FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Transparent Application Adjustment for Efficient and Elastic Execution in the Cloud

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

Rostyslav Zabolotnyi
Registration Number 1128386

to the Faculty of Informatics
at the TU Wien

Advisor: Univ.Prof. Schahram Dustdar

External reviewers:
Adam Barker. University of St Andrews, England.
Frank Leymann. University of Stuttgart, Germany.

Vienna, 24th August, 2015

Rostyslav Zabolotnyi	Schahram Dustdar

Declaration of Authorship

Rostyslav Zabolotnyi
Tigergasse 23-27 20/2, Wien, Austria

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 24th August, 2015

Rostyslav Zabolotnyi

Acknowledgements

This PhD thesis defines an ultimate accomplishment of my academic research career in the Distributed Systems Group at TU Wien. Looking back over the last four years, I can clearly confirm that it was an interesting experience that broadened my understanding of computer science and university life.

First, I would like to thank my aunt Oksana Senchuk for finding and recommending me such an interesting opportunity. Likewise, I would like to thank Prof. Hannes Werthner, Prof. Hans Tompits and Ms. Clarissa Schmid for welcoming me as a participant of the Vienna PhD School of Informatics and helping me with any study- or research-related issue I had. Equally, I would like to express my gratitude to Prof. Schahram Dustdar that agreed to work with me, accepted me in the Distributed Systems Group, and guided me through my research and studies.

Furthermore, I want to thank all my colleagues from the Distributed Systems Group and the PhD School for the fruitful discussions, support and collaboration. Notably, I appreciate Philipp Leitner, Waldemar Hummer, Vitaliy Liptchinsky, Benjamin Satzger, and Alessio Gambi for the help in shaping my research and a fruitful cooperation that developed into the results that I am presenting in this thesis. Similarly, I am thankful to Philipp Hoenisch who served me as the good example in almost every research and development-related activity. I am also immensely grateful to Ass. Prof. Stefan Schulte, whose facilitation and guidance over the last two years were perhaps the main reason I actually completed my research work at all.

My special thanks are devoted to my family and friends, particularly to Serwah Sabetghadam, Konstantin Selyunin, and Maryna Kostikova, who supported me emotionally, added colors, and adventures during my stay in Vienna.

Finally, I am grateful for financial support from the Vienna PhD School of Informatics and European Community's Seventh Framework Programme under grant agreement 318201 (SIMPLI-CITY).

Abstract

Cloud computing rapidly surpassed the phase of initial adoption and quickly gains momentum on the market of information technologies. Companies, startups and regular users leverage the cloud to avoid infrastructure or middleware costs, to gain flexibility in computing usage and to obtain unlimited computational or storage resources available on demand. However, along with cloud computing's benefits, new challenges arrived. In order to achieve the advantages of the cloud, developers have to redesign their existing applications and build new ones with scalability and elasticity in mind. Additionally, as the market of cloud providers developed, two competing application development paradigms emerged. When bringing an application to the cloud, developers have to decide if they follow the Infrastructure as a Service model, which provides flexibility and software architecture freedom, or the Platform as a Service model that offers a higher level of abstraction and a simpler application development process.

This thesis addresses emerging cloud computing challenges presenting a transparent application distribution approach based on the JCloudScale middleware. The described cloud application development approach hides boilerplate cloud interaction code from developers and allows focusing on the business logic of the application instead. Providing benefits common to Platform as a Service solutions, the discussed approach allows keeping flexibility and freedom that is missing in alternatives. However, this approach brings in a set of distinctive challenges, that are targeted in this work. To solve the issue of transparent application code integrity and synchronization, a solution for seamless code distribution was built. To simplify the complexity of elastic application management, a scaling definition language based on complex event processing application architecture was designed. Finally, targeting effective cloud resource usage, a profiling-based task scheduling solution was proposed. Each solution and framework, presented in this thesis, denote the steps on the ongoing road to achieve the declared goal of transparent cloud application distribution.

Developed approaches and solutions were thoroughly validated using accomplished user studies and performance evaluations. The obtained results show that the presented transparent cloud application development approach is appealing to developers, increases productivity and simplifies cloud migration or cloud application construction without significant performance costs.

Contents

List of Figures

List of Tables

List of Listings

Earlier Publications

This thesis is based on previous work published in scientific conferences, workshops, books and journals. For reasons of brevity, these core papers, which build the foundation of this thesis, are listed here once, and will generally not be explicitly referenced again. Parts of these papers are contained in verbatim. Please refer to Appendix 9.3 for a full publication list of the author of this thesis.

- Rostyslav Zabolotnyi, Philipp Leitner, Waldemar Hummer, Schahram Dustdar "JCloudScale: Closing the Gap Between IaaS and PaaS", *Transactions on Internet Technology (TOIT)*, vol. 15, no. 3, Jul. 2015.
- Rostyslav Zabolotnyi, Philipp Leitner, Stefan Schulte, Schahram Dustdar "SPEEDL – A Declarative Event-Based Language for Cloud Scaling Definition", *The Future of Software Engineering FOR and IN Cloud* visionary track of *IEEE World Congress on Services*, pp. 71–78, 2015.
- Rostyslav Zabolotnyi, Philipp Leitner, Schahram Dustdar "Profiling-Based Task Scheduling for Factory-Worker Applications in Infrastructure-as-a-Service Clouds", *40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 119–126, 2014.
- Rostyslav Zabolotnyi, Philipp Leitner, Schahram Dustdar "Building Elastic Java Applications in the Cloud: A Middleware Framework" in *Handbook of Research on Architectural Trends in Service-Driven Computing*, Volume II, IGI Global, pp. 661–685, 2014.
- Philipp Leitner, Rostyslav Zabolotnyi, Alessio Gambi, Schahram Dustdar "A Framework and Middleware for Application-Level Cloud Bursting on Top of Infrastructure-as-a-Service Clouds", *6th IEEE/ACM Utility and Cloud Computing Conference (UCC)*, pp. 163–170, 2013.
- Rostyslav Zabolotnyi, Philipp Leitner, Schahram Dustdar "Dynamic Program Code Distribution in Infrastructure-as-a-Service Clouds", *5th International Workshop on Principles of Engineering Service-Oriented Systems (PESOS)*, co-located with *International Conference on Software Engineering (ICSE)*, pp. 29–36, 2013.

CHAPTER 1

Introduction

In recent years, the cloud computing paradigm [1] has provoked a significant push towards more flexible provisioning of IT resources, including computing power, storage and networking capabilities. Cloud computing simplifies the implementation of innovative ideas for small companies or individuals, and lowers production and maintenance costs for industrial applications [2].

Usage of provided hardware or platform simplifies company organization and allows avoiding private infrastructure management and hiring additional personnel that is not directly related to core business activities [3]. Applications designed with the cloud in mind (so-called *cloud-native applications*) allow developers elastically adapting to market changes and optimizing resource consumption.

Even huge companies that maintain their own data centers, benefit from cloud computing: it allows them performing rapidly small or temporal demonstration projects in situations when standard resource obtaining and usage reporting procedures may cause intolerable delays [3].

At the same time, not only software developing companies benefit from cloud computing. Cloud platform providers successfully compete against in-house infrastructure solutions by focusing resources on the single task of mass infrastructure resource provisioning, thus offering cheaper and better resource provisioning solutions controlled by professional staff with higher level of expertise.

Cloud computing usually implies pay-as-you-go pricing, which allows users of the cloud to view IT costs as expenses rather than investments [4]. This is closely related to the advantage most commonly associated with the cloud: IT costs can be kept low by reducing the upfront infrastructure investments close to zero, and paying only what is actually used [1]. The flexibility offered by the cloud is particularly interesting for start-ups. In the words of Amazon's Jinesh Varia: "*In the past, if you got famous and your systems or your infrastructure did not scale, you became a victim of your own success. Conversely, if you invested heavily and did not get famous, you became a victim of your failure*" [5]. Besides economic factors, the core driver behind this cloud computing hype

is the idea of elastic computing. Elastic applications are able to increase and decrease their resource usage based on the current application load, for instance by adding and removing computing nodes. Optimally, elastic applications are cost and energy efficient, while still providing the expected level of application performance [6].

Elastic applications are typically built using either the IaaS (Infrastructure as a Service) or the PaaS (Platform as a Service) paradigm [2]. In IaaS, users rent virtual machines from the cloud provider, and retain full control (e.g., administrator rights) over the virtual machines. In PaaS, the level of abstraction is higher, as the cloud provider is responsible for managing virtual resources, while users utilize provided APIs in order to focus on the business goal of a developed application.

1.1 Motivation

However, despite all benefits that come with the cloud, developers have to face new set of challenges that come with this modern technology. In order to use the cloud, programmers need to adapt existing applications to work in the new environment. While application adaptation allows running code in the cloud and thus decreases infrastructure costs, simple execution of legacy software in the cloud is not the ultimate goal of cloud application development. Instead, in order to fully benefit from the cloud, programmers need to redesign their applications to be elastic and dynamic by design [7], what often is a challenge on its own.

While developing cloud native applications, one has to decide which cloud paradigm (i.e., IaaS or PaaS) to follow. PaaS offers a higher level of abstraction, therefore allows for more efficient cloud application development, as less boilerplate code (e.g., for creating and destroying virtual machines, monitoring, load balancing or application code distribution) is required. However, practice has shown that today's PaaS offerings (e.g., Windows Azure, Google AppEngine, or Amazon's Elastic Beanstalk) come with significant disadvantages, which render this option infeasible for many developers.

These disadvantages include:

1. strong vendor lock-in [8], as one is typically required to program against a proprietary API;

2. limited control over the elastic behavior of the application (e.g., developers have very little influence on when to scale up and down);

3. no root access to the virtual servers running the actual application code, what complicates remote debugging, operating system tweaking, network traffic monitoring or software management on cloud hosts [9];

4. little support for building applications that do not follow the basic architectural patterns assumed by the PaaS offering [10] (i.e., usually Apache Tomcat based web applications are expected).

All in all, developers are often forced to fall back to IaaS for many use cases, despite the significant advantages that the PaaS model promises.

Building IaaS-based elastic cloud applications is not an easy task, and makes developers face an entire new range of challenges. While programmers get full software development freedom, they also become responsible for the complete application execution stack. This slows down application development and requires broader software development knowledge. Also programmers need to be ready to face the following challenges:

1. a significant amount of platform-dependent boilerplate code to control rented virtual machines;
2. home-made cloud host state monitoring and fault recovery systems are needed;
3. domestic scaling policy behavior and elasticity enforcement components are required;
4. the vendor lock-in caused by proprietary APIs that provides basic cloud management mechanisms.

These tasks are orthogonal to the applications' mission, but introduce significant complications and bury the applications' actual business logic deep under a mountain of platform-dependent code, what has to be developed over and over again for each new application or platform version.

1.2 Problem Statement

The challenges described above expose a significant variety of possible research questions and issues to solve. To narrow down the set of challenges we address and to define the scope of this thesis, we outline the core problem statement in this section.

The main decision that cloud application developers currently have to settle before designing the cloud application, is the cloud paradigm they want to target. Whenever a new cloud application is developed or some old solution needs to start running in the cloud, developers have to clearly understand whether they target IaaS or PaaS. The reason why this decision has to be done so early is that cloud applications developed for each cloud paradigm differ significantly in their architecture, capabilities and design.

The Existing separation into PaaS and IaaS paradigms is not a coincidence. Even though there are some drawbacks of each approach, both of them appeared as the result of a progressive evolution of the existing application and framework design paradigms. Each cloud paradigm attempts to provide a generic solution for a broad range of applications, while having different priorities and targeting a different niche of the cloud usage. IaaS provides full access to the virtual infrastructure, granting a control layer which is as thin as possible [11]. PaaS usually provides an environment for comfortable and flexible application development and management, concealing all infrastructure administration from the cloud developers [11].

While currently developers have to make a distinctive choice between the available cloud paradigms, in this thesis an alternative option is explored.

The main goal of this thesis is to *provide transparency and development convenience of cloud applications as in modern PaaS solutions, while maintaining the flexibility and complete control over the developed cloud application.*

The defined research statement means that the developed solution has to be capable of:

1. simplifying or even completely eliminating the need to define cloud-specific code which is not related to the business logic of the designed application, thus allowing developers to focus on the business logic of the developed product;

2. hiding platform-specific APIs behind the unified and developer-controlled code, thus avoiding a vendor lock-in;

3. allowing flexible application design and enforcing minimum code or architecture restrictions;

4. granting full access to the cloud solution code during development, testing and execution;

Such a solution would be attractive from multiple perspectives. On the one side, such approach should be easy to use and fast to start experiencing the benefits of the cloud computing as most of cloud-specific code would be handled by the platform. On the other side, developers should have the complete control over the application, used technologies and execution environment in case some advanced tweaking is necessary or some particular application component has to behave differently from the rest of the application.

1.3 Research Questions

The research performed within this thesis was mainly aligned along the following main research questions:

RQ 1: How can an application be transparently distributed over the cloud?

In the most general sense, an application becomes a cloud application only when it is distributed over a set of cloud hosts and its components are communicating with each other. Therefore, in order to function properly, cloud applications need to be aware of the cloud and be able to leverage dynamic cloud resources to solve a predefined task. However, this leads to the cloud-related issues discussed in Section 1.1 and the need to design applications accordingly to the IaaS or PaaS cloud paradigm. Therefore, the solution we target has to provide a way to distribute applications within the cloud as transparently as possible in order to decrease the amount of changes required to switch to another cloud platform or distribution approach. This can be achieved by the efficient and robust middleware that supports multiple cloud platforms and allows abstracting from

the underlying API. However, in order to avoid having just another lock in, discussed middleware has to be as lightweight and transparent as possible.

Some simple cases or partial solutions are already available in the field (see Chapter 3 for more details), while there is no general and elastic solution that would provide both, transparent code distribution and cloud elasticity.

RQ 2: Which instruments and capabilities allow efficient, flexible and elastic execution of transparent cloud applications?

Whenever a new cloud application is designed, developers have to focus heavily on mastering the tools that allow leveraging cloud resources efficiently. During the process of designing a transparent and efficient cloud platform, automatic application distribution and management tools are usually coming to mind. However, even though such tools should be a solid ground for the reasonable defaults, application developers usually deeply relay on the powerful and flexible gears of cloud control. Therefore, an application distribution solution should be able to provide an adaptable platform-independent way to state the rules that control application distribution and behavior in the cloud. These rules should be generic enough to cover most of the algorithms that are usually used in cloud computing, while they should also allow specifying sophisticated and domain-specific behaviors and actions in a developer-friendly manner.

Efficient resource usage in the cloud is usually understood as the minimal amount of hosts used by an application while satisfying performance requirements [12]. However, a problem of efficient task ordering and execution within a single cloud host is usually overlooked: As the resources of a single cloud machine are limited, only a limited amount of tasks may be assigned to run there at any particular point of time. Additionally, as the resource usage of each task fluctuates over execution time, resources for each task have to be allocated considering a maximal resource usage spike. This allows having functional application even in situations when resource usage spikes of all concurrent tasks overlap. This does not happen frequently. Therefore, such reserved resources usually remain unused. If the designed solution could provide a method to efficiently avoid these resource usage spike overlaps, the system could schedule more tasks to the same amount of hosts.

Even though the field of code distribution and resource management in the cloud is a hot research and industrial topic, the work there is mainly focused on the areas of load balancing [13] and auto scaling [14]. Such approaches usually cover simple or specific scenarios of cloud applications. Additionally, there was hardly any work on task organization within the cloud hosts. In our work we focused on providing a generic solution that tightly integrates with developed applications and simplifies cloud behavior specification for sophisticated and advanced use cases.

RQ 3: How to verify if the designed transparent application distribution approach is useful and fits developers' needs?

Whenever a new middleware, platform or framework is designed, its ideas and APIs usually are highly influenced by former experience of the architects [15]. However, even

if the architects are experienced cloud developers or researchers, their vision of the final solution might be completely different from the one expected by users. Due to this, the designed solution may be of no interest to practitioners, independently of the quality of ideas or benefits it provides. Similarly, even if the core idea is sound and finds positive feedback within the target audience, the designed solution might still fail to attract people due to some small misconceptions or design faults that were missed by the architects.

In order to surpass this issue, a transparent cloud application distribution solution has to be constantly validated by a reasonable set of future users. Evaluation has to be organized to validate every important aspect of user experience, as each of them is crucial for product success. Therefore, evaluation should determine not only if users were able to design an application, but should also measure their convenience, satisfaction, and interest in the proposed solution. Following this approach, designers can ensure that the result of their work is indeed interesting to the target audience and will solve the problems users are currently facing while using alternative approaches.

1.4 Scientific Contributions

In order to solve the research questions formulated in Section 1.3, the following contributions were carried out in the scope of this thesis.

Contribution 1: A middleware for dynamic transparent application distribution in the cloud. Targeting to provide a transparent application distribution solution, the conceptual middleware named JCloudScale, initially presented in [16], was extended with configuration, communication and discovery components in order to become a fully-functional and transparent platform. Similar to PaaS, JCloudScale takes over virtual machine management, application monitoring, load balancing, and code distribution. However, given that JCloudScale is a client-side solution instead of a complete hosting environment, programmers retain full control over the developed application. Furthermore, the selected application distribution method allows JCloudScale to support a wider range of applications than common PaaS platforms. JCloudScale applications run on top of any IaaS cloud, making JCloudScale a viable solution to implement applications for private or hybrid cloud settings [17]. In summary, we claim that the JCloudScale model is a promising compromise between IaaS and PaaS, combining many advantages of both worlds.

Details on JCloudScale middleware are discussed in Chapter 5. Contribution 1 has originally been presented in [16] and further extended in [18, 19].

Contribution 2: A transparent dynamic application code distribution framework. During the development and evaluation of JCloudScale, we faced the issue of code integrity and synchronization between cloud hosts. Whenever cloud software developers are making any changes in developed code, they have to ensure that old and new components are interface-compatible or tear down all running cloud instances and rebuild virtual machine images to include only updated code. While this may be a reasonable approach for slowly-changing projects with a few updates every week, it is unacceptable for fast-paced applications.

An alternative way to achieve program code distribution is to include facilities for dynamic code search and distribution in the underlying framework. Therefore, in order to solve the issue of frequent code updates, a solution for transparent program code distribution was developed. Developed approach transparently distributes program updates over a set of cloud hosts in order to ensure that a compatible version is used by every user and application component. Additionally, the developed code distribution solution encapsulates each component or client in a separate container, thus allowing multiple code versions of the same component running in parallel at the same cloud host. While the discussed code distribution solution was developed for JCloudScale, it is highly independent and can be used separately as well.

Details on discussed transparent code distribution approach are discussed in Section 5.2. Contribution 2 has originally been presented in [20].

Contribution 3: An application scaling and cloud management definition language. Cloud platform management and application distribution logic have the key role in cloud applications development process. In order to achieve elastic and efficient resource usage, developers need to describe application behavior in a developer-friendly and cloud platform-agnostic form. To simplify this task, the SPEEDL (Scaling Policy Extensible Event-based Declarative Language) language was designed. SPEEDL is a declarative and extensible domain-specific language [21] (DSL) that simplifies the creation of elastic, application-specific cloud scaling behavior on top of IaaS clouds. SPEEDL allows the definition of *scaling policies*, i.e., a set of event-condition-action (ECA) rules managing the amount and types of resources (e.g., VM instances) acquired from the cloud, as well as the mapping of incoming tasks to these resources for processing. Unlike existing industrial solutions, SPEEDL tightly integrates and communicates with the developed application. Additionally, SPEEDL is extensible and highly favors domain and application specific rules and behaviors.

Details on SPEEDL language are presented in Chapter 6. Contribution 3 has originally been presented in [22].

Contribution 4: Profiling-based task scheduling for factory-worker applications. Addressing the issue of effective cloud resource usage, a task scheduling approach for uniform tasks based on profiling data was designed. The developed scheduler aims to avoid overlapping peak resource usages of running tasks, hence allowing to execute more tasks in parallel on the same virtual machine. This is achieved with a task scheduler that monitors task resource consumption and constantly improves future resource usage estimations for each used host. These predictions allow for the effective scheduling of subsequent tasks and forecasting when an application should scale up or down, thus improving elastic system behavior in the cloud and optimizing resource usage. Designed solution proved to be efficient and effective on factory-worker applications (also known as the producer-consumer pattern, and strongly related though not identical to the master-slave pattern [23]), but the approach may be partially useful for other applications as well.

Details on profiling-based task scheduling are presented in Chapter 7. Contribution 4 has originally been presented in [24].

Contribution 5: A qualitative and quantitative user study that validates the developed cloud application development middleware. In order to evaluate the developed solution for application distribution in the cloud, a user study was performed. In this user study, JCloudScale and its components were compared to both, existing IaaS (OpenStack and Amazon EC2) and PaaS (Amazon Elastic Beanstalk) solutions. The main goal of the user study was to validate the original claims and expectations as well as to evaluate development productivity and user acceptance.

The discussed user study, along with JCloudScale performance evaluation, is presented in Chapter 8. Contribution 5 has originally been presented in [19].

1.5 Structure of the Work

The rest of this thesis is organized as follows:

- Chapter 2 provides fundamental information about technologies and concepts used in the remainder of this thesis. Particularly, the concepts of cloud and elastic computing (Section 2.1), communication in the cloud (Section 2.2), event-driven architecture (Section 2.3), and aspect-oriented programming (Section 2.4) are described.
- Chapter 3 presents scientific and industrial aspects of work related to the contributions presented in the thesis.
- Chapter 4 discusses the case study of JavaScript testing cloud service JSTaaS ("JavaScript Testing-as-a-Service") that serves as the illustrative example for each contribution discussed in this thesis.
- Chapter 5 describes the fundamental contribution of the thesis, the JCloudScale middleware.
- Chapter 6 focuses on the most important aspect of JCloudScale middleware, discussing how cloud solutions can leverage JCloudScale in order to optimize application resource usage in the cloud.
- Chapter 7 dives into a particular problem of task scheduling within the cloud host, where the problem of efficient task execution within limited host resources is discussed and the designed solution is presented.
- Chapter 8 provides the evaluation of the designed transparent cloud application development middleware. Particularly, the presented work is evaluated using an extensive user study (Section 8.2.2) that validates user perception of the designed middleware, followed by the performance evaluation which validates performance and functionality of the presented solution.
- Chapter 9 concludes the thesis and presents a glance over the future research directions that emerged due to this work.

CHAPTER 2

Background

This chapter brings in some basic concepts that are crucial for understanding the remaining of the thesis. First of all, Section 2.1 presents the concept of cloud computing, which is the target environment and basement for every presented contribution of this thesis. The notion of elasticity, discussed in this section, is one of the essential characteristics of cloud computing [25, 26] and plays a vital role and main motivation for the scaling behavior contributions presented in this work. Following, Section 2.2 presents the concept of distributed system communication and discusses the available alternatives for cloud communication middleware. This section presents also the messaging and messaging-based architecture that was intentionally selected as the reliable and efficient communication infrastructure for the JCloudScale middleware. Afterwards, Section 2.3 describes the concept of monitoring events, event reaction and event-driven software architecture that allow building reactive and adaptive cloud applications. Finally, Section 2.4 presents the idea of aspect-oriented programming, the software development technique that allows application post-processing, permitting to achieve transparent cloud application distribution that is the foundation of this thesis.

2.1 Cloud Computing

Even though the concept of *cloud computing* can be traced back to 1995 [27], establishing and framing of cloud computing is still an ongoing process. The notion of cloud computing is fuzzy and spans from the computation distributed over Internet [28] through *utility computing* [2] up to Web 2.0 concepts [29].

2.1.1 Definition of Cloud Computing

The US National Institute of Standards and Technology (NIST) defines cloud computing as *"...a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications,*

and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [25].

This definition is quite general to include most of the concepts behind the cloud computing trend. Within this thesis, the concept of cloud computing will mainly follow the idea of utility computing available on demand [2]. Therefore, the following aspects of cloud computing, coined out in [2] will be most prominent and important in the following text:

1. *"The illusion of infinite computing resources available on demand."*
2. *"The elimination of an up-front commitment by cloud users."*
3. *"The ability to pay for use of computing resources on a short-term basis as needed."*

The promise of infinite resources is critical for cloud computing. In reality it is fulfilled by the enormous scale of cloud providers data centers that make almost any reservation of computing resources possible. The ability to obtain additional host at any point of application execution time shapes the behavior of cloud applications and defines the reactions on increasing demand. While in the past, applications had to know the upper limit of resources they could obtain and developers were programming distribution solutions knowing this limit in advance, with cloud computing, developers tend to develop infinitely scaling applications [30] that can scale from a few hosts up to hundreds and thousands.

The elimination of an up-front commitment allows cloud computing users to avoid thinking of the required amount of resources ahead of time. This approach opens possibilities and eliminates the risks for applications and start-ups when popularity or resource requirements are not known beforehand. While previously developers would need to risk by defining estimates in advance [5], cloud computing allows obtaining and releasing resources whenever it is required.

The ability to pay for computing use on a short-term basis brings in the *"pay-as-you-go"* concept [2]. Cloud applications are usually billed on a regular basis depending on their actual use. This allows avoiding sophisticated advance resource booking algorithms and brings in the monitoring and planning mechanisms that predict resource usage on a short notice. Additionally, this allows treating infrastructure as operational expenses rather than investments as it is used to be prior to cloud computing.

2.1.2 Elastic Computing

The key aspects of cloud computing, coined out above, fostered a new trend of *"elastic computing"* [31]. While the term *"elasticity"* is commonly referenced to economics [32] or physics [33], practitioners and scientists tend to notice a similar behavior in cloud computing applications [31]. In cloud computing, elasticity is usually defined as the degree to which a system autonomously adapts its capacity to workload over time [34]. However, this definition defines only the elasticity of used resources, while similar behavior

in cloud applications is also observed for operation costs or quality of service [26]. In the scope of this thesis, only the resource elasticity is usually in the focus.

2.1.3 Cloud Computing Service Models

Cloud computing resources may be provided under different terms and business models. Currently the most popular approach is to use either *"Public Cloud"* or *"Private Cloud"* [2]. The public cloud provisioning model assumes that computational resources are publicly available and shared equally among users following the "pay-as-you-go" model. Contrary, private cloud usually refers to internal data centers functioning similarly to public clouds, but not available to the public [2]. NIST defines [25] two additional cloud deployment models *"Community Cloud"*, that defines infrastructure that is shared within a specific community of customers, and *"Hybrid Cloud"*, which stands for computational resources that are implemented as composition of multiple cloud infrastructures [25].

Considering the flexibility and tooling, cloud solutions can be categorized into three distinct categories [25]:

- *Software as a Service (SaaS)* represents consumer applications executing on the cloud infrastructure. Consumers, contrary to traditional software usage model, do not control or own the software, but rather exploit it on demand, mainly on subscription basis. While the term itself appeared prior to cloud computing popularization [2], the concept of SaaS is perfectly aligned with the cloud computing paradigm and architecture.
- *Platform as a Service (PaaS)* defines the cloud-based middleware that allows customers to deploy onto the cloud infrastructure customer-created or acquired applications. While application developers do not control PaaS and underlying infrastructure, they get the comfortable and easy to use API that simplifies the creation of cloud applications [25].
- *Infrastructure as a Service (IaaS)* delimits the provisioning of processing, storage, networks or other fundamental computing resources where consumers are able to deploy their software or data [25]. Customers do not control or manage physical machines, while they are provided with API to obtain and release virtual resources on demand.

The presented cloud solution categories, often referenced as *"Service Models"* [25], are usually considered as the cloud software development stack. Figure 2.1 illustrates the relations of these layers and presents some popular examples of the software operating on each service model.

2.2 Cloud Computing Communication

The moment applications became distributed, the need for communication between components emerged. Over the past decades, a significant amount of distributed archi-

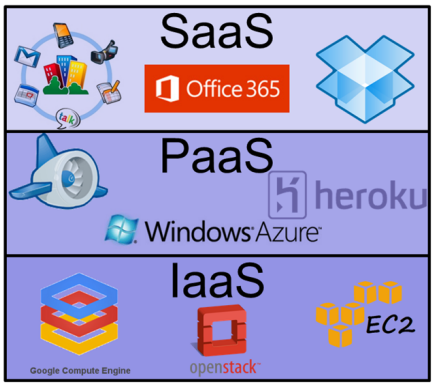


Figure 2.1: Cloud computing service models stack

tectures [35] and communication protocols appeared [36], but all of them can be placed between the two theoretical extremes: client-server and peer-to-peer architectures [37].

Distributed application development practices gradually evolved adapting to the changing communication environment, increasing computing performance and shifting software requirements. Initial distributed applications were usually executed in closed corporate or university environments and were highly heterogeneous [38]. The appearance of the Internet and worldwide communication brought in the need of standardization and unification of communication protocols. These trends finally resulted in standardization and Service Oriented Architecture (SOA), which is described in more details in [39]. However, resulting standard appeared to be too generic, heavy-weight and slowly adapting to the changing conditions of the market, what caused alternative concurrent wave of communication simplification to gain an increasing amount of recognition. The trend of communication simplification lead to the development of Representational State Transfer (REST) software architecture and RESTful web services definition [40].

2.2.1 Communication Middleware

The growing heterogeneity of the distributed computing environment, caused by the need to communicate with external services, resulted in exponential growth of communication complexity. This exponential growth, along with other requirements such as reliability, traceability, and distribution of communication, caused the appearance of the Enterprise Service Bus (ESB) [41] software architecture. This architecture brought in a central component, named ESB, that was responsible for communication routing, simplification and harmonization.

While conceptually similar, Message Oriented Middleware (MOM) and Enterprise Messaging System (EMS) [42] emerged out of different reasoning. The EMS architecture also introduces the middleware component that is responsible for communication, but the design focus is on messaging protocols instead of the service implementation, as in SOA. EMS focuses on guarantees of message delivery, secure communication, message routing, and pattern-based subscription. While EMS systems are also capable of handling

multiple communication protocols at the same time, this is not the main goal of such systems. Therefore, usually MOM is based on a single protocol that all components understand and are able to process.

MOM introduces a conceptual change into distributed system communication paradigm. While originally the communication was performed in a synchronous, "request-response" mode, MOM brought in the asynchronous message passing communication. Asynchronous communication significantly increases performance, as each component does not have to wait for a message to be sent, delivered and responded to. Instead, the communication tends to happen in a "fire and forget" mode, when the task is considered completed if the resulting message is scheduled for sending. Mainly this is possible due to delivery guarantees of MOM, as most of such systems ensure that once the message enters the middleware, it will be delivered to the endpoint, except of configured dead message handling situations. Such approach allows building loosely coupled systems that are not taking responsibility for communication issues at all. Additionally, this simplifies creation of the data stream processing applications [43], where application components are built in a chain or a tree around incoming data flows.

However, not every communication can be performed asynchronously. Whenever EMS applications require to have a synchronous communication between components, they have to imitate synchronous communication with the available means of MOM. Even though most of existing MOM implementations have specific APIs that simplify simulation of synchronous calls, this may bring in additional performance overhead and communication delay in comparison to direct communication methods.

2.2.2 Communication in the Cloud

Building applications in the cloud brings in a different set of requirements and limitations to communication protocols and middleware. However, often these requirements are conflicting and require developers to make some compromise between them. For instance, the requirement to build an infinitely scaling application requires to avoid any single central components as they may lead to bottlenecks. However, the need to build fail-safe application in unreliable environments requires to have replicated components and some monitoring and routing service that is responsible for message guidance and transmission.

Depending on cloud application specifics and architecture, developers can select peer-to-peer communication with RESTful services, classic SOA design or message-based communicating components. Each of the architectures has its advantages and disadvantages, and none of them fits every cloud application. Modern cloud applications tend to fall into two categories. If reliability is the key feature of the application, developers have to design some central, often duplicated, component that ensures reliable execution. Such applications usually develop around messaging middleware that provides delivery and execution guarantees. Whenever the application key requirement is rapid and unlimited scaling, developers should consider peer-to-peer architectures, that are nowadays in the cloud tightly relevant to the idea of *"micro-services"* [44].

The JCloudScale middleware, discussed in this thesis, also required to decide which communication architecture will it follow. While initially the idea was to select

a micro-services architecture using RESTful web services for communication [16], after an extensive period of usage in the actual cloud environment, the decision was made to follow a more reliable, while less scaling approach. This allowed improving reliability of the distributed system on a relatively small cost of performance overhead. However, in future releases of JCloudScale, this may be configurable to allow developers deciding what is more important to them.

2.3 Event-Driven Architecture

In order to define truly adaptive behavior for distributed application under dynamic load, developers have to take into account the concept of application state monitoring. However, it is insufficient to provide a method for host or component state retrieval that will be periodically queried by some specific component. Such *"active monitoring"* approach increases the load on each component of the distributed system, raises the amount of traffic circulating in the network and does not assist application scalability [45]. Instead, distributed applications are usually designed using a *"passive monitoring"* approach [45], where each component periodically reports state information to some predefined channel, usually named *"stream"*. Neither the active nor the passive monitoring approach forbids additional monitoring configuration or adaptation. Both approaches support configuring the notification frequency or the amount of provided information. However, applications implemented with passive monitoring should be aware that the monitoring information stream is rather reactive, thus requested changes will not result in immediate change of the monitored components behavior.

Each monitoring notification from the remote component is usually considered as an *"event"*. An event is a thing that happens, especially one of importance [46]. Within this work, it will be assumed that event is an object that encodes some important situation, action or state change of an application. On each incoming event, the monitoring component that processes events from an event stream, adapts information about the current state of an application. This information is used to adapt the future application execution or resource usage.

Usually information from a single monitoring event should not influence application execution much. A single event is vulnerable to monitoring or application execution anomalies, thus a set of sequential events has to be aggregated. An *"event processing"* [47] mechanism, using statistical functions and algorithms, generates *"complex events"* that contain much more valuable information than in a single event. Even a simple averaging of the subsequent events values significantly increases usefulness of performance measurements in comparison to the data from a single event.

Additionally, complex event processing allows the creation of other events that are more important to an application. Such *"application-specific events"* [48] represent a high-level application behavior or a state that is hard to observe using simple events obtained from the monitored components. For example, traffic monitoring application can be interested in "accident" complex events, which are produced when the amount of simple events about cars moving slowly increases at some location.

The idea of events and event processing significantly influences an application architecture. In order to experience the full power of event processing, cloud application developers have to design their solutions or monitoring components around the streams of events that flow through an application, as it is shown in Figure 2.2. Such application architecture, usually named *"event-driven architecture"* [49] significantly differs from the classic 3-tier architecture [50] as the analyzed data is dynamic and constantly changing, what requires different information processing apparatus.

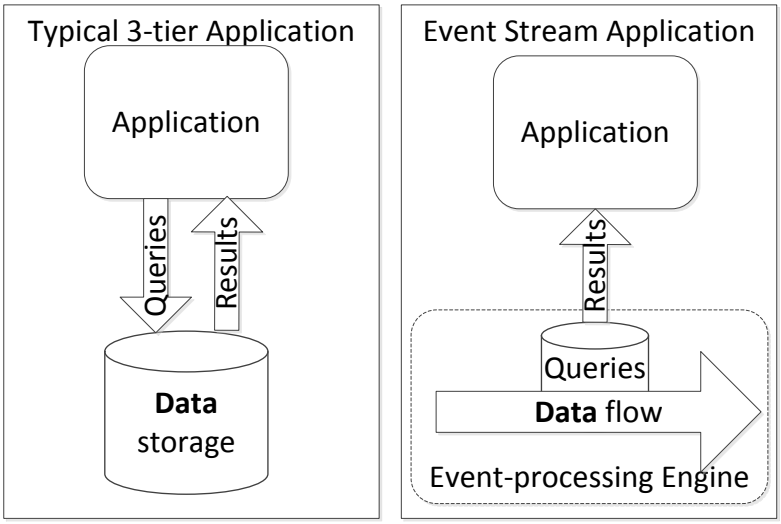


Figure 2.2: Comparison of event stream applications to typical 3-tier applications

Addressing the issue of state monitoring and effective adaptation of cloud applications to the load and resource requirements, the solutions presented in this thesis follow the ideas of complex event processing and event-driven application behavior.

2.4 Aspect-Oriented Programming

Software development is a complex process during which programmers have to keep in mind a significant amount of loosely coupled objects and relations between them. Even though during the whole history of software development programmers and theoreticians favored decoupling and modularization [51], in reality this is not always achievable. Some loosely coupled processes such as payment and product shipment are easy to separate. However, other activities are so much entangled together that it is impossible to split them neither from the programming nor from the architectural point of view.

Consider the example of an abstract method implementation, shown in Figure 2.3. In addition to the business logic, which is the reason of the method creation, a programmer has to perform a significant amount of other activities that are not directly related to the business logic that needs to be coded. For example, a developer has to check input parameters, handle execution errors and log method invocation. While each of

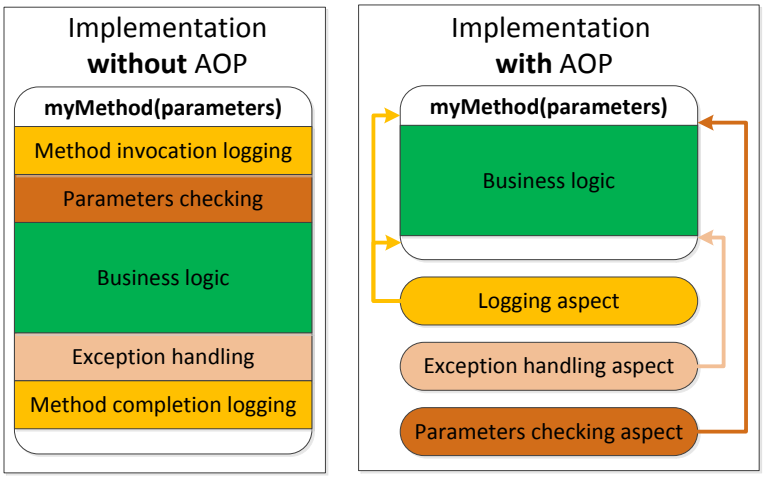


Figure 2.3: Structure of the method implemented using usual and aspect oriented programming techniques.

these activities is usually just a few lines of relatively simple code, together they form a significant portion of the method body. Even though this code is simple and easy to understand, it hides the business logic of the method, what may significantly slow down method understanding and modification by another programmer.

While the cost of all these side activities may be bearable for a single method, on the scale of an application this accumulates to a significant amount of code that dramatically influences code readability and refactoring. However, programmers can not simply discard all this additional code. While it usually does not solve the main goal of an application, it needs to be in place to simplify application debugging, reaction to extraordinary situations, execution control or error handling.

In order to solve this problem, the concept of *"Aspect Oriented Programming"* (AOP) was introduced. This patented [52] programming paradigm increases modularity in situations when multiple sideways activities are mixing into the main application execution flow. Such activities (in terms of AOP, – *"cross-cutting concerns"*) are extracted into a separate entity named *"aspect"* (see Figure 2.3). Aspects contain only the functionality related to the described concern (e.g., "logging", "parameter checking", etc.) and are usually represented in a form of a common entity for the used programming language (in case of Java it is usually a separate class). In addition, aspects are associated with *"pointcuts"*: the statements in a special domain-specific language that declare the places in an application where the particular part of an aspect (i.e. *"advice"*) has to be injected. For example, such pointcut may declare that at the beginning of every method within some specific package or class `startLogging` advice from `Logging` aspect has to be injected.

AOP does not change the source code of an application, thus a developer is not distracted from the business logic implementation. Instead, AOP platforms provide the runtime application modification or the post-compilation step that applies all necessary

aspects at the predefined places.

From the one side, such approach cleans up the source code of an application as it allows extracting repetitive and distractive code from the main flow of the application execution. However, it also makes an application harder to understand for developers that are unfamiliar with AOP or work with an Integrated Development Environment (IDE) that has no AOP assistance. Therefore, AOP is another technology that provides an interesting solution to the existing and important problem of code decoupling and separation of concerns, but brings along a new set of challenges and issues that developers have to be aware of before starting using AOP in any application.

Addressing the issue of transparent application execution in the cloud, concepts of AOP appeal to be an interesting starting point. However, with the time it became clear that AOP is not a solution, but just an instrument that still requires significant amount of work in order to obtain a holistic solution to the problem of the transparent code distribution. Solutions presented in this thesis form around the JCloudScale middleware, which uses AOP paradigm to distribute applications and inject cloud management code.

CHAPTER 3

Related Work

In the following chapter, a survey of the related research work is presented. Primarily the focus is on the scientific and industrial work that is related to the JCloudScale middleware in general. Secondly, the current research trends that are related to the important parts of the thesis are discussed as well.

3.1 Related Work on Transparent Distribution Frameworks

There are essentially two schools of thought of how one should build a distributed system. On the one hand, many approaches aim at hiding the complexity of distribution behind convenient abstractions, such as remote procedure call systems. On the other hand, some claim that such abstractions always have to be leaky, and, hence, should be avoided altogether (for just one example of this argument, see [73]). JCloudScale follows the former school of thought. Essentially, JCloudScale provides an abstraction that makes elastic applications running on top of an IaaS cloud seem like regular, non-distributed Java applications. Hence, JCloudScale implements the ideal for building elastic applications mentioned in [74], a "single shared global memory space of mostly unbounded capacity". In doing so, JCloudScale is complementary to a number of related commercial platforms and research works.

A comprehensive starting point for research work on elastic cloud application development is provided by the survey in [75]. According to the taxonomy used in this paper, our work clearly falls into the category of container-level scalability in the platform layer (the container being JCloudScale in this case). The group of container-based scalability systems mainly consists of PaaS solutions, that typically provide an environment or a sandbox for customer code execution.

The main disadvantage of all such systems is that they imply a significant loss of control for the developer. They typically require the usage of a given public cloud (typically

Table 3.1: High-level comparison of distributed and cloud computing systems.

	Transparent Remoting	**Transparent Elasticity**	**Local Testing**	**Unrestricted Architecture**	**Transparent VM Management**	**Cloud Provider Independence**
Remoting Frameworks						
Java RMI	yes	no	yes	yes	no	no
Enterprise Java Beans (EJB)	yes	partial	yes	yes	no	no
Elastic Remote Methods [10]	yes	yes	no	yes	yes	yes
Aneka [53] [54]	partial	no	no	yes	yes	yes
A^2-VM [55]	partial	yes	no	yes	yes	yes
PaaS Systems						
Google AppEngine [56]	yes	yes	partial	no	yes	no
Amazon Elastic Beanstalk [57]	yes	yes	no	no	yes	no
Heroku [58]	yes	yes	partial	no	yes	no
AppScale [59] [60]	yes	yes	no	no	yes	yes
ConPaaS [61] [62]	yes	yes	no	partial	yes	yes
BOOM [63]	yes	yes	no	no	no	yes
Google Cloud Dataflow [64]	yes	yes	no	no	no	no
Esc [65]	yes	yes	no	no	no	yes
Granules [66]	yes	yes	yes	no	no	yes
ElasticThrift [67]	yes	partial	no	no	yes	yes
ElasticJava [67]	yes	partial	no	yes	yes	yes
Cloud Deployment & Test Frameworks						
JClouds [68]	no	no	no	yes	no	yes
Docker [69]	no	no	yes	yes	no	yes
Cafe [70]	no	no	yes	no	no	yes
MADCAT [71]	no	no	yes	no	no	yes
OpenTOSCA [72]	no	partial	no	yes	yes	yes
JCloudScale	yes	partial	yes	yes	yes	yes

provided by the same vendor), imply the usage of proprietary APIs, and restrict the types of applications that are supported (typically, these platforms support only Tomcat-based Online Transaction Processing (OLTP) style systems). JCloudScale, on the other hand, allows application developers to retain full control over their application and influence application behavior on any level starting from cloud operating system configuration. JCloudScale applications are not bound to any specific cloud provider, are easy to migrate, work well in the context of private or hybrid clouds, and support a wider variety of applications, while still providing an abstraction comparable to commercial PaaS solutions.

As the scope of JCloudScale is rather wide, there is a large number of systems that are partially related to JCloudScale. The main dimensions used to compare all mentioned below frameworks are:

1. to what extent they transparently handle remoting and elasticity;
2. whether they handle scaling up and down;
3. how easy it is to locally test and debug applications;
4. whether the framework restricts types and architectures of applications that can be built;
5. whether the framework handles and manages cloud virtual machines;
6. whether the framework is bound to one specific cloud provider.

A high-level comparison of the various frameworks along these dimensions is demonstrated in Table 3.1. All systems are evaluated along discussed dimensions, and assigned *"yes"* (strong support), *"no"* (no real support), or *"partial"* (some support). All evaluations are based on tool documentations or publications.

Firstly, JCloudScale can be compared to traditional distributed object middlewares [76], such as Java RMI or EJB. These systems provide transparent remoting features, but clearly do not provide any support for cloud specifics, such as VM management. It can be argued that EJB provides some amount of transparent elasticity, as EJB containers can be clustered. However, it is not easy to scale such clusters up and down. A recent work [10] has introduced the idea of Elastic Remote Methods, which extends Java RMI with cloud-specific features. This is comparable in goals to our contribution. However, the technical approach is quite different. Aneka [53, 54], a well-known .NET based cloud framework, is a special case of a cloud computing middleware that also exhibits a number of characteristics of a PaaS system. It is arguable if Aneka's abstraction of remoting is perfect, as developers still need to be rather intimately aware of the underlying distributed processing. To the best of our knowledge, Aneka does not automatically scale systems, and provides no local testing environment. [55] also presented an approach that can be considered related to JCloudScale. Their A^2-VM framework schedules Java threads over a compute cluster.

Secondly, many of JCloudScale's features are comparable to common PaaS systems (e.g., Google AppEngine, Amazon Elastic Beanstalk, or Heroku). All of these provide transparent remoting and elasticity, and take over virtual machine management for the user. However, they usually tie the user tightly to one specific cloud provider. Support for local testing is limited, although most providers nowadays have at least some tooling or emulators available for download.

In addition to these commercial PaaS systems, there are also multiple platforms coming out of a research setting. For instance, AppScale [59, 60] is an open-source implementation of the Google AppEngine model. AppScale can also be deployed on any IaaS system, making it much more vendor-independent than other PaaS platforms. This is similar to the ConPaaS open source platform [61, 62], which originates from a European research project of the same name. ConPaaS follows a more service-oriented style, treating applications as collections of loosely-coupled services. The recent IBM concepts named ElasticJava and ElasticThrift [67], that originate from Elastic Remote Methods [10], are more similar to JCloudScale as the stated goal is also to achieve transparent application distribution. However, instead of transparent application adaptation, authors propose Java language extensions and alternative Java compiler that are supposed to bring in transparent application distribution. Due to this, the approach advocated in this thesis is preferable, as developers do not need to learn new programming concepts and redesign their applications following the new syntaxis.

In scientific literature, there are also a number of PaaS systems which are more geared towards data processing, e.g., BOOM [63], Esc [65], or Granules [66]. These systems are hard to compare with JCloudScale, as they generally operate in an entirely different fashion as compared to JCloudScale or the commercial PaaS operators. However, they typically only support a very restricted type of (data-driven) application model, and often do not actually interact with the cloud by themselves. This makes them cloud provider independent, but also means that developers need to implement the actual elasticity-related features themselves.

A particularly interesting case of data processing solutions is the recently announced Google Cloud Dataflow platform [64]. This Big Data processing cloud platform was presented as a further development of the map/reduce paradigm [77]. This platform presents a convenient tooling to operate with the big amounts of data with the paradigm of pipes and filters and is based on previously announced Google internal technologies Flume and MillWheel [77].

Thirdly, there is a need to compare JCloudScale to a number of cloud computing related frameworks, which cover a part of the functionality provided by JCloudScale middleware. JClouds [1] is a Java library that abstracts from the heterogeneous APIs of different IaaS providers, and allows decoupling Java applications from the IaaS system that they operate in. JCloudScale internally uses JClouds to interact with the multiple cloud providers. However, by itself, JClouds does not provide any actual elasticity. Docker [2] is a container framework geared towards bringing testability and portability

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

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

to cloud computing. Essentially, Docker has similar goals to the local test environment of JCloudScale and may be used as testing or portable cloud environment in the future. However, Docker should be treated more as a tool that is highly useful in cloud environment rather than as a complete cloud middleware.

JCloudScale also has some relation to the various cloud deployment models and systems that have recently been proposed in literature, e.g., Cafe [70], MADCAT [71], or OpenTOSCA [72], which is an open source implementation of an OASIS standard. These systems do not typically cover elasticity by themselves (although TOSCA has partial support for auto-scaling groups), but they are usually independent of any concrete cloud provider.

By design, JCloudScale supports most of the characteristics we discuss here. However, especially in comparison to PaaS systems, developers of JCloudScale applications are not entirely shielded from issues of scalability.

Architecture restrictions of different solutions require more discussion. Industrial PaaS systems (e.g., AppEngine, Beanstalk, Heroku) are generally geared towards a very specific type of application (transaction-based Web applications). These systems assume that requests are (to a large extent) independent, and take very little time to process (AppEngine, for instance, has a hard upper limit of 30 seconds of processing time per request). This model is useful for many typical use cases in a Web context, e.g., blogs or Web shops. However, developers aiming to build other kinds of applications (e.g., the JSTaaS example discussed in Chapter 4, banking solutions, video streaming platforms, etc.) have to switch to IaaS or struggle with the architectural and technical restrictions imposed by those PaaS systems. Other remoting and cloud frameworks (e.g., Java RMI or Docker) do not have such restrictions (e.g., Docker is useful for more or less arbitrary applications), however, these systems are also not concerned about providing automated scaling and elasticity. JCloudScale, as well as the related ANEKA framework [53] and the Elastic Remote Methods proposed by [10], strive for a middle ground. They do not inherently assume a specific, narrow type of application, and can in principle be used to implement a wide range of elastic applications. However they are mostly suitable for applications with durable request- or task-processing activities, such as, for instance, video-audio encoding, web-crawling, sentiment analysis, or image rendering.

Additionally, JCloudScale provides significant benefits for applications that use cloud resources only to cover activity bursts [78]. JCloudScale is less suitable for connection-oriented and latency-sensitive applications, such as streaming services or online games. Further, for big data centric applications, JCloudScale is arguably less intuitive to use than state-of-the-art models (e.g., Hadoop or Spark SQL).

3.2 Related Work on Transparent Code Distribution

The problem of code distribution, discussed in Section 5.2, is not a novel topic [79]. The trivial solution would be to update code manually prior to execution. This solution is good enough for systems that update rarely, or in situations when the network speed is insufficient for code transmission or version verification at runtime. However, with further

development of networking and network-aware applications, automated code updating has become common. Nowadays, applications often check for updates periodically or at startup, and download updated code versions when necessary. This approach is suitable and becomes a standard for common user-oriented applications, but in other cases more sophisticated methods are needed.

One scientific area that inherently faces the problem of code distribution is grid computing [80, 81]. Software development in grid is usually focused on parallelization and computation-intensive execution [82]. Therefore, it is applicable to distribute code to the appropriate grid nodes prior to execution either manually or automatically. Some approaches to distribute program code and additional data on-a-fly were proposed in [83]. Still, code distribution in grid computing is different from the taken approach in presented work. In grid computing, developers solve the problem of initial long-running code distribution or "hot patching" [84]. Instead, discussed approach focuses on running different code versions in parallel. This is important for development, testing and multi-tenancy scenarios [85].

The transparent code distribution approach presented in Section 5.2 is more similar to the idea of mobile agents in agent-based computing. With this paradigm, applications are able to migrate from one computer to another autonomously and continue their execution on the destination computer [86]. Code distribution is a vital concept for such applications and a lot of research has been conducted to achieve different goals and improve code migration [87, 88, 89, 90, 91]. However, in contrast to JCloudScale, mobile agents are active and choose themselves where and if to migrate between computers at any time during their execution [92].

In JCloudScale, the application is distributed transparently and is not aware that the code is being distributed. From this point of view, the presented approach is more similar to the idea of remote code evaluation [93], when a task is transmitted to the server to execute. Also, transparent code distribution in the frames of JCloudScale exhibits some features of the code on demand approach [89], when missing code and related files can be fetched from the remote location on demand.

Finally, it should be noted that the code distribution solution, presented within this thesis, falls into the larger class of weak code mobility [90], as both code and data is transmitted, but not the application state.

3.3 Related Work on Scaling Behavior Definition

The problem of task scheduling did not originate in the cloud computing area. Clearly, the workload distribution challenge is present in any distributed or parallel system [94]. With the appearance and maturing of common scheduling algorithms [95], DSLs for scheduling and distributed systems started to appear. Nowadays task scheduling, often in a form of DSLs, is researched for instance within the fields of high performance computing [96] and embedded systems [97].

In the area of cloud computing, scheduling is usually performed in a form of balancing [95] or greedy [98] workload distribution in order to parallelize execution or optimize

resource usage. Such approaches satisfy data processing or classic three-tier [99] cloud applications, and usually do not require complex DSLs or special scheduling frameworks. However, when task distribution needs to address such dynamic or domain-specific features as data locality [100] or system heterogeneity [101], the necessity of an additional layer of abstraction becomes more plausible. The work presented in Section 6.3 does not focus on advances in cloud task scheduling. Instead, a holistic approach is developed that contains a significant amount of common algorithms and allows developers to address their workload management needs as easy and clear as possible.

Resource management in general, and the elasticity concept particularly play a vital role in cloud computing. Mainly, research is focusing on SLA-conformance [102], cost optimization [103] and "green" computing [98]. However, there are multiple DSLs and frameworks that are facilitating the problem of resource management by providing a user-friendly API and a predefined set of behaviors [104]. Nevertheless, these DSLs and frameworks are either completely outside of the developed application and provide some uniform means of resource management like TOSCA [105], or have a limited set of access APIs from within the developed application that allow passing information to some external decision module [104]. Instead, scaling solution presented within this thesis focuses on providing a tightly-integrated, extensible, cloud management framework that is running within the developed application and does not require any standalone components.

The major difference between the discussed solutions and scaling definition approach presented in Section 6.3 is that we are aiming at providing a cloud management component that (1) does not enforce any specific application design or architecture, (2) allows developer-friendly configuration and extension using the same language as the developed application, and (3) performs all actions within the developed application, allowing full usage of application-specific knowledge.

3.4 Related Work on Profile-Based Task Scheduling

Effective scheduling in cloud computing influences dramatically the overall performance of an application as well as resource usage [106]. Smart resource usage is clearly important since the pay-as-you-go model is an intrinsic feature of cloud computing [1]. However, resource usage optimization research in the area of cloud scheduling is mainly focused on QoS [107, 108] cost-awareness [102, 109, 110] or SLA-conformance [111, 112].

Research works about QoS-based task scheduling usually focus on user perception of the service execution and performance [113]. QoS became an important characteristic with the invention and popularization of telephony [114] where it was used to define the perceived quality of communication between users. In cloud computing area, research that is focusing on QoS usually discusses service composition [115], workflow execution [107], and IaaS or PaaS application organization [116]. Similar areas of research are explored within the SLA-conformance [111] area [117, 118, 119]. The main difference between QoS and SLA-oriented research is in perspective: QoS research focuses on user perception, while the research targeting SLA conformance assumes to have a formal agreement

between a service provider and consumers that can be formally verified [120]. SLA-based formal agreement is not contradicting or excluding QoS analysis. Instead, it is often considered as an extension or a formal method to ensure the provided service qualities expected by a user [121, 122].

The issue of cost-aware application execution is more specific to the cloud environment as it exploits the idea of dynamic cloud pricing [123]. Despite of that, the scope of topics explored in cost-aware cloud computing is similar to the ones of QoS and SLA [124, 125, 126].

Relatively new, but nevertheless a quite prominent topic in cloud computing is energy-efficiency or "green" workload scheduling [127, 128], that focuses more on effective power usage and operation in data centers or application adaptation in order to require a minimal amount of resources.

Another scheduling approach that focuses more on reliability and guaranteed task completion in faulty environments is redundant scheduling [129, 130] approach. Research in this direction targets the issue of fault detection, prevention, and recovery, while in Chapter 7 the main issue is the effective usage of the particular host resources.

From the perspective of profiling, in the cloud computing domain it is common to use it in order to develop elastic applications [131, 132] that adapt themselves to varying loads. This is usually treated in a broader way than the approach presented in Chapter 7: usually profiling is used to determine under-utilized or over-utilized machines, in order to balance existing load or manage the amount of used resources.

Each of the discussed approaches operates on higher levels of business requirements, resource usage and service quality, rather than the approach presented in Chapter 7. While most of the research mentioned here abstracts from the actual historical resource usage measurements, presented approach manages actual and predicted task resource usage values in order to achieve smooth and controlled resource usage curve on the level of a single host. This approach allows lowering resource requirements for planned jobs and allows executing more tasks in parallel on the same amount of cloud hosts.

CHAPTER 4

Case Study

Every contribution of this thesis is illustrated using the JSTaaS ("JavaScript Testing-as-a-Service") case study. JSTaaS is a JavaScript application testing cloud service, inspired by the real-life service provided by the New York based startup Codeship[1]. Even though the Codeship company and the service they provide is real, the details discussed below are not related to the real architecture or implementation of the Codeship product.

The role of JavaScript is rapidly increasing in the modern software development world [133]. JavaScript is a dynamic weakly-typed scripting language [133], therefore the main way of ensuring that particular code is correct, is to invest more efforts in code testing. However, in order to continuously test significant amounts of code, developers have to spend a substantial amount of money on computing infrastructure and personnel designated only for this purpose. Due to this situation, an imaginary team of developers decided to provide a standard, scalable and convenient infrastructure for JavaScript code testing.

Clients interested in automated code testing have to register their code repositories with the unit and integration tests in JSTaaS. Afterward, provided test suites are launched periodically following customer requirements. Test execution reports, accompanied by billing information, are sent back to the code owners.

Such business model allows companies to have automated continuous testing with minimum efforts required from their side. A very rough illustration of JSTaaS service is given in Figure 4.1.

As the developers had limited initial funding and unclear perspectives, it has been decided that the preliminary version of JSTaaS will be deployed in a small private cloud, sharing available infrastructure with other cloud applications. However, JSTaaS appeared to be a success, and soon grew out of available resources in this private data center.

[1] `https://codeship.com`

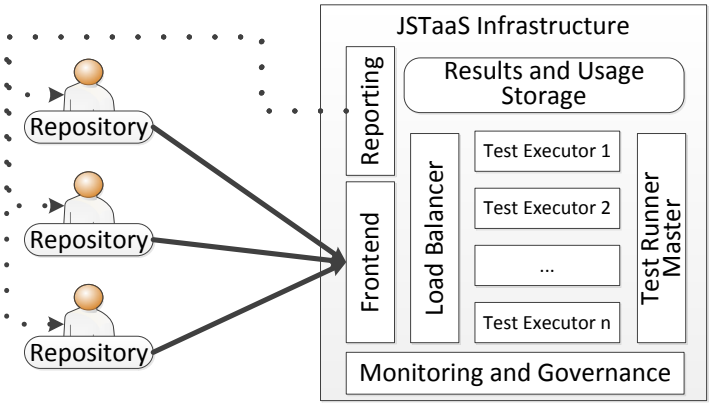


Figure 4.1: JSTaaS usage model and behavior

In order to facilitate further service enlargement, developers decided to incorporate resources of the Amazon EC2 public cloud, so that infrastructure costs will only grow in line with the actual demand. Furthermore, to save costs, local and remote virtual machines, which will be used to launch actual tests, should be utilized to the highest degree possible. This means that tests should be co-located on the same virtual machines as far as possible, and idle machines should be released if they are not required any longer. To this end, the core of JSTaaS needs to continuously monitor the utilization of all hosts, as well as the execution time of tests, in order to decide which hosts to keep online and which to tear down.

In order to address these ideas, the JSTaaS service infrastructure had to continuously extend over the available private and public cloud infrastructure, as it is shown in Figure 4.2. However, infrastructure management and monitoring APIs are completely different for each environment, what requires from JSTaaS developers to rewrite a significant amount of infrastructure management code in order to encapsulate and handle this difference.

Using standard tools, this application is not trivial to implement. Developers need to split the application into task manager, load balancer and workers to execute the tests, setup virtual machines, install the respective application components on these virtual machines and, at runtime, monitor to make sure that the application is not over- or under-provisioned. Tools such as AWS Elastic Beanstalk (deployment) or CloudWatch (monitoring) can be used to ease these tasks to some extent. However, such tools are available only for the respective provider's environment (i.e., Amazon) and handcrafted aggregating infrastructure and applications to merge data from multiple platforms are still required. Additionally, developers have to solve the issues related to ongoing JSTaaS development. Previously they used the same infrastructure for production and development of JSTaaS, but now they need to develop an algorithm to define how their changes will propagate to the public cloud. Similarly, if some client relays on the previous version of JSTaaS, developers have to ensure that the code of such client is executed within the right version of JSTaaS.

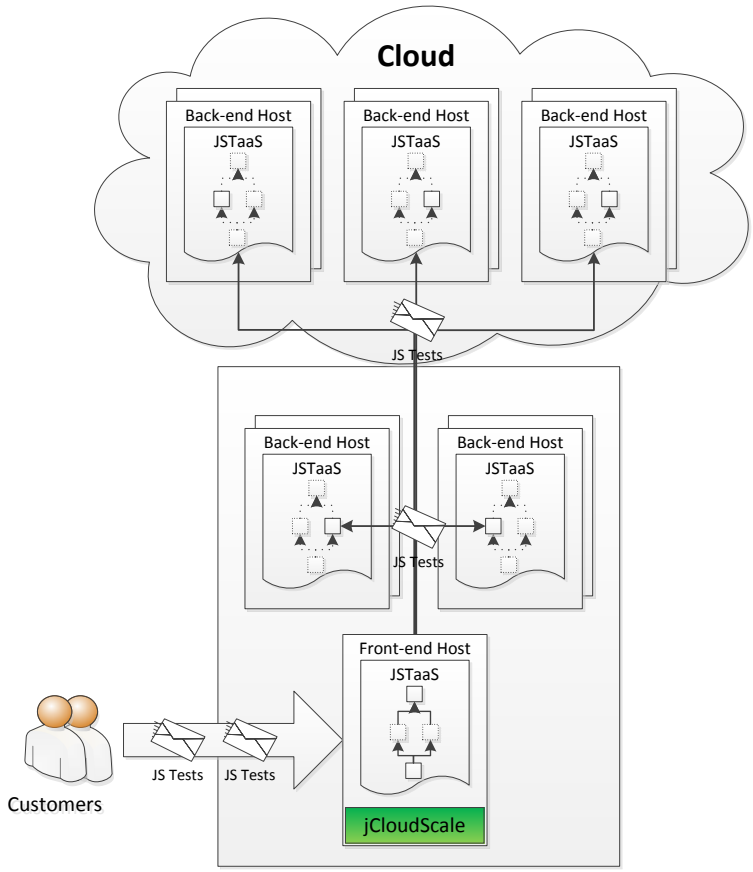


Figure 4.2: Extended model of JSTaaS service infrastructure

Hence, the developers decide to use the JCloudScale middleware to build JSTaaS. The team hopes that JCloudScale will help its few developers to efficiently write this elastic application, quickly master unfamiliar cloud APIs, and devise good scaling and scheduling policies. Furthermore, the integrated event-based monitoring of JCloudScale will also be useful for customer billing. Additionally, building their application on top of the JCloudScale abstraction, allows the team to easily migrate or extend their application, e.g., to an OpenStack-based private cloud, should they decide to move away from Amazon EC2 in the future. Finally, JCloudScale requires minimum changes to the existing solution and much less efforts from developers to have a first functional prototype than alternatives. This also allows JSTaaS developers to try out the distribution approach provided by JCloudScale really fast and easily fall back to any alternative solution in case they are not satisfied with the result.

In the remainder of this thesis, it will be shown:

1. How an application can be migrated or designed using JCloudScale middleware;

2. How code changes can be propagated over the infrastructure;
3. How JCloudScale allows defining application elasticity;
4. How unit tests can be efficiently scheduled over the booked resources;

CHAPTER 5

The JCloudScale Middleware

In this chapter, the basis for all contributions of this thesis is presented. While a basic version of the JCloudScale middleware itself was initially introduced before [16], this chapter discusses the main concepts and focuses on the work done within this thesis. The descriptions and code listings in this and following chapters use the case study discussed in Chapter 4 for practically explaining ideas. Structurally, this chapter starts with the basic concepts and notions, introducing more advanced features and internal details as they become related.

5.1 Basic Notions

JCloudScale is a Java-based middleware for building elastic IaaS applications. The ultimate aim of JCloudScale is to facilitate developers to implement cloud applications (in the following referred to as *target applications*) as local, multi-threaded applications, without even being aware of the actual cloud deployment. That is, the target application is not aware of the underlying physical distribution, and does not need to care about technicalities of elasticity, such as program code distribution, virtual machine instantiation and destruction, performance monitoring, and load balancing. This is achieved with a declarative programming model (implemented via Java annotations) and with AOP techniques, provided by AspectJ [1] framework that allows injecting required bytecode modifications into the target application. To a developer, JCloudScale appears as an additional library (e.g., a set of jar-files in classpath or Maven[2] dependency) plus a post-compilation build step. This puts JCloudScale in stark contrast to most industrial PaaS solutions, which require applications to be built specifically for these platforms. Such PaaS applications are usually not executable outside of the targeted PaaS environment. Contrary, JCloudScale encourages developers to build applications

[1] https://eclipse.org/aspectj/
[2] https://maven.apache.org/

that not only execute, but also achieve a stated goal with or without JCloudScale post-compilation processing enabled. This paradigm simplifies application development and testing, as developers can develop and verify application business logic using familiar local development environment.

The primary entities of JCloudScale are *cloud objects* (COs). COs are object instances which execute in the cloud. COs are deployed to, and executed by, so-called *cloud hosts* (CHs). CHs are virtual machines acquired from the IaaS cloud, which run a JCloudScale server component. They accept COs to host and execute on client request.

Furthermore, in order to follow the traditional cloud application development paradigm, CHs are currently not shared between the different target applications. They are acquired or instantiated by a single JCloudScale client application, and are only usable from this application. However, this limitation is more ideological than technical, as most of JCloudScale components support multi-tenancy [85] by default what will be shown in the following.

The program code responsible for managing virtual machines, dispatching requests to the virtual machines, class loading, and monitoring is injected into the target application as a post-compilation build step via bytecode modification. Optimally, COs are highly cohesive and loosely coupled to the rest of the target application, as, after cloud deployment, every further interaction with the CO constitutes a remote invocation over the network.

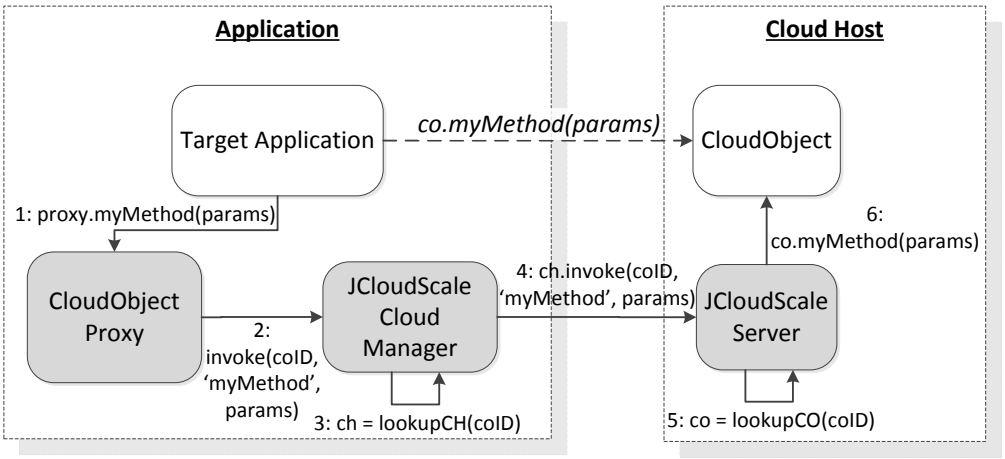


Figure 5.1: Basic interaction with cloud objects

In the JSTaaS example, implementations of test runners are good candidates for COs. Test execution potentially produces high computational load, and little interaction between the test runner and the rest of the application is necessary during test execution. Figure 5.1 illustrates the basic operation of JCloudScale in an interaction diagram. Whenever application invokes the `myMethod` on an object, JCloudScale middleware transparently intercepts the call and forwards it to the correct cloud host through the

internal proxy. The gray boxes indicate the code that is injected. Hence, these steps are transparent to the application developer. However, note that Figure 5.1 is simplified for readability. Some technicalities, such as classloading or data marshalling, have been omitted, but will be discussed in more detail later in this section.

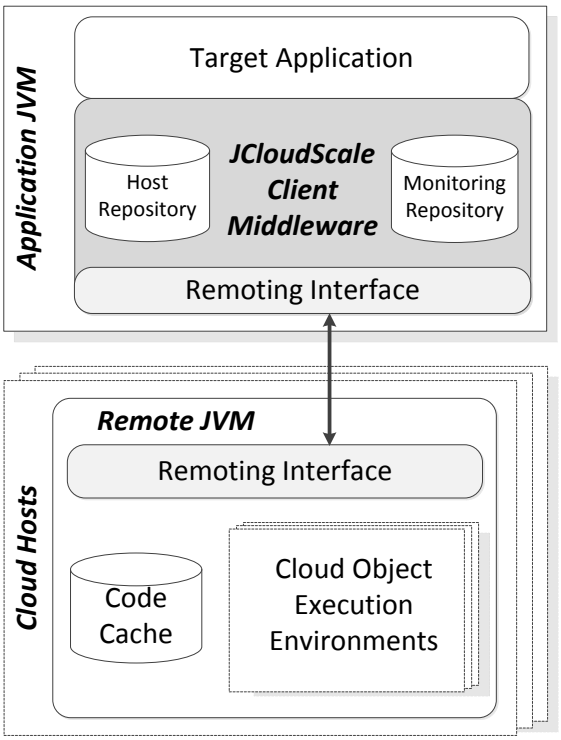


Figure 5.2: System deployment view

Figure 5.2 shows a high-level deployment view of a JCloudScale application. The gray box in the target application JVM also indicates injected components. Note that CHs are conceptually "thin" components, i.e., most of the actual JCloudScale business logic is running on the client side in the target application JVM. Such activities as cloud host management (illustrated by *Host Repository* component) and monitoring data collection and analysis (*Monitoring Repository*) are happening within the target application JVM. In its current version, JCloudScale does not support target applications that are themselves distributed.

CHs consist mainly of a small server component that accepts requests from clients, a code cache used for classloading, and sand boxes for executing COs. In order to achieve efficient yet light-weight encapsulation, these sand boxes are currently implemented via custom Java class loaders. On the client side, the JCloudScale middleware collects and aggregates monitoring data, and maintains the list of CHs and COs. Further, the client-side middleware is responsible for scaling up and down based on user-defined policies.

5.1.1 Interacting With Cloud Objects

Listing 5.1: Declaring COs in target applications

```
1  @CloudObject
2  public class MyTestExecutor {
3
4    @CloudGlobal
5    private static String myTestExecutorName;
6
7    @EventSink
8    private EventSink eventSink;
9
10   @DataSource(name = "testresults")
11   private static Datastore datastore;
12
13   public MyTestExecutor(){}
14
15   @Local // Constructed object will execute locally.
16   public MyTestExecutor(boolean local){}
17
18   public TestResult execute(@ByValueParameter TestSuite tests) {
19      ...
20   }
21
22   @DestructCloudObject
23   public void cleanup(){}
24 }
```

Application developers declare and adjust COs in their application code via simple Java annotations. These annotations serve as markers for JCloudScale to hint the places of target application that should be altered in order to successfully and efficiently distribute application code over the CHs. In the following, we refer to the minimal example given in Listing 5.1. A more comprehensive example, which also includes a step-by-step tutorial, is available in Appendix 9.3.

The main annotation that declares a class as a CO is `@CloudObject`. Any instance of the class, which is annotated with this annotation, is typically considered a CO. This means that every creation of an object of this class is altered to deploy the CO to the particular CH instead. However, in some cases not all instances of a class annotated with `@CloudObject` annotation should become COs. In order to support alternative behaviors, JCloudScale allows annotating a particular constructor with `@Local` annotation (see Listing 5.1, lines 15-16). This declares that all objects constructed using this constructor will execute locally as if there is no `@CloudObject` annotation on the class. Occasionally even this approach is not granular enough and only a certain set of class instances should be moved to the cloud. In this case, a programmatic interface (see Appendix 9.3) for proxy object creation can be used instead of the `@CloudObject` class annotation.

Another concern that target application developers should be aware of is the specifics of the code executed within the constructor of CO. In order to create a proxy object that will be used for interaction with an actual cloud object running in the cloud (see

Figure 5.1), JCloudScale creates a dynamic sub-class of the CO. However, to instantiate this proxy class, JCloudScale has to execute the parent constructor (i.e., the initially invoked constructor of the CO). Similarly, the constructor code of the CO will be executed on creation of the CO within the designated CH. Therefore, developers should be aware that the constructor code will be executed twice during CO creation. Usually this is fine as constructor code should only prepare the object itself for execution. However, if the constructor code is not idempotent or is computation-intensive, developers should move such code into a separate initialization method that application will invoke after the CO is already instantiated.

As is the case for any object in Java, the target application can fundamentally interact with COs in two different ways: invoking CO methods, and getting or setting CO member fields. In both cases, JCloudScale intercepts the operation, executes the requested operation on the CH, and returns the result (if any) back to the target application. In the meantime, the target application is blocked (more concretely, the target application remains in an "idle wait" state while it is waiting for the CH response). Fundamentally, JCloudScale aims to preserve the functional semantics of the target application after bytecode modification. That is, every method call or field operation behaves functionally identical to a regular Java program, except of the issues described above.

Finally, developers should be aware of the resource cleanup behavior of JCloudScale. By default, JCloudScale tries to follow default Java behavior and destroys COs whenever they are collected by the garbage collector within the target application. While this behavior is a reasonable default and should fit most applications, it may cause unnecessary resource consumption in the cloud and postpone resource cleanup, resulting in higher application operation costs. In order to improve this behavior, the `@DestructCloudObject` annotation allows declaring the point in application execution sequence when the particular CO can be removed from the appropriate CH. Note that after the method annotated with `@DestructCloudObject` is executed, every following interaction with the CO will result in a `JCloudScaleException`.

5.1.2 Static Fields and Methods in Cloud Objects

Another interesting topic is the behavior of the classes annotated as CO that contain static fields and methods. Operations on those are by default not intercepted by JCloudScale for performance reasons, as this would introduce a significant overhead even if the target application only reads from such static fields, what is usually the case. However, in some situations skipping interceptions may lead to a problem that we refer to as *JVM-local updates*: if code executing on a CH (for instance a CO instance method) changes the value of a static field, only the copy in this CH JVM will be changed. Other COs, or the target application JVM, will not be aware of the change. Hence, in this case, the value of the static field is tainted, and the execution semantics of the application changes after JCloudScale bytecode injection. To prevent this problem and preserve standard Java language semantics, static fields can be annotated with the `@CloudGlobal` annotation (see Listing 5.1, lines 4-5). Changes to such static fields are maintained in the target

application JVM, and all CH JVMs are operating on the target application JVM value via callback.

Particularly interesting is the problem of static method invocation if they are declared in a CO. If such a method is invoked from the target application, JCloudScale does not intercept the call and invokes it locally. The main reason for this behavior is that such invocation can not be associated with any particular CO, thus JCloudScale does not know on which CH this method should be invoked. However, if the static method is invoked from an instance of a CO, this method invocation will be associated with the particular CO and thus will execute on the CH that hosts this CO. While this approach improves performance of utility method invocation (what is usually the case with static methods), this may break synchronization patterns in user code. A summary of CO interaction semantics is shown in Table 5.1.

Table 5.1: JCloudScale interaction semantics

Target application ...	**JCloudScale ...**
... invokes CO constructor	... intercepts this method call, creates a new CO and deploys it on the CH.
... invokes CO constructor with `@Local` annotation	... does not intercept the object creation and local object is created.
... invokes a (non-static) CO method	... intercepts this method call and schedules its execution on the CH copy.
... invokes a static method	... does not intercept this operation. The static method will execute in the invoking application VM.
... gets or sets a (non-static) CO field	... intercepts this operation and gets or sets the value on the CH copy instead.
... gets or sets a static field	... does not intercept this operation. The static field value of the invoking VM will be used.
... gets or sets a static field with `@CloudGlobal` annotation	... intercepts this operation and uses the value from the target application VM.

5.1.3 Passing Data Objects

Evidently, most JCloudScale applications require parameter objects to be passed from the target application to the COs, or between COs. As the purpose of these objects is typically to transport data, we refer to them as *data objects*. JCloudScale supports data object passing via three common strategies, as summarized in Table 5.2. If small, primitive data objects (e.g., identifiers or numerical parameters) need to be passed, the common strategy is to pass them using `copy-by-value`. This strategy is simple and has a low overhead for small objects. However, it requires the objects to be marshallable. Technically, this means that they need to support standard Java serialization mechanisms. `by-reference` is more powerful, but should be used with care, as any interaction with the by-reference proxy results in additional remoting. Target applications often use `by-reference` to implement callback mechanisms, allowing for flexible asynchronous programming models. Furthermore, the `by-reference` mechanism allows COs to get access to a proxy of a different CO instance, hence, enabling communication between different COs. Finally, JCloudScale also supports the `shared` strategy, in which (serialized) data objects are stored in a persistent data store. This data store is shared

Table 5.2: Data object passing strategies

Strategy	Description
`copy-by-value`	Sends a deep copy of the object. Changes in the copy will not be reflected in the original object.
`by-reference`	Sends a proxy object (*by-reference proxy*) instead of a copy. Invocations of the proxy are redirected back to the original object.
`shared`	Data objects are exchanged by storing them in a shared data store. All CHs and the target application operate on the same copy of the data (ensured by transactional mechanisms and concurrency control).

among all CHs and the target application. This approach is commonly used if large chunks of data need to be passed around multiple times, as is the case for many scientific computing applications [134]. The `shared` strategy is also helpful if the JCloudScale application is expected to interface with the external data producers or consumers.

`copy-by-value` and `by-reference` are defined on Java method and field level, i.e., by annotating a parameter of a CO method, or a CO member field. In Listing 5.1, `TestResult` is a `by-reference` parameter, while the actual test suite is passed `copy-by-value`.

The `shared` model needs to be triggered explicitly by requesting JCloudScale to inject a connection handle to a shared database (*data store* in JCloudScale terms) into the Java application via dependency injection. For example, in Listing 5.1, a CouchDB NoSQL data store [135] is injected. The application code then explicitly reads from or writes to this data store. JCloudScale internally uses a custom data mapping framework, which allows to serialize arbitrary "Plain Old Java Objects" to a wide array of relational and non-relational data stores, including CouchDB, Riak, HBase and any SQL database compatible with the Java Persistence API (JPA). The `shared` approach also has the additional advantage that conflicts are detectable on database level via optimistic concurrency control [136]. Optimistic concurrency control essentially implies that each revision of a data object is associated with a numerical version flag. Whenever the data object is updated, the version flag is incremented. Whenever a data object should be updated, and the version flag in the data store is higher than the version of the object that should be written, a conflicting change is detected and reported back to the user via a `DataStoreException`. For `by-reference` and `copy-by-value`, developers need to make sure that different COs do not override changes of other COs manually, just as it is the case for any other multi-threaded Java application.

5.1.4 Fault Handling

Distributing applications with JCloudScale can potentially introduce faults, which are not apparent as long as the target application is executed locally. For instance, transient network outages can mean that a subset of COs is temporarily not available, or a terminated CH can lead to a permanent loss of COs. At this stage, JCloudScale does not provide sophisticated features to deal with these situations. However, JCloudScale notifies the target application via a custom exception type (`JCloudScaleException`)

and a more detailed exception message about such problems, and hence allows developers to deal with these issues as required in the target application.

Fundamentally, JCloudScale is most suitable for applications where the loss of individual COs or CHs is non-critical. This is in line with standard cloud architectures, which typically promote designing for failure, e.g., by adopting statelessness and redundancy [137]. Built-in support for redundancy is not part of the current JCloudScale release, but is part of our future research. Additionally, approaches for autonomous fault detecting [138] are considered as an option.

5.2 Application Code Distribution Framework

When an IaaS application has to scale up (i.e., use more virtual machines than before), one problem is how the availability of the current version of the application code, configuration files and other resources can be ensured on the new host. In the following, we will use the term "*program code*" as the shorthand for the application code and all dependent files. The trivial approach is to either send the correct version of the code to each machine on every request or to include the program code in the virtual machine base image. The first approach introduces significant overhead for application performance, while the second one is reasonable only in situations when the program code is entirely static and will not be modified during application lifetime. However, real-life applications are typically different. The program code often evolves over time, and developers would need to rebuild all cloud images related to the developed application. In such scenarios, hard coding the program code and other files into the virtual machine images becomes hard or even impossible. Additionally, it becomes even more problematic if we consider situations when multiple different code versions can be operating at the same CH over some time or even in parallel. The only alternative way to achieve the program code distribution in such conditions is to include facilities for dynamic code search and distribution on the middleware level.

This section introduces a framework for transparent runtime program code distribution. While this framework was developed within the JCloudScale, it is independent enough to be used separately in a different setup or use case that requires dynamic and transparent code distribution in the cloud environment.

5.2.1 Program Code Distribution Challenges

Dynamically distributing program code in a real-life IaaS cloud requires a number of aspects to be addressed. (1) Firstly, the framework needs to detect when the code that is to be executed is not available at all, and request the code from a code server (the machine that has the correct version of the application binaries). For instance, if the JSTaaS application described in Chapter 4 wants to delegate the generation of a test report to a CH, the required program code needs to be available at this virtual machine. (2) If this is not the case, the CH has to find a trusted code storage where appropriate code can be retrieved from. In our case, usually the application itself will act as a

Table 5.3: Summary of code distribution challenges

#	Challenge Name	Challenge Synopsis
1	Missing Code Detection	CHs need to be able to dynamically detect if program code needs to be loaded on demand.
2	Trusted Code Storage	CHs need to be able to locate a code storage service (typically the target application or a dedicated code server in the cloud).
3	Communication Middleware	CHs need to have access to a suitable communication middleware that allows them to dynamically load code.
4	Communication Protocol	CHs and target applications need to use an efficient protocol for minimizing the communication overhead incurred by dynamic code loading.
5	Code Versioning	CHs need to be aware that program code can change, and that the loaded program code is not valid indefinitely.

code server and deliver the required program code on demand, including the correct versions of all missing dependencies that are required to execute this code. Alternatively, it is possible to install a specific dedicated code server in the cloud, which then takes over this task from the target application to reduce its load. Evidently, it is possible to hard-code all required program code directly into the virtual machine images used by JCloudScale, but this drastically reduces the flexibility of the system and makes maintenance of the system cumbersome and time-consuming. (3) Thirdly, some means of communication need to be established which allow program code to be transferred at runtime from the trusted code storage to the cloud virtual machines. This communication can be handled either in a point-to-point fashion (e.g., via Web services technology, such as SOAP) or via a messaging middleware. (4) Fourthly, for practical performance reasons, the middleware needs to optimize the communication protocol between application and the CHs. For instance, it is typically not feasible to initiate dynamic code exchange routines separately for each missing class of application code. Instead, the middleware needs to smartly decide which additional code and non-code resources (e.g., images, configuration files) will also be required in addition to the already detected missing code. These dependencies should be distributed over the cloud at the same time to minimize the overhead of the code distribution. (5) Fifthly, after dynamically loading the program code, the middleware needs to decide how long this code and its dependencies can now be considered as valid. To this end, it is required that the JCloudScale middleware is able to detect when a different version of the program code is needed.

These challenges are summarized in Table 5.3. In the remaining of the section, we will discuss our approach to solve these challenges within the JCloudScale middleware and discussed code distribution framework.

5.2.2 Code Distribution Framework Overview

Whenever a CH in the JCloudScale middleware has to execute a new task, the system has to ensure that all necessary resources are available and schedule the execution of the task. Now we will describe our approach and show how we try to achieve efficient and seamless code distribution, solving the challenges described above.

When the target application approaches a code segment that can be delegated to the cloud, it schedules the execution on the CHs that are available at the moment. On the CH, the scheduled code starts executing, while a special class loader on the platform level maintains and fetches all required program code and other relevant resources, such as configuration files. The architecture of our solution is visualized in Figure 5.3, where the target application started from the application host can be seen. This application is distributing the work to the set of CHs that retrieve necessary code from the cloud code cache, code storage or directly from the target application. Due to this architecture, code that is being executed does not have to care about code availability and version, as the underlying infrastructure handles these problems seamlessly and transparently.

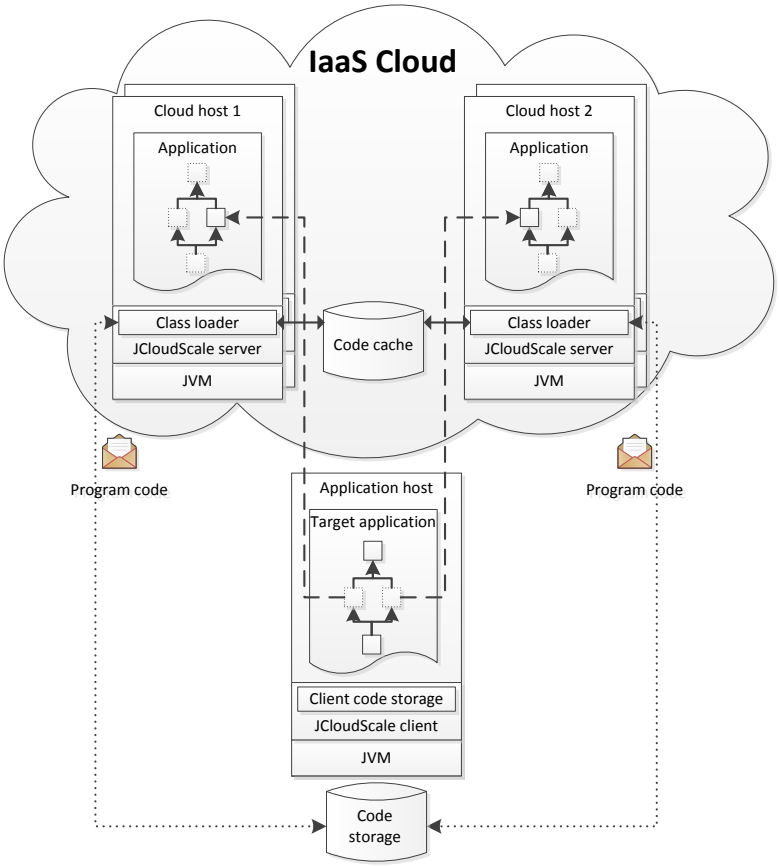


Figure 5.3: Overview of program code distribution model

5.2.3 Missing Code Detection

In most programming languages (including Java, as used by JCloudScale), detection of missing resources (both of code and non-code nature) can be handled by the developer through a special APIs. However, in order to avoid misbehaviors, solve stated challenges,

and be able to control code availability and load sequence, we implemented, basing on the available APIs, a special module in our middleware to intercept all requests for a program code at cloud hosts. Concretely, we intercept the class loading mechanism of the programming language to check against a set of already resolved classes. If the required code has already been loaded, it can be provided again without any additional work required from the class loader. If the required code has not been loaded before during this execution, the class loader checks a code cache for it, as shown in Figure 5.4. The details of this mechanism will be explained later. If the code was not found in the cache, the class loader requests the code from the target application (or from a trusted code storage), and waits for the response (see Figure 5.3).

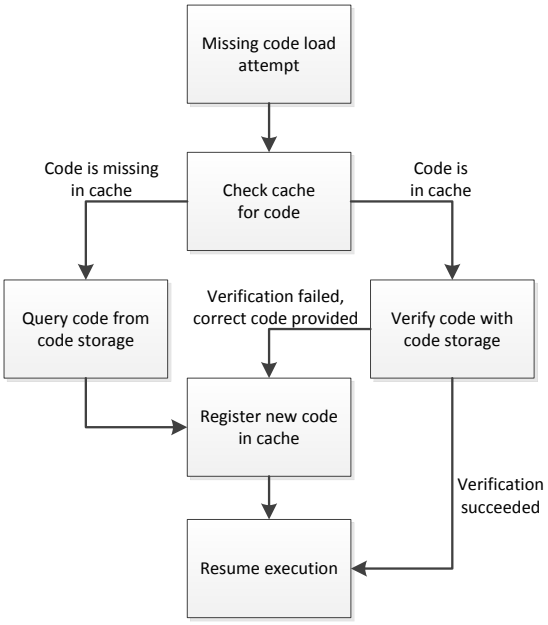


Figure 5.4: Code loading strategy

5.2.4 Communication Middleware

Our code loading system does not have any specific requirements for a particular communication channel, and usually can be used over the same communication facilities as used by the rest of the application or parent middleware (i.e., JCloudScale). Resource loading works based on simple blocking calls and may require the ability to initiate communication with the trusted code storage facility. The only communication channel properties that are important for this use case are reliability and a reasonable data transfer speed. Channel speed is very important as communication delay is directly influencing the application performance on the cloud. Evidently, any network communication is slow as compared to local code retrieval, and the slower communication is, the lower the

performance benefits that can be reached by distributing the application over the cloud in the first place are.

Reliability is also vital. Transfer errors, which CH can detect with the help of check sums, can dramatically impact communication speed due to code retransmission. In case of a communication failure, the application has to shutdown gracefully, as there is no code to continue executing the task.

In the current version of JCloudScale, a JMS-compatible message queue is used (i.e., Apache ActiveMQ[3]) to provide the communication channel for all client-host communication, including dynamic code loading.

5.2.5 Trusted Code Storage Location

While the creation of a dedicated code server may improve reliability and performance of the code distribution framework, for some cases this solution is not desirable. Sometimes it is required to be able to fetch actual code directly from the target application. For example, during software development or testing, it makes more sense to use the target application startup machine for code distribution instead of a dedicated server that has to be updated prior to every run. In such situations, code distribution service has to be provided from within the target application. Moreover, the target application is typically the most reliable source of the code, as the target application codebase contains exactly the code that the developer expects to run. Therefore, by default, the target application always runs the code distribution service, even in situations when a dedicated code server is expected to be used. This simplifies framework configuration and allows using the target application to update or verify code on the code server, or as a fallback option in case the code server is off line or overloaded.

The code distribution service within the target application has to be able to provide the code to the CHs without interrupting the main application thread. To achieve this, the service is started in a dedicated thread. When the code distribution service receives a request, it checks for an availability of the requested code and decides what to send. The code provided by the trusted source is then stored in the cache on the CH and execution is resumed. Additionally, inside the cache it is mapped to the appropriate target application in order to enable multi-tenancy and fast code verification for the following requests.

5.2.6 Code Versioning

To solve the challenges related to the code version control and updated code propagation, a code verification system was implemented as a part of the JCloudScale class loading mechanism. In case the code is available in cache, the class loader still has to ensure that the code has the same version as the target application expects. Therefore, the class loader carries out the code verification based on the last modification date and the size of the code files, as depicted in Figure 5.4. Evidently, some other alternatives to implement

[3] http://activemq.apache.org/

code verification are feasible as well (e.g., using hash-codes, explicit versioning via version numbers, or partial transfer), but we deemed the selected heuristic approach to be the fastest, while still being reliable enough for practical applications. This point of view is supported by the fact that similar approaches are used in other state-of-the-art solutions, e.g., RSync,[4] Apache Ant,[5] GNU Make[6] and others.

As code is stored in the cache, not only the required program code itself, but rather all files that were provided previous time for the same code request are verified. For each file in this set, the client either confirms that this is the expected code or provides the file that should be used. After this, the class loader delivers the correct code for the execution and, if necessary, updates the cached version.

5.2.7 Code Caching

In JCloudScale, CHs execute each separate request in a particular sand box. To this end, the code retrieval infrastructure on each CH resolves all resources for each target application separately. This allows parallel execution of different requests using different code bases, and restricts any possible influence between requests. However, evidently this approach introduces some redundant code transmission, because if the same program code should be used more than once, still it will be transmitted separately for each target application. To avoid this redundancy, we introduce a smart code caching mechanism.

For the first code request, when the required code is not yet cached, it has to be downloaded from a trusted code storage, while each of the following requests only uses the code available from the cache (if code verification is successful). When changes are detected during verification, the outdated code is either replaced or used in parallel to the updated version, depending on the cache usage and configuration policy. When there are no changes, the cached code can be used without transmission through the communication channel.

Table 5.4: Cache deployment selection tradeoff

	Host Private Cache	**Cloud Cache**
Cloud-Based Code Storage	+ code access speed - low cache hit rate	- no speed up + good cache hit rate
External Code Storage	+ code access speed - low cache hit rate	+ code access speed + good cache hit rate

The main task of the caching mechanism is to provide faster code fetching in situations when the same code is requested multiple times. Therefore, code from the cache has to be accessible faster than from a trusted code storage (e.g., the target application). The fastest possible location of the cache is the hard drive or even memory of the CH. This will give ideal access speed, but will reduce the cache hit rate, as each CH will have to

[4] http://rsync.samba.org/
[5] http://ant.apache.org/
[6] http://www.gnu.org/software/make/

maintain its own cache. In case of some distributed applications, this approach may give no benefits at all, as it is shown in Table 5.4. Another possible approach is to create a dedicated cache server or share one cache between multiple CHs. This is a good solution if the code is initially transmitted through an unreliable or slow channel. But if the application is already using a dedicated code service, a shared cache in the cloud hardly makes any sense, as access speed will be almost identical as to the code server.

From the situation described above, it is clear that we face a trade-off illustrated in Table 5.4. Depending on the environment configuration and situation, different approaches will be more efficient and, hence, preferable. Therefore, to achieve the best performance, it makes sense to allow the target application to decide on the preferred caching strategy.

5.2.8 Batch Loading

When the class loading infrastructure receives a request for new classes or resources to be loaded, there is not much information available to make some assumptions about the data that should be loaded. The only thing that is available is the name of the resource that should be retrieved. Therefore, the cloud host has to send a request to the storage facility with only required resource name specified (as described above, the situation is slightly different when there is code already available in the cache; for the sake of simplicity, we will omit this case now).

When the code retrieval request arrives at the storage facility, an appropriate service has to find the required piece of code and decide what to send along with it. Of course, the simplest scenario would be to send only the requested resource, but this would increase the cost of dynamic code loading and slow down an application, especially at the startup. Another extreme would be to send the complete application code at the first request: this would decrease the amount of messages, but might introduce even longer delay for the very first request, when the entire set of libraries and code base is transmitted even if most of them are not necessary at all for CO execution. Considering the fact that usually not all code would be required on each cloud host, this option may introduce even more overhead than the first one.

One possible option to solve this trade-off would be to allow the target application to configure the amount of code that should be transferred for each request. However, this approach would be rather cumbersome for the developers and against the primary design goals of JCloudScale (making it transparent and easy to build cloud applications). Another choice would be to use heuristics, which would propose a satisfying solution for common usage scenarios.

For example, if the requested class belongs to a library (e.g., a jar file), it makes sense to send the entire library instead, as the chances that other resources from that library will be requested are high. Similarly, if the class belongs to a package, it makes sense to consider sending the entire package. Also, if the class has some dependencies or belongs to a hierarchy of classes or interfaces, other classes are very likely to be needed as well.

All of these heuristics have their own benefits and problems and it is complicated to determine which of them should be used as the default behavior. To determine the

influence of these factors on real-life applications, we included a number of different batch loading algorithms in our numerical evaluation, presented in the original work [20].

5.2.9 Summary

Summarizing said above, the transparent code distribution framework allows for distributing code to the cloud on demand and transparently to the target application. Presented approach targets seamless application execution in a distributed environment and permits alternative code versions running on each cloud host in parallel. The framework was evaluated in a real-life application and the overhead of different code distribution and caching strategies were compared and analyzed. The evaluation showed that selected code distribution approach provides a list of benefits over alternatives and minimum overhead for the users, while requiring insignificant amount of time to configure and use.

5.3 Target Application Development Process

To show how the development process of JCloudScale-based applications looks like, we go through the set of steps necessary to bring a Maven-based application (as introduced in Chapter 4) to the Amazon EC2 cloud. In this section, only the core elements of application development process are highlighted. This should allow reader to understand the level of complexity and intrusion that target application developers have to face in order to start using JCloudScale. In more details this process is described in Appendix 9.3.

While here we focus on Maven-based applications, JCloudScale allows building target applications without Maven as well. Differences from the Maven-based sequence are highlighted in Appendix 9.3.

The target application development process consists of the following three fundamental steps:

1. The project setup has to be changed to reference JCloudScale;
2. COs have to be selected and necessary annotations have to be added;
3. JCloudScale has to be configured to efficiently and elastically scale the target application.

5.3.1 Target Application Setup

The first step on a way to build a JCloudScale-based cloud application requires the modification of a `pom.xml` file to reference JCloudScale and to apply post-compilation processing required to inject JCloudScale behavior code into the target application.

In order to add a reference to JCloudScale, target application developers have to add another `dependency` block which is shown in Listing 5.2. This dependency references the current version of JCloudScale and provides the core JCloudScale functionality that is required for almost any JCloudScale-based application. In case

developers want to leverage some additional JCloudScale functionality (e.g., application bursting, database interaction, or scaling behavior definition modules), developers would need to add corresponding dependencies as well.

Listing 5.2: Introducing JCloudScale dependency

```
1 <dependency>
2     <groupId>jcloudscale</groupId>
3     <artifactId>jcloudscale.core</artifactId>
4     <version>0.4.0</version>
5 </dependency>
```

If developers build the target application now, they will see that the build process fails with a dependency resolution error. This happens because by default Maven locates dependencies in local or central repositories. However, none of them contains JCloudScale artifacts. While we are continuously working on the JCloudScale inclusion into a Maven central repository, currently the JCloudScale project is just a research prototype and does not match every requirement necessary to be included into the Maven central repository. Until this process is successfully completed, developers need to reference a private repository that contains JCloudScale artifacts and all necessary dependencies. In order to do this, the code from Listing 5.3 has to be included into a `pom.xml` file. This code defines that Maven has to include external repository into the artifact discovery process.

Listing 5.3: Referencing infosys maven repository

```
1 <repositories>
2     <repository>
3         <id>infosys-repository</id>
4         <url>http://www.infosys.tuwien.ac.at/mvn</url>
5     </repository>
6 </repositories>
```

After the correct repository is referenced, the target application should successfully load the necessary dependencies and compile the available source code of the target application. Finally, concluding this step, developers need to add the code post-compilation processing definition, as defined in Listing 5.4. The plugin description presented in Listing 5.4 alters the usual application build process in order to include additional step of AspectJ processing. AspectJ is an aspect-oriented framework (see Section 2.4) that allows post-compilation modifications of Java applications. JCloudScale uses AspectJ to apply necessary code modifications that allow application distribution in the cloud.

5.3.2 COs Selection

The idea of JCloudScale is based on the notion of cloud objects, as introduced in Section 5.1. As the most common and resource-intensive task of JSTaaS application is

Listing 5.4: Applying JCloudScale post-compilation processing

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>aspectj-maven-plugin</artifactId>
    <version>1.4</version>
    <configuration>
        <source>1.7</source>
        <target>1.7</target>
        <complianceLevel>1.7</complianceLevel>
        <verbose>true</verbose>
    </configuration>
    <executions>
        <execution>
            <configuration>
                <XnoInline>true</XnoInline>
                <aspectLibraries>
                    <aspectLibrary>
                        <groupId>jcloudscale</groupId>
                        <artifactId>jcloudscale.core</artifactId>
                    </aspectLibrary>
                </aspectLibraries>
            </configuration>
            <goals>
                <goal>compile</goal>
                <goal>test-compile</goal>
            </goals>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.7.0</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>1.7.0</version>
        </dependency>
    </dependencies>
</plugin>
```

to execute customers' tests, the class `MyTestExecutor` that wraps the separate test suite execution is a good candidate. Furthermore, this class is strongly decoupled and requires minimal interaction with other components of the application, what perfectly fits the notion of an ideal CO.

Listing 5.5 shows the test execution class that is being distributed by JCloudScale. Mainly it consists of application-specific business logic with a number of JCloudScale annotations added. As the `MyTestExecutor` class is annotated with an `@CloudObject` (see line 1 in Listing 5.5), all interactions with the instances of this class are intercepted by JCloudScale and scheduled to the appropriate CH. In addition, to optimize performance, some method parameters and return values are annotated with the appropriate parameter passing annotations that allow treating parameters either as *copy-by-value*

Listing 5.5: The skeleton of the test execution class

```
1  @CloudObject
2  public class MyTestExecutor {
3          @CloudObjectId
4          private UUID coId;
5
6          @DataSource(name = "testresults")
7          private Datastore datastore;
8
9          public @ByValueParameter UUID getId() {
10                 return coId;
11         }
12
13         public void setSuite(@ByValueParameter TestSuite suite, int testId){
14                 ...
15         }
16
17         public void execute(TestSuiteExecution statuses,int suiteNr){
18                 ...
19         }
20
21         @DestructCloudObject
22         public void cleanup(){}
23 }
```

(see lines 6 and 13 in Listing 5.5) or as *by-reference* (see line 17 in Listing 5.5). For example, as the `statuses` parameter of the `execute` method (line 17 in Listing 5.5) is not annotated by any specific annotation, it is treated as *by-reference* and all changes applied to this object in CH are retransmitted back to the target application.

Another important annotation is `@DestructCloudObject` on the `cleanup` method (see line 21 in Listing 5.5) . This annotation specifies that this is the last invocation on this CO and, after invocation of this method is finished, this CO can be destructed. This allows optimizing resource usage and cleaning unnecessary objects from the CHs.

Separately we would like to note the dependency injection feature of JCloudScale. Two fields of this class (`coId` (line 4 in Listing 5.5) and `datastore` (line 7 in Listing 5.5)) are annotated with appropriate annotations to allow additional interaction with the JCloudScale middleware. For example, the `coId` field annotated with `@CloudObjectId` annotation allows using JCloudScale-defined CO Id of this particular object.

5.3.3 Configuring JCloudScale

At this point, the JSTaaS application is already distributed by JCloudScale. However, the distribution is happening in the so-called "debug" mode: instead of using separate cloud hosts, JCloudScale spawns new JVMs on the same host the target application is started. This mode is perfect for debugging and ensuring that everything works as expected prior to deploying the application to the cloud. In order to deploy the application on a real cloud, an appropriate configuration has to be provided.

There is a number of ways to configure JCloudScale, described in more details in Appendix 9.3. In this section, system properties will be used to configure the developed target application. An example of such code-based JCloudScale configuration provider is shown in Listing 5.6. The system property `jcloudscale.configuration` has to define either the path to an XML file containing a serialized JCloudScale configuration, or the name of a class that has a static method with the `@JCloudScaleConfigurationProvider` annotation (see line 1 in Listing 5.6) that returns an instance of `JCloudScaleConfiguration` class (see line 2 in Listing 5.6). Within this method, developer can either load configuration from some application-specific storage or build configuration on the fly using `JCloudScaleConfigurationBuilder` class, as it is shown in lines 4–15 of Listing 5.6. During the application run-time, JCloudScale will load configuration from the configured location on demand.

Listing 5.6: An example of JCloudScale configuration provider

```
1         @JCloudScaleConfigurationProvider
2         public static JCloudScaleConfiguration getConfiguration()
3         {
4                 return new JCloudScaleConfigurationBuilder(
5                         new EC2CloudPlatformConfiguration()
6                         .withAccessKey(EC2_ACCESS_KEY)
7                         .withSecretKey(EC2_SECRET_KEY)
8                         .withAwsEndpoint(AWS_ENDPOINT)
9                         .withInstanceType(INSTANCE_TYPE)
10                        .withSshKey(SSH_KEY)
11                        )
12                .withMQServerHostname(serverAddress)
13                .with(new ScalingPolicy())
14                .withLogging(Level.INFO)
15                .build();
16        }
```

If an appropriate Amazon EC2 configuration is specified, the application can already be distributed in the Amazon EC2 cloud. However, the default host managing policy will not be optimal for this particular target application. In order to adjust it, a custom scaling policy based on the monitoring information or domain-specific logic has to be developed.

It is challenging to design an elastic and effective scaling policy. To address this issue, JCloudScale provides a configurable scaling policy definition language, described in Chapter 6. This language is available as an JCloudScale extension, but a custom scaling policy can be defined without it. Fundamentally, every scaling policy has to answer a set of questions to define how target application should behave in the cloud. The basic requirements for each custom scaling policy are presented in Listing 5.7. Developers have to implement the method `selectHost` (see lines 9–13 in Listing 5.7) that defines to which CH each CO has to be deployed. Additionally, developers should also define when each CH is not needed any more and may be terminated. This has to be defined in the `scaleDown` method (see lines 15–18 in Listing 5.7) that is periodically invoked for each active CH. Finally, programmers may implement the `initialize` (lines 3–7

in Listing 5.7) and the `close` (lines 20–24 in Listing 5.7) methods that are invoked on JCloudScale initialization and shutdown accordingly. These methods are intended for initialization and termination of any background activities, such as performance monitoring, related to the core scaling policy behavior.

Listing 5.7: A scaling policy example

```
1  public class ScalingPolicy extends AbstractScalingPolicy {
2
3          @Override
4          public void initialize(IHostPool hostPool){
5                  ... // here we define how cloud environment
6                  // should be prepared on application startup.
7          }
8
9          @Override
10         public synchronized IHost selectHost(
11                                 ClientCloudObject cloudObject, IHostPool pool){
12                 ... // here we define where to deploy the new cloud object.
13         }
14
15         @Override
16         public boolean scaleDown(IHost scaledHost, IHostPool hostPool){
17                 ... // here we define if the specified host should be shut down.
18         }
19
20         @Override
21         public void close(){
22                 ... // here we define how environment should
23                 // be cleaned up on application shutdown.
24         }
25 }
```

After completing these changes, our JSTaaS application is fully capable of running over the Amazon EC2 cloud, where we can further adapt it to achieve the required performance and resource consumption level.

5.3.4 Development Process

As JCloudScale makes it easy to switch between different cloud environments, the middleware supports a streamlined development process for elastic applications, as sketched in Figure 5.5. The developer typically starts by building an application as a local, multi-threaded Java application using common software engineering tools and methodologies. Once the target application logic is implemented and tested, the developer adds the necessary JCloudScale annotations, as well as scaling policies, monitoring metric definitions, and JCloudScale configuration as required. Using configuration, the developer specifies a deployment in the local environment first. This allows application testing and debugging within the developer's machine, including tuning and customizing the scaling policy. Finally, once the developer is satisfied with the target application behavior, the application can be configured to run in an actual cloud environment.

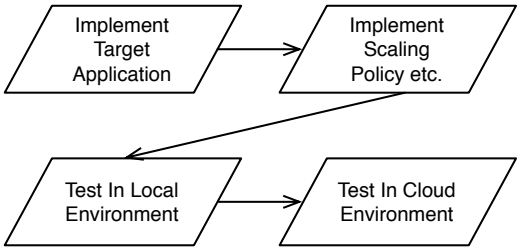


Figure 5.5: Conceptual development process

The presented process aims to significantly decrease the amount of difficulties that developers experience when building cloud applications, as it allows fixing errors and issues on the stage where they appear, decreasing time-consuming application testing on an actual cloud. Following this process, developers can fix issues related to business logic while the application is still running in the preferable IDE. Similarly, the solid part of distribution problems are already visible in a simulated local distribution environment.

However, of course this process is idealized. Practical usage shows that developers will have to go back to a previous step in the process on occasion. For instance, after testing the scaling behavior in the local environment, the developer may want to slightly adapt the target application to better support physical distribution.

CHAPTER 6

Scaling Behavior

While the previous chapter presented the basic notions and internal activities of JCloudScale, this chapter focuses on building elastic applications and the instruments that JCloudScale provides for cloud application developers.

Transparent application distribution on its own is not the main feature of JCloudScale middleware. Transparent code distribution for JCloudScale is only a method that allows for elastic and efficient application execution in the cloud. The presented architecture and paradigm of JCloudScale permits developers to completely separate the target application distribution logic from the business logic of the application. This approach simplifies the development of an independent and compact scaling behavior that is only weakly dependent on the core application. Therefore, application elasticity can be programmed by a separate team of developers or easily shared between multiple applications. Additionally, scaling behavior can be easily substituted during any stage of application execution, what allows for multiple independent scaling strategies improving or competing with each other.

Targeting to have as few restrictions as possible, JCloudScale on its own provides a very generic and abstract API for scaling behavior definition. Target application distribution behavior has to be presented in the form of a *Scaling policy* that allows defining used host management activities and task distribution. While such an API is flexible enough to develop a scaling policy of almost any complexity and behavior, it requires from developers significant efforts to develop even a simple scaling algorithm. To address this issue and to simplify the creation of a scaling behavior, the approaches and frameworks, presented in this chapter, were developed.

6.1 Autonomic Elasticity via Complex Event Processing

One central advantage of JCloudScale in comparison to modern PaaS platforms, is that it allows for building elastic applications by manually mapping requests to a dynamic pool of CHs. This encompasses three related tasks:

1. Performance monitoring;

2. CH provisioning and de-provisioning;

3. CO-to-CH scheduling and CO migration.

One design goal of JCloudScale is to abstract from technicalities of these tasks, but still grant developers the necessary low-level control over the elasticity behavior of the target application.

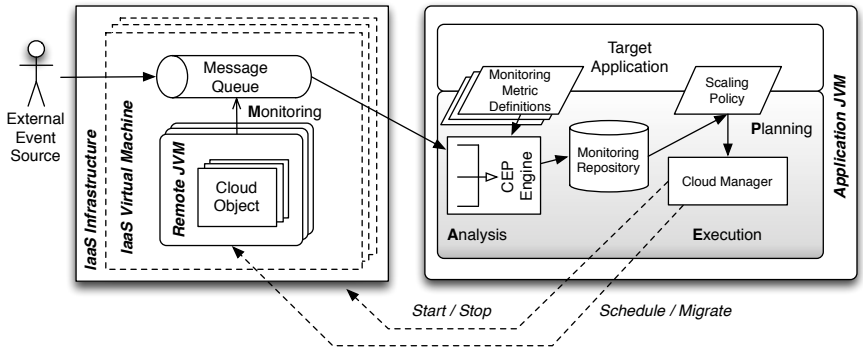


Figure 6.1: Autonomic elasticity

An overview over the JCloudScale components related to elasticity, and their interactions, is given in Figure 6.1. Conceptually, JCloudScale implements the well-established autonomic computing control loop of monitoring-analysis-planning-execution [139] (MAPE). The base data of monitoring is provided using event messages. All components in a JCloudScale system (COs, CHs, as well as the middleware itself) trigger a variety of predefined lifecycle and status events, indicating, for instance, that a new CO has been deployed or that the execution of a CO method has failed. Additionally, JCloudScale makes it easy for target applications to trigger custom (application-specific) events. Finally, events may also be produced by *external event sources*, such as an external monitoring framework. All these events form a consolidated stream of monitoring events in a *message queue*, by which they are forwarded into a *complex event processing (CEP) engine* [140] for analysis. CEP is the process of merging a large number of low-level events into high-level knowledge, e.g., many atomic execution time events can be merged into meaningful performance indicators for the system in total.

Developers steer the scaling behavior by defining a *scaling policy*, which implements the planning part of the MAPE loop. This policy is invoked whenever a new CO needs to be scheduled, and is also responsible for deciding whether to de-provision an existing CH at the end of each Billing Time Unit (BTU). A simplistic example that demonstrates how round-robin task scheduling policy is implemented, is shown in Listing 6.1. This policy schedules COs in a round-robin fashion among existing CHs, and never scales up.

Listing 6.1: Example round-robin scaling policy

```
1  public class RoundRobin extends AbstractScalingPolicy {
2
3    int index = 0;
4
5    public IHost selectHost(ClientCloudObject newCloudObject, IHostPool hostPool) {
6      return hostPool.getHosts().get((index++) % hostPool.getHostsCount());
7    }
8
9    public boolean scaleDown(IHost host, IHostPool hostPool) {
10     return host.getCloudObjects().size() == 0;
11   }
12 }
```

The policy terminates a host if it is unused (that is, there are no COs deployed at it) at the end of the CH BTU.

Clearly, most real scaling policies are more complex than the one in Listing 6.1. Using the `ClientCloudObject`, `IHostPool`, and `IHost` APIs, defined in Appendix 9.3, developers are able to schedule the provisioning of new CHs (optionally asynchronously), migrate existing COs between CHs, and schedule COs to a CH. Oftentimes, these decisions will be based on monitoring data. Hence, developers can define any number of *monitoring metrics*. Metrics are simple 3-tuples <*name, type, cep-statement*>. CEP-statements are defined over the stream of monitoring events. An example, which defines a metric `AvgEngineSetupTime` of type `java.lang.Double` as the average `duration` value of all `EngineSetupEvents` received in a 10 second batch, is given in Listing 6.2.

Listing 6.2: Example of defining monitoring metrics via CEP

```
1  MonitoringMetric metric =
2    new MonitoringMetric();
3  metric.setName("AvgEngineSetupTime");
4  metric.setType(Double.class);
5  metric.setEpl(
6    "select avg(duration)
7     from EngineSetupEvent.win
8       :time_batch(10 sec)"
9  );
10 EventCorrelationEngine.getInstance()
11   .registerMetric(metric);
```

Monitoring metrics range from very simple and domain-independent (e.g., calculating the average CPU utilization of all CHs) to rather application-specific ones, such as the example given in Listing 6.2. Whenever the CEP-statement is triggered, the CEP engine writes a new value to an in-memory *monitoring repository. Scaling policies* have access to this repository, and make use of its content in their decisions. In combination with monitoring metrics, scaling policies are a well-suited tool for developers to specify how the application should react to changes in its workload. Hence, sophisticated scaling policies that minimize cloud infrastructure costs or that maximize utilization [141] are

easy to integrate. As part of the JCloudScale release, we provide a small number of default policies that developers can use out of the box. However, these policies are usually too simplistic to be used in an actual application. Therefore, they are mainly assumed to serve as an example, while developers will write their own domain-specific scaling policies. This has proven necessary as, usually, no generic scaling policy is able to cover the needs of every application and the only way to achieve effective application distribution is to integrate application-specific parameters and criteria into the scaling policy.

Finally, the *cloud manager* component, which can be seen as the heart of the JCloudScale client-side middleware and the executor of the MAPE loop, enacts the decisions of the policy by invoking the respective functions of the IaaS API and the CH remote interfaces (e.g., provisioning of new CHs, de-provisioning of existing ones, as well as the deployment or migration of COs).

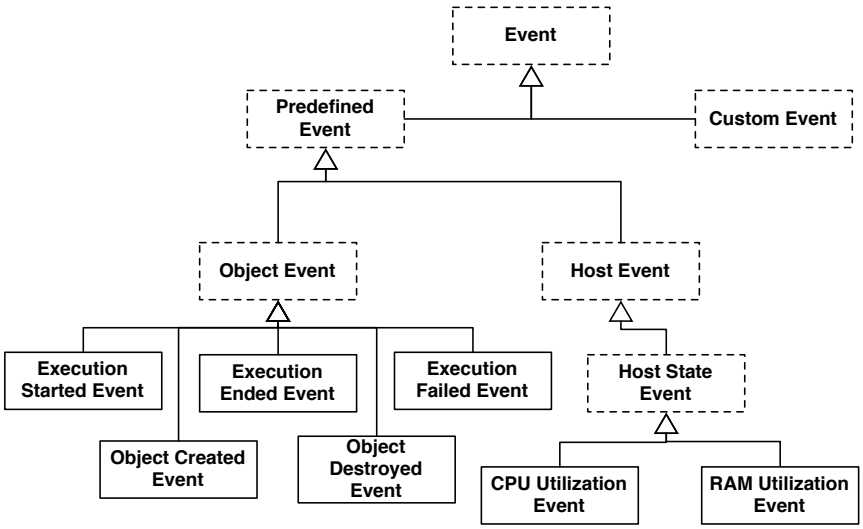


Figure 6.2: Monitoring event hierarchy

Figure 6.2 depicts the type hierarchy of all predefined events in JCloudScale. Dashed classes denote abstract events, which are not triggered directly, but serve as classifications for groups of related events. All events further contain a varying number of event properties, which form the core information of the event. For instance, for `ExecutionFailedEvent`, the properties contain the CO, the invoked method, and the actual error. Developers and *external event sources* can extend this event hierarchy by inheriting from `CustomEvent`, and writing these custom events into a special event sink (injected by the middleware, see Listing 5.1). This process is described in more detail in [48].

6.2 Cloud Targeting and Bursting

As all code that interacts with the IaaS cloud is injected, the JCloudScale programming model naturally decouples Java applications from the cloud environment that they are physically deployed to. This allows developers to re-deploy the same application to a different cloud simply by changing the respective parts of the JCloudScale configuration. JCloudScale currently contains three separate cloud backends, supporting OpenStack-based private clouds, the Amazon EC2 public cloud, and a special *local environment*. The local environment does not use an actual cloud at all, but simulates CHs by starting new JVMs on the same physical machine as the target application. Support for more IaaS clouds, for instance Microsoft Azure's virtual machine cloud, is an ongoing activity. Moreover, we aim to introduce systematic testing to ensure reliable deployment of CHs, which is a key requirement for elasticity [142]. Figure 6.3 illustrates the different types of environments supported by JCloudScale.

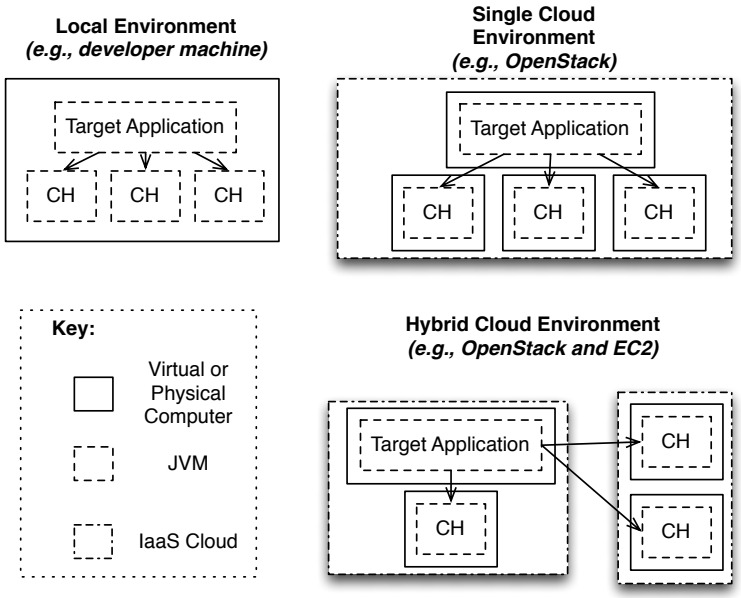


Figure 6.3: Supported deployment environments

Modern cloud applications are usually developed targeting one of two possible infrastructures. Some applications target *private clouds*, which are usually represented by virtualized private data centers of a limited size under the complete jurisdiction of the same entity as the developed application; or *public clouds*, which are usually represented by independent extra-large shared virtualized data centers [1].

Nowadays the decision to execute applications in a public or private cloud usually has to be taken prior to application deployment. Moreover, in order to change this decision, developers usually may need to update application behavior to address the peculiarities of the selected cloud platform.

The ability to switch the targeting platform with a simple change in the configuration allows JCloudScale-based applications avoiding these problems at all. Furthermore, JCloudScale unlocks opportunities for *cloud bursting* application development.

The cloud bursting concept [143] targets the idea of building applications that are able to spread over multiple cloud environments (i.e., "burst") whenever resources in one environment are insufficient or inappropriate. This incorporates the concept of migration between multiple environments in order to decrease execution costs and the ability to execute different types of jobs in different environments addressing security or performance concerns. Nowadays cloud bursting is mostly a research idea[144].

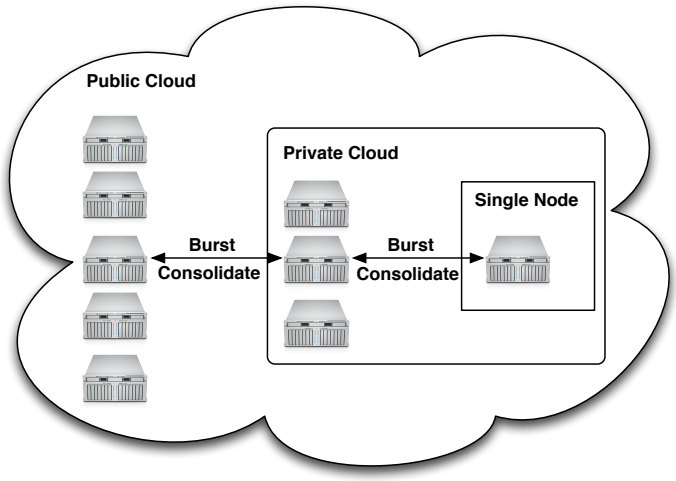


Figure 6.4: Basic three-phase cloud bursting model

Figure 6.4 illustrates the idea of cloud bursting in a model that will be referenced further. This approach is in line with the existing research on the concept of cloud bursting [144].

Describing cloud bursting using the use case from Chapter 4, initially the JSTaaS application runs in a single host. Actually, up to some amount of tests to execute, the performance of such deployment is higher than of a distributed application, what is mainly caused by the communication overhead which is avoided when everything runs within a single host. However, once the amount of tests exceeds the capabilities of a single host, JSTaaS has to burst into a private cloud. At this point JSTaaS is already distributed, but all communication happens over a fast local area network, thus performance is still good and the impact of networking is still comparable to a single host deployment. Finally, when the load exceeds the capabilities of a private cloud, JSTaaS has to burst into a public cloud. Whenever this happens, the JSTaaS application has to take into account the workload transmission costs and the communication overhead. Additionally, at this point developers have to consider privacy issues, as not every user would like to have its private code to be accessible, even theoretically, by someone in a

public cloud.

Such JSTaaS behavior may be an interesting trade-off between the resource utilization, application performance and execution costs. The presented approach allows JSTaaS accessing unlimited resources of the public cloud, while providing offers for privacy-concerned customers and being able to execute critical short-running tests within the private infrastructure.

Unfortunately, designing cloud bursting behavior is quite challenging nowadays. There are no standardized tools that take care of managing multiple environments at once, thus developers have to develop code distribution, application monitoring and environment management code for each used cloud over and over again. Moreover, developers have to manually design cloud bursting behavior that fits particularly their application, what significantly holds back the global adaptation of the cloud bursting idea.

In order to address this issue, the cloud bursting extension to JCloudScale was developed [78]. The resulting framework provides a transparent application performance monitoring and automatically decides when to burst or retreat from every used environment. Additionally, the presented framework provides an API that allows developing custom cloud bursting scaling policies of varying complexity. The developed cloud bursting policy and framework are presented in more details and evaluated in the original work [78].

6.3 A Declarative Event-Based Scaling Policy Language

As the development of an elastic and effective scaling policy is an important part of cloud application development, tools provided by a cloud platform should allow crafting an efficient scaling behavior as easy as possible. Designing such tools, cloud providers face an important trade off between the functionality and ease to use. From the one side, the idea to design a generic and transparent solution that "just works" without any labor required from a cloud application developer should definitely satisfy everyone. However, such universal application will either fit only a subset of cloud applications or fail to achieve the desired productivity due to the generality of approach. Therefore, some developers will still need to have a powerful mechanism to design a custom scaling behavior that achieves better performance than the generic approach.

Modern Platform-as-a-Service solutions (e.g., Google Appengine[1] or IBM Bluemix[2]) provide simple automated, rule-based solutions to this problem, which e.g., add and remove servers based on CPU utilization thresholds. Those simple solutions are a perfect fit for many three-tier web applications [99]. However, there are many real-life applications that do not fit this model. For some applications, incoming tasks differ substantially in resource usage per request, or the architectural design requires non-trivial mapping of tasks to resources [145]. Similarly, problems appear when legislative rules regarding data handling apply. For example, the European Union establishes specific rules for how medical data is to be handled by service providers [146].

[1] `https://appengine.google.com/`

[2] `http://www.ibm.com/software/bluemix`

In these situations, cloud developers generally fall back to Infrastructure-as-a-Service clouds, which allow more fine-grained elasticity control. However, choosing IaaS also implies that developers have to create their own cloud management solutions, which are both, cumbersome and error-prone. Further, manual development of elasticity behavior is repetitive, as conceptually the same kind of abstract behavior needs to be implemented in many different applications.

Similarly to IaaS, the JCloudScale scaling capabilities presented above target to provide a powerful environment to design a custom scaling policy rather than an universal scaling behavior. After a while, it became clear that scaling policy definition in JCloudScale is a time-consuming process that is hardly different from the custom solution, which developers have to create for an IaaS-based application. The discussed JCloudScale scaling definition approach was intentionally selected to have minimum limitations on developers and to collect some usage statistics and best practices to design a scaling behavior for JCloudScale-based applications.

For this, the SPEEDL language was developed. SPEEDL is a declarative and extensible domain-specific language [21] (DSL) that simplifies the creation of elastic, application-specific cloud scaling behavior on top of IaaS clouds. SPEEDL allows for the definition of scaling policies in form of a set of event-condition-action (ECA) rules managing the amount and types of resources (e.g., VM instances) acquired from the cloud, as well as the mapping of incoming tasks to these resources for processing. Unlike existing industrial solutions, SPEEDL is extensible and allows for application-specific rules development.

While SPEEDL is designed to be a generic and universal scaling policy definition language, it is based on our previous experience with JCloudScale scaling policy definitions. Moreover, the reference implementation of SPEEDL is presented as a JCloudScale plug-in[3] and can be transparently integrated into the core scaling definition architecture of JCloudScale. Nevertheless, the reference implementation of SPEEDL is not hard-wired to JCloudScale and can be easily used stand-alone or as part of a third-party solution as well.

6.3.1 Language Design Considerations

While every cloud application has its own specifics and unique requirements, cloud applications typically all make use of a number of general constructs defining how cloud resources should be acquired and used. With SPEEDL, these requirements were structured, formalized, and represented as a declarative DSL. The design and architecture of SPEEDL, as well as the concrete out-of-the-box rules provided, are influenced by existing industrial cloud systems and platforms, ongoing parallel research activities in the field [104, 110] and our former experience with building and supporting elastic applications [16, 19, 24].

Existing cloud research typically models elasticity either in the form of a control loop (e.g., in the sense of autonomic computing [139]), or, more reactively, as a set of

[3] `https://github.com/xLeitix/jcloudscale/tree/master/ext`

ECA rules [147]. While the former approach is often preferred in scientific work, those solutions often struggle with being narrow for a specific domain and challenging to reuse or adapt to fit different applications. The ECA-based approach avoids this problem [147]. Hence, SPEEDL was built on the notion of CEP [140], which provides reactiveness and responsiveness to complex scenarios and application behaviors. An additional advantage is that the basic declarative event-based model used by SPEEDL is conceptually close to how practitioners define elasticity behavior in common PaaS services [148]. Hence, we argue that the SPEEDL approach integrates better with current cloud developer's mindsets.

6.3.2 SPEEDL Overview

Scaling behavior in SPEEDL is defined by the developer as a scaling policy SP. Every application makes use of exactly one scaling policy, which can be understood as a 2-tuple $SP =< TM, RM >$, with TM being a set of task management rules, and RM a set of resource management rules. Both, TM and RM are allowed to be the empty set ($TM, RM = \{\}$). In this case, SPEEDL does not consider request scheduling, or does not actually scale up or down. Every concrete rule $r \in TM \cup RM$ is in turn a 3-tuple $r =< E, C, A >$, with E, C and A being sets of triggering events, guarding conditions, and resulting actions correspondingly. Actions differ for task and resource management rules. For example, task management actions often entail scheduling a task to one specific resource. The notion of "task" in this scope represents any workload or application component that needs to be executed on a cloud resource. Resource management actions may, for instance, entail starting a new resource of a specific type. The ECA structure of SPEEDL defines a distinct responsibility of each part of the scaling policy and provides clear and effective ways to configure the behavior of each rule. Additionally, this allows applications to quickly react to changes in the system state, without requiring periodic background checks as it is common in other approaches [147].

The implementation is technically realized as a fluent interface [149]. This makes the actual DSL concise, expressive, and easy to understand. Using method cascading, developers can simply invoke required rules separated by dots and produce compact and tidy code that can be read like a declarative sentence.

Figure 6.5 gives a high-level overview over the main components and interactions of a SPEEDL-based application. SPEEDL integrates with the actual application business logic as a third-party component (i.e., a library in the Java implementation). The SPEEDL implementation mainly executes a defined scaling policy, which consists of task and resource management rules. All rules are triggered via events from an event bus. This bus receives and correlates, in the sense of CEP, events from the cloud resources, the application, and SPEEDL itself. Task management rules instruct the application to execute specific tasks on specific hosts, while resource management rules interact with the cloud to acquire and release resources. For both, events and rules, SPEEDL contains a set of predefined constructs, which were defined based on requirements and features of other literature and existing products. Additionally, developers have the opportunity to extend these sets of predefined events and rules with application-specific ones.

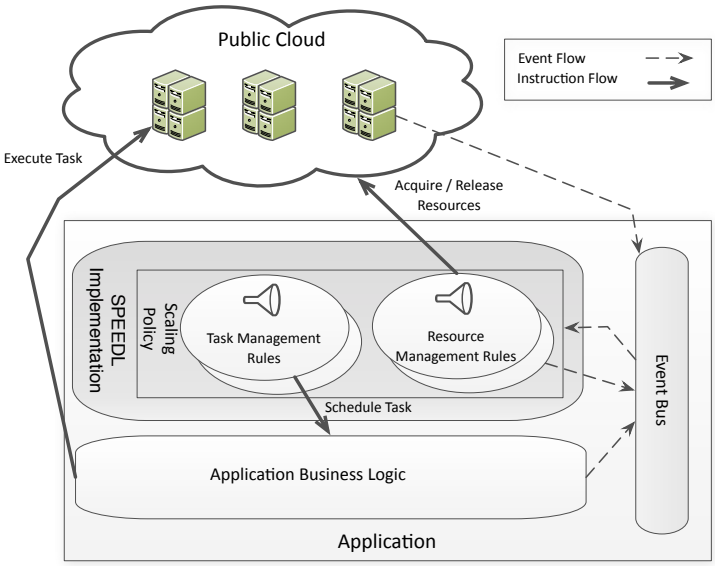


Figure 6.5: Using SPEEDL for elasticity control

Categorizing rules to scale up, scale down, task scheduling and task migration sets allows defining the behavior of a cloud application clearly and intuitively. Any typical action in the cloud can be assigned to one of these groups or separated on a few components that fall into the presented categories. Defining SPEEDL rules, common patterns and actions observed in the related literature and our previous experience were incorporated. While this covers a broad spectrum of possible scaling scenarios, SPEEDL was designed with extensibility in mind. Therefore, even in unique cases, developers can leverage SPEEDL to achieve the desired application behavior.

6.3.3 Top-Level Language Grammar

We discuss the formal SPEEDL grammar using the Backus Normal Form (BNF). In this section, we focus only on the most important details, while the full grammar is available in Appendix 9.3. The top level of a SPEEDL definition, shown in Grammar 6.1, consists of a rules sequence, followed by the optional `validation` section and the terminal statement (`build`). Rules are split into `Scale Up`, `Scale Down`, `Scheduling` and `Migration` sets.

The `validation` section allows triggering an optional consistency validation of the scaling policy. SPEEDL distinguishes two types of validation: (1)internal rule validation warns about rules that are internally inconsistent (e.g., scaling up validates that the number of hosts to spawn is larger than -1), while (2) external validation checks for inter-rule inconsistencies. Out-of-the-box, SPEEDL currently only supports internal rule validation. External validation logics need to be provided by the developer, if required.

Grammar 6.1: Top-level formal language specification of SPEEDL

⟨*ScalingPolicy*⟩ ::= ⟨*SPConfigElements*⟩

⟨*SPConfigElements*⟩ ::= ⟨*Rule*⟩ ⟨*SPConfigElements*⟩
| ⟨*Validation*⟩ ⟨*SPTerminalStatement*⟩
| ⟨*SPTerminalStatement*⟩

⟨*SPTerminalStatement*⟩ ::= 'build'

⟨*Rule*⟩ ::= ⟨*ScaleUpRule*⟩
| ⟨*ScaleDownRule*⟩
| ⟨*SchedulingRule*⟩
| ⟨*MigrationRule*⟩

In the following, we discuss each group of the rules in more detail and introduce the available out-of-the-box constructs. The focus is on the most central and interesting features of the language and code examples based on the JCloudScale Java implementation of SPEEDL.

6.3.4 Event-Driven Elasticity

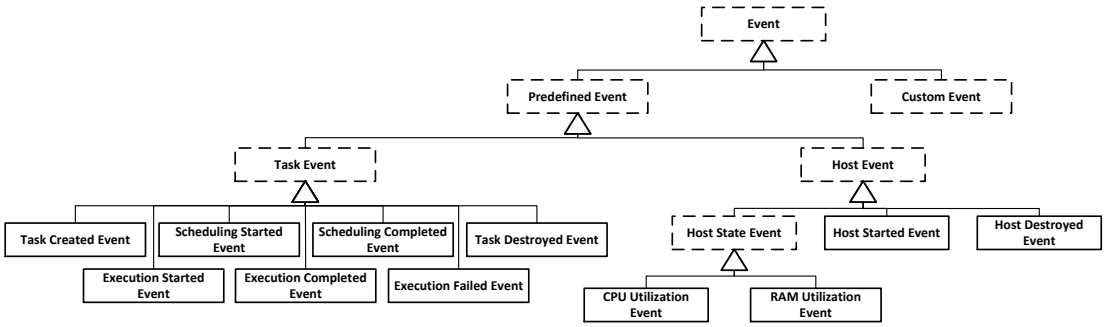


Figure 6.6: Simplified hierarchy of predefined events

Events form the basis of all rules in SPEEDL. Naturally, different systems and implementations make available different predefined events. In the Java implementation of SPEEDL, the events depicted in the event hierarchy in Figure 6.6 are available. SPEEDL predefined events are based on the monitoring event hierarchy from JCloudScale core (see Figure 6.2). However, the core hierarchy was extended and improved in order to provide more flexible and adaptive application monitoring and evolution.

Predefined events are mainly produced and consumed within the framework itself and cover a range of common elasticity-related situations, such as task scheduling or execution, host lifetime and resource usage. Additionally, application developers can implement custom, domain-specific events, which are typically triggered either in custom

rules or directly in the application. In the JSTaaS scenario from Chapter 4, a potential domain-specific event may be the creation of a test suite that is not allowed to be scheduled to a public cloud due to privacy reasons.

6.3.5 Task Management

Task management rules focus on how to map tasks to resources. While these rules may take into account the state of the cloud infrastructure, the only actions that are initiated is that one or more tasks are assigned to exactly one host for execution. Task management rules come in two flavors, the *task scheduling* or the *task migration* rule sets. Task scheduling represents the initial mapping of a new task to a resource, while task migration re-maps an already-existing task. Task migration controls the dispersion and load of each resource by arranging and moving tasks in order to maintain overall system stability.

Grammar 6.2: Formal specification of task scheduling rules

⟨*SchedulingRule*⟩ ::= '`Schedule`' ⟨*ScheduledTaskType*⟩ ⟨*SchedulingHostFilter*⟩
⟨*SelectedSchedulingRule*⟩
| ⟨*customSchedulingRuleImplementation*⟩

⟨*ScheduledTaskType*⟩ ::= '`task`' ⟨*allowedTaskType*⟩
| '`task`' ⟨*allowedTaskPredicate*⟩
| ''

⟨*SchedulingHostFilter*⟩ ::= '`allHosts`'
| '`onRandom`' ⟨*hostCount*⟩
| '`onHosts`' ⟨*hostToBooleanPredicate*⟩
| '`onHosts`' ⟨*hostTaskToBooleanPredicate*⟩
| ''

⟨*SelectedSchedulingRule*⟩ ::= '`greedy`' ⟨*GreedyRule*⟩
| '`balance`' ⟨*BalancingRule*⟩

Task Scheduling Rules

There are two prevailing approaches to distribute tasks in the cloud. (1) Balancing rules [95] aim to evenly distribute tasks over all available hosts, with the ultimate goal of achieving a close-to-uniform distribution of tasks over hosts, while (2) greedy rules [98] aim to saturate a single resource before using the next. Both of these behaviors have merits, and domain- and application knowledge is required to select which of those fundamental strategies is more suitable.

Additionally, each balancing or greedy rule is further shaped by a set of restrictions. Developers can specify a criterion that selects the set of hosts that should be considered. Alternatively, developers can specify which type of tasks this scheduling rule applies

Listing 6.3: Greedy scheduling rule

```
1  Schedule
2          .tasks(MyTestExecutor.class)
3          .onHosts(
4                  (host,task)->
5                          host.getType() == (canRunInCloud(task) ?
6                                                  PUBLIC_CLOUD : PRIVATE_CLOUD))
7          .greedy()
8          .maxTasks(4);
```

to, as well as a maximal amount of concurrent tasks running per host. Finally, specific scoring criteria, comparable to a fitness function in optimization, can be specified for each host or scheduled task. This criterion allows developers to balance tasks depending on application-specific task properties, thus achieving better, domain-specific, scheduling results by exploiting data locality [100] or achieving cost-effectiveness.

The formal definition of a scheduling rule is provided in Grammar 6.2. An illustrative example of a rule that distributes test execution tasks between private and public cloud depending on a custom developer-defined predicate is shown in Listing 6.3 using the syntax of the SPEEDL Java implementation.

Task Migration Rules

In many applications, especially those with long-running tasks (e.g., scientific computing), it may often make sense to re-assign tasks that have already been started to execute on a cloud host. The technical process of task migration is out of scope of SPEEDL. However, SPEEDL provides a set of rules that allow the definition of a migration strategy as part of the scaling policy, if the underlying application is able to suspend and move tasks, as it is the case in JCloudScale. The formal structure and main rule categories are again defined using BNF in Grammar 6.3.

Grammar 6.3: Formal specification of migration rules

⟨*MigrationRule*⟩ ::= ‘`migration`’ ⟨*MigrationType*⟩
| ⟨*customMigrationRuleImplementation*⟩

⟨*MigrationType*⟩ ::= ⟨*MigrationHostFilter*⟩ ‘`integrate`’ ⟨*IntegrationRule*⟩
| ⟨*MigrationHostFilter*⟩ ‘`optimize`’ ⟨*OptimizationRule*⟩

⟨*MigrationHostFilter*⟩ ::= ‘`allHosts`’
| ‘`hosts`’ ⟨*hostToBooleanPredicate*⟩
| ‘’

The process of task migration consists of four distinct phases. At first, situations that require migration need to be detected (*detection phase*). As SPEEDL is based on the notion of ECA rules, this phase is implemented via events. Next, tasks that should be

migrated are selected (*task selection phase*). By default, SPEEDL prefers to migrate tasks that have been started last, but oftentimes an application developer will want to substitute this behavior with application-specific logic. After that, the destination host to which the task should be migrated, needs to be selected (*host selection phase*). By default this is controlled by the same metric as the migration condition (e.g., when high RAM usage is detected, objects are migrated to hosts with the least RAM usage). However, again developers are able to customize this selection strategy or provide their own implementation. Finally, the actual migration needs to be performed (*migration phase*).

An example of an optimization migration rule that allows decreasing the load on the private JSTaaS infrastructure by moving some tasks to the public cloud during working hours is shown in Listing 6.4.

Listing 6.4: Optimizing migration rule

```
1  Migration.
2          .hosts(host -> host.getType() == PRIVATE_CLOUD)
3          .optimize()
4          .withMoreTasks(4)
5          .migrateTo(host -> host.getType() == PUBLIC_CLOUD)
6          .ifViolatedFor(ofMinutes(5))
7          .minActionInterval(ofMinutes(10))
8          .canMigrate(task -> canRunInCloud(task))
9          .arrangeTasks(task -> task.getStartTime(), DESCENDING)
10         .isEnabled(WorkingSchedule.isWorkingTime(now()));
```

6.3.6 Resource Management

Resource management rules provide a mechanism to control and adapt the resources that the application requests from the cloud infrastructure. While rules may take into account CO resource usage, task executions or the application state, the main outcome of all resource management rules is a change in the number and/or types of available resources. This happens primarily through the *scale-up* and *scale-down* rule sets. Industrial PaaS platforms usually take scale up and scale down decisions based on resource usage metrics, e.g., average CPU load. This generic approach is also supported by SPEEDL. However, resource-based scalability is reactive, cumbersome to write and hard to tweak [150], as all decisions have to be based on the current resource usage. Therefore, SPEEDL provides additionally an alternative approach that allows taking resource management decisions based on application-specific events and conditions [48]. This allows adapting cloud resource usage in advance, e.g., based on domain-specific predictions of future load. For example, in the JSTaaS motivating scenario, application developers often know in advance when a large batch of unit tests is going to appear, based on their development schedule.

Scale-Up Rules

Scaling up is usually controlled via one or more application-dependent metrics (e.g., CPU/RAM usage, task throughput, predictions of future load). The behavior of all those rules is similar – if a metric threshold is exceeded, a scale-up action is executed. Hence, we created a single configurable behavior policy that accepts a controlled metric and additional configuration that allows defining the actual action, e.g., how many and which resources to start. The event-based nature of SPEEDL gives us the ability to flexibly adjust thresholds and actions, depending on an application's needs. Further, by leveraging CEP, developers have access to powerful means of data aggregation and analysis when defining metrics. However, in addition to these metric-threshold based rules, SPEEDL also contains other rules for scale-up. For long-running applications, SPEEDL also provides time-based scale-up rules. These rules do not trigger based on changes in the actual or predicted load, but ensure that a proper amount of hosts is running at specified points in time. This model is suitable for applications with well-known periods of high usage. A formal definition of SPEEDL scale-up rules is given in Grammar 6.4.

Grammar 6.4: SPEEDL scale up rules specification.

⟨*ScaleUpRule*⟩ ::= '`scale up`' ⟨*ScaleUpHostFilter*⟩ ⟨*SelectedScaleUpRule*⟩
| ⟨*customScaleUpRuleImplementation*⟩

⟨*ScaleUpHostFilter*⟩ ::= '`allHosts`'
| '`hosts`' ⟨*hostToBooleanPredicate*⟩
| ''

⟨*SelectedScaleUpRule*⟩ ::= ⟨*CPUBasedScaleUpRule*⟩
| ⟨*RAMBasedScaleUpRule*⟩
| ⟨*TaskCountScaleUpRule*⟩
| ⟨*TimeBasedScaleUpRule*⟩
| ⟨*TaskQueueLengthScaleUpRule*⟩
| ⟨*CustomMetricScaleUpRule*⟩

A sample scale-up rule that scales from 1 to 20 cloud hosts when we have more scheduled test suites over the next hour than we have processing resources, is shown in Listing 6.5.

Scale-Down Rules

While scaling up is often based on a current or predicted load, scaling down in contemporary IaaS cloud systems should be aligned with the BTU of the cloud provider. In IaaS cloud systems, computing resources are typically billed periodically (e.g., hourly in Amazon EC2[4], per minute after the first 10 minutes in Google[5]). Economically, it

[4] `http://aws.amazon.com/ec2/`
[5] `https://cloud.google.com/compute/pricing`

Listing 6.5: A scale-up rule based on a domain-specific metric

```
1  ScaleUp
2          .hosts(host ->
3                  host.getType() == PUBLIC_CLOUD)
4          .when(hosts ->
5                  countTaskCapacity(hosts) < TestsSchedule
6                                  .scheduled(now(), ofHours(1)))
7          .checkEvery(ofMinutes(5))
8          .minHosts(1)
9          .maxHosts(20)
10         .scaleUpStep(1)
11         .newHostType("PUBLIC_CLOUD", "m1.small")
12         .minScaleUpInterval(ofMinutes(10));
```

makes little sense to release a resource while it is still paid for. Hence, the evaluation whether resources should be scaled down or not in SPEEDL is triggered briefly before the resource would enter the next billing period. A second peculiarity of scaling down is that it often needs to integrate with migration (see Section 6.3.5) in order to move tasks still scheduled to a host that is about to be scaled down. Aside from those aspects, scaling down is conceptually similar to scaling up. A formal definition of scale-down rules in BNF is presented in Grammar 6.5.

Grammar 6.5: SPEEDL scale down rules specification.

⟨*ScaleDownRule*⟩ ::= 'scale down' ⟨*SelectedScaleDownRule*⟩
| ⟨*customScaleDownRuleImplementation*⟩

⟨*SelectedScaleDownRule*⟩ ::= ⟨*CPUBasedScaleDownRule*⟩
| ⟨*RAMBasedScaleDownRule*⟩
| ⟨*TaskCountBasedScaleDownRule*⟩
| ⟨*TaskQueueLengthScaleDownRule*⟩
| ⟨*HostIdleTimeScaleDownRule*⟩
| ⟨*TimeBasedScaleDownRule*⟩
| ⟨*CustomMetricScaleDownRule*⟩

An example of a scale-down rule that releases cloud resources when no longer needed during public holidays is shown in Listing 6.6.

Hosts that are currently running tasks may also be scaled down. In some cases, it is safe to restart the aborted task on another host. This is a common assumption in many state-of-the-art PaaS platforms, which primarily deal with HTTP requests as tasks. However, this is not always the case. Sometimes, tasks cannot be aborted due to high startup costs, or the possibility of introducing state inconsistencies. In such cases, the host either has to be left running until the tasks are finished or, if this is possible, the tasks have to be migrated to another host. In SPEEDL this is controlled by the `ifWithTasks` condition. It defines whether tasks can be discarded, left running

Listing 6.6: A scale-down rule based on task count

```
1  ScaleDown.runningTasks(0)
2                  .checkAdvance(ofMinutes(1))
3                  .minHosts(
4                          host -> host.getType() == PUBLIC_CLOUD, 1)
5                  .isEnabled(
6                          host -> host.getType() == PUBLIC_CLOUD &&
7                                  WorkingSchedule.isHoliday(now()))
```

or migrated to another host. In more sophisticated cases, developers can perform any custom actions with a particular host (including task migration or abortion) within the custom `isEnabled` predicate that allows determining if particular scale down rule is applicable to this host. As an example, such custom predicate is used in Listing 6.6 to release only hosts from a public cloud during official holidays.

6.3.7 Summary

The rapid elasticity of cloud applications is an essential characteristic of cloud computing [25]. However, development of an effective and self-adjusting scaling behavior is connected with the significant amount of such purely programming challenges as distributed communication, event correlation, and thread-safe development. In order to address these challenges and provide a useful tool for a comfortable scaling policy definition, the SPEEDL language was developed. The domain-specific declarative language SPEEDL simplifies defining advanced task and resource management policies for IaaS cloud applications. Contrary to existing approaches, SPEEDL is aiming to provide cloud management abilities as the part of the cloud application rather than via an external system, thus allowing developers to incorporate domain-specific information and flexible application design. SPEEDL categorizes typical scaling behaviors into four distinctive sets (i.e., `Scale Up`, `Scale Down`, `Scheduling`, and `Migration`) and provides a collection of typical customizable algorithms within each of the sets.

CHAPTER 7

Profiling-Based Task Scheduling and Execution

While the previous chapters introduced the core and elasticity components of JCloudScale, this chapter presents the automatic task scheduling and execution management approach that improves resource consumption and task management within a single cloud host.

Different types of applications require different approaches to task scheduling and monitoring. The scheduling approach proposed in this chapter is most useful for factory-worker applications (also known as the producer-consumer pattern, and strongly related though not identical to the master-slave pattern [23]). In factory-worker, a single host or a set of hosts (named "factory" hosts) create tasks while a (typically large) number of worker hosts processes them. This architectural pattern is commonly used in situations where the system has to process a set of tasks generated from user requests or by splitting the bigger problem into smaller chunks. Applications designed this way often achieve high scalability and performance while keeping interaction code simple. These advantages make the factory-worker architectural pattern a common choice for applications that run in a distributed environment or the cloud. Another distinctive feature of factory-worker systems is that the set of possible tasks is usually homogeneous or limited. This allows predicting future resource usage based on previously gathered profiling data.

7.1 Resource-Aware Task Scheduler

Usually, tasks executed within the cloud do not use resources (e.g., memory, CPU or network bandwidth) uniformly. Instead, over the task run, resource usage varies, causing usage peaks and valleys. In order to achieve effective and predictable execution times, developers have to reserve resources considering the maximum expected usage [12]. This causes resource over-provisioning for, often significant, parts of the task execution time.

If multiple tasks are to be processed on the same machine, resource over-provisioning is even higher, as developers have to reserve resources accordingly to the worst-case scenario, when resource usage peaks overlap. For instance, to process multiple tasks in the cloud, with a 1GB peak memory usage each, developers have to either use hosts with 1GB of RAM, and execute tasks sequentially on each host, or reserve machines with more memory, thus allowing parallel task execution. However, if this memory usage peak takes only a short period of the task processing time (e.g., during data serialization), while remaining memory usage is much lower, all reserved memory for that peak demand is wasted most of the time, as it is shown in Figure 7.1.

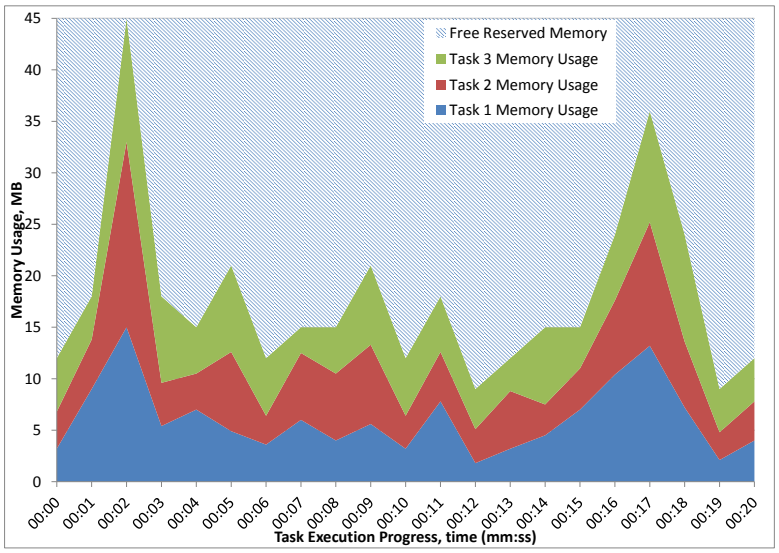


Figure 7.1: Host memory usage in case of memory peaks overlapping

The task scheduling approach presented in this chapter allows effective resource usage for uniform tasks based on profiling data. In order to achieve this, a scheduler that monitors task execution and constantly improves future resource usage estimations for each used host was developed. These predictions allow the effective scheduling of subsequent tasks, thus improving elastic system behavior in the cloud and optimizing resource usage. Additionally, the discussed scheduler aims to avoid overlapping peak resource usages of tasks, hence allowing to run more tasks in parallel on the same virtual machine.

7.2 JSTaaS as a Factory-Worker Application

JSTaaS can be partially considered as such factory-worker application. At its core, JSTaaS collects the tests that need to be executed and schedules them over the available computation resources. Thus, its behavior is similar to factory-worker pattern. Even though it is hard to predict and categorize resource usage of a set of abstract tests, it is

Table 7.1: Resource types summary

Resource Name	**Measuring Units**	**Resource Type**	**Description**
CPU	operations per second	Competitive	Specifies application execution speed within existing environment.
Memory	bytes	Cumulative	Specifies amount of used memory within machine.
Network Traffic Usage	bytes	Cumulative	Specifies amount of data transferred over the network.
Network Bandwidth	bytes per second	Competitive	Specifies current throughput of the network.
Storage	bytes	Cumulative	Specifies amount of occupied storage within available disk space.
Storage Read-/Write	bytes per second	Competitive	Specifies speed of read/write operations of the storage.
Database Read-/Write	transactions per second	Competitive	Specifies amount of successful transactions between system and database.

fair to assume that resource usage is not uniform over the test execution. There may be some memory spikes for test configuring or result collecting, test executions may include busy-waitings or sleeping and network data may be retrieved or sent during different tested activities. Additionally, a significant amount of tests do a similar job (i.e., invoking some short-running activity and analyzing result), thus their resource usage patterns should be similar. Even if this similarity is insufficient to obtain an accurate and usable resource usage history, JSTaaS developers can generate a distinctive profile for each test. As all tests are executed periodically, resource usage of each test usually stays the same, while the changes caused by tested code modifications can be handled by error correction algorithms, presented below.

7.3 Resource Types and Control Limitations

At first, before diving into the details of our approach, it is necessary to discuss the nature of different computational resources and possible ways to control their usage. On the highest level of abstraction, cloud-provided computational resources (see Table 7.1) in the following text are divided into two classes: *competitive* and *cumulative*.

On the one hand, *cumulative* resources can be profiled and predicted relatively easily and confidently. For example, if memory allocation is required and we are handling multiple tasks in parallel, we can assume that the total amount of used memory is the sum of each task usage. However, it is not trivial to reduce cumulative resource usage at a specific point of time. For example, when some task will need more memory than available in the system, we cannot reduce memory usage of other tasks, therefore, we have to suspend our task until the total memory usage decreases. Additionally, whenever we suspend the execution of a task, the usage of cumulative resources remains constant (i.e., the usage of cumulative resources does not decrease when suspending a task), limiting the effectiveness of task suspension for such resources.

On the other hand, *competitive* resources are easy to manage with task suspension

and resuming. For example, when we are approaching some timing-critical CPU-intense computation stage of one task, we can suspend other tasks on the same host and thus ensure that all computational resources are allocated to the critical task. However, competitive resources usage profiling is not as predictable as for cumulative resources. For example, when two tasks are competing over the CPU of a virtual machine, their execution time is hardly predictable because of concurrency issues. This can be somehow managed by task or thread priorities, but mainly concurrency performance depends on operating system implementation and state.

Additionally, we need to keep in mind that resource usage on application level is not entirely predictable in practice. For instance, some requests or request sequences can significantly influence the overall usage and productivity of some resource. For example, database queries of different complexity can take different amounts of time. If concurrent requests work with completely distinctive parts of a database, an execution may significantly slowdown due to the frequent cache misses. Such behavior is hard to predict during profiling, therefore, our approach relies on a profiling error correction system, which improves prediction accuracy over time.

7.4 Approach Overview

The system we present in this chapter provides a fine-grained scalability and adaptability of an application as decisions are based on the actual application behavior and current activities, instead of general resource usage trends as utilized in related approaches [151, 152, 153, 154]. This is achieved by using a specifically designed distributed profiling solution that allows collecting runtime information from the distributed application in the key points of task execution. Whenever an application schedules a task, it invokes the discussed task scheduling system. The global overview of the task scheduling and execution process is presented in Figure 7.2.

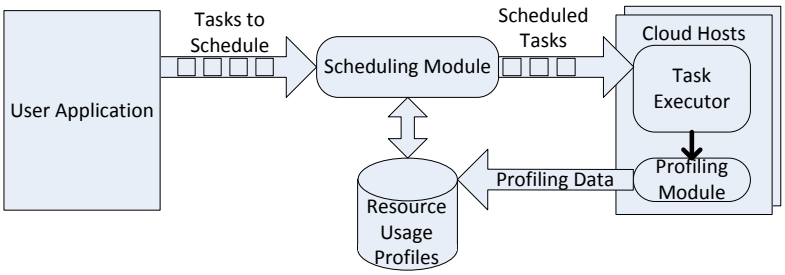


Figure 7.2: Overview of the profiling-based scheduling approach

To have current and accurate information on application behavior from each used worker (CH), an application is monitored via a special *Profiling Module*. The profiler is running within an application on each used CH, profiles the resource consumption of a *Task Executor*, and collects the information necessary for scaling decisions. Currently, this information includes memory usage by objects related to the tasks executed on this

CH and CPU usage of the machine. Extension to other types of resources is part of our ongoing work.

Collected information is matched to the tasks executed on this node and their progress, either via push notifications sent by the task that is being executed, or after each task is finished. The Profiling Module is mainly responsible for information collection, but also handles starting and suspending tasks, in order to prevent overlapping and to avoid peak load aggregation. Collected profiling data is accumulated in a *Resource Usage Profiles* storage. All profile data processing and the creation of task schedules is happening in a separate *Scheduling Module*, which is a conceptually independent component that processes information collected from *Profiling Modules*. The Scheduling Module is a central planner for the approach, and can be easily deployed to a separate cloud host to not interfere with an application or workers performance.

7.5 Resource Profiling

As described above, the Scheduling Module relies on resource usage information obtained from profiling previous runs of similar or identical tasks. During each task execution, Profiling Modules are collecting resource usage traces and periodically send this information to Scheduling Module. For a profiled task τ we measure the current usage or the usage delta (u_i) of each resource ($\forall \rho \in P$). Therefore, the task execution trace (U) is a mapping of resource usage measurements to the time when the measurement was performed for each measurement point $0..n$, as shown in Equation 7.1:

$$U \equiv \forall \rho \in P, \forall i \in [0..n] :< t_i, u_i > \tag{7.1}$$

This trace information, for each resource, can be visually represented as in Figure 7.3 (the figure exemplifies a trace for memory usage).

After receiving multiple of these task execution traces, the Scheduling Module can build an estimated *Aggregated Resource Profile* ($I_{\tau,\rho} \equiv t \in [t_0; t_x], (\iota_{\tau,\rho,t_0} \dots \iota_{\tau,\rho,t_x})$) by averaging collected traces (U). After x executions (e) of a task of type τ, we hence end up with x traces for each resource ρ, which we describe via a set $U_{\tau,\rho} = \{U_1, U_2, \dots, U_x\}$. Each resource usage trace may be not completely accurate and may represent only partial information or may provide distorted data due to external system activity, network problems, or other unpredictable events. To improve the predicted profile, all separate traces must be compared and analyzed to minimize side-effects and minimize statistical errors.

For example, for a memory execution profile ($\rho =$ memory), a statistically plausible way to build the average is to take the arithmetic mean of all traces for each point in time t, as shown in Equation 7.2. As the calculation of each point is generally independent, the algorithm does not have to wait for the whole trace to become available (e.g., wait for all executions to finish). Instead, the Aggregated Resource Profile can be calculated and further improved with each new measurement point of next task executions ($u_i \in U_{\tau,\rho}$), if the measured value diverges from the predicted one more than a configured error rate (ξ). The graphical representation of the trace averaging is shown in Figure 7.4.

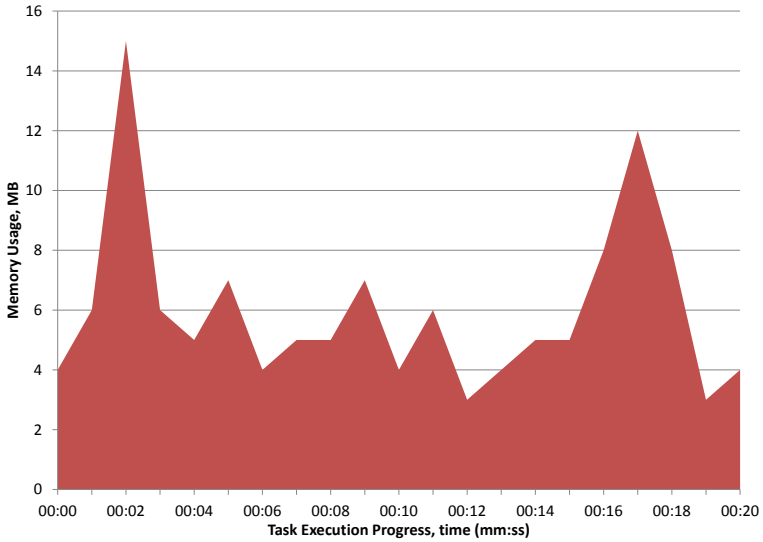


Figure 7.3: The measured memory usage profile for a specific task execution

$$I_{\tau,\rho} \equiv \forall t \in [t_0; t_x] : \iota_{\tau,\rho,t} = \frac{\sum_{U=U_1}^{U_x} u_t}{x} \tag{7.2}$$

This Aggregated Resource Profile allows predicting the future load for new tasks of type τ. Therefore, if the scheduling infrastructure knows the current execution point of each task on a specific worker in the system, the Scheduling Module is able to estimate the future load of each worker and to adapt the task execution schedule to fit the required resource usage limitations within each host.

7.5.1 Resource Profiling Modes

From an implementation point of view, profiling can be performed in *active* or *passive mode*. In *passive mode*, profiling is happening seamlessly to the profiled application. The Profiling Module is configured to perform resource measurements in fixed intervals and has no knowledge of the profiled task execution state. This approach gives more freedom to profiled application developers as it does not require any awareness of task profiling. However, it does not provide fine-grained profile information, and may miss some resource usage spikes or misinterpret an application profile because of the interpolation of periodical measurements (see Figure 7.5). For example, if a task is periodically acquiring and releasing memory, profiling may provide some random memory usage curve because measurements happened on different stages of the periodic process. While an aggregated task profile should improve after multiple task executions, the passive mode is still more appropriate for resources that do not exhibit significant short-term spikes and profiling long-running tasks that only gradually vary their resource usage.

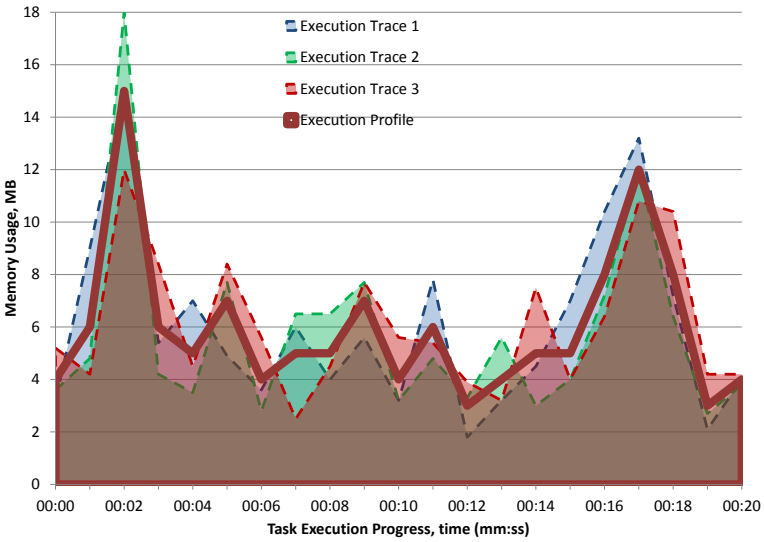


Figure 7.4: Averaging of measured memory usages to obtain aggregated memory usage profile

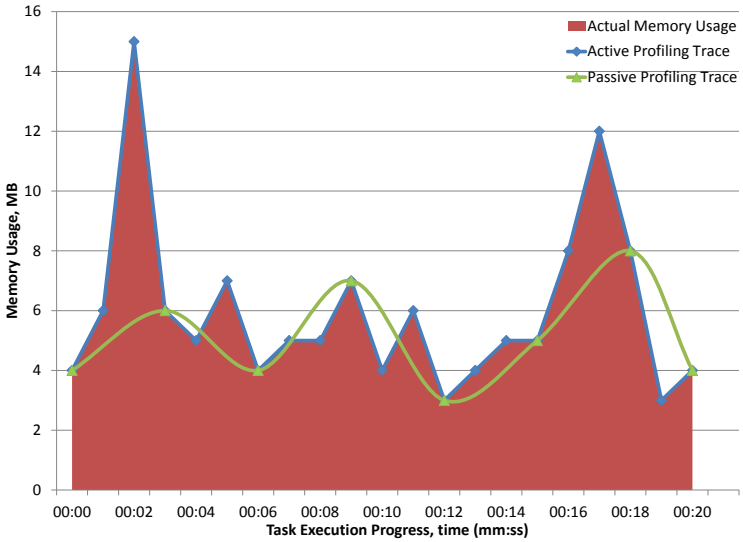


Figure 7.5: Comparison of active and passive profiling technique on highly dispersing task execution

In situations when passive profiling is not appropriate or shows insufficient results, the *active profiling mode* should be used. In this mode, the profiled task is actively triggering the Profiling Module to measure resource usage at crucial execution points. This allows obtaining a context-aware resource usage profile that exposes actual task behavior, leading to more confident and reliable scheduling actions. However, this approach has a

bigger impact on application performance, requires full awareness of developers and often needs some amount of iterations to achieve the required granularity. It is preferable for short-running tasks or applications with short resource usage spikes that can be missed in passive profiling mode.

Another important benefit of active profiling is that it allows the Profiling Module to suspend task execution on profiling points. This opens up additional scheduling possibilities for the Scheduling Module, allowing for better resource usage results. In case of passive profiling, task executions, once started, cannot be suspended in our system. However, when active profiling is used, the Profiling Module can pause some tasks to wait for a specific execution point of other tasks, therefore achieving better resource usage at the minimal cost of overall application execution speed.

7.6 Task Scheduling

Whenever the Profiling Module on any worker reports host resource usage (see Figure 7.6), it also includes information about the execution progress of each currently running task. This allows to scale and align executions of multiple tasks from multiple machines to one Aggregated Resource Profile for each distinctive task type existing in the profiled application (see Figure 7.4). In addition, this profile is further refined to correspond to new measurements, thus improving the overall quality of prediction and adapting the Aggregated Resource Profile if resource usage changes gradually over time.

Based on the currently available profiling data and the task execution state of each worker, the Scheduling Module can construct resource usage predictions for each worker. These predictions play a key role in the process of scheduling new tasks. Every time new tasks need to be scheduled, the Scheduling Module constructs the current prediction for each worker and tries to schedule each new task to start as soon as possible. Generally, the scheduling problem is isomorphic to the well-studied bin-packing problem, which is known to be an NP-hard problem [155]. Hence, our scheduling approach is currently based on a heuristic greedy algorithm. We will consider other implementations, for instance, based on evolutionary algorithms [156] in our future research.

In order to formally define our scheduling goal, we need to define some additional preliminaries. The algorithm schedules instances of different types of tasks ($\tau_n \in T_i$), where each type of task T_i has known or previously measured expected resource usage profiles for each profiled resource $\rho \in P$ ($\forall \rho \in P : I_\rho$), as explained in Section 7.5. Additionally, each type of the tasks has an expected duration (t_{T_i}), which is the minimal constraint for each task instance execution time ($t_{\tau_i} \geq t_{T_i}$). The expected task execution time (t_{τ_i}) is determined during the scheduling process and is caused by delays because of inter-task competition over computational resources or deliberate task suspension. Each profiled resource ρ_x on each cloud host $h_i \in H$ has a usage limitation ρ'_x, after which either task execution slows down due to competition with concurrent tasks if resource is competitive (e.g., if multiple tasks are competing for a single CPU), or an application runs out of available resources and crashes if resource is cumulative (e.g., with an `OutOfMemoryError` for Java applications). Additionally, if some obtained

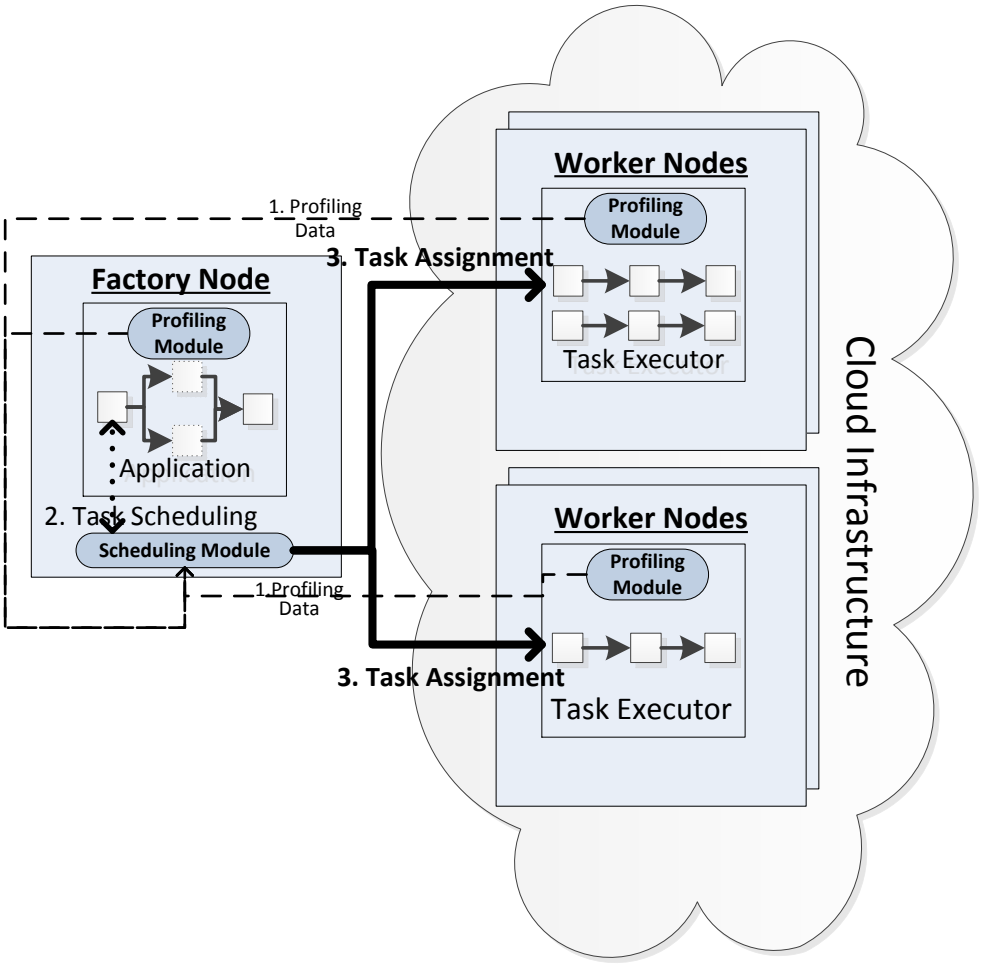


Figure 7.6: Architecture overview of the profiler-based scaling

cloud resource is not utilized above some boundary value ρ_{b_x}, the corresponding host is assigned a penalty $p(\rho_x, \rho_{b_x})$. Based on this formal model, our scheduling approach aims to minimize the total task processing time after the initial startup time t_0, while keeping resource wasting (as captured via penalties) as low as possible (see Equation 7.3). Effective resource usage and task execution time often represent conflicting choices, hence, application developers can additionally specify which of these criteria is more important via the coefficients (A, B).

$$S = A \sum_{h_i \in H} \left(\sum_{\rho_x \in P} (p_{h_i}(\rho_x, \rho_{b_x})) \right) + B(max(t_n \in T) - t_0) \rightarrow min \tag{7.3}$$

The greedy scheduling approach is shown in Figure 7.7. First, the Scheduling Module sorts all new tasks in correspondence to their deadline (sooner first) and expected resource usage (larger tasks first). This allows to schedule and run more prioritative and

demanding tasks sooner while there are more scheduling options available (less tasks currently running or scheduled). After this ordering, for each new task, an appropriate host and starting time is selected. To do this, the Scheduling Module tries to determine how soon the current task can be started on each available host, while satisfying all defined resource usage constraints. This is done by including the current task into the execution plan of the host and detecting if any resource constraint is violated. If this is the case, the Scheduling Module tries to postpone the task further by moving only the first point when the resource constraint was hit. If the required delay for this point is found, a new schedule is calculated using the newly shifted startup time. Note that the scheduling algorithm cannot use the current time as task startup time, as it has to postpone the current task for at least the amount of time required to transfer the task over the network to the worker and start the execution there. This time is a parameter of our approach, and can be either measured on startup (e.g., via the round-trip of a packet of appropriate size plus task initialization time), detected by observing previous scheduling results, or preconfigured by an application developer. Additionally, if the task arrives at the host later than it was scheduled, the task is returned back to the Scheduling Module for re-scheduling. After a worker and task startup time is defined by the heuristic, the task is sent directly to the worker for execution to not miss the scheduled start time.

One additional scheduling technique available to the Scheduling Module if active profiling is used, is task suspension. The task that is being scheduled can be suspended at developer-defined points of execution (checkpoints) to allow other tasks to pass their resource usage pikes, therefore allowing to fit task execution within the resource constraint even in situations when a non-suspending Scheduling Module would need to postpone the task startup time after a resource usage peak.

On each task checkpoint, the Scheduling Module determines whether the following task execution can violate resource usage constraint or not. In case it does, the Scheduling Module pauses the task execution and starts awaiting the moment when the task execution can be resumed without resource usage constraint violations. In order to minimize the impact of task suspensions on the task execution time, the Scheduling Module actively manages a state of each task whenever they reach the next checkpoint. On each checkpoint, the Scheduling Module analyzes the execution profile of the current task few steps ahead and decides how the upcoming execution of the task influences the overall resource usage. Whenever the task is going to increase resource usage, it is suspended (unless this is the last task to execute). In case the resource usage decreases, the Scheduling Module decides if any of the suspended tasks can be resumed, preferring the ones that were suspended earlier and have higher resource usage upcoming.

7.7 Summary

While cloud computing brings the ability to acquire and release resources according to application needs, common resource usage patterns may lead to resource over-provisioning and wastage. In this chapter, the focus was on a specific subset of cloud computing

applications that consist of known sets of uniform tasks with non-uniform resource usage patterns. In order to optimize resource usage of each used cloud host, a task scheduling and execution approach was developed. This approach is based on task execution profiles and resource usage restrictions defined by application developers. The presented task scheduling and execution management approaches allow concurrent task execution with resource usage constraints in applications that support task suspension or not.

In the original work [24], the profile-based task scheduling approach is presented and evaluated in more details. The evaluation results indicate that the approach allows controlling the resource usage, while not influencing drastically the overall performance of an application. In case of the evaluation application, presented approach managed to cut 33% memory usage while adding only 1% of execution time overhead.

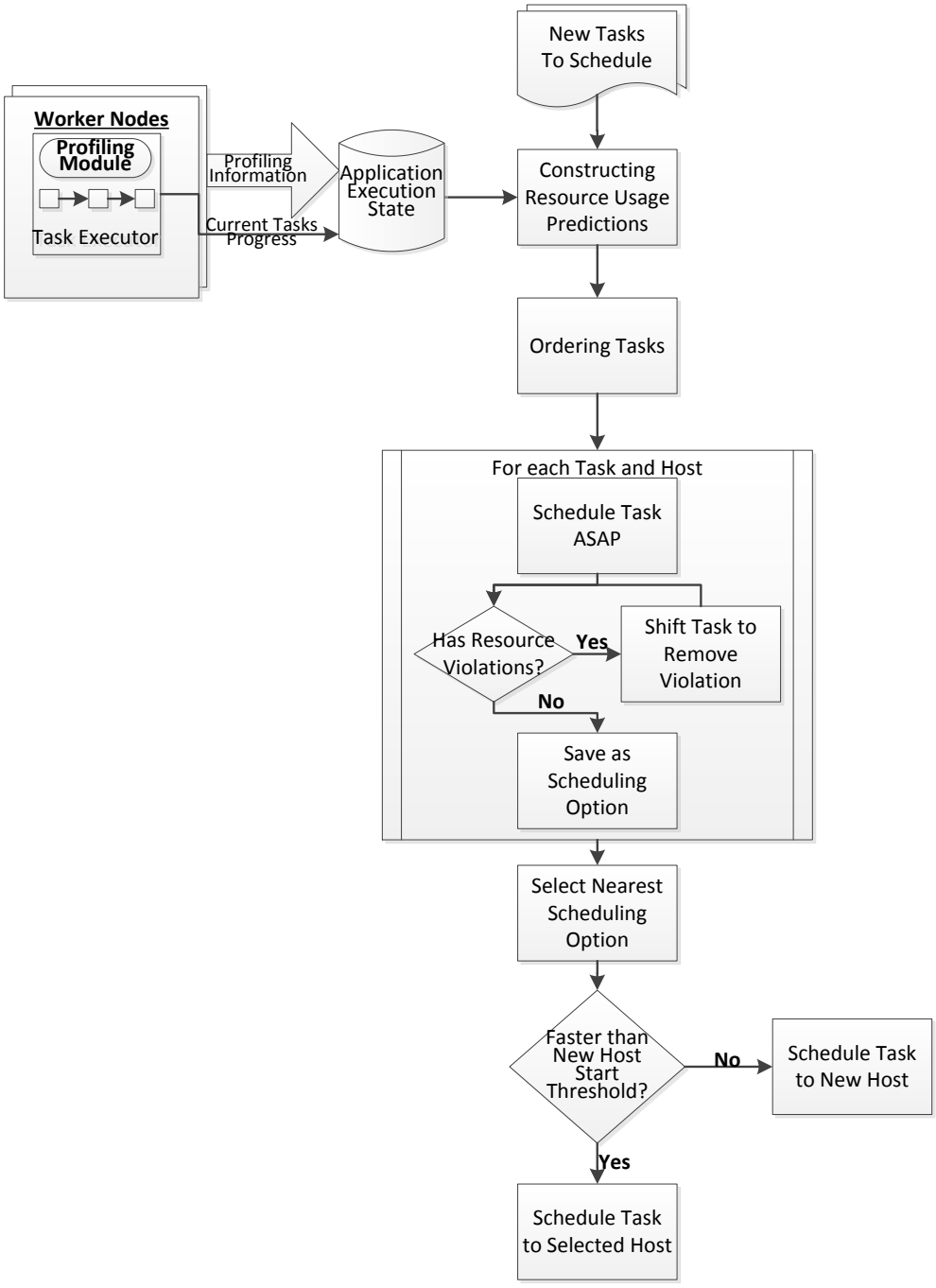


Figure 7.7: Overview over the task scheduling heuristic

CHAPTER 8

Evaluation

In this chapter, the contributions presented in this thesis are evaluated. Addressing the research questions discussed in Section 1.3, a user study and performance evaluations were performed. The results show that the developed JCloudScale middleware and its extensions solve the problems addressed by this thesis.

8.1 Evaluation Setup

The evaluation of contributions presented in this thesis consists of two parts. Section 8.2 starts with comparison of essential cloud application development features provided by JCloudScale and popular IaaS and PaaS platforms. This is followed by a comprehensive user study that validates these claims.

After that, Section 8.3 provides a performance evaluation of JCloudScale. The main question addressed in this section is the overhead that developers experience using JCloudScale. The performance of JCloudScale-based applications is compared to the performance of cloud-native equivalent IaaS applications.

Finally, the evaluation is concluded with a discussion about open issues and an analysis of possible validity threats.

8.2 Usability and Usefulness Evaluation

JCloudScale is designed to simplify the task and boost productivity of the higher-level developers. In order to verify how JCloudScale is achieving its goals, one can observe over time how popular it is, how real developers tend to use it and what is their opinion about the discussed software. While this may be a reasonable approach for advanced industrial solutions, research prototypes can not follow this methodology due to numerous reasons. The main reason for this is that research prototypes mainly assess the quality of the presented ideas or discovered principles, rather than the completeness and quality of

the solution in general. Therefore, research prototypes hardly ever provide the level of quality, performance, advertisement and support that users are expecting from a popular middleware.

An alternative evaluation approach that provides the desired answers while allowing focusing on the noteworthy parts of the developed functionalities are usability and usefulness evaluations.

Usability evaluations usually focus on the theoretical ability of developers to achieve their goal using presented technology [157]. This approach allows verifying how much time developers spent solving the stated task and how complete and effective their solution was. Usefulness evaluations focus on developers' perception of the presented tool or the middleware [157]. Usefulness evaluations allow assessing how effective or constraint developer feel while using the evaluated technology, how eager developer will be to use it again or to recommend it to others.

In this section the usability and usefulness of JCloudScale middleware and its separate components are validated. At first, we quickly recap the core difference between JCloudScale and state-of-the-art IaaS and PaaS solutions, followed by the user study discussion and analysis.

8.2.1 Comparison with Other Platforms

In the following, we briefly compare application development using JCloudScale with building an IaaS application directly on top of Amazon EC2 (without specific tooling except for the EC2 API) and using a PaaS service, such as Amazon Elastic Beanstalk (AEB) or Google AppEngine. The main goal here is to show what advantages an in-between solution such as JCloudScale has.

Starting with API complexity, JCloudScale requires knowledge of a reasonably small amount of API functions, while offering large capabilities for application development. This is mainly caused by the way how applications are built on top of JCloudScale and the amount of necessary changes to the target application. While both, EC2 and AEB assume developers to develop a new application for this platform specifically, and based on the provided APIs, JCloudScale aims at seamless development and ease of bringing existing distributed applications to the cloud. In addition to that, JCloudScale provides specific tools and methodologies for cloud application debugging, which are missing for EC2 or AEB. This is mostly provided by the special local execution environment of JCloudScale, which scales applications in a sandbox on a local machine, while developers of applications for EC2 or AEB can only debug application while the target platform is available and only through a limited set of tools available for the selected platform.

A core advantage of any IaaS approach is that it provides freedom regarding supported frameworks and application architecture designs. JCloudScale, on the other hand, is by its nature restricted to the Java programming language. Other than that, the restrictions imposed by JCloudScale are minimal. AEB, on the other hand, induces quite significant limitations on application design, and restricts the application developer significantly, both with regard to what API functions can be used and what architecture

Table 8.1: Feature comparison of JCloudScale and alternative IaaS and PaaS solutions

Feature	**Amazon EC2**	**JCloudScale**	**AEB**
Complexity of API	Small	Small	Significant
Amount of Platform Interaction Code	Significant	Small	Small
Application Debugging Simplicity	Manual/None	Simple	Reasonable
Architecture Limitations	None	Small	Significant
Scaling Configuration Convenience	Manual/None	Good	Basic
Code Distribution and Update	Manual/None	Semi-Automatic	Automatic
Monitoring Features	Manual/None	Advanced	Basic
Backend Server Access	Unrestricted	Unrestricted	None
Hybrid Cloud Support	Manual	Built-in	None
Developer Lock-in	Small	Small	Significant
Programming Language Support	Any	Java	Java, Python, PHP, JavaScript, Ruby, .NET

an application needs to follow. Another thing that the generic IaaS approach is good for is for having full access to the back-end servers, providing developers complete flexibility and control over the resource usage and operating system configuration. JCloudScale aims to hide the complexity of virtual machines and developers can build cloud applications without even controlling virtual machines, however, it does not forbid developers to modify the virtual machine as long as the core components of JCloudScale are still running.

The generic PaaS model has significant advantages as well. One example of such a benefit of PaaS is code distribution and application scalability. While PaaS approaches scale applications mostly automatically, for EC2-based applications, developers have to create their own rules and approaches to achieve elastic application scaling. From this point of view, JCloudScale provides a reasonable alternative. Scalability is achieved by injected code and appears to be seamless to developer, while additional scaling rules can be provided separately. Provided rules leverage the flexible monitoring framework that allows controlling not only basic parameters such as CPU load and memory usage, but also a high-level application-specific metrics.

Finally, JCloudScale offers support for applications that are scaling over multiple clouds (forming so-called "hybrid clouds"), what allows minimizing application operating costs and extends application flexibility beyond the limits of one cloud provider. This model is not supported by AEB or Google AppEngine at all. Using an IaaS service such as EC2, it is possible to implement hybrid clouds, but this requires a significant amount of development and configuration work. In contrast, setting up a hybrid cloud with JCloudScale comes at almost no effort to the developer.

Table 8.1 demonstrates a qualitative summary of the features that were considered as important for application development. Compared to other systems, JCloudScale significantly simplifies the application development process, hides complexity of code distribution and cloud management, while providing convenient and configurable debugging and development experience. Therefore, it is plausible to believe that developers (especially the ones new to cloud computing) will benefit from using JCloudScale and will be able to develop applications and bring them to the cloud faster than with existing

tools.

8.2.2 User Study

In order to verify the claims asserted in Section 8.2.1 and evaluate the usability and usefulness of JCloudScale, a user study with 14 participants was performed in order to assess the developers' experience with JCloudScale as compared to using standard tools. Following the general ideas of action research [158], we aimed at a study methodology that focused on how real developers would actually use our middleware to build two separate, non-trivial cloud applications.

Note that JCloudScale is deliberately not compared to more domain-specific platforms (such as Apache Hadoop, which is a state-of-the-art implementation of the map/reduce idea [159]). JCloudScale aims to be more general with regard to the use cases that it can support, hence, such comparison would necessarily be unfair.

Study Setup and Methodology

The user study was conducted with 14 male master students of computer science at TU Vienna (participants P01 to P14), and based on two different non-trivial implementation tasks. The first task was to develop a parallel computing implementation of a genetic algorithm (**T1**). The second task required the participants to implement a service that executes JUnit test cases on demand (**T2**). Both tasks required solutions that were elastic, i.e., participants needed to demonstrate that their solutions were able to react to changes in load dynamically and automatically by scaling up and down in the cloud. Both **T1** and **T2** required roughly one to two developer weeks of effort (assuming that the respective participant did not have any particular prior experience with the used technologies).

The study ran in two phases. In Phase (1), JCloudScale running on top of OpenStack was compared with programming directly via the OpenStack API, without any specific middleware support. This phase reflected a typical private cloud [8] use case of JCloudScale. In Phase (2), JCloudScale on top of Amazon EC2 was compared with AEB. This reflects a common public cloud usage of the middleware. In both study phases, the participating developers were asked to build solutions for both tasks using JCloudScale and the respective comparison technology, and to compare the developer experience based on quantitative and qualitative factors. The choice of OpenStack and Amazon EC2 was motivated by the fact that those two platforms currently form the most well-known, as well as most widely used, private and public IaaS systems. Especially EC2 has established a quasi-standard in terms of API support, which many other IaaS systems also adhere to. Consequently, AEB was chosen as a PaaS system in order to stay within the same cloud ecosystem, so as to keep results as comparable as possible.

Phase (1) of the study lasted two months. We initially presented JCloudScale and the comparison technologies to the participants, and randomly assigned which of the tools each participant should be using for T1. Participants then had one month of time to submit a working solution to the task along with a short report, after which they

Table 8.2: Relevant background for each participant of the study.

ID	Phase	Java Exp.	Cloud Exp.	JCS/OS	OS	JCS/EC2	AEB
P01	Phase (1)	+	+	T1	T2	–	–
P02	Phase (1)	+	+	T1	T2	–	–
P03	Phase (1)	∼	∼	T2	T1	–	–
P04	Phase (1)	-	-	T1	T2	–	–
P05	Phase (1)	∼	-	T2	T1	–	–
P06	Phase (1)	+	-	T2	T1	–	–
P07	Phase (1)	+	+	T2	T1	–	–
P08	Phase (1)	+	∼	T2	T1	–	–
P09	Phase (1)	+	∼	T2	T1	–	–
P10	Phase (2)	+	+	–	–	T2	T1
P11	Phase (2)	+	∼	–	–	T1	T2
P12	Phase (2)	+	-	–	–	T1	T2
P13	Phase (2)	+	∼	–	–	T2	T1
P14	Phase (2)	+	+	–	–	T2	–

could start working on T2 with the remaining technology. Similar to T1, participants were given one month of time to submit a solution and a short report. Based on the lessons learned from Phase (1), we slightly clarified and improved the task descriptions and gave participants more time (1.5 months per task) for Phase (2). Other than that, Phase (2) was executed identically to Phase (1).

After one month, each participant submitted his solution via mail, and wrote a semi-structured report summarizing his experience. In the second phase, each participant that was using JCloudScale for the first phase was assigned with one of the comparison technologies, and vice versa. All participants had again one month of time to implement and submit T2. This time, we asked not only for a report of the second task, but also for a qualitative comparison of the used technologies across both tasks.

Table 8.2 summarizes the relevant background for each participant of the study. To preserve anonymity, we classify the self-reported background of participants related to their Java or cloud experience into three groups: relevant work experience (+), some experience (∼), or close to no experience (-). The last four columns indicate whether the participant submitted solutions for JCloudScale running on top of OpenStack (JCS/OS), OpenStack directly (OS), JCloudScale running on top of EC2 (JCS/OS), or AEB, as well as which tasks the participant solved.

For the OpenStack-related implementations, we used a private cloud system hosted at TU Vienna. This OpenStack instance consists of 12 dedicated Dell blade servers with 2 Intel Xeon E5620 CPUs (2.4 GHz Quad Cores) each, and 32 GByte RAM, running on OpenStack Folsom (release 2012.2.4). All servers are redundantly connected through 3 GBit switches. For the study, each participant was alloted a quota of up to 8 small cloud instances (1 virtual CPU, and 512 MByte of RAM), which they could use to implement and test their solutions. For the AWS-related implementations, participants were assigned an AWS account with sufficient credit to cover their implementation and testing with no particular limitations. More information regarding the questionnaires and anonymized participant reports is available in an on line appendix [1] to the work

[1] http://www.infosys.tuwien.ac.at/staff/phdschool/rstzab/papers/TOIT14/

where the user study was originally introduced [19].

Comparison of Development Efforts

Table 8.3: Solutions sizes in lines of code.

	Phase (1)					Phase (2)				
	JCS/OS		OS			JCS/EC2		AEB		
	$\tilde{A}$	σ_A	$\tilde{B}$	σ_B	$\tilde{A}-\tilde{B}$	$\tilde{C}$	σ_C	$\tilde{D}$	σ_D	$\tilde{C}-\tilde{D}$
T1										
Business Logics	200	176	552	215	-352	388	152	825	947	-437
Cloud Management	100	112	180	86	-80	163	24	676	742	-513
Other Code	170	157	286	226	-116	1590	1203	897	1127	693
Entire Application	400	416	1050	434	-650	2141	1331	2790	2660	-649
T2										
Business Logics	450	292	375	669	75	800	434	208	280	592
Cloud Management	100	48	250	364	-150	118	745	223	38	-105
Other Code	325	213	300	297	25	140	2240	1290	972	-1150
Entire Application	1025	461	1500	901	-475	1000	3328	2184	968	-1184

In order to perform a quantitative evaluation of developer efforts, we asked participants to report on the size of their solutions (in lines of code, without comments and blank lines). The results are summarized in Table 8.3. $\tilde{A}$ to $\tilde{D}$ represent the median size of solutions, while σ_A to σ_D indicate standard deviations. It can be seen that using JCloudScale generally reduces the total source code size of applications. Most importantly, the size of the entire application was substantially smaller when using JCloudScale than in the comparison cases. Going into the study, we expected JCloudScale to mostly reduce the amount of code necessary for interacting with the cloud. However, our results indicate that using JCloudScale also often reduced the amount of code of the application business logics, as well as assorted other code (e.g., data structures). When investigating these results, we found that participants considered many of the tasks that JCloudScale takes over as "business logics" when building the elastic application on top of OpenStack or AEB. To give one example, many participants counted code related to performance monitoring towards "business logics".

Note that, due to the open nature of our study tasks, the standard deviations are all rather large (i.e., solutions using all technologies varied widely in size). Further, the large difference in T1 sizes (for JCloudScale on top of OpenStack and EC2) between Phase (1) and Phase (2) solutions can be explained by clarifications in the task descriptions. In Phase (1), some formulations in the tasks led to much simpler implementations, while our requirements were formulated much more unambiguously in Phase (2), leading to more complex (and larger) submissions. Hence, we caution the reader to not compare results from Phase (1) with those from Phase (2).

Summarizing, the median JCloudScale solution across both tasks is only a little over 80% of the size in lines of code as the median OpenStack based solution. Furthermore, 7 out of 9 participants reported that their JCloudScale solution is smaller than their OpenStack solution, independent of which task they used which technology for. 1 participant reported that both solutions are about the same size, and for 1 participant

Table 8.4: Development time spent in full hours.

	Phase (1)					Phase (2)				
	JCS/OS		OS			JCS/EC2		AEB		
	$\tilde{A}$	σ_A	$\tilde{B}$	σ_B	$\tilde{A}-\tilde{B}$	$\tilde{C}$	σ_C	$\tilde{D}$	σ_D	$\tilde{C}-\tilde{D}$
T1										
Tool Learning	7	2	12	5	-5	28	18	16	1	12
Coding	4	10	30	17	-26	42	25	54	23	-12
Bug Fixing	7	7	18	12	-11	14	8	20	14	-6
Other Activities	13	9	14	14	-1	5	0	6	6	-1
Entire Application	31	25	76	33	-45	127	25	121	36	6
T2										
Tool Learning	8	6	2	1	6	15	10	23	18	-8
Coding	30	11	25	17	5	36	5	30	14	6
Bug Fixing	10	11	10	7	0	16	16	5	0	11
Other Activities	7	4	11	10	-4	5	0	9	9	-4
Entire Application	62	13	46	17	16	125	40	102	13	23

the outcome of the JCloudScale solution was significantly larger than the solution he built directly on OpenStack.

However, looking at lines of code alone is not sufficient to validate our hypothesis, as it would be possible that the JCloudScale solutions, while being more compact, are also more complicated (and, hence, take longer to implement). That is why we also asked participants to report on the time they spent working on their solutions. The results are compiled in Table 8.4. $\tilde{A}$ to $\tilde{D}$ represent the median time spent, while σ_A to σ_D indicate standard deviations.

The work hours were classified into a number of different activities: initially learning the technology, coding, testing and bug fixing, and other activities (e.g., building OpenStack cloud images). The results indicate that the initial learning curve for JCloudScale is lower than for working with OpenStack directly. However, in comparison with AEB, some participants reported equal or even more complexity of JCloudScale, mainly because less information about JCloudScale is available on Internet. For coding, JCloudScale appeared to be the much faster tool for participants who had at least some prior experience with cloud computing. Generally, for task T2, JCloudScale proved troublesome for some participants. In this task, JCloudScale generally did not improve productivity over either OpenStack or AEB. Further research will be required to analyze why the results between task T1 and T2 vary in this regard.

Summarizing, our results indicate that JCloudScale indeed improves developer efficiency. We also analyzed qualitative feedback by the participants in their reports. Multiple developers have reported that they felt more productive when using JCloudScale. For instance, P01 has stated that *"the coolest thing about* JCloudScale *is the reduction of development effort necessary, to host applications in the cloud (...) [there] are a lot of thing you do not have to care about in detail."* P03 also concluded that using JCloudScale *"went a lot smoother than [using OpenStack directly]"*. P07 also seemed to share this sentiment and stated that *"[After resolving initial problems] the rest of the project was without big problems and I was able to be very productive in coding the solution."* In comparison to AEB, participants indicated that the core idea

behind JCloudScale is easier to grasp for starting cloud developers than the one behind state-of-the-art PaaS systems. For example, P13 indicated that *"the API is easier to understand and more intuitive to use. Also it fits more into a Java-like programming model, instead of the weird request based approach of the Amazon API"*. However, some participants noted that the fact that AEB is based on common technology also appeals to them. For instance, P10 specified that *"[In case of AEB,] Well-known technology is the basis for everything (Tomcat/Servlet)"*. Hence, the participant argued that this allows developers who are already familiar with these platforms to become productive sooner.

Summarizing the qualitative study results, the data suggests that JCloudScale indeed allows for higher developer productivity for task T1. For T2, JCloudScale solutions are indeed more compact, but it took participants longer to implement them. More research is required to substantiate the underlying reasons for this discrepancy.

Comparison of Developer-Perceived Qualities

It should be noted that both metrics reported so far (lines of code and work hours) do not consider whether JCloudScale, AEB and OpenStack solutions differ in quality or features. While this is hard to judge objectively, most participants commented that they think their OpenStack solutions are more basic than their JCloudScale submission. Participants that were comparing JCloudScale with AEB mainly could not clearly state which solution is more advanced. Mainly this was caused by the time constraints and development speed on each platform. Hence, it is possible that the metrics reported here are pessimistic, i.e., that actual savings in lines of code and development time are in fact larger than reported in our study.

In order to analyze the qualitative experience with JCloudScale, we were interested in the participant's subjective evaluation of the used technologies. Hence, we asked them to rate the technologies along a number of dimensions from 1 (very good) to 5 (insufficient). We report on the dimensions "simplicity" (how easy is it to use the tool?), "debugging" (how easy is testing and debugging the application?), "development process" (does the technology imply an awkward development process?), and "stability" (how often do unexpected errors occur?). A summary of our results is shown in Table 8.5. $\tilde{A}$ to $\tilde{D}$ represent the median ratings, while σ_A to σ_D indicate standard deviations.

For T1, participants rated all used technologies similarly, while JCloudScale was appreciated more for T2. However, JCloudScale was rated worse than the comparison technologies in terms of "stability". This is not a surprise, as JCloudScale still is a research prototype in a relatively early development stage. Participants indeed mentioned multiple stability-related issues in their reports (e.g., P10 mentions that *"When deploying many cloud objects to one host there were behaviors which were hard to reason about"*). Further, some technical implementation decisions in JCloudScale were not appreciated by our study participants. To give an example, P11 noted that *"It is confusing in the configuration that the field AMI-ID actually expects the AMI-Name, not the ID"*. In contrast, JCloudScale has been rated slightly better in terms of simplicity and ease-of use, especially for T2. For example, participant P09 claimed that "JCloudScale *is the clear winner in ease of use. If you quickly want to just throw some Objects in the cloud,*

Table 8.5: Subjective participant ratings from 1 (very good) to 5 (insufficient).

	Phase (1)					Phase (2)				
	JCS/OS		OS			JCS/EC2		AEB		
	$\tilde{A}$	σ_A	$\tilde{B}$	σ_B	$\tilde{A} - \tilde{B}$	$\tilde{C}$	σ_C	$\tilde{D}$	σ_D	$\tilde{C} - \tilde{D}$
T1										
Simplicity	3	0.6	3	1.2	0	2	0	2	1.4	0
Debugging	3	1.5	3	1	0	4	0	3.5	0.7	0.5
Development Process	4	1.7	3.5	0.5	0.5	2	1.4	3	0	-1
Stability	2	1.4	2	0.8	0	2	1.4	1.5	0.7	0.5
Overall	3	0.6	3	0.8	0	2	0	2	1.4	0
T2										
Simplicity	2	0.4	3	1.4	-1	2	1.5	3	1.4	-1
Debugging	2	0.7	4	1.4	-2	4	0	4	0	0
Development Process	2	0.6	3	1.4	-1	2	0	2.5	0.7	-0.5
Stability	2	1.5	1	0	1	3	0.5	2.5	0.7	0.5
Overall	2	0.4	3	0	-1	3	0.5	3	1.4	0

it's the clear choice." Similarly, P12 reported *"[*JCloudScale *is] programmer friendly. All procedure is more low level and as a programmer there are more things to tune and adjust."*. In terms of debugging features, all used technologies were not rated overly well. JCloudScale was generally perceived slightly better (arguably due to its local development environment), but realistically speaking, all compared systems are currently deemed too hard to debug if something goes wrong. Finally, in terms of the associated development process, JCloudScale is generally valued highly, with the exception of T1 and JCloudScale on OpenStack. We assume that this is a statistical artifact, as the development process of JCloudScale is judged well in all other cases. Concretely P01 stated that with JCloudScale, *"You are able to get application into the cloud really fast. You are not forced to take care about a lot of cloud-specific issues."*

Independently of the subjective ratings, multiple participants stated that they valued the flexibility that the JCloudScale concept brought over AEB. Particularly, P11 noted that *"[*JCloudScale *provides] more flexibility. The developer can decide when to deploy hosts, on which host an object gets deployed, when to destroy a host, etc"*. Additionally, participants favored the monitoring event engine of JCloudScale for performance tracking over the respective features of the PaaS system. For example, P12 specified as an JCloudScale advantage that *"programmatic usage of different events with a powerful event correlation framework [is] in combination with listeners extremely powerful."*

Concluding our discussion regarding qualitative results, we note that JCloudScale indeed has some way to go before it is ready for industrial usage. The general concepts of the tool are valued by developers, but currently, technical issues and lack of documentation and technical support make it hard for developers to fully appreciate the power of the JCloudScale model. One aspect that needs more work is how developers define the scaling behavior of their application. Both tasks in our study required the participants to define non-trivial scaling policies, e.g., in order to optimally schedule genetic algorithm executions to cloud resources, which most participants felt unable to do with the current API provided by JCloudScale. Overall, in comparison to working

directly on OpenStack, many participants preferred JCloudScale, but compared to a mature PaaS platform, AEB still seems slightly preferable to many. However, it should be noted that JCloudScale still opens up use cases for which using AEB is not an option, for instance for deploying applications in a private or hybrid cloud [78].

8.2.3 SPEEDL Evaluation

Participants of the user study presented above were using only the core functionality of JCloudScale without any extensions. The main reason for this was that the user study was organized to evaluate the core principles behind the JCloudScale middleware. Additionally, the complexity of the stated tasks did not explicitly require any functionalities provided in the JCloudScale extensions. However, during the user study, participants complained about the complexity of scaling policy defining as they had to define rather complex scaling behaviors using plain Java language. This caused a lot of problems with event management, synchronization and scaling policy debugging.

Because of these reasons, solutions developed within the user study are applicable to be used as a baseline for evaluation of SPEEDL scaling language, presented in Section 6.3.

Evaluation Setup

In order to prepare the basis for the SPEEDL evaluation, the code that was responsible for scaling and task distribution was extracted from the anonimized solutions of the user study participants. After that, code formatting was unified and unnecessary elements were removed. The obtained scaling code differed dramatically in size and complexity. The shortest was only 27 Lines of Code (LoC), while the longest one was 177 LoC (median length was 75 LoC). Informal inspection of these code snippets revealed that many study participants indeed struggled with getting the scaling behavior right, and ended up building rather fragile, "hacky" Java solutions (see Listing 8.1 for an example). After detailed analysis of the used algorithms and behaviors, the equivalent code in the SPEEDL DSL was implemented. Both, extracted scaling code and the equivalent SPEEDL scaling policies are available on line[2] within the supportive materials of the original publication [22].

Results and Discussion

Comparing the original Java-based solutions to SPEEDL, the most interesting points are the amount of code necessary to represent the same behavior using both methods and to what extend the out-of-the-box rules of SPEEDL are useful for expressing the scaling behavior that the participants of our study wanted to implement.

In terms of LoC, the SPEEDL representations indeed turned out to be substantially shorter than the equivalent pure Java code (shortest was 8 LoC, longest was 21 LoC, with a median of 11.5 LoC). This is illustrated in Figure 8.1, which depicts the LoC for

[2] http://www.infosys.tuwien.ac.at/staff/phdschool/rstzab/papers/SERVICES15/

Listing 8.1: Snippet from real-life scaling code

```
synchronized (lock)
{
        try {
                // dirty hack to get correct
                // host.getCloudObjectsCount()
                Thread.sleep(1000);
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
        while(selectedHost == null) { ... }
}
```

each scaling behavior next to the size of an equivalent SPEEDL policy. Additionally, it is arguable that the more compact SPEEDL equivalents are also easier to read and comprehend. For instance, it appeared to be possible to replace a complex 90-line multi-method scaling behavior, which included "sleep" statements, nested iterations, and global locking for synchronization, with the SPEEDL policy shown in Listing 8.2.

Listing 8.2: Complete example of a SPEEDL scaling policy

```
SmartPolicy
        .create(Schedule.greedy()
                        .maxTasks(
                          schedulerConfig.getMaxCloudObjects()))
        .add(ScaleUp
                .queueLength(
                  schedulerConfig.getMaxCloudObjects())
                        .maxHosts(schedulerConfig.getMaxNodes())
                        .newHostsType(schedulerConfig.getFlavor()))
        .add(ScaleDown.runningTasks(0))
```

As a second step, the coverage of real-life application developers' needs by the default SPEEDL rules was evaluated. The analysis showed that equivalent versions of 71% of all scaling behaviors in the used dataset could be built using out-of-the-box rules alone. 100% of all behaviors could be represented with a small amount of custom rules. The scaling behaviors that required custom rules are plotted in darker color in Figure 8.1.

Finally, another interesting observation was that 4 of the scaling behaviors in the dataset contained minor errors (29%). 3 scaling behaviors are not correctly synchronized and can potentially fail due to race conditions. Similarly, another code has (based on the intent shown in a code comment) incorrectly defined "if"-conditions, which would lead to unwanted scaling in edge cases. Hence, the usage of the out-of-the-box rules of SPEEDL does not only simplify the definition of scaling behavior, but also reduces the potential for developer errors.

the original scaling policies were received as the result of a long development and testing process, what is clearly visible in the code. For example, the author of *scaling policy 3* adapted an initially complex approach to behave much simpler. Similarly, *scaling*

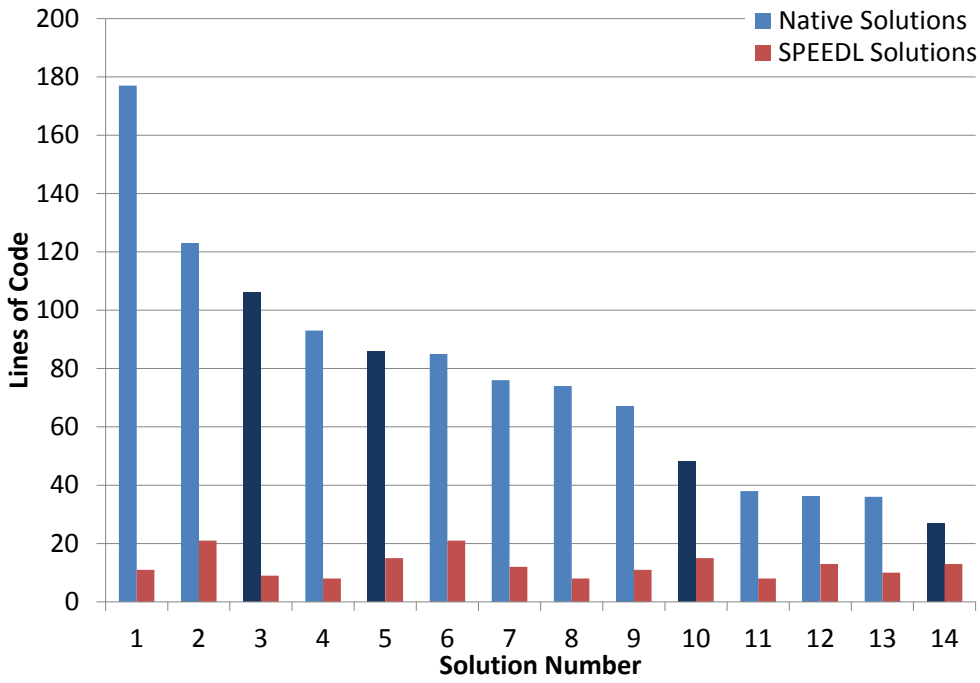


Figure 8.1: Length comparison of evaluated scaling policies

policy 4 and 5 use multiple inefficient sleep-waits to wait for some condition, instead of using an event-based approach.

It is plausible to conclude that SPEEDL indeed provides a noteworthy improvement in scaling policy readability and length. While the existing out-of-the-box rules provided by SPEEDL cannot cover everything a developer would want to express, it was still possible to re-implement 71% of all scaling behaviors using out-of-the-box rules alone. Including custom rules, all scaling behaviors were possible to implement in a much more shorter and precise way. Finally, we have seen that the complications of building real-life scaling behavior can easily lead to hard-to-detect bugs, such as race conditions. Using the out-of-the-box rules of SPEEDL greatly reduces the risk of such bugs.

8.3 Performance Evaluation

Finally, we investigated whether the improved convenience of JCloudScale leads to significantly reduced application performance. Therefore, the main goal of these experiments was to compare the performance of the same application built on top of JCloudScale and using an IaaS platform (OpenStack or EC2) directly.

Experiment Setup

To achieve this, we built a simple sample implementation of the JSTaaS case study application on top of Amazon EC2 and our private OpenStack cloud. For OpenStack, we used the same private cloud as for the user study. For EC2, we deployed our application in the `eu-west-1` region and used instances of size `t1.micro`. Initial experiments with AEB have shown that there is no substantial performance difference between EC2 and AEB. Hence, we omit AEB in our discussions here.

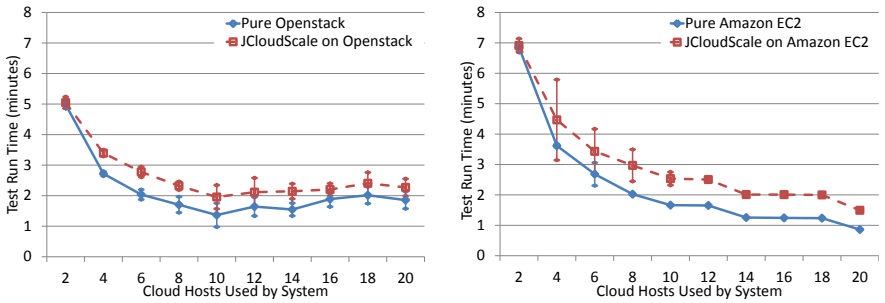
Secondly, we also implemented the same application using JCloudScale. As the main goal was to calculate the overhead introduced by the JCloudScale, we designed both implementations to have the same behavior and reuse as much business logic code as possible. In addition, to simplify our setup, to focus on execution performance evaluation and to avoid major platform-dependent side effects, we limited ourselves to a scenario, where the number of available cloud hosts is static. The source code of both applications is available on line [3].

All four solutions (directly on OpenStack, directly on EC2, and using JCloudScale on both, OpenStack and EC2) follow a simple master-slave pattern: a single node (the master) receives tests through a SOAP-based web service and schedules them over the set of available worker nodes. All solutions were tested with a test setup that consisted of 40 identical parallelizable long-running test suites (each suite execution takes around 30 seconds in our OpenStack cloud environment), scheduled evenly over the set of available cloud machines. Each test suite consisted of a set of dummy JavaScript tests calculating Fibonacci numbers. During the evaluation, we measured the total time of an entire invocation of the service (i.e., how long a test request takes end-to-end, including scheduling, data transmission, result collection, etc.). A single experiment run consisted of 10 identical invocations of the testing web service, each time with a different number of CHs (ranging from 2 to 20 CHs). In all experiment setups, our evaluation shared a physical cloud environment with other tenants, as would be the case in real-life usage. To reduce the performance impact of other tenant's activities, we repeated each experiment 10 times over the course of a day.

Experiment Results

Figure 8.2a and Figure 8.2b show the median total execution time for different numbers of hosts, including error bars indicating standard deviations. In general, both applications show similar behavior in each environment, meaning that both approaches are feasible and have similar parallelizing capabilities with minor differences in overhead. In both environments, there is an overhead of JCloudScale that is proportional to the amount of used CHs and approximately equal to 2 to 3 seconds per introduced host for a multiple minutes running evaluation application. This overhead may be significant for performance-critical production applications, but it is a reasonable price to pay in the current development stage of the JCloudScale middleware. Furthermore, how large this overhead is, depends largely on how much the target application is required to

[3] `http://www.infosys.tuwien.ac.at/staff/phdschool/rstzab/papers/TOIT14/`



(a) Execution time on OpenStack platform (b) Execution time on EC2 platform

communicate with the CHs. The main reason for this is that messaging in JCloudScale is more expensive in comparison to a pure OpenStack or EC2 solution, as JCloudScale appends some platform-specific metadata to remote invocations (e.g., which CH or CO to address, which code to run, etc.). The evaluation application required a substantial amount of coordination between target application and CHs, hence there are reasons to believe that these overhead measurements are relatively conservative. However, detailed investigations (and, subsequently, reductions) of the overhead introduced by JCloudScale is planned for future releases of the middleware.

Another important issue that is visible in Figure 8.2a and Figure 8.2b is the cloud performance stability and predictability. With an increasing number of hosts, the total execution time is expected to monotonously decrease, up to a limit when the overhead of parallelization is larger than the gain of having more processors available. This indeed happens in case of Amazon EC2. However, starting with 10 used hosts in OpenStack, the overall application execution time remains almost constant or even increases. This is mainly caused by the limited size of the used private cloud. Starting with 10 hosts, physical machines start to get over-utilized, and virtual machines start to compete for resources (e.g., CPU or disk IO).

8.4 Threats to Validity

The major threat to validity, with the results relating to the usability and usefulness evaluation (see Section 8.2), is that the small sample size of 14 study participants, along with relatively open problem statements, does not allow establishing statistical significance. However, due to the reports received from participants, as well as due to comparing the solutions themselves, we are convinced that the result that JCloudScale lets developers build cloud applications more efficiently was not a coincidence.

Further, another threat is that participants were aware that JCloudScale is our own system. Hence, there is a chance that participants gave their reports a more positive spin. However, given that all reports contained both negative and positive aspects for all evaluated platforms, we are confident that most participants reported truthfully.

There is also the possibility that the study design, which required all participants to work on two projects, skewed the results, as it can be expected that participants learned from the first project for the second one. This has been mitigated by letting a subset of the participants work with JCloudScale first, and the remainder start with one of the comparison platforms.

Finally, there is the threat that JCloudScale solutions, while being implemented more efficiently, are also of lower quality. This threat was mitigated by (partially) automatic testing of solutions against defined requirements, as well as by manual inspection and comparison of solutions.

In terms of external validity, it is possible that the two example projects that were chosen for the user study are not representative of real-world applications. However, we argue that this is unlikely, as the projects have specifically been chosen based on real-life examples that the study organizers are aware of or had to build themselves in the past. Another threat to external validity is that the participants of the study are all students at TU Vienna. While most of them have some practical real-life experience in application development, none can be considered senior developers.

In terms of performance evaluation (see Section 8.3), the major threat to external validity is that the application that was used to measure overhead on is necessarily simplified, and not guaranteed to be representative. Real applications are hard to replicate in exactly the same way on different systems, hence comparative measurements amongst such systems are always unfair. To minimize this risk, the core features of cloud applications were carefully preserved, even in the simplified measurement application.

Similarly, there is a threat to the generalizability of our study related to user study. It is possible that the results of our study would have been substantially different if we had chosen different comparison systems, e.g., Microsoft Azure or Google App Engine. However, we consider this threat small, as at the time we executed the study, the core features provided by most IaaS and PaaS providers were comparable, to the extent required by our study.

CHAPTER 9

Conclusions

The final chapter summarizes the results achieved during the work on this thesis. Section 9.1 outlines the research results and solutions presented in this work. In Section 9.2 the research questions formulated in Section 1.3 are aligned with the contributions presented in this thesis. Finally, Section 9.3 presents an outlook on future research that can be performed using the results achieved here.

9.1 Summary

The work within this thesis significantly extended the JCloudScale middleware. Now, JCloudScale is a fully-functional Java-based middleware that eases the development of elastic cloud applications on top of an IaaS cloud. JCloudScale follows a declarative approach based on Java annotations, which removes the need to actually adapt the business logics of target applications to use the middleware. Hence, JCloudScale support can easily be turned on and off for an application, leading to a flexible development process that clearly separates the implementation of the target application business logics from implementing and tuning the scaling behavior.

However, such approach brings up problems that are not apparent in other cloud application platforms and solutions. In order to provide an extensive and fully-functional cloud application distribution middleware, the contributions presented in this thesis target eliminating these problems and achieving transparent application distribution.

In order to solve the issue of application code distribution and versioning, a transparent application distribution framework was developed and integrated into JCloudScale. To simplify application bursting and distribution over different clouds, a cloud bursting solution was presented. To face the problem of scaling behavior definition complexity, a declarative and extensive scaling policy definition language was developed. To address the issue of effective resource usage in the cloud, a profile-based task scheduling and distribution framework was implemented.

Finally, the JCloudScale architecture and approach were evaluated through a qualitative and quantitative user study and a performance evaluation. The results indicate that JCloudScale is well received among initial developers. The results support the claim that the general JCloudScale model has advantages in comparison to both, working directly on top of an IaaS API or on industrial PaaS systems.

9.2 Research Questions Revisited

The research questions, introduced in Section 1.3, are revisited here. In the following, it is summarized how the research questions were addressed within this work and what limitations are still remaining despite the development of the JCloudScale middleware.

RQ 1: How can an application be transparently distributed over the cloud?

Chapter 5 introduced the APIs and the usage of the JCloudScale middleware with the main focus on transparent application distribution over the cloud. Addressing this issue, JCloudScale limits the impact on the target application by avoiding any constructs and requirements that change default application behavior or architecture. This allows keeping the application execution flow intact, while providing the ability to distribute application code over a set of CHs. This is achieved by using the means of aspect-oriented programming (see Section 2.4), which allows modifying the target application binary code after the application is compiled. As the source code and execution flow stays mostly the same, application distribution happens transparently to application developers. Finally, following the idea of transparency, JCloudScale presents the means to distribute and update on demand the application binary code and data files, thus hiding target application distribution from developers in code and dependency management as well.

However, this is not the end of the JCloudScale middleware development and there are still some places where the transparency can be increased. Currently target application developers have to manually select the code that needs to be distributed. Also developers have to manually figure out where data has to be transferred by-value and where – by-reference; manually detect and annotate the cut-points where application can be split with less harm to performance. These tasks still remain unsolved in the journey of providing truly transparent application distribution platform.

RQ 2: Which instruments and capabilities allow efficient, flexible and elastic execution of transparent cloud applications?

While working on distributed and cloud applications it becomes clear that application distribution and elastic behavior are challenging to achieve. Every IaaS and PaaS solution offers some means to control application distribution. Some are leaning towards simplicity, some are extending the means of control. However, on every platform, the target application developer has to develop his own scaling whenever the behavior of the application falls out of the default assumptions stated by platform designers.

Trying to build best approach for application scaling while following the main goal of transparent application distribution is not an easy task. Over some time the scalability assisting tools were developed and JCloudScale received an individual set of instruments that simplify scaling definition. Some of these tools, such as the cloud bursting platform, the SPEEDL language, and the profiling-based task scheduling approach, were presented in Chapter 6 and Chapter 7. However, as it was shown in the evaluation (see Section 8.2), provided tools allow achieving transparent application distribution often easier and more convenient than using conventional distribution strategies. Despite that, there is still a long road ahead in a process of finding the balance between the functionality, flexibility and convenience in defining instruments for elastic application distribution programming.

RQ 3: How to verify if the designed transparent application distribution approach is useful and fits developers' needs?

The flexible, functional, transparent, while still convenient for developers solution can not be designed solely following someone's initial idea and personal experience. In order to design a middleware that is of use to actual developers, designers have to be in a constant dialog with the target audience and listen and adapt to the market needs and preferences.

Following this concept, JCloudScale was evaluated in an extensive user study, presented in Section 8.2, during which participants were asked to design mature cloud applications using JCloudScale and modern IaaS or PaaS platforms. Results of this user study allowed us determining the actual issues that users face during application development and the next steps in JCloudScale development that will further improve developers' experience. The next step would be to involve an even wider circle of actual cloud developers to participate in JCloudScale-based application development.

9.3 Future Work

While this thesis presented solutions to a number of important issues, there are still some questions that were missed or initially stated by this work. These questions define the starting point for the ongoing research and further improvement of JCloudScale middleware and transparent application distribution in general.

- An automatic CO detection seems to be the biggest issue. While developing a transparent application distribution solution, one expects to have a tool that distributes any or at least some applications without any efforts from the application developer. While this approach is possible in general, usually it brings up a trade-off between the universality of supported applications and the performance impacts. Transparent code and data distribution was studied in the past [160, 161, 162, 163], but JCloudScale takes an alternative path and requires developers to hint the places where an application can be distributed with minimum effect on performance and stability. This allows achieving better distribution in comparison to completely automatic approaches because someone who has a solid understanding of the

source code and application behavior assists in this process. However, often this approach is unclear and puzzling for developers as they do not fully understand which components of their application should be distributed. Targeting to improve this behavior, some profiling tool or algorithm that hints or benchmarks provided CO selection should be developed in future.

- Scaling policy definition is a complex process of measuring, tweaking and adapting distribution code. Usually this process runs in parallel to application business logic development. As the user study showed, usually this is the hardest part in the whole process of cloud application development. Future work on transparent application distribution needs to explore the abilities to automatically consider the costs and penalties of application scaling and CO scheduling. Additionally, extended usage of aspect-oriented programming may improve and simplify scheduling performance, application monitoring and seamless code distribution. In addition to these internal tweaks, the developed high-level SPEEDL API has to be evaluated and extended in order to cover more scaling policy patterns and common cloud application behaviors. Finally, the profile-based task scheduling approach, presented in Chapter 7, needs to be improved and integrated as the default JCloudScale cloud management solution.

- While JCloudScale provides some basic fault handling techniques, currently this area is a weak point of the middleware. Designing an application for the cloud is usually tightly connected with "design for failure" [164]. This means that a distributed application needs to be designed assuming that the hardware will fail often, what is completely different to local application execution. This approach influences the whole design of the application, therefore it is hardly possible to achieve it transparently for the application developer. However, JCloudScale can bring in some aspects of "design for failure" into distributed application. For example, automated fault detection, that is actively discussed in research [138, 165] can be incorporated into the JCloudScale middleware. Additionally, redundant deployment, transactional memory, data storage management behavior or automatic connectivity recovery can be transparently injected into an application.

- The Internet of things (IoT) [166] is gaining more attention as the performance and capabilities of IoT devices significantly increase. Moreover, recently appeared the trend of IoT and cloud computing convergence [167, 168] and now it is gaining significant attention. Considering these tendencies, JCloudScale may be of interest in this area. Particularly, transparent application distribution, cloud management and elastic resource usage, that are already integrated into JCloudScale, are actively discussed in this raising domain [169].

- Continuing the process of further JCloudScale refinement and improvement, following user studies and external middleware evaluations are necessary. While the presented user study allowed showing the usefulness of JCloudScale, following studies are required to strengthen these claims, as the limited scale of the initial

study was not sufficient to clear all doubts about the viability of the system. Also there are plans to extend the initial user study using a more heterogeneous and larger group of developers, in order to resolve the threats to validity identified in Section 8.4. An alternative way to evaluate and popularize JCloudScale is to try migrating to the cloud some popular or third-party developed applications. This approach allows determining how the designed middleware behaves in unexpected situations and how flexible it is. Initial tests allowed determining that load tester Apache JMeter or the service composition engine JOpera [170] may be good applications to start with.

Bibliography

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Fut. Gener. Comp. Syst.*, vol. 25, no. 6, pp. 599–616, Jun. 2009.

[2] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[3] N. Leavitt, "Is cloud computing really ready for prime time?" *Computer*, vol. 42, no. 1, pp. 15–20, 2009.

[4] A. Talukder, L. Zimmerman, and A. Prahalad, "Cloud economics: Principles, costs, and benefits," in *Cloud Computing*. Springer, 2010, pp. 343–360.

[5] J. Varia, "Cloud architectures," *White Paper of Amazon*, p. 16, 2008.

[6] A. Berl, E. Gelenbe, M. Di Girolamo, G. Giuliani, H. De Meer, M. Q. Dang, and K. Pentikousis, "Energy-efficient cloud computing," *The Computer Journal*, vol. 53, no. 7, pp. 1045–1051, 2010.

[7] G. Reese, *Cloud Application Architectures: Building Applications and Infrastructure in the Cloud*. O'Reilly Media, Inc., 2009.

[8] T. Dillon, C. Wu, and E. Chang, "Cloud Computing: Issues and Challenges," in *24th IEEE International Conference on Advanced Information Networking and Applications (AINA'10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 27–33. [Online]. Available: http://dx.doi.org/10.1109/AINA.2010.187

[9] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications an empirical study on software development for the cloud," *arXiv preprint arXiv:1409.6502*, 2014.

[10] K. Jayaram, "Elastic remote methods," in *Proceedings of Middleware 2013*, ser. Lecture Notes in Computer Science, D. Eyers and K. Schwan, Eds., vol. 8275. Springer Berlin Heidelberg, 2013, pp. 143–162. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-45065-5_8

[11] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," in *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on.* Ieee, 2009, pp. 44–51.

[12] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, "Efficient resource provisioning in compute clouds via vm multiplexing," in *7th international conference on Autonomic computing.* ACM, 2010, pp. 11–20.

[13] M. Randles, D. Lamb, and A. Taleb-Bendiab, "A comparative study into distributed load balancing algorithms for cloud computing," in *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on.* IEEE, 2010, pp. 551–556.

[14] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.* ACM, 2011, p. 49.

[15] P. Kruchten, R. Capilla, and J. C. Dueas, "The decision view's role in software architecture practice," *Software, IEEE*, vol. 26, no. 2, pp. 36–42, 2009.

[16] P. Leitner, B. Satzger, W. Hummer, C. Inzinger, and S. Dustdar, "Cloudscale: a novel middleware for building transparently scaling cloud applications," in *27th Annual ACM Symposium on Applied Computing (SAC '12).* New York, NY, USA: ACM, 2012, pp. 434–440.

[17] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet Computing*, vol. 13, no. 5, pp. 14–22, Sep. 2009. [Online]. Available: http://dx.doi.org/10.1109/MIC.2009.119

[18] R. Zabolotnyi, P. Leitner, and S. Dustdar, *Handbook of Research on Architectural Trends in Service-Driven Computing.* IGI Global, 2014, ch. Building Elastic Java Application Services Seamlessly in the Cloud: A Middleware Framework, pp. 661–685.

[19] R. Zabolotnyi, P. Leitner, W. Hummer, and S. Dustdar, "JCloudScale: closing the gap between IaaS and PaaS," *ACM Transactions on Internet Technology (TOIT)*, vol. 15, no. 3, Jul. 2015.

[20] R. Zabolotnyi, P. Leitner, and S. Dustdar, "Dynamic program code distribution in infrastructure-as-a-service clouds," in *Proceedings of the 5th International Workshop on Principles of Engineering Service-Oriented Systems (PESOS 2013), co-located with ICSE 2013*, 2013.

[21] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005. [Online]. Available: http://doi.acm.org/10.1145/1118890.1118892

[22] R. Zabolotnyi, P. Leitner, S. Schulte, and S. Dustdar, "SPEEDL – a declarative event-based language for cloud scaling definition," in *The Future of Software Engineering FOR and IN Cloud visionary track of The IEEE SERVICES 2015*, 2015.

[23] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing.* Wiley, 2007.

[24] R. Zabolotnyi, P. Leitner, and S. Dustdar, "Profiling-based task scheduling for factory-worker applications in infrastructure-as-a-service clouds," in *40th EUROMICRO Conference on Software Engineering and Advanced Applications.* IEEE, 2014, pp. 119–126.

[25] P. Mell and T. Grance, "The nist definition of cloud computing (draft)," *NIST Special Publication*, vol. 800, p. 145, 2011.

[26] S. Dustdar, Y. Guo, B. Satzger, and H. Truong, "Principles of elastic processes," *Internet Computing, IEEE*, vol. 15, no. 5, pp. 66–71, 2011.

[27] A. Regalado, "Who coined "cloud computing"," *Technology Review*, vol. 31, 2011.

[28] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali, "Cloud computing: Distributed internet computing for it and scientific research," *Internet Computing, IEEE*, vol. 13, no. 5, pp. 10–13, 2009.

[29] P. Neto, "Demystifying cloud computing," in *Proceeding of Doctoral Symposium on Informatics Engineering*, 2011.

[30] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.

[31] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud computing: principles and paradigms.* John Wiley & Sons, 2010, vol. 87.

[32] G. J. Tellis, "The price elasticity of selective demand: A meta-analysis of econometric models of sales," *Journal of marketing research*, pp. 331–341, 1988.

[33] E. M. Arruda and M. C. Boyce, "A three-dimensional constitutive model for the large stretch behavior of rubber elastic materials," *Journal of the Mechanics and Physics of Solids*, vol. 41, no. 2, pp. 389–412, 1993.

[34] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not." in *ICAC*, 2013, pp. 23–27.

[35] C. Leopold, *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches.* John Wiley & Sons, Inc., 2001.

[36] R. L. Probert and K. Saleh, "Synthesis of communication protocols: survey and assessment," *Computers, IEEE Transactions on*, vol. 40, no. 4, pp. 468–476, 1991.

[37] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, 2001, pp. 101–102.

[38] G. R. Andrews, *Foundations of parallel and distributed programming*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[39] T. Erl, *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2005.

[40] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly Media, Inc., 2008.

[41] D. Chappell, *Enterprise service bus*. O'Reilly Media, Inc., 2004.

[42] E. Curry, "Message-oriented middleware," *Middleware for communications*, pp. 1–28, 2004.

[43] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002, pp. 1–16.

[44] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[45] M. Natu and A. S. Sethi, "Active probing approach for fault localization in computer networks," in *End-to-End Monitoring Techniques and Services, 2006 4th IEEE/IFIP Workshop on*. IEEE, 2006, pp. 25–33.

[46] O. Dictionaries, "Oxford dictionaries," *Oxford University Press*, vol. 15, 2010.

[47] O. Etzion and P. Niblett, *Event processing in action*. Manning Publications Co., 2010.

[48] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar, "Application-level performance monitoring of cloud services based on the complex event processing paradigm," in *Proceedings of the 2012 5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, 2012, pp. 1–8.

[49] B. M. Michelson, "Event-driven architecture overview," *Patricia Seybold Group*, vol. 2, 2006.

[50] W. W. Eckerson, "Three tier client/server architectures: achieving scalability, performance, and efficiency in client/server applications," *Open Information Systems*, vol. 3, no. 20, pp. 46–50, 1995.

[51] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, *Aspect-oriented software development.* Addison-Wesley Professional, 2004.

[52] G. J. Kiczales, J. O. Lamping, C. V. Lopes, J. J. Hugunin, E. A. Hilsdale, and C. Boyapati, "Aspect-oriented programming," Oct. 15 2002, uS Patent 6,467,086.

[53] C. Vecchiola, X. Chu, and R. Buyya, "Aneka: a software platform for .net based cloud computing," in *Proceedings of the High Performance Computing Workshop*, 2008, pp. 267–295.

[54] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya, "The Aneka platform and QoS-driven resource provisioning for elastic applications on hybrid Clouds," *Future Generation Computer Systems*, vol. 28, no. 6, pp. 861–870, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.future.2011.07.005

[55] J. Simao, J. Lemos, and L. Veiga, "A2-vm: A cooperative java vm with support for resource-awareness and cluster-wide thread scheduling," in *Proceedings of the 19th International Conference on Cooperative Information Systems (CoopIS 2011)*, September 2011.

[56] A. Zahariev, "Google app engine," *Helsinki University of Technology*, 2009.

[57] J. Vliet, F. Paganelli, S. Van Wel, and D. Dowd, *Elastic Beanstalk.* O'Reilly Media, Inc., 2011.

[58] N. Middleton, R. Schneeman *et al.*, *Heroku: Up and Running.* O'Reilly Media, Inc., 2013.

[59] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski, "Appscale: Scalable and open appengine application development and deployment," in *Cloud Computing*, ser. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, D. Avresky, M. Diaz, A. Bode, B. Ciciani, and E. Dekel, Eds. Springer Berlin Heidelberg, 2010, vol. 34, pp. 57–70. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12636-9_4

[60] C. Krintz, "The appscale cloud platform: Enabling portable, scalable web application deployment," *IEEE Internet Computing*, vol. 17, no. 2, pp. 72–75, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1109/MIC.2013.38

[61] G. Pierre, I. El Helw, C. Stratan, A. Oprescu, T. Kielmann, T. Schütt, J. Stender, M. Artač, and A. Černivec, "Conpaas: An integrated runtime environment for elastic cloud applications," in *Proceedings of the Workshop, Posters and Demos Track (Middleware'11).* New York, NY, USA: ACM, 2011, pp. 5:1–5:2. [Online]. Available: http://doi.acm.org/10.1145/2088960.2088965

[62] G. Pierre and C. Stratan, "Conpaas: A platform for hosting elastic cloud applications," *IEEE Internet Computing*, vol. 16, no. 5, pp. 88–92, 2012.

[63] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears, "Boom analytics: Exploring data-centric, declarative programming for the cloud," in *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, 2010, pp. 223–236.

[64] S. Krishnan and J. L. U. Gonzalez, "Google cloud dataflow," in *Building Your Next Big Thing with Google Cloud Platform*. Springer, 2015, pp. 255–275.

[65] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: Towards an elastic stream computing platform for the cloud," in *IEEE 5th International Conference on Cloud Computing (CLOUD'11)*, 2011, pp. 348–355.

[66] S. Pallickara, J. Ekanayake, and G. Fox, "Granules: A lightweight streaming runtime for cloud computing with support for map-reduce," in *Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09)*. IEEE, 2009, pp. 1–10.

[67] K. Jayaram, "Towards explicitly elastic programming frameworks," in *Proceedings of the Internetional Conference on Software Engineering (ICSE 2015), New Ideas and Emerging Results Track*, 2015.

[68] A. Thiery, T. Cerqueus, C. Thorpe, G. Sunye, and J. Murphy, "A dsl for deployment and testing in the cloud," in *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 376–382.

[69] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[70] 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)*, R. Meersman, T. Dillon, and P. Herrero, Eds. Springer Berlin / Heidelberg, 2009, vol. 5870, pp. 357–364. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-05148-7_24

[71] C. Inzinger, S. Nastic, S. Sehic, M. Vögler, F. Li, and S. Dustdar, "MADCAT: a methodology for architecture and deployment of cloud application topologies," in *Proceedings of the 8th International Symposium on Service Oriented System Engineering (SOSE)*, April 2014, pp. 13–22.

[72] 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 the 11th International Conference on Service-Oriented Computing (ICSOC)*, 2013, pp. 692–695.

[73] S. Vinoski, "Convenience over correctness," *IEEE Internet Computing*, vol. 12, no. 4, pp. 89–92, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/MIC.2008.75

[74] P. Sampaio, P. Ferreira, and L. Veiga, "Transparent scalability with clustering for java e-science applications," in *11th International Conference on Distributed Applications and Interoperable Systems*. Springer, 2011, pp. 270–277.

[75] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 1, pp. 45–52, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1925861.1925869

[76] W. Emmerich, *Engineering Distributed Objects*. John Wiley & Sons, 2000.

[77] "Mapreduce successor google cloud dataflow is a game changer for hadoop thunder," http://cloudtimes.org/2014/07/07/mapreduce-successor-google-cloud-dataflow-is-a-game-changer-for-hadoop-thunder/, Last accessed: 2015.08.04.

[78] P. Leitner, Z. Rostyslav, A. Gambi, and S. Dustdar, "A framework and middleware for application-level cloud bursting on top of infrastructure-as-a-service clouds," in *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, ser. UCC '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 163–170. [Online]. Available: http://dx.doi.org/10.1109/UCC.2013.39

[79] J. W. Armitage and J. V. Chelini, "Ada software on distributed targets: a survey of approaches," *ACM SIGAda Ada Letters*, vol. 4, no. 4, pp. 32–37, 1985.

[80] C. Dumitrescu, I. Raicu, M. Ripeanu, and I. Foster, "Diperf: An automated distributed performance testing framework," in *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE, 2004, pp. 289–296.

[81] G. Von Laszewski, E. Blau, M. Bletzinger, J. Gawor, P. Lane, S. Martin, and M. Russell, "Software, component, and service deployment in computational grids," in *Component Deployment*. Springer, 2002, pp. 244–256.

[82] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop, 2008. GCE'08*. Ieee, 2008, pp. 1–10.

[83] A. Van Hoff, J. Payne, and S. Shaio, "Method for the distribution of code and data updates," Jul. 6 1999, uS Patent 5,919,247.

[84] S. Bratus, J. Oakley, A. Ramaswamy, S. Smith, and M. Locasto, "Katana: Towards patching as a runtime part of the compiler-linker-loader toolchain," *International Journal of Secure Software Engineering (IJSSE)*, vol. 1, no. 3, pp. 1–17, 2010.

[85] C.-P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. t. Hart, "Enabling multi-tenancy: An industrial experience report," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2010.5609735

[86] M. Ghorbel, M. Mokhtari, and S. Renouard, "A distributed approach for assistive service provision in pervasive environment," in *Proceedings of the 4th International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots*. ACM, 2006, pp. 91–100.

[87] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer, "Mole–concepts of a mobile agent system," *World Wide Web*, vol. 1, no. 3, pp. 123–137, 1998.

[88] K. Rothermel, F. Hohl, and N. Radouniklis, "Mobile agent systems: What is missing?" *Distributed Applications and Interoperable Systems (DAIS'97), Chapman & Hall*, pp. 111–124, 1997.

[89] M. Baldi, S. Gai, and G. Picco, "Exploiting code mobility in decentralized and flexible network management," in *Mobile Agents*. Springer, 1997, pp. 13–26.

[90] G. Cabri, L. Leonardi, and F. Zambonelli, "Weak and strong mobility in mobile agent applications," in *Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java (PA JAVA 2000), Manchester (UK)*, 2000.

[91] A. Carzaniga, G. Picco, and G. Vigna, "Designing distributed applications with mobile code paradigms," in *Proceedings of the 19th International Conference on Software Engineering*. ACM, 1997, pp. 22–32.

[92] V. CeronmaniSharmila and V. KomalaValli, "Enhanced security through agent based non-repudiation protocol for mobile agents," *International Journal of Power Control Signal and Computation(IJPCSC)*, vol. 3, no. 1, 2012.

[93] E. Sanchis, "Mobility and remote-code execution," in *Mobile Wireless Middleware, Operating Systems, and Applications-Workshops*. Springer, 2009, pp. 85–97.

[94] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 138–153, 1990.

[95] P. Naik, S. Agrawal, and S. Murthy, "A survey on various task scheduling algorithms toward load balancing in public cloud," *American Journal of Applied Mathematics*, vol. 3, no. 1-2, pp. 14–17, 2015.

[96] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," *Parallel Computing*, vol. 38, no. 1–2, pp. 37–51, 2012.

[97] F. Singhoff and A. Plantec, "AADL modeling and analysis of hierarchical schedulers," in *2007 Annual ACM SIGAda International Conference on Ada*. ACM, 2007, pp. 41–50.

[98] C.-C. Lin, P. Liu, and J.-J. Wu, "Energy-aware virtual machine dynamic provision and scheduling for cloud computing," in *4th International Conference on Cloud Computing (CLOUD 2011)*. IEEE, 2011, pp. 736–737.

[99] J. Bi, Z. Zhu, R. Tian, and Q. Wang, "Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center," in *3rd International Conference on Cloud Computing (CLOUD 2010)*. IEEE, 2010, pp. 370–377.

[100] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, "BAR: an efficient data locality driven task scheduling algorithm for cloud computing," in *The 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '11)*. IEEE, 2011, pp. 295–304. [Online]. Available: http://dx.doi.org/10.1109/CCGrid.2011.55

[101] H. Choi, W. Choi, T. M. Quan, D. Hildebrand, H. Pfister, and W.-K. Jeong, "Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems," *IEEE Transactions on Visualization and Computer Graphics,*, vol. 20, no. 12, pp. 2407–2416, 2014.

[102] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar, "Cost-efficient and application sla-aware client side request scheduling in an infrastructure-as-a-service cloud," in *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing (CLOUD '12)*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 213–220.

[103] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimization of resource provisioning cost in cloud computing," *IEEE Transactions on Services Computing*, vol. 5, no. 2, pp. 164–177, 2012.

[104] G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar, "SYBL: an extensible language for controlling elasticity in cloud applications," in *The 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '13)*. IEEE, 2013, pp. 112–119.

[105] 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.

[106] J. Hu, J. Gu, G. Sun, and T. Zhao, "A scheduling strategy on load balancing of virtual machine resources in cloud computing environment," in *Parallel Architectures, Algorithms and Programming (PAAP), 2010 Third International Symposium on*. IEEE, 2010, pp. 89–96.

[107] M. Xu, L. Cui, H. Wang, and Y. Bi, "A multiple qos constrained scheduling strategy of multiple workflows for cloud computing," in *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*. IEEE, 2009, pp. 629–634.

[108] H. Qi-yi and H. Ting-lei, "An optimistic job scheduling strategy based on qos for cloud computing," in *2010 Inter. Conference on Intelligent Comput. and Integrated Systems*. IEEE, 2010, pp. 673–675.

[109] P. Hoenisch, S. Schulte, and S. Dustdar, "Workflow Scheduling and Resource Allocation for Cloud-based Execution of Elastic Processes," in *6th IEEE International Conference on Service Oriented Computing and Applications (SOCA 2013)*. IEEE, 2013, pp. 1–8.

[110] S. Schulte, D. Schuller, P. Hoenisch, U. Lampe, S. Dustdar, and R. Steinmetz, "Cost-driven optimization of cloud resource allocation for elastic processes," *International Journal of Cloud Computing*, vol. 1, no. 2, pp. 1–14, 2013.

[111] R. Buyya, S. K. Garg, and R. N. Calheiros, "Sla-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions," in *Cloud and Service Computing (CSC), 2011 International Conference on*. IEEE, 2011, pp. 1–10.

[112] L. Wu, S. K. Garg, and R. Buyya, "Sla-based resource allocation for software as a service provider (saas) in cloud computing environments," in *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. IEEE, 2011, pp. 195–204.

[113] C. Gronroos, "Service quality: the six criteria of good perceived service," *Review of business*, vol. 9, no. 3, p. 10, 1988.

[114] J. R. d. A. Amazonas and C. R. Barra, "Experimental characterization and modeling of the qos for real time audio and video transmission," *Interpretation*, p. 800, 1960.

[115] L. Zhao, S. Sakr, A. Liu, and A. Bouguettaya, "Qos-aware service compositions in cloud computing," in *Cloud Data Management*. Springer, 2014, pp. 119–133.

[116] A. Abdelmaboud, D. N. Jawawi, I. Ghani, A. Elsafi, and B. Kitchenham, "Quality of service approaches in cloud computing: A systematic mapping study," *Journal of Systems and Software*, vol. 101, pp. 159–179, 2015.

[117] L. Wu, S. K. Garg, S. Versteeg, and R. Buyya, "Sla-based resource provisioning for hosted software-as-a-service applications in cloud computing environments," *Services Computing, IEEE Transactions on*, vol. 7, no. 3, pp. 465–485, 2014.

[118] X. Zhao, L. Shen, X. Peng, and W. Zhao, "Toward sla-constrained service composition: An approach based on a fuzzy linguistic preference model and an evolutionary algorithm," *Information Sciences*, vol. 316, pp. 370–396, 2015.

[119] F. Jrad, J. Tao, I. Brandic, and A. Streit, "Sla enactment for large-scale healthcare workflows on multi-cloud," *Future Generation Computer Systems*, vol. 43, pp. 135–148, 2015.

[120] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web services agreement specification (ws-agreement)," in *Open Grid Forum*, vol. 128, 2007, p. 216.

[121] V. Stantchev and C. Schröpfer, "Negotiating and enforcing qos and slas in grid and cloud computing," in *Advances in grid and pervasive computing*. Springer, 2009, pp. 25–35.

[122] P. Patel, A. H. Ranabahu, and A. P. Sheth, "Service level agreement in cloud computing," 2009.

[123] M. D. De Assunçao, A. Di Costanzo, and R. Buyya, "Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*. ACM, 2009, pp. 141–150.

[124] E. N. Alkhanak, S. P. Lee, and S. U. R. Khan, "Cost-aware challenges for workflow scheduling approaches in cloud computing environments: Taxonomy and opportunities," *Future Generation Computer Systems*, 2015.

[125] R. W. de Medeiros, N. S. Rosa, L. Ferreira Pires *et al.*, "A metamodel for modeling cost behavior in service composition," in *Computer Systems and Applications (AICCSA), 2014 IEEE/ACS 11th International Conference on*. IEEE, 2014, pp. 84–91.

[126] M. D. De Assuncao, M. A. S. Netto, L. Renganarayana, and C. C. Young, "System, method and program product for cost-aware selection of stored virtual machine images for subsequent use," May 19 2015, uS Patent 9,038,085.

[127] R. Buyya, A. Beloglazov, and J. Abawajy, "Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges," *arXiv preprint arXiv:1006.0308*, 2010.

[128] J. Baliga, R. W. Ayre, K. Hinton, and R. Tucker, "Green cloud computing: Balancing energy in processing, storage, and transport," *Proceedings of the IEEE*, vol. 99, no. 1, pp. 149–167, 2011.

[129] Y. C. Lee and A. Y. Zomaya, "Rescheduling for reliable job completion with the support of clouds," *Future Generation Computer Systems*, vol. 26, no. 8, pp. 1192–1199, 2010.

[130] G. Suciu, C. Cernat, G. Todoran, V. Suciu, V. Poenaru, T. Militaru, and S. Halunga, "A solution for implementing resilience in open source cloud platforms," in *Communications (COMM), 2012 9th International Conference on*. IEEE, 2012, pp. 335–338.

[131] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[132] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *Network and Service Management (CNSM), 2010 International Conference on*. IEEE, 2010, pp. 9–16.

[133] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 155–165.

[134] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber, "Scientific data management in the coming decade," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 34–41, 2005.

[135] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1978915.1978919

[136] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions Database Systems*, vol. 6, no. 2, pp. 213–226, Jun. 1981. [Online]. Available: http://doi.acm.org/10.1145/319566.319567

[137] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications – an empirical study on software development for the cloud."

[138] C. Schneider, A. Barker, and S. Dobson, "Autonomous fault detection in self-healing systems: Comparing hidden markov models and artificial neural networks," in *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems*. ACM, 2014, p. 24.

[139] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003. [Online]. Available: http://dx.doi.org/10.1109/MC.2003.1160055

[140] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional, May 2002.

[141] S. Genaud and J. Gossa, "Cost-wait trade-offs in client-side resource provisioning with elastic clouds," in *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, ser. CLOUD '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2011.23

[142] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Testing idempotence for infrastructure as code," in *ACM/IFIP/USENIX Middleware Conference*, 2013, pp. 368–388.

[143] T. Guo, U. Sharma, P. Shenoy, T. Wood, and S. Sahu, "Cost-aware cloud bursting for enterprise applications," *ACM Transactions on Internet Technology (TOIT)*, vol. 13, no. 3, p. 10, 2014.

[144] S. Imai, T. Chestna, and C. A. Varela, "Elastic scalable cloud computing using application-level migration," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing (UCC'12)*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 91–98.

[145] D. Warneke and O. Kao, "Exploiting dynamic resource allocation for efficient parallel data processing in the cloud," *IEEE Transactions Parallel and Distributed Systems*, vol. 22, no. 6, pp. 985–997, 2011.

[146] R. Fears, H. Brand, R. Frackowiak, P.-P. Pastoret, R. Souhami, and B. Thompson, "Data protection regulation and the promotion of health research: getting the balance right," *QJM*, vol. 107, no. 1, pp. 3–5, 2014.

[147] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring alternative approaches to implement an elasticity policy," in *4th International Conference on Cloud Computing (CLOUD 2011)*. IEEE, 2011, pp. 716–723.

[148] G. Galante and L. C. Bona, "A survey on cloud computing elasticity," in *2012 IEEE Fifth International Conference on Utility and Cloud Computing (UCC 2012)*. IEEE, 2012, pp. 263–270. [Online]. Available: http://dx.doi.org/10.1109/UCC.2012.30

[149] M. Fowler, *Domain Specific Languages*. Pearson Education, 2010.

[150] J. Allspaw, *The Art of Capacity Planning: Scaling Web Resources*. O'Reilly Media, Inc., 2008.

[151] F.-f. Han, J.-j. Peng, W. Zhang, Q. Li, J.-d. Li, Q.-l. Jiang, and Q. Yuan, "Virtual resource monitoring in cloud computing," *Journal of Shanghai University (English Edition)*, vol. 15, pp. 381–385, 2011.

[152] "Amazon autoscaling," http://aws.amazon.com/autoscaling/, Last accessed: 2015.08.04.

[153] "Rackspace cloud monitoring," http://copperegg.com/rackspace/, Last accessed: 2015.08.04.

[154] S.-k. Kwon and J.-h. Noh, "Implementation of monitoring system for cloud computing environments."

[155] S. Martello and P. Toth, *Knapsack problems*. Wiley New York, 1990.

[156] N. Radcliffe and P. Surry, "Formal Memetic Algorithms," *Evolutionary Computing*, vol. 865, pp. 1–16, 1994.

[157] "Usefulness vs. usability – expero – simplifying complexity," http://experoinc.com/usefulness-vs-usability/, Last accessed: 2015.08.03.

[158] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen, "Action research," *Commun. ACM*, vol. 42, no. 1, pp. 94–97, Jan. 1999. [Online]. Available: http://doi.acm.org/10.1145/291469.291479

[159] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[160] E. Tilevich and Y. Smaragdakis, "Transparent program transformations in the presence of opaque code," in *Proceedings of the 5th international conference on Generative programming and component engineering*. ACM, 2006, pp. 89–94.

[161] E. Farcas, C. Farcas, W. Pree, and J. Templ, "Transparent distribution of real-time components based on logical execution time," *ACM SIGPLAN Notices*, vol. 40, no. 7, pp. 31–39, 2005.

[162] J. I. Landman, H. N. Cofer, R. Gomperts, and D. Mikhailov, "Transparent distribution and execution of data in a multiprocessor environment," Jul. 24 2007, uS Patent 7,249,357.

[163] T. J. Collins III, S. R. Anderson, S. J. McDowall, C. H. Kratsch, and J. P. Larson, "System for software distribution in a digital computer network," Dec. 1 1998, uS Patent 5,845,090.

[164] P. Beerthuizen, "Designing for failure," in *DASIA 2003*, vol. 532, 2003, p. 70.

[165] B. S. J. Costa, P. P. Angelov, and L. A. Guedes, "Fully unsupervised fault detection and identification based on recursive density estimation and self-evolving cloud-based classifier," *Neurocomputing*, vol. 150, pp. 289–303, 2015.

[166] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[167] C. Doukas and I. Maglogiannis, "Bringing iot and cloud computing towards pervasive healthcare," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*. IEEE, 2012, pp. 922–926.

[168] E. Sun, X. Zhang, and Z. Li, "The internet of things (iot) and cloud computing (cc) based tailings dam monitoring and pre-alarm system in mines," *Safety science*, vol. 50, no. 4, pp. 811–815, 2012.

[169] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497–1516, 2012.

[170] C. Pautasso and G. Alonso, "JOpera: A toolkit for efficient visual composition of web services," *International Journal of Electronic Commerce*, vol. 9, no. 2, pp. 107–141, Jan. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1278095.1278101

JCloudScale Documentation

Introduction

JCloudScale is a framework for deploying and managing applications in an *Infrastructure-as-a-Service* cloud, for instance Amazon EC2. Essentially, JCloudScale wants to allow you to write a distributed, elastic application like a regular (local) Java app. The general concept of JCloudScale is to use Aspect-Oriented Programming (AOP) techniques to dynamically modify the bytecode of Java-based applications, and transparently move designated (via annotations) parts of the application (which we refer to as cloud objects) to virtual resources in the cloud (referred to as cloud hosts). This process is transparent to the application developer, and is completely automated. In the end, applications built on top of JCloudScale look like regular (local) Java applications, but are actually executed in a distributed fashion.

What Kind of Applications Can Profit from JCloudScale?

JCloudScale is perfectly suited to help you build applications that are:

- *multi-threaded*: they inherently want to do multiple things in parallel.
- *computation-heavy*: some of the things the application wants to do take a long time to finish and produce significant processor load.
- *memory-heavy*: application operations require significant amount of memory that are hard to satisfy on a single machine.
- *elastic*: load on the application is not always the same, and the application should adapt its usage of cloud resources based on current load.

How does JCloudScale help you with such applications? It lets you focus on what matters - your business logics. All the pesky interactions with the cloud are handled by JCloudScale. In many applications, you do not even see the cloud anymore in your application.

Required Software

As the JCloudScale project and necessary infrastructure is based on Java, any operation system may be used. We have tested JCloudScale on many versions of Windows, Mac OS X, and Linux.

Before you start, please ensure that the following software is available or installed on your machine (recommended version is specified in italics, however newer versions should work as well, unless specified otherwise).

1. Java JDK (*Oracle Java SE JDK 1.7*)
2. Apache Maven (*Maven 3.0*) - JCloudScale is developed as Maven project and is available as maven artifact. The easiest approach for you would be to base your own application on Maven as well. Of course, this is not mandatory, but otherwise dependency management and aspect weaving become more cumbersome to configure correctly.
3. **Optional:** Apache ActiveMQ (*Apache ActiveMQ 5.8.0*) - JCloudScale uses ActiveMQ service for communication between clients and cloud hosts. For production deployment, a stand-alone ActiveMQ instance should be used. If you are just toying around or testing your solution, you do not need an ActiveMQ installation, as JCloudScale is able to instantiate an integrated message queue if no external MQ is available.

Javadocs

Please find the automatically generated API documentation for JCloudScale here[1].

Current Version

This documentation describes the latest stable JCloudScale release **0.4.0**. As we are continuing working on this project, latest version can be found in our maven repository[2]. Note, however, that newer versions do not necessarily have the same API and behavior may differ from the one described here.

Basic Usage

This section focuses on Maven-based projects. If you want to use JCloudScale *without Maven, read Section 9.3 first.*

As a first step to get started with JCloudScale. you need to add it to your project's list of dependencies in your Maven `pom.xml` file as it is shown in Listing 1.

In addition, as JCloudScale is not yet registered in public maven repositories, you need to add a reference to the TU Wien Infosys maven repository as well (see Listing 2).

[1] `http://xleitix.github.io/jcloudscale/apidocs/`
[2] `http://www.infosys.tuwien.ac.at/mvn/jcloudscale/`

Listing 1: Maven configuration to include JCloudScale dependency

```
1  <dependency>
2          <groupId>jcloudscale</groupId>
3          <artifactId>jcloudscale.core</artifactId>
4          <version>0.4.0</version>
5  </dependency>
```

Listing 2: Maven configuration to reference TU Wien maven repository

```
1  <repositories>
2      <repository>
3          <id>infosys-repository</id>
4          <url>http://www.infosys.tuwien.ac.at/mvn</url>
5      </repository>
6  </repositories>
```

However, this is not the only required change to the build process. Additionally, as JCloudScale uses AspectJ to weave its code into your application, you will need to add compile-time weaving as another step of your compilation process. Many things can be tweaked here, but for most cases it should be sufficient to simply add this plugin configuration to your `pom.xml` file (into `plugins` subsection of a `build` section) (see Listing 3).

Now it is time to actually start using JCloudScale in the application. The key concept of the JCloudScale is the notion of *cloud objects*. Cloud objects are represented by Java classes annotated with the `@CloudObject` annotation. Whenever JCloudScale finds the creation of the new instance of the class using this annotation, it replaces the constructor call with the code necessary to select a remote host for the object and deploy it there, providing only a proxy object to the application. Following, any invocations on this proxy object actually become remote method invocations on the cloud object.

A simple cloud object is defined in Listing 4.

Every instance of `MyCloudObject` is instantiated on a remote host, and all invocations to this object are redirected to this host. As soon as the method `MyCloudObject.iAmDone()` is invoked, the cloud object is destroyed on the remote host (and any subsequent invocations to this object will trigger `JCloudScaleException`).

Using this cloud object in your code is simple. It is just like using any other Java object, what is shown in Listing 5.

Note that this simple code snippet actually already triggers some serious back-and-forth between your client application and the remote host, as indicated in Figure 5.2.

When using JCloudScale. you should also indicate when your application does not require JCloudScale anymore and can be shut down. To do this, you can use the `@JCloudScaleShutdown` annotation. After the execution of the method annotated with it, JCloudScale will insert necessary calls to gracefully destroy all remaining

Listing 3: Maven configuration to enable AspectJ post-compilation

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>aspectj-maven-plugin</artifactId>
    <version>1.4</version>
    <configuration>
        <source>1.7</source>
        <target>1.7</target>
        <complianceLevel>1.7</complianceLevel>
        <verbose>true</verbose>
    </configuration>
    <executions>
        <execution>
            <configuration>
                <XnoInline>true</XnoInline>
                <aspectLibraries>
                    <aspectLibrary>
                        <groupId>jcloudscale</groupId>
                        <artifactId>jcloudscale.core</artifactId>
                    </aspectLibrary>
                </aspectLibraries>
            </configuration>
            <goals>
                <goal>compile</goal>
                <goal>test-compile</goal>
            </goals>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.7.0</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>1.7.0</version>
        </dependency>
    </dependencies>
</plugin>
```

cloud objects and shutdown any additional infrastructure created by the framework to communicate with the remote hosts. An example of such annotation usage is shown in Listing 6

Alternatively, you can invoke `JCloudScaleClient.closeClient()` in order to shutdown all JCloudScale infrastructure manually. Don't forget, that any interaction with JCloudScale after shutdown will cause exceptions.

Using JCloudScale without Maven

While we encourage users to use Maven for the JCloudScale-based projects, it does not mean that it is impossible to use JCloudScale without Maven. In this short

Listing 4: An example of cloud object class

```
1  @CloudObject
2  public class MyCloudObject
3  {
4
5   public String doThings() { ... }
6
7   public void doOtherThings(String param1, String param2) { ... }
8
9   @DestructCloudObject
10  public void iAmDone() { ... }
11
12 }
```

Listing 5: An example of using cloud object

```
1  System.out.println("Starting");
2  MyCloudObject object = new MyCloudObject();
3  System.out.println(object.doThings());
4  object.doOtherThings("first", "second");
5  object.iAmDone();
6  System.out.println("Done");
```

Listing 6: An example of method that shuts down JCloudScale

```
1  @JCloudScaleShutdown
2  public static void main(String[] args) { ... }
```

guide we will try to describe the necessary steps you need to perform in order to use JCloudScale without Maven.

Introduction

As Maven is a project management tool, everything that needs to be performed differently is the way JCloudScale has to be referenced and integrated into your project.

Note, that this section of documentation does not reflect the way we perform testing or encourage others to use JCloudScale. Therefore, it may be a bit outdated or expose some issues that do not occur with default approach. Whenever you face such situation, be free to inform us in order to fix these issues.

Adding JCloudScale dependency

In order to access JCloudScale functionality or compile the code that references JCloudScale. you need to add JCloudScale jars to your project setup. JCloudScale jar can be obtained directly from our maven repository[3], while you still might

[3] http://www.infosys.tuwien.ac.at/mvn/jcloudscale/jcloudscale.core/

have difficulties finding jars that JCloudScale depends on. You can either try to figure out all necessary for your particular use case dependencies manually (by adding missing jars as long as you get `ClassNotFoundException`) or, if you do have Maven on your machine, you can run `mvn package` command in the directory with the `pom.xml` file with the following content to collect all dependencies of JCloudScale (some of them might be not necessary for your particular use case, but if you include them all, you will definitely be on the safe side). Appropriate `pom.xml` file is shown in Listing 46.

Applying AspectJ Aspects

As JCloudScale seamless integration depends on AspectJ, we need to apply AspectJ aspects defined by JCloudScale to your project in order to achieve the same features as default Maven-based setup provides. Of course, if you don't plan to use annotation-based features of JCloudScale and plan using JCloudScale API, you can skip this step.

AcpectJ provides 3 types of aspect weaving:

1. *compile-time weaving*: compile either target source or aspect classes via dedicated AspectJ compiler;

2. *post-compile weaving*: inject aspect instructions to already compiled classes;

3. *load-time weaving*: inject aspect instructions to the byte code during class loading, i.e. load instrumented class instead of the "raw" one;

Note, If you are using any IDE, it may have plugins or embedded features that simplify AspectJ usage (e.g., AJDT for Eclipse, or AspectJ support in IntelliJ IDEA). In this manual we briefly describe raw AspectJ usage independently from any IDE.

Compile-time weaving

Compile-time weaving requires source code of the application to be compiled by AspectJ Compiler(ajc)[4] instead of default javac compiler (assuming `ajc` is in PATH and `aspectjrt.jar` is in CLASSPATH), as it is shown in Listing 7.

Listing 7: Compile-time aspect weaving

```
1 ajc -aspectpath jcloudscale.core-0.4.0.jar
2                 -classpath <your classpath here> -1.7
3                 -sourceroots <source folder here>
4                 -outjar <compiled jar here>
```

A concrete example of this command is shown in Listing 8.

After this, application can be started from the resulting jar as it is shown in Listing 9.

[4] `https://eclipse.org/aspectj/downloads.php`

Listing 8: Compile-time aspect weaving example

```
1 ajc -aspectpath lib/jcloudscale.core-0.4.0.jar
2                 -classpath "lib/*" -1.7 -sourceroots test
3                 -outjar code.jar
```

Listing 9: Application starting after the compile-time aspect application

```
1 java -cp<your classpath here> <specify your main class here>
2
3 java -cp code.jar;lib/* test.Main
```

Post-compile weaving

Post-compile weaving gives you more freedom to compile your source files the way it is usually done, while introduces another step after compilation and before application running. Post-compile weaving is performed by the same `ajc` utility as compile-time, but with a slightly different set of parameters, as it is shown in Listing 10.

Listing 10: Post-compile aspect weaving

```
1  ajc -aspectpath jcloudscale.core-0.4.0.jar -classpath <your classpath here>
2                 -1.7 -inpath <your compiled classes or jars here>
3                 -outjar <result jar here>
```

A concrete example of such invocation is shown in Listing 11.

Listing 11: Post-compile aspect weaving example

```
1 ajc -aspectpath lib/jcloudscale.core-0.4.0.jar -classpath "lib/*"
2                 -1.7 -inpath target -outjar code.jar
```

After this, application can be started from the resulting jar as usually as it is shown in Listing 9.

Load-time weaving

Load-time weaving does not require anything specific from the code compilation stage. This allows working with code in any IDE or environment without any limitations. However, load-time weaving applies some run-time application code processing and changing, therefore influencing application performance, what might be critical.

In order to enable AspectJ runtime weaving, we need to provide an `aop.xml` file, located in `META-INF` folder in classpath. This file for aspects defined in JCloudScale may have the content shown in Listing 12.

Listing 12: AspectJ configuration for load-time weaving

```
1  <aspectj>
2      <aspects>
3        <aspect name=
4            "at.ac.tuwien.infosys.jcloudscale.aspects.CloudObjectAspect"/>
5        <aspect name=
6            "at.ac.tuwien.infosys.jcloudscale.aspects.JCloudScaleManagementAspect"/>
7      </aspects>
8
9      <weaver options="-verbose">
10         <!-- Ignore all classes within -->
11         <exclude within="javax..*"/>
12         <exclude within="java..*"/>
13         <exclude within="org.aspectj..*"/>
14         <exclude within="at.ac.tuwien.infosys.jcloudscale..*"/>
15
16         <!-- Process classes within (place your packages instead of 'test')-->
17         <include within="test..*"/>
18     </weaver>
19 </aspectj>
```

Additionally, you need an aspectj weaver[5], that provides necessary functionality for AspectJ runtime weaving. Finally, we need to add `-javaagent` JVM option to application startup, as it is shown in Listing 13.

Listing 13: Application startup with load-time aspect weaving

```
1  java -javaagent:aspectjweaver.jar -cp<specify your classpath here>
2          <specify your main class here>
```

After you run this command, your application should run with JCloudScale aspects applied.

Interacting With Cloud Objects

As is the case for any object in Java, clients can fundamentally interact with cloud objects in two different ways: (1) invoking methods of cloud objects, and (2) getting and setting member fields directly. Additionally, cloud objects (more concretely, the classes defining cloud objects) may contain static fields and static methods. It is important for users to understand what technically happens in JCloudScale in each of those cases.

- *Client invokes a (non-static) method*: JCloudScale will intercept this method call and schedule its execution on one of the remote hosts. The client will block until the cloud host returns the result of this invocation (or signals completion in case of void methods) (see lines 1–2 in Listing 14).

[5] `http://repo1.maven.org/maven2/org/aspectj/aspectjweaver/1.7.0/aspectjweaver-1.7.0.jar`

- *Client sets a (non-static) field*: JCloudScale will intercept this set operation and instruct the cloud host that is responsible for this object to set the value in his copy instead. The proxy in the client VM maintains the old value. The client will block until the value is successfully changed on the server. (see line 4 in Listing 14).
- *Client gets a (non-static) field*: JCloudScale will intercept this get operation and request the current value from the cloud host that is responsible for this object. This value is returned to the client. The proxy in the client VM does not change. (see line 6 in Listing 14).
- *Client invokes a static method*: JCloudScale will not intercept this operation. The static method will execute in the client VM.(see line 8 in Listing 14)
- *Client gets or sets a static field*: JCloudScale will not intercept this operation. The static field in the client VM will be used. (see lines 10-11 in Listing 14)

Listing 14: Interaction examples with JCloudScale

```
1
2  CloudObject co = new CloudObject();
3  co.invokeMe();
4
5  co.publicField = "hugo";
6
7  System.out.println(co.publicField);
8
9  CloudObject.invokeMeStatically();
10
11 CloudObject.staticPublicField = "hugo_static";
12 System.out.println(CloudObject.staticPublicField);
```

Additionally, there is also another case that needs to be discussed in this place. Sometimes, cloud objects (i.e., code running on a cloud host, not the client) might want to get or set the value of static fields, as it is shown in Listing 15.

Listing 15: Static fields access from cloud object

```
1  @CloudObject
2  public class MyCO {
3
4   public static String someValue;
5
6   public void doSomething() {
7          someValue = "newValue";
8   }
9  }
```

This is somewhat problematic, as the semantics outlined so far will likely not be what the author of this code intended. To be concrete, in this case, the static field

`someValue` is what we call *JVM-local* i.e., every remote host (and the client JVM, if this is relevant) have a separate value for `someValue`, and the value is not synchronized between different hosts. Put differently, the value of this field is depending on which host a cloud object is physically deployed on. This is generally a *bad thing* in JCloudScale. Hence, it is possible to explicitly demark (non-final) shared static fields in cloud objects using the `@CloudGlobal` annotation (see Listing 16).

Listing 16: Static fields access with @CloudGlobal annotation

```
1  @CloudObject
2  public class MyCO {
3
4   @CloudGlobal
5   public static String someValue;
6
7   public void doSomething() {
8           someValue = "newValue";
9   }
10
11 }
```

The semantics of interacting with cloud-global fields are as follows:

- *Cloud object sets a cloud-global field*: JCloudScale will intercept this set operation and instruct the client via callback that it should set this value instead. The cloud object will block until the value is successfully changed on client-side.

- *Cloud object gets a cloud-global field*: JCloudScale will intercept this get operation and request the current value from the client. This value is returned.

Likely, this will capture the intend of the author of the above code snippet better. However, users should keep in mind that getting and setting cloud-global fields involves remoting and is hence significantly more expensive than regular static field access. Furthermore, note that getting and setting cloud-global fields is by default just as unsynchronized as interaction with regular static fields. If one cloud object sets a static field, there is no guarantee that another cloud object will not swoop in and override this value immediately.

Warning: using the reflection API for interacting with cloud objects is problematic, as it will partially circumvent the mechanisms we use for intercepting method invocations, and get and set operations. Unfortunately, many libraries and third-party middleware use reflection internally, for instance to create objects on the fly. Hence, we have gone through some pain to make the basic uses cases of JCloudScale play nicely with reflection, however, in some cases (especially in combination with the by-reference semantics) users may encounter bugs and unexpected behavior when using reflection.

Passing Parameters By-Value and By-Reference

Whenever data is passed between regular Java objects running in the JVM of the client application and cloud objects (for instance, as parameters or return values of method invocations of cloud objects, or as values of fields of cloud objects), two different semantics can be used: *copy-by-value* or *by-reference*. The following simple rules apply.

Data is passed **by-value** iff it is:

- a Java primitive type (`int`, `short`, etc.)
- wrapper of a primitive type (`Integer`, `Short`, etc.)
- of type `String`
- Enum Type
- either class, invocation or invocation parameter is annotated with `@ByValueParameter`

In all other cases, data is passed *by-reference*. Demonstration examples are shown in Listing 17.

Note that by-value data passing requires the object to be serializable (i.e., to implement the `Serializable` interface). For by-reference data, this is not required as such data is never actually sent over the wire. However, types used for by-reference data passing need to provide a default no-arg constructor.

Parameters that are passed by-reference continue to exist only within the client JVM. Hence, invoking methods of by-reference parameters always leads to a callback to the client application. Users should keep this in mind when interacting with by-reference parameters (i.e., in general, invocations to by-reference parameters should be minimized).

Warning: at the moment, we are not providing any convenience functions for deciding on by-reference or by-value in addition to the simple rules stated above. For instance, if you declare a field as by-value, it will not automatically be considered by-value in other contexts (for instance, when using this field as return value of a getter). Declare by-value explicitly whenever you want data to be serialized or annotate declared parameter type itself if you want it to be passed by-value in all cases.

Restrictions on Cloud Objects and By-Reference Classes

Whenever user constructs the new instance of Cloud Object or passes parameter to the Cloud Object invocation by reference, JCloudScale constructs a proxy object that is used to intercept all invocations to this object with the help of CGLib[6]. This approach simplifies the code and extends possibilities, but has some limitations. Here's the list of the things that users should be aware of.

Both, Cloud Objects and objects passed by-reference:

[6] `http://cglib.sourceforge.net/`

Listing 17: Parameters passing examples in JCloudScale

```
1  @CloudObject
2  public class MyCO {
3
4   @ByValueParameter
5   public MyComplexObject field1; // by-value
6
7   public String field2; // by-value
8
9   public MyComplexObject field3; // by-reference
10
11  public @ByValueParameter MyComplexObject getSomething() {
12      ... // return value by-value
13  }
14
15  public void doSomething(MyComplexObject  parameter) {
16      ... // parameter by-reference
17  }
18
19  @ByValueParameter
20  static class MyParameter implements Serializable {
21    // class is annotated with @ByValueParameter,
22    // all invocations where this type will be involved will be processed by-value.
23  }
24
25  private MyParameter updateMyParameter(MyParameter param) {
26   ... //return value and parameter are passed by-value
27       // as the class MyParameter is annotated with @ByValueParameter
28  }
29
30  // However, even if MyParameter will be passed here,
31  // parameter will be passed by-reference!
32  public void updateMyParameter(Object obj) {
33
34  }
```

- *Must have* empty (no-args) constructor. (this limitation might be loosened in future);

- *Must not* use methods `finalize()` and `clone()` as they are not handled correctly. (this limitation might be loosened in future)

- *Must not* use any finalized methods (e.g., `getClass()`, `notify()`, `wait()`, etc. or user-defined methods with modifier final) as they won't be intercepted at all.

- *Should avoid* whenever it is possible to use these objects in hash maps or sorting lists as each invocation to methods `compareTo()`, `equals()` and `hashCode()` with require remote call and will slow down execution a lot. Consider to base sorting and hashing on some by-value passed parameter or value that identifies this object. (e.g., `String`, `int`).

JCloudScale Configuration

Creating Configuration

JCloudScale can be configured either directly from code (likely the easier version), or via an XML configuration file. Listing 18, illustrates a simple code snippet that showcases how to create and modify the configuration.

Listing 18: Creating custom configuration for JCloudScale

```
1 JCloudScaleConfiguration config = new JCloudScaleConfigurationBuilder().build();
2 config.common().clientLogging().setDefaultLoggingLevel(Level.OFF);
```

The configuration can also be stored to a file or loaded from a file, as it is shown in Listing 19.

Listing 19: Serializing and deserializing configuration from file

```
1 config.save(new File("config.xml"));
2 config = JCloudScaleConfiguration.load(new File("config.xml"));
```

Specifying Configuration

After you obtained an instance of `JCloudScaleConfiguration`, you have to inform JCloudScale framework to use this configuration. You can do that multiple ways, each of them has own benefits and restrictions.

1. *You can specify configuration manually.* To do this, you have to provide an instance of the `JCloudScaleConfiguration` class to the static method `setConfiguration` of the `JCloudScaleClient` class. However, you have to do that *prior to any interaction* with the JCloudScale framework, because otherwise some components might be initialized with the default configuration before you provide correct one (see Listing 20).

Listing 20: Manually defining the configuration to JCloudScale

```
1 JCloudScaleConfiguration config = ...
2 ...
3 JCloudScaleClient.setConfiguration(config);
```

2. *You can specify where to get the configuration from.* To do this, you have to set system property `jcloudscale.configuration` (specified by the public constant `JCloudScaleClient.JCLOUDSCALE_CONFIGURATION_PROPERTY`) to point

either to the file where configuration is stored or to the class that has the static parameterless method annotated with `@JCloudScaleConfigurationProvider` annotation and returns an instance of `JCloudScaleConfiguration` (see Listing 21).

Listing 21: Defining JCloudScale configuration through system property

```
1 @JCloudScaleConfigurationProvider
2 public static JCloudScaleConfiguration createConfiguration() {
3  ... // obtain configuration instance and configure it
4  return config;
5 }
```

3. *You can set system property from the code.* This approach is illustrated in Listing 22. However, it does not give you any benefits in comparison to the first option: anyways you have to do that before any interaction with the JCloudScale framework.

Listing 22: Defining JCloudScale configuration through system property from code

```
1 System.setProperty("jcloudscale.configuration", "config.xml");
```

4. *You can set system property before application starts*The real benefit of defining JCloudScale configuration through system property approach is that this property can be specified before running the application. To do this with maven, you can apply the changes shown in Listing 23 the `pom.xml` (to the `exec-maven-plugin` in the `<plugins>` configuration section). You can do that without maven as well. In this case you have to provide mentioned above line as command-line argument to the java process that starts your application, as it is shown in Listing 24.

Listing 23: Defining JCloudScale configuration through pom.xml file

```
1 <configuration>
2     <executable>java</executable>
3     <arguments>
4                 ...
5         <argument>-Djcloudscale.configuration=config.xml</argument>
6         ...
7     </arguments>
8 </configuration>
```

The last configuration specification approach is recommended, as it minimizes amount of possible issues and problems with configuration specification. However, in order to ensure that your configuration is indeed used by JCloudScale. verify that your changes to configuration influence JCloudScale behavior. For example, you can set logging to INFO or ALL and see that the amount of logging messages significantly increased.

Listing 24: Defining JCloudScale configuration through system property without maven

```
1
2  java -Djcloudscale.configuration=config.xml app.main.Class
```

Also you should see the following message: *INFO:* JCloudScale *successfully loaded configuration from <your configuration source here>.*

Additionally, you may verify if your configuration is indeed used by all JCloudScale components. If you see the following message (ensure WARNING logging is enabled in configuration you provide), than you are either setting configuration multiple times (e.g., from system property and from the code) or providing configuration too late and some components are already using default one: *WARNING:* JCloudScale *configuration redefinition: Replacing configuration instance. Some components might be still using the previous version of the configuration.*

Configuration Structure

To understand better what can be configured within the JCloudScale framework, here is the complete list of the configuration modules with short explanations. Some modules will be explained in more detail below.

1. *Common Configuration*: contains parameters that are shared by client and server.
    a) *Class Loader Configuration*: contains type and configuration specific to the appropriate class loader. Default implementation is the `CachingClassLoaderConfiguration`.
    b) *Client Id*: The unique identifier of the client that allows server to distinguish between clients and communicate with them. For example, when application restarts, cloud host can detect that this is already a different application run.
    c) *Client Logging Configuration*: The configuration of the Logging of the JCloudScale components that work on the client.
    d) *Communication Configuration*: The configuration of the Message Queue connection and data transferring parameters.
    e) *Monitoring Configuration*: The configuration of the JCloudScale state monitoring and events processing.
    f) *Scaling Policy*: A user-provided Java class that specifies the rules how the Cloud Host is selected for the new instance of the Cloud Object. By default, `HostPerObjectScalingPolicy` is used.
    g) *UI Configuration*: The configuration of the management interface, which allows monitoring what is actually happening within the application.

2. *Server Configuration*: contains parameters that are specific for the server.

    a) *Cloud Platform Configuration*: The configuration specific for the selected platform. By default it is represented by `LocalCloudPlatformConfiguration` to allow application testing on local machine. In order to make use of the real cloud, it should be replaced by an instance of `EC2CloudPlatformConfiguration` or `OpenstackCloudPlatformConfiguration`.

    b) *Server Logging Configuration*: The configuration of the Logging for the JCloudScale components that work on the server.

3. *Version*: the version of the JCloudScale platform that created this configuration object.

Writing Scaling Policies

JCloudScale framework itself cannot detect how Cloud Objects should be mapped to the Cloud Hosts and how many hosts should be used. Hence, you should select or write a scaling policy that fits your needs. You can use predefined scaling policies from the package `at.ac.tuwien.infosys.jcloudscale.policy.sample`, but to get really the best performance and resource usage, you might need to write your own scaling policy that will decide which host to select based on the number of already running machines, size of the task, planned amount of machines, current load of each machine or some other rules that might be important for your application.

To create your own scaling policy, you have to create a class that extends `at.ac.tuwien.infosys.jcloudscale.policy.AbstractScalingPolicy` class and provide an instance of this class to JCloudScale configuration, as it is shown in Listing 25.

As you can see, you have to implement two methods that will allow JCloudScale to scale up and down. The first method, `selectHost` will be called on each Cloud Object creation and has to answer the question *"Which host should this object be deployed to"*? It receives as parameters a descriptor of the new Cloud Object (containing unique ID of the cloud object, it's class, and proxy object that will be used to perform all invocations after object will be deployed to the cloud host) and reference to the cloud hosts' pool that allows to perform various manipulations with the set of available hosts (like, get available hosts, start new host or shutdown existing host). JCloudScale expects this method to return the reference to the cloud host that `newCloudObject` should be deployed on.

Second method from this interface has slightly different semantics. As cloud providers usually have billing based on time periods (billing period, usually a hour) starting from the host startup time, JCloudScale allows user to scale down similarly. Whenever the new host is started, `scaleDown` method will be called periodically to determine if the host specified by `scaledHost` parameter should be scaled down now or not. If the user answers true, all objects running on this host will be destroyed and the host will be

Listing 25: Implementing custom scaling policy in JCloudScale

```
1  public abstract class AbstractScalingPolicy
2  {
3      /**
4       * Selects the virtual host for the new cloud object.
5       * @param newCloudObject The descriptor of the new
6           *              cloud object.
7       * @param hostPool The host pool that contains the
8           *              set of available hosts and allows
9       * to perform additional scaling operations on these hosts.
10      * @return The host that should be used to deploy
11          *              new cloud object.
12      */
13     public abstract IHost selectHost(ClientCloudObject newCloudObject,
14                                      IHostPool hostPool);
15
16     /**
17      * This method is called periodically
18      * (with the period specified by <b>scaleDownInterval</b>
19          *              from common configuration)
20      * to perform scaling down and cloud usage optimization
21          *              tasks.
22      * @param hostPool The host pool that contains the set
23          *              of available hosts and allows
24      * to perform additional scaling operations on these hosts.
25      * @param scaledHost Indicates the host that reached
26          *              the next <b>scaleDownInterval</b>.
27      * @return <b>true</b> if the specified host should be
28          *              scaled down. Otherwise,
29      * if the specified host has to stay online for another
30          *              scaling interval, <b>false</b>.
31      */
32     public abstract boolean scaleDown(IHost scaledHost, IHostPool hostPool);
33 }
```

terminated (therefore, you should either never terminate hosts with objects running on them or never use objects deployed to this host after host termination). Scaling down method invocation interval can be configured within common section of configuration.

The decision in both methods should be based on the type of the object, expected operations on this object and current state of running cloud hosts. For each cloud host you have following information available:

- *Host ID*, that allows identifying host uniquely. This property is available only when host is online.
- *Host IP Address*, that gives you the IP address of the host. This property is available only when host is online.
- *Startup Time*, that specifies when the host was started up (if the host is running already).
- *Last Request Time*, that specifies when was the last interaction with this host.

- *Online*, that allows you to determine if host is running. If this value is false, the host may be either still starting up or already shutting down. You can detect this by checking Startup Time value: if host did not start yet, this value will be empty.

- *Cloud Objects*, that allows you to count or iterate cloud objects that are currently deployed on this host.

- *Current CPU Load/Current RAM Usage*, that allow you to determine the last measurement of CPU and RAM usage on this host. To be able to use this values, you have to enable monitoring (see configuration section). In addition, you should be aware that these values are not actual and show the last reported usage, what may different from the current values. The interval to measure these and other metrics can be configured within monitoring configuration as well.

Local vs. Cloud Deployment

Testing cloud applications can be tricky, as debugging physically disparate applications is not an easy task. Hence, JCloudScale provides two separate modes of deployment. Firstly, the *local deployment* mode is your default for building and testing applications. In local deployment, separate JVM instances are used instead of actual remote hosts. That is, whenever an application would normally want to request a new host from the cloud, it starts a new JVM on localhost and deploys the cloud object there. No actual distribution over multiple hosts happens.

If the application is working as expected in local deployment mode, you can switch to *cloud deployment* mode, which will actually physically distribute the application. Currently JCloudScale supports Openstack and Amazon EC2 as cloud deployment platform. In this mode, all platform-specific operations (e.g., machine startup/shutdown) will be performed over the JClouds API[7], while all other communication with cloud hosts and cloud objects will happen over the message queue.

Configuring Message Queue Server

While in local deployment mode JCloudScale manages Message Queue Server seamlessly for the application, in cloud deployment mode it is recommended to configure ActiveMQ Server[8] manually or use preconfigured image for selected cloud platform. When the Message Queue server is up and running, configure JCloudScale to access it (e.g., by using `withMQServer` method of `JCloudScaleConfigurationBuilder` configuration class). Additionally, ensure that Message Queue server is accessible, by the configured hostname and port, to other cloud hosts that will be started by JCloudScale in order to scale your application, as otherwise your application will fail with timeout exception waiting for cloud hosts to start and connect to message queue.

[7] `http://jclouds.apache.org/`
[8] `http://activemq.apache.org/`

JCloudScale-based Application Architecture

In general, any JCloudScale-based solution consists of cloud workers (cloud hosts that host and execute cloud objects), *messaging server* (that handles and routes all communication between components) and *initial application* (that coordinates the whole solution and starts any interaction). Generally, it is recommended (but not strictly required) to deploy all parts of your application within the cloud, as this decreases amount of communication that needs to be performed beyond the boundaries of the cloud and dramatically speeds up your application.

Each cloud worker is always deployed on the separate cloud host and should be the only resource-intensive process on that particular host. To remove any influence and simplify solution architecture, messaging server and initial application should also be deployed on the separate hosts. However, if initial application does not require a lot of resources, messaging server and initial application can be co-located on the same host. This may increase performance (as a significant part of messaging will be performed within the localhost) and decrease overall maintenance and costs (as there's one server less to maintain).

Using Openstack Cloud Platform

Before you can start using Openstack from JCloudScale-based application, you have to create a virtual machine image that has an instance of JCloudScale Server *of the same version* running.

To get such an image, either refer to our tutorial on building server images, or download our pre-built image (0.4.0).

If you don't want to specify the name or id of the created image in the application configuration explicitly, you have to name it accordingly to JCloudScale conventions. If neither name nor id of the virtual machine image is specified, JCloudScale tries to find the image that is named `"JCloudScale"+"_v"+JCloudScaleConfiguration.CS_VERSION` (e.g., `JCloudScale_v0.4.0` if application is using JCloudScale with version 0.4.0). Additionally, an Apache ActiveMQ instance needs to be available that is accessible both from the client and from cloud hosts.

In addition, to speedup application startup process, you can use *static* JCloudScale *instances*. Static instance is the virtual machine that has JCloudScale service running and is started prior to application startup. At startup, JCloudScale queries for running cloud hosts through message queue. Hosts that answered are used as static hosts. Therefore, to become a static instance, JCloudScale service should be properly configured to connect to the correct message queue server.

Static instances will not be shut down on application termination unless application shuts down them explicitly. To start a new static host, you can use JCloudScale *Static Host Management Tool* that allows you to start and stop available static hosts. Note, that this tool requires a file with serialized configuration (see configuration section) that will be used by your main application to configure static hosts accordingly. (The tool is under development and will be available soon)

Note that you do not need access to a real IaaS cloud for local deployment. However, access credentials to an IaaS cloud are required as soon as you switch to cloud deployment.

Further note that local deployment is *not meant for production*. Realistically, we cannot think of a good reason to ever write an application with JCloudScale if you do not intend to deploy it to an actual cloud. Local deployment is a tool to ease the development and testing process, and nothing more.

Listing 26: Selecting local deployment mode

```
1 new JCloudScaleConfigurationBuilder()
2   .build();
```

Selecting the deployment mode is part of JCloudScale configuration. Just selecting local deployment without further customization can be done as in Listing 26. Local deployment can be additionally configured as it is shown in Listing 27.

Listing 27: Selecting local deployment mode with additional configuration

```
1 new JCloudScaleConfigurationBuilder(new LocalCloudPlatformConfiguration()
2         .withStartupDirectory("target/")
3         .withClasspath("lib/*")
4         .withJavaHeapSizeInMB(6000))
5     .build();
```

Openstack mode is selected similarly, but requires some configuration properties to be able to operate with running instances in the cloud. The simplest configuration looks as it is shown in Listing 28.

Listing 28: Selecting openstack deployment mode

```
1 new JCloudScaleConfigurationBuilder(
2         new OpenstackCloudPlatformConfiguration
3             (identityPublicURL, tenantName, imageName, login, password))
4  .build();
```

To avoid these cumbersome and huge list of parameters, you can use another overload that accepts instance of Properties object with all Openstack-specific parameters as it is shown in Listing 29.

The properties that are expected are:

- `OS_AUTH_URL` that specifies Authentication URL.
- `OS_TENANT_NAME` that specifies Openstack tennant name.
- `OS_USERNAME` that specifies Openstack user login.
- `OS_PASSWORD` that specifies Openstack user password.

Listing 29: Selecting openstack deployment mode using properties-based configuration

```
1  Properties openstackProperties = new Properties();
2  ... //fill in properties here
3  new JCloudScaleConfigurationBuilder(
4    new OpenstackCloudPlatformConfiguration(openstackProperties))
5    .build();
```

In addition, for testing or prototyping purposes mostly, as it hardly makes sense to store access credentials in plain text, JCloudScale allows to specify the path to the file where these properties can be loaded from (see Listing 30).

Listing 30: Selecting openstack deployment mode using properties-based configuration from file

```
1  String openstackPropertiesFilename = "...";
2  new JCloudScaleConfigurationBuilder(
3    new OpenstackCloudPlatformConfiguration(openstackPropertiesFilename))
4    .build();
```

Additionally, you can conveniently specify other configuration properties of JCloudScale or cloud platform as it is shown in Listing 31

Listing 31: Selecting openstack deployment mode with additional configuration

```
1  new JCloudScaleConfigurationBuilder(
2      new OpenstackCloudPlatformConfiguration
3          (identityPublicURL, tenantName, imageName, login, password)
4          .withSshKey(sshKeyName)
5          .withInstanceImage(imageName))
6    .with(scalingPolicyInstance)
7    .withLogging(loggingLevel)
8   .withMqServer(hostname, port)
9   .withMonitoring(true)
10   .build();
```

All other parts of the application are not dependent on the selected platform and everything should work the same way independently of selected cloud platform. However, after switching for cloud deployment mode, the application will be actually distributed over multiple machines with, possibly, different operation system and file system, what might cause difference in execution and behavior.

Deployment in EC2

To the largest extend, deployment in EC2 works exactly like explained above for Openstack. The main difference is that instead of using `OpenstackCloudPlatformConfiguration`, an instance of `EC2CloudPlatformConfiguration` should be passed to the JCloud-

Scale configuration. This configuration mostly has the same property values as explained for Openstack. The core difference is that it requires the availability of an AWS properties file with the access and secret keys, as it is shown in Listing 32

Listing 32: Defining AWS properties file

```
1 accessKey = YOUR_ACCESS_KEY_IN_AWS
2 secretKey = YOUR_SECRET_KEY_IN_AWS
```

Furthermore, an AMI that implements the JCloudScale server component is required. For version 0.4.0, you can use the public image `ami-4f645e26` (US East, N. Virginia region) or `ami-80ce38f7` (EU, Ireland region) in EC2. Alternatively, you can easily build your own image based on our tutorial.

Event-Based Monitoring

Writing useful scaling policies is often the hardest part of building your JCloudScale application. For simple policies, users can use the information provided directly through the API, as discussed in Scaling Policies documentation section. However, oftentimes, users may want to scale based on data that is more application-specific than CPU or RAM utilization (for instance based on the number of violations of Service Level Agreements on each host). Such use cases are covered by JCloudScale event-based monitoring interface.

JCloudScale integrates a powerful Esper[9]-based complex event processing (CEP) engine. Via Esper, useful metrics can be defined as CEP statements on streams of monitoring data produced by JCloudScale and the cloud objects.

Available Default Events

If monitoring is enabled via the configuration, JCloudScale automatically triggers a set of predefined events that can be used in CEP statements. Monitoring is enabled as it is shown in Listing 33.

Listing 33: JCloudScale configuration that enables monitoring

```
1 new JCloudScaleConfigurationBuilder()
2   .withMonitoring(true)
3   .build();
```

The following predefined events then become available:

- State Events

[9] `http://esper.codehaus.org/`

  - `monitoring.CPUEvent`: triggered periodically, and contains the current CPU load at a given host.
  - `monitoring.RAMEvent`: triggered periodically, and contains the current utilization of the Java heap space of the host JVM.

- Cloud Object Events
  - `monitoring.ObjectCreatedEvent`: triggered whenever a new instance of a cloud object is deployed.
  - `monitoring.ObjectDestroyedEvent`: triggered whenever an instance of a cloud object is destroyed.
  - `monitoring.ExecutionStartedEvent`: triggered whenever a method of a cloud object starts to execute.
  - `monitoring.ExecutionFinishedEvent`: triggered whenever a method of a cloud object completes successfully.
  - `monitoring.ExecutionFailedEvent`: triggered whenever an execution of a cloud object throws an exception.

Note: on server-side, the generation of State Events (`CPUEvent`, `RAMEvent`) can be obtained in multiple ways. By default, JCloudScale tries using the best available approach. The most precise and complete implementation is the one based on a Java-based monitoring library called Sigar[10], which uses the Java Native Interface (JNI) to call operating system functions. If you are using the *local* development mode of JCloudScale and you want to receive State Events generated by Sigar implementation, you need to configure `LocalCloudPlatformConfiguration` with the path to a correct native `libsigar` binaries for your platform. To do this, you have to specify `java.library.path` system variable as it is shown in Listing 34. If you are building a server image yourself, don't forget to include this system variable into the script you use to start JCloudScale server process.

Listing 34: Java library path definition using custom JVM arguments

```
1 localCloudPlatformConfiguration.addCustomJVMArgs(
2         "-Djava.library.path=<folder_with_sigar_native_libraries>");
```

Triggering Custom Events

In addition to the predefined events listed above, users may want to define and trigger their own evens from their cloud objects. To do so, users should first define their event classes as regular Java beans, which subclass `monitoring.Event`. An example is given in Listing 35.

[10] `http://www.hyperic.com/products/sigar`

Listing 35: Custom JCloudScale event definition

```
1  public class DurationEvent extends Event {
2
3      private static final long serialVersionUID = 1L;
4
5      private long duration;
6      private String hostId;
7
8      public DurationEvent() {}
9
10     public DurationEvent(long duration, String hostId) {
11         this.duration = duration;
12         this.hostId = hostId;
13     }
14
15     public long getDuration() {
16         return duration;
17     }
18     public void setDuration(long duration) {
19         this.duration = duration;
20     }
21     public String getHostId() {
22         return hostId;
23     }
24     public void setHostId(String hostId) {
25         this.hostId = hostId;
26     }
27 }
```

Before this new event can be triggered and used in CEP statements, it needs to be registered to the event engine. This is also done as the part of the configuration (see Listing 36).

Listing 36: Custom JCloudScale event registration

```
1  new JCloudScaleConfigurationBuilder()
2    .withMonitoring(true)
3    .withMonitoringEvents(DurationEvent.class)
4    .build();
```

Triggering such event in cloud objects is rather easy. Cloud objects can let JCloudScale inject event sinks, which new events are then written to as it is demonstrated in Listing 37.

Scaling Based on Events

Both, predefined and custom, events can be used in monitoring metrics within scaling policies. Before usage, monitoring metrics need to be registered with JCloudScale (see Listing 38).

Listing 37: Custom JCloudScale event creation

```
1  @CloudObject
2  public class MyCloudObject {
3
4   @EventSink
5   private IEventSink events;
6
7   public void myMethod() {
8     events.trigger(
9       new DurationEvent(DURATION, HOST_ID)
10    );
11  }
12 }
```

Listing 38: Expected JCloudScale custom metric registration

```
1  MonitoringMetric metric = new MonitoringMetric();
2  metric.setName("AvgProcessingTimeMetric");
3  metric.setEpl(
4    "select avg(duration) as avg_dur from ClassificationDurationEvent.win:time(10 sec)"
5  );
6  EventCorrelationEngine.getInstance().registerMetric(metric);
```

Once the metric is registered, current (as well as historical) values of the metric can be retrieved as it is shown in Listing 39.

Listing 39: Received event values retrieval

```
1  HashMap mapValue = (HashMap) EventCorrelationEngine.getInstance()
2    .getMetricsDatabase().getLastValue("AvgProcessingTimeMetric");
3  double avgDuration = (Double) mapValue.get("avg_dur");
```

If a metric should not be monitored any longer, it should be unregistered (see Listing 40).

Listing 40: Custom metric unregistration

```
1  EventCorrelationEngine.getInstance().unregisterMetric("AvgProcessingTimeMetric");
```

The `epl` field in `MonitoringMetric` can contain any valid Esper CEP statement. At this point, we do not provide a complete documentation of what users can do with Esper, but check the very exhaustive online documentation on the Esper web page.

JCloudScale API

Typically, JCloudScale applications follow a declarative approach, where objects that should be scaled are annotated with the `@CloudObject` annotation. In most use cases, this approach works fine. However, occasionally, users might want to follow a more traditional and imperative approach.

JCloudScale supports the imperative programming style via the `at.ac.tuwien.infosys.jcloudscale.api.CloudObjects` API. For most users, the most relevant method provided by this API is `CloudObjects.create(...)`, which allows to use an instance of arbitrary type as cloud object as it is shown in Listing 41.

Listing 41: Creating cloud object through API

```
1 MyRegularObject myObject = CloudObjects.create(
2                            MyRegularObject.class, "constructorParam");
```

Note that `MyRegularObject` is not annotated with `@CloudObject`, but will behave like an instance of an annotated type from now on. However, evidently, destruction of such objects also needs to be triggered explicitly (see Listing 42).

Listing 42: Destruction of cloud object through API

```
1 CloudObjects.destroy(myObject);
```

Advantages and Disadvantages

This syntax has a a number of advantages and disadvantages as compared to the declarative, annotations-based programming model.

- Advantages
  - *Flexible*: unlike the declarative model, which follows more of an all-or-nothing mentality, the imperative programming model allows users to specify on instance level which instances should be deployed to the cloud and which should remain regular local Java objects. Note that this can also be implemented (somewhat cumbersomely) in the declarative model using local constructors (i.e., constructors with the @Local annotation).
  - *Supports third-party types*: one disadvantage of the declarative model is that it requires source-code access to the type used as cloud object (to add the required annotations). The imperative model allows to deploy instances of types which user cannot modify.
- Disadvantages

– *Intrusive*: the biggest disadvantage of this model is that it couples users much more strongly to JCloudScale than the declarative model. Typically, any JCloudScale application can be run without JCloudScale simply by disabling AspectJ weaving during build. With the imperative model, this is not quite as straight-forward anymore.
– *Separation-of-concerns*: one of the claims of JCloudScale is that it enables a clean separation of concerns, with JCloudScale handling cloud deployment and the application handling the actual business logics. With the imperative model, this separation of concerns is somewhat more diluted, as the application now explicitly contains code to create and destroy cloud objects.

As a general rule of thumb, we suggest using the declarative model unless users have a specific use case that requires using the imperative programming model.

File Dependencies

If the Cloud Object's code or any code that is invoked from the Cloud Object needs some external files to run (e.g., configuration files, initial data or state), after the code distribution this files obviously will be missing and code will fail with `FileNotFoundException`.

To inform JCloudScale that this class has some additional file dependencies, `@FileDependency` annotation can be used. Whenever JCloudScale detects first usage of class annotated with this annotation, it transfers additional dependencies specified with this annotation to the cloud host (this happens during class loading, therefore all files should be ready at the moment of creation and deployment of the Cloud Object).

Files that should be deployed to the cloud host can be specified in 2 different ways.

1. In case the set of files is fixed and known prior to application compilation, they can be enumerated inside the annotation as it is shown in Listing 43.

Listing 43: File dependencies enumerated within the annotation

```
1 @FileDependency(files = {"1.txt", "resources/2.txt", "files/3.txt"})
2 public class ClassWithDependencies
3 {
4   ...
5 }
```

2. If the set of files that should be loaded is not known on the application compilation time or developer would prefer to populate this list dynamically, dependency provider approach can be used. To use this approach, `@FileDependency` annotation's property `dependencyProvider` should be initialized with the class that implements `IFileDependencyProvider` and provides the set of files necessary to the annotated class (see Listing 44).

Listing 44: File dependencies enumerated within the dependency provider class

```
1  @FileDependency(dependencyProvider= MyDependencyProvider.class)
2  public class ClassWithDependencies {
3    ...
4  }
5  public class MyDependencyProvider
6                  implements IFileDependencyProvider {
7      @Override
8      public DependentFile[] getDependentFiles() {
9          List<DependentFile> dependentFiles =
10                                 new ArrayList<DependentFile>();
11         File fileFolder = new File("files");
12         if(fileFolder.exists()) {
13             for(File file : fileFolder.listFiles()) {
14                 if(file.isFile())
15                     dependentFiles.add(
16                                                 new DependentFile(file.getPath()));
17             }
18         }
19         return dependentFiles.toArray(
20                                 new DependentFile[dependentFiles.size()]);
21     }
22 }
```

Note that it is preferable to keep all dependent files within application directory and specify them with the relative path from the current working directory. However, if the specified file is outside of the working directory, JCloudScale will still try to deploy it on the cloud host.

Dynamic File Dependency Handling

In case you have a dynamic set of files that you need to transport to cloud hosts, and this set of files is not known during classloading (or may change during application runtime), the approach described above will not work.

In this case, the `getResourceAsStream` method of classloaders can be used to obtain files dynamically from the client on demand.

Note that in order to be loaded successfully, the files you are trying to load have to be on the application classpath of the client. Otherwise `null` will be returned.

To use this feature, you need a JCloudScale-aware classloader. The best way to obtain such classloader in the code of a cloud object or in any dependent code is to use classloader that loaded the cloud object as it is shown in line 1 of Listing 45.

Listing 45: Manual file loading through classloader

```
1  ClassLoader myClassLoader = this.getClass().getClassLoader();
2  String fileClasspath = "1.txt";
3  InputStream fileAsStream = myClassLoader
4                          .getResourceAsStream(fileClasspath);
```

After that, application can request the input stream of the needed file and work with it normally (see line 2-3 of Listing 45). Note that as long as file you are trying to load is on classpath, you can use any classloader in client-side code (the code that is not executing on remote host). However, described above approach universal and should work for any code, thus we recommended to use it always to load resources from classpath.

Listing 46: Maven configuration to collect all JCloudScale dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation=
        "http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>jcloudscale</groupId>
    <artifactId>jcloudscale.server</artifactId>
    <version>1.0.0</version>
    <name>JCloudScale</name>
    <description>JCloudScale</description>
    <packaging>pom</packaging>

    <properties>
        <!-- Specify desired \cs version here -->
        <jCloudScaleVersion>0.4.0</jCloudScaleVersion>
        <!-- Specify the folder to store all dependencies here -->
        <libraryDirectory>lib</libraryDirectory>
    </properties>

    <dependencies>
        <dependency>
            <groupId>jcloudscale</groupId>
            <artifactId>jcloudscale.core</artifactId>
            <version>${jCloudScaleVersion}</version>
        </dependency>
    </dependencies>

    <repositories>
        <repository>
            <id>infosys-repository</id>
            <url>http://www.infosys.tuwien.ac.at/mvn</url>
        </repository>
    </repositories>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <version>2.5</version>
                <executions>
                    <execution>
                        <id>copy-dependencies</id>
                        <phase>package</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>${libraryDirectory}</outputDirectory>
                            <overWriteReleases>false</overWriteReleases>
                            <overWriteSnapshots>false</overWriteSnapshots>
                            <overWriteIfNewer>true</overWriteIfNewer>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

JCloudScale Application Development Tutorial

Obtaining JCloudScale source code

If you want to experiment with the code itself, you can check out the code as it is shown in Listing 47.

Listing 47: JCloudScale source code loading using git

```
1 git clone https://github.com/xLeitix/jcloudscale.git
```

Make sure to have Git, Java 7 and Maven 3 installed. You can run the end-to-end tests of JCloudScale as in Listing 48.

Listing 48: JCloudScale compilation and test execution

```
1 mvn clean verify -P local-tests
```

If you don't want to mess with the JCloudScale source code, following documentation explains how to work with the binaries using maven.

Introduction

If you want to try out JCloudScale with your own application and don't have time for Documentation, here's the illustrative example that shows complete JCloudScale integration process. Our example will be an application for prime numbers searching. The original ("un-cloudified") version of this application can be found at 0.initialSampleApplication[11].

This application represents a typical scalable application, which will allow us to demonstrate the main features and usage scenarios of the JCloudScale framework. The

[11] https://github.com/xLeitix/jcloudscale/blob/master/docs/0.initialSampleApplication.zip?raw=true

application is built with the Maven[12] ideology in mind, but can be run in Eclipse[13] as well. The main goal of the application is to calculate the amount of prime numbers within a specified integer range. With the default configuration, the application is supposed to execute in less than 30 seconds on modern hardware, however you can play with it and configure running time to fit your needs or wishes. You can run the application from the console with `mvn compile exec:exec` or `mvn test exec:exec` to run tests as well. (`mvn clean compile exec:exec` / `mvn clean test exec:exec` in case you want to clean and recompile application first).

The application consists of two main packages: `prime.searcher`, which represents the prime numbers searching algorithm, and `prime.scheduler`, which parallelizes the initial task to use desired amount of threads. The application entry point is in the class `prime.Main` that defines search scope and used algorithms. Look through the code and play with it: the sample should be reasonably easy to understand. After this you should be ready to walk through these few easy steps to move this application to the cloud. To understand JCloudScale better, we encourage you to apply all following modifications yourself to the clean application provided above. However, after each step there is a link to the updated version of the application as well.

Step 1: Applying JCloudScale to the Application

The presented application is simple and nicely scalable, but whenever you try to increase the range for prime numbers search, you hit the problem that the execution will take very long even on a multi-core machine, independently of the amount of threads you spawn. The only option we have to be able to scale further and receive results faster is to scale out our application by distributing it over multiple nodes, e.g., using the cloud computing paradigm.

To be able to use any features of JCloudScale, we should apply some modifications to the Maven Project File (`pom.xml`).

Adding JCloudScale Dependency

At first, we have to add the JCloudScale dependency. To do this, we open the project's `pom.xml` file and insert the following code into the `<dependencies>` section. The position of this particular dependency does not matter for maven. Also you may need to change version to match the one you actually target. Discussed code is shown in Listing 49.

In addition, as JCloudScale is not registered in public maven repositories, you need to add a reference to the TU Wien Infosys maven repository (see Listing 50) to the root of the `pom.xml` file (not into `<dependencies>` section, but create a new `<repositories>` section).

[12] `http://maven.apache.org/`
[13] `http://www.eclipse.org/`

Listing 49: JCloudScale dependency definition

```
1 <dependency>
2     <groupId>jcloudscale</groupId>
3     <artifactId>jcloudscale.core</artifactId>
4     <version>0.4.0</version>
5 </dependency>
```

Listing 50: JCloudScale repository reference

```
1 <repositories>
2     <repository>
3         <id>infosys-repository</id>
4         <url>http://www.infosys.tuwien.ac.at/mvn</url>
5     </repository>
6 </repositories>
```

Now you can try to build the project again and maven should download all necessary dependencies. As we did not change the code yet, application execution should not change. In case you have any problems, ensure that you have access to the repository (you can try to open the repository link in browser) and read carefully the error message that maven provided after the build.

Applying JCloudScale Aspects

After we added necessary dependencies and ensured that everything still works, it's time to configure AspectJ. As mentioned in the introduction, JCloudScale is using AspectJ to weave the appropriate cloud management code into the application. This happens as a separate weaving step after compilation of the actual application. To allow this, we have to add new plugin configuration to the `<plugins>` section within section `<build>` as it is shown in Listing 51.

This section tells maven that at compile and test-compile stages it should process code with AspectJ and apply aspects defined into the library specified by the group id and artifact id. If you compile project again, execution should still be the same, but in addition there should appear new tasks `aspectj-maven-plugin:1.4:compile/test-compile` with some warnings of not-applied aspects. This is everything we need to do before starting to use JCloudScale in our application. If you had some difficulties applying changes listed here, the complete source code as it should be after successfully performing all changes can be found in 1.Introduction[14].

[14] https://github.com/xLeitix/jcloudscale/blob/master/docs/1.Introduction.zip?raw=true

Listing 51: AspectJ post-compilation processing plugin

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>aspectj-maven-plugin</artifactId>
    <version>1.4</version>
    <configuration>
        <source>1.7</source>
        <target>1.7</target>
        <complianceLevel>1.7</complianceLevel>
        <verbose>true</verbose>
    </configuration>
    <executions>
        <execution>
            <configuration>
                <XnoInline>true</XnoInline>
                <aspectLibraries>
                    <aspectLibrary>
                        <groupId>jcloudscale</groupId>
                        <artifactId>jcloudscale.core</artifactId>
                    </aspectLibrary>
                </aspectLibraries>
            </configuration>
            <goals>
                <goal>compile</goal>
                <goal>test-compile</goal>
            </goals>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.7.0</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>1.7.0</version>
        </dependency>
    </dependencies>
</plugin>
```

Step 2: Selecting Cloud Objects

At this point you should have all necessary maven configuration applied to start using JCloudScale and the project should still successfully run. However, still no actual change in the behavior of the application will occur, as we have not yet designated any cloud objects. Now we will start modifying the code of the application to start using JCloudScale.

The whole scaling concept of JCloudScale works around the Cloud Objects: the instances of classes that do heavy work and are deployed on the cloud hosts.

Selecting the right classes in your application to become Cloud Objects is very important. Keep in mind that Cloud Objects are very expensive to create and invoke.

They may look like regular Java objects, but interacting with them from the rest of the application always requires remoting via the message bus.

In case of our sample application it is easy to see that the searcher class (`SimpleSearcher`) looks like a perfect candidate to become a Cloud Object. It has only 2 public methods, one of which is computation-intensive, it does not rely on any additional dependencies and depends only on 3 files within the same package (`ISearcher`, `SimpleSearcher` and `Range`). Therefore, we are going to declare this class a Cloud Object. To do this, we add the `@CloudObject` annotation on top of `SimpleSearcher` class as it is shown in Listing 52.

Listing 52: Adding @CloudObject annotation on top of the class

```
1  @CloudObject
2  public class SimpleSearcher implements ISearcher
3  {
4      ...
5  }
```

Now all non-static method invocations to the instance of the class MyCloudObject will be redirected to the appropriate cloud host and executed there seamlessly for application.

In principle, this is everything we need to do to introduce all necessary cloud-related code to deploy this object to the cloud. However, to make it work properly and not cause any problems for our application, we need to add a few more annotations.

One important thing we should care about is the life time of the Cloud Object on the cloud host: in case you want to control the life of the Cloud Objects on the cloud hosts, you should annotate some method that will be the last one you call on the Cloud Object with `@DestructCloudObject` annotation. After the invocation of this method, the cloud host will be informed that this Cloud Object can be destroyed. In the case of our demo application, `getResult()` can be used as such a method. Hence, we annotate it is shown in Listing 53.

Listing 53: Adding @DestructCloudObject annotation on top of the method

```
1  @DestructCloudObject
2  @Override
3  public long getResult()
4  {
5      return result;
6  }
```

In addition to specifying when Cloud Objects should be destroyed, we have to specify when the JCloudScale infrastructure won't be needed any more and can be shut down. To do this, you can use the `@JCloudScaleShutdown` annotation. After the execution of the method annotated with it, JCloudScale will insert necessary calls to gracefully destroy all Cloud Objects and shutdown any additional infrastructure created by the framework to communicate with the cloud hosts.

In our demo application we can see that the whole application execution is within the static main method from the `prime.Main` class. We can annotate this method with the `@JCloudScaleShutdown` annotation (see Listing 54), which will cause JCloudScale to release all resources and close all connections at the moment we exit from the main method. In your application, however, you can annotate any method that marks the point when JCloudScale is no longer needed or application is going to shut down. You can even create an empty method that you will call only to shut down JCloudScale.

Listing 54: Adding @JCloudScaleShutdown annotation on top of the method

```
1 @JCloudScaleShutdown
2 public static void main(String[] args)
3 {
4     System.out.println("Starting...");
5     ...
6 }
```

The last thing we should consider are parameters that are passed into the Cloud Object's method invocations and returned from them. These parameters can be delivered in two ways: *Copy-By-Value* (when the object is serialized and delivered to the other side, therefore creating the copy of this object there) or *By-Reference* (when only the proxy of the object is transferred and both sides can change the object simultaneously and observe results). By default, JCloudScale tries to mimic Java default behavior: passing by-value primitive types and passing by-reference all class types. However, this does not always correspond to the needs of the developers as passing by-reference introduces communication overhead to the application execution.

To influence JCloudScale defaults and pass some complex parameters by-value instead of by-reference approach, you can annotate your parameters in method execution with `@ByValueParameter` annotation as it is shown in Listing 55.

Listing 55: Adding @ByValueParameter annotation to method parameters

```
1 SimpleSearcher(@ByValueParameter Range range)
2 {
3         if(range.getFrom() <= 0|| range.getTo() <= 0)
4                 throw new RuntimeException(
5                         "Range contains negative or zero parameters.");
6         this.range = range;
7         this.result = 0;
8 }
```

Alternatively, we can annotate the type itself to always pass it by value as it is shown in Listing 56. Note, that in this case cloud hosts will operate over the copy of the passed parameter and if they change passed object, they will have to provide it back to the client explicitly to see changes there.

Congratulations! Your application is now cloud-aware. However, for now, instead of using any real cloud, it creates a new Java Virtual Machine to simulate the new cloud

Listing 56: Adding @ByValueParameter annotation to the class

```
1  @ByValueParameter
2  public class Range implements Serializable
3  {
4     ...
5  }
```

host. This approach is called the *"Local"* mode of JCloudScale framework, and is used to test the behavior of your application locally before deploying it to the cloud. As you should have noticed, the running time actually increased. This is caused by the added overhead needed to start virtual machines (one per object), deploy code there and synchronize execution. We will learn how to change the default mode of the JCloudScale framework to actually use the cloud in the following steps. The complete code that we should have at this point can be found in 2.Cloudified[15].

Step 3: JCloudScale Configuration

As you saw in the previous sections, using JCloudScale is pretty easy and does not require applying any sophisticated changes to your application. However, when you are not satisfied with the default behavior of JCloudScale, you should be able to change it to fit your needs. To do this, you have to change the default configuration of the JCloudScale framework.

Specifying configuration

To be as flexible as possible and satisfy most of the users' needs, JCloudScale can be configured in a few different ways.

In our demo application, we will specify configuration by creating special configuration providing method in the `prime.Main` class as it is shown in Listing 57.

Listing 57: Defining configuration providing method

```
1  @JCloudScaleConfigurationProvider
2  public static JCloudScaleConfiguration getConfiguration()
3  {
4      return new JCloudScaleConfigurationBuilder()
5                      .withLogging(Level.SEVERE)
6              .build();
7  }
```

To make JCloudScale use this method, we will change application startup declared in `pom.xml` file as in Listing 58.

[15] `https://github.com/xLeitix/jcloudscale/blob/master/docs/2.Cloudified.zip?raw=true`

Listing 58: Defining the source of the configuration

```
<configuration>
    <executable>java</executable>
    <arguments>
        <argument>-Djcloudscale.configuration=prime.Main
                                    </argument>
        <argument>-classpath</argument>
        <classpath />
        <argument>prime.Main</argument>
    </arguments>
</configuration>
```

Now whenever JCloudScale will need configuration, it will check system property `jcloudscale.configuration` and load it from the class specified there.

Logging Configuration

Logging is the easiest way to monitor the state of the running application and detect errors. However, when you're starting to use JCloudScale in your application, you don't have to do anything for output and logging redirection. To receive the output of your own code, you don't have to use JCloudScale logging infrastructure: by default, JCloudScale redirects all standard and error output to the client application. Therefore, if you add some output to standard or error stream from Cloud Object (`SimpleSearcher`), as it is shown in Listing 59, you should see it during execution as if it was printed locally (prefixed with the IP address of the host actually executing this object).

Listing 59: Application output from cloud host

```
System.out.println("#### In "+range+" found "+ result +" prime numbers.####");
```

You can change that behavior in the server logging configuration.

Scaling Policy

To scale application according to our needs, we need to create a scaling policy. For our demo case it will be really simple one (similar to the `SingleHostScalingPolicy` from default JCloudScale policies set), but it will give us some reasonable information on when and how scaling policies are used. In your application you will need to write more sophisticated scaling policies. The developed scaling policy is shown in Listing 60.

Cloud Platform Selection

For this sample application, we will limit ourselves with *Local* JCloudScale mode, while you may try to extend this application to use the actual cloud machines from your cloud. The only change that is needed for application is the change of configuration to know

Listing 60: Simple custom scaling policy example

```
1  @XmlRootElement
2  public class MyScalingPolicy extends AbstractScalingPolicy {
3
4      // The method is synchronized to avoid race conditions
5          // between different cloud objects
6      // being scheduled for execution at the same time.
7      @Override
8      public synchronized IHost selectHost(
9                          ClientCloudObject newCloudObject,
10                         IHostPool hostPool) {
11         if(hostPool.getHostsCount() > 0)
12         {
13             IHost selectedHost = hostPool.getHosts().iterator().next();
14             System.out.println(
15                                 "SCALING: Deploying new object "+
16                 newCloudObject.getCloudObjectClass().getName() +
17                 " on "+selectedHost.getId());
18             return selectedHost;
19         }
20         else
21         {
22             System.out.println(
23                                 "SCALING: Deploying new object "+
24                 newCloudObject.getCloudObjectClass().getName() +
25                 " on the new virtual machine.");
26             // Here we return a host started asynchronously
27                         // to minimize time inside synchronized section.
28             return hostPool.startNewHostAsync();
29         }
30     }
31
32     @Override
33     public boolean scaleDown(IHost scaledHost, IHostPool hostPool) {
34         // We will not scale down for this sample application as
35         // JCloudScale will shut down all hosts at the end,
36                 // but you may need that.
37         return false;
38     }
39 }
```

how to deploy the code into the real cloud (see Listing 61). When this configuration will

Listing 61: Openstack cloud platform selection

```
1  new JCloudScaleConfigurationBuilder(
2      new OpenstackCloudPlatformConfiguration
3          (identityPublicURL, tenantName, imageName, login, password))
4      .build();
```

be used, JCloudScale will operate on Open Stack virtual machines instead of the local Java virtual machines.

The complete source with all discussed configuration-related changes can be found at

3.Configured[16].

Using File Dependency

Let's extend our completely configured application to work even faster. If you look through the code of `SimpleSearcher`, you will see that the method `isPrime()`, that is executed quite often, does pretty stupid job: verifies if the provided number is divisible by each odd number. However, it would make more sense to try dividing only on prime numbers. This leads us to the point when we need to have some sort of prime numbers cache within which we will verify each provided number.

This prime numbers cache can be either calculated dynamically on the startup or loaded from some file. Let's go for the second approach, as it sounds more interesting. You can create cache file by modifying any version of our application to write each found number to some file (i.e., `primes.txt`). However, note that you should keep cache size reasonable, as otherwise searchers will have to load more numbers from cache than they actually need to generate.

To implement this, let's create new class `CachedNumbersSearcher`, shown in Listing 62, that uses the cache of prime numbers instead of comparing to each odd number as `SimpleSearcher` does. Note that we have to declare `range` parameter as

Listing 62: Prime numbers searching class that uses cache for small prime numbers

```
1  @CloudObject
2  public class CachedNumbersSearcher extends SimpleSearcher
3  {
4      public static final String CACHE_FILE_NAME = "primes.txt";
5      private List<Long> cachedPrimes = null;
6      private long maxCachedPrime = 0L;
7      CachedNumbersSearcher(@ByValueParameter Range range)
8      {
9          super(range);
10     }
11   ...
12 }
```

passed by value again and annotate our class as `@CloudObject`.

Also we should not load cache in the constructor as current version of the JCloudScale executes constructor code on both client and cloud machines. To load it lazily, let's create a helper method shown in Listing 63.

Now, when we loaded the cache, we have to override `isPrime` method as it is demonstrated in Listing 64 (don't forget to change visibility from `private` to `protected` in the parent class). Also don't forget to change `SearcherFactory` to return the new searcher we created.

Now we came to the main point of this section: in this class we're using an external file, while JCloudScale is completely unaware of that. This will lead to the problem

[16] `https://github.com/xLeitix/jcloudscale/blob/master/docs/3.Configured.zip?raw=true`

Listing 63: Lazy cache loading method

```
1  private void loadCache() {
2      try {
3          File cacheFile = new File(CACHE_FILE_NAME);
4          if(!cacheFile.exists())
5              throw new FileNotFoundException(
6                          "File "+CACHE_FILE_NAME+" was not found."+
7                          "CachedNumbersSearcher cannot continue");
8          cachedPrimes = new ArrayList<Long>();
9          maxCachedPrime = 0L;
10         try(Scanner scanner = new Scanner(cacheFile)) {
11             while(scanner.hasNextLong()) {
12                 long nextPrime = scanner.nextLong();
13                 cachedPrimes.add(nextPrime);
14                 if(nextPrime > maxCachedPrime)
15                     maxCachedPrime = nextPrime;
16             }
17         }
18         System.out.println("Loaded "+cachedPrimes.size()+
19                                    " prime numbers.");
20     }
21     catch(IOException ex) {
22         throw new RuntimeException("Failed to load cache", ex);
23     }
24 }
25 private List<Long> getCache() {
26     if(cachedPrimes == null)
27         loadCache();
28
29     return cachedPrimes;
30 }
```

that when this code will be executed on the cloud host, this file won't be available and application will fail with exception. To avoid this, we have to notify JCloudScale to "capture" additional file along with the code. To do this, we add `@FileDependency` as it is described in documentation and shown in Listing 65.

Now the application can run successfully in any environment as the specified file will be provided along with the code.

In case you had some difficulties, complete source code with these changes is available at 3.ConfiguredWithFileDependency[17].

[17] https://github.com/xLeitix/jcloudscale/blob/master/docs/3.ConfiguredWithFileDependency.zip?raw=true

Listing 64: Cache-aware prime searching method implementation

```
1  @Override
2  protected boolean isPrime(long number)
3  {
4      // calculating the maximum number we have to check
5      long max = (long)Math.floor(Math.sqrt(number));
6      // checking with the cached numbers
7      for(long i : getCache())
8      {
9          if(i > max)
10             break;
11         if(number % i == 0)
12             return false;
13     }
14
15     // if we checked already enough numbers, we're done
16     if(max < maxCachedPrime)
17         return true;
18
19     // otherwise we need to continue checking after
20         // biggest loaded prime number
21     for(long i = maxCachedPrime + 2; i < max; i+=2)
22         if(number % i == 0)
23             return false;
24
25     return true;
26 }
```

Listing 65: File capturing through file dependency annotation

```
1  @CloudObject
2  @FileDependency(files = {CachedNumbersSearcher.CACHE_FILE_NAME})
3  public class CachedNumbersSearcher extends SimpleSearcher
4  {
5      public static final String CACHE_FILE_NAME = "primes.txt";
6       ...
7  }
```

SPEEDL Grammar Definition

Top-level SPEEDL Formal Specification

<ScalingPolicy> ::= <SPConfigElements>

<SPConfigElements> ::= <Rule> <SPConfigElements> | <Validation> <SPTerminalStatement> | <SPTerminalStatement>

<SPTerminalStatement> ::= *"build"*

<Validation> ::= *"validate"* <ExternalValidationRules> <ValidationConfiguration>

<ExternalValidationRules> ::= <externalValidationRule> <ExternalValidationRules> | ""

<ValidationConfiguration> ::= <ValidationConfigurationElement> <ValidationConfiguration> | ""

<ValidationConfigurationElement> ::= *"abortOnWarnings"* <boolean> | *"runtimeValidation"* <boolean> | *"messageListener"* <validationMessageListener> | *"isEnabled"* <boolean>

<Rule> ::= <SchedulingRule> | <MigrationRule> | <ScaleUpRule> | <ScaleDownRule>

Task Scheduling Rules Formal Specification

Scheduling Rules Initialization

<SchedulingRule> ::= *"Schedule"* <ScheduledTaskType> <SchedulingHostFilter> <SelectedSchedulingRule> | <customSchedulingRuleImplementation>

<ScheduledTaskType> ::= *"tasks"* <allowedTaskType> | *"tasks"* <allowedTaskPredicate> | ""

<SchedulingHostFilter> ::= *"allHosts"* | *"onRandom"* <hostCount> | *"onHosts"* <hostToBooleanPredicate> | *"onHosts"* <hostTaskToBooleanPredicate> |""

Scheduling Rules

<SelectedSchedulingRule> ::= *"greedy"* <GreedyRule> | *"balance"* <BalancingRule>

<GreedyRule> ::= <BestHostSelectionSchedulingRule> | <ResourceBasedGreedySchedulingRule> | <TaskCountBasedGreedySchedulingRule> | <CustomHostScoreGreedySchedulingRule> | <CustomTaskScoreGreedySchedulingRule>

<ResourceBasedGreedySchedulingRule> ::= <CPUBasedGreedySchedulingRule> | <RAMBasedGreedySchedulingRule>

<BalancingRule> ::= <ResourceBasedBalanceSchedulingRule> | <TaskCountBasedBalanceSchedulingRule> | <CustomHostScoreBalanceSchedulingRule> | <CustomTaskScoreBalanceSchedulingRule>

<ResourceBasedBalanceSchedulingRule> ::= <CPUBasedBalanceSchedulingRule> | <RAMBasedBalanceSchedulingRule>

<TaskCountBasedBalanceSchedulingRule> ::= *"taskCount"* <TaskCountBalanceRuleConfig>

<CustomHostScoreBalanceSchedulingRule> ::= *"hostScore"* <hostToScorePredicate> <CustomHostScoreBalanceRuleConfig>

<CustomTaskScoreBalanceSchedulingRule> ::= *"taskScore"* <taskToScorePredicate> <CustomTaskScoreBalanceRuleConfig>

<CPUBasedBalanceSchedulingRule> ::= *"cpuUsage"* <CPUUsageType> <ResourceBasedBalanceRuleConfig>

<RAMBasedBalanceSchedulingRule> ::= *"ramUsage"* <RAMUsageType> <ResourceBasedBalanceRuleConfig>

<BestHostSelectionSchedulingRule> ::= *"bestHost"* <hostComparisonIntPredicate> <BestHostRuleConfig> | *"bestHost"* <hostToBooleanPredicate> <BestHostRuleConfig>

<TaskCountBasedGreedySchedulingRule> ::= *"maxTasks"* <runningTaskCount> <SchedulingCommonConfig>

<CustomHostScoreGreedySchedulingRule> ::= *"hostScore"* <hostToScorePredicate> <CustomSubscribtionRuleConfig> <CustomScoreGreedyTerminal>

<CustomTaskScoreGreedySchedulingRule> ::= *"taskScore"* <taskToScorePredicate> <CustomTaskScoreConfig> <CustomScoreGreedyTerminal>

<CPUBasedGreedySchedulingRule> ::= *"cpuUsage"* <CPUUsageType> <ResourceBasedRuleConfig> <ResourceBasedGreedyTerminal>

<RAMBasedGreedySchedulingRule> ::= *"ramUsage"* <RAMUsageType> <ResourceBasedRuleConfig> <ResourceBasedGreedyTerminal>

Scheduling Rules Configuration

<SchedulingCommonConfig> ::= *"isEnabled"* <hostsToBooleanPredicate> <SchedulingCommonConfig> | *"isEnabled"* <hostsTaskToBooleanPredicate> <SchedulingCommonConfig> | ""

<CustomSubscribtionRuleConfig> ::= *"subscribeTo"* <customEventsList> <CustomSubscribtionRuleConfig> | <SchedulingCommonConfig> <CustomSubscribtionRuleConfig> | ""

<TaskCountBalanceRuleConfig> ::= *"maxTasks"* <runningTaskCount> <TaskCountBalanceRuleConfig> | <SchedulingCommonConfig> <TaskCountBalanceRuleConfig> | ""

<CustomHostScoreBalanceRuleConfig> ::= <CustomScoreGreedyTerminal> <CustomHostScoreBalanceRuleConfig> | <CustomSubscribtionRuleConfig> <CustomHostScoreBalanceRuleConfig> | ""

<CustomTaskScoreConfig> ::= *"isScoreDynamic"* <boolean> <CustomTaskScoreConfig> | <CustomSubscribtionRuleConfig> <CustomTaskScoreConfig> | ""

<CustomTaskScoreBalanceRuleConfig> ::= <CustomScoreGreedyTerminal> <CustomTaskScoreBalanceRuleConfig> | <CustomTaskScoreConfig> <CustomTaskScoreBalanceRuleConfig> | ""

<ResourceBasedRuleConfig> ::= *"inLast"* <measurementTimeWindow> <ResourceBasedRuleConfig> | <SchedulingCommonConfig> <ResourceBasedRuleConfig> | ""

<ResourceBasedBalanceRuleConfig> ::= <ResourceBasedGreedyTerminal> <ResourceBasedBalanceRuleConfig> | <CPUBasedRuleConfig> <ResourceBasedBalanceRuleConfig> | ""

<BestHostRuleConfig> ::= *"subscribeTo"* <customEventsList> <BestHostRuleConfig> | <SchedulingCommonConfig> <BestHostRuleConfig> | ""

<ResourceBasedGreedyTerminal> ::= *"maxUsage"* <resourceUsageValue>

<CustomScoreGreedyTerminal> ::= *"maxHostScore"* <hostScore>

Task Migration Rules Formal Specification

Migration Rules Initialization

<MigrationRule> ::= *"migration"* <MigrationType> | <customMigrationRuleImplementation>

<MigrationType> ::= <MigrationHostFilter> *"integrate"* <IntegrationRule> | <MigrationHostFilter> *"optimize"* <OptimizationRule>

<MigrationHostFilter> ::= *"allHosts"* | *"hosts"* <hostToBooleanPredicate> | ""

Migration Rules

<IntegrationRule> ::= <IntegrationCPUBasedMigrationRule> | <IntegrationRAMBasedMigrationRule> | <IntegrationTaskCountMigrationRule> | <IntegrationHostScoreMigrationRule>

<OptimizationRule> ::= <OptimizationCPUBasedMigrationRule> | <OptimizationRAMBasedMigrationRule> | <OptimizationTaskCountMigrationRule> | <OptimizationHostScoreMigrationRule>

<IntegrationCPUBasedMigrationRule> ::= *"withLessCpuUsage"* <resourceUsageValue> <CPUUsageType> <MigrationCommonConfig>

<IntegrationRAMBasedMigrationRule> ::= *"withLessRamUsage"* <resourceUsageValue> <RAMUsageType> <MigrationCommonConfig>

<IntegrationTaskCountMigrationRule> ::= *"withLessTasks"* <runningTaskCount> <MigrationCommonConfig>

<IntegrationHostScoreMigrationRule> ::= *"withSmallerScore"* <hostToScorePredicate> <hostScore> <MigrationCustomScoreConfig>

<OptimizationCPUBasedMigrationRule> ::= *"withMoreCpuUsage"* <resourceUsageValue> <CPUUsageType> <OptimizationMigrationCommonConfig>

<OptimizationRAMBasedMigrationRule> ::= *"withMoreRamUsage"* <resourceUsageValue> <RAMUsageType> <OptimizationMigrationCommonConfig>

<OptimizationTaskCountMigrationRule> ::= *"withMoreTasks"* <runningTaskCount> <OptimizationMigrationCommonConfig>

<OptimizationHostScoreMigrationRule> ::= *"withHigherScore"* <hostToScorePredicate> <hostScore> <OptimizationMigrationCustomScoreConfig>

Migration Rules Configuration

<MigrationCommonConfig> ::= *"minActionInterval"* <timeInterval> <MigrationCommonConfig> | *"violationInterval"* <timeInterval> <MigrationCommonConfig> | *"inLast"* <timeInterval> <MigrationCommonConfig> | *"migrateTo"* <hostToBooleanPredicate> <MigrationCommonConfig> | *"migrateTo"* <hostsToHostPredicate> <MigrationCommonConfig> | *"arrangeDstHosts"* <hostToScorePredicate> <ordering> <MigrationCommonConfig> | *"isEnabled"* <hostsToBooleanPredicate> <MigrationCommonConfig> | *"canMigrate"* <taskToBooleanPredicate> <MigrationCommonConfig> | ""

<MigrationCustomScoreConfig> ::= *"checkOn"* <customEventsList> <MigrationCustomScoreConfig> | *"checkEvery"* <timeInterval> <MigrationCustomScoreConfig> | <MigrationCommonConfig> <MigrationCustomScoreConfig> | ""

<OptimizationMigrationCommonConfig> ::= *"arrangeTasks"* <taskToScorePredicate> <OptimizationMigrationCommonConfig> | *"migrateTop"* <runningTaskCount> <OptimizationMigrationCommonConfig> | <MigrationCommonConfig> <OptimizationMigrationCommonConfig> | ""

<OptimizationMigrationCustomScoreConfig> ::= <MigrationCustomScoreConfig> <OptimizationMigrationCustomScoreConfig> | <OptimizationMigrationCommonConfig> <OptimizationMigrationCustomScoreConfig> | ""

Scale Up Rules Formal Specification

Scale Up Rules Initialization

<ScaleUpRule> ::= *"scaleUp"* <ScaleUpHostFilter> <SelectedScaleUpRule> | <customScaleUpRuleImplementation>

<ScaleUpHostFilter> ::= *"allHosts"* | *"hosts"* <hostToBooleanPredicate> | ""

Scale Up Rules

<SelectedScaleUpRule> ::= <CPUBasedScaleUpRule> | <RAMBasedScaleUpRule> | <TaskCountScaleUpRule> | <TimeBasedScaleUpRule> | <TaskQueueLengthScaleUpRule> | <CustomMetricScaleUpRule>

<CPUBasedScaleUpRule> ::= *"whenCpuUsageHigher"* <resourceUsageValue> <CPUUsageType> <aggregationMethod> <ScaleUpCommonConfig>

<RAMBasedScaleUpRule> ::= *"whenRamUsageHigher"* <resourceUsageValue> <RAMUsageType> <aggregationMethod> <ScaleUpCommonConfig>

<TaskCountScaleUpRule> ::= *"taskCountHigher"* <runningTaskCount> <aggregationMethod> <ScaleUpCommonConfig>

<TimeBasedScaleUpRule> ::= *"at"* <timeValue> <TimeBasedScaleUpRuleConfig>

<TaskQueueLengthScaleUpRule> ::= *"queueLength"* <tasksWaitingScheduleCount> <ScaleUpCommonConfig>

<CustomMetricScaleUpRule> ::= *"when"* <hostsToBooleanPredicate> <CustomMetricScaleUpRuleConfig>

Scale Up Rules Configuration

<ScaleUpCommonConfig> ::= *"minHosts"* <hostCount> <ScaleUpCommonConfig> | *"maxHosts"* <hostCount> <ScaleUpCommonConfig> | *"newHostsType"* <hostsTypeParams> <ScaleUpCommonConfig> | *"scaleUpStep"* <hostCount> <ScaleUpCommonConfig> | *"isEnabled"* <hostsToBooleanPredicate> <ScaleUpCommonConfig> | *"minActionInterval"* <timeInterval> <ScaleUpCommonConfig> | *"ifViolatedFor"* <timeInterval> <ScaleUpCommonConfig> | *"overLastWindow"* <timeInterval> <ScaleUpCommonConfig> | ""

<TimeBasedScaleUpRuleConfig> ::= *"minHosts"* <hostCount> <TimeBasedScaleUpRuleConfig> | *"maxHosts"* <hostCount> <TimeBasedScaleUpRuleConfig> | *"newHostsType"* <hostsTypeParams> <TimeBasedScaleUpRuleConfig> | *"isEnabled"* <hostsToBooleanPredicate> <TimeBasedScaleUpRuleConfig> | ""

<CustomMetricScaleUpRuleConfig> ::= *"checkOn"* <customEventsList> <CustomMetricScaleUpRuleConfig> | *"checkEvery"* <timeInterval> <CustomMetricScaleUpRuleConfig> | <ScaleUpCommonConfig> <CustomMetricScaleUpRuleConfig> | ""

Scale Down Rules Formal Specification

Scale Down Rules Initialization

<ScaleDownRule> ::= *"scaleDown"* <SelectedScaleDownRule> | <customScaleDownRuleImplementation>

Scale Down Rules

<SelectedScaleDownRule> ::= <CPUBasedScaleDownRule> | <RAMBasedScaleDownRule> | <TaskCountBasedScaleDownRule> | <TaskQueueLengthScaleDownRule> | <HostIdleTimeScaleDownRule> | <TimeBasedScaleDownRule> | <CustomMetricScaleDownRule>

<CPUBasedScaleDownRule> ::= *"ifCpuUsageLower"* <resourceUsageValue> <CPUUsageType> <aggregationMethod> <WindowBasedScaleDownConfig>

<RAMBasedScaleDownRule> ::= *"ifRamUsageLower"* <resourceUsageValue> <RAMUsageType> <aggregationMethod> <WindowBasedScaleDownConfig>

<TaskCountBasedScaleDownRule> ::= *"runningTasks"* <runningTaskCount> <WindowBasedScaleDownConfig>

<TaskQueueLengthScaleDownRule> ::= *"queueLength"* <tasksWaitingScheduleCount> <WindowBasedScaleDownConfig>

<HostIdleTimeScaleDownRule> ::= *"hostIdle"* <timeInterval> <ScaleDownCommonConfig>

<TimeBasedScaleDownRule> ::= *"at"* <timeValue> <TimeBasedScaleDownRuleConfig>

<CustomMetricScaleDownRule> ::= *"when"* <hostToBooleanPredicate> <WindowBasedScaleDownConfig>

Scale Down Rules Configuration

<ScaleDownCommonConfig> ::= *"ifWithTasks"* <withTasksScaleDownBehavior> <ScaleDownCommonConfig> | *"checkAdvance"* <timeInterval> <ScaleDownCommonConfig> | *"minHosts"* <hostToBooleanPredicate> <hostCount> <ScaleDownCommonConfig> | *"minHosts"* <hostCount> <ScaleDownCommonConfig> | *"isEnabled"* <hostToBooleanPredicate> <ScaleDownCommonConfig> | *"isEnabled"* <hostsToBooleanPredicate> <ScaleDownCommonConfig> | *"isEnabled"* <hostsHostToBooleanPredicate> <ScaleDownCommonConfig> | *"migrateTo"* <hostsToHostPredicate> <ScaleDownCommonConfig> | ""

<WindowBasedScaleDownConfig> ::= *"overLastWindow"* <timeInterval> <WindowBasedScaleDownConfig> | <ScaleDownCommonConfig> <WindowBasedScaleDownConfig> | ""

<TimeBasedScaleDownRuleConfig> ::= *"arrangeHosts"* <hostComparisonIntPredicate> <TimeBasedScaleDownRuleConfig> | *"arrangeHosts"* <hostToScorePredicate> <TimeBasedScaleDownRuleConfig> | *"maxHosts"* <hostCount> <TimeBasedScaleDownRuleConfig> |<ScaleDownCommonConfig> <TimeBasedScaleDownRuleConfig> | ""

Enumeration Expressions Definition

<boolean> ::= *"TRUE"* | *"FALSE"*

<ordering> ::= *"ASCENDING"* | *"DESCENDING"*

<aggregationMethod> ::= *"AVERAGE"* | *"MAX"* | *"MEDIAN"* | *"MIN"* | *"SUM"* | *"DISPERSION"*

<withTasksScaleDownBehavior> ::= *"NOT_SHUTDOWN"* | *"SHUTDOWN"* | *"MIGRATE"*

<CPUUsageType> ::= *"PROCESS"* | *"SYSTEM"* | *"SYSTEM_AVG"*

<RAMUsageType> ::= *"PROCESS_FREE"* | *"PROCESS_USED"* | *"PROCESS_USED_PERCENT"* | *"SYSTEM_FREE"* | *"SYSTEM_USED"* | *"SYSTEM_USED_PERCENT"*

Constants Definition

<allowedTaskType> The value of domain-specific task type that can be scheduled using this rule.

<customEventsList> The list of domain-specific or generic events that should be considered in this rule.

<hostCount> A positive integer that defines the amount of hosts to consider.

<hostScore> A double value that defines the threshold score value for host.

<hostsTypeParams> Array of string parameters that should be passed to cloud-specific API in order to create a required cloud host.

<resourceUsageValue> Double value that defines the threshold resource usage value.

<runningTaskCount> A positive integer that defines the amount of tasks running on the particular host.

<tasksWaitingScheduleCount> A positive integer that defines the amount of tasks waiting to be scheduled.

<timeInterval> A platform-specific constant that defines the particular time interval (e.g., *"15 seconds"*, *"5 mins"*, *"TimeInterval.fromSeconds(5)"*).

<timeValue> A platform-specific definition of the time when action has to be performed (e.g., *"2015/02/02 15:02"*).

Predicates Definition

<customMigrationRuleImplementation> A custom native language implementation of the required domain-specific migration behavior.

<customScaleDownRuleImplementation> A custom native language implementation of the required domain-specific scale down behavior.

<customScaleUpRuleImplementation> A custom native language implementation of the required domain-specific scale up behavior.

<customSchedulingRuleImplementation> A custom native language implementation of the required domain-specific scheduling behavior.

<externalValidationRule> A custom native language implementation of the required domain-specific external validation behavior.

<hostComparisonIntPredicate> A custom predicate that takes two hosts as an input and returns a value that is below, equal or above zero (*"(host1, host2) -> int"*).

<hostTaskToBooleanPredicate>	A custom predicate that takes a host and a task as an input and returns boolean value that indicates whether the particular task can be processed by the host (*"(host, task) -> boolean"*).
<hostToBooleanPredicate>	A custom predicate that takes a host as an input and returns boolean value that indicates whether the particular host should be considered by the rule (*"(host) -> boolean"*).
<hostToScorePredicate>	A custom predicate that takes a host as an input and returns a double value that indicates the fittness of the particular host to perform required action (e.g., accept task for processing) (*" (host) -> double"*).
<hostsTaskToBooleanPredicate>	A custom predicate that takes a list of hosts and a task as an input and returns a boolean value that indicates whether the particular action can be performed now. (*"(hosts, task) -> boolean"*).
<hostsToBooleanPredicate>	A custom predicate that takes a list of hosts as an input and returns a boolean value that indicates whether the particular action can be performed now. (*"(hosts) -> boolean"*).
<hostsHostToBooleanPredicate>	A custom predicate that takes a list of hosts and a particular host as an input and returns a boolean value that indicates whether the particular action on a particular host can be performed now. (*"(hosts, host) -> boolean"*).
<hostsToHostPredicate>	A custom predicate that takes a list of hosts as an input and selects the host that should be used in the particular situation (*"(hosts) -> host"*).
<taskToBooleanPredicate>	A custom predicate that takes a task as an input and determines whether this task can be used in the declared action or not (*"(task) -> boolean"*).
<taskToScorePredicate>	A custom predicate that takes a task as an input and returns a double value that defines the fittness function of each task to be used in particular action or rule (*"(task) -> double"*).
<validationMessageListener>	A custom predicate that takes a validation message as an input and performs required application-specific actions (*"(message) -> void"*).

Curriculum Vitae

MSc Rostyslav Zabolotnyi

12.8.2015

Tigergasse 23-27 20/2, Vienna 1080, Austria
+43 660 166 23 19
rstzab@gmail.com
www.linkedin.com/in/rstzab

PERSONAL INFORMATION

Date of birth: 17.11.1987
Place of birth: Kyiv, Ukraine
Citizenship: Ukrainian
Gender: Male

EDUCATION

National University of "Kyiv-Mohyla Academy"

01.09.2004 – 28.06.2008
Computer Science, **Bachelor diploma**.

National University of "Kyiv-Mohyla Academy"

01.09.2008 – 28.06.2010
Information Control Systems and Technologies, **Master Diploma with honors**.

PROFESSIONAL EXPERIENCE

.NET Developer | Module cooperative, Kyiv, Ukraine

10.01.2008 – 15.09.2011

Designing and implementing SCADA-oriented software for Windows.

Used technologies: C# .NET, VB .NET, WCF, XML, SOAP, LINQ, Win API, Enterprise Library (Unity, Logging, Prism), Infragistics Controls, Dundas Charting, MS SQL, Versant DB4O, SVN.

PhD Student | Vienna University of Technology, Austria

10.10.2011 – 30.09.2015

PhD Student/Project Assistant in Distributed Systems Department.
Employed by SIMPLI-CITY EU WP7 project, **working on** jCloudScale research prototype.

Used technologies: LaTeX, Java, OSGi, Apache CXF, JMS, AspectJ, Maven, OpenStack, Amazon S3, Amazon EC2, Git.

CO-AUTHOR OF THE PAPERS:

- **Cloud Computing, Cloud Platform Engineering**

  - Rostyslav Zabolotnyi, Philipp Leitner, Hummer Waldemar, Schahram Dustdar " *JCloudScale: Closing the Gap Between IaaS and PaaS*" , ACM Transactions on Internet Technology (TOIT) journal , 2015
  - Rostyslav Zabolotnyi, Philipp Leitner, Stefan Schulte, Schahram Dustdar "*SPEEDL – A Declarative Event-Based Language for Cloud Scaling Definition*" ,IEEE SERVICES, TFoSEC visionary track, 2015
  - Rostyslav Zabolotnyi, Philipp Leitner, Schahram Dustdar "*Profiling-Based Task Scheduling for Factory-Worker Applications in Infrastructure-as-a-Service Clouds*" , Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2014
  - Rostyslav Zabolotnyi, Philipp Leitner, Schahram Dustdar "*Dynamic Program Code Distribution in Infrastructure-as-a-Service Clouds*" Workshop on Principles of Engineering Service-Oriented Systems (PESOS), co-located with ICSE, 2013
  - Philipp Leitner, Rostyslav Zabolotnyi, Alessio Gambi, Schahram Dustdar, "*A Framework and Middleware for Application-Level Cloud Bursting on Top of Infrastructure-as-a-Service Clouds*", IEEE/ACM Utility and Cloud Computing Conference (UCC), 2013.
  - Georgia Fragkiadaki, Petro Kazmirchuk, Nattakarn Phaphoom, Ourania Smyrnaki, Rostyslav Zabolotnyi "*Wireless technologies in datacenter management*", Short Papers from the SummerSOC '12, Report Number: RC25348, Date of Report: January 27, 2013

- **Cloud Computing, Software Engineering**

  - Benjamin Satzger, Rostyslav Zabolotnyi, Schahram Dustdar, Martin Gaedke, Stefan Wild, Steffen Goebel, Tobias Nestler, "*Software Engineering Leveraging the Crowd*" in Economics-Driven Software Architecture, Elsevier, 2014

- **Cloud Computing, Software Testing**

  - Alessio Gambi, Rostyslav Zabolotnyi, Schahram Dustdar "*Improving Cloud-based Continuous Integration Environments*", International Conference on Software Engineering (ICSE) 2015 (Poster)

- **Cloud Computing, Hybrid Services, Human-as-a-Service**

  - Muhammad Z.C. Candra, Rostyslav Zabolotnyi, Hong-Linh Truong, and Schahram Dustdar, "*Virtualizing Software and Human for Elastic Hybrid Services*", Web Services Handbook, Springer-Verlag, 2013.
  - Rostyslav Zabolotnyi, Hong-Linh Truong, Schahram Dustdar, "*Intelligent Request Routing in Clouds of Hybrid Services*", Advanced School on Service Oriented Computing, 2012 (Poster)

- **Social Computing**

  - Vitaliy Liptchinsky, Benjamin Satzger, Rostyslav Zabolotnyi, Schahram Dustdar "*Expressive Languages for Selecting Groups from Graph-Structured Data*" , World Wide Web Conference, Springer, 2013

COMPUTER LITERACY

☐ **Operating Systems:**

Windows 9x – 10:	Advanced user
Linux:	Basic user
Solaris:	Beginner
ReactOS:	Minor contributor

☐ **Programming Languages:**

Languages	*Experience*	*Code Examples*
C#,VB.NET:	3 years of industry experience	
Java:	4 years of experience	JCloudScale
C++:	deep interest and a hobby	UGH Burner, SEAA 2014
VB, Pascal, C:	intermediate knowledge	
Scala, F#, LISP:	intermediate knowledge	
JavaScript, Python:	beginner	

☐ **Used frameworks and Technologies:**

CCNA/CCNP:	courses during bachelor/master studies;
MPI/OpenMP:	courses during bachelor studies;
OSGi:	used as the integration solution in SIMPLI-CITY.

FOREIGN LANGUAGES

☐ Ukrainian, Russian: **native**.
☐ English: good level. (TOEFL iBT, 15 April 2011) – **100 points**.
☐ German: fair level (TU Wien, 31 May 2012) – **B1.2**

EXTRACURRICULAR ACTIVITIES

☐ Member of Ukrainian Scout Organization “Plast”
☐ Sports: riding bicycle.