Universität Stuttgart

Design Support for Performance- and Cost-efficient (Re)Distribution of Cloud Applications

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Santiago Gómez Sáez

aus Montevideo, Uruguay

Hauptberichter:	Prof. Dr. Dr. h. c. Frank Leymann
Mitberichter:	Prof. Kostas Magoutis
	Dr. Vasilios Andrikopoulos

Tag der mündlichen Prüfung: 23.07.2019

Institut für Architektur von Anwendungssystemen der Universität Stuttgart

2019

Contents

Zusammenfassung

Im letzten Jahrzehnt hat die IT-Landschaft einen revolutionären Wandel sowohl in der Forschung als auch in der Industrie erlebt, der auf die starke Entwicklung und Akzeptanz von Cloud-Anbietern und -Diensten zurückzuführen ist. Der Anstieg der verfügbaren Cloud-Dienste in Kombination mit der Anwendung von DevOps-Methoden hat eine grundlegende Veränderung erzwungen, wie Software entwickelt und betrieben wird. Heutzutage können Unternehmen ein breites Spektrum an Cloud-Services nutzen, um ihre Anwendungskomponenten in der Cloud neu zu entwickeln, zu migrieren, großflächig zu verteilen oder bestehende Komponenten durch native Cloud-Anwendungen zu ersetzen. Die Umwandlung von verschiedenen IT-bezogenen Ausgaben (wie beispielsweise der Betrieb eigener Hardware) in operative Kosten ist Realität geworden.

Das Aufkommen von Cloud-Diensten brachte jedoch eine erhebliche Komplexität beim Design und Betrieb von Cloud-Anwendungen mit sich. Genauer betrachtet bedienen Cloud-Anwendungen immer eine

Vielzahl von Konsumenten gleichzeitig, die Rechenzeit, Netzwerk- und Speicherressourcen auf On-Demand-Basis nutzen. Diese Art der virtualisierten Server-Umgebung bewirkt, dass Leistung variabel und heterogen abgerufen wird, was sich wiederum auf die Kostenstruktur der Anwendungen auswirkt. Bei der Migration von Anwendungen in die Cloud führt diese Heterogenität zu komplexeren architektonischen Entscheidungen, was vor allem auf das oft begrenzte Wissen über Cloud Services zurückzuführen ist.

Diese Arbeit nimmt sich der oben genannte Herausforderung der Entscheidungsfindung an, indem sie ein Framework zur Unterstützung von Designentscheidungen namens SCARF definiert und prototypisch realisiert, um IT-Organisationen bei der effizienten Migration ihrer Anwendungen in die Cloud zu unterstützen. Der Schwerpunkt liegt dabei auf der Optimierung von Kosten und Leistung. SCARF ist also ein Framework zur Unterstützung von Entscheidungen, um Anwendungen systematisch in die Cloud zu migrieren. SCARF bietet Anwendungsarchitekten dabei einen Entwurfsprozess inklusive Software-Tools um (i) ihre Zielarchitekturen zu modellieren, (ii) potenzielle Cloud-Services automatisch zu identifizieren, um bestehende Komponenten zu (re-)implementieren, (iii) sie hinsichtlich ihrer Kosten und Leistung zu bewerten und (iv) den Verlauf von Leistung und Kosten während der Laufzeit zu überwachen und zu analysieren. SCARF baut dabei auf bestehenden Konzepten und Techniken im Bereich von Cloud-Anwendungstopologien auf, um die Migration traditioneller Anwendungen in die Cloud zu vereinfachen. Dazu zählen das *case-based reasoning*, die Ähnlichkeitsanalyse und die *utility theory*.

Die technische Unterstützung für SCARF und SCARF-T umfasst eine integrierte Werkzeugkette, die auf existierenden Standards auf-

baut und bestehende Technologien und Tools wiederverwendet. Die Evaluation von SCARF und SCARF-T zeigt einen effizienteren und einfacheren Entscheidungsprozess bei der Migration von Anwendungen in die Cloud auf. Diese Aussage wird zusätzlich durch zahlreiche wissenschaftliche Publikationen belegt, welche die Erkenntnisse und Ergebnisse dieser Arbeit untermauern. Es ist festzustellen, dass SCARF als Ausgangspunkt betrachtet werden soll, um IT-Organisationen bei der Entscheidungsfindung zu unterstützen, ob und wie sie ihre Anwendungslandschaft in die Cloud migrieren. Dabei liegt der Schwerpunkt auf der Optimierung des gesamten Spektrums von nicht-funktionalen Aspekten.

Abstract

In the last decade the IT landscape has experienced a revolutionary change in both the research and industry domains due to the strong emergence and adoption of cloud providers and services. The increase of available cloud services together with the materialization of DevOps methodologies have forced a change in how software is engineered and operated. Nowadays, organizations can leverage a wide spectrum of cloud services to build, migrate, distribute, and even replace their application components in the cloud. In summary, the transformation of capital into operational expenditures when using private or public clouds has become a reality.

The emergence of cloud services, however, brought with it a substantial complexity when designing and running cloud applications. More specifically, cloud environments are intrinsically multi-tenant, where different tenants (or cloud consumers) consume computational, network, and storage resources on an on-demand basis. This type of virtualized environment causes performance to be variable and

heterogeneous, therefore having consequences on application costs. When migrating applications to the cloud, this heterogeneity causes architectural decision making to be more complex, mainly due to the limited amount of knowledge about cloud services.

This work tackles the aforementioned decision making challenge, by defining and realizing a design decision support framework called SCARF to assist IT organizations to efficiently migrate their applications to the cloud, with a focus on optimizing their cost and performance. In summary, SCARF is a design and decision support framework for *systematically (re)distributing applications in the cloud*. SCARF equips application architects with a design process and tools (i) to model their application architectures, (ii) to automatically discover potential cloud services to (re-)deploy their application components, (iii) to evaluate them with respect to cost and performance, and (iv) to monitor and analyze application costs and performance evolution during runtime. SCARF builds upon existing concepts and techniques in the domain of cloud application topologies, case base reasoning, similarity analysis, and utility theory, in order to simplify the steps for re-engineering and migrating traditional applications to the cloud.

The technological support for SCARF, SCARF-T, comprises an integrated tool chain built upon prior standards and reusing existing technologies and tools. The evaluation of SCARF and SCARF-T show a more efficient and simplified decision making process for migrating applications to the cloud, as also shown by the publications supporting this thesis. Overall, SCARF can be considered as a starting point for enabling IT organizations with architectural decision support for migrating their applications to the cloud, with a focus on optimizing the complete spectrum of application non-functional aspects.

Acknowledgements

Writing this thesis has been a journey that is probably hard to describe with words. For those who know me, you will realize that one of my biggest passions in life is sailing. I would like to use this as an analogy.

Before starting a sailing trip, also known as a passage, there are some tasks that are mandatory. These are (i) finding and setting the destination in a chart, (ii) calculating the course to steer, and (iii) calculating the impact of wind and stream on the course to steer. In some cases, the destination is located upwind, meaning that it wind is coming in the opposite direction to the destination point. Sailing upwind is physically not possible. The only way to sail upwind is by tacking – zig-zagging following a 45-degree angle w.r.t. the wind direction. Doing this efficiently takes more skill and practice than anything else in sailing. If you learn it well, then you can sail anywhere. This can be compared to writing a dissertation, as in a first step a research domain and general challenge are defined. However, it is practically impossible to navigate and resolve it straightforward.

The challenge should be decomposed into small problems, which can be tackled and solved independently, in a similar way as to zig-zagging upwind. When solving them all, the journey is completed.

First and foremost, I want to deeply thank my supervisor Prof. Dr. Dr. h. c. Frank Leymann not only for giving me the opportunity to join this journey, but also for his support, knowledge, enthusiasm, and encouragement during this thesis. Working with you and being part of the Institute of Architecture of Application Systems (IAAS) has been a wonderful opportunity. I also want to thank Prof. Kostas Magoutis for finding the time to review the thesis and for being a member of the Ph.D. committee. There are probably not enough words to describe my gratitude to Dr. Vasilios Andrikopoulos. Starting working with you as a student research assistant and ending up writing this thesis was a very long journey working with you in different projects, having plenty of discussions, and most importantly, learning everyday something new and keeping motivation high.

During my time in the IAAS institute, I had the pleasure to meet and work with a group of brilliant professionals, of which some have become good friends. I would like to start thanking Steve Strauch, not only for introducing me into the world of cloud computing, but also for helping and motivating me during this thesis. Big thanks also to Dr. Marigianna Skouradaki, specially for all those fun moments at the office. To the SimTech IAAS colleagues, Prof. Dr. Dimka Karastoyanova, Dr. Andreas Weiß, and Michael Hahn, thanks for such a pleasant collaboration. Thank you Dr. Andrè van Hoorn for sharing your knowledge in the field of performance engineering. Thanks also to Dr. Oliver Kopp for sharing his technical knowledge of OpenTOSCA. Thanks to each and every IAAS member for your rich feedbacks and great moments working with you.

To the ALLOW Ensembles and SitOPT consortia, many thanks for the opportunity to meet and collaborate with international researchers, as well as for the chance to learn new fields. In particular, thanks to Dr. Marina Bitsaki, Georgios Koutras, and Alina Psycharaki for our joint work and your hospitality. Special thanks go also to the group of students that have worked with me throughout these years. Roberto, Sherif Yassin, Jhonny, Florian, Hao, Maria Elena, and Kushal, thank you all for the great moments and learnings during your theses. To my colleagues at Volkswagen Financial Services (VWFS), many thanks for your patience and motivation, with special thanks to Vincenz Kopp for taking his time to review the thesis.

Lastly but not least, I would like to thank my family and friends for their continuous support and motivation. To my parents Gustavo and Cecilia, thank you for your unconditional love and for showing me the path and facilitating me to achieve what I like. To my brother Matias, and my sister Florencia, thank you for your support and encouragement. Especially to my future wife Natalia, your love and patience have no boundaries. Thank you for standing by and encouraging me all these years always with a smile. Kocham cię bardzo!!

Chapter 1

Introduction

1.1 Motivation

In the last years, cloud computing entered the IT landscape and forced a change in how software is designed, developed, and executed. The Rightscale State of the Cloud Report[1] discovered in 2016 that 95% of the organizations are using any type of cloud service, either in their testing and/or production environments, from which 81% are moving towards the adoption of hybrid clouds for their production environments.

NIST defines cloud computing as a *model for enabling 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*

[1] RightScale: https://www.rightscale.com/lp/state-of-the-cloud

effort or service provider interaction [MG11; BGPV11]. The cloud computing model is composed of three service models – Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS) – and four deployment models – *Private Cloud*, *Public Cloud*, *Hybrid Cloud*, and *Community Cloud*. For a more detailed description of the service and delivery models, the reader can refer to the complete NIST definition in [MG11].

Cloud environments bring to organizations business and operational advantages when designing and running their applications [EPM13]. On the one hand, Varia points out in [Var10] that business benefits are mainly related to the transformation of capital into operational expenditures. In other words, cloud computing requires almost zero upfront investments, as it is based on a pay-as-you-go model, and allows for an increase of operational expenditures tailored to business needs. For instance, if we consider a Web startup, it may need to rapidly provision cloud resources (scale up) to support an unpredictable spike in demand when it becomes popular. However, the spike will in the long term stabilize, therefore balancing infrastructure demands and scaling down to its initial infrastructure needs [AFG+10].

Concentrating on the management of resources, operating applications can be highly simplified and rapidly adapted in cloud environments. Cloud infrastructures are API-driven, meaning that managing resources can be automated by calling management operations, e.g., scale up or down resources, during the complete development and production life cycle [Var10]. Such management simplicity and adaptation features are aligned with the fundamental DevOps principles for enabling the rapid and automated development, release, and adaptation of software systems [BWZ15].

The years of the cloud landscape have been marked by a strong evo-

lution towards the Everything-as-a-Service (*aaS) delivery model and by the emergence of cloud providers and services [HK10; HK11]. For instance, Cloud Spectator[2] already identified 25 major and 10 major cloud providers in their 2016 European[3] and 2017 North America[4] cloud performance reports, respectively. Moreover, Gartner[5] forecasts a public cloud revenue growth of 17.3 % in 2019. The wide spectrum of available cloud providers and services have boosted the number of possibilities to migrate and build applications in the cloud [ABLS13]. Together with the adoption of DevOps principles [HM11], organizations are aiming at fully exploiting the diversity of cloud services to partially or completely distribute their applications [ABLS13], in order to rapidly respond to changing market demands.

Focusing on the three-tiered sample Web shop application depicted in Fig. 1.1, we can derive different distribution alternatives for partially or completelly migrating its three tiers, *Presentation*, *Business Logic*, and *Data* [Fow02]. The migration is strictly dependent on organizational constraints, as well as business and operational requirements. For instance, security and privacy policies may constitute the main constraints for migrating application data to a specific cloud region. Therefore, one possible alternative is to keep the business logic and data tiers in an on-premise OpenStack[6] private cloud, while migrating the presentation logic and caching functionalities to an

[2] Cloud Spectator: http://cloudspectator.com

[3] Cloud Spectator 2016 European Report: http://cloudspectator.com/reports/2016-Top-Ten-European-Cloud-Service-Providers.pdf

[4] Cloud Spectator 2017 North America Report: http://connect.cloudspectator.com/top-cloud-iaas-providers-benchmark-2017-pdf-download

[5] Gartner Forecast 2019:https://gtnr.it/2CTlJGA

[6] OpenStack: https://www.openstack.org

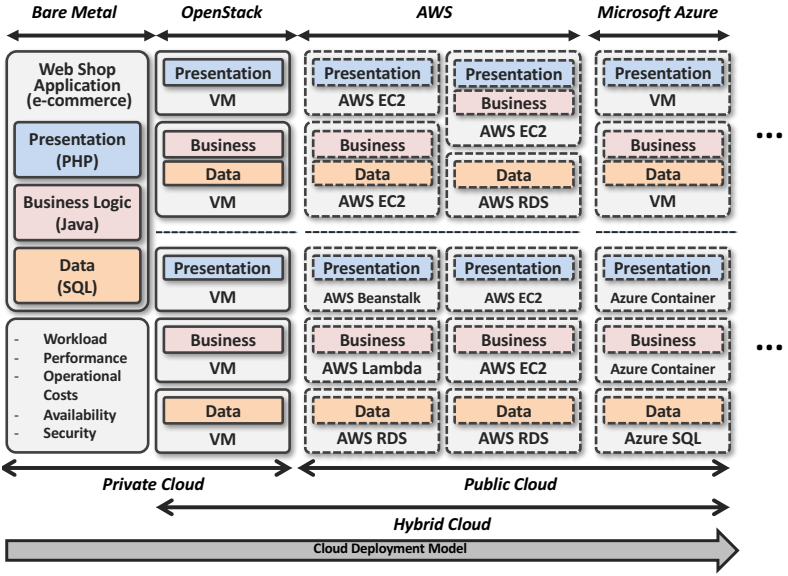


Figure 1.1: Web Shop Application Distribution - Example

Amazon EC2[7] or Microsoft Azure VM[8] instance.

Further possibilities comprise the usage of PaaS solutions to host the presentation and business tiers using Amazon Beanstalk[9] or Microsoft Azure Container Service[10]. In the case of no data privacy constraints, the Web shop's application data tier can leverage the usage of a

[7] AWS EC2: https://aws.amazon.com/ec2

[8] Microsoft Azure VM Service: https://azure.microsoft.com/en-us/services/virtual-machines

[9] AWS Elastic Beanstalk: https://aws.amazon.com/elasticbeanstalk

[10] Microsoft Azure Container Service: https://azure.microsoft.com/en-us/services/container-service

Database-as-a-Service (DBaaS) offering, such as Amazon RDS[11] or Microsoft Azure SQL[12].

The example in Fig. 1.1, however, considers two major providers and three service delivery models. In reality, there exists a vast amount of service providers and offerings. For instance, Schaffer identifies 44 types of services across different providers [Sch09]. Migrating the application to the cloud and distributing its components among heterogeneous cloud services is, therefore, a challenging task for IT organizations. There exists nowadays a common need for decision support mechanisms that can assist the discovery of viable cloud offerings during migration of applications to cloud environments. Several EU Projects – listed and monitored in the CloudWATCH EU Project[13] – have been launched in this direction, and focus on the different aspects of cloud migration, such as cost analysis, security compliance, development estimation, Quality of Service (QoS), etc., independently. This work also falls under this umbrella, but goes a step further, by means of combining QoS and costs aspects in a decision support framework for assisting the migration and distribution of applications in the cloud.

The remainder of this chapter presents and details the fundamental problems addressed in the scope of this work.

[11] AWS RDS: https://aws.amazon.com/rds

[12] Microsoft Azure SQL: https://azure.microsoft.com/en-us/services/sql-database

[13] CloudWATCH - A Portfolio of offers for trusted and secure services: http://www.cloudwatchhub.eu/sites/default/files/A-portfolio-of-offers-for-trusted-and-secure-services_Web.pdf

1.2 Problem Definition & Scope

The existence of a wide variety of cloud offerings has allowed organizations to migrate their applications to the cloud, by means of spanning their components among different cloud services and providers. However, as briefly highlighted in the previous section, deciding among cloud providers and services in an efficient manner is a challenging task. In other words, there exists a necessity for providing decision support mechanisms to assist IT organizations to efficiently select and configure distributions of their application components in the cloud.

Migrating applications to the cloud regularly requires the realization of architectural and compliance tasks, which are organized w.r.t the application's functional and non-functional aspects. Functional-related tasks consist of specifying the necessary underlying resources, such as computational, storage, etc. These also include development tasks, such as the adaptation of components and the development of deployment artifacts. On the other hand, non-functional-related tasks are related to the analysis and optimization of cost, performance, QoS, security, etc. As in [GALS14a], optimizing application non-functional aspects consists of solving a multi-dimensional problem, where each non-functional aspect represents one axis of the problem. The utilization and configuration of the different cloud services – each building towards a viable distribution of the application components – has an impact on each and every non-functional aspect.

Although existing decision support mechanisms target the optimal fulfillment of application's non-functional aspects, these typically address each aspect independently (an extended explanation is provided in Sec. 2.3). Since cost and performance are part of the highest organizational cloud migration concerns [Lal13], such dimensions are

analyzed jointly in this work. In other words, our mechanisms seek for the efficient selection of cloud offerings, focusing on the trade-off between cost and performance. Towards such a goal, there are two fundamental observations that we identified in [GALS14b]. Typically, cloud application topologies are used to depict the distribution of the applications components in the cloud, as mentioned in the literature [BFL+12; ARSL14]. Firstly, the distribution of the application in the cloud has a severe effect on its cost and performance– however it is not always obvious whether the effect is beneficial or detrimental. Secondly, a realistic application workload fluctuates over time, therefore requiring the adaptation of its distribution, i.e. a redistribution of its components.

Typically, cloud migration decisions are part of the *Application Architect* tasks. Rightscale identified in its 2016 State of the Cloud Report[14] an increase of 40% of application architect roles in organizations. For the scope of this work, we define the application architect tasks and responsibilities (aligned with Gesvindr et al. [GB16]) as:

- the selection and configuration of cloud providers and offerings,
- the feasibility study of the application's fitness for its successfull operation in a cloud environment w.r.t. its business and operational requirements, and
- the architectural design of the application considering the intrinsic characteristics of the selected cloud services, such as availability, scalability, etc.

Further roles identified and considered in this work are the *Business Architects*, *Application Developers*, and *End Users*. Business architects

[14] RightScale: `https://www.rightscale.com/lp/state-of-the-cloud`

are responsible for the definition, analysis, and maintenance of business objectives and capabilities [Kat97]. Application developers' tasks are related to the design and development of the application itself, as well as developing automated provisioning mechanisms and artifacts for application components. Lastly, end users utilize the application towards satisfying their individual needs, e.g., purchasing articles in a Web shop application.

The main beneficiary from this work are application architects and application developers, as these are responsible for deciding for cloud services during the application design's or migration phases [GB16]. Without the necessary knowledge and tooling support, decisions may drive towards an inefficient planning and allocation of monetary and computational resources, as cloud services offer contrasting performance levels for different types of applications. The existence of dedicated service types – following the *aaS delivery model –, each of them with independent cost models and QoS levels, demands for the enhancement of decision support mechanisms during the design phase of the application. This leads to defining the generic problem that this work addresses as:

> How to migrate applications to the cloud – by means of *distributing* the application components among multiple cloud services – while *optimizing for cost and performance*?

The decision support mechanisms in this work allow application architects to smoothly discover optimal allocations of cloud resources to host their distributed applications. Such mechanisms use as the basis past knowledge of the same or similar applications, and the analysis of the application's business and operational requirements.

It must be clarified beforehand that this work refers to migration as both the (i) redeployment and adaptation actions of the application components (according to the selected application distribution), and the (ii) actual operation of the resulted application distribution, as supported in [ABLS13].

This work puts a strong focus on business applications, i.e., applications that are built towards achieving an economic gain, according to Microsoft's application categorization[15]. Morover, the main assumption in our work relies on the certainty that the decision to migrate an application to the cloud has already been taken. We exclude from the discussion the development of cloud native applications, although the techniques and mechanisms in this work can also be partially applied for such scenarios.

1.3 Vision & Challenges

The central vision of this work relies on providing application architects with mechanisms and support for re-engineering their applications as part of their migration to the cloud. In particular, this work concentrates simultaneously on facilitating design decision making support for the discovery of cloud services, and for the distribution of application components among cloud offerings.

A major part of this work consists of providing the means to define, exploit, and evaluate applications' performance- and cost-related knowledge as part of the decision making support. Such a knowledge can be leveraged and combined with evaluation mechanisms to enhance the decision making for discovering, selecting, and dis-

[15] Microsoft - Application Types: https://blogs.msdn.microsoft.com/jmeier/2010/07/29/application-types-app-types-the-early-years

tributing application components among different cloud offerings. In particular, application architects can potentially be provided with alternative solutions for distributing their applications, by leveraging knowledge from previous decisions and complying to application functional and non-functional constraints. The main focus and outcome of the decision making support relates to analyzing and evaluating the trade-off between the cost and performance for different distribution alternatives of their applications, leading to the selection of optimal distributions of applications.

The remainder of this section identifies the challenges in this work, which are grouped w.r.t. two main aspects: the (i) *Cloud Application Migration Requirements* and the (ii) *Discovery, Selection & Evaluation of Cloud Offerings*. The former identifies the constraints when migrating applications to the cloud, focusing on the cost and performance aspects. The latter relates to achieving efficiency in such a migration, by means of identifying the challenges for discovering, selecting, and evaluating cloud offerings w.r.t. the application's business and operational requirements.

1.3.1 Cloud Application Migration Requirements

The migration of applications to the cloud is a major concern for organizations, as legacy applications were originally designed using traditional software engineering methods and techniques, without any cloud related concerns [JAP13]. The cloud allows the distribution of applications, by means of partially or completely distributing its components among heterogeneous cloud services, which can potentially replace original application components [ABLS13]. Such concerns leads to the definition of the following research challenges:

RCh.1 How to identify the requirements for migrating applications to the cloud?

Organizations require mechanisms to identify their business and operational requirements before and during the migration phase of applications to the cloud. More specifically, organizations need to identify or design IT performance indicators aligned with the business objectives.

RCh.2 How to model cloud applications considering cost and performance characteristics?

The development of cloud application architectural models has been part of major research and industrial projects, according to the EU Cordis organization[16] and the EU Project CloudWatch[17]. The modeling of application functional aspects has already been covered in research works and standards. However, there exists a gap for enabling the definition of non-functional aspects.

Focusing on the cost and performance of applications, we find the need for defining an enhanced architectural model that includes (i) application performance and cost constraints, and (ii) consolidates the intrinsic characteristics of the application's workload. More precisely, such an architectural model aims at capturing performance knowledge that can be leveraged towards constructing performance behavioral models, as defined in [Fei02; CS93; KYTA12; MHCD10; RAKJ15]. This enhanced cost- and performance-aware architectural model can

[16] EU FP7 Software & Services Projects Portfolio: https://cordis.europa.eu/projects/en

[17] EU Cloudwatch Project: http://www.cloudwatchhub.eu/sites/default/files/BookletA4_June2017_inner_v04_web.pdf

be the leveraged in the applications' design and decision making tasks.

1.3.2 Cloud Offerings Discovery, Selection & Evaluation

The previous section identified the challenges for defining migration requirements for cloud applications. However, the simple definition of constraints does not imply efficiency of a cloud migration. In other words, decision support mechanisms are necessary for automating the discovery and the efficient selection of cloud services.

A first step in decision making consists of identifying the decision making points, i.e., identifying the cloud migration concerns, as defined by Andrikopoulos et al. in [ADKL14; ASL13; SAB+13]. A second step consists of automating the decision making mechanisms seeking for optimal migration alternatives. Focusing on the cost and performance of cloud applications, this work identifies the following challenges:

> **RCh.3** How to facilitate cloud offering decision making, considering the application's evolving workload and cloud service's variable performance?

Cloud services are heterogeneous in terms of their performance and cost characteristics. On the one hand, performance of cloud offerings varies among providers and time. Moreover, application workloads can fluctuate over time. On the other hand, cloud services expose different cost models, which typically vary among providers and services.

Application architects lack of full knowledge of cloud providers and offerings. There exists a necessity for automating the decision making by means of developing mechanisms that:

1. automate the discovery of potential cloud providers and offerings, which can facilitate
2. the construction of cloud application distributions compatible with its functional aspects, while
3. considering business and operational performance and costs constraints.

Ideally, such mechanisms can leverage previous and similar knowledge and decisions.

> **RCh.4** How to identify the optimal performance and cost trade-off among cloud offerings for a specific application as aspect of decision making?

The single discovery and construction of compatible cloud application distributions still does not provide support for seeking optimality in the decision making of this work. Application architects require mechanisms to evaluate the trade-off between cost and performance of viable cloud application distributions. In particular, these need to aggregate (i) previous and present performance and cost knowledge, and (ii) mechanisms to evaluate during design time the business and operational impact of a viable distribution on business and operational requirements.

> **RCh.5** What tooling support is required for the cloud migration decision making tasks?

There exist tools that partially cover decision making aspects for the migration of applications to the cloud [KSBT11]. This work must analyze and evaluate existing tooling support, and must build the

necessary new ones, towards providing a tool chain to assist in the cost- and performance-efficient migration and distribution of applications in the cloud.

1.4 Research Contributions

This section summarizes the research contributions developed in this work. These are part of the following peer reviewed publications:

- 5 Conference papers published in the IEEE International Conference on Cloud Computing (IEEE CLOUD), in the European Conference on Service-Oriented and Cloud Computing (ESOCC), in the International Conference on Cloud Computing and Services Science (CLOSER), and in the International Conference on Advanced Information Systems Engineering (CAiSE).
- 2 Journal articles published in the IEEE Transactions on Services Computing (TSC) and in the ACM Transactions on Internet Technology (TOIT).
- 1 Chapter in the Cloud Computing and Services Science book.

A detailed overview of the publications associated with each contribution and research challenge is depicted in Fig. 1.2. Further publications, such as [AGKW13; AGKW14; ABG+16], are exclusively used as case studies for the evaluation in this work. These are introduced and depicted in Chapter 10. The remainder of this section identifies and summarizes the contributions of this work.

RCo.1 Empirical Study on Performance Variability and Design Challenges among Cloud Providers and Offerings

The cost and performance variability among cloud providers and services is an existing engineering challenge in organizations. A first step in this work consists of practically studying such a challenge. In particular, this work empirically studies the performance variation among cloud providers and offerings for different types of applications and cloud services.

The performance fluctuation among cloud services influences on how cloud applications should be engineered and where its components should be placed. The specification of application business and operational requirements must be decoupled from a concrete provider and must incorporate the necessary placeholders to specify business and operational requirements in a generic manner.

RCo.2 Cost- and Performance-aware Topology Model

The distribution of applications in the cloud requires an architectural model to describe the application components and cloud resources in a generic and portable manner. Cloud application topologies are well-known in the cloud domain, and therefore used as the basis in this work. A complete presentation of cloud application topologies is provided in Chapter 2.

Conventional cloud topology models, however, do not naturally incorporate cost- or performance-awareness. A further step is required, which is associated with enriching such models with the necessary artifacts to specify (i) performance requirements, (ii) business requirements, and (iii) workload behavioral characteristics.

RCo.3 SCARF: Systematic Cloud Application (Re)Distribution Framework

The concepts developed in this contribution serve as the pillar of the engineering framework proposed in this work. More specifically, we propose a theoretical framework, denoted as SCARF, which consists of a (i) life cycle for the cost- and performance-aware migration of cloud applications, and (ii) a method geared towards the cost- and performance-efficient design of cloud applications.

The main focus of SCARF is to *foster the proactive and collaborative (re)distribution of cloud applications in a cost- and performance-efficient manner*. SCARF focuses on analyzing the performance and cost trade-off in all steps of the application life-cycle, therefore ensuring proactive decision making. SCARF also influences on how applications are designed, by means of enabling collaboration not only between application and business architects, but also among organizations.

RCo.4 Discovery and Construction of Viable Cloud Application Distributions

The method proposed in RCo.3 requires automated decision making mechanisms to assist application architects to discover and select viable distributions for their applications. In short, a viable application distribution *consists of a technologically feasible distribution of its components, according to the application requirements and represented as a topology model*. This contribution proposes concepts and mechanisms to analyze application operational requirements, and to discover cloud offerings that can be part of an application's viable distribution.

These mechanisms consist of a two-step case- and similarity-based processing of the application's functional and non-functional aspects. Firstly, graph similarity enables the discovery of similar applications by analyzing structural similarity among application architectures.

Secondly, the non-functional discovery mechanisms use case-based reasoning and similarity analysis as its fundamental support. In particular, these leverage the usage of previous and similar knowledge, i.e., using previous solutions (viable application distributions) for new ocurring problems (new application distribution demands).

RCo.5 Utility-based Decision Making for Performance- and Cost-aware Optimal Cloud Application Distribution

Decision making support in this work is provided by means of quantitatively analyzing viable cloud application distributions. In particular, this work uses utility theory as the basis to evaluate the trade-off between cost and performance.

The utility framework enables the definition of custom utility functions, each dealing with the analysis of business and operational requirements. A pool of utility functions allows to reuse utility models among different types of applications and viable distributions.

RCo.6 SCARF-T: Tool Chain for the Performance- and Cost-efficient Design of Cloud Applications

The last contribution consists of reusing and/or developing, when deemed necessary, a tool chain supporting the theoretical framework developed in RCo.3. The tool chain is denoted as SCARF-T, and comprises a set of tools to support the complete performance- and cost-aware migration of applications to the cloud.

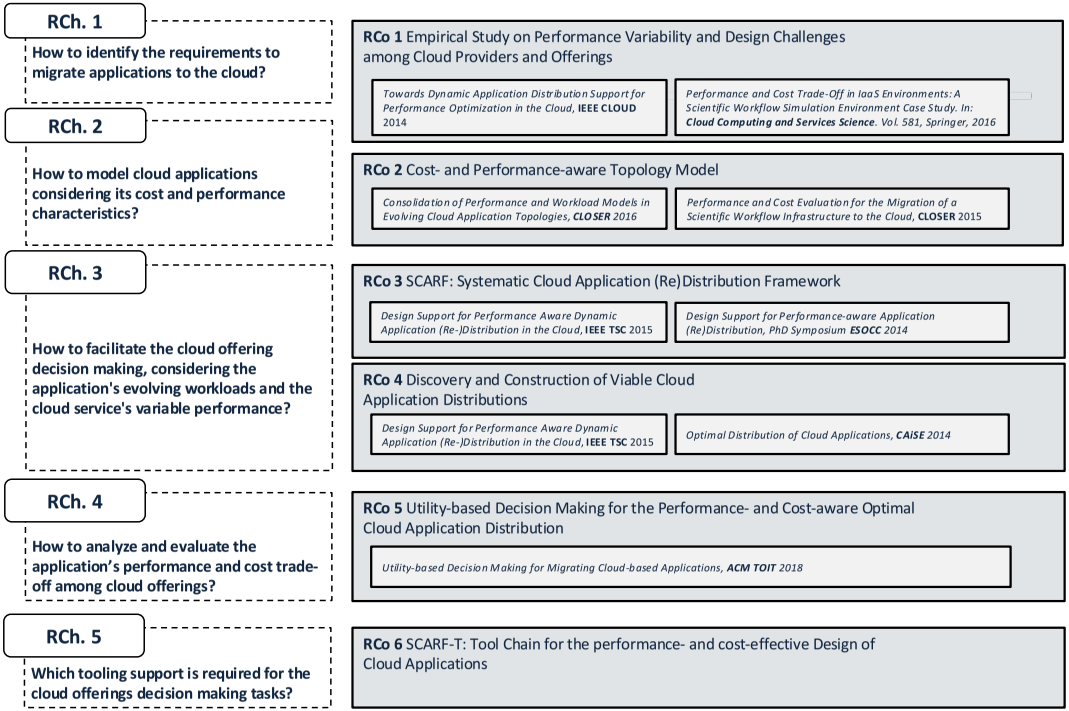


Figure 1.2: Main Publications Overview - Mapping to Challenges & Contributions

1.5 Structure of the Thesis

This thesis is organized as depicted in Fig. 1.3. Chapter 1 opens this work by introducing its motivation, scope, research challenges, and research contributions. Chapter 2 summarizes the necessary background and foundations of SCARF. It outlines methods and decision support frameworks for engineering and migrating cloud applications. Chapter 3 empirically evaluates performance variability among cloud providers and services. These serve as the basis for introducing SCARF in Chapter 4, a generic framework for systematically engineering cloud applications. SCARF consists of a life-cycle and method focusing on reducing decision making complexity when migrating applications to the cloud. The following chapters are structured referring to SCARF's method, SCARM.

Chapter 5 proposes an enhancement model for cloud application topologies, which focuses on characterizing performance and workload knowledge. Chapter 6 implements a technology-agnostic topology model for SCARF, which focuses on reusing and generating viable cloud application topologies. This model allows application architects to model application specific sub-topologies, while discovering and reusing non-application specific sub-topologies. The discovery and construction of cloud application topologies relies on a method built upon case-based reasoning and similarity analysis, which is introduced in Chapter 7. Decision making support is one of the main pillars in SCARF, as it assists business and application architects to select optimal cloud application topologies. Such a support is based on utility theory, and is formally developed in Chapter 8. This utility model focuses on analyzing the trade-off between cost and performance for a cloud application viable topology.

Chapter 9 presents the technical realization of SCARF, which comprises a set of integrated tools in a tool chain denoted as SCARF-T. It presents a generic architecture. and describes each tool, by means of discussing design principles, and adopted and developed technologies. The evaluation of SCARF is performed in Chapter 10. In particular, two cloud migration case studies are driven, using a Wiki and a Smart City application for this purpose. Chapter 11 provides a conclusion and outlook, which summarizes the contributions of this thesis and describes future research challenges to enhance SCARF and overcome its limitations.

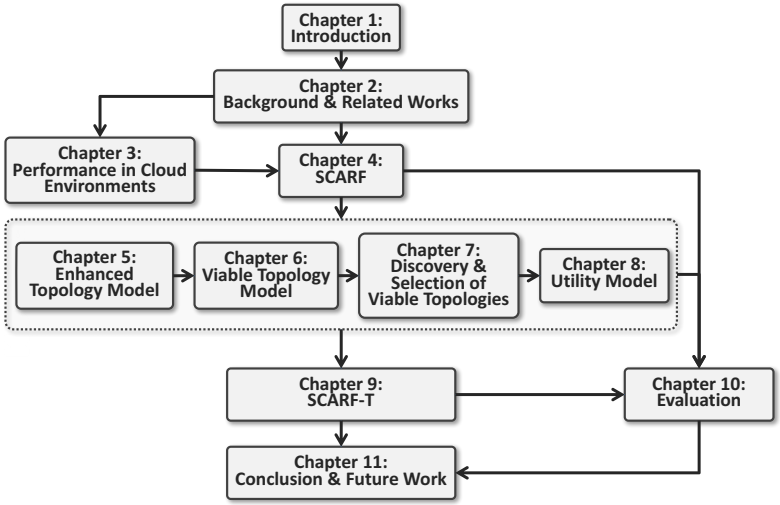


Figure 1.3: Structure of the Thesis

Chapter 2

Background & Related Works

2.1 Chapter Introduction

The migration of applications to the cloud has been for more than a decade a major topic in both the industry and research. This section opens with the underpinning concepts to consider when migrating applications to the cloud. In particular, we address the most relevant architectural principles and guidelines, together with current standards and approaches, to specify deployable cloud applications in Sec. 2.2. Section 2.2 enriches the discussion with an overview of the services landscape currently offered by major cloud providers.

Since decision making is a key aspect in this work, Sec. 2.3 addresses cost and performance analysis, and cloud service selection during

the design phase of cloud applications, including a review of projects focusing on building multi-cloud decision support frameworks. Performance analysis models and approaches are detailed in 2.4, and focus on how to model, evaluate, and capture the evolution of performance and workload of cloud applications.

As briefly introduced in the previous section, the trade-off analysis mechanism between cost and performance is built in this work using utility as the basis for decision making. Sec. 2.5 familiarizes with utility theory and its usage in computing systems. The chapter closes with Sec. 2.6, which introduces a conceptual background related to using Case-based Reasoning (CBR) as the basis for the discovery of viable application deployments.

2.2 Engineering Applications for the Cloud

The cloud computing paradigm has brought together an increase of design and operation complexity, demanding the modernization and enhancement of traditional IT engineering techniques [Var10]. The design of cloud applications strongly requires a solid understanding of how cloud environments work [Kav14]. In other words, migrating monolithic applications as a Virtual Machine (VM) does not exploit the benefits of the cloud, therefore making the migration a simple hosting solution rather than a scalable cloud-based solution.

Migrating applications to the cloud is a complex procedure, which consists of (i) cloud-enabling the application, (ii) delegating the management of resources to cloud providers, and (iii) observing and ensuring that the application requirements are fulfilled. According to Andrikopoulos et al., cloud-enabling applications is basically a matter of maximizing the profit from past software investments, rather

than completely abandoning previous design and development efforts [ABLS13]. Of course, this fact does not constrain the possibility of partially replacing application components with cloud-native implementations, i.e., implementations addressed specifically for the cloud.

The remainder of this section firstly focuses on discussing the architectural benefits and guidelines when designing distributed cloud application architectures, by means of fully exploiting the *aaS delivery model landscape. This section closes with a technical discussion of standards and approaches to realize cloud application architectures.

2.2.1 Design of Cloud Application Architectures

The shift in cloud computing towards the *aaS delivery model had a considerable impact when analyzing how to enable traditional applications for the cloud, as these are typically built with pre-conceived mindsets using traditional software engineering techniques [Var10]. Cloud environments expose different service delivery and deployment models. Although it is not our goal to detail them, as these are well-defined in [MG11], we review the fundamental cloud characteristics identified by the National Institute of Standards and Technology (NIST): (i) the *on-demand self-service* of computational resources, (ii) *broad network access* of capabilities, (iii) *resource pooling* using a multi-tenant model, (iv) *rapid elasticity* of resources, appearing to be unlimited, and a (v) *measured service* enabling monitoring, control, and reporting [MG11]. Such characteristics must be considered by application architects, who must comply to and be aware of the benefits and drawbacks when migrating existing or building new applications in the cloud.

Efforts in the last years have focused on establishing architectural principles, guidelines, and standards for migrating and/or building cloud applications. For instance, Andrikopoulos et al. focus on migrating applications to cloud environments, and identify four types of migrations [ABLS13]: *I. Replacement*, *II. Partial Migration*, *III. Migration of the whole Software Stack*, and *IV. Cloudifying the Application*. In a nutshell, the replacement type consists of substituting application components with cloud offerings, e.g. substituting a local database with a DBaaS offering [SAT+13]. A partial migration consists of moving some of the application functionality to the cloud, e.g. some of the business logic to AWS Lambda[1]. The migration of the complete software stack packaged in a VM is the most classic example. Cloudifying the application consists of completely migrating the application to the cloud, by means of realizing the application as a composition of cloud services.

On the other hand, building *Cloud Native Applications* – applications built for the cloud – are analyzed by Leymann et al. and Fehling et al. in [LFWW16] and [FLR+14], respectively. Fehling et al. identify the *IDEAL* properties for cloud native applications: ***I**solation of state*, ***D**istribution*, ***E**lasticity*, ***A**utomated Management*, and ***L**oose Coupling*. Ideally, a native cloud application must be designed following Service Oriented Architecture (SOA) principles of loose coupling and distribution by spanning their components among multiple cloud services. This distribution must also be automated in terms of provisioning, deployment, and monitoring, and it must enable scalability at all levels of the application software stack. One aspect that requires an independent analysis is the handling of the application's state. Cloud architectures rely on Basically Available, Soft State, and Eventu-

[1] AWS Lambda: `https://aws.amazon.com/lambda/details/`

ally Consistent (BASE) transactions. BASE transactions acknowledge that resources can fail and the data will eventually become consistent [Kav14]. This transactions-based model is a threat to traditional applications, as legacy applications rely on Atomicity, Consistency, Isolation, and Durability (ACID) transactions [Kav14]. This challenge must be evaluated and the application must be adapted prior to the migration, by means of analyzing and coping with trade-offs defined in the CAP theorem [Bre12].

Cloud architectures best practices and guidelines have mostly appeared in the industry domain, such as in Amazon Web Services (AWS), as a mean of incentive and transparency to architects willing to migrate their applications to public clouds. When migrating applications to the cloud, application architects must analyze and adapt their applications to partially or completely comply with cloud application architectural principles. Among the cloud architectural guidelines proposed in the industry, such as by AWS [Var10], or in research works by Fehling et al. [FLR+14] and Reese [Ree09], the mostly required property when designing cloud applications is its ability to cope with system failures. Interpreted in another way, Varia recommends to be a pessimist when designing architectures in the cloud, assuming that things will fail [Var10]. Highly failure resistant cloud applications can be realized by adopting mechanisms that:

- automate the management of resources, such as dynamic scaling techniques [VRB11] or spinning up a VM from a previous snapshot stored as a Virtual Machine Image (VMI), e.g., as an Amazon Machine Image (AMI)[2], and by

[2] AMI:
http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

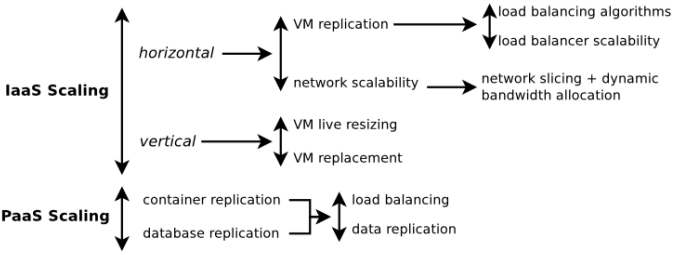


Figure 2.1: Scalable Mechanisms in Cloud Environments [VRB11]

- decoupling application components and minimizing the dependencies among them [FLR+14].

Focusing on the former, cloud applications must be elastic by nature. Scalability techniques for cloud applications have been widely studied in the literature, mostly for satisfying optimal QoS levels in [VRB11] while minimizing the so-called *elastic costs* – costs derived from scaling up an application [SSJL12]. Figure 2.1 depicts scalable mechanisms in IaaS and PaaS cloud environments. IaaS environments offer horizontal scalability of virtual hardware, by means of replicating VMs or scaling network properties. Vertical scalability consists of allocating more hardware for a VM, by means of resizing the hardware properties or replacing a running VM with a more powerful one. Scalability in PaaS environments is based on replicating containers and databases hosting the application and its data. It is crucial to identify which scaling approach suits better the application requirements, as horizontally scaling is typically cheaper as vertical scaling, due to the instant availability of replicated resources [EPM13].

Focusing on decoupling application components following SOA

principles, microservices architectures have been a hype in the last years. Microservices architectures consist of developing a single application as a suite of small services, which interact towards building business capabilities and are independently deployable in a fully automated manner [FL14; LFWW16]. Such architectural style perfectly fits with DevOps principles, fostering the agile and rapid development, adaptation, and deployment of software architectures [HM11].

Further cloud application architectural concerns are related to maximizing parallelization wherever possible, as well as keeping dynamic data closer to the business logic while static data closer to the end-user, e.g., using elastic caching mechanisms. Security is not covered in our work, but is a fundamental aspect when migrating applications to the cloud. In particular, Varia recommends to implement security at every layer of the application architecture, due to its distributed nature [Var10].

2.2.2 *aaS (*Everything*-as-a-Service) Delivery Models

Cloud computing is built upon the premise of *converting capital expenses into operating expenses* [AFG+10]. The main economic benefit of cloud computing is the possibility to deliver hardware following the pay-as-you-go model – purchasing cloud resources non-uniformly in time. Cloud providers have seen this as an opportunity to go beyond the original SaaS, PaaS, and IaaS service delivery models defined by the NIST [MG11]. Schaffer identified in [Sch09] 44 types of services from different providers, which go from delivering databases in a DBaaS delivery model, to delivering more fined grade services, such as Security-as-a-Service (SECaaS), Identity-as-a-Service (IDaaS) or Desktop-as-a-Service (DaaS), among others.

Cloud providers are nowadays gearing their efforts towards building new and expanding existing service models that are suitable to specific kind of applications. For instance, AWS Elastic Compute Cloud (EC2)[3] provides different types of VM instances, which are optimized for different types of applications, such as compute, memory, or storage intensive. The same VM specialization is observed in Microsoft Azure VM Service[4]. Focusing on workflow applications, WSO2 Stratos Service[5] is optimized for the management and execution of workflows, and the IBM Business Process Manager[6] offers the necessary tools and abstraction levels for developing, deploying and monitoring workflows in the Cloud. AWS offers AWS Lambda[7], which allows a smoothly deployment of code which is billed only when executed.

The existence of multitude of cloud services, which in practice use different technologies, protocols, and formats, has become a major challenge for cloud architects [HK10]. Apart from that, cloud providers are mostly vague or even avoid to describe the internals of their services, making interoperability and decision making a complex task. Even the usage of the term *aaS in the literature lacks of a unified view, according to Duan et al. [DFZ+15]. For purposes of organizing and categorizing existing cloud services and providers, there exist a considerable amount of research works that focus on creating taxonomy representations of the cloud landscape. For instance, Hoefer et al. propose a taxonomy that allows quick comparisons, by giving the user a set of choices that are related to application-related character-

[3] AWS EC2 Instance Types: https://aws.amazon.com/ec2/instance-types

[4] Microsoft Azure VM: https://azure.microsoft.com/en-us/services/virtual-machines

[5] WSO2 Stratos Service: http://wso2.com/cloud/stratoslive/

[6] IBM BP Manager: http://www-03.ibm.com/software/products/en/business-process-manager-cloud

[7] AWS Lambda: https://aws.amazon.com/lambda

istics [HK10; HK11]. Rimal et al. categorize different cloud services based on application architectural features, such as computing, load balancing, interoperability, etc. [RCL09]. Focusing exclusively on the application data layer, Strauch et al. categorize different deployment possibilities for migrating databases to cloud environments [SKLU11]. Kachele et al. also limit their focus, by means of classifying storage and networking cloud services [KSHD13].

It is clear that cloud computing is shifting towards enhancing the end-user experience, by means of reshaping the cloud computing landscape and creating the next wave of dynamic services for supporting new agile lines of business [Rob+08].

2.2.3 Specification of Cloud Application Architectures

The underlying complexity of cloud computing environments has forced application architects to move beyond traditional modeling and specification techniques and languages. Cloud computing requires and allows application architects to define and view, respectively, the complete application landscape, by means of describing the components, services, hardware, and the interactions among them. Moreover, the possibility to build a multi-cloud environment demands such application definitions to comply with multiple clouds, therefore fostering the portability of applications.

The concept of topologies has been widely used, for instance, in distributed systems and computer networks [TV07]. Moving towards enterprise applications, Binz et al. propose the usage of an Enterprise Topology Graph (ETG), which is defined as a *graph-based model for enterprise topologies capturing all entities of enterprise IT and their logical, functional, and physical relationships* [BFL+12]. In other words,

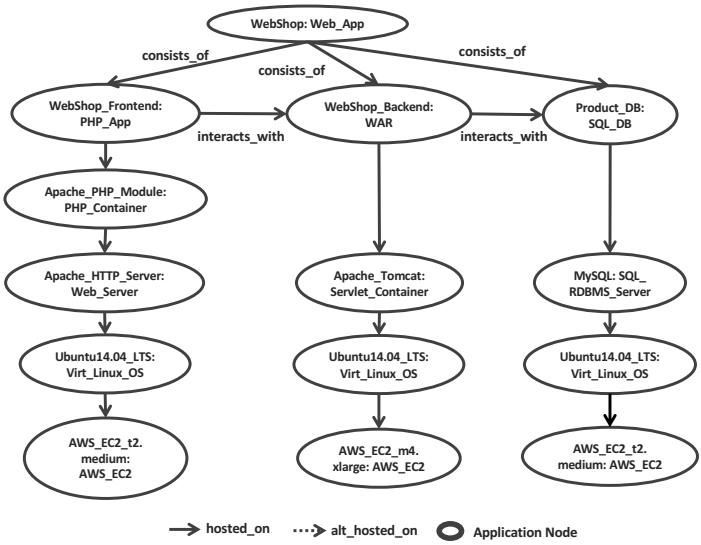


Figure 2.2: Topology Example - Three-Tiered Web Shop Application

an ETG can be basically used to depict a big picture of a system consisting of multiple interacting applications. Cloud applications require a further step towards its specification, as cloud environments offer the possibility to automatically compose and manage cloud services and applications potentially hosted in different clouds [PvdH11]. For instance, Fig. 2.2 depicts a graph-based topology model for a sample three-tiered Web shop application. The topology is essentially a directed acyclic graph, where nodes define the application components, and edges define the relationships between them. The usage of topologies to represent the architecture of cloud applications is used as the basis throughout this work.

Cloud application topologies can be expressed in different ways,

such as Domain Specific Language (DSL), visual templates, or graphical models, which are typically specific for each tool or provider. For instance, the Flexiant Cloud Orchestrator (FCO)[8] uses the so-called infrastructure blueprints, which are templates describing application environments. AWS CloudFormation[9] is the DSL for defining AWS resources, and is graphically supported by the CloudFormation Designer. OpenStack Heat[10] application templates are evolving towards providing compatibility with AWS CloudFormation templates. Ansible[11] is geared towards the declarative configuration management of multi-cloud applications. As already depicted, there exists a wide amount of DSL, or Cloud Modeling Languages (CML) as defined in [BBF+18]. Bergmayr et al. introduce in [BBF+18] a framework capturing and categorizing characteristics of various CMLs.

Approaches for modeling portable cloud applications have arised in both research and industry domains. The Topology and Orchestration Specification for Cloud Applications (TOSCA) standard is probably the most extended standard for enabling portability of cloud applications, and is used in this work as the central technology. TOSCA is based on the definition of component-related types and templates [BBKL14a]. In particular, application topologies are defined as service templates, which comprise the application topology template, node and relationship types, and the application management plans. Node and relationship templates define the application components and their corresponding interactions, respectively. TOSCA fosters reusability of components by means of using node and relationship types, which depict the skeleton of the component properties, e.g., user credentials for

[8] FCO: http://www.flexiant.com/flexiant-cloud-orchestrator/
[9] AWS CloudFormation: https://aws.amazon.com/cloudformation/
[10] OpenStack Heat: https://wiki.openstack.org/wiki/Heat
[11] Ansible: https://www.ansible.com

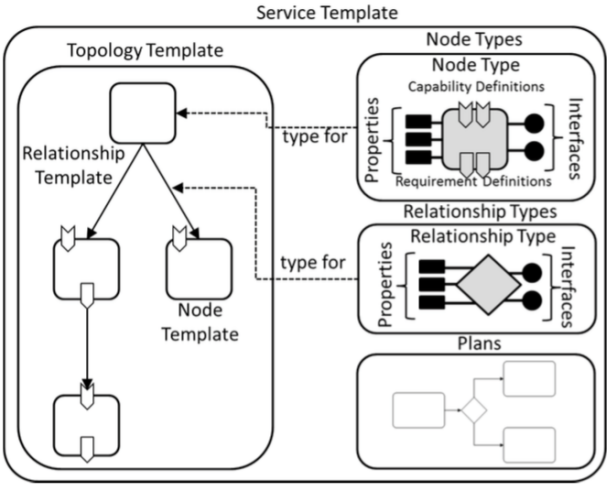


Figure 2.3: TOSCA Standard Model [BBKL14a]

a cloud service, and define the management interfaces. Management plans are defined as workflows, and depict the management operations for the whole application stack, such as VM instantiation, package installation, artifact deployment, etc. TOSCA is widely accepted and technologically supported by the OpenTOSCA environment[12], as well as in industry tools as Cloudify[13] and EU projects, such as SeaClouds[14]. The specification of application non-functional aspects in TOSCA is supported by the Policy4TOSCA specification [WWB+13], which is also adopted in this work.

A similar approach to TOSCA are *Cloud Blueprints* [PvdH11]. Cloud blueprints allow the specification, configuration, and deployment of

[12]OpenTOSCA: `http://www.opentosca.org`
[13]Cloudify: `http://getcloudify.org`
[14]SeaClouds: `http://www.seaclouds-project.eu`

applications on virtual machines. Blueprints are meant to facilitate cloud service selection, customization and composition of service-based applications. Blueprints allow developers to define their requirements in terms of functional capabilities, QoS characteristics, as well as the deployment and provisioning of resources as target blueprints. Cloud Application Management for Platforms (CAMP)[15] is probably the major TOSCA competitor, as it defines a self-management API for PaaS environments. Apache Brooklyn[16] is the tooling support for CAMP, where developers can define platform-, plan-, and assembly-related resources within a Platform Deployment Package (PDP) archive containing the plan file together with the application content. The Composite Application Framework (CAFE) environment provides the means to describe composite service-oriented applications and deploy them automatically across cloud providers [MUL09]. The fundamental difference towards CAFE is its variability model containing variability points for parameterizing and optimizing the application deployment model, as discussed and compared in Sec. 2.3.2.

Using Model-Driven Engineering (MDE) as the basis, CLOUDML [BM12; FRC+13] is a DSL for topology modeling. Its framework consists of a runtime environment for enacting, provisioning, and adapting these models. Topology models define the nodes of the cloud infrastructure, as well as the software artifacts that are deployed on these nodes. Both nodes and artifacts are typed, which allows for reasoning on the topology models. CLOUDML is used in the MODA-CLOUDS EU project for modeling and managing multi-cloud applications [ADM+12]. MDE is also used in Models@Runtime to enable dynamic adaptation of resources defined as CLOUDML mod-

[15] CAMP v1.1: `http://docs.oasis-open.org/camp/camp-spec/v1.1/`
[16] Apache Brooklyn: `https://brooklyn.apache.org`

els [CdFD+14].

From the previous discussion, It can be observed that there exist multiple approaches in the form of proprietary DSL or standards, therefore creating a spacious spectrum for modeling and specifying cloud applications. The last approach discussed in this section gears towards the generalization of such approaches. More specifically, the Generalized Topology Language (GENTL) consolidates similarities among TOSCA and blueprints, and creates a language for developing a generic application topology that can then be mapped or transformed to target(s) technological environment(s) [ARSL14].

2.3 Migration of Applications to the Cloud

The materialization of cloud providers moving towards delivering *aaS opened opportunities for organizations to migrate their legacy applications typically hosted in on-premise environments [Sch09]. However, the migration of applications to the cloud is not a trivial task, as the intrinsic characteristics of cloud environments demand considerations and tasks that go beyond traditional application design and maintenance techniques. For instance, Kousiouris et al. identify main challenges when porting legacy applications to the cloud [KKMV12]. Focusing on Small-to-Medium Enterprises (SMEs), da Silva et al. focus on identifying challenges and tools to support small organizations with limited cloud knowledge and resources to migrate their applications to the cloud [dSdFCM18]. Among others, the most crucial cloud migration challenges relate to (i) the lack of knowledge on infrastructure environments and cloud offerings, (ii) the impact of multi-tenancy when competing for shared computational resources, and (iii) the accommodation of different application

types with specific cloud offerings.

In the last years, industry and research works focused on migrating the whole application stack to virtual machines, either to on-premise or off-premise IaaS enviroments. Some examples are the works of LLoyd [LPD+11], Khajeh [KGSS12], and Menzel [MR12]. They mainly focus on assisting in the selection, configuration, and evaluation of provisioned VMs. However, simply placing an application in a VM does not automatically imply the full exploitation of capabilities offered by cloud environments [LFWW16].

The existence of different cloud migration possibilities and a wide spectrum of cloud offerings in the *aaS model broadens even more the challenges that application architects must face, building a multi-dimensional problem. Decision support tools and mechanisms are a current trend, with the purpose of independently or jointly targeting the each problem dimension [JAP13]. Although these are in a very formative stage [KSBT11], there are several projects in this domain. In the remainder of this section we extend on such a challenge, analyze the impact of cost and performance, and assemble existing decision making methods and tools for migrating applications to the cloud.

2.3.1 Multi-Dimensional Cloud Service Analysis

As previously discussed, the distribution of application components spanning cloud environments incorporates fundamental architectural challenges, which are directly related to the business and operational objectives of organizations. In other words, application architects face a multi-dimensional problem, where business objectives and operational requirements frame the different parameters of such a problem.

Andrikopoulos et al. identify four main decision points when migrating applications to the cloud [ASL13; ADKL14]. These are directly correlated with analytical tasks analogous to deciding among cloud providers and services to host the application. Fig. 2.4 outlines and extends the most relevant concerns when selecting cloud services, and highlights the ones addressed in the remainder of this section. When migrating applications to the cloud, analytical tasks related to such concerns deal with evaluating the trade-off among them. For instance, an organization may focus only on analyzing security and compliance of a cloud service, while others might be more interested in tackling cost and interoperability. The main foreseen challenge when building decision support systems are related to enabling the automated execution of these tasks, which can then produce an aggregated and quantified result representing the trade-off between different concerns. This result can be used to compare the expected outcome when utilizing different cloud services.

Although there exists a plethora of related works in cloud service selection, the majority of them typically focus on one axis of the problem, e.g. independently analyzing security, cost, or performance. There exist works that identify and discuss the influence and affections between the tasks, such as [ADKL14] and [RCL09], but lack automated mechanisms and tools capable of quantifying the trade-off between them. The simultaneous analysis of concerns is addressed in the areas of cloud brokerage, e.g. using multi-objective optimization through genetic algorithms [AV16], or using enriched feature diagrams to represent the application architecture [QRD16]. The MADCAT methodology proposes the iterative refinement of cloud application architectures for changing requirements among stakeholders [INS+14]. However, these approaches are not capable of

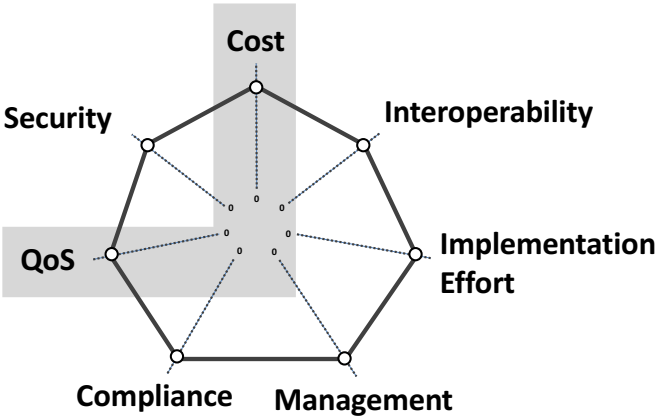


Figure 2.4: Multi-Dimensional Analysis - Cloud Service Selection

representing during the complete application life cycle the trade-off between concerns when analyzing different cloud services. Essentially, these works either focus on one specific concern, or support optimization during application design or execution phases. This work simplifies the decision making process by exploiting past experiences and by facilitating a utility-based ranking mechanism that exploits the usage of performance and cost knowledge in the whole application life-cycle (see Ch. 8).

Laliberte situates performance concerns in the third position in a survey of major organizations, after security and data protection [Lal13]. Laliberte also highlights the need for proactively taking migration decisions by collaboratively sharing performance experiences when using cloud services, rather than waiting for performance issues to arise [Lal13]. The fourth position of such a survey positions budget

constraints. Most of the works in the domain of decision making systems for cloud service selection target pricing analysis as their decision making pillar. There exist further migration concerns that are covered independently in the literature, such as consistency [Aba12]. In short, Laliberte's highlight is aligned with the scope of this work, as we focus on solving the problem of proactively taking migration decisions focusing on cost and performance by analyzing experiences among applications. The remainder of this section presents fundamentals and related works in the domain of cost- and performance-aware selection of cloud services.

2.3.1.1 Cost-aware Selection of Cloud Services

Cost optimization tasks for migrating applications to the cloud have encompassed in the last years a major part in the research agenda. Focusing on the application migration costs, Andrikopoulos et al. decompose these into (i) pricing models of service providers, and (ii) software licensing and infrastructure costs derived from elastic mechanisms [ABLS13]. Cloud providers intrinsically include software licensing costs, e.g., for the operating system, and offer distinct pricing models, which are summarized in [ABLS13] as follows:

1. Per-use: enables the access to resources on an on-demand basis and without any upfront payments. Prices typically vary between provider offerings.
2. Subscription: consists of reserving in advance computational resources by means of signing a contract and making an upfront investment, which is tied to the type of contract and provider.
3. Prepaid per-use: similar to the pre-paid model used in mobile

network operators. The billing is made on a pre-paid account, and services are blocked when the credit is consumed.

4. Subscription + per-use: combines the reservation of resources of the subscription model and enables the on-demand provisioning of additional resources on pay-per-use basis. There exist providers, such as Microsoft Azure, that offer a discount in their subscription contracts when allocating additional resources which are not included in a subscription.

The existence of different pricing models and units among cloud providers and services make cost optimization and prediction tasks complex. Cloud providers tried to facilitate such tasks by means of offering calculation services for their offerings, such as the AWS Simple Monthly Calculator[17], Microsoft Azure Calculator[18], or the Google Compute Platform (GCP) Calculator[19]. However, providers limit their calculation services to their own offerings, rather than spanning their calculation to multiple cloud providers. Such a limitation is addressed in calculation services offered by the Cloud Calculator[20], Cloud Norado[21], or in research works [XA13; ARML14; ARSL14].

Existing cost optimization mechanisms are directly related to estimating costs during the design phase of applications, by means of analyzing the application's workload and performance constraints. More specifically, these focus on estimating the resource costs for different application workloads [MMZV12; MMV13; FACD13], or on es-

[17] AWS Simple Calculator: https://calculator.s3.amazonaws.com/index.html

[18] Microsoft Azure Calculator: https://azure.microsoft.com/en-us/pricing/calculator/

[19] GCP Calculator: https://cloud.google.com/products/calculator/

[20] The Cloud Calculator: http://thecloudcalculator.com

[21] CloudDorado: https://www.cloudorado.com

timating the costs for each application component and type [FPLH11; CCB+15]. Approaches like [TD10] also enhance cost calculation with monitoring capabilities to control overall infrastructure costs. However, resource estimations are mainly based on estimating IaaS resources, and using in the majority of the cases simulation techniques. More advanced approaches are geared towards maximizing the profit of applications. In particular, Tsakalozos et al. [TKS+11] focus on calculating the optimal number of VMs by means of converging towards a MC (Marginal Cost) = MR (Marginal Revenue) equilibrium in IaaS environments. Wei et al. use vector arithmetic to model the objective balancing of VMs [CQWH12]. Focusing on the composition of cloud services, Ye et al. focus on using economic and Bayesian Network (BN) models to compose cloud services [YBZ14; YBZ12]. However, the economic model is tightly coupled with the type of applications, forcing architects to develop independent BN models for their applications. Moreover, approaches like [TKS+11] focus on analyzing if cost variation is similar or equal to revenue variation, rather than using a mechanism that aggregates multiple non-functional aspects into one final analytical value.

Cost efficient analysis and selection of cloud services is a very mature topic in the research domain. However, we outline the necessity for (i) providing resources cost calculations in multi-cloud scenarios, and (ii) facilitating more generic and simplified techniques and tooling support for estimating and evaluating infrastructure costs focusing more on application workloads, without requiring full knowledge of the underlying infrastructure model.

2.3.1.2 QoS-aware Selection & Performance Prediction of Cloud Services

QoS levels offered by cloud services play a fundamental role when deciding among cloud providers and offerings, as these depict levels of performance, reliability, and availability offered by a service or infrastructure hosting an application [ACC+14]. Cloud providers pursue to optimize the trade-off between QoS levels and operational costs, seeking for the delivery of acceptable QoS levels described in their Service Level Agreement (SLA) [WB12]. However, cloud environments offer a high performance heterogeneity and continuously changing scheduling mechanisms, therefore challenging the planning, analysis, and monitoring of QoS levels [ACC+14].

The analysis of QoS is directly related to specifying and evaluating the fulfillment of application demands in cloud environments. QoS metrics are used to specify performance and availability requirements, which can subsequently be evaluated during the execution of applications [SCD+97]. However, these kind of evaluations entail a reactive response to QoS variations, e.g., horizontally scaling the application when resources are over consumed. There has been a considerable effort in research and industry towards proactively estimating QoS parameters prior to the application production's phase, e.g., during its design phase. For instance, Kousiouris et al. denote the necessity to identify workload and QoS patterns for application components and cloud services, which can be fostered to create categories of services fulfilling common requirements and to facilitate the selection of cloud providers and services [KKMV12].

The modeling and prediction of QoS is a step further in performance-aware engineering of cloud applications. Since performance engi-

neering deals with modeling performance and workload aspects of applications, predicting QoS levels in the cloud requires evaluating the application's performance hosted in such shared virtualized environments. During decision making tasks for migrating applications to the cloud, simulation techniques can be applied to predict the amount and behavior of infrastructure resources. Simulation approaches have been proposed in the literature, mostly focusing on IaaS environments [LTRK10; NVC+12]. However, such approaches do not provide the knowledge or means to discover potential services to migrate and host applications, as well as to evaluate them. Furthermore, existing cloud simulators require the definition of an infrastructure model and offer a varying level of accuracy [ASS18]. More sophisticated approaches are used in combination with discovery and optimization techniques, such as *Genetic Algorithms*, *Utility Theory* or *Analytic Hierarchy Process (AHP)*. For instance, Ye et al. and Lee. et al. use genetic algorithms to produce different combinations for composing cloud services, which are then evaluated using simulation techniques and objective functions [YZB11; LTRK10]. Kritikos et al. use utility and application deployment knowledge to find optimal IaaS configurations [KMP16]. Unuvar et al., on the other hand, use utility theory to find the optimal cloud availability zone for deploying an application, focusing on the end-user satisfaction [UDS+14]. Using Grey TOPSIS and AHP in combination, SELCLOUD aims at providing support for cloud brokers for ranking and selecting IaaS cloud services based on their QoS [JGFB18]. In spite of such enhancements, the aforementioned approaches can be taken a step further, by means of exploiting principles demanded in [KKMV12] and considering the application's end-user satisfaction. In particular, the collaborative sharing and processing of performance and cost knowledge of previous cloud mi-

grations, as well as the end-user satisfaction, can considerably enhance decision making tasks when designing and migrating applications to the cloud.

2.3.2 Cloud Services Decision Support Approaches

Previous sections denoted the challenging tasks for application architects related to deciding among different cloud providers and services to host their applications. Towards reducing such challenges, different approaches of Decision Support System (DSS) – as defined in [Pow08] – have been implemented. There are also a considerable number of EU projects that focus on the (i) assisted modeling, (ii) portable deployment, and (iii) monitoring of multi-cloud applications.

Table 2.1 summarizes existing migration DSS approaches that focus on selecting cloud providers and offerings to host applications in a distributed manner. Moreover, Table 2.1 classifies these approaches w.r.t. the following properties:

1. Deployment Extension: which deployment models are supported, i.e., (i) single cloud or (ii) multi-cloud deployments.
2. Migration Type: as defined in [ABLS13] and discussed in Sec. 2.2, i.e., I. Replacement, II. Partial, III. Whole Stack, and IV. Cloudification.
3. Offerings Discovery: supported mechanism to manual and (semi-)automatically discover cloud providers and offerings.
4. Offerings Analysis & Evaluation: supported (automated) mechanism to evaluate offerings w.r.t. application workload, offerings cost, and performance.

5. Tooling Support: existing approach entails tooling support for *Modeling* of deployment models, *Discovery* of cloud offerings, and *Provisioning & Monitoring*.

Based on the previous classification, Table 2.1 further provides a quantitative analysis on how many aspects are covered by each of the approaches.

Focusing on the different approaches, the CloudDSF migration decision support framework provides a conceptual overview and the corresponding tooling support for representing tasks, actions, and the relationships among them, that application architects must cope with [ADKL14; ASL13]. Moving towards migration decision support systems that operate on the application's architecture, MOCCA focuses on facilitating cloud migration, by means of providing a method and tool chain support to distribute applications in multi-cloud environments [LFM+11]. MOCCA's decision making support is based on manual architecture model partitioning and requirements evaluation using optimization algorithms, such as simulated annealing. MOCCA strongly focuses on the portable deployment of applications, as the optimization does not consider, for example, the application workload. Jamshidi et al. propose a pattern-based cloud application method using cloud architecture migration patterns as the basis [JPM16]. The composition of migration patterns derive in the creation of domain specific migration plans.

The usage of simulation in cloud migration DSS is well established. CDOSim, for instance, supports optimization by means of simulating the performance and cost of a Cloud Deployment Option (CDO) [FFH12]. CDOSim itself only supports single cloud deployment scenarios and is probably the most used simulator in several projects. Its integration and extension in the CLOUDML environment

	Deployment Extension			Migration Type				Offerings Discovery		Offerings Analysis & Evaluation					Tooling Support			
	Single Cloud	*Multi-Cloud*	*I. Replacement*	*II. Partial*	*III. Whole Stack*	*IV. Cloudification*	*Manual Specification*	*Semi-Automated Matching*	*Automated*	*Workload*	*Cost*	*Performance*	*Ranking*	*Modeling*	*Discovery*	*Provisioning & Monitoring*	*# Supported Aspects*
MOCCA [LFM+11]	✓	✓	✓	✓	✓	✓	✓				✓	✓		✓		✓	11
CDOSim [CRB+11]	✓		✓	✓	✓	✓	✓			✓	✓	✓		✓	✓		11
iCanCloud [NVC+12]	✓			✓	✓		✓			✓	✓	✓		✓			8
TOSCA-MART [SBB+16]	✓	✓	✓	✓	✓	✓	✓	✓						✓	✓	✓	11
CloudMIG [FH11]	✓			✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓		12
CloudDSF [ADKL14]	✓	✓	✓	✓	✓	✓	✓										12
CloudGenius [MR12]	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓		✓	✓		12
CloudWard Bound [HSS+10]	✓			✓	✓	✓	✓				✓	✓					7
Cloud Adoption Toolkit: [KGSS12]	✓	✓	✓	✓	✓	✓	✓				✓	✓	✓				10
(MC2) 2 [MST13]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	16
SeaClouds[a] [BFI+15]	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	14
PaaSage[b] [Kol13]	✓	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓	14
CLOUDML[c] [ADM+12; CdFD+14]	✓	✓	✓	✓	✓	✓	✓							✓		✓	9
Cloud4SOA[d] [ZDB+13]	✓	✓	✓	✓	✓	✓	✓		✓			✓	✓	✓		✓	12
Cactos[e] [ÖGW+14; LKHE15]	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓		✓	✓	✓	14
mOSAIC[f] [MAD+11; PDV+13]	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓		✓	✓	✓	13

Table 2.1: Migration Support Methods & Tools Comparison

[a] SeaClouds EU Project: http://www.seaclouds-project.eu
[b] PaaSage EU Project: http://www.paasage.eu
[c] CLOUDML Project: http://cloudml.org/
[d] Cloud4SOA: http://www.cloudwatchhub.eu/cloud4soa
[e] Cactos EU Project: http://cactos-cloud.eu
[f] mOSAIC EU Project: http://www.mosaic-cloud.eu

allows the simulation of multi-cloud scenarios [BM12], which is realized within the EU Project MODAClouds [ADM+12]. MODAClouds enables simulation and decision support mechanisms to score cloud providers w.r.t. application cost and risk requirements. The CloudMIG approach adopts the CDOSim as part of its environment, and develops a framework capable of semi-automatically classifying cloud deployment alternatives [FH11]. In particular, CloudMIG is capable of classifying CDOs into suitability groups, such as *cloud incompatible*, *cloud ready*, etc.

Further simulation-based approaches, such as iCanCloud, simulate only performance of IaaS environments and are limited to single cloud deployments, as these basically emulate the behavior of hypervisors when provisioning and executing VMs [NVC+12]. CloudGenius goes a step further, by automating the decision making with simulation analysis [MR12]. In particular, it uses multi-criteria decision making based on AHP to discover and evaluate combinations of VM images. However, the authors denote that examining application workloads during the AHP process in CloudGenius is not supported and could considerably improve the decision making process. The DSS developed in the EU mOSAIC project is also constrained to IaaS environments, and it is built by capturing cloud knowledge using ontologies, and by building a cloud brokerage system capable of automatically and transparently negotiating cloud services [PDV+13].

There also exists a number of approaches that do not use simulation as the basis. TOSCA-MART, for instance, fosters reusability of cloud application topologies by discovering and matching cloud offerings [SBB+16]. TOSCA-MART is a method for completing and reusing TOSCA cloud application topologies in a semi-automated manner, by means of matching requirements and capabilities. The CloudWard

Bound is a cost estimation framework focusing on hybrid cloud deployments [HSS+10]. A similar environment is the one provided in the Cloud Adoption Toolkit, which examines application constraints and estimates the cost for using multiple services [KGSS12]. Focusing on private deployment scenarios and providing ranking capabilities based on multi-criteria-based decision making, (MC2) 2 generates a ranking over different deployment alternatives. w.r.t. diverse infrastructure and operational costs, such as investment, maintenance, training, integration, regulation, etc. [MST13].

In the last five years, there has been a strong interest for multi-cloud decision making support in the research community. This resulted in launching several EU projects, such as SeaClouds[22], PaaSage[23], Cloud4SOA[24], and Cactos[25]. Although they all target multi-cloud scenarios, they focus on different aspects of applications. SeaClouds provides seamless adaptive multi-cloud management of applications, with a strong focus on design matchmaking and deployment optimization of cost and performance, as supported by the works in the scope of TOSCA-MART [SBB+16; BS16]. However, SeaClouds focuses on generating multiple deployment alternatives, without any kind of ordering- or ranking-based navigation support.

In the domain of optimizing deployment of PaaS applications, PaaSage allows the model-based development, configuration, and optimization of application deployment alternatives [Kol13]. PaaSage basically generates a policy-based ranked list of matching services specified in its DSL CAMEL. Moreover, PaaSage also focuses on capturing and exploiting history and evolution of distributed application

[22] SeaClouds EU Project: `http://www.seaclouds-project.eu`
[23] PaaSage EU Project: `http://www.paasage.eu`
[24] Cloud4SOA: `http://www.cloudwatchhub.eu/cloud4soa`
[25] Cactos EU Project: `http://cactos-cloud.eu`

deployments in order to classify cloud resources [PM13]. Cloud4SOA supports the discovery and matching of PaaS offerings suitable to developers' needs. Its matchmaking algorithm computes the degree of relation between the semantic descriptions of cloud offerings and application profiles, and generates a ranked list of PaaS-based deployment alternatives [ZDB+13]. However, both PaaSage and Cloud4SOA require cloud providers to elaborate and publish semantic descriptions of their services, using – in the case of PaaSage – its DSLs.

Finally, Cactos is probably the closest approach to this work. Cactos analyses application behavior and performance, and uses mathematical models to determine best fitting resources. Cactos is shipped with a simulation environment, CactoSim, capable of simulating the performance of diverse application workloads [ÖGW+14; LKHE15]. CactoSim is used as the basis to generate performance and cost predictions. However, fully basing the decision support on simulation mechanisms implies acquiring and implementing the knowledge and models of an infrastructure, which is essentially very complex. In this work we leverage the usage of collaborative knowledge among similar applications and their different deployment experiences.

2.4 Performance Engineering of Cloud Applications

The deployment of applications in the cloud is not a direct guarantee of high performance, scalability, and optimized QoS [GB16]. Software engineers and architects typically face crucial challenges when designing, deploying, and maintaining performance-efficient cloud applications. Performance constraints and objectives must be built into the application life cycle as early as possible, in order to ensure an application performance within expected parameters. For instance,

Molyneaux highlights the importance of not only considering the performance of independent application components, as these may independently perform very well, but also to consider the performance impact when these interact with other internal or third-party components [Mol09].

Optimizing performance of applications can be seen as a feedback cycle comprising the following main tasks. Software architectures must incorporate performance goals and constraints, which can allow for the optimization of architectural models. According to Aleti et al., performance and cost concerns sum up a 44% and 39%, respectively, in the software design and development related literature [ABG+13a]. An efficient implementation and testing prior to the production phase can heavily impact on achieving an optimal performance. More specifically, there are two main approaches for describing and improving performance: (i) an *early-cycle predictive model approach*, and a *late-cycle measurement-based approach*. Woodside emphasizes that the future trend focuses on a necessity for converging them during the entire development cycle [WFP07]. During the (pre-)production phase of applications, the allocation and configuration of resources play a fundamental role. A well designed capacity planning process consists of identifying business models and measurable goals, identifying the system architecture, as well as developing models for analyzing, characterizing, and predicting the application workload and the behavior of the underlying resources [AM02]. The main goal when planning capacity is to ensure that a system meets desired performance goals without excessive over-provisioning [Fei15].

The remainder of this section provides an overview on fundamentals and existing approaches focusing on design and development of performance-aware cloud applications. In particular, this section

breaks down the following three key aspects: *modeling performance in cloud applications*, *modeling and specifying workload behavioral characteristics*, and *analyzing and evaluating application's performance and workload evolution*.

2.4.1 Performance Modeling

A fundamental task when engineering well-performing applications consists of creating or deriving performance models that are part of an application profile. Molyneaux considers a well-performing application when functionalities exposed to end users can be utilized without perceived delays or irritations [Mol09]. An elemental challenge when engineering well-performing applications is to proactively detect performance issues prior to the application's production phase. Therefore, the main challenge in performance modeling is to create accurate models that comply with application business and operational requirements that can help to predict its performance.

The definition, usage, and analysis of key indicators in performance models have been a primary line of research in software engineering. Considering the design and operations of cloud applications, two critical landscapes can be clearly identified. On the one hand, application architects are responsible for developing, implementing, and using winning Key Performance Indicators (KPIs) [Par15; WT06]. KPIs serve to establish, analyze, and evaluate the performance and success of an organization. Winning KPIs are introduced by Parmenter as a small set of KPIs, normally less than 10, that are fully aligned and provide focus in an organization [Par15]. In software engineering, KPIs represent a set of measures that correspond to internal performance indicators for designing, developing, and operating software, such as

its delivery time, usage or reuse of new and existing technologies, end user perception, etc. [Par15; Mol09]. Considering the usage of cloud environments to host applications, application architects must go a step further in the definition of KPIs, by means of analyzing each cloud provider SLAs and by defining each Service Level Objective (SLO) for their applications [FRL13]. Frey et al. identify in [FRL13] a set of KPIs and measures depending on the utilized cloud service. For instance, cloud storage KPIs for storing and retrieving data encompass response time, throughput, read and write speed, and resource provisioning time. Cloud CPU KPIs comprise the utilization of CPU and memory, as well as the average migration and interruption time. Such classification is aligned with the generic service-oriented and efficiency-oriented user perception indicators identified by Molyneaux in [Mol09], which define the ability of a system to actually provide a service to end users, and the quality of such a service, respectively.

KPIs-based performance models can be used in several steps of software development. According to Bailey et al., these can assist to understand and predict performance of applications [BS05]. In particular, performance models can be leveraged to design and tune both application and underlying resources, as well as to estimate runtime parameters during the design phase of the application. Palladio Component Model (PCM) is probably the most used performance model in the research domain. PCM is a domain specific performance modeling language to describe performance aspects of component-based software architectures [Hub14]. Its main benefit relies on enabling early performance and cost prediction for architectural models. A similar approach using the RESOLVE specification language as the basis is driven by Sitaraman et al. [SKK+01]. However, the main restriction of approaches tackling the performance prediction of component-

based systems is the necessity of modeling the environment used to host the application. In cloud environments, the configuration of underlying resources is driven by monitoring, predicting, and defining elasticity parameters for the provisioned resources. For instance, AppRAISE supports the specification of performance parameters of distributed applications for managing and resizing virtualized server environments [WCG+09].

Due to the high performance variation observed in cloud environments, several works tackle the derivation of performance and prediction models for providing architectural feedback loops for production-based architectures, i.e., during the production phase of the application. Such models can be derived as software performance anti-patterns [SW02; SW03; Tru11], incorporating reoccurring performance problems and their corresponding solutions. The adoption of such concepts in the cloud is ongoing research work, as seen in Trubiani [Tru15; Tru11], having as its main building block the conduction of architectural feedback activities. Performance prediction in cloud environments typically follow a bottom up approach, meaning that testing the performance of cloud services can assist in defining performance models and elastic strategies, as discussed by Mian et al. [MMZV13] and Hwang et al. in [HBS+16].

On the other end of the spectrum, cloud providers' interests are related to extracting performance models of resources utilization to predict consumption and fulfill SLA levels. For instance, Watson et al. propose a probabilistic performance modeling methodology for virtualized environments [WMG+10]. The probabilistic performance modeling methodology has as foundation the characterization of workloads, which can be leveraged to estimate response time distributions for a variety of applications. A similar characterization technique is

employed by Rahimizadeh et al. on a highler level [RAKJ15]. More specifically, the workload characterization approach consists of identifying groups of VMs that have certain workload patterns and can be categorized into logical clusters with unique prediction models.

2.4.2 Workload Modeling & Specification

The specification of workloads and derivation of workload models is a substantial challenge when designing and operating applications. Understanding workload trends or behavioral patterns can significantly help application architects to efficiently design and optimize application and infrastructure architectures [JVS98]. In other words, characterizing application workloads and deriving workload models can lead to efficiently designing, evaluating, and refining application architectures [JVS98].

Workload modeling and specification has been extensively covered in the literature. Under the umbrella of workload modeling and characterization approaches, the work of Feitelson is probably the most extensive one [Fei15]. Feitelson defines a workload model as an *attempt to create a simple and general model, which can then be used to generate synthetic workloads at will*. Workload models can be leveraged in the evaluation tasks of applications, as it can be typically used to emulate the workload that an application handles during its production phase. Among the benefits of workload models introduced by Feitelson, the most pertinent in the scope of our work are related to:

1. providing a *generalized* and more *understandable* view of the workload characteristics, which can be

2. *adjusted* and *controlled* to control the influence of different behavioral parameters, e.g., user arrival rate, and can help to
3. *predict* in order to optimize the system's performance.

The derivation of workload models often starts with analyzing measured data about the workloads. More specifically, this analysis consists of processing traces or logs during a certain time interval, which allow to create a descriptive workload model depicting the characteristics of a workload. Workload descriptive models are typically comprised of statistical summaries of observed workloads that are used in some way to characterize workloads w.r.t. their behavior. The biggest challenge when deriving descriptive workload models is to first identify the degree of detail or number and type of parameters in the model that will enable the characterization of a workload [Fei15]. This challenge is addressed in a domain-specific manner. For instance, if the system to test is an operating system, the defined workload parameters are related to the number of processes or jobs, I/O, etc. However, when evaluating a Web application, further parameters must be considered, such as the arrival rate of users and the mix of their operations, network rates, etc. [BM11].

Feiltelson identifies two main types of workloads: *static workloads* and *dynamic workloads*, exemplified in Fig. 2.5 [Fei15]. Static workloads describe unique jobs that are executed in a system, while dynamic workloads describe a continuous arrival of jobs. Dynamic workloads are the ones that are typically seen in Web applications, where servers interact with multiple clients, each with an independent behavior impacting on the state of the application. Such behavior can be modeled as Markov chains, as seen in Fig. 2.5 and proposed by Van Hoorn in [VRH08]. An adapted and generic vision on [VRH08]

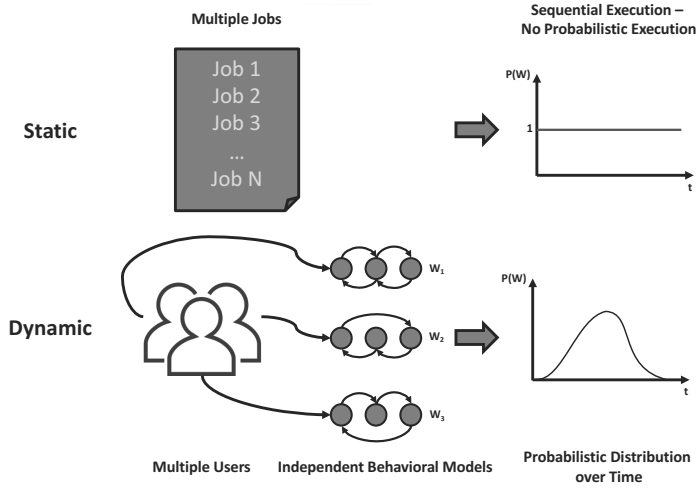


Figure 2.5: Static vs. Dynamic Workload

is adopted in Ch. 5 for defining cloud application workload models. One important aspect of dynamic workloads is that these are probabilistically distributed over time, as seen in Fig. 2.5. On the other hand, static workloads do not expose any kind of continuous arrival, therefore not following a probabilistic distribution. Although multiple works deal with characterizing workloads in different types of systems, such as centralized or network systems [CS93], the literature covered in the remainder of this section is scoped to the characterization and generation of workload models in Web and cloud environments.

Focusing on Web-based systems, the characterization of workloads is fundamental for creating a reusable workload model. Bahga et al. introduce in [BM11] steps for modeling workloads through characterizing and estimating workload attributes, by means of selecting

a probabilistic model describing the arrival of users, by estimating its parameters, and measuring the goodness of fit. Focusing on realistic applications, Bodik et al. focus on identifying and analyzing the behavior of workload spikes in stateful applications [BFF+10], while Arlitt and Jin analyzes the workload characteristics of the 1998 world cup Web site using statistical techniques and characterizing the popularity of Web pages [AJ00]. Bodik's analysis highlights the importance of deriving pages access patterns in order to proactively cache such a content to reduce the load on the Web application.

The characterization of workloads in cloud environments has been in the last years a prominent research domain, due to the necessity to scale resources in a proactive manner. Mishra et al. focus on classifying workloads of Google compute clusters, which are constituted by tens thousands of daily tasks [MHCD10]. Their approach consists of defining categories of tasks using k-means clustering techniques, which can help system designers define capacity planning techniques. Clustering and classification techniques have been leveraged in the derivation of *Workload Demand Patterns*, which capture workload cyclical behaviors in observed time intervals [GRCK07]. More specifically, Gmach et al. processed traces of Hewlett Packard (HP) clusters and extracted cyclic workload behavioral demands and used similarity analysis on several resource consumption metrics, such as % of CPU usage, to extract time-based workload demand patterns. Such patterns aim at assisting in the automation of efficient usage of resource pools when hosting large numbers of enterprise services [GRCK07]. A more coarse-grained approach is adopted by Khan et al., which consists of categorizing groups of VMs in a data center that frequently exhibit correlated workload demand patterns in concrete time intervals [KYTA12].

As previously discussed, one main advantage when characterizing and modeling workloads is the possibility to emulate production-based environments when testing performance of applications. Performance testing is typically driven with synthetic benchmarks, and its main goal is to test and stress a system. Workload models can serve as the input for generating synthetic workloads, as performed in [BM11; VRH08]. Nowadays, there exists a significant amount and types of benchmarks for independently and jointly evaluating performance of software applications and their underlying infrastructure. For instance, Cloud Spectator released the top 10 cloud vendor benchmarks[26].

2.4.3 Performance Evaluation & Evolution

A central challenge in designing performance-efficient applications in cloud environments relies on the facts that (i) workloads may evolve over time, and that (ii) the usage of virtual environments has an impact on the allocation of physical resources [Fei15]. Focusing on the workload variation, these are categorized by Feitelson as stationary, cyclic or trend workloads [Fei15]. Concentrating on cloud applications, Fehling et al. [FLR+11] identify a set of workload patterns for cloud applications and describe their impact on resources provisioning and decommission. In particular, Fehling et al. identify cloud workload behaviors as static or periodic, once in a lifetime, continuously changing, leading to an unpredictable behavior.

Common practices to tackle the evolution of application workloads and their impact on shared resources can be folded into two main categories w.r.t. how the system reacts to such evolution: (i) *proac-*

[26] CloudSpectator Top 10 Cloud Vendor Benchmarks: `http://connect.cloudspectator.com/cloud-vendor-benchmark-2016-pdf-download`

tive and (ii) *reactive*. Proactive approaches consist of evaluating and deriving methods and mechanisms to predict performance at the different levels of the application, based on past observations. For instance, benchmarks are typically used to drive performance and capacity tests for application components and infrastructure resources, respectively. Some examples are the TPC-*[27] (used as database benchmark for cloud services in Ch. 3), SPEC [28], and the YCSB [29]. These are used during design and pre-production phases of applications in order to evaluate in advance the performance of systems and environments. Focusing on the design of applications, Meier et al. provide a set of best practices and patterns for designing performance-aware applications [MFB+07]. Concentrating on the performance of infrastructure resources, prediction techniques are commonly used, such as in [GRCK07; GCF+10]. These typically leverage the usage of statistical models to derive usage patterns that can be subsequently used in predictive mechanisms.

Reactive approaches deal mainly with adapting infrastructure resources exploring optimal scalability techniques. Scalability is a wide research topic [oNeu94], and it is not the aim of this section to elaborate on it, but on giving an overview on scalability strategies to satisfy SLO and QoS levels. Ahluwalia defines a set of patterns for designing scalable applications [Ahl07]. The most relevant are patterns that enable optimized decentralization and scalability automation. Focusing on the strategies for scalability, Huber et al. presents a scalable mechanism based on the usage of strategies, tactics, and actions [HvHK+14]. The evaluation of scalable application configu-

[27] Transaction Processing Council: http://www.tpc.org/default.asp
[28] Standard Performance Evaluation Corporation: https://www.spec.org
[29] Yahoo! Cloud Serving Benchmark: https://github.com/brianfrankcooper/YCSB/wiki

rations are also part of current research works, such as [HBS+16], and typically use benchmarks to evaluate different scaling parameters used for scaling-out/-up applications.

Moving the discussion to cloud environments, several works demonstrated high performance variation of cloud resources. Ambrust et al. observed that the sharing of CPU and memory works surprisingly well in cloud environments, while network and disk resources are more problematic [AFG+10].

When comparing different cloud providers, there currently exists a big performance variation among providers, as exposed in the Cloud Spectator benchmark report[30]. The Cloud Spectator report is in line with the one-year performance experiment report of Iosup et al. in [IYE11]. In particular, Iosup et al. conclude that there exist yearly and daily workload patterns, and that workload variations are dependent on the application type. According to Armbrust, the biggest variation among providers is observed in the storage and network resources. Cloud environments also expose a high performance variation among their availability zones, as demonstrated by Unuvar et al. [UTD+15]. Focusing on one cloud provider, Schad et al. discovered in [SDQ10] a significant performance variability for the same cloud service provisioned in different regions.

Most of performance evaluation approaches are either based on simulation mechanisms or on independent benchmarks. We observe a lack of support for automatically capturing and processing performance knowledge in a collaborative manner, i.e., based on experiences for different types of applications. The collaborative sharing and analysis of performance knowledge can help to discover cloud services to suit different types of applications, and can be used to avoid the

[30] Cloud Spectator: `http://cloudspectator.com`

complexity introduced by, e.g. simulation models.

2.5 Utility Theory in Computing Systems

Utility theory emerged in the domain of economics, but its usage has also been adopted in provisioning and management of computational resources in distributed environments. Focusing on its definition, utility is defined in different manners. For instance, Marshall defines utility as *the perceived satisfaction when consuming a good or service* [Mar09], while Strunk and Fishburn define utility as *a value representing the desirability of a particular state or outcome* [STFG08; Fis70]. All definitions have in common the fact that utility is a *measure of preferences over a set of goods or services*, and is typically used in game theory and decision making mechanisms, e.g., multi-attribute utility theory [KR93]. Considering the relationship among preferences established by Ingersoll as [Ing87]:

$$U(X) \geq U(Y) \implies X \succeq Y \tag{2.1}$$

where, X and Y are goods or services, and the utility function $U : Z \rightarrow [0,1]$ is a mean to establish a preference, i.e., an order between X and Y.

Focusing on the usage of utility theory in computer systems, leveraging utility-based techniques is basically distributed under two main umbrellas: (i) autonomic computing, and (ii) management of cloud applications and environments. In the domain of the former, self-optimization is a well-known sub-domain in autonomic computing systems. Autonomic computing systems have the capability of manag-

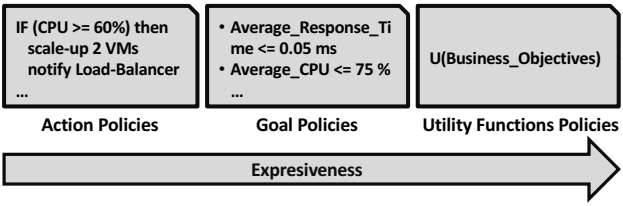


Figure 2.6: Self-Management Policies

ing computational resources in a self-adaptive manner, by means of evaluating its internal behavior and adapting them to satisfy system requirements. In other words, Kephart envisions autonomic computing systems as systems that manage themselves according to organization's operational goals [KC03]. Self management is achieved by the definition of policies, which are categorized as *Action Policies* and *Goal Policies* in the literature.

Figure 2.6 exemplifies the different policies and depict the degree of expressiveness among them. Action policies define an action to be driven when a system is in a given state, and are typically formulated as IF(Condition) THEN(Action) clauses [KC03]. Goal policies represent the desired state of the system, which are responsible for computing one or multiple actions to achieve such a state [KC03]. However, these policies are restricted in terms of expressing fine distinctions of preferences. *Utility Function Policies* address such a restriction and are viewed as an extension of goal policies. In particular, utility function policies assign a real-valued scalar representing the desirability of each state. Utility function policies iteratively compute the utility of each desired state in advance, and allow the efficient decision making by analyzing the trade-off between system states w.r.t. given preferences. Kephart and Das enhanced IBM products, such as the

Web-Sphere Extended Deployment (WXD) and the Tivoli Intelligent Orchestrator (TIO), with capabilities to define and adapt resources w.r.t. utility function policies [KD07]. Their experiments showed an improved efficiency of the WXD and the TIO when using utility to allocate resources. Although the main advantage of utility function policies are their fine degree of expressiveness, these entail challenges for eliciting multi-dimensional system objectives. In particular, such challenges relate to deriving and realizing the underlying utility model, and the decision support algorithms, as these are typically domain and application specific. For instance, Walsh et al. propose and realize a data center management system allowing the definition of utility function policies depicting high-level business and service-level attributes for Web applications [WTKD04]. More specifically, Walsh et al. contributed towards Unity, a self-adaptive management system built atop of utility function policies, and developed a utility model for managing allocation of resources for monolithic Web applications [WTKD04]. However, Unity is not capable of handling complex distributed and scalable applications, as well as calculating the impact of resources reallocation on the business.

Due to the shift in the technological landscape towards the usage of cloud environments, current investigations focus on using utility as an optimization mechanism for the dynamic allocation and configuration of cloud resources. Most approaches focus on fulfilling certain satisfaction level, usually defined in the SLAs between cloud providers and consumers, while optimizing the cloud provider's operational costs [ACC+14]. Cloud service providers are mainly concerned with maximizing the usage of resources, while minimizing infrastructure and management costs. This requires to build optimal resource scheduling mechanisms that can leverage the usage of

utility functions, and used to ensure the satisfaction of SLOs. For instance, Minarolli and Freisleben analyze the trade-off between QoS and operational costs for maximizing a global provider's utility in IaaS environments [MF11]. The global utility is maximized by locally maximizing the utility of allocated VMs in each physical node, considering the CPU allocation and the costs per CPU in each physical node. Goudarzi and Pedram follow a similar approach, and focus on finding an optimal resource allocation to optimize the total profit gained from SLA contracts [GP11]. More specifically, Goudarzi and Pedram analyze the profitability of application by means of computing the end-user utility when consuming processing, memory, and communication resources. In the domain of storage systems, Strunk et al. leverage the usage of utility functions to provision and maintain distributed storage systems [STFG08]. Strunk et al. propose a utility model that aggregates performance, data protection, and cost metrics, and computes the estimated profit when distributing and allocating data resources. The work presented by Paton et al. exclusively focuses on using utility to coordinate the execution of workloads in cloud environments, without triggering the adaptation of underlying resources [PDL+09].

From the perspective of cloud consumers, migrating an application to the cloud requires selecting and provisioning concrete services to host its components. In other words, application architects or cloud brokers face a multi-dimensional decision making problem. Works in this domain typically consist of decision making support mechanisms and automated negotiation frameworks for facilitating the selection and provisioning of cloud resources. Antonescu et al. propose in [ARB12] a policy and action-based approach that matches and dynamically adapts the application topology and the configuration of

infrastructure resources. Unuvar et al. propose a utility model to evaluate cloud availability zones [UDS+14]. Utility-based mechanisms for automated negotiations are typically realized within cloud brokers, and consist of exploiting past SLA data towards evaluating the fulfillment of business objectives. For instance, Macías and Guitart provide a utility model towards classifying cloud providers [MG10], while Ronaldo and Zimeo [RZ13] propose a utility function that allows to evaluate the impact on the overall system's capacity before accepting a new service contract. In the scope of the PaaSage[31], Krikikos et al. define a set of utility functions to optimally select and configure IaaS cloud services [KMP16].

As previously introduced, the usage of utility theory in computing systems simplifies and benefits solving multi-dimensional decision making problems related to the (i) configuration of computing systems, and (ii) the selection of computational services. The usage of utility theory in the scope of this work has one major goal: *assessing business and IT experts in the decision making tasks when spanning their applications among multiple and heterogeneous cloud services and providers*. Strunk identified the challenge of utility elicitation, denoting the complexity and time consuming constraints when creating utility functions. Moreover, application developers may or not have have complete knowledge of utility theory [Str08]. A possible identified solution in this work consists of delivering a utility-based framework incorporating business models and operational costs. Towards such a goal, Ch. 8 presents a utility-based decision making support framework to evaluate the trade-off between cost and performance when selecting and configuring specific cloud services.

[31] PaaSage EU Project: `http://www.paasage.eu`

2.6 Case-based Reasoning (CBR)

Case-based Reasoning (CBR) emerged in the 80s in research fields of *Cognitive Science*, *Machine Learning*, and *Knowledge Based Systems*, motivated from the work of Schank [Sch83]. CBR builds upon the premise that *similar problems are best solved with similar solutions* [Lea96], and introduces methodologies and mechanisms to solve real world problems based on how humans think, reason, and solve them [Sol16]. Knowledge acquired from previous experiences can be therefore exploited, as *similar* problems tend to have *similar* solutions.

CBR differs from classic Artificial Intelligence (AI) techniques in the sense that it does not require an identical problem to be solved in the past [She03]. By analyzing new and previous problems with similar characteristics, CBR can propose potential solutions based on previously acquired knowledge. Due to the scope of this work w.r.t. the usage of CBR, the remainder of this section presents CBR fundamentals, and subsequently positions the discussion and related works in the domain of applying CBR for engineering applications. A detailed presentation on CBR can be found in the works of K. Pal and Ck. Shiu [PS04], Shepperd [She03], and Aamod and Plaza [AP94].

2.6.1 Case-based Reasoning Foundations

According to Shepperd, CBR relies on utilizing memory of *previously experienced situations* to tackle *new encountered situations* [She03]. Such situations are denoted in CBR as *cases*, which can be either *past cases* or *new cases*. Past cases correspond to previous problem solving experiences, while new cases are new emerged problems to be solved. A case typically groups a problem description and a description of its corresponding solution. In practice, the procedure when utilizing

CBR is presented by Aamodt and Plaza [AP94], and consists of the CBR cycle depicted in Section 2.6.1.1, which allows problem solving and knowledge enhancement in CBR. Since CBR cycle relies on the usage of *similarity analysis* among past and new cases, this section introduces an overview of similarity analysis in CBR.

2.6.1.1 CBR Cycle

The CBR cycle depicted in Fig. 2.7 is also known as the R^4 model, which consists of the following four processes introduced in [AP94]:

1. **RETRIEVE** a group of past cases most similar to the new case
2. **REUSE** retrieved knowledge from similar past cases in order to solve a new case
3. **REVISE** the proposed solution
4. **RETAIN** partially or completely the new solution in the knowledge base for future problem solving

In particular, the R^4 model comprises a common central *General Knowledge*. When a new problem arises, it must be arranged as a problem description, which is basically a vector of features representing a new case and is used as the basis for retrieving similar cases from the general knowledge [She03]. A new case is also referred in the literature as a *query*, and is processed in four main steps. The *Retrieve* step consists of retrieving one or multiple past cases, each containing solutions that can be reused to solve a new case. According to [AP94], the retrieval task is divided into identifying new case features, initially matching the new case with previous cases by searching in the general knowledge, and selecting the best match. Since there exists a low

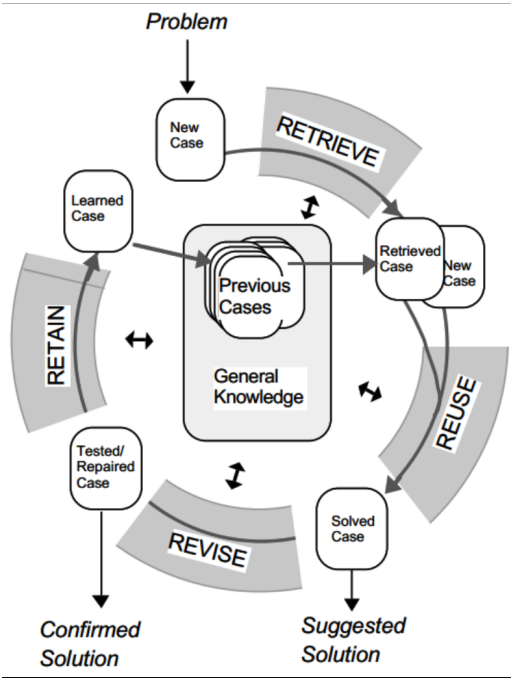


Figure 2.7: Case-Based Reasoning Cycle [AP94]

likelihood of finding an exact match between new and past cases in the knowledge base, the initial match and search subtasks rely on the usage similarity analysis (see Sec. 2.6.1.2) and its corresponding *similarity measures*. Similarity measures allow to retrieve past cases that are sufficiently similar to the new case. This task can also be referred as a *k*-nearest neighbor retrieval task using a *similarity function* as the basis [BAM+09]. The efficiency of the retrieval task has been questioned and investigated, as it directly depends on the

size of the general knowledge, and the utilized similarity measure. There exist approaches in CBR that specifically target the case retrieval efficiency [Ber02].

During the *Reuse* step, the retrieved solutions are reused to solve problems in the context of new cases. The reuse step first focuses on identifying the differences between a new case and the retrieved past case, and on determining which parts of the retrieved past case solutions can be reused. The output of such a step is a list of one or multiple *suggested solutions* for the new case, which can be generated in a manual or dynamic manner. The *Revision* encompasses the possibility to evaluate and repair, if necessary, the suggested solution. A *confirmed solution* is then used in the *Retain* learning step, which could consist of simply extracting the knowledge of how the new case was solved, indexing such a knowledge, and adding the confirmed solution to the general knowledge. However, there are two main challenges in these subtasks, which are related to the problem of knowledge acquisition and storage. On the one hand, extraction and addition of knowledge must be selective. Otherwise, efficiency of the retrieval may be affected as the general knowledge increases. Existing works propose using competence models [SM01] or learning global and local similarity measures [CH08; Sta05], which is introduced in Sec. 2.6.1.2. On the other hand, indexing of cases is not straightforward, as it relates to identifying what type of indexes to use for organizing and adapting the search space of indexes [Ber02]. There exist few works that address the problem of learning adaptation knowledge, such as [HK96] and [CJR01].

2.6.1.2 Similarity Analysis

When examining CBR, it can be clearly highlighted the continuous and repeated appearance of terms related to *similarity*. CBR technologies are built upon the usage and the computation of similarity measures, which are their underlying models for retrieving *approximately matching* information [Sta05]. In other words, similarity measures are used to calculate similarity among queries and past cases. Liao et al. identify two major case-based retrieval approaches that leverage the usage of similarity measures: *distance-based* and *representational* [LZM98]. Distance-based approaches consist of determining the distance between new and past cases, and the most similar ones are obtained by analyzing their distances. On the other hand, representational approaches use indexing mechanisms within the structure in knowledge bases to interconnect similar cases.

The purpose of driving similarity analyses in CBR systems is depicted in Fig. 2.8. In a nutshell, similarity analysis enables the encoding and selective retrieval of knowledge in the R^4 model (see Sec. 2.6.1.1) of CBR, and can be exploited to estimate the utility of retrieved past cases for a given new case, also denoted as query. When a new query arrives to the CBR system, the *Retrieval Engine* is mainly responsible for (i) retrieving potential similar cases from the knowledge base, and for (ii) analyzing and computing the similarity among them. The outcome from such operations allows the establishment of a preference relation among retrieved past cases. The efficiency and degree of details in the similarity calculation process is obviously a challenge in CBR systems.

There exist different levels of abstraction for computing similarities among cases, such as the *Local-Global Principle*, introduced by

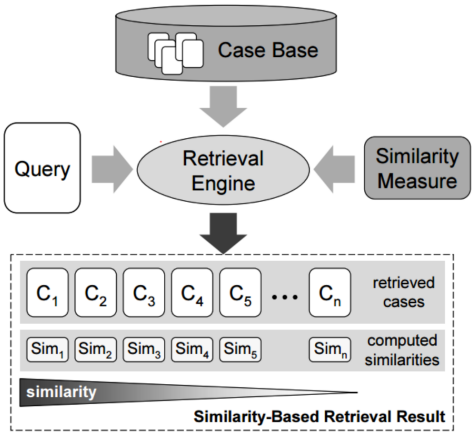


Figure 2.8: Similarity-Based Retrieval in CBR Systems [Sta05]

Bergman [Ber02] and Richter [Ric07]. The remainder of this section discusses fundamentals on similarity measures and its usage in local-global similarity analysis.

Similarity Measures The study of similarity measures is a wide research topic in CBR systems, and its foundations rely on mathematical models. Stahl defines a similarity measure that estimates the utility of a given case for a given query q as a function $Sim = \mathbb{D}_D x \mathbb{D}_D \longrightarrow [0,1]$ [Sta03]. There also exists various ways for represent similarity measures, which are identified by Richter as [Ric93]:

1. A binary predicate $SIM(x,y) \subset \mathbb{D}_D^2$ meaning that *x and y are similar*.
2. A binary predicate $DISSIM(x,y) \subset \mathbb{D}_D^2$ meaning that *x and y are not similar*.

3. A ternary relation $S(x, y, z) \subset \mathbb{D}_D^3$ meaning that *y is at least as similar to x as z is to x*.
4. A quaternary relation $S(x, y, u, v) \subset \mathbb{D}_D^4$ meaning that *y is at least as similar to x as u is to v*.
5. A function $sim(x, y) : \mathbb{D}_D x \mathbb{D}_D \rightarrow [0, 1]$ measuring the degree of similarity between x and y.
6. A function $d(x, y) : \mathbb{D}_D x \mathbb{D}_D \rightarrow \mathbb{R}$ measuring the distance between x and y.

According to Stahl, the previously denoted similarity measures definitions contain an increasing order of information [Sta03]. More specifically, SIM and $DISSIM$ cannot be leveraged for ranking retrieved similar results, as these are binary predicates. On the other hand, S and R contain only ordinal information, while sim and d also contain cardinal information. In practice, CBR systems use similarity or distance functions to induce similarity relations. Due to the scope of this work, we focus on leveraging the state-of-the-art w.r.t. similarity measures models. In particular, we adopt the notion of similarity measure as the function Sim defined in [Sta03], rather than defining a new mathematical model. The underlying mathematical model of Sim is detailed in [Sta03].

There exist numerous traditional similarity and distance measure models depicted in the literature and listed in [Sta05], such as the *Hamming Distance*, *Simple Matching Coefficient (SMC)*, *Weighted (SMC)* for binary attributes, and *City Block Metric*, *Euclidean Distance*, or the *Minkowsky Norm* and *Weighted Minkowsky Norm* for numeric attributes. However, the previous models are constrained to case representations consisting of attributes with single value types, such as binary. Such a restriction makes such models inappropriate for using

when case representations are defined in a more complex manner, i.e., consisting of attributes with different value types. To address such a challenge, the *local-global principle* proposed by Richter [Ric04] allows an approximation oriented representation of knowledge. In the following sections we outline the fundamentals of such a principle and its benefits.

Local-Global Principle The *local-global principle* was first proposed by Richter in [Ric04] and consists of breaking the computation of similarity among cases into a local and a global part. More specifically, this principle decomposes similarity calculation into a local part, only considering *local similarities* between single attribute values of each case, and a global part, aggregating a *global similarity* among the different cases, each exhibiting a local similarity measure [Sta03].

Local similarity measures represent the utility among cases by analyzing their details, i.e., their attributes. Depending on the type of each attribute, the similarity measure model varies. Stahl generalizes in [Sta03] a local similarity measure as $sim_A : \mathbb{A}_{range} \times \mathbb{A}_{range} \rightarrow [0, 1]$, where $\mathbb{A}_{range}$ corresponds to the value range of an attribute A. However, there exist distinct similarity measures that depend on the value type of the attributes. In the case of computing similarity measures among *discrete value types*, Richter [Ric08] proposes the usage of lookup tables known as *Similarity Tables*. A similarity table is a matrix structure representing similarity between a query value and a case value. Fig. 2.9 depicts a similarity table for comparing the casing of personal computers. In particular, a customer will not be satisfied when demanding a *laptop* (query q) and receiving a *big-tower* (case c). However, they will be mostly satisfied when demanding a *mini-tower* and receiving a *midi-tower*. Note that the matrix diagonal is 1, as in the

q \ c	laptop	mini-tower	midi-tower	big-tower
laptop	1.0	0.2	0.1	0.0
mini-tower	0.3	1.0	**0.9**	0.5
midi-tower	0.2	**0.7**	1.0	0.7
big-tower	0.1	0.4	0.6	1.0

Figure 2.9: Similarity-Table [Sta03]

case of demanding a *laptop* and receiving a *laptop*, a customer is fully satisfied. Similarity tables can be leveraged in assessing modeling tasks of cloud applications, as shared past knowledge can be exploited towards finding how similar applications were previously distributed.

Similarity measures for *numeric value types* adopt the usage of functions calculating the distance between two values. According to Stahl [Sta03], *difference-based similarity functions* can compute the similarity value $sim_A(\delta(q,c)) = s$ using the difference function $\delta : \mathbb{A}_{range} \times \mathbb{A}_{range} \rightarrow \mathbb{R}$ as the basis. Difference-based similarity functions assume a decrease of similarity when an increase of the distance is observed. Therefore, the correct adoption of similarity functions in CBR systems is crucial for retrieving past similar cases in an accurate manner. Stahl identifies two common difference functions, such as the *linear difference* and the *logarithmic difference* [Sta03], and four typical similarity functions, such as the *threshold*, *linear*, *exponential*, and *sigmoid* functions. When computing similarity among *structured value typed* attributes, the information relevant for the similarity is typically encoded within the type itself [Sta03]. In particular, *ordered relations* can be defined by the user based on symbols, or symbols can be structured arranged in a *taxonomy tree generalization relation*.

By using such structures, the similarity definition relies then on the underlying structure.

Having analyzed local similarity among cases' attributes, a second step in the local-global principle consists of calculating the *global similarity measure* among the cases. More specifically, the global similarity corresponds to using an *aggregation function* that computes similarity based on the previously calculated local similarity values. In a formal way, Stahl defines a global similarity measure $Sim : \mathbb{D}_D x \mathbb{D}_D \rightarrow [0,1]$ as [Sta03]:

$$Sim(q,c) = \pi(sim_1(q.a_1, c.a_1), ..., sim_n(q.a_n, c.a_n), \overrightarrow{w}) \qquad (2.2)$$

where sim corresponds to a local similarity function for each attribute, $\overrightarrow{w}$ to a vector of weights, and $\pi : [0,1]^{2n} \rightarrow [0,1]$ is a monotonously increasing aggregation function in the arguments representing the local similarity values. In practice, CBR systems move towards the usage of simple functions, such as the *Weighted Average Aggregation*, *Minkowski Aggregation*, *Maximum Aggregation*, or *Minimum Aggregation*. Again, an adequate selection of an aggregation function is crucial for accurately retrieving similar cases.

2.6.2 CBR in Software Architectures

Besides the previously introduced CBR definition, Shepperd also defines CBR as *a technique for managing and using knowledge that can be organized as discrete abstractions of events or entities that are limited in time and space*. If such a definition is extrapolated to the domain of software architecture, cases basically consist of vectors of features or characteristics that represent a software architecture. In particular,

such a vector may contain number of interfaces, size of software packages, development methods, etc. A CBR system dedicated to retrieving similar previous software architectural solutions can be visualized as a repository containing previous software architectural solutions for appeared problems. The usage of CBR for architecting software falls into two main categories: *prediction* and *reuse* of applications [She03].

The usage of CBR systems for prediction purposes consists of effectively making cost and quality predictions of software projects. Using CBR systems for reuse purposes aim at increasing the productivity of software projects. For example, using patterns or software experience can help in gaining productivity while maintaining or increasing the quality of software [She03]. There exist several works for enabling prediction and reuse of CBR systems as the basis. For example, the Estor system [MVP92], and the COCOMO [Boe+81] and FACE [BM95] approaches are positioned in the category of software architecture prediction. Semantic networks, faceted index approaches, or *Experience Factory (EF)* [BCR94] and *Leasons Learned (LL)* [WAB01] systems are common in the reuse of software architectures. EF systems are in practice more beneficial as LL systems, as these include an explicit notion of context, and empirical evidence is used to evaluate potential new cases. A more extensive overview on existing concepts and technological applications of CBR techniques are available in [She03].

Moving towards the exploitation of CBR systems in cloud computing, Soltani et al. [SME14] presents QuaRAM, a case-based reasoning system for selecting IaaS services. QuaRAM analyzes application requirements, constraints, and preferences, and retrieves similar cases from its knowledge base. In other words, an application profile constitutes the query, and the cloud provider and VM configuration correspond to

the solution for a particular case. However, there are further aspects to consider when analyzing similarity among applications in cloud environments, such as (i) the distribution of application components among different services, (ii) the impact of the application workload behavior on the performance of cloud services, and (ii) the usability and quality of retrieved cases w.r.t. the given problem. For the latter, Stahl [Sta03] motivated the usage of utility functions to generate a preference relation among retrieved cases, which is leveraged in this work as the evaluation mechanism for estimating the profitability of application distributions. CBR is used in this work as the basis for capturing and processing application performance and workload knowledge. In particular, the mechanisms for the dynamic discovery and construction of viable cloud application distributions in Ch. 7 use CBR as the basis for generating viable distributions of application components using past knowledge from similar applications.

Chapter 3

Performance Characteristics in Cloud Environments

3.1 Chapter Introduction

The intrinsic properties of cloud infrastructures raise two major performance challenges when migrating applications to the cloud: (i) resources are typically shared among multiple users and applications, each of them exhibiting different behaviors and characteristics, and (ii) cloud providers aim at maximizing resource utilization, thereby impacting offered performance to cloud consumers [MG11].

Performance evaluation frameworks are a well-known research

topic in the cloud domain. Cloud benchmarks, such as [LML+11; BLL+14; YZSD12; FFRR15], evaluate different types of cloud services, cloud providers, and their corresponding SLAs. However, there exists a gap when analyzing performance variation of cloud services over time. In particular, there are few works, such as [SDQ10; LC16], that study the inherent performance volatility of cloud providers and services using benchmarks that target each service component independently, e.g., CPU or memory consumption for VM-based services.

By capturing and analyzing performance requirements and workload knowledge during design and production phases of applications, application architects can efficiently refine architectural models to fit changing performance needs [GB16]. In particular, application architects can refine the selection and configuration of cloud providers and services. However, such a knowledge must be firstly harvested, and must incorporate the means to capture performance characteristics of different types of applications, cloud services, and cloud providers. This section focuses on identifying and deriving the necessary knowledge and ingredients by empirically analyzing performance characteristics through a set of controlled experiments, in line with Pfleeger's experimental guidelines [Pfl95].

3.2 Experimental Methodology

In the field of experimental analysis in software engineering, Pfleeger defines a hypothesis as a *statement stated by the examiner at the beginning of an experiment, which is believed to be truth, unless proven otherwise* [Pfl95]. The experimental methodology is built by defining a structured list of hypotheses, since there are multiple concerns and actions related to performance among cloud services and providers

that need to be proofed. In particular, the hypotheses are the following:

> **H1** There exists a seasonal performance variation among cloud providers and services.

Cloud environments rely on virtualizing resources, which are shared among multiple applications. According to the pay-per-use cloud model, each cloud consumer can use cloud resources in irregular intervals [MG11]. However, application workloads may partially or completely follow seasonal behaviors (also denoted as usage patterns) [LC16; FLR+14].

There exists prior evidence of a variation in performance among cloud providers in services, as discussed in [IYE11]. This investigation shows the existence of a monthly performance variability, which is mainly due to the combined effects of system size, workload variability, virtualization overhead, and resource time-sharing.

> **H2** The execution of applications in the cloud suffers from contrasting performance and cost variations, depending on the cloud provider and offering.

When observing performance among cloud providers and services, there exists a fluctuation w.r.t. their offerings. The utilization of specialized cloud services, such as compute or memory optimized VM instances, for instance, may increase the performance of cloud applications. However, these typically entail higher monetary costs.

H3 There exists a boost in application performance when distributing its components, by spanning them among different cloud services within each provider.

Migrating legacy applications and placing their entire stack in a VM does not fully exploit the benefits of cloud environments [ABLS13; LFWW16]. However, distributing and placing application components among specialized cloud offerings can significantly benefit the overall application's performance.

H4 The separation of application data and business logic tiers in separate cloud services leads to a decrease of application's performance while increasing networking costs.

This hypothesis is related to Gray's statements in the *Distributed Computing Economics* [Gra08]. Gray basically concludes that separating data computation and data storage, by means of hosting them in different machines, has a negative implication on the overall application's performance and cost.

H5 The utilization of specialized cloud services boosts the overall application's performance.

Cloud providers have moved in the last year towards offering specialized cloud services. For instance, AWS has specialized the EC2 service, by means of moving further than the *General Purpose*, and offering *Compute Optimized*, *Memory Optimized*, and *Storage Optimized* VMs instance types. The utilization of such specialized instances can boost the overall application's performance, depending on their type.

Table 3.1: Mapping of Experiments & Hypotheses

	Hypothesis				
	H1	H2	H3	H4	H5
Experiment 1 (Exp.1)	✓	✓			✓
Experiment 2 (Exp.2)		✓	✓	✓	
Experiment 3 (Exp.3)		✓			✓

The previously introduced hypotheses have as fundamental goals in this work (i) to establish the coverage of driven experiments, (ii) to investigate and update the state-of-the-art on analyzing variations in performance among cloud offerings, (iii) to evaluate the effect on performance when distributing applications, and (iv) to extract, scope, and build the necessary knowledge to capture and analyze performance and its variability over time. The establishment of these hypotheses basically serve as the basis for identifying performance concerns and constructing knowledge boundaries for the remainder of this work. Table 3.1 provides an overview of the defined hypotheses an driven experiments, with a focus on mapping which hypothesis(es) each experiment verifies or falsifies.

3.3 Experiment 1 (Exp.1): Business Application

Business applications are typically built with the goal of achieving a certain economic profitability[1]. For instance, e-commerce applications are expected to be utilized for commercial transactions, such as the trade of articles.

There exists several business applications in the market. However,

[1] Microsoft Application Types: https://blogs.msdn.microsoft.com/jmeier/2010/07/29/application-types-app-types-the-early-years/

very few are publicly available for experimental purposes. In this set of experiments we use the well-known TPC-H[2] application, part of the TPC benchmark suite, which emulates a three-layered business application [Fow02]. The experiments discussed in this section appeared also as part of the conference paper [GALS14b].

3.3.1 Experimental Methodology & Setup

This experiment focuses on migrating application data to the cloud. More specifically, it aims at evaluating performance variability among offerings, when migrating application data to different cloud services and configurations, e.g., AWS EC2 or AWS Relational Database Service (RDS). The following infrastructures are used:

- an on-premise virtualized server on 4 CPUs Intel Xeon 2.53 GHz with 8192KB cache, 4GB RAM, running Ubuntu 10.04 Linux OS and MySQL 5.1.72,
- an off-premise virtualized server (IaaS) hosted in Flexiscale[3] consuming 8GB RAM, 4 CPUs AMD Opteron 2GHz with 512KB cache, and running Ubuntu 10.04 Linux OS and MySQL 5.1.67,
- an off-premise virtualized server (IaaS) Amazon EC2[4] *m1.xlarge* instance hosted in the EU (Ireland) zone and running Ubuntu 10.04 Linux OS and MySQL 5.1.67,
- and an off-premise Amazon RDS[5] DBaaS *db.m1.xlarge* database instance hosted in the *eu-west-1* (Ireland) region and running MySQL 5.1.69.

[2] TPC-H Benchmark: http://www.tpc.org/tpch/
[3] Flexiscale: http://www.flexiscale.com
[4] Amazon EC2: http://aws.amazon.com/ec2/
[5] Amazon RDS: http://aws.amazon.com/rds/

When migrating the application's data layer to the cloud, the following application distribution scenarios are considered: the application data is hosted in (i) a MySQL server on-premise, (ii) in a DBaaS offering, and (iii) in a MySQL server deployed in an IaaS offering. In all cases, 1GB of application data is generated using the TPC-H benchmark query generator. Since the focus of this experiment is on analyzing performance variability, we measured performance across 10 rounds on average per day for a period of three weeks in the last quarter of 2013.

```
select sacctbal,sname,nname,p_partkey,p_mfgr,
    s_address,s_phone,scomment
from part,supplier,partsupp,nation,region
where p_partkey = ps_partkey and
s_suppkey = ps_suppkey and
p_size = 47 and p_type like '%BRASS' and
s_nationkey = n_nationkey and
n_regionkey = r_regionkey and
r_name = 'MIDDLE EAST' and
ps_supplycost =
(select min(ps_supplycost) from partsupp,
    supplier, nation, region where p_partkey =
    ps_partkey and
s_suppkey = ps_suppkey and
s_nationkey = n_nationkey and
n_regionkey = r_regionkey and
r_name = 'MIDDLE EAST')
order by s_acctbal desc, n_name, s_name,
    p_partkey limit 100';
```

Listing 3.1: TPC-H Benchmark - Sample Query

The experimental workload consists of a set of 23 TPC-H queries, which are used as the basis to generate an uniformly distributed workload of 1000 queries. A sample TPC-H query is provided in Listing 3.3.1. The workload is subsequently loaded in Apache JMeter 2.9[6], used as the application load driver to emulate the application business logic layer. In all scenarios, the load driver remains on-premise.

3.3.2 Experimental Results & Findings

Figures 3.1, 3.2, and 3.3 show the dispersion of measured response times among multiple experimental rounds.

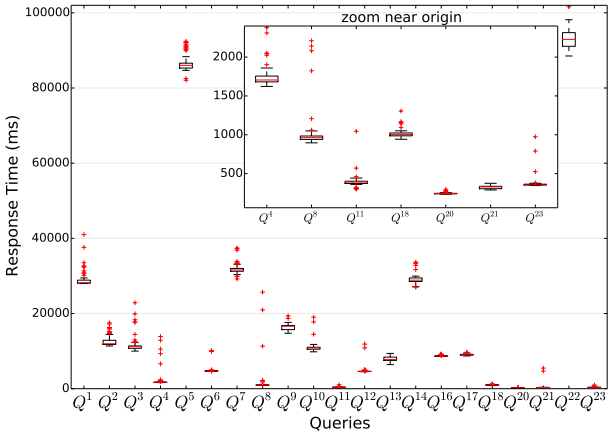


Figure 3.1: On-premise Performance Results [GALS14a].

For the on-premise deployment results in Fig. 3.1, the latency experienced by the application is distributed into three main intervals: (80000, 100000) ms, (20000, 40000) ms, and (0, 20000) ms. A

[6]Apache JMeter: http://jmeter.apache.org

slightly improved latency is observed when executing the workload in a database deployed in the AWS EC2 IaaS offering (see Fig. 3.2). However, when executing the workload in a database deployed in a AWS RDS DBaaS offering, the previous behavior is reverted, as the latency is significantly reduced by approximately 90% on average (see Fig. 3.3).

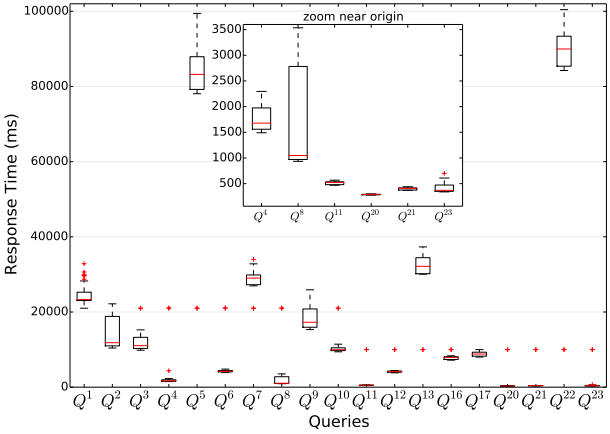


Figure 3.2: AWS EC2 (IaaS) Performance Results [GALS14a].

The improvement when deploying the database in AWS RDS is attributed to the fact that the database's engine configuration in a DBaaS solution is highly optimized for complex (retrieval) query processing. However, such performance increase is not observed across all sampled workload operations. More specifically, more than 50% of the workload operations present a high amount of outliers. Such variations, however, do not always follow a trend towards a performance degradation, as observed in the on-premise scenario

depicted in Fig. 3.1. For example, for workload operations $Q^{(8)}$, $Q^{(12)}$, $Q^{(14)}$, and $Q^{(16)}$, there are latency observations which improve the performance in approximately 10% in average with respect to the median.

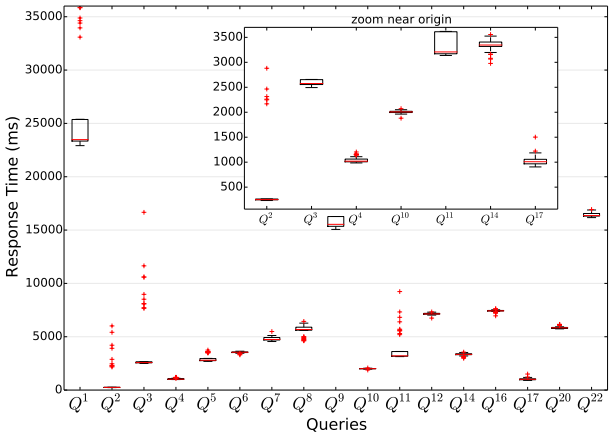


Figure 3.3: AWS RDS (DBaaS) Performance Results [GALS14a].

The previous results show for most scenarios a divergent performance which fundamentally depends on the concrete workload operation (database query) and the infrastructure where it is executed (on-premise, AWS EC2, or RDS). In particular, scenarios where the data layer is hosted on-premise and on AWS RDS present more outliers than the scenario where the data layer is hosted on an AWS EC2 VM. Despite the observed performance variation in all scenarios and workload operations, a categorization of workload operations is still possible, as their performance fluctuates between certain values. For example, the on-premise scenario in Fig. 3.1 showed three main

performance intervals: (80000, 100000) ms, (20000, 40000) ms, and (0, 20000) ms.

As previously introduced, the fluctuation of application workloads can highly vary in a virtualized environment. The fluctuation can occur according to different aspects, e.g. end-user demands, popularity of the data, season of the year, etc. The second part of this experiment consists of finding a seasonal performance behavior, as similarly analyzed in [GRCK07], and depicted in Fig. 3.4.

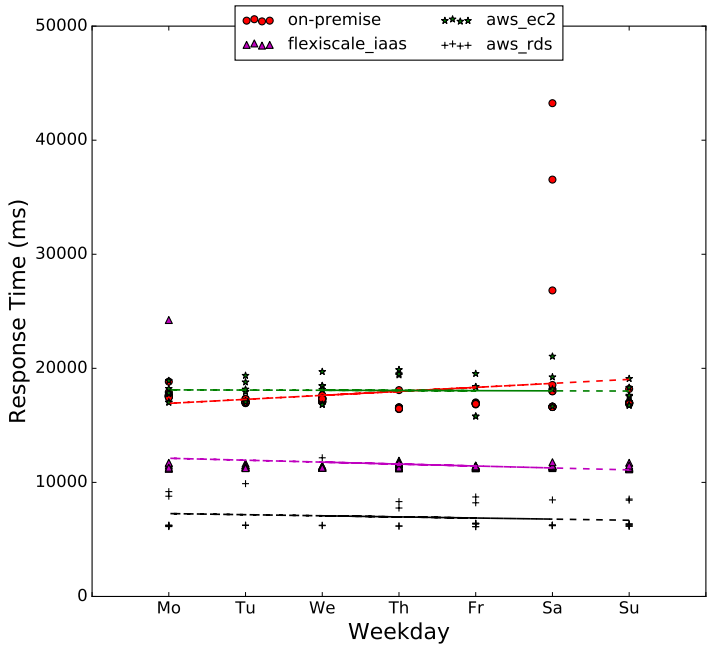


Figure 3.4: TPC-H Benchmark - Cloud Provider Weekly Analysis [GALS14a].

The experiments show that there exists an overall performance

improvement when hosting application data off-premise. Furtermore, there exists a clear gain in performance when using DBaaS offerings, such as AWS RDS. The observations in this analysis also show a performance variation when executing the initial workload under different deployment scenarios. The on-premise and DBaaS scenarios present a highest performance variation, while a steadier performance is observed for the IaaS scenarios. Focusing on the performance trend for each scenario, off-premise scenarios present a rising trend when reaching weekends, while the on-premise scenario behaves in an inverse manner. Such performance degradation trend observed in the on-premise scenario relates to scheduled maintenance tasks running in parallel to our experiments at the University of Stuttgart. However, the on-premise deployment scenario shows, for example, a better performance until Wednesday, when comparing it to the AWS EC2.

3.4 Experiment 2 (Exp.2): Wiki Application

The second experiment in this work appears as part of the article [GALS14a]. This experiment uses the MediaWiki[7] application as its basis, used as the underlying technological support for the Wikipedia[8], which is daily utilized by millions of users.

MediaWiki is a two-tiered PHP-based open source application. Since the release of real access traces of several Wikipedia mirror servers, different research works have driven benchmark-related investigations using this data [UPV09; Pet09; CJMB11; AMC07].

[7] MediaWiki: `https://www.mediawiki.org`
[8] Wikipedia Project: `https://www.wikipedia.org`

3.4.1 Experimental Methodology & Setup

This experiment aims at moving a step forward w.r.t. Exp.1, by considering the application in a holistic manner and by analyzing further application constraints, e.g., its data transfer intensity. Moreover, this experiment focuses on distributing the tiers of MediaWiki among different cloud services. In particular, this experiment considers the following application distributions:

- Hosting the whole application stack in an on-premise virtualized server, configured with 4 vCPUs, 4GB RAM, and running Ubuntu 14.04 Virtual Linux LTS OS.
- Hosting the whole application stack in an AWS *m3.xlarge* EC2 instance.
- Hosting the business logic tier an on-premise or in an off-premise AWS EC2 *t2.medium*, while migrating the application back-end data tier to:
  1. a MySQL server hosted on an AWS EC2 *t2.medium* instance
  2. a *m3.large* DBaaS instance in AWS RDS.

The MediaWiki benchmark Wikibench[9] is used to sample and generate custom workload distributions. This experiment uses access traces of Wikipedia and its database dump from the year 2008. These are sampled using WikiBench, and then used to randomly generate a synthetic workload consisting of 200K HTTP requests. The final workload consists of retrieving, editing, and adding Wikipedia articles.

Apache JMeter version 2.9 is used for all experimental rounds as the load driver, creating 10 concurrent users and uniformly distributing

[9] Wikibench: http://www.wikibench.eu/

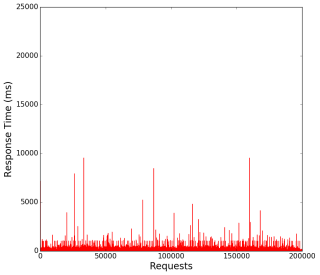
operations among them. Both the generated workload and the JMeter load driver configuration are publicly accessible in Bitbucket[10]. In all experimental rounds, the latency experienced by the application's end user is measured.

3.4.2 Experimental Results & Findings

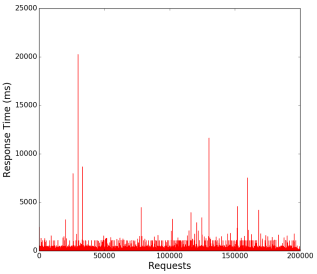
Figure 3.5 summarizes all scenarios of this experiment. When migrating the MediaWiki back-end data tier off-premise, experiments show that there is a performance degradation of approximately 300% in average when deploying its back-end database tier in an off-premise infrastructure, such as AWS EC2 and AWS RDS (see Figs. 3.5a, 3.5c, and 3.5e). This is attributed to the network latency between the two application tiers. However, due to the fact that the database server is being executed off-premise, most of the resources previously hosting the complete MediaWiki stack are released.

Figures 3.5b, 3.5d, and 3.5f, on the other hand, correspond to the migration of the complete MediaWiki application stack off-premise (both front- and back-end tiers). These scenarios first evaluate the deployment of the MediaWiki data tier in the same VM as its presentation and business logic layers, and secondly evaluates the deployment of the data tier in independent AWS EC2 and RDS VM and database instances, respectively. The latency experienced by the application's end-user is in average 6.7% reduced w.r.t. the scenarios where the front-end tier is hosted on-premise and the back-end tier off-premise. Such decrease is due to the existence of a lower overall network latency.

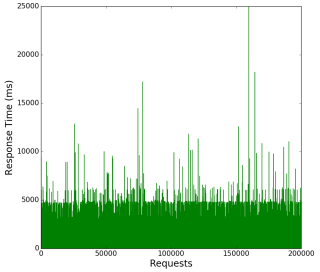
[10]Load Driver Profile & Sample - Bitbucket Git Repository:
`http://bitbucket.org/sgomezsaez/mediawiki_load_driver`



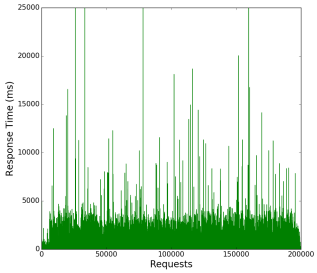
(a) single on-premise VM



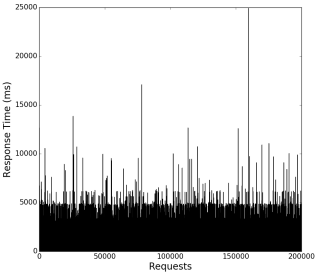
(b) single AWS EC2 *m3.xlarge* VM



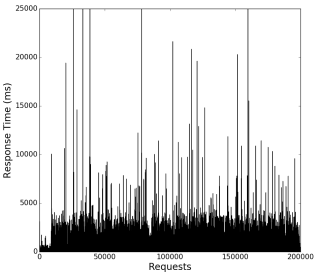
(c) front-end tier on-premise and back-end tier on an AWS EC2 *t2.medium* VM



(d) front- and back-end tiers on two separate AWS EC2 *t2.medium* VMs



(e) front-end tier on an on-premise VM, and back-end tier on AWS RDS *m3.large*



(f) front-end tier on an AWS EC2 *t2.medium* VM, and back-end tier on AWS RDS *m3.large*

Figure 3.5: Observed Latency - On- vs. Off-premise Deployment of MediaWiki's Front- and Back-end Tiers.

When comparing both scenarios and focusing on using specialized services, such as DBaaS, experiments show that its usage helps in improving the performance when deploying the application database off-premise. For instance, for the first scenario (on-premise deployment of the MediaWiki front-end tier), the performance improvement is approximately 1.79% while in the second scenario (off-premise deployment of the MediaWiki front-end tier) is 6.78%.

3.5 Experiment 3 (Exp.3): Scientific Workflow Simulation Application

The third experiment focuses on evaluating performance variation using a Scientific Workflow Simulation application. This experiments are part of the conference paper and book chapter publications [GAH+15] and [GAH+16], respectively.

Simulation workflows, a well-known topic in the field of eScience, describe the automated and flexible execution of simulation-based experiments. Common characteristics among simulation workflows are that they are long-running as well as executed in an irregular manner. However, during their execution, a wide amount of resources are typically provisioned, consumed, and released on-demand. The inherent characteristics of simulation workflows make them ideal candidates to be migrated to the cloud.

Scientific workflows need complex middleware systems, known as Scientific Workflow Management System (SWfMS). A SWfMS comprises a workflow engine, a messaging system, several databases, an auditing system, and an application server running simulation services. The workflow engine provides an execution environment for the workflows. The messaging system serves as communication layer

between the modeling and monitoring tool, the workflow engine, and the auditing system. The auditing system stores workflow execution for analytical and provenance purposes [SK10].

3.5.1 The OPAL Simulation Workflow

This experiment uses the Opal Simulation Application, as defined in [SK10]. The OPAL Simulation Application consists of a set of services which are controlled and orchestrated through a main OPAL workflow (the *Opal Main* process depicted in Fig. 3.6). The simulation services are implemented as Web services and divided into two main categories: (i) resource management, e.g. distributing the workload among different servers, and (ii) simulation services, i.e., as discussed in [BS03; MBHS10].

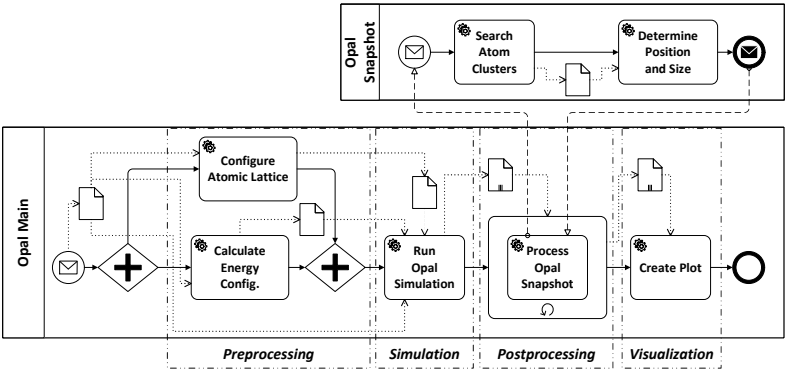


Figure 3.6: Simplified View of Simulation Workflows Constituting the OPAL Simulation Environment [SK10]

The main workflow can be divided into four phases, as shown in Fig. 3.6: preprocessing, simulation, post-processing, and visualization.

During the preprocessing phase, all data needed for the simulation is prepared. In the simulation phase, the workflow starts the simulation by invoking a corresponding Web service. In regular intervals, the Opal simulation creates intermediate results (snapshots). For each of these snapshots, the main workflow initiates post-processing tasks, which are realized in a separate workflow (see *Opal Snapshot* process in Fig. 3.6). When the simulation is finished and all intermediate results are post-processed, results of the simulation can be visualized.

3.5.2 Experimental Methodology & Setup

The OPAL environment is basically composed by two main systems: the SimTech SWfMS [SK10; SHK+11], and a set of Web services bundling resource management and Kinetic Monte Carlo (KMC) simulation tasks. The main goal is to evaluate the performance of the OPAL application and environment, by means of utilizing optimized on- and off-premise VM-based cloud services to host the complete OPAL application stack.

Table 3.2 depicts different IaaS VM instances used for the experiments. In particular, this experiment evaluates different types of instances and configurations, such as *Micro Instances*, *General Purpose Instances*, *Compute Optimized Instances*, and *Memory Optimized* VM instances. These vary in terms of their performance, as denoted by the specifications given by their cloud providers.

In scenarios comprising on-premise deployments, this experiment uses an IBM System x3755 M3 server[11] with an AMD Opteron Processor 6134 exposing 16 CPU of speed 2.30 GHz and 65GB RAM. In all

[11] IBM System x3755 M3: http://www-03.ibm.com/systems/xbc/cog/x3755m3_7164/x3755m3_7164aag.html

Table 3.2: IaaS Ubuntu Linux On-demand Instances Categories per Provider (in January 2015).

Instance Category	Cloud Provider	Instance Type	vCPU	Memory (GB)
Micro	on-premise	micro	1	1
	AWS EC2	t2.micro	1	1
	Windows Azure	A1	1	1.75
	Rackspace	General 1	1	1
General Purpose	on-premise	large	2	4
	AWS EC2	m3.large	2	7.5
	Windows Azure	A2	2	3.5
	Rackspace	General 2	2	2
Compute Optimized	on-premise	compute3.large	4	4
	AWS EC2	c3.large	2	3.75
	Windows Azure	D2	2	7
	Rackspace	Compute 1-3.75	2	3.75
Memory Optimized	on-premise	memory4.large	2	15
	AWS EC2	r3.large	2	15.25
	Windows Azure	D3	4	14
	Rackspace	Memory 1-15	2	15

scenarios, middleware components are deployed on an Ubuntu server 14.04 LTS with 60% of the total OS memory dedicated to the SWfMS.

In terms of the workload, it comprises 10 concurrent users, each of them sequentially sending 10 random and uniformly distributed simulation requests. This load aims at emulating a shared utilization

of the simulation infrastructure. Apache JMeter 2.9 is used as the load driver. Due to the asynchronous nature of the OPAL workflow, a custom plugin in JMeter was realized towards receiving and correlating the asynchronous simulation responses. The experiment measures the latency perceived by end users (scientists) in milliseconds (ms). Towards minimizing network latency, the load driver is deployed in all scenarios on a VM located in the same region as the simulation environment.

3.5.3 Experimental Results & Findings

Figure 3.7 shows the average latency for the different VM categories depicted in Table 3.2. The scenarios using *Micro* instances have been excluded from the comparison due to the impossibility to finalize the execution. More specifically, the on-premise micro-instance was capable of stably running approximately 80 requests, while in the off-premise scenarios the system was saturated with approximately 10 requests. For the scenario utilizing Rackspace, the VM micro instance was saturated immediately after sending the first set of 10 concurrent simulation requests.

With respect to the scenarios using the instances *General Purpose*, *Compute Optimized*, and *Memory Optimized*, the following performance variations are observed:

1. The on-premise scenario shows in average a latency of approximately 320K ms over all categories, a 40% on average higher than the perceived latency in the off-premise scenarios.
2. However, the performance is not constantly improved when migrating the simulation environment off-premise. For example, the *General Purpose* Windows Azure VM instance shows a

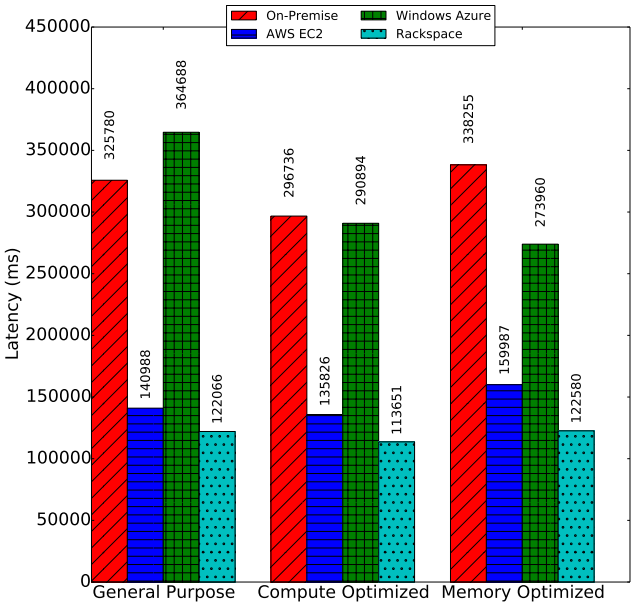


Figure 3.7: Average Latency per Provider & VM Category [GAH+15].

degraded performance of 11%, while the Windows Azure *Compute Optimize* VM instance shows only a slightly performance improvement of 2%, when compared with the on-premise scenario.

3. When migrating the complete simulation environment to the Cloud, the performance improves by approximately 56% and 62% for the AWS EC2 and Rackspace *General Purpose* VM instances, respectively,

4. 54%, 2%, and 61% for the AWS EC2, Windows Azure, and Rackspace *Compute Optimized* VM instances, respectively, and

5. 52%, 19%, and 63% for the AWS EC2, Windows Azure, and Rackspace *Memory Optimized* VM instances, respectively.

When comparing the average performance improvement among different optimized VM instances, the *Compute Optimized* and *Memory Optimized* instances enhance the performance by 12% and 6%, respectively.

3.6 Overview - Experimental Findings

The previous experiments verified the hypotheses depicted in Sec. 3.2. In particular, these used different types of applications as the basis, in order to evaluate the intrinsic performance characteristics among cloud services.

Focusing first on comparing characteristics among utilized workloads:

- the TPC-H workload used in *Exp.1* stresses the application's data tier, while on the other hand,
- the Wikipedia workload used in *Exp.2* groups data transfer requests consisting of retrieving, modifying, and storing Wikipedia articles.
- The OPAL simulation workflow in *Exp.3* is a special kind of workload, since it combines both computation and data intensive characteristics. However, its concurrency, i.e., scientists performing simulations at the same time, is considerably lower than in the previous workloads.

Although *Exp.1* specifically focuses on migrating the application data tier, experimental results already reveal most of performance

challenges discussed in Sec. 3.2. More specifically, *Exp.1* verifies the hypotheses *H1*, *H2*, and *H5*. *Exp.1* demonstrated the existence of a variability in performance among different AWS offerings, such as EC2 and RDS, showing an increased performance volatility when using AWS EC2. Focusing on seasonal trends, *Exp.1* detects a seasonal trend in off-premise scenarios showing a performance improvement when approximating to weekend days. In an opposite manner, on-premise scenarios behave in an inverse manner. *H5* is verified when migrating the application data to a DBaaS offering. In particular, a significant performance increase is observed when using AWS RDS to host the application's data tier.

Exp.2 focuses on distributing application components, by means of leveraging different types of cloud services. This experiment verifies *H4* when migrating only the database tier to an off-premise environment. In particular, the performance is degraded by approximately 300% when migrating the database to AWS EC2 or RDS. However, when migrating the whole application stack to AWS, such behavior is overturned, and hypotheses *H2* and *H3* are verified. In particular, we can observe an increase of performance of approximately 7% w.r.t. the on-premise deployment scenarios. More specifically, this happens when utilizing specialized services, such as AWS EC2 and RDS to host the application's business logic and data, respectively

Lastly, *Exp.3* contributes to verifying hypotheses *H2* and *H5*. The migration of the OPAL simulation environment to the cloud shows an impact on the overall system's performance, which is beneficial or detrimental, depending on the chosen VM provider and offering category. A detrimental performance is observed when selecting *Micro* VM instances, as these did not offer a minimum capacity for the OPAL application. Focusing on the remaining scenarios, the majority of

selected off-premise IaaS services improve the performance of the simulation environment. However, the *General Purpose* instances showed a performance degradation when compared to other IaaS instance types, such as *Compute Optimized*. In particular, compute optimized instances show the best performance. Such behavior is in line with the compute intensity nature of the OPAL simulation environment.

Chapter 4

SCARF - Systematic Cloud Application (Re)Distribution Framework

4.1 Chapter Introduction

The migration of traditional applications to the cloud entails several engineering and re-engineering challenges for application architects, as previously discussed in Chapter 2. In particular, traditional software engineering methods cannot be completely reused without any modification, as these were not originally built with cloud concerns [JAP13].

Up until this point, this work analyzed performance variability aspects among cloud services and providers. From such a point, this work raised the challenge for identifying the necessary ingredients, tasks, and abstraction levels, to support the partial and complete modeling of cloud applications, considering their performance characteristics. The objective of this chapter is to structure such ingredients and to build a software development process aimed at distributing and redistributing cloud applications.

This chapter establishes the foundations of this work. In particular, it introduces a Systematic Cloud Application (Re)Distribution Framework (SCARF), a generic framework used as the basis in this work and geared towards the systematic development and management of distributed cloud applications. SCARF eases the migration and re-engineering of traditional applications, by means of providing the necessary artifacts to design, provision, and manage distributed cloud applications. Moreover, SCARF is based on distributing and redistributing cloud application components, focusing on operational and business performance demands.

SCARF is comprised of two main pillars: (i) a life-cycle for the design of performance- and cost-aware cloud applications, and (ii) a Systematic Cloud Application (Re)Distribution Method (SCARM), a set of manual and automatic tasks supporting the SCARF life-cycle to be performed by application and business architects. Both fundamental elements of SCARF have been discussed in [GALS14a] and [GAWM14].

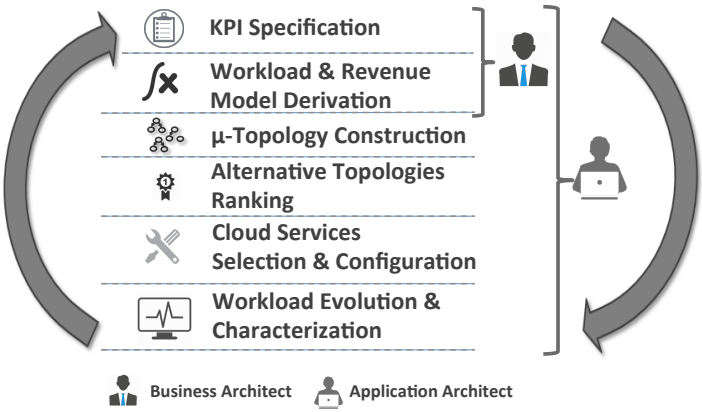


Figure 4.1: SCARF - Life-Cycle

4.2 SCARF - Life-Cycle

Chappel remarked in [Cha08] the complexity and necessity for defining an Application Lifecycle Management (ALM) for IT organizations, due to the existence of different perspectives. The generic view of an ALM introduced in [Cha08] takes a broad perspective and divides it into three areas: governance, development, and operations. These cover the complete set of events demarcating an application's life cycle. This work folds into the *operations* area – the efforts required to run and manage applications.

The first ingredient in SCARF consists of the SCARF Life-Cycle and is depicted in Fig. 4.1. The SCARF life-cycle entails a set of six phases for operating cloud applications, by means of distributing and redistributing their components, while focusing on optimizing performance and cost. Moreover, the life-cycle requires interaction

and collaboration of application and business architects, as identified in the scope of this work (see Sec. 1.2).

Focusing on the phases depicted in Fig. 4.1, the first phase consists of *KPI Specification*, by means of defining and specifying an application's business and operational requirements. Application architects are responsible for defining functional and non-functional operational requirements, while business architects focus on defining business requirements and constraints. Subsequently, application and business architects are required to bring together their knowledge in order to derive a first *Workload & Revenue Model*. A workload model comprises a description of potential application workload behavioral models, while the revenue model includes prediction of revenues for the application.

In the *μ-topology Construction* phase, an application's μ-topology is built, as discussed in Chapter 5. Informally, a μ-topology is a graph-based architectural representation of an application, which contains all possible distributions of its components. A μ-topology comprises different alternative sub-topologies, which can be used to provision and deploy a complete application stack in a distributed manner. A formal definition of μ-topologies is provided in Chapter 6. Until this phase, the μ-topology is, however, exclusively built based on functional aspects and requirements of applications. The definition of an application profile, which contains non-functional characteristics and requirements, serves as the basis in the *Alternative Topologies Ranking* phase, which generates every alternative topology and ranks them based on their expected utility, i.e., the expected gain of a concrete application distribution. The utility-based evaluation of alternative topologies is discussed further in Chapter 8, and consists of using utility functions to evaluate the trade-off between performance

and cost of cloud applications distributions, represented by their corresponding viable topologies. The actual generation of alternative topologies is discussed in more depth in Chapter 7.

Subsequently to analyzing different scenarios for distributing a cloud application – represented as viable topologies of an application, cloud resources can be selected and configured in the *Selection & Configuration* phase. Once the application is deployed, i.e., during its production phase, the *Workload Evolution & Characterization* phase consists of retrieving performance data and analyzing workload evolution using, for instance, monitoring techniques.

4.3 SCARM - Systematic Cloud Application (Re)Distribution Method

The second pillar of SCARF consists of a Systematic Cloud Application Redistribution Method (SCARM), a method for efficiently distributing cloud applications. SCARM fosters the collaboration among application architects, and focuses on optimizing the trade-off between cost and performance when operating cloud applications. For this, SCARM targets the analysis and processing of application operational and business requirements, with a strong focus on studying infrastructure's performance and application's workload behavioral characteristics [GALS14a].

SCARM strongly focuses on analyzing and optimally solving the multi-dimensional problem identified in Sec. 2.3, by organizing tasks and tools to systematically analyze and revise the distribution of applications in the cloud w.r.t. their cost and performance. SCARM supports the SCARF life-cycle and allows application architects to distribute and redistribute cloud applications to cope with changing

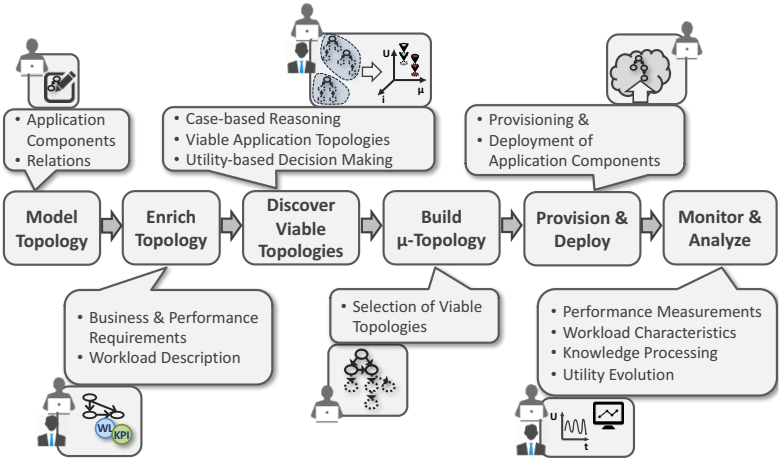


Figure 4.2: SCARF: Systematic Cloud Application (Re)Distribution Method (SCARM)

business and operational requirements, and fluctuating workloads.

The remainder of this section details SCARM, depicted in Fig. 4.2, which consists of the following tasks: (i) *Model Topology*, (ii) *Enrich Topology*, (iii) *Discover Viable Topologies*, (iv) *Build μ-Topology*, (v) *Provision & Deploy*, and (vi) *Monitor & Analyze*.

i. Model Topology The modeling of application topologies is mainly a responsibility of application architects. Application architects define application topology models using a modeling environment, as defined in Chapter 2, and represent application functional and non-functional aspects, such as the relationships among components, their dependencies, security and performance constraints, etc.

As previously discussed in Chapter 2, migrating applications to the

cloud requires as a first step to represent an application's architecture using cloud application topology models. Cloud application topologies are defined using domain or provider specific languages. For instance, cloud application topologies can be specified using generic languages, such as GENTL [ARSL14], or using technology specific languages, such as TOSCA [BBKL14a], Blueprints [PvdH11], AWS CloudFormation[1], or CAMP[2]. SCARM is not bound to a concrete topology language, and is therefore proposed as a technology agnostic method.

There are two modeling possibilities for application architects for defining application topologies in SCARM: (i) modeling a complete application topology, by means of defining its components and their relations. The second modeling possibility consists of (ii) partially modeling the application stack, by means of defining a sub-topology for components that are specific to the application. This application-specific sub-topology is informally referenced in the remainder of this chapter as an α-topology. Chapter 6 formally defines alpha-topologies.

ii. Enrich Topology The enrichment task in SCARM allows application architects to optionally enrich an application specific topology, i.e., an α-topology, with operational and business performance requirements, as detailed in Chapter 5. The enrichment can be driven using the same modeling environment as in the modeling step.

The enriched topology model plays a fundamental role when discovering and generating viable topologies. In particular, an enriched topology model contains knowledge for scoping the domain of compatible cloud services, and can determine the size of μ-topologies.

[1] AWS CloudFormation: https://aws.amazon.com/cloudformation/

[2] OASIS CAMP: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=camp

iv. Discovery & Evaluation An enriched topology model essentially constitutes an application profile, which represents a new case for distributing a cloud application in SCARM. An application profile contains architectural characteristics, performance requirements, and workload knowledge built upon the information provided during the modeling and enrichment steps. This profile is used in this step for discovering and evaluating viable application topologies.

The discovery of viable application topologies is driven in two substeps, targeting first a (i) *architectural-based discovery*, and followed by (ii) a *non-functional based similarity analysis*. Both steps analyze and infer on μ—topology graphs of similar applications. This allows for the generation of viable topologies using previous knowledge extracted from similar applications. However, the generated set of viable topologies can be too extensive.

A subsequent non-functional based analysis consists of exploiting previous knowledge and reasoning on previous cases, i.e., using similarity analysis on past application distributions. The similarity analysis limits the number of viable application topologies to be evaluated. Once the architectural and non-functional analyses take place, SCARM uses utility theory to calculate the utility of each viable topology (see Chapter 8). This calculation is used as the basis for constructing a ranked list of viable distributions, where each viable topology is linked to an expected utility value. This list can be then analyzed by application architects, who are then responsible for manually or automatically selecting viable topology models to construct their application's μ-topology.

v. Provision & Deploy Subsequently to deciding for viable topologies, application architects can select a viable topology to be instan-

tiated. In that case, the deployment of the application takes place. This consists of instantiating the selected viable topology model, by means of automatically provisioning cloud resources and deploying the application. SCARM is not bound to a concrete technology for dynamically provisioning an application stack. The deployment can be done, e.g., using the OpenTOSCA Container[3].

vi. Monitor & Analyze During the production phase of an application, it is necessary to capture its performance knowledge. More specifically, it is crucial to monitor and capture metrics relevant to specified performance and operational requirements, as well as behavioral characteristics of the application's workload.

The analysis of performance evolution is a critical step in SCARM. Application architects must analyze and decide for (i) redistributing application components, (ii) re-configuring cloud resources, or (iii) adapting initial operation and business performance demands, e.g., refining performance demands previously specified in the KPIs & WL analysis step.

The application's performance evolution is analyzed in SCARM by means of measuring the evolution of utility for the selected viable topologies. For instance, a continuous decrease of utility may indicate the need for redistributing an application.

4.4 Discussion & Roadmap

The SCARF framework provides artifacts for re-engineering and migrating traditional applications to cloud environments, focusing on

[3] OpenTOSCA Container:
http://www.iaas.uni-stuttgart.de/OpenTOSCA/

optimizing cost and performance. This optimization is based on evaluating cloud services that can be used to span application components, and based on using past knowledge from similar cases as the basis.

Although there are approaches that focus on optimizing software architectures, these present the following disadvantages: (i) they are not ready for the cloud, (ii) they either focus specifically on functional aspects or on one non-functional aspect, (iii) or they focus on migrating a specific application layer. Under the umbrella of software optimization methods not ready for the cloud, Aleti et al. review the most relevant engineering methods and group them w.r.t. which non-functional aspect the authors try to optimize [ABG+13a]. However, this categorization is meant for traditional software systems, without considering the possibility to cloudify application components.

Optimization of cloud application architectures w.r.t. functional aspects are, for example, tackled in [SBB+16] and [INS+14]. TOSCA-MART [SBB+16] fosters reusability of application topologies by matching functional requirements with capabilities among sub-topologies, while MADCAT [INS+14] proposes a methodology for iteratively refining cloud native architectures based on the refinement of stakeholders' requirements. Focusing on the application's data layer, Strauch et al. [SAK+14] target different problem dimensions when migrating the application database layer to the cloud. SCARF goes a step further in the state-of-the-art, as it consolidates the analysis and optimization of (i) functional and non-functional application's aspects, and (ii) offers support for migrating and distributing the complete application stack to private and public cloud environments.

Since SCARM is the engineering method of SCARF, the remainder chapters are realized and structured according to the different tasks of SCARM. In particular, Chapter 5 presents an enhancement model

for cloud application topologies and identifies necessary performance knowledge for the modeling and enrichment tasks in SCARM. Chapter 6 goes a step further in SCARM, and presents a formal model for defining viable and reusable application topology models. Chapter 7 introduces an automatic mechanism for discovering and constructing μ—topologies based on analyzing similarity among application architectures and non-functional requirements. A formal utility model for evaluating viable topologies is presented in Chapter 8.

Since this work focuses on assisting application architects in decision making tasks for migrating their applications to the cloud, deployment, production, and monitoring tasks are not conceptually developed in the remainder of this document. More specifically, SCARF focuses on exploiting existing concepts and technologies for the deployment, production, and monitoring tasks. Further technological details for these tasks are provided as part of the technological framework of SCARF, called SCARF-T, which is discussed in Chapter 9.

Chapter 5

Enhanced Cloud Application Topology Model

5.1 Chapter Introduction

Section 2.2.3 discussed using application topologies to model the architecture of cloud applications. Application topologies define application components, and the relationships among them. Existing cloud application topology definition approaches, such as TOSCA [BBKL14a], Cloud Blueprints [PvdH11], or CloudML [BM12], provide modeling and language constructs to represent architectural and non-functional characteristics of cloud applications.

For non-functional characteristics, current topology modeling approaches, however, provide coarse grained constructs that must be extended to cover the desired set of non-functional requirements (see Sec. 2.2.3 for more details). Moreover, these generic constructs are tightly bound to specific technologies. For instance, Policy4TOSCA is a policy language for TOSCA-based cloud applications [WWB+13]. Policy4TOSCA provides a policy definition construct, which can be attached to TOSCA service templates, which must be extended and customized for each application and requirement separately. There exists therefore a necessity for going a step further in the domain of specifying non-functional requirements related to performance. In particular, this work identifies the necessity for (i) identifying the necessary performance knowledge capturing the performance of applications in the cloud, and for (ii) establishing modeling and language constructs to define application performance information during the design phase and to capture it during the runtime phase, such as performance requirements, metrics, and the evolution workload behavioral characteristics.

The usage of an enhanced cloud application model in SCARF enables application architects to specify during application's design phase their performance requirements. This information is processed in SCARF in two steps: (i) during the discovery and construction of viable topologies, and (ii) during the knowledge capturing of application performance in the production phase. This knowledge can be used to optimize the performance of future application distributions.

The remainder of this chapter firstly focuses on identifying the necessary application performance knowledge to capture performance and workload characteristics. This knowledge can be leveraged throughout design and runtime phases of applications. Subsequently,

this chapter defines a set of fine-grained modeling constructs, which can be used to enhance application topology models with performance and cost characteristics and demands. The presented modeling constructs are part of research contributions presented in [GAL16].

5.2 Characterization of Performance Knowledge

The previous experiments in Ch. 3 showed variability in performance among cloud providers and services. Towards efficiently analyzing and evaluating an application's performance under different cloud distribution alternatives, its variability must be observed, captured, and analyzed using significant attributes during design and production phases [SDQ10; LML+11].

The remainder of this section identifies necessary attributes that build the performance knowledge in SCARM. In particular, it firstly focuses on identifying and characterizing performance metrics, which can be used to analyze performance fluctuation. Secondly, it puts emphasis on the application's workload, by means of identifying the main ingredients to describe and capture workload behavioral characteristics. Both identification and characterization of relevant performance metrics and workload attributes have been driven as part of the research works [Gan15] and [Pin16]. More specifically, these conducted a literature survey focusing on:

- identifying common performance metrics among different types of applications, such as *Enterprise Applications*, *e-Science Applications*, *Internet of Things (IoT) Applications*, and *Cloud Applications*,
- identifying common KPIs for cloud services, and

- identifying metrics and patterns that can be used to describe workload characteristics of cloud applications.

The remainder of this section presents identified common metrics and workload attributes, and groups them into different categories. SCARM is not limited to these categorization, as these can be extended with further performance and worload metrics.

5.2.1 Identification of Relevant Performance Metrics

Sec. 2.4 presented some fundamentals related to performance modeling and engineering of applications. A fundamental success factor for organizations is the definition of efficient KPIs. As a recap from Sec. 2.4, Parmenter identifies KPIs as the means for establishing, analyzing, and evaluating the success of an organization [Par15].

In the IT domain, KPIs are associated with metrics used to quantitatively analyze different aspects of software development and production. Focusing on the application's production phase, the focus is typically on analyzing the end-user perception and the utilization of resources. Table 5.1 summarizes a characterization of relevant metrics for cloud applications. In particular, these metrics build the performance knowledge in SCARF for cloud application topology models. In addition, these are leveraged in the cloud application distribution decision making process presented in Chapters 6 and 7.

Focusing on Table 5.1, four categories build the performance knowledge in SCARM: (i) *Resource Capacity*, comprising metrics that are relevant or highly influence the application's end-user perception, (ii) *Resource Utilization*, grouping typical resource-related metrics, (iii) *Elasticity*, depicting cluster-based metrics, and (iv) *Availability*, comprising common infrastructure availability parameters.

Table 5.1: Cloud Application Performance Metrics (adapted from [Pin16])

Metric Category	Metric	Common Unit
Resource Capacity	Response Time	ms
	Throughput	Req./s
	Read / Write Speed	Rev./s
	Latency	ms
	Network Bandwidth	Mb/s
	Number of VMs	$\mathbb{Z}^+$
	I/O Operations	$\mathbb{Z}^+$
Resource Utilization	Network Utilization	pct
	Memory Utilization	pct
	Disk Utilization	pct
	CPU Utilization	pct
	VMs Utilization	pct
Elasticity	Bandwidth	Mb/s
	Resource Acquisition Time	s
	Resource Provisioning Time	s
	VM Startup Time	s
	Deployment Time	s
	Resource Release Time	s
Availability	Service Uptime	pct
	Resource Uptime	pct
	Mean Time Between Failures	h
	Mean Time to Repair	h

5.2.2 Identification of Relevant Application Workload Attributes

The identification and modeling of application workload characteristics is identified in Sec. 2.4 as a challenging task, due to the complexity and variability of workloads. However, the construction of workload models by means of capturing or specifying its characteristics can be highly beneficial when deciding among viable distributions of a cloud application. In particular, the remainder of this section focuses on identifying a minimal set of workload attributes that build an application's workload knowledge, which can be leveraged during decision making tasks in SCARM.

Table 5.2 summarizes workload attributes that build the workload knowledge for cloud applications: (i) *Seasonal Pattern*, (ii) *User Arrival Rate Distribution*, (iii) *Behavioral Model*, (iv) *Average Number of Users*, (v) *Average Number of Transactions*, and (vi) *Time Interval*.

The identified seasonal patterns are inline with Fehling's cloud computing workload patterns in [FLR+14]. The representation of the user arrival rate is typically done through statistical distributions [Fei15]. The most typical distribution in Web-based systems is, for example, the Poisson distribution [BM11]. The behavioral model essentially captures the probability distribution of a workload. Finally, the average number of users, average number of transactions in the system, and the time interval represent statistical indexes of a workload.

The workload knowledge can be retrieved from two sources, as discussed in [GAL16]. A top-down approach directly involves application architects in defining expected workload characteristics, while a bottom-up approach consists of deriving models from production data, such as server logs. The exploitation of workload knowledge is further discussed in Chapter 7.

Table 5.2: Cloud Application Workload Attributes (adapted from [Pin16])

Workload Attribute	Possible Values
Seasonal Pattern	Continuously Changing Once-in-a-lifetime Periodic Static Unpredictable
User Arrival Rate Distribution	Normal Poisson Logarithmic Gamma Uniform
Behavioral Model	Normal Poisson Logarithmic Gamma Uniform
Average Number of Users	$\mathbb{Z}^+$
Average Number of Transactions	$\mathbb{Z}^+$
Time Interval	$\mathbb{Z}^+$

5.3 Running Example

The remainder of this chapter uses as running example the Web shop application topology depicted in Fig. 5.1. The Web shop application is a two-tiered application, comprising a PHP front-end and a MySQL

database in its back-end. The topology is represented in a technology-agnostic manner, as discussed in Section 2.2.3, i.e., using common graph notation.

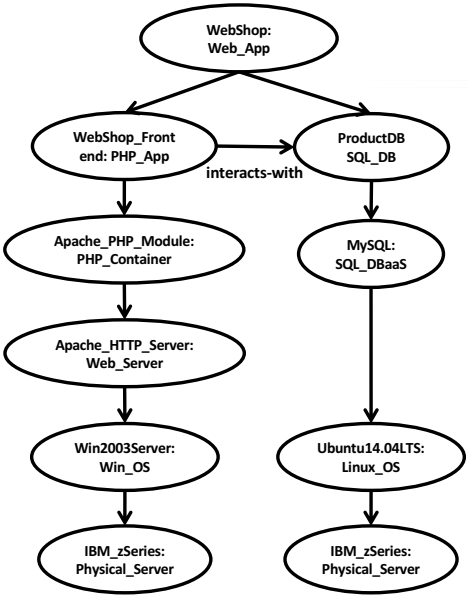


Figure 5.1: Web Shop Application - Running Example

Both front- and back-end tiers are initially hosted in separate on-premise physical servers, and the objective tackled in this chapter relates to creating a performance aware topology model that can be used as the basis by application architects in SCARF to migrate this application to the cloud.

5.4 Enriched Application Topology Model

A major gap identified in this work is the lack of support in topology specification approaches for defining performance and cost indicators for cloud applications. Since cloud application topology models basically represent architectural characteristics of cloud applications, this work opts to build atop of them. More specifically, the remainder of this section defines a generic enrichment mechanism using the GENTL language proposed in [ARSL14] as the basis.

GENTL is a generic application topology language that can be used as an abstraction from specific topology languages and technologies, as it provides mappings to and from other topology languages, such as TOSCA or cloud blueprints, as discussed in Sec. 2.2.3. GENTL is merely used in the remainder of this section for representing the enriched application topology model.

Figure 5.2 summarizes both performance specific constructs and their relation with generic application topology constructs. In particular, Fig. 5.2 defines (i) generic modeling constructs for cloud application topologies, (ii) modeling constructs for defining and capturing performance and workload characteristics of cloud applications, and (iii) specifies potential enrichment points at different levels of the application topology.

Performance characteristics of applications basically consist of a set of *Performance Requirements*, which can be specified through performance indicators grouped within *Metric Categories*. Application workloads, which are defined as *Workload Profiles*, have an impact on performance metrics. The composition of the previous constructs is used as the basis to enrich cloud application topology models at its different levels, depicted as follows:

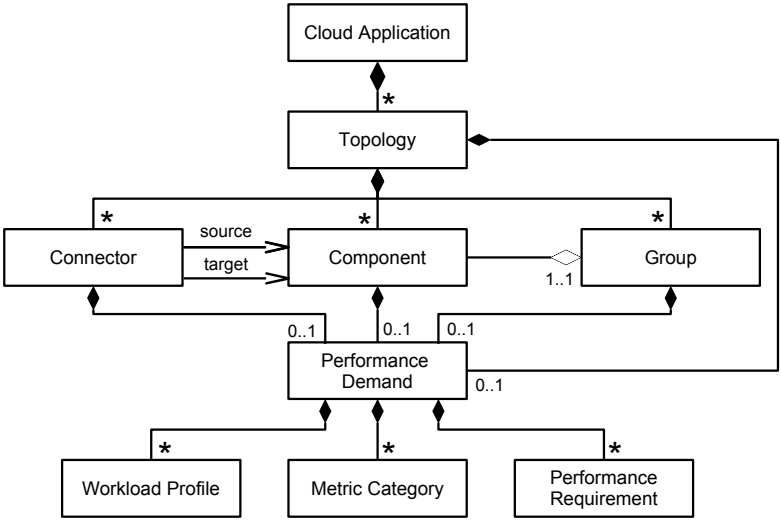


Figure 5.2: Application Topology Enrichment Meta-model for Performance Characteristics. Adapted from [GAL16].

- a fine-grained description describes performance characteristics on each topology *Component* or *Connector*, while a
- coarse-grained description enriches the application topology model in a high-level, aggregated as a *group* manner.

Running Example: Fig. 5.3 depicts the enhanced application topology for the Web shop application example. Considering performance requirements and workload profile as the fundamental input for enhancing topologies with performance information, these can be applied at different levels of the application topology. A fine-grained enrichment consists of defining performance information on each

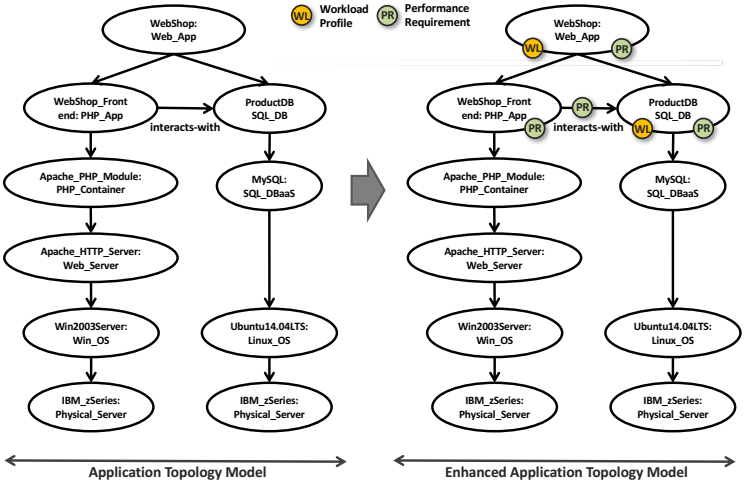


Figure 5.3: Enhanced Application Topology Example

application component. For the sample Web application, this includes both front- and back-end tiers individually. For example, performance requirements can be specific to the PHP front-end or the SQL back-end database. For the workload profile, on possibility is to define it exclusively for the back-end database, e.g., describing the distribution of SQL requests. Relationship constructs can also be enriched with performance demands, by means of enriching their *Connectors*. For instance, network latency demands among front- and back-end tiers can fold into this fine-grained level of description. On the other hand, a coarse-grained performance demand specification consists of defining both performance requirements and workload profile for the application as a whole, i.e., in the root node of the Web shop application topology.

The enhancement of cloud application topologies with performance information is leveraged during the design and production phases of applications. During design phase, it is used for automating decision making tasks in SCARM, such as the ones related to automating the discovery of viable topologies (see Chapter 4). During the production phase, the enhancement serves as the basis for monitoring the fulfillment of performance requirements, and for building performance knowledge for discovering other application distributions.

5.5 Performance Requirements

Business and operational requirements are typically defined as KPIs, e.g., maximum allowed latency, end-user satisfaction, etc. Figure 5.4 establishes design constructs for defining application performance indicators, together with their corresponding metrics.

According to the meta-model in Fig. 5.4, application *Performance Requirements* can be partitioned into two main groups (see Fig. 5.4): *Operational* and *Business*. Operational indicators relate to the technical operation of applications, such as availability, throughput, etc., while business indicators define business objectives, e.g. expected revenue per user, maximum expenditure, etc. Both can be quantitatively analyzed using *Metrics* (see Fig. 5.4).

For simplification purposes, *Metric Categories* are used to classify different metrics. For instance, there are metrics which focus on capacity or utilization of different types of resources, while others measure the infrastructure's availability. Metrics can be quantitatively analyzed by processing *Measure Samples*, which are samples taken from monitoring logs, e.g., server throughput, and are limited to a predefined time interval. In particular, measure samples consist of a

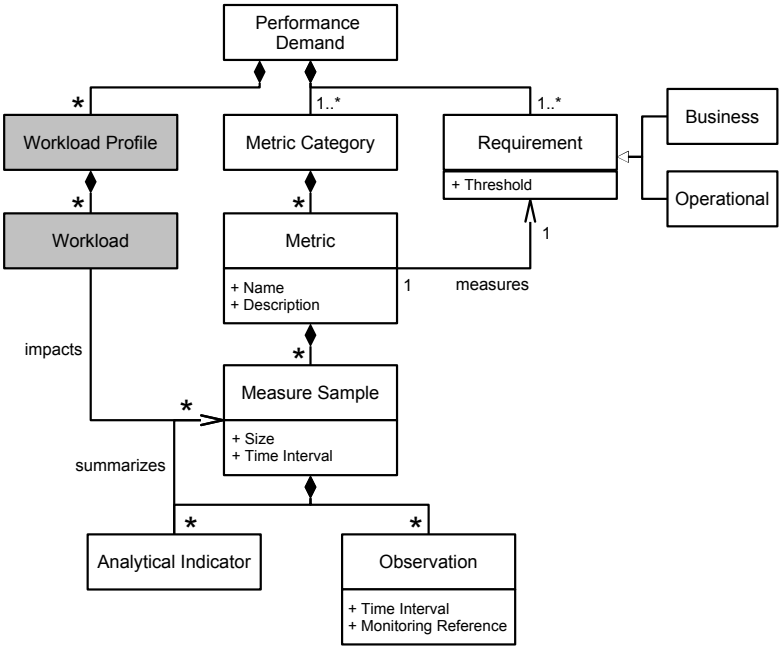


Figure 5.4: Performance Demand Evolution Meta-model for Cloud Applications [GAL16].

sequence of *Observations* taken over a period of time. The representation of a statistical summary of measure samples is supported in the proposed meta-model, by means of associating *Analytical Indicators* for each measure sample. As examples of analytical indexes, application architects can use standard deviation or mean indicators, among others, in order to analyze the existence of volatility or the existence of a central tendency in the fulfillment of application requirements.

The application performance oscillates due to the impact of applica-

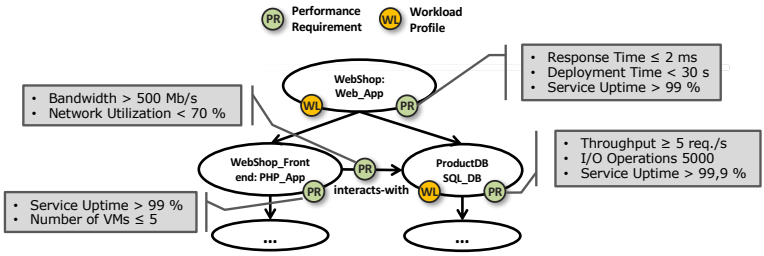


Figure 5.5: Enhanced Application Topology Example - Performance Requirements

tion workloads with different behavioral characteristics – represented as application workload profiles and discussed in the following section. In particular, such an oscillation is observed in each application measure sample, and corresponding to a concrete metric (see Fig. 5.4).

Running Example: Focusing on the Web shop application running example, business and operational performance requirements can be specified at the different levels of its topology. Fig. 5.5 provides sets of requirements for its front-end, back-end, and for the holistic application. Requirements are associated with their corresponding metrics (as previously defined in Table 5.1), and are specific to the application. For example, the Web shop front-end requires the usage of not more than 5 VMs, due to cost constraints, and its back-end expects a SQL server capable of handling more than 5 database req./s.

5.6 Workload Profile

The previous section highlighted a strong relation between application workloads and their impact on performance requirements. More specifically, workload fluctuations are reflected in measured samples, which are consequently analyzed in performance metrics. These are used to verify the fulfillment of performance requirements. This section analyzes and identifies the properties of application workloads, and defines the necessary modeling and knowledge constructs for application architects to specify and capture characteristics of application workloads.

Firstly, let's define for the scope of this work application workloads as the *description of a set of business transactions which are probabilistically distributed over a time interval, have an impact on the application state, and define the behavioral characteristics of its corresponding users* [ARSL14].

Figure 5.6 depicts a meta-model for describing cloud application workloads, which builds upon statistical descriptions, and simplifies the model presented in [VRH08]. An application *Workload Profile* consists of a set of *Workload* samples, each comprising an *Usage Profile*, a *Workload Mix*, a *Behavioral Model*, and a *Seasonal Pattern*. The workload *Usage Profile* describes the evolution of the application *Users* behavior, in terms of their *Arrival Rate* and the specification of concurrent or non-concurrent requests. Every *User* executes a set of *Transactions* that are part of a *Workload Mix*. A workload mix essentially contains the complete set of operations that users execute. The *Behavioral Model* represents the probabilistic distribution of the workload, i.e., the probability of the workload to occur. Finally, the *Seasonal Pattern* relates to the cloud computing workload patterns

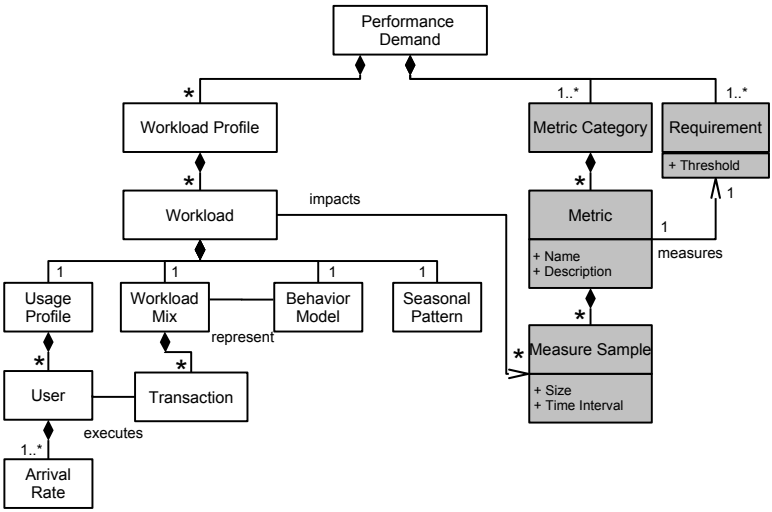


Figure 5.6: Workload Behavior Specification. Adapted from [GAL16].

defined in [FLR+14], e.g., periodic, static, once-in-a-lifetime, etc.

As previously denoted, workload characteristics can be either specified during the design of cloud applications, or can be captured during its execution. For the latter, it can be essentially done by (i) monitoring and capturing application or Application Programming Interface (API) logs, and by (ii) processing them in order to dynamically compute and derive characteristics of workload profiles.

Running Example Figure 5.7 exemplifies the specification of workload behaviors for the Web shop application example. In particular, the Web shop application's topology can be enhanced with workload information at two levels: (i) at the application as a whole, and (ii)

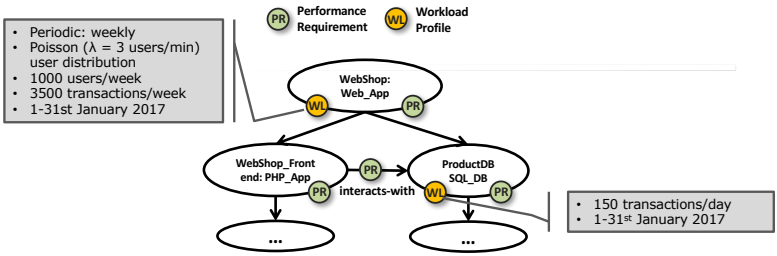


Figure 5.7: Enhanced Application Topology Example - Workload Behavioral Information

at a specific application tier. A workload specification for the complete application consists of describing the workload affecting all application tiers, i.e., front-end and back-end. In this case, workload periodicity, as well as intrinsic characteristics of workloads (users, arrival rate of users, etc.), have an influence on the overall application performance. However, there are workloads that only impact one application component, such as database workloads for the back-end tier in Fig. 5.7.

5.7 Discussion

Enhanced cloud application topology models can be leveraged in decision support tasks during both design and runtime phases. During design phase, application architects can define performance requirements that can be dynamically processed during decision making tasks. These tasks generate potential application distributions to fulfill such requirements. During runtime, performance knowledge can be captured and processed to optimize the decision making process.

The previously presented enhancement model allows the capturing of performance knowledge within application topology models. This knowledge can be exploited in SCARM to refine and optimize application topology models, as SCARM leverages and exploits performance knowledge among similar applications, using similarity-based analysis as the basis. However, the enhancement of application topology models depicted in this chapter only establishes the modeling and knowledge baseline in SCARM. In particular, SCARM's discovery and evaluation tasks require support for reusing application sub-topologies, which is not yet covered up to this point. Such a re-usability demand is covered in the *Viability of Cloud Application Topology* model introduced in Ch. 6, which complements the application topology enhancement model in SCARM.

Chapter 6

Formal Model for the Viability of Cloud Application Topologies

6.1 Chapter Introduction

Application topologies are the de facto specification approach for migrating applications to cloud environments [AGLW14; BBKL14a]. Chapter 2 presented different works that aim at technologically supporting design, provisioning, and production of cloud application topologies. However, such approaches lack of support for reusability or generation of feasible application topologies, based on architectural functional and non-functional constraints.

Towards this goal, Ch. 5 builds the support and establishes knowledge and modeling baselines for enhancing application topologies with performance information. This chapter complements the enhanced topology model, and presents a formal model for cloud application viable topologies, geared towards reusing and generating cloud application topologies. In particular, this formal model serves as the basis for building viable cloud application topologies in SCARM, which is based on a discovery mechanism developed in Chapter 7.

In a nutshell, this chapter establishes a formal technology-agnostic baseline for reusing cloud application topologies and for generating viable cloud applications topologies in the decision making process of SCARM. Although this works uses TOSCA as technology for defining cloud application topologies, this technology-agnostic formal model maps to the GENTL language proposed in [ARSL14]. GENTL supports the mapping of technology-agnostic cloud application topologies to concrete specifications, such as Blueprints or TOSCA. Exploring and proofing the mapping of this formal model to concrete specifications for cloud applications is out of scope in this work.

The remainder of this chapter aggregates research works contributed in [AGLW14], [Din16], and [Pin16]. In particular, the initiatives developed in this chapter make possible for application architects to explore their application's design space. Moreover, the formal model of viable cloud application topologies serves as the foundation for the decision support mechanisms presented in the following chapters.

6.2 Application Topology - Formal Definitions

Previous chapters introduced informal concepts for cloud application topologies. Let's first start by formally describing a cloud application

topology, using [AGLW14] for this purpose:

Definition 6.1 (Application Topology)
An application topology *is a directed acyclic labeled graph* $G = (N^L, E^L, s, t)$ *where* N *is a set of nodes,* E *is a set of edges,* L *a set of labels, and* s, t *the source and target functions* $s, t : E^L \rightarrow N^L$*. The topology graph is called* typed*, if the label set* L *contains only elements* <name:type> *(for nodes) and* <type> *(for edges), in which case the graph is denoted by* T*.*

This model is implicitly used by approaches like TOSCA [BBKL14a] or Cloud Blueprints [PvdH11], therefore representing one possible instance of an application topology. More specifically, each instance only depicts one possible distribution of an application, without covering the space of alternative distributions for the application components. Such a weakness is addressed by introducing the notion of *typed graph with inheritance* – as formally defined in [BED+03] and [dLBE+07] –, therefore enabling the exploration of application distribution possibilities, and defined as follows:

Definition 6.2 (Type Graph with Inheritance, as in [BED+03])
A type graph with inheritance TG_I *is a triple* (TG, I, A) *consisting of a type graph* $TG = (N, E, s, t)$ *(with a set of nodes* N*, a set of edges* E *and a target function* $s, t : E \rightarrow N$*), an inheritance graph* I *sharing the same set of nodes* N*, and a set* $N^A \subseteq N$*, called abstract nodes. For each node* $n \in I$ *the* inheritance clan *relation is defined by* $clan(n)_I = \{n' \in N \mid \exists \text{ path } n' \xrightarrow{*} n \in I\}$ *where* $n \in clan(n)_I$ *(i.e. the* path *of length 0 is included).*

The usage of a typed graph with inheritance, as in UML class diagrams, allows the usage of *abstract* nodes, as defined in [BED+03]

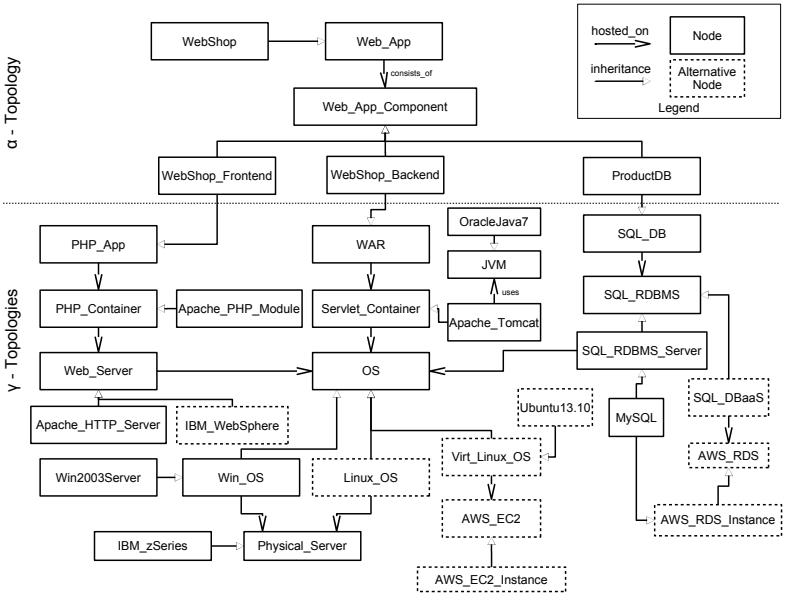


Figure 6.1: μ-topology of a three-tiered Web Shop Application. Adapted from [AGLW14]

and exemplified in Fig. 6.1, for types that have only inheritance relations with other nodes. Figure 6.1 is an extended version of the Web shop application introduced in Sec. 5.4. In particular, it is designed and realized as a three-tiered application. Abstract nodes, as depicted in Fig. 6.1, can be used to denote generic classes of nodes like, e.g., Web Server type for a Web server middleware component. The clan morphism relation $clan(n)_I$ is defined in [BED+03] and adopted in this work. The $clan(n)_I$ allows for navigating the inheritance-type edges in TG_I graphs, which is instrumental in producing typed graphs,

therefore producing multiple typed topology graphs. For instance, the 'Apache HTTP Server' and 'IBM WebSphere' nodes in Fig. 6.1 are results for the 'Web Server' node, while 'Win OS', 'Linux OS', and 'Virtual Linux OS', are results for the 'OS' node.

6.3 Viable Application Topologies

6.3.1 Formal Definitions

The concept of *viable or feasible application topologies* has been previously used to denote a possible distribution of applications. Building on the concept of clan morphism defined in [BED+03], a viable topology, according to [AGLW14], can be defined as follows:

Definition 6.3 (Viable Topology)
A typed topology T is viable *w.r.t. a type graph with inheritance TG_I, iff all elements of T are labeled (typed) over the elements of TG_I, i.e. there exists a graph homomorphism $m : TG_I \rightarrow T$ which uses the inheritance clan relation.*

Based on this definition, the viable topologies of the Web Shop application of Fig. 6.2 can therefore be classified as viable under the TG_I graph of Fig. 6.1. Focusing on the front-end component of the Webshop application in Fig. 6.2, it is hosted on a concrete instance of a γ-topology offering an 'Apache PHP Module' hosted on an 'Apache HTTP Server', which is hosted on a 'Windows 2003 Server' and an 'IBM zSeries' physical server. For hosting the Webshop back-end and database components, two possible γ-topology instances can be used for each, such as hosting the back-end in a virtual linux in an AWS EC2 instance, and hosting the database on an AWS RDS DBaaS instance.

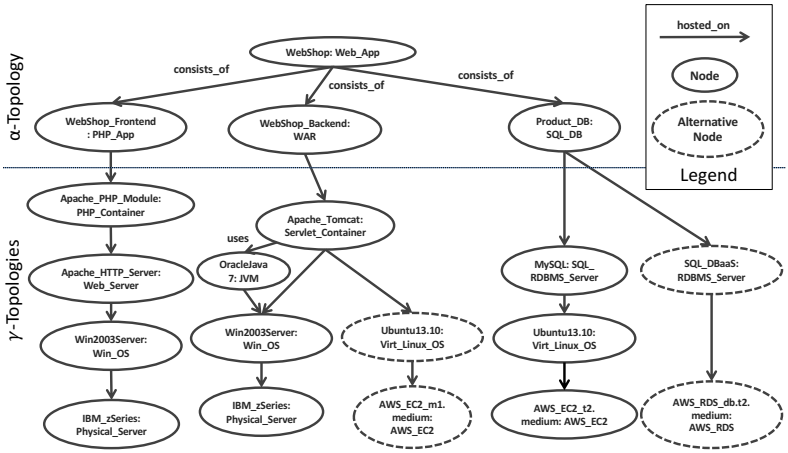


Figure 6.2: Sample three-tiered Web Shop Application Viable α- and γ-Topologies. Adapted from [AGLW14]

There are two design views of the morphism m that translates TG_I to T: *top-down*, with T being generated or validated against TG_I, and *bottom-up*, with TG_I being abstracted from one particular typed topology T and potentially being reused across different viable topologies. Both top-down and bottom-up design views help towards generating a simplified fragmented view of topologies, which is defined as follows:

Definition 6.4 (μ, α and γ-topology)
The typed graph with inheritance TG_I for a viable application topology T is called its μ-topology. *We denote by* α-topology *the sub-graph of a μ-topology that contains the components and relations that are specific to the application. A* γ-topology *is a sub-graph of a μ-topology that contains components or services that are not specific (and therefore reusable) to the application.*

Focusing on Fig. 6.1 and Fig. 6.2, the upper part depicts the Webshop application's α-topology, depicting application specific components, while the lower part depicts γ-topologies, each representing a reusable sub-topology. The application μ-topology is basically the graph representing all viable distribution of the application.

Definition 6.5 (ζ-topology)
A ζ-topology *is a sub-graph of a* μ*-topology, which encompasses an* α-topology *and exactly one* γ-topology *for each leaf node of the* α-topology.

A ζ-topology essentially represents a viable and deployable cloud application topology. Focusing on Fig. 6.2, a viable ζ-topology is depicted using solid lines for its corresponding nodes and edges.

6.3.2 Generation of Viable Application Topologies

So far, α-, γ-, and μ-topologies have been defined. However, the generation of viable topologies (defined in a μ-topology) has not yet been covered. The remainder of this section summarizes the generation of viable topologies – each denoted as a ζ-topology in the remainder of this work– from a constructed application's μ-topology.

Let's first define the set of viable ζ—topologies as $\mathcal{Z}$. Viable topologies are generated for a given application's α-topology, by means of first analyzing the space of existing γ-topologies, and by constructing a first μ-topology. γ-topologies are reusable sub-topologies depicting the *non application-specific* stack.

Using the generated μ-topology, a set of viable topologies can then be inferred by applying different morphisms $m^{(i)} : TG_I \rightarrow T^{(i)}, i \geq 1$ to it, resulting in different topologies $T^{(i)} \in \mathcal{Z}$. We assume, that there

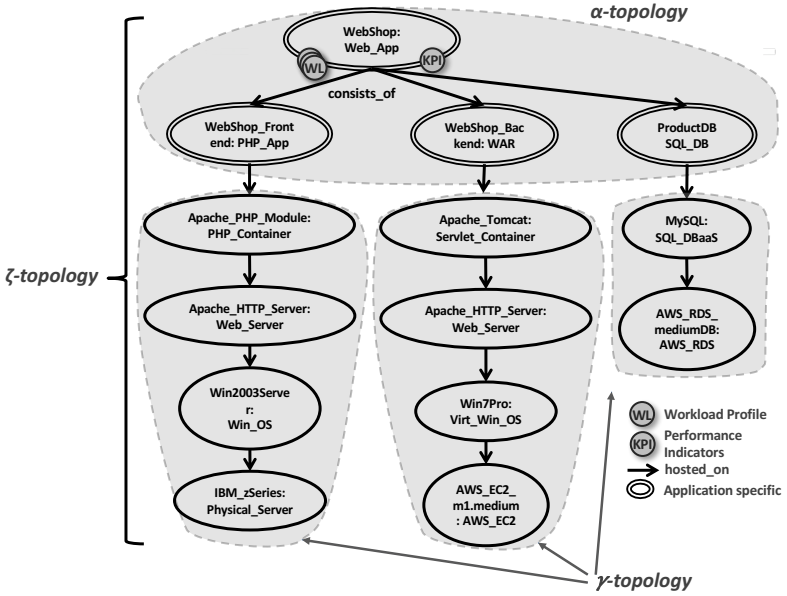


Figure 6.3: Web Shop Viable Application Topology (ζ-Topology)

always exists a viable topology for an application, i.e. $|\mathcal{Z}| \geq 1$. For instance, one possible application viable ζ—topology for the μ-topology represented in Fig. 6.1 is depicted in Fig. 6.3. The upper part of the topology (depicted with a double line) represents the application specific α-topology. The lower part of the topology groups a set of reusable γ-topologies, which have been encountered by applying the morphisms on the μ-topology.

6.4 Discussion

Previous sections established a formal baseline for reusing topologies, by introducing α-, γ-, and μ-topologies. α-topologies define application specific characteristics. These are used as the basis to discover and construct μ-topologies, which represent a set of viable application ζ-topologies. However, the model presents limitations w.r.t. how μ-topologies are built and viable topologies are discovered, which are discussed in the remainder of this section.

For the ζ-topology example in Sec. 6.3.2, the following assumptions were made. Firstly, it was assumed that the μ-topology for the Web shop application existed, i.e., was previously constructed or discovered. The existence of a μ-topology that combines all viable topologies allows a direct application of morphisms $m^{(i)}$ to derive viable ζ-topologies. Secondly, the applied morphisms $m^{(i)}$ use exclusively the inheritance clan relation as the basis for generating all possible viable topologies. This means that the richer the size of available types is, the bigger the set of viable topologies to evaluate will get, therefore making decision making tasks during migration to the cloud cumbersome.

Such gaps are covered and introduced in the following chapter. SCARM aims at exploiting knowledge among viable topologies. Particularly, a μ-topology can be interpreted as the evolution of how an application is distributed over time, capturing all application topologies during its life time. An application α-topology mostly remains constant, since this part of the application only changes due to major architectural decisions affecting its components, e.g., moving from a two- to a three-tiered architectural pattern. However, ζ-topologies change more often, as its underlying components or services depicted

as γ-topologies may not be sufficient for the identified application requirements, and can be regularly replaced. This fact motivates to consider all instantiated γ-topologies as part of an application's μ-topology, therefore describing how applications are distributed during their life-time.

Moving forward into how viable topologies are discovered during design time, a first step in SCARM consists of modeling an application specific topology, i.e., its α-topology. For discovering viable topologies for a given α-topology, SCARM applies a *Discovery Method* introduced in the following chapter, in order to discover viable ζ-topologies to build a first μ-topology. The discovery method reuses architectural and performance knowledge, e.g., μ-topologies, from similar applications. The usage of the enriched application topology model introduced in Chapter 5 plays a fundamental role in when discovering viable ζ-topologies, since non-functional requirements can be exploited when applying morphisms on μ-topologies. Moreover, SCARM enhances the inheritance clan relation with similarity analysis in order to trim down the space of viable topologies, therefore reducing the number of generated ζ-topologies.

Chapter 7

Discovery & Selection of Viable Cloud Application Topologies

7.1 Chapter Introduction

As discussed in previous chapters, SCARM aims at assisting application architects to migrate and distribute their applications in cloud environments. The model for viability of cloud application topologies presented in the previous chapter allows to design cloud applications in two ways: (i) depicting only application specific components (e.g., exclusively an α-topology), or (ii) depicting a first viable topology.

For both modeling approaches, SCARM constructs a μ- topology

based on architectural aspects of applications, e.g. structural and non-functional, using past knowledge from similar applications as the basis. The previous chapter started the discussion on inferring ζ-topologies – viable topologies–, by means of applying morphisms to μ-topologies. However, how μ-topologies can be discovered and constructed has not yet been discussed. This chapter addresses such a challenge, by means of presenting one possible mechanism using Case-based Reasoning (CBR) as the basis.

As outlined in Ch. 2, CBR relies on previous experienced situations to tackle new encountered problems [She03]. In a nutshell, SCARM *leverages CBR and applies similarity analysis by means of exploring applications' previous migration and distribution decisions in the cloud and assists in constructing a first set of viable ζ-topologies constituting a μ-topology model*. There are two fundamental aspects tackled in this chapter when analyzing similarity with previous application migration decisions and distributions: (i) similarity w.r.t. its architectural aspects, such as its structure and characteristics of its components, and (ii) similarity w.r.t. its performance requirements and workload characteristics.

This chapter first introduces a *Discovery & Construction of Viable Topologies* method, which aims at facilitating the process for constructing μ-topologies. This method is aligned with SCARM and focuses on tasks related to discovering and selecting viable topologies, and for aggregating them into μ-topologies, given a set of α-topologies. As previously remarked, this method is one possible way for discovering and constructing μ-topologies. This chapter is structured following such a method, and conceptually dives into each step. Concepts introduced in this chapter have been developed in the scope of the research works [Din16] and [Pin16].

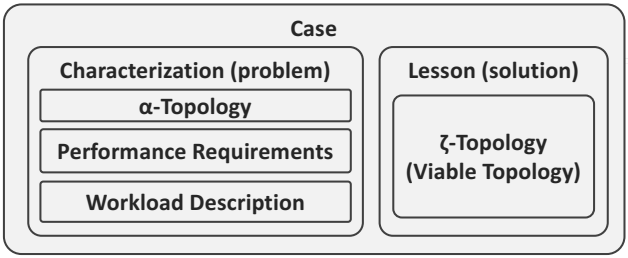


Figure 7.1: Case Representation [Pin16]

7.2 Discovery & Selection Method Overview

The discovery of viable topologies in SCARM is built upon capturing and processing knowledge of applications previously migrated to the cloud. More specifically, SCARM uses CBR together with similarity analysis in order to analyze viable topologies among applications. As introduced in Section 2.6, the R^4 model in CBR consists of (i) retrieving past cases similar to the new one, (ii) reusing past knowledge to solve the new case, (iii) revising the proposed solution, and (iv) partially or completely retaining the new solution for future problem solving [AP94]. Before describing how we implement the R^4 model in the discovery method, it is first necessary to build a representation of the case used in the R^4 model.

A *Case* essentially consists of a viable cloud application distribution (see Figure 7.1). Application architects model application specific-components as α-topologies, therefore specifying its architectural characteristics, such as the structure of its architecture, type of components, relationships among them, etc. An α-topology can be enriched with application non-functional aspects, such as performance require-

ments. An enriched α-topology –containing performance demands and workload characteristics– builds a *Characterization* of a case, i.e., the characteristics of a *Problem*. A concrete application distribution, represented as a ζ-topology, corresponds to a *Lesson* or *Solution* to the given problem. Lessons can evolve over time, as applications may need to be redistributed due to changing performance demands and increased costs of the current solution. In such a scenario, problem characteristics vary, and therefore it is necessary to build a new case.

Having the case representation in place, the remainder of this section presents a method depicted in Figure 7.2 aimed at discovering and building μ-topologies for given α-topologies. As previously discussed, the discovery method is built atop of CBR, which uses cases as the basis for capturing and processing past knowledge. These build a knowledge base of application distributions, consisting of viable ζ-topologies, which can be aggregated into μ-topologies, and performance knowledge (see Fig. 7.2).

Focusing now on the discovery method, application architects in SCARM typically model their application-specific architectures as α-topologies during design time, as depicted in the *α-Topology* step in Fig. 7.2. A subsequent *Reasoning Parameters* step consists of deriving criteria for analyzing how similar cases are in a knowledge base. Reasoning parameters are presented in Sec. 7.3, and allow to describe past and new cases, as well as used for analyzing similarity analysis among cases. This set of *Modeling* tasks create a characterization of a case, i.e., a representation of an application distribution problem, which is used as the input in the *Discovery* steps.

Discovery and construction of μ-topologies in SCARM is partitioned into two subsequent set of tasks: (i) *Similarity Analysis* and (ii) *Selection & Construction*. Similarity analysis tasks compute similarity

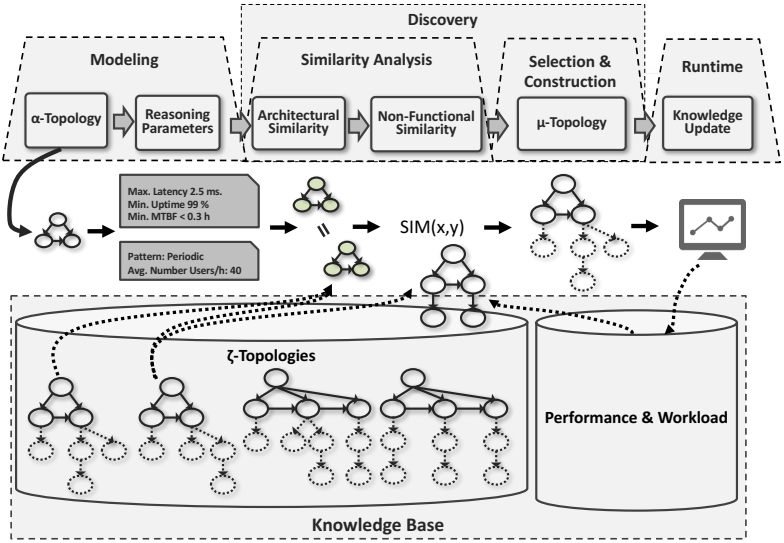


Figure 7.2: Discovery & Construction of Viable Topologies - Method Overview

measures based on reasoning parameters of new problems and previous cases in the knowledge base. Similarity is analyzed in two steps: (i) *Architectural Similarity* and (ii) *Non-Functional Similarity*. Although these are depicted as sequential steps for illustrative purposes, these can also be executed in parallel. The architectural similarity step computes similarity of application architectural characteristics, such as its structure and type of components, while the non-functional similarity focuses on application non-functional parameters. A detailed description of similarity calculation is provided in Sec. 7.4, and provides a similarity measure, which is used as the basis for deciding if a ζ-topology will be considered for constructing a μ-topology.

The *Selection & Construction* step dynamically or manually selects ζ-topologies to be aggregated into a μ-topology. Dynamic selection requires the specification of an accepted similarity measure threshold, while manual selection requires interaction with application architects. Once ζ-topologies are selected, these are aggregated into a μ-topology. Application architects can then evaluate ζ-topologies individually, e.g., using utility-based decision making techniques, as presented in Ch. 8. When application architects select a ζ-topology candidate to be provisioned, *Runtime* steps update the knowledge base with the selected ζ-topology, and runtime performance metrics and workload attributes knowledge are updated in the knowledge base.

The aforementioned method is one possible approach for facilitating the discovery of viable ζ-topologies, using similarity analysis and CBR as the basis. However, such approach is not exclusive, as application architects can also model μ-topologies by directly combining α- with γ-topologies. The remainder of this chapter discusses further the different steps of the discovery and selection method depicted in Fig. 7.2. In particular, this chapter first identifies reasoning parameters for the case characterization, which are used as the basis for subsequently defining similarity with cases in the knowledge base. Subsequently, it presents the local-global principle and similarity functions common to both functional and non-functional analyses. Finally, similarity analysis, selection of cases, and aggregation of knowledge are discussed.

7.3 Reasoning Parameters

The adoption of CBR for discovering viable applications requires to first define reasoning parameters, which are used for analyzing simi-

larity among new and past cases in the knowledge base. Reasoning parameters constitute the problem characterization of cases defined in the previous section. In particular, Fig. 7.3 identifies reasoning parameters related to the application's architecture, which are represented in α-topologies.

Architectural Parameters Application architectures are defined within topology models in the scope of this work. In particular, α-topology models capture application-specific architectural aspects, as these depict its structure, its components, type of components, and relationships among them. As depicted in Fig. 7.3, three parameters are considered in α-topologies for similarity analysis when discovering viable topologies: (i) node type definition, (ii) relationship type definition, and (iii) graph structure.

Node types define which type of component is represented in a node. For the Web shop application illustrated in Fig. 7.3, its parent application node is of type *Web Application*, and its front- and back-end components are of type *PHP Application*, *WAR*, and *SQL_Database*. The relationship among components are of type *consists_of* and *interacts_with*. Such type definitions essentially depict a Web application comprising a PHP-based front-end, and a back-end consisting of a Java-based Web application and an SQL database. The α-topology graph describes structural characteristics of an application architecture. For instance, the α-topology for the previous Web shop application reveals that it is built as a multi-tiered application with three-tiers: (i) front-end, (ii) back-end, and (iii) data tiers.

Performance Parameters Application performance parameters are defined as part of the enriched model of α-topologies. In particular,

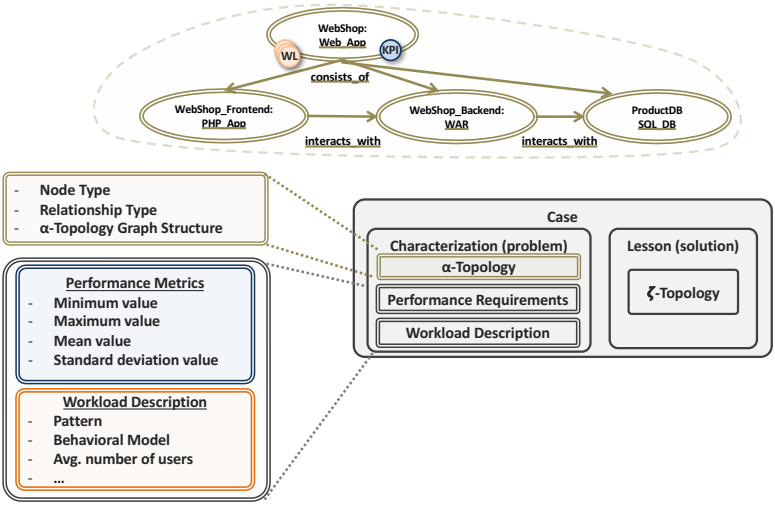


Figure 7.3: Reasoning Parameters - Functional & Non-Functional Application Characteristics

these are related to the performance metrics and workload attributes presented in Ch. 5. In summary, Ch. 5 categorizes relevant performance metrics towards providing application architects with a baseline to specify application performance requirements. Moreover, Ch. 5 also identifies a set of workload attributes that can be leveraged for describing application workload behaviors in α-topologies.

Performance metrics identified in Ch. 5 are used as the basis for building performance reasoning parameters. Application architects can specify performance requirements by means of defining thresholds for each performance metric, which are then analyzed during runtime. Metrics are associated with statistical analytical indicators, such as minimum and maximum, mean, and standard deviation values, and

serve to describe the intrinsic characteristics of a performance metric. These four indicators constitute one part of the *Performance* knowledge in a case (see Fig. 7.3).

Workload attributes constitute the other part of performance knowledge in a case. In particular, these are specified by application architects during design time, and can vary during runtime. For instance, average number of users may be a significant workload reasoning parameter for the Web shop application depicted in Fig. 7.3, as well as workload patterns, which may significantly change during peak shopping periods.

7.4 Similarity Analysis of Viable Topologies

As discussed in the previous sections, the discovery method in SCARM uses similarity measures for evaluating similarity among new and past cases in the knowledge base. Similarity measures are the underlying ingredient in CBR for retrieving approximately matching information [Sta05]. Section 2.6 identified two approaches for calculating similarity: *distance-based* and *representational*. Distance-based approaches are used when applying the *local-global* principle of similarity analysis and determine the distance between new and past cases. More information of the local-global principle is introduced in the remainder of this section. On the other hand, representational approaches use indexing mechanisms to connect similar cases.

SCARM uses distance-based similarity and applies the *local-global* principle, because SCARM computes similarity among different types of attributes and aggregates them into a single similarity value. Similarity analysis is split into evaluating architectural and performance similarity, as elaborated in the remainder of this section. Architec-

tural similarity analysis considers application architectural parameters, such as the type of an applications and its components, its graph structure, etc. On the other hand, performance similarity analysis correlates new with past cases with similar performance parameters, specified as performance metrics and workload attributes.

7.4.1 Local-Global Principle Overview

Before introducing conceptual foundations for functional and non-functional similarity in SCARM, let's first introduce how SCARM leverages the *local-global* principle. In a nutshell, SCARM applies the local-global principle by first computing local similarity on every local attribute, and then by subsequently aggregating them in a global similarity value.

The local-global principle consists of breaking the computation of similarity among cases into a local and global part [Sta03]. Local similarity is applied on single attribute values of each case, while global similarity goes a step further, by aggregating local similarities among cases in a generic measure. Section 2.6 defined local similarity measures as $sim_A : \mathbb{A}_{range} \times \mathbb{A}_{range} \rightarrow [0, 1]$, where $\mathbb{A}_{range}$ corresponds to the value range of an attribute A. SCARM computes local similarity using *difference-based* similarity functions, which calculate the absolute difference between two values. In other words, difference-based functions assume that the decrease of similarity stands in some relation with increasing difference of the values to be compared [Sta03].

There are different similarity functions typically used, such as *Threshold*, *Linear*, *Exponential*, and *Sigmoid* [Sta03]. SCARM is not conceptually restricted to the usage of a specific similarity function. However, for exemplification purposes, lets assume the usage of an

exponential similarity function, which is defined in [Sta03] as follows:

$$sim(x,y) = \frac{1}{e^{|\delta(x,y)|}} \tag{7.1}$$

where $\delta(x,y)$ is a difference function. Typical difference functions, as defined in [Sta03], are linear or logarithmic. Condensing both of them:

- linear $\delta(x,y) = y - x$,
- logarithmic $\delta(x,y) = \begin{cases} ln(x) - ln(y) & \text{for } x,y \in \mathbb{R}_+ \\ -ln(-x) + ln(-y) & \text{for } x,y \in \mathbb{R}_- \\ undefined & \text{else} \end{cases}$

As depicted in the logarithmic $\delta(x,y)$, if an attribute contains an undefined value, the difference function $\delta(X,Y)$ returns an *undefined* value. In such a case, this attribute is not considered when computing global similarity. So far the local similarity has been introduced. Going a step above, SCARM aggregates local similarity values into a single global similarity that can be presented to application and business architects. Although there are several aggregation functions, such as *Weighted Average Aggregation*, *Minkowski Aggregation*, *Maximum Aggregation*, and *Minimum Aggregation*. SCARM utilizes the Weighted Average Aggregation function, due to its simplicity for computing a global similarity measure [MC11]. It is defined as follows:

$$\pi(sim_1, ..., sim_n, \vec{w}) = \sum_{i=1}^{n} w_i \,.\, sim_i \tag{7.2}$$

considering equal values w_i for each defined similarity measure

sim_i for obtaining a normalized distance. For example, if we consider two applications and the measures *latency* and *CPU Utilization*, the local similarity is calculated by independently calculating the distance for each measure among both applications. Towards aggregating both similarities into a single measure, the Weighted Average Aggregation function calculates a global similarity measure considering both measures equally important, i.e., with equal weights. The following section assembles the previous concepts into different steps where local and global similarity is calculated in SCARM.

7.4.2 Application Similarity Calculation Overview

Function *ApplicationSim* depicted in Algorithm 7.1 calculates a global similarity measure among two applications, by means of sequencing local and global similarity calculations for application architectural characteristics and non-functional attributes. In particular, this function in Algorithm 7.1 receives as input a new and an existing application (denoted as *case*), and first computes local similarities:

- among application types (see line 2),
- among the structure of their architectures (see line 3),
- and among performance requirements and workload characteristics (see lines 4 and 5)

Once local similarities are calculated, the Weighted Average Aggregation function is computed for all local similarities to generate a global similarity measure (see line 12). The output of the *ApplicationSim* function is essentially a global similarity value describing how similar two applications are. The remainder of this section elaborates

on how local similarity is computed for both architectural structure, performance requirements, and workload characteristics.

Algorithm 7.1 Application similarity calculation - Adapted from [Pin16]. *Similarity* is abbreviated as *Sim*.

```
1:  function ApplicationSim(new_app, case)
2:      appTypeSim ← AppTypeSim(new_app, case)
3:      archStructureSim ← ArchitectureSim(new_app, case)
4:      performanceSim ← PerformanceSim(new_app, case)
5:      workloadSim ← WorkloadSim(new_app, case)
6:      localSimList ← CreateList
7:      add(localSimList, appTypeSim)
8:      add(localSimList, structuralSim)
9:      add(localSimList, componentsSim)
10:     add(localSimList, performanceSim)
11:     add(localSimList, workloadSim)
12:     appSim ← WeightedAvgAggregation(localSimList)
13:     return appSim
14: end function
```

7.4.3 Architectural Similarity

Focusing on the application's architecture, application architects specify during design time in SCARM α-topologies, which describe (i) application and component type definitions, (ii) and relationships among its components. This information is distributed among the topology at different levels, such as nodes and relationships, and graph.

Evaluating the similarity of application architectures in single-step manner is not possible due to the heterogeneity among reasoning parameters within α-topologies. Therefore, SCARM calculates the similarity among application architectures in two steps. First, SCARM

calculates similarity between applications, i.e., among their types, in order to find analogous applications in the knowledge base. Subsequently, SCARM computes the similarity among application components and the relationships among them, both defined in α-topologies. The remainder of this section independently elaborates on both steps.

7.4.3.1 Similarity among Application Types

Evaluating similarity among types of applications presents one major challenge, since these are typically described using discrete values. As summarized in Ch. 5, Richter proposes the usage of *Similarity Tables* to represent similarity among non-numerical attributes [Ric08]. Particularly, these help to answer questions like *how similar is application A to application B?*, or *how similar is component A to component B?*

Similarity tables consist of a matrix representing similarity between a query value and a case value. In other words, similarity tables contain similarity measures with values between 0 and 1 structured as a matrix, where rows and columns correspond to nonnumerical discrete values. Figure 7.4 depicts a similarity table for different types of applications, e.g., Web shop, E-commerce, etc. Similarity tables are initially empty, in the sense that only its matrix diagonal is initialized with 1's. Similarity tables in SCARM evolve over time, as new types of applications emerge. These must be, therefore, maintained by domain experts, typically application architects sharing knowledge in an architecture community.

Focusing on a Web shop application as example, application architects define the application type at the root level of α-topologies during design time. When analyzing similarity among application types in the knowledge base, SCARM prescribes to inspect the root

	Web Shop	E-Commerce	Rich Internet	Mobile Backend	Customer Relationship	News Feed	Email
Web Shop	**1**	0.5	0	0	0	0.1	0
E-Commerce	0.5	**1**	0	0	0	0	0
Rich Internet	0.3	0.3	**1**	0	0.3	0	0
Mobile Backend	0	0	0	**1**	0	0	0
Customer Relationship	0.1	0	0	0	**1**	0	0
News Feed	0	0	0	0	0.1	**1**	0.5
Email	0	0	0	0	0	0.5	**1**

Figure 7.4: Sample Similarity Table for different Application Types

node type definitions and to generate queries with such a type as query values. For the Web shop application example and focusing on the application similarity, the most similar application in Fig. 7.4 is an E-commerce application with a similarity measure of 0.5. This similarity measure is subsequently aggregated into an application's architectural global similarity measure, as discussed in the following section.

7.4.3.2 Similarity of α-topologies

The previous section focused on calculating similarity among types of applications, which are defined in root nodes of α-topologies. However, such approach covers only a first step when calculating the similarity between application architectures. The next step consists of analyzing the similarity among application architectures, by means of analyzing how similar their α-topologies are.

Nodes and relationships in α-topology models represent architectural characteristics of applications, such as its design pattern, e.g.

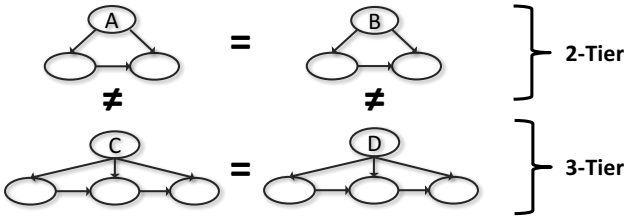


Figure 7.5: Architectural Structure - Example Graph Matching

multi-tier. For instance, Fig. 7.5 depicts α-topologies for two- and three-tiered applications. Structural similarity of α-topology graphs can be calculated by comparing the number of components and relationships among them. In Fig. 7.5, applications *A* and *B*, and *C* and *D*, have equivalent structural characteristics, while *A* and *C*, and *B* and *D*, do not, as they differ on the number of child nodes.

Algorithm 7.2 encompasses operations to calculate similarity among application-specific architectures represented as α-topologies. More specifically, the function *ArchitectureSimilarity* analyzes the intrinsic characteristics of α-topologies, which are evaluated as follows: (i) by calculating the structural similarity among application architectures, e.g., their α-topology graphs, and (ii) by calculating the similarity among application components, each represented as a leaf typed node in an α-topology, and (iii) by aggregating both similarities into a similarity value that represents how similar α-topologies are.

Architectural Structure Similarity Architectural structure similarity is evaluated in SCARM by applying graph matching as the basis (see Algorithm 7.2). The function *ArchitectureSimilarity* first retrieves the α-topology graphs (see lines 3 and 4). Subsequently, it analyzes

the structural similarity among α-topologies, by computing graph matching in the function *graphMatching* (see line 5). SCARM does not conceptually limit which type of graph matching must be used, as both *Exact Graph Matching* and *Inexact Graph Matching* defined in [Ben02], could be supported. For instance, the exact graph matching for applications C and D in Fig. 7.5 returns a similarity of 1, while for applications A and C returns a similarity of 0.

Similarity of Application Components In a second step, the function *ArchitectureSimilarity* in Algorithm 7.2 calculates the similarity of application components, by using similarity tables as the basis. Function *getAppNodes* returns a list of leaf nodes in an α-topology (see lines 7 and 8), were each node represents an application component. This list is then iterated in order to calculate the similarity among the node types. Similarity is essentially calculated using the similarity table *nodeTypesSimTable* in the function *calculateNodeTypeSimilarity* (see line 15). For example, for α-topologies A and B in Fig. 7.5, the function *getAppNodes* first returns the list of leaf nodes for each α-topology. After, the function *ArchitectureSimilarity* iterates through both lists and calculates similarity for each node type, e.g., if both nodes represent back-end SQL databases, then the similarity is 1. If one node represents an SQL database, while the other represents a NoSQL database, then their similarity value will be ≈ 0.

Aggregating Similarities Once application structural and component similarities are calculated, the function *ArchitectureSimilarity* in Algorithm 7.2 computes the Weighted Average Aggregation of all similarities previously calculated, and returns the similarity among α-topologies (see line 20). This is then used as input for calculating a

global application similarity, as discussed in Sec. 7.2.

Algorithm 7.2 α-Topologies Architectural Similarity Calculation - Adapted from [Pin16]. *Similarity* abbreviated as *Sim*.

```
1: function ArchitectureSimilarity(new_app, case)
2:     archSimilarities ← createList
3:     alphaTopology ← getAlphaTopology(new_app)
4:     caseAlphaTopology ← getAlphaTopology(case)
5:     graphsMatch ← graphMatching(
           alphaTopology, caseAlphaTopology)
6:     add(archSimilarities, graphsMatch)
7:     appNodes ← getAppNodes(appAlphaTopology)
8:     caseAppNodes ← getAppNodes(caseAlphaTopology)
9:     nodeTypesSimTable ← getNodeTypesSimTable
10:    i ← size(appNodes)
11:    j ← size(caseAppNodes)
12:    while i > 0 and j > 0 do
13:        nodeType ← getNodeType(appNodes, i)
14:        caseNodeType ← getNodeType(caseAppNodes, j)
15:        nodeTypeSim ← calculateNodeTypeSim(
               nodeTypesSimTable, nodeType, caseNodeType)
16:        add(archSimilarities, nodeTypeSim)
17:        i ← i − 1
18:        j ← j − 1
19:    end while
20:    architectureSim ← WeightedAvgAggregation(
           archSimilarities)
21:    return architectureSim
22: end function
```

7.4.4 Similarity of Performance Requirements

So far, previous sections only focused on analyzing similarity among functional application characteristics. Non-functional application characteristics, such as performance, are also evaluated in SCARM. This section elaborates on conceptual foundations for similarity among performance requirements.

Similarity analysis among performance parameters is simpler, as these attributes take numeric values, which are typically measured over an application's lifetime. In particular, difference functions introduced in Sec. 7.4.1 are used to calculate absolute difference between performance characteristics among applications. As a recap of the reasoning parameters introduced in Sec. 7.3, performance reasoning parameters include different statistical descriptors for each metric, such as the *minimum and maximum*, *mean*, and *standard deviation* values. This requires to firstly apply local similarity on each statistical descriptor, and then aggregate their similarity measures into a global similarity.

Algorithm 7.3 sketches how performance similarity is calculated in the function *PerformanceSim*. In a first step, the function gathers performance metrics for the new application and for the existing one in the knowledge base (see lines 2 and 3). For each performance metric in the new case, local similarity is firstly applied for every performance reasoning parameter (see line 5). Essentially, for each performance metric, the *Performance Similarity* function first extracts each metric descriptor, and calculates the similarity among them (see line 9). Once the similarity among metric descriptors are processed, a single similarity measure for all of them is calculated (see line 16). This happens iteratively for all metrics.

In general, SCARM uses difference functions for calculating similarity between numeric attributes, as depicted in line 11. After calculating similarity for each performance metric, these are aggregated using the Weighted Average Aggregation function for calculating similarity among all performance metrics (see line 18).

Algorithm 7.3 Performance similarity calculation - Adapted from [Pin16]. *Similarity* is abbreviated as *Sim*.

```
1:  function PerformanceSim(new_app,case)
2:      performanceMetrics ← performanceMetrics(new_app)
3:      casePerformanceMetrics ← performanceMetrics(case)
4:      performanceSimList ← createList
5:      for each metric ϵ performanceMetrics do
6:          queryMetricDescriptors ← getDescriptors(metric)
7:          caseMetric ← getMetric(case, metric)
8:          descriptorSimList ← createList
9:          for each descriptor ϵ queryMetricDescriptors do
10:             caseMetricDescriptor ← getDescriptor(
                    caseMetric, descriptor)
11:             difference ← computeDifference(
                    descriptor, caseMetricDescriptor)
12:             descriptorSimilarity ← e^(abs(difference) ∗ −1)
13:             add(descriptorSimilarity, descriptorSimList)
14:         end for
15:         metricDescriptorSimilarity ←
                WeightedAvgAggregation(descriptorSimList)
16:         add(performanceSimList, metricDescriptorSimilarity)

17:     end for
18:     performanceSim ←
            WeightedAvgAggregation(performanceSimList)
19:     return performanceSim
20: end function
```

7.4.5 Similarity of Workload Models

Application workload models in SCARM have been previously introduced in Ch. 5. α-topologies are enriched with parameters describing workload behavioral characteristics, such as average number of users, behavioral pattern, etc. These are part of the reasoning parameters in each case of SCARM, as depicted in Sec. 7.3.

Algorithm 7.4 Workload similarity calculation. Adapted from [Pin16] and *Similarity* abbreviated as *Sim*.

```
1:  function WorkloadSim(new_app,case)
2:      workloadAttr ← getWorkloadAttrs(new_app)
3:      workloadSimList ← createList
4:      for each workloadAttr ϵ workloadAttrs do
5:          workloadCaseAttr ← getAttr(case, workloadAttr)
6:          workloadAttrType ← getAttrType(workloadAttr)
7:          if workloadAttrType = numeric then
8:              difference ← computeDifference(workloadAttr,
                    workloadCaseAttr)
9:              workloadAttrSim ← e^(abs(difference) ∗ −1)
10:         else
11:             simTable ← getSimTable(workloadAttrType)
12:             workloadAttrSim ← retrieveSim(simTable,
                    workloadAttr, workloadCaseAttr)
13:         end if
14:         add(workloadSimList, workloadAttrSim)
15:     end for
16:     workloadSim ←
            WeightedAvgAggregation(workloadSimList)
17:     return workloadSim
18: end function
```

Focusing on how similarity among application workloads is analyzed in Algorithm 7.4, function *WorkloadSim* coordinates the calcu-

lation of similarity for both numeric workload attributes, e.g., average number of users, and non-numeric workload attributes, e.g., workload pattern. For the nonnumerical workload attributes, SCARM uses similarity tables like the ones for application types. These must also be maintained by domain experts, such as application architects, in order to evolve them based on experience.

This function iterates over all application workload attributes, and computes similarity for each attribute (see line 4). As previously described, similarity for nonnumerical values is calculated using similarity tables. For similarity among numeric values, SCARM uses difference functions, as part of the similarity function (see lines 8 and 9). After calculating the similarity for each workload attribute, these are aggregated using the Weighted Average Aggregation function for calculating the similarity among all workloads (see line 16).

7.4.6 Aggregation into a Global Similarity

As previously discussed, SCARM leverages the local-global principle to calculate similarity between α-topologies and past cases. Before selecting similar topologies to construct an application μ-topology, a last step calculates the global similarity using the Weighted Average Aggregation function (see Eq. 7.2).

The outputs from the functions in Algorithms 7.1, 7.2, 7.3, and 7.4 are local similarity values that used as input for each sim_i to calculate a global similarity value (see Eq. 7.2). In the case of having local similarity values that are zero, these are also considered when calculating the global similarity. This is mainly because similarity analysis in SCARM does not aim at discarding cases using similarity analysis, as this is part of the selection step of similar topologies introduced in

Section 7.5. In any case, having low or zero local similarity values essentially have an impact on the global similarity value, and therefore during the selection of similar topologies.

With this, the discovery method can present to application architects with a list of similar application ζ-topologies, each with an aggregated similarity value. These are the input for the selection step of similar topologies and the construction of μ-topologies.

7.5 Selection of Similar Topologies & Construction of μ-Topologies

So far, this chapter introduced conceptual foundations for discovering viable distributions using similarity analysis as the basis. The next step in SCARM consists of (i) selecting topologies that are used (ii) to construct an application's μ-topology. In other words, application architects can define which discovered ζ-topologies are aggregated into a μ-topology (see Fig. 7.2).

We previously defined that the selection step can be executed in an automated or manual manner. In both cases, application architects must interact in some degree with the system. For a dynamic selection, a similarity acceptance threshold must be defined. For manual selection, the selection step presents an descendant ordered set of discovered viable distributions (ζ-topologies), based on the similarity measures calculated during the discovery step for the original α-topology. In both manual and automatic cases, the selection step in the discovery method corresponds to the *Revise* and *Retain* phases of the CBR R^4 model (see Ch. 2). Application architects can select an existing ζ-topology or refine it, therefore creating in both cases a new case in the knowledge base.

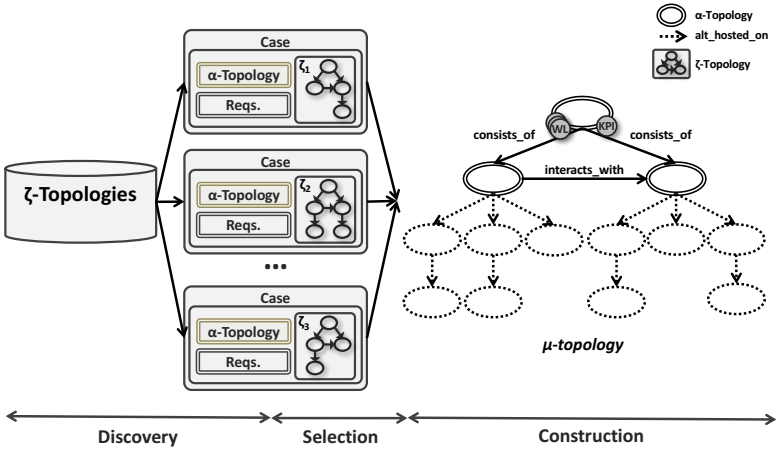


Figure 7.6: Aggregation - Construction of μ-Topologies

Once application architects select one or multiple viable ζ-topologies, these are aggregated to construct an application's μ-topology. Figure 7.6 illustrates the steps for discovering, selecting, and aggregating ζ-topologies into a μ-topology. More specifically, it assumes that three ζ-topologies, ζ_1, ζ_2, and ζ_3, are discovered. In the case of selecting the three of them, these are aggregated into a single μ-topology. The μ-topology can be subsequently used to evaluate each possible application distribution, e.g., using utility analysis, in order to make a final decision on which ζ-topology is instantiated. SCARM uses past cases and similarity analysis as the basis for discovering alternative ζ-topologies. However, the discovery of alternative topologies is not strictly limited to using past cases, as another way of generating ζ-topologies could consist of applying morphisms after modeling a complete μ-topology in the first step, as discussed in Ch. 6.

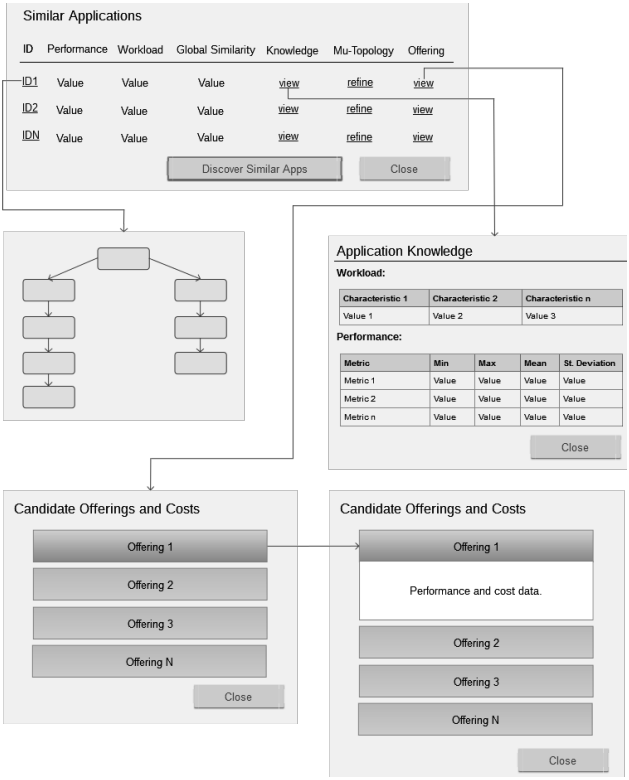


Figure 7.7: Interface Layout for Manual Selection of ζ-Topologies to build μ-Topologies [Pin16].

To illustrate the selection step in SCARM, Fig. 7.7 depicts an interface layout sample used by application architects for selecting similar viable ζ-topologies for their application μ-topologies. The upper part shows a summary of discovered similar applications, each associated with a global similarity value, and with further analytical functionali-

ties, such as (i) knowledge viewing and (ii) cost calculation of cloud offerings in ζ-topologies.

7.6 Knowledge Update

Knowledge aggregation in SCARM is performed after selection and provisioning, and during runtime of a viable ζ-topology. During runtime, performance knowledge is aggregated in the performance and workload knowledge bases. Some knowledge can be both manual and automatically updated, such as similarity tables depicting how similar application types are (see Sec. 7.4.3.1). Other knowledge, e.g., application response time or throughput, can be automatically aggregated using monitoring tools. Overall, knowledge update does not only serve as support for application architects in SCARM for discovering viable topologies with similar performance characteristics, but also for analyzing how the application performance evolves.

During this step, performance knowledge is aggregated for reasoning parameters provided by application architects. In particular, SCARM currently supports performance and workload reasoning parameters depicted in Sec. 7.3, i.e., workload attributes, and performance metrics. The collection and aggregation of performance knowledge is supported by the instrumentation provided in SCARF-T (to be discussed in Ch. 9). The knowledge aggregation phase must, therefore, not only collect workload and performance measurements, but also compute statistical indicators, such as minimum and maximum values, and mean and standard deviation.

Chapter 9 elaborates on the technical implementation for supporting the previously introduced similarity analysis in SCARM. In particular, it provides technical insights on how the discovery method

is realized. It introduces the implementation of CBR, similarity analysis algorithms and functions, and show how these can be used by application architects in SCARF-T.

Chapter 8

Utility-based Evaluation of Viable Cloud Application Topologies

8.1 Chapter Introduction

The existence of a vast catalog of cloud services is an actual challenge for application architects. This work introduced SCARF, a framework supporting the migration of existing or the design of new cloud applications in a cost- and performance-efficient manner.

As discussed in the previous chapters, SCARM incorporates manual, automatic, and synergistic tasks supporting the complete application's life cycle. In particular, it allows application architects to model

their application-specific α-topologies, enrich them with performance information, dynamically discover viable ζ-topologies, and construct their application's μ-topology. The discovery method introduced in the previous chapter, however, does not fully reduce the complexity of having a wide number of viable application topologies. In particular, the solution space may still be too complex for application architects' decision making tasks, due to the abundance of alternatives for the deployment of the application [KH18].

This chapter goes a step further in SCARM, and introduces a utility-based evaluation mechanism, used as the basis for decision making support in SCARM. In short, *Utility* emerged in the economics domain and is used to measure users' perceived satisfaction when consuming a good or service [Mar09] (an extensive introduction of utility theory is provided in Sec. 2.5). The usage of utility functions has been adopted in the domain of management of computational and storage resources, as discussed in [MF11] and [STFG08], respectively. In the domain of cloud applications, Andrikopoulos et al. motivate its usage and propose a cost-oriented utility function for designing cloud applications [AGLW14]. Utility functions are typically domain specific. In particular, different utility functions can be represented depending on the specific domain requirements taken into consideration.

This chapter addresses a utility model for cloud applications. It is geared towards quantifying the economic impact of each application viable ζ-topology, focusing on analyzing the trade-off between its cost and performance concerns. In short, our utility model points towards analyzing the profitability of viable topologies. The utility model presented in this chapter is essentially the utility model built in SCARF and a previous version has been published in [GAB+18]. The utility model presented here deviates from that version in the following: (i)

it enhances and simplifies formal definitions, (ii) it restructures its elaboration, and (iii) it renames some of the definitions for clarity purposes.

Sec. 8.2 opens with a study of the economic benefits and design challenges when using cloud environments to host business applications. Since utility is used as the basis for decision making support in SCARF, Sec. 8.2 builds on the premise of using utility as its underlying mechanism for decision making support. The SCARF utility model is formally defined in Sec. 8.3, where a profitability utility model is presented. This chapter closes with a discussion in Sec. 8.4.

8.2 Utility & Decision Making in SCARF

The cloud computing paradigm enables the shifting of capital expenditures to operational expenditures. In particular, studies have shown that organizations running their own infrastructure invest approximately only 20% in designing and running their applications, while 80% of the time is spent on administrating and maintaining the data center [Kep11]. Cloud computing helps organizations in flipping this ratio, by means of allowing IT departments to spend 80% of their efforts in the core business and design of applications, while minimizing the maintenance expenditures to 20% [Kep11]. More specifically, it allows organizations to focus on the economic impact of their application architectures.

Cloud computing has created, however, several decision making challenges. Applications cannot be designed and provisioned in a cost-efficient manner using traditional methods and techniques, due to the fundamental properties of cloud infrastructures, such as multi-tenancy and demand-side aggregation [HY10]. Application architects

must rethink their application architectures and (re)design their decision making processes for new or migrated applications to cloud environments. In particular, applications cannot be simply packed and deployed in virtual machines, as these does not exploit the full spectrum of benefits of cloud environments [ABLS13]. Moreover, each application requirement builds towards creating a multi-dimensional business and operational problem, where each dimension entails one specific concern, e.g., cost, security, performance, etc. In a nutshell, there exists a strong need to bring together business and operational perspectives when designing cloud applications, by means of leveraging the economic benefits of cloud computing to minimize operational costs and to maximize its business revenue. In particular, there is a general necessity for decision making mechanisms and tools to support a cost-efficient design and provisioning of cloud applications.

As a recap of Sec. 2.5, utility progressively gained support in the management of computer systems and distributed environments. Focusing on its definition, utility is defined as a value representing the desirability of a particular state or outcome [STFG08; Fis70]. More specifically, it is a measure of preferences over a set of goods and services [Mar09], which is typically used in game theory and decision making, e.g., in multi-attribute utility theory [KR93]. In other words, utility is a way to represent customer's satisfaction when consuming a good or a service w.r.t. the original demand.

The usage of utility as the underlying mechanism for decision making during the design of cloud applications has two main objectives in the scope of this work. Firstly, this work brings together IT and business perspectives during the design of cloud applications, by means of bridging the design gap between operational and business perspectives of cloud applications. Secondly, this work leverages the

usage of utility theory, and develops a utility model for business and application architects, which can be leveraged in decision making tasks related to the selection of viable cloud application distributions (depicted as ζ—topologies).

Utility functions can assist in solving multi-dimensional problems related to the trade-off among application requirements. In particular, this work focuses on optimizing application cost and performance when distributing its components and spanning them among different cloud services. The focus on the profitability (as depicted in Sec. 8.3) as the fundamental ingredient of utility of cloud applications is, however, one possible alternative for the optimal cloud application distribution problem. There may exist further utility functions, such as optimizing security or privacy, which are considered outside of the scope of this work.

8.3 Utility Model

The remainder of this chapter establishes a formal utility model, denoted as SCARF Utility Model, which is used as the basis for the decision making mechanism in SCARF. For the scope of this work, the SCARF utility model focuses on the *quantitative representation of the monetary cost and performance trade-off of viable distributions of an application (depicted as a set of ζ-topologies aggregated into a μ-topology) spanning different cloud services for a time period* [GAB+18].

For illustrative purposes, Fig. 8.1 is used as the example while developing the utility model. Firstly, this section introduces preliminary definitions, which are subsequently used for introducing an example utility function (see Sec. 8.3.2). The proposed utility function quantifies the cost and performance trade-off, by means of computing

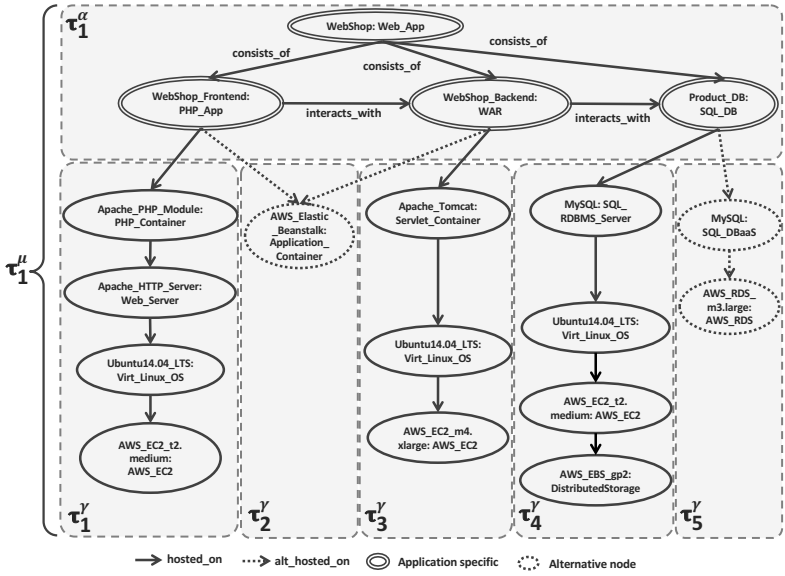


Figure 8.1: $\tau^\alpha, \tau^\gamma,$ and τ^μ topologies mapping for the viable topologies of the Web Shop Application

the revenue, cost, and end user satisfaction when distributing an application.

8.3.1 Preliminary Definitions

Previous to introducing the utility function, this section formalizes the variables and sets used in the utility model. Let's first assume that we have a fixed application to distribute A, which is represented by at least one α-topology and has at least one viable ζ-topology. Some of the functions and constants depicted in the remainder of this section are dependent on the application A, which are explicitly denoted as

such. The life time of an application is defined as the set of sets $\Psi = \{\psi_m \mid m \geq 1\}$, where each $\psi_m \in \Psi$ is an ordered discrete time interval $\{t_1, ..., t_n\}$ that represents the period in time when an application viable topology is instantiated, i.e., its components are distributed according to the viable topology. For example, if an application is redistributed on a weekly basis, then ψ_1 would be, for example, $\psi_1 = \{Mo\ 18.7.2016, ..., Su\ 24.7.2016\}$. Since a cloud application's μ-topology is decomposed into an application-specific α-topology and multiple non-application specific (and reusable) γ-topology models (see the enhanced topology model in Chap. 5), we define:

- The set of application-specific α-topologies as $\mathrm{T}^{\alpha} = \{\tau_p^{\alpha}\}$. For the Web shop application in Fig. 8.1, T^{α} contains one α-topology τ_1^{α}, which is indicated by double lines.
- The set of all available and reusable application non-specific γ-topologies as $\mathrm{T}^{\gamma} = \{\tau_q^{\gamma}\}$, where each τ_q^{γ} represents the underlying resources, such as middleware or cloud service instances, that can be used to host one or multiple application components. For the Web shop application in Fig. 8.1, $\mathrm{T}^{\gamma} = \{\tau_1^{\gamma}, ..., \tau_5^{\gamma}\}$.
- The set of viable ζ—topologies derived from a μ-topology of an application as $\mathrm{T}^{\zeta} = \{\tau_i^{\zeta}\}$, where a function $f : \mathrm{T}^{\zeta} \rightarrow \Psi$ maps each $\tau_i^{\zeta} \in T^{\zeta}$ to its associated time interval $\psi_m \in \Psi$ in the application's life time. A morphism function $m^{(i)} : \mathrm{T}^{\mu} \rightarrow \tau_i^{\zeta}$ maps a μ-topology in T^{μ} to a viable topology $\tau_i^{\zeta} \in \mathrm{T}^{\zeta}$, as depicted in [AGLW14] and summarized in Sec. 6.3.2.

For the Web shop application μ-topology in Fig. 8.1 and for a time interval ψ_1, a possible viable ζ—topology τ_1^{ζ} can comprise the subset of γ-topologies $\{\tau_1^{\gamma}, \tau_3^{\gamma}, \tau_4^{\gamma}\} \subset \mathrm{T}^{\gamma}$ with the α-topologies subset $\{\tau_1^{\alpha}\} \subset \mathrm{T}^{\alpha}$.

So far, we defined the architectural aspects of the application. However, non-functional aspects, such as performance and cost, play a fundamental role when deciding among different cloud offerings. These are typically analyzed by measuring the system, e.g., its response time, or its availability. Formally, let's define:

- The set of measurements $\mathcal{V} = \{V_n\}$. Each measurement $V \in \mathcal{V}$ consists of measurement values $V = \{v_i\}$ and corresponds to a time interval $\psi \in \Psi$ by means of a bijective map $f : V \rightarrow \Psi$. The measurement value v_i has been measured at a time $f(v_i) \in \Psi$.
- The set of business and operational requirements for an application α-topology as $R = \{r_j \mid 1 \leq j \leq |\mathcal{V}|\}$. For each requirement $r \in R$, a bijective map $g : R \rightarrow \mathcal{V}$ associates the requirement r with the set of corresponding measurement values $g(r) \in \mathcal{V}$. This allows to determine if the requirement r has been met at time $f(g(r))$.

For instance, lets consider the average latency for analyzing performance and the time intervals $\psi_1 = \{Mo\ 18.7.2016, ..., Su\ 24.7.2016\}$ and $\psi_2 = \{Mo\ 25.7.2016, ..., Su\ 30.7.2016\}$. Each set of measurement values $V_n \in \mathcal{V}$ essentially contains the average latency for each time interval, e.g., $V_1 = \{1.5\,ms, 2\,ms\}$. Since this example only considers the latency measurements, then $\mathcal{V} = \{\{1.5ms, 2ms\}\}$, where $\{1.5ms, 2ms\}$ refers to the latency performance requirement.

Application workloads play a fundamental role in calculating utility, since these comprise distributions of transactions that arrive over a time interval and performed by the different application users [GAL16]. Lets first define the set of all workloads for a specific application as $\Omega_A = \{\omega_x\}$. These workloads arrive at different time intervals, meaning that an application receives for each time interval $\psi_m \in \Psi$ a subset

of workloads in Ω_A. Therefore, we define the set of sets of application workloads as $\mathcal{W} = \{W_k \mid 1 \leq k \leq |\Psi| \ \wedge \ W_k \subseteq \Omega_A\}$, and a function $h : \mathcal{W} \ \rightarrow \ \Psi$ that maps each set of workloads $W_k \in \mathcal{W}$ to the time interval $\psi_m \in \Psi$ where it occurs. Each $W \in \mathcal{W}$ is probabilistically distributed, meaning that each $w \in W$ has a probability to occur in a time interval ψ_m. For example, a Poisson distribution with $\lambda = 4$ may be used to describe the distribution of workloads in each $W_k \in \mathcal{W}$.

8.3.2 Utility Function

Utility functions in SCARF constitute the main pillar for decision making, and must be jointly defined by business and application architects. In the remainder of this section we formalize a utility function that focuses on calculating the economic impact of a ζ-topology. This function considers application requirements, workloads, and calculates the trade-off between its expected revenue and cost. Looking at the landscape of business applications, this utility function can be widely applied, since these typically exist to generate business revenue.

The utility function $u : T^{\zeta} \times R \times \mathcal{V} \times \mathcal{W} \times \Psi \rightarrow \mathbb{R}$, for an application ζ-topology $\tau^{\zeta} \in T^{\zeta}$, the requirements $r \in R$ and associated set of measurement values $V \in \mathcal{V}$, the set of workloads $W \in \mathcal{W}$, and its time interval $\psi \ \in \Psi$ is defined as:

$$u(\tau^{\zeta}, r, V, W, \psi) = revenue(\tau^{\zeta}, V, W, \psi) \ \times \ sat(\psi, W) - cost(\tau^{\zeta}, r, V, \psi) \tag{8.1}$$

where $revenue : T^{\zeta} \times \mathcal{V} \times \mathcal{W} \times \Psi \rightarrow \mathbb{R}$ is a function calculating the application's revenue during a time interval $\psi \in \Psi$, and $cost : T^{\zeta} \times R \times \mathcal{V} \times \Psi \rightarrow \mathbb{R}$ is a function estimating the associated resource

costs for an application viable topology $\tau^{\zeta} \in T^{\zeta}$. Functions $revenue$ and $cost$ are detailed in Sec. 8.3.3 and Sec. 8.3.4, respectively. The application's utility is influenced by the overall satisfaction of its end users, which can be calculated by a function $sat : \Psi \times \mathcal{W} \rightarrow \mathbb{R}_{\geq 0}$, and impacts the total application's revenue. If customer attrition is the main focus for measuring satisfaction, one possible way of calculating $sat(\psi, W)$ is:

$$sat(\psi, W) = \prod_{w \in W} \frac{user_{gain}(\psi, w)}{user_{loss}(\psi, w)} \tag{8.2}$$

where $user_{gained} : \Psi \times W \rightarrow \mathbb{R}_{\geq 0}$ returns the average amount of end users gained in a $\psi_m \in \Psi$, and $user_{loss} : \Psi \times W \rightarrow \mathbb{R}_{\geq 0}$ returns the average amount of non-returning (lost) end users. For instance, lets consider workloads $w_1, w_2 \in W$ observed in a time interval ψ_2. Functions $user_{gain}(\psi_2, w_1)$ and , $user_{gain}(\psi_2, w_2)$, and $user_{loss}(\psi_2, w_1)$ and $user_{loss}(\psi_2, w_2)$, can compute the estimation of users gained and lost, respectively, by observing workloads w_1 and w_2.

8.3.3 Revenue Function

The first fundamental constituent in SCARF's utility model is the $revenue$ function. The remainder of this section focuses on the viable topology $\tau^{\zeta} \in T^{\zeta}$ previously introduced for exemplification purposes. The revenue function $revenue : T^{\zeta} \times \mathcal{V} \times \mathcal{W} \times \Psi \rightarrow \mathbb{R}$ determines the monetary revenue of an application distribution for a time interval

$\psi \in \Psi$. More specifically, $revenue$ is defined as:

$$revenue(\tau^{\zeta}, V, W, \psi) = \sum_{w \in W} P(w) \cdot [((1 - \varepsilon_A)\, tpu_A(w)) \cdot |\psi| \cdot rpu_A(\psi) \cdot users(w) \cdot av(\tau^{\zeta}, \psi, V)\,] \tag{8.3}$$

where its symbols are explained as follows. $P(w)$ is the probability of receiving a concrete workload $w \in W$ in a time interval $\psi_m \in \Psi$, e.g., for w_1, $P(w_1) = 0.05$, and for workload w_2, $P(w_2) = 0.3$. The beginning of this section introduced a fixed application *A*. The constant ε_A, and functions tpu_A and rpu_A, are related to the application *A*, as these are specific to an application and to the domain where it is generating business value. In particular, function $tpu_A : W \rightarrow \mathbb{R}_{\geq 0}$ depicts the average number of transactions per end user (customer) in each workload $w \in W$, and $0 \leq \varepsilon_A \leq 1$ represents the error rate of the application A in its workload $w_k \in W$. For example, for the workloads w_1 and w_2, the average number of transactions per end user may be 1 and 3 transactions, respectively, while the average transaction error rate of the application could be $\varepsilon = 0.03$.

The function $rpu_A : \Psi \rightarrow \mathbb{R}$ is also an application-specific business function that estimates the average monetary revenue per end user during a time interval $\psi_m \in \Psi$. This function is typically developed by business architects and based on analyzing seasonal monetary revenues. One possible example could be a step function depicting the average revenue per user per transaction on a monthly basis, returning an average of 80 U$ per user for the months of December and January (Christmas period), and 55 U$ for the remainder months.

The number of end users constituting a workload $w_k \in W$ is returned by the function $users : W \rightarrow \mathbb{R}_{\geq 0}$. The availability of the underlying environment highly impacts the application's revenue. In this utility model, the availability function $av : T^{\zeta} \times \Psi \times \mathcal{V} \rightarrow [0,1]$ returns the availability in a time period $\psi_m \in \Psi$ of the cloud services in a ζ-topology $\tau_i^{\zeta} \in T^{\zeta}$. Considering the previous example and the uptime described in the SLA of Amazon Web Services (AWS)[1], both $av(\tau_1^{\zeta}, \psi_1, ...)$ and $av(\tau_2^{\zeta}, \psi_2, ...)$ are 0.9995.

Based on the previous parameter values, the revenue for τ_2^{ζ} during the time interval ψ_2, for the workloads $w_1, w_2 \in W$, for 3.5K users during ψ_2, and for a set of observed measures $V \in \mathcal{V}$ can be calculated as follows:

$$\begin{aligned} revenue(\tau_2^{\zeta}, V, \{w_1, w_2\}, \psi_2) = \\ P(w_1) \cdot [((1-\varepsilon)\, 1\, tx/user) \cdot 2\, months \cdot 80\, U\$) \\ \cdot 3.5K\, user \cdot av(\tau_2^{\zeta}, \psi_2, V)\,] + \\ P(w_2) \cdot [((1-\varepsilon)\, 3\, tx/user) \cdot 2\, months \cdot 80\, U\$) \\ \cdot 3.5K\, user \cdot av(\tau_2^{\zeta}, \psi_2, V)\,] = 27024.2 + 486435.6 \simeq 513K\, (U\$) \end{aligned} \tag{8.4}$$

8.3.4 Cost Function

A second influence factor in the utility function is associated to the resources utilization costs (see Eq. 8.1). Given a viable application topology $\tau^{\zeta} \in T^{\zeta}$, the requirements $r \in R$, and the measurement values $V \in \mathcal{V}$ for a time interval $\psi \in \Psi$, the cost function $cost$:

[1] Amazon Web Services (AWS) SLA: https://aws.amazon.com/ec2/sla/

$T^{\zeta} \times R \times \mathcal{V} \times \Psi \rightarrow \mathbb{R}_{\geq 0}$ is defined as:

$$\begin{aligned} &cost(\tau^{\zeta}, r, V, \psi) = \\ &C_{fixed}(\tau^{\zeta}, r, \psi) + \\ &C_{variable}(\tau^{\zeta}, r, V, \psi) \end{aligned} \tag{8.5}$$

Before introducing C_{fixed}, let's define the function $services : T^{\zeta} \rightarrow S(\tau^{\zeta})$ where $S(\tau^{\zeta}) \equiv \{s \,|\, s \, is \, a \, cloud \, service \, of \, \tau^{\zeta}\}$. Function $services$ essentially returns a set S containing all cloud services in an alternative topology $\tau^{\zeta} \in T^{\zeta}$. The function $C_{fixed} : T^{\zeta} \times R \times \Psi \rightarrow \mathbb{R}_{\geq 0}$ returns the fixed costs for provisioning and maintaining cloud services in a viable topology $\tau^{\zeta} \in T^{\zeta}$:

$$\begin{aligned} C_{fixed}(\tau^{\zeta}, r, \psi) = \\ C_{fixed}(services(\tau^{\zeta}), r, \psi) = \\ \sum_{s \in S(\tau^{\zeta})} C_{service}(s, r, \psi) \end{aligned} \tag{8.6}$$

where $C_{service} : S(\tau^{\zeta}) \times R \times \Psi \rightarrow \mathbb{R}_{\geq 0}$ calculates the provisioning and maintenance costs for each cloud service $s \in S$. Focusing on the Web shop application example in Fig. 8.1 distributed w.r.t. τ_2^{ζ} during the

time interval ψ_2:

$$
\begin{aligned}
&C_{fixed}(services(\tau_2^\zeta), R, \psi_2) = \\
&C_{fixed}(S(\tau_2^\zeta), r, \psi_2) = \sum_{s \in S(\tau_2^\zeta)} C_{service}(s, r, \psi_2) = \\
&C_{service}(AWS\_EC2\_t2.medium, r, \psi_2) + \\
&C_{service}(AWS\_EC2\_m4.xlarge, r, \psi_2) + \\
&C_{service}(AWS\_EC2\_t2.medium, r, \psi_2) + \\
&C_{service}(AWS\_EBS\_gp2, r, \psi_2)
\end{aligned}
\tag{8.7}
$$

where $S(\tau_2^\zeta) = \{ AWS\_EC2\_t2.medium, AWS\_EC2\_m4.xlarge, AWS\_EC2\_t2.medium, AWS\_EBS\_gp2 \}$ is the set of cloud services used in τ_2^ζ.

However, the fulfillment of the requirements in R depends on (i) the performance offered by a cloud provider, and on (ii) the application workload behavior. Therefore, we must also consider incurred costs due to the scaling of resources to satisfy every requirement in R, e.g., scaling out a virtual machine to satisfy a workload peak interval. Let's define the function calculating cloud services variable costs $C_{variable} : T^\zeta \times R \times \mathcal{V} \times \Psi \rightarrow \mathbb{R}_{\geq 0}$ as:

$$
\begin{aligned}
&C_{variable}(\tau^\zeta, r, V, \psi) = \\
&C_{variable}(services(\tau^\zeta), r, V, \psi) = \\
&\sum_{s \in S(\tau^\zeta)} \sum_{v \in V} C_{adapt}(s, r, v, \psi)
\end{aligned}
\tag{8.8}
$$

where *services* is the function previously defined. The function

$C_{adapt} : S(\tau^{\zeta}) \times R \times V \times \psi \rightarrow \mathbb{R}_{\geq 0}$ returns a sum of incurred scaling (additional) costs for every utilized cloud service $s \in S(\tau^{\zeta})$. This sum function is performed for every time interval $\psi_m \in \Psi$ to satisfy every requirement in R evaluated through each set of measurement values in V. In line with the previous example (see Fig. 8.1), a potential adaptation cost to satisfy a requirement $r_1 = database_{throughput} \geq 10\ reqs./sec.$ could consist of provisioning an additional VM instance *t2.medium* during a time interval ψ_2. More specifically, one possible definition of C_{adapt} can be:

$$C_{adapt}(s, r, v, \psi) = \begin{cases} C_{service}(s, r, \psi) & if\ ev(r, v, \psi) \\ 0 & otherwise \end{cases} \tag{8.9}$$

where $ev : R \times V \times \Psi \rightarrow \{0, 1\}$ is a function that evaluates the fulfillment of a requirement in $r \in R$ for a time interval $\psi_m \in \Psi$ and w.r.t. its set of measurement samples $v \in V$.

8.3.5 Utility Evolution

As supported by SCARM, cloud applications can be redistributed during their life time. Therefore, the utility model must consider the impact on an application's utility when redistributing its components, i.e. when adopting a new viable ζ—topology in T^{μ}. *Marginal Utility* is used to calculate the difference in utility when redistributing an application. Given the application A introduced in the beginning of the section, its marginal utility $\triangle u_A : T^{\zeta} \times \Psi \rightarrow \mathbb{R}$ for an application viable topology in $\tau^{\zeta} \in T^{\zeta}$, given its requirements in R, its corresponding

measurement values in $\mathcal{V}$, and the workloads in $\mathcal{W}$, is defined as:

$$\triangle u_A(\tau^\zeta, \psi) = u(\tau_i^\zeta, r, V, W, \psi_m) - u(\tau_{i-1}^\zeta, r, V, W, \psi_{m-1}) - cost_{red}(\tau_{i-1}^\zeta, \tau_i^\zeta) \quad (8.10)$$

where $cost_{red} : T^\zeta \times T^\zeta \rightarrow \mathbb{R}_{\geq 0}$ is a function calculating redistribution costs due to the transition from τ_{i-1}^ζ to τ_i^ζ. For instance, a redistribution of the Web shop application depicted in Fig. 8.1 from a τ_1^ζ to a new τ_2^ζ in the first week of December 2016 – previous to the Christmas season– could entail the migration of the Web shop's front-end to a cluster of two VMs, each Ubuntu-based *Amazon EC2 t2.large* using the *EBS gp2* elastic storage system for caching purposes. Therefore, redistributing the application would consist of provisioning τ_2^ζ during ψ_2. The calculation of the marginal utility (see Eq. 8.10), therefore, entails the calculation of redistribution costs $cost_{red}(\tau_1^\zeta, \tau_2^\zeta)$ for migrating the front-end to an *AWS Beanstalk* container, such as the costs produced for a planned operational downtime, which could be 2500 U$. Therefore, considering the new utility $u(\tau_2^\zeta, ...) = 25000$ U$ and the previous utility $u(\tau_1^\zeta, ...) = 15000$ U$, and an infrastructure cost $cost(\tau_2^\zeta, ...) = 5000$ U$, the marginal utility in the example is $\triangle u(\tau_2^\zeta, \psi_2) = 7500$ U$.

Focusing on calculating the utility for an application A and its lifetime Ψ, which contains multiple time intervals ψ_m, we can define the utility of an application's life time $u_A : \Psi \rightarrow \mathbb{R}$ as:

$$u_A(\Psi) = \sum_{i=1}^{|\Psi|} \lambda_i \cdot u(\tau_i^\zeta, r, V, W, \psi_i) \quad (8.11)$$

where $0 \leq \lambda_i \leq 1$ and $\sum_{i=1}^{|\Psi|} \lambda_i = 1$, are the weights reflecting the preference over the time intervals in Ψ for the life time of the application. For example, during a business year, there may be monthly periods where an application depicted by its topology T^{α} may be the associated with the primary line of business of the application owner. In such a case, $\lambda_i \approx 1$ for the corresponding months.

8.4 Discussion

The major goal of the SCARF utility model is to enhance the SCARM method with a decision making mechanism for evaluating viable ζ-topologies. In particular, the previously introduced utility model can be leveraged to orient and assist application architects w.r.t. the existence of different viable ζ-topologies in application's μ-topologies. This assistance leads to an efficient distribution and redistribution of cloud applications focusing on their cost and performance.

SCARF's utility model introduces a layer of complexity in terms of incorporating cost- and performance-related parameters, as well as the need for defining application-specific functions, such as tpu, rpu, and sat. These are indeed not trivial, and would require some mathematical acumen and sufficient data by business and application architects. However, once these are define, there is space for reusability. For instance, the satisfaction function sat can be reused for similar types of applications across different business domains, as it specifically focus on calculating the end user satisfaction when consuming an application.

As previously discussed, one possible realization of this utility model can be geared towards delivering to decision makers a ranked list of ζ-topologies based on their calculated utility. Chapter 9 follows that di-

rection, and presents the technological implementation to support the creation of such a ranked list. For validation and evaluation purposes, Chapter 10 evaluates the previously introduced utility model under different distribution scenarios and using different utility models.

Chapter 9

SCARF-T: Systematic Cloud Application (Re)Distribution Framework - Tooling Support

9.1 Chapter Introduction

Previous chapters focused on elaborating conceptual foundations for SCARF. As a recap, SCARF is introduced in Chapter 4, which proposes

a generic framework to assist application and business architects to migrate applications to the cloud. In particular, SCARF incorporates a life-cycle and a method (SCARM), both focusing on exploiting application knowledge to minimize complexity of decision making when selecting and utilizing cloud services to migrate and distribute applications. SCARF-T is essentially a technological realization of SCARF, i.e., an integrated tool chain supporting different steps in SCARM with built-in mechanisms for both its life-cycle and method.

Before introducing SCARF-T, let's first make an informal synopsis of the different steps of SCARM. Starting with a *Model Topology* step, application architects can partially or completely define their application topologies, which depict its components and relationships among them. A partial topology model is defined in the scope of this work as an α-topology, while a μ-topology represents a complete application topology model. α-topologies can be subsequently enriched in an *Enrich Topology* step, by means of specifying operational and business requirements. In the scope of this work, SCARM limits such enrichment to performance and cost information.

Once an enriched α-topology is defined, SCARF analyzes and processes this information and enters into the *Discover Viable Topologies* step. Such discovery step leverages existing application knowledge and analyzes topologies of similar applications to generate alternative distributions, each defined as a viable ζ-topology. The amount of viable topologies may be broad, hence not efficiently assisting application architects to decide among them. Utility assists decision making, and computes a value representing the expected benefit when for each viable ζ-topology. After one or multiple viable ζ-topologies are selected, the *Build μ-topology* step constructs an application μ-topology, which comprises all selected viable ζ-topologies. Then, the *Provision*

& Deploy step occurs, where a selected viable ζ-topology is instantiated. During production, the *Monitor & Analyze* step takes place, and focuses on capturing and building application performance and cost knowledge.

The remainder of this chapter is structured similarly to the sequence of steps in SCARM. This chapter opens with an architectural overview of SCARF-T. Each architectural component is subsequently elucidated, by means of introducing design principles and decisions, used technologies, integration points, etc. In general, the development of SCARF-T is based on three main pillars: (i) leveraging existing standards, (ii) maximizing reusability of technologies and tools, while developing new ones when merely necessary, and (iii) fostering collaboration with researchers.

9.2 Architectural Overview

As previously summarized, SCARF-T is a technological framework designed and developed to support SCARF. The remainder of this section presents a generic architecture for SCARF-T, and expands on every component covering different functionalities to support SCARF.

Figure 9.1 depicts an achitecture for SCARF-T. SCARF-T is designed and developed following service-oriented architecture principles[1]. In particular, tools in SCARF-T are built as a set of loosely-coupled and self-contained services, where each service is independent but can interact with others through standardized interfaces accessible over the network [GP08; WCL+05]. Services developed and adopted in SCARF-T are essentially realized as Web applications implementing

[1] Service-Oriented Architecture Standards - The Open Group: www.opengroup.org

RESTful interfaces [RR08].

There are three architectural blocks in SCARF-T: *Modeling*, *Decision Support*, and *Provisioning & Execution* (as depicted in Fig. 9.1). The *Modeling* block supports all functionalities for application architects in SCARM related to modeling and enriching cloud application topologies. In addition, the modeling and enrichment block also provides a Web graphical interface integrating all services supporting different steps in SCARF-T. A *Decision Support* block enhances modeling with decision support, by offering business and application architects with discovery and assessment mechanisms related to optimally distributing their applications in the cloud. In other words, decision support is a key ingredient in SCARF-T, since it allows to (i) capture and persist knowledge of application topologies, (ii) exploit such knowledge to discover and build alternative ζ-topologies, (iii) support creation of custom utility functions, and (iv) to use such functions to estimate the benefit of each alternative ζ-topology focusing on cost and performance requirements. Finally, tools in the *Provisioning & Execution* block serve for topology instantiation and knowledge filtering and retrieval, mostly used during the execution of applications.

The remainder of this section opens all components in each architectural block of SCARF-T, providing more details on their functionality.

Topology Modeling & Enrichment During modeling, SCARF-T provides application architects a *Topology Modeling & Enrichment* tool, which supports modeling of cloud application topologies. As discussed in Chapter 6, SCARM supports defining partial application topology models – denoted as α-topologies –, and complete application topology models – denoted as alternative ζ-topologies–. It allows to define application topologies, by depicting their components as

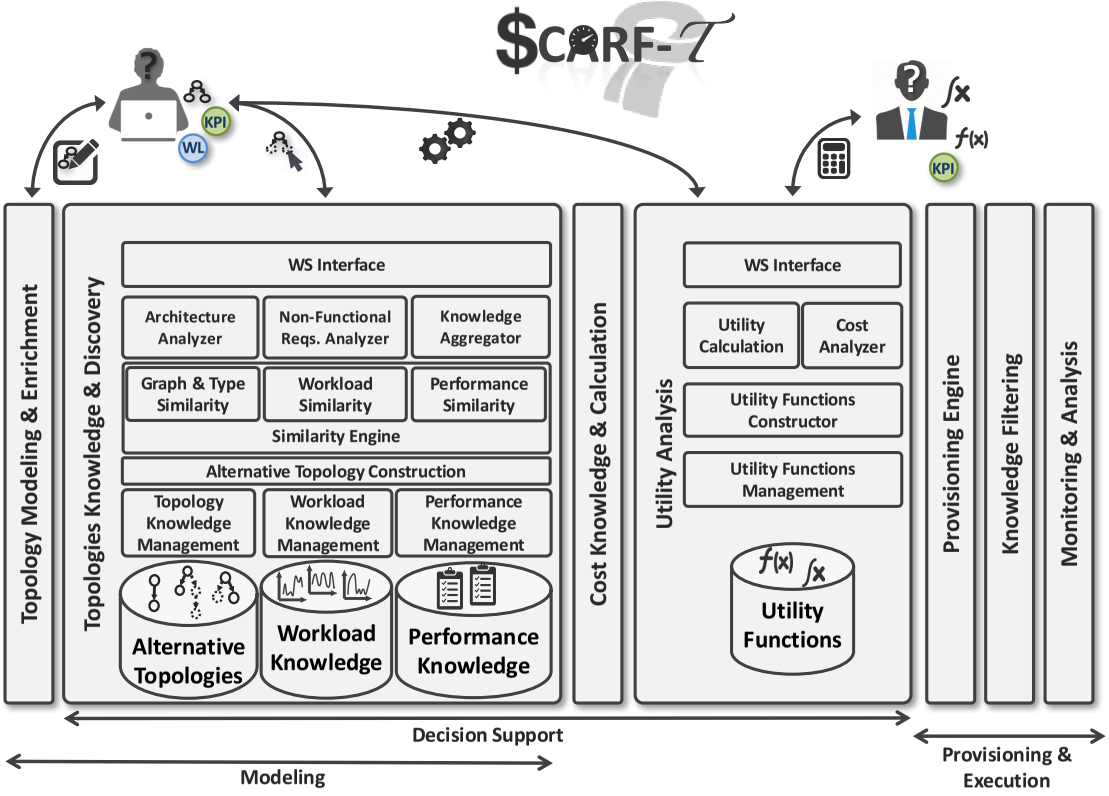


Figure 9.1: SCARF-T Architectural Overview

topology nodes and the relationship among them. Topologies can be enriched with performance and workload information. For example, specifying required throughput, response time, mean time to recover, workload behavior, number of users and transactions per month, etc. This enrichment is leveraged in subsequent decision making steps of SCARF.

The *Topology Modeling & Enrichment* tool is packaged together with a repository, which can be used to locally store topology models. Topologies can be exported and imported as packages, which can be reused in other topology modeling environments or can be instantiated using a provisioning engine. In addition, this tool also serves as a graphical interface that guides application architects through all steps in SCARM.

Topologies Knowledge & Discovery As previously denoted, application architects can partially or completely model their application topologies, by means of defining α- or ζ-topologies, respectively. Such topology models are subsequently enriched with performance and workload information. A *Topologies Knowledge & Discovery* tool processes such information and discovers alternative ζ-topologies based on existing knowledge in the following knowledge bases: (i) *Alternative Topologies*, *Workload Knowledge*, and *Performance Knowledge*. All three knowledge bases have a *Knowledge Management* interface for capturing and retrieving knowledge in them.

The *Alternative Topologies* knowledge base contains a repository of α-topologies and reusable γ-topologies, thus constituting μ-topologies for all applications used in SCARF-T. In a similar way, *Workload* and *Performance Knowledge* bases persist workload and performance knowledge for all instantiated ζ-topologies, respectively. As intro-

duced in Chapter 7, SCARF leverages similarity analysis for discovering alternative ζ-topologies. A *Similarity Engine* orchestrates similarity analysis on three levels: *Architectural*, *Non-Functional*, and *Global Similarity*.

Architectural similarity essentially comprises (i) graph similarity, and (ii) application, component, and relationship type similarity, which are introduced in Section 7.4.3. As a recap, architectural similarity first calculates similarity among types of applications, and then computes similarity for its architectural structure, depicted as an α-topology graph. Finally, it aggregates both similarities using the local-global similarity principle. The *Graph & Type Similarity* component in the *Similarity Engine* is responsible for calculating these similarities. *Non-Functional* similarity is jointly computed by *Workload Similarity* and *Performance Similarity* components. For all previous analyses, SCARF-T first extracts both architectural and non-functional information from application topologies using an *Architecture Analyzer* and *Non-Functional Requirements Analyzer*. In the case of architectural similarity, SCARF-T analyzes all relevant functional information contained in the application topology model: (i) graph structure, i.e., nodes and relationships among them, and (ii) types of nodes and relationships. The *Graph & Type Similarity* component interacts with the *Alternative Topologies* knowledge base to compute graph similarity for each topology.

Non-functional Similarity is based on calculating local and global similarity on performance and workload knowledge. For such a purpose, the *Non-Functional Requirements Analyzer* first extracts such information from topology models, which is then used in the *Similarity Engine* to compute local and global similarity. The *Similarity Engine* retrieves performance and workload knowledge of other appli-

cation topologies from the *Workload* and *Performance Knowledge* bases. These contain knowledge of application workloads and performance models for every instantiated application alternative ζ-topology. For example, an alternative topology T_1^{ζ} in the *Alternative Topologies* knowledge base contains its complete runtime performance and workload knowledge in the workload and performance knowledge bases, respectively.

Cost Knowledge & Calculation Cost analysis is one of the main pillars in SCARF for decision making. For this, a *Cost Knowledge & Calculation* tool offers two main functionalities: (i) aggregates pricing knowledge for different cloud providers and services, and (ii) exposes a cost calculation interface to estimate costs for given application profiles. Cost knowledge can be either aggregated manually in the knowledge base or crawled from cloud provider's web sites.

Pricing knowledge in the *Cost Knowledge & Calculation* is essentially based on predefined configurations for cloud services. For example, for AWS EC2[2], it contains cost knowledge for all possible configurations of AWS EC2 instances, e.g., t2.medium, m2.large, etc. Cost calculation, on the other hand, leverages such knowledge for a given application profile, e.g., number of months, location zone, etc.

Utility Analysis SCARF leverages utility to assist application and business architects to decide which cloud provider and service optimally suit their application needs (see Chapter 8). For such a purpose, a *Utility Analysis* tool is the main pillar for decision making in SCARF-T. It allows business architects to define new or reuse application specific utility functions, e.g. focusing on estimating how profitable

[2]AWS EC2: https://aws.amazon.com/ec2/

an alternative application ζ-topology can be. Application specific utility functions typically comprise multiple functions, which can be reused by other applications, covering different aspects, e.g., resources cost, workload model, etc., and can be combined into a single utility function.

The *Utility Analysis* tool essentially offers three functionalities:

1. creation of reusable functions in a *Utility Functions Constructor*, which can be shared among different applications and stored in a repository of *Utility Functions*,
2. integration with the *Cost Knowledge & Calculation* tool through a *Cost Analyzer* component, for functions requiring cost calculation, and
3. the calculation of utility for alternative ζ-topologies, by means of orchestrating newly defined or referenced functions in the *Utility Functions* repository and computing utility in the *Utility Calculation* component.

Provisioning Engine Once application architects finalize the decision making phase in SCARF, an alternative topology T^{ζ} is selected. However, the selected T^{ζ} is not yet instantiated, i.e., its components are not yet provisioned and deployed. A *Provisioning Engine* is responsible for materializing an alternative ζ-topology. More specifically, a *Provisioning Engine* orchestrates provisioning and deployment of all application components, by means of inferring nodes and relations in its topology. For example, if an *Apache HTTP Server* is hosted on an *AWS EC2 m1.medium* VM, the latter is first provisioned and the former is subsequently installed. In addition, a provisioning engine ensures that each application component is configured accordingly

to its topology model description, e.g., access keys, application ports, auto scaling configuration, etc.

Monitoring & Analysis After provisioning and deployment, applications are finally accessible to end users. During the execution phase, it is of key importance to monitor all indicators to analyze an application's performance. A *Monitoring & Analysis* tool has two fundamental goals: (i) to monitor all available application metrics, e.g. all related to cost, performance, workload, etc., and (ii) to provide an interface for application and business architects to continuously observe and analyze different aspects of their applications in production. Metrics retrieved during monitoring can be aggregated and translated into knowledge that can be leveraged for decision making in SCARM, e.g., for optimizing the process of finding further alternative ζ-topologies for the same or similar applications.

Knowledge Filtering Since SCARF-T *Monitoring & Analysis* aims at monitoring all possible metrics, it is necessary to filter performance and workload information, which are relevant only for the discovery and analysis of alternative application ζ-topologies in SCARF. A *Knowledge Filtering* tool filters all relevant performance and workload information of SCARF-T, and translates it into a knowledge format supported in SCARF-T.

9.3 Topology Modeling

The *Topology Modeling & Enrichment* tool in SCARF-T is one of the key tools for application architects, as it allows them to partially or completely model their application topologies. In addition, this tool

is essentially used as the main graphical interface in SCARF-T, as it is integrated with all underlying tools supporting all steps in SCARF.

Figure 9.2: Winery Repository - Visual Interface

The implementation of the *Topology Modeling & Enrichment* tool is based on OpenTOSCA[3] [KBBL13]. OpenTOSCA is a technological materialization of the TOSCA standard, which gained in the last years a considerable traction and is widely used in both research and industry [KBBL13; BBH+13; BBKL14b]. OpenTOSCA is delivered as a set of Web applications covering the complete life cycle of TOSCA-based cloud applications [KBBL13; BBH+13; BBKL14b]. In short, OpenTOSCA consists of three main applications: *Winery Modeling Tool*, *OpenTOSCA Container*, and *Vinothek Self-service Portal*. SCARF-T particularly adopts the first two. The remainder of this section focuses on describing adaptations in the OpenTOSCA *Winery Modeling Tool* (referred as Winery in the remainder of this section) for SCARF-T. Winery is a Web-based modeling environment that allows modeling of TOSCA-based cloud application topologies. Summarizing TOSCA, it is based on defining component-related types and templates [BBKL14a].

[3] OpenTOSCA: `http://www.iaas.uni-stuttgart.de/OpenTOSCA/`

Application topologies are specified as service templates comprising an application's topology, node and relationship types, and management plans. An extended introduction of TOSCA is provided in Chapter 2.

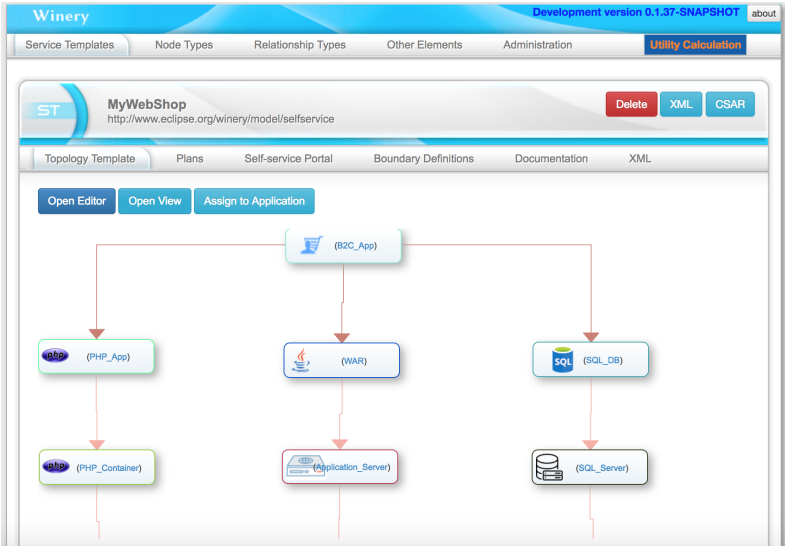


Figure 9.3: Winery Repository - Web Shop Application

Winery is built upon three components: a *Repository*, an *Elements Manager*, and a *Topology Modeler*. Focusing on the modeling phase in SCARF, the *Repository* and *Elements Manager* components provide a graphical Web interface and support the storage of all necessary application artifacts, such as service templates, node and relationship types, management plans, deployment artifacts, etc. (see Fig. 9.2). In addition, such interface also implements visualization elements for already modeled topologies, as shown in Fig. 9.3. Winery's *Topology Modeler* is an independent Web application that can be accessed from

the Winery repository, and supports the actual modeling of partial or complete application topologies (see Fig. 9.4). Application architects can easily access both Winery repository and modeler locally or remotely using a Web browser. The *Topology Modeler* contains a palette of available node types that can be used to model application topologies using a drag & drop functionality, as well as a graphical interface to configure node templates and their corresponding relationships.

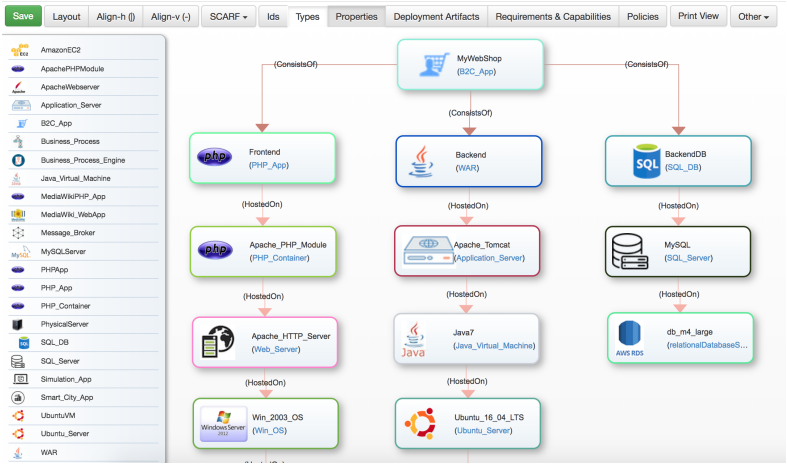


Figure 9.4: Topology Modeling Tool - Web Shop Application

As Winery already comes with a solid foundation for modeling application topologies, SCARF-T effectively adopts it and builds atop the necessary graphical interface extensions for interacting with the different tools of the SCARF-T framework. For instance, application architects can leverage functionalities of SCARF-T by means of accessing its functionalities under the SCARF navigation bar (see Fig. 9.4). Winery is extended to support modeling of partial topologies, denoted

as α- and γ-topologies, and modeling of alternative ζ-topologies, and their aggregation into μ-topologies. When modeling partial topologies, the SCARF-T implementation in Winery integrates with the *Topology Modeling & Enrichment* tool for discovering and building alternative ζ-topologies. Focusing on business architects, Winery is also extended to support the creation and configuration of utility functions within the *Utility Calculation* module (see Fig. 9.2).

Summarizing Winery's underlying technology in SCARF-T, both *Topology Modeler* and *Elements Manager* interfaces are built using Hypertext Markup Language (HTML) 5[4], JavaScript[5], Java 8[6] and JavaServer Pages (JSP)[7] technologies. The integration of both interfaces with SCARF-T tools is done using Representational State Transfer (REST) and Asynchronous JavaScript and XML (AJAX)[8]. Winery's *Repository* is built as a RESTful application using Java API for RESTful Web Services (JAX-RS) and Java. Such RESTful interface allows the storage and retrieval of TOSCA artifacts, such as Cloud Service Archive (CSAR) packages.

9.4 Topology Enrichment

The previous section focused on introducing Winery and corresponding extensions built atop of it in SCARF-T. In summary, it solely focused on modeling functional aspects in application topologies. A second step in SCARM relates to enriching cloud application topologies with

[4] HTML5: https://www.w3.org/TR/html5/

[5] JavaScript: https://www.javascript.com/

[6] Java 8: http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html

[7] JSP: http://www.oracle.com/technetwork/java/index-jsp-138231.html

[8] AJAX: https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

performance and workload information, as discussed in Chapter 5. Both performance and workload enrichment support are depicted in the remainder of this section, and are built atop of the *Topology Modeler* in OpenTOSCA Winery.

9.4.1 Performance Enrichment

In summary, Chapter 5 provided a model for the performance enhancement of cloud application topologies. Such enrichment model allows application architects to specify during design phase application performance requirements. Chapter 5 characterized performance attributes, such as resource capacity and utilization, elasticity, and availability, that are leveraged in SCARF for decision making.

Particularly for defining application capacity performance requirements, Fig. 9.5 shows a graphical interface built in the Winery *Topology Modeler* in SCARF-T. In a similar way, SCARF-T extends such *Topology Modeler* to allow defining all performance attributes contemplated in SCARF. This extension is mainly developed within the *Topology Modeler* using JSP[9] tags and jQuery[10] support in JavaScript[11].

The technical realization for enhancing cloud application topologies with performance requirements in SCARF-T is fully presented in [GAL16] and is based on Policy4TOSCA definitions [WWB+13]. Policy4TOSCA is an extension of the TOSCA specification that allows attachment of non-functional requirements to TOSCA service templates. In detail, Policy4TOSCA enables the definition of *Policy Types* and *Policy Templates* comprising actions which must be per-

[9] JavaServer Pages: http://www.oracle.com/technetwork/java/index-jsp-138231.html
[10] jQuery: https://jquery.com/
[11] JavaScript: https://www.javascript.com/

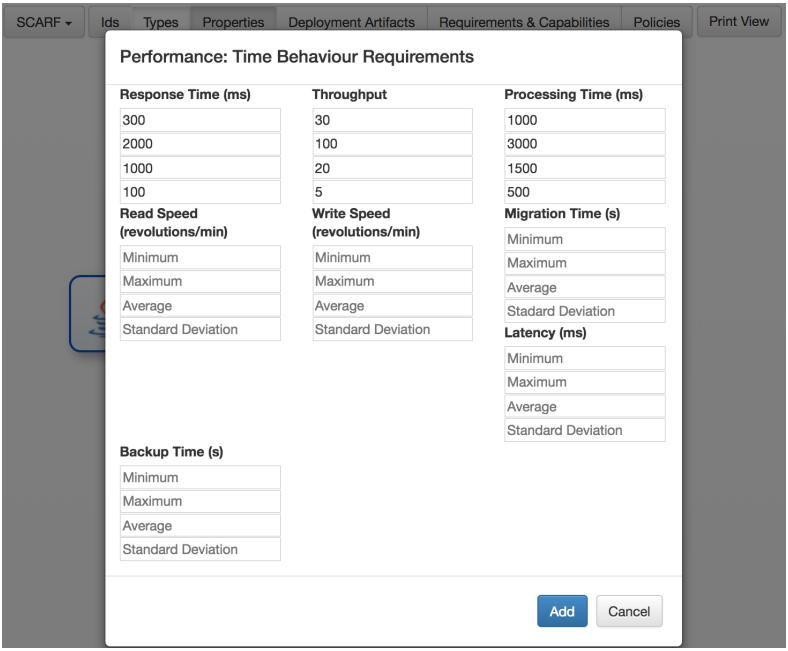


Figure 9.5: Topology Modeling & Enrichment Tool - Performance Enrichment

formed at concrete phases of an application's life cycle [WWB+13]. Policy4TOSCA policies can be attached to TOSCA templates, which can be subsequently included in exported CSAR packages. At the moment, SCARF-T does not fully profits from processing policies in CSAR packages, but utilizes such information to build application performance knowledge in its *Topologies Knowledge & Discovery* tool. This knowledge is utilized in SCARF for discovering alternative ζ-topologies.

9.4.2 Workload Enrichment

In a similar way as for performance requirements, SCARF-T supports enrichment of topology models with workload information. Chapter 5 identified and categorized application workload attributes relevant to SCARF, such as workload patterns, user arrival distribution, user behavioral model, average number of users and transactions, etc. In subsequent steps of SCARF, such information is processed as part of similarity analysis, in order to discover and build alternative ζ-topologies.

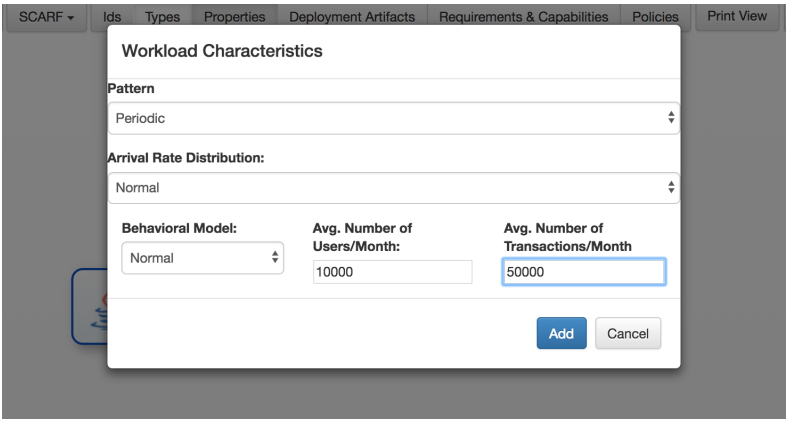


Figure 9.6: Topology Modeling & Enrichment Tool - Workload Behavior Enrichment

The workload enrichment graphical interface in SCARF-T's *Topology Modeling & Enrichment* tool is depicted in Fig. 9.6. Such interface is built as an HTML form atop of Winery's *Topology Modeler* using JSP tags and JQuery. By these means, application architects can enrich their cloud application topology models with workload information.

Enriched cloud application topology models can be stored locally w.r.t. the machine hosting Winery, i.e. in a local development machine, or in a remote server hosting Winery.

9.5 Alternative Topologies Knowledge Base

Previous sections focused on the tooling support for modeling and enriching cloud application topologies in SCARF-T. SCARF utilizes enriched cloud application topology models to discover and build alternative ζ-topologies. The next logical step in SCARF would be to introduce the technical support in SCARF-T to discover and build alternative ζ-topologies. However, in order to provide such a capability, SCARF requires technical support to persist and retrieve knowledge for α-, γ-, ζ-, and μ-topologies.

Chapter 6 introduced a formal model for viable cloud application topologies based on graph theory. SCARF-T leverages graph databases for storing, processing, and retrieving application topologies. Graph databases have gained interest in the last years due to their native support for storing and retrieving information represented as graphs [RWE13]. Moreover, graph databases are highly optimized for doing operations on graphs, such as graph isomorphism. Therefore, the architectural decision for using a graph database as the underlying mechanism for storing and processing knowledge about alternative topologies relies on features that graph databases offer out-of-the-box. In particular, SCARF-T uses Neo4j[12], which is a state-of-the-art graph database management system, according to the *Database Engines*

[12]Neo4j Graph Database: `https://neo4j.com/`

Ranking[13].

This section introduces an underlying graph database model for persisting and processing α-, γ-, and μ-topologies, developed in collaboration with the research work [Din16]. This data model is also extended to support enhancement of application topologies with performance and workload information (see Chapter 5). In a first step, this section introduces a graph database notation used in Neo4j, which is then used throughout this chapter. Subsequently, it presents a graph database data model for persisting viable topologies, which is then enhanced for capturing performance characteristics.

9.5.1 Notation

Figure 9.7 represents a graphical notation used in Neo4j and throughout this chapter. *Nodes* are used to represent entities, which can contain one or multiple *Properties*. A property is basically a key-value pair. Each node can be assigned one or more *Labels*, which assigns roles or types to each node, typically used to group nodes with similar characteristics.

Nodes are interconnected in an acyclic graph through *Relationships*. Each relationship has a source and end node, representing the direction of a relationship. Relationships can also contain properties, although these are not explicitly denoted in Figure 9.7.

9.5.2 Topology Graph Database Model

As previously denoted, developing a graph database model capable of supporting the persistence and discovery of viable topologies in

[13] Database Engines Ranking:
https://db-engines.com/en/ranking/graph+dbms

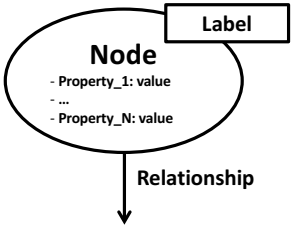


Figure 9.7: Graph Database - Notation

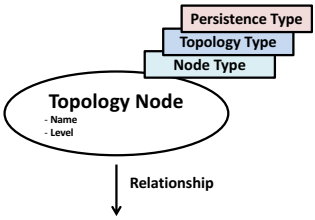


Figure 9.8: Topology Persistence - Node Definition

SCARF is fundamental. Prior to introducing the data model for α-, γ-, ζ-, and μ-topologies, this section firstly presents in Figure 9.8 an underlying definition of typed nodes used for persisting application topology components in SCARF.

Topology nodes comprise a set of *Properties*, *Labels*, and *Relationships*. Properties are used to aggregate application component attributes and properties. In particular, two properties are used: a (i) *Name* property to identify a specific component, and a (ii) *Level* attribute complements a node definition to identify if such node is a root or leaf in a topology graph.

As described in the previous section, labels are used to assign one or multiple types to nodes. For node definitions in Fig. 9.8, three labels

are used: *Node Type*, *Topology Type*, and *Persistence Type*. A *Node Type* represents the type of a component in a topology. For instance, an Apache Tomcat server is a Servlet Container used for deploying Java-based Web applications. Therefore, it is of type *Java Servlet Container*. *Topology Type* labels are used to group nodes as part of α- and γ-topologies, i.e., application specific and reusable components, respectively.

SCARF fosters reusability of topologies, by means of discovering and constructing alternative ζ-topologies based on application functional and non-functional aspects, and existing knowledge. For such a purpose, it is necessary to handle, structure, and relate different types of nodes in graphs used for defining an application topology. *Persistence Type* labels are used to identify different types of nodes in the data model for viable topologies. In particular, these can be as follows:

- *Abstract Nodes* represent generic application components or services, such as a Web Server. Abstract nodes are basically used to represent components, which are part of reusable γ-topologies.
- *Concrete Nodes* go a step further in specializing abstract nodes in γ-topologies. In particular, these describe a concrete implementation for an abstract node. For instance, an Apache HTTP Server is an implementation of a Web server.
- *Instance Nodes* are instances of concrete nodes. More specifically, instance nodes are specific to applications, and represent nodes belonging to specific α-topologies and viable application ζ-topologies.
- *Topology Index Nodes* are used to identify instance nodes that

belong to a concrete application viable topology, and contain the corresponding technology-specific description. For instance, if TOSCA [BBKL14a] is used to instantiate an application ζ-topology, then its topology index nodes contain its corresponding TOSCA Extensible Markup Language (XML) specification.

- *Requirement Nodes* are used to describe technical requirements of applications in their corresponding α-topologies. These are then matched to capabilities exposed by γ- topologies during the discovery of μ-topologies and construction of alternative ζ-topologies.
- *Capability Nodes* depict technical capabilities of components in γ- topologies. For instance, an Apache Web server supports Hypertext Transfer Protocol (HTTP) protocol, and XML and JavaScript Object Notation (JSON) messaging formats.
- *Workload Nodes* capture workload information of instantiated application ζ-topologies.
- *Performance Nodes* capture performance evolution of instantiated application ζ-topologies, and are typically associated with instance nodes of a viable distribution.

The remainder of this section explores and exemplifies the previously introduced graph database modeling artifacts for storing and processing each part of alternative ζ-topologies.

9.5.3 α-topology Graph Database Model

As introduced in Chapter 6, α-topologies define an application-specific stack, i.e., components and relations specific to an application. These

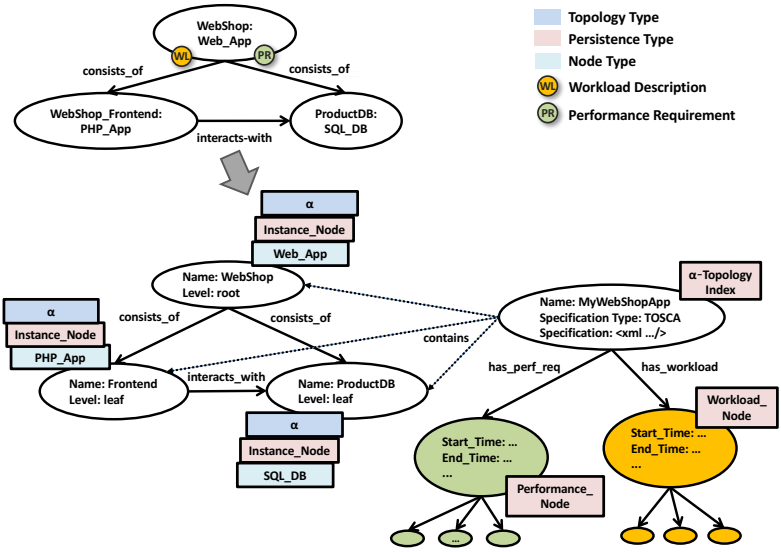


Figure 9.9: Topology Data Model - α-topology Example

can be enriched with performance information, i.e., business and operational performance requirements, and workload behavioral models.

Figure 9.9 depicts a data model for persisting α-topologies, using a Web shop application as example. In particular, α-topologies are labeled as α and specified as instance nodes, since instance nodes are meant to capture information specific to an application. This Web shop application is a two-tiered application with a front-end of type *PHP_App* and a back-end database of type *SQL_DB*. These are related as depicted in Fig. 9.9.

So far, this α-topology is labeled using instance nodes, which support the persistence of α-topologies in a technology agnostic man-

ner. However, production systems typically depend on specific technologies for provisioning and deploying applications. Therefore, it is necessary to connect such technology agnostic topology model with its corresponding technical description, e.g. TOSCA *Service Templates* [BBKL14a]. *α-Topology Index Nodes* are similar to instance nodes, and are used to connect α-topology models with technology specific descriptions, by means of relating these to instance nodes constructing an α-topology (see Fig. 9.9).

The Web shop application's α-topology depicted in Fig. 9.9 can be enhanced with performance requirements, by means of defining workload and performance nodes. These are then related to α-topology index nodes using specific relations, such as *has_performance_req* (see Fig. 9.9). Performance and workload nodes capture the complete history of performance and workload requirements for each α-topology, and are further explained in Sec. 9.5.4.

9.5.4 Performance & Workload Graph Database Model

This chapter defined so far a graph-based data model for persisting and processing functional aspects of α-topologies. Application non-functional aspects, such as performance requirements and workload descriptions are not yet covered, and are necessary in SCARF for discovering and constructing μ- and generating ζ-topologies. This section extends the previous α-topology graph database model and enhances it with support for defining performance and workload information for the life time of applications. Application performance and workload descriptions are defined when modeling application α-topologies, and relate to an application's α-topology index node, by means of defining *Performance* and *Workload Nodes*. Figure 9.10

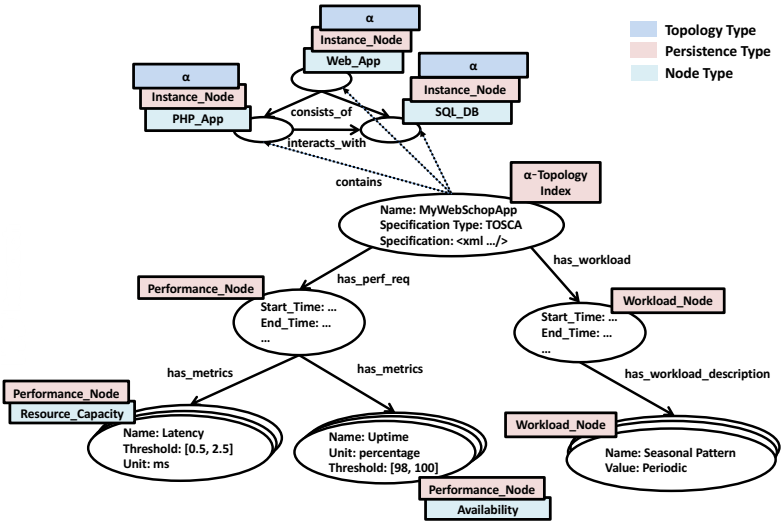


Figure 9.10: Topology Data Model - Performance Requirements & Workload Specification

exemplifies the definition of performance requirements and workload descriptions using the Web shop application as example.

Performance Nodes define two properties: (i) a time interval when such performance information applies, and (ii) a category to which each performance measure belongs to. Defining performance requirements and knowledge for different time intervals in the application life time allows to specify, e.g., seasonal requirements. For a Web shop application, it might be necessary to define stronger performance requirements during peak shopping seasons, such as Christmas. Focusing on the Web shop application in Fig. 9.10, a time interval for a group of performance requirements and knowledge is defined us-

ing the relationship *has_perf_req* and a *Performance Node*. Expected performance measures, such as the ones related to latency or uptime, are related to a specific time interval, e.g., from January to March 2017, by defining a relationship *has_measure_req*. As defined in Chapter 5, performance metrics can be grouped into different categories, therefore simplifying their specification by application architects. For categorizing such metrics, the graph database model leverages the usage of node labels for defining categories for different metrics, such as resource capacity or availability categories for latency and uptime, respectively.

Workload Nodes, on the other hand, capture the workload behavior in different time intervals. These nodes are related to α-topologies using a relation *has_workload*, and essentially capture the workload attributes introduced in Chapter 5, such as workload's seasonal pattern, user arrival distribution, average number of users, etc. Focusing on the Web shop application in Fig. 9.10, a workload node firstly defines a time interval using the relation *has_workload*, which is then related to a subsequent workload node that defines a specific workload attribute, such as its seasonal pattern.

9.5.5 γ-topology Graph Database Model

So far, previous sections presented a graph database model for persisting and processing enriched application specific α-topologies. SCARF, however, also focuses on exploiting knowledge of non-application specific topologies, defined as γ-topologies. γ-topologies are reusable topology fragments, which appear in multiple application topologies, and are used to construct application μ-topologies in combination with α-topologies. Using these, SCARF can then discover alternative

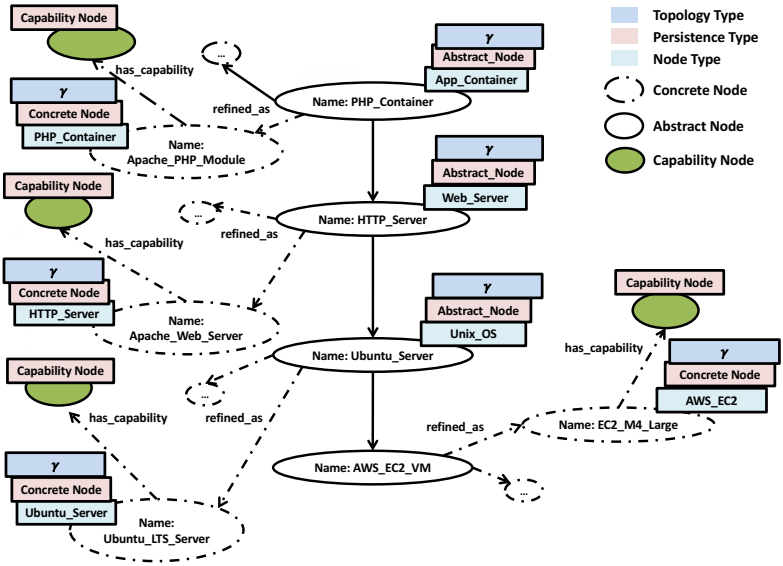


Figure 9.11: Topology Data Model - γ-topology Example

ζ-topologies. Informally, γ-topologies typically represent reusable middleware components, Operating System (OS), and cloud services, which can be reused among applications.

Persistence and processing γ-topologies introduce a higher degree of complexity, since these must naturally support reusability. As such, γ-topologies are defined using inheritance of components, which are usually depicted as nodes, in μ-topologies. Thus, reusable components and services in γ-topologies are represented in this topology graph database model using *Abstract Nodes* and *Concrete Nodes*. Figure 9.11 illustrates a graph database model for persisting γ-topologies, using a reusable sub-topology as example. In particular, this sub-topology

depicts a PHP container hosted on an HTTP server, which is subsequently hosted on an Ubuntu server in an AWS EC2 VM. In principle, this sub-topology could be reused by all applications requiring a PHP container.

Generic application components in γ-topologies are represented using abstract nodes (see Fig. 9.11). Abstract nodes provide a generic representation of components, such as a *PHP Container* or an *HTTP Server*, as depicted in Fig. 9.11. These types of components can be used to host all applications requiring a PHP application container. Concrete implementations of abstract nodes, e.g., an Apache PHP Module implementation of a PHP container in Fig. 9.11, are represented using *Concrete Nodes*. An Apache PHP module is one possible implementation of a PHP container. Concrete nodes are connected with abstract nodes using relationships of type *refined_as*.

Lastly, *Capability Nodes* in γ-topologies expose capabilities of concrete nodes. More specifically, capability nodes are mainly used during discovery and construction of μ-topologies, which are discussed in the following Sec. 9.6. *Capability Nodes* are used to match technical requirements of α-topologies with technical capabilities of γ-topologies. For instance, a capability node of an Apache Web Server may define supported messaging protocols or encryption mechanisms.

The remainder of this chapter focuses on going further in the SCARF process, by means of presenting its underlying technological support for discovering viable topologies, for supporting decision making using utility, and provisioning and monitoring instantiated alternative ζ-topologies.

9.6 Discovery of Viable Topologies

A main advantage of SCARF consists of automatically discovering alternative ζ-topologies, each representing a viable application distribution. As a summary of Chapter 7, application architects are not required to model complete cloud application topologies, but to model α-topologies, representing components and relationships specific to applications. Subsequently, SCARF leverages CBR and *Similarity Analysis* to construct μ-topologies and derive from these alternative ζ-topologies, which can then be selected for provisioning and deploying their applications. In particular, Chapter 7 introduced a discovery method that leverages knowledge of similar applications previously deployed in the cloud.

Similarity analysis and CBR in SCARF focus on two main application aspects: *Architectural* and *Non-Functional*. In short, architecture similarity analysis evaluates application architectural characteristics, such as the number of components, types and relationship among them, etc., while non-functional analysis targets performance and cost aspects among applications. The remainder of this section introduces an underlying technological support in SCARF-T developed to support both types of analyses in SCARF. Such implementation has been driven as part of collaborative research works in [Din16] and [Pin16]. Firstly, this section presents an overview of such functionalities visually integrated in the *Topology Modeling & Enrichment* tool's Web interface, and subsequently introduces the technicalities in the *Topologies Knowledge & Discovery* tool for performing both architectural and non-functional analysis to discover ζ-topologies.

9.6.1 Modeling Interface Overview

As previously discussed, SCARF-T is delivered with a *Topology Modeling & Enrichment* tool based on OpenTOSCA Winery. It provides a graphical Web interface for application architects to perform all tasks supported in SCARF.

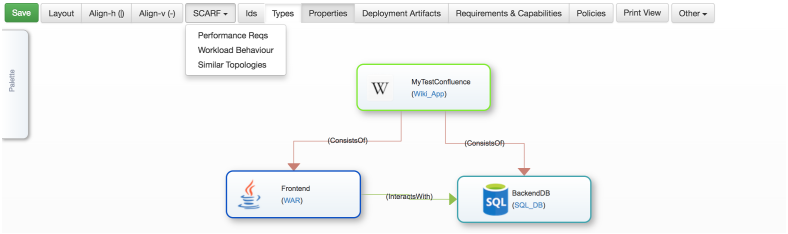


Figure 9.12: Topology Modeling & Enrichment Tool - α-Topology

Figure 9.12 shows the *Topology Modeling & Enrichment* tool in SCARF-T, using an α-topology of a Wiki application as example. In this modeling environment, application architects can model their application specific α-topologies. A SCARF menu contains the entry point for discovering similar topologies for a given α-topology. Once in this entry point, a dialog lists all alternative ζ-topologies (see Fig. 9.13). In it, a detailed view of similarity values for workload and performance similarity is presented, as well as the possibility to access application knowledge related to each alternative ζ-topology. Moreover, the *Topology Modeling & Enrichment* tool also supports refinement, cost calculation, and utility calculation for each application's alternative ζ-topology, which are introduced in the following sections.

As previously denoted, SCARF-T extends the native implementation of OpenTOSCA Winery. In all cases, the *Topology Modeler* serves as

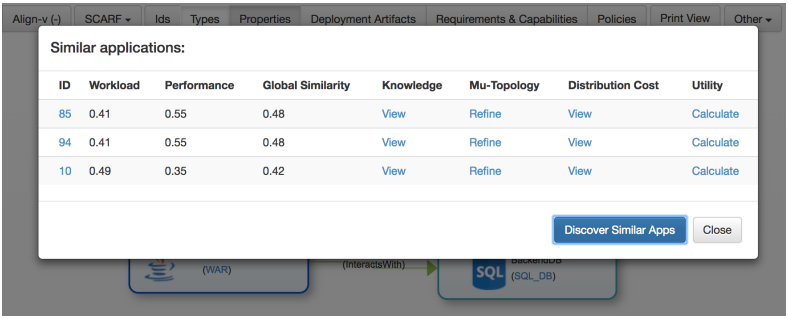


Figure 9.13: Topology Modeling & Enrichment Tool - Similarity Analysis

a graphical Web interface, which is integrated to back-end services implementing different functionalities in SCARF-T. Extensions in the *Topology Modeler* are developed in JavaScript[14] and integrated using JSP[15] tags and jQuery[16]. AJAX[17] is used as integration framework for communicating with back-end RESTful services of SCARF-T, e.g. related to the *Topologies Knowledge & Discovery*, *Cost Knowledge & Calculation*, or *Utility Analysis* tools.

9.6.2 Architectural Similarity

Architectural similarity among applications in SCARF is essentially driven in two steps. Considering an application α-topology as input, SCARF firstly analyses the similarity between the given application

[14] JavaScript: https://www.javascript.com/
[15] JavaServer Pages: http://www.oracle.com/technetwork/java/index-jsp-138231.html
[16] jQuery: https://jquery.com/
[17] AJAX: https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

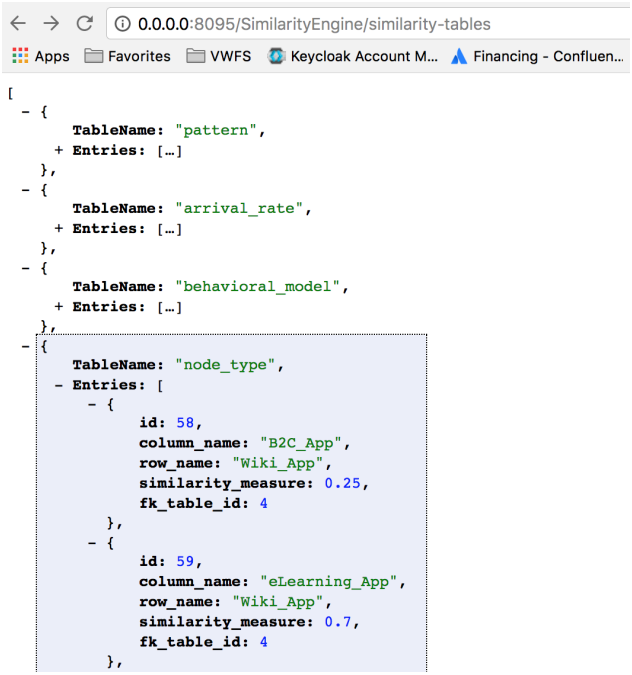


Figure 9.14: Similarity Table - Application Type

and applications in the knowledge base. In particular, SCARF first computes similarity among application types. Chapter 7 introduced the usage of so-called *Similarity Tables*, which are maintained by domain experts, and define how similar applications are. In a second step, SCARF analyses the structural similarity among application architectures (depicted as α-topologies) in the knowledge base, by means of computing graph similarity.

The functional similarity steps previously introduced are executed

in a *Similarity Engine* (see Fig. 9.1). The *Similarity Engine* contains logic and interfaces for calculating local and global similarities. This engine computes local similarity for individual application aspects, such as similarity among type of applications, and subsequently computes a global similarity that encompasses all local similarities. The *Similarity Engine* is developed as a Java-based RESTful service using Spring[18] and MySQL[19]. Similarity tables are used for calculating similarity among types of applications. These are persisted in MySQL databases and a RESTful interface allows their continuous update by domain experts. A sample similarity table HTTP response is depicted in Fig. 9.14.

Structural similarity of α-topologies in SCARF-T is also calculated in the *Similarity Engine*, which leverages graph matching functionalities offered in the Neo4j[20] graph database. More specifically, SCARF-T computes graph matching between a given α-topology and all α-topologies in the alternative topologies knowledge base. Essentially, calculating similarity among α-topologies consists of analyzing similarity (i) among their architectures, e.g. two- or three-tiered, and (ii) among their application components, as discussed in Section 7.4.3. A RESTful interface atop of the graph database system coordinates both similarity analyses and executes graph database queries developed in the Cypher[21] language of Neo4j.

[18] Spring: `https://spring.io/`
[19] MySQL: `https://www.mysql.com/`
[20] Neoj4: `https://neo4j.com/`
[21] Neo4j Cypher: `https://neo4j.com/developer/cypher/`

9.6.3 Non-Functional Similarity

Chapter 7 introduced conceptual foundations for processing local and global similarities of performance and workload knowledge of applications. In short, local similarity among performance and workload is computed based on numeric metrics and non-numeric characteristics. For the former, the *Similarity Engine* retrieves metrics from the *Performance Knowledge Base* for each α-topology and computes a local similarity for each metric. For the latter, the *Similarity Engine* retrieves workload numeric and non-numeric values from the *Workload Knowledge Base*, and computes local similarities. In the case of non-numeric values, the computation is mainly done using similarity tables.

In all cases, calculation of local and global similarity calculation is implemented in the *Similarity Engine*, which is developed as a Java-based RESTful application using Spring and MySQL. Similarity tables, e.g., used for calculating similarity among workload characteristics, are persisted in a MySQL database, and a RESTful interface allows their continuous update by domain experts. The *Similarity Engine* also integrates a RESTful interface atop of the *Workload Knowledge* and *Performance Knowledge* bases, in order to retrieve workload and performance historical data, respectively. Such knowledge is then processed in local and global similarity functions implemented in the *Similarity Engine*. This implementation is realized as Java packages that compute similarity for numeric and non-numeric attributes.

Subsequently to calculating similarity in the *Similarity Engine*, SCARF-T provides application architects in the *Topology Modeling & Enrichment* tool with alternative ζ-topologies, each representing a viable application distribution. For each alternative ζ-topology, ap-

plication architects can analyze (i) incurred infrastructure costs, e.g. generated costs for using an AWS EC2 instance for a given period of time, and (ii) expected utility when selecting a specific alternative ζ-topology. The remainder of this chapter addresses such support in SCARF-T, by means of introducing cost and utility calculation support, used as the basis for decision support in SCARF.

9.7 Cost Knowledge & Calculation

SCARF aims at supporting decision making tasks related to selecting viable application distributions, denoted as alternative ζ-topologies. Chapter 8 introduced a utility model used as the basis for decision making support, which incorporates cost calculation as one of its pillars. The remainder of this section introduces the technological support in SCARF-T for cost knowledge, analysis, and calculation.

SCARF-T adopts Nefolog as framework for supporting cost knowledge and calculation. In summary, Nefolog is developed in [ARML14; XA13] as a python-based Web application with the following characteristics: (i) contains cost knowledge for different cloud providers and offerings, (ii) implements a cost calculation engine for given application profiles, and (iii) offers a RESTful API that exposes all pricing knowledge and cost calculation functionalities [XA13]. Nefolog captures pricing knowledge for 6 cloud providers, such as AWS, Google, Microsoft Azure, etc., and a total of 58 offerings. Each offering is associated with a cloud service, e.g. AWS RDS or Microsoft Azure Compute. Since pricing also varies within cloud services, Nefolog incorporates pricing knowledge for 520 configurations of specific cloud services. Such knowledge is leveraged for calculating costs when using a specific cloud service and configuration. For instance, Fig. 9.15

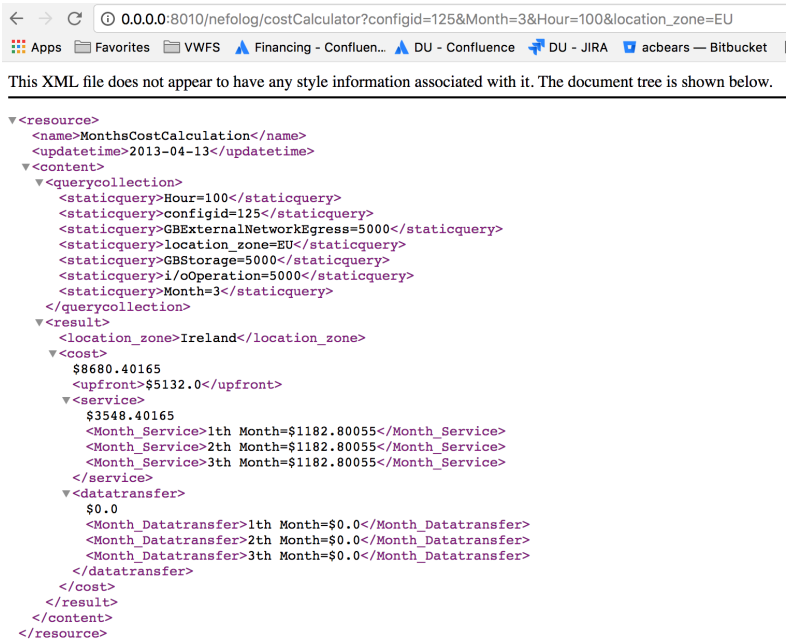


Figure 9.15: Nefolog - Cost Calculation

depicts the XML response containing the price calculation for an AWS EC2 reserved instance in the eu-west-1 (Ireland) region and for a duration of three months. Given an application profile, e.g. number of months, hourly usage, location zone, among others, Nefolog's cost calculation engine can estimate monthly costs for a specific cloud service configuration.

SCARF-T integrates Nefolog for fulfilling two objectives. Firstly, the *Topology Modeling & Enrichment* tool in SCARF-T provides a graphical Web interface for application architects to visually calculate costs for

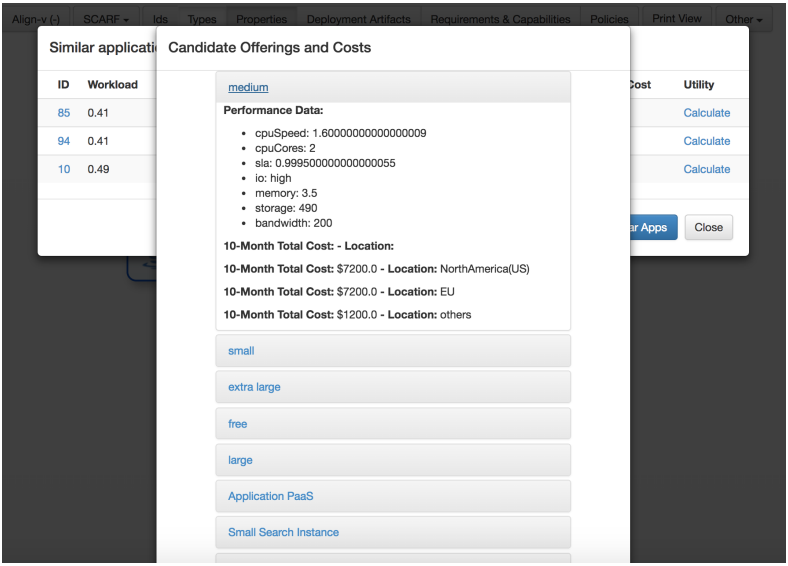


Figure 9.16: Topology Modeling & Enrichment Tool - Cost Analysis

each alternative ζ-topology (see Fig. 9.16). In such, cloud offerings are shown for a given application alternative ζ-topology, with an indicative cost estimation. Secondly, utility models used in SCARF incorporate cloud offerings cost calculation as one of its pillars. Therefore, SCARF-T integrates its *Utility Analysis* tool with Nefolog in order to calculate cloud offerings costs for each alternative ζ-topology. This integration is detailed in the remainder of this chapter.

9.8 Utility Analysis

As previously introduced in Chapter 8, SCARF leverages utility as its underlying mechanism for decision making support. In practice, utility is used to measure perceived satisfaction when consuming a good or service [Mar09]. Chapter 8 formally introduced a potential utility model for evaluating expected monetary profitability when distributing cloud applications w.r.t. specific ζ-topologies. This utility model introduces computation of revenue, end-user satisfaction, and cost calculation functions, which are combined in a single utility function.

SCARF-T supports such utility model in its *Utility Analysis* tool and goes a step further, by means of allowing application and business architects to custom and visually define, reuse, and compute revenue, satisfaction, and cost functions. Such support, however, requires a *Utility Analysis* tool to (i) provide a repository of reusable functions, (ii) interpret functions depicted in a specific syntax, (iii) interpret function parameters in the specific functions, and (iv) provide an interface to interact with the *Topology Modeling & Enrichment* tool.

The remainder of this section firstly introduces a technical overview of the *Utility Analysis* tool, which is then opened to present its different functionalities to represent, process, and calculate utility functions. Finally, this section closes describing the integration between the *Topology Modeling & Enrichment* and *Utility Analysis* tools.

9.8.1 Overview

Section 9.2 introduced an architectural overview of the *Utility Analysis* tool in SCARF. In general, the *Utility Analysis* tool is developed as a Java Web application exposing a RESTful API to manage and compute

utility functions. Its Web service interface implementation is based on JAX-RS[22] and Java Architecture for XML Binding (JAXB)[23], and its logic is developed in Java. Its repository for utility functions is based on MySQL[24], and is mainly used for persisting reusable functions. The *Utility Analysis* tool has been developed in collaboration with research works presented in [Fre16] and [GAB+18].

The *Utility Functions Constructor* in the *Utility Analysis* allows to define *Functions* as reusable components, which are then referenced in *Utility Sub-Functions*. Functions can be of a specific type, such as revenue, cost, or user satisfaction, as discussed in Chapter 8. Utility sub-functions are then the main building blocks for *Utility Functions*, and allow to use all available revenue, satisfaction, and cost functions in its repository. In short, utility sub-functions connect utility functions with revenue, cost, and satisfaction functions. Finally, functions typically incorporate parameters, which can be of different value types, e.g. integer, float, etc., or might be retrieved from an external source or service. For the latter case, cost calculations in Nefolog are an example.

All previous artifacts are represented in the *Utility Analysis* tool as RESTful resources, enabling application and business architects to search, clone, and create reusable functions in its repository. So far, structural characteristics of functions in the *Utility Analysis* have been presented. The following section focuses on how functions can be defined, interpreted, and processed for calculating utility.

[22] JAX-RS: https://docs.oracle.com/javaee/6/tutorial/doc/giepu.html

[23] JAXB: http://www.oracle.com/technetwork/articles/javase/index-140168.html

[24] MySQL Relational Database System: https://www.mysql.com/

9.8.2 Processing of Functions

Two fundamental building blocks in the *Utility Analysis* tool are the *Utility Functions Constructor* and the *Utility Functions Management*, as these enable application and business architects to build custom utility functions by creating new or reusing existing functions in a repository of functions. Both components allow to represent functions as strings, which are subsequently parsed and processed. The remainder of this section focuses on introducing an underlying mechanism to define, interpret and process functions in the repository.

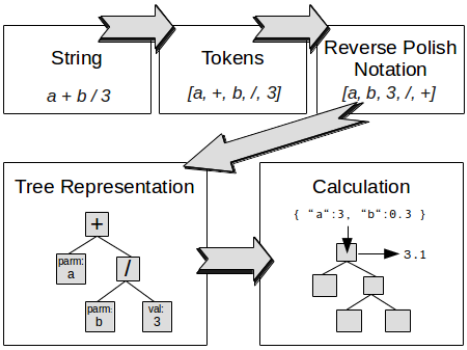


Figure 9.17: Function Calculation - Process [Fre16]

Figure 9.17 depicts a process for parsing and calculating functions in the *Utility Analysis* tool. In particular, functions are represented as strings, which (i) are parsed into string tokens, and then transformed using reverse polish notation, in order to (iii) determine a tree representation of such functions. Given certain parameters, this tree representation is then processed and its result is calculated. The remainder of this section details different steps for calculating functions, by means of introducing a syntax for defining functions, and the

underlying algorithms for parsing and creating its tree representation.

9.8.2.1 Syntax

SCARF-T allows the definition of function *Constants* and *Variables*. In particular, constants must be specified following a regular expression $[0-9]+[[.][0-9]+]?$. For instance, accepted values in functions are numbers. On the other hand, variables are also necessary to define functions, which can be defined according to a regular expression $[a-z,A-Z][a-z,A-T,0-9]*[[\_][a-z,A-Z]+]?$. In the case of indexing variables for defining collection of values, e.g. T_i, these are denoted as var_index, e.g. T_i. Constants may also be expressed as variables that stand for a fixed number, e.g. Euler's number e or π. In such cases, the *Utility Functions Constructor* detects and processes these special constants automatically.

Focusing on defining operations within functions, the *Utility Functions Constructor* parses two types of operators: *Basic* and *Boolean Operators*. In both cases, operations can be grouped using brackets, i.e. [], or parenthesis, i.e. (). Basic operators are supported for defining addition, substraction, multiplication, division, exponentiation, etc. For instance, function $E = m * c^2$ can be defined as $m * c\hat{}2$. Boolean operators are also supported in SCARF-T. For instance, $OR(a, b)$ takes the value of 1 if a and/or b are not zero, and zero in all other cases. Further boolean operators are $XOR(a, b)$, $AND(a, b)$, $<(a, b)$ (lower than), $>(a, b)$ (greater than), $EQU(a, b)$ (equal), $ESM(a, b)$ (lower or equal), $EBG(a, b)$ (greater or equal), $NOT(a)$. For example, a logical expression $\neg a \vee b$ is defined as $OR(NOT(a), b)$.

When specifying functions, the *Utility Functions Constructor* distinguishes between two types of functions: *Basic* and *Referred Functions*.

Basic functions examples are square, square root, root, if, maximum, minimum, sine, cosine, tangent, factorial, sum, product, and definite integral. For instance, $SUM_x(x + a_x)$ computes the following addition function $\sum_{x=x_{min}}^{x_{max}}(x + a_x)$. For referred functions, the *Utility Functions Constructor* expects a syntax $FCT(function\_alias, assignment)$. *Function_alias* is basically a name of the function that FCT is referring to. The *assignment* binds a variable or parameter to each parameter in a referred function. Parameter assignments are separated by "*$*" and ":". For instance $FCT(remote, a: 1.5\$b: 3)$ uses a *remote* function with variables assignments $a = 1.5$ and $b = 3$.

The following section introduces how the previously defined syntax is parsed and converted into a tree representation, which is subsequently processed for calculation purposes.

9.8.2.2 Parsing & Calculation

As previously discussed, application and business architects can define new and reuse existing functions in a *Utility Functions* repository. The *Utility Functions Constructor* and *Utility Functions Management* are Java-based components implementing function parsing and processing in the *Utility Analysis* tool of SCARF-T. Functions can be defined as strings following a syntax introduced in the previous section.

In a first step, string representations of functions are parsed. This parsing consists of detecting tokens from an input string, which can be of type *operand*, *operator*, *parenthesis*, or *comma*. Detected tokens are then processed using a lightly modified version of the *Dijkstra's Shunting-Yard Algorithm* in order to generate a Reverse Polish Notation (RPN) representation [UK12]. A complete description of the algorithm is provided in [Fre16]. In summary, this algorithm starts with a list of

input tokens, an empty stack for persisting operators, and an empty output (or result) list, and generates a RPN that can be used for function calculation.

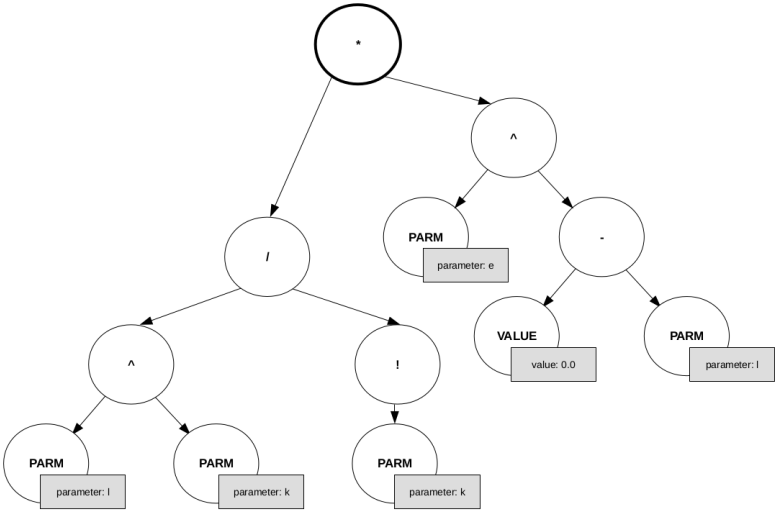


Figure 9.18: Calculation Tree - Example $\frac{l^k}{k!}e^{-l}$ [Fre16]

A *Utility Calculation* component in the *Utility Analysis* is responsible for processing the previously generated RPN representation. In particular, it generates a tree representation of the function, as depicted in Fig. 9.18, where root and parent nodes represent operators, and leaf nodes can be of three types: *Value*, *Parameter*, or *Function Reference*. Value nodes are associated with concrete values in a function, e.g. 0.0 (see Fig. 9.18). Parameter nodes are assigned with function parameters, e.g. k. Function reference nodes are assigned with parameters of referred functions. During calculation, the *Utility Calculation* component traverses the tree representation, substitutes parameters and

function references with input values, and calculates the function's result by analyzing operators in root and parent nodes.

9.8.3 Topology Repository & Modeler Integration

Previous sections focused on establishing technical foundations for defining, interpreting, and calculating functions provided by application and business architects when calculating utility. This section provides a technical overview and examples on how functions can be defined and used through the API of the *Utility Analysis* tool, and it introduces a graphical Web interface in the *Topology Modeling & Enrichment* tool for constructing functions and calculating utility for alternative ζ-topologies.

$$\sum_{x=0}^{n} \sum_{y=0}^{m} \frac{a_{x,y}}{b} \tag{9.1}$$

```
1 SUM_x(SUM_y(a_xy/b))
```

Listing 9.1: Syntax representation of function 9.1

Focusing on the *Utility Analysis* tool, previous sections introduced that its implemented as a Java Web application exposing a RESTful API to manage and calculate utility. Focusing on Function 9.1 as example, its representation using the previously introduced syntax is denoted in Listing 9.1. Considering this function as already existing in the *Utility Functions* repository, it can be used by executing a HTTP GET request with a path and query parameters defined in Listing 9.2. In such, specific values for parameters x, y, b, as well as all possible values for a_{xy}, are defined as query parameters in the HTTP request.

```
http://<url>/UtilityAnalysis/Function/myFct/calc
    ?
x=[0,2]&
y=[0,1]&
a_xy=[[-1,2.2],[1,2],[-1.6,4]]&
b=0.777
```

Listing 9.2: GET Request Path & Parameters for calculating function 9.1

When a utility function calculation is triggered, the *Utility Analysis* RESTful API responds with an XML message containing the result, the function's string representation, a RPN of the function, and the provided function parameters in the request (see Listing 9.3). This API, however, may not be in practice efficient for application and business architects for defining and using their functions during utility analysis of alternative ζ-topologies. Therefore, SCARF-T integrates its *Topology Modeling & Enrichment* tool with this API, and provides a graphical Web interface where utility functions can be managed and used in a simpler way, which is introduced in the remainder of this section.

```
<?xml version="1.0" encoding="UTF-8" standalone
    ="no"?>
<calculation>
  <result>8.49</result>
  <formula>SUM_x(SUM_y(a_xy/b))</formula>
  <toks>[SUM_x, (, SUM_y, (, a_xy, /, b, ), )]</
     toks>
  <rpn>[a_xy, b, /, SUM_y, SUM_x]</rpn>
  <parameters> { "a_xy":
     [[-1,2.2],[1,2],[-1.6,4]], "b": 0.777, "x":
      [0,2], "y": [0,1] } </parameters>
```

```
</calculation>
```

Listing 9.3: XML Representation: Function Calculation

SCARF-T integrates its *Topology Modeling & Enrichment* and *Utility Analysis* tools, in order to provide a graphical Web interface for defining and utilizing utility functions in a simple manner. As previously introduced, the *Topology Modeling & Enrichment* tool is based on OpenTOSCA Winery [KBBL13]. In particular, SCARF-T extends Winery's Topology Repository using JSP[25] tags and jQuery[26] developed in JavaScript[27]. Moreover, AJAX is used as the underlying framework for integrating with the *Utility Analysis* RESTful interface. Figure 9.19 depicts a graphical Web interface for defining reusable functions. In such, application and business architects can define custom functions, e.g. their application's revenue function and its corresponding parameters, as exemplified in Figure 9.19. This revenue function can then be used when calculating utility for alternative ζ-topologies. Functions are defined using the previously introduced syntax and its parameters are specified together with their data types, such as number or array of numbers.

Although SCARF-T supports defining custom utility functions for applications, it provides a default utility function defined in Chapter 8. In short, the default utility function is based on the expected profitability for an alternative ζ-topology, and depicted in Chapter 8 and Eq. 9.2. For such, SCARF-T provides its default *revenue*, *satisfaction*, and *cost* functions. These are specified in Listing 9.4 following the

[25] JavaServer Pages: http://www.oracle.com/technetwork/java/index-jsp-138231.html
[26] jQuery: https://jquery.com/
[27] JavaScript: https://www.javascript.com/

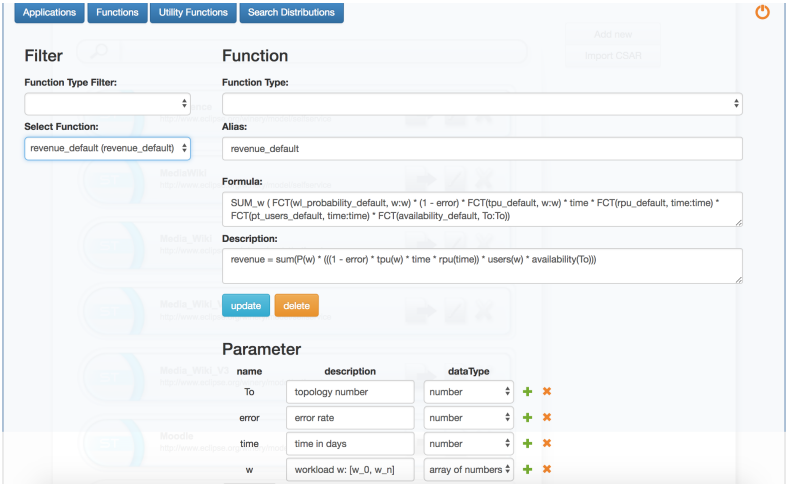


Figure 9.19: Topology Repository Tool - Utility Functions Graphical Interface

Utility Analysis tool syntax.

$$u(T^{\zeta},R,M,W,\Psi_m) = revenue(T^{\zeta},M,W,\Psi_m) \times sat(\Psi_m,W) - cost(T^{\zeta},R,M,\Psi_m) \quad (9.2)$$

```
SUM_w ( FCT(wl_probability_default, w:w) * (1 -
    error) * FCT(tpu_default, w:w) * time * FCT(
    rpu_default, time:time) * FCT(
    pt_users_default, time:time) * FCT(
    availability_default, To:To))*
2 user_gained / user_loss -
  FCT(nefolog_calculation,
```

```
configurationID:configurationID)
```

Listing 9.4: SCARF-T Default Utility Function - Syntax

Subsequently to defining custom functions for calculating utility in the *Topology Repository* of SCARF-T, application architects can make use of such functions after discovering alternative ζ-topologies for their applications. As previously introduced, application architects can define their application specific α-topologies in a *Topology Modeling & Enrichment* tool, and utilize its discovery functionality to view and select alternative ζ-topologies.

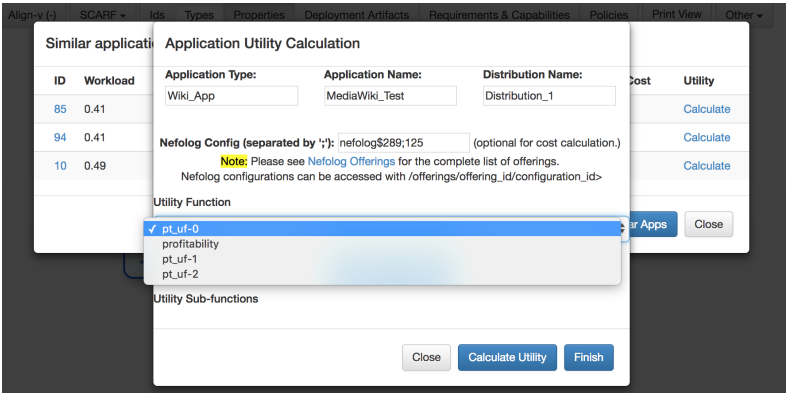


Figure 9.20: Topology Modeling & Enrichment Tool - Utility Calculation Interface

After alternative ζ-topologies are discovered, the *Topology Modeling & Enrichment* tool provides a graphical Web interface that integrates with the calculation support in the *Utility Analysis* tool, and allows utility calculation for each alternative ζ-topology. Figure 9.20 illustrates a starting point when calculating utility for an alternative ζ-topology. In such, application architects can review and define relevant cost

configurations in Nefolog (*Cost Knowledge and Calculation* tool), and can select their desired utility function. When a utility function is selected, the *Topology Modeling & Enrichment* tool interacts with the *Utility Analysis* tool and dynamically renders all necessary data for a selected utility function, such as its parameters.

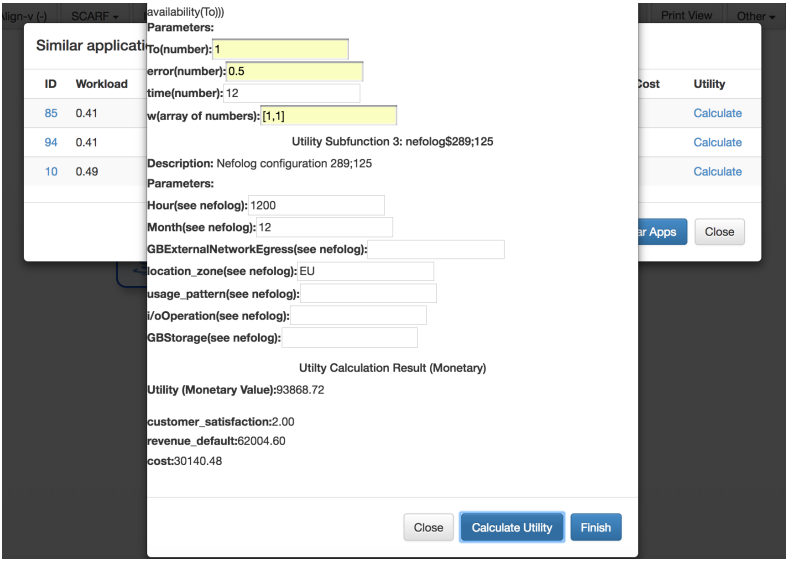


Figure 9.21: Topology Modeling & Enrichment Tool - Utility Calculation

Figure 9.21 depicts the step after selecting a desired utility function. In particular, the Web graphical interface expands for specifying parameters of a selected utility function. The visual interface is dynamically rendered depending on a selected function and based on interacting with the *Utility Analysis* tool for retrieving function characteristics.

So far, this section introduced the underlying technological support in SCARF-T for utility analysis and calculation. In such, application and business architects can specify custom and reusable utility functions, which can be leveraged for decision making support after discovering alternative ζ-topologies. The following section goes a step further in SCARF, and focuses on the technological support for selecting and refining alternative ζ-topologies.

9.9 Selection & Refinement

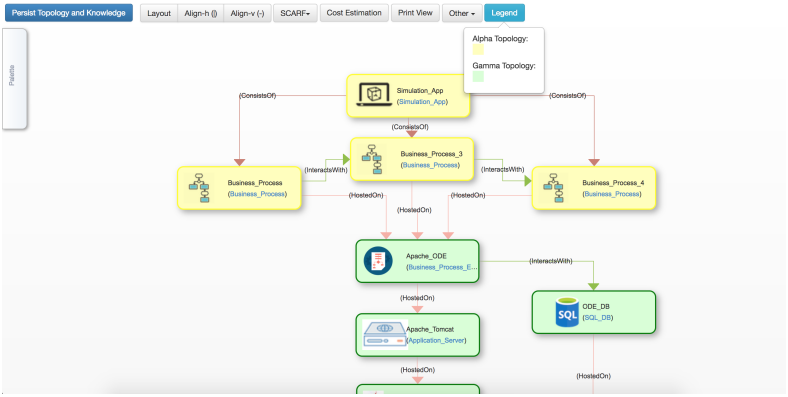


Figure 9.22: Topology Modeling & Enrichment Tool - Refinement of Alternative Topologies

The *Topology Modeling & Enrichment* tool in SCARF-T is extended in order to provide support for visualizing ζ-topologies. In this view, application architects can visually differentiate modeled α-topologies and discovered reusable γ-topology fragments. Figures 9.22 and 9.23 provide an example for a simulation application and a Wiki appli-

cation, respectively. In both examples, the modeling tool visually differentiates application specific and application reusable topologies, depicted as α- and γ-topologies, respectively. Application architects can refine a selected alternative ζ-topology, as the *Topology Modeling & Enrichment* tool is extended but maintaining original modeling capabilities of Winery [KBBL13], such as its modeling palette.

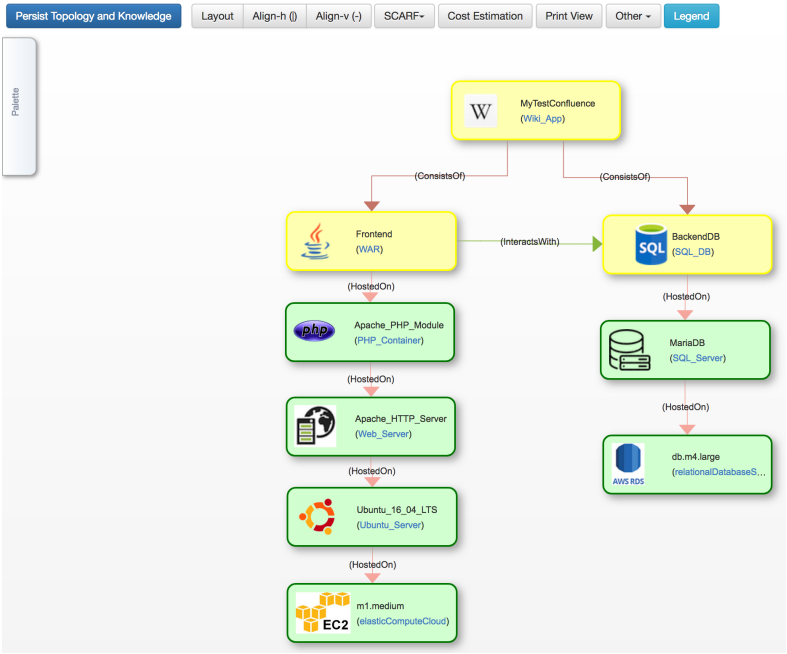


Figure 9.23: Topology Modeling & Enrichment Tool - Refinement of Alternative Topology

After refining a selected alternative ζ-topology, application architects can update the *Alternative Topologies* knowledge base with their selected ζ-topology. SCARF-T then associates a selected alternative

ζ-topology with an application's α-topology, and prepares *Workload* and *Performance* knowledge bases for aggregating performance and workload monitoring information once the application is provisioned and deployed.

9.10 Provisioning Engine

After application architects finalize decision making and selection phases in SCARM, an alternative ζ-topology is ready to be provisioned. Such ζ-topology is the input for the *Provisioning Engine* in SCARF-T, which is responsible for instantiating it. In other words, a provisioning engine orchestrates provisioning of underlying resources, such as cloud services, servers, etc., and finally deploys a CSAR. Essentially, it prepares an application's environment according to its ζ-topology.

As previously discussed, SCARF-T aims at maximizing reusability of existing tools and components. Since the implementation of SCARF-T is based on OpenTOSCA [KBBL13] and compliant with TOSCA [BBKL14a], it can leverage the OpenTOSCA's tool chain for provisioning and deploying cloud application topologies. In particular, the *Topology Modeling & Enrichment* tool supports exporting CSAR packages. CSAR are packages, which are portable across different TOSCA-compliant implementations [KBBL13]. These packages can then be imported into TOSCA-compliant provisioning engines, such as the *OpenTOSCA Runtime Container* [BBH+13].

Due to the implementation and research scope in this work, SCARF-T is delivered without a TOSCA-compliant runtime container, as these are typically vendor specific and packed as separate components. However, SCARF-T recommends to use the open source *OpenTOSCA*

Runtime Container[28].

9.11 Monitoring & Knowledge Retrieval

After provisioning alternative ζ-topologies, applications enter into their execution phase. In such, SCARF focuses on aggregating and processing application performance and workload monitoring data. For instance, CPU or memory utilization, as well as arrival rate of requests, might be metrics of interest for deciding future application distributions. In short, performance and workload data is necessary to feed and build both, *Workload* and *Performance Knowledge* bases.

SCARF-T integrates support for updating performance and workload information, by means of enhancing the *Topologies Knowledge & Discovery* tool in SCARF-T. In summary, it exposes a RESTful API developed using JAX-RS to update on regular time intervals performance and workload knowledge. Such knowledge can then be associated with provisioned ζ-topologies. Due to the scope of SCARF-T, a monitoring tool is not developed or provided in the scope of this work. However, all monitoring tools capable of consuming a RESTful API are in principle supported. For concrete examples of tools, a survey of cloud monitoring tools is provided in [ABDP13; FEH+14]. Moreover, cloud providers, such as AWS, provide services like CloudWatch[29] that can be easily integrated in order to propagate application monitoring data to external data sources, such as SCARF-T's *Topologies Knowledge & Discovery* RESTful API.

[28] OpenTOSCA: `http://www.iaas.uni-stuttgart.de/OpenTOSCA/`
[29] AWS CloudWatch: `https://aws.amazon.com/cloudwatch/`

9.12 Prototype Delivery

Previous sections focused on describing a prototypical implementation of SCARF-T. As previously introduced, SCARF-T's tool chain consists of multiple loosely coupled Web applications, each implementing a piece of functionality for SCARF. Due to the underlying complexity of SCARF-T in terms of set-up and usage, this section focuses on describing how SCARF-T is delivered in a portable and simplistic manner.

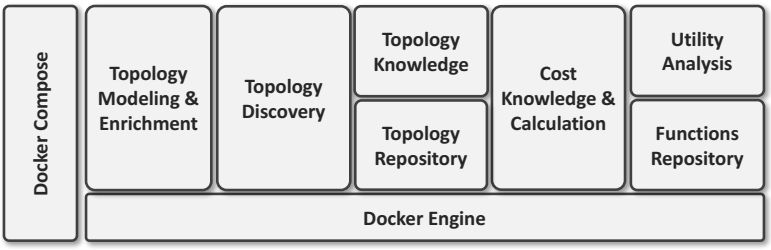


Figure 9.24: SCARF-T - Prototype Delivery

Figure 9.24 illustrates the environment setup for SCARF-T based on Docker[30] [Tur14]. In particular, SCARF-T is executed in a Docker environment, where all its tools are distributed using Docker containers. Tools for *Topology Modeling & Enrichment*, *Topology Discovery*, *Topology Knowledge*, *Cost Knowledge & Calculation*, and *Utility Analysis* are independent Docker images, each described as Docker files. In case of applications using databases, such as the *Topology Repository* or the *Functions Repository*, database servers are containerized in separate Docker images and provisioned as independent Docker containers. In order to automate the provisioning of all Docker containers, SCARF-T

[30]Docker: https://www.docker.com/

leverages Docker Compose[31]. A Docker compose file specifies containers and relationships among them, as well as networking configuration and links among containers. With this delivery approach, application and business architects can provision SCARF-T's tool set in a Docker environment with minimum configuration and set-up overhead.

9.13 Chapter Summary

SCARF was introduced in previous chapters as a generic framework assisting application and business architects to migrate their applications to the cloud. Such framework leverages application knowledge, similarity analysis, and utility in order to minimize the complexity of decision making when selecting and configuring cloud services. This chapter introduced SCARF-T, an implementation supporting SCARF. SCARF-T incorporates multiple tools implementing functionalities for modeling application topologies, persisting them in a reusable manner, and supporting all decision support tasks in SCARF.

SCARF-T is shipped with the following set of tools implemented as loosely coupled Web applications:

- A *Topology Modeling & Enrichment* tool for modeling and enriching application topologies.
- A *Topologies Knowledge & Discovery* tool for discovering alternative application distributions, each specified as a ζ-topology. This tool integrates a knowledge base capturing all possible distributions of applications, depicted as μ-topologies, which essentially comprise all α-topologies, and their compatible and reusable γ-topologies. μ-topologies are used as the basis for

[31] Docker Compose: https://docs.docker.com/compose/

discovering alternative ζ-topologies using similarity analysis as the basis.

- A *Cost Knowledge & Calculation* tool enhances SCARF-T with cost calculation capabilities for different cloud providers and services. This tool contains cost knowledge for a variety of cloud providers and services.
- A *Utility Analysis* tool implements utility support in SCARF-T. Application and business architects can define new or reuse existing functions to calculate utility of alternative ζ-topologies.
- A *Provisioning Engine* responsible for the instantiation of alternative ζ-topologies, i.e. provisioning of cloud resources, application servers, etc., as well as their corresponding configuration.
- *Monitoring & Analysis* and *Knowledge Filtering* tools for monitoring performance and workload metrics necessary to aggregate, update, and build application performance and workload knowledge.

This chapter provided insights into all aforementioned tools, by means of introducing key implementation details for new or extended tools, and summarizing adopted functionalities of existing tools. Finally, an overview on how SCARF-T is delivered is provided. In such, its tool set is shipped as containers using Docker, in order to ensure portability, and minimize set-up and configuration overhead.

Chapter 10
Evaluation

10.1 Chapter Introduction

SCARF was introduced in this work as a design decision support framework for assisting application and business architects to migrate their applications to the cloud. This chapter validates SCARF (introduced in Chapter 4) and its underlying technological support SCARF-T (introduced in Chapter 9), using the *MediaWiki* and *ALLOW Ensembles Trip Booking Ensemble* applications as case studies for cloud migration.

MediaWiki[1] is an open-source Web application used for supporting Wikipedia[2], part of the Wikimedia Foundation (WMF)[3]. The Trip Booking Ensemble application showcases the capabilities offered by the concept of Collaborative, Dynamic, and Complex (CDC) systems

[1] MediaWiki: https://www.mediawiki.org
[2] Wikipedia Project: https://www.wikipedia.org
[3] Wikimedia Foundation: https://wikimediafoundation.org

introduced in [ABG+13b; AGKW13; AGKW14] and developed in the scope of the FP7 EU project ALLOW Ensembles[4]. It essentially implements a *Pervasive Application* for Smart Cities, and is realized using SOA principles.

The remainder of this chapter evaluates different aspects of SCARF, with a strong focus on evaluating utility as the underlying decision support mechanism. These evaluations have been driven in research works [GAB+18] and [Moh16], respectively. For each application, this chapter first provides a technical overview specific to each application and experimental setup, i.e., its enriched α-topology. Subsequently, SCARM and SCARF-T are used to emulate the decision support process for migrating both applications to the cloud. Finally, evaluation findings are discussed for both applications, focusing on using different utility models for decision making support.

10.2 MediaWiki: A Wikipedia Case Study

10.2.1 Application Overview & Setup

MediaWiki is an open-source and highly scalable Wiki application used for Wikipedia, which enables users to create, edit, and read articles in a simplified and collaborative manner. Wikipedia uses MediaWiki in a highly distributed manner, as it distributes and replicates application and database nodes on-premise to serve Wikipedia content across the globe. The WMF identified in its Y2016 Annual Plan[5] a number of potential risks, among which are the following: (i) failure in tech-

[4] ALLOW Ensembles:
https://cordis.europa.eu/project/rcn/106345/factsheet/en

[5] WMF Y2016 Annual Plan Report:
https://de.wikipedia.org/wiki/Datei:WMF2015-16AnnualPlan.pdf

nology infrastructure cause a disruption of WMF operations, and (ii) efforts to build large scale, high performance features result in delays or failures. Migrating MediaWiki to the cloud opens a wide set of possibilities, as it can leverage from on-demand usage of resources, high availability, and a reduction of on-premise costs.

This evaluation uses a realistic Wikipedia workload as the basis. More specifically, it focuses exclusively on the English Wikipedia, since it represents ≈ 46% of all Wikimedia projects, and received the highest load ($\approx 7,869M$ views) in January 2016, showing a 9% increase w.r.t. the previous year, according to Wikistats[6].

Wikidumps[7] is used as the basis for generating a realistic workload and content representing the English Wikipedia. The generated workload is a scaled version of the original Wikidumps workload, and specifically focuses on the time period between Jan. 1 2016 and Jan. 31 2016, as Wikipedia organizes donation campaigns starting in December and ending in January. The generated workload aggregates a total of 79K users and 632K requests. Further information on the used Wikipedia content and generated workload is provided in [GAB+18].

10.2.2 α-Topology

Focusing on MediaWiki's architecture, Fig. 10.1 depicts the α-topology of MediaWiki modeled with SCARF-T's *Topology Modeling & Enrichment* tool introduced in Chapter 9. MediaWiki aggregates two tiers: a (i) front-end PHP application comprising business logic and caching functionalities, and a (ii) backend SQL database, where data and meta-data for articles, users, and content modifications are persisted.

[6] Wikistats: `https://stats.wikimedia.org/EN/TablesPageViewsMonthlyCombined.htm`

[7] Wikidumps: `https://dumps.wikimedia.org/`

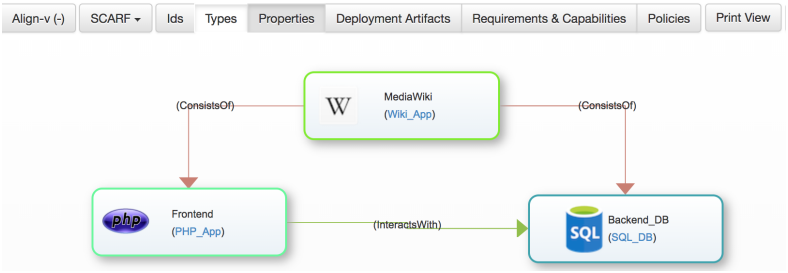


Figure 10.1: MediaWiki - α-Topology

Following SCARF's method, a subsequent step consists of enriching MediaWiki's α-topology (see Table 10.1), using Wikipedia's performance and workload knowledge for the month of January 2016 as the basis. For simplicity purposes, only minimum performance requirement values are specified for *Throughput*, *Network Utilization*, *Disk Utilization*, *VM Utilization*, and maximum performance requirements values for *Response Time* and *VM Startup Time*.

10.2.3 Construction of the μ-Topology

After enriching MediaWiki's α-topology, the *Discover Viable Topologies* step in SCARM is applied to retrieve alternative ζ-topologies. Subsequently, the construction of MediaWiki's μ-topology is driven manually, i.e., application architects manually select which alternative ζ-topology is aggregated into a final μ-topology. Table 10.2 lists the set of alternative ζ-topologies discovered by SCARM and aggregated into its μ-topology.

Alternative ζ-topologies in Table 10.2 rely on two cloud providers using VM- and container-based services. Prices are based on the on-

Table 10.1: MediaWiki α-Topology Enrichment - Sample Performance & Workload

	Category	Metric	Value
Performance	*Resource Capacity*	Response Time	30 s
		Throughput	5 req./s
	Resource Utilization	Network Utilization	70 %
		Disk Utilization	60 %
		VMs Utilization	70 %
	Elasticity	VM Startup Time	20 s
Workload	*Seasonal Pattern*	-	Periodic
	User Arrival Distribution	-	Poisson
	Average Number of Users	-	2,5 K
	Average Number of Transactions	-	20 K
	Time Interval	-	31 days

demand usage of cloud services for hosting MediaWiki's front- and back-end tiers. There are further costs, e.g., load balancer instances or network egress charges, which are not considered in the scope of this evaluation.

10.2.4 Construction of the Utility Function

Before calculating utility, application and business architects must create their application's revenue and satisfaction functions in the *Utility Analysis* tool in SCARF-T (see Sec. 9.8). The remainder of this

Table 10.2: Evaluation Setup - Alternative ζ-Topologies (T^{ζ}) of the MediaWiki Application. Summarized from [GAB+18].

T^{ζ}	Cloud Service	MediaWiki Front-end	MediaWiki Back-end	Total Price (USD/h)
T_1^{ζ}	EC2	m4.large	m4.large	0.234
T_2^{ζ}	EC2	m4.xlarge		0.264
T_3^{ζ}	EC2 + RDS	m4.large	db.m4.large	0.325
T_4^{ζ}	Beanstalk	(2x) t2.small	db.m4.large	(2x) 0.028 + 0.193
T_5^{ζ}	ECS	(2x) t2.small	db.m4.large	(2x) 0.028 + 0.193
T_6^{ζ}	VM	DS2	DS2	0.292
T_7^{ζ}	VM	DS3		0.292
T_8^{ζ}	Container	(2x) DS1	DS2	(2x) 0.073 + 0.146

Amazon Web Services Microsoft Azure

section summarizes [GAB+18] for defining revenue per user, user satisfaction, workload, and cloud provider's availability functions.

Workload Probability Function The workload probability function is based on Wikipedia's realistic workload and content data only for its donation campaign in January 2016. Therefore, it assumes a constant workload probability function $P(w) = 1$, as only the Wikipedia workload for January 2016 is considered. This function should be adapted in case of analyzing a bigger time interval for the evaluation, e.g. 6 months.

Average Transactions Per User Function In order to derive a function representing the average number of transactions per user, we analyzed and created a simplified version of the English Wikipedia's access traces and number of users provided by Wikidumps. This resulted in a workload comprising a total of 632K requests and 79K users. Considering a uniform distribution of requests among users, we derive for January 2016 the constant function $tpu(w) = 8$. If further months are considered, the function $tpu(w)$ would be represented as a step function.

Revenue per User The MediaWiki financial data from Wikipedia's 2015 Fundraising Data Report[8] was analyzed in [GAB+18], by means of processing the total daily income for December 2015, which is based on donations. The revenue for December 2015 follows an exponential function, as discovered in [GAB+18], and the average revenue per Wikipedia user is 0.000534 USD.

Availability Function The availability function is extracted from the SLAs provided by the cloud providers. More specifically, the availability function $av(T_i^{\zeta}, \Psi_m, V)$ depicted in Sec. 9.8 is defined for the scope of this evaluation as a step function, which returns the availability defined in the cloud provider's SLA for each T_i^{ζ}. In particular, it returns 0.9995 for AWS and 0.9995 for Microsoft Azure.

[8] WMF Fundraising 2015: https://meta.wikimedia.org/wiki/Fundraising#December_2015_Campaign_Launch_Update

User Satisfaction Function For building the satisfaction function, Wikipedia's Measuring User Search Satisfaction Schema 2.0.0[9] was investigated. In summary, the maximum waiting time for a user to retrieve a Wiki page before dropping out is 30s [GAB+18]. Moreover, user satisfaction is also impacted by unsuccessful requests, i.e., requests with an HTTP return code 40x or 50x. The satisfaction function is defined as:

$$sat(\Psi_m, W) = 1 - \frac{reqs_{>30s}(\Psi_m, W_1) + reqs_{error}(\Psi_m, W_1)}{reqs_{total}(\Psi_m, W_1)} \quad (10.1)$$

where W_1 is the realistic Wikipedia workload for January 2016, $reqs_{>30s} : \Psi_m \times W \rightarrow \mathbb{R}_{\geq 0}$ returns the total requests in the workload W with a latency higher than 30s, $reqs_{error} : \Psi_m \times W \rightarrow \mathbb{R}_{\geq 0}$ returns the total number of unsuccessful requests in the workload W, and $reqs_{total} : \Psi_m \times W \rightarrow \mathbb{R}_{\geq 0}$ returns the total number of requests in the workload W.

10.2.5 Utility-based Decision Making & Application Provisioning

Once custom application functions for MediaWiki are defined in the *Utility Functions* repository of the *Utility Analysis* tool (see Sec. 9.8), SCARM enters into the decision making and selection step. In such, utility analysis is leveraged to assist application and business architects

[9] Measuring User Search Satisfaction Schema 2.0.0: https://meta.wikimedia.org/wiki/Research:Measuring_User_Search_Satisfaction

to decide and select an alternative ζ-topology.

$$SCARF\ Utility : T_7^\zeta \succ T_2^\zeta \succ T_6^\zeta \succ T_1^\zeta \succ T_3^\zeta \succ T_8^\zeta \succ T_4^\zeta \succ T_5^\zeta \quad (10.2)$$

The preference order in 10.2 provides a ranked list in descendant order for MediaWiki's ζ-topologies. Results show that the optimal alternative topology is T_7^ζ with an utility of $\approx$ 480USD (US Dollars). On the other side, alternative topologies T_4^ζ and T_5^ζ offer the lowest utility, with values $\approx$ 60 USD and $\approx$ 1 USD, respectively. Although utility results seem to be low for the size of the English Wikipedia, it should be remarked that the generated workload is a scaled down version of the English Wikipedia workload for January 2016. If extrapolated to the real Wikipedia workload, the utility for T_7^ζ would be $\approx$ 480K USD, since this experiments used a scaled version of it.

During the selection step of SCARM, it can be safely assumed that application architects choose the alternative ζ-topology with highest utility, i.e. T_7^ζ. The selection step in SCARF-T represents updating the *Alternative Topologies* knowledge base and instantiating the selected ζ-topology. Although the provisioning and deployment step is not included in this evaluation, it can be performed using any TOSCA compliant *Provisioning Engine*, as discussed in Sec. 9.10. During runtime, application workload behavior and performance metrics are monitored and aggregated into the knowledge bases for workload and performance.

10.2.6 Findings & Limitations

The previous sections focused on evaluating SCARF and its underlying technological support SCARF-T using the MediaWiki application and

Wikipedia's realistic workload and data. More specifically, SCARM was applied to distribute MediaWiki to satisfy demands w.r.t. Wikipedia's donation campaign in January 2016. After discovering alternative ζ-topologies and constructing the application's μ-topology, utility was used as the underlying decision support mechanism for selecting an alternative ζ-topology to be instantiated.

As utility can be represented using different functions, the utility model presented in Sec. 9.8 is evaluated with other possible functions depicted in Eq. 10.3. The $u_{opex}(T_i^\zeta, ...)$ cost function originates in [AGLW14], and calculates utility based operational expenditures, while $u_{av}(T_i^\zeta, ...)$ focuses only on the availability of cloud services. Both are defined as follows:

$$\begin{aligned} u_{opex}(T_i^\zeta, R, V, W, \Psi_m) &= opex_{max} - opex(T_i^\zeta, R, V, \Psi_m) \\ u_{av}(T_i^\zeta, R, V, \Psi_m) &= av(T_i^\zeta, \Psi_m, V) \end{aligned} \tag{10.3}$$

where $opex_{max}$ represents the maximum operational cost, and $av(T_i^\zeta, \Psi_m, M)$ returns the availability of the cloud services in T_i^ζ. The ordered preference in 10.4 introduces the complete comparison elaborated in [GAB+18].

When using the function $u_{opex}(T_i^\zeta, ...)$ for calculating utility, T_7^ζ, T_6^ζ, and T_8^ζ offer the highest utility. However, alternative topologies T_2^ζ or T_4^ζ are not in high positions of the raking, which in practice offer a better performance and cost trade-off. Calculating utility using the function $u_{av}(T_i^\zeta, ...)$ is possible, but not practical. In particular, an ordered rank cannot be established, since both AWS and Microsoft

Azure guarantee an availability of 99.95% , according to their SLAs.

$$\begin{aligned} SCARF\ Utility &: T_7^\zeta \succ T_2^\zeta \succ T_6^\zeta \succ T_1^\zeta \succ T_3^\zeta \succ T_8^\zeta \succ T_4^\zeta \succ T_5^\zeta \\ Cost &: T_7^\zeta \succ T_6^\zeta \succ T_8^\zeta \succ T_2^\zeta \succ T_1^\zeta \succ T_4^\zeta \succ T_3^\zeta \succ T_5^\zeta \\ Availability &: T_1^\zeta \succeq T_2^\zeta \succeq T_3^\zeta \succeq T_4^\zeta \succeq T_5^\zeta \succeq T_6^\zeta \succeq T_7^\zeta \succeq T_8^\zeta \end{aligned} \tag{10.4}$$

SCARF's utility model analyses performance and costs deeper w.r.t. the other utility models, as it incorporates three main ingredients: (i) *revenue*, (ii) *user satisfaction*, and (iii) *operational costs*. However, this utility model introduces more complexity as the others. Further limitations of SCARF's utility model relate to supported cloud services cost models. In particular, only linear cost models are supported, such as hourly utilization or reserved instances. Spot instances, for example, are currently not supported. Moreover, this evaluation focused on single cloud distribution scenarios, i.e., hosting the complete application stack in one cloud provider. Using SCARF for multi-cloud scenarios could open a spectrum of possibilities for migrating specific workloads among multiple off-premise cloud providers.

10.3 Trip Booking Ensemble: A Smart City Application Case Study

10.3.1 Application Overview & Setup

The Trip Booking Ensemble application is a Smart City application realized for the EU ALLOW Ensembles[10] project. It introduces concepts of *entities, cells* and *ensembles* as the basis to realize Colective

[10] ALLOW Ensembles:
https://cordis.europa.eu/project/rcn/106345/factsheet/en

Adaptive Systems (CAS) [AGKW13; ABG+13b; ABG+14]. In short, *Entities* represent both software and human actors, which aggregate functionalities specified as *cells*. Entities can interact with each other, by mean of exposing their functionalities using cells (realized as service orchestrations), therefore resulting into *ensembles* (realized as service choreographies) [AGKW14].

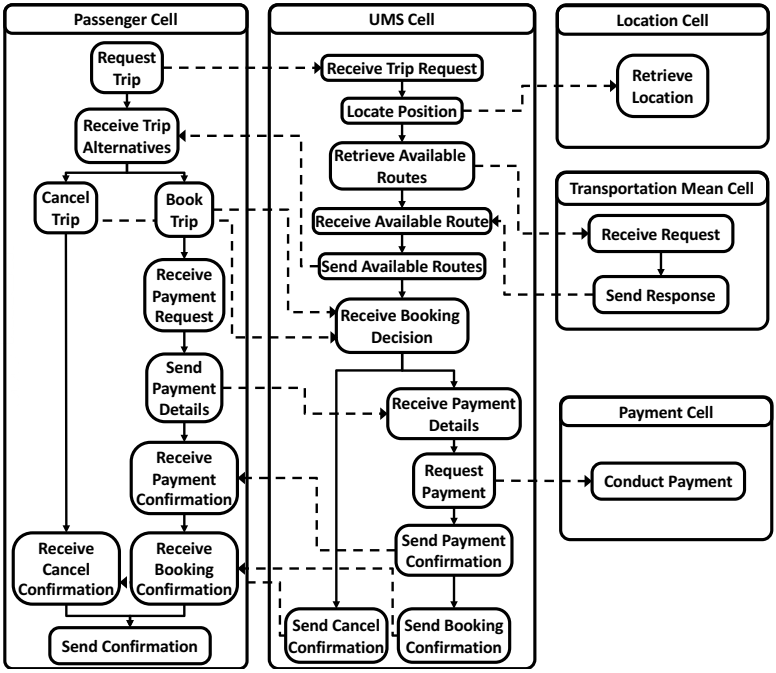


Figure 10.2: Trip Booking Ensemble Application

The Trip Booking Ensemble application implements a Smart City application for validating CDC systems using the city of Trento (Italy) as case study in [AGKW13; ABG+13b; ABG+14]. This ensemble

comprises multiple entities and cells, as depicted in Fig. 10.2. Passengers willing to commute between two points can interact with a so-called *Urban Mobility System (UMS)*, which acts as a city planner, as it discovers and adapts routes of different transportation means.

As illustrated in Fig. 10.2, entities collaborate with each other through their cells. Technically, ensembles are realized as BPEL4Chor choreographies [DKLW07], which are transformed into executable cells defined as WS-BPEL processes [OAS07]. BPEL4Chor choreographies are not executable, and must therefore be transformed before deployment into executable WS-BPEL processes [ABG+14].

10.3.2 α-Topology

Figure 10.3 depicts the application's α-topology modeled with SCARF-T's *Topology Modeling & Enrichment* tool introduced in Chapter 9. Essentially, the application-specific components of the Trip Booking Ensemble are a set of interconnected business processes, as depicted in Fig. 10.3. These are all specified using WS-BPEL standards and can be executed in any compliant WS-BPEL process orchestration engine.

Following SCARM, the α-topology is subsequently enriched with performance and workload requirements derived from [ABG+14] and depicted in Table 10.3. The application must handle at least 160 concurrent requests in time intervals of 15 min., and users should not experience a latency greater than 5 sec. This evaluation adopts the workload generated in [ABG+14], which emulates a regular start of a workday in the city of Trento (Italy). In particular, it contains a total of 10K users that arrive to the system following a Poisson distribution with $\lambda = 5$. Each user's trip booking consists of two requests: (i) requesting trip alternatives and (ii) paying for a transportation ticket

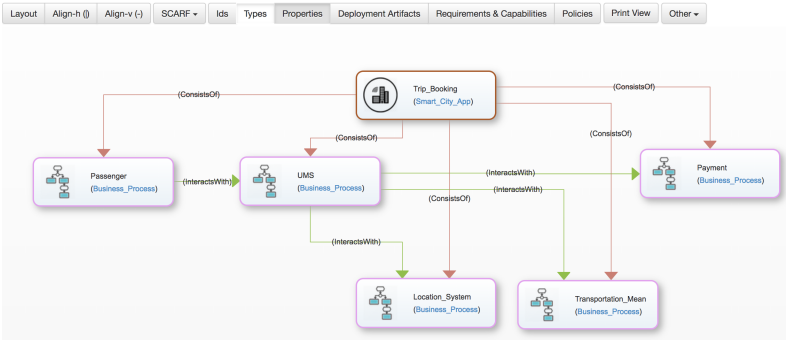


Figure 10.3: Trip Booking Application - α-Topology

(see Fig. 10.2). Each user interacts with the application twice per day in average, as we assume that these are commuting to and back from a point in the city, e.g., commuting to work in the morning and back home in the evening.

10.3.3 Construction of μ-Topology

The *Discovery Viable Topologies* step in SCARM is executed to retrieve alternative ζ-topologies. Table 10.4 depicts alternative ζ-topologies using the Apache ODE (Orchestration Director Engine)[11] as process orchestration engine. Apache ODE is developed as a Java Web Application, which supports the execution of WS-BPEL processes. Its architecture essentially comprises an Java-based *Orchestration Engine* that requires an Application Server and a backend *Auditing DB* in a MySQL database [Moh16].

For clarification purposes, the derived alternative ζ-topologies de-

[11] Apache ODE: `http://ode.apache.org/`

Table 10.3: Trip Booking Ensemble Application α-Topology Enrichment - Performance & Workload

	Category	Metric	Value
Performance	*Resource Capacity*	Response Time	5 s
		Throughput	0.2 req./s
	Resource Utilization	Network Utilization	40 %
		VMs Utilization	90 %
Workload	*Seasonal Pattern*	-	Periodic
	User Arrival Distribution	-	Poisson
	Average Number of Users	-	10K
	Average Number of Transactions	-	2
	Time Interval	-	4 hs.

picted in Table 10.4 are structured into two groups. The first group comprises *Single Instance Deployment* scenarios, which essentially deploy all application's WS-BPEL processes in a single instance of Apache ODE. The second group is highlighted in gray, and considers *Distributed Instances Deployment* scenarios, which distributes the application's WS-BPEL processes among multiple interconnected instances of Apache ODE.

When deploying all application's WS-BPEL processes in a single Apache ODE instance, SCARM derives alternative ζ-topologies that (i) host both orchestration engine and the auditing database on a single VM instance, e.g., T_1^{ζ},...,T_6^{ζ}, and that (ii) host the orchestration

Table 10.4: Evaluation Setup - Alternative ζ-Topologies (T^{ζ}) of the Trip Booking Ensemble Application. Summarized and adapted from [Moh16].

T^{ζ}	Cloud Service	Orchestration Engine	Auditing DB	Total Price (USD/h)
T_1^{ζ}	AWS EC2	m4.xlarge		0.264
T_2^{ζ}	AWS EC2	c4.xlarge		0.238
T_3^{ζ}	AWS EC2	r3.xlarge		0.371
T_4^{ζ}	Azure Compute	DS3 v2		0.277
T_5^{ζ}	Azure Compute	F4S		0.242
T_6^{ζ}	Azure Compute	DS12 v2		0.371
T_7^{ζ}	AWS EC2 & RDS	m4.large	db.t2.large	0.132 + 0.146
T_8^{ζ}	AWS EC2 & RDS	c3.large	db.t2.large	0.12 + 0.146
T_9^{ζ}	AWS EC2 & RDS	r3.large	db.t2.large	0.185 + 0.146
T_{10}^{ζ}	AWS EC2 & RDS	(2x)t2.small	(2x)db.t2.large	(2x) 0.028 + (2x) 0.146
T_{11}^{ζ}	AWS EC2 & RDS	(2x)c3.large	(2x)db.t2.large	(2x) 0.12 + (2x) 0.146
T_{12}^{ζ}	AWS EC2 & RDS	(2x)r3.large	(2x)db.t2.large	(2x) 0.185 + (2x) 0.146

Single Instance Deployment Distributed Instances Deployment

engine on a VM instance and the auditing DB in a separate DBaaS instance, e.g., T_7^{ζ},...,T_9^{ζ}. When distributing the application's WS-BPEL processes across multiple instances, SCARM derives alternative ζ-topologies using two VM instances to host the orchestration engine and two DBaaS instances to host the auditing DB [Moh16]. For

all cases, SCARM also discovers alternative ζ-topologies including different types of VM instances, such as *General Purpose m4.xlarge* and *t2.small*, *Compute Optimized c3.large*, and *Memory Optimized r3.large*.

The construction of the application μ-topology is manually driven by selecting alternative ζ-topologies derived in SCARF-T. Essentially, application architects can manually select among the alternative ζ-topologies, i.e., T_1^{ζ},...,T_{12}^{ζ}, discovered by SCARM.

10.3.4 Construction of the Utility Function

Utility functions must be first specified in order to compute the utility for each alternative ζ-topology. In such tasks, application and business architects specify the intrinsic functions of the application's revenue and satisfaction functions of the utility model in the *Utility Analysis* tool in SCARF-T (see Sec. 9.8).

Workload Probability Function The application workload generated in [ABG+14] emulates the start of a workday in the city of Trento (Italy). Therefore, its probability function represents the probability of having one workday within a month, assuming no national or local holidays. Considering a total of 30 natural days and 22 workdays, the probability of observing a workday workload is $P(w) = 0.73$, while for a non-workday is $P(w) = 0.27$.

Average Transactions Per User Function As in Sec. 10.3.2, each commuter in Trento (Italy) executes in average 2 transactions per day, as he commutes in the morning to its destination point, e.g., to work, and back home in the evening.

Revenue per User The bus system in Trento charges 1.20 € for a trip of 70 minutes[12]. Assuming that passengers commute in average 10 km and considering the analysis in [ABG+14], passengers need at least 3,5 min/km in average, therefore requiring 35 min in average to travel 10 km. When using a taxi, costs ramp up to 15,1 € for such a distance[13]. The average revenue per user is essentially a step function that depends on the used transportation mean: (i) 1.20 € for a bus and (ii) 15,1 € for a taxi trip.

Availability Function As for the MediaWiki evaluation in Sec. 10.2, the availability function is extracted from each SLA provided by the cloud providers. In particular, the availability function $av(T_i^{\zeta}, \Psi_m, V)$ returns 0.9995 for AWS and 0.9995 for Microsoft Azure.

User Satisfaction Function The user satisfaction function used for the Trip Booking Ensemble Application is defined in Eq. 10.5. We assume that users' satisfaction in both workloads W_1 and W_2 is impacted when the application's response time is greater than 20 s and by requests that derive failed responses.

$$sat(\Psi_m, W) = \prod_{k=1}^{2} 1 - \frac{reqs_{>20s}(\Psi_m, W_k) + reqs_{error}(\Psi_m, W_k)}{reqs_{total}(\Psi_m, W_k)} \quad (10.5)$$

[12] Trento Ticket Price: https://international.unitn.it/incoming/bus-and-train-travel-card

[13] Taxi Trento: https://www.taxitrento.it/en/fares/

10.3.5 Utility-based Decision Making & Application Provisioning

After defining the previous functions, SCARM enters into the decision making and selection step for the Trip Booking Ensemble Application.

$$SCARF\ Utility : T^{\zeta}_{12} \succ T^{\zeta}_{2} \succ T^{\zeta}_{9} \succ T^{\zeta}_{8} \succ T^{\zeta}_{7} \succ T^{\zeta}_{6} \succ T^{\zeta}_{5} \succ T^{\zeta}_{4} \succ T^{\zeta}_{1} \succ T^{\zeta}_{3} \succ T^{\zeta}_{10} \succ T^{\zeta}_{11} \tag{10.6}$$

Equation 10.6 provides a ranked list in descendant order for the discovered ζ-topologies. Results show that the optimal alternative topology is T^{ζ}_{12}, which distributes the application processes among AWS EC2 memory optimized and AWS RDS instances. T^{ζ}_{12} offers a utility of $\approx\ 20,6K\ USD$ (US Dollars) for the evaluated period. T^{ζ}_{2} and T^{ζ}_{9} are next in the ranked list, offering a utility of $\approx\ 13K\ USD$ and $12K\ USD$. Such a monetary difference is mainly due to latency and failed requests observed in the application, which has a negative impact on the user's satisfaction. On the other hand, T^{ζ}_{10} and T^{ζ}_{11} offer the smallest utility: $\approx\ 8K\ USD$ and $6K\ USD$.

Based on the previous utility results, application architects choose during the selection phase of SCARM the alternative ζ-topology with highest utility, i.e. T^{ζ}_{12}. When selected, SCARF-T updates the *Alternative Topologies* knowledge base and instantiates the selected ζ-topology. The provisioning step in SCARF-T can be driven using any TOSCA compliant *Provisioning Engine*, as discussed in Sec. 9.10.

10.3.6 Findings & Limitations

In the previous sections, SCARM and SCARF-T were leveraged to distribute the ALLOW Ensembles Trip Booking application. This evaluation adopted a workload defined in [ABG+14], which emulates the start of a working day in the city of Trento (Italy). SCARM was applied to build the application's μ-topology and to evaluate using utility its alternative ζ-topologies. This section analyzes findings and limitations w.r.t. the evaluation, with a strong focus on comparing SCARF's utility function with other possible functions.

$$\begin{aligned} u_{opex}(T_i^{\zeta}, R, V, W, \Psi_m) &= opex_{max} - opex(T_i^{\zeta}, R, V, \Psi_m) \\ u_{av}(T_i^{\zeta}, R, V, W, \Psi_m) &= av(T_i^{\zeta}, \Psi_m, V) \\ u_{sat}(\Psi_m, W) &= 1 - \frac{reqs_{>20s}(\Psi_m, W) + reqs_{error}(\Psi_m, W)}{reqs_{total}(\Psi_m, W)} \end{aligned} \tag{10.7}$$

As previously introduced in Sec. 9.8, SCARF's utility function relies on three pillars: (i) application revenue, (ii) resources costs and availability to host the application, and (iii) user satisfaction. We evaluate the default function with other possible utility functions, such as a cost function $u_{opex}(T_i^{\zeta}, ...)$, an availability function $u_{av}(T_i^{\zeta}, ...)$, and a user satisfaction function $u_{sat}(\Psi_m, W)$, all depicted in Eq. 10.7. In particuar, $u_{opex}(T_i^{\zeta}, ...)$ focuses exclusively on operational costs, $u_{av}(T_i^{\zeta}, ...)$ focuses on infrastructure availability, and $u_{sat}(\Psi_m, W)$ is based only on the user satisfaction.

Equation 10.8 depicts a ranked list in descendant order of alternative ζ-topologies for all utility functions defined in Eq. 10.7. When considering only operational costs when calculating utility, T_2^{ζ}, T_5^{ζ}, and T_1^{ζ} offer the best utility while T_6^{ζ}, T_{11}^{ζ}, and T_{12}^{ζ} offer the least utility.

If compared to SCARF's default utility function, the major observed difference is on T_{12}^{ζ}, as T_{12}^{ζ} is the most expensive ζ-topology. However, due to a better performance, it provides an improved end-user satisfaction and capacity for processing more trip requests.

$$
\begin{aligned}
\mathit{SCARF\ Utility} &: T_{12}^{\zeta} \succ T_{2}^{\zeta} \succ T_{9}^{\zeta} \succ T_{8}^{\zeta} \succ T_{7}^{\zeta} \succ T_{6}^{\zeta} \succ T_{5}^{\zeta} \succ T_{4}^{\zeta} \succ \\
&\quad T_{1}^{\zeta} \succ T_{3}^{\zeta} \succ T_{10}^{\zeta} \succ T_{11}^{\zeta} \\
\mathit{Cost} &: T_{2}^{\mu} \succ T_{5}^{\zeta} \succ T_{1}^{\zeta} \succ T_{8}^{\zeta} \succ T_{4}^{\zeta} \succ T_{7}^{\zeta} \succ T_{9}^{\zeta} \succ T_{10}^{\zeta} \succ \\
&\quad T_{3}^{\zeta} \succ T_{6}^{\zeta} \succ T_{11}^{\zeta} \succ T_{12}^{\zeta} \\
\mathit{User\ Satisfaction} &: T_{12}^{\zeta} \succ T_{6}^{\zeta} \succ T_{4}^{\zeta} \succ T_{5}^{\zeta} \succ T_{8}^{\zeta} \succ T_{2}^{\zeta} \succ T_{9}^{\zeta} \succ T_{10}^{\zeta} \succ \\
&\quad T_{7}^{\zeta} \succ T_{1}^{\zeta} \succ T_{11}^{\zeta} \succ T_{3}^{\zeta} \\
\mathit{Availability} &: T_{1}^{\zeta} \succeq T_{2}^{\zeta} \succeq T_{3}^{\zeta} \succeq T_{4}^{\zeta} \succeq T_{5}^{\zeta} \succeq T_{6}^{\zeta} \succeq T_{7}^{\zeta} \succeq T_{8}^{\zeta} \succeq \\
&\quad T_{9}^{\zeta} \succeq T_{10}^{\zeta} \succeq T_{11}^{\zeta} \succeq T_{12}^{\zeta}
\end{aligned}
\tag{10.8}
$$

When considering only user satisfaction for utility, T_{12}^{ζ}, T_{6}^{ζ}, and T_{4}^{ζ} are situated among the three optimal alternative ζ-topologies, while T_{1}^{ζ}, T_{11}^{ζ}, and T_{3}^{ζ} are the least optimal. In this case, utility results are more similar to the ones of SCARF utility, as the user satisfaction is implicitly impacted by a low application performance. Finally, when availability is the only basis for calculating utility, a strict ranking cannot be defined, as both cloud providers AWS and Microsoft Azure offer the same availability in their SLAs.

The previous utility functions comparison showed that SCARF's utility model is more accurate, as its utility function combines three essential aspects: (i) *application revenue* and *availability*, (ii) *user satisfaction*, and (iii) *operational costs*. However, encountered limi-

tations in this evaluation are similar to the ones in the MediaWiki application. Defining all functions for SCARF's utility model may be complex, as it requires business architects to have previous data, experience, and knowledge to define revenue and satisfaction functions. One possible solution observed in this evaluation is to calculate utility only based on end-user satisfaction as a starting point, at its results approximated the ones of SCARF's utility model. Moreover, reserved or spot instances are not supported in SCARF, and could therefore not be leveraged to reduce operational costs. Focusing on the evaluation, the distribution of the Trip Booking Ensemble application is limited to a cluster of two machines. Distributing its business processes, e.g., among a bigger cluster of smaller machines may positively impact the application's utility.

10.4 Chapter Summary

This chapter evaluated SCARF and its technological realization SCARF-T, with a strong focus on analyzing the impact of using utility for decision making. Two applications were used as case study: (i) the MediaWiki application, used for Wikipedia, and (ii) the Trip Booking Ensemble Application, developed in the ALLOW Ensembles EU Project. For the former, realistic Wikipedia workload for January 2016 was used as the basis. For the latter application, we generated a workload that emulates transportation demands in the city of Trento (Italy).

For both case studies, the evaluation followed the same methodology. After defining the application-specific α-topologies, these were enriched with performance and workload information. SCARM was then applied to derive alternative ζ-topologies, which allowed to manually build both applications' μ-topologies. In summary, μ-topologies

for the MediaWiki and the Trip Booking Ensemble applications consisted of 8 and 12 alternative ζ-topologies, respectively. A subsequent step consisted of defining the application's *revenue*, *user satisfaction*, and *availability functions*, used as the basis for SCARF's utility calculation. Then, utility was calculated for all alternative ζ-topologies, and compared with other possible utility functions.

Evaluation results essentially showed an improved accuracy when using SCARF's utility model, in comparison to other possible utility functions, e.g., only considering operational costs or user satisfaction. However, one major encountered limitation relates to the complexity for defining revenue and user satisfaction functions, especially when no previous business data exists. SCARF and SCARF-T are able to some extent to mitigate this problem, as SCARF-T is not coupled to a specific utility model, but provides the flexibility to define custom utility functions for calculating the utility of alternative ζ-topologies in μ-topologies.

Chapter 11

Conclusions & Future Work

11.1 Summary

In the last years, the cloud computing landscape has enabled the Everything-as-a-Service (*aaS) delivery model, therefore building a broad spectrum of cloud providers and services. Nowadays, applications can be migrated to the cloud by means of spanning their components among different cloud providers and services. This had a positive impact on IT organizations, as the possibilities to shift their capital into operational expenditures considerably increased [Var10]. However, such a wide set of possibilities also increased the complexity of migrating applications to the cloud, especially when deciding which cloud services are the most suitable to host or replace each application

component.

Cloud services are intrinsically heterogeneous, as these typically vary in terms of security, cost, performance, and availability, among other aspects. This work focuses on cost and performance, and targets the generic migration decision problem defined in Chapter 1 as: *how to migrate applications to the cloud – by means of distributing the application components among multiple cloud services – while optimizing for cost and performance?* For this purpose, we introduced the Systematic Cloud Application (Re-)Distribution Framework (SCARF), a design decision support framework that assists application and business architects to efficiently migrate their applications to the cloud.

SCARF was formally introduced in Chapter 4 and comprises a lifecycle and a method named SCARM. SCARM aims at assisting the migration and re-engineering tasks of traditional applications, by means of providing the necessary artifacts and steps for design, decision making, provisioning, and management of distributed cloud applications. Although there are enough works targeting decision support for migration and engineering of applications in the cloud, these generally lack of support for analyzing simultaneously multiple non-functional aspects, e.g., cost and performance (see Chapter 2). In summary, these tackle either portability of cloud applications among cloud providers, or focus on supporting non-functional aspects independently. SCARF goes a step further in this domain, as it establishes foundations for migrating and distributing applications in the cloud focusing on optimizing the trade-off between cost and performance.

The remainder of this chapter summarizes research results of this work, by means of answering the research challenges identified in Chapter 1, and mapping them to specific solutions introduced in the chapters throughout this work. Subsequently, this chapter closes with

a summary of encountered limitations and research opportunities that can be incorporated in future investigations in the domain of migrating and engineering cloud applications.

11.2 Research Results

This section wraps up the research results of this work. In particular, it summarizes concepts and tools developed to address the research challenges presented in Sec. 1.3, and maps them to the chapters describing their realization.

> **RCh.1** How to identify the requirements for migrating applications to the cloud?

When migrating applications to the cloud, IT organizations must first identify functional and non-functional application characteristics, which are typically aligned with business objectives. Cloud services are built atop of shared infrastructure resources, therefore impacting the business and operational performance experienced by cloud consumers. Therefore, Chapter 3 empirically identified the necessity for capturing and analyzing performance and workload knowledge during design, and provisioning and execution phases of cloud applications, by means of incorporating such knowledge into cloud applications' architectures.

Focusing on non-functional application aspects, Chapter 5 leveraged experimental results in Chapter 3 and identified necessary performance and workload metrics and attributes. These were subsequently characterized in order to build a meaningful application workload and performance knowledge, which can be used to analyze performance

and workload variation over time.

Focusing on application architectural aspects, Chapter 6 formally defined a cloud application topology in SCARF, which is implicitly used by existing approaches, e.g., TOSCA. Essentially, cloud application topologies describe the architecture of cloud applications, by means of defining application components and relationships among them in a directed acyclic labeled graph.

RCh.2 How to model cloud applications considering cost and performance characteristics?

Chapter 2 summarized related works in both industry and research domains related to defining cloud application architectures and evaluating performance of cloud services. However, there exists a gap (i) for capturing and analyzing performance variation of cloud services over time, (ii) for incorporating such knowledge in cloud application architectures, and (iii) for analyzing application performance and cost knowledge in combination during decision making.

Chapter 3 introduced a meta-model for enriching cloud application topologies with cost and performance characteristics in a fine- or coarse-grained manner. This model is aimed at (i) defining application performance requirements and workload profiles during design phase, and (ii) capturing application performance and workload knowledge during execution phase. Performance and workload knowledge can be leveraged in SCARF to estimate the monetary profitability of a cloud application topology. Although the enrichment model focuses on performance and workload requirements and knowledge, it can be extended for other type of application requirements, e.g., security.

RCh.3 How to facilitate cloud offering decision making, considering the application's evolving workload and cloud service's variable performance?

Migrating applications to the cloud is a complex task, specially when analyzing and deciding among a wide spectrum of available cloud services. As previously discussed, application and business architects usually lack sufficient knowledge of existing cloud providers and offerings, particularly w.r.t. their costs and performance. SCARF bridges this gap in Chapter 4, by proposing SCARM, a method that defines a set of tasks to guide the decision making process for (i) designing, (ii) discovering, (iii) selecting, (vi) evaluating, (v) running, and (vi) evolving cloud application distributions, focusing on optimizing their cost and performance.

Chapter 6 introduced a formal model for viable and reusable cloud application topologies, which defines α-, γ-, ζ-, and μ-topologies. In summary, application architects can first model and enrich α-topologies, which define only the application-specific architecture. γ-topologies represent non-application specific components, which can be reused among multiple applications. These are used in SCARM as the basis to discover and build viable ζ-topologies, each representing a possible cloud application distribution. During selection, ζ-topologies are chosen and aggregated, therefore building the application's μ-topology.

The automation of discovery and selection of viable application topologies is developed in Chapter 7, and builds atop of the enrichment model defined in Chapter 5. CBR and similarity analysis are proposed as one possible mechanism to automatically discover and build

viable ζ-topologies, based on performance and workload knowledge of similar cloud applications. ζ-topologies can then be dynamically or manually selected to construct an application's μ-topology.

> **RCh.4** How to identify the optimal performance and cost trade-off among cloud offerings for a specific application as aspect of decision making?

The discovery and construction of viable application topologies reduces significantly the decision making complexity, considering that the set of application distribution possibilities may still be broad. In addition to that, however, it is necessary to identify the optimal trade-off between the cost and performance of each viable cloud application topology. Towards this challenge, Chapter 8 develops a utility model built upon a set of (utility) functions calculating how profitable a viable ζ-topology is. In particular, the functions presented in this work focus on application and business performance, end-user satisfaction, and resources costs.

This utility model is extensively evaluated in Chapter 10. The evaluation shows an improved accuracy when compared to other possible utility models. However, defining these functions requires application and business architects to have previous application and business knowledge.

> **RCh.5** What tooling support is required for the cloud migration decision making tasks?

Chapter 9 presented SCARF-T, the technological realization of SCARF. SCARF-T is essentially a tool chain that leverages existing standards, reuses and extends tools and technologies, and develops new

ones when necessary. Tools in SCARF-T are built as loosely-coupled and independent services that build three main architectural blocks for supporting the migration of applications to the cloud: *Modeling*, *Decision Support*, and *Provisioning & Execution*.

SCARF-T adopts existing technologies and standards discussed in Chapter 2 like TOSCA, the OpenTOSCA environment, the Nefolog cost knowledge and calculation framework, and builds a set of RESTful applications that are fully integrated with and accessible from the topology modeling and enrichment tools in SCARF-T.

11.3 Limitations & Research Opportunities

As discussed throughout this work, SCARF aims at assisting IT organizations to migrate and distribute their applications among cloud services. However, SCARF and its technological realization SCARF-T encounter several limitations, therefore opening a space for further research opportunities. These are highlighted in the remainder of this section.

One major assumption denoted in Chapter 1 relates to organizations having already decided to migrate their applications to the cloud. There exist frameworks, such as [ADKL14] and [SAB+13], that could be integrated into SCARF to provide an end-to-end automated decision support migration process, which could incorporate evaluating whether the migration should take place or not. SCARF also assumes that (i) application and business architects have full knowledge of their application architecture and business model and requirements, and (ii) enforces the development of cloud application topologies, i.e., α- or ζ-topologies, from the very beginning. Transforming existing application architectures, e.g., defined as UML component or deploy-

ment diagrams, into SCARF's TOSCA specification, would leverage the reusability of traditional application architectural specifications.

This work also established the boundaries w.r.t. considered application non-functional aspects. In particular, SCARF limits its decision support to cost and performance. Expanding support to analyze and evaluate the trade-off among further non-functional aspects, such as adaptation efforts, compliance, security, etc., would positively impact the adoption of SCARF, specially in regulated industry domains, such as financial institutions.

Focusing on cloud application costs, SCARF and SCARF-T are limited to the traditional pay-per-use cloud pricing model. However, in the last years cloud providers have developed and adopted more complex pricing models, specially for reserved and spot instances of resources. This would require adapting the utility model defined in Chapter 8. In line with cloud application costs, SCARF's application knowledge is essentially captured during execution. This has a negative cost impact at early times of applications, in particular if a "green field" approach is implemented where past data is not available or obsolete. In order to minimize costs during decision making, a simulation phase could be incorporated in the framework, supporting the simulation of ζ-topologies before their execution. Some examples tools are [ADM+12], [MGAD13], and [CRB+11], which provide a simulation environment for cloud services.

SCARF's utility model focuses on evaluating the profitability of cloud applications, by incorporating revenue, end-user satisfaction, and cost functions into it. As observed in the evaluation in Chapter 10, defining all functions can be a complex task for business architects, specially if no previous business and application knowledge exists. A possible way to mitigate this could be to investigate and develop a

mechanism to emerge and evolve utility functions over time, based on application and business needs and knowledge. Moreover, SCARF-T lacks at the moment technical support for comparing the expected utility during design time and the observed utility during runtime.

Focusing on further improvements in SCARF and SCARF-T, the following are recommended. Firstly, evaluating SCARF by means of driving a field study in practice would help to increase its adoption, specially in the industry domain. Focusing on SCARF-T, an enhancement of the topology modeling and enrichment tool to capture and graphically represent the evolution of utility over time would provide an efficient monitoring of the state of the application w.r.t. the performance and cost trade-off. Secondly, the enrichment of cloud application topologies with performance and workload aspects is technically possible at a coarse-grained level. This means that requirements are specified for the application as a whole. Enriching different topology constructs, such as components and relationships, would provide further improvements to the topology enrichment model of SCARF. Thirdly, Chapter 9 highlighted that integrating monitoring and knowledge capturing tools is out of scope for implementation. Future works in this direction consist of analyzing monitoring frameworks for cloud environments and integrating them into SCARF-T.

Appendix A

List of Acronyms

ACID Atomicity, Consistency, Isolation, and Durability

AHP Analytic Hierarchy Process

AI Artificial Intelligence

AJAX Asynchronous JavaScript and XML

ALM Application Lifecycle Management

AMI Amazon Machine Image

API Application Programming Interface

AWS Amazon Web Services

BN Bayesian Network

BASE Basically Available, Soft State, and Eventually Consistent

CAFE Composite Application Framework

CAMP Cloud Application Management for Platforms

CAS Colective Adaptive Systems

CBR Case-based Reasoning

CDO Cloud Deployment Option

CSAR Cloud Service Archive

DBaaS Database-as-a-Service

DaaS Desktop-as-a-Service

DSL Domain Specific Language

DSS Decision Support System

***aaS** Everything-as-a-Service

EF Experience Factory

EC2 Elastic Compute Cloud

ETG Enterprise Topology Graph

FCO Flexiant Cloud Orchestrator

GCP Google Compute Platform

GENTL Generalized Topology Language

HP Hewlett Packard

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IaaS Infrastructure-as-a-Service

IDaaS Identity-as-a-Service

JSON JavaScript Object Notation

JSP JavaServer Pages

JAXB Java Architecture for XML Binding

JAX-RS Java API for RESTful Web Services

KMC Kinetic Monte Carlo

KPIs Key Performance Indicators

LL Leasons Learned

MDE Model-Driven Engineering

NIST National Institute of Standards and Technology

OS Operating System

PaaS Platform-as-a-Service

PCM Palladio Component Model

PDP Platform Deployment Package

QoS Quality of Service

RDS Relational Database Service

REST Representational State Transfer

RPN Reverse Polish Notation

SCARF Systematic Cloud Application (Re)Distribution Framework

SCARM Systematic Cloud Application (Re)Distribution Method

SWfMS Scientific Workflow Management System

SLA Service Level Agreement

SLO Service Level Objective

SMEs Small-to-Medium Enterprises

SOA Service Oriented Architecture

SaaS Software-as-a-Service

SECaaS Security-as-a-Service

TIO Tivoli Intelligent Orchestrator

TOSCA Topology and Orchestration Specification for Cloud Applications

VM Virtual Machine

VMI Virtual Machine Image

XML Extensible Markup Language

WMF Wikimedia Foundation

List of Figures

List of Tables

Bibliography

[Aba12] D. J. Abadi. “Consistency Tradeoffs in Modern Distributed Database System Design.” In: *Computer-IEEE Computer Magazine* 45.2 (2012), p. 37 (cit. on p. 48).

[ABDP13] G. Aceto, A. Botta, W. De Donato, and A. Pescapè. “Cloud Monitoring: A Survey.” In: *Computer Networks* 57.9 (2013), pp. 2093–2115 (cit. on p. 249).

[ABG+13a] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. “Software Architecture Optimization Methods: A Systematic Literature Review.” In: *IEEE Transactions on Software Engineering* 39.5 (2013), pp. 658–683 (cit. on pp. 59, 120).

[ABG+13b] V. Andrikopoulos, A. Bucchiarone, S. Gómez Sáez, D. Karastoyanova, and C. A. Mezzina. “Towards Modeling and Execution of Collective Adaptive Systems.” In: *Proceedings of the Eighth International Workshop on Engineering Service-Oriented Applications (WESOA’13)*. Springer, Dec. 2013, pp. 1–12 (cit. on pp. 254, 264).

[ABG+14] V. Andrikopoulos, M. Bitsaki, S. Gómez Sáez, D. Karastoyanova, C. Nikolaou, and A. Psycharaki. “Utility-based Decision Making in Collective Adaptive Systems.” In: *Proceedings of the Fourth International Conference on Cloud Computing and Ser-*

vice Science (CLOSER'14). SciTePress, Apr. 2014, pp. 308–314 (cit. on pp. 264, 265, 269, 270, 272).

[ABG+16] V. Andrikopoulos, M. Bitsaki, S. Gómez Sáez, M. Hahn, D. Karastoyanova, G. Koutras, and A. Psycharaki. "Evaluating the Effect of Utility-based Decision Making in Collective Adaptive Systems." In: *Proceedings of the 6th International Conference on Cloud Computing and Service Science (CLOSER 2016)*. Rome, Italy: SciTePress, Apr. 2016, pp. 1–10 (cit. on p. 24).

[ABLS13] V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch. "How to Adapt Applications for the Cloud Environment." In: *Computing* 95.6 (2013), pp. 493–535 (cit. on pp. 13, 19, 20, 33, 34, 48, 53, 90, 182).

[ACC+14] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang. "Quality-of-Service in Cloud Computing: Modeling Techniques and their Applications." In: *Journal of Internet Services and Applications* 5.1 (2014), p. 1 (cit. on pp. 51, 72).

[ADKL14] V. Andrikopoulos, A. Darsow, D. Karastoyanova, and F. Leymann. "CloudDSF–the Cloud Decision Support Framework for Application Migration." In: *Proceedings of Fourth European Conference on Service-Oriented and Cloud Computing (ESOCC'14)*. Springer. 2014, pp. 1–16 (cit. on pp. 22, 46, 54, 55, 283).

[ADM+12] D. Ardagna, E. Di Nitto, P. Mohagheghi, et al. "MODAclouds: A Model-driven Approach for the Design and Execution of Applications on multiple Clouds." In: *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*. IEEE. 2012, pp. 50–56 (cit. on pp. 43, 55, 56, 284).

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. "A

View of Cloud Computing." In: *Communications of the ACM* 53.4 (2010), pp. 50–58 (cit. on pp. 12, 37, 69).

[AGKW13] V. Andrikopoulos, S. Gómez Sáez, D. Karastoyanova, and A. Weiß. "Towards Collaborative, Dynamic & Complex Systems." In: *Proceedings of the 6th International Conference on Service-Oriented Computing and Applications (SOCA'13)*. IEEE Computer Society. IEEE, Dec. 2013, pp. 1–5 (cit. on pp. 24, 254, 264).

[AGKW14] V. Andrikopoulos, S. Gómez Sáez, D. Karastoyanova, and A. Weiß. "Collaborative, Dynamic & Complex Systems: Modeling, Provision & Execution." In: *Proceedings of the Fourth International Conference on Cloud Computing and Service Science (CLOSER'14)*. SciTePress, Apr. 2014, pp. 276–286 (cit. on pp. 24, 254, 264).

[AGLW14] V. Andrikopoulos, S. Gómez Sáez, F. Leymann, and J. Wettinger. "Optimal Distribution of Applications in the Cloud." In: *Advanced Information Systems Engineering*. Ed. by M. Jarke, J. Mylopoulos, and C. Quix. Lecture Notes in Computer Science (LNCS). Springer. Springer, June 2014, pp. 75–90 (cit. on pp. 141–146, 180, 185, 262).

[Ahl07] K. S. Ahluwalia. "Scalability Design Patterns." In: *Proceedings of the 14th Conference on Pattern Languages of Programs (PLoP'07)*. ACM. 2007, p. 2 (cit. on p. 68).

[AJ00] M. Arlitt and T. Jin. "A Workload Characterization Study of the 1998 World Cup Web Site." In: *IEEE Network* 14.3 (2000), pp. 30–37 (cit. on p. 66).

[AM02] V. A. Almeida and D. A. Menascé. "Capacity Planning: an Essential Tool for Managing Web Services." In: *IT professional* 4.4 (2002), pp. 33–38 (cit. on p. 59).

[AMC07] R. Almeida, B. Mozafari, and J. Cho. “On the Evolution of Wikipedia.” In: *Proceedings of the Fifth Conference on Web and Social Media (ICWSM’07)*. 2007 (cit. on p. 98).

[AP94] A. Aamodt and E. Plaza. “Case-based Reasoning: Foundational Issues, Methodological Variations, and System Approaches.” In: *AI communications* 7.1 (1994), pp. 39–59 (cit. on pp. 75–77, 153).

[ARB12] A.-F. Antonescu, P. Robinson, and T. Braun. “Dynamic Topology Orchestration for Distributed Cloud-Based Applications.” In: *Proceedings of the Second Symposium on Network Cloud Computing and Applications (NCCA’12)*. 2012, pp. 116–123 (cit. on p. 73).

[ARML14] V. Andrikopoulos, A. Reuter, X. Mingzhu, and F. Leymann. “Design Support for Cost-efficient Application Distribution in the Cloud.” In: *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD’14)*. IEEE Computer Society, 2014 (cit. on pp. 49, 231).

[ARSL14] V. Andrikopoulos, A. Reuter, Santiago Gómez Sáez, and F. Leymann. “A GENTL Approach for Cloud Application Topologies.” In: *Proceedings of 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC’14)*. Springer, Sept. 2014, pp. 148–159 (cit. on pp. 17, 44, 49, 117, 131, 137, 142).

[ASL13] V. Andrikopoulos, S. Strauch, and F. Leymann. “Decision Support for Application Migration to the Cloud.” In: *Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER’13)* (2013), pp. 149–155 (cit. on pp. 22, 46, 54).

[ASS18] D. Alshammari, J. Singer, and T. Storer. “Performance Evaluation of Cloud Computing Simulation Tools.” In: *2018 IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. IEEE. 2018, pp. 522–526 (cit. on p. 52).

[AV16] A. Amato and S. Venticinque. “Multiobjective Optimization for Brokering of Multicloud Service Composition.” In: *ACM Transactions on Internet Technology (TOIT)* 16.2 (Apr. 2016), 13:1–13:20 (cit. on p. 46).

[BAM+09] R. Bergmann, K.-D. Althoff, M. Minor, M. Reichle, and K. Bach. “Case-Based Reasoning – Introduction and Recent Developments.” In: *KI - Künstliche Intelligenz, German Journal on Artificial Intelligence - Organ des Fachbereiches "Künstliche Intelligenz" der Gesellschaft für Informatik e.V. (KI)* 23.1 (Jan. 2009), pp. 5–11 (cit. on p. 77).

[BBF+18] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, and F. Leymann. “A Systematic Review of Cloud Modeling Languages.” In: *ACM Comput. Surv.* 51.1 (Feb. 2018), 22:1–22:38. URL: http://doi.acm.org/10.1145/3150227 (cit. on p. 41).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner. “OpenTOSCA - A Runtime for TOSCA-based Cloud Applications.” In: *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC’13)*. Vol. 8274. LNCS. Springer Berlin Heidelberg, Dec. 2013, pp. 692–695 (cit. on pp. 207, 248).

[BBKL14a] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. “TOSCA: Portable Automated Deployment and Management of Cloud Applications.” In: *Advanced Web Services*. Springer, 2014, pp. 527–549 (cit. on pp. 41, 42, 117, 123, 141, 143, 207, 218, 220, 248).

[BBKL14b] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann. “Vinothek - A Self-Service Portal for TOSCA.” In: *Proceedings of the 6th Central-European Workshop on Services and their Composition (ZEUS 2014)*. Ed. by N. Herzberg and M. Kunze. Vol. 1140. CEUR Workshop Proceedings. CEUR-WS.org, März 2014, pp. 69–72 (cit. on p. 207).

[BCR94] V. R. Basili, G. Caldiera, and H. D. Rombach. “Experience Factory.” In: *Encyclopedia of Software Engineering* (1994) (cit. on p. 85).

[BED+03] R. Bardohl, H. Ehrig, J. De Lara, O. Runge, G. Taentzer, and I. Weinhold. *Node Type Inheritance Concept for Typed Graph Transformation*. Technische Universität Berlin, Fakultät IV-Elektrotechnik und Informatik, 2003 (cit. on pp. 143–145).

[Ben02] E. Bengoetxea. “Inexact Graph Matching Using Estimation of Distribution Algorithms.” PhD thesis. Paris, France: Ecole Nationale Supérieure des Télécommunications, Dec. 2002 (cit. on p. 167).

[Ber02] R. Bergmann. *Experience management: foundations, development methodology, and internet-based applications*. Springer-Verlag, 2002 (cit. on pp. 78, 80).

[BFF+10] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. “Characterizing, Modeling, and Generating Workload Spikes for Stateful Services.” In: *Proceedings of the 1st ACM Symposium on Cloud computing*. ACM. 2010, pp. 241–252 (cit. on p. 66).

[BFI+15] A. Brogi, M. Fazzolari, A. Ibrahim, J. Soldani, J. Carrasco, J. Cubo, F. Durán, E. Pimentel, E. Di Nitto, and F. D Andria. “Adaptive Management of Applications across Multiple Clouds: The SeaClouds Approach.” In: *CLEI Electronic Journal* 18.1 (2015), pp. 2–2 (cit. on p. 55).

[BFL+12] T. Binz, C. Fehling, F. Leymann, A. Nowak, and D. Schumm. "Formalizing the Cloud through Enterprise Topology Graphs." In: *Proceedings of the 2012 IEEE International Conference on Cloud Computing (CLOUD'12)*. IEEE Computer Society Conference Publishing Services, June 2012 (cit. on pp. 17, 39).

[BGPV11] L. Badger, T. Grance, R. Patt-Corner, and J. Voas. "Cloud Computing Synopsis and Recommendations (draft), NIST Special Publication 800-146." In: *Recommendations of the National Institute of Standards and Technology, Tech. Rep* (2011) (cit. on p. 12).

[BLL+14] A. H. Borhani, P. Leitner, B.-S. Lee, X. Li, and T. Hung. "WPress: Benchmarking Infrastructure-as-a-Service Cloud Computing Systems for On-line Transaction Processing Applications." In: *Proceedings of the 18th IEEE International Enterprise Distributed Object Computing Conference (EDOC'14)*. 2014 (cit. on p. 88).

[BM11] A. Bahga and V. K. Madisetti. "Synthetic Workload Generation for Cloud Computing Applications." In: *Journal of Software Engineering and Applications* 4 (2011), pp. 396–410 (cit. on pp. 64, 65, 67, 128).

[BM12] E. Brandtzæg and P. Mohagheghi. "Towards a Domain-Specific Language to Deploy Applications in the Clouds." In: *In 3rd International Conference on Cloud Computing, GRIDs, and Virtualization*. Citeseer. 2012 (cit. on pp. 43, 56, 123).

[BM95] R. Bisio and F. Malabocchia. "Cost Estimation of Software Projects Through Case-Based Reasoning." In: *International Conference on Case-Based Reasoning*. Springer. 1995, pp. 11–22 (cit. on p. 85).

[Boe+81] B. W. Boehm et al. *Software Engineering Economics*. Vol. 197. Prentice-hall Englewood Cliffs (NJ), 1981 (cit. on p. 85).

[Bre12] E. Brewer. “CAP twelve years later: How the "rules" have changed.” In: *Computer* 45.2 (2012), pp. 23–29 (cit. on p. 35).

[BS03] P. Binkele and S. Schmauder. “An atomistic Monte Carlo Simulation of Precipitation in a Binary System.” In: *Zeitschrift für Metallkunde* 94.8 (2003), pp. 858–863 (cit. on p. 103).

[BS05] D. H. Bailey and A. Snavely. “Performance Modeling: Understanding the Past and Predicting the Future.” In: *Proceedings on the European Conference on Parallel Processing (Euro-Par'05)*. Springer. 2005, pp. 185–195 (cit. on p. 61).

[BS16] A. Brogi and J. Soldani. “Finding Available Services in TOSCA-compliant Clouds.” In: *Science of Computer Programming* 115 (2016), pp. 177–198 (cit. on p. 57).

[BWZ15] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015 (cit. on p. 12).

[CCB+15] R. Chard, K. Chard, K. Bubendorfer, L. Lacinski, R. Madduri, and I. Foster. “Cost-aware Cloud Provisioning.” In: *Proceedings of IEEE 11th International Conference on e-Science (e-Science'15)*. IEEE. 2015, pp. 136–144 (cit. on p. 50).

[CdFD+14] L. Cianciaruso, F. di Forenza, E. Di Nitto, M. Miglierina, N. Ferry, and A. Solberg. “Using Models at Runtime to support adaptable Monitoring of Multi-Clouds Applications.” In: *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'14)*. IEEE. 2014, pp. 401–408 (cit. on pp. 44, 55).

[CH08] W. Cheng and E. Hüllermeier. “Learning Similarity Functions from Qualitative Feedback.” In: *Proceedings of the European Conference on Case-Based Reasoning (ECCBR'08)*. Springer. 2008, pp. 120–134 (cit. on p. 78).

[Cha08] D. Chappell. “What is Application Lifecycle Management.” In: *White Paper, December* (2008) (cit. on p. 113).

[CJMB11] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. “Workload aware Database Monitoring and Consolidation.” In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD’11)*. ACM. 2011, pp. 313–324 (cit. on p. 98).

[CJR01] S. Craw, J. Jarmulak, and R. Rowe. “Learning and Applying Case-Based Adaptation Knowledge.” In: *International Conference on Case-Based Reasoning*. Springer. 2001, pp. 131–145 (cit. on p. 78).

[CQWH12] W. Chen, X. Qiao, J. Wei, and T. Huang. “A Profit-Aware Virtual Machine Deployment Optimization Framework for Cloud Platform Providers.” In: *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD’13)*. Ed. by R. Chang. IEEE, 2012, pp. 17–24 (cit. on p. 50).

[CRB+11] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. “CloudSim: a Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning algorithms.” In: *Software: Practice and Experience* 41.1 (2011), pp. 23–50 (cit. on pp. 55, 284).

[CS93] M. Calzarossa and G. Serazzi. “Workload Characterization: A Survey.” In: *Proceedings of the IEEE* 81.8 (1993), pp. 1136–1150 (cit. on pp. 21, 65).

[DFZ+15] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu. “Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends.” In: *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing (CLOUD’15)*. IEEE. 2015, pp. 621–628 (cit. on p. 38).

[Din16] H. Ding. “Persistence and Discovery of Reusable Cloud Application Topologies.” MA thesis. Univ. of Stuttgart, 2016 (cit. on pp. 142, 152, 215, 225).

[DKLW07] G. Decker, O. Kopp, F. Leymann, and M. Weske. “BPEL4Chor: Extending BPEL for Modeling Choreographies.” In: *Proceedings of the International Conference on Web Services (ICWS'07)*. IEEE Computer Society, 2007, pp. 296–303 (cit. on p. 265).

[dLBE+07] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. “Attributed Graph Transformation with Node Type Inheritance.” In: *Theoretical Computer Science* 376.3 (2007), pp. 139–163 (cit. on p. 143).

[dSdFCM18] H. C. da Silva Filho, G. de Figueiredo Carneiro, E. S. M. Costa, and M. Monteiro. “Tools to Support SMEs to Migrate to the Cloud: Opportunities and Challenges.” In: *Information Technology-New Generations*. Springer, 2018, pp. 159–165 (cit. on p. 44).

[EPM13] T. Erl, R. Puttini, and Z. Mahmood. *Cloud computing: Concepts, Technology, & Architecture*. Pearson Education, 2013 (cit. on pp. 12, 36).

[FACD13] D. Franceschelli, D. Ardagna, M. Ciavotta, and E. Di Nitto. “Space4cloud: A Tool for System Performance and Cost Evaluation of Cloud Systems.” In: *Proceedings of the 2013 International Workshop on Multi-Cloud Applications and Federated Clouds (MultiCloud'13)*. ACM. 2013, pp. 27–34 (cit. on p. 49).

[FEH+14] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn. “A survey of Cloud Monitoring Tools: Taxonomy, Capabilities and Objectives.” In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2918–2933 (cit. on p. 249).

[Fei02] D. G. Feitelson. “Workload Modeling for Performance Evaluation.” In: *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation*. Springer. 2002, pp. 114–141 (cit. on p. 21).

[Fei15] D. G. Feitelson. *Workload modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 2015 (cit. on pp. 59, 63, 64, 67, 128).

[FFH12] F. Fittkau, S. Frey, and W. Hasselbring. "CDOSim: Simulating Cloud Deployment Options for Software Migration Support." In: *2012 IEEE 6th International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA)*. IEEE. 2012, pp. 37–46 (cit. on p. 54).

[FFRR15] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. "An Updated Performance Comparison of Virtual Machines and Linux Containers." In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2015, pp. 171–172 (cit. on p. 88).

[FH11] S. Frey and W. Hasselbring. "The CloudMIG approach: Model-based Migration of Software Systems to Cloud-optimized Applications." In: *International Journal on Advances in Software* 4.3 and 4 (2011), pp. 342–353 (cit. on pp. 55, 56).

[Fis70] P. C. Fishburn. *Utility Theory for Decision Making*. Tech. rep. DTIC Document, 1970 (cit. on pp. 70, 182).

[FL14] M. Fowler and J. Lewis. "Microservices." In: *Viittattu* 28 (2014), p. 2015 (cit. on p. 37).

[FLR+11] C. Fehling, F. Leymann, R. Retter, D. Schumm, and W. Schupeck. "An Architectural Pattern Language of Cloud-based Applications." In: *Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP'11)*. ACM, 2011, pp. 49–62 (cit. on p. 67).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. *Cloud Computing Patterns*. Springer Wien, Jan. 2014, p. 394 (cit. on pp. 34–36, 89, 128, 138).

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley Professional, 2002 (cit. on pp. 13, 92).

[FPLH11] A. Floratou, J. M. Patel, W. Lang, and A. Halverson. "When Free is not really Free: What does it Cost to run a Database Workload in the Cloud?" In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2011, pp. 163–179 (cit. on p. 50).

[FRC+13] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg. "Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-Cloud Systems." In: *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD'13)*. IEEE Computer Society, 2013, pp. 887–894 (cit. on p. 43).

[Fre16] F. Frech. "Utility-based Analysis of Evolving Cloud Application Topologies." Diploma Thesis. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Jan. 2016 (cit. on pp. 235, 236, 238, 239).

[FRL13] S. Frey, C. Reich, and C. Lüthje. "Key Performance Indicators for Cloud Computing SLAs." In: *Proceedings of the Fifth International Conference on Emerging Network Intelligence (EMERGING'13)*. 2013, pp. 60–64 (cit. on p. 61).

[GAB+18] S. Gómez Sáez, V. Andrikopoulos, M. Bitsaki, F. Leymann, and A. van Hoorn. "Utility-Based Decision Making for Migrating Cloud-Based Applications." In: *ACM Transactions on Internet Technology (TOIT)* 18.2 (Feb. 2018), 22:1–22:22 (cit. on pp. 180, 183, 235, 254, 255, 258–260, 262).

[GAH+15] S. Gómez Sáez, V. Andrikopoulos, M. Hahn, D. Karastoyanova, F. Leymann, M. Skouradaki, and K. Vukojevic-Haupt. "Performance and Cost Evaluation for the Migration of a Scientific Workflow Infrastructure to the Cloud." In: *Proceedings of the*

5th International Conference on Cloud Computing and Service Science (CLOSER'15). SciTePress, May 2015, pp. 352–361 (cit. on pp. 102, 107).

[GAH+16] S. Gómez Sáez, V. Andrikopoulos, M. Hahn, D. Karastoyanova, F. Leymann, M. Skouradaki, and K. Vukojevic-Haupt. "Performance and Cost Trade-Off in IaaS Environments: A Scientific Workflow Simulation Environment Case Study." In: vol. 581. Cloud Computing and Services Science. Springer, Feb. 2016, pp. 153–170 (cit. on p. 102).

[GAL16] S. Gómez Sáez, V. Andrikopoulos, and F. Leymann. "Consolidation of Performance and Workload Models in Evolving Cloud Application Topologies." In: *Proceedings of the 6th International Conference on Cloud Computing and Service Science (CLOSER'16)*. Rome, Italy: SciTePress, Apr. 2016, pp. 160–169 (cit. on pp. 125, 128, 132, 135, 138, 186, 211).

[GALS14a] S. Gómez Sáez, V. Andrikopoulos, F. Leymann, and S. Strauch. "Design Support for Performance Aware Dynamic Application (Re-)Distribution in the Cloud." In: *IEEE Transactions on Services Computing* 8.2 (Dec. 2014), pp. 225–239 (cit. on pp. 16, 94–98, 112, 115).

[GALS14b] S. Gómez Sáez, V. Andrikopoulos, F. Leymann, and S. Strauch. "Towards Dynamic Application Distribution Support for Performance Optimization in the Cloud." In: *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD'14)*. IEEE Computer Society, June 2014, pp. 248–255 (cit. on pp. 17, 92).

[Gan15] K. Ganguly. "Performance Aware Cloud Application Topology Enrichment." Diploma Thesis. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Apr. 2015 (cit. on p. 125).

[GAWM14] S. Gómez Sáez, V. Andrikopoulos, F. Wessling, and C. C. Marquezan. “Cloud Adaptation & Application (Re-)Distribution: Bridging the two Perspectives.” In: *Proceedings of the first International Workshop on Engineering Cloud Applications & Services (EnCASE'14)*. IEEE Computer Society Press, Sept. 2014, pp. 163–172 (cit. on p. 112).

[GB16] D. Gesvindr and B. Buhnova. “Performance Challenges, Current Bad Practices, and Hints in PaaS Cloud Application Design.” In: *ACM SIGMETRICS Performance Evaluation Review* 43.4 (2016), pp. 3–12 (cit. on pp. 17, 18, 58, 88).

[GCF+10] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. “Statistics-driven Workload Modeling for the Cloud.” In: *Proceedings of the IEEE 26th International Conference on Data Engineering Workshops (ICDEW'10)*. IEEE. 2010, pp. 87–92 (cit. on p. 68).

[GP08] D. Georgakopoulos and M. P. Papazoglou. *Service-Oriented Computing*. The MIT Press, 2008 (cit. on p. 199).

[GP11] H. Goudarzi and M. Pedram. “Multi-dimensional SLA-based Resource Allocation for Multi-Tier Cloud Computing Systems.” In: *Proceedings of the 2011 IEEE International Conference on Cloud Computing (CLOUD'11)*. IEEE. 2011, pp. 324–331 (cit. on p. 73).

[Gra08] J. Gray. “Distributed Computing Economics.” In: *Queue* 6.3 (May 2008), pp. 63–68 (cit. on p. 90).

[GRCK07] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. “Workload Analysis and Demand Prediction of Enterprise Data Center Applications.” In: *Proceedings of the IEEE 10th International Symposium on Workload Characterization (IISWC'07)*. 2007, pp. 171–180 (cit. on pp. 66, 68, 97).

[HBS+16] K. Hwang, X. Bai, Y. Shi, M. Li, W.-G. Chen, and Y. Wu. “Cloud Performance Modeling with Benchmark Evaluation of Elastic Scaling Strategies.” In: *IEEE Transactions on Parallel and Distributed Systems* 27.1 (2016), pp. 130–143 (cit. on pp. 62, 69).

[HK10] C. N. Hoefer and G. Karagiannis. “Taxonomy of Cloud Computing Services.” In: *2010 IEEE Globecom Workshops*. IEEE. 2010, pp. 1345–1350 (cit. on pp. 13, 38, 39).

[HK11] C. Höfer and G. Karagiannis. “Cloud Computing Services: Taxonomy and Comparison.” In: *Journal of Internet Services and Applications* 2.2 (2011), pp. 81–94 (cit. on pp. 13, 39).

[HK96] K. Hanney and M. T. Keane. “Learning Adaptation Rules from a Case-Base.” In: *European Workshop on Advances in Case-Based Reasoning*. Springer. 1996, pp. 179–192 (cit. on p. 78).

[HM11] J. Humble and J. Molesky. “Why Enterprises must Adopt DevOps to enable Continuous Delivery.” In: *Cutter IT Journal* 24.8 (2011), p. 6 (cit. on pp. 13, 37).

[HSS+10] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. “Cloudward Bound: Planning for beneficial Migration of Enterprise Applications to the Cloud.” In: *ACM SIGCOMM Computer Communication Review*. Vol. 40. 4. ACM. 2010, pp. 243–254 (cit. on pp. 55, 57).

[Hub14] N. M. Huber. “Autonomic Performance-Aware Resource Management in Dynamic IT Service Infrastructures.” PhD thesis. Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2014, 2014 (cit. on p. 61).

[HvHK+14] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev. “Modeling Run-time Adaptation at the System Architecture Level in Dynamic Service-Oriented Environments.” In: *Service*

Oriented Computing and Applications 8.1 (2014), pp. 73–89 (cit. on p. 68).

[HY10] R. Harms and M. Yamartino. “The Economics of the Cloud.” In: *Microsoft whitepaper, Microsoft Corporation* (2010) (cit. on p. 181).

[Ing87] J. E. Ingersoll. *Theory of Financial Decision Making*. Vol. 3. Rowman & Littlefield, 1987 (cit. on p. 70).

[INS+14] C. Inzinger, S. Nastic, S. Sehic, M. Voegler, F. Li, and S. Dustdar. “MADCAT - A Methodology For Architecture And Deployment Of Cloud Application Topologies.” In: *Proceedings of the 8th IEEE International Symposium on Service Oriented System Engineering (SOSE’14)*. IEEE. 2014 (cit. on pp. 46, 120).

[IYE11] A. Iosup, N. Yigitbasi, and D. Epema. “On the Performance Variability of Production Cloud Services.” In: *Proceedings of the 11th IEEE/ACM international symposium on Cluster, Cloud and Grid Computing (CCGrid’11)*. Ed. by IEEE. 2011, pp. 104–113 (cit. on pp. 69, 89).

[JAP13] P. Jamshidi, A. Ahmad, and C. Pahl. “Cloud Migration Research: A Systematic Review.” In: *IEEE Transactions on Cloud Computing* 1.2 (2013), pp. 142–157 (cit. on pp. 20, 45, 111).

[JGFB18] C. Jatoth, G. Gangadharan, U. Fiore, and R. Buyya. “SELCLOUD: a Hybrid Multi-criteria Decision-making Model for Selection of Cloud Services.” In: *Soft Computing* (2018), pp. 1–15 (cit. on p. 52).

[JPM16] P. Jamshidi, C. Pahl, and N. C. Mendonça. “Pattern-based Multi-Cloud Architecture Migration.” In: *Software Practice and Experience* (2016) (cit. on p. 54).

[JVS98] L. K. John, P. Vasudevan, and J. Sabarinathan. “Workload Characterization: Motivation, Goals and Methodology.” In: *Proceedings of the first Workload on Workload Characterization (WWC’98)*. 1998 (cit. on p. 63).

[Kat97] B. Katzy. “The Business Architect—The Concept of Enterprise Integration Revisited.” In: *Enterprise Engineering and Integration*. Springer, 1997, pp. 329–338 (cit. on p. 18).

[Kav14] M. J. Kavis. *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS)*. John Wiley & Sons, 2014 (cit. on pp. 32, 35).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann. “Winery - A Modeling Tool for TOSCA-based Cloud Applications.” In: *Proceedings of the 10th International Conference on Service-Oriented Computing (ICSOC’13)*. Vol. 8274. LNCS. Springer Berlin Heidelberg, Dec. 2013, pp. 700–704 (cit. on pp. 207, 242, 247, 248).

[KC03] J. O. Kephart and D. M. Chess. “The Vision of Autonomic Computing.” In: *Computer* 36.1 (2003), pp. 41–50 (cit. on p. 71).

[KD07] J. O. Kephart and R. Das. “Achieving Self-management via Utility Functions.” In: *IEEE Internet Computing* 11.1 (2007), pp. 40–48 (cit. on p. 72).

[Kep11] B. Kepes. “Cloudonomics: the Economics of Cloud Computing.” In: *Diversity Limited* (2011) (cit. on p. 181).

[KGSS12] A. Khajeh-Hosseini, D. Greenwood, J. W. Smith, and I. Sommerville. “The Cloud Adoption Toolkit: Supporting Cloud Adoption Decisions in the Enterprise.” In: *Software: Practice and Experience* 42.4 (2012), pp. 447–465 (cit. on pp. 45, 55, 57).

[KH18] K. Kritikos and G. Horn. “IaaS Service Selection Revisited.” In: *Proceedings of the European Conference of Service-Oriented and Cloud Computing (ESOCC’18)*. IEEE. 2018, pp. 170–184 (cit. on p. 180).

[KKMV12] G. Kousiouris, D. Kyriazis, A. Menychtas, and T. Varvarigou. “Legacy Applications on the Cloud: Challenges and Enablers focusing on Application Performance Analysis and Providers Characteristics.” In: *Proceedings of the IEEE 2nd International Conference on Cloud Computing and Intelligence Systems (CCIS’12)*. Vol. 2. IEEE. 2012, pp. 603–608 (cit. on pp. 44, 51, 52).

[KMP16] K. Kritikos, K. Magoutis, and D. Plexousakis. “Towards Knowledge Based Assisted IaaS Selection.” In: *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom’16)*. IEEE. 2016, pp. 431–439 (cit. on pp. 52, 74).

[Kol13] B. Koller. “Model Based Cloud Application Development using PaaSage.” In: *Innovatives Supercomputing in Deutschland* 11.1 (2013) (cit. on pp. 55, 57).

[KR93] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Trade-offs*. Cambridge University press, 1993 (cit. on pp. 70, 182).

[KSBT11] A. Khajeh-Hosseini, I. Sommerville, J. Bogaerts, and P. Teregowda. “Decision Support Tools for Cloud Migration in the Enterprise.” In: *Proceedings of the 2011 IEEE International Conference on Cloud Computing (CLOUD’11)*. IEEE. 2011, pp. 541–548 (cit. on pp. 23, 45).

[KSHD13] S. Kächele, C. Spann, F. J. Hauck, and J. Domaschka. “Beyond IaaS and PaaS: An extended Cloud Taxonomy for Computation, Storage and Networking.” In: *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud*

Computing. IEEE Computer Society. 2013, pp. 75–82 (cit. on p. 39).

[KYTA12] A. Khan, X. Yan, S. Tao, and N. Anerousis. "Workload Characterization and Prediction in the Cloud: A Multiple Time Series Approach." In: *2012 IEEE Network Operations and Management Symposium*. IEEE. 2012, pp. 1287–1294 (cit. on pp. 21, 66).

[Lal13] B. Laliberte. *Got Clouds? Optimize Application Performance*. ESG Research Report. Jan. 2013 (cit. on pp. 16, 47).

[LC16] P. Leitner and J. Cito. "Patterns in the Chaos & Mdash; A Study of Performance Variation and Predictability in Public IaaS Clouds." In: *ACM Trans. Internet Technol.* 16.3 (Apr. 2016), 15:1–15:23 (cit. on pp. 88, 89).

[Lea96] D. B. Leake. *Case-Based Reasoning: Experiences, Lessons and Future directions*. MIT press, 1996 (cit. on p. 75).

[LFM+11] F. Leymann, C. Fehling, R. Mietzner, A. Nowak, and S. Dustdar. "Moving Applications to the Cloud: An Approach based on Application Model Enrichment." In: *IJCIS* 20.3 (Oct. 2011), pp. 307–356 (cit. on pp. 54, 55).

[LFWW16] F. Leymann, C. Fehling, S. Wagner, and J. Wettinger. "Native Cloud Applications: Why Virtual Machines, Images and Containers Miss the Point!" In: *Proceedings of the 6th International Conference on Cloud Computing and Service Science (CLOSER'16)*. Rome: SciTePress, Apr. 2016, pp. 7–15 (cit. on pp. 34, 37, 45, 90).

[LKHE15] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth. "Performance-based Service Differentiation in Clouds." In: *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'15)*. IEEE. 2015, pp. 505–514 (cit. on pp. 55, 58).

[LML+11] A. Lenk, M. Menzel, J. Lipsky, S. Tai, and P. Offermann. “What are you Paying for? Performance Benchmarking for IaaS Offerings.” In: *Proceedings of the 2011 IEEE International Conference on Cloud Computing (CLOUD'11)*. IEEE. 2011, pp. 484–491 (cit. on pp. 88, 125).

[LPD+11] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas. “Migration of Multi-Tier Applications to Infrastructure-as-a-Service clouds: An Investigation using Kernel-based Virtual Machines.” In: *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing (ICGC'11)*. IEEE. 2011, pp. 137–144 (cit. on p. 45).

[LTRK10] G. Lee, N. Tolia, P. Ranganathan, and R. H. Katz. “Topology-aware Resource Allocation for Data-Intensive Workloads.” In: *Proceedings of the first ACM asia-pacific workshop on Workshop on systems (APSys'10)*. ACM. 2010, pp. 1–6 (cit. on p. 52).

[LZM98] T. W. Liao, Z. Zhang, and C. R. Mount. “Similarity Measures for Retrieval in Case-Based Reasoning Systems.” In: *Applied Artificial Intelligence* 12.4 (1998), pp. 267–288 (cit. on p. 79).

[MAD+11] F. Moscato, R. Aversa, B. Di Martino, T.-F. Fortiş, and V. Munteanu. “An Analysis of Mosaic Ontology for Cloud Resources Annotation.” In: *Proceedings of the 2011 Federated Conference on Computer Science and Information Systems (FedCSIS'11)*. IEEE. 2011, pp. 973–980 (cit. on p. 55).

[Mar09] A. Marshall. *Principles of Economics: Unabridged Eighth Edition*. Cosimo, Inc., 2009 (cit. on pp. 70, 180, 182, 234).

[MBHS10] D. Molnar, P. Binkele, S. Hocker, and S. Schmauder. “Multi-scale Modelling of Nano Tensile Tests for different Cu precipitation States in α-Fe.” In: *Proceedings of the 5th International Conference on Multiscale Materials Modelling (MMM'10)*. 2010, pp. 235–239 (cit. on p. 103).

[MC11] J. M. Merigó and M. Casanovas. “A new Minkowski Distance based on Induced Aggregation Operators.” In: *International Journal of Computational Intelligence Systems* 4.2 (2011), pp. 123–133 (cit. on p. 161).

[MF11] D. Minarolli and B. Freisleben. “Utility-based Resource Allocation for Virtual Machines in Cloud Computing.” In: *Proceedings of the 2011 IEEE Symposium on Computers and Communications (ISCC’11)*. IEEE. 2011, pp. 410–417 (cit. on pp. 73, 180).

[MFB+07] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea. *Performance Testing Guidance for Web Applications: Patterns & Practices*. Microsoft press, 2007 (cit. on p. 68).

[MG10] M. Macias and J. Guitart. “Using Resource-level Information into Non-additive Negotiation Models for Cloud Market Environments.” In: *Proceedings of the 2010 IEEE Network Operations and Management Symposium (NOMS’10)*. IEEE. 2010, pp. 325–332 (cit. on p. 74).

[MG11] P. Mell and T. Grance. “The NIST Definition of Cloud Computing.” In: (2011) (cit. on pp. 12, 33, 37, 87, 89).

[MGAD13] M. Miglierina, G. Gibilisco, D. Ardagna, and E. Di Nitto. “Model based Control for Multi-Cloud Applications.” In: *Proceedings of the 5th International Workshop on Modeling in Software Engineering (MiSE’13)*. 2013, pp. 37–43 (cit. on p. 284).

[MHCD10] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. “Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters.” In: *ACM SIGMETRICS Performance Evaluation Review* 37.4 (2010), pp. 34–41 (cit. on pp. 21, 66).

[MMV13] R. Mian, P. Martin, and J. L. Vazquez-Poletti. “Provisioning Data Analytic Workloads in a Cloud.” In: *FGCS* 29 (2013), pp. 1452–1458 (cit. on p. 49).

[MMZV12] R. Mian, P. Martin, F. Zulkernine, and J. L. Vazquez-Poletti. “Estimating Resource Costs of Data-Intensive Workloads in Public Clouds.” In: *Proceedings of the 10th International Workshop on Middleware for Grids, Clouds and e-Science (MGC’12)*. ACM. 2012, p. 3 (cit. on p. 49).

[MMZV13] R. Mian, P. Martin, F. Zulkernine, and J. L. Vazquez-Poletti. “Towards Building Performance Models for Data-intensive Workloads in Public Clouds.” In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE’13)*. 2013 (cit. on p. 62).

[Moh16] S. Y. M. Mohamed. “Evaluation of Alternative Distributions of a Choreography-based Smart City Transportation Application among Cloud Offerings.” Master Thesis. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Sept. 2016 (cit. on pp. 254, 266, 268).

[Mol09] I. Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. " O’Reilly Media, Inc.", 2009 (cit. on pp. 59–61).

[MR12] M. Menzel and R. Ranjan. “CloudGenius: Decision Support for Web Server Cloud Migration.” In: *Proceedings of the 21st international conference on World Wide Web (WWW’12)*. Lyon, France: ACM, 2012, pp. 979–988 (cit. on pp. 45, 55, 56).

[MST13] M. Menzel, M. Schönherr, and S. Tai. “(MC2) 2: Criteria, Requirements and a Software Prototype for Cloud Infrastructure Decisions.” In: *Software: Practice and experience* 43.11 (2013), pp. 1283–1297 (cit. on pp. 55, 57).

[MUL09] R. Mietzner, T. Unger, and F. Leymann. “Cafe: A Generic Configurable Customizable Composite Cloud Application Framework.” In: *On the Move to Meaningful Internet Systems: OTM 2009*. Springer, 2009, pp. 357–364 (cit. on p. 43).

[MVP92] T. Mukhopadhyay, S. S. Vicinanza, and M. J. Prietula. “Examining the Feasibility of a Case-Based Reasoning Model for Software Effort Estimation.” In: *MIS quarterly* (1992), pp. 155–171 (cit. on p. 85).

[NVC+12] A. Núñez, J. L. Vázquez-Poletti, A. C. Caminero, G. G. Castañé, J. Carretero, and I. M. Llorente. “iCanCloud: A flexible and scalable Cloud Infrastructure Simulator.” In: *Journal of Grid Computing* 10.1 (2012), pp. 185–209 (cit. on pp. 52, 55, 56).

[OAS07] OASIS. *Web Services Business Process Execution Language Version 2.0*. Apr. 2007. URL: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html (cit. on p. 265).

[ÖGW+14] P.-O. Östberg, H. Groenda, S. Wesner, J. Byrne, D. S. Nikolopoulos, C. Sheridan, J. Krzywda, A. Ali-Eldin, J. Tordsson, E. Elmroth, et al. “The CACTOS Vision of Context-Aware Cloud Topology Optimization and Simulation.” In: *2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom’14)*. IEEE. 2014, pp. 26–31 (cit. on pp. 55, 58).

[oNeu94] B. C. ord Neuman. “Scale in Distributed Systems.” In: *ISI/USC* (1994) (cit. on p. 68).

[Par15] D. Parmenter. *Key Performance Indicators: Developing, Implementing, and Using Winning KPIs*. John Wiley & Sons, 2015 (cit. on pp. 60, 61, 126).

[PDL+09] N. Paton, M. A. De Aragão, K. Lee, A. A. Fernandes, and R. Sakellariou. “Optimizing Utility in Cloud Computing through Autonomic Workload Execution.” In: *Bulletin of the Technical Committee on Data Engineering* 32.1 (2009), pp. 51–58 (cit. on p. 73).

[PDV+13] D. Petcu, B. Di Martino, S. Venticinque, M. Rak, T. Máhr, G. E. Lopez, F. Brito, R. Cossu, M. Stopar, S. Šperka, et al. “Experiences in Building a mOSAIC of Clouds.” In: *Journal of Cloud Computing: Advances, Systems and Applications* 2.1 (2013), p. 1 (cit. on pp. 55, 56).

[Pet09] L. Petrazickis. “Deploying PHP Applications on IBM DB2 in the Cloud: MediaWiki as a Case Study.” In: *Proceedings of CASCON’09*. IBM Corp. 2009, pp. 304–305 (cit. on p. 98).

[Pfl95] S. L. Pfleeger. “Experimental Design and Analysis in Software Engineering.” In: *Annals of Software Engineering* 1.1 (1995), pp. 219–253 (cit. on p. 88).

[Pin16] J. V. Pincay. “Knowledge Capturing and Usage of Evolving Cloud Application Topologies.” MA thesis. Univ. of Stuttgart, 2016 (cit. on pp. 125, 127, 129, 142, 152, 153, 163, 168, 170, 171, 175, 225).

[PM13] A. Papaioannou and K. Magoutis. “An Architecture for Evaluating Distributed Application Deployments in Multi-Clouds.” In: *Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom’13)*. Vol. 1. IEEE. 2013, pp. 547–554 (cit. on p. 58).

[Pow08] D. J. Power. “Decision Support Systems: A Historical Overview.” In: *Handbook on Decision Support Systems 1*. Springer, 2008, pp. 121–140 (cit. on p. 53).

[PS04] S. K. Pal and S. C. Shiu. *Foundations of Soft Case-Based Reasoning*. Vol. 8. John Wiley & Sons, 2004 (cit. on p. 75).

[PvdH11] M. P. Papazoglou and W.-J. van den Heuvel. “Blueprinting the Cloud.” In: *Internet Computing* 15.6 (2011), pp. 74–79 (cit. on pp. 40, 42, 117, 123, 143).

[QRD16] C. Quinton, D. Romero, and L. Duchien. “SALOON: A Platform for Selecting and Configuring Cloud Environments.” In: *Software: Practice and Experience* 46.1 (2016), pp. 55–78 (cit. on p. 46).

[RAKJ15] K. RahimiZadeh, M. AnaLoui, P. Kabiri, and B. Javadi. “Performance Modeling and Analysis of Virtualized Multi-Tier Applications under Dynamic Workloads.” In: *Journal of Network and Computer Applications* 56 (2015), pp. 166–187 (cit. on pp. 21, 63).

[RCL09] B. P. Rimal, E. Choi, and I. Lumb. “A Taxonomy and Survey of Cloud Computing Systems.” In: *INC, IMS and IDC* (2009), pp. 44–51 (cit. on pp. 39, 46).

[Ree09] G. Reese. *Cloud Application Architectures: Building Applications and Infrastructure in the Cloud*. " O’Reilly Media, Inc.", 2009 (cit. on p. 35).

[Ric04] M. M. Richter. “Logic and Approximation in Knowledge Based Systems.” In: *Logic vs. Approximation*. Springer, 2004, pp. 184–203 (cit. on p. 82).

[Ric07] M. M. Richter. “Foundations of Similarity and Utility.” In: *FLAIRS Conference*. 2007, pp. 30–37 (cit. on p. 80).

[Ric08] M. M. Richter. “Similarity.” In: *Case-Based Reasoning on Images and Signals*. Springer, 2008, pp. 25–90 (cit. on pp. 82, 164).

[Ric93] M. M. Richter. “Classification and Learning of Similarity Measures.” In: *Information and Classification*. Springer, 1993, pp. 323–334 (cit. on p. 80).

[Rob+08] S. Robison et al. “The Next Wave: Everything as a service.” In: *Executive Viewpoint: www.hp.com* (2008) (cit. on p. 39).

[RR08] L. Richardson and S. Ruby. *RESTful Web Services*. " O’Reilly Media, Inc.", 2008 (cit. on p. 200).

[RWE13] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O’Reilly Media, Inc., 2013 (cit. on p. 214).

[RZ13] N. Ranaldo and E. Zimeo. “Capacity-aware Utility Function for SLA Negotiation of Cloud Services.” In: *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC’13)*. IEEE. 2013, pp. 292–296 (cit. on p. 74).

[SAB+13] S. Strauch, V. Andrikopoulos, T. Bachmann, D. Karastoyanova, S. Passow, and K. Vukojevic-Haupt. “Decision support for the migration of the application database layer to the cloud.” In: *Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom’13)*. Vol. 1. IEEE. 2013 (cit. on pp. 22, 283).

[SAK+14] S. Strauch, V. Andrikopoulos, D. Karastoyanova, F. Leymann, N. Nachev, and A. Stäbler. “Migrating enterprise applications to the cloud: methodology and evaluation.” In: *International Journal of Big Data Intelligence 5* 1.3 (2014), pp. 127–140 (cit. on p. 120).

[SAT+13] S. Strauch, V. Andrikopoulos, B. Thomas, D. Karastoyanova, S. Passow, and K. Vukojevic-Haupt. “Decision Support for the Migration of the Application Database Layer to the Cloud.” In: *Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom’13)*. Vol. 1. IEEE. 2013, pp. 639–646 (cit. on p. 34).

[SBB+16] J. Soldani, T. Binz, U. Breitenbücher, F. Leymann, and A. Brogi. “ToscaMart: A Method for Adapting and Reusing Cloud Applications.” In: *Journal of Systems and Software* 113 (2016), pp. 395–406 (cit. on pp. 55–57, 120).

[SCD+97] B. Sabata, S. Chatterjee, M. Davis, J. J. Sydir, and T. F. Lawrence. “Taxonomy for QoS specifications.” In: *Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems*. IEEE. 1997, pp. 100–107 (cit. on p. 51).

[Sch09] H. E. Schaffer. “X as a Service, Cloud Computing, and the Need for Good Judgment.” In: *IT professional* 11.5 (2009), pp. 4–5 (cit. on pp. 15, 37, 44).

[Sch83] R. C. Schank. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press, 1983 (cit. on p. 75).

[SDQ10] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. “Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance.” In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 460–471 (cit. on pp. 69, 88, 125).

[She03] M. Shepperd. “Case-based Reasoning and Software Engineering.” In: *Managing Software Engineering Knowledge*. Springer, 2003, pp. 181–198 (cit. on pp. 75, 76, 85, 152).

[SHK+11] M. Sonntag, S. Hotta, D. Karastoyanova, D. Molnar, and S. Schmauder. “Workflow-based Distributed Environment for Legacy Simulation Applications.” In: *ICSOFT (1)*. 2011, pp. 91–94 (cit. on p. 104).

[SK10] M. Sonntag and D. Karastoyanova. “Next Generation Interactive Scientific Experimenting Based On The Workflow Technology.” In: *Proceedings of the 21st IASTED International Conference on Modelling and Simulation (MS’10)*. Ed. by R. Alhajj,

V. Leung, M. Saif, and R. Thring. ACTA Press, July 2010 (cit. on pp. 103, 104).

[SKK+01] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. N. Reddy. "Performance Specification of Software Components." In: *ACM SIGSOFT Software Engineering Notes*. Vol. 26. 3. ACM. 2001, pp. 3–10 (cit. on p. 61).

[SKLU11] S. Strauch, O. Kopp, F. Leymann, and T. Unger. "A Taxonomy for Cloud Data Hosting Solutions." In: *Proceedings of the International Conference on Cloud and Green Computing (CGC '11)*. IEEE Computer Society, Dec. 2011 (cit. on p. 39).

[SM01] B. Smyth and E. McKenna. "Competence Models and the Maintenance Problem." In: *Computational Intelligence* 17.2 (2001), pp. 235–249 (cit. on p. 78).

[SME14] S. Soltani, P. Martin, and K. Elgazzar. "QuARAM Recommender: Case-Based Reasoning for IaaS Service Selection." In: *Proceedings of the 2014 International Conference on Cloud and Autonomic Computing (ICCAC'14)*. IEEE. 2014, pp. 220–226 (cit. on p. 85).

[Sol16] S. Soltani. "IaaS Cloud Service Selection using Case-Based Reasoning." PhD thesis. 2016 (cit. on p. 75).

[SSJL12] B. Suleiman, S. Sakr, R. Jeffery, and A. Liu. "On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure." In: *Journal of Internet Services and Applications* 3.2 (2012), pp. 173–193 (cit. on p. 36).

[Sta03] A. Stahl. *Learning of Knowledge-Intensive Similarity Measures in Case-Based Reasoning*. dissertation. de, 2003 (cit. on pp. 80–84, 86, 160, 161).

[Sta05] A. Stahl. “Learning Similarity Measures: A Formal View based on a Generalized CBR Model.” In: *Proceedings of the 2005 International Conference on Case-Based Reasoning (ICCBR'05)*. Springer. 2005, pp. 507–521 (cit. on pp. 78–81, 159).

[STFG08] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. “Using Utility to Provision Storage Systems.” In: *FAST*. Vol. 8. 2008, pp. 1–16 (cit. on pp. 70, 73, 180, 182).

[Str08] J. D. Strunk. *Using Utility Functions to Control a Distributed Storage System*. ProQuest, 2008 (cit. on p. 74).

[SW02] C. U. Smith and L. G. Williams. “New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot.” In: *Int. CMG Conference*. 2002, pp. 667–674 (cit. on p. 62).

[SW03] C. U. Smith and L. G. Williams. “More new Software Performance Antipatterns: Even more Ways to Shoot Yourself in the Foot.” In: *Computer Measurement Group Conference*. Citeseer. 2003, pp. 717–725 (cit. on p. 62).

[TD10] H.-L. Truong and S. Dustdar. “Composable Cost Estimation and Monitoring for Computational Applications in Cloud Computing Environments.” In: *Procedia Computer Science* 1.1 (2010), pp. 2175–2184 (cit. on p. 50).

[TKS+11] K. Tsakalozos, H. Kllapi, E. Sitaridi, M. Roussopoulos, D. Paparas, and A. Delis. “Flexible Use of Cloud Resources Through Profit Maximization and Price Discrimination.” In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE'11)*. IEEE. 2011, pp. 75–86 (cit. on p. 50).

[Tru11] C. Trubiani. “Automated Generation of Architectural Feedback from Software Performance Analysis Results.” PhD thesis. PhD thesis, Univ. of L'Aquila, 2011 (cit. on p. 62).

[Tru15] C. Trubiani. “Introducing Software Performance Antipatterns in Cloud Computing Environments: Does it Help or Hurt?” In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE’15)*. ACM. 2015, pp. 207–210 (cit. on p. 62).

[Tur14] J. Turnbull. *The Docker Book*. Lulu. com, 2014 (cit. on p. 250).

[TV07] A. S. Tanenbaum and M. Van Steen. *Distributed Systems*. Prentice Hall, 2007 (cit. on p. 39).

[UDS+14] M. Unuvar, Y. Doganata, M. Steinder, A. Tantawi, and S. Tosi. “A Predictive Method for Identifying Optimum Cloud Availability Zones.” In: *Proceedings of the 2014 IEEE 7th International Conference on Cloud Computing (CLOUD’14)*. IEEE. 2014, pp. 72–79 (cit. on pp. 52, 74).

[UK12] G. Ucoluk and S. Kalkan. *Introduction to Programming Concepts with Case Studies in Python*. SpringerLink : Bücher. Springer Vienna, 2012 (cit. on p. 238).

[UPV09] G. Urdaneta, G. Pierre, and M. Van Steen. “Wikipedia Workload Analysis for Decentralized Hosting.” In: *Computer Networks* 53.11 (July 2009), pp. 1830–1845 (cit. on p. 98).

[UTD+15] M. Unuvar, S. Tosi, Y. N. Doganata, M. G. Steinder, and A. N. Tantawi. “Selecting Optimum Cloud Availability Zones by Learning User Satisfaction Levels.” In: *IEEE Transactions on Services Computing* 8.2 (2015), pp. 199–211 (cit. on p. 69).

[Var10] J. Varia. “Architecting for the Cloud: Best Practices.” In: *Amazon Web Services* (2010) (cit. on pp. 12, 32, 33, 35, 37, 277).

[VRB11] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. “Dynamically Scaling Applications in the Cloud.” In: *ACM SIGCOMM Computer Communication Review* 41.1 (2011), pp. 45–52 (cit. on pp. 35, 36).

[VRH08] A. Van Hoorn, M. Rohr, and W. Hasselbring. “Generating Probabilistic and Intensity-varying Workload for Web-Based Software Systems.” In: *Performance Evaluation: Metrics, Models and Benchmarks*. Springer, 2008, pp. 124–143 (cit. on pp. 64, 67, 137).

[WAB01] R. Weber, D. W. Aha, and I. Becerra-Fernandez. “Intelligent Lessons Learned Systems.” In: *Expert systems with applications* 20.1 (2001), pp. 17–34 (cit. on p. 85).

[WB12] L. Wu and R. Buyya. “Service Level Agreement (SLA) in Utility Computing Systems.” In: *IGI Global* (2012) (cit. on p. 51).

[WCG+09] Z. Wang, Y. Chen, D. Gmach, S. Singhal, B. J. Watson, W. Rivera, X. Zhu, and C. D. Hyser. “AppRAISE: Application-level Performance Management in Virtualized Server Environments.” In: *IEEE Transactions on Network and Service Management* 6.4 (2009), pp. 240–254 (cit. on p. 62).

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005 (cit. on p. 199).

[WFP07] M. Woodside, G. Franks, and D. C. Petriu. “The Future of Software Performance Engineering.” In: *Future of Software Engineering (FOSE’07)*. IEEE. 2007, pp. 171–187 (cit. on p. 59).

[WMG+10] B. J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, and Z. Wang. “Probabilistic Performance Modeling of Virtualized Resource Allocation.” In: *Proceedings of the 7th international conference on Autonomic computing (ICAC’10)*. 2010 (cit. on p. 62).

[WT06] A. Weber and R. Thomas. “Key Performance Indicators: Measuring and Managing the Maintenance Function.” In: *Ivara Corporation* (2006) (cit. on p. 60).

[WTKD04] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. “Utility Functions in Autonomic Systems.” In: *Proceedings of the 2004 International Conference on Autonomic Computing (ICAC'04)*. IEEE. 2004, pp. 70–77 (cit. on p. 72).

[WWB+13] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, B. Mitschang, A. Nowak, and S. Wagner. “Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing.” In: *Proceedings of the 2013 On the Move to Meaningful Internet Systems (OTM'13)*. 2013 (cit. on pp. 42, 124, 211, 212).

[XA13] M. Xiu and V. Andrikopoulos. *The Nefolog & MiDSuS Systems for Cloud Migration Support*. Technical Report 2013/08. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Nov. 2013, p. 19 (cit. on pp. 49, 231).

[YBZ12] Z. Ye, A. Bouguettaya, and X. Zhou. “QoS-aware Cloud Service Composition based on Economic Models.” In: *Proceedings of the 2012 IEEE International Conference on Service Oriented Computing (ICSOC'12)*. Springer. 2012, pp. 111–126 (cit. on p. 50).

[YBZ14] Z. Ye, A. Bouguettaya, and X. Zhou. “Economic Model-Driven Cloud Service Composition.” In: *ACM Trans. Internet Technol.* 14.2-3 (Oct. 2014), 20:1–20:19 (cit. on p. 50).

[YZB11] Z. Ye, X. Zhou, and A. Bouguettaya. “Genetic Algorithm based QoS-aware Service Compositions in Cloud Computing.” In: *Proceedings of the 23rd International Conference on Database Systems for Advanced Applications (DASFAA'18)*. Springer. 2011, pp. 321–334 (cit. on p. 52).

[YZSD12] L. Ye, H. Zhang, J. Shi, and X. Du. “Verifying Cloud Service Level Agreement.” In: *Proceedings of the 2012 IEEE Global Communications Conference (GLOBECOM’12)*. 2012 (cit. on p. 88).

[ZDB+13] D. Zeginis, F. D’andria, S. Bocconi, J. Gorronogoitia Cruz, O. Collell Martin, P. Gouvas, G. Ledakis, and K. A. Tarabanis. “A User-Centric Multi-PaaS Application Management Solution for Hybrid Multi-Cloud Scenarios.” In: *Scalable Computing: Practice and Experience* 14.1 (2013), pp. 17–32 (cit. on pp. 55, 58).

All URLs were verified on the November 3, 2019.