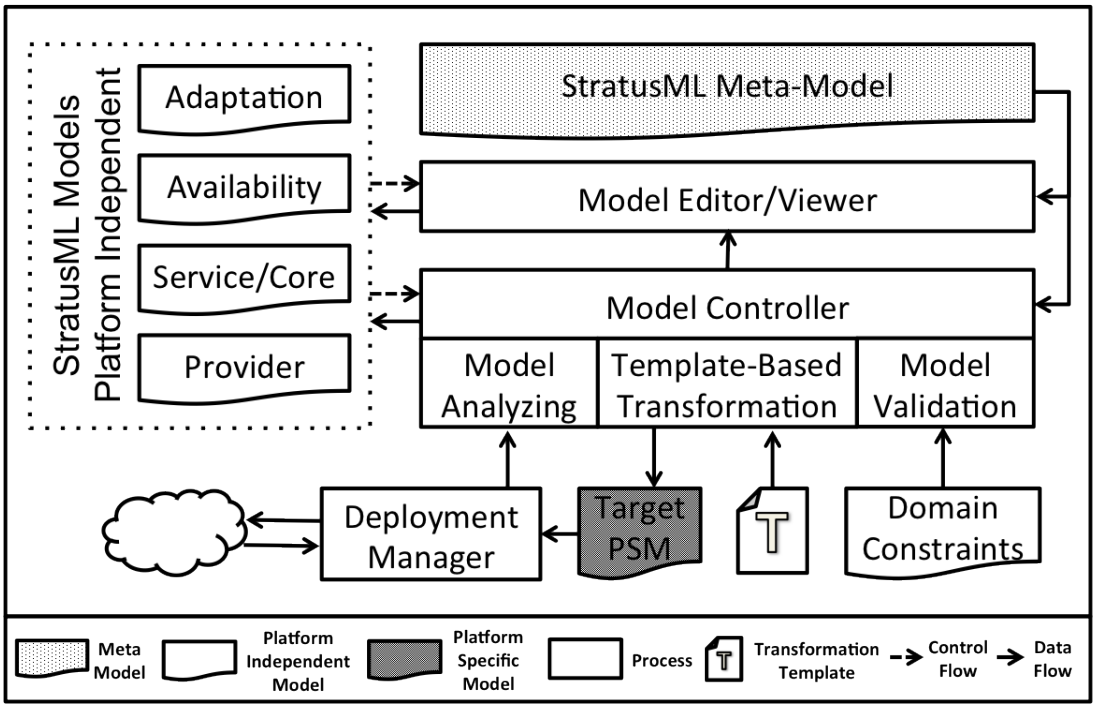


Figure 4.1: The StratusML Framework Architecture

The StratusML meta-model integrates five different meta-models with one *core* meta-model to address five different, but interleaved concerns of the cloud applications (i.e., *core* (service composition), *availability* (distibution and replication), *adaptation* (elasticity and dynamic behavior), *provider*, *performance* and *workflow*). The usage scenario explained in this chapter focuses on the integration of the core model with three of its perspectives (i.e., availability, adaptation, and provider), leaving the performance and workflow meta-models for the next chapter. The StratusML meta-models are explained in detail in Section 4.3 and made available online for reference in [64].

As shown in Figure 4.2, each of the StratusML meta-models has its own layer to be viewed on the modeling IDE. This integration of all the meta-models is what gives the cloud models their unique characteristics.

The StratusML framework supports two types of transformations, a *model transformation* that is used to specialize the Platform Independent Models (PIM) into Provider Specific Models (PrSM), and a *template-based transformation* [86], which is used to generate the Platform Specific Models (PSM) for the target platforms. Unlike other approaches that incorporate model-driven engineering to solve the vendor lock-in problem, StratusML em-

ploys template-based transformation that is supported with an automatic schema matching technique [68] to generate the different cloud model artifacts. The template-based transformation is a key feature in the StratusML framework. The transformation engine uses the validated StratusML model, and applies template transformation to it in order to generate a target model. The model encapsulates the essential data about the entities that need to be generated, while the template dictates the syntax of the target model. The template transformation engine produces the target model by replacing the template internal references with real data coming from the model according to the transformation rules specified in a procedural way in the template. Template-based transformation provides flexibility; as you can generate all types of models without changing the transformation engine, and portability; as the data model and engine are not touched. Moreover, the transformation syntax is simple, which facilitates reusability and productivity. A sample template that generates Windows Azure configuration from StratusML models is provided in Section 4.5.3.

4.2.3 The StratusML Methodology and Implementation

To build StratusML, we started from the domain artifacts of some popular cloud providers (i.e., Amazon AWS, Windows Azure and GAE), from which we created variability models to highlight their commonalities and differences. We built one *core* meta-model [66] that defines the abstract syntax, well-formedness rules, and semantics of the language.

We built several views around the core meta-model by extending it with the required components to address various concerns. The result was a huge meta-model, where each of the model elements inherits one of six meta-elements. Each one of the meta-elements represents one of the six different meta-models explained in Chapter 3. Each element in the meta-model is then mapped to one visual component to define the language concrete-syntax.

The StratusML framework has been realized as an extension of Microsoft Visual Studio 2012. In particular, we used the Microsoft DSL toolkit[37] to design the StratusML visual designers and Microsoft Text Template Transformation Toolkit (T4) to produce the different artifacts generators. Microsoft DSL model designer was used to define the different meta-models and then map each concept in these meta-models to its corresponding visual component shapes and decorators. A custom code has been used for creating the layering feature and to provide advanced validation. By combining T4 and StratusML meta-models, the user of the StratusML framework can easily generate artifacts for any target platform. Creating a new transformation template is no different than writing a simple procedural

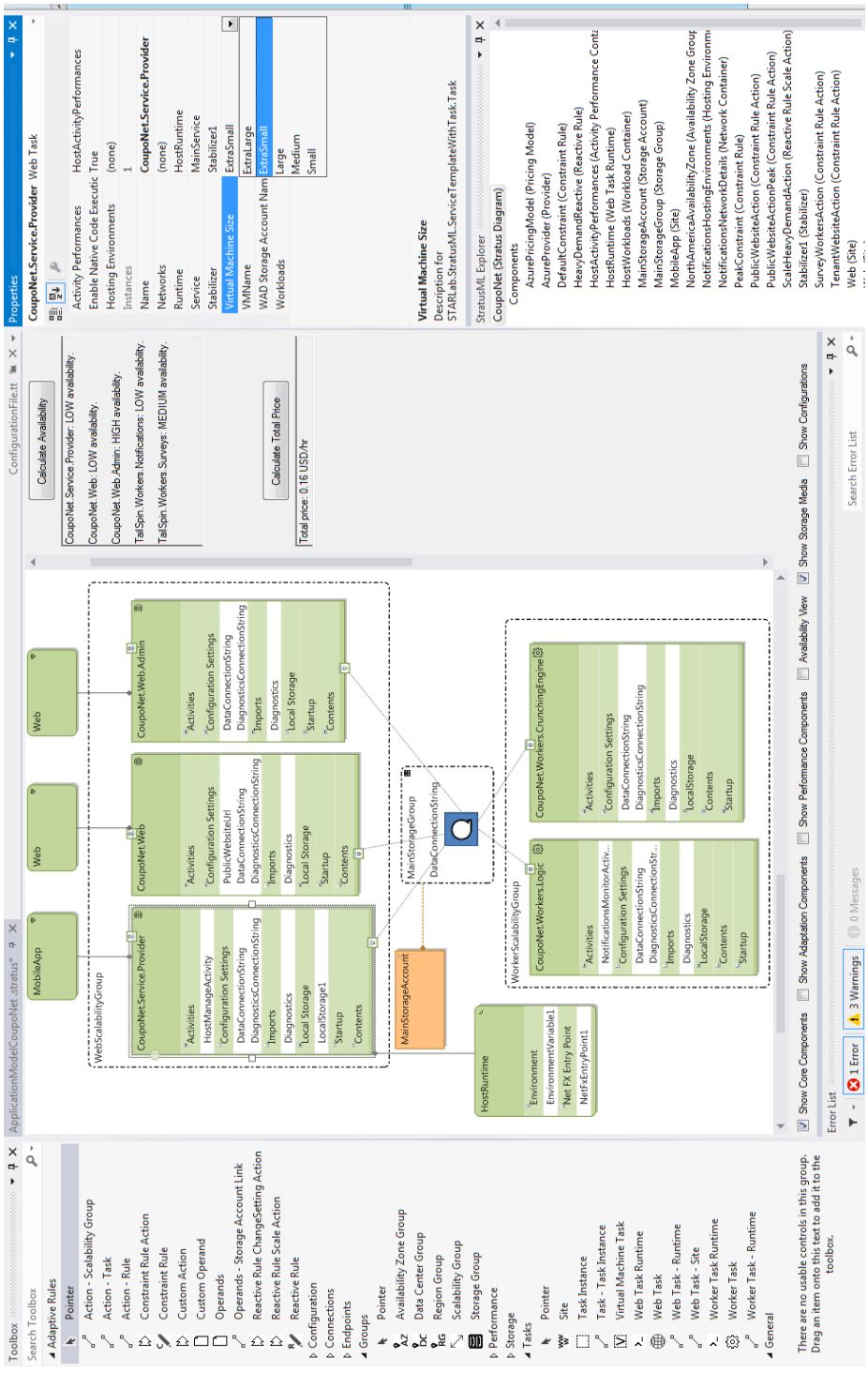


Figure 4.2: A Snapshot of the Design Window of the StratusML Modeling Framework

program. This alleviates the need to learn complex transformation syntax (e.g., XSLT). StratusML enables stakeholders to add new files of (.stratus) extension to build their model and create or utilize the provided transformation templates to transform the models into the desired output files.

Figure 4.2 shows a snapshot of the design window of the StratusML modeling framework. The tool box on the left side can be used to explore categories, and expose the different components. Users can drag and drop components as needed in the model design window, connect them and view and change their properties using the property window on the far right hand side. StratusML offers more than 60 domain concepts and more than 150 domain constraints. These constraints can be used to ensure model completeness and to steer the stakeholders' decisions to the right design. In the middle right-hand side of the figure, you can find the model information tab. The upper part is used to calculate the components availability (e.g., low, medium, high) based on the number of instantiated instances of each module and how they are distributed into different geographic locations. The lower part of the model information tab is used to estimate the price of the current configuration based on a selected provider. Finally, on the bottom of the figure is the views bar that allows the users to toggle between partial and holistic views that represent different aspects of the model. StratusML installation instructions and a detailed step-by-step usage scenario is provided in the StratusML Webpage [64], which also presents and explains the different meta-models and validation rules that represent the StratusML syntax and semantics.

4.3 The Stratus Modeling Language Specifications

This section focuses on the StratusML language specifications that enable capturing the cloud application configuration space. We used the process of cleaning and elimination to divide the Stratus meta-model into smaller meta-models to facilitate explaining them. The section starts by describing the StratusML syntax and semantics by means of class diagrams and natural language. Then, it gives examples of how the well-formedness rules have been created and documented through utilizing the Microsoft DSL constraint language.

4.3.1 The StratusML Abstract Syntax

Normally, a meta-model consists of abstract syntax, well-formedness rules and semantics. In this subsection, class diagrams are used to capture the syntax of the language and some

of the semantics that can be expressed through cardinality. Moreover, natural language is used to specify the semantics beyond cardinality.

4.3.1.1 The Service (Core) Meta-Model

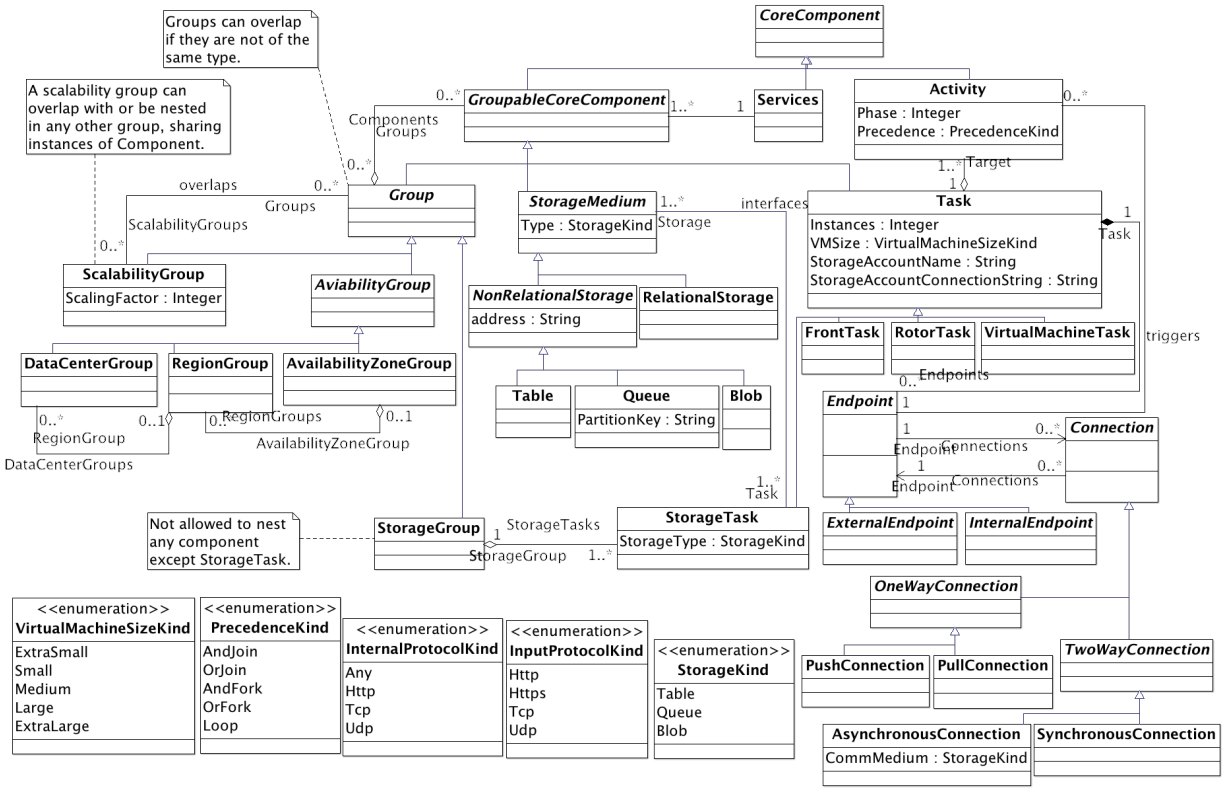


Figure 4.3: The Core View Meta-Model

The service meta-model allows developers to describe cloud services using platform independent components, providing a high level description of resource requirements independently from any target platform, and specifying operational requirements to be enforced and validated. More particularly, the *core* meta-model enables service developers to: (i) describe the structure of a cloud service composed of one or more *Tasks*, the types of tasks, and their relationships, (ii) cluster tasks into groups for easy management, (iii) assign each task a predefined or custom service template type, and (iv) assign availability level to every task (e.g., low, medium, high), which constrains how it should be instantiated and replicated in the availability model. A service template specifies the computation power,

memory and storage of the virtual machines available to host a task. Service template types (e.g., small, medium, large, extra-large) vary based on the provider. The templates are specified using the provider meta-model.

Figure 4.3 depicts the components of the core meta-model. The core meta-model is an abstraction layer on top of the PaaS layer. It is defined after manually inspecting three cloud platform application packaging requirements; namely, Amazon Web Services (AWS), Windows Azure, and Google Application Engine (GAE). For example, an *Elastic Beanstalk App* in AWS, a *Role* in Azure, and a *Module* in GAE all refer to a virtual appliance that is prepared with the required software stack to run a certain family of applications (i.e., web, back-end). Each of these concepts is abstracted with the concept of a *Task* in our meta-model.

The *core* meta-model captures all the concepts that are needed to specify a cloud configuration space, and facilitates dealing with inconsistencies between the different providers' file structures. The following is a brief description of the core meta-model components.

A *service* inherits the *CoreComponent*. It contains at least one *GroupableCoreComponent*. A service is one level higher than a *task* in granularity. The core meta-model facilitates creating several services within the same application. Each of them consists of several virtual appliances (a.k.a. *Tasks*), *Groups*, and/or *StorageMediums*.

A *Cloud Task* is a composable unit, which consists of a set of actions that utilize services to provide specific functionality to solve a problem. It is a mutated unit that can be copied to other virtual machines to allow horizontal and vertical scalability. When composed with other tasks, tasks should satisfy the following principals: statelessness, low coupling, modularity, and semantic interoperability. A task is semantically connected to other tasks in the cloud through the *roles* it plays in order to satisfy a specific business requirement, which is bound by obligations or *responsibilities*. A cloud task is uniquely identified by a global Dynamic Domain Name Service (DDNS) that can be assigned to a dynamic virtual IP address at run time. This makes the task highly available and fault-tolerant. Moreover, it allows the cloud application to be dynamically upgradable without any interruptions.

A *CloudTask* can be classified into:

(a) *CloudFrontTask*: An entry point to the cloud application that handles user requests, which are distributed by a load balancer. A *CloudFrontTask* must support the interactive request-response pattern. It is usually a web application (*WebTask*) hosted on the cloud datacenter where a web-server is always enabled. However, it can also be a web-service (*ServiceTask*), which is provided by a third party. A *ServiceTask* uses the Enterprise Service Bus (ESB) to discover and access remote or enterprise services.

(b) *CloudRotorTask / CloudWorkerTask*: This task runs in the background of the *CloudFrontTask* on the cloud datacenter. It is not directly accessible from outside the cloud datacenter. Mainly, it helps other tasks by performing a particular functionality. *CloudRotorTask* must support event-driven communication patterns. Grid computing tasks are common examples of *CloudRotorTask*s.

(c) *CloudPersistenceTask / CloudStorageTask*: The main role of *CloudPersistenceTask*s is to manage storage accounts. *CloudPersistenceTask*s manage the access control and login to cloud storages. A cloud storage (e.g., blob, table, queue) does not have any access control mechanism; it is the responsibility of the persistence task to provide the authorization and authentication services. *CloudPersistenceTask*s create containers, which are analogous to folders but with no nesting. Containers are accessible through a unique Uniform Resource Identifier (URI). *CloudPersistenceTask*s assign persistency to containers and give them a unique URI that is either privately or publicly accessible. The *CloudPersistenceTask* supports three main types of cloud storages that are reliable, can scale out, simple, and inexpensive. These types are: unstructured data (blobs), structured data (tables) and asynchronous messaging (queues).

i *Blob* : A blob is an unstructured large data file along with its meta-data. It can be stored as a sequence of blocks or pages. The blob is the simplest and the largest cloud storage unit. A cloud drive storage is a blob.

ii *Table*: A cloud table is a structured data file that is more complex than a blob. A cloud table is much simpler than a relational database (RDB) table. It is more like a datasheet. This simplicity makes it more suitable for massive scalability. A cloud table consists of a set of entities and its associated properties. It uses two types of keys: partition keys and row keys. A cloud table does not support SQL queries. It has no schema, and it uses optimistic concurrency for updates and deletions.

iii *Queue* : A queue is scalable message storage that supports the polling-based model used in message passing between tasks. A message can be stored for long periods (i.e. days) before it is read and then removed from the queue. A cloud queue is different from a conventional queueing system. A cloud queue must support fault-tolerance. Unlike conventional queues, a read message does not delete the message from the queue. Instead, the message is set into the hidden mode until it is successfully processed. It is the responsibility of the processing task to delete the message. The queue is the main communication mechanism between *CloudFrontTask*s and *CloudRotorTask*s, which makes it one of the most frequently used design patterns in the cloud. This design pattern not only reliefs the end-user from waiting for a long time until a task processes the message, but also makes scalability easier.

An *activity* is a sub-process of a *task*. Each *task* has at least one activity. *Activities* are essential for performance analysis. *GroupableCoreComponent*s can be nested so that certain properties apply to all of them. A *Group* is a container that applies the composite pattern; it inherits the *GroupableCoreComponent* and can contain many *GroupableCoreComponent* elements at once.

The core meta-model distinguishes three types of groups: *ScalabilityGroup*, *StorageGroup*, and *AvailabilityGroup*. A *ScalabilityGroup* can overlap with or nest in other groups that are regulated by adaptation rules. Components in the same *ScalabilityGroup* can be scaled via different *ScalingFactors*. A *StorageGroup* can only nest *StorageTasks*; although it inherits *Group*, it may not nest any other *GroupableCoreComponent*. This is enforced by a validation constraint. Finally, an *AvailabilityGroup* nests components, which need to be hosted for the same location. It is a superclass for the three geolocation groups (i.e., Zone, Region, and DataCenter). Availability groups will be discussed in Section 4.3.1.

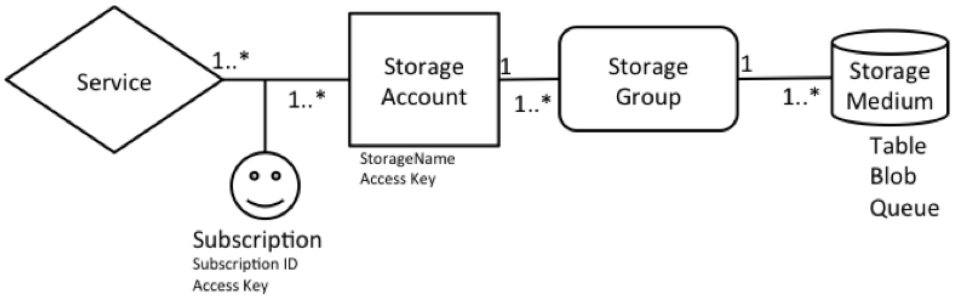


Figure 4.4: Storage Management in StratusML

Figure 4.4 shows how storage management is addressed in StratusML. StratusML abstracts the way cloud storage management is handled by different cloud platform providers. In a nutshell, a *StorageGroup* is a container of storage mediums; it contains at least one storage medium. Each *StorageGroup* has a storage account that specifies the access mechanism and credentials (i.e., storage account ID and account key) to access the *StorageGroup*. A service can have multiple storage accounts. This flexibility can be leveraged to address locality considerations, and reduce traffic overhead by keeping the data and processes close to each other.

By organizing the storage management service in a hierarchical fashion, cloud platform providers can support a wide verity of data multi-tenant architectures. Examples of these architectures include: one subscription per tenant, one subscription for a group multiple tenants, one storage account per tenant, one storage account for group multiple tenants,

one table per tenant, one table with one partition key per tenant, and one container per tenant.

Finally, relationships between *Tasks* can be determined by *EndPoint*s. *EndPoint*s are ports through which a *CloudTask* can connect to other tasks or to the environment. Each *Task* has one or more *EndPoint*s. An *EndPoint* can be classified based on several criteria. Whether it is publicly visible (external) or only accessible within the Cloud Application (internal), load balanced at the network level or not, or whether it allows *inbound* or *outbound* communication. An *inbound EndPoint* is bound to one or more activities that represent the interaction points with the task. Each *EndPoint* uses an access mechanism, which uses a semantic interaction pattern for the coordination of message exchange. These patterns are based on specific protocols that determine the syntax and semantics of the messages that are exchanged between the two communication parties. Message Exchange Patterns (MEP) can be classified into two main categories, one-way or two-way. The one-way MEP is usually referred to as the event driven MEP, or publish subscribe (pub/sub), in which the participating parties are not fully aware of each other. A temporary storage in the form of a queue is usually used to accomplish this. One party will push a message, and the second will pull it from the queue. This is one of the common communication patterns between *CloudFrontTask*s and *CloudRotorTask*s. On the other hand, the two way MEP is usually referred to as request/response MEP. It can either be a synchronous (blocking) or asynchronous (non-blocking). The two way MEP is an interactive communication that is usually needed when you have direct interaction with the user. *CloudFrontTask*s must support this type of interaction with the application user.

4.3.1.2 The Adaptation Meta-Model

The adaptation meta-model provides the components required to specify the adaptation rules and actions for each task or group of tasks in the core model. This enables many dynamic features of the system, such as elasticity and security. It also helps administrators ensure that the system continuously satisfies operational requirements (e.g., minimum operational cost, high performance, and high availability). A rule can control the number of virtual appliance instances, or enable a security guard or policy. A rule uses a set of key performance indicators as *operands*. These are usually collected though instrumentation and trace summarization.

Figure 4.5 shows the adaptation meta-model. Each component or group of components is associated with a set of *actions*. The meta-model enables two types of actions, *predefined* and *custom* actions. While predefined actions are known ahead of time (e.g., scaler actions),

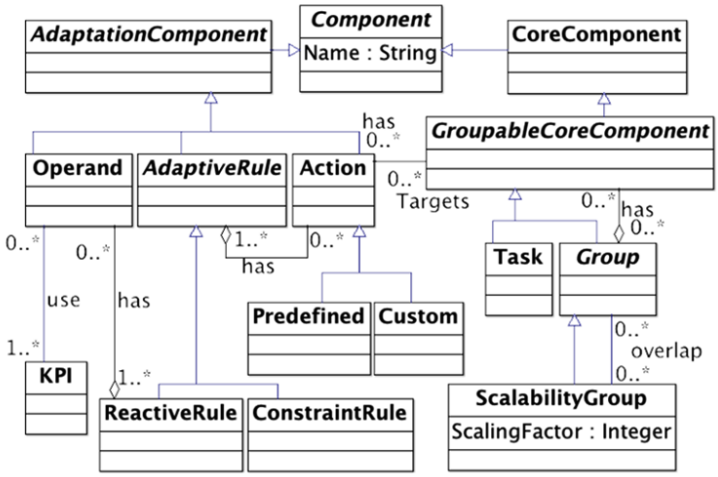


Figure 4.5: The Adaptation View Meta-Model

custom actions allow assigning external processes. An *action* is triggered based on a constraint or reactive *adaptation rule*. A *ConstraintRule* is a predefined (static) constraint, such as the minimum or maximum number of instances allowed at a certain time. A *ReactiveRule* is dynamic rule. It is based on evaluating runtime environment parameters against a set of key performance indicators (e.g., CPU utilization, queue length, response time).

4.3.1.3 The Availability Meta-Model

The availability meta-model provides the components required to specify the configuration parameters of the tasks that can be specified and modified at runtime without need to stop the running application or redeploy it. Example of these parameters include: the size of the virtual machine (*VMSize*), number of instances (*numberOfInstances*), storage size (*DBSize*), *bandwidth*, the location (*locationProximity*) where the task instances will be instantiated and whether they belong to the same affinity group or not. This information account for the dynamic aspects of cloud application such elasticity (i.e., the ability to scale in and out).

Figure 4.6 shows the availability meta-model. An availability zone refers to the distinct

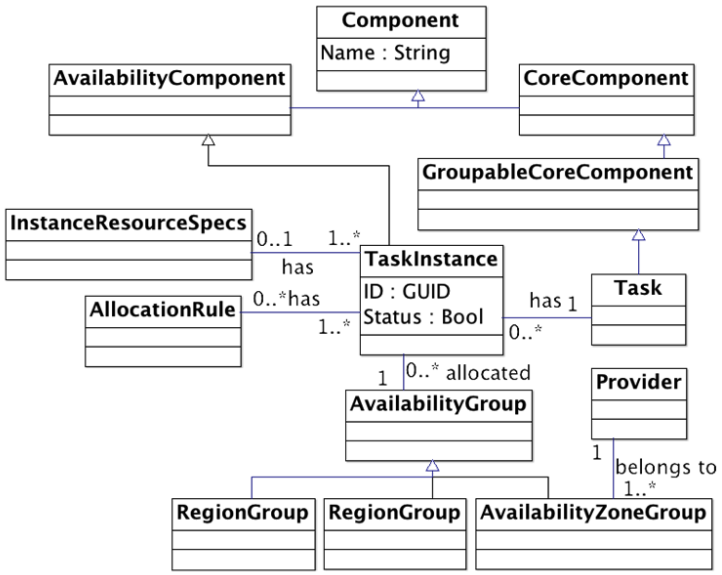


Figure 4.6: The Availability View Meta-Model

Table 4.1: Availability and Fault Recovery Levels

Specified Requirements (Core)		**Validation Constraint to be Checked**
Availability	**Fault Recovery**	
Very Low	N/A	A task has one instance
Low	Fast	A task has two instances in the same region
Medium	Average	A task has two instances distributed into two different regions
High	Slow	A task has two instances distributed into two availability-zones
High	Fast	A task has three instances, two in the same region, and one in a different availability-zone
Very High	Fast	A task has four instances, two in the same region, one in different region but same zone, and one in different availability zone

physical location of the available hosting data-centers of a provider. An *AvailabilityZone-Group* nests components that need to be hosted in the same availability zone. It contains several *RegionGroups* that vary based on the provider. A *RegionGroup* nests components that need to be hosted in the same region and contains many *DataCenterGroups*. Lastly, a *DataCenterGroup* nests components that need to be hosted in the same datacenter. Selecting a specific datacenter for an application is still not supported by most providers.

For each *task* in the core model, the availability meta-model allows administrators to specify the number of instances (*TaskInstance*) to instantiate (replication), and their distribution to different geolocations (*AvailabilityGroups*). Recall that in the core model a service developer specifies a provider for each task, and the required availability and fault recovery levels. The availability model assigns one of that provider's available geolocations for each task. This enables hybrid cloud deployments, where an application can span multiple providers. Once the administrator creates the availability model, and before the actual artifacts required for packaging and deploying the application on a target platform are generated, a set of constraint rules are validated to ensure that the availability model conforms with requirements. Table 4.1 shows a list of availability and fault recovery objectives, and their corresponding validity constraints.

4.3.1.4 The Provider Meta-Model

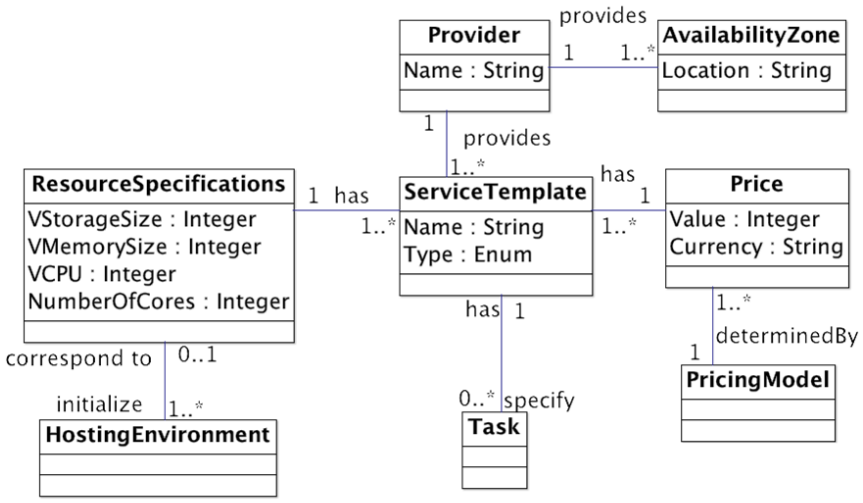


Figure 4.7: The Provider View Meta-Model

The *Provider* meta-model aims to model different providers' templates, offers and costs. Figure 4.7 depicts the *Provider* meta-model. Each *provider* provides a list of *availability zones*, and *service templates*. Availability zones represent the physical locations of provider data-centers, while service templates capture different resource specification bundles (i.e., CPU speed, number of cores, memory size, disk space) that serve as templates for service

tasks. The ability to specify resources using predefined templates is a turning point in automating performance analysis. This meta-model enables reusability at the resource model level, providing ready to use templates that represent actual cloud provider templates. Each template has a *price* that captures the cost of the resource configurations per bundle. These prices usually depend on a provider's *pricing models*, contract period and terms (e.g., free, weekly, pay-as-you-go).

4.3.2 The StratusML Model Validation

There is always a need to check the correctness and completeness of the models created or generated using a DSML. Validating a model includes checking:

(a) **The model structural constraints and well-formedness rules**: Structural and well-formedness rules can be specified by superclasses along with the multiplicity and type information expressed in the language meta-model. These are *hard* constraints that are normally verified automatically by the modeling framework. An example of a structural constraint is setting the lower cardinality of a component to one in the meta-model. This constraint enforces that a model must be created with at least one component of this type (i.e., an empty model is invalid).

(b) **The model semantic constraints**: Semantic constraints can be checked by defining *invariants* to ensure model correctness and domain conformance. These *invariants* can be implemented as soft or hard constraints (e.g., each task must have a unique name). Moreover, depending on the validation context there may also be a need to define *pre-conditions* and *post-conditions* as in the case of *method* validation to ensure context state validity before and after the execution of the method.

(c) **The transformation constraints**: These constraints are used to verify the correctness and completeness of the information needed to generate the target model before a transformation.

StratusML utilizes Microsoft DSL framework to define the validation rules required to ensure that the specified model satisfies the basic domain requirements and provides the information required to generate the target platform specific artifacts. Microsoft DSL automatically generates the validation methods for the structural constraints (e.g., checking minimum multiplicity based on the defined meta-model). However, for semantic and transformation constraints, the validation constraints must be explicitly defined by adding validation methods to the domain classes or relationships of the DSL.

In Microsoft DSL, validation constraints can be classified into hard and soft constraints. Hard constraints cannot be violated by the user (i.e., it prevents the user from making modeling mistakes). In contrast, soft constraints can be violated, but still create warnings and errors to guide the user to the correct decisions. For example, a constraint that enforces that no two tasks in a model should have the same name could be implemented either as a hard constraint or as a soft constraint. If it is implemented as a hard constraint, then a user will not be allowed "at any point in time" to create a task that has the same name as another task. However, if it is implemented as a soft constraint, then a user is allowed to create a task with the same name as another task, but an error will be generated to instruct the user of the problem. Hard and soft constraints provide different usability experiance. While hard constraints make the design environment rigid, lots of soft constraint may result in deferring errors discovery to the time of saving the model. This could lead to error accumulation and hence user frustration. StratusML mixes between soft and hard constraints. A StratusML model cannot be saved if errors still exist. For example, in the case of CoupoNet each of the web tasks external endpoints must be secure. The model validator ensures this by checking if each of these endpoints is using SSL for communication and that an SSL certificate is assigned for each external port. If an external endpoint is not using SSL or does not have a certificate, an error message will be displayed and the model will not be saved unless the error is resolved. In addition to hard and soft constraints, StratusML also differentiates between validation rules based on generality into domain specific or generic rules. Both domain specific and generic rules are used to specify semantics of the cloud concepts beyond the meta-model, such as the rules of group nesting. Domain specific constraints are classified in StratusML based on the layer of the concepts it validates (e.g., availability, adaptation). On the other hand, generic rules are the rules used to enhance the modeling experience in general, such as those that check for name duplication.

Each validation rule is applied to a *scope* (i.e., class, relationship). The rule scope specifies the *context* (i.e., name of the class or relationship) that the rule is validating and its attributes or methods. Running a validation, either by a user or under program control, executes some or all of the validation methods. Each method is applied to each instance of its class. Moreover, there can be several validation methods in each class. Validation is executed as a response to an event *trigger*. The types of triggers supported in Microsoft DSL and used in StratusML are: File *Open*, *Load*, *Save*, and *Custom* triggers. A *Custom* trigger is normally implemented as an event handler. Each rule validation method is implemented using a list of invariants, which are the conditions the rule validates. Whether the rule invariant is *valid* or *invalid* a set of actions can be activated. The actions can be *error* or *warning* messages, or custom actions.

The following are two examples of the validation rules used in StratusML (i.e., a soft and a hard rule). The complete list of the validation constraints and the validation code can be found in the StratusML webpage [64].

Example 1 - A Hard Validation Rule: Figure 4.8 is an example of a validation code for a hard rule. The first line represents the context where the rule is applied (i.e., *Group* class). As shown in lines three and four, the rule is triggered in response to a custom event, which is a model element creation of the type *GroupableComponent*. Line five is the invariant, which validates that if the created element has a *Group*, and the element is of type *Storage* then the *Group* must be of the type *StorageGroup*. As a result, if the invariant is invalid then an error will appear and the component created will automatically be deleted. This is an example of hard constraint, as it prevents the user from creating the component by deleting the created component, in order to preserve the initial state of the context .

```
1. [RuleOn(typeof(GroupableComponent))]
2. public sealed class OnlyStorageInStorageGroupRule : AddRule{
3.   public override void ElementAdded(ElementAddedEventArgs e){
4.     GroupableComponent component = e.ModelElement as
       GroupableComponent;
5.     if (component.Group != null && !(component is Storage) &&
           component.Group is StorageGroup) {
6.         Helpers.ShowErrorMessage
           ("Only a storage can be nested in a storage group.");
7.         component.Delete();}}} //reverse action
```

Figure 4.8: Example of Validation Code for a Hard Rule

Figure 4.9 is the documentation of the rule in Listing 4.8. In which, we specify the rule name, description, context, trigger event, invariants, and pre- and post- conditions. StratusML rules have been documented by utilizing the same template in Figure 4.9. Notice that the rule *pre-conditions*, *invariants* and *post-conditions* are declarative (i.e., do not change the rule context state). StratusML does not support validation rules with imperative actions with the exception of hard rules that validate the creation of components. These rules support *delete* actions to reverse the creation action effect in order to maintain the initial state of the context .

Example 2 - A Soft Validation Rule: Figure 4.10 is an example of a soft validation rule that is applied to the adaptation *action* concept. The rule is triggered by the model's *save* event. The rule validates that an action is associated with a *Task* or *ScalabilityGroup* as a target. If this invariant is invalid, an error message will be generated. Moreover, for

Rule:	`OnlyStorageInStorageGroupRule`
Decryption:	checks that a `StratusDiagram` class contains at least one `Task` class.
Context:	**<class>** `GroupableComponent`
Trigger Event:	**<custom> adding** `GroupableComponent`.
Pre-conditions:	`GroupableComponent` **exists**, `GroupableComponent.Group` **exists**
Invariants:	¬ ((`GroupableComponent.Group` **is** `StorageGroup`) ∨ (`GroupableComponent.Group` **is** `StorageGroup`) ∧ (`Component` **is** `Storage`)) ⇒ **valid.**
Post-conditions:	**On invalid** **display <error>** *"Only a storage can be nested in a storage group"*. **delete** `GroupableComponent`.

Figure 4.9: Example of Hard Validation Rule

Rule:	`ActionMustHaveAtLeastOneTargetRule`
Decryption:	checks if an `Action` class references at least one `ScalabilityGroup` or `Task`
Context:	**<class>** `Action`
Trigger Event:	**< Save>**
Pre-conditions:	`Action` **exists**
Invariants:	`ReactiveRuleScaleAction.TaskTargets.Count` > `0 ||` `ReactiveRuleScaleAction.ScalabilityGroupTargets.Count` > 0 ⇒ **valid**
Post-conditions:	**On invalid** **display <error>** *"Actions must have at least one target (task or scalability group)"*. **Set Focus** `<Action> A`

Figure 4.10: Example of Soft Validation Rule

better user experience a *set focus* action will be used to direct the user to the *component* (i.e., the action) that generated the error.

4.4 Mapping the Stratus Meta-Models to Cloud Specific Platforms

The Stratus meta-model presented in Section 4.3.1 is platform independent. It provides a set of generic concepts, vocabulary, and rules for specifying the configuration space of an

application apart from the target cloud platform vendor constraints. In this section, we provide a mapping between the StratuML elements and their corresponding elements in Windows Azure, Google App Engine and Amazon AWS. The later platforms were chosen based on their popularity (i.e., they are major players in the cloud space with hundreds of thousands of customers), their good documentation and the fact that they provide a set of development tools at the PaaS level. They provide a diverse range of services including but not limited to: compute, storage, security, networking and content delivery, databases (relational, NoSQL, columnar, and caching), application services (queuing, orchestration, etc.), and deployment and management services.

Table 4.2 shows a high level mapping between the StratusML meta-models and their corresponding artifacts in Windows Azure, Google App Engine and Amazon Web Services. For example, most model constructs of the StratusML core meta-model can have correspondences in both Azure service definition (*.csdef) and configuration files, in GAE deployment descriptor and configuration files and in AWS Elastic Beanstalk and EC2 templates.

Table 4.2: Mapping Meta-Data Artifacts

StratusML Artifact	Azure Artifact	GAE Artifact	AWS Artifact
Core Model	Service Definition File & Service Configuration File	Deployment Descriptor & Configuration File	Elastic Beanstalk Template & EC2 Template
Adaptation Model	WASABi Adaptation rules	Configuration File	AWS AutoScaling Template
Provider Model	Azure Provider Offering	GAE Provider Offering	AWS Provider Offering

As shown in Table 4.2, there is no one-to-one mapping between StratusML models and providers' artifacts. This is because different providers use configuration artifacts that have different file structures. More about this problem and an example of these file structures is provided in Chapter 6.

In this section, we focus on how the different concepts within these files are manually mapped to the StratusML meta-model. Chapter 6 shows how such mapping can be simi-automated. This section starts by a sample mapping of the StratusML core model elements, followed by a sample of the provider meta-model concepts, then a sample of adaptation model concepts. Note that the availability model is an instance model of the core service model. Hence, no mapping is needed.

Table 4.3: Mapping of Platform Specifications

StratusML::Core	Azure Artifact	Google GAE Artifact	Amazon AWS Artifact
Compute			
Core:Service	Azure Service	GAE Application	AWS Application
Core:Task	Role	Module	Elastic Beanstalk Task
Task:CloudFrontTask	Web Role	Frontend Module	Elastic Beanstalk WebApp
Task:CloudRotorTask	Worker Role	Backend Module	Elastic Beanstalk Background Task
Storage			
StorageTask:StorageAccount	Azure Storage Account	Google Cloud Storage Account	AWS Storage Gateway Service
StorageMedium: NonRelationalStorage:Table	Azure Table	Datastore	Amazon SimpleDB
StorageMedium: NonRelationalStorage:Queue	Azure Queue	Task Queue	Simple Queue Service Simple Notification Service
StorageMedium: NonRelationalStorage:Blob	Azure Blob AzureDrives	Blobstore	Amazon S3 Elastic Block Store (EBS)
StorageMedium: RelationalStorage	SQL Azure	Google Cloud SQL	Amazon RDS
Configuration and Communication			
Service:ServiceAccount	Azure Account	GAE Account	AWS Account
Task:DNS	Azure DNS	Cloud DNS	Route53
Task:LoadBalancer	Azure Traffic Manager & Load Balancer	Compute Engine Load Balancer	Amazon Elastic Load Balancer
Task:Endpoint	Role:Endpoint	Module:Endpoint	BeansTalk:Endpoint

4.4.1 Mapping the Core Model Components

Table 4.3 shows a sample of a mapping between the StratusML core meta-model elements and it corresponding platform specific elements from Windows Azure, Google App Engine, and Amazon Web Services. Generally speaking, the core cloud platform components can be divided into compute elements, storage elements, configuration elements and connectors. We start by mapping the main building blocks: the cloud compute elements and persistence elements, followed by the elements that create the connections between the building blocks.

Compute Elements

As explained in Chapter 3, a StratusML *Service* is composed of *Tasks*. A *Task* is the main compute element, it refers to a composable unit that represents a virtual machine image, microservice or container. A StratusML *Task* is best alligned to Windows Azure *Role*,

GAE *Module*, and AWS *BeansTalk*. A *Role* is a specialized instance of a virtual machine that is customized for specific tasks and it is the main composition unit of an *Azure Service* [108]. A *Module* refers to a component of an application that provides a single service or configuration. An *App Engine Application* is made up of one or more modules. Each module consists of source code and configuration files to handle a specific task[58]. On the other hand, a Beanstalk is the main composition unit of *AWS Application*. A Beanstalk is a fully managed application container service that provides built-in deployment features for a variety of application frameworks [5].

StratusML offers two types of Tasks: *FrontTask* and *RotorTask* (i.e., also referred to as *Worker Task* in the StratusML modeling framework). A *FrontTask* is best mapped to a *Web Role* in Azure , a *Frontend Module* in GAE, or a *Elastic Beanstalk WebApp* in AWS. These concepts describe an task that is customized for web application programming. On the other hand, a *RotorTask* is best mapped to Azure *Worker Role*, GAE *Backend Module*, or AWS *Elastic Beanstalk Background Task*, which all refer to long running background tasks that dont require user interaction.

Persistence Elements

In addition to compute services, the StratusML core meta-model provides persistence building blocks that enable specifying the application database and storage options. Table 4.3 shows a mapping of these components to the different platform providers.

StratusML storage components include the *StorageAccount* in addition to building blocks that represent the configuration constructs of both the non-relational storages and relational database. In StratusML, each *StorageTask* manages the *StorageAccount* and offers options for recovery and backup. A *StorageAccount* maintains the information required to access the storage medium. A *StorageAccount* is mapped to *Azure Storage Account*, *GAE Cloud Storage Account* and *AWS Storage Gateway Service*.

The StratusML non-relational storages (i.e., Table, Queue, Blob) are mapped to Azure (Table, Queue and Blob), GAE (Datastore, Task Queue, and Blobstore) , or to AWS (SimpleDB, Simple Queue, and Simple Storage (S3)) respectively. Finally, The StratusML RelationalStorage is mapped to SQL Azure, Google Cloud SQL and Amazon RDS respectively.

Configuration and Communication

In addition to the compute and storage components, StratusML provides a set of miscellaneous components that cover aspects such as security, access control, networking and content delivery. This section covers some of these components.

The *ServiceAccount* component specifies the subscription information of the user on the deployment portal. The information specified using this component is mapped to its equivalent account information in Azure, GAE and AWS.

A Task *DNS* specifies the settings for a domain name server (e.g., the name and IP address of the DNS server). A StratusML *DNS* is mapped to *Azure DNS*, *Cloud DNS* and *Route53* components in Azure, GAE and AWS respectively. Similarly a Task *Load Balancer* specifies the configuration setting used to divide traffic (workloads) between Taks. StratusML *Load Balancer* is mapped to Azure, GAE and AWS *Load Balancer*.

Endpoint Protocols refer to the communication protocols used in cloud applications. Cloud *Tasks* can communicate with internal tasks (within the same cloud), with external services and cloud applications, or with end users depending on the type of the task and the role it plays. In Windows Azure, WebRoles use HTTP/HTTPS in order to communicate with end users. WorkerRoles can be accessed using HTTP/HTTPS protocols as well as the TCP protocol. WebRoles and WorkerRoles communicate with each other via message passing using azure storage services in a *Restful* way. The same concepts are applied to Google App Engine and AWS. Both uses HTTP/HTTPS protocols for incoming and outgoing communication, REST over HTTP for communicating with web-services, as well as queues to schedule tasks for execution, based on the requests received from the FrontEnd Modules and WebApp Beanstalks

4.4.2 Mapping Dynamic Behavior (Adaptation Meta-Model)

Adaptation capabilities allow you to automatically change the application structure or configuration at runtime. Today, most cloud platforms offer variations of autoscaling services that enable adding or removing instances to handle sudden changes in traffic and reduce cost. In this section we map the StratusML adaptation meta-model elements to the autoscaling templates provided by Azure, Google and Amazon platforms.

Table 4.4 provides a summary of this mapping. For example, a StratusML *AdaptiveRule* is mapped to a *ScalingRule* in Windows Azure Autoscaling Application Block (WASABi), to *Autoscaling Policy* in GAE Managed Instances Configuration, or Scaling Policy in AWS

Table 4.4: Mapping of Adaptation Rules

StratusML Adaptation	Azure WASABi	GAE Managed Instances Configuration	AWS Autoscaling Template
AdaptiveRule	ScalingRule	Autoscaling Policy	Scaling Policy
ReactiveRule	ReactiveRule	Autoscaling policy based on target monitoring metrics	Dynamic AutoScaling
ConstraintRule	ConstraintRule	Scheduling Policy	Scheduled AutoScaling
Action	ScaleAction	Instance Group Updater (startup script)	ScaleEvent
Scalability Group	ScaleGroup	Managed Instance Groups (Target Pool)	AutoScaling::AutoScalingGroup

Autoscaling template. All the aforementioned frameworks support both reactive and constraint rules. Also, they allow creating custom actions and setting a scalability group as a target. However, as in previous mappings, these platforms use different terminologies as shown in the table, and may have different constraints. For example, in GAE there is a cap limitation of 100 instances per module. In contrast, the limit in Azure is 50 instances.

4.4.3 Mapping Providers' Specifications (Provider Meta-Model)

Recall that each *Task* in the StratusML core model is assigned a *service template* that specifies the computation power, memory and storage of the virtual machines available to host a task. This section maps the StratusML service templates to its equivalent provider templates.

Table 4.5: Mapping of StratusML Privider Service Templates to Existing Cloud Providers' Offerings

StratusML:Provider :ServiceTemplates	Window Azure	Google Cloud Engine	Amazon EC2
Small	Medium (A2)	n1-highcpu-2	t2.medium
Medium	Large (A3)	n1-highcpu-4	c3.xlarge
Large	Extra Large (A4)	n1-highcpu-8	c3.2xlarge

Table 4.5 shows a sample mapping of comparable small, medium and large instances. While Table 4.6 shows the hourly price (the cost of using a VM with a specific service

Table 4.6: Comparing The Service Templates's Pricing and Resource Specifications

	Hourly Price	CPU	Memory	Storage
	Small			
Window Azure	$0.12	2 Cores	3.5 GB	Off Instance Only
Google Cloud Engine	$0.088	2 Cores	1.8 GB	Off Instance Only
Amazon EC2	$0.068	2 vCPU	4 GB	Off Instance Only
	Medium			
Window Azure	$0.24	4 Cores	7 GB	Off Instance Only
Google Cloud Engine	$0.176	4 Cores	3.6 GB	Off Instance Only
Amazon EC2	$0.21	4 vCPU	7.5 GB	2 x 40 GB SSD
	Large			
Window Azure	$0.48	8 Cores	14 GB	Off Instance Only
Google Cloud Engine	$0.352	8 Cores	7.2 GB	Off Instance Only
Amazon EC2	$0.42	8 vCPU	15 GB	2 x 80 GB SSD

template under the designated provider) and hardware specifications (i.e., number of cores, memory size, disk space) for each of the service templates. For example a *small* service template in StratusML is matched with a *medium (A2)* instance in Azure, a *n1-highcpu-2* instance in Google and a *c3.large* and *t2.medium* instances in Amazon. Each of these instances has two core CPUs, a main memory between 1.8 and 3.5 GB. Moreover, they use off-instance storage.

Comparing the performance for running a service on different providers, given only the providers information is impossible [36]. While the aforementioned service templates are equivalent in terms of hardware specifications, it is difficult to infer that they can guarantee the same performance. This makes finding the right resource specification match a daunting task. In our work we adopt the results obtained by Cloud Harmony Inc. benchmark [36] for service template mappings.

4.5 Case Study: Demonstrating the Stratus Framework Capabilities

A platform provider (e.g., Windows Azure) should empower the developers to build high quality applications. A modeling language (e.g., StratusML) assists stakeholders to make the right decisions to fully utilize the platform. This section shows, by example, how

StratusML contributes to solve the typical challenges that face any organization adopting cloud computing.

4.5.1 The CoupoNet Scenario

Let us consider CoupoNet; a fictitious startup company that offers coupon services on the cloud. The software is a multi-tier and multi-tenant application that works as follows: CoupoNet tenants obtain a free trial or paid subscription that allows them to design and post coupons and publish them based on the target customers' geolocation. The application stores the buying and selling data and performs sophisticated analytics to rank and position the offers, provides statistical data to the subscribers, and analyzes the users interactions to dynamically update the CoupoNet business model. The CoupoNet team decided to host their application on the cloud to harness its benefits; cut initial costs, provide reliable services, and minimize administrative and configuration tasks. As a start-up, there are many challenges to be met. The CoupoNet team is unsure how many resources they need. The coupons market is vibrant; coupons can be seasonal, time limited and susceptible to the slash dot effect; no one can predict when a coupon will become popular. The CoupoNet site should be able to respond quickly to increasing demand. CoupoNet is also unsure of how to distribute the services geographically to insure the highest availability and minimize the traffic overhead, or how to assure security and compliance. Cost wise, CoupoNet is unsure which provider to select; while currently provider X provides the cheapest services, CoupoNet developers are familiar with provider Y's technologies. Moreover, CoupoNet expects a more competitive offer next year from CloudKick, a third provider. From a technical point of view, CoupoNet's architect designed the application to ensure components have lowest coupling and highest cohesion. However, at the time of the deployment, the system administrator, who is unaware of the architect design decisions, may redistribute the components to minimize cost and initialize them with arbitrary ratios based on his best estimation. As the demand fluctuates the administrator needs to update the deployment model and reconfigure the different services.

4.5.2 Using the StratusML Framework Modeling Features

This section uses the CoupoNet example to demonstrate the capabilities of StratusML. Table 4.7 summarizes how StratusML can be used to address the CoupoNet team challenges. The first column shows the challenges that the CoupoNet team faces. The second column shows the StratusML layer that is used to address each challenge, while column

Table 4.7: Addressing CoupoNet Challenges

CoupoNet Challenges	Modeling Layer	StratusML Solutions
Model multi-tenant applications	Core	Provide different groups (i.e., storage, availability, and scalability groups) that address multi-tenancy by applying different service and data partitioning strategies.
Model multi-tier applications	Core	The core meta-model provides platform independent task-templates for frontend, backend and cloud-storages. It also provides connections to describe the different interactions between *Tasks*.
Communicate architectural decisions to administrators	Core	StratusML groups can overlap with each other. Using these groups, architects can ensure that their original decisions, which aim to reduce coupling and increase cohesiveness, are maintained. Grouped components will always stay, migrate and scale together.
Distribute services into multiple geographic locations	Availability	Provide availability groups that facilitate managing the service instances locations and counts.
Uncertainty of required resources	Adaptation	Facilitate modeling for adaptation rules and actions with the focus on scalability actions. A user can specify constraint and reactive rules, and associate them to task or scaling groups. The framework generates the required rule-based configurations to automate resource provisioning.
Evaluate different provider offerings	Provider	A user can select one of the available providers or create a custom provider. The system will estimate the cost of deploying the application on the selected provider.
Migrate between different providers	Provider	Provide a set of templates that can be customized to any platform in order to automatically generate all the target platform artifacts.
Minimize administration and configuration tasks and model co-evolution	Provider	Provide a set of templates that can be customized to any platform in order to automatically generate all the target platform artifacts.

three provides the specific features that address the challenge. The table demonstrates the benefits of using the layering feature, and how the different views can foster collaboration between the different cloud stakeholders.

This chapter focuses on how to use StratusML to capture the application deployment configuration and to generate the required artifacts to deploy a cloud application on a target platfrom (e.g. Windows Azure), by transforming the StratusML provider independent models into provider specific configurations. Capturing the application deployment configuration includes: (i) the application static structure (a.k.a, service model) (ii) the application runtime model (i.e., how the different components are instantiated, distributed and replicated), and (iii) the application dynamic behavior at runtime (a.k.a., adaptation model). Here is how StratusML is used to specify such information:

Define a Cloud Application Service Model (Structure): The core layer provides a set of visual components that corresponds to the StratusML core meta-model elements. Using the *core* meta-model service, developers can describe the structure of a cloud service composed of one or more *Tasks*, the types of tasks, and their relationships. For example, the model in the design window of Figure 4.2 shows a provider independent deployment model that corresponds to the CoupoNet example. The model is composed of one service (not shown in the model as it is part of the hidden configuration view) with three web tasks (i.e., frontend modules) and two worker tasks (i.e., backend modules). Each web task has at least one external endpoint that must be secure and is a frontend web MVC-style application to be accessed by specific user groups (i.e., Coupon Providers, Coupon Buyers, Admins and Marketing Researchers). The first worker task corresponds to the application backend that handles all operations (logic tier). The second is the analytics/data-crunching engine, which processes buy/sell data. There is also a storage tier, which consist of blobs for storing data collected from buy/sell dumps, and queues for asynchronous communication between worker and web tasks. The core layer furnishes various groups (i.e., storage, availability, and scalability) that assist in modeling multi-tenancy using different service/data partitioning strategies.

Specifying the Replication and Distribution of the Modules: The availability of the system depends on how the different components and modules are replicated and distributed into different regions. StratusML provides a modeling view that makes it easier to manage the locations and the number of instances of each cloud task. This helps administrators model cloud service distribution to multi-geographic locations. Figure 4.11 shows how the CoupoNet tasks are replicated and distributed. For example, three instances have been instaniated for the task "CoupoNet.Web.Admin" in two different regions both in North America, one instance in the North Central "Chicago" datacenter and two instances

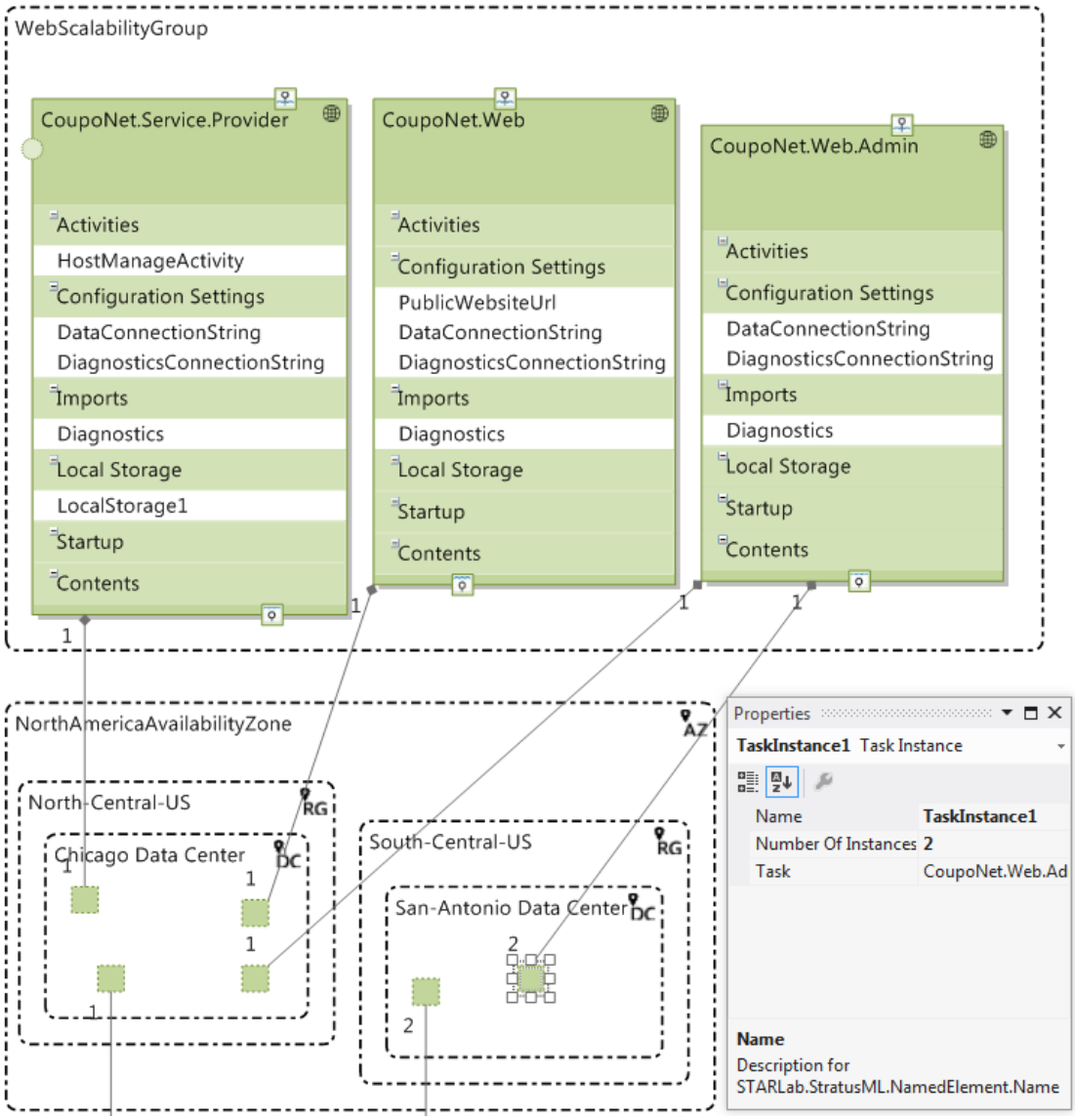


Figure 4.11: StratusML Availability View Excerpt

in the South Central "San Antonio" datacenter as shown from the link cardinality. An administrator can easily instruct the cloud fabric to relocate the instance into a different location within the same provider or to a different provider by changing the properties of the instance.

Specifiying the Application Behaviour (Adaptation Model): Using the adaptation layer, StratusML facilitates modeling adaptation rules and actions, with a focus on scalability actions. A user can specify constraints and reactive rules, and associate them with a task or scaling group. The framework generates the rule-based configurations required to automate resource provisioning. This solves the problem when required resources cannot be estimated at the beginning of the project.

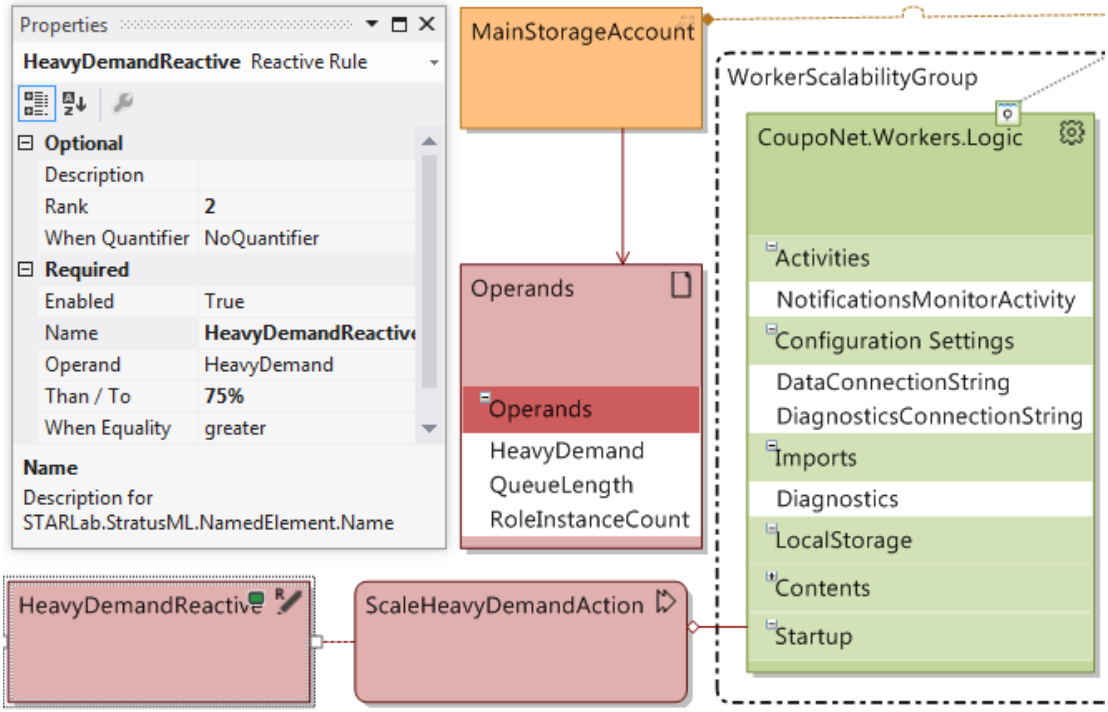


Figure 4.12: StratusML Adaptation Rule and Action Excerpt

Figure 4.12 shows a screenshot of a task (*CoupoNet.Workers.Logic*) that is associated with an adaptation action (*ScaleHeavyDemandAction*) that is activated based on a reactive rule (*HeavyDemandReactive*). The reactive rule has the rank 2. It can be enabled or disabled when needed. The activation of the reactive rule depends on the value of the operand (*HeavyDemandOperand*) and activated when the demand is greater than 75%. The value of the operand is collected at runtime. The *MainStorageAccount* component is used to set the configuration parameters of the diagnostic API.

Creating and Selecting a Cloud Provider: StratusML users can use one of the sup-

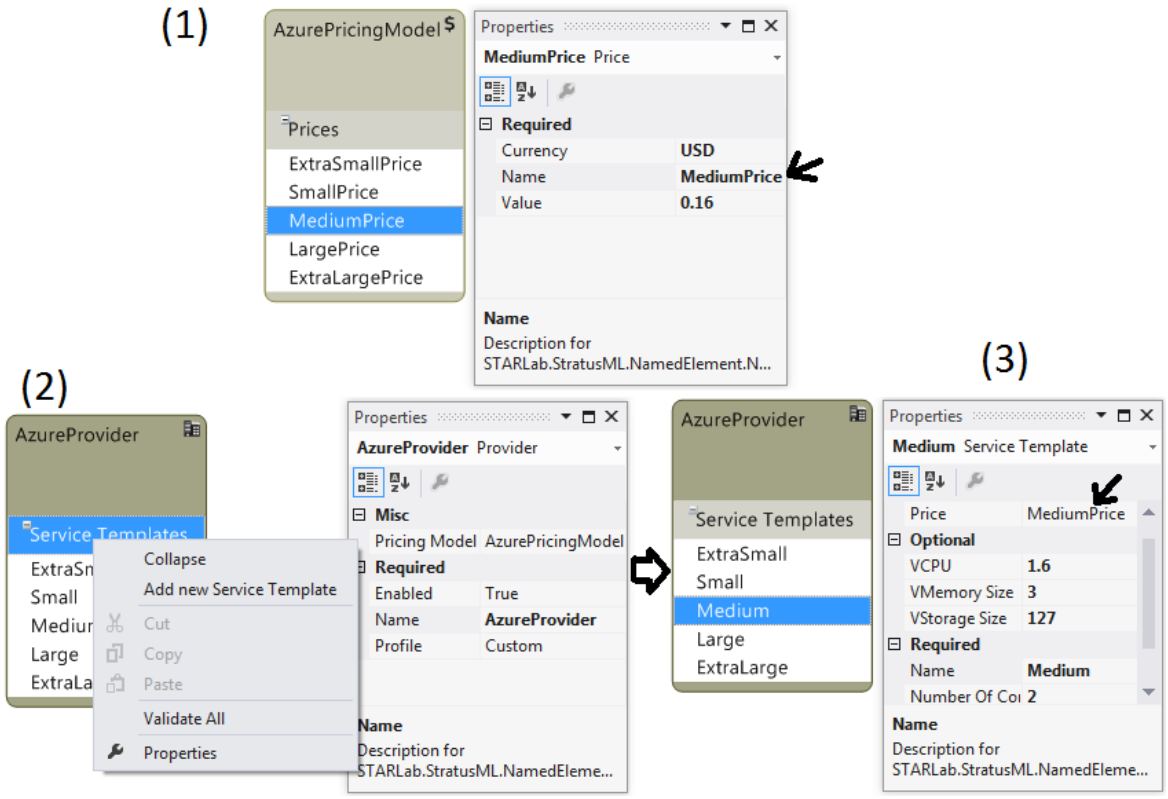


Figure 4.13: StratusML Provider Snapshot

ported providers or create a new one. The provider layer provides a set of components to describe providers' specifications. This includes; specifying (i) the *service templates*, which describe the resources provided in bundles (i.e., CPU speed, number of cores, memory size, disk space), (ii) the *availability zones* that represent the physical locations of provider data-centers, and (iii) the *pricing profiles* which specify the cost of using a VM with a specific service template under the designated provider. The information specified can be used to estimate the system performance and calculate the cost of deployment under various providers. Using and generating analytical performance models from the specified information in the deployment and service models will be discussed in detail in the next chapter.

Figure 4.13 corresponds to Windows Azure provider model. The figure shows the list of service templates offered (e.g., small, medium) by Azure, the pricing model of Azure and how each of the service templates can be assigned a pricing profile.

4.5.3 Artifacts Generation

The purpose for designing a visual DSML is to generate artifacts (e.g., code, configuration) from the models written in that language to reduce the efforts of manually creating them. By generating the cloud configuration space artifacts, StratusML reduces the administration and configuration efforts. StratusML facilitates application migration between different cloud providers, by modeling the service structure and configuration independently from the platform specifications, then generating the configuration artifacts (i.e., text XML files) required to run the application on the target platfrom. There are multiple approaches with DSL tools to transform models into code and text files [42]. StratusML uses transformation templates. Particularly, StratusML utilizes the Text Templating Transformation Toolkit (T4), which is a text transformation technology developed by Microsoft for artifacts generation.

Models created using visual editors, such as StratusML are already loaded in memory. This eliminates the need to parse and serialize the model. A template-based transformation uses text files, called transformation templates. A transformation template contains the code to import the model loaded in memory, navigates through the model, and generates textual artifacts based on the transformation rules (i.e., code) specified in the template. A transformation template consists of three parts: a static part that represents the structure of the output file, a dynamic part that corresponds to the code logic used for model navigation, patterns identification and transformation rules, and template directives which specify how the template should be processed. T4 organizes these parts into blocks. These blocks are distinguished by their opening control markers. The following are the main blocks provided in T4:

(a) Standard control blocks '$<\#$ *statements* $\#>$': Contain statements written in C# or Visual Basic to control the flow of processing in the text template.

(b) Expression control blocks '$<\# =$ *expressions* $\#>$': Contain expressions that are evaulated and automatically converted to strings to be inserted into the output file.

(c) Class feature control blocks '$<\# +$ *methods* $\#>$': Contain methods, fields and properties. It is used to add reusable pieces of template, such as transformation helper functions.

(d) Directive blocks '< @ *directives* # >': Provide instructions to the T4 template engine. For example directive blocks can be used to specify that the output file should have a '.cscfg' extension, or it should be splitted into multiple files.

Anything written outside the boundaries of these blocks are considered part of the static content, which will be emitted to the output file without processing.

Selecting a platform provider calls a hard imperative rule that transforms the model into provider specific, but platform independent model. If the model passes validation with no errors, then the save action will trigger the template based transformation to generate the actual platform specific artifacts. StrartusML provides a set of transformation templates out of the box and allows creating custom ones for a new providers.

Listing 4.4 is an example for part of a template that is used to generate Windows Azure service definition file from the StratusML model. The complete Azure defenition template as well as the other templates required to fully generate all the configurations and artifacts to deploy any StratusML model into Windows Azure are available in the StratusML web page [64]. The code in Listing 4.4 starts with a set of directives that specifies the output file name extension '*.csdef*', defines the name of a class that makes the link between our model and the T4 engine '*StratusMLDirectiveProcessor*' and loads the source instance model '*ApplicationModelCoupoNet.stratus*' of the application as modeled using the StratusML language.

The code from line 7 to 20 of the standard control block represents the primitive transformation function of the template. It utilizes three supportive transformation helper functions that are implemented within class feature control blocks to be reusable. Those are: *GenerateBasicRoleNodes*, *GenerateWebRoleNodes*, and *GenerateWorkerRoleNodes*. The *GenerateBasicRoleNodes* helper method generates the essential elements that are presented in all three types of tasks.

```
1  <#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation"#>
2  <#@output extension=".csdef" #>
3  <#@StratusML processor="StratusMLDirectiveProcessor" requires="fileName='ApplicationModelCoupoNet.stratus'"
    #>
4  <#@ import namespace="System.Collections.Generic" #>
5  <# Write("<ServiceDefinition name=\"{0}\" ", StratusDiagram.Name);
6    ... //omitted for paper presentation
7    List<Component> tasks = GetAllTasks();
8    foreach (Task task in tasks) {
9      if (task is WebTask) {
10       WriteLine("<WebRole name=\"{0}\" enableNativeCodeExecution=\"{1}\" vmsize=\"{2}\">", task.Name, (task
           as WebTask).EnableNativeCodeExecution, task.VirtualMachineSize);
11       GenerateWebRoleNodes(task as WebTask);
12       WriteLine("</WebRole>");
13     } else if (task is WorkerTask) {
```

```
14          WriteLine("<WorkerRole name=\"{0}\" enableNativeCodeExecution=\"{1}\" vmsize=\"{2}\" >", task.Name, (
              task as WorkerTask).EnableNativeCodeExecution, task.VirtualMachineSize);
15          GenerateWorkerRoleNodes(task as WorkerTask);
16          WriteLine("</WorkerRole>");
17        } else if (task is VirtualMachineTask) {
18        ... //omitted for paper presentation
19        } else
20        { throw new Exception("Invalid task type for task: " + task);}
21      }
22      GenerateNetworkTrafficRulesNode(tasks);
23      PopIndent();
24      Write("</ServiceDefinition>");
25  #>
26  <#+ void GenerateWorkerRoleNodes(WorkerTask role) {
27        GenerateBasicRoleNodes(role);
28        PushIndent(" ");
29        if (role.Runtime != null) {
30          WriteLine("<Runtime executionContext=\"{0}\">", role.Runtime.ExecutionContext);
31          PushIndent(" ");
32          GenerateEnvironmentNode(role.Runtime.Environment);
33          if (role.Runtime.NetFxEntryPoint != null && role.Runtime.NetFxEntryPoint.Count == 1) {
34            WriteLine("<EntryPoint>");
35            PushIndent(" ");
36            foreach (NetFxEntryPoint netFxEntryPoint in role.Runtime.NetFxEntryPoint) {
37              WriteLine("<NetFxEntryPoint assemblyName=\"{0}\" targetFrameworkVersion=\"{1}\" />",
                  netFxEntryPoint.AssemblyName, netFxEntryPoint.TargetFrameworkVersion);
38              break; // Only output one
39                  }
40            PopIndent();
41            WriteLine("</EntryPoint>");
42                } else if (role.Runtime.ProgramEntryPoint != null && role.Runtime.ProgramEntryPoint.Count == 1) {
43          ... // //omitted for paper presentation
44                  }
45            PopIndent();
46            WriteLine("</EntryPoint>");
47                }
48          PopIndent();
49          WriteLine("</Runtime>");
50            }
51        if (role.Startup != null && role.Startup.Count > 0) {
52          ... //omitted for paper presentation
53          }
54          PopIndent();
55          WriteLine("</Startup>"); }
56        if (role.Contents != null && role.Contents.Count > 0) {
57        ... // omitted for paper presentation
58      }
59  #>
60  <#+ List<Component> GetAllTasks() {
61         List<Component> tasks = StratusDiagram.Components.FindAll(x => x is Task);
62        foreach (ScalabilityGroup scalabilityGroup in StratusDiagram.Components.FindAll(x => x is ScalabilityGroup)) {
63          tasks.AddRange(GetAllTasks(scalabilityGroup)); }
64        return tasks;
65        }
66  #>
```

Listing 4.4: T4 Transformation (StratusML To Azure Definition).

The *GenerateWebRoleNodes* and *GenerateWorkerRoleNodes* helper methods generate nodes that are specific to web tasks and worker tasks, respectively. Both of these methods call *GenerateBasicRoleNodes*, before they start running. The template then checks that all the values were added correctly in the template. We used this template along with the templates provided in the StratusML webpage to generate the complete configurations required to deploy the CoupoNet example on Windows Azure platfrom.

4.6 Summary

This chapter presents StratusML, a modeling framework and a layered modeling language for cloud applications. The distinctive advantages of StratusML over existing cloud modeling frameworks are the following: it uses layers to provide partial and holistic views for the different cloud application concerns, it facilitates visual modeling of adaptation rules and actions, and it uses template-based transformation to deal with the cloud platforms heterogeneity.

StratusML provides the components needed to model platform independent and quality cloud applications. Moreover, it simplifies the applications' management through employing "model once deploy everywhere" model driven approach and minimizing the assumptions about the target cloud platform through offering out of the box provider's templates.

StratusML supports the cloud applications DevOps process, by covering the main architectural views that represent the application's structure, behavior and configuration. It promotes flexibility, portability, reusability and productivity.

To demonstrate the capabilities of the StratusML framework to satisfy the cloud stakeholder needs, a scenario-based example (CoupoNet) has been presented. In which we showed how the StratusML is used to model the CoupoNet example and to generate the artifactes required to deploy it in the cloud.

The next chapter investigates the relationship between the cloud application configuration space artifacts and the artifacts required to automatically generate performance models for the corresponding cloud systems.

Chapter 5

StratusPM: Generating Layered Cloud Performance Models

"Transparency is valuable, but while many things can be made transparent in distributed objects, performance isn't usually one of them."
– Martin Fowler, 2003

Chapter 4 presents StratusML: an architectural framework and a modeling language for the cloud configuration space. This chapter aims at extending StratusML to support generating analytical performance models for cloud applications by reusing the information used to configure the platform for the application deployment and operation.

Constructing an analytical performance model includes two steps: (i) modeling the application structure and the underlying infrastructure resources, (ii) modeling the user interactions and workload characterization. Traditional approaches usually consider the structural part to be *variable*, as it depends on the target infrastructure specifications, but *static* as it rarely changes after deployment. Accordingly, the automatic generation of the structural resource models is given less attention in comparison to the workload characterization and the behavioral part of the performance model. In cloud *both the platform and the application are fluid* and continuously changing after deployment. Additionally, cloud providers offer instances and service templates with known specifications (e.g., Table 4.5 and 4.6) that provides opportunity to automate the process of generating performance models for cloud applications. However, even with such opportunity, the problem of constructing accurate performance models is complex, because it requires knowledge of sparse information that changes continuously according to the decisions of different stakeholders.

This includes: the application service architecture, the platform specification, the system runtime parameters (i.e., service time, number of calls and their probabilities, response time, throughput, etc.), and the target analytical model formalism and notations.

The goal of this chapter is to empower collaborations and reduce the efforts needed to specify the performance models by reusing other stakeholders artifacts. We extend the StratusML framework to support annotating the cloud models with runtime performance parameter, and enable transforming the annotated StratusML models into Layered Queuing Network (LQN) models.

The rest of this chapter is organized as follows. Section 5.1 evaluates the performance modeling requirements in context of cloud artifacts. Section 5.2 extends the StratusML to capture the application performance. Section 5.3 explains the reasoning behind selecting the Layered Queuing Network as a target analytical performance model to analyze the cloud application performance. Section 5.4 explains the rules needed to transform the StratusML models that are annotated with StratusPM components into Layered Queuing Network (LQN) models. Section 5.5 provides a case study that demonstrates how to use StartusPM to model the cloud application performance and how to generate LQN models by applying the proposed transformation rules. Finally, Section 5.6 summarizes the chapter.

5.1 Evaluating the Requirements of Performance Modeling

A performance model is a quantitative measure of how the system operations use its *resources* (which resources, for how long, and in what order) [128]. It is used to predict resource utilization and estimate the required resources. There is an overlap between the information that is required to analytically analyze the system performance and the information that is required to deploy and run an application on a particular platform. Deploying an application on a cloud platform requires specifying how the application service model uses the platform *resources* of that particular provider. Specifying an application performance model requires specifying the characteristics of the software application and its underlying platforms [129]. Therefore, both deployment artifacts and the performance model use the same *resource model*, which depends on the providers' templates and service architecture.

The following are the two different sets of information that are required before the model is ready to be solved analytically.

(a) The structural model, which is represented by:

A1. The application components.

A2. The resources (e.g., cpu, disks) under which the application components operate.

A3. Multiplicity of resources and components (i.e., how many of each).

A4. Scheduling policies for all components, tasks and their priorities.

(b) The behavioral model, which is represented by:

B1. The service time of each component.

B2. The interaction between components (i.e., call paths).

B3. The number of calls to a service (i.e., probability a call path is taken).

Table 5.1 shows where the aforementioned information that are required to build a performance model for a cloud system could potentially be found within the artifacts that describe a cloud application deployment. The table uses examples from three different cloud providers (i.e., Azure, GAE, and AWS), in addition to the StratusML models explained in Chapter 4.

Table 5.1: Mapping Performance Modeling Requirements to Cloud Artifacts

Performance Modeling Components	**Cloud Artifacts (The Field Within Each Artifact)**			
	Azure	**GAE**	**AWS**	**StratusML Model**
A1. Application components	Definition File (Role)	Deployment Descriptor (Module)	AWSTemplate ElasticBeanstalk (Beanstalk)	Service Model (Task)
A2. Resources	Definition Files (VMSize) + Provider Specs (e.g., Table I)	Configuration File (InstanceClass) + Provider Specs (e.g., Table I)	AWSTemplate AutoScaling (instance type) + Provider Specs (e.g., Table I)	Service Model (Service Template) Provider Model (Resource Specification)
A3. Multiplicity of resources and components	Configuration File (Instances count)	Configuration File (instances)	AWSTemplate AutoScaling ServerCapacity	Availability Model (NumberOfInstances)
A4. Scheduling policies	Adaptation rules WASABi	Configuration File (autoscaling)	AWSTemplate AutoScaling	Adaptation Model
B1. Service time of each component	N/A	N/A	N/A	N/A*
B2. Interaction between components	Partial Info in Def and Config Files	Partial Info in Dispatch file	Partial Info in Route 53 file	N/A*
B3. Number of calls to a service	N/A	N/A	N/A	N/A*

*Addressed in this chapter

In Table 5.1, an *application component* refers to a virtual appliance [67] or virtual machine image (e.g., a *Role* in Windows Azure, a *Module* in GAE, or *Task* in StratusML

terminologies). Recall that such components can be found in the deployment package definition files. The description of the underlying resources that is required to run these virtual appliances is determined based on the virtual machine template, as specified in the configuration or definition files, while the actual hardware specification of this template is provided in the PaaS specifications as defined in the service level agreement. The multiplicity and replication of each component is another important factor that can affect the performance of the application. Such information can be extracted from the configuration files of the deployment package. Moreover, using the adaptation rules and scaling policies, the minimum and maximum number of instances allowed can be determined. This can be used to tune the simulation parameters to study the application performance under different configurations.

Table 5.1 shows that it could be possible to generate the structural part of the performance model, in addition to some of the interactions between the components from the cloud application package artifacts. Since StratusML models are able to generate these artifacts, it would also be possible to generate the structural part directly from StratusML models. Using StratusML models to generate a performance model can have a significant advantage over generating transformation through mapping each of the provider specific artifacts to its corresponding performance components. Using StratusML models as pivot models can reduce the number of transformations required from $N \times M$ to $N + M$, where N is the number of providers and M is the number of target performance models.

Table 5.1 also shows that in order to have a fully functional analytical performance model, there is a need to extend the StratusML models to capture the information related to the behavioral-model. This includes; (i) the usage model, which describes the interaction between the users, and (ii) the system and the resource utilization parameters, which show the load on the different resources as a result of user requests. This information can be presented as annotations in a separate view, because it is usually specified by different stakeholders (i.e., Performance Engineers), and it is based on a separate process (i.e., the application instrumentation and measurement). Section 5.2 shows how the StratusML can be extended to capture the missing performance parameters.

5.2 StratusPM: A Pivot Model for Cloud Performance

This section describes an extenstion to the StratusML meta-model. This extension is based on the evaluation provided in Section 5.1. The objectives of this extension is to provide a set of modeling components to capture the cloud application resource utilization parameters and usage model. We refer to the pivot model that carries the informaition

that is required to generate analytical perfromance models (e.g., LQNS, Opera, JMT) for cloud applications as the StratusPM. The StratusPM meta-model consits of two sub meta-models that extend the StratusML core meta-model components. Those are the perfromance meta-model and the workflow meta-model.

5.2.1 The Performance Meta-Model

Figure 5.1 depicts the performance meta-model. Its elements were inspired by the UML performance analysis modeling profile (PAM), which is part of MARTE profile [123]. As shown in Figure 5.1, there are five association relationships that connect three core components to corresponding performance aspects. These links represent integration points between the cloud resource model and its performance specifications.

Each *Activity* is linked to *ActivityPerformance*, *Workload*, and to itself through the *Call* reflexive relationship. The *ActivityPerformance* specifies the activity server performance parameters; such as, the activity resource demand, think time that is optionally used to specify pure delay, and the desired response time of that activity. The *Workload* specifies the intensity of demand on that activity. The *Call* component specifies the number of calls that each activity makes to others, the types of these calls, the communication mechanism (*InteractionType*) used, as well as the size of the input and output messages associated with the calls. While an activity can be the source of several calls, each call is associated with exactly two activities; a source and a target activity. An *InteractionType* can be synchronous, asynchronous, or forward, with the same semantics as in LQN [52]. The *Endpoint* is connected to a*Connection* component. The *Connection* links endpoints. All calls that cross activities between Tasks must go through the *Connection*, which acts as pipe between two endpoints. The *Network* performance component specifies the connection parameters for a *Connection*. Finally, each *Task* is connected to a *HostingEnvironment*. The *HostingEnvironment* parameters can be inferred based on the task type and the target PaaS provided as specified in the provider meta-model.

In a nutshell, each *task* can perform several *activities*. Each *activity* has a *phase* and a *type*. The *phase* specifies the order of execution within the *task*, while the *type* specifies whether it is a *normal* activity or a *join*, *fork*, or *loop* activity. An *activity* can be linked to other activities within the same task or with external tasks. Requests directed to activities in other tasks are sent to one of the external *endpoints* of the target task. A phase zero activity is always connected to one of the *external endpoint*s of the task. A phase zero activity receives the external requests that are directed to that *external endpoint*.

Each *activity* belongs to one or more *workloads*. A *workload* logically groups all *activi-*

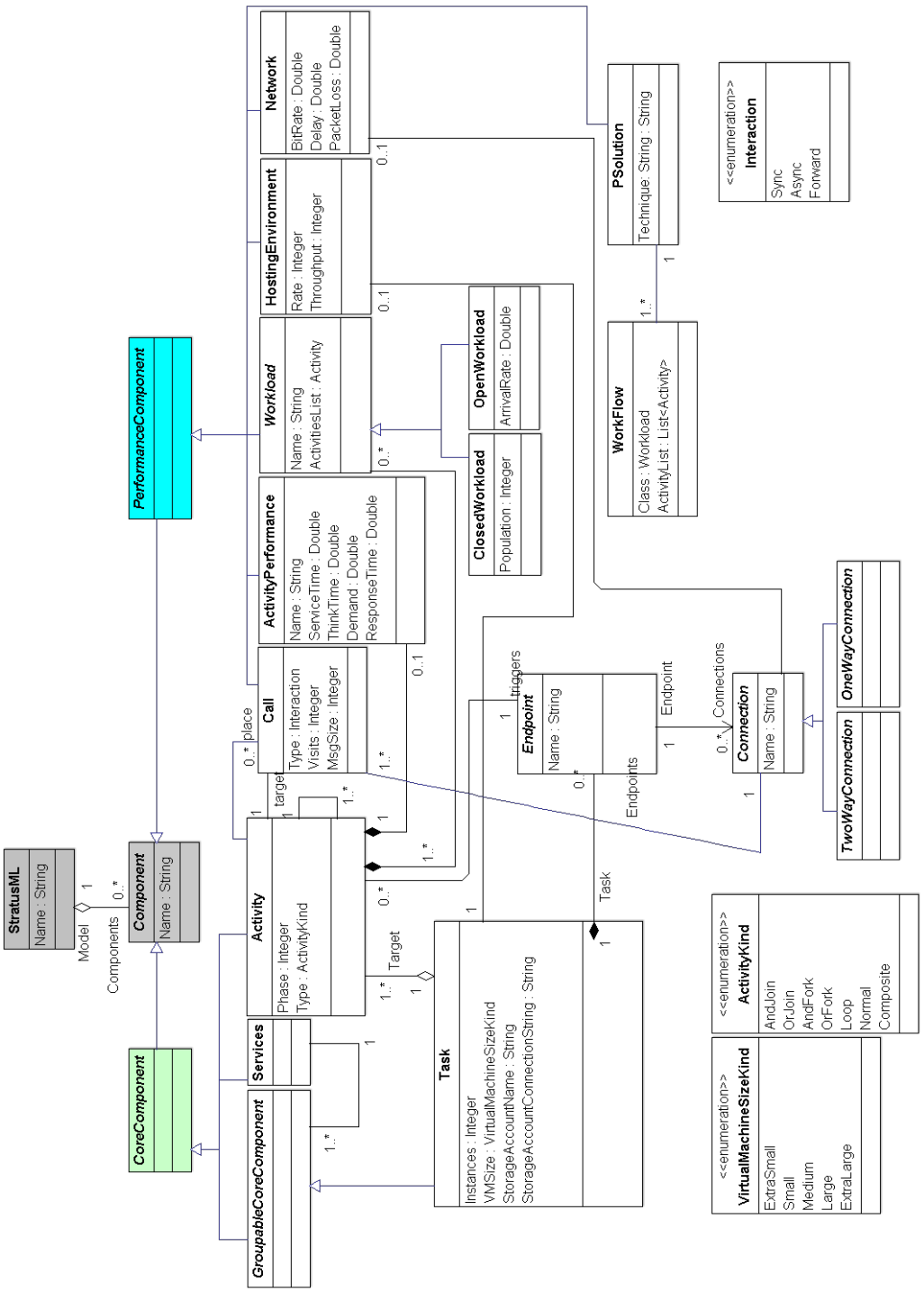


Figure 5.1: The Performance View Meta-Model

ties that are affected by the same traffic class and hence should be executed together, while a *task* groups all *activities* that share and utilize the same underlying resources. There are two types of workloads: *open workloads*, where requests arrive at a given rate; and *closed workloads*, where requests are generated by a fixed number of users (*population*). Since there is many-to-many relationship between *activities* and *workloads*, another entity is needed to specify the activity execution path for each *workload*. The *workflow* entity is part of the workflow view. Each performance solution (*PSolution*) has a list of *workflows*. Each *workflow* has a designated *workload* and an ordered list of the *Activities* performed by that *workflow*. A performance solution (*PSolution*) also specifies the solution technique used in solving the performance model (i.e., analytical, or simulation).

5.2.2 The Workflow Meta-Model

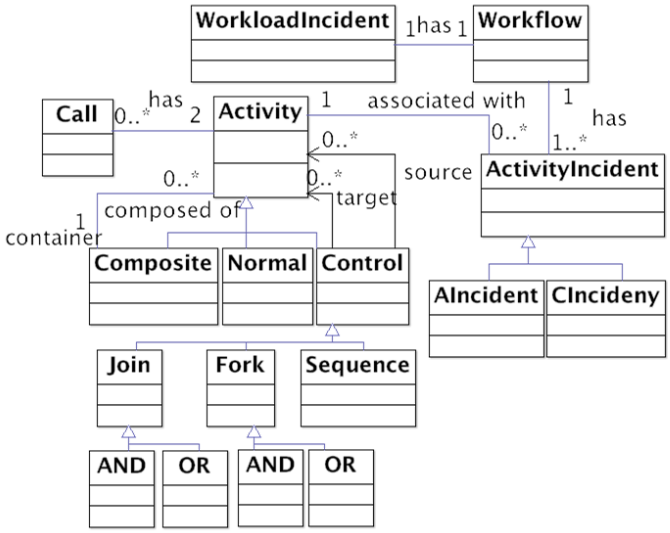


Figure 5.2: The Workflow View Meta-Model

To represent system behavior and enable operational analysis, the (5+1) model adopts a simple *workflow* meta-model. It specifies activity control structures that need to be executed in sequence to perform a certain work. As shown in Figure 5.2, the two main components in the *workflow* meta-model are *activities* and *activity incidents*. A workflow model represents the actual occurrence of the activities that are defined in the core model along with the workload generator (from the performance model) that is responsible for the activity enactment. Each *activity incident* belongs to one *activity*. This allows us

to compose different workflows (execution scenarios) for the activities according to their runtime execution.

An *activity* can be classified as a *normal*, *composite* or *control* activity. A *control activity* links source and target activities and describes their execution sequence and concurrency. Depending on the control activity type, it can link one source to one target (*sequence*), one source to many targets (*fork*), or many sources (*join*). Activities that succeed a fork control can all be executed concurrently (*and fork*), or one at a time based on evaluating a boolean (*or fork*). Similarly, in an (*and join*) all predecessor activities must be executed before the join, while in an (*or join*) the execution of any of the predecessor activities is enough to start execution of the join activity. The *normal* activity is an elementary activity that cannot be further decomposed, while a *composite* activity contains (n) normal or complex activities and their control joint activities.

Even though the workflow model is essential for performance analysis, it is better to model activity definitions, performance specifications and actual usage scenarios seperately. This separation of concerns maximizes modeling flexibility, fosters reuse, and facilitates modeling of the application's behavior at runtime.

5.3 The Target Analytical Performance Model

While it is possible to transform StratusPM models into any analytical performance model, this thesis adopts the layered queuing network as a target model. This section starts by a brief introduction of the basic concepts of LQN, followed by the reasons of adopting LQN as a target analytical performance model.

A Layered Queuing Network (LQN) is a set of extensions and notations for extended queuing networks to model systems with nested resources [52]. Figure 5.3 describes the LQN metamodel. Here, we briefly describe the main concepts of the LQN model. An LQN model contains software resources called tasks which are allocated to hosts called processors. A task provides a set of services called entries. A task also owns a queue where requests for entries are waiting for the service based on scheduling type. The process of an entry is described using a set of activities connected by different precedence relationship. An activity represents the smallest operation unit in LQN model.

Activities can also be described in the model in the form of phases, where they are constrained to be executed in a sequence of one to three. A request from an activity (or phase) to entry can be synchronous, asynchronous, or forwarding. In a synchronous request, a client is blocked waiting for a reply. In an asynchronous request, there is no

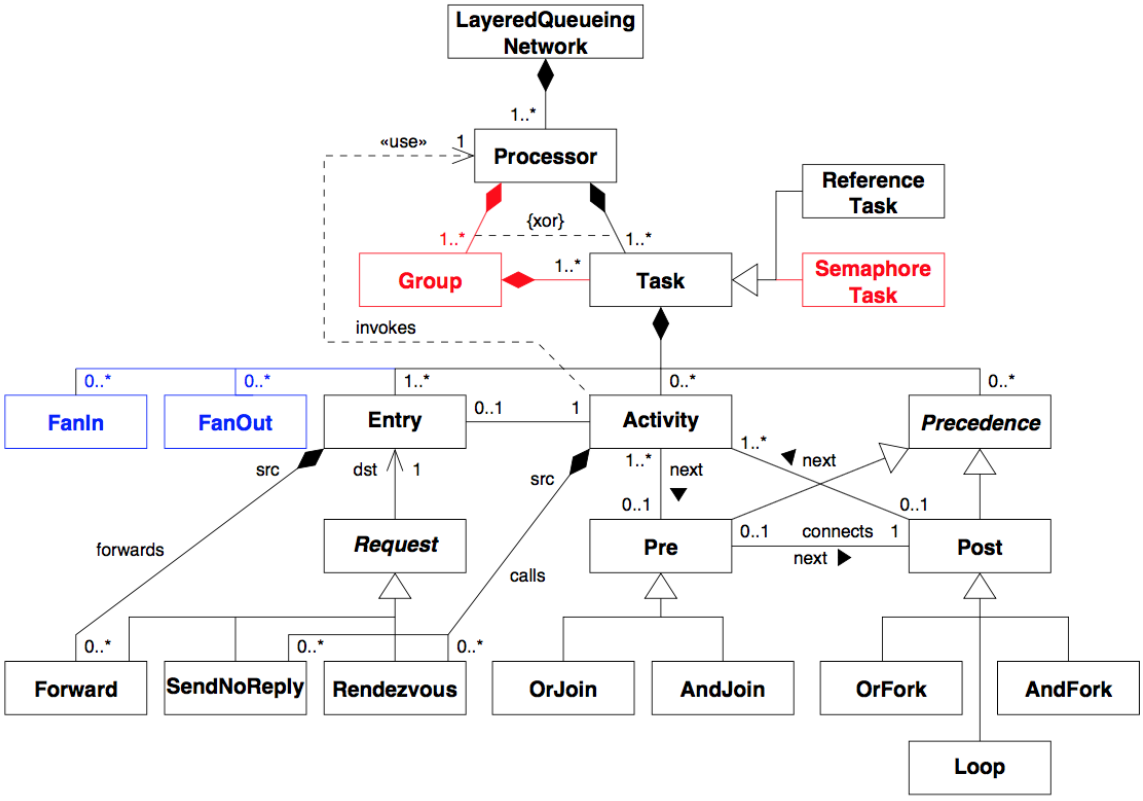


Figure 5.3: LQN Meta Model [159]

reply. Finally, in a forwarding request, the client request is processed by a chain of servers: the first server in the chain will forward the request to the second. The request will keep forwarded until it reaches the last server in the chain, which will reply to the client.

LQN is appropriate for modeling complex systems that exhibits complex characteristics such as multi-tiers of service, fork/join interactions, and asynchronous communication. This makes LQN a good fit to model the performance of cloud applications. More particularly, the reasons to select the layered queuing network as a target model are the following:

(a) LQN is a form of Extended Queuing Networks for systems with nested multiple resource possession. This ability to model the performance of systems with multiple layers is a perfect match for cloud application modelling. This is because the cloud computing service models also follow a strictly layered architecture (i.e., IaaS, PaaS, and SaaS).

(b) LQN provides a multi-sever multi-class solvers that can easily model replication. Replication is a key characteristic in the cloud that is used to ensure the availability scalability, and performance requirements of a cloud application.

(c) LQN supports multi-threaded and multi-processor sever modeling. This makes it possible to write transformation rules that provide direct mapping between the cloud application constructs and the LQN model constructs.

(d) LQN supports analyzing models that have Synchronous, Asynchronous and Forward communication patterns. This covers the main communication patterns supported in the cloud.

(e) LQN has an up-to-date meta-model and XML model definition language, which makes it possible to have a model to model and model to text transformation.

(f) LQN provides both analytical and simulation solvers (LQNS) that allow us to easily compare the results once the application model and workload information have been provided.

(g) The LQN model can be used for different types of performance analysis. It can be used in sensitivity analysis, capacity planning, and to estimates the response times and throughputs of the users and services.

In the next section, we describe the process of generating LQN models from StratusPM models.

5.4 Generating LQN Performance Models from StratusPM Models

The ability to capture the providers' predefined templates (*Resource Specifications*) represent a turning point in automating the performance analysis process. In the previous sections, we showed how StratusPM can capture the cloud application performance, through (i) reusing the StratusML components and the cloud providers' specific templates, and (ii) annotating these models with resource usage and utilization models. In this section, we present the transformation process and rules to generate LQN models from the StratusPM models.

As shown in Figure 5.4, the transformation process is composed of two levels: In the first level, the LQN structural model is generated using the StratusML models and the providers' templates. In the second level, the LQN performance parameters are determined using the StratusPM workflow and performance models.

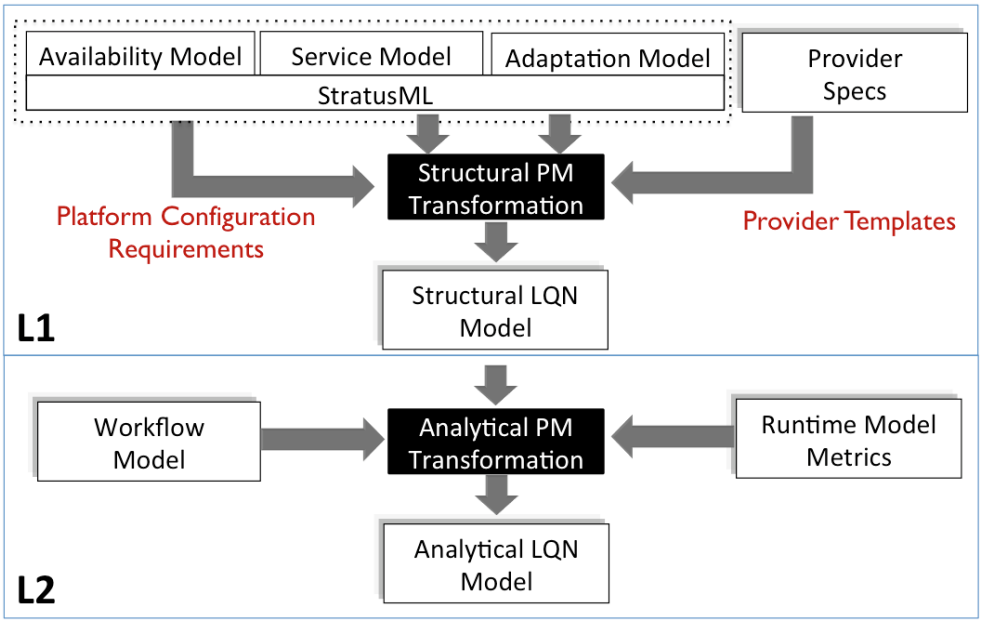


Figure 5.4: Transformation Process from StratusPM to LQN

5.4.1 Level 1: Generating the LQN Structural Models

Deriving the LQN structural model can be achieved through two refinement steps. In the first step, the structural skeleton is generated from the StratusML service model (i.e., cloud tasks and their interaction mechanisms). In the second step, the structural model dynamic parameter (i.e., the components replication and multiplicity) are specified. The following describes the two steps to derivate the structural model:

Step 1: From StratusML Service Model to LQN Structural Model

In this step, the service model is used to generate the skeleton of the performance model. The outcome of this step is an LQN structural model that specifies the main tasks and

hosts without specifying any deployment configuration parameters (e.g., tasks replications) or target provider specifications (e.g., resource multiplicity). The following is the set of transformation rules that are used to transform a cloud service model into an LQN structural performance model.

- **L1-S1-R1**: Each *cloud task* (i.e, web , worker) is translated into:
  - An LQN task.
  - A *processor* that is annotated with the *size* of the VM hosting the task.
- **L1-S1-R2**: Each inbound *endpoint* in a cloud task is translated into an LQN *entry* place holder. This place holder will be replaced with the actual set of entries based on the activities that are bound to the endpoint as we will see later.
- **L1-S1-R3**: Each *web task* will be connected to one or more LQN *reference tasks*. Each represents the external requests of one class of users. The number of reference tasks is equal to the number of *external endpoints*(e.g., input endpoint in Azure) of the web task mutiplied by the number of distinct types of user request per endpoint. Moreover, for each of the reference tasks, an entry will be created on the web task. In this step we only create a place holder for an LQN *reference tasks*. The number of distinct types of user request, in addition to the number of users and request rate for each of the reference tasks will be determined in later steps.
- **L1-S1-R4**: Each *connection (pipe)* between two task *endpoints* is transformed by:
  - Creating a *pseudo LQN task* that represents the communication overhead. An LQN *synchronous call* is created between the *source task entry* and the *pseudo LQN task entry*. Moreover, an LQN *call* is created between the *source LQN task entry* and the *target task entry*. The second LQN call can be *synchronous*, *asynchronous* or *forwarding* depending on the communication mechanism.
  - Creating a *processor* that represents a pure delay center.
- **L1-S1-R5**: Each cloud *storage task* is translated into an LQN *task* and pure delay *processor*.

The aforementioned rules focus on generating the static structural model. The next step focuses on the dynamic structural model (i.e., the model multiplicity and replication).

Step 2: Specifying the Multiplicity and Replication of the Structural Model

Multiplicity and replication give the cloud resource model its fluid nature. LQN has been selected as a target model due to its ability to model the performance and solve systems with replication and multiplicity. LQN supports multiplicity and replication for both tasks and processors.

The *multiplicity of an LQN task* refers to the number of threads or sessions that can run concurrently. In the cloud service model, this number can vary based on the cloud task type (i.e., web, worker), instance size and the target cloud provider. On the other hand, *the LQN recourse multiplicity* refers to the number of cores of an LQN processor. This can be used to model the number of CPUs in a Virtual Machine Instance (VMI). The recourse multiplicity of an LQN processor can be inferred from the size of a cloud task (e.g., small, medium) and the target provider specifications. Recourse multiplicity can be used to model *vertical scaling* in the cloud.

LQN uses replication to reduce the number of nodes in the LQN model by (i) combining tasks and processors with identical behavior into a single object, and then (ii) specifying the number of replicas of that object (node). Replication can be used to immitate the cloud *horizontal scaling* (i.e., scaling out behavior). The replication number of a cloud task (i.e., web, worker) represents the number of server or container instances.

Figure 5.5 shows some of the different scenarios that LQN supports by varying the multiplicity and replication parameters. Note that in multiplicity a single queue is served by *multiple* servers (i.e., tasks or processors). On the other hand, queues of the servers are also copied in replication. Hence, requests must be routed to the various queues. Requests may be scheduled using different queueing disciplines. By default first come first serve is used as queuing discipline for both LQN tasks and processors unless specified otherwise. Case (A) shows an example of a task that runs on a multiprocessor host. Case (B) represents a multithreading scenario, where several copies of a task are running on one host with two core processor. Case (C) uses replication and corresponds to horizontal scaling in the cloud. Finally, case (D) represents horizontal scaling at the infrastructure level.

The artifacts that contribute to the input data of the second step are the following: (i) the structural model as derived in the first step, (ii) the platform as a service (PaaS) specific configuration, and (iii) the cloud application packaging and deployment configuration. As explained earlier, all this information is captured by the StratusML models

The transformation rules to specify multiplicity and replication are as follows:

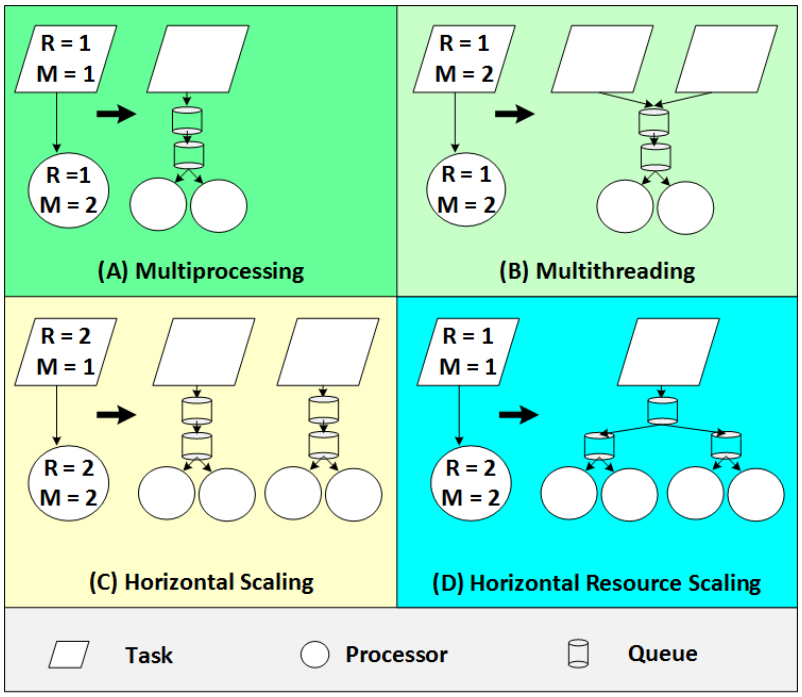


Figure 5.5: Different Scenarios of Varying Multiplicity (M) and Replication (R) in LQN

- **L1-S2-R1: Multiplicity**: For each *cloud task*, the corresponding template that represents the platform specific configuration will be applied to further specify the task. For instance if the *VMSize = small* and the target platform is Windows Azure. Then, the multiplicity of the processors annotated with small will be replaced by 2 according to Table 4.6. The rules below show the mapping relationship, the PS on the arrow indicates that this mapping rule depends on platform specific data.

  Task::ServiceTemplate::ResourceSpecifications:NoOfCores $\overset{PS}{\longmapsto}$ LQN::Processor:multiplicity

  Task::ServiceTemplate::ResourceSpecifications:vCPU $\overset{PS}{\longmapsto}$LQN::Processor::speedfactor

- **L1-S2-R2: Replication**: Modeling for replication is given special attention in our framework. This is because elasticity in the cloud is based on replication (i.e., the ability to scale out). The following steps show the replication mapping rules. The PI on the arrow indicates that the mapping process depends on platform independent data.

  - **L1-S2-R2-1**: Set the replication of the processor to equal the number of copies of the Task

Task:Instances $\overset{PI}{\longmapsto}$ LQN::Processor:replication

- **L1-S2-R2-2**: Set the replication of the Task

Task:Instances $\overset{PI}{\longmapsto}$ LQN::Task:replication

The advantage of providing mapping rules for replication and multiplicity is to insure correctness, by reducing the human errors in manual modeling. Replication of tasks may occur at any and all layers. Moreover, the replicated model may consist of a mixture of replicated and non-replicated components.

By completing this step the structural LQN model is complete. In the next section we will discuss how to map StratusPM parameters to the LQN runtime parameters.

5.4.2 Level 2: Mapping StratusPM Workflow and Performance Parameters to LQN Parameters

In the previous section, we showed how to construct an LQN structural model by reusing the cloud artifacts as modeled in StratusML. In this section, we focus on the behavioral aspects of the LQN performance model.

As explained earlier the performance behavioral model includes: the service time of each component, call paths, number of calls to a service in addition to the workload information. These parameters are captured in the StratusPM performance and workflow models. In this section, we describe the set of transformation rules needed to map the StartusPM parameters to its corresponding LQN parameters. To do so, we follow a similar approach to the one proposed by Petriu et al. [130] to transform PUMA models into LQN models.

The following is the set of transformation rules to generate the LQN behavioral model elements (i.e., reference tasks, entries, phases activities, and calls) from StratusPM models.

L2-R1: Generating LQN reference tasks:

In StratusPM, each workflow is annotated with its workload that represents the load generator for a specific classe of users. The workload can be open or closed. An open workload is specified using user population and think time, while a closed workload is specified using arrival rates. In LQN, load generators and system users are modeled as *reference tasks*. To map the StarusPM *workload* parameters to its corresponding LQN *reference task* parameters, the following rule is applied:

For each *workflow* in StratusPM, we create a *reference task*, then we map the *workload* parameters associated with that workflow to its corresponding *entry* paramters of the created *reference task* as follows:

Workflow::ClosedWorkloadIncident:Population $\longmapsto$ LQN:ReferenceTask:Entry:multiplicity

Workflow::ClosedWorkloadIncident:ThinkTime $\longmapsto$ LQN:ReferenceTask:Entry:thinktime

Workflow::OpenWorkloadIncident:ArivalRate $\longmapsto$ LQN:ReferenceTask:Entry:openarrivalrate

L2-R2: Generating LQN entries and phases:

In the previous subsection, an entry placeholder was created for each task endpoint. Here, we replace the placeholders with the actual entries. Moreover, we specify the boundaries of the phases of each entry and calculate the service time for each phase based on the activities involved in the execution of that phase.

Recall that a service model describes the high level interaction between tasks that represent software servers, while the workflow model represents the interactions between activate that are running on these servers. When a task received a request, a set of activities will be executed. The exaction of these activities may trigger the execution of other internal activities within the task or external actions in other tasks. We assume that all these activities are atomic.

For each external call (request for a service) that passes through a connection (pipe) we create an entry in the task that recieves the request through an inbound endpoint. For example, in Figure 5.7, the task *CoupoNet.Worker.Logic* recieves two external calls one from *CoupoNet.Web* and the second from *CoupoNet.Provider*. Hence, two entries will be created on the *CoupoNet.Worker.Logic* task. The number of calls on any request is extracted automatically from the annotated StratusPM model as follows:
Workflow::ActivityIncident::Call:Visits $\longmapsto$ Task::Entry:Request:calls-mean

For each of the entries created, the number of phases (or the activity graph) and their boundaries need to be determined. Several cases need to be addressed:

L2-R2-Case1: The task that receives the request executes a series of activities and returns a response to the task that requested the service at the end; after completing the execution of all the activities. In this case, we have a single-phase execution. The service demand of the phase calculated by adding of the execution demands of all the activities, between receiving external request until sending the response. Note that any of the executed activities may perform external calls to other tasks; however, the response for these calls should be received before the phase is completed.

L2-R2-Case2: The task that receives the service request continues to perform other activities after sending a response message to the task that requested the service. In this

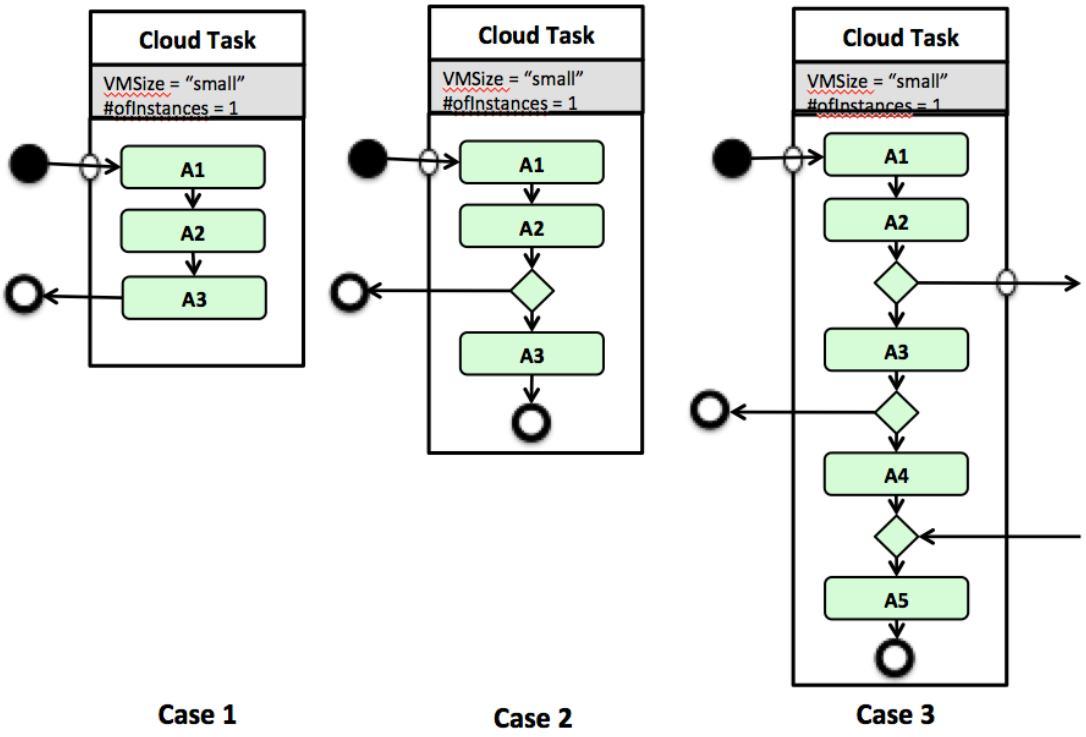


Figure 5.6: The Different Cases to Generating Entries and Phases

case, we have a two-phase entry, where the first phase is completed and the second phase is started after sending the response message. If the execution of the second phase continues to completion, then the execution demand of each of the two phases is calculated by adding all the activities within each phase.

L2-R2-Case3: The activity graph of the task that receives the request contains branches that make it impossible to divide the entry into one or two phases. Several solutions can be adopted in this case. For example, an LQN activity graph can be created that corresponds to the activities within the task. This is the simplest approach as there is one-to-one mapping between the StratusPM workflow activities and the LQN activities graph. Unfortunately, although the LQN model generated from this mapping may be solvable using simulation, it may not have analytical solution. To generate LQN model that has analytical solution, we adopted a transformation approach that is inspired by Islam et al. [78] simplified LQN technique. The technique is based on generating a simplified LQN model by combining the activities, tasks and processors in such away that the model is

always analytically solvable. Here, we only simplify activity graphs with complex branches as follows:

- Substitute the activities of an entry by the total entry demand (S_e). The total entry demand is calculated by aggregating the demands caused by executing each of the activities (S_i) in the activity graph, each multiplied by the number of times each activity is executed (c_i) per request to the entry (e) according to the following equation:

$$S_e = \sum_i c_i \times S_i \tag{5.1}$$

- Substitute the calls (y_{id}) from the aggrigated activities to any other destination entry (d) by calls from the entry (e) of the aggrigated activities. For each destination entry (d) the number of calls (y_{ed}) from the source entry (e) equals the sum of the calls from the activities as in the following equation:

$$y_{ed} = \sum_i c_i \times y_{id} \tag{5.2}$$

L2-R3: For each communication link between two tasks, we set the host demand of the entry of the pseudo task that has been created to represent the network delay to the value of the network delay associated with that connection.

Connection::Network:Delay $\longmapsto$ Task::Entry:host-demand-mean

In addition to these transformation rules, delay centers will be generated, when appropriate, to model the platform delays as a result of load balancing services and network communication delays.

5.5 Case Study: CoupoNet

In this section, a scenario-based example will be used, first to demonstrate how the StratusPM components can be used to annotate a StratusML model with performance and workflow parameters, and second to show how the transformation approach proposed in this chapter can be applied to generate LQN models from the annotated StratusML models.

5.5.1 Annotating CoupoNet with Performance and Workflow Parameters

Chapter 4 uses the CoupoNet example to demonstrate the capabilities of StratusML to specify a cloud application service model, deployment configuration, provider specifications and adaptation rules and actions. In this section, the same CoupoNet example is used to demonstrate how the StratusPM components introduced in this chapter can be used to specify the CoupoNet service model further with performance parameters and workflow details.

Recall, that the CoupoNet example presented in Section 4.5 has three types of users (i.e., Coupon Providers, Coupon Customers and CoupoNet Administrators). Figure 5.7 shows two workflows (i.e., W1: fetch coupon and W2: create coupon) that represent two of the interaction scenarios for two of the CoupoNet users. Those are the coupon customers and coupon providers. The figure shows the workload generators $\ll Workload \gg$, which specify the classes of interactions. It also shows the tasks involved in each of these interactions, the interconnections between the tasks (i.e., endpoints and connections (pipes)), the specification of the provider specific virtual instances that host these tasks (e.g., medium azure A2 instance) - which consist of the memory, number of cores and the virtual CPUs clock speed - and the activities that run within each task.

Both workflows (i.e., W1 and W2) start with a request sent to the request handler, which accordingly distributes the request to the appropriate activity. For simplification, in this scenario there is only one type of request directed to each of the tasks. W1 (i.e, fetch coupon workflow) consists of a sequence of seven activity incidents that are distributed into three different tasks. The workflow represents an example of a synchronous blocking client server communication. On the other hand, W2 (i.e., create coupon workflow) consists of a sequence of eight activity incidents that are also distributed between three tasks. The workflow is an example of a non-blocking asynchronous communication. The activity that submits the coupon does not need to wait for the coupon to be stored and can continue to accept new requests from users. The pull coupon activity will determine the appropriate location that is close to the target customer and save it when the storage is available.

As shown in the figure, the two workflows (W1 and W2) share the resources of two tasks. Using the performance components, each of the connections between two task endpoints can be annotated with the network data-flow performance parameters $\ll Network \gg$. On the other hand, the calls between two activities can be annotated with the number of visits and message size $\ll Call \gg$. Note that when data-flow parameters are not available, delays can be estimated as part of the system performance calibration and can be modeled as delay centers. This is common in the cloud, as network traffic depends on where the

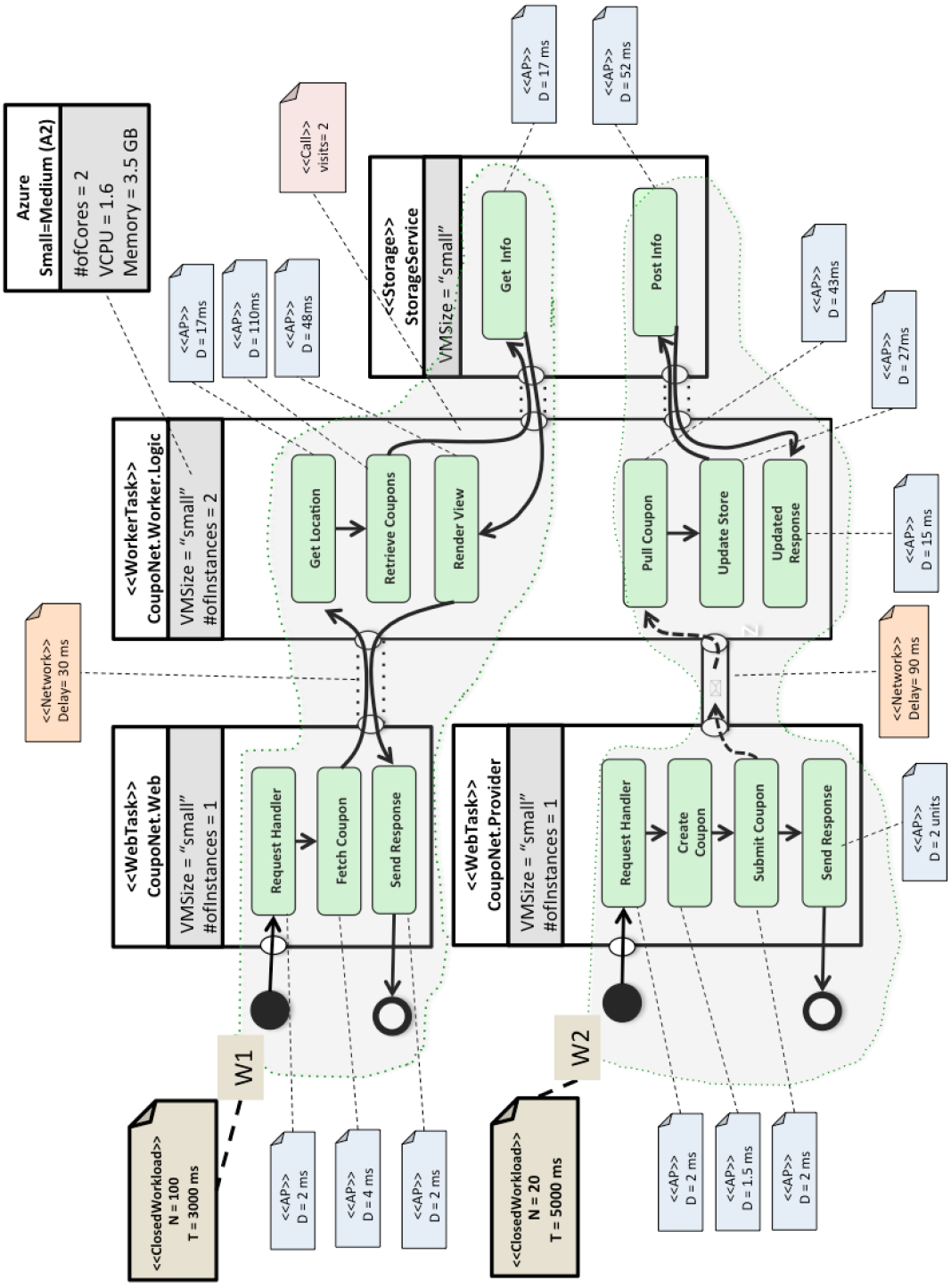


Figure 5.7: Annotating CoupoNet with Performance Parameters

cloud provider places the tasks within each region and the datacenter infrastructure (i.e., routers and switches) used by each particular provider. This information is hidden from the SaaS users. To keep the model clean, only two connections (pipes) and one call have been annotated as an example.

Finally, each activity is annotated with its service demand using the activity performance component $\ll AP \gg$. Moreover, each workflow is annotated with the workload (i.e., close, open) parameters applied to it. In this figure a closed workload is applied on each of the workflows. The workload is specified by the number of users and the think time (i.e., average time between two requests).

5.5.2 Applying the Transformation Rules to the CoupoNet Example

In this subsection, we show how to generate an LQN model for the annotated CoupoNet example presented in section 5.5.1. For the purpose of presentation, the transformation process is explained through representative examples and in steps.

Level 1: Generating LQN Structural Model

Figure 5.8 shows the results of applying the rules in *step 1* of the first level of transformations, which aims to generate the LQN structural model from a StratusML core model. The StratusML core model in Figure 5.8(a) corresponds to part of the CoupoNet example presented earlier (Figure 5.7). Figure 5.8(b) shows the corresponding Azure service definition file, while Figure 5.8(c) shows the LQN model generated, as a result of applying the rules in step 1.

In Figure 5.8(c), the LQN tasks are presented with bold rectangles with entries that represent atomic microservices attached to them (i.e., the non-bold rectangles). The service entries use lower layer services through request and reply interactions indicated by call arcs between the entries. The LQN host processors are represented as ellipses following the LQN notations.

The LQN tasks (CoupoNet.Web) and (CoupoNet.Worker.Login) in addition to their LQN processors have been generated from applying the rule L1-S1-R1. Each processor is annotated with the "*vmsize*" value (e.g., *small*). An entry placeholder (e.g., FetchCoupon, RetrieveCoupon) is created for each of the generated LQN tasks (e.g. CoupoNet.Web, CoupoNet.Worker.Login) by applying the rule L1-S1-R2. The created entry placeholders

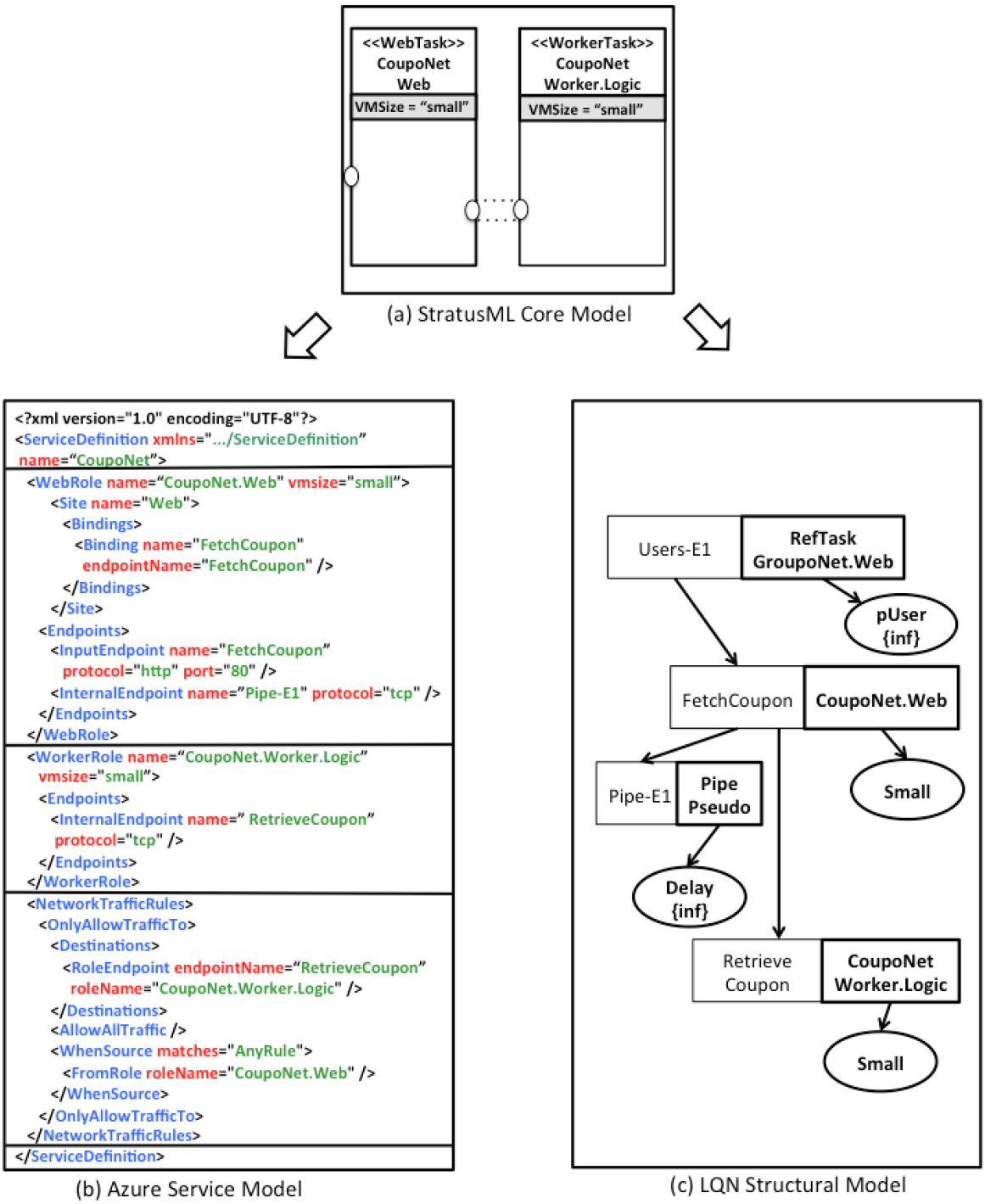


Figure 5.8: Applying the Structural Transformation Rules

are dummy entries; they do not specify any performance parameters (e.g., service time). The name of each dummy entry is determined from the inbound endpoint name. The name appears in the endpoint specification in StratusML (not shown in the figure). The same name appears in the specification of the inbound endpoint as shown in the Azure service definition (Figure 5.8(b)). Following the rule L1-S1-R3, a reference task placeholder with its pure delay processor is created for the web task (CoupoNet.Web). Only one placeholder is created as the web task has only one external endpoint. Based on rule L1-S1-R4 the connection between the two tasks (i.e., CoupoNet.Web and CoupoNet.Worker.Login) is transformed into a pseudo task with a pure delay processor. A synchronous call from the entry of the task with the outbound endpoint to the entry of the created pseudo task is created. The communication overhead will be determined in a later step.

To specify the multiplicity of the processors in addition to the replication of the tasks the rules in step 2 are applied as shown in Figure 5.9. First the multiplicity of each of the processor (i.e., number of cores) is specified by applying the rule L1-S2-R1. Since the size of each of the tasks hosting server is small and the target provider is Azure, the multiplicity, which corresponds to the number of cores, will be set to two and the processor speed factor will be set to 1.6 GHz, which corresponds to the specifications of Azure medium (A2) instance. Second, the replication of each task is specified based on the number of instances of the task according to L1-S2-R2.

The multiplicity of the reference task (#n) refers to the number of active users (population) circulating in the system. This number is specified in the workflow model as shown in the next step. Similarly, the multiplicity of the task refers to the number of concurrent threads. This is a run-time parameter that is specified in the workflow model.

Level 2: Generating LQN Analytical Model

Figure 5.10 shows the complete analytical LQN model that corresponds to the first workflow (W1) in the CoupoNet example. The analytical LQN model has been generated from applying the second level of transformation rules to the fetch coupon workflow.

Using rule L2-R1 the reference task placeholder (RefTask CoupoNet.Web) will be replaced with a concrete reference task. The *entry* parameters of the *reference task* will be specified based on the W1 workload parameters. Here, we have a closed workload with 100 users and 3 seconds think time. Accordingly, the multiplicity of the LQN reference task will be set to 100 and the think time parameter (Z) will be set to 3 seconds.

Using the rule L2-R2 the set of entries (e.g., Fetch Coupon, Retrieve Coupon and Get Info) and the number of calls to and from these entries are determined. Then, depending

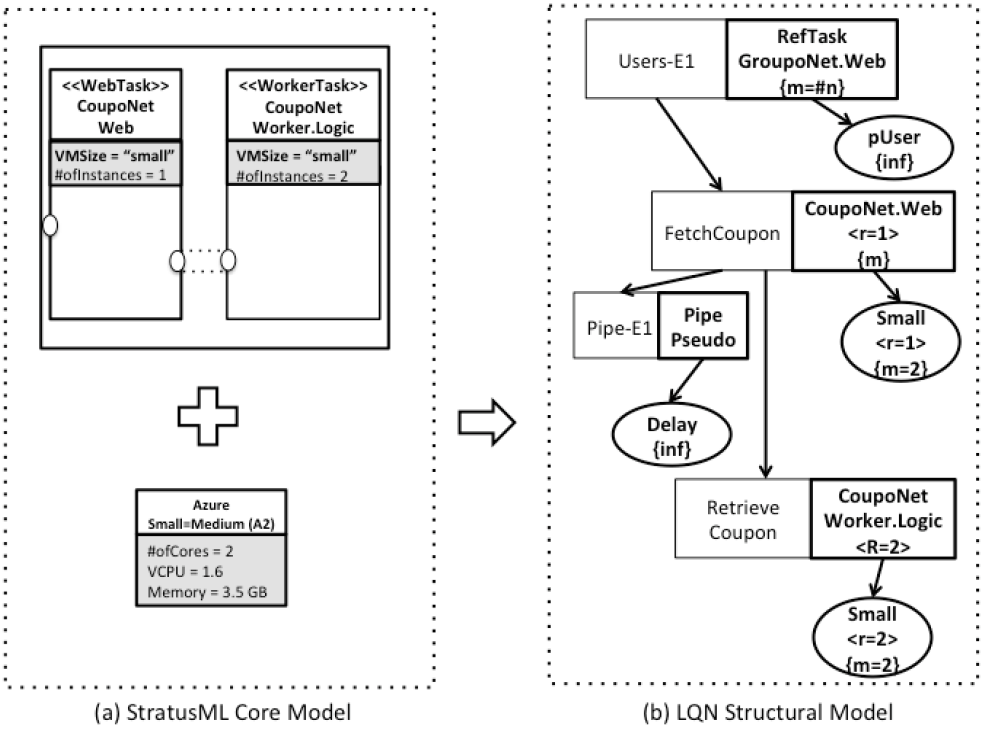


Figure 5.9: Applying the Rules of Multiplicity and Replication

on the different cases explained as part of L2-R2, the entry demand is calculated. The entry demand of the *FetchCoupon* entry is specified by applying the L2-R2-Case1 rule. This is because, the *WebTask* receives external requests from the (reference task) through the external endpoint *FetchCoupon*. Then, it executes a series of activities and returns a response (i.e., case1). The entry demand of the *FetuchCoupon* entry is equal to the sum of the demand of the *WebTask* activities (i.e., (Request Handler, 2 ms), (Fetch Coupon, 4 ms) and (Send Response, 2 ms)). The same rule (i.e., L2-R2-Case1) is used to calculate the entry demand of both the *Retrieve Coupon* and *Get Info* entries.

Rule L2-R3 is used to assign the demand for the entries of both *Pipe Pseudo1* and *Pipe Pseudo2* tasks.

5.5.3 Results

Figure 5.11 shows the complete target LQN model as a result of applying the rules explained in this chapter to the CoupoNet example. Notice that the load-balancing task has been

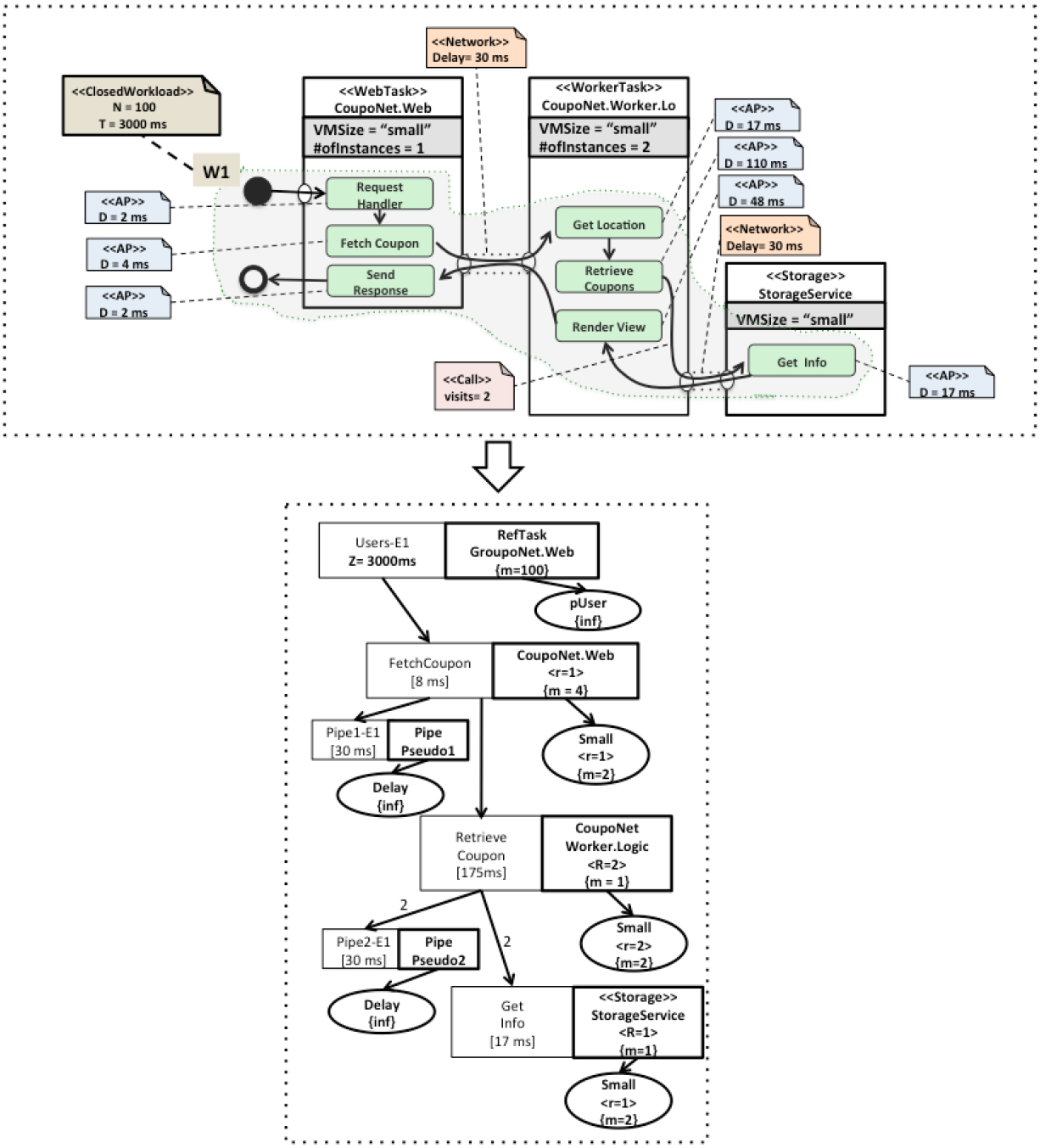


Figure 5.10: Generating Analytical LQN Model for the Fetch Coupon Workflow Scenario

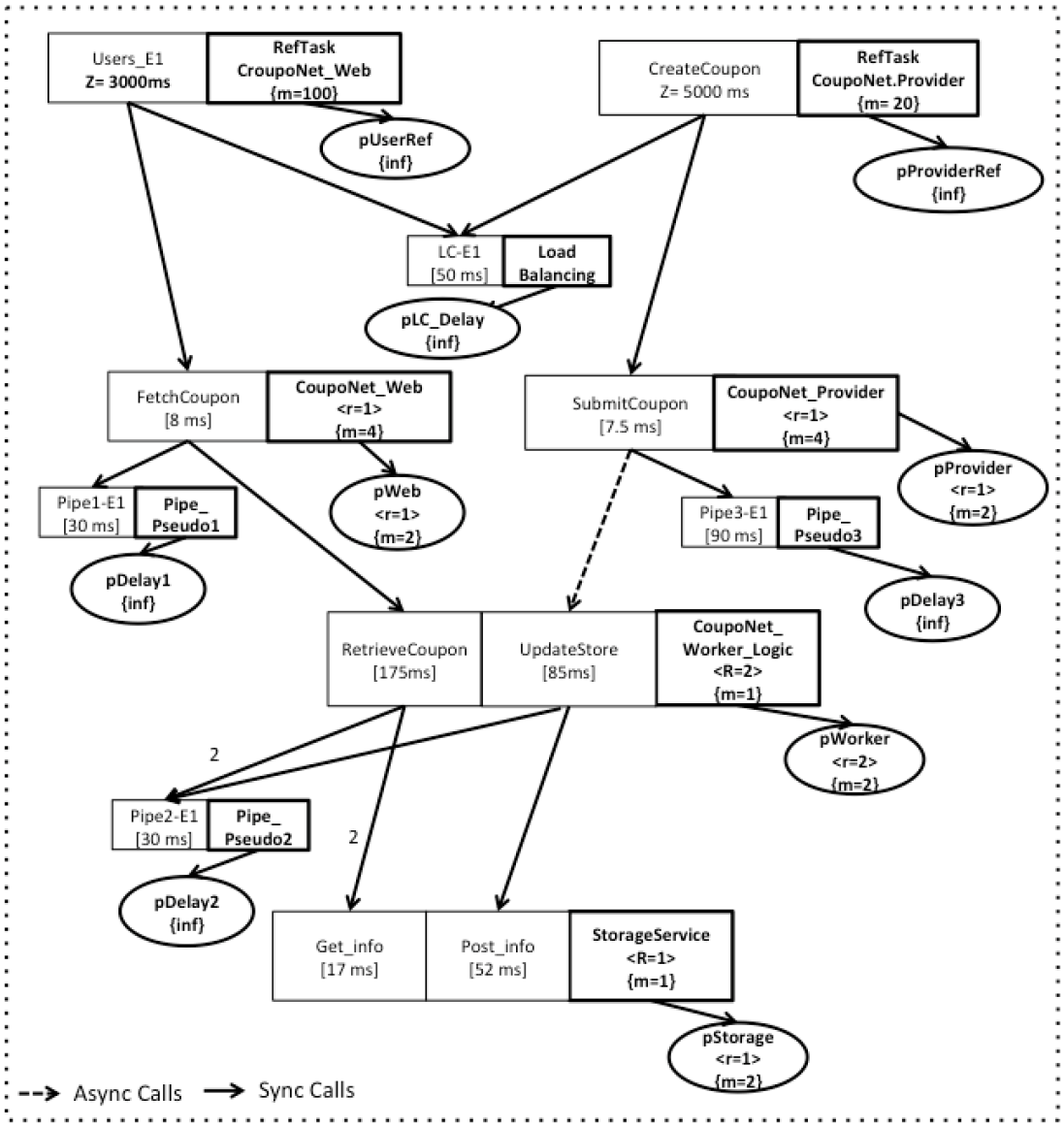


Figure 5.11: The CoupoNet LQN Model

added to compensate for the delay introduced by the target platform load balancing and networking overhead.

To validate the model generated as valid LQN model. We used the W3C XML Schema (XSD) Validator[1]. W3C XML Schema (XSD) Validator can be used to validate an XML file against a schema to check if it conforms to the schema semantics. The tool also checks the well-formedness of the XML file. Using the W3C XML Schema (XSD) Validator we were able to validate the syntactical correctness and the well-formedness of the generated LQN model.

The LQN Solver (LQNS) is used to solve the generated LQN models. Appendix B shows the LQNS output results of the CoupoNet example. Using the results, a system administrator can analyze the response time and throughput of the system. By varying the usage profiles (e.g., number of users) and the deployment configuration, the administrator can also analyze the system performance under different deployment configuration or target providers. They can also study the system bottleneck and adjust the configuration, or define adaptation rules to enact auto-scaling actions.

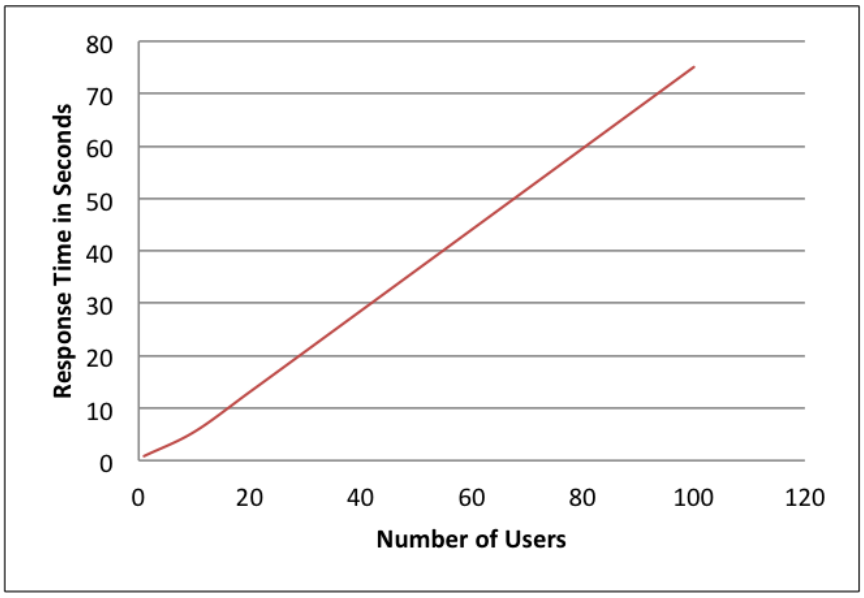


Figure 5.12: The Change in Fetch Coupon Response Time with User Population

Figure 5.12 shows the results of analyzing the system response time for the fetch coupons scenario as the population of the coupon users increases. Such analyses are used to evaluate

[1]http://www.utilities-online.info/xsdvalidation

the system scalability (maximum users supported with a specified response time). Several other analysis techniques and algorithms can also be applied on the generated LQN models. The LQN community created algorithms to identify the location of the system bottlenecks and to tune the multiplicity and replication parameters of software threads and processors to improve the system performance [122]. However, the evaluation of these techniques is out of the scope of this work.

5.6 Summary

This chapter extends the StratusML language to facilitate annotating the cloud models with the runtime performance parameters and generating performance models from the cloud applications on the fly. The chapter argued that LQN models are one of the suitable candidates as a target performance analysis formalism to represent the performance of cloud applications.

The chapter makes the following contributions:

(a) Evaluating performance modeling requirements based on the existing cloud deployed artifacts.

(b) Extending our StratusML framework to capture the application runtime parameters to enable generating analytical performance models from the StratusML models.

(c) Defining a set of transformation rules to automatically transform the StratusML models into LQN formalism.

A case study has been presented to show, in a step by step manner how an annotated StratusML models with performance parameters and how to apply the proposed transformation rules to generate LQN models by utilizing the cloud application service deployment and configuration artifacts.

The meta-models presented in this chapter sum the description of the concepts of the (5+1) view models presented in Chapter 3. Moreover, this chapter completes the realization of the StarusML framework presented in Chapter 4. The next chapter focuses on the framework evolution support. A set of semi-automated techniques for mapping domain concepts will be presented to support the process of meta-model creation, evolution, and transformation.

Chapter 6

Liberate: A Schema Matching Approach to Support Service Migration in the Cloud

"These vendors will never agree on standardized schema, but they have agreed on how they describe their schema."
– Jamie Lewis, president of the Burton Group.

Cloud providers impose using specific schemas to specify the application services for deployment and operation. However, mismatches between providers' schemas hinder application portability and lead to vendor lock-in. To address this problem, this thesis follows a model driven approach that facilitates specifying the cloud service using provider-agnostic constructs, and uses transformation techniques to generate provider specific configuration. As shown in the previous chapters, such model driven approach, depends on the ability to find mappings between the concepts of providers schemas. This is a daunting, error prone and time consuming task that is usually performed by the domain experts manually. To alleviate the efforts needed in this manual process, this chapter adopts a semi-automated schema matching approach. Schema matching can be used to identify the correspondences between the different providers' schemas. This is important for (i) creating domain reference models (i) keeping the reference models up-to-date as new domain concepts evolve, and (iii) creating mappings between the different reference models, or between the reference models and the platform specific models.

Schema matching exploits structural and linguistic similarities between schemas to identify matchings. This is a well known practice for ontology alignment and database

migration [145] in many domains, e.g: linked-data, e-commerce, and bioinformatics [47]. However, it has been less frequently exploited in the cloud domain. This is because, not all cloud providers' schemas follow the same structural conventions. Moreover, the terms in these schemas use industrial jargons that may not share linguistic similarities. In this chapter, Liberate: a novel semi-automated schema matching approach has been devised to deal with these challenges. Liberate applies a generic structural similarity based on a modified similarity-flooding algorithm [103], and utilizes web search results, and gloss-based semantic similarity to incorporate domain knowledge in element-based similarity evaluation.

The rest of this chapter is organized as follows: Section 6.1 illustrates why schema matching is required and how generic schema matching solutions may fail to address the mismatch problem in the cloud. Section 6.2 presents our proposed solution, and how it has been realized. Section 6.3 evaluates the different matchers proposed against public schemas of two major cloud providers (i.e., Windows Azure and Google App Engine) and private schemas of two reference cloud models (StratusML and TOSCA). Finally, an executive summary and suggestions for future work is presented in Section 6.4.

6.1 Preliminaries

Several approaches could be used to address the vender lock-in problem in the cloud [68]. Most of these approaches are based on model driven engineering. Regardless of the approach selected, finding alignments between the different providers' schemas, standards and models is unavoidable for mismatch resolution. This section shows some examples of the mismatches between cloud providers' schemas and highlights the challenges of automatic schema matching in the cloud domain.

6.1.1 Mismatches Between Platform Providers

Migrating an application from one cloud platform to another requires transforming the configuration space artifacts that are used to deploy the application from the the first provider format to the target format. This process requires identifying the related elements between the source and target providers' models.

Figure 6.1 shows a sample of two configuration files for public cloud providers: The first (6.1a) describes the deployment configuration of an application on Windows Azure. The second (6.1b) specifies the deployment configuration of the same application under

```
<ServiceConfiguration>
   <Role name="ShoppingCartProcessing">
   <Instances count="2"/>
      <ConfigurationSettings>
         <Setting name="DataConnection"
            value="UseDevelopmentStorage=true"/>
      </ConfigurationSettings>
   </Role>
</ServiceConfiguration>
```

(a) Example of Azure Service Configuration

```
<appengine-web-app>
  <application>My App</application>
  <module>ShoppingCartProcessing</module>
  <version>uno</version>
  <instance-class>F4</instance-class>
  <manual-scaling>
    <instances>2</instances>
  </manual-scaling>
</appengine-web-app>
```

(b) Example of GAE Configuration

Figure 6.1: A Sample of Public Providers' Configuration Artifacts

Google Application Engine (GAE). Likewise, Figure 6.2 shows a sample of two configuration artifacts that are conforming to the specifications of two different cloud languages: The first (6.2a) shows a service template file that is specified according to TOSCA standard language [17], while the second (6.2b) shows a service description file that is specified according to the CloudML language [57].

```
<ServiceTemplate name="MyFirstMachine">
  <NodeTemplate type="csp:Machine">
   ...
       <MachineConfiguration>
         <cpu>4</cpu>
         <memory>64000</memory>
         <disk>
           <capacity>512000</capacity>
           <format>NTFS</format>
           <initialLocation>C:</initialLocation>
         </disk>
       </MachineConfiguration>
  </NodeTemplate>
</ServiceTemplate>
```

(a) Example of TOSCA Service Template File

```
<serviceDescription:ServicesType version="AmazonM1">
...
  <nodes ID="M1 Medium Instance 1">

    <ram size="3.75" unit="GB"/>
    <cpu Architecture="64Bit" Cores="2"/>
    <functionality functionality="Server"/>
    <operatingSystem operatingSystem="Ubuntu"/>

  </nodes>
  <locations country="US"/>
  <locations country="Europe"/>
...
</serviceDescription:ServicesType>
```

(b) Example of CloudML Service Description

Figure 6.2: A Sample of Standard Languages' Configuration Artifacts

By analyzing the artifacts in Figures 6.1a and 6.1b, we notice that both Azure and GAE files are essentially describing the same information, but using different provider specific concepts and file structure. For example, by referring to Appendix C, the concepts of a *module* and a *role* are very similar. Both describe a basic modular component (i.e., VMI) in the cloud configuration space from which a cloud application is composed. For each role and module, the configuration files specify the resource requirements and number of instances required. Similarly, Figures 6.2a and 6.2b show that there is great similarity

between TOSCA service template and CloudML service description. Again, both describe the same information, but each adheres to the format of the that standard format.

Mismatching between cloud schemas is inescapable. It occurs due to: (i) the poor coordination between providers, as they develop their platforms; (ii) the continuous updates to the schemas, as a result of adding new features; in addition to (iii) branding and positioning practices as part of the providers' marketing campaigns. In order to migrate the application from one provider to another or from one format to another, the mismatch between the different schemas needs to be resolved.

6.1.2 Schema Matching in the Cloud

Even though schema matching has obvious utility to uncover mismatch between schemas, schema matching has been poorly exploited within the cloud domain. This section explains why traditional schema matching approaches may fail in the cloud.

As explained in Section 2.2.4, most schema matching approaches use linguistic techniques to evaluate the similarity between schema elements. Unfortunately, techniques that are solely linguistic may fail to uncover matches across cloud schemas.

Table 6.1: Path Length Pairwise Similarity

Terms	Role	Module	BeansTalk	Task	Node
Role	1	0.12	0.07	0.33	0.14
Module	0.12	1	0.10	0.11	0.14
BeansTalk	0.07	0.10	1	0.06	0.10
Task	0.33	0.11	0.06	1	0.36
Node	0.14	0.14	0.10	0.09	1

For example, Table 6.1 shows a similarity matrix[1] obtained by applying *path length pairwise similarity* [81] to a set of similar cloud domain concepts extracted from five cloud schemas (i.e., Azure, GAE, AWS, StratusML, TOSCA). The domain definition of each of these concepts is provided in Appendix C. Path length is a linguistic similarity measure between zero and one that depends on node-counting. The path length score is inversely proportional to the number of nodes along the shortest path between the synsets. If two synsets are the same, then the length of the shortest path between them is 1, which means 100% similarity.

[1]The similarity values in the matrix have been computed using the WordNet similarity web interface implementation by Ted Pedersen and Jason Michelizzi http://maraca.d.umn.edu/cgi-bin/similarity/similarity.cgi

As explained earlier, the concept of "Role" in Azure is similar to a "Module" in GAE. The same concepts can also be mapped to "Beanstalk" in AWS, "Task" in StratusML and "Node" in TOSCA. However, the similarity matrix shows very weak linguistic semantic correlation between the concepts, even though these concepts refer to the same thing within the cloud domain. Other linguistic semantic techniques may produce worse results. For example, using Jiang & Conrath [81] measure, the similarity between a "Role" and "Beanstalk" is zero.

In a nutshell, there is a need for techniques that are able to capture correlation between concepts beyond linguistic syntactic or semantic similarity. These techniques should be able to capture the similarity by utilizing domain knowledge.

6.2 The Liberate Approach

To address the schema matching problem in the cloud, we introduce Liberate. An overview of the Liberate approach is illustrated in Figure 6.3.

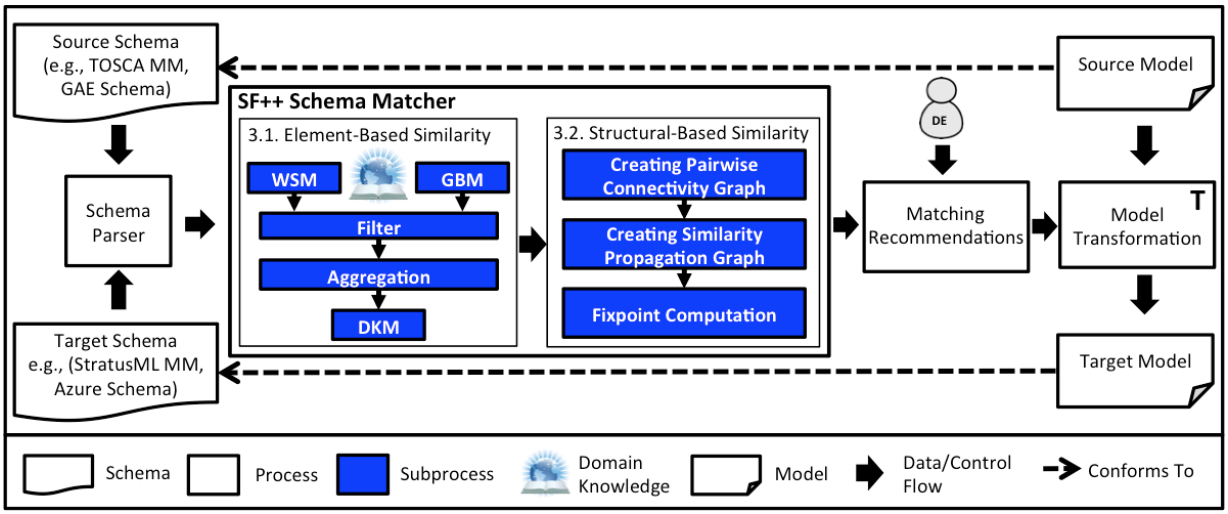


Figure 6.3: Liberate - A Schema Matching Approach for Model-Driven Migration

Given two providers' schemas, the schemas are first parsed, then passed to a semi-automated schema matching process (i.e., the SF++ schema matcher). More about the SF++ will follow. For now, the semi-automated schema matching process generates a set of matching recommendations. A domain expert (DE) then reviews the matching results

to identify the valid matches. The valid matches are used to create a set of transformation rules to enable transforming any source model that conforms to the first provider schema into a target model that conforms to the second provider schema. The valid matches can also be used to create a domain reference model, or update an existing one. This chapter concentrates on the schema matching process (SF++ Schema Matcher). The following two requirements need to be considered in the design of the schema matching process so that it can be applied in the cloud domain:

(i) The process should be generic enough to be applied to a wide range of providers' schemas.

(ii) The process should incorporate domain and provider specific knowledge.

To meet the aforementioned requirements, we adopt a schema matching process that is inspired by the Similarity-Flooding (SF) algorithm proposed by Melnik et al. [103]. We call the new process the SF++ schema matching process.

The original SF algorithm is a generic graph-based schema matching technique based on the intuition that similar elements have similar neighbors. It combines element and structural based similarity. Given two directed, labeled graphs, the SF algorithm matches each pair in the graph using string-based similarity. Then, it applies fixpoint [103] computations to propagate similarities between adjacent elements.

Figure 6.3 shows that similar to the SF algorithm, the SF++ consists of two phases (i.e., Element-Based and Structural-Based Similarity). Moreover, it creates pairwise connectivity graph and similarity computation graph, and uses fix-point computation to propagate similarities between the neighbor elements. However, the SF++ extends the similarity-flooding algorithm by:

(i) Replacing the similarity computation step that is based on string-matching with a more advanced similarity metric that takes domain knowledge into consideration. We call this new metric the Domain-Knowledge-Matcher (DKM). The DKM aggregates the results of calculating two metrics that incorporate domain knowledge from two different sources in the matching process: the Web-Similarity-Matcher (WSM)[68] and the Gloss-Based-Matcher (GBM).

(ii) Generalizing the (SF) algorithm to accept non-labeled directed acyclic graphs instead of directed labeled graphs to create a pairwise connectivity graph (PCG).

The details of the SF++ are explained next.

6.2.1 The SF++ Element-Based Similarity

The SF++ uses two different matching metrics (i.e., WSM and GBM) to incorporate domain knowledge. Both calculate the similarity between each pair of elements of the source and target schemas. Accordingly, each produce a similarity matrix. The WSM is a statistical similarity matching technique based on web-search results. Hence, it depends on the availability of online content. On the other hand, the GBM compares the similarity between concepts based on the domain definition (gloss) attached to each concept, as provided by schema providers. This section explains each of these metrics and how they are filtered and aggregated into a DKM.

The Web Similarity Matcher

The WSM is a similarity metric that uses search engine results to infer the similarity between concepts. It is based on the Normalized Web Distance ($\texttt{NWD}_{\texttt{SE}}$) [59] that is a more generic form of the Normalized Google Distance (NGD) [34]. The difference is that the $\texttt{NWD}_{\texttt{SE}}$ uses the results obtained from *any* search engine (SE), such as Bing, Yahoo or other engines, not only Google. The $\texttt{NWD}_{\texttt{SE}}$ is a metric that measures the distance between two terms based on their co-occurrence in search engine results. Given two terms, τ_1 and τ_2, and the total number of pages in the search engine index N, the $\texttt{NWD}_{\texttt{SE}}$ calculates the relatedness between the two terms by finding the number of hits (i.e., page counts) returned by the search engine that contains each of the terms separately $f(\tau_1)$, $f(\tau_1)$, and both terms together $f(\tau_1||\tau_2)$ (i.e., the concatenation $||$ between τ_1 and τ_2). Accordingly, it applies Equation (1) to calculate the distance between the two terms. A zero distance indicates 100% correlation, while ∞ means no correlation.

$$\texttt{NWD}_{\texttt{SE}}(\tau_1, \tau_2) = \frac{\max\{\log f(\tau_1), \log f(\tau_2)\} - \log f(\tau_1||\tau_2)}{\log N - \min\{\log f(\tau_1), \log f(\tau_2)\}} \tag{6.1}$$

It is worth mentioning that Equation (1) measures distance between the terms rather than similarity. Moreover, the output of Equation (1) is not bounded (i.e., the distance ranges from 0 to ∞). To transform distance into similarity and normalize the results to be bounded between 0 and 1, we use the exponential function $g(x) = e^{-x}$ in composition with the $\texttt{NWD}_{\texttt{SE}}$. The exponential function is a decreasing transformation function, where $(g \circ \texttt{NWD}_{\texttt{SE}}) : [0, \infty) \mapsto [1, 0)$. We refer to this composition as the Web-Similarity-Matcher (WSM), shown in Equation (6.2). The value of the WSM is bounded between zero and one, where zero indicates no similarity and one is 100% similarity.

$$\texttt{WSM}_{\texttt{SE}}(\tau_1, \tau_2) = e^{-\texttt{NWD}_{\texttt{SE}}(\tau_1, \tau_2)} \tag{6.2}$$

To calculate the WSM value, *three queries* to the search engine API are needed. In each case, the page counts is retrieved, and then substituted in the equation. For example, searching for the keyword "Azure" on Google returns 22,000,000 pages that contain this term. Similarly, searching for the keyword "AWS" returns 34,600,000 pages. Searching for both terms together returns 2,390,000 pages that represent their co-occurrence. All these numbers are based on a search preformed at the time of writing this thesis. Searching the same terms today may return different results. By referring to Equation (6.2), there is still one missing value, which is the the total number of pages (N) in the search engine index. In this example, the value used for Google is 30 trillion. Based on these values NWD("Azure", "AWS") ≈ 0.2 and WSM("Azure", "AWS") ≈ 0.82. However, since the value of N changes continuously and is different from one search engine to another, and in order to avoid using a hardcoded value, in our implementation we use the following simple trick that enables us to infer the number of pages indexed at run time. First, we query the search engine for the exact match of a common term that appears in almost every web page (e.g., "the"). Assuming that the average frequency for this term is constant per page, then we can find the total page count (N).

Table 6.2: WSM Pairwise Similarity

Terms	**Azure Role**	**GAE Module**	**AWS BeansTalk**	**StratusML Task**	**TOSCA Node**
Azure Role	1	0.75	0.78	0.37	0.58
GAE Module	0.75	1	0.89	0.36	0.67
AWS BeansTalk	0.78	0.89	1	0.64	0.65
StratusML Task	0.37	0.36	0.64	1	0.36
TOSCA Node	0.58	0.67	0.65	0.36	1

Table 6.2 shows the results of applying the WSM to the same cloud domain concepts in Table 6.1. Each cell in the table is obtained by calculating the WSM between the two concepts that intersect in that cell. To limit the search within a specific domain context: (i) we specify the domain name before the term to be searched (e.g., "*Azure* Role" instead of "Role"). Moreover, (ii) we set a number of tuning parameters, in the API calls, to limit the search within a specific time period and remove duplicates.

Table 6.2 demonstrates the ability of the WSM to identify correlation between the domain concepts even when these concepts do not share linguistic similarity. However, while the WSM is able to uncover the correlation between an "Azure Role" and "AWS BeansTalk", it is unable to discover the strong correlation between "TOSCA Node" and "StratusML Task", or between "Azure Role" and "StratusML Task". One possible reason is that the WSM relies on the availability of a rich online content pertaining to the given schemas. The availability of such content is not always guaranteed. For example in this case the availability of online content that combines StratusML and TOSCA or Azure is limited. Moreover, this content is sometimes misleading specially in the case of private and new providers' schemas. The next section discusses another matcher that is able to incorporate domain knowledge regardless of the availability of online content.

The Gloss-Based Matcher

In ontology engineering, a gloss refers to a description for the meaning of a concept that renders important knowledge for understanding [79]. A gloss is represented in *short sentences* in natural language. It is part of schema documentation and a reliable source of domain knowledge in provider-driven schemas and standard reference models that target the public use.

Different approaches have been devised to measure similarity between text segments and short sentences [1, 106, 134]. Achananuparp et al. [1] categorized these approaches into three categories: (i) *word overlap measures*, which compute the similarity of two sentences based on the number of words shared between them, (ii) *TF-IDF measures*, which refer to a family of statistical techniques (e.g., vector similarity) that use *Term Frequency* (i.e., the number of times a word appears in a document) and *Inverse Document Frequency* (the frequency of a word in a corpus) to infer the importance of a word in a document, and (iii) *linguistic measures*, which use the words' semantic relations and syntactic composition to find the similarity between sentences.

Based on the findings of [1], generally speaking linguistic measures outperform word overlap and TF-IDF measures for identifying similarities between arbitrary short sentences. In Liberate, we adopt a variation of the Mihalcea metric [109]. Mihalcea metric is a linguistic measure that combines metrics of word-to-word similarity and word specificity[2].

In our approach, we use a simplified version of Mihalcea metric that uses the sentence length to normalize the results, instead of word specificity [96]. The later metric is simpler

[2]Word specificity is determined using the inverse document frequency (idf), defined as the total number of documents in the corpus divided by the total number of documents including that word [109]

and has been proved to be a good indicator of the semantic similarity of sentences [1].

Following the same approach, given two input sentences (i.e., domain definitions in our case). First, each sentence is partitioned into a bag of words b_1 and b_2, where function words (e.g., articles, pronouns, conjunctions, auxiliary verbs) are removed. Then, each word w is assigned a corresponding part of speech tag (e.g., noun, verb, preposition) using a Brill tagger [23]. After that, Porter stemming algorithm [132] is applied to remove the common morphological and inflexional endings of words. Then, the word to word similarity is calculated to measure the similarity of each word in b_1 and all words in b_2, and vise versa. Different corpus-based and knowledge-based techniques for word semantic similarity can be used (e.g. WordNet similarities or Latent Semantic Analysis). Particularly, in this study WordNet similarities is adopted in the same way it is implemented in the SEMILAR toolkit [141]. Finally the similarity score between the two bags of words (b_1 and b_2) is calculated as shown in the following equation:

$$GBM(b_1, b_2) = \frac{\sum_{w \in b_1} maxSim(w, b_2) + \sum_{w \in b_2} maxSim(w, b_1)}{|b_1| + |b_2|} \tag{6.3}$$

where, $maxSim(w, b_2)$ is the highest semantic similarity obtained between the word ($w \in b_1$) and all the words in b_2 that belong to the same class (e.g., verb, noun). $|b_1|$ and $|b_2|$ are the number of words in the first and second word bags respectively (i.e., the length of the sentences after removing function words).

Table 6.3: Gloss-Based Pairwise Similarity

Terms	**Azure Role**	**GAE Module**	**AWS BeansTalk**	**StratusML Task**	**TOSCA Node**
Azure Role	1	0.57	0.48	0.59	0.44
GAE Module	0.57	1	0.56	0.60	0.49
AWS BeansTalk	0.48	0.56	1	0.56	0.42
StratusML Task	0.59	0.60	0.56	1	0.46
TOSCA Node	0.44	0.49	0.42	0.46	1

Table 6.3 shows the results obtained by applying the GBM to the same concepts that we compared earlier. However, in this comparison the GBM uses the domain glosses

corresponding to each of these concepts as defined by the cloud providers. The list of concepts and their glosses is provided in Appendix C.

The GBM score gives values between zero and one. The results in Table 6.3 show that despite the absence of syntactic or linguistic semantic between the terms compared, the domain gloss can provide a good sense of similarity between the concepts. In most cases, the domain glosses of the equivalent domain concepts are at least 50% similar. Moreover, the GBM does not rely on web search results, hence it can be applied equally for new schemas and private ones. Unfortunately, the GBM requires the concepts domain glosses that are not always available.

In Liberate, we compare the concept glosses of the source and the target schemas. The similarity matrix produced is then filtered and aggregated with the WSM into a Domain-Knowledge-Matcher as shown next.

Filtering and Aggregation

The output of each of the element-based similarity matching techniques is a similarity matrix that shows the similarity between each element in the first schema and the elements of the second schema. As explained earlier, each metric provides a different perspective and has its pros and cons. The goal of this section is to aggregate the results obtained through the WSM and GBM into a single metric: the domain knowledge matcher (DKM). As there is no golden rule of how to combine these metrics, we are aiming at providing a configurable similarity score that combines these metrics based on the current available knowledge and the degree of contribution of each metric.

Before aggregating the results, the first step is to filter insignificant matches. This process is usually referred to as *alignment extraction*. Alignment extraction techniques can broadly be divided into two categories: (i) threshold-based extraction, and (ii) mapping-based extraction. *Threshold-based* techniques select matches with a similarity greater than a predefined value. Whereas, *mapping-based* techniques use mapping algorithms to select matches. Examples of mapping algorithms include, the Maximum Weight Bipartite Graph (MWBG) [114], the Perfect Monogamy (PM) and the Stable Marriage (SM) [97, 103], and Hierarchical Mapping (HM) [142].

In this work, we apply a configurable threshold-based filtration technique. In which, we use the percentile to filter the results below a given percentage in the range of similarity results obtained for each pair of concepts (c and t) in the similarity matrix between the source and target schema.

After filtering the results that are obtained from the different similarity matrices, the similarity matches are aggregated into a Domain Knowledge Matcher (DKM) using the weighted geometric mean, as in the following equation:

$$DKM(c,t) = \sqrt[\Sigma_{i=1}^{n}\mu_i]{WSM(c,t)^{\mu_1} \times GBM(c,t)^{\mu_2} \times \dots} \tag{6.4}$$

The geometric mean is defined as the n^{th} root of the product of n numbers. The geometric mean produces consistent rankings independent of the reference data based on which measurements are normalized; it is generally considered as the most reliable way to calculate the mean for normalized measurements [49]. Furthermore, by using a weighted geometric mean different preferences regarding the aggregated metrics can also be taken into account. For example, if the domain glosses are available and trustworthy, the user may give the GBM more weight, so that it can contribute more to the DKM. The user can give the WSM a weight of 1 and GBM a weight of 2 and hence the DKM will be the third root of the product of the WSM and GBM, each raised to its corresponding weight . By default equal weights (μ) are given for all metrics. However, both the threshold and the weights have been implemented as configurable parameters that can be tuned at runtime. Selecting the optimal values for those parameters is out of the scope of this work.

6.2.2 The SF++ Structural-Based Similarity

As explained earlier, the work in this chapter extends the Similarity Flooding (SF) algorithm in two ways: (i) it incorporates the domain knowledge in the matching process to facilitate generating the initial alignments between the schema elements, and (ii) it generalizes the (SF) algorithm to accept non-labeled directed graphs instead of directed labeled graphs to create a *pairwise connectivity graph* (PCG). This section focuses on how the node similarity results can be used to create a PCG, how to convert a PCG into a *propagation graph*, and how to apply *fixpoint computation* to propagation graphs.

Step1: Creating Pairwise Connectivity Graph

In order to apply the SF algorithm, each of the schemas that need to be structurally compared must first be consolidated into a Pairwise Connectivity Graph (PCG). The PCGs are then converted into propagation graphs. A PCG between two graphs G_1 and G_2 is defined as follows: $((x_1,y_1),r,(x_2,y_2)) \in PCG \iff (x_1,r,x_2) \in G_1$ and $(y_1,r,y_2) \in G_2$. Therefore, a PCG is a set of triples (n_1,r,n_2) such as: $n_1 = (x_1,y_1)$ and $n_2 = (x_2,y_2)$,

where x_1 and x_2 are labeled nodes of G_1, y_1 and y_2 are labeled nodes of G_2, and r is a labeled arc.

While using labels on arcs simplifies the process of creating a PCG; labels are not always available, and may limit the ability of the algorithm to detect structural similarities in schemas that use different naming standards for specifying the types of relationships between schema entities. For this reason, this section generalizes the (SF) algorithm to accept graphs with no labels on the arcs. This has been accomplished through extending the PCG algorithm to enable it to infer PCG graphs from the available information: the similarity matrix generated from element-based similarity matching, and the two schema graphs.

input : Graphs: G_1, G_2 and Similarity Matrix S
output: PCG Graph
foreach *Node* $n \in G_1$ **do**
  C_n: the set of children of node n;
  foreach *Node* $x \in C_n$ **do**
    S_x: the set of nodes $\in G_2$ similar to x;
  end
  S_n: the set of nodes $\in G_2$ similar to n;
  foreach *Node* $m \in S_n$ **do**
    C_m: the set of children of node m;
    foreach *Node* $y \in C_m$ **do**
      S_y: the set of nodes $\in G_1$ similar to y;
    end
  end
  $u_{n,y} := C_n \cap S_y$, where $C_n \neq \phi$ & $S_y \neq \phi$;
  $v_{m,x} := C_m \cap S_x$ where $C_m \neq \phi$ & $S_x \neq \phi$;
  $\text{PCG}_n : \bigcup_{\forall m \in S_m} \text{PCG}_{n,m}$,
  $| \text{PCG}_{n,m} : (n, m) \mapsto u_{n,y} \times v_{m,x}, \forall y \& x$;
end
$\text{PCG} \longleftarrow \bigcup_{\forall n \in G_1} \text{PCG}_n$;

Figure 6.4: A Generic PCG Algorithm

Figure 6.4 shows our PCG algorithm. To explain the algorithm, let's consider the example in Figure 6.5, which consists of (i) two directed graphs G_1 and G_2, each represent a different schema. (ii) an arbitrary similarity matrix that corresponds to the output of the element-based similarity analysis after filtration (i.e., only values with similarity $\geq \theta$ are considered), and (iii) the outcome PCG graph. The algorithm starts from the top

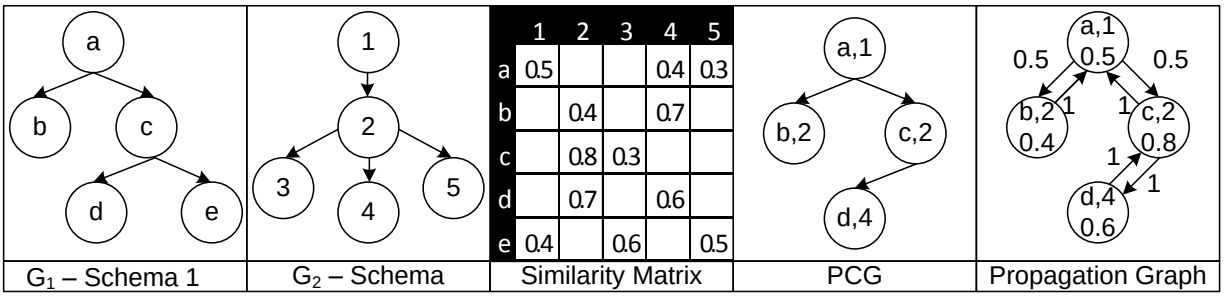


Figure 6.5: A PCG Example

(root node). For each node in the graph it retrieves the directly connected children of that node, and the set of similar elements to that node from the filtered similarity matrix. For example, in Figure 6.5 the children of the node 'a' are $C_a = \{b, c\}$, and the nodes similar to 'a' form G_2 are $S_a = \{1, 4, 5\}$. For each child node returned, the algorithm finds the nodes that are similar to it in the other graph $S_b = \{2, 4\}$, $S_c = \{2, 3\}$. Similarly, for each similar node returned, the algorithm returns the set of directly connected children $C_1 = \{2\}$, $C_4 = \phi$ and $C_5 = \phi$, where ϕ refers to empty set. Lastly, for each of the children returned from the second schema, the algorithm uses the similarity matrix to return the similar nodes from the first schema $S_2 = \{b, c, d\}$.

To find the pairwise connectivity graph for node 'a' ($\texttt{PCG}_\texttt{a}$), first the algorithm finds the intersection between the children set of 'a' (C_a) and each of the similarity sets of the corresponding children sets in the second graph (e.g., in this example its only one set S_2), $v_{a,2} = C_a \cap S_2 = \{b, c\}$. Then it finds the intersection between the children sets of 'a' similar nodes in G_2 (e.g., C_1) and the similarity sets of 'a' children (e.g., S_b and S_c). $u_{1,b} = C_1 \cap S_b = \{2\}$ and $u_{1,c} = C_1 \cap S_c = \{2\}$. Based on these results, $\texttt{PCG}_\texttt{n}$ will be given by the union of all PCG triples that has the node n from G_1 and one of its similar m correspondences in G_2 as a vertex, and the map of this vertex to the cartesian product of the sets ($v_{n,y} \times u_{m,x}$). In our example $\texttt{PCG}_\texttt{a}$ has only one source (i.e., (a,1)) and two destinations (i.e., (b,2) and (c,2)) vertices. $\texttt{PCG}_\texttt{a} = \{\texttt{PCG}_{\texttt{a},1}\} = \{\{(a,1) \longmapsto (b,2)\}, \{(a,1) \longmapsto (c,2)\}\}$.

The final PCG is the union of all $\texttt{PCG}_\texttt{n}$ for all the nodes in G_1. After creating the PCG the rest of the SF algorithm will be applied in the same way as proposed by Melnik, Garcia-Molina and Rahm[103]. However, to keep the chapter self-contained, step2 and 3 of the SF algorithm are briefly explained.

Step2: Creating Similarity Propagation Graph

A similarity propagation graph (SPG) is a two way directed weighted graph. It can be generated from a PCG, by replacing the one way directed arcs between the PCG nodes with two way weighted arcs using the following steps: (i) For each arc $(n_1, r, n_2) \in$ PCG, a reverse arc (n_1, r, n_2) is created. (ii) For each node $n \in$ PCG the number of fan-outs (F_{out}) (i.e., number of arcs out of that node) are computed. The weight on each arc that flows out of that node (a.k.a., propagation coefficient) is given by ($\frac{1}{F_{out}}$). For instance, for the PCG in Figure 6.5 , the node (a,1) has two fan-outs, each will have a propagation coefficient of 0.5. On the other hand, the node (b,2) has one fan-out, hence its propagation coefficient is 1.

Step3: Fixpoint Computation

Recall that each node n in a PCG represents a pair of label nodes $n = (x, y)$, where the first element belongs to the first schema $x \in G_1$ and the second element belongs to the second schema $y \in G_2$. Let σ_n^i be the similarity measure between the two elements x and y of n at iteration i. The initial similarity between any two elements is given by σ^0. The fixpoint computation starts by assigning initial similarity values to each node in the propagation graph. Then, it propagates the initial values through the graph.

The initial similarity values are assigned according to the values obtained from element based similarity (Section 6.2.1). To propagate the similarity values, the SF algorithm performs iterative computations. In each iteration, new similarity values (σ^{i+1}) are computed. The process is repeated until a stable similarity value is reached. Malnek et. al. [103] defined four different ways (i.e., formulas) to perform fixpoint computations. These ways are differentiated based on the weights given to the similarity propagation (i.e., the propagation of neighbors similarity in the propagation graph) in comparison to the node initial and current similarity values. The SF++ adopts Equation (6.5) as it gives the initial similarity σ^0, and the current similarity σ^i values equivalent importance. According to [103] this formula usually has the best results and the best convergence properties.

$$\sigma_n^{i+1} = normalize(\sigma_n^0 + \sigma_n^i + \sum_{c \in C_n} \omega(n, c) \times (\sigma_c^0 + \sigma_c^i)) \tag{6.5}$$

Given a propagation graph with N nodes. Equation (6.5) finds the similarity value of node $n \in N$ at iteration $(i + 1)$. This formula uses node n initial similarity σ_n^0 and its current similarity value σ_n^i, in addition to the similarity propagation from the neighbors of node n. The similarity propagation to node n is the summation of the similarity of all

directly connected neighbors ($c \in C_n$) to the node n each multiplied by the propagation coefficient $\omega(c, n)$ from the child node c to n. At the end of each iteration, the similarity values are normalized, by dividing them by the maximum similarity value computed in that iteration.

Using Equation (6.5), the similarity value after the first iteration for the node (a,1) in the propagation graph in Figure 6.5 is given by: $\sigma^1_{(a,1)} = 0.5 + 0.5 + 1 * (0.8 + 0.8) + 1 * (0.4 + 0.4) = 3.4$. Similarly, $\{\sigma^1_{(b,2)} = 1.3, \sigma^1_{(c,2)} = 3.3, \sigma^1_{(d,4)} = 2.8\}$. After the first iteration, all values will be normalized by dividing the results by the maximum similarity value of the first iteration, which is 3.4. Hence, the new values after normalization are: $\{\sigma^1_{(a,1)} = 1, \sigma^1_{(b,2)} = 0.382, \sigma^1_{(c,2)} = 0.97, \sigma^1_{(d,4)} = 0.823\}$.

The fix-point computation is performed iteratively until no gain above a particular ϵ value is provided by the last iteration. If the computation does not converge, the process terminates after a maximal number of steps. For more information on the complexity and the convergence the reader can refer to Melnik et. al. [103].

6.2.3 Implementations

To implement the matchers proposed in this work, Liberate extends the Open Information Integration (OpenII) Harmony framework [112]. OpenII Harmony is an open source schema-matching framework that was implemented in collaboration between the MITRE Corporation and several industrial (e.g., Google, Microsoft, IBM, Yahoo) and academic collaborators (e.g., University of Wisconsin, University of Pennsylvania, University of California, Berkeley).

OpenII Harmony is a mature and extensible project. It supports multiple schema formats (e.g. XSD) and multiple output formats (e.g., spreadsheets, and cvs) for matching results. In addition, it provides high-end GUI that allows users to refine the suggested mappings, confirm or reject matches, add annotations, and specify transformation functions. All these features make it suitable to deal with the complex cloud schemas that come with different formats, spread into multiple files, and may require non-trivial transformations.

Our implementation uses the OpenII loaders, mappers and code-generator. It also extends the matcher module by adding four extra matchers: three element-based matchers (i.e., web-semantic-matcher, gloss-based-matcher, and domain-knowledge-matcher) and one structural matcher (i.e., the modified similarity flooding matcher). All the implemented element-based matchers extend the matcher class. They have been implemented to work as composite matchers that can work togather and with other matchers implemented within

the OpenII framework. This facilitates combining and comparing the results of different matchers.

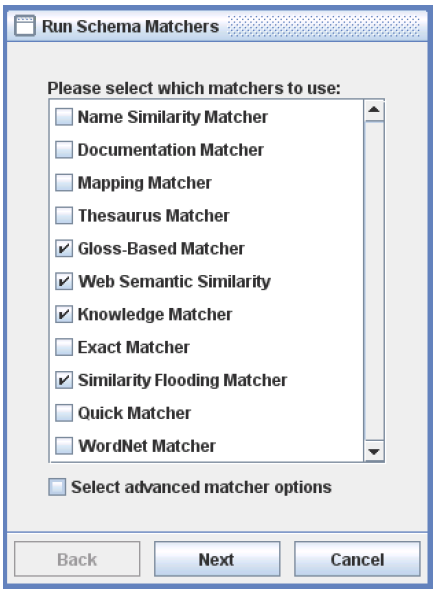


Figure 6.6: Configuring a Composite Matcher

Figure 6.6 shows how the implemented matchers can be used as composite matchers together and with other matchers within OpenII.

The WSM has been implemented to use several search engines. One of the main issues that we had to deal with is to reduce the number of calls to the search engines' APIs. This is because, calling a search engine API and waiting for a response is the most time consuming operation in the WSM process. Moreover, most search engines limit the number of calls per second and the maximum number of calls per month or day for each user. Assuming that the two schemas S_1 and S_2 have the same number of elements (n), the number of search requests required by the WSM process, is given by:

$$\texttt{TotalSearchRequests}_{\texttt{withoutcache}} = 3 \times n^2$$

To reduce this number, we implemented a caching strategy. Using caching, we were able to reduced the number of calls by order of three as shown in the following equation:

$$\texttt{TotalSearchRequests}_{\texttt{withcache}} = n \times (n + 2)$$

While from a theoretical point of view, the number of calls will still have a polynomial (quadratic) growth with respect to the number of elements in the schemas, the gain achieved in practice is worth the effort.

On the other hand, the Gloss-Based Matcher has been implemented by extending the sentence-to-sentence similarity from the SEMantic simILARity (SEMILAR) toolkit. Then, the GBM has been integrated with the OpenII framework. The SEMILAR toolkit includes libraries that facilitate textual preprocessing (e.g., collocation identification, part-of-speech tagging, phrase or dependency parsing, etc.) and semantic similarity computation at both word-level and sentence-level [141]. In our implementation of the GBM, we used these libraries to implement a modified version of the Mihalcea, Corley, and Strappavara (MCS) method for sentence-to-sentence similarity.

Finally, the modified similarity flooding matcher has been implemented in a separate module, and combined as a pipeline architecture.

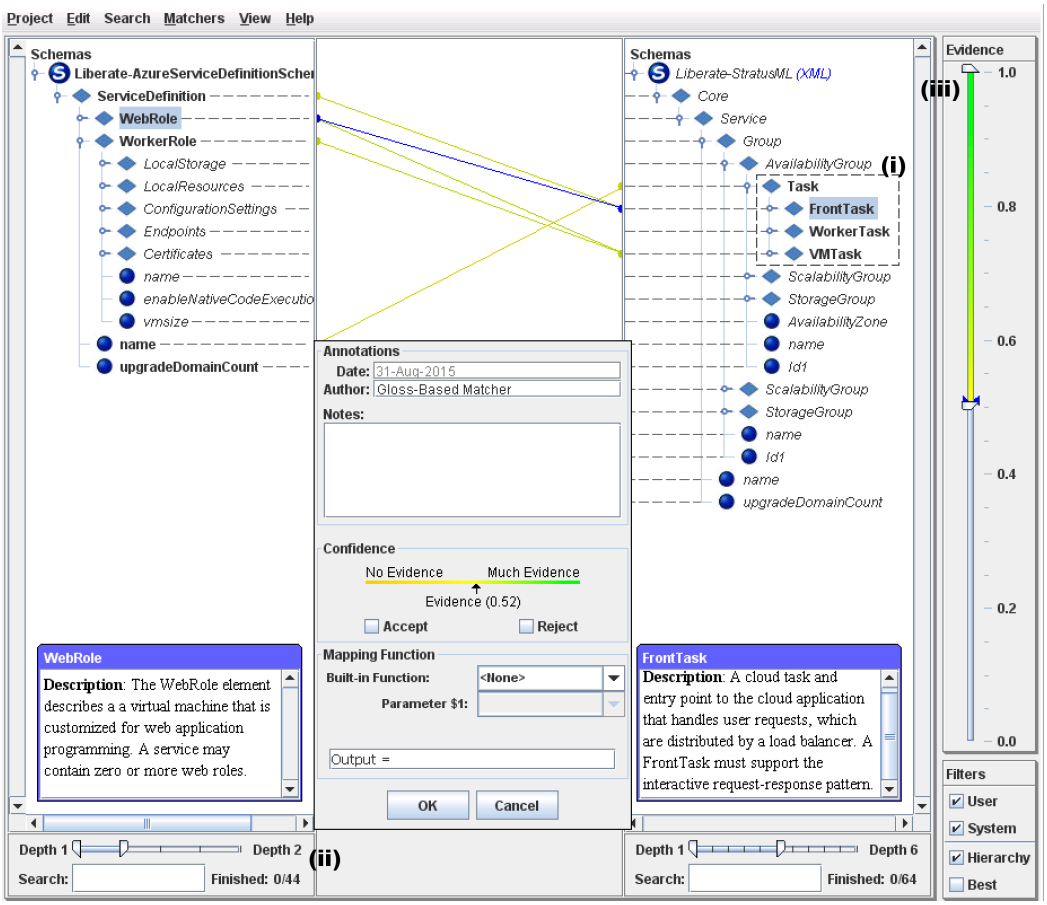


Figure 6.7: Applying the GBM on AZURE Service Definition and StratusML Core Schemas

Figure 6.7 is a snapshot that shows the results of applying the gloss-based matcher on two cloud schemas: a public schema that belongs to Microsoft Azure Service Definition and a private schema that corresponds to the StratusML core meta-model. The figure shows three of the filtration methods that have been used to reduce the noise of the false positive matches:

(i) *Set focus:* This filter refines the results by showing only the relationships between the concept selected from the first schema, including its sub-tree elements and all the corresponding matches from the other schema. For example, Figure 6.7 shows how the set focus filter has been used to refine the matching results of the GBM matcher to show only the "Task" component and its corresponding matching elements on Azure Service Definition.

(ii) *Depth:* This filter is used to show only schema elements that appear at a particular nested depth. The filter appears at the bottom of Figure 6.7, where we used it to show the matching results at depth one in the Azure Service Definition and depth five in StratusML schema. Filtering based on the depth can help alleviate some of the structural differences between the schemas.

(iii) *Evidence threshold:* Shown on the right side of Figure 6.7, this filter refines the matching results below a certain threshold. In the figure, this filter has been set to show the matching results with evidence greater than or equal to 0.45.

Figure 6.7 shows that using the GBM and the selected filters, it was possible to reveal the similarity between the Azure "WorkerRole" and the StratusML "WorkerTask" concepts. The figure shows a mapping with confidence (evidence = 0.48).

The next section discusses the tests performed to evaluate the proposed system. To make this research reproducible, our modified OpenII framework implementation, in addition to the datasets used in this chapter and the outcome matching results have been made publicly available online at the Liberate webpage[3].

6.3 Experimental Setup and Evaluation

The objective of this section is to evaluate the performance of the different matchers implemented within the Liberate approach. We answer the following questions:

[3]http://www.stargroup.uwaterloo.ca/~mhamdaqa/liberate/

Q1. *How will the proposed Web Similarity Matcher perform when applied to public and private schemas?*

Q2. *How will the proposed Domain Knowledge Matcher perform in comparison to the WSM approach?*

Q3. *How will the structural based matching affect the results?*

First, we explain the experimental setup, then the evaluation results, followed by a discussion and finally the threats to validity.

6.3.1 Experiment Setup

Three experiments with a total of 12 matching tasks have been conducted to provide answers for the aforementioned three questions. Table 6.4 provides a summary of the experiments. For each experiment, the table specifies the set of matchers that have been compared and the schemas that the matchers have been applied to.

Table 6.4: Experiment Set

Name	Description	Matchers	Target Schemas	Schema 1	Schema 2
Experiment 1	Web vs. Linguistic	WSM vs. (NS and WN)	Public	Azure Service Definition	GAE
			Private	StratusML	TOSCA
Experiment 2	Domain Knowledge	WSM vs. GBM vs. DKM	Public	Azure Service Definition	GAE
			Private	StratusML	TOSCA
Experiment 3	Domain Knowledge vs. Structural	DKM vs. SF++	Public	Azure Service Definition	GAE
			Private	StratusML	TOSCA

In each experiment, the different matchers were evaluated as they were applied to public and private cloud schemas. From public cloud schemas, we experimented with Microsoft Azure "service definition" schema and Google App Engine "appengine-web" schema. From private cloud schemas, we experimented with StratusML "core meta-model" schema, and TOSCA "service template" schema. Two variations of each schema have been used. In the first variation, schema elements were not annotated with domain glosses, while in the second variation elements were annotated with their domain glossaries.

The first experiment evaluates the performance of the WSM against the OpenII Name Similarity (NS), and WordNet (WN) matchers. The NS matcher is a linguistic syntactic matcher that is based on edit distance [116]. The WN matcher, by contrast, is a bag matcher that uses the WordNet dictionary as a thesaurus [112]. For this experiment, the weight for the WSM was set to 100%, and GBM to 0%. The WSM was set to use Microsoft

Bing as a search engine, and the search was bound to a context by appending the provider's name before the schema concepts. Moreover, the time between requests was set to 0.25 seconds to avoid being classified as a denial of service (DoS) attack.

The second experiment compares the results obtained from WSM against the ones obtained from applying the GBM and DKM. The matchers have been applied to the same public and private cloud schemas as before. To focus on the results of the GBM, we set the weight of the GBM to 100% and WSM to 0%. Then, to compare the results of both the WSM and GBM to the hybrid DKM approach, we set the weight of GBM to 75% and the weight of WSM to 25%. This percentage was selected based on the intuition that glossaries, when available, may contain more accurate domain knowledge in comparison to the public web. This is because glossaries explain concepts as defined by the providers' themselves.

The third experiment compares the results obtained from DKM with the results of the SF++ approach in order to investigate the impact of structural matching on the final results. The comparison was conducted against the same schemas and with the same settings for the DKM as before.

We ran the experiments using the different matchers that have been selected. This creates a total of 12 tasks; each task runs one matcher against two schemas. The results were then filtered based on different threshold evidence values. Then, the mappings were exported and evaluated. To make the comparisons fair, no manual selection of mappings has been considered in this evaluation.

6.3.2 Evaluation of Results

In order to evaluate the proposed matchers, we use the matching results obtained from running the matchers. The results are then compared to a set of reference alignments. In this study, we use a set of predefined alignments that have been manually prepared by domain experts. For each alignment, we determine the true positive, false positive and false negative matches. Accordingly, the values of precision, recall, and f-measure were computed according to Equations 6.6, 6.7, and 6.8 respectively. The outcome of these equations is between zero and one, the higher the value the better the result.

$$\text{Precision} = \frac{|\{\text{ReferenceAlignments}\} \cap \{\text{MatchResults}\}|}{|\{\text{MatchResults}\}|} \tag{6.6}$$

Precision and recall are widely accepted measures for evaluating the results in information retrieval. Precision measures correctness; how many of the matches discovered by

the matcher are correct in comparison to the total number of matches retrieved. A higher precision means less wrong matches.

$$\text{Recall} = \frac{|\{\text{ReferenceAlignments}\} \cap \{\text{MatchResults}\}|}{|\{\text{ReferenceAlignments}\}|} \tag{6.7}$$

On the other hand, recall measures completeness; how many correct matches are retrieved in comparison to the total number of correct matches in the reference alignment. A higher recall means more matches have been found.

$$f-\text{measure} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{6.8}$$

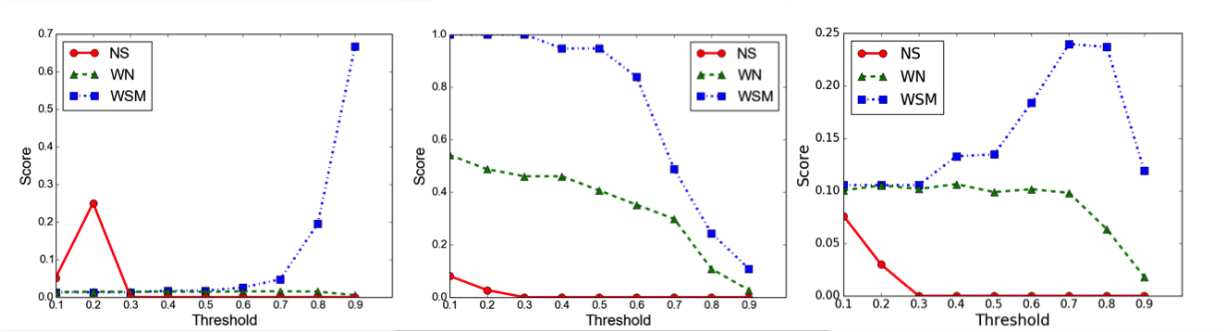
In practice, maximizing precision could lead to a fewer retrieved matches (lower recall), while maximizing recall may result in a poor precision. To have a better perception of the quality of the different matchers, the values of both precision and recall should be combined. One way to combine both of them is through the f-measure, which is the harmonic mean of precision and recall. The f-measure provides a global measure of the matching quality. However, the f-measure gives equivalent weights to both precision and recall. Weighing precision and recall of a tool the same is not correct. As described by Berry [15], the human effort to find a true positive manually measured by recall is different from that to reject a false positive measure by precision. If the human effort to find a missing match is β times the efforts to vet a matching result, then recall should be weighted β times the precision in calculating the f-measure. The weighted f-measure (f_β) can be caculated using Equation 6.9.

$$f_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} \tag{6.9}$$

In Liberate, the effort of finding a missing match is approximately 10 times the effort to vet a match. Accordingly, in the evaluation, we use the weighted f-measure equation f_β with $\beta = 10$.

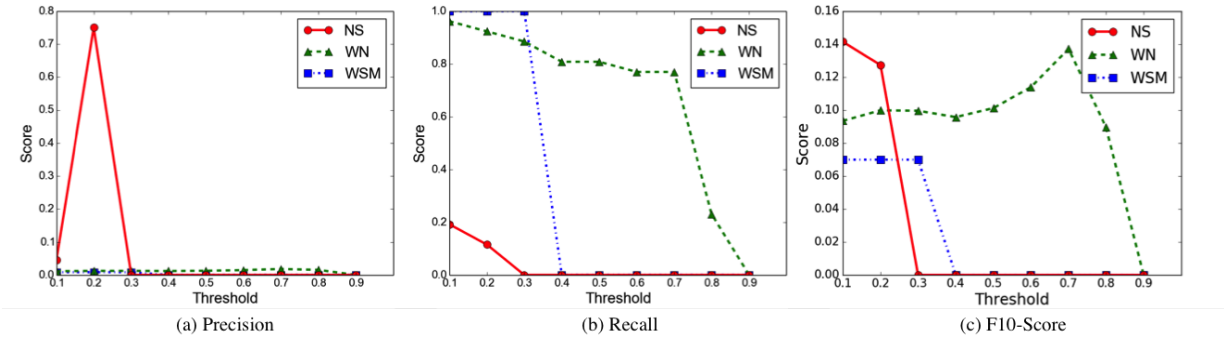
Experiment1: Web vs. Linguistic Matchers

Figure 6.8 shows the results of applying the WSM, WN and NS matchers to public and private cloud schemas respectively.



(a) Precision (b) Recall (c) F10-Score

(A) Public Schemas (Azure vs. GAE)



(a) Precision (b) Recall (c) F10-Score

(B) Private Schemas (StratusML vs. TOSCA)

Figure 6.8: Experiment 1 - Results

Figure 6.8A shows that the WSM approach tends to provide higher precision and recall. The value of f_{10}-measure is also higher in comparison to NS and WN matchers for public clouds. In particular, as the evidence threshold increases the precision increases. The precision of the NS matcher starts higher than the WSM. It increases until the evidence threshold reaches 20% then it hits zero at 30% evidence threshold. This sudden change can be reasoned by reviewing the recall figure, which shows very poor recall at 20% that drops to zero at 30% evidence. The relationship between the number of matches found and the evidence threshold is inversely proportional. This is expected, because when the evidence is low the matcher retrieves more results; both correct and incorrect. The reason WSM has the highest recall is due to the fact that it is a statistical measure. Given the large number of pages on the Internet, in most cases, the value of the WSM is greater than zero, which indicates some statistical relatedness. However, this is not the case for linguistic based similarities, where the similarity can easily hit zero.

Based on the f_{10}-measure results, the NS matcher is the worse performer when applied to public cloud schemas, while the WSM is the best performer. The f_{10}-measure results in Figure 6.8A shows a proportional relationship between the WSM results and evidence threshold up to 0.8 evidence. At this point, the f_{10}-measure value drops due to the sharp drop in the recall (i.e., very few values retrieved, some of them are correct mappings).

Figure 6.8B shows the results of repeating the same experiment for private cloud schemas. This time, the WSM scored the worse in comparison to both linguistic matchers (i.e., NS and WN). The reason might be due to the limited content online that refers to the concepts in these schemas. The WSM produced very poor matching precision and high recall that sharply fall down to zero at 30% evidence threshold. On the other hand, the behavior for both the NS and WN seems to be consistent for both public and private cloud schemas, in both cases, they had poor performance. This is expected, as both matchers do not incorporate domain knowledge in the matching process, and the concepts in both public and private cloud schemas do not share a linguistic similarity. The f_{10}-measure confirms that none of the matchers are suitable for private clouds with linguistic approaches being more favorable.

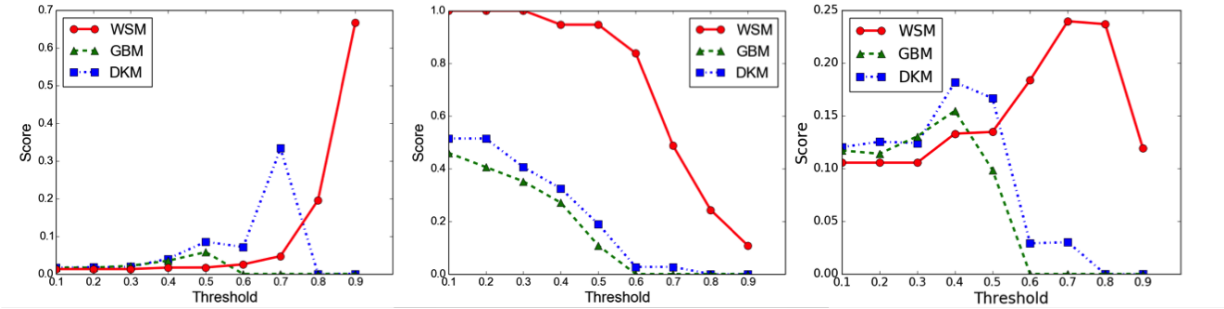
Experiment2: Domain Knowledge Matchers

Figure 6.9 shows the results of evaluating the performance of the WSM, GBM and DKM matchers as they are applied to public and private cloud schemas respectively.

Figure 6.9A shows that the WSM is still the dominant matcher for public cloud schemas. WSM performs better than both GBM and DKM. In particular, the GBM shows poor precision, moderate recall, and relatively lower f_{10}-measure when the evidence threshold is below 0.6. Nevertheless, the matching outcome of the GBM still outperforms the results obtained by the pure linguistic matchers used in the first experiment.

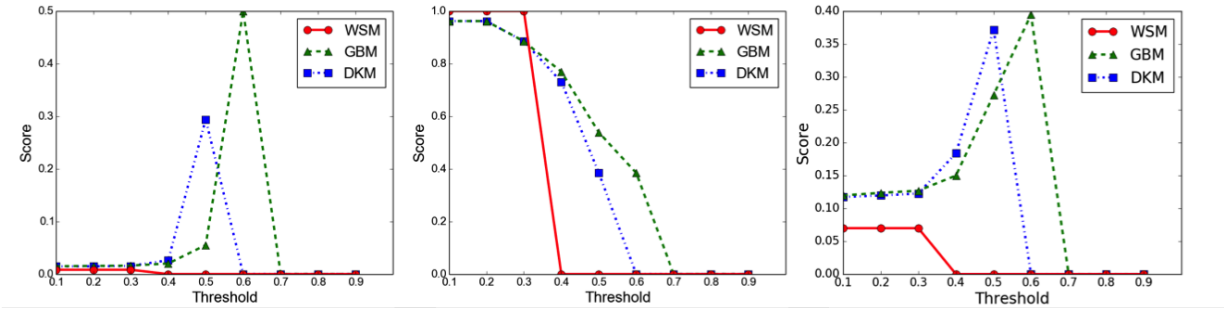
As expected the results obtained by the DKM is between those obtained by the GBM and WSM. This is because the DKM combines the results of both matchers. However, since more weight was given to the GBM, the DKM results are closer to the GBM results.

Figure 6.9B shows the results of applying the same matchers on private cloud schemas. As in the first experiment, the WSM fails as a matcher for private cloud schemas. On the other hand, the GBM shows promising results when applied to private cloud schemas. The precision of the GBM increases as the evidence threshold increases. The precision value reaches its peak at 0.6 threshold. The value then dropped to zero as the total number of retrieved matches (recall) dropped to zero at 0.7 evidence threshold. The f_{10}-measure score shows that the GBM outperforms both the DKM and WSM. On average the GBM results



(a) Precision (b) Recall (c) F10-Score

(A) Public Schemas (Azure vs. GAE)



(a) Precision (b) Recall (c) F10-Score

(B) Private Schemas (StratusML vs. TOSCA)

Figure 6.9: Experiment 2 - Results

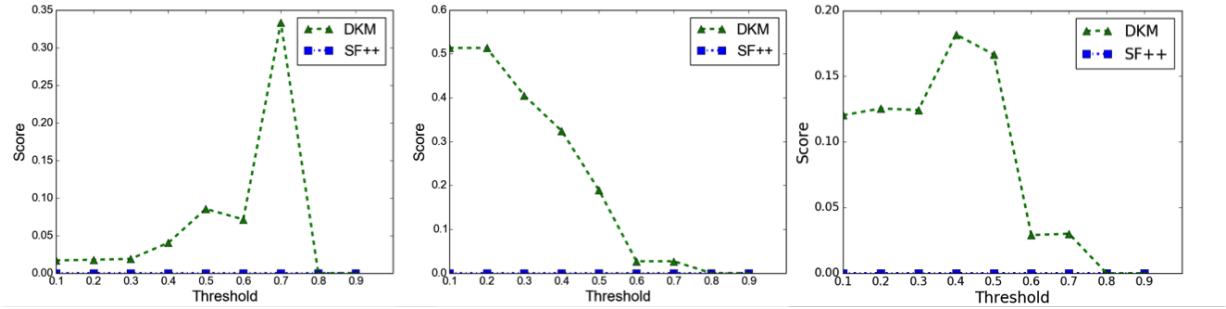
were 8% higher than DKM and 70% higher than the WSM when applied to TOSCA and StatusML schemas.

Experiment3: Structural Matcher

The third experiment aims at examining the effect of structural similarity. Figure 6.10 shows the results of applying SF++ to public and private cloud schemas. The results are compared with the DKM results obtained from the second experiment.

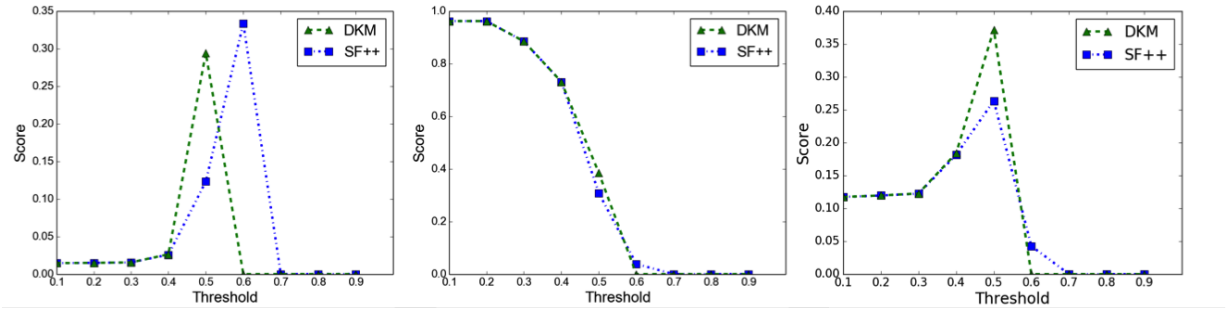
The observations from Figure 6.10A, and 6.10B are that:

(i) The public cloud schemas used in this experiment are structurally dissimilar. This is clear from the zero values of precision, recall and f_{10}-measure.



(a) Precision (b) Recall (c) F10-Score

(A) Public Schemas (Azure vs. GAE)



(a) Precision (b) Recall (c) F10-Score

(B) Private Schemas (StratusML vs. TOSCA)

Figure 6.10: Experiment 3 - Results

(ii) The private cloud schemas used in this experiment expose some kind of structural similarity, but over all they still have significant structural differences. As shown in the precision results in Figure 6.10B, the SF++ improved the precision of the DKM, particularly at a higher evidence threshold (e.g., 0.6).

(iii) The SF++ seems to have a negative effect on recall. While this was not that obvious for private clouds, it was the main factor that brought the precision and f_{10}-measure values to zero in public clouds.

The SF++ can have negative impact on the initial matching results when the schemas do not expose structural similarity. This is due to the effect of the fix point computation, which affects the similarity magnitude of the initial matches. Degrading the initial similarity magnitude affects the number of true positive values, which is the reason behind the poor recall values.

Discussion

In this subsection we revisit our questions based on the experimental results.

Q1. *How will the proposed Web Similarity Matcher perform when applied to public and private schemas*?

The first experiment shows that the WSM has higher precision recall and f_{10}-measure values than the pure linguistic approaches, when applied to public cloud schemas. Unfortunately, all the matchers used in the first experiment, including the WSM, preformed badly when applied to private cloud schemas.

Q2. *How will the proposed Domain Knowledge Matcher perform in comparison to the WSM approach?*

The results of the second experiment show that the WSM workes better to detect and retrieve matches of the public cloud schemas, while GBM works better for private clouds. To combine the benefits of both matchers, the DKM measure can be used in both cases and produces decent results. The DKM is more favorable in scenarios where there is limited knowledge about the schemas. For example in scenarios where we are not sure how much online content is available, or if we have incomplete glossaries.

Q3. *How will the structural based matching affect the results?*

As shown in the results of the third experiment , for the selected schemas the SF++ did not add value to the matching process. However, using the SF++ can reveal important knowledge about how much two schemas are structurally deviated, which can serve as an indicator on the manual efforts required for schema alignment, and transformation template generation. Moreover, the SF++ can be used in cases of schema evolution (i.e., to discover mismatches between different versions of the same schema).

It is worth to be noted that, although the conclusions were the same; the results reported in this thesis, which were obtained using the weighted f_{10}-measure, were stronger than those obtained using f_1-measure. This confirms Berry's [15] observation.

6.3.3 Threats to Validity

In this section four validity threats are discussed. Two of these threats are external threats that affect the generalization of the approach; those are the veracity of search results and

the size of the experiment. The other two are internal threats that make some of the obtained results questionable; those are, the matching confidence and the ground truth alignments used.

- **Veracity of search results:** Data veracity refers to uncertain, imprecise or inaccurate data [54]. The WSM metric uses the number of hits retrieved from search results. Therefore, the accuracy of the WSM depends on the search results' precision. Unfortunately, it has been noted that search engine results are not always precise (i.e., based on estimates). Search engines, such as Google or Yahoo, rely on cached results to answer search queries; unless, the cached results are below a certain threshold. For instance, using Google to search the term "car", the number of pages returned is 965 million, while if you search a more precise term such as "car games" the number of pages returned is over a billion. Obviously, this is misleading as the search results of "car" should involve "car games". In order to address this issue, the WSM implementation enforces the use of deep search through parameter tuning, which is a more sophisticated search option that provides more accurate results, by avoiding the use of precached results.

- **Size of the experiment:** Currently, our experiments use a sample of four schemas and apply the proposed matchers to them. Two of the sample schemas are public and two are private. However, to generalize the results of this study, a wider range of schemas that cover other cloud vendors and providers should be examined.

- **Matching confidence:** The current approach uses naive threshold-based extraction technique for extracting significant mappings. This is because some matchers are generous by nature, while others are greedy. For example the WSM is a generous matcher; it always returns some similarity results. However, if the similarity between one concept from the source schema and all other concepts from the target schema is always the same (i.e., high or low). This means, the matcher has failed to clearly identify matches for that concept. In case of the WSM, this could be because the concept is a common concept that appears in all search results.

- **Ground truth alignments:** The matchers that have been introduced in this chapter have been evaluated using a set of ground truth alignments. These alignments have been manually created by a domain expert; hence, they are susceptible to human errors. This challenges the evaluation results. In this study, the credibility of the manually created reference alignments stems from the fact that these alignments have previously been used to create a domain independent modeling language for the cloud [69, 70]. The alignments were realized as transformation rules to automate

mapping the provider-independent models created using our modeling language into cloud provider-specific deployment artifacts, and deploy actual applications on the cloud.

Even with these validity threats and concerns, the advantages of the approach and matchers introduced in this chapter are evident and presents a forward step toward automatic migration and transformation for cloud models and applications.

6.4 Summary

This chapter argues that there is a need to investigate new techniques for schema matching that are able to deal with the mismatches between the cloud providers' schemas. Cloud schemas are unconventional vendor driven schemas, where similar domain concepts may not be identifiable via linguistic similarities. Hence, there is a need for advanced techniques that incorporate domain knowledge in the matching process.

This chapter presents a semi-automated schema matching process that solves the problem of obtaining domain knowledge and making the complex alignments required to facilitate model driven migration of the cloud service models between different providers. Particularly, this chapter makes the following contributions:

(i) It uses schema matching to attack the problem of cloud vendor lock-in: we cover two different matching scenarios between public schemas and private schemas.

(ii) It devises a new element-based matcher: the new matcher combines gloss-based and web-based similarity to incorporate domain and provider specific knowledge in the matching process, to find correspondences between similar concepts that may not share linguistic or semantic features.

(iii) It generalizes the similarity flooding algorithm to work with any directed acyclic graph, instead of only a labeled directed graph.

We call the new approach "Liberate" because it frees users from vendor lock-in. Liberate has been realized as an extension to the OpenII framework. The approaches implemented within liberate have been examined against public and private cloud schemas. The results show that the WSM performs better when applied to public cloud schemas. On the other hand the GBM works better on private cloud schemas. Generally speaking the combination

of both matches (i.e., the DKM) performs better than traditional purely linguistic schema matching approaches when applied to providers specific cloud schemas. Finally while providers schemas are usually structural dissimilar, structural similarity (SF++) can reveal important knowledge about the amount of manual efforts required for schema alignment.

Although the approach presented in this work and the results are promising, there are some issues that need to be addressed: (i) the generalized similarity flooding (SF++) contribution to the matching results was not significant. However, we argued its benefit as indication of the efforts needed for manual alignment and transformation template creation, or to uncover mismatches between different revisions of the same schema. This is another interesting problem that we are planning to address in the future. (ii) As in all schema-matching approaches scalability is still a major issue that affects matching performance in terms of execution time. However, due to its dependency on online search results, the WSM takes even more time than other processes to discover the alignments. The current implementation addresses the scalability problem by caching. The performance of the current approach can be further improved in the future by implementing the similarity-matching task as a parallel process and using partitioning to allow several machines to perform similarity matching at the same time. (iii) In some cases the relationships between elements is difficult to be revealed using both the domain glossaries and web-similarity matching. However, a domain expert is able to infer this relationship from the available schema examples; another venue to be explored in the future. Finally, (iv) there is a need for a repository for thesauri, acronyms, dictionaries, and mismatch lists for schemas across cloud vendors.

Chapter 7

Conclusions and Future Directions

"Research is to see what everybody else has seen, and to think what nobody else has thought."
– Albert Szent-Gyorgyi, 1937

This thesis investigates the process of building, deploying and managing multi-cloud applications, by harnessing the underlying cloud platform and infrastructure capabilities in a cost-effective manner. This chapter summarizes the findings of this thesis and presents future directions. Section 7.1 presents the contributions of the thesis, Section 7.2 sums up the answers to the research questions , Section 7.3 discusses future work that would extend this research, finally, Section 7.4, makes concluding remarks.

7.1 Contributions

The contributions of this thesis stated in Chapter 1. This section discusses these contributions in the light of the material that is covered in chapters 2-6. These contributions include:

- devising a new architectural-view-model that is tailored to address the cloud DevOps process: The (5+1) View Model. The (5+1) view model is an architectural framework that consists of five model views, namely: the availability, adaptation, performance, workflow, and provider models. What distinguishes the (5+1) view model from other software architecture frameworks is that it bridges the gap between service

design and operation. The (5+1) view model provides a set of modeling concepts and views to address the dynamic nature of cloud applications (e.g., the adaptation view), and to model its performance under diverse configurations. Moreover, the (5+1) view model captures the common practices of the different cloud application stakeholders for creating, interpreting, analyzing, and using architecture descriptions to specify cloud applications deployment and management aspects. The description of the core model was first introduced in the first International Conference on Cloud Computing and Services Science [66]. At CASCON [69], I presented the different views of the (5+1) view model and their corresponding meta-models. Moreover, at IC2E [70], I used scenario-based examples to show how the introduced views can facilitate architecting malleable applications that can change their structure and behavior at runtime through configuration.

- developing a modeling framework for multi-clouds applications: The StratusML framework. StrarusML is a realization of the (5+1) view model. The StratusML framework provides model editor and generator for cloud application services that have been implemented to facilitate model once deploy everywhere approach and keep the architectural models in synchrony with the configuration space artifacts as applications evolve. Moreover, StratusML offers a model validator to ensure the correctness and completeness of the created models, and a model analyzer to analyze the anticipated performance, availability and cost of the created models under different configurations and providers. Some of the distinctive features of StratusML are layers, template-based-transformations and the ability to model adaptation rules and actions. Layers and template-based-transformations facilitate fragmenting the models into artifacts that are easy to modify and enable weaving the fragments into model views to empower collaboration between the cloud stakeholders. The StratusML framework features and implementation were covered in parts in several publications [66, 69, 70, 65].

- extending the StratusML language to capture the cloud application runtime performance parameters and to generate corresponding layered queuing network models; we refer to this new extension as StratusPM. The distinctive advantage of StratusPM over other performance model generators is its ability to address the fluid nature of the cloud service and deployment models, by reusing these models in addition to the provider specification to automatically generate corresponding analytical performance models for cloud applications. Using StratusPM, cloud performance engineers can generate analytical performance models on the fly from the information specified by the architects and system administrators. This not only reduces the efforts

needed to specify analytical performance models, but it also maintains consistency between models as models evolve. Performance engineers will only be responsible for specifying the runtime measurements and execution scenarios. This will alleviate them from the need to know the target platform specifications and deployment models and configurations. Chapter 5 shows an example of how to transform StratusML models that are annotated with StratusPM components into layered queuing network models.

- automating vendor-specific schema matching and meta-models creation and evolution. I devised a number of schema matching approaches (i.e., PrisonBreak and Liberate) to facilitate discovering correspondences between the concepts of the different cloud schemas in order to create model abstractions (meta-models) for the cloud providers' schemas, keeping them up to date, and generating mappings between the providers' independent abstractions and the providers' specific schemas. What distinguishes these approaches is their ability to incorporate domain knowledge in the schema matching process by considering statistical web-based similarities, and domain glossaries to uncover similarities between similar domain concepts even if they are linguistically dissimilar. The approaches proposed in this thesis showed significant improvement over existing schema matching approaches when applied to vendor specific schemas, as demonstrated in several publications [68, 71].

7.2 A Summary of Research Questions

This section revisits the research questions presented in this thesis and provides short answers based on the findings presented in the different chapters.

RQ1. How is architecting cloud applications different from architecting other applications?

Chapter 3 of this thesis provides an elaborate answer to this question, by eliciting the requirements for architecting cloud-native applications. The following are the main distinctive architectural characteristics that distinguish the process of architecting a cloud application:

(i) a cloud application is a malleable (elastic and dynamic) application. It changes its structure and behavior at runtime through utilizing the cloud infrastructure as a code capability to meet performance, availability, and scalability targets. Accordingly,

architecting for the cloud should focus on addressing the concerns of service operation rather than the service implementation.

(ii) architecting an application for the cloud includes complex architectural, operational and financial decisions. These decisions are made by different stakeholders and span to after the application's delivery.

The (5+1) view model presented in Chapter 3 is constructed around the application service and deployment model. It captures all the essential information for architecting malleable applications that can change their structure and behavior at runtime through configuration.

RQ2. How should an application be architected to be deployed into multiple clouds and how should cloud modeling and configuration spaces be bridged?

To architect an application to be deployed into multiple clouds, there is a need to first, separate the application domain and architectural concerns from the provider's operational, technical and financial concerns, second, provide an abstraction layer for each of the concerns. In this thesis, the separation between concerns has been achieved through introducing a new architectural style for developing malleable cloud applications. Moreover, the abstraction layer has been implemented through using meta-models. On the other hand, to bridge the cloud modeling and configuration spaces there is a needed for a modeling framework that provides (i) abstract and providers independent modeling constructs to incrementally build cloud models that address the different cloud concerns and (ii) tools for model weaving, fragmentation, and transformation to enable generating the configuration space artifacts for the target provider. As shown in Chapter 4, this has been achieved through utilizing model-driven engineering to build a multi-layer modeling language that uses template-based transformations to enable generating the target providers' artifacts.

RQ3. How to extend the framework to support generating analytical performance models for cloud applications?

As explained in Chapter 5, constructing an analytical performance model includes two steps: (i) modeling the application structure and infrastructure resources, (ii) modeling the user interactions and workload characterization. On the other hand, deploying an application on a cloud platform requires specifying the structure and infrastructure information of the application. Accordingly, to generate analytical performance model from the cloud service and deployment artifacts, the modeling framework that has been created to model and generate these artifacts (StratusML) needs to be extended to first, capture the

resource usage model, which describes the interaction between the users and the system, second, capture the resource utilization parameters, which show the load on the different resources as a result of user requests.

RQ4. How should the framework be kept up-to-date?

Maintaining a framework that uses model-driven-engineering up-to-date requires first, keeping the abstraction layer (meta-models) that the framework uses to describe the syntax and semantic of the underlying modeling language in that framework up-to-date, second, generating new transformation rules to map the concepts in the abstraction layer to its corresponding concepts on the target provider schema. Such mapping requires checking continuously for mismatches and similarities between the different vendors' schemas and the modeling framework meta-models. Uncovering such similarities and mismatches between the different schemas can be a daunting task as it requires in-depth knowledge of the semantics of the various concepts in that domain. Chapter 6 shows that one possible solution to this problem is to first, automate the process of discovering mismatches and similarities, by exploiting some of the machine learning techniques used in similarity matching, and second incorporate domain knowledge in the matching process to address cases where concepts do not share linguistic similarities, which is a common case in industrial schemas.

7.3 Future Work

The development of the StratusML framework and Liberate approach laid the groundwork for several future directions. The following is a list of possible future work:

- extending the framework to support micro-services architecture and containers technologies: As microservices and container-based applications are getting more momentum, there is more demand on frameworks that facilitates (i) designing applications by composing micro-services independently from any container technology (e.g. Docker), and (ii) enabling the linkage between the build systems and infrastructure deployment artifacts. StratusML alleviates some of the challenges regarding service integration and operation, by facilitating generating the platform artifacts that are required to schedule and manage the deployment of containers. One of the direct extensions of the StratusML framework would be to enable generating the configuration artifacts of the container images. This will require an additional abstraction

layer that is closer to the containers technology to allow creating the containers build files. There will also be a need for another layer of abstraction, at a higher level, that facilitates utilizing microservice architectural patterns and templates to generate their corresponding artifacts.

- supporting a wider range of cloud providers: StratusML currently provides out of the box templates for generating the platform configuration for windows azure and google application engine. In the future, I will work on creating more templates to support more platform providers. Moreover, I will work on providing transformations to some of the standard cloud description languages (e.g., TOSCA [17])

- extending the StratusML to support IoT services: IoT-based services require management frameworks that support managing both the devices at the edge and the supporting services at the cloud platform. StratusML supports managing only the cloud platform side of the network, more work will be required to support the modeling and configuration of the edge devices. Moreover, bandwidth and latency limitations at the edge impose challenges on transferring data from the smart edge devices to the cloud. To deal with these limitations there is a need for a technological layer that sets between the edge devices and the cloud and addresses scalability and latency and provides intelligent control for the data that is generated at the edge devices. CISCO refers to this layer as Fog Computing [20]. Similar to cloud computing, the fog consists of compute and storage devices. However, different from the cloud dedicated datacenter model, the fog uses an ad-hoc model that depends on mobile devices that are close to the edge devices. The analytical applications that run on the fog have specific requirements. For example, these applications need to adhere to the different set of standards and protocols that are used by the different devices. In the future, I will work on developing an abstraction layer, similar to the one that I developed for the cloud, in order to facilitate deploying IoT services on the fog and automatically configuring them to the target edge devices.

- automating partitioning of applications into microservices: The current framework assumes the developers can make the right decisions in terms of partitioning the application into modules (microservices), and then distributing them into geographic locations to achieve the desired availability level. However, partitioning applications into cohesive modules is difficult and requires knowledge about the dependency relationships between the different modules. In the future, I am planning to use graph theory and clustering techniques to extend the framework to support the automatic partitioning of applications into micro-services that can be hosted into containers

and to group micro-services into container pods. This fine-grain decomposition will have a huge impact on applications' malleability, self-adaptation and its availability.

- implementing feedback loops between the runtime model and the performance model. Currently, StratusPM requires manually specifying the performance model parameter. Future work will investigate combining the application performance monitoring (APM) and round trip model driven techniques to fully automate the application adaptation process. I will be utilizing the platform centralized logging capabilities to facilitate the management of distributed services to remediate issues or to alert operators as needed.

- supporting other target performance models. Currently, the proposed framework supports generating LQN models from the StratusPM models. In the future, I am planning to support transformations to other performance modeling languages that address modeling software components and thier underlying platforms such as Palladio Component Model (PCM) [11]. Supporting other performance modeling frameworks will give the users of the StratusML framework more flexibility and options to compare the performance results.

- expanding the application of domain schema matching to automate model-driven migration: In this thesis, the liberate approach has been proposed to deal with the vendor lock-in problem, by uncovering the alignments between the concepts of the provider specific schemas. Liberate is a generic approach that can be equally applied to other domains. In the future, I will examine the approach in the model driven-engineering domain to semi-automate the process of creating transformation rules between the source and the target models on the fly. Particularly, I will investigate how to automatically formulate mappings (transformation rules) based on the generated matches.

- further evaluation for the framework: In this thesis, I used scenario-based examples to demonstrate the usefulness and capabilities of the proposed framework. In the future, I am planning to conduct a comprehensive empirical study to evaluate the impact of the proposed framework on the roles of the cloud stakeholders and the cloud DevOps process. Particularly, I will evaluate the usability, domain coverage, correctness, maintainability, and domain specificity. That is the DSML small enough, leaving out language features that do not contribute to the purpose of the language.

7.4 Conclusion

Cloud platforms advances have changed the application development landscape. Cloud platforms abstract the complexity of application delivery to enable the applications' rapid development and easy management. This changes the way development teams need to think about and deal with the underlying resources while building and managing their applications. In cloud computing, architecture evolves during deployment; therefore, runtime operation needs as much architectural modeling as functional design does. This thesis describes a new method supported by a modeling framework to enable organizations that build cloud applications, SaaS providers, to exploit the cloud platform building blocks to leverage the flexibility, reliability, and scalability that these platforms provide to the application layer. I have successfully realized the architectural view model as a cloud DSML that uses layers to toggle between partial and holistic views, model dynamic behaviors using adaptation rules and actions, and "weave" stakeholder concerns together to generate useful artifacts for the supported target platforms. While the framework currently supports generating artifacts for three cloud platforms, support for framework evolvability has been considered, using template-based transformation and advanced schema matching techniques.

APPENDICES

Appendix A

LQN Transformation Template

```
<lqn-model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" name="TailSpin">
  <solver-params conv_val="1" it_limit="50" print_int="0" underrelax_coeff="0.5" />
  <processor name="RefTask_TailSpin.Services.Surveys.Host.AzureProcessor" replication="1"
   multiplicity="1" speed-factor="0.5">
    <task name="RefTask_TailSpin.Services.Surveys.Host.Azure" multiplicity="1" replication="1">
      <entry name="RefEndpointToEndpoint1" type="PH1PH2">
      </entry>
      <entry name="RefEndpointToHostInternal" type="PH1PH2">
        <entry-phase-activities>
          <activity name="HostActivity1" phase="1" host-demand-mean="0">
            <synch-call dest="WebInternal" calls-mean="0" />
          </activity>
        </entry-phase-activities>
      </entry>
      <task-activities>
        <activity name="HostActivity1" host-demand-mean="0" />
        <activity name="HostActivity2" host-demand-mean="0" />
      </task-activities>
    </task>
  </processor>
  <processor name="TailSpin.Services.Surveys.Host.AzureProcessor" replication="1" multiplicity="1"
   speed-factor="0.5">
    <task name="TailSpin.Services.Surveys.Host.Azure" multiplicity="1" replication="1">
      <entry name="Endpoint1" type="PH1PH2">
      </entry>
      <entry name="HostInternal" type="PH1PH2">
        <entry-phase-activities>
          <activity name="HostActivity1" phase="1" host-demand-mean="0">
            <synch-call dest="WebInternal" calls-mean="0" />
```

```
        </activity>
      </entry-phase-activities>
    </entry>
    <task-activities>
      <activity name="HostActivity1" host-demand-mean="0" />
      <activity name="HostActivity2" host-demand-mean="0" />
    </task-activities>
  </task>
</processor>
<processor name="RefTask_TailSpin.WebProcessor" replication="1" multiplicity="1" speed-factor=
"0.5">
  <task name="RefTask_TailSpin.Web" multiplicity="1" replication="1">
    <entry name="RefEndpointToHttpsIn" type="NONE">
    </entry>
    <entry name="RefEndpointToWebInternal" type="NONE">
    </entry>
  </task>
</processor>
<processor name="TailSpin.WebProcessor" replication="1" multiplicity="1" speed-factor="0.5">
  <task name="TailSpin.Web" multiplicity="1" replication="1">
    <entry name="HttpsIn" type="NONE">
    </entry>
    <entry name="WebInternal" type="NONE">
    </entry>
  </task>
</processor>
<processor name="RefTask_TailSpin.Web.Survey.PublicProcessor" replication="1" multiplicity="1"
speed-factor="0.5">
  <task name="RefTask_TailSpin.Web.Survey.Public" multiplicity="1" replication="1">
    <entry name="RefEndpointToHttpIn" type="NONE">
    </entry>
    <entry name="RefEndpointToPublicInternal" type="NONE">
    </entry>
  </task>
</processor>
<processor name="TailSpin.Web.Survey.PublicProcessor" replication="1" multiplicity="1" speed-
factor="0.5">
  <task name="TailSpin.Web.Survey.Public" multiplicity="1" replication="1">
    <entry name="HttpIn" type="NONE">
    </entry>
    <entry name="PublicInternal" type="NONE">
    </entry>
  </task>
</processor>
```

```
  <processor name="TailSpin.Workers.NotificationsProcessor" replication="1" multiplicity="1" speed-
   factor="0.5">
    <task name="TailSpin.Workers.Notifications" multiplicity="1" replication="1">
      <entry name="NotificationsInternal" type="NONE">
      </entry>
    </task>
  </processor>
  <processor name="TailSpin.Workers.SurveysProcessor" replication="2" multiplicity="2" speed-
   factor="1">
    <task name="TailSpin.Workers.Surveys" multiplicity="1" replication="2">
      <entry name="SurveysInternal" type="NONE">
        <entry-phase-activities>
          <activity name="SurveysActivity1" phase="1" host-demand-mean="0">
            <synch-call dest="SurveysInternal" calls-mean="0" />
          </activity>
        </entry-phase-activities>
      </entry>
      <task-activities>
        <activity name="SurveysActivity1" host-demand-mean="0" />
      </task-activities>
    </task>
  </processor>
</lqn-model>
```

Listing A.1: The Generated LQNX Model.

Appendix B

CoupoNet LQNS Results

```
Generated by: lqns, version 5.7

Copyright the Real-Time and Distributed Systems Group,

Department of Systems and Computer Engineering

Carleton University, Ottawa, Ontario, Canada. K1S 5B6

Invoked as: lqns C:\Users\Hamdaqa\Desktop\CoupoNet.lqn

Input: C:\Users\Hamdaqa\Desktop\CoupoNet.lqn

Comment: LQN for the Hamdaqa CoupoNet Example

Convergence test value: 6.63235e-006

Number of iterations: 9

Solver:

   User: 0:00:00.00

   System: 0:00:00.00

   Elapsed: 0:00:00.00

   Submodels: 4
```

```
    MVA Core(): 36

    MVA Step(): 394

    MVA Wait(): 133836

Processor identifiers and scheduling algorithms:

Processor Name Type Copies Scheduling

pDelay1 Inf 1 DELAY

pDelay2 Inf 1 DELAY

pDelay3 Inf 1 DELAY

pLC_Delay Inf 1 DELAY

pProvider Mult(2) 1 FCFS

pProviderRef Inf 1 DELAY

pStorage Mult(2) 1 FCFS

pUserRef Inf 1 DELAY

pWeb Mult(2) 1 FCFS

pWorker Mult(2) 2 FCFS


Task information:

Task Name Type Copies Processor Name Pri Entry List

CoupoNet_Provider Uni 1 pProvider 0 SubmitCoupon

CoupoNet_Web Uni 1 pWeb 0 FetchCoupon

CoupoNet_Worker_Logic Uni 2 pWorker 0 RetrieveCoupon, UpdateStore

LoadBalancing Uni 1 pLC_Delay 0 LC_E1

Pipe_Pseudo1 Uni 1 pDelay1 0 Pipe1_E1
```

```
Pipe_Pseudo2 Uni 1 pDelay2 0 Pipe2_E1

Pipe_Pseudo3 Uni 1 pDelay3 0 Pipe3_E1

RefTaskCoupNet_Web Ref(100) 1 pUserRef 0 Users_E1

RefTaskCoupoNet_Provider Ref(20) 1 pProviderRef 0 CreateCoupon

StorageService Uni 1 pStorage 0 Post_info, Get_Info



Entry execution demands:


Task Name Entry Name Phase 1

CoupoNet_Provider SubmitCoupon 7.5

CoupoNet_Web FetchCoupon 8

CoupoNet_Worker_Logic RetrieveCoupon 175

              UpdateStore 85

LoadBalancing LC_E1 50

Pipe_Pseudo1 Pipe1_E1 30

Pipe_Pseudo2 Pipe2_E1 30

Pipe_Pseudo3 Pipe3_E1 90

RefTaskCoupNet_Web Users_E1 3000

RefTaskCoupoNet_Provider CreateCoupon 5000

StorageService Post_info 52

              Get_Info 17

Mean number of rendezvous from entry to entry:
```

```
Task Name Source Entry Target Entry Phase 1

CoupoNet_Provider SubmitCoupon Pipe3_E1 1

CoupoNet_Web FetchCoupon RetrieveCoupon 1

              FetchCoupon Pipe1_E1 1

CoupoNet_Worker_Logic RetrieveCoupon Pipe2_E1 2

              RetrieveCoupon Get_Info 2

              UpdateStore Pipe2_E1 1

              UpdateStore Post_info 1

RefTaskCoupNet_Web Users_E1 FetchCoupon 1

              Users_E1 LC_E1 1

RefTaskCoupoNet_Provider CreateCoupon SubmitCoupon 1

              CreateCoupon LC_E1 1


Mean number of non-blocking sends from entry to entry:


Task Name Source Entry Target Entry Phase 1

CoupoNet_Provider SubmitCoupon UpdateStore 1


Phase type flags:

All phases are stochastic.


Squared coefficient of variation of execution segments:

All executable segments are exponential.


Open arrival rates per entry:
```

```
All open arrival rates are 0.

Type 1 throughput bounds:

Task Name Entry Name Throughput

CoupoNet_Provider SubmitCoupon 0.0102564

CoupoNet_Web FetchCoupon 0.00325733

CoupoNet_Worker_Logic RetrieveCoupon 0.00371747

              UpdateStore 0.00598802

LoadBalancing LC_E1 0.02

Pipe_Pseudo1 Pipe1_E1 0.0333333

Pipe_Pseudo2 Pipe2_E1 0.0333333

Pipe_Pseudo3 Pipe3_E1 0.0111111

RefTaskCoupNet_Web Users_E1 0.0297885

RefTaskCoupoNet_Provider CreateCoupon 0.00388538

StorageService Post_info 0.0192308

              Get_Info 0.0588235

Mean delay for a rendezvous:

Task Name Source Entry Target Entry Phase 1

CoupoNet_Provider SubmitCoupon Pipe3_E1 0

CoupoNet_Web FetchCoupon RetrieveCoupon 468.931

              FetchCoupon Pipe1_E1 0

CoupoNet_Worker_Logic RetrieveCoupon Pipe2_E1 0

              RetrieveCoupon Get_Info 0
```

```
                UpdateStore Pipe2_E1 0

                UpdateStore Post_info 0

RefTaskCoupNet_Web Users_E1 FetchCoupon 74257

                Users_E1 LC_E1 16.8454

RefTaskCoupoNet_Provider CreateCoupon SubmitCoupon 92.4308

                CreateCoupon LC_E1 16.0421

Mean delay for a send-no-reply request:


Task Name Source Entry Target Entry Phase 1

CoupoNet_Provider SubmitCoupon UpdateStore 446.825

Service times:

Task Name Entry Name Phase 1

CoupoNet_Provider SubmitCoupon 97.5

CoupoNet_Web FetchCoupon 775.931

CoupoNet_Worker_Logic RetrieveCoupon 269

                UpdateStore 167

LoadBalancing LC_E1 50

Pipe_Pseudo1 Pipe1_E1 30

Pipe_Pseudo2 Pipe2_E1 30

Pipe_Pseudo3 Pipe3_E1 90

RefTaskCoupNet_Web Users_E1 78099.8
```

RefTaskCoupoNet_Provider CreateCoupon 5255.97

StorageService Post_info 52

Get_Info 17

Service time variance (per phase)

and squared coefficient of variation (over all phases):

Task Name Entry Name Phase 1 coeff of var **2

CoupoNet_Provider SubmitCoupon 25706.2 2.70414

CoupoNet_Web FetchCoupon 1.37e+006 2.27549

CoupoNet_Worker_Logic RetrieveCoupon 50077 0.692044

UpdateStore 25724.8 0.922399

LoadBalancing LC_E1 2500 1

Pipe_Pseudo1 Pipe1_E1 900 1

Pipe_Pseudo2 Pipe2_E1 900 1

Pipe_Pseudo3 Pipe3_E1 8100 1

RefTaskCoupNet_Web Users_E1 1.70828e+010 2.80065

RefTaskCoupoNet_Provider CreateCoupon 2.12688e+007 0.769905

StorageService Post_info 2704 1

Get_Info 289 1

Throughputs and utilizations per phase:

Task Name Entry Name Throughput Phase 1 Total

CoupoNet_Provider SubmitCoupon 0.00380519 0.371006 0.371006

```
CoupoNet_Web FetchCoupon 0.00128041 0.993513 0.993513

CoupoNet_Worker_Logic RetrieveCoupon 0.00128041 0.344431 0.344431

              UpdateStore 0.00380519 0.635468 0.635468

Total: 0.00508561 0.979899 0.979899

LoadBalancing LC_E1 0.00508561 0.25428 0.25428

Pipe_Pseudo1 Pipe1_E1 0.00128041 0.0384124 0.0384124

Pipe_Pseudo2 Pipe2_E1 0.00636602 0.190981 0.190981

Pipe_Pseudo3 Pipe3_E1 0.00380519 0.342468 0.342468

RefTaskCoupNet_Web Users_E1 0.00128041 100 100

RefTaskCoupoNet_Provider CreateCoupon 0.00380519 20 20

StorageService Post_info 0.00380519 0.19787 0.19787

              Get_Info 0.00256083 0.0435341 0.0435341

Total: 0.00636602 0.241404 0.241404


Utilization and waiting per phase for processor: pDelay1


Task Name Pri n Entry Name Utilization Phase 1

Pipe_Pseudo1 0 1 Pipe1_E1 0.0384124 0


Utilization and waiting per phase for processor: pDelay2


Task Name Pri n Entry Name Utilization Phase 1

Pipe_Pseudo2 0 1 Pipe2_E1 0.190981 0
```

Utilization and waiting per phase for processor: pDelay3

Task Name Pri n Entry Name Utilization Phase 1

Pipe_Pseudo3 0 1 Pipe3_E1 0.342468 0

Utilization and waiting per phase for processor: pLC_Delay

Task Name Pri n Entry Name Utilization Phase 1

LoadBalancing 0 1 LC_E1 0.25428 0

Utilization and waiting per phase for processor: pProvider

Task Name Pri n Entry Name Utilization Phase 1

CoupoNet_Provider 0 1 SubmitCoupon 0.028539 0

Utilization and waiting per phase for processor: pProviderRef

Task Name Pri n Entry Name Utilization Phase 1

RefTaskCoupoNet_Provider 0 20 CreateCoupon 19.026 0

Utilization and waiting per phase for processor: pStorage

Task Name Pri n Entry Name Utilization Phase 1

StorageService 0 1 Post_info 0.19787 0

Get_Info 0.0435341 0

```
Task Total: 0.241404

Utilization and waiting per phase for processor: pUserRef

Task Name Pri n Entry Name Utilization Phase 1

RefTaskCoupNet_Web 0 100 Users_E1 3.84124 0

Utilization and waiting per phase for processor: pWeb

Task Name Pri n Entry Name Utilization Phase 1

CoupoNet_Web 0 1 FetchCoupon 0.0102433 0

Utilization and waiting per phase for processor: pWorker

Task Name Pri n Entry Name Utilization Phase 1

CoupoNet_Worker_Logic 0 1 RetrieveCoupon 0.224072 0

                      UpdateStore 0.323442 0

Task Total: 0.547514
```

Appendix C

Cloud Concepts

App Engine Module is a component of an application that provides a single service or configuration. Modules let developers factor large applications into logical components that can share stateful services and communicate in a secure fashion. A module has a collection of versions that define a specific set of code used to implement the functionality of that module. At the highest level, an App Engine application is made up of one or more modules. Each module consists of source code and configuration files to handle a specific task. You can deploy multiple versions of the same module, to account for alternative implementations or progressive upgrades as time goes on.
Source:
https://cloud.google.com/appengine/docs/java /modules/

AWS Beanstalk is a PaaS service from Amazon Web Services that allows users to create applications and push them to a definable set of AWS services. A Beanstalk allows you to configure an entire virtual machine based on one of several available baseline configurations and then customize it through a powerful configuration system. AWS Beanstalk is a fully managed application container service. It's based on a PaaS (Platform as a Service) model where your application is provisioned by infrastructures that are automatically managed by AWS there is no need for you to manually build and maintain them. As a container service, it also provides built-in deployment features for a variety of web app frameworks.
Sources:
http://aws.amazon.com/elasticbeanstalk/faqs/
http://www.eclipse.org/jetty/documentation /9.2.6.v 20141205 /elastic-beanstalk.html

Azure Role is a specialized instance of a virtual machine. Azure Roles are modeled

around the application layers. Each role instance is actually a dedicated Azure VM that is customized for a specific task. A Web Role describes a role that is customized for web application programming, while a Worker Role is typically used for long running background tasks that don't require user interaction. It's a dedicated Azure VM running the Azure runtime environment and your Worker Role application.
Sources:
https://msdn.microsoft.com/en-us/library/azure/

StratusML Task is a composable unit represents Virtual Machine Image, Microservice or Container. It consists of a set of activities that utilize services to provide a specific functionality to solve a problem. It is a mutated unit that can be copied to other virtual machines in order to allow horizontal and vertical scalability. Each task has a service template that specifies the computation power, memory and storage of the virtual machines available to host a task. When composed, tasks should satisfy the following principals: statelessness, low coupling, modularity, and semantic interoperability. Tasks are semantically connected to other tasks in the cloud through the roles they play in order to satisfy a specific business requirement, which is bounded by obligations or responsibilities.
Source:
Hamdaqa, Mohammad, Tassos Livogiannis, and Ladan Tahvildari. "A Reference Model for Developing Cloud Applications". CLOSER 2011.

TOSCA Nodes are the base components (e.g. a VM) and the edges are the relationships between the components. Both nodes and relationships are first defined outside of a concrete service, as they usually represent reusable building blocks. They are called Node Types and Relationship Types. A Node Type is a node of the type VM when used outside of a topology, but once added to a template (potentially with some properties attached), it becomes a node template. TOSCA Node Type is a reusable entity that defines the type of one or more Node Templates. As such, a Node Type defines the structure of observable properties via a Properties Definition, i.e. the names, data types and allowed values the properties defined in Node Templates using a Node Type or instances of such Node Templates can have. In nutshell Node Types represent processes. Node Type allows definition of properties, attributes and lifecycle management of a given process group.
Sources:
http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd03 /TOSCA-v1.0-csd03.html
Stephan Ulbricht, Wolfram Amme, Thomas S. Heinze, Simon Moser, Hans-Dieter Wehle."Portable Green Cloud Services". CLOSER 2014.

References

[1] Palakorn Achananuparp, Xiaohua Hu, and Xiajiong Shen. The Evaluation of Sentence Similarity Measures. In *Data Warehousing and Knowledge Discovery*, pages 305–316. Springer, 2008.

[2] Syed A Ahson and Mohammad Ilyas. *Understanding the Cloud Computing Landscape*, pages 1–16. Cloud Computing and Software Services: Theory and Techniques. CRC Press, 2011.

[3] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, 1984.

[4] Mohamed Almorsy, John Grundy, and AmaniS. Ibrahim. Adaptable, Model-Driven Security Engineering for SaaS Cloud-Based Applications. *Automated Software Engineering*, 21(2):1–38, 2013.

[5] Amazon Inc. AWS Documentation. Online: http://aws.amazon.com /documentation/. [Accessed: 10- May- 2016].

[6] Danilo Ardagna, Elisabetta Di Nitto, Mohagheghi Mohagheghi, et al. MODAClouds: A Model-driven Approach for the Design and Execution of Applications on Multiple Clouds. In *the Workshop on Modeling in Software Engineering*, pages 50–56, 2012.

[7] David Aumueller, Hong-Hai Do, Sabine Massmann, and Erhard Rahm. Schema and Ontology Matching with COMA++. In *the International Conference on Management of Data*, pages 906–908, 2005.

[8] AWS CloudFormation. Online: https://aws.amazon.com/cloudformation/. [Accessed: 10- May- 2016].

[9] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[10] Falko Bause. Queueing Petri Nets-A formalism for the Combined Qualitative and Quantitative Analysis of Aystems. In *the International Workshop on Petri Nets and Performance Models*, pages 14–23, 1993.

[11] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio Component Model for Model-Driven Performance Prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[12] Michael Behrendt, Bernard Glasner, Petra Kopp, Robert Dieckmann, Gerd Breiter, Stefan Pappe, Heather Kreger, and Ali Arsanjani. Introduction and Architecture Overview IBM Cloud Computing Reference Architecture 2.0, March 2011. Retrieved from: https:// www. opengroup.org /cloudcomputing /uploads /40/ 23840 / CCRA.IBMSubmission.02282011.doc.

[13] Zohra Bellahsene, Angela Bonifati, and Erhard Rahm. *Schema Matching and Mapping*, volume 57 of *Data-Centric Systems and Applications*. Springer, 2011.

[14] Alexander Bergmayr, Alessandro Rossini, Nicolas Ferry, Geir Horn, Leire Orue-Echevarria, Arnor Solberg, and Manuel Wimmer. The Evolution of CloudML and its Manifestations. In *the International Workshop on Model-Driven Engineering on and for the Cloud*, pages 1–6, 2015.

[15] Daniel M. Berry. Requirements for Tools for Hairy Requirements or Software Engineering Tasks. Technical report, University of Waterloo, 2016. Online: https://cs.uwaterloo.ca/dberry/FTP_SITE/tech.reports/HTpaper.pdf. [Accessed: 20- Aug- 2016].

[16] Antonia Bertolino and Raffaela Mirandola. *CB-SPE Tool: Putting Component-Based Performance Engineering into Practice*, volume 1 of *Component-Based Software Engineering*, pages 233–248. Springer, 2004.

[17] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. TOSCA: Portable Automated Deployment and Management of Cloud Applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.

[18] Gordon Blair, Nelly Bencomo, and Robert France. Models@ Run. Time. *Computer*, 42(10):22–27, 2009.

[19] Egor Bondarev, Johan Muskens, Peter de With, Michel Chaudron, and Johan Lukkien. Predicting Real-Time Properties of Component Assemblies: A Scenario-Simulation Approach. In *the Euromicro Conference*, pages 40–47, 2004.

[20] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and Its Role in the Internet of Things. In *the Workshop on Mobile Cloud Computing*, pages 13–16. ACM, 2012.

[21] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley Professional, 2005.

[22] Eirik Brandtzæg, Sébastien Mosser, and Parastoo Mohagheghi. Towards CloudML, a Model-based Approach to Provision Resources in the Clouds. In *the European Conference on Modelling Foundations and Applications*, pages 18–27, 2012.

[23] Eric Brill. Transformation-based Error-driven Learning and Natural Language Processing: A Case Study in Part-of-speech Tagging. *Computational Linguistics*, 21(4):543–565, 1995.

[24] Gerard Briscoe and Alexandros Marinos. Digital Ecosystems in the Clouds: Towards Community Cloud Computing. In *the International Conference on Digital Ecosystems and Technologies*, pages 103–108, 2009.

[25] Andreas Brunnert, André van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, et al. Performance-oriented DevOps: A Research Agenda. *arXiv preprint arXiv:1508.04752*, 2015.

[26] CA Labs. Cloud Computing Web-Services Offering and IT Management Aspects. In *the Conference on Object Oriented Programming Systems Languages and Applications*, pages 27–39, 2009.

[27] Faruk Caglar, Kyoungho An, Shashank Shekhar, and Aniruddha Gokhale. Model-Driven Performance Estimation, Deployment, and Resource Management for Cloud-Hosted Services. In *the workshop on Domain-specific modeling*, pages 21–26, 2013.

[28] Alison Cartlidge, Ashley Hanna, Colin Rudd, Ivor Macfarlane, John Windebank, and Stuart Rance. *An introductory overview of ITIL® V3*, volume 1. The UK Chapter of the itSMF, 2007.

[29] Clovis Chapman, Wolfgang Emmerich, Fermín Galán Márquez, Stuart Clayman, and Alex Galis. Software Architecture Definition for On-demand Cloud Provisioning. *Cluster Computing*, 15(2):79–100, 2012.

[30] Stuart Charlton. Model Driven Design and operations for the Cloud. In *the Conference on Object Oriented Programming Systems Languages and Applications*, pages 17–26, 2009.

[31] Chef Software Inc. Online: https://www.chef.io. [Accessed: 10- May- 2016].

[32] Mihai Christodorescu, Reiner Sailer, Douglas Lee Schales, Daniele Sgandurra, and Diego Zamboni. Cloud Security is Not (Just) Virtualization Security: A Short Paper. In *the workshop on Cloud computing security*, pages 97–102, 2009.

[33] Andrea Ciancone, Antonio Filieri, Mauro Luigi Drago, Raffaela Mirandola, and Vincenzo Grassi. *KlaperSuite: An Integrated Model-Driven Environment for Reliability and Performance Analysis of Component-Based Systems*, volume 1 of *Objects, Models, Components, Patterns*, pages 99–114. Springer, 2011.

[34] Rudi Cilibrasi and Paul Vitanyi. The Google Similarity Distance. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):370–383, 2007.

[35] Allan Clark, Stephen Gilmore, Jane Hillston, and Mirco Tribastone. *Stochastic Process Algebras*, volume 1 of *Formal Methods for Performance Evaluation*, pages 132–179. Springer, 2007.

[36] Cloud Harmony. State of the Cloud - Compute. Technical report, Cloud Harmony Inc., 2014.

[37] Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2008.

[38] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-based software performance analysis*. Springer Berlin Heidelberg, 2011.

[39] Vittorio Cortellessa and Raffaela Mirandola. Deriving a queueing network based performance model from UML diagrams. In *the international workshop on Software and performance*, pages 58–70, 2000.

[40] Giuseppina Cretella and Beniamino Di Martino. Semantic and matchmaking technologies for discovering, mapping and aligning cloud providers's services. In *the*

International Conference on Information Integration and Web-based Applications & Services, pages 380–384, 2013.

[41] Isabel Cruz, Flavio Palandri Antonelli, and Cosmin Stroe. AgreementMaker: Efficient Matching for Large Real-world Schemas and Ontologies. *Proceedings of the VLDB Endowment*, 2(2):1586–1589, 2009.

[42] Krzysztof Czarnecki and Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[43] Andrea D'Ambrogio and Paolo Bocciarelli. A Model-Driven Approach to Describe and Predict the Performance of Composite Service. In *the international workshop on Software and performance*, pages 78–89, 2007.

[44] Margaret Dawson. Debunking the Top Three Cloud Security Myths. Online: https://blog.cloudsecurityalliance.org/2011/03/30/debunking-the-top-three-cloud-security-myths/. [Accessed: 10- May- 2016].

[45] Thijmen de Gooijer, Anton Jansen, Heiko Koziolek, and Anne Koziolek. An Industrial Case Study of Performance and Cost Design Space Exploration. In *the joint WOSP/SIPEW international conference on Performance Engineering*, pages 205–216, 2012.

[46] David Emery and Rich Hilliard. Every Architecture Description Needs a Framework: Expressing Architecture Frameworks Using ISO/IEC 42010. In *the European Conference on Software Architecture*, pages 31–40, 2009.

[47] Jérôme Euzenat and Pavel Shvaiko. *Ontology Matching*. Springer, 2007.

[48] Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin, and Arnor Solberg. Managing Multi-Cloud Systems with CloudMF. In *the Second Nordic Symposium on Cloud Computing & Internet Technologies*, pages 38–45, 2013.

[49] Philip Fleming and John Wallace. How Not to Lie With Statistics: The Correct Way to Summarize Benchmark Results. *Communications of the ACM*, 29(3):218–221, 1986.

[50] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-2, Electrical Engineering and Computer Sciences University of California at Berkeley, 2009.

[51] Davide Franceschelli, Danilo Ardagna, Michele Ciavotta, and Elisabetta Di Nitto. SPACE4CLOUD: A Tool for System Performance and Costevaluation of Cloud Systems. In *the international workshop on Multi-cloud applications and federated clouds*, pages 27–34, 2013.

[52] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. Enhanced Modeling and Solution of Layered Queueing Networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2009.

[53] Sören Frey and Wilhelm Hasselbring. Model-Based Migration of Legacy Software Systems into the Cloud: The CloudMIG Approach. In *the Workshop on Software-Reengineering*, pages 1–2, 2010.

[54] Hansjorg Fromm and Stephan Bloehdorn. Big Data Technologies and Potential. In *Enterprise Integration*, VDI-Buch, pages 107–124. Springer Berlin Heidelberg, 2014.

[55] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse is Still So Hard. *IEEE Software*, 26(4):66–69, 2009.

[56] Gartner Research. Forecast: Public Cloud Services, Worldwide, 2011-2017, 4Q13 Update. Technical Report G00248731, Gartner Inc., 2011.

[57] Glauco Gonçalves, Patricia Endo, Marcelo Santos, Djamel Sadok, et al. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *the International Conference on Cloud Computing Technology and Science*, pages 399–406, 2011.

[58] Google Inc. Python Runtime Environment - Google App Engine. Online: http://code.google.com/appengine/docs/python/runtime.html. [Accessed: 10- May- 2016].

[59] Jorge Gracia and Eduardo Mena. Web-Based Measure of Semantic Relatedness. In *the International Conference on Web Information Systems Engineering*, pages 136–150, 2008.

[60] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *ACM SIGCOMM - Computer Communication Review*, 39:68–73, December 2008.

[61] Michele Guerriero, Michele Ciavotta, Giovanni Paolo Gibilisco, and Danilo Ardagna. A Model-Driven DevOps Framework for QoS-Aware Cloud Applications. In *the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 345–351, 2015.

[62] Joaquín Guillén, Javier Miranda, Juan Manuel Murillo, and Carlos Canal. A Service-Oriented Framework for Developing Cross Cloud Migratable Software. *Journal of Systems and Software*, 86(9):2294–2308, 2013.

[63] Joaquín Guillén, Javier Miranda, Juan Manuel Murillo, and Carlos Canal. A UML Profile for Modeling Multicloud Applications. In *Service-Oriented and Cloud Computing*, pages 180–187. Springer, 2013.

[64] Mohammad Hamdaqa. The Stratus Modeling Language. Online: http://www.stargroup.uwaterloo.ca/ mhamdaqa/stratusml/. [Accessed: 10- May- 2016].

[65] Mohammad Hamdaqa. A Birdś-Eye View on Modelling Malleable Multi-Cloud Applications. In *the International Conference on Cloud Engineering*, pages 505–507, 2015.

[66] Mohammad Hamdaqa, Tassos Livogiannis, and Ladan Tahvildari. A Reference Model for Developing Cloud Applications. In *the International Conference on Cloud Computing and Services Science*, pages 98–103, 2011.

[67] Mohammad Hamdaqa and Ladan Tahvildari. Cloud Computing Uncovered: A Research Landscape. *Advances in Computers*, 86:41–85, 2012.

[68] Mohammad Hamdaqa and Ladan Tahvildari. Prison-Break: A Generic Schema Matching Solution to the Cloud Vendor Lock-in Problem. In *the International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based System*, pages 37–46, 2014.

[69] Mohammad Hamdaqa and Ladan Tahvildari. The (5+1) Architectural View Model for Cloud Applications. In *the IBM Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 46–60, 2014.

[70] Mohammad Hamdaqa and Ladan Tahvildari. StratusML: A Layered Cloud Modeling Framework. In *the International Conference on Cloud Engineering*, pages 96–105, 2015.

[71] Mohammad Hamdaqa and Ladan Tahvildari. Liberate: On the Application of Schema Matching for Public and Private Cloud. *Journal of Systems and Software*, 2016. (Under Review).

[72] Till Haselmann, Gottfried Vossen, Stefanie Lipsky, and Theurl Theresia. Cooperative Community Clouds for Small and Medium Enterprises: Facilitating Cloud Computing for Small and Medium Enterprises with the Cooperative Paradigm. In *the International Conference on Cloud Computing and Services Science*, pages 104–109, 2011.

[73] Bernhard Haslhofer and Wolfgang Klas. A Survey of Techniques for Achieving Metadata Interoperability. *ACM Computing Surveys*, 42(2):1–42, 2010.

[74] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process Algebra for Performance Evaluation. *Theoretical computer science*, 274(1):43–87, 2002.

[75] Paul Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. *Computing Systems*, 15:1–40, 2001.

[76] International Organization for Standardization and the International Electrotechnical Commission. ISO/IEC/IEEE 42010, Systems and Software Engineering - Architecture Description, December 2011.

[77] Mohd Adham Isa, Mohd Zaki, and Dayang Jawawi. A Survey of Design Model for Quality Analysis: From a Performance and Reliability Perspective. *Computer and Information Science*, 6(2):55–70., 2013.

[78] Farhana Islam, Dorina Petriu, and Murray Woodside. *Simplifying Layered Queuing Network Models*, pages 65–79. Computer Performance Engineering. Springer International Publishing, 2015.

[79] Mustafa Jarrar. Towards the Notion of Gloss, and the Adoption of Linguistic Resources in Formal Ontology Engineering. In *the International Conference on World Wide Web*, pages 497–503, 2006.

[80] Keith Jeffery, Nikos Houssos, Brigitte Jörg, and Anne Asserson. Research Information Management: the CERIF Approach. *International Journal of Metadata, Semantics and Ontologies*, 9(1):5–14, 2014.

[81] Jay Jiang and David Conrath. Semantic Similarity Based on Corpus Statistics and Lexical Taxonomy. In *the International Conference on Research in Computational Linguistics*, pages 1–15, 1997.

[82] Ben Kepes. Moving your Infrastructure to the Cloud: How to Maximize Benefits and Avoid Pitfalls. White paper, Diversity Limited and Rackspace, March 2011. Online : http://www.rackspace.com/knowledge _center/whitepaper/ moving-your-infrastructure-to-the-cloud-how-to-maximize-benefits-and-avoid-pitfalls.

[83] Lennard Kerber. Scenario-Based Performance Evaluation of SDL/MSC-Specified Systems. In *Performance Engineering*, pages 185–201. Springer, 2001.

[84] Anthony Klug and Dennis Tsichritzis. Multiple View Support within the Ansi/Sparc Framework. In *the International Conference on Very Large Data Bases*, volume 3, pages 477–488, 1977.

[85] Bastian Koller. Model Based Cloud Application Development using PaaSage. *Innovatives Supercomputing in Deutschland*, 11(1), 2013.

[86] Dimitrios Kolovos, Louis Rose, and James Williams. Using Model-to-Text Transformation for Dynamic Web-based Model Navigation. In *the International Workshop on Models at Runtime*, pages 1–12, 2011.

[87] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. Winery - A Modeling Tool for TOSCA-Based Cloud Applications. In *Service-Oriented Computing*, pages 700–704. Springer, 2013.

[88] Samuel Kounev, Fabian Brosig, and Nikolaus Huber. The Descartes Modeling Language. *Department of Computer Science, University of Wuerzburg, Tech. Rep*, 2014.

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

[90] Kyriakos Kritikos, Jorg Domaschka, and Alessandro Rossini. SRL: A Scalability Rule Language for Multi-Cloud Environments. In *the International Conference on Cloud Computing Technology and Science*, pages 1–9, 2014.

[91] Philippe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[92] Grace Lewis. Role of Standards in Cloud-Computing Interoperability. In *the International Conference on System Sciences*, pages 1652–1661, 2013.

[93] Xiao-Yong Li, Li-Tao Zhou, Yong Shi, and Yu Guo. A Trusted Computing Environment Model in Cloud Architecture. In *the International Conference on Machine Learning and Cybernetics*, volume 6, pages 2843–2848, 2010.

[94] Peter Linington. RM-ODP: the Architecture. In *Open Distributed Processing*, pages 15–33. Springer, 1995.

[95] Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Damien Tamburri. Providing Architectural Languages and Tools Interoperability Through Model Transformation Technologies. *IEEE Transactions on Software Engineering*, 36(1):119–140, 2010.

[96] Rahul Malik, Venkata Subramaniam, and Saroj Kaushik. Automatically Selecting Answer Templates to Respond to Customer Emails. In *the International Joint Conference on Artifical Intelligence*, pages 1659–1664, 2007.

[97] Anan Marie and Avigdor Gal. On the Stable Marriage of Maximum Weight Royal Couples. In *the International Workshop on Information Integration on the Web*, pages 1–6, 2007.

[98] Alexandros Marinos and Gerard Briscoe. Community Cloud Computing. *Cloud Computing*, pages 472–484, 2009.

[99] Chris Matthews, Stephen Neville, Yvonne Coady, Jeff McAffer, and Ian Bull. Overcast: Eclipsing High Profile Open Source Cloud Initiatives. In *the Conference on Object Oriented Programming Systems Languages and Applications*, pages 7–15, 2009.

[100] Michael Maximilien, Ajith Ranabahu, Roy Engehausen, and Laura Anderson. Toward Cloud-Agnostic Middlewares. In *the Conference on Object Oriented Programming Systems Languages and Applications*, pages 619–626, 2009.

[101] Nenad Medvidovic, Eric Dashofy, and Richard Taylor. Moving Architectural Description From Under the Technology Lamppost. *Information and Software Technology*, 49(1):12–31, 2007.

[102] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Recommendations of the National Institute of Standards and Technology Special Publication 800-145, National Institute of Standards and Technology, 2009.

[103] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In *the International Conference on Data Engineering*, pages 1–12, 2002.

[104] Daniel A Menasce, Virgilio AF Almeida, Lawrence W Dowdy, and Larry Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall PTR, 2004.

[105] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.

[106] Donald Metzler, Susan Dumais, and Christopher Meek. *Similarity Measures for Short Segments of Text*. Springer, 2007.

[107] Trend Micro. Top 5 Myths of Cloud Computing Security. Online: http://cloudsecurity.trendmicro.com/top-5-myths-of-cloud-computing-security/. [Accessed: 10- May- 2016].

[108] Microsoft. Overview of Windows Azure Service Architecture. Online: http://msdn.microsoft.com/en-us/library/dd179341.aspx. [Accessed: 10- May- 2016].

[109] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. Corpus-based and Knowledge-based Measures of Text Semantic Similarity. In *the National Conference on Artificial Intelligence - Volume 1*, pages 775–780, 2006.

[110] Michael Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, 100(9):913–917, 1982.

[111] Jon Moore. Cloud Computing Economics: Technical and Business Information about Cloud Computing. Online: http://www.cloudcomputingeconomics.com/2009/01/experience-curves-for-data-center.html. [Accessed: 10- May- 2016].

[112] Peter Mork, Len Seligman, Arnon Rosenthal, Joel Korb, and Chris Wolf. The Harmony Integration Workbench. In *Journal on Data Semantics XI*, volume 5383 of *Lecture Notes in Computer Science*, pages 65–93. 2008.

[113] Francesco Moscato, Rocco Aversa, Beniamino Di Martino, T Fortis, and Victor Munteanu. An analysis of mOSAIC Ontology for Cloud Resources Annotation. In *the Federated Conference on Computer Science and Information Systems*, pages 973–980, 2011.

[114] James Munkres. Algorithms for The Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[115] National Institute of Standards and Technology (NIST). NIST Cloud Computing Reference Architecture, V.1, March 2011. Online: http:// collaborate.nist.gov/ twiki-cloud-computing/ pub/ CloudComputing/ ReferenceArchitectureTaxonomy/ NIST_CC _Reference_Architecture_v1_March_30_2011.pd.

[116] Gonzalo Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[117] Dinh Khoa Nguyen, Francesco Lelli, Mike P Papazoglou, and Willem-Jan Van Den Heuvel. Blueprinting Approach in Support of Cloud Computing. *Future Internet*, 4(1):322–346, 2012.

[118] Hung Quoc Viet Nguyen, Xuan Hoai Luong, Zoltán Miklós, Tho Thanh Quan, and Karl Aberer. Collaborative Schema Matching Reconciliation. In *On the Move to Meaningful Internet Systems*, volume 8185, pages 222–240, 2013.

[119] Michael Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 1st edition, 2007.

[120] Department of Defense Architecture Framework Working Group et al. DoD Architecture Framework, Version 1.5. Department of Defense, USA, 2007.

[121] Daniel O'Leary. Impediments in the Use of Explicit Ontologies for KBS Development. *International Journal of Human-Computer Studies*, 46(2):327–337, 1997.

[122] Tariq Omari, Greg Franks, Murray Woodside, and Amy Pan. Solving Layered Queueing Networks of Large Client-Server Systems with Symmetric Replication. In *the international workshop on Software and performance*, pages 159–166, 2005.

[123] Object Management Group (OMG). A UML Profile for MARTE: Modeling And Analysis Of Real-Time Embedded Systems, V.1.1, 2011.

[124] Douglas Parkhill. *The Challenge of the Computer Utility*. Addison-Wesley Pub. Co., first edition, January 1966.

[125] Dana Petcu, Beniamino Di Martino, Salvatore Venticinque, et al. Experiences in Building a mOSAIC of Clouds. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):12, 2013.

[126] Dana Petcu, Georgiana Macariu, Silviu Panica, and Ciprian Crăciun. Portable Cloud Applications-From Theory to Practice. *Future Generation Computer Systems*, 29(6):1417–1430, 2013.

[127] Dorin Petriu and Murray Woodside. *Software Performance Models from System Scenarios in Use Case Maps*, pages 141–158. Software Performance Models from System Scenarios in Use Case Maps. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[128] Dorin Petriu and Murray Woodside. An Intermediate Metamodel with Scenarios and Resources for Generating Performance Models from UML Designs. *Software & Systems Modeling*, 6(2):163–184, 2007.

[129] Dorina Petriu. *Software model-based performance analysis*, volume 1 of *Model-Driven Engineering for Distributed Real-Time Systems: MARTE Modeling, Model Transformations and their Usages*, pages 1–19. John Wiley & Sons Inc., 2010.

[130] Dorina Petriu, Mohammad Alhaj, and Rasha Tawhid. Software Performance Modeling. In *Formal Methods for Model-Driven Engineering*, pages 219–262. Springer, 2012.

[131] Dorina Petriu and Murray Woodside. Performance Analysis with UML. In *UML for Real*, pages 221–240. Springer, 2003.

[132] Martin Porter. An Algorithm for Suffix Stripping. *Program*, 14(3):130–137, 1980.

[133] Puppet Labs. Online: https://puppet.com. [Accessed: 10- May- 2016].

[134] Xiaojun Quan, Gang Liu, Zhi Lu, Xingliang Ni, and Liu Wenyin. Short Text Similarity Based on Probabilistic Topics. *Knowledge and Information Systems*, 25(3):473–491, 2010.

[135] Clément Quinton, Daniel Romero, and Laurence Duchien. Cardinality-Based Feature Models with Constraints: A Pragmatic Approach. In *the International Software Product Line Conference*, pages 162–166, 2013.

[136] Erhard Rahm and Philip Bernstein. A Survey of Approaches to Automatic Schema Matching. *International Journal on Very Large Data Bases*, 10(4):334–350, 2001.

[137] Ajith Ranabahu, E Maximilien, Amit Sheth, and Krishnaprasad Thirunarayan. Application Portability in Cloud Computing: An Abstraction Driven Perspective. *IEEE Transactions on Services Computing*, 99(1):1–14, 2013.

[138] Ajith H Ranabahu, Eugene Michael Maximilien, Amit P Sheth, and Krishnaprasad Thirunarayan. A Domain Specific Language for Enterprise Grade Cloud-Mobile Hybrid Applications. In *the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, pages 77–84, 2011.

[139] George Reese. The Economics of Cloud Computing. Online: http://broadcast.oreilly.com/2008/10/the-economics-of-cloud-c.html. [Accessed: 10- May- 2016].

[140] Benny Rochwerger, David Breitgand, Eliezer Levy, Alex Galis, Kenneth Nagin, Ignacio M Llorente, Ruben Montero, Yaron Wolfsthal, Erik Elmroth, Juan Caceres, et al. The Reservoir Model and Architecture for Open Federated Cloud Computing. *IBM Journal of Research and Development*, 53(4):4–1, 2009.

[141] Vasile Rus, Mihai C Lintean, Rajendra Banjade, Nobal B Niraula, and Dan Stefanescu. SEMILAR: The Semantic Similarity Toolkit. In *the Annual Meeting of the Association for Computational Linguistics*, pages 163–168, 2013.

[142] Rohjans Santodomingo, Sebastian Rohjans, Mathias Uslar, JA Rodríguez-Mondéjar, and Miguel Sanz-Bobi. Ontology Matching System for Future Energy Smart Grids. *Engineering Applications of Artificial Intelligence*, 32:242–257, 2014.

[143] Bran Selic, Alan Moore, Murray Woodside, Ben Watson, Morgan Bjorkander, Mark Gerhardt, and Dorina Petriu. UML Profile for Schedulability, Performance and Time. *OMG document*, 1.1, 2005.

[144] Pavel Shvaiko and Jérôme Euzenat. A Survey of Schema-Based Matching Approaches. In *Data Semantics IV*, volume 3730 of *Lecture Notes in Computer Science*, pages 146–171. Springer, 2005.

[145] Pavel Shvaiko and Jérôme Euzenat. Ontology Matching: State of the Art and Future Challenges. *IEEE Transactions on Knowledge and Data Engineering*, 25(1):158–176, 2013.

[146] Michael Smit. Supporting Software Evolution to the Multi-cloud with a Cross-Cloud Platform. In *the International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pages 85–85, 2013.

[147] Connie Smith and Lloyd Williams. *Performance solutions: a practical guide to creating responsive, scalable software*, volume 1. Addison-Wesley Reading, 2002.

[148] James Staten, Ted Schadler, John Rymer, and Chenxi Wang. Q & A : By 2011 , CIOs Must Answer The Question, Why not Run In The Cloud? Technical report, Forrester Research Inc., 2009. Retrieved from: http://www.forrester.com/rb/Research/q

[149] Clemens Szyperski, Jan Bosch, and Wolfgang Weck. Component-Oriented Programming. In *the European Conference on Object-Oriented Technology*, pages 795–795, 1999.

[150] Richard N Taylor, Nenad Medvidovic, and Peyman Oreizy. Architectural Styles for Runtime Software Adaptation. In *the Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, pages 171–180, 2009.

[151] Johannes Thones. Microservices. *IEEE Software*, 32(1):116–116, 2015.

[152] Nidhi Tiwari and Prabhakar Mynampati. Experiences of Using LQN and QPN Tools for Performance Modeling of a J2EE Application. In *the CMG-CONFERENCE*, volume 1, pages 537–547, 2006.

[153] Javier Troya, Hugo Brunelière, Martin Fleck, Manuel Wimmer, Leire Orue-Echevarria, and Jesús Gorroñogoitia. ARTIST: Model-Based Stairway to the Cloud. In *the Software Technologies: Applications and Foundations conference - Project Showcase*, L'Aquila, Italy, July 2015.

[154] Wei-Tek Tsai, Xin Sun, and Janaka Balasooriya. Service-Oriented Cloud Computing Architecture. In *the International Conference on Information Technology: New Generations*, pages 684–689, 2010.

[155] Konstantinos Tsakalozos, Mema Roussopoulos, Vangelis Floros, and Alex Delis. Nefeli: Hint-Based Execution of Workloads in Clouds. In *the International Conference on Distributed Computing Systems*, pages 74–85, 2010.

[156] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[157] TOGAF Version 9. The Open Group Architecture Framework. *The Open Group*, 1:692, 2009.

[158] Padmal Vitharana, Hemant Jain, and Fatemeh Mariam Zahedi. Strategy-Based Design of Reusable Business Components. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 34(4):460–474, 2004.

[159] Murray Woodside and Greg Franks. Tutorial Introduction to Layered Modeling of Aoftware Performance. Tech. rep, Department of Systems and Computer Engineering, Carleton University, 2013.

[160] Murray Woodside, Dorina Petriu, Dorin Petriu, Hui Shen, Toqeer Israr, and Jose Merseguer. Performance by unified model analysis (PUMA). In *the international workshop on Software and performance*, pages 1–12, 2005.

[161] Xiuping Wu and Murray Woodside. Performance modeling from software components. *ACM SIGSOFT Software Engineering Notes*, 29(1):290–301, 2004.

[162] John Zachman. A Framework for Information Systems Architecture. *IBM Systems Journal*, 26(3):276–292, 1987.

[163] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *the European conference on Computer systems*, pages 265–278, 2010.

[164] Liang-Jie Zhang and Jia Zhang. Architecture-Driven Variation Analysis for Designing Cloud Applications. In *the International Conference on Cloud Computing*, pages 125–134, 2009.

[165] Joe Zou and Christopher Pavlovski. Modeling Architectural Non Functional Requirements: From Use Case to Control Case. In *the International Conference on e-Business Engineering*, pages 315–322, 2006.