FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Trustworthy Measurement and Arbitration of Service Level Agreements in the Cloud

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christian Schubert, BSc
Matrikelnummer 0925131

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Dipl.-Ing. Michael Borkowski

Wien, 1. Jänner 2018

Christian Schubert Stefan Schulte

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Trustworthy Measurement and Arbitration of Service Level Agreements in the Cloud

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Christian Schubert, BSc
Registration Number 0925131

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dr.-Ing. Stefan Schulte
Assistance: Dipl.-Ing. Michael Borkowski

Vienna, 1st January, 2018

Christian Schubert	Stefan Schulte

Erklärung zur Verfassung der Arbeit

Christian Schubert, BSc
Erdbergstraße 17, 1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Jänner 2018

Christian Schubert

Acknowledgements

I would first like to thank my advisor Assistant Prof. Dr.-Ing. Stefan Schulte of the Distributed Systems Group, who made the writing of this thesis possible. Besides my advisor, I am also grateful to Dipl.-Ing. Michael Borkowski for his exceptional support throughout the whole process of realizing this work. The regular meetings, the fast feedback on any questions that arose and the enlightening discussions were extremely helpful and motivating for me.

I thank my parents, Manfred and Ingrid, who encouraged and supported me in all concerns during my entire lifetime. Without them, the studies would not have been possible. A very special thanks goes out to my girlfriend Tamara for her endless patience, and for giving me the strength to finish my studies.

Finally, I would like to thank all other persons who accompanied me in all the years of studying, including my friends, family and fellow students.

Kurzfassung

Service-Level-Agreements sind vertragliche Vereinbarungen zwischen Anbietern und Kunden einer Dienstleistung, welche die zu erfüllende Qualität und Rahmenbedingungen der Dienstleistung festlegen. Im Bereich der IT-Services, beispielsweise bei Cloud-Services, werden diese eingesetzt, um Anforderungen in Bezug auf den Verbrauch und die Bereitstellung von Rechenressourcen zu definieren. Damit die Vertragseinhaltung überprüft werden kann, ist eine Echtzeitüberwachung der vereinbarten Qualitätsparameter erforderlich.

Ein grundlegendes Problem bei der Überwachung von Service-Level-Agreements stellt die Vertrauenswürdigkeit dar. Die Durchführung von Messungen oder Entscheidungen bei einer der beiden Vertragsparteien, d.h. Dienstanbieter oder Kunde, führt zu Vertrauensproblemen der anderen Partei, da unterschiedliche wirtschaftliche Interessen verfolgt werden.

Im Zuge dieser Arbeit wird ein Monitoring-System entwickelt, welches eine vertrauenswürdige Überwachung von Service-Level-Agreements bei Cloud-Services ermöglicht. Eine vertrauenswürdige dritte Partei fungiert dabei als Entscheidungsinstanz, welche Messungen von Anbieter- sowie Kundenseite analysiert und gegebenenfalls auftretende Vertragsverletzungen erkennt. Technologien wie aspektorientierte Programmierung oder Softwareagenten erlauben die Extraktion von Qualitätsparametern, ohne dass ein Eingriff in bestehende Software notwendig ist.

Es erfolgt eine Implementierung des vorgestellten Ansatzes, sowie eine Evaluierung in einer Public-Cloud-Testumgebung. Ergebnisse zeigen, dass die Überwachung des Systems präzise und vertrauenswürdig ist, während die Anforderungen an die Transparenz, sprich geringer Ressourcenverbrauch und wenig Intrusion, eingehalten werden.

Abstract

Service Level Agreements state commitments between service providers and customers, in which quality aspects and constraints are regulated. In the field of computing, for instance cloud computing, they are used to define requirements in terms of resource provisioning and consumption. In order to verify the contract compliance, a real-time monitoring of the agreed quality parameters is required.

A fundamental problem in the monitoring of Service Level Agreements is trustworthiness. If the measurement or arbitration is done at one of the two signature parties, i.e., service provider or customer, trust issues of the opposing party arise, since different economic interests are pursued.

In the course of this work, a framework is developed which enables a trustworthy monitoring of Service Level Agreements at cloud services. A Trusted Third Party that is employed for the arbitration analyzes measurements from both the provider and the customer to detect possibly occurring contract violations. Technologies like aspect-oriented programming or software agents are used to allow for the extraction of quality parameters without being invasive to the existing software.

An implementation of the presented approach as well as an evaluation in a public cloud test environment is carried out. The results show that the monitoring of the framework is accurate and trustworthy, while requirements for the transparency, i.e., low resource consumption and little intrusion, are met.

Contents

CHAPTER 1

Introduction

Progress in network and Internet technologies led us to the possibility to outsource computing services or even whole computing infrastructures over the Internet. Providers offer their hardware, while consumers utilize them in a service-like manner and pay the providers according to the usage. While that brings a lot of advantages to both the providers and the consumers of the services, questions about the accountability arise. What happens if, for instance, the availability of the provisioned service does not fulfill what was promised? Since services are not hosted by the consumers themselves, but by an external company, how can deviations from agreed requirements be proven? If one involved party claims to encounter violations, how can these assertions be trusted by the other party?

Especially in cloud computing, these are fundamental issues, since software and virtual hardware are provided as a service, and are therefore not maintained and monitored by the consumer.

1.1 Cloud Computing

Cloud computing is a paradigm where computing resources are offered as "on demand services", where users pay for what they need, formed on an agreement between the user and the provider [BYV$^+$09]. Resources like storage, computational power, applications or servers can be shared via the Internet. Clouds are characterized by the term elasticity, meaning that there is an autonomous resource allocation and release to scale the capacity according to the emerging load [TPD14].

The following cloud service models can be distinguished [MA15]:

Software as a Service (SaaS) In this model, a software application is hosted in the cloud by a provider. Customers can access this application on demand via Internet-enabled devices, e.g., with a Web browser on a personal computer or applications on smartphones. The target group are mainly end-users.

Platform as a Service (PaaS) This term describes a model where the users are able to obtain a hosted development platform. Therefore, developers can build and deploy custom services and Web-based applications on this platform. These applications can be developed using the provided tools and frameworks, amongst other things like storage, databases or server softwares. There is no need for the developer to maintain or configure the operating system, software licenses, etc. because everything is provided ready-to-use. However, the users are dependent on the offer since there is no possibility to access the operating system and install other tools. Furthermore, the provided virtual hardware cannot be customized by the user.

Infrastructure as a Service (IaaS) One speaks of IaaS when the user of the service is capable of accessing virtualized hardware, i.e., an IT infrastructure, over the Internet. Thus, the user has full control over the underlying operating system and the software, which is in contrast to PaaS or SaaS. The virtual hardware can often be configured as desired by the customer, e.g., by selecting CPU, memory or storage. The costs for operation may rise with the machine's performance.

Companies profit from the advantages of cloud computing since they do not have buy and maintain hardware themselves, they simply acquire and access required resources in the cloud [MA15]. The resources are dynamically scaled to the current workload, i.e., in times of increased load, the infrastructure is scaled up, and in times of decreased load, the infrastructure is scaled down, adapting the costs of operating the infrastructure to current demand. Because the infrastructure is managed by the provider, additional efforts and costs for maintenance for the consumer decrease.

Due to the dynamical nature of the cloud, providers are challenged to efficiently allocate resources for their customers. Hardware virtualization allows them to share the same resources amongst multiple users [MEMB12]. If the provider underestimates the requirements it may lead to losses of performance for some customers. Therefore, need arises for a regulation of resource provisioning between consumer and provider, so that a possible non-compliance does not result in a disadvantage for the opposing party.

1.2 Service Level Agreement Monitoring

Service Level Agreements (SLAs) play a major role in cloud computing [BYV$^+$09], although their usage is not strictly limited to this area only. They also find their application in grid computing, Service-Oriented Architectures (SOAs) or generic Web services. SLAs are negotiated between the provider and the consumer of a service and specify the relation between those two parties regarding functional and non-functional constraints the provided service has to fulfill. Possible requirements regarding the Quality of Service (QoS) include availability, response time or data throughput. The consumer may also have the interest that hardware resources like the CPU, memory or network are made available for an application as it was agreed.

SLAs aim to protect the consumers of the services, as they can also form penalties for providers, in case agreed constrains are not adhered by them. It is often the case that the consumer is awarded with credits by the provider when SLA violations are detected [Bas12]. These credits can be spent for future service payments.

The detection of SLA violations can be accomplished by real-time monitoring of the provided environment as well as monitoring the services themselves at application level. The cloud provider also has interest in monitoring, because otherwise he may be allocating too many resources for a customer/service, which may result in unnecessary costs. In addition, if the allocation was too narrow, the provider may risk non-compliance with defined SLAs, as the consumer's resource requirement can suddenly increase above the provision [MEMB12].

1.3 Motivation and Problem Statement

The company *ImageCompany* wants to outsource their image processing service into the cloud, thus, they rent a cloud server from another company called *CloudCompany* (IaaS model). Among other things, the service from *ImageCompany* performs tasks like post-processing, resizing or categorizing images. The outsourcing has economical benefits for the company, as they do not have to buy and maintain a server, but only pay per usage of the infrastructure. Figure 1.1 shows the deployment of the service.

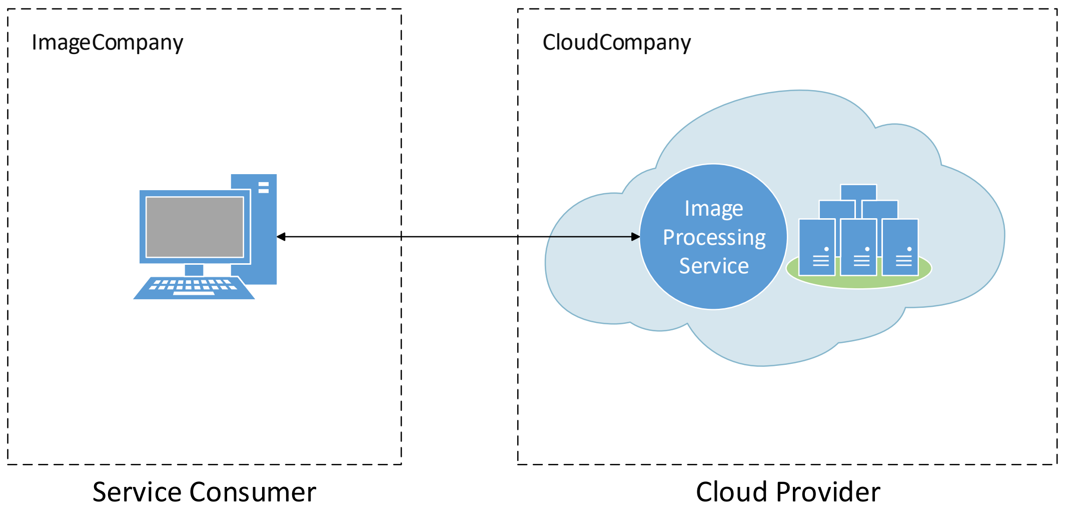


Figure 1.1: Deployment of the Image Processing Service in the Cloud

The service *Image Processing Service* is clearly no longer in the domain of the company, but installed on servers of another company. Of course, *ImageCompany* can access their service via the Internet, still, some concerns arise with this deployment model.

ImageCompany has some specific QoS requirements for their image processing service, therefore, the company negotiates an SLA with the cloud provider. This SLA is defined at application level, meaning that monitoring at application level is a requirement. The consumer may not be interested solely in the metrics of the server or infrastructure on which the service is running, but in metrics that describe the service's quality.

Many cloud providers leave the obligation of proving violations of SLAs to the customer [Bas12]. However, this is a hard task for the customer, in this case for the company *ImageCompany*, since there is a wide range of different parameters that would be required to be monitored. Some providers offer integrated monitoring tools, but they often work in a provider-specific behavior, meaning that installations on other systems will not work. Often built-in tools are not customizable or do not show enough details for the user [MEME15].

If SLA monitoring is only done by one involved party, i.e., only *ImageCompany* or only *CloudCompany* performs the monitoring, the problem of trust arises: Who decides whether the monitored values indicate correct behavior, or whether they indicate an SLA violation? Both parties seek to keep costs as low as possible and therefore have opposing interests, so there is a need to consider the arbitration process in detail.

Many approaches for monitoring of cloud services are invasive to the service, i.e., applying them requires significant modification of the application's source code [SWWM10, CLX09]. In contrast, non-invasive solutions allows to accomplish the monitoring with no or little modification of the existing infrastructure or the source code [MHJ$^+$11]. In our scenario, *ImageCompany* wants to avoid introducing extensive changes to the code of their image processing service before outsourcing, since such modifications also would cause significant cost.

All of the abovementioned issues lead us to the formulation of the problem statement for this thesis.

Problem statement: What requirements must be considered when designing and implementing a non-invasive framework that is capable of monitoring software services deployed in a cloud, in order to allow both the provider and the consumer of a service to trust the other?

This work aims to provide a framework to monitoring and arbitration of SLAs for cloud services while ensuring trustworthiness. To this end, we introduce a combination of agents and Aspect-Oriented Programming (AOP) to measure application-level metrics in a non-invasive way on both the provider and the consumer side of a service. A third party, which has the trust of both sides, is employed for the arbitration of SLAs. It provides capabilities for violation detection, but also performs grouping and aggregating of measurements to higher-level metrics at runtime, based on Complex Event Processing

(CEP). We create a reference implementation of the proposed framework. To test the solution, a testbed containing an exemplary service provider and a consumer of this service is developed. Using this testbed, several simulation scenarios are created that show the accuracy and the trustworthiness of the monitored SLA metrics. Another goal of this thesis is to perform a survey of existing, related monitoring approaches. Based on the research, a comparison of the approaches is done.

1.4 Thesis Organization

The remainder of this work is structured as follows. First, Chapter 2 entails a survey of related work. In our research, we present and discuss approaches that are related to the problem of monitoring cloud SLAs in a trustworthy manner.

We provide the reader with background information in Chapter 3, where we lay out a foundation for our proposed framework by describing important terms and core technologies we encountered during our research. Also, a breakdown of measurable SLA metrics is presented.

Based on the research presented in Chapters 2 and 3, we form our SLA monitoring and arbitration framework in Chapter 4. First, we discuss the requirements we aim to fulfill with our framework. Subsequently, we show the development of the reference implementation, including all design decisions taken in the process.

Following the conceptual design and implementation of our approach, we create a test environment, described in Chapter 5. The experimental testbed we develop resembles our problem scenario, i.e., we create an exemplary cloud service and a consumer.

In Chapter 6, various simulation scenarios for our framework are presented. We perform tests in our developed test environment to show the accuracy of measured SLA metrics. We reflect how our developed solution contributes to trustworthiness and evaluate the performance of the system in terms of intrusiveness.

We then conclude the work in Chapter 7 with a summary. Furthermore, areas for future research are identified and outlined.

CHAPTER 2

Related Work

In this chapter, we survey existing approaches that are related to our given problem. A certain focus is put on cloud services, however, other closely related realms like SOA or grid computing are analyzed as well.

First, in Section 2.1, we provide an overview of existing literature, but also present commercial tools. In Section 2.2, we then discuss and compare the presented approaches, and their applicability in the context of our motivational scenario. The related work also serves as starting point for the development and evaluation of our own approach. Ideas or concepts that we have adopted to form our proposed framework are later stated in Section 4.2.

2.1 State-of-the-Art Monitoring Approaches

We divide the related work into three categories. Sections 2.1.1 and 2.1.2 present strategies within the context of SOA, Sections 2.1.3 and 2.1.4 include concepts for grid applications, and Sections 2.1.5 through 2.1.13 provide an overview of approaches considering cloud services.

2.1.1 A Monitoring Framework for SOA by Balfagih et al.

In “Agent based Monitoring Framework for SOA Applications Quality” [BH10], Balfagih et al. suggest a framework to automate monitoring of SLAs in a SOA. SOA is a paradigm where an architecture is created out of loosely coupled services with standardized interfaces to map a business process. Since services may be owned by different parties or enterprises, the paper describes problems of trust for the consumer that arise when measuring of SLA metrics is only done by the provider. In order to achieve the goal of creating a trustworthy framework, a third party is proposed that analyzes violations and forces penalties for the causative party.

In the described framework, groups of software agents are used on the provider, the consumer and on the third party side. Each group consists of multiple agents that work together to reach a goal. Every component in the architecture has access to the SLA file. SOAP messages between a client and a service, i.e., the communications between those components, are analyzed by an agent and checked against the definitions in the agreement, then forwarded to the actual receiver. If the consumer agent or the provider agent detects a violation, it is forwarded to the third party. The third party then requests log files from both involved sides and performs another analysis. If the outcome of this evaluation is in fact a violation of SLAs, penalties are applied.

Neither an implementation nor an evaluation of the approach is presented with this paper, however, an implementation is currently being realized.

2.1.2 SLA Monitor for SOA by Ameller et al.

Another approach to monitor SLAs in a SOA is described by Ameller et al. in “Service Level Agreement Monitor (SALMon)” [AF08]. Based on the ISO/IEC 9126-1[1] quality model for software engineering, a list of attributes which can be monitored at a Web service are identified. Properties of the model that describe software design characteristics, e.g., maintainability, portability or usability are excluded since they cannot be monitored in real-time. Availability, time behavior and accuracy are the result subsets of the analysis.

The proposed architecture for monitoring is composed of following services.

Monitor Several measure instruments belonging to the Monitor perform measurements in predefined intervals of time (intervals are specified together with a metric) at the SOA system. The monitor then stores gathered information into a database.

Analyzer This component analyzes the collected data. When detecting a violation, it is reported to the next component, the Decision Maker. The conditions are read out of an SLA document that is defined for each service, or are configured manually by an user.

Decision Maker When triggered by the Analyzer, the Decision Maker performs actions that aim to solve SLA violations. To restore the order, it selects the best solution to perform a treatment to the SOA system.

A first implementation that allows to measure the response time and the availability of a SOA system is presented, however, no evaluation was performed for this approach.

2.1.3 Accurate Monitoring for Grid Applications by Ropars et al.

An approach to monitor applications in the grid is presented by Ropars et al. in “GAMoSe: An Accurate Monitoring Service For Grid Applications” [RJM07]. The basis of the

[1]ISO/IEC 9126-1 quality model; `https://www.iso.org/standard/22749.html`.

described grid architecture are applications that are distributed over different grid nodes. A node may be a single PC or a computer cluster. The authors mention the problem that nodes are heterogeneous and volatile. Also, similar to SOA or cloud computing, resources may be managed by other enterprises or domains.

Since an application consists of multiple components running on the grid nodes, Ropars et al. try to solve the stated problems by equipping each node with a so-called *Component Monitor*, i.e., a software that collects information of the local node and propagates it to an *Application Monitor*. The *Application Monitor* manages a single application, and therefore all nodes belonging to it. Besides measuring the resource consumption of the nodes, e.g., CPU and memory consumption, the proposed system also gathers the states of all application components and provides a failure detection and recovery. The measurements can be used to check if defined SLAs are respected. An important constraint for the system is to be completely transparent to the applications, i.e., no modifications thereof must be required.

The authors of the work provide an implementation of the monitoring system, which has been successfully integrated into the Vigne Grid Operating System. Various tests have been carried out to assess the impact of the monitoring framework on the performance of the grid, as the authors are targeting high-performance computing.

2.1.4 Grid-Enabled OMIS Monitor by Baliś et al.

In “Monitoring Grid Applications with Grid-Enabled OMIS Monitor” [BBF+04], Baliś et al. introduce a monitoring infrastructure for interactive grid applications. As a basis they use the standard *OMIS*, which specifies an interface for accessing monitoring tools.

The infrastructure describes a decentralized and distributed system consisting of different components. For each node in the grid, there is a local component that receives requests to monitor applications in this node. There may be multiple applications per node that can be monitored simultaneously. The requests are sent by a manager component that exists permanently for each site of the grid. Also, this entity provides the *OMIS* interface to the monitoring tools. If an application requires to be measured at application level, monitoring modules can be embedded.

Baliś et al. conclude by describing how their proposed infrastructure meets their specified design requirements such as efficiency, scalability, security or transparency to the user, however, no detailed evaluation of the approach is presented.

2.1.5 An AOP Approach by Mdhaffar et al.

In “AOP4CSM: An Aspect-Oriented Programming Approach for Cloud Service Monitoring” [MHJ+11], Mdhaffar et al. describe a method to monitor QoS parameters in cloud environments at the software layer, again without being invasive to the cloud services, neither to the server nor to the client implementation. This is achieved by the use of AOP. By the interception of methods on the server as well as on the client side, data is collected and evaluated by the AOP code. There are also different instants of time that are gathered when service requests and responses are carried out. They help to calculate parameters such as the response time or the execution time.

Code interceptions are performed in the following situations.

1. When the client invokes a request
2. When the server receives a request
3. When the server sends the response
4. When the client receives the response

Besides time-related QoS parameters, *AOP4CSM* is also able to measure the parameters throughput and availability. Both are calculated by using the additionally monitored data. For example, to calculate the availability, the number of successful requests and the number of all requests is stored by the interceptors, the availability is then expressed by the ratio of the two numbers. The proposed approach by Mdhaffar et al. is also able to handle multiple clients, i.e., all implemented measurements can be performed for multiple clients simultaneously. To achieve this, clients are distinguished by their IP address on the server side.

The *AOP4CSM* system has been integrated into an existing fault tolerance framework, which uses the monitored QoS parameters to perform a graceful recovery in case of a failure. An experiment showed improvements in failure recovery while the computational overhead for the monitoring was considered as a negligible value.

2.1.6 Application Level Monitoring by Mastelic et al.

Mastelic et al. present an agent-server architecture in their work “M4Cloud - Generic Application Level Monitoring for Resource-shared Cloud Environments” [MEMB12]. In the approach, agents describe stand-alone applications that measure application-level metrics like CPU usage, response time or memory of a cloud service, amongst other measurements. The agents are also deployed into the cloud and run parallel to other applications. Monitored metrics are communicated to a separate server that stores the gathered information into a database. The server is also responsible for managing all agents.

The two main components of this approach, i.e., the agent and the server are evaluated in the cloud. Several tests are performed in order to show the agent-based monitoring of CPU and memory usage metrics. Furthermore, an analysis of the agent’s resource consumption is carried out, which revealed that the consumption is negligible. The server evaluation showed that all packages transmitted by the agents are received by the server, however, the database is identified as a bottleneck, since the database can not keep up with the high amount of insertions.

2.1.7 Multi-Layer Cloud Monitoring by Trihinas et al.

In “JCatascopia: Monitoring Elastically Adaptive Applications in the Cloud” [TPD14], Trihinas et al. introduce a cloud monitoring system that is capable of monitoring in multiple layers of the cloud environment. In order to achieve this goal, the approach suggests to deploy one monitoring agent, that is, an autonomous software that performs monitoring, per layer.

The proposed architecture installs agents in the following three levels of the cloud.

1. Application Level
2. Virtual Machine (VM) Level
3. Physical Level

An advantage of the agent-based monitoring is that they are non-intrusive and transparent to the existing software or infrastructure, since no code or system modifications are required. The agents run independently beside the other components. In the approach by Trihinas et al., different probes collect low-level metrics and forward them to the agents. An Application Programming Interface (API) allows developers to extend an agent with new probes. Amongst other things, *JCatascopia* provides probes for monitoring the CPU, memory, network and the hard disk.

A key feature of *JCatascopia* is that the agents of the system perform in a fully automatic manner, they can be dynamically added or removed. They communicate with a monitoring server through a *publish and subscribe* message pattern. The server also applies a filter to the received values, this helps to reduce the storage overhead. With the use of a specially designed rule language, users are able to express mappings that transform low level metrics to higher level metrics.

Trihinas et al. provide experiments within their work that show the dynamics of their system, working in a testbed of public and private cloud infrastructures. These proved that the system is capable of scaling to an emerging workload while having a low runtime footprint.

2.1.8 CEP-Based Monitoring by Leitner et al.

In “Application-level performance monitoring of cloud services based on the complex event processing paradigm” [LIH+12], Leitner et al. focus on the collection and correlation of high-level application performance metrics. Their approach is based on an existing middleware called *Cloud-Scale* that is able to scale Java applications transparently by moving application parts to the cloud. Essentially, various events that describe the current system state are emitted by *Event Emitters* and collected by the middleware. By applying CEP to the received events, filtering as well as mapping of low-level metrics to high-level metrics is done.

For a preliminary evaluation of the approach, the CPU and network overhead of the system is analyzed. The results show that the overhead is reasonably low, thus, the framework is feasible even at high event rates.

2.1.9 SLA Violation Detection by Emeakaroha et al.

Emeakaroha et al. propose a cloud monitoring architecture in their work "CASViD: Application Level Monitoring for SLA Violation Detection in Clouds" [EFN+12]. The focus of this work not only lies in measuring SLA metrics, but also in detection of metric violations. Therefore, thresholds are defined that are stricter than the violation thresholds. A transgression of these values warns the cloud provider's system before a real violation occurs, so it can react quickly to counteract. The work by Emeakaroha et al. also provides tools for scheduling and deployment as well as resource allocation in the cloud.

The discussed framework is evaluated by monitoring the throughput and the response time of a ray tracing program with different measurement intervals. It is shown that the number of detected violations is increasing with the rate of measurement. At larger intervals, occurring violations are missed if they occur between the intervals.

2.1.10 Federated Cloud Services Monitoring by Moustafa et al.

Similar to Mastelic et al. [MEMB12], Moustafa et al. describe in "SLAM: SLA Monitoring Framework for Federated Cloud Services" [MEME15] an agent-server architecture to monitor SLA metrics. Besides application-relevant metrics, this approach allows users to also collect various system-level performance information. The framework provided with this work is platform-independent, i.e., it does not rely on a specific cloud provider.

Part of the *SLAM* framework is a dashboard where users can input SLA parameters and the respective thresholds. According to defined rules, the system performs a mapping of the entered parameters to low-level metrics that can be monitored. The authors of this work distinguish between simple and complex mappings. The simple mapping describes the case when an SLA parameter is directly assigned to a single metric, while a complex mapping assigns multiple metrics to an SLA parameter. A coordinator component acts as central component in the system. Agents deployed in the same VM as the applications receive tasks from the coordinator and send back their gathered data. The central unit stores the information into a monitoring repository that is represented by a database. Users can then generate customized reports out of the repository data. The implemented prototype for measurement is built on top of an existing monitoring software called Zabbix[2].

To show the functionality of the framework, tests have been performed in a private and a public cloud environment. Moustafa et al. evaluate the overhead of their system by comparing the responsiveness of the target application with and without the framework

[2]Zabbix is an open source monitoring software for IT infrastructure; `https://www.zabbix.com`.

deployed in the cloud. Also, several frequencies of monitoring, i.e., the rate when a measurement is updated, have been tested on how they impact the system differently.

2.1.11 A Trusted Third Party for Monitoring by Maarouf et al.

Maarouf et al. address the problem of trust in SLA monitoring in “Towards a Trusted third party based on Multi-agent systems for automatic control of the quality of service contract in the Cloud Computing” [MMH15]. Therefore, they introduce a Trusted Third Party (TTP) as a control authority between a cloud service provider and the service consumer. The proposed TTP is extended with a multi-agent system that is composed of four different agents that work together. When detecting violations, penalties to the party not adhering to the contract are enforced by an agent.

The authors of this work provide no implementation or evaluation of the introduced framework.

2.1.12 SLA Monitoring for RESTful Services by Al-Shammari et al.

In “MonSLAR: A Middleware for Monitoring SLA for RESTFUL Services in Cloud Computing” [ASAY15], Al-Shammari et al. propose a monitoring framework that considers the server as well as the client side for measuring SLA parameters. To reach this goal, a RESTful protocol is used, i.e., Hypertext Transfer Protocol (HTTP) methods are extended by embedding monitored data in headers of requests and responses.

The architecture is composed of two different middlewares, one for the client and one for the server. When a client software performs a service request, the middlewares forward the message to the service. This also applies to responses of the service. When forwarding, a middleware performs measurements and appends the information to the HTTP header, so metrics like response or communication time can be inferred. The client’s middleware also calculates the Quality of Experience (QoE) from monitored metrics. Each has the built-in functionality to compare measurements against values in the agreement. Additionally, on the server side, an agent takes care of monitoring the machine’s resources. The authors state that their architectural design leads to the advantage that no additional communication overhead to a broker or similar is necessary.

To evaluate the approach, a comparison with other monitoring frameworks is presented. The authors argue that the introduced framework is one of the few which considers the QoE and takes into account client and server measurements. Furthermore, they claim that no additional broker or similar is needed for the management of monitoring.

2.1.13 CloudWatch by Amazon Web Services

CloudWatch [Ser17] is a commercial service provided by Amazon Web Services (AWS) to monitor applications and resources in the Amazon Cloud.

From an extensive number of implemented metrics, costumers can select those they are interested in. Users may observe metrics of VM instances, e.g., CPU usage, disk activity, communication overhead, or view detailed information about used AWS resources, e.g., databases, messages queues, load balancers and so on. An API allows developers to submit custom metrics for their applications to *CloudWatch*. Another feature is an alarm that can be set for any metric. Is a threshold exceeded or not reached, predefined actions to counteract can be performed automatically.

The service provides a free but limited contingent of metrics to monitor, however, various price models exist to extend the contingent.

Similar monitoring tools also exist for other cloud platforms, for example, Stackdriver Monitoring for the Google Cloud Platform[3] or CloudMonix for Microsoft Azure[4].

2.2 Discussion of Related Work

This section aims to provide a comparison and reflection of the presented related work.

We first define properties that describe the monitoring frameworks in terms of technology and in respect to SLA monitoring and arbitration. This helps us to carry out a comparison and to decide whether a framework is applicable to solve our problem or not. We define the following properties.

Target Model This property distinguishes which model or paradigm the approach considers, i.e., whether it is targeted for the grid, cloud or SOA.

Measurement Technology Another important distinction considers the technology that is used to perform monitoring of the involved components, e.g. agents, AOP, middleware, etc.

Measurement Side It is important to define if measurements are done on both involved parties (consumer and provider), or if only one side is considered.

Non-Invasive Approaches may be invasive to an existing client or service software, i.e., they require modifications or access to the source code. This property characterizes if an approach is able to monitor in a non-invasive manner.

SLA Mapping In this property, we determine whether the monitored low-level metrics are mapped to higher-level SLA metrics.

SLA Violation Detection This property states whether the given framework has built-in capabilities to detect SLA violations.

Trustworthiness This property indicates whether considerations about the trustworthiness of monitored parameters are taken in the given approach.

[3]Stackdriver Monitoring; `https://cloud.google.com/monitoring`.
[4]CloudMonix; `http://www.cloudmonix.com/`.

Requirements Definition This property defines how the SLA requirements are specified, i.e., whether an approach uses existing standards, or a custom solution.

After defining our properties, we compare the presented related work with regards to technology and SLAs in Tables 2.1 and 2.2, respectively.

To the best of our knowledge, there is no approach that is entirely suitable to our problem statement. In the following, we describe the approaches coming closest to our solution, and which points do not comply.

In [RJM07], the monitoring provides an overview of resource consumption and states of distributed applications in the grid. The authors clearly follow another goal with this approach: This information helps to react on node failures and to provide high availability of a service, rather than to provide a trustworthy monitoring for provider and consumer. Also, if SLAs are negotiated, only measurements of hardware resources can be used for this purpose. These may say little about the actual performance of an application or service. Higher-level QoS metrics provide more information about the behavior, however, these are not considered at all with this approach.

Another interesting framework is provided in [MHJ$^+$11]. The use of AOP allows to monitor metrics in a non-invasive manner on the server and on the client side. However, the authors use the monitoring capabilities for providing a fault-tolerance system, the SLA management domain is not targeted. Besides that, the mechanism provides only a limited set of possible metrics and can not be easily extended.

Table 2.1: Comparison of Monitoring Frameworks in Terms of Technology

Framework	**Target Model**	**Measurement Technology**	**Measurement Side**	**Non-Invasive**
[BH10]	SOA	Agent	Provider + Client	✓
[AF08]	SOA	Agent	Provider	✓
[RJM07]	Grid	Agent-like	Provider	✓
[BBF$^+$04]	Grid	OMIS + Tools	Provider	
[MHJ$^+$11]	Cloud	AOP	Provider + Client	✓
[MEMB12]	Cloud	Agent	Provider	✓
[TPD14]	Cloud	Multi-Layer Agents	Provider	✓
[LIH$^+$12]	Cloud	Event Emitters	Provider	
[EFN$^+$12]	Cloud	Agent	Provider	
[MEME15]	Cloud	Agent	Provider	✓
[MMH15]	Cloud	Agent	Provider + Client	✓
[ASAY15]	Cloud	Middleware	Provider + Client	✓
[Ser17]	Cloud	Proprietary	Provider + Client[1]	✓

[1] Measurements on the client side may be enabled via custom metrics.

Table 2.2: Comparison of Monitoring Frameworks in Respect of SLAs

Framework	**SLA Mapping**	**SLA Violation Detection**	**Trust-worthiness**	**Requirements Definition**
[BH10]	✓	✓	✓	SLA File
[AF08]	✓	✓		WSLA / User-Defined
[RJM07]				—
[BBF+04]				—
[MHJ+11]				—
[MEMB12]	✓			WSLA
[TPD14]	✓	✓		Custom Def. Language
[LIH+12]	✓	✓		—
[EFN+12]	✓	✓		User-Defined
[MEME15]	✓	✓		User-Defined
[MMH15]	✓	✓	✓	—
[ASAY15]	✓	✓		WSLA
[Ser17]				—

The tool CloudWatch [Ser17] works in a provider-dependent way, i.e., it can only be used with AWS. The gathered metrics are more oriented towards developers using this platform to have a control over their resources and applications. The measurements can not be used to protect the customer of the cloud, as the tool lacks the possibility to combine measured metrics with SLAs.

In [BBF+04], another monitoring mechanism for the grid is shown, focusing on gathering low-level metrics. Besides not being targeted to cloud applications, a possibility to specify or arbitrate SLAs is not provided, making this approach unsuitable to our problem.

Although they consider SLAs, various approaches propose strategies for measuring metrics on the provider's side only [AF08, MEMB12, TPD14, LIH+12, EFN+12, MEME15]. In these approaches, the client side is not considered at all. However, it is generally in the interest of the consumer to distrust values monitored solely on the server side. In order to provide and maintain trustworthiness, we must take into account performing measurements on the client side.

[ASAY15] is a framework that provides the capability to monitor SLA metrics which reflect the QoE of the consumer. However, this approach is limited to work with RESTful Web services only. Other service technologies may be hard to adapt as the authors rely on specific HTTP request messages. The approach also does not aim to provide a trustworthy arbitration.

To the best of our knowledge, only a few approaches in related work take trustworthiness into account significantly. Both [BH10] and [MMH15] propose to include a trusted broker or trusted party that validates measured SLA parameters. [MMH15] is merely a conceptual framework that has its focus in the management and enforcement of SLAs. In terms of SLA monitoring, the authors do not go much into detail and provide no concrete solutions, thus, makes an adaption of this framework difficult for our problem scenario.

[BH10] covers almost all the properties we are taking into account. Still, some points make an adaption of this framework insufficient for our needs. First, the approach is not completely transparent for the service or client software, since the communication between these two is not direct, i.e., all messages have to to pass the agents on consumer and provider side. This is likely to result in additional communication overhead and may require adjustments in the service or client software. Furthermore, for the functioning of this approach, a set of agents are required to be installed on both the consumer as well as the provider side. Especially for the service consumer, e.g., an end-user of a cloud application, this may not be reasonable. The authors of the work also do not provide a solution to monitor system/hardware resources, but only analyze the communication messages between provider and consumer.

CHAPTER 3

Background

Within this chapter, we provide background information and theoretical foundations for our work. At the beginning, in Section 3.1, we introduce and define basic terms related to our topic. Then, in Section 3.2, various technologies playing an important role in the design and development of our monitoring framework are presented. For a basic understanding of different SLA parameters, we elaborate a breakdown in Section 3.3, where quantitative measurable metrics are identified and discussed.

3.1 Term Definitions

Quality of Service This term describes an objective model to quantify a service in terms of performance. QoS is related to the term QoE, however, the latter uses subjective measurements to denote the quality. QoE combines non-technical and technical parameters to describe the perception from the user's perspective [FHTG10], where QoS only uses technical, objective parameters.

Metric The quantitative measure to describe properties of an entity is referred to as metric [Fen94]. The use of metrics makes it possible to quantitatively analyze the entity and compare it with others. For example, Web service performance metrics may be used to describe the efficiency of a service numerical.

Message-Oriented Middleware Message-Oriented Middleware (MOM) describes software that provides a distributed system with the ability to communicate asynchronous messages [GST03]. The use of an MOM results in a system where the components are loosely coupled. A message broker works as central unit for the storage and distribution of all messages. It is a reliable component that ensures reliable message delivery avoiding duplicated messages [TMR04].

RESTful Web Services Web services allow computer systems to communicate with each other over the Internet. Representational State Transfer (REST) describes a paradigm or architectural style to realize (stateless) Web services [ASAY15]. The main idea behind this approach is to rely on existing standards, e.g., HTTP is typically used as communication protocol. Data or functions are exposed as resources which are identified by an Uniform Resource Identifier (URI). HTTP methods like *GET, POST, PUT, DELETE* etc. can be used to perform operations on these resources [Cor13].

3.2 Key Technologies

In this section, we present a brief introduction into central frameworks and technologies we use within our monitoring approach. Each of these concepts requires the definition of certain terms, which are also discussed in the following sections.

3.2.1 Web Service Level Agreement

The Web Service Level Agreement (WSLA) standard published by IBM specifies a conceptual framework for defining and monitoring SLA parameters in Web services [KL03].

To this end, the framework consists of a language to describe SLA parameters and the relation and obligations of involved parties, and describes a runtime architecture that manages the entire lifecycle of an SLA. It was designed for the use in Web service environments, but can also be applied to other inter-domain management scenarios (i.e., between enterprises) [KL03]. This characteristic makes the framework also applicable to cloud services [PRS09, MEMB12]. An alternative to the WSLA specification, which serves the same purpose, is *WS-Agreement* by the Open Grid Forum (OGF) [For07].

WSLA Language Features

The WSLA language is based on an XML schema, hence a WSLA can be expressed by an XML file. A proper agreement is defined by the following three major parts [LKD$^+$03].

Parties This part specifies the involved parties, which includes at least the two signature parties, i.e., the service provider and the service consumer. Additional parties can be defined in this section too, such as supporting parties that help to carry out measurements or manage SLAs.

Service Definition The Web service is described further in this section. All operations offered by a service are listed, for each operation the agreed SLA parameters are defined. Keller et al. specify an SLA parameter as a metric that is put into the context of a specific customer [KL03]. The metric to which an SLA is assigned is also expressed in the Service Definition part, while further context is given in the Obligations, as described below. The WSLA specification distinguishes between resource metrics and composite metrics. While resource metrics can be gathered directly by an involved component, composite metrics are built out of several resource metrics or even other composite metrics by applying an algorithm to the values. Algorithms, for example, calculate average values over a time span, or pick out the minimum or maximum value of a series. The WSLA language provides an extensive operation set to express various kinds of metrics, such as arithmetic operations or time windows to which a calculation applies.

Obligations The obligations contain conditions which signature parties committed to promise, and actions they have to perform in case of violations. Conditions are boolean expressions that can be described using the WSLA language. For example, conditions evaluate whether measured parameters exceed or undercut a given threshold.

SLA Management Lifecycle

The WSLA framework proposes a lifecycle for automating the entire management process of SLAs. The lifecycle includes five stages: negotiation, deployment, measurement, corrective management and termination of SLAs.

In the first stage, the two signature parties negotiate and sign an SLA for a given service. This process can be tool-assisted. The outcome is a WSLA document as described in the last section. After that, the deployment process invokes a distribution of the document to the involved parties. The document may be deployed as a whole or only by relevant sections. In this step, roles and duties are assigned to the parties. In the measurement stage, services perform monitoring of SLA parameters according to their assignments. Also, an evaluation service checks the values against predefined conditions. In the fourth stage, when a violation is detected, the management service enforces actions a signature party obliged to do. The last stage is the SLA termination. An SLA is terminated when thee negotiated termination conditions are met. Note that the WSLA language also allows to specify an expiration date for the termination [LKD+03].

3.2.2 Complex Event Processing

CEP is a paradigm which enables a continuous analysis of data-streams in real-time, as proposed by David Luckham in “The Power of Events” [Luc01].

A system that implements the CEP paradigm is in contrast to a database management system, where persistent data is analyzed. CEP has the ability to combine lots of different data sources and to process the data right after events are registered at the data source. Events often describe a change of a state, e.g., an activity, a decision or a process [BD15]. Among other things, CEP provides capabilities to filter, aggregate or correlate events. An abstraction of multiple events into a higher-level event, resulting through an operation at the events, is called complex event [BD15]. Often, a processing language is used to express patterns and rules, in order to apply CEP [LEM16].

3.2.3 Aspect-Oriented Programming

The concept of AOP was first introduced by Kiczales et al. [KLM+97]. While Object-Oriented Programming (OOP) is a programming paradigm that helps to modularize code by encapsulating related attributes and functions into *objects*, the AOP paradigm allows to modularize code responsible for concerns that *cross-cut* basic functionalities of a system [KLM+97].

Cross-cutting concerns are reoccurring functionalities that are implemented in multiple different methods in the code, and cannot easily be encapsulated with the concept of OOP alone. For example, consider functions that are only allowed to be executed if a user is authenticated. Without the use of AOP, each of these functions must include the code to perform a verification of the user before executing. AOP solves this problem by allowing to isolate the concern of verification from the functions using it. An advantage resulting from the isolation of concerns is that the program code is easier read, to maintain and to develop [KLM+97].

The modularized cross-cutting concerns are referred to as aspects. The phase where aspects are integrated into the code is called *weaving* [KNJ15]. Multiple different weaving methods exist, and can be categorized as follows:

Compile-Time Weaving This method is applied by a special compiler that generates a binary file with woven-in aspects out of source code files.

Post-Compile Weaving Similar to the first approach, the result of this method is a binary file that has the aspects included. This approach, however, does not require access to the source code of the application, but can perform weaving on the already compiled application binary file. Sometimes, this method is also called binary weaving.

Load-Time Weaving This method also does not need access to the source code of the application. In contrast to the post-compile weaving, no new binary file is yielded. This time, the weaving process is performed when parts or units of the original program are loaded into memory. The code is then enhanced or modified with aspects. When carrying out load-time weaving on a Java application, this is the case when the Java Virtual Machine (JVM) performs class-loading [Asp05].

Run-Time Weaving Some AOP frameworks also support run-time weaving. As the name indicates, the weaving is done during the execution of an application at run-time. This is accomplished by using dynamic proxy objects that act as intermediaries to the actual objects [Spr16].

Because some weaving methods (e.g., load-time weaving or post-compile weaving) have the ability to add aspects to existing binary files without modifying them, AOP is also suitable for adding additional functionality to applications in a non-invase way [MHJ+11].

Other important terms for AOP are join points and pointcuts. Join points refer to the position in the code were an aspect is woven into, while pointcuts *pick out* these join points [Asp03]. Pointcuts can be defined to trigger aspects by using expressions, e.g., to describe when a specific function is executed or called. Advices, i.e., the aspect code implementation, are then executed when the expression of a pointcut is matched.

3.3 Breakdown of SLA Metrics

For the purpose of this work, we identify and analyze SLA metrics with a certain focus on Web services. For every metric, we describe how it can be measured and calculated. Different views in the calculation or interpretation of the values exist, so we compare and discuss the different methods to decide which one is best suited for which problem scenario. A detailed description of which representation we refer to in our monitoring approach is given in Section 4.2.3.

A classification is made by analyzing the trustworthiness of the measurement, i.e., whether metrics can be monitored in a trustworthy manner. For this purpose, we also examine the possible parties from which metrics can be read, such as the provider or consumer of the service, or a neutral, independent party.

First, we identify possible SLA metrics for real-time monitoring. For this purpose, Ameller et al. survey the ISO/IEC 9126-1 quality model for software engineering in [AF08]. They argue that software design characteristics, namely, maintainability, portability, usability and reliability, are of no interest for monitoring, because they do not change during execution time. The outcome of the survey are characteristics like availability, time behavior and accuracy. However, we argue that the described quality model might be too generic for cloud/Web services, and refer to the OASIS Open Standard for Web Services Quality Factors (WS-Quality-Factors) [Ope12]. This standard defines various quality levels for Web services, e.g., business value quality, service level measurement quality, manageability quality or security quality, to name a few. The service level measurement quality comprises quantitative, dynamically changing attributes which describe the QoS. Consequently, these attributes are highly suitable for real-time SLA monitoring. We observe that these metrics are a subset of the evaluation results in [AF08].

In the following, we present the breakdown based on the quality factors in [Ope12]. This section also includes metrics that describe the quality of provided system resources for a service. We consider metrics that are as general as possible, because highly application-specific metrics might be too diverse to be described in a single breakdown.

3.3.1 Response Time

In [Ope12], the *response time* is defined as the duration between the time when a request is sent and the time when a response is received.

The metric consists of three types of latencies:

$$\mathit{ResponseTime} = \mathit{ClientLatency} + \mathit{NetworkLatency} + \mathit{ServerLatency} \tag{3.1}$$

The latency types can be described as follows.

Client Latency This latency refers to the time a client system needs to process the messages a service request comprises, that is, preparing the request message and handling the response message. For example, when the system needs to deal with security attributes or needs to process additional data for reliable messaging, this latency may increase [MHJ+11].

Network Latency This latency describes the time a network needs to deliver the request message and the response message from one endpoint to the other.

Server Latency Similar to the client latency, this metric comprises the processing times of the messages. However, this metric also includes the execution time, meaning, the time to execute a request on the server.

To measure the response time on the client side, (3.1) can be simplified to a single duration between two instants in time, shown in (3.2). The formula reflects the above definition.

$$\mathit{ResponseTime} = t_2 - t_1 \tag{3.2}$$

where t_1 is the *request sent timestamp* and t_2 is the *response received timestamp.*

The proposed framework in [MHJ+11] measures the response time metric on the client side, and the execution time metric on the server side. Since they neglect the processing time, they are also able to infer the network latency as the difference of both measured attributes. As in (3.2), they record the *request sent timestamp* and the *response received timestamp* on the client, and then calculate the response time.

Another view on this SLA metric is presented in [MEMB12]. Since Mastelic et al. perform monitoring on the server side, they calculate the response time from the duration between *request received timestamp* and *response sent timestamp* from the server's point of view.

Although the response time might differ a little depending on the point of measurement, we argue that in order to objectively analyze the metric, readings from both sides of a service must be considered and compared. As long as no major deviation in both values is detected, they can be trusted as valid. However, the measurement on the client side fits better the definition of the response time, since it also takes into account the client latency.

A closer examination of the resulting difference of the measurement side is carried out in the evaluation in Chapter 6.

3.3.2 Throughput

Besides the response time, the WS-Quality-Factors standard specifies the maximum throughput to measure the quality of Web services, that is, the maximum number of requests that can be processed by a service provider in a given time span. In (3.3), the calculation of the general throughput is described. The formula in (3.4) then applies the maximum function, so, the outcome is the mentioned metric.

$$\mathit{Throughput} = \frac{\mathit{NumberOfRequestsProcessedByServiceProviderInMeasuredTime}}{\mathit{MeasuredTime}} \tag{3.3}$$

$$\mathit{MaximumThroughput} = \max(\mathit{Throughput}) \tag{3.4}$$

SLAs may define other aggregation functions besides the maximum, which are to be applied to the throughput. The WSLA language, for example, provides several functions for any metric to be used [LKD+03]. Therefore, we argue that it is more flexible to express and measure the general metric, in this case the throughput – see (3.3) – and if required, any aggregation function can be applied.

In [MHJ+11], the throughput is calculated at the client side, using (3.5). It is defined as the number of successful requests for a period of time T.

$$\mathit{Throughput}_{\mathit{ClientSide}} = \frac{\mathit{SuccRequests}}{T} \tag{3.5}$$

While T clearly corresponds to the denominator of (3.3), i.e., the measured time, we have to further define which requests are classified as *successful*. When a service processes a request from a client and returns a successful response message, the service request is defined as *successful* for the client [MHJ+11]. Therefore, every request successfully measured by the client was also processed by the service, meaning the throughputs in (3.3) and (3.5) basically describe the same.

Note that the formula for the throughput in (3.3) evaluates the whole service provider, therefore, the resulting throughput is at system level. A service provider may run several services independent from each other. To provide an application-level measurement, i.e., to cover only the Web service which is considered for the measurement, we adapt the formula a little:

$$\mathit{Throughput}_{\mathit{ApplicationLevel}} = \frac{\mathit{NumberOfRequestsProcessedByServiceInMeasuredTime}}{\mathit{MeasuredTime}} \tag{3.6}$$

If an SLA defines the throughput metric at user/client basis, then the definition in (3.5) is sufficient. In the other cases, we must consider that a service may also processes requests from multiple users/clients in a given period of time. A single client may not have insight into the number of service requests of all clients, as the server does. Therefore, measurements from the server side are required to calculate the throughput in (3.6).

To provide trustworthiness, successful requests measured by clients and processed requests measured by the service can be correlated and compared if they match. The actual calculation of the throughput over a time period is then carried out using the readings from the service.

3.3.3 Availability

Another important metric for describing QoS is availability. It is defined as the ratio of uptime in a given time period [Ope12, MA15]. The calculation of this percentage is shown in the following:

$$Availability = \frac{Uptime}{MeasuredTime} \tag{3.7}$$

The uptime of a service refers to the time in which the service is running and ready to use. In contrast, downtime expresses the time in which the service is not usable. As displayed in (3.8), it is more convenient to calculate the availability out of the downtime instead of the uptime [Ope12]:

$$Availability = 1 - \frac{DownTime}{MeasuredTime} \tag{3.8}$$

Another approach to calculate the availability of a service is given in [MHJ+11]. The authors express the metric as a ratio of successful requests to all requests in a given time. This alternative approach can be seen in (3.9). However, in [Ope12], this ratio defines the successability metric, which is closely related and explained in Section 3.3.4. Since this does not match the definitions of availability from [Ope12, MA15], we use (3.8) in our approach.

$$Availability_{AlternativeApproach} = \frac{SuccRequests}{AllRequests} \tag{3.9}$$

To calculate the availability, certain knowledge of the downtime or uptime of a service in a given time period is required. To this end, the service can be tested at predefined intervals whether it is up or down.

Another crucial aspect is the point of measurement. It may be difficult to perform monitoring on the server the service is running on, due to several reasons: If the service is unavailable because the entire server is down, then there is also no possibility to do measurements on the server. Even if the service is operative for the monitoring component, other issues, e.g., network problems could exist, so clients may not be able to access the application. This may remain undetected by the server.

We argue that a client is also not suitable to measure the availability. If the availability requires to be measured in longer time spans, e.g., uptime within a month or year, the client must be running permanently and perform repeated tests. Besides that, the service provider may not trust these measurements.

The WSLA framework proposes to use an independent third party to probe a service for availability [KL03]. As described, the party can detect downtimes through measurements at predefined intervals, but also can perform the calculation of availability. If both provider and consumer trust this third party, the monitoring of the metric is in fact trustworthy.

3.3.4 Successabilty

Successability is a quality aspect of a Web service that puts in relation the number of (successful) response messages to the number of request messages in a given time period [Ope12]. The concept of a successful response has already been discussed in Section 3.3.2. The formula for the calculation of the QoS metric is defined as follows:

$$Successabilty = \frac{NumberOfResponseMessages}{NumberOfRequestMessages} \tag{3.10}$$

Similar to the throughput metric, the service can track whether the processing of a request was successful and whether a successful response message was sent to the client. Therefore, both the client and the service can measure the number of successful invocations. For the calculation of the metric, we have to consider two cases: If the SLA parameter is agreed at user/client level, the successability can be calculated at the client using (3.10). In the other case, if the parameter describes the successability of the entire service (including multiple client requests), a different strategy must be applied. Again, a third party can collect the number of request and response messages from all involved clients and can calculate the overall successability of the service, using (3.10). To provide trustworthiness, a TTP can be utilized that correlates and compares measurements from client and server before the actual metric is computed.

3.3.5 Accessibility

Despite the similar name, the terms accessibility and availability describe different characteristics of a system. Being available does not imply that a system is also accessible [IBM02].

The accessibility metric of the WS-Quality-Factors standard allows to express the probability in which a service provider platform is accessible while availability is given [Ope12]. It is defined as the number of acknowledgements from the platform to the number of requests from clients in a given time, see (3.11).

$$Accessibility = \frac{NumberOfAckMessages}{NumberOfRequestMessages} \tag{3.11}$$

This metric describes the state of the service platform rather than that of a service or application. However, we propose a similar method for the measurement and calculation of this metric like we did for the successability metric. The only difference is the point of measurement on the provider platform. This time, it is not the service that is monitored, but the platform itself, which sends the acknowledgment messages to the clients.

3.3.6 CPU Usage

An operating system's process scheduler takes over the task of assigning the available CPUs to the running processes. By giving the processes fixed time slots to execute their instructions on the CPUs, this assignment is carried out. The accumulated time for a given process to be executed on the CPUs is referred to as CPU time. For defining the CPU usage metric, this time metric is essential.

In [MEMB12], the CPU usage is expressed as a percentage of the CPU time for a given application:

$$CPUusage = \frac{CPUtime_{Application}}{CPUtime_{System}} * 100 \tag{3.12}$$

To put this into context, we provide an example. A CPU usage of 80% for a process indicates that the task scheduler assigns 80% of its time slots to this single process. The remaining percentage is allocated for other processes and the operating system itself.

The formula in (3.12) also goes in line with the representation of CPU usage in the Linux operating system [Lin17a, Lin17b]. However, there are two different approaches. While the `ps` command line tool [Lin17a] measures the usage since the start of the process, i.e., during its entire lifetime, the `top` tool [Lin17b] forms the percentage based on the usage since the last screen update. The update time of `top` can be manually configured.

Furthermore, different percentage representations exist. On a multi-core or multi-processor system, CPU usage may reach a multiple of 100%. For example, considering a system with 4 processor cores, a percentage of up to 400% is possible. Another approach is to divide the usage by the number of CPUs, scaling the maximum value to 100%. The Linux `top` command [Lin17b] has both representations built-in. They can be toggled via the `IrixSolaris_Mode_toggle`[1].

To retrieve the metrics CPU time or usage, readings from the operating system must be carried out. This task differs depending on the used system. As mentioned, the representations may also vary between the systems, therefore, they must be unified before they can be used as SLA metrics.

Different to the previously discussed SLA metrics, this metric can only be measured by one party, i.e., on the side where the application or service to be monitored is installed. This is due to the need for access to the operating system for monitoring this CPU

[1]Linux `top` command - man page; `https://linux.die.net/man/1/top`.

metrics. For a trustworthy measurement, a mechanism must be applied to ensure that readings can not be altered by the service provider.

3.3.7 Memory Usage

Like the CPU usage, this metric also belongs to the group of low-level system metrics. In [MEMB12], various metrics are defined that describe the memory consumption of a process: resident, shared and virtual memory usage. A value of interest is the resident memory usage, as it describes the actual physical memory (RAM) a task occupies.

Memory usage can also be expressed as a percentage of the resident memory in relation to the available physical memory. The Linux operating system uses this representation within the `top` command line tool [Lin17b]:

$$MemoryUsage = \frac{Memory_{Resident}}{Memory_{Total}} * 100 \tag{3.13}$$

Memory usage metrics have the same requirements for the (trustworthy) measurement as the CPU metrics because system access is required for reading (see Section 3.3.6).

3.3.8 Recap and Further Metrics

Up to this point, we have identified and described several high-level QoS attributes for Web/cloud services as well as low-level resource metrics. The former are distinguished by the fact that mostly two parties can perform measurements while the latter can only be measured on the server side, i.e., on the host the application is running on.

From the low-level resource metrics, we have outlined the measurement and calculation of CPU and memory usage. SLAs may be agreed on other system or application resources as well, however, our breakdown aims to provide a broad overview of generic application-level metrics and is not exhaustive. For a further distinction, we refer to other work: In [TPD14], there is a focus on resource metrics. Besides CPU and memory, the authors also describe disk, network or database utilization metrics. Also, in [LIH$^+$12], the additional resources disk and network are monitored.

CHAPTER 4

Trustworthy SLA Monitoring and Arbitration

This chapter presents our cloud service SLA monitoring and arbitration approach. It is based on software agents and AOP modules that serve the purpose of measuring generic application-level metrics. The monitoring components are loosely coupled to a TTP that uses the measurements to check whether negotiated SLA parameters are fulfilled.

The chapter is structured as follows: In Section 4.1, we formulate requirements for the mechanism based on an analysis of the given problem statement. However, we aim to provide a generic solution that also works for other, related problems.

After discussing the basis, we then proceed with describing our proposed framework in detail in Section 4.2, considering the requirements in the design of our solution.

In Section 4.3, we present a reference implementation of our idea. The implemented prototype is used to evaluate our approach, as shown in Chapter 6.

4.1 Requirements

In this section, we elaborate and discuss requirements that are the basis for our approach. We start by defining our target model. Afterwards, we describe requirements for monitoring as well as for system design.

4.1.1 Target Model

We aim to perform monitoring in cloud environments. In detail, we measure Web services deployed in the cloud, characterized by providing functionality or data to other computer systems. Therefore, at the core of our model, we consider communication over the

Internet constituted by requests to and responses from a service. Typically, a client application in another computer network calls the service's functions.

Monitoring self-contained applications in the cloud, i.e., applications that do not offer Internet services, is not our primary goal. They have no interaction with clients or similar, therefore, only provider-side measurements are possible. However, we also intend to take into account metrics that describe system resources at application level. When using SLAs for this kind of applications, the resource metrics could serve as measurements to check if the agreements are complied to.

Assumptions

In this work, we adhere to the WSLA standard for the definition of SLAs. We have already described the SLA lifecycle which is proposed by this standard (see Section 3.2.1). To recall, the lifecycle is composed of multiple stages that aim to automate the SLA management process. The focus of this work lies in the measurement and evaluation stage. Covering the other areas is out of scope, however, it is possible to extend our work to cover them in the future. The negotiation and management of SLAs is well researched, we refer to other work that covers this topic [DDK$^+$04, BMLD09]. Therefore, for this work, we assume a negotiated WSLA document as already given.

Furthermore, we assume that the consumer side consists of a single client that invokes a service. This concerns both the implementation and the evaluation of our framework. Consideration of multiple consumers and clients can be done in future work.

4.1.2 Requirements in SLA Monitoring

We have defined various requirements regarding the monitoring of SLA metrics, as discussed in the following:

Application Level First of all, we specify the level of measurement. Since the defined model requires a Web service to be monitored, we argue that system-level metrics, i.e., measurements of the entire server or cloud infrastructure, say little about the performance of this service. Thus, we define as a requirement that monitoring should be done at application level.

Generic The measured SLA metrics should be generic in their usage, that is, they should not be designed for a particular application but work with any Web service. We identified generic application-level metrics in Section 3.3. That is to say, we consider metrics describing the QoS of Web services, but also metrics that describe system resources allocated for the applications.

Trustworthy We strive for a mechanism that ensures trustworthiness in measured metrics for the provider and the consumer of a service. Since both parties have opposing interests, the question arises who decides whether monitored values are violations, or whether those values are correct. Consequently, our approach must provide a trustworthy arbitration of measurements.

Decentralized Measurements are required to be done on both involved sides of a Web service. More precisely, they must be carried out on the provider side, i.e., on the cloud where the service is running on, and on the consumer side, i.e., on the client software that calls the service's functions. If only one of the two parties performs measurements, it may be hard for the other party to trust these values. However, some metrics may only be observable on one side, e.g., metrics that describe system resources on the provider side. Considerations must be given to this.

Accurate The monitoring must be as precise as possible. An evaluation of measured metrics is required to show that the approach provides accurate readings. Imprecise or even wrong measurements may not be accepted and trusted by the involved parties.

4.1.3 Requirements in System Design

When designing a trustworthy and non-invasive monitoring mechanism, we pose special demands for the system design. Their primary purpose is to increase the quality of our solution. In the following, they are specified.

Reliability A key requirement is to provide reliable communication between all monitoring and arbitration components of the system. It must be ensured that every measurement reaches the component that performs arbitration so that it is checked for compliance. If readings are lost, possibly occurring violations can not be detected. We argue that a reliable communication helps to increase trust in the system because the involved parties can be confident that the measurements on their side are considered.

Platform Independence We want to ensure interoperability between different cloud providers. The monitoring mechanism should not be designed in a way that it is dependent on functions of a specific cloud provider. Also, if possible, measurements should work on various operating systems. We set as a goal to support Unix-like and Windows systems.

Transparency We define the requirement that the monitoring should be transparent for the service and for the client. It is important that the two components are able to communicate with each other just as they would without the monitoring system. Besides that, a transparent monitoring system is lightweight and has negligible resource consumption. Especially on the provider side, a low utilization is beneficial because the consumer might pay the provider per usage. Also, the services' performance must not be negatively affected by the use of the mechanism, e.g., the response time of the service must not degrade considerably. We define 5% as an acceptable margin of degradation.

Non-Invasiveness Related to the transparency requirement, this requirement aims to provide a monitoring mechanism without being invasive to the existing infrastructure and software. That is, measuring metrics of an application, be it the service on the provider side or the client software on the consumer side, should be made possible without any modification thereof or of the underlying system.

4.2 Proposed Framework

At this point, we have defined a necessary theoretical basis, along with the requirements concerning the design and implementation of our approach. We now describe the framework that aims to meet the stated requirements.

First, we discuss concepts from related work that inspired our approach. We then continue with the presentation of the architecture and state the roles of all involved parties. Furthermore, we describe our approaches for measuring the identified metrics, and for ensuring trustworthiness in monitoring and arbitration.

4.2.1 Concepts from Related Work

During the conducted literature study, we have found several different technologies to monitor application-level SLA parameters. One commonly applied approach is the use of software agents for the measurement [AF08, TPD14]. However, some SLA parameters like response time or successability are difficult to measure with this technology, as they require an intervention in the service to retrieve them. If the consumer side is monitored too, there is the necessity to permanently run an agent at this side as well. The use of AOP, as presented in [MHJ$^+$11], copes with the downsides of agents. However, this time, resource usage or utilization metrics are difficult to measure continuously, since the AOP code is only executed when the service is requested. To facilitate the advantages of both technologies, we adopt and combine both approaches to perform the measurements.

In [MMH15], a TTP is proposed, which acts as a control authority to enforce SLAs. With [BH10], we have another approach that involves a TTP for the validation of monitored parameters. To provide trustworthiness for our monitoring approach, we also adopt the concept of using a third party as trustworthy authority.

Although the approach in [LIH$^+$12] follows another goal, we adopt the usage of CEP for the correlation of SLA parameters. In addition, we also implement the recognition of SLA violations with this paradigm.

4.2.2 Architecture Overview

We describe our approach by first discussing its architecture. An abstracted overview of the entire monitoring system is depicted in Figure 4.1. In the following, we outline each individual party and their components involved in the monitoring mechanism.

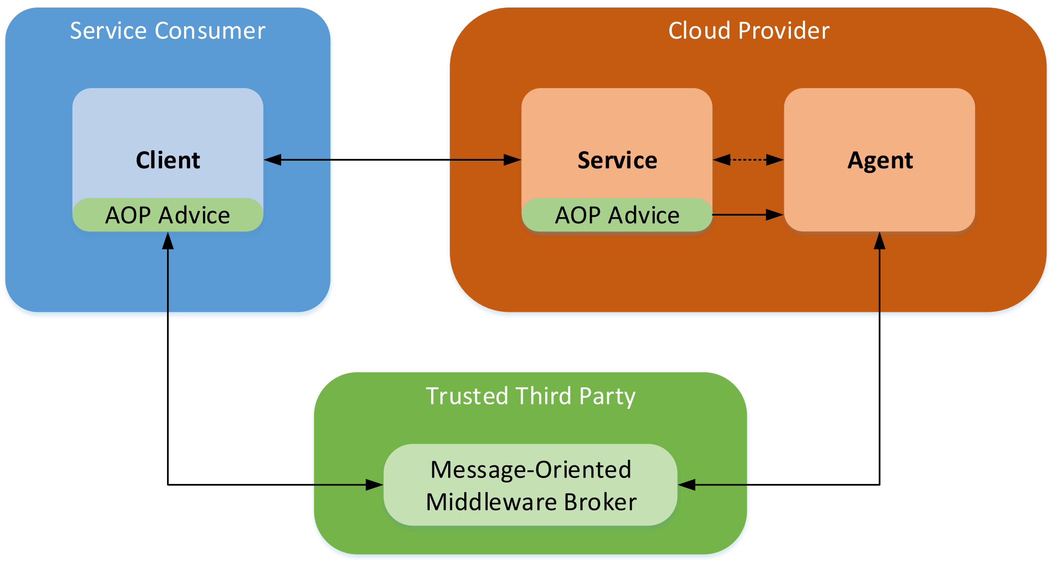


Figure 4.1: Abstract Overview of the Proposed Architecture for Monitoring

Trusted Third Party

Besides the two signature parties, i.e., service consumer and provider, we introduce a third party (see Figure 4.1). This third party has the trust of both signature parties, we therefore denote it as TTP. Its task is to manage the entire monitoring lifecycle of an SLA. Measurements are performed on the consumer and on the provider side, but neither provider nor consumer interprets the measurements. The responsibility of arbitrating metrics, i.e., deciding exact values and checking for violations, is outsourced to the TTP. This ensures objectivity, since the TTP is neutral for the signature parties and able to compare measurements from both.

Sometimes there is the case that a higher-level view of the measurements is required to calculate metrics. This applies, for example, to aggregation metrics that are composed of other metrics. Also, in the breakdown in Section 3.3, we identified metrics that cannot be calculated on the client side if the SLA is agreed for the entire service, e.g., the throughput or successability metric. The TTP has access to this higher-level view, as it collects all necessary information from the signature parties to form the metrics.

In our approach, we utilize CEP to continuously analyze measurements from both consumer and provider in real-time. From the agreement between the signature parties, the TTP is able to infer conditions under which an SLA parameter is violated. Violations are stored persistently, so, a proof of their occurrence can be given at a later time.

The TTP employs an MOM broker for the communication with service consumer and cloud provider. The broker may run on the same machine as the TTP software, but can also run on a different system, however, it is managed by the supporting party. The use of MOM enables a reliable and asynchronous communication between the parties and ensures that messages are delivered exactly once. Furthermore, some implementations provide a scalable mechanism so a higher communication load can be satisfied.

We propose to also deploy the TTP server software in a cloud instance. A TTP server may manage multiple SLA monitoring processes at the same time for different signature parties. While we do not implement this in our approach, but postpone scaling to future work, the computational resources can be dynamically scaled during times of high demand.

Service Consumer

The service consumer side is represented by the client software. This client invokes the Web service that is deployed in the cloud (see Figure 4.1).

For the monitoring, we provide an AOP library that can be woven into the client software. Every time a service request is executed, the AOP code starts measuring time-relevant metrics. Also, various information about the subsequent response from the service is extracted by the advice. After the invocation is performed, a message containing the measurements is sent to the MOM broker which is managed by the TTP. More on the functioning of this approach is described in the implementation in Section 4.3.2.

The use of AOP allows a transparent and non-invasive measurement on the service consumer side. It neither requires access to the source code, nor is a change of the client software required.

Cloud Provider

Similar to the consumer side, there is an AOP advice on the provider side that is executed within the service. This time, measurements regarding the QoS are carried out on the opposite side, directly at the service.

Apart from the service, there is another component involved in provider-side monitoring, we refer to it as agent (see Figure 4.1). This component is an autonomous software that mainly measures resources that are provided for a service, e.g., CPU and memory. The agent runs at the application level of the cloud like any other application, including the services to monitor. The layer model is depicted in Figure 4.2.

The agent has the capability to monitor multiple services or applications simultaneously, therefore, one agent per cloud instance is sufficient.

Since we have at least two monitoring units on the provider side (the agent, and an AOP advice for each monitored service), each must forward their measurements to the TTP. To this end, we let the advices send their readings over an interface to the agent, which regulates the communication to the MOM broker. This interface may also be used to extend the measurement capabilities of the agent with additional probe applications.

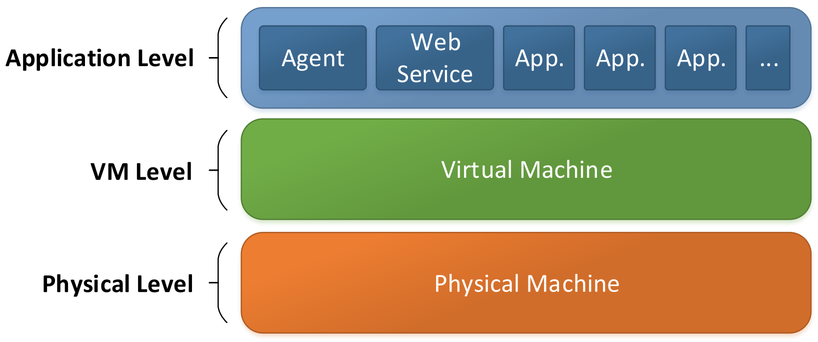


Figure 4.2: Monitoring Agent in the Cloud Layer Model

The combination of agents and AOP allows us to measure a variety of different metrics that may be difficult to measure with one technology alone. In addition, monitoring is isolated from the existing infrastructure. Little to no intrusion is required to perform monitoring on the cloud provider side, as required by Section 4.1.3.

4.2.3 Monitored Metrics

In Section 3.3, we identified and described various candidates for SLA metrics. Often, several different approaches exist in the calculation or representation thereof. This subsection aims to define how we perform the measurements and expressions of the metrics in our approach.

Some metrics are read directly by a monitoring component, e.g., the CPU usage is measured directly by the agent, while others must be calculated from lower-level measurements. In any case, additional aggregation functions can be applied to the metrics, if required. The TTP component of our framework provides this capability by utilizing CEP features. Therefore, a measurement of the general metric is sufficient, as the TTP takes care of optional aggregating.

In the following, we discuss the general metrics that can be monitored with our system.

Response Time This metric is measured and calculated by an AOP advice that is woven into the client software. For the calculation, we use the simplified formula that requires two instants in time, i.e., the time when a request is sent and the time when the associated response is received, see (3.2).

Similarly, at the server side, a related timespan is measured: the execution time. For the calculation, start and end time of the request execution are recorded and the difference is formed. This metric adds objectivity in deciding values because the TTP can analyze measurements from both signature parties.

Throughput Since we focus on application-level monitoring, we decide to use (3.6) for calculating the throughput. It requires the number of processed requests on the server side. As mentioned, for every service request, both the AOP advice of the client and the advice of the service build and send a message containing measurements to the TTP for the analysis. The TTP is then able to form the number of processed requests by counting these messages from the service advice. If only one client is considered, the client messages can also be counted, using (3.5). We evaluate the formula for the throughput over a definable measure time using CEP at the TTP.

Successability Similar to the throughput, the calculation of this metric is performed by the TTP using information in the messages from the AOP advices. We use the equation we have identified for the successabilty, defined in (3.10). It puts in relation the number of successful responses to the number of requests. The AOP advices send their messages with observations whether a request was successful or not. They include information about the success of a request, therefore, the TTP is able to infer the successability metric.

CPU Usage The agent component reads various CPU time metrics at assignable time intervals. The CPU usage can then be expressed as described in (3.12). For the monitoring framework, we define the percentage as the usage since the last interval update, since we already measure on an interval basis. This is different to the notion that describes the usage from the start of the process.

In this work, we use the representation where the total CPU usage is divided by the number of CPUs. This means that a full load at a computer with 4 CPU cores is represented as 100%, not 400%.

Memory Usage Like the CPU metrics, memory metrics are measured by the agent at definable rates. By forming the percentage of the resident memory, i.e., the amount of RAM occupied by an process, the usage is calculated, see (3.13).

Another metric we have identified is the accessibility metric. However, this metric describes a quality aspect of the entire service platform the service is running on, hence it is a system-level metric. We have defined the requirement that monitoring should be based on application level, thus, we do not currently include the accessibility metric.

From the system resource metrics, we selected CPU and memory usage. We show the feasibility of our agent-based monitoring approach by measuring these two. Other resource metrics can be added to our solution in the future.

To monitor the availability in respect to the uptime of a service, we adhere to the idea in [KL03]. It proposes to equip a third party with a measurement service that periodically checks whether the service is up and running. The architecture of our framework makes it possible to incorporate the availability probe within the TTP.

4.2.4 Further Considerations of Trustworthiness

A major requirement for our approach concerns the trustworthiness. So far, we provide a TTP which overtakes the arbitration part. A reliable communication ensures that every measurement by provider or consumer arrives and is considered. Mostly, readings from both parties can be objectively compared and analyzed. We have stated how metrics are interpreted in our approach so that a possible ambiguity between consumer or provider in interpretation and representation can be excluded.

We advise against implementing the monitoring mechanism as a closed source software project. The involved parties should be able to inspect the source code of the system, ensuring that neutrality is granted and no party is favored.

Furthermore, we propose to use code signing[1]. This ensures that the implementations of the monitoring components have not been tampered with to a disadvantage of the opposing party. Also, with low-level resource metrics, we have metrics that can only be measured on the provider side. Especially in this case, code signing allows the consumer to verify that the agent component has not been altered. A similar objective is pursued with the use of a Message Authentication Code (MAC)[2] in our approach. This time, however, it is not the software that is secured against tampering, but the communication. For every message that is sent to the MOM broker for distribution, a MAC is appended that is verified by the TTP.

We perform a comprehensive evaluation of the system's trustworthiness in Section 6.5.

4.3 Reference Implementation

In the following, we discuss a reference implementation of the proposed framework presented in Section 4.2. The discussion also provides a more detailed insight in the functioning of the approach.

We divide the discussion into three sections, each describing the development of an individual component. We begin with the description of the agent in Section 4.3.1. After that, the implementation of the monitoring via AOP is presented in Section 4.3.2. The last part depicts the development of another central component, the TTP, as discussed in Section 4.3.3.

4.3.1 Agent-Based Monitoring: Implementation

The agent component of the monitoring framework is entirely written in Java 8 as a server application.

[1]Code signing is a mechanism that allows users to verify the source and integrity of an application by creating a cryptographic hash signature using a public-key approach.

[2]With a MAC, message receivers are able to verify the source and integrity of the message based on a checksum.

Table 4.1: CPU and Memory Metrics and Their Corresponding Sigar Functions

Type	Metric	Sigar Function
CPU	System Time [ms]	ProcCpu.getSys()
	User Time [ms]	ProcCpu.getUser()
	Total Time [ms]	ProcCpu.getTotal()
	CPU Usage [%]	ProcCpu.getPercent() / CpuCount * 100
Memory	Virtual Memory [Byte]	ProcMem.getSize()
	Resident Memory [Byte]	ProcMem.getResident()
	Shared Memory [Byte]	ProcMem.getShared()
	Memory Usage [%]	ProcMem.getResident() / TotalMemory * 100

At the current stage, the agent supports monitoring application-level CPU and memory usage metrics of a given application/Web service. To accomplish the task of measuring these metrics, the Java bindings for the Sigar library[3] are used. The library also provides functions that allow to access system-level metrics, for example, the *total* CPU, memory or file system usage, however, in this thesis, we aim at solving the task of application-level monitoring. Table 4.1 presents the measuring functions that are implemented in the agent. Note that the variables *CpuCount* and *TotalMemory* are not queried each time a measurement is performed. They are never changing, therefore, they are cached. The Sigar library has built-in functions to get the CPU count as well as the total memory of a system.

To retrieve the metrics of a specific process, which in our case is the service to be measured, we need the Process Identifier (PID) as a parameter for the Sigar library functions. The PID is a unique key that is created and maintained by most operating systems when a process is started, used to clearly identify a process [MSD17, Lin17a].

One approach to get the PID of a process is to query a detailed list of all running processes of the operating system and search for the process name. However, we implement another approach, consisting of a component that is able to start applications and fetch their PIDs. This component is also integrated in the agent software. Starting the processes from within the agent has several advantages over querying a process list:

- Getting the PID of a process after starting it is easier than searching the entire process list, since an operating system may run hundreds of tasks simultaneously [MEMB12]. It is difficult to distinguish and find them only by the application name, because an operating system may run multiple instances of one application, resulting in multiple occurrences of the name in the process list. Furthermore, Java applications pose another special case: Since they require a runtime environment for the execution, they show up with the process name of the *Java Application*

[3]Sigar is a software library to access information of the operating system activity and the underlying hardware; `https://github.com/hyperic/sigar`.

Launcher (`java/javaw`). To retrieve more information about a Java process, a tool like the Java Virtual Machine Process Status Tool (jps)[4] is required.

- Starting the application allows more control over it. Hence, the agent is able to measure application metrics right away from the process start, without missing possibly occurring violations. Additionally, when implementing a communication protocol between agent and TTP in future versions, e.g., to command the agent to start and stop a service for monitoring, this capability is useful as well.

- Applications can be started with additional parameters. This advantage is especially used in the AOP part of the monitoring system. To enable aspect weaving for an application, a weaving agent and a library containing the aspects are specified via command-line parameters. More implementation details on this topic are described in Section 4.3.2.

Java 8 provides no OS-neutral function to retrieve the PID of a process that was started with the Process API. In this work, we use an approach that utilizes the Reflection API[5] to inspect the process instance and retrieve this information from private fields[6]. Depending on the operating system, different implementations for a process are used by the Java runtime. This approach works for Unix-like systems as well as for Microsoft Windows. However, to work with Windows, the Java Native Access (JNA)[7] library is required, since kernel functions must be called to get the PID.

The newest release of Java (Version 9), includes an enhanced Process API, where it is possible to get the PID with a single call to the function pid()[8], working cross-platform. However, at the time of the development, Java 8 was the current version.

When monitoring, it must be taken into account that an application can start other processes, belonging to the application. These so called child processes may spawn other child processes as well, resulting in a hierarchical tree structure. Figure 4.3 represents such a relationship of processes. The process *P1* is the parent process of *P1.1*, *P1.2* and *P1.3*. To the processes *P1.2.1* and *P1.2.2*, *P1.2* is a parent, which is also a child of *P1*. Since in our approach, the agent starts the applications before the monitoring, it can be referred to as the parent process *P1*.

[4]jps is a command line tool for retrieving the state of running Java processes; `http://docs.oracle.com/javase/7/docs/technotes/tools/share/jps.html`.

[5]The Java Reflection API allows to inspect or modify program properties at runtime.

[6]Retrieving the PID of a Java Process; `https://github.com/apache/nifi/blob/master/nifi-bootstrap/src/main/java/org/apache/nifi/bootstrap/util/OSUtils.java`.

[7]JNA is a Java library which provides functions to access native shared libraries; `https://github.com/java-native-access/jna`.

[8]See Java 9 Process API; `http://download.java.net/java/jdk9/docs/api/java/lang/Process.html#pid`.

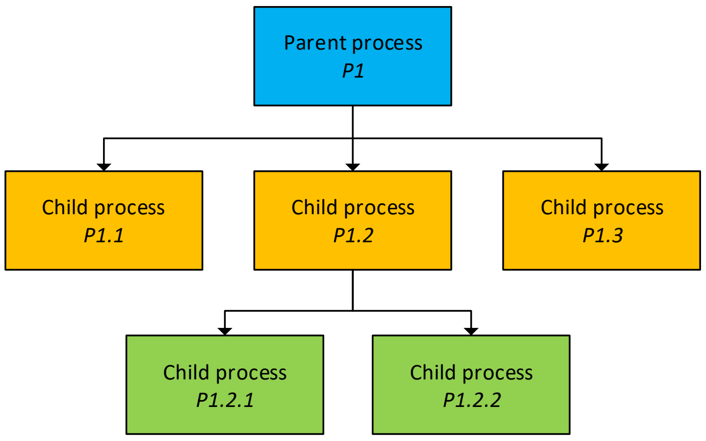


Figure 4.3: Monitoring Agent Process Tree

To build a list of all processes to monitor, Mastelic et al. propose to report the PID back to its parent and therefore back to the agent, once a sub-application has started [MEMB12]. Since we aim to provide a non-invasive solution, we propose another method for building the process list.

We utilize the Sigar library for retrieving a list of all processes managed by the operating system. For each process in the list, we can query the PID of its parent, also called Parent Process Identifier (PPID). By recursively climbing up the process tree until we have a process with the PPID of our main application, we know that all visited processes are children of our application process.

In Listing 4.1, the implementation of our approach for building a process list is presented. Listing 4.2 shows the recursive function for checking whether a process is a child of another one, which is called by Listing 4.1. For every application to observe, the agent queries a process list, and keeps it updated by periodically calling the function in Listing 4.1 at a predefined rate. To reduce the overhead of the monitoring agent, a set of already checked processes is managed by the methods. Before retrieving the parent of a process, a lookup is performed to verify whether a check is already done. Using this mechanism, unnecessary and expensive recursions are prevented. Otherwise, double-checking multiple nodes is not excluded, since we iterate a list of all processes and recursively walk up the tree for each process.

After acquiring a PID list of all processes belonging to the application we want to monitor, we utilize the functions of the Sigar library to get the application-level metrics listed in Table 4.1. We use a timer that triggers the measure functions at a definable rate. All child processes belonging to the application are measured too, and the results are summed up to form a total metric. Listing 4.3 shows how the measurement and aggregation for CPU metrics are performed by the agent. Analogously, the memory metrics are monitored.

Listing 4.1: Retrieving a Set of All Processes to Monitor

```java
// Global sets
Set<Long> processesToMonitor = new HashSet<>();
Set<Long> evaluatedProcesses = new HashSet<>();

Set<Long> findProcessesToMonitor(long pid) {
  // Reset sets
  processesToMonitor.clear();
  evaluatedProcesses.clear();

  // Add the parent process itself to the sets
  processesToMonitor.add(pid);
  evaluatedProcesses.add(pid);

  // Retrieve pids from all processes
  long[] pids = sigar.getProcList();

  for (int i = 0; i < pids.length; i++) {
    // Evaluate each process in the list
    checkProcess(pids[i], pid);
  }

  return processesToMonitor;
}
```

The SLA formed between the consumer and the provider of a service may not always include agreements concerning the CPU or memory usage. To this end, it is possible to define for each application separately which metrics must be observed by the agent. This helps to reduce the runtime and network overhead of the agent since collecting and sending of metrics of no interest is omitted.

The agent's main task is to monitor instructed metrics. It neither interprets the data, nor performs the arbitration of monitored values. We equip the TTP with this ability and let the agent communicate the measurements to the TTP. Since we use MOM as a communication approach, the agent has to address the message broker for this purpose. In our implementation, the communication is achieved by using the Java Message Service (JMS) API[9]. The agent builds messages out of the metrics and sends it via JMS to a queue of the message broker, which is also accessible by the TTP.

Besides the monitored values, there is also other information attached to the messages. The description of the metric to which the value belongs, and a timestamp are added to make the measurement comprehensible and traceable. It must also be considered that

[9]JMS specifies a set of interfaces to allow communication over MOM for Java programs; `http://download.oracle.com/otndocs/jcp/7195-jms-1.1-fr-spec-oth-JSpec`.

Listing 4.2: Recursive Method for Checking If a Process is a Child of Another

```java
boolean checkProcess(long pid, long parentId) {
  if (evaluatedProcesses.contains(pid)) {
    // Already evaluated -> return known result
    return processesToMonitor.contains(pid);
  }
  evaluatedProcesses.add(pid);

  ProcState state = sigar.getProcState(pid);
  if (state.getPpid() == parentId) {
    // Process is a child process -> add to list
    processesToMonitor.add(pid);
    return true;
  } else if (state.getPpid() == 0) {
    // No parent process anymore or parent is scheduler
    return false;
  }

  // Climb up process tree, check parent
  boolean result = checkProcess(state.getPpid(), parentId);
  if (result) {
    // Parent is a child -> this process is a child too
    processesToMonitor.add(pid);
  }
  return result;
}
```

Listing 4.3: Measurement of Application Level CPU Metrics (Multiple Processes)

```java
... // Remainder omitted
double sumCpuUsagePerc = 0;
long sumCpuTotal = 0;
long sumCpuUser = 0;
long sumCpuKernel = 0;

for (Long pidToMonitor : processesToMonitor) {
  ProcCpu procCpu = sigar.getProcCpu(pidToMonitor);
  sumCpuUsagePerc += procCpu.getPercent() * 100 / cpuCount;
  sumCpuTotal += procCpu.getTotal();
  sumCpuUser += procCpu.getUser();
  sumCpuKernel += procCpu.getSys();
}
... // Remainder omitted
```

the TTP can receive messages from multiple agents and agents can measure multiple applications simultaneously. For this case, additional fields must be added to make the messages distinguishable. To overcome the problem of multiple agents, we choose to attach the IP address of the machine an agent is running on. To distinguish applications, we simply add the application name. The MAC is attached in this step too. In detail, we use a Keyed-Hash Message Authentication Code (HMAC) with the cryptographic hash function SHA-1.

In order to not permanently load the network by constantly sending every measured value immediately, we implement an approach where a component performs the collection, aggregation and sending of all messages at a fixed rate. For each monitored application, a queue is managed. Every time a measurement is performed, the measured values are enqueued into this queue. When triggered by a timer, the aggregator component dequeues all application queues with the metrics and aggregates the message to send. When finished, this message is sent to the JMS broker.

We reduce the number of messages with CPU measurements by filtering out readings that have zero CPU usage. If no load occurred, then there is also no change in the CPU time metrics, e.g., user time or total time. A similar strategy is applied to the memory measurements. Only if there is a change in allocation compared to the last reading, a message is sent to the broker. The network load is reduced using these actions, especially in time spans during which no requests to a service are sent.

The agent's monitoring capabilities can be extended with additional *probes*. Probes are small programs that perform measurements and forward them to the agent. We choose TCP sockets for the communication between these two parties and JSON[10] as the data format for the messages. This makes it possible to write the probes in any programing language that implements sockets and JSON. Messages received by the agent (sent by the probes) are temporarily stored in a queue. The aggregator component mentioned before collects and aggregates these messages too before sending to the MOM broker.

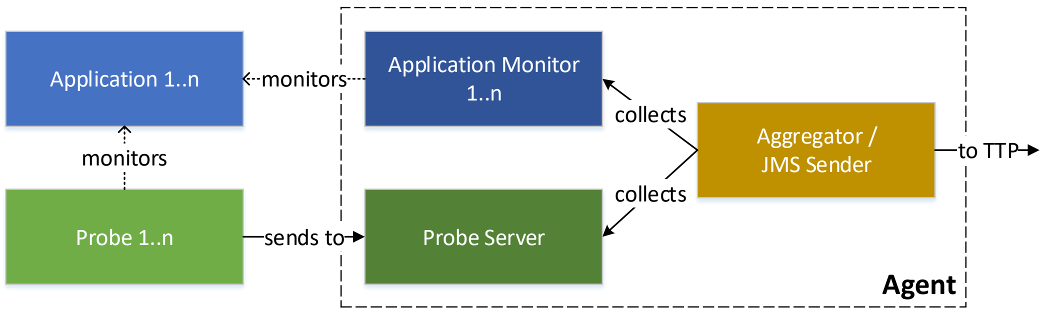


Figure 4.4: Overview of an Agent's Infrastructure

[10]JSON is a lightweight and human-readable data-interchange format; `http://www.json.org`.

Figure 4.4 illustrates a simplified overview of the agent's infrastructure to show how the components work together. Note that the entire depicted infrastructure is deployed in the cloud.

4.3.2 AOP-Based Monitoring: Implementation

In addition to the agent-based monitoring approach described in Section 4.3.1, we also implemented an AOP-based approach. This allows us to acquire different metrics in a non-invasive way that may not be possible or difficult to retrieve using an agent only.

There are two aspects added to the infrastructure, one is woven into the client and one is woven into the service. Similar to the agent, the aspects measure various low-level metrics and forward them to the TTP, which takes care of processing them to higher-level SLAs. On the client side, the aspect measures the response time of the service, as perceived by the client. Furthermore, information about the success of a service call can be extracted. This information will be used in another step to calculate the successability metric. The AOP advice is invoked when the client starts a service request. The aspect woven into the service monitors the execution time, i.e., how long it takes to execute the service function to fulfill the client's request. The invocation starts after receiving a request. Figure 4.5 illustrates how AOP is applied to the two units client and service, in respect to the request and the subsequent response.

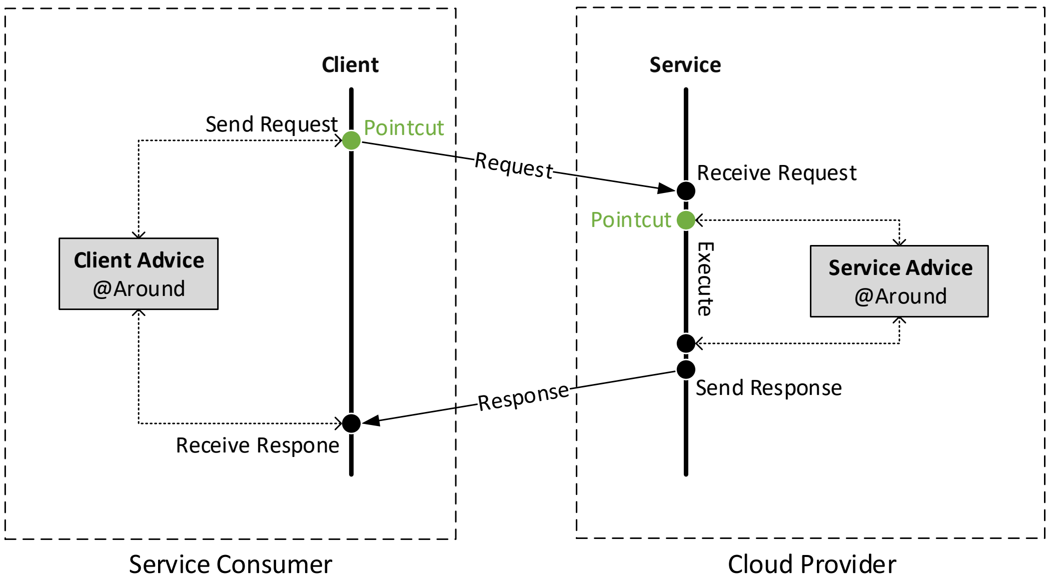


Figure 4.5: AOP Pointcuts and Advices

The two AOP aspects are completely independent from each other, so one could run without the other. However, to provide a trustworthy measurement and arbitration, both are crucial. The TTP checks the measured values for reasonability by comparing both the execution time as well as the response time.

We provide an implementation for the AOP-based monitoring approach that has been tested to work with RESTful Web services written in the Java programming language. To be more precise, we use the reference implementation Jersey[11] of the *Java API for RESTful Web Services* (JAX-RS) specification. We show that our approach is feasible, and other Java frameworks can be adapted as well in a similar manner. The approach explained in this section may also be applicable to other programming languages, since AOP extensions also exist for many programming languages besides Java. Some extensions would be AspectDNG[12] or LOOM.NET[13] for .NET-based languages, AspectC++[14] for C and C++, Pytilities[15] for Python, Go! AOP PHP[16] for PHP or AspectJS[17] for JavaScript.

We utilize the open source Java extension AspectJ[18] for enabling AOP in our software modules. AspectJ supports the following three different weaving approaches [Asp05]:

1. Compile-Time Weaving
2. Post-Compile Weaving
3. Load-Time Weaving

Recalling Section 3.2.3, compile-time weaving requires access to the source code of the software the aspect is woven into, and a new binary file with included aspects is generated by the compiler. Post-compile-time weaving does not need access to the source code, only to the already compiled binary file. Like in compile-time weaving, this creates a new woven binary file. The third approach, load-time weaving, does not modify the binary file at all, and aspects are woven into the byte-code dynamically when class-loading is performed by the runtime environment, in our case by the JVM.

The third process, load-time weaving, is the approach we are striving for. It does not require access to the source code and keeps the existing binary file as it is, resulting in the most transparent and non-invasive method to solve our problem.

[11] Jersey is a framework for building RESTful Web services; `https://jersey.github.io`.
[12] AspectDNG; `https://sourceforge.net/projects/aspectdng`.
[13] LOOM.NET; `https://loom.codeplex.com`.
[14] AspectC++; `http://www.aspectc.org`.
[15] Pytilities; `http://pytilities.sourceforge.net/doc/1.1.0/guide/aop`.
[16] Go! Aspect-Oriented Framework; `https://github.com/goaop/framework`.
[17] AspectJS; `http://aspectjs.com`.
[18] AspectJ was developed by Xerox PARC and is now part of the Eclipse Foundation; `http://www.eclipse.org/ajdt`.

AspectJ requires a load-time weaving agent to enable load-time weaving [Asp05]. For this purpose, we use the `aspectjweaver.jar` that is shipped within the AspectJ bundle. Additionally, aspects to be woven must be known to the weaving agent before the classes to be intercepted are loaded by the JVM [Asp05]. To this end, we add the library containing the aspects to the Java classpath before starting the application. In respect to the requirements for load-time weaving, we end up with the following command to perform non-invasive weaving to a given Java application `application.jar` with the aspects in `aspects.jar`:

```
~ $ java -classpath aspects.jar
         -javaagent:aspectjweaver.jar
         -jar application.jar
```

In Section 4.3.1, we described how our monitoring agent also starts the applications before performing measurements. This is beneficial because the agent can add additional command line arguments and therefore start the application with the above command. Our aspects are woven into the application at load time, and there is no additional action required to enable weaving on the server side. To enable weaving on the client side, the client application can also be started using the same command, but by loading the client aspects. Note that we have implemented two different, independent aspect libraries, one for a client and one for a service.

In the following sections, we present our two implementation approaches, one that yields the aspect library for the client side and the other one that yields aspects for the server side.

Client Side

In order to measure parameters on the client side, we first have to identify suitable positions for join points in the program code, where a monitoring aspect can be woven into. For the acquisition of the response time, we need to measure the time between placing a request and receiving a response from the service, as has already been shown in Figure 4.5.

To this end, we inspect the source code of the client of the JAX-RS reference implementation Jersey. In the class `HttpUrlConnector`[19] we find the method `_apply` which places an HTTP request and reads the response. Listing 4.4 displays this crucial part in the code.

In line 5 of the listing, we see a check whether a request has an entity to send or not. While HTTP methods like POST or PUT send data (in this case the entity), this is not the case with the method GET. However, HTTP headers may also be be sent when using

[19]Jersey Client API - HttpUrlConnector.java full source code; `https://github.com/jersey/jersey/blob/master/core-client/src/main/java/org/glassfish/jersey/client/internal/HttpUrlConnector.java`.

Listing 4.4: Jersey Client API - Send Request, Receive Response (Reduced)

```
 1  private ClientResponse _apply(final ClientRequest request) {
 2    final HttpURLConnection uc;
 3    ... // Remainder omitted
 4    final Object entity = request.getEntity();
 5    if (entity != null) {
 6      ... // Remainder omitted
 7      request.writeEntity();
 8    } else {
 9      setOutboundHeaders(request.getStringHeaders(), uc);
10    }
11
12    final int code = uc.getResponseCode();
13    final String reasonPhrase = uc.getResponseMessage();
14    ... // Remainder omitted
15    return responseContext;
16  }
```

Listing 4.5: Definition of the Pointcut for the Execution of the Function `_apply`

```
@Pointcut("execution(* _apply(..))")
public void applyExecution() {
}
```

a GET method (line 9). Right after transmitting the data, in form of a header and/or entity, the response is read, at first the response code (line 12). Internally, the function `getReponseCode` in the `HttpUrlConnection` class acquires an `InputStream` from the connection to the other endpoint and then proceeds to read the first field in the HTTP header to get the response code. The moment when reading from the `InputStream` is possible, we already have a response from the server. Thus, the `_apply` function is a suitable position in the program code to place a join point for our aspect.

At first, we define a pointcut `applyExecution()` which picks out the join point (see Listing 4.5). We use annotation-based style for aspect declaration in the program code.

We want our advice to be triggered when the method `_apply` is called, but we want to limit it to the class `HttpUrlConnector` of the Jersey Client API. For this purpose, AspectJ load-time weaving is configurable via an XML file [Asp05]. Therefore, we include only this class for the weaver, so no other calls outside this class to retrieve the response code start a measurement.

We form the advice *around* the join point, so we can measure the time before and after the method is invoked. The time frame between these two instants of time represents

Listing 4.6: Measurement of the Response Time Metric Using AOP

```java
@Around("applyExecution()")
public Object applyAdvice(final ProceedingJoinPoint
    proceedingJoinPoint) {
  ... // Remainder omitted
  long startTime = System.nanoTime();
  Object result = proceedingJoinPoint.proceed();
  long responseTime = System.nanoTime() - startTime;
  ... // Remainder omitted
  return result;
}
```

Listing 4.7: Extraction of Additional Request and Response Parameters

```java
... // Remainder omitted
ClientRequest clientRequest =
     (ClientRequest) proceedingJoinPoint.getArgs()[0];
URL url = clientRequest.getUri().toURL();
String requestMethod = clientRequest.getMethod();

ClientResponse clientResponse = (ClientResponse) result;
int responseCode = clientResponse.getStatus();
... // Remainder omitted
```

the response time metric, to be more precise, the duration between the time when a request is sent and the time when a response is received (see Section 3.3). Listing 4.6 represents an extract of the advice to measure the response time. The around advice gives the control to proceed the execution of the intercepted method at any time.

Besides the response time metric, we also extract other important information in the same advice. Note that the `_apply` function (see Listing 4.4) takes a `ClientRequest` object as a parameter and returns a `ClientResponse`. AspectJ provides us with the possibility to inspect parameters of a function, consequently, we can access the `ClientRequest` since it is the first parameter. Also, the around advice allows us to retrieve the result of the original function, i.e., the `ClientResponse`. Using both objects, we extract the (cached) response code, the connection's URL and the HTTP request method (see Listing 4.7).

The response code helps us to draw conclusions about the result of the operation that was performed by the service. An HTTP status code of 2 as the first digit (i.e., 2xx) means that the operation was successful [RFC99]. Using this information, we are able to form the successability metric in the TTP. Similar to the monitoring agent, the aspect itself does not interpret the measured data, the information is simply forwarded to the

TTP, which is dedicated to this purpose. The URL and the HTTP method help us to assign measured values to a specific HTTP endpoint, since a service may provide multiple endpoints/functions. We need this to correlate a monitored execution time on the service to the response time on the client.

Information monitored by the client side aspect is directly sent to the TTP using the JMS API. The TTP can work as a central unit for multiple clients that may perform monitoring at the same time. To distinguish the messages from each other and to carry out an assignment, the client aspect appends the IP address of the machine he is running on to each message. A timestamp that describes the point in time when a measurement was performed is also included for traceability. Again, the MAC is added as well.

Additionally, we define an annotation in our AOP client library. Methods marked with this annotation trigger an AOP advice that measures the time needed for the execution of this method. When programming a client in Java that does not rely on the Jersey Client API for communicating with the service, this annotation can be used to mark custom methods that perform a service request. The execution time of this method, therefore, reflects the response time. The programmer of the client must include the URL of the service that is requested and the HTTP method as parameters in the annotation, so an assignment of the measured time to the endpoint can be done. To state that a request was *not* successful, an exception must be thrown by the method. The AOP advice then can derive the status of the operation – whether an exception is thrown or not – and forward it the TTP. The TTP then counter-checks the measured values of the client with the values of the service to provide trustworthiness when using this annotation approach.

Server Side

The AOP measurement approach on the server side is similar to the one on the client side, with slight differences that are discussed in the following. Currently, we measure the execution time parameter. To this end, an advice is defined that is woven *around* the functions that provide a service endpoint for the clients. The JAX-RS specification defines annotations to label functions with a request method designator that corresponds to an HTTP method [Cor13]. A method labeled with a `@POST` annotation, for example, serves as an endpoint to process HTTP POST requests and so on. We define an AOP pointcut for every annotation described in the specification plus a pointcut for the execution of an arbitrary method (see Listing 4.8). Consequently, we implement an `@Around` advice, so it is woven when a method is executed that is labeled with one of the annotations. Without performing any modifications, this approach of monitoring the execution time may work with other JAX-RS implementations too, besides Jersey. The condition is that they use the same annotations for defining endpoints. As we did with the client AOP library, we provide an annotation to label endpoint methods not conforming with the JAX-RS specification, therefore, these can be measured too.

In order for the TTP to be able to make an assignment of a measured execution time to a specific endpoint of the service, the server aspect has to add additional information to

Listing 4.8: Definition of Pointcuts for the Execution Time

```
@Pointcut("execution(* *(..))")
public void atExecution() {
}

@Pointcut("@annotation(javax.ws.rs.POST)")
public void hasPostAnnotation() {
}

@Pointcut("@annotation(javax.ws.rs.GET)")
public void hasGetAnnotation() {
}

@Pointcut("@annotation(javax.ws.rs.PUT)")
public void hasPutAnnotation() {
}

... // Remainder omitted

@Pointcut("atExecution() &&
   (hasPostAnnotation() || hasGetAnnotation() || )")
public void monitorExecution() {
}
```

the measurement before forwarding it to the TTP. The aspect, therefore, adds the HTTP method and the relative URI path the function is assigned to. The HTTP method is inferred from the request method annotation, and the URI path is built from extracting and concatenating parameters of the surrounding `@Path` annotations. According to the JAX-RS specification, `@Path` annotations describe a relative URI where a Java class or function is hosted [Cor13].

On the server side, measured metrics are not directly sent to the TTP via JMS, but forwarded to the monitoring agent first. As discussed in Section 4.3.1, the agent can be extended with probes that communicate their measurements to the component via TCP sockets. The server aspect uses this interface. The information measured is converted to a JSON message and sent to the agent component, which takes care of aggregation, and regulates sending to the TTP.

4.3.3 Trusted Third Party: Implementation

The server component, which fulfills the task of arbitration, is also written in Java 8 as a server application. We assume that the negotiation of SLAs between provider and consumer is already done and the outcome of this process is a WSLA file. We also assume that the TTP has access to this document.

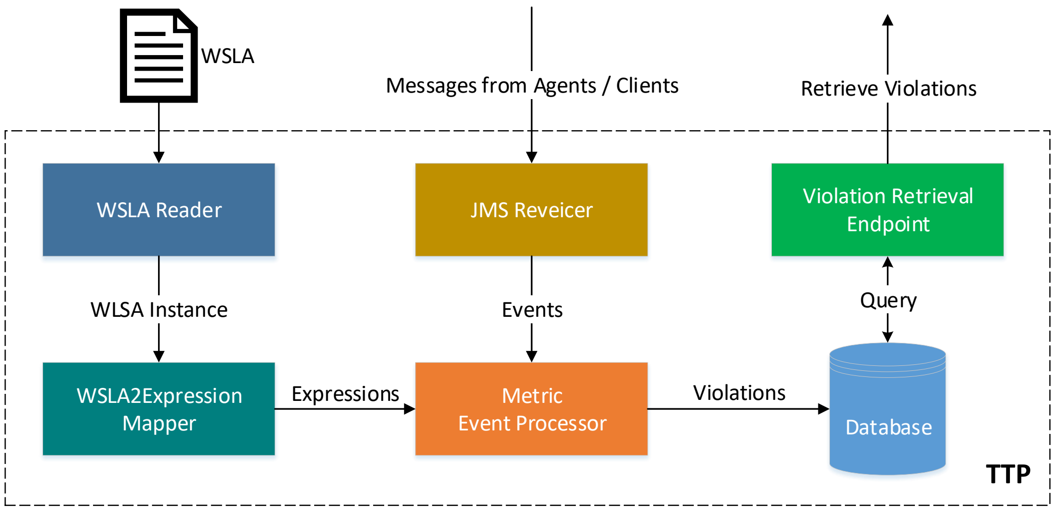


Figure 4.6: Infrastructure of the TTP Implementation

The TTP uses CEP to perform the mapping from low-level measurements to higher-level metrics, and to detect violations. By extracting relevant information from a WSLA document, CEP statements are created and added to the CEP engine. Received measurements from clients and agents are then sent as events to the CEP engine as well. When detecting violations, a detailed log about the deviation is stored into a database. The involved signature parties, i.e., the provider and the consumer, may request detailed information about violations over a Web service. An infrastructure overview of the TTP implementation is shown in Figure 4.6. In the following sections, we explain the most important components of the infrastructure in greater detail.

WSLA Reader

When providing a WSLA document to the TTP, the file is read and mapped to Java objects. Since the document structure is rather complex, the mapping has the advantage that we can easily traverse and retrieve its elements. From the XML schema definition of the WSLA specification [LKD+03], we generate Java classes by using the `xjc`[20] binding compiler. The mapping is done by utilizing the Java Architecture for XML Binding (JAXB) API[21]. The outcome of this component – a WSLA instance – is forwarded to the next next component, i.e., the *WSLA2Expression Mapper*.

[20]xjc is included in the Java Development Kit; `https://docs.oracle.com/javase/8/docs/technotes/tools/unix/xjc.html`.
[21]JAXB allows to bind an XML file to Java classes; `http://www.oracle.com/technetwork/articles/javase/index-140168.html`.

WSLA2Expression Mapper

The *WSLA2Expression Mapper* is a core component of our TTP implementation. The outcome of this mapper are statements for the CEP engine. For every SLA parameter in the WSLA document that requires arbitration, one statement is built.

As described in Section 3.2.1, a WSLA file includes three major sections – parties, service definition and obligations. For every operation a service provides, SLA parameters and their connected metrics are defined. We distinguish between simple SLA parameters, i.e., parameters that can be directly mapped to measured metrics, and complex SLA parameters that are composed from several metrics and may apply functions to them.

To put this into context, we provide an example of a complex parameter – the average response time of a service operation. Listing 4.9 shows how this parameter could be defined using the WSLA language. The definition of the SLA parameter starts in line 3, and the corresponding metric is given in line 4. The *AverageResponseTime* metric is calculated by applying an average function to another metric – the *ResponseTime* (see lines 10-15). This metric is also defined (lines 16-18). It is assigned to a measurement by the consumer.

Listing 4.9: Extract of a Service Definition Section with a Complex SLA Parameter

```
1  <ServiceDefinition>
2    <Operation name="Operation"
        xsi:type="WSDLSOAPOperationDescriptionType">
3      <SLAParameter name="OperationAverageResponseTime" type="double"
          unit="milliseconds">
4        <Metric>AverageResponseTime</Metric>
5        <Communication>
6          <Source>ServiceConsumer</Source>
7        </Communication>
8      </SLAParameter>
9      ...
10     <Metric name="AverageResponseTime" type="double"
          unit="milliseconds">
11       <Source>ImageServiceConsumer</Source>
12       <Function xsi:type="Average" resultType="double">
13         <Metric>ResponseTime</Metric>
14       </Function>
15     </Metric>
16     <Metric name="ResponseTime" type="long" unit="milliseconds">
17       <Source>ServiceConsumer</Source>
18     </Metric>
19     ...
20   </Operation>
21   ...
22 </ServiceDefinition>
```

The service definition section does not specify the requirements for an SLA parameter, this is done in the obligation section. An exemplary objective is shown in Listing 4.10. Boolean conditions that must be satisfied for an SLA parameter are specified via expressions (lines 8-13). In this case, it is a violation if the average response time exceeds a threshold of 2500 milliseconds.

Listing 4.10: Extract of an Exemplary Obligation Section for an SLA parameter

```
 1 <Obligations>
 2   <ServiceLevelObjective name="AverageResponseTimeSLO">
 3     <Obliged>ServiceProvider</Obliged>
 4     <Validity>
 5       <Start>2017-01-01T14:00:00.000-05:00</Start>
 6       <End>2018-01-01T14:00:00.000-05:00</End>
 7     </Validity>
 8     <Expression>
 9       <Predicate xsi:type="LessEqual">
10         <SLAParameter>AverageResponseTime</SLAParameter>
11         <Value>2500</Value>
12       </Predicate>
13     </Expression>
14     <QualifiedAction>...</QualifiedAction>
15   </ServiceLevelObjective>
16    ...
17 </Obligations>
```

Using the information of how SLA parameters and their requirements are defined in a WSLA document, we can build CEP statements that are given to the CEP engine to perform mapping and arbitration of the agreed parameters.

In our implementation, we use Esper[22] for CEP. The tool provides a so-called Event Processing Language (EPL) with an SQL-like syntax to build statements.

Internally, our mapper component uses predefined “skeleton statements” that are filled with parameters, functions and requirements for SLAs at runtime. Also, some predefined blocks may be appended to extend the functionality of the statements, e.g., to consider values within a specified time window only. Various “skeleton statements” exist to express simple and complex SLA parameters, but also to express a ratio of an occurrence of a particular event. Messages received from consumer or provider may contain multiple monitored metrics at once, but also contain meta data, e.g., source IP addresses and the measured service endpoint or application. In order to perform a mapping of WSLA-defined low-level metrics to the specific fields in the messages, the mapper stores predefined assignments of metrics to fields.

[22]Esper is an open source product that allows processing and correlation of event streams; `http://www.espertech.com/products/esper.php`.

The WSLA language specifies some functions that can be applied to metrics [LKD+03], e.g., *Average, Median, Sum, Max*, to name a few. Our mapper component directly translates these functions to equivalent EPL functions. Similar, expressions in the obligations section are translated to clauses the Esper CEP engine understands.

Based on our example with the average response time, the EPL statement in Listing 4.11 is created by the *WSLA2Expression Mapper*.

Listing 4.11: Generated EPL Statement for the Average Response Time

```
1 SELECT  AVG(responseTime) AS monitoredvalue,
2         'responseTime' AS metrictype,
3         'avg<=2500.0' AS requirementdesc, *
4         FROM ClientInfoMessage(responseTime >= 0)
5         GROUP BY serviceName
6         HAVING AVG(responseTime) > 2500.0 AND COUNT(*) >= MIN_QUANTITY
```

The statements in Listing 4.11 bear the following semantics:

- As mentioned, we retrieve the message type and the corresponding field for the metric from a predefined assignment. Here, this is the property `responseTime` (lines 1, 4, 6) of the object `ClientInfoMessage` (line 4).

- Messages with negative and therefore invalid values are not considered and filtered out first (line 4).

- Since the TTP may perform arbitration for many independent services, we cannot apply an aggregate function over all received messages that describe measurements from different services. Hence, we group the messages beforehand. We choose a property/field that clearly assign measurements to a service or application. As mentioned, messages have meta-information included. We can use the name of the service endpoint or application for grouping (line 5).

- A condition selects only events fulfilling this clause (line 6). To detect violations of SLA parameters, we build a condition that is the opposite of the expression that the WSLA obligation states. We also define a minimum threshold for a quantity of messages that must be reached before an average is calculated. Else, when starting with arbitration, outliers in the first measurements could already lead to a violation.

- We select various information (lines 1-3) which will help us later to insert violations to the database. The value that led to the violations is selected, but also a description of the obligation. Other information, like time of measurement, source IP address and so on is included in the message, thus, the entire message is picked via the asterisk selector.

Metric Event Processor

This component provides the interface to the Esper CEP engine. Statements passed by the *WSLA2Expression Mapper* are added to the engine. Furthermore, when a message containing metrics is received, its MAC is verified first (by the JMS Receiver component), then it is sent as an event to the CEP engine as well. An integrated listener component receives callbacks when a violation is detected. It has the values we selected with the EPL statements as parameters. The information is processed and stored into a relational database.

Violation Retrieval Endpoint

Our implementation of the TTP exposes a RESTful Web service. Currently, it allows provider and consumer of a service to retrieve detailed logs of detected violations from the database. The involved parties may perform polling at some interval on this resource, so they would be notified if an SLA violation occurred. We reserve a more advanced notification system for future work.

CHAPTER 5

Experimental Testbed

Following the implementation of our monitoring framework, we design and develop an experimental testbed in this chapter. This allows us to execute various simulation scenarios that contribute to the evaluation of our approach. We strive for an easily configurable solution, which is also suitable for use in a cloud environment.

We start with a brief presentation of our exemplary cloud service, which plays a major role in our testbed, in Section 5.1.

Section 5.2 then explains the complete test environment. This also concerns the configuration as well as the workload that is used for experiments.

5.1 Exemplary Cloud Service

Based on the motivational scenario for our work (see Section 1.3), we implement an exemplary image processing service. This service and its consumer are the two main components that are monitored to evaluate our approach. The development of a custom service has the advantage that we can modify it for various simulation scenarios. For example, we can inject longer execution times that simulate a shortcoming of resource provision from the cloud provider.

We develop a RESTful Web service in Java that provides an API to resize or rotate images. We use the server library of the Jersey framework (i.e., the reference implementation of the JAX-RS specification) for this purpose. The application is embedded within a Grizzly[1] HTTP container and exported as *uber-jar*[2]. This has the advantage that the

[1]Grizzly is a multi-protocol framework based on the Java NIO API; `https://javaee.github.io/grizzly`.

[2]An *uber-jar* (also known as *jar with dependencies* or *fat jar*) is a Java archive which contains all application dependencies and resources.

service works standalone and can easily be started and measured by the monitoring agent component.

The image processing API is designed in such a way that only a single request is necessary to resize and/or rotate an image. Within the request, an image and the settings for processing are provided. The service then processes this image and returns the result with the response. More precisely, the endpoint is defined as follows:

```
POST /imageprocessor/resize HTTP/1.1
Accept: multipart/form-data
```

The endpoint accepts the mediatype `multipart/form-data`[3]. Several parameters can be transmitted with this request message, as described in Table 5.1. Note that only the image parameter is required. The other parameters are optional and can be combined as desired, e.g., an optional rotation and/or different approaches to resize the image can be applied.

Table 5.1: Parameters for the Exemplary Image Service (`multipart/form-data`)

Key	Type	Description
image	File	The image file to process (jpg, png, gif)
width	Text	The target width in pixel
height	Text	The target height in pixel
size	Text	The longer side of the image is shrunken to this size while keeping the original ratio (can be used instead of width and height)
rotation	Text	The target rotation (CW_90, CW_180, CW_270, FLIP_HORZ or FLIP_VERT)

When an image has been processed successfully, the following response message is returned by the service:

```
HTTP/1.1 200 OK
Content-Type: image/jpg or image/png or image/gif
IMAGE PAYLOAD
```

A successful response returns the HTTP status code 200 (OK) [RFC99] and the image. The image file format is not altered by the service, therefore, the content type in the response message depends on the file format of the source image.

The consumer of the described image processing service is a simple client written in Java. In order to invoke the endpoints of the service, the Jersey client library is used.

[3]The media-type `multipart/form-data` can be used to upload (binary) files over HTTP; see RFC 2388 `https://www.ietf.org/rfc/rfc2388.txt`.

5.2 Test Environment

Up to this point, we have implemented all components that are required to evaluate our approach. In addition to the components for monitoring, i.e., agent, AOP libraries and TTP server software, we now have an exemplary cloud service and a consumer of it. What is still missing is the outline of a suitable test environment where we conduct experiments required for the evaluation. In the following, we discuss our solution.

5.2.1 Overview

To be as close to a real-world scenario as possible, we carry out the experiments in a distributed cloud environment. The setup consists of two public cloud servers and a local PC. As depicted in Figure 5.1, we use three instances which reflect the parties involved in the monitoring process, such as cloud provider, service consumer and the TTP. The cloud servers are hosted in the Amazon Elastic Compute Cloud (EC2)[4]. EC2, also part of AWS, is an IaaS [MHJ+11] service, i.e., we are provided with the virtual hardware and have full control over the operating system and software.

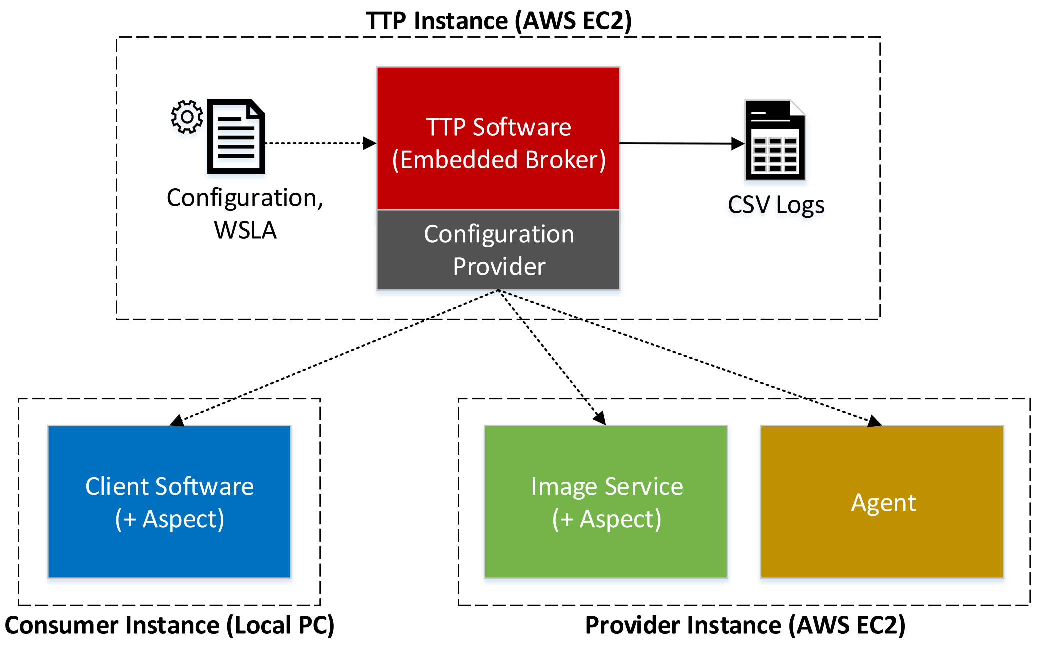

TTP Instance (AWS EC2)
Configuration, WSLA
TTP Software (Embedded Broker)
Configuration Provider
CSV Logs
Client Software (+ Aspect)
Image Service (+ Aspect)
Agent
Consumer Instance (Local PC)
Provider Instance (AWS EC2)

Figure 5.1: Overview of the Test Environment

[4]Amazon EC2 provides virtual computing environments; `https://aws.amazon.com/ec2`.

Each component in the test environment can be configured individually. This allows a variety of different experiments which can be conducted, however, with three distributed computers this may be laborious. To simplify the configuration, we provide a single, central settings file from the TTP instance, since this is also the central unit in our monitoring mechanism. The file is served via HTTP by a so-called *Configuration Provider* that is embedded within the TTP software. When starting the agent, service or client software, the URI of the configuration endpoint can be specified via command line arguments. The configuration allows different negotiated SLAs to be set for the arbitration, i.e., by providing a WSLA document.

The TTP software logs every received measurement as well as every detected violation. The logs are written to different CSV[5] files. These are used for analysis and to evaluate our approach, which is shown in Chapter 6.

5.2.2 Hardware and Operating System Configuration

The basic system configuration of our experimental testbed is described in Table 5.2. The two cloud servers, i.e., the provider and the TTP instance, are hosted in the region "US East (Ohio)", whereas the consumer instance is a regular PC. Note that the cloud server's CPUs are virtual and may be shared among several users/instances.

Table 5.2: Hardware Configuration of the Test Environment

Name	Operating System	CPU	Cores	Memory
Consumer	Windows 10	Intel Core i5 @ 3.4 GHz	4	8 GB
Provider	Ubuntu Server 16.04	Intel Xeon E5 @ 2.4 Ghz	1	1 GB
TTP	Ubuntu Server 16.04	Intel Xeon E5 @ 2.4 Ghz	1	1 GB

5.2.3 Experimental Workload

The workload for the experiments comprises several images in different resolutions and consequently different file sizes. The images[6] are illustrated in Figure 5.2 and a brief comparison regarding their dimensions and file sizes is given in Table 5.3.

With the mixed selection of images, we intend to cover possible real-world scenarios. Different dimensions are expected to result in different execution times for resizing. Preliminary experiments on the public cloud showed that the execution times to shrink the images to a size of 300x200 pixel vary from about 15 milliseconds (smallest image) to about 700 milliseconds (biggest image). The size of the images may also affect the overall response time for the consumer, since they must be transfered over the network. We will verify this using experiments in Chapter 6.

[5]CSV (Comma-Separated Values) is a text-based file format; see RFC 4180 `https://tools.ietf.org/html/rfc4180`.

[6]All images are under Creative Commons license (CC0); see source `https://pixabay.com`.

Table 5.3: Comparison of the Experimental Workload

Name	Resolution	File Size
Image1 - Small	640×426	90 kB
Image2 - Medium	1280×898	239 kB
Image3 - Big	1920×1280	692 kB
Image4 - Huge	4896×3264	2,400 kB

(a) Image1 - Small

(b) Image2 - Medium

(c) Image3 - Big

(d) Image4 - Huge

Figure 5.2: Experimental Workload Images

CHAPTER 6

Evaluation and Results

In this chapter, we carry out the evaluation of our monitoring framework presented in Chapter 4. To this end, we perform several simulations in our experimental testbed, which we have described in Chapter 5, and discuss the results.

We start by providing a complete simulation scenario for our mechanism in Section 6.1. This includes the discussion of the configuration (Section 6.1.1) and the interpretation of the outcome of the experiment (Section 6.1.2).

Further experiments are conducted in Section 6.2 to assess the correctness and accuracy of the metrics of which measurements are implemented in the monitoring framework. After that, in Section 6.3, we examine and discuss options for the measurement interval of the low-level resource metrics. Finally, one of our key concerns, the trustworthiness in SLA monitoring and arbitration is evaluated in Section 6.5.

In Appendix A, we also provide detailed instructions for repeating all the experiments shown in this chapter.

6.1 Exemplary Test Run

We present in detail one exemplary scenario. In this scenario, we simulate a shortcoming of resource provisioning by the provider to show the functioning of our monitoring mechanism. In case of correct execution, the failures should be recognized by the TTP. To this end, we define an SLA that specifies a maximum threshold for the response time of the image processing service. Furthermore, we inject poor execution times into this service with a certain possibility. These are also reflected in the response times perceived by the service consumer, since the execution time is part of the response time.

6.1.1 Configuration of the Scenario

We start with the description of the most relevant parameters for the test run. The settings are presented in Listing 6.1. Note that this is just a subset of all adjustable parameters. The remaining parameters are explained in more detail in the corresponding experiments.

Listing 6.1: Configuration for the Exemplary Test Run

```
 1  wsla.file=response_time_agreement.xml
 2
 3  image.type=small
 4  image.target.size=300
 5  image.rotation=NONE
 6
 7  request.count=1000
 8
 9  execution.time.delay.rate=0.02
10  execution.time.delay.time=1000
11  execution.time.delay.variation=100
```

In line 1, we define the WSLA file that is provided to the TTP. The document is quite similar to the one we have shown in the TTP implementation in Section 4.3.3. The full WSLA file for this scenario is provided in Appendix B. We define a threshold of 1000 milliseconds for the response time, which means that any transgression of this value must be reported as violation.

Lines 3 through 5 are concerned with settings for image processing. The first parameter specifies the image file that is sent to the service for processing, such as the ones shown in Section 5.2.3. Based on the size identifier (small, medium, etc.), the corresponding image from the experimental workload is selected. The requested size for resizing is given with the parameter in line 4. The value 300 indicates that the longer side of the provided image is shrunken to 300 pixel, while the aspect ratio is maintained. The last parameter for processing defines whether and how a rotation is applied to the image.

An experiment is composed of multiple successive requests that are invoked by our client software. The number of requests is specified in line 7, in this case we let a *single* client execute 1000 service invocations. Note that all our simulations in this chapter are conducted with one client and not several, as described in Chapter 5.

The parameters in lines 9 through 11 are used to inject longer execution times into to service. The first value describes a percentage of requests that are deliberately delayed, whereas the second value describes the delay time in milliseconds. Note that the delay time is added to the actual execution time and does not substitute it. To avoid that all delays are of the same length, we introduce a third parameter in line 11. A random time between zero and the given value (in milliseconds) is also added to the above parameter to form the effective delay time. The sample configuration can be described as follows:

At an injection rate of 2%, the service influences 20 out of 1000 invocations. These 20 requests are delayed by 1000 to 1100 milliseconds.

All these entries indicate the necessity for a central configuration. Otherwise, each component must be individually configured, since the settings affect the behavior of several components. In this case, we have the WSLA definition for the TTP, the request parameters for the client, and the delay settings for the image processing service.

6.1.2 Test Run Results

We run the simulation and then analyze the log files created by the TTP. Figure 6.1 depicts the response times measured by the client aspect. Note that we show the first 500 requests only, as this results in a clearer presentation. However, the evaluation is carried out with all 1000 measurements. In addition to the response times, we display the threshold and mark all by the TTP detected violations.

The TTP receives all 1000 measurements from the client as well as from the service, thus, we have a first verification that the communication in our monitoring system is reliable. In the experiment, response times above the specified threshold were correctly detected as SLA violations. A total of 20 violations was logged, which corresponds to the delay injection rate of 2%.

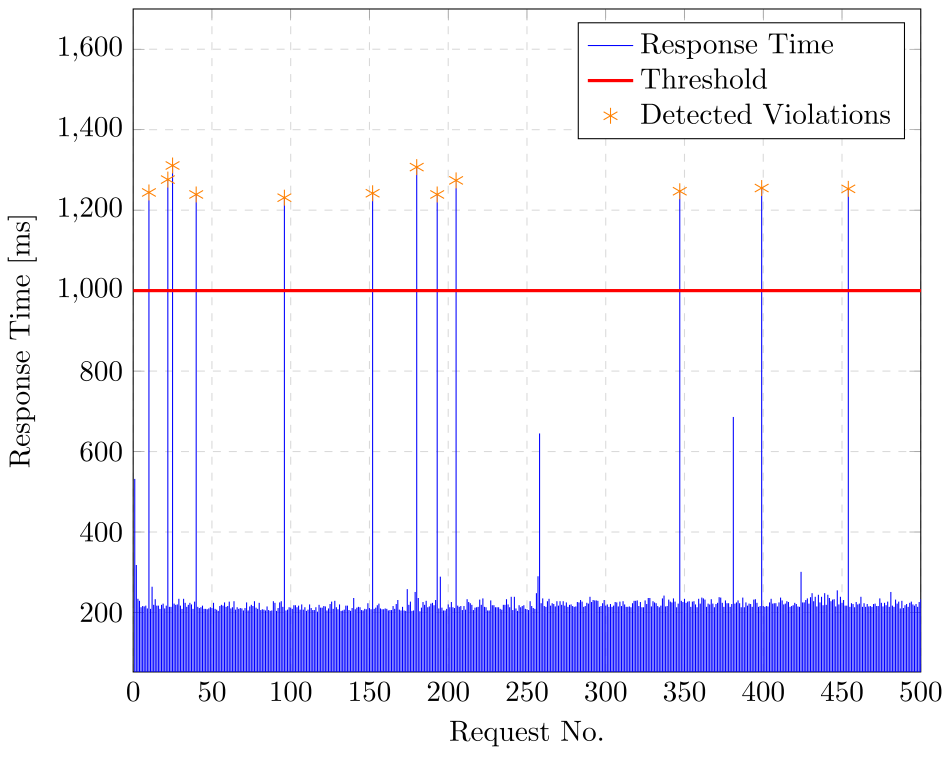


Figure 6.1: Monitored Response Times

For comparison, we also present the measured execution times of the service in Figure 6.2. We mark the occurred (response time) violations in this plot too. What is immediately noticeable are the injected execution times, which have contributed to the response time violations. Overall, the execution times are much shorter, as they do not include the network and client latency. If we omit the injected times, we have an average value of about 220 milliseconds for the response time and an average value of about 17 milliseconds for the execution time to shrink the small image. The difference between the two is the network and client latency, of which the former is likely to have more weight.

In Figure 6.1, apart from the injected times, other high values can be found which are far above average, e.g., request numbers 258 or 382. However, their corresponding execution times are not conspicuously high (see Figure 6.2). These outliers are therefore mainly caused by bad network performance. As in this case, a high network latency could even result in a response time violation, although this may not be the fault of the cloud provider. The necessity arises to counter-check the measurements from both the consumer and the provider. The TTP can then decide whether a real SLA violation because of resource under-provisioning by the provider has occurred.

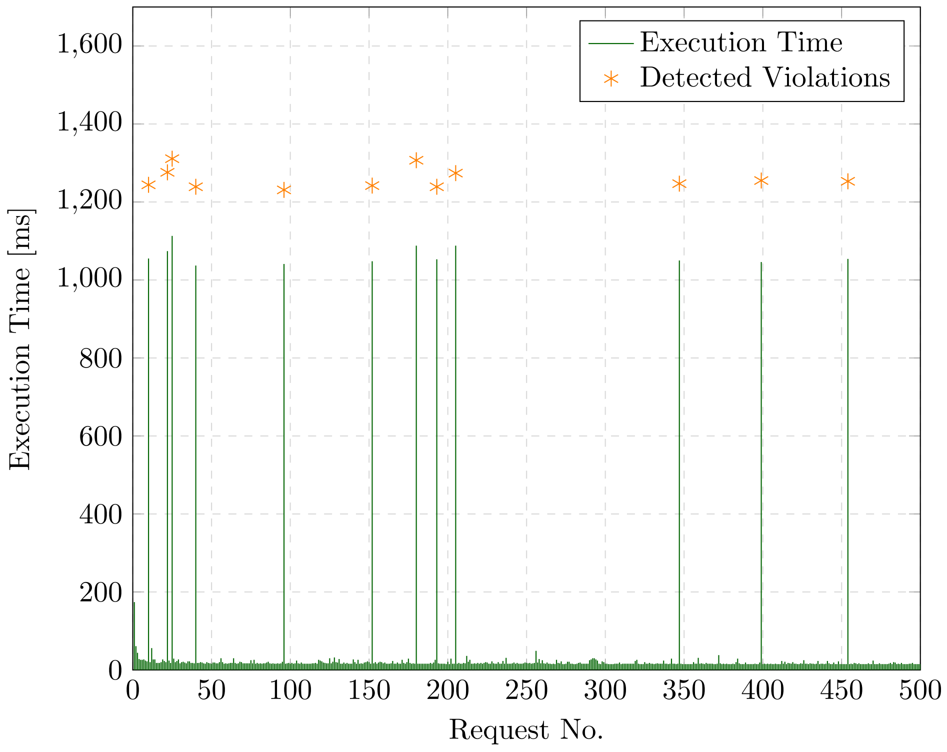


Figure 6.2: Monitored Execution Times

The very first reading of the response or execution time is strikingly high too. We assume that this is due to the behavior of the Just-in-Time (JIT)[1] compiler of the Java VM, where no code optimizations have been done yet. We argue that in reality, the applications, in particular the service, are not restarted each time before use, but rather are active for a longer time. For this reason, we discard the first reading of response or execution time in the following experiments. Other parameters should not be affected by this behavior, since they do not require measurements of time.

6.2 Correctness and Accuracy

In this section, we show that our implemented approach for monitoring provides correct and accurate readings. For each metric that our mechanism is capable of measuring, we carry out experiments and analyze their results in detail.

6.2.1 CPU Usage

The measurement of the application-level CPU usage is implemented in the agent component of our monitoring mechanism. For this simulation, we also use the Linux `top` tool [Lin17b], which displays CPU and memory-related information of running processes, and log its output. A comparison of both measurements is then carried out.

The following command line options are used for the `top` tool.

```
top -b -d DELAY_TIME -p PID
```

The `-b` option switches to batch-mode operation[2]. This allows for easier logging. We set the delay time, i.e., the refresh rate, with the `-i` option. Finally, we specify the application for monitoring with the PID of the corresponding process (`-p` option).

For an easier handling of the simulation, we implement the startup and logging of the `top` tool in the agent. This is done as soon as the PID of the Web service is retrieved (see the agent implementation in Section 4.3.1). In order to allow a comparison, the update rate of the `top` tool is set to the same as for the agent-based monitoring.

The simulation configuration has the option to specify whether the CPU usage is logged with the Linux `top` tool. For this scenario, we enable this switch and set a monitoring interval of 1000 milliseconds for both the agent and the `top` tool:

```
log.usage.top=true
system.metrics.monitor.interval=1000
```

To provide a continuous load over a period of time for the service, we run the experiment with 2 000 image resizing requests from the client. Figure 6.3 depicts the results of this experiment.

[1]A JIT compiler translates program code into machine code at run-time as soon as it is needed.

[2]Linux `top` command - man page; `https://linux.die.net/man/1/top`.

The first few seconds in Figure 6.3 represent the startup of the exemplary service. Now the advantage of starting the application by the agent opens up, otherwise these readings are difficult to capture. After all client requests have been processed, the CPU usage goes zero, since no more requests occur (at 590 seconds).

The CPU usage is expressed in percent and describes the use of the CPU by the application in the last measured period, in our case in the last second. Note that the two different measurement approaches are not completely in sync, which is why the readings differ slightly. Their synchronization is a hard task because the `top` tool has its own timer for updating, thus, we can not log the usage at the same time as the agent does measuring. Especially at peak values, the deviation is noticeable. However, the values before or after the peak compensate for this. For instance, consider the peak at 230 seconds of runtime. The `top` tool measures a CPU usage of 67% while the agent measures only 35%. If we look at the next value, the `top` tool has 8% and the agent 39%. As a result, taking into consideration a slightly larger time frame, the usages are almost the same again. Therefore, the deviation is due to the slightly offset measurement intervals, i.e., because one approach assigns a high but short load to a different interval than the other approach.

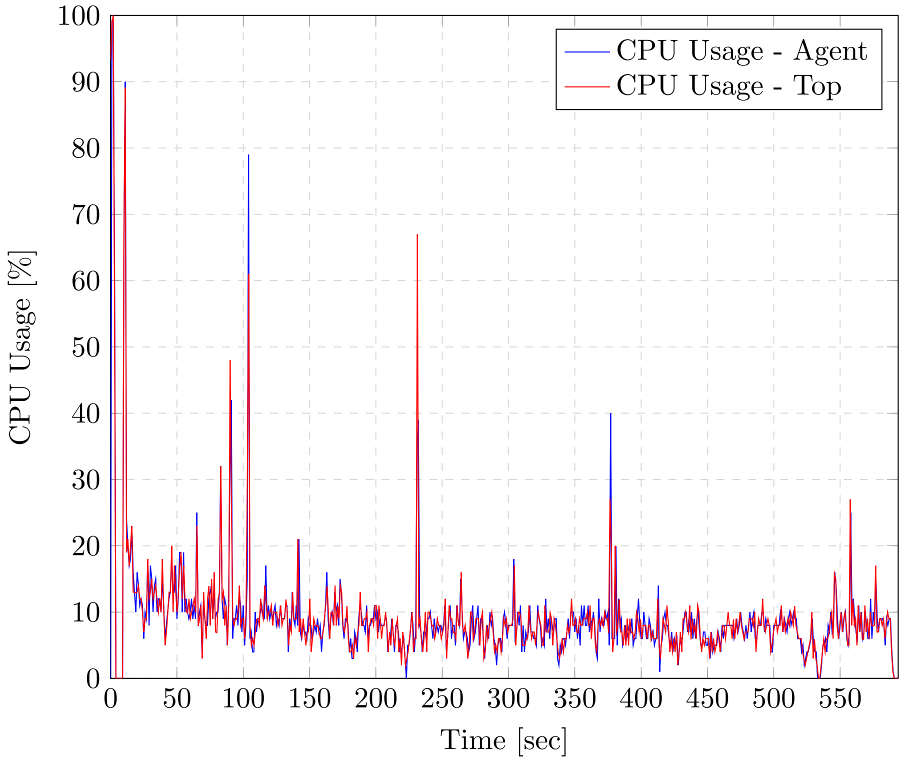


Figure 6.3: Comparison of the CPU Usage Measurements

We perform a paired t-test with the measurements of both approaches. The parameters and the result of the test are displayed in Table 6.1. We do *not* reject the null hypothesis, meaning there is no significant difference between the two population means. We were able to find consistent results in a similar experiment using Microsoft Windows as the operating system, therefore, the platform independence requirement (see Section 4.1.3) is satisfied for this parameter.

We have seen that short CPU usage peaks are difficult to capture. If a negotiated SLA requires their detection, the measurement interval must be shortened. However, the shortening may come with other disadvantages, e.g., a higher resource consumption of the agent or a higher network load may arise as a result. Since the monitoring interval also concerns other resource metrics, we discuss it separately in Section 6.3.

Table 6.1: Paired t-Test for the CPU Usage in %

	Agent	Top
Mean	9.177	9.156
Variance	76.548	75.823
Observations	593	593
Hypothesis H_0	$\mu_1 = \mu_2$	
Hypothesis H_1	$\mu_1 \neq \mu_2$	
Significance Level α	0.05	
t Statistic	0.145	
p Value (2-tail)	0.885	
t Critical (2-tail)	1.964	
Supported Hypothesis	H_0	

6.2.2 Memory Usage

We evaluate the accuracy of the memory usage using the same methodology as for the CPU usage, i.e., by comparing the measurements with the Linux `top` tool output. The results are shown in Figure 6.4. Note that the memory usage has been recorded in the same experiment as the CPU usage presented in the last subsection, therefore, the simulation settings are the same. The update rate is also one second.

Similar to the CPU usage evaluation, the reading intervals of the two different approaches are slightly offset. Apart from that, there is high correspondence between both measurements. A paired t-test (see Table 6.2) affirms this statement. Again, we do *not* reject the null hypothesis, i.e., we found no statistically significant difference between the two measurements. We also verify the results with the Windows operating system. Generally, it is noticeable that the memory usage is not changing as fast as the CPU usage. Furthermore, we did not observe as many peak values. Even after all requests have been handled (at 590 seconds), there is no instant release of the resource by the application, which is significantly different to the CPU usage.

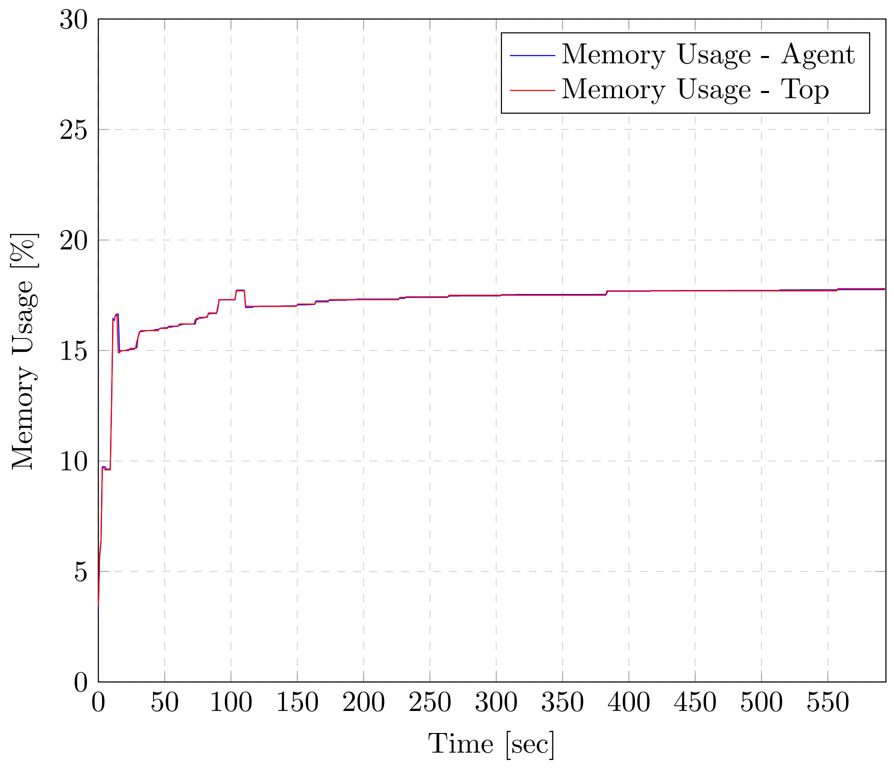


Figure 6.4: Comparison of the Memory Usage Measurements

6.2.3 Response Time

In order to show the accuracy of the response time metric in our approach, we introduce an additional measurement *directly* in the exemplary client software so that we can make a comparison with the AOP measurement. Optimally, there should be hardly any differences between the readings, otherwise a measurement with AOP is not suitable for this metric.

We run the experiment with all four images in our experimental workload with the purpose to cover a wide range of different response times. To achieve this within a single simulation run, we set the image type to *batch* in the configuration.

```
image.type=batch
```

With this setting, the client divides the total number of requests equally among the various image types. For instance, in 1000 requests, the first 250 have the small image as a parameter, the next 250 the medium sized image, and so on.

The result of the experiment is shown in Figure 6.5. The gray line depicts the regression line. Note that the first reading is discarded, so we have both sets with 999 measurements each from the client as well as from the AOP advice.

Table 6.2: Paired t-Test for the Memory Usage in %

	Agent	Top
Mean	17.162	17.161
Variance	1.484	1.485
Observations	593	593
Hypothesis H_0	$\mu_1 = \mu_2$	
Hypothesis H_1	$\mu_1 \neq \mu_2$	
Significance Level α	0.05	
t Statistic	-0.395	
p Value (2-tail)	0.693	
t Critical (2-tail)	1.964	
Supported Hypothesis	H_0	

The evaluation shows that the response time measurements with AOP are quite accurate and agree with those measured directly in the client software. There is an average difference of 1.35 milliseconds, where the measured time by the AOP aspect is always shorter. We argue that this difference results from the point of measurement in the code. The aspect is woven into the lower-level functions of the REST client library, where the communication via HTTP is implemented. Directly from the exemplary client, only the higher-level library functions are available to track when a response from the service is received. Therefore, the difference occurs through the processing time of the REST

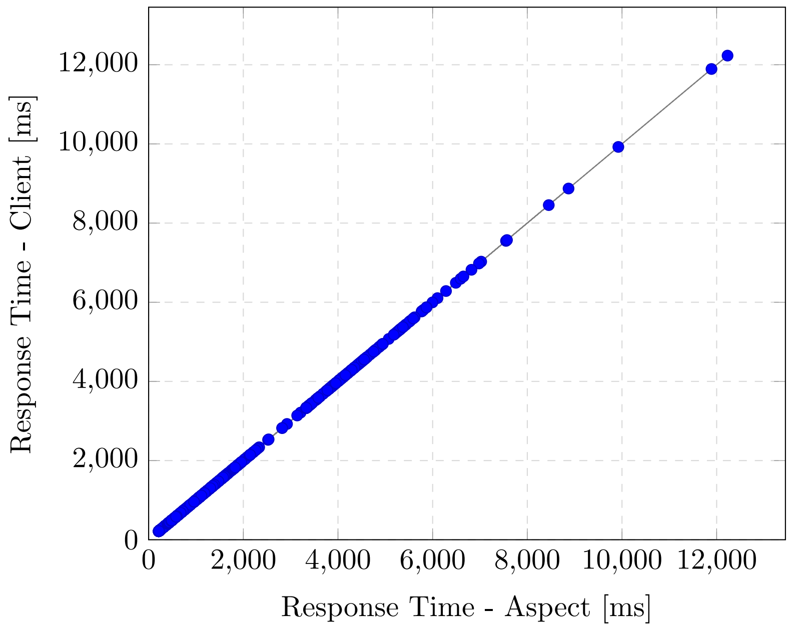


Figure 6.5: Comparison of the Response Time Measurements

client library. Depending on the interpretation of the client latency (which is part of the response time), this additional processing time may be added to the response time or not. However, we argue that the difference is negligible, since it usually accounts for less than 1% of the response time.

6.2.4 Successability

We evaluate the successability metric by deliberately injecting failures into the exemplary service and observing the outcome of the monitoring. Similar to the execution time delay rate in the exemplary test run (see Section 6.1), we introduce a simulation parameter for that purpose:

```
request.fail.rate=0.03
```

This setting causes 3% of all service invocations to fail. Internally, the Web service returns the HTTP status code 500 (Internal Server Error) [RFC99] to signalize a failure. For this scenario, we let the client make 500 successive requests.

The results of the evaluation are presented in Figure 6.6. In order to observe the behavior of the successability metric in relation to the responses, we have plotted both.

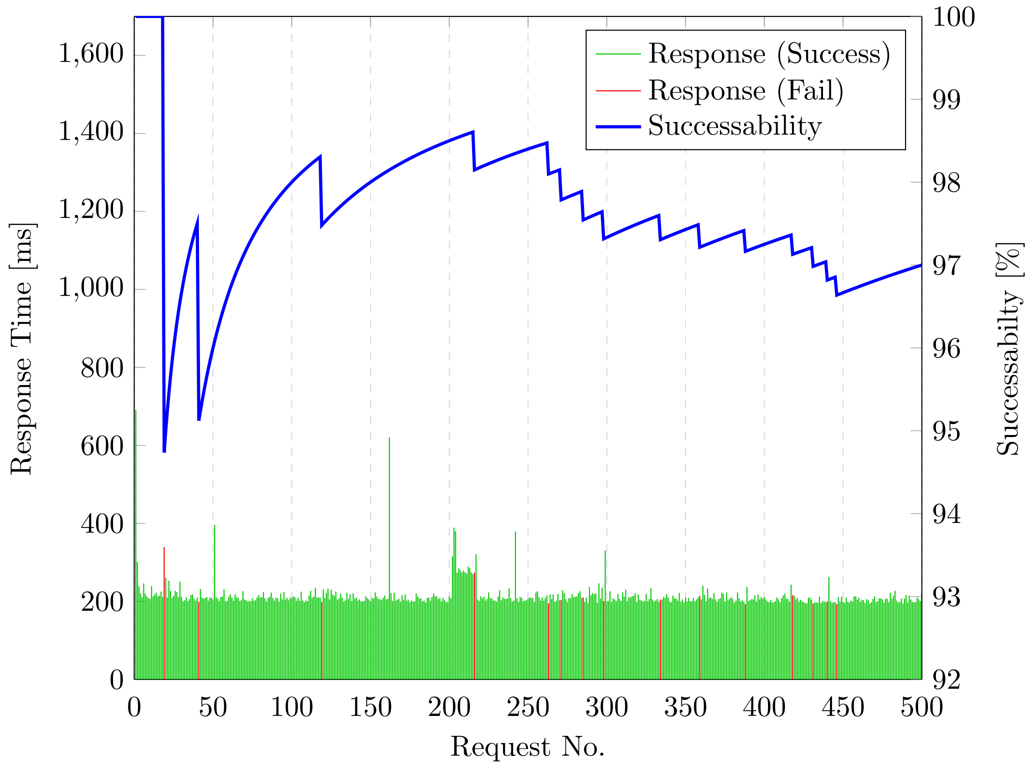


Figure 6.6: Monitored Sucessability

At 3% of 500 requests, we expect 15 errors. We can easily verify that they have been detected correctly. Furthermore, after all service responses have been analyzed by the TTP, the expected successability of 97% is calculated. In Figure 6.6, it can also be seen that there is a high fluctuation of the metric at the beginning. It stabilizes when more and more measurements are considered for the calculation. When negotiating an SLA for this metric, e.g., by specifying a lower bound that must be reached at least, this behavior must be taken into account. For instance, if the very first service invocations fail, the bound may be undercut very fast, although all successive invocations are successful. To this end, it makes sense to collect at least a certain number of response messages before calculating and arbitrating this metric. The CEP statements generated by the TTP consider this minimum quantity for aggregations (see Section 4.3.3).

Furthermore, there is also the need for an expiration time. Let us take into account the following scenario: On most days, the successability of a given service is 100%, while one day all service invocations fail. The metric is already stabilized in a way so that this one day only slightly lowers it, if at all. This clearly penalizes the service consumer. An additional time window that is specified together with the successability bound would solve this problem. For the calculation of the successability metric, the TTP then only considers measurements within this time window.

6.2.5 Throughput

Generally, the throughput metric describes the number of requests successfully processed by the service in a given time. In our specific case, this is the number of images processed by the exemplary image service. For the evaluation of this metric, we run another simulation, where we inject execution times into the service which are longer than usual. We expect the throughput to deteriorate significantly, since the injections block further requests from processing for a while.

The following simulation parameters are used in the experiment:

```
execution.time.delay.rate=0.01
execution.time.delay.time=2000
execution.time.delay.variation=0
```

The effects of the parameters have already been discussed in Section 6.1. The calculated throughput is logged by the TTP every second, thus, for a noticeable difference in the result, we set a delay of two seconds for the injections. We configure the client to execute 1000 service invocations with the small image. Results of the simulation are presented in Figure 6.7.

Apart from the throughput, the response time measurements are also displayed in Figure 6.7. The relationship between the two is thereby clarified. With a constant good response time, the throughput varies between 4 and 5 requests per second (e.g., from 0 to 20 seconds). With an average response time of about 220 milliseconds, resulting from the processing of the small image (see Section 6.1), we verify the correctness of

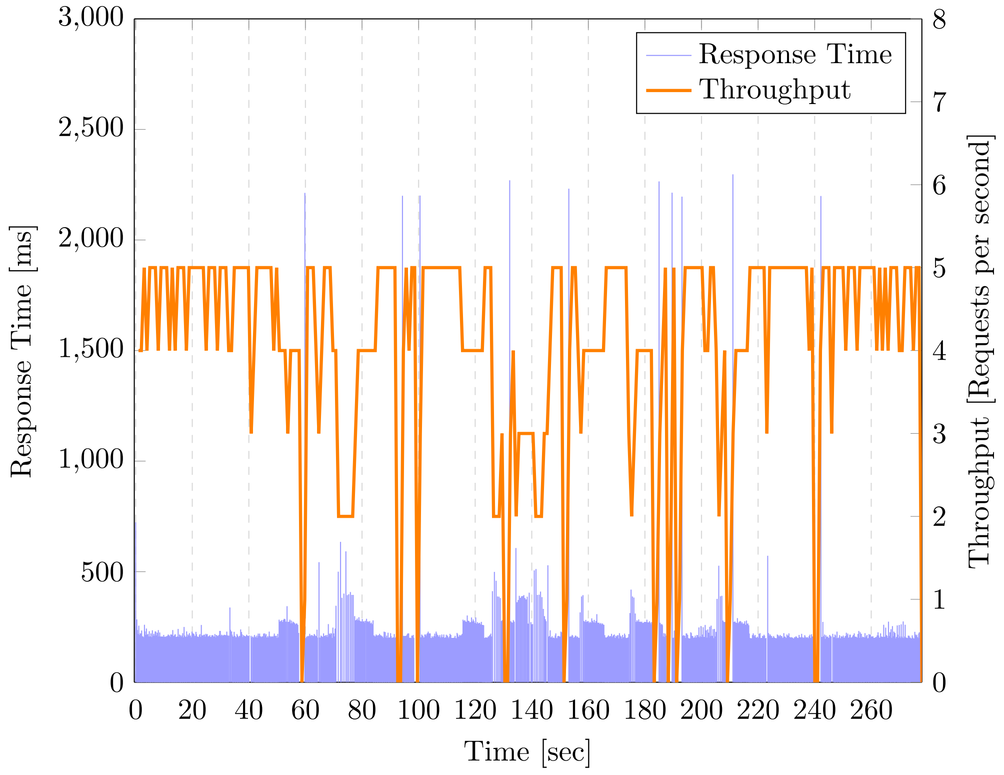


Figure 6.7: Monitored Throughput

this throughput measurement. The drops in the throughput caused by the injections are also clearly recognizable. We count 10 drops; this is consistent with the number of injections. Other accumulations of above-average response times, possibly due to bad network performance during the experiment, also degrade the throughput (e.g., from 70 to 76 seconds).

6.3 Measurement Interval

We inspect in detail the measurement interval for the hardware resource metrics. Currently, these are the CPU and memory usage measurements to which this applies. The other metrics implemented in our framework do not depend on any monitoring interval, but their parameters are measured or extracted at service invocations.

6.3.1 Agent Resource Consumption

We start by comparing the resource consumption of the agent with different measurement intervals. This is important because a high consumption of computational power caused by the agent may lead to additional costs for the cloud service customer, especially when a pay-per-use model is used.

For the investigation, we run a complete monitoring scenario four times under the same conditions, where we only adjust the measurement interval of the agent. The tested intervals are 0.5, 1, 2 and 5 seconds. As discussed in Section 4.3, the agent also retrieves measurements from the AOP advice for each service invocation. We do not want to neglect this during the experiments. For this reason, we also provide load to the service by instructing the client to make recurring requests.

To monitor the resource consumption of the agent itself, i.e., CPU and memory usage, we employ another agent instance. We refer to this instance as *uber-agent*. We have shown in Section 6.2.1 and 6.2.2 that the agent provides accurate readings, hence this approach is well suited for our purposes. In our experiments, the *uber-agent* starts and monitors the usual agent, which in turn starts and monitors the service. The results of the four experiments are depicted in Figure 6.8. A further comparison of how the different intervals affect the agent is shown in Table 6.3. Note that the results in Table 6.3 do not consider the first 2 seconds of the measurement. We want to measure the impact of the agent during operation, and therefore do not want to take into account one-time events like the initialization phase of the agent.

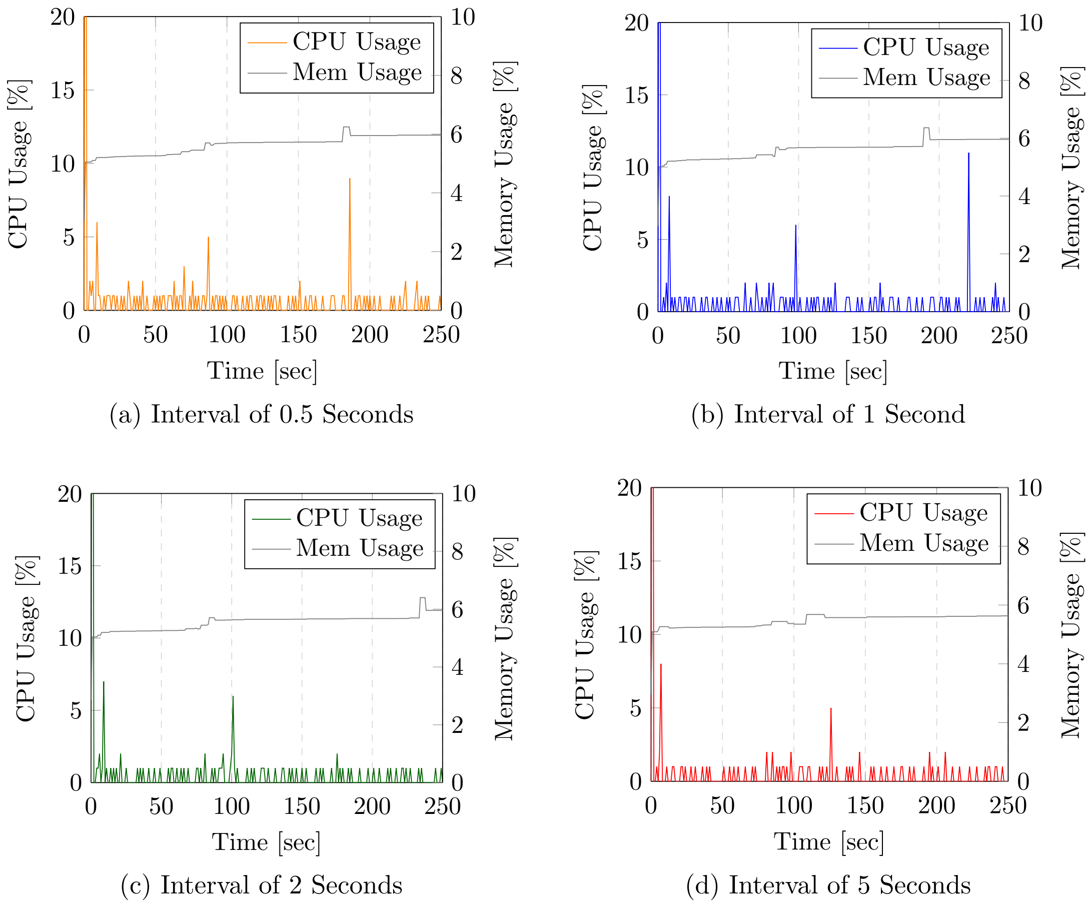
With regard to the resource consumption of the agent, the monitoring interval does not seem to make a noteworthy impact. Nevertheless, it can be seen that the consumption increases with the frequency of the update rate, albeit very little. With a mean CPU usage below 1% and a mean memory usage of 50 to 60 megabytes, we argue that the agent is lightweight enough to run and monitor permanently. Even with a high rate of monitoring, the resource consumption is negligible. Further experiments have shown that in times when the service is not invoked, the agent's consumption is even less, since no messages from the AOP advice must be processed. All in all, the agent satisfies the transparency requirement, which we defined in Section 4.1.3.

Table 6.3: Comparison of the Agent's Resource Consumption (Standard Deviation in Brackets)

Measurement Interval [sec]	Mean CPU Usage [%]	Mean Memory Usage [%]
0.5	0.512 (0.918)	5.632 (0.289)
1	0.460 (1.052)	5.566 (0.264)
2	0.383 (0.760)	5.495 (0.204)
5	0.343 (0.768)	5.442 (0.171)

6.3.2 Amount of Data

Besides the resource consumption on the provider side, we must also consider the amount of data resulting from the permanent measurements of the CPU and memory usage, because every reading must be sent to the TTP via the Internet. It is obvious that the data increases proportionally with the measurement rate, e.g., an interval of 1 second yields twice as much data as an interval of 2 seconds. This behavior must be taken into



(a) Interval of 0.5 Seconds
(b) Interval of 1 Second
(c) Interval of 2 Seconds
(d) Interval of 5 Seconds

Figure 6.8: Resource Usage of the Agent With Different Measurement Intervals

account when setting the update rate of the agent. Generally speaking, the measurement interval should only be set as precise as necessary, depending on the negotiated SLA. There are cases where the CPU usage is considered over a long period of time only. For example, AWS provides instance types where the consumer is rewarded with credits every hour if the CPU usage was below a specified baseline[3]. In such cases, a relatively long monitoring interval may be sufficient to observe this baseline. In contrast, if the detection of high but very short CPU usage peaks is required, even an interval of 1 second is not sufficient (shown in Section 6.2.1). Consequently, a much shorter one must be set, e.g., 0.5 seconds or less.

To further overcome the problem with high amounts of data, different intervals may be defined for each metric individually. We have seen that the memory usage does not change as fast as the CPU usage (see Section 6.2.2), hence a longer interval for the

[3] AWS T2 Instance - User Guide; `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html`.

memory monitoring can be set. We have also implemented a method in the agent that filters successive identical values. As a result, only changes in the readings must be propagated over the Internet. We note that other mechanisms to reduce the amount of data are possible, but are outside of the scope of this work.

6.4 Monitoring Intrusion

Related to the transparency requirement (see Section 4.1.3), we further examine the intrusion of our monitoring framework, i.e., its impact on the monitored service. In Section 6.3.1, we have already shown that the agent is lightweight in terms of resource consumption, thus, the agent is barely intrusive. Now we inspect the performance of the exemplary service with regard to the response time – both with and without the monitoring on the server side. For the acceptance of the monitoring system, it is important that the responsiveness of the service is not degraded considerably when enabling the monitoring.

The experiment consists of two test runs. During the first run, the monitoring components on the server side, i.e., the AOP library and the agent, are enabled. During the second run, these components stay disabled, therefore, only the Web service is active. In each trial, 1000 requests of each workload image are performed by the client, which also measures the service's response time.

When running the described experiment in the cloud, we experience strong fluctuations in the response time, although the execution times are inconspicuous. We test the results of two independent test runs with the *same* configuration using a t-test. It reveals that there is a significant difference between the two population means (see Table 6.4), even though the opposite should be the case. We assume that this is due to bad network/Internet performance or other factors we can not influence. This behavior makes the intrusion test in the cloud inconclusive, which is why we perform the test under a controlled environment, i.e., on our local PC.

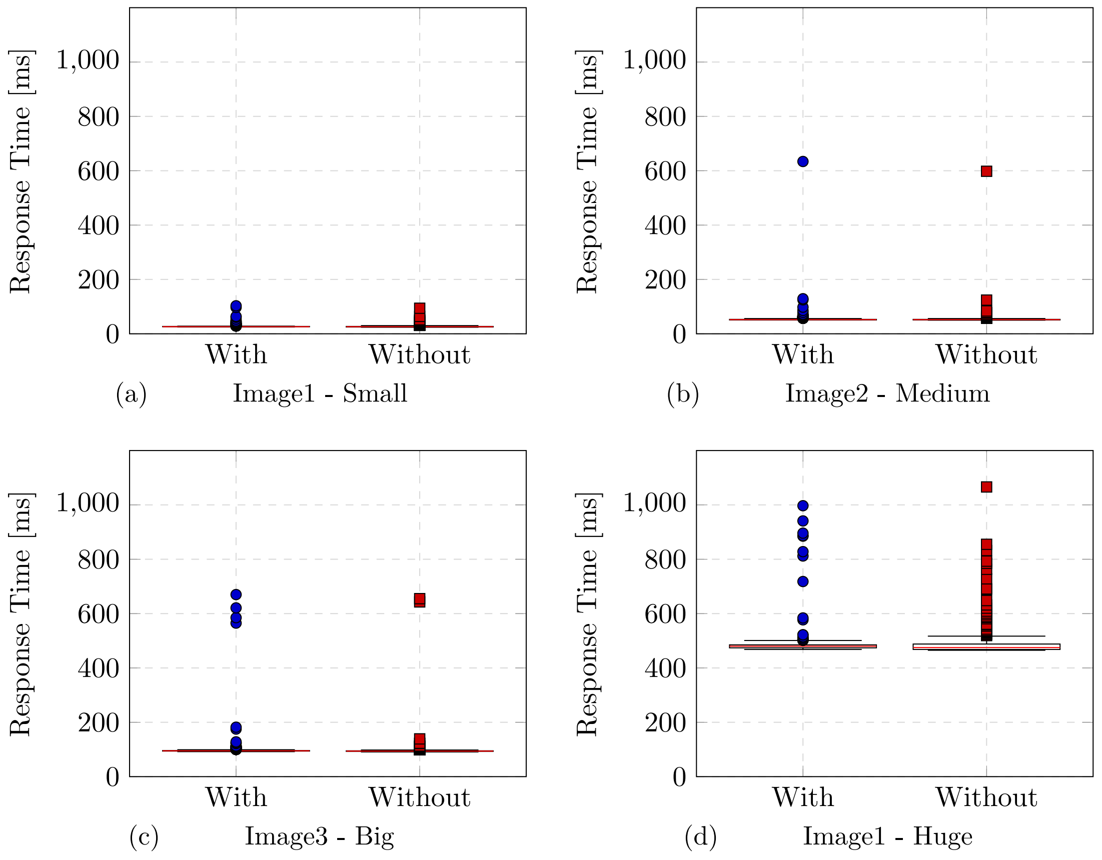
Table 6.4: t-Test for Two Independent Runs With the Same Configuration

	Run 1	Run 2
Mean	252.471	229.068
Variance	7415.889	2781.139
Observations	2000	2000
Hypothesis H_0	$\mu_1 = \mu_2$	
Hypothesis H_1	$\mu_1 \neq \mu_2$	
Significance Level α	0.05	
t Statistic	10.365	
p Value (2-tail)	8.566E-25	
t Critical (2-tail)	1.961	
Accepted Hypothesis	H_1	

The results of the experiments on the local PC are shown in Figure 6.9 and in Table 6.5. Since no communication via the Internet is necessary, the response times of the service are much lower than in the cloud, however, the intrusiveness can still be evaluated. It can be seen that the monitoring adds overhead to the service when comparing the results without the monitoring. Nevertheless, the overhead is a maximum of 2% and within the acceptable range of 5%, which we have stated in the transparency requirement (see Section 4.1.3).

Table 6.5: Comparison of the Response Times
(Standard Deviation in Brackets)

Workload Image	Mean Response Time *With* Service Monitoring [ms]	Mean Response Time *Without* Service Monitoring [ms]
Image1 - Small	27.054 (4.375)	26.524 (3.524)
Image2 - Medium	53.537 (18.943)	53.383 (17.664)
Image3 - Big	97.540 (32.987)	95.732 (25.026)
Image4 - Huge	483.707 (36.481)	482.326 (36.443)



(a) Image1 - Small
(b) Image2 - Medium
(c) Image3 - Big
(d) Image1 - Huge

Figure 6.9: Response Times With/Without Service Monitoring

We also note that with the size of the image, the number of outliers increases. Different to the outliers experienced in the exemplary test run (see Section 6.1), these outliers also occur in the execution time measurements (we take those measurements from the test with service monitoring). We assume that the operating system can not catch up with the resource provisioning for the service.

6.5 Evaluation of Trustworthiness

One of the major aspects we consider in our work is the ensuring of trustworthiness in monitoring. The monitoring process includes the measurement *and* the evaluation of SLA parameters, thus, it is important that both of these areas grant trustworthiness to a certain extent. To indicate this necessity, we provide the following scenario: If the measurements are indeed trusted by the consumer and the provider, but the arbitration is done at one of them, the opposing party will have a reason to distrust these decisions. In contrast, also a trustworthy arbitration alone does not solve the problem if the readings cannot be trusted.

Furthermore, it must be taken into account that our approach requires the sending of measurements to the TTP, hence providing trustworthiness in the communication must not be excluded in the evaluation. In the following, we discuss how our considerations contribute to trustworthiness in each of these areas.

6.5.1 Trustworthiness in Measurement

An important aspect in the measurement of SLA parameters is the correctness and the accuracy of those. Consumer and provider must be able to rely on the monitoring system that the readings are correct and that those do not lead to unjustified disadvantages for themselves. In our work, we devoted a whole section where we have shown that the readings are appropriately accurate (see Section 6.2). It can also be seen that all measurements are objectively performed and that no party is preferred. Our entire monitoring system is open source, thus, the users can additionally ensure that the objectivity of the measure methods is given.

From the breakdown of SLA metrics (see Section 3.3), we have seen that different interpretations of the metrics exist (e.g., throughput can be evaluated for a single client vs. throughput is evaluated for an entire service provider), and that different representations of the readings exist (e.g., CPU usage can be divided by the number of CPUs, or not). Due to these discrepancies, if each party developed its own measurement mechanism, it might be difficult to compare the readings. For instance, the consumer might state that the throughput of a service was only 2 requests per second, where the provider might argue to have measured a throughput of 5 requests per second (because the provider calculates the metric differently). By using our monitoring system, such misconceptions are avoided. The entire system comes a from a single source, hence there is the same interpretation/representation for measurements on both the consumer and the provider side. Of course, this also applies to the TTP as it further processes the readings in our approach.

We have established a decentralized measurement, i.e., if possible, a metric is measured on the service consumer side and on the cloud provider side. This approach creates further confidence in the readings since the TTP can confirm their correctness later by counter-checking the values from both parties. Any major deviation would be noticed and can be investigated. Even with the response time metric, where the readings differ depending on the point of measurement, informative comparisons can still be made. In our work, we have compared the response times of a service with the corresponding execution times. Those are closely related, because the execution times are reflected in the response times.

The agent component of our framework monitors metrics that can only be read on the provider side, namely the CPU and memory usage. Our proposed idea to digitally sign the executable of the agent allows the consumer or the TTP to verify the integrity of the application. Therefore, it can be ensured at any time that the cloud provider has not modified the agent to his advantage, hence these readings can be trusted too. The code signing mechanism may be applied to other monitoring components as well to increase the trust.

6.5.2 Trustworthiness in Communication

Measurements must be transfered via the Internet to the TTP, both from the consumer and from the provider, therefore, it is important that the communication is trusted. In our approach, we have enabled a reliable communication through the use of MOM. A MOM broker employed by the TTP takes care of the (reliable) delivery of all messages. Since the broker is also in the domain of the TTP, it can be trusted too. Single points of failures in the communication may be eliminated by a clustering of brokers. In the experiments, we were able to verify that the TTP has always received all the measurements from the client or the agent by comparing the logs of all components. If this were not the case, it would also be difficult for the signature parties to trust the arbitration, because important readings may have been lost.

To further increase the trust, we have secured the communication by attaching a MAC to the messages. This allows the TTP to verify the integrity and the authentication of the measurements. If the readings were tampered with during the transmission, then the TTP would notice as soon as they are received.

6.5.3 Trustworthiness in SLA Evaluation

Following the measurement and communication phase, received parameters are used to evaluate SLAs. We have introduced a TTP for this purpose. This neutral authority completely takes over the task of arbitration, which offers a solution to solve the conflict of interest between consumer and provider. Measurements are evaluated objectively and independently of the interests. Again, the objectivity can be verified by the inspection of the source code and the utilization of code signing.

Many SLA parameters are only calculated at the TTP. This especially applies to aggregated metrics. As a result, only the absolutely necessary amount of measurement work must be done at the signature parties. The remaining part is outsourced to the TTP, which is trusted by both.

Another problem concerns the burden of proof, i.e., how can credible evidence of a violation be provided? Especially if only the signature parties are involved in the monitoring, this may be a difficult task. In our approach, this also lies within the responsibility of the TTP. With every detected violation, information about the circumstances is stored into a database. As a result, a clear proof for every violation is given that can be requested at any time by the consumer and provider.

CHAPTER 7
Conclusion

This chapter provides a summary of the work presented within this thesis. Furthermore, we give a brief outlook on possible fields for future work.

7.1 Summary

Cloud computing is an increasingly popular strategy for outsourcing IT resources. Amongst other benefits, more control and flexibility are granted to both the cloud providers and the customers. However, unlike with traditional outsourcing, the rented resources are shared between several customers, thus, the necessity to use SLAs arises. SLAs allow to stipulate boundaries on quality aspects that provided services must fulfill.

In this thesis, we have addressed the problem of trustworthiness in SLA measurement and arbitration. We have conducted a literature research which revealed that many works have investigated the monitoring of SLAs, however, only a few have considered trustworthiness. Regarding these approaches, certain shortcomings have been identified, so that an adaption of a given approach would not have met our criteria. Hence, we have designed and implemented a novel solution for a trustworthy monitoring of SLAs.

We have proposed a framework consisting of the core components TTP, agent and AOP libraries. The agent mainly measures application-level resource metrics on the cloud provider, e.g., CPU or memory usage of a service. Measurements of these metrics depended on an interval, as they change over time. Further measurements are done by the AOP libraries. They are woven into the service and into the requesting client at runtime. This has the advantage that no intrusion into existing software is necessary. Information that is required to calculate QoS metrics such as response time, throughput or successability is extracted by the AOP code at service invocations. The measurement components propagate their readings to the TTP. The TTP is trusted by the provider and the customer and is responsible for the trustworthy arbitration of SLAs. At this component, received measurements are further processed and checked for compliance with the agreement.

Subsequently, we have implemented a prototype of the proposed monitoring framework. During the development, we have put emphasis on transparency and platform independence. For the definition of SLAs, we have adhered to the WSLA standard. Furthermore, we have provided the agent with an open interface, thus, developers are free to add additional monitoring probes to our framework.

To evaluate our solution in a real-world cloud environment, we have developed a fully functional testbed containing an exemplary cloud service and a consumer. The implemented service offers the functionality to process images. For the experiments, we have deployed the service on a public cloud server. Another cloud server has been used for the operation of the TTP component.

Several experiments have been conducted in the testbed. First, we have presented an exemplary scenario, where we investigated the detection of SLA violations. We have shown that the TTP is capable of identifying non-compliances with agreed SLA parameters. Measurements from the consumer as well as from the provider side can be considered for this purpose. We have presented this on the basis of the response time and execution time metrics.

For each metric that is measured by our mechanism, we have provided an experiment to evaluate the correctness and accuracy of the readings. This has shown that both agent-based monitoring and AOP-based monitoring are appropriate tools for the measurement, while the accuracy is sufficient for most applications. We also have investigated the impact of the monitoring interval of the agent. Results have demonstrated that the agent is lightweight enough to monitor frequently, however, it is advisable to set the interval only as short as necessary. Otherwise, there arises a considerable amount of data that has to be transmitted over the Internet to the TTP.

We have then compared the service's response times with and without enabled monitoring. This has shown that using our monitoring mechanism adds a slight overhead to the service, however, the response times only increase by a maximum of 2%. We have argued that this overhead is acceptable for the most use cases.

Finally, we have discussed how all the measures we have realized with our approach help to ensure trustworthiness for both the provider and the customer. To this end, we have analyzed the trustworthiness in the following three areas: measurement, communication and SLA evaluation. One of the most important aspects is the establishment of measurements on the provider and on the consumer side. In case of a discrepancy between those readings, a further examination can be done immediately. Moreover, by outsourcing the entire SLA arbitrations to the TTP, it can be ensured that provider and customer trust its decisions.

7.2 Future Work

We have presented a first prototypical implementation of our framework which enables trustworthy monitoring of cloud services. Although the prototype is fully operational in its current state, we have identified several aspects for an improvement, as discussed in the following:

- In our work, we have covered the monitoring and arbitration stage of SLAs. Other stages to complete the WSLA lifecycle, as described in the WSLA standard, can be added in the future to complement our implementation. Currently, the negotiation or deployment of SLAs requires external tools that are not integrated within our architecture. This also concerns the enforcement of penalties after a violation has been detected by the TTP. Furthermore, the support of other SLA standards such as WS-Agreement is considered a possible addition.
- A larger-scale evaluation of our monitoring framework is still required to analyze its behavior in the real world, i.e., considering multiple independent services and consumers. This scenario affects the TTP in particular, because it then has to manage and monitor multiple SLAs simultaneously.
- One drawback of the TTP approach is the resulting amount of data that must be sent over the Internet. Even though some ideas to reduce this amount, such as filtering successive identical values, have already been implemented in our software, it is desirable to decrease it even further. Especially when dealing with a slow Internet connection, this improvement would help to keep the TTP monitoring timely.
- The current implementation focuses on the monitoring of application-level metrics. A possible domain of future work is to enable the measurement and arbitration of system or VM-level metrics. The main value of this feature is that the quality of the other provided cloud layers can be observed as well.
- With the realized AOP libraries, we provide the possibility to extract required parameters from the client and the service in a non-invasive manner. In the current form, Java applications, which utilize the reference implementation of the JAX-RS specification, are supported. One possible goal that could be pursued in the future would be to support more frameworks and programming languages for our AOP-based monitoring approach.

APPENDIX A

Reproduction of Experiments

In this appendix, we describe the steps necessary to repeat the simulations which we have shown in Chapter 6.

The source code of our monitoring framework is accessible at `https://github.com/christianschubert/cloudmonitoring`. The provided GitHub repository also contains the workload images, the simulation configuration and several exemplary WSLA files. In the experiments, we used version 0.3 of the project with the git commit hash `deb05ea4729c3fbcd7022a374bb70a509c9f501a`.

A.1 Test Environment Setup

To recreate the test environment, two cloud servers (IaaS) and a computer with Internet connection are required. There are several options for the operating system. We tested the monitoring framework on Ubuntu Server 14.04 LTS and on Microsoft Windows 10, however, other operating systems should work as well[1].

In order to establish the communication between the three parties, several TCP ports must be opened on the two cloud servers for incoming requests. SSH is also required for the setup and startup of the applications. All necessary ports are shown in Table A.1. Every machine requires the Java Development Kit[2] (Version 8) and an Apache Maven[3] installation (Version 3.*) to compile and run our software. To clone the git repository, a git installation is helpful, but not required. There is also the possibility to manually download the project from GitHub.

[1]The monitoring framework is entirely written in Java, hence platform independence is given. Prerequisite for this is the availability of the Java 8 runtime. The agent also requires a native Sigar library matching the operating system.

[2]Java SE Development Kit 8; `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`.

[3]Apache Maven; `https://maven.apache.org/download.cgi`.

Table A.1: Required TCP Ports for the Monitoring System

Machine	Port	Description
Provider Server	22	SSH
	8080	Image Processing REST API
TTP Server	22	SSH
	8090	Violation REST API
	61616	JMS broker

When using Ubuntu Server 14.04 LTS, every required software can be installed with the following command (on cloud servers, the commands can be executed via SSH):

```
~ $ sudo apt-get update &&
    sudo apt-get install default-jre default-jdk maven git
```

The git repository of our monitoring framework is cloned with the `git clone` command:

```
~ $ git clone https://github.com/christianschubert/cloudmonitoring.git
```

Once the cloning is done, the complete project can be built with Apache Maven (Note that this may take a while until the project dependencies are downloaded):

```
~ $ cd cloudmonitoring
~/cloudmonitoring $ mvn install
```

If everything succeeded, the following output is shown:

```
......
[INFO] ------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] cloudmonitoring ................ SUCCESS [  0.164 s]
[INFO] common ......................... SUCCESS [  0.610 s]
[INFO] service_aspect ................. SUCCESS [  0.066 s]
[INFO] monitoring_service ............. SUCCESS [  2.117 s]
[INFO] monitoring_agent ............... SUCCESS [  2.977 s]
[INFO] monitoring_server .............. SUCCESS [  2.452 s]
[INFO] client_aspect .................. SUCCESS [  0.051 s]
[INFO] monitoring_client .............. SUCCESS [  3.073 s]
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------
[INFO] Total time: 11.642 s
[INFO] Finished at: 2017-11-13T17:57:37+01:00
[INFO] Final Memory: 94M/741M
[INFO] ------------------------------------------------------
```

A.2 Conducting Experiments

Before executing simulations, several parameters for the test must be set. The configuration is defined in the file `etc/settings.properties` (relative to the project directory). We have already described most of the parameters in Chapter 6, however, it is also necessary to specify the location of the message broker and the service. Otherwise, there can be no communication between the parties. Note that the message broker is embedded in the TTP server software and is started together with it.

The URLs must be provided in the following format, in which `TTP_SERVER` and `PROVIDER_SERVER` are the respective cloud servers:

```
# url of ttp message broker
broker.url=tcp://TTP_SERVER:61616

# url of exemplary service
service.url=http://PROVIDER_SERVER:8080
```

The WSLA file specified in the configuration is searched for in the `etc/agreements` folder. For the exemplary test run shown in Section 6.1, we have used the `response_time_agreement.xml`. Other parameters not mentioned in this work are explained in the file and allow further control over the software. After a completed simulation, the CSV log files can be found in the `etc/logs` directory.

In a common simulation scenario, the TTP is started first, then the agent (who starts the service), and finally the client. For some components, there are several ways to execute them, allowing for different configurations. In the following, we explain all possible execution targets for each component.

TTP

The TTP server software can be started with the following command:

```
~/cloudmonitoring $ mvn exec:exec -pl monitoring_server
```

The settings file is automatically passed to the application via command line parameters. In it, the JMS broker and the violation database are embedded.

Agent

The agent requires the message broker (and therefore the TTP) to be active. When starting with the following command, the agent also starts and monitors the exemplary image processing service:

```
~/cloudmonitoring $ mvn exec:exec -pl monitoring_agent
```

Again, the configuration is automatically provided.

In Section 6.3, we discussed the ability of the agent to monitor another instance of itself. This can be accomplished with the following execution target:

```
~/cloudmonitoring $ mvn exec:exec@uberagent -pl monitoring_agent
```

The *uber-agent* starts and monitors the usual agent, who in turn starts and monitors the image processing service. No other configuration is required. The logs of the *uber-agent* are also written in the default log directory. They have appended the string *_uber* in the filename.

Client

The client can be executed as soon as the TTP and the service are up and running. When invoking this command, the AOP library is woven into the client at runtime:

```
~/cloudmonitoring $ mvn exec:exec -pl monitoring_client
```

As with the agent, the configuration is provided. To test the client without the AOP monitoring, the `@noaspect` can be specified:

```
~/cloudmonitoring $ mvn exec:exec@noaspect -pl monitoring_client
```

Service

The service may also be executed without the monitoring of the agent. Similar to the client, it can be started with or without the AOP library. However, if no agent is running to collect the measurements from the AOP advice, there is also no sense in using the library. This leaves us the following command:

```
~/cloudmonitoring $ mvn exec:exec@noaspect -pl monitoring_service
```

Integration Test

We also provide an integration test that can be run locally on a single computer. Of course, since no real network/Internet communication is involved, results different to the ones shown in our work are expected. However, it is a good start to test the basic functionality of the monitoring mechanism. This command starts up all components and then executes the simulations defined in the `etc/settings.properties` file:

```
~/cloudmonitoring $ mvn exec:exec@integration -pl monitoring_client
```

APPENDIX B

Exemplary Test Run: WSLA

Listing B.1: WSLA Document with the Response Time Parameter

```xml
<?xml version="1.0"?>
<SLA xmlns="http://www.ibm.com/wsla"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/wsla wsla.xsd"
      name="ResponseTimeAgreement">

  <!-- Definition of the involved parties, the signatory parties
      as well as the supporting ones -->
  <Parties>
    <ServiceProvider name="ImageServiceProvider">
      <Contact>
        <Street>PO BOX 218</Street>
        <City>Yorktown, NY 10598, USA</City>
      </Contact>
    </ServiceProvider>
    <ServiceConsumer name="ImageServiceConsumer">
      <Contact>
        <Street>19 Skyline Drive</Street>
        <City>Hawthorne, NY 10532, USA</City>
      </Contact>
    </ServiceConsumer>
    <SupportingParty name="Trusted3rdParty">
      <Contact>
        <Street>Hursley Park</Street>
        <City>Winchester, England, SO21 2JN</City>
      </Contact>
      <Sponsor>ImageServiceProvider</Sponsor>
      <Sponsor>ImageServiceConsumer</Sponsor>
      <Role>ConditionEvaluationService</Role>
    </SupportingParty>
  </Parties>

```

```xml
    <!-- The definition of the service in terms of the service
       parameters and their measurement. -->
    <ServiceDefinition>
      <Operation name="ShrinkImage"
          xsi:type="WSDLSOAPOperationDescriptionType">
        <SLAParameter name="ResponseTimeService" type="long"
          unit="milliseconds">
          <Metric>ResponseTime</Metric>
          <Communication>
            <Source>ImageServiceConsumer</Source>
          </Communication>
        </SLAParameter>

        <Metric name="ResponseTime" type="long" unit="milliseconds">
          <Source>ImageServiceConsumer</Source>
          <MeasurementDirectiveVariable>
          </MeasurementDirectiveVariable>
        </Metric>

        <WSDLFile>
          http://provider:8080/imageprocessor/application.wadl
        </WSDLFile>
        <SOAPBindingName>
          http://provider:8080/imageprocessor/shrink
        </SOAPBindingName>
        <SOAPOperationName>POST</SOAPOperationName>
      </Operation>
    </ServiceDefinition>

    <!-- The obligations of the parties, referring to parameters
       defined above. -->
    <Obligations>
      <ServiceLevelObjective name="ResponseTimeSLO">
        <Obliged>ImageServiceProvider</Obliged>
        <Validity>
          <Start>2017-01-01T14:00:00.000-05:00</Start>
          <End>2099-01-01T14:00:00.000-05:00</End>
        </Validity>
        <Expression>
          <Predicate xsi:type="LessEqual">
            <SLAParameter>ResponseTimeService</SLAParameter>
            <Value>1000</Value>
          </Predicate>
        </Expression>
        <EvaluationEvent>NewValue</EvaluationEvent>
      </ServiceLevelObjective>
    </Obligations>
  </SLA>
```

List of Figures

List of Tables

Acronyms

AOP Aspect-Oriented Programming. 4, 9, 10, 14, 15, 22, 23, 31, 34, 36–39, 41, 46, 47, 51, 61, 72, 73, 77, 79, 85–87, 92, 95

API Application Programming Interface. 11, 14, 41, 43, 47, 49, 51, 53, 59, 60, 90

AWS Amazon Web Services. 13, 14, 16, 61, 78

CEP Complex Event Processing. 4, 11, 21, 34, 35, 37, 38, 53–57, 75

EPL Event Processing Language. 55–57

HMAC Keyed-Hash Message Authentication Code. 45

HTTP Hypertext Transfer Protocol. 13, 20, 48–52, 59, 60, 62, 73, 74

IaaS Infrastructure as a Service. 2, 3, 61, 89

JIT Just-in-Time. 69

JMS Java Message Service. 43, 45, 51, 52, 90, 91

jps Java Virtual Machine Process Status Tool. 41

JVM Java Virtual Machine. 22, 47, 48

MAC Message Authentication Code. 39, 45, 51, 57, 82

MOM Message-Oriented Middleware. 19, 35, 36, 39, 43, 45, 82

OOP Object-Oriented Programming. 22

PaaS Platform as a Service. 2

PID Process Identifier. 40–42, 69

PPID Parent Process Identifier. 42

QoE Quality of Experience. 13, 16, 19

QoS Quality of Service. 2, 4, 9, 10, 15, 19, 23, 26, 27, 29, 32, 36, 85

REST Representational State Transfer. 19, 47, 57, 59, 73, 90

SaaS Software as a Service. 1, 2

SLA Service Level Agreement. 2–5, 7–9, 12–17, 19–21, 23–25, 27–29, 31, 32, 34–37, 43, 46, 52, 54–57, 62, 65, 67, 68, 71, 75, 78, 81–83, 85–87, 96

SOA Service-Oriented Architecture. 2, 7–9, 14, 15

TTP Trusted Third Party. 13, 27, 31, 34–39, 41, 43, 45–47, 50–54, 56, 57, 61, 62, 65–68, 75, 77, 81–83, 85–87, 90–92

URI Uniform Resource Identifier. 20, 52, 62

VM Virtual Machine. 11–13, 69, 87

WSLA Web Service Level Agreement. 20, 21, 25, 27, 32, 52–56, 62, 66, 67, 86, 87, 89, 91

Bibliography

[AF08] D. Ameller and X. Franch. Service Level Agreement Monitor (SALMon). In *7th International Conference on Composition-Based Software Systems (ICCBSS)*, pages 224–227, 2008.

[ASAY15] S. Al-Shammari and A. Al-Yasiri. MonSLAR: A Middleware for Monitoring SLA for RESTFUL Services in Cloud Computing. In *9th IEEE International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA)*, pages 46–50. IEEE, 2015.

[Asp03] AspectJ. The AspectJ Programming Guide, 2003. `https://eclipse.org/aspectj/doc/released/progguide/index.html`. Accessed: 2017-12-29.

[Asp05] AspectJ. The AspectJ Development Environment Guide, 2005. `https://eclipse.org/aspectj/doc/next/devguide/index.html`. Accessed: 2017-12-29.

[Bas12] S. A. Baset. Cloud SLAs: Present and Future. *ACM SIGOPS Operating Systems Review*, 46(2):57–66, 2012.

[BBF$^+$04] B. Baliś, M. Bubak, W. Funika, T. Szepieniec, R. Wismüller, and M. Radecki. Monitoring Grid Applications with Grid-Enabled OMIS Monitor. In *1st European Across Grids Conference (EAGC)*, pages 230–239. Springer, Berlin, Heidelberg, 2004.

[BD15] R. Bruns and J. Dunkel. *Complex Event Processing: Komplexe Analyse von massiven Datenströmen mit CEP*. Springer, Wiesbaden, 2015.

[BH10] Z. Balfagih and M. F. B. Hassan. Agent Based Monitoring Framework for SOA Applications Quality. In *International Symposium on Information Technology (ITSim)*, volume 3, pages 1124–1129, 2010.

[BMLD09] I. Brandic, D. Music, P. Leitner, and S. Dustdar. VieSLAF Framework: Enabling Adaptive and Versatile SLA-Management. In *6th International Workshop on Grid Economics and Business Models (GECON)*, pages 60–73. Springer, Berlin, Heidelberg, 2009.

[BYV+09] 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. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[CLX09] B. Cao, B. Li, and Q. Xiag. A Service-Oriented Qos-Assured and Multi-Agent Cloud Computing Architecture. In *1st International Conference on Cloud Computing (CloudCom)*, pages 644–649. Springer, Berlin, Heidelberg, 2009.

[Cor13] Oracle Corporation. JSR-000339 JAX-RS 2.0 Specification, 2013. `http://download.oracle.com/otndocs/jcp/jaxrs-2_0-fr-eval-spec/index.html`. Accessed: 2017-12-29.

[DDK+04] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web Services on Demand: WSLA-Driven Automated Management. *IBM Systems Journal*, 43(1):136–158, 2004.

[EFN+12] V. C. Emeakaroha, T. C. Ferreto, M. A. S. Netto, I. Brandic, and C. A. F. De Rose. CASViD: Application Level Monitoring for SLA Violation Detection in Clouds. In *IEEE 36th Annual Computer Software and Applications Conference (COMPSAC)*, pages 499–508. IEEE, 2012.

[Fen94] N. Fenton. Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, 1994.

[FHTG10] M. Fiedler, T. Hossfeld, and P. Tran-Gia. A Generic Quantitative Relationship Between Quality of Experience and Quality of Service. *IEEE Network*, 24(2):36–41, 2010.

[For07] Open Grid Forum. Web Services Agreement Specification (WS-Agreement), 2004-2007. `https://www.ogf.org/documents/GFD.107.pdf`. Accessed: 2017-12-29.

[GST03] S. Goel, H. Sharda, and D. Taniar. Message-Oriented-Middleware in a Distributed Environment. In *International Workshop on Innovative Internet Community Systems (IICS)*, pages 93–103. Springer, Berlin, Heidelberg, 2003.

[IBM02] IBM. Understanding Quality of Service for Web Services, 2002. `https://www.ibm.com/developerworks/library/ws-quality/index.html`. Accessed: 2017-12-29.

[KL03] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, Berlin, Heidelberg, 1997.

[KNJ15] S. K. Khatri, P. Narayan, and P. Johri. Weaving Techniques and Its Impact on Execution of Classes in Aspect Oriented Programming. In *4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*, pages 1–6, 2015.

[LEM16] H. Ladour and A. El-Maouhab. Monitoring "Grid-Cloud Model" Using Complex Event Processing (CEP). In *2nd International Conference on Open Source Software Computing (OSSCOM)*, pages 1–9, 2016.

[LIH+12] 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 *5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8. IEEE, 2012.

[Lin17a] Linux. Linux Manual Page: ps - Report Process Status, 2017. `http://man7.org/linux/man-pages/man1/ps.1.html`. Accessed: 2017-12-29.

[Lin17b] Linux. Linux Manual Page: top - Display Linux processes, 2017. `http://man7.org/linux/man-pages/man1/top.1.html`. Accessed: 2017-12-29.

[LKD+03] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck. Web Service Level Agreement (WSLA) Language Specification, v1.0, 2003. `http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf` Accessed: 2017-12-29.

[Luc01] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[MA15] G. J. Mirobi and L. Arockiam. Service Level Agreement in Cloud Computing: An Overview. In *International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pages 753–758, 2015.

[MEMB12] T. Mastelic, V. C. Emeakaroha, M. Maurer, and I. Brandic. M4Cloud - Generic Application Level Monitoring for Resource-shared Cloud Environments. In *2nd International Conference on Cloud Computing and Services Science (CLOSER)*, pages 522–532, 2012.

[MEME15] S. Moustafa, K. Elgazzar, P. Martin, and M. Elsayed. SLAM: SLA Monitoring Framework for Federated Cloud Services. In *IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pages 506–511. IEEE/ACM, 2015.

[MHJ+11] A. Mdhaffar, R. B. Halima, E. Juhnke, M. Jmaiel, and B. Freisleben. AOP4CSM: An Aspect-Oriented Programming Approach for Cloud Service Monitoring. In *IEEE 11th International Conference on Computer and Information Technology (ICCIT)*, pages 363–370. IEEE, 2011.

[MMH15] A. Maarouf, A. Marzouk, and A. Haqiq. Towards a Trusted Third Party Based on Multi-Agent Systems for Automatic Control of the Quality of Service Contract in the Cloud Computing. In *International Conference on Electrical and Information Technologies (ICEIT)*, pages 311–315, 2015.

[MSD17] MSDN. Windows Developer Center: About Processes and Threads, 2017. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms681917(v=vs.85).aspx`. Accessed: 2017-12-29.

[Ope12] OASIS Open. Web Services Quality Factors Version 1.0. Candidate OASIS Standard 01, 2012. `http://docs.oasis-open.org/wsqm/WS-Quality-Factors/v1.0/WS-Quality-Factors-v1.0.html`. Accessed: 2017-12-29.

[PRS09] P. Patel, A. Ranabahu, and A. Sheth. Service Level Agreement in Cloud Computing. In *Cloud Workshops at the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[RFC99] RFC2616. HTTP/1.1 Status Codes, 1999. `https://tools.ietf.org/html/rfc2616#section-6.1.1`. Accessed: 2017-12-29.

[RJM07] T. Ropars, E. Jeanvoine, and C. Morin. GAMoSe: An Accurate Monitoring Service For Grid Applications. In *6th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 40–40, 2007.

[Ser17] Amazon Web Services. Amazon CloudWatch Product Overview, 2017. `https://aws.amazon.com/cloudwatch`. Accessed: 2017-12-29.

[Spr16] Spring.io. Spring Framework Reference Documentation - Aspect Oriented Programming with Spring, 2016. `https://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html`. Accessed: 2017-12-29.

[SWWM10] J. Shao, H. Wei, Q. Wang, and H. Mei. A Runtime Model Based Monitoring Approach for Cloud. In *IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pages 313–320. IEEE, 2010.

[TMR04] S. Tai, T.A. Mikalsen, and I. Rouvellou. Using Message-Oriented Middleware for Reliable Web Services Messaging. In *2nd International Workshop on Web Services, E-Business, and the Semantic Web (WES)*, pages 89–104. Springer, Berlin, Heidelberg, 2004.

[TPD14] D. Trihinas, G. Pallis, and M. D. Dikaiakos. JCatascopia: Monitoring Elastically Adaptive Applications in the Cloud. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 226–235. IEEE/ACM, 2014.