# Scalable and Live Trace Processing in the Cloud

Bachelor's Thesis

Phil Stelzer

April 7, 2014

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Florian Fittkau

**Eidesstattliche Erklärung**

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

_____

# Abstract

Nowadays, Cloud computing is gaining importance in the modern IT industry. The Cloud offers different benefits over traditional deployments. To enable services to make use of the advantages of the Cloud environment they need to be adapted and configured.

This thesis describes the design and deployment of an application-level monitoring solution to a Cloud environment. The resulting architecture provides self-scaling and resource management mechanics to automatically react to different workload demands. To evaluate our approach and test the scaling behavior of the deployed services we run different workload scenarios in the Cloud. The results of our test scenarios show the scaling of services and the reaction to applied workload patterns. Future work includes the further development of our capacity management solution and the refinement of the scaling behavior of the monitoring solution.

# Contents

Chapter 1

# Introduction

Nowadays, Cloud Computing is gaining importance in the modern IT industry. It changes the way we design and operate software. Especially for start-ups and new, innovative software ideas the possibility to deploy software without investing in own, expensive hardware is beneficial. Given this development, Infrastructure-as-a-Service Provider, e.g., Amazon[1] or Rackspace[2] are becoming increasingly relevant. In the Cloud, functionality and responsiveness of large software landscapes is no longer bound to hardware limitations, but dependent on how well the system scales with growing and shrinking workload. As most software landscapes are heterogeneous, all parts of the system have to have the possibility to scale along with the rest of it. For example, a growing load on the web frontend and therefore an increasing number of queries to the database backend results in the need for scaling web as well as database servers.

We make use of application-level monitoring frameworks to gain knowledge of the internal structure and behavior of applications. However, when doing live analysis in a dynamic scaling Cloud environment, a large number of traces have to be processed by the framework.

In the Cloud, such frameworks need to scale with the monitored application.

## 1.1 Motivation

Fittkau et al. 2013b present an architecture for a multi-level monitoring solution. In order to be able to use this approach in real world applications and in the Cloud, it has to be evaluated how well the solution performs in different workload scenarios.

## 1.2 Goals

In the present Bachelor's thesis, the three goals (G1 to G3) and an optional goal (G4) will be addressed, which will be described in the following section.

[1] Amazon Web Services: http://aws.amazon.com/
[2] Rackspace: http://www.rackspace.com/

### G1: Deploy Envisioned Architecture to the Cloud

The first goal is to deploy the architectures described in [Fittkau et al. 2013b] to the Cloud. To achieve this goal, it has to be ensured that the used tools are working as expected and offer the features we need in later stages of the project. In particular, the tools need to scale, monitor and in general orchestrate the Cloud. The goal is satisfied, when various, pre-defined deployment architectures are able to scale and run correctly in the Cloud.

### G2: Collect Monitoring Data in Different Scaling Scenarios

In G1 the necessary infrastructure for our project was created and it is now possible to run different performance evaluations in the Cloud. The second goal includes collecting monitoring data from monitoring nodes and determining how workload affects our monitoring architecture.

### G3: Analysis of Collected Data and Discussion of Results

We will analyze our collected data and make statements on how different workload and deployment scenarios affect the monitoring nodes. This goal will yield in guidelines on how to deploy and scale monitoring solutions in the Cloud.

### G4 (optional): Visualize Collected Data with ExplorViz

The visualization of our monitoring data can be done by ExplorViz, a tool further described in Chapter 2. ExplorViz enables organizing the data and displays a landscape view of our monitored application architecture. This goal is optional and not critical to the success of our research project.

## 1.3 Document Structure

The thesis is organized as follows. Chapter 2 presents technologies and foundations used throughout this thesis. In Chapter 3 we outline the state of the art of monitoring applications in the Cloud and describe different approaches. Chapter 4 shows our approach and architecture we chose for this project. The following chapter is about the actual implementation (Chapter 5) of our architecture. The evaluation of our approach is set out in Chapter 6. We end the thesis with a conclusion and possible future work (Chapter 8).

Chapter 2

# Foundations and Technologies

In this section, we describe different technologies which are used throughout our thesis. The technical foundations, the project is based on, are also explained in this section.

## 2.1 Dynamic Analysis and Monitoring

Monitoring frameworks are used to observe applications at runtime. It can be used to discover erroneous or unintentional behavior or to measure the performance of the application. In our project the monitored application generates traces which are then processed by the analysis worker nodes. The traces contain the monitoring data, for example, method calls. To extract the desired information of the raw traces ExplorViz (Section 2.6) is used.

## 2.2 Deployment Strategies for Monitoring in the Cloud

In a Cloud environment and especially when adding and removing instances dynamically to and from the environment, it is important when and how this is conducted. In our work, we summarize the decision making in this process as deployment strategies. Different approaches have been undertaken and described in [Fittkau et al. 2013b] and [Brunst et al. 2003a].

## 2.3 Cloud Computing

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.(Definition by Mell and Grance 2011) In our project, we will use a private Cloud which is owned and managed by our working group and running OpenStack. This enables us to run our experiments in an environment under control, without possible interference from other users.

## 2.4 OpenStack

OpenStack is an IaaS (Infrastructure as a Service) framework and is used to run our experiments on. OpenStack enables to orchestrate virtual machines running on different computing nodes and also provides facilities for virtual networking. The framework is developed by the OpenStack Foundation since late 2010. It is used by companies such as AT&T, CERN, and Sony either to directly sell Cloud computing resources or as a foundation to provide dynamically scaling SaaS (Software as a Service) to customers. For this project the private Cloud of our working group is at our disposal with a capacity of about 256 virtual machines we can run simultaneously.

## 2.5 Automatic Adaptive Capacity Management for IaaS

OpenStack lacks the ability to balance network load between virtual machines and an automatic scaling mechanism. For this project, we need to evaluate load-balancing solutions as well as scaling frameworks. A possible candidate for these tasks is SLAstic Lite, a framework which is based on SLAstic (van Hoorn et al. 2009a) and forked by a student in his bachelor's thesis (Koppenhagen 2013). To start or stop instances through software in a Cloud, we need an interface to the used Cloud framework. OpenStack offers a REST API for applications to manage instances and to gather information about the Cloud status.

## 2.6 ExplorViz

In our project we collect monitoring data from an application. To visualize and organize this data we use ExplorViz, a tool written by Florian Fittkau and described in [Fittkau et al. 2013a]. ExplorViz offers a browser interface to view the landscape model of the monitored application environment and makes it possible to interact with it.

## 2.7 Monitoring Analysis Worker

The monitoring analysis worker are programs running on their own node each which collect traces from the monitored application and deliver it to the master node. These nodes need to be scaled with the application.

Chapter 3

# Monitoring and Scaling Strategies

In the following, monitoring and scaling strategies which are used in business tools and production environments are described. State of the art for monitoring in the Cloud is outlined. Finally, approaches which are suitable for our project are presented. Possible strategies are described in [Fittkau et al. 2013b] and [Brunst et al. 2003b].

## 3.1 State of the Art

Nowadays, monitoring information technology infrastructure serves different purposes. Deploying a monitoring solution provides support in detecting, for example, errors and performance bottlenecks or to analyze high load periods. To provide these features, it is necessary to collect specific information of the observed system. Different sections of the system might need section specific monitoring techniques. For this reason a large amount of tools are available to gather data about particular nodes (e.g., CPU usage or HDD throughput), a specific application (e.g., method runtime or RAM usage) or the network itself (e.g., throughput or response time).

In the Cloud, we usually deal with large, dynamic scaling system landscapes. Due to this circumstance, the demands on monitoring tools have changed and new challenges need to be solved. As it is common in Cloud environments to scale applications horizontally instead of vertically, nodes are added and removed from the computing environment very frequently. This raises the need to parallelize the monitoring solution. In this context, it is also favorable to minimize the footprint the monitoring application introduces per node.

As the generated monitoring data in the Cloud grows large and complex, it becomes difficult to handle with traditional data processing applications. NoSQL databases (like Redis[1] or Cassandra[2]) are used and provide key-value databases, which are, in comparison to relational database systems, optimized to scale horizontally. The term 'big data' has established itself and describes data sets with sizes, usually in the petabyte to exabyte range, which are hard to manage with commonly used tools in a tolerable amount of time. To analyze and search this data, the MapReduce[3] programming model can be used and has been implemented by many different projects (a popular open-source project is Apache

[1] Redis: http://www.redis.io/
[2] Cassandra: http://cassandra.apache.org/
[3] Google MapReduce: http://research.google.com/archive/mapreduce.html

Hadoop[1]).

Existing monitoring applications like Nagios[2], impose difficulties when used in a Cloud environment and don't offer the needed flexibility when it comes to dynamic infrastructure changes during runtime. Configuration management systems may help to balance these shortcomings by applying configuration changes to these components on-the-fly.

Live application-level monitoring in the Cloud leads to other challenges. Due to the fine granularity of monitoring data and the challenge to visualize given data in real time, the resource requirements on the processing architecture are high. Especially when the CPU load of the monitored application is coupled to the amount of produced monitoring data, it is important to be aware of the scaling requirements of the employed monitoring solution. To overcome these problems, the monitoring system needs to use the advantages of the Cloud, for example, the possibility to rapidly scale to workload demands.

In the following, we describe one possible approach to large-scale application monitoring.

### 3.1.1 Parallel Analysis Infrastructure Approach

Brunst and Nagel 2003 describe an architecture to monitor and analyze performance of parallel applications. Figure 3.1 shows their system architecture. The green nodes are

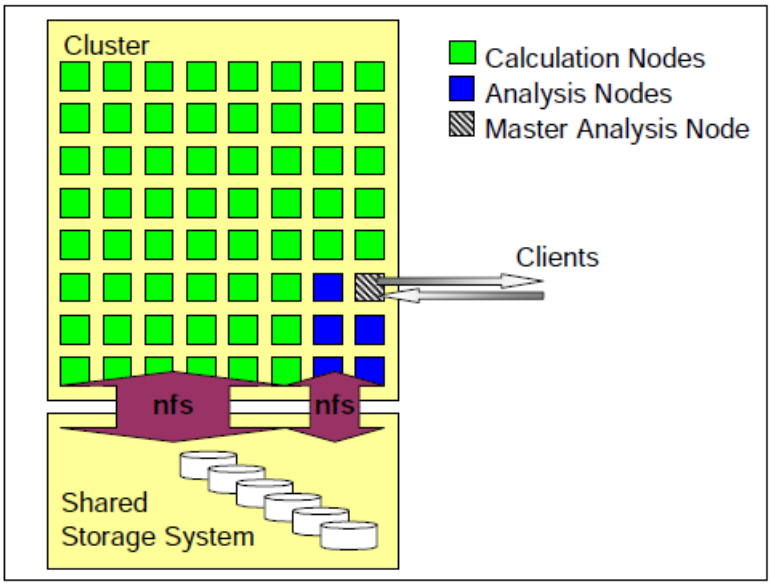


**Figure 3.1.** Parallel analysis architecture (taken from [Brunst et al. 2003a])

[1]http://hadoop.apache.org/
[2]Nagios: http://www.nagios.org/

monitored applications and the blue nodes are analysis worker. The generated monitoring data is distributed to the analysis nodes through a network file system. To save the collected data, this approach uses a specific format which can be read by the analysis nodes.

## 3.2 Single-Tier Approach

The single-tier approach only uses one layer of analysis worker nodes to collect and analyze the trace data. The workers then send their data to the master server. The master server aggregates the trace data and provides an interface to visualize it. To distribute the monitoring data and thus the workload evenly between the workers, we need to deploy a load balancing solution. This approach focuses on scaling out analysis workers in a single layer.

## 3.3 Multi-Tier Approach

The multi-tier approach uses additional layers of analysis worker nodes to process trace data. As the single-tier approach lacks the ability to scale out the master server, the multi-tier approach adds this functionality. The architecture shown in Figure 3.2 illustrates

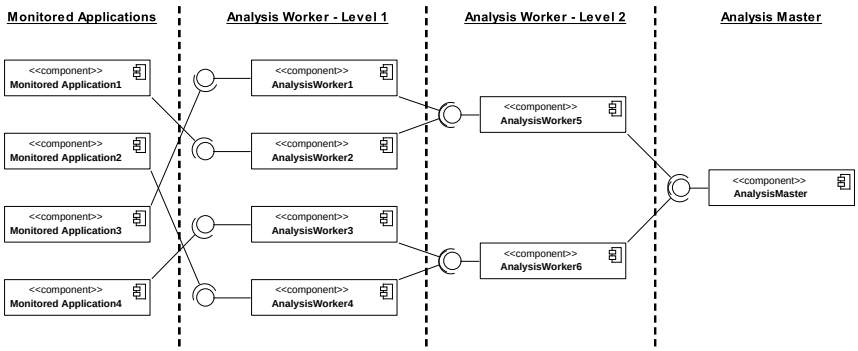


**Figure 3.2.** Multi-tier monitoring architecture (taken from [Fittkau et al. 2013b])

this approach. The monitoring data is reached from the monitored applications layer to the first layer of analysis worker load. The additional layer of analysis worker is added or removed by the capacity management solution when the analysis master is over- or underutilized.

Chapter 4

# Our Monitoring and Scaling Approach

This chapter describes our approach for a monitoring and scaling architecture. We outline what needs to be considered to design and implement this environment.

## 4.1 Service-Oriented Architecture

We design our architecture in a service-oriented approach. Service-oriented architectures benefit from a better reusability of services and follow various design principles. As we plan to integrate different software components into our environment it is favorable to consider some of these guidelines. We will follow these guidelines:

- service loose coupling,
- service reusability,
- service statelessness and
- service abstraction.

As we want to be able to exchange every software component in our environment, we try to maintain a loose coupling between the services. This makes it also easier to reuse different components from our project in other projects. The service statelessness is especially important for our approach as it enables services to be scaled independently from others. The abstraction of our components is conducted by defining only the necessary functions to the outside and providing consistent interfaces.

## 4.2 General Architecture

The architecture of our approach consists of three different layers of components which will be described in detail in the next subsections. The components are designed to comply to our service-oriented structure.

### 4.2.1 Scalable Application

We want to observe the scaling behavior of different components, thus we need an application which can be deployed easily and independent from others and serve as a generator for monitoring data. In our case we use a web application and instrument methods which are called during a web request. The web application communicates to the analysis worker through a defined interface. As our application needs to be scalable it is important to maintain the statelessness of this component. A critical part of this venture will be the persistence of log files as this data makes our service stateful.

### 4.2.2 Workload Generation

As the amount of our generated monitoring data is now dependent on how much HTTP-requests the web application has to process, we will deploy a HTTP-request generator in front of the application servers to simulate users trying to reach the application. Another important aspect of the generated workload is the realistic distribution of requests, as it should reflect real user behavior.

### 4.2.3 Analysis Worker

The third component of our architecture is the monitoring backend. It includes all software parts responsible for receiving, handling and forwarding traces. As we want to scale this component to the applied workload we need to make sure the analysis service is stateless as well.

### 4.2.4 Specific to Scaling Strategy 1

As laid out in Section 3.2 we want to scale out a single group of analysis worker. This can be achieved by using a capacity manager described in Section 4.3. As this strategy doesn't take the load of the analysis master into account it is not necessary for the capacity management to be aware of this instance. Hence, this scaling strategy leads to exactly two service groups. One group for the monitored application and the other one for the analysis layer.

### 4.2.5 Specific to Scaling Strategy 2

This strategy leads to a more complex environment configuration as for the first scaling strategy. The capacity manager has to keep track of the analysis master and make decisions based on its state. Also, service groups need to be dynamically added and removed during runtime from the architecture.

## 4.3 Load Balancing and Capacity Management

To automatically add and remove instances to the Cloud environment, it is necessary to employ services to observe the current amount of available instances and the number of required instances to suffice the current workload. Also these services need to be able to communicate the will to start or remove an instance and how this instance should be configured. We assign a group to each node managed by our capacity management service. This way, different service groups can be scaled independently from each other.

Another important functionality which depends on the capacity management is the load balancing between the service groups. The workload needs to be evenly distributed between the nodes and thus the load balancers need to be aware of the instances they can route traffic to and in which service group each node is.

Chapter 5

# Implementation

Different challenges came up concerning different components of our architecture. The implementation details and challenges encountered at the monitored application component are described in Section 5.1. In Section 5.3 we describe how workload is produced and distributed throughout our architecture. Section 5.4 explains how our infrastructure dynamically changes during runtime and how this process is automated. As the state and behavior of our infrastructure needs to be observed, we deployed a log data aggregation system which is described in Section 5.5. The provisioning and configuration of Cloud instances is shown in Section 5.6 and finally an overview of our architecture in Section 5.7 concludes this chapter.

## 5.1 Monitored Application

To simulate workload on analysis worker nodes we must have a source of monitoring records. In our experiment we use an adapted and instrumented version of JPetStore. JPetStore is a Java web application which serves a simple online shop for pets. It makes use of the iBATIS[1] framework as its persistence layer and Apache Struts[2] as its web application framework. The simple structure makes the build and integration process into a server landscape easy, because there are no external dependencies, for example, to database servers.

We plan to deploy the JPetStore application to a Apache Tomcat[3] server. As a lot of HTTP requests are generated for the Tomcat to process, we have to change the default configuration of it to handle high workload. There are different configuration options to tune. First, as the Tomcat server is a Java application and therefore runs in a Java Virtual Machine (JVM), we can pass arguments to the JVM to change its behavior.[4] *-server* can be used to enable server runtime optimizations in exchange for longer startup time. We also change the heap size to match our server hardware capabilities, i.e., as we plan to use virtual machines with 4 GB of RAM for our "monitored-application" nodes we set the parameter *-Xms64m -Xmx4g* accordingly.

[1] MyBATIS: http://blog.mybatis.org/
[2] Apache Struts: http://struts.apache.org/
[3] Apache Tomcat: http://tomcat.apache.org/
[4] JVM Parameter: http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

### 5.1.1 Instrumentation and Monitoring of JPetStore

The instrumentation of JPetStore is accomplished with AspectJ. AspectJ provides the possibility to add additional method calls to a Java program before it is ran. Therefore, we can add calls to our monitoring framework without the need to have the source code of the program we want to monitor. We specify which methods are instrumented and therefore monitored using a configuration file. As AspectJ supports load-time weaving, as described above, we don't have to recompile the target program after a change in the AspectJ configuration, but only have to restart it to pick up the new configuration.

For our use-case, the number of instrumented methods in JPetStore directly correlates to the amount of methods processed by the *analysis worker* nodes. We use the configuration in Listing 5.1 for weaving.

**Listing 5.1.** aop.xml

```
1 <aspectj>
2         <weaver options="">
3                 <include within="com.ibatis..*" />
4                 <include within="org.apache.struts..*" />
5         </weaver>
6
7         [...]
8 </aspectj>
```

In line 3 and 4 we define that all methods in all classes in the packages *com.ibatis* and *org.apache.struts* will be instrumented and monitored. This instrumentation covers large parts of the JPetStore application like every database access and every call to the web-framework related methods will be monitored.

### 5.1.2 JPetStore Adaptation

Due to the fact, that in JPetStore most requests are processed within 2 milliseconds, it is difficult to generate appropriate CPU load for our scaling strategy. We decided to adapt JPetStore to generate additional CPU load per request and implemented a method, shown in Listing 5.2, to accomplish that.

**Listing 5.2.** JPetStore adaptation

```
1 public final int compute() {
2   int retVal = 0;
3
4   int i = 0;
5   while (i < 12582912) {
6     if(i==0)
7       retVal = new Random().nextInt();
```

```
 8      retVal = retVal + 2;
 9      i++;
10    }
11    return retVal;
12  }
```

To prevent the JIT compiler from optimizing the code and thus not generating load during the execution of this method, we introduced line 6 and 7. After introducing this change we observed a 50% higher execution time per request and larger amount of CPU load.

### 5.1.3 Load Balancing the Monitoring Workload

The collected monitoring data needs to be distributed to the *analysis worker* layer. To make the workload on the *analysis worker* predictable and therefore easier to scale out, it is necessary to distribute the load evenly between the nodes. We accomplished that by frequently changing the connected *analysis worker* instance and thus letting the load balancer decide which node receives the workload. The downside of this approach is, that there are short periods of time were no analysis worker is connected. The monitoring data collected during this time is cached in a buffer on the client. Depending on how many requests the JPetStore application currently serves, this may lead to high client side memory consumption and workload spikes on the new connected analysis worker when the big buffer is sent to it.

Our experiments demonstrated that a connection refresh interval of five seconds is a good tradeoff between distributing workload and client side memory consumption.

## 5.2 Monitoring Layer

Our monitoring layer is responsible to receive and process monitoring records from the monitored applications. We employ ExplorViz analysis worker nodes to collect these records and reach them to their ExplorViz master server. Similar to the JPetStores we scale the analysis workers with the workload they receive.

## 5.3 Workload Generation

To test the scalability of applications, we need to generate appropriate workload. In our scenario we want to produce load to use the JPetStore nodes to full capacity and therefore make our Capacity manager (Section 5.4) add instances to withstand the workload. In the context of web servers, workload usually stands for HTTP requests processed by the server. Consequently, we crafted a large number of HTTP requests and sent them to the JPetStores. The *workload-generation* layer is deployed in front of the *monitored-application*

layer. To distribute HTTP requests from the *workload-generation* layer to the web servers, we deploy load balancers (Section 5.4.2) to choose which node receives the request.

As every web application is different in terms of reacting to load, it is important to adapt how and where workload is applied specifically for each application. This fact needs to be considered when creating stress tests and also plays a role when evaluating tools for this task. To be as flexible and versatile as possible we chose JMeter as our HTTP request generation tool.

### 5.3.1 JMeter

JMeter is a Java tool to load test web applications. It has been in development since 2001 and the latest version was released in 2014. It provides a GUI to create and execute tests and a CLI mode to run tests on headless servers. JMeter uses threads to send a pre-defined sequence of HTTP requests to a specified server. Therefore the number of used threads for a test determines the amount of workload produced. HTTP requests in JMeter can be configured in different ways and various options may be applied.

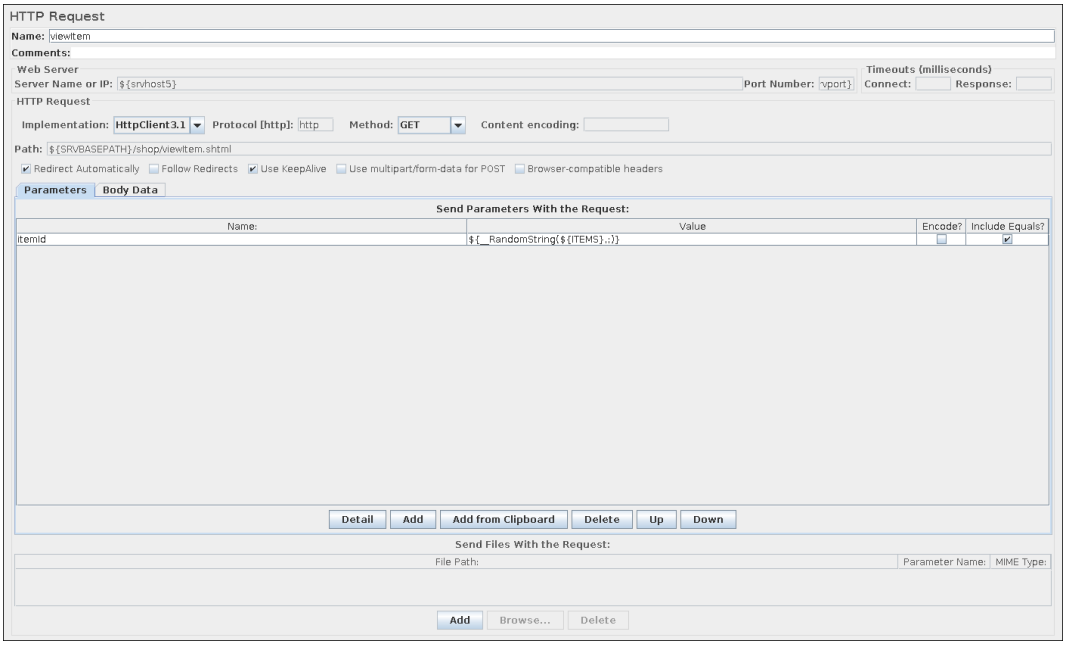


**Figure 5.1.** JMeter request editor

Figure 5.1 shows a screenshot of the JMeter GUI to setup a request. In the image we can see basic options like protocol, receiver IP address, receiver port and more advanced options like connection timeouts and different implementations of HTTP engines used for

this request. It is also possible to attach payload, like files and parameters to a request.

As JMeter lacks the ability to change the number of active threads during runtime of a test, we extend the functionality of JMeter by using a plugin called Markov4JMeter.

#### Markov4JMeter

Markov4JMeter is a JMeter plugin and was developed by André van Hoorn [van Hoorn et al. 2009a]. The plugin allows us to alter the number of active threads during a stress test and to change the used HTTP requests by defining a Markov Model. These features make it possible to model a realistic workload curve. For our web shop we use two different user behaviors. One, where the user resembles a casual browser without buying anything and the other represents a user who uses the login, shopping cart and ultimately buys a pet. The behaviors each have a certain probability of occurring.

We shape the workload by defining a function which takes the number of minutes elapsed since start of the test and returns the number of active sessions for this moment in time. In Listing 5.3 the used function is shown. The resulting workload of this function is further explained in Section 6.4.

**Listing 5.3.** Markov4JMeter function

```
1  private static double numSessions (double min){
2    double retVal = 0;
3    retVal += (min<=230)?1/((min+80)/20d)*1.2:0;
4    retVal += (min>230 && min<=320)?(min-160)*0.001d:0;
5    retVal += (min>320 && min<=410)?(min-280)*0.004d:0;
6    retVal += (min>410 && min<=620)?Math.cos((min-545)/120d)*1.2:0;
7    retVal += (min>620 && min<=650)?(min-1600)*-0.001d:0;
8    retVal += (min>650 && min<=690)?(min+1250)*0.0005d:0;
9    retVal += (min>690 && min<=850)?Math.cos((min-790)/200d)*1.1:0;
10   retVal += (min>850 && min<=965)?(min-330)*0.0020d:0;
11   retVal += (min>965)?Math.cos((min-1150)/200d)*2.1:0;
12 }
```

## 5.4 Capacity Management

Capacity management in the Cloud describes the process of keeping track of deployed resources, like server instances and making decisions based on the utilization of these managed resources. In our scenario, our capacity management solution is responsible of initiating the addition and removal of nodes to our server landscape. The decision when the infrastructure needs more resources is based on the current load of individual machines and clusters of machines. To automate this process we employ SLAstic Lite.

### 5.4.1 SLAstic Lite

SLAstic Lite is a Java application and is based on SLAstic, a tool written by André van Hoorn [van Hoorn et al. 2009a]. In general, SLAstic Lite performs two tasks in our environment. First, it observes and manages two scaling groups and secondly, it updates the load balancers on infrastructure changes.

The application follows the client-server principle. The CPU utilization monitor clients are deployed on every machine that should belong to the managed infrastructure of SLAstic Lite. CPU utilization monitor clients connect to the SLAstic server and frequently deliver information about the machine state to it. The delivered data contain current CPU load and RAM usage and are sent out every second. The server manages groups of machines (scaling groups) and assigns each received workload record to a node in a scaling group. Thus, the server is able to calculate the average CPU load of every group and can therefore make decisions based on those information. It is possible to define rules when the server starts and removes nodes. For example, for our experiments we chose a lower CPU threshold of 15 percent to shutdown one node of the concerned scaling group (down scaling) and 60 percent as the upper limit before a new node is added (up scaling). To initiate a new instance boot, the server needs access to our private Cloud framework. In OpenStack, this can be done by using a provided REST API, which is accessible from all virtual machines within OpenStack.

Figure 5.2 shows how our capacity management is integrated in our server architecture. The diagram shows two scaling groups. The green group contains our monitored application layer (Section 5.1) whereas the red group includes all nodes deployed to the analysis worker layer. The load balancers shown in the lower left side are not assigned to a scaling group and therefore are not monitored for CPU usage by the SLAstic server. Nevertheless they are updated by the server component through a HTTP interface. The OpenStack Nova node represents the Cloud framework and houses its API to manage the virtual instance repository.

### 5.4.2 Load Balancer

As mentioned earlier, a feature of SLAstic Lite is to populate available resources to the load balancers. This is important, as the incoming load should always be evenly distributed to the different layers. Also, unavailable nodes shouldn't receive any traffic and thus should be deleted from the load balancers. In our experiment the load balancers provide a HTTP interface to conduct changes to them. IP addresses can be retrieved per scaling group and are given out using the round-robin algorithm.

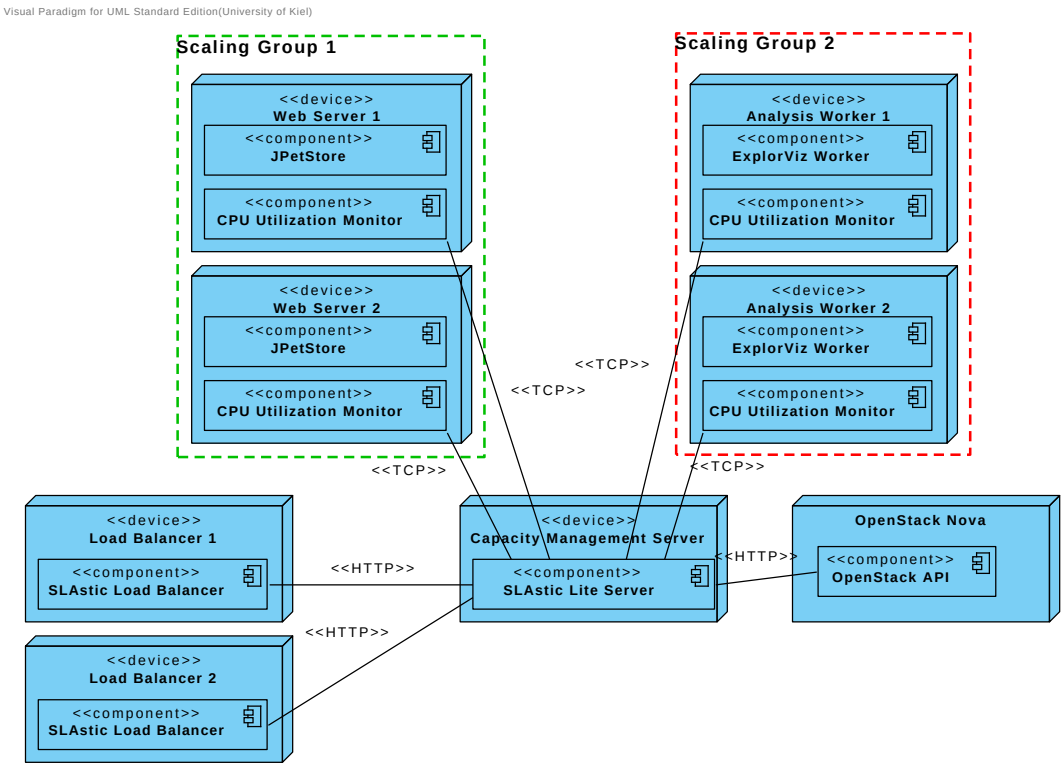


**Figure 5.2.** Capacity management architecture

## 5.5 Log Data Aggregation in Distributed Environments

In distributed server landscapes the state and health of the environment is determined by server-specific metrics. These metrics could contain data like, for example, current CPU load, memory consumption or application specific details like the amount of answered HTTP requests with HTTP status 200 (request succeeded). Usually this information is saved in log files on the server. In order to discover faulty behavior of services quickly, it is desirable to employ monitoring tools which observe the state and sanity of servers from an external point, or to accumulate and save the log data in a central repository. Both approaches have pros and cons but in a frequently changing, dynamic Cloud environment only a central data repository is a viable option for us.

Dynamic scaling in our project is characterized by adding and removing nodes during application runtime. While adding servers to the cluster works, removing nodes imposes problems as this action also expunges all locally acquired data, like log files, from the node (we denote the storage device of this node as *ephemeral*). This carries the risk of losing errors happened while this node was running and prevents the possibility of debugging

and fixing faulty software. To circumvent this issue, we decided to collect log data in a remote storage location as they occur. This exposes errors instantly and also provides us with the ability to search and filter data after shutting down the source server.

Our evaluation for appropriate tools lead to Logstash, which will be further described in the following section.

### 5.5.1 Logstash

Logstash[1] is a tool to manage and collect logs. It consists of an application to send log data from a server and a web interface (*Kibana*) to browse and filter collected logs. As its backend datastore, Logstash can be used with ElasticSearch[2], a search database written in Java, which provides features like full-text search and a RESTful web interface. Eventually, we use ElasticSearch to store the logs delivered by Logstash and let Kibana query the ElasticSearch API to search through and filter logs. Kibana is a web application written in HTML and JavaScript and can easily be deployed on a local web server.

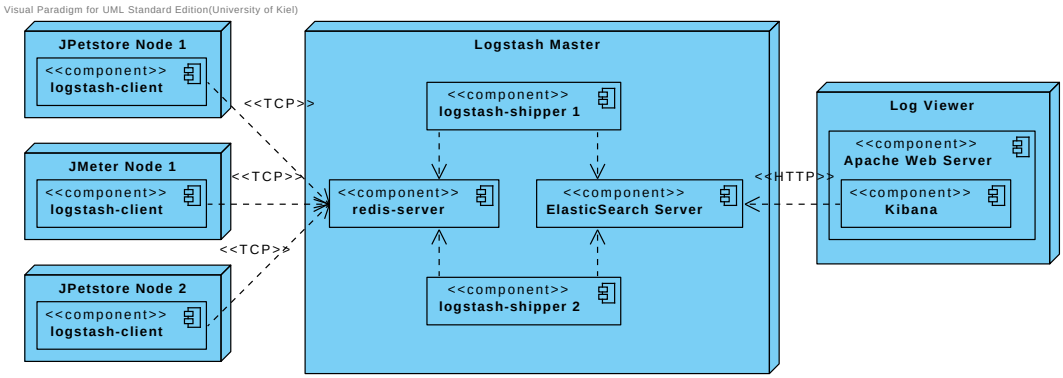


**Figure 5.3.** Logging architecture

Figure 5.3 illustrates our logging architecture and how each component is deployed. As our environment is variable, the number of JPetStore and JMeter nodes may change at any given point in time. The *redis-server* component is a Redis[3] Server deployment, an in-memory key-value storage which serves as a 'broker' between remote Logstash clients and the ElasticSearch backend.

In the following, we list an example Logstash client configuration file which shows how an input log file is delivered to the Redis server component.

**Listing 5.4.** Logstash client configuration

[1]Logstash: http://logstash.net/
[2]ElasticSearch: http://www.elasticsearch.org/
[3]Redis: http://redis.io/

```
input {
  file {
    path => "/opt/slastic/applications/jpetstore/logs/*request.log"
    type => "jpetstore"
  }
}

filter {
  grok {
    match => { "message" => "%{COMBINEDTOMCATLOG}" }
  }
}

output {
  redis {
    host => "{{ pillar['redis-server']['ip'] }}"
    port => "{{ pillar['redis-server']['port'] }}"
    data_type => "list"
    key => "jpetstore"
  }
}
```

We can identify three different parts in this configuration file: The *input*, where we define all sources for log data. In our example these are log files for the JPetStore web server. Following, the *filter* section parses and classifies all incoming log data lines. The last section *output* defines where the collected log data will be written, in our case the aforementioned Redis server. The *host* and *port* line contain variables which are set by our *Configuration Management* tool, which is described in detail in Section 5.6.

## 5.6 Cloud Orchestration and Management

Dynamic scaling in the Cloud leads to the necessity of having a convenient way of configuring and controlling instances. In a vertical scaling architecture, adding and removing nodes to the cluster is rare because scaling is done by changing the available resources of a single server (e.g. adding more RAM). This technique is called *scaling up* and imposes the risk of over- or under-provisioning of servers and is inflexible when it comes to unexpected workload spikes.

In contrast to this method, a technique called *scaling out* or horizontal scaling is possible. This technique favors adding or removing servers to the cluster according to workload demands. In the Cloud, scaling out brings many advantages over other approaches as starting and stopping nodes is cheap in terms of working hours.

The downside is a considerable configuration overhead as we now have to deal with many servers instead of one server and configuration changes during runtime are harder to implement. This fact is a major drawback especially in our project as we expect to frequently change server and application configurations. To overcome this disadvantage we employ a *configuration management* tool which helps us to manage our Cloud instances and application stack life cycle.

During our evaluation of appropriate tools, we decided to use SaltStack for global configuration management.

### 5.6.1 SaltStack

SaltStack is a configuration management tool and enables us to save and apply the configuration of virtual machines in our environment in a reusable format. Configuration, in this context includes

- configuration files of applications,
- state of services (for example, started, stopped, etc.),
- downloading and building applications,
- installing and updating packages.

SaltStack uses a client-server model and applies configuration of a node over TCP connections. A typical SaltStack deployment consists of a master server and clients (so-called minions) which are configured by the master. The master server itself contains the configuration of all nodes and applies it when a new minion connects. Which configuration is applied to which node is determined by the hostname of the minion.

For our project this implies that we use two base images. One image contains a SaltStack master server installation and is started before a minion exists. The other one contains SaltStack minion installation. On start of a Cloud instance with the minion image it automatically connects to the master and applies the given configuration.

The configuration is saved in files using YAML[1] (YAML Ain't Markup Language). YAML is a data-oriented serialization format and thus can easily be used in conjunction with a version control system. Listing 5.5 shows the configuration of a JPetStore node.

**Listing 5.5.** JPetStore configuration

```
1 checkout-git:
2   git.latest:
3     - name: https://repository.org/gitlab/explorviz/slastic-lite.git
4     - rev: pst-bachelor
5     - target: /opt/slastic
```

[1] YAML: http://www.yaml.org/

```
6
7  /etc/init/jpetstore.conf:
8    file.managed:
9      - source: salt://jpetstore/upstart.conf
10     - template: jinja
11
12 jpetstore:
13   service:
14     - running
15     - enable: True
16     - watch:
17       - git: checkout-git
18       - file: /etc/init/jpetstore.conf
19     - require:
20       - sls: java
```

The example in Listing 5.5 shows a common use case of a configuration management tool. In Line 1 to 5 an application from a git repository is checked out. Line 7 initiates the replacement of the file */etc/init/jpetstore.conf* with a file from the master server (line 9). The service *jpetstore* will be started in line 12. In line 20 we define that this service depends on the package *java* to be executed.

We are now able to configure instances at boot time. After a configuration change at runtime, we can trigger a configuration refresh for all connected minions with the command shown in Listing 5.6.

**Listing 5.6.** Refresh configuration

```
salt '*' state.highstate
```

Another useful feature provided by SaltStack is the execution of specific commands on minions without connecting to it through ssh. It is also possible to execute a command on more than one machine in parallel. The command in Listing 5.7 returns the service status for all connected minions with hostname infix 'jpetstore'.

**Listing 5.7.** Status of JPetStores

```
salt '*jpetstore*' cmd.run 'service jpetstore status'
```

This feature makes it easy to keep the overview of the server environment.

## 5.7 Architecture Overview

As our architecture follows the principles of SOA (Service-Oriented Architecture), we structure the components as illustrated in Figure 5.4. On the left side we see the workload

generation layer which contains all servers dedicated to producing requests for our monitored applications Section 5.3. For distributing the workload the load balancers between each layer of components are responsible. The load balancers are managed and regularly updated by SLAstic Lite (Section 5.4). The workload is reached from the workload generation further to the web servers who process the incoming requests in Section 5.1. While processing these requests the web servers produce monitoring data for the monitoring layer (Section 5.2). The resulting workload for the analysis worker nodes is also load balanced and finally reached to the analysis master.

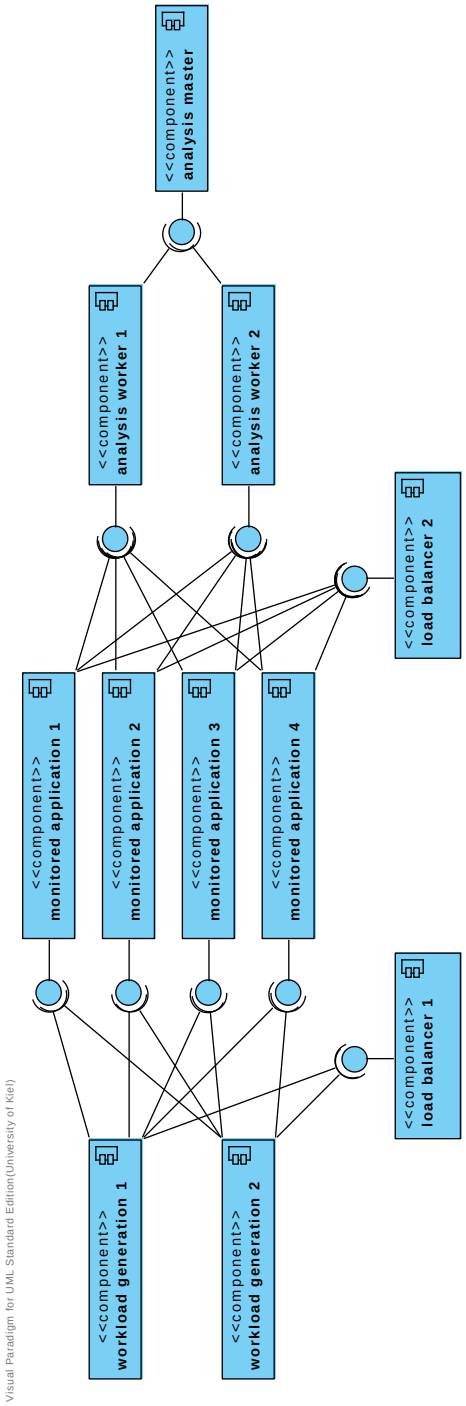


**Figure 5.4.** Final architecture

Chapter 6

# Evaluation

We want to use the results of our research to find guidelines on how to deploy monitoring solutions in the Cloud. These guidelines enable to use our results in real world projects and as reference when deploying monitoring nodes to the Cloud. In general, this chapter describes and evaluates our results. The goals of our analysis are outlined in Section 6.1. Following, we describe our research methodology in Section 6.2. As our experiments yield various data, we categorize this data in assessable metrics in Section 6.3. Ongoing, the experiment setup (Section 6.4) is described. After that, different scenarios (Section 6.5) and their results in Section 6.6 are shown. We conclude this chapter with a discussion of our results (Section 6.7) and possible threats to the validity of our collected data in Section 6.8.

## 6.1 Goals

We evaluate the behavior of our architecture and applications with respect to

- scaling,
- long durations of high load,
- impact of monitoring on performance

The following proposed and conducted scenarios help us finding characteristics of our environment and enable us to better analyze our monitoring system.

## 6.2 Methodology

In a dynamic environment it is difficult to gather data as a lot of resources in it are ephemeral. Thus, it is necessary to deploy systems which collect data and persist them for later use. In our implementation we use a system to aggregate log data from all active nodes (Section 5.5). As we use this software to debug and keep the overview of our environment it is consistent to use it to filter the maintained data to gain metrics. Missing data, which is not used to debug the software landscape but is required to form a metric can easily be added through additional log clients or changed log configurations.

## 6.3 Significant Metrics

Our deployed server environment produces a lot of data which we can use to evaluate the components in it. To keep the overview over this data we need to separate unnecessary from relevant information and choose sufficient metrics. We then use these metrics to evaluate the performance and other criteria of our server landscape. It is also essential to combine metrics to confirm possible relationships between them.

As the goal of our project is to examine and observe the scaling behavior of monitoring workers, we focus our data collection on metrics associated with scaling in the Cloud. Not mentioned, conceivable metrics throughout this chapter most likely fall into this category. An example for such a metric is the disk I/O of our nodes.

In the following, we outline significant metrics for our particular use case.

### 6.3.1 HTTP Requests

A metric related to the monitored application layer (Section 5.1) in our environment and web servers in general is the amount of processed HTTP requests in a given time frame. The rate of HTTP requests can be an indicator of the load the web server just endures. It may also be related to the disk I/O of the server, depending on how many disk intensive operations, like file write or reads, the hosted application contains.

HTTP request can be divided into two categories. Successful requests, usually provided with a HTTP status code 200 and unsuccessful or faulty requests (status codes like, for example, 404). For our project both categories are important, because error in requests indicate misbehavior of a component in our environment.

We track the processed HTTP requests per second for every web application node in our environment.

### 6.3.2 Active Nodes

Cloud instances are constantly added and removed during our scenarios. To comprehend the scaling behavior of our server landscape, we watch the numbers of active nodes. As our capacity manager assigns each managed node a scaling group (Section 5.4), we can determine the number of nodes for a scaling group.

### 6.3.3 CPU Load

The CPU load is a significant metric as it represents how busy a node or a group of nodes currently is. We log the CPU load of every managed node and form the average CPU load for its assigned scaling group.

### 6.3.4 Method Calls

The monitoring workers (Section 5.2) receive monitoring records from the monitored applications and process them. The records contain traces of, for example, all method calls which were done during a web request. To measure the throughput of our monitoring solution deployment we track the number of method calls contained in the monitoring records for each monitoring worker node. This metric is related to the amount of HTTP requests processed as for every request a trace is created and sent out.

## 6.4 Experimental Setup

To conduct our experiments we prepare our runtime environment. These settings are the same for all scenarios described in Section 6.5 and will be reset before each run. In the following we will show the configuration of components in our environment.

### 6.4.1 Workload Generation

Our workload generation layer is responsible to apply workload to the monitored application. We generate HTTP requests to accomplish this. Thus, prior to our experiments we configure JMeter with appropriate parameters. We deploy one workload generation node to our environment.

Figure 6.1 shows the produced workload graph. We measured the current workload of the web application nodes in *http requests per second*.

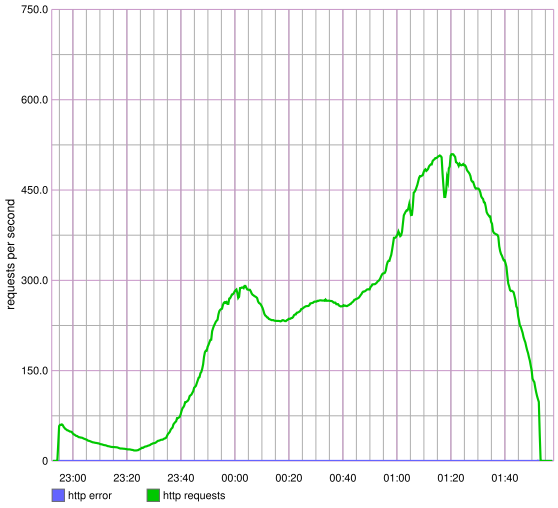


**Figure 6.1.** HTTP workload

The displayed graph resembles a workload curve of a website for 24 hours. The load increases for a first peak at noon and goes on for its highest peak in the evening around 8:00 pm.

In all our scenarios we use this workload and stretch out or compress its length as needed.

### 6.4.2 Scaling Groups

As explained in Section 5.4 we categorize nodes into scaling groups. Before applying workload, we will start exactly one node for each scaling group. This means, we will start one node with a started JPetStore service and one node with an analysis worker deployment.

### 6.4.3 Capacity Management

The SLAstic Lite server is started at the beginning and is ready to add or remove instances to or from our scaling groups. The scaling thresholds are the same for all scenarios. The CPU load needs to be below 15 percent to shutdown one instance of a scaling group and above 50 percent to make the capacity manager add a new one to it. As the SLAstic Lite server is also responsible to update the load balancers, we deploy and configure one separate node to balance the workload.

## 6.5 Scenarios

This section explains our two scenarios we conducted within our evaluation.

### 6.5.1 Hardware

For all our scenarios we use the following hardware for each instance:

- Workload Generator: 16 VCPUs, 60 GB RAM,
- Monitored Application Layer: 2 VCPUs, 4 GB RAM,
- Analysis Worker Layer: 2 VCPUs, 4 GB RAM and
- Load Balancer: 4 VCPUs, 12 GB RAM.

### 6.5.2 Scenario 1

In our first scenario we disable the monitoring component. Hence, this scenario yields no monitoring data for the analysis worker layer. By this scenario we test the scaling of our web servers and plan to use the results as comparable basis for our other scenarios.

The runtime of this experiment is three hours as it is not planned to use these results for reference in long-term scenarios.

### 6.5.3 Scenario 2

Our second scenario is conducted with enabled monitoring and thus we expect to see higher CPU load on the monitoring layer. In addition to the active nodes and CPU load metric for both scaling groups, we keep track of the processed method calls on the analysis master. We plan to compare the results from scenario 1 to this and thus, the duration of this scenario is also three hours.

## 6.6 Results

We will now show the results of our scenarios described above.

### 6.6.1 Scenario 1

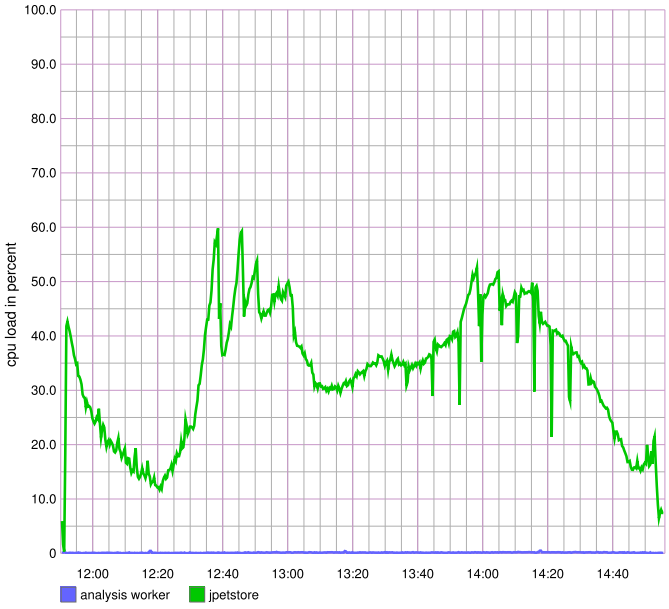


**Figure 6.2.** CPU load without monitoring

Figure 6.2 illustrates the workload curve for our monitored application layer. The workload curve always stays under 60% CPU consumption. As we disabled the monitoring for scenario 1, the analysis worker layer doesn't indicate any load.

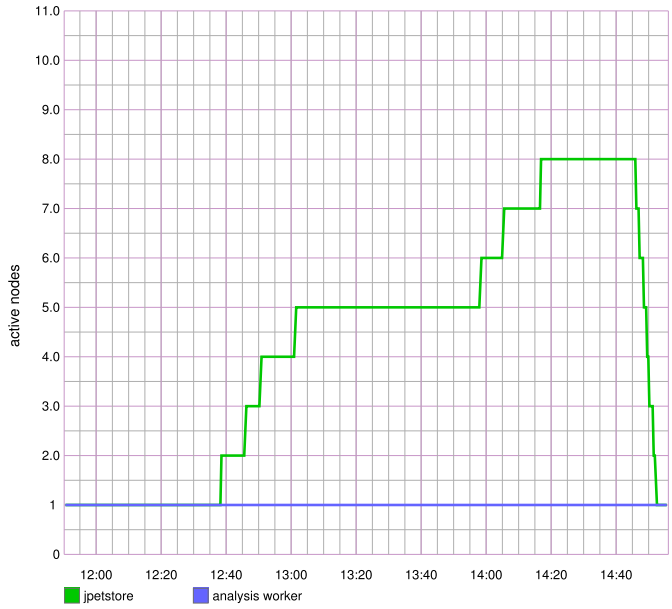


**Figure 6.3.** Active nodes without monitoring

The node count during our experiment is shown in Figure 6.3. The capacity manager starts up to five nodes for the first peak and goes on to scale up to eight instances on the highest point of the curve.

### 6.6.2 Scenario 2

For the second scenario we enabled the monitoring layer and thus are able to collect metrics about the amount of processed method calls.

Figure 6.4 shows the CPU load for our web servers and, in addition to that, the CPU load of our second scaling group, the monitoring layer. The CPU load curve for the analysis worker is similar to the curve of the JPetStores except its steeper on workload gradients. As we see in Figure 6.5 we scale to a maximum of nine JPetStore nodes and four analysis worker nodes. The counted method calls are shown in Figure 6.6 and peak to about 375000 per second in the second highest point.

### 6.6.3 Combined Results

The graphs in Figure 6.7 and Figure 6.8 show the combined CPU load and combined active nodes graphs from our two scenarios for direct comparison. The CPU load for the JPetStore instances is higher for the scenario where monitoring is enabled.

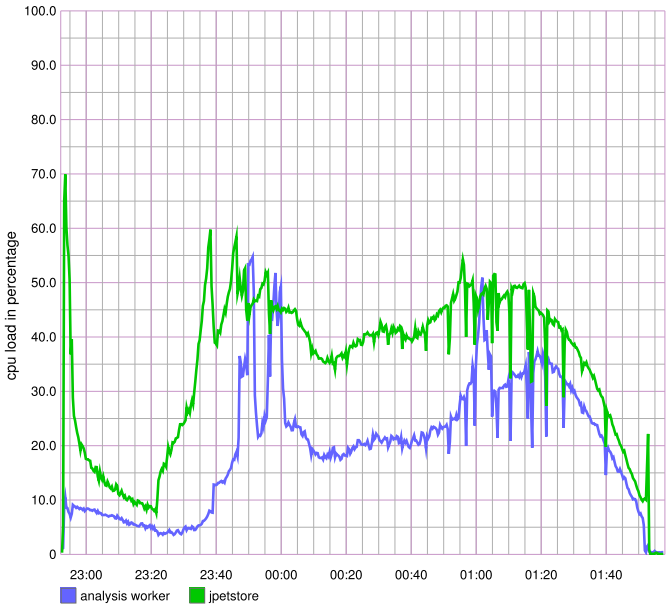


**Figure 6.4.** CPU load with monitoring

The node count for our second scenario is also higher in the peak load of our workload curve.

## 6.7 Discussion of the Results

### 6.7.1 Scalability

The results of our first scenario show that the basic scaling of groups works in our environment. The CPU load graph without monitoring in conjunction with the corresponding active nodes show, that every time the CPU load of our web servers rises above 50% a new node is started. This happens while our workload curve increases to the first peak and continues during the rise up to the second peak. The active nodes curve approximately resembles the shape of our workload curve.

The first scenario also assures, that the load balancers evenly distribute the traffic to our scaling group and are updated automatically when a new node is started. We can verify this circumstance by watching the CPU load drop after every start of a new node.

As expected, the second scenario yields roughly the same shape for the CPU load graph as seen in scenario 1. Our monitoring of the JPetStore instances is now enabled and thus leads to the CPU load curve for the analysis worker. The analysis worker itself are scaled up to four nodes, which is half the nodes we need for our web server layer. This yields

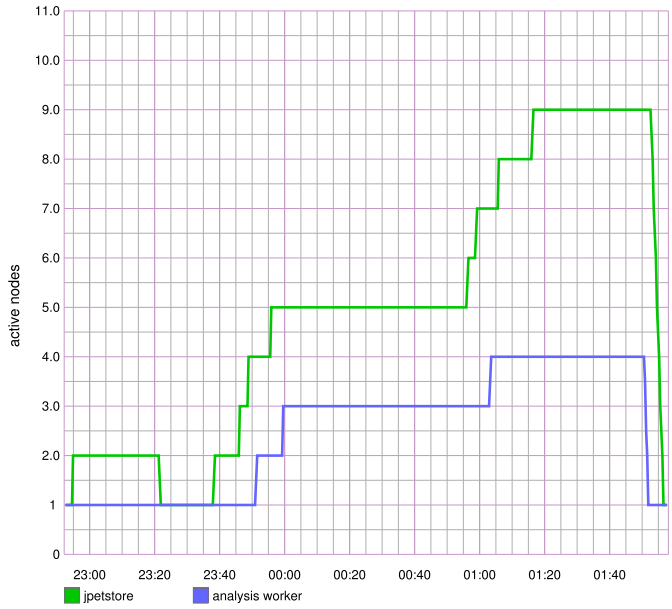


**Figure 6.5.** Active nodes with monitoring

a ratio of two web servers per analysis worker. Also, the load balancing for this layer distributes the monitoring records evenly.

The load curve of the analysis worker nodes show a steeper gradient and even load spikes when adding web servers to the monitored application layer. The analysis worker seem to have a per-connection CPU load spike which could be further investigated in additional scenarios.

Eventually, our results show that the analysis worker group scales linearly with the number of active web server nodes for our given workload.

### 6.7.2 Durations of High Load

Our scenarios both lasted three hours. Our results show, that during this testing our architecture worked as expected. To further confirm our results we could conduct additional, longer lasting scenarios in our architecture.

### 6.7.3 Monitoring Performance

The performance impact of our monitoring solution to the instrumented application can be seen in our combined graphs in Figure 6.7 and Figure 6.8. The CPU load on the monitored applications is higher during the second scenario and even leads to an additional started

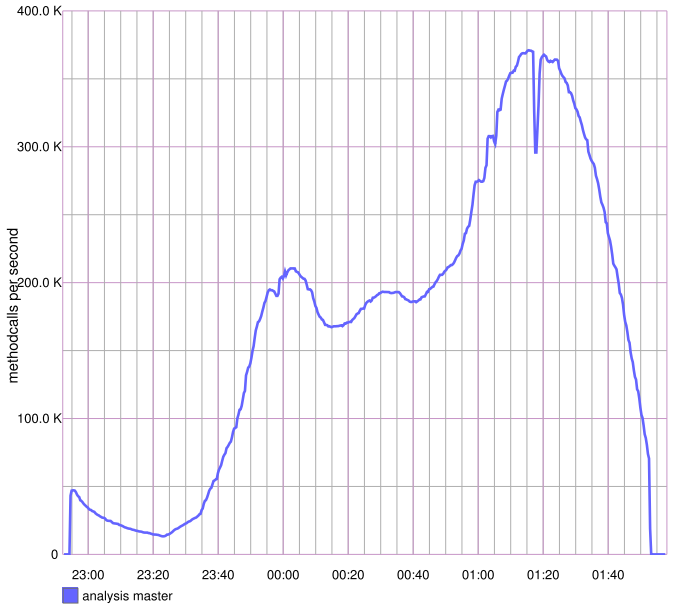


**Figure 6.6.** Method calls on the analysis master

node during the highest point in the workload curve. It is hard to determine the exact impact on performance with our collected monitoring data as it is not possible to compare CPU load curves with different amount of active nodes.

## 6.8 Threats to Validity

We conduct our scenarios in our private Cloud. As this is a foreclosed environment it induces difficulties to realistic tests in it. In our case we want to stress test a web application. Due to low latencies associations with network links between nodes in the Cloud our results may not be universally applicable.

Furthermore, the duration of our scenarios may impose problems as the results may not be valid for longer running tests. To eliminate this issue more experiments have to be conducted.

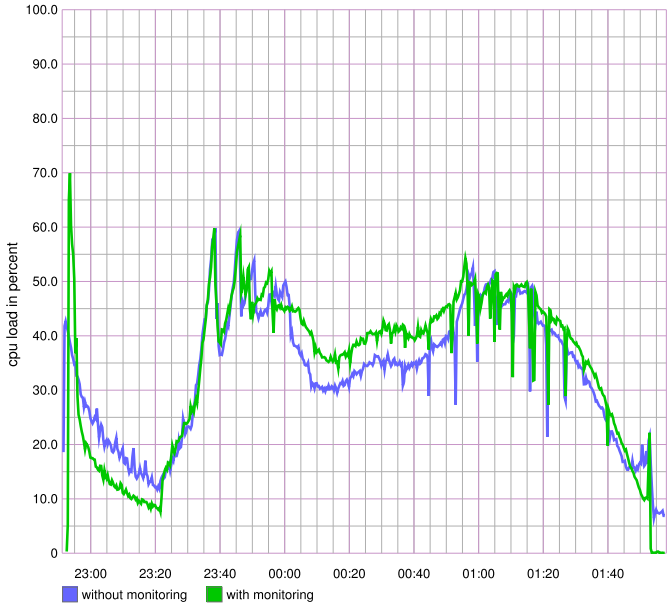


**Figure 6.7.** Combined CPU load

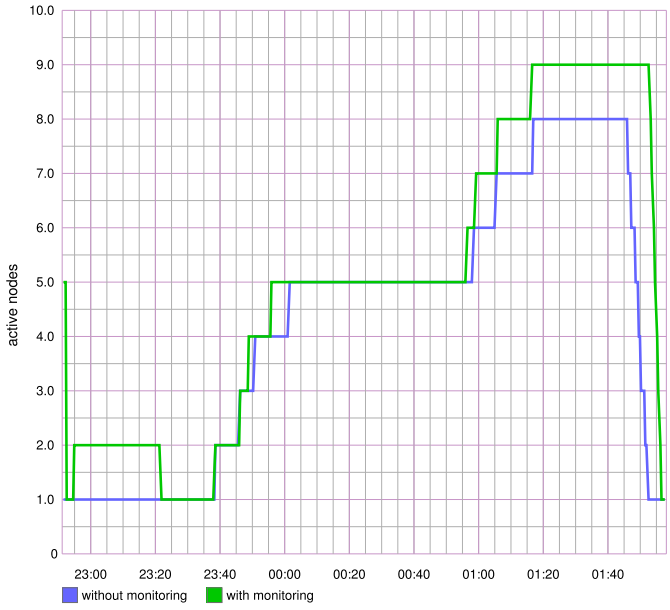


**Figure 6.8.** Combined active nodes

Chapter 7

# Related Work

In [Meng and Liu 2013] and [Meng et al. 2010] the authors propose a Monitoring-as-a-Service solution. To monitor the complex infrastructure of modern Cloud datacenters they developed a scalable and flexible monitoring topology consisting of different services. They optimized the resource consumption of their Cloud monitoring environment so that the deployed services don't put additional burden on the Cloud infrastructure. In difference to our approach they provide not only application level monitoring but monitoring for whole datacenter equipment like infrastructure, platform and applications.

Brunst and Nagel 2003 show an approach on a parallel analysis infrastructure. The proposed architecture is already illustrated and described in Section 3.1.1. In comparison to our approach Brunst and Nagel 2003 don't use TCP connections to distribute produced monitoring workload to the analysis worker nodes. Instead a shared storage infrastructure is used and the monitoring data is saved in files on it.

Chapter 8

# Conclusions and Future Work

This chapter draws the conclusions of the thesis and proposes possible future work.

## 8.1 Conclusions

During the thesis a Cloud architecture to test and observe the scaling behavior of our monitoring solution has been designed. First, different approaches to design a dynamic scaling Cloud environment have been laid out. Ongoing, these strategies have been implemented to our private Cloud platform. The resulting architecture is capable of dynamically adapt itself to workload demand. We then went on to conduct tests to examine the scalability of the deployed monitoring solution. The collected data make the dynamic behavior of our Cloud environment visible and enable us to understand how our deployed applications react to different workload. Furthermore, the results show that the analysis worker layer scales correctly and we are able to collect monitoring data from a monitored application.

The thesis provides a working Cloud deployment. It is possible to stress test the monitored application and observe its behavior. Furthermore, as the proposed architecture follows the principles of a service-oriented environment (Section 4.1) the used services are loosely coupled and can be substituted with other services as long as the defined interfaces are used. Thus, the integration of foreign services into our architecture is easy and the required adaptation effort is low. In addition to that, the deployed capacity management solution is suited to scale different groups of services and managing the load balancing facilities.

In respect to our objectives in Section 1.2 the thesis succeeded in fulfilling the three mandatory goals. The proposed architecture has been deployed to the Cloud and monitoring data has been collected. We then analyzed and discussed our results in Section 6.7.

## 8.2 Future Work

Different workload scenarios were evaluated in our monitoring architecture. To further confirm our results discussed in Section 6.7 more tests can be conducted. For example, a workload scenario with a duration of 24 hours or more would be helpful to examine the long-time behavior of the scaling groups. To rule out scheduling interferences of the hyper

visor from our results the hardware configuration for our instances could be varied. As we optimized our capacity manager to work in an OpenStack driven Cloud environment future work includes extending the functionality of it to other Cloud frameworks. Eventually, the load balancers currently don't support passing through TCP connections but only provide a HTTP interface to pull the IP addresses. This feature implies limitations on the clients and could be substance for future work.

Chapter 9

# Attachments

## DVD Contents

| | |
|---|---|
| **/ubuntu1310saltmaster.img** | Salt master image |
| **/ubuntu1310saltminion.img** | Salt minion image |
| **/monitored-application/** | Monitored-application |
| **/load-balancer/** | Load Balancer |
| **/slastic-lite/** | SLAstic Lite Server |
| **/worker/** | ExplorViz Analysis Worker |
| **/common-monitoring/** | ExplorViz Library |

# Bibliography


[Brunst and Nagel 2003] H. Brunst and W. E. Nagel. Scalable performance analysis of parallel systems: concepts and experiences. In: *Proceedings of the 10th Conference on Parallel Computing: Software Technology, Algorithms, Architectures, and Applications*. Edited by G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter. Dresden, Germany: Elsevier, 2003. (Cited on pages 6 and 37)

[Brunst et al. 2003a] H. Brunst, W. E. Nagel, and A. D. Malony. A distributed performance analysis architecture for clusters. In: *IEEE International Conference on Cluster Computing, Cluster 2003*. Hong Kong, China: IEEE Computer Society, Dec. 2003, pages 73–81. (Cited on pages 3 and 6)

[Brunst et al. 2003b] H. Brunst, A. D. Malony, S. S. Shende, and R. Bell. Online remote trace analysis of parallel applications on high-performance clusters. In: *High Performance Computing*. Volume 2858. Lecture Notes in Computer Science. Springer, Nov. 2003, pages 440–449. (Cited on page 5)

[Fittkau et al. 2013a] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: the ExplorViz approach. In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. 2013. (Cited on page 4)

[Fittkau et al. 2013b] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance — Joint Kieker/Palladio Days 2013 (KPDays 2013)*. CEUR Workshop Proceedings, 2013. (Cited on pages 1–3, 5, and 7)

[Koppenhagen 2013] E. Koppenhagen. Evaluation von Elastizitätsstrategien in der Cloud im Hinblick auf optimale Ressourcennutzung. Bachelor thesis. Kiel University, 2013. (Cited on page 4)

[Mell and Grance 2011] P. Mell and T. Grance. The NIST definition of cloud computing (draft). *NIST special publication* 800.145 (2011), page 7. (Cited on page 3)

[Meng and Liu 2013] S. Meng and L. Liu. Monitoring-as-a-service in the cloud: spec phd award (invited abstract). In: *Proceedings of the ACM/SPEC international conference on International conference on performance engineering*. ACM. 2013, pages 373–374. (Cited on page 37)

[Meng et al. 2010] S. Meng, L. Liu, and V. Soundararajan. Tide: achieving self-scaling in virtualized datacenter management middleware. In: *Proceedings of the 11th International Middleware Conference Industrial Track*. Middleware Industrial Track '10. Bangalore, India: ACM, 2010, pages 17–22. (Cited on page 37)

[Van Hoorn et al. 2009a] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring. An adaptation framework enabling resource-efficient operation of software systems. In: *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010*. WUP '09. Cape Town, South Africa: ACM, 2009, pages 41–44. (Cited on pages 4, 17, 18)

[Van Hoorn et al. 2009b] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Research Report. Kiel University, 2009.

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: a framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, 2012, pages 247–248.