Cost-effectiveness of tenant-based allocation model in SaaS applications running in a public Cloud

Wojciech Stolarz
This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author:
Wojciech Stolarz
E-mail: voytec0dh@gmail.com

University advisor:
Prof. Lars Lundberg
School of Computing
ABSTRACT

Context. Cloud computing is getting more and more interest with every year. It is an approach that allows Internet based applications to work in distributed and virtualized cloud environment. It is characterized by on-demand resources and pay-per-use pricing. Software-as-a-Service (SaaS) is a software distribution paradigm in cloud computing and represents the highest, software layer in the cloud stack. Since most cloud services providers charge for the resource use it is important to create resource efficient applications. One of the way to achieve that is multi-tenant architecture of SaaS applications. It allows the application for efficient self-managing of the resources.

Objectives. In this study I investigate the influence of tenant-based resource allocation model on cost-effectiveness of SaaS systems. I try to find out weather that model can decrease the system's actual costs in commercial public cloud environment.

Methods. I am implementing two authorial SaaS systems: first tenant-unaware and then using tenant-based resource allocation model. Then they are deployed into Amazon public cloud environment. Tests focused on measuring over- and underutilization are conducted in order to compare cost-effectiveness of the solutions. Public cloud provider's billing service is used as a final cost measure.

Results. The tenant-based resource allocation model proved to decrease my system's running costs. It also reduced the system's resources underutilization. Similar research was done, but the model was tested in private cloud. In this work the systems were deployed into commercial public cloud.

Conclusions. The tenant-based resource allocation model is one of the method to tackle under-optimal resource utilization. When compared to traditional resource scaling it can reduce the costs of running SaaS systems in cloud environments. The more tenant-oriented the SaaS systems are the more benefits that model can provide.

Keywords: Cloud computing, SaaS, multi-tenancy, cost-effectiveness.
# Table of contents

LIST OF ABBREVIATIONS.........................................................................................................................1

1 INTRODUCTION..................................................................................................................................2
  1.1 PURPOSE OF THE MASTER THESIS.........................................................................................2
  1.2 AIM AND OBJECTIVES...............................................................................................................3
  1.3 RESEARCH QUESTIONS / HYPOTHESES..................................................................................4
  1.4 LIMITATIONS...............................................................................................................................4

2 BACKGROUND......................................................................................................................................6
  2.1 CLOUD COMPUTING....................................................................................................................6
  2.2 SOFTWARE-AS-A-SERVICE.........................................................................................................7
  2.3 JAVA ENTERPRISE EDITION......................................................................................................8
  2.4 CLOUD ECONOMY.......................................................................................................................8
  2.5 MULTI-TENANCY............................................................................................................................9
  2.6 RELATED WORK.............................................................................................................................9

3 RESEARCH METHODOLOGY.............................................................................................................11
  3.1 METHODOLOGY...........................................................................................................................11
  3.2 MEASURES...................................................................................................................................12
    3.2.1 Overutilization.......................................................................................................................13
    3.2.2 Underutilization.....................................................................................................................14
    3.2.3 Cost.......................................................................................................................................15
  3.3 WORKLOAD GENERATION............................................................................................................15

4 SYSTEM DESIGN..............................................................................................................................18
  4.1 BASE SYSTEM................................................................................................................................18
    4.1.1 SaaS platform.......................................................................................................................20
    4.1.2 Database schema....................................................................................................................22
    4.1.3 Resource Monitor....................................................................................................................22
  4.2 TENANT-AWARE SYSTEM............................................................................................................24
    4.2.1 Tenant-based isolation............................................................................................................25
    4.2.2 Tenant-based VM allocation...................................................................................................27
    4.2.3 Tenant-based load balancing................................................................................................29
    4.2.4 VM Management....................................................................................................................30
  4.3 USED TECHNOLOGIES..................................................................................................................32
  4.4 AWS CLOUD..................................................................................................................................33

5 TESTS....................................................................................................................................................36
  5.1 TEST BED DEPLOYMENT..............................................................................................................37
  5.2 TEST PLAN.....................................................................................................................................39

6 RESULTS AND ANALYSIS................................................................................................................42
  6.1 RESULTS.......................................................................................................................................42
  6.2 ANALYSIS......................................................................................................................................48
    6.2.1 Performance.............................................................................................................................48
    6.2.2 Cost analysis............................................................................................................................53

7 CONCLUSIONS...................................................................................................................................57

8 SUMMARY AND FUTURE WORK.....................................................................................................60

9 REFERENCES.......................................................................................................................................62
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>CWA</td>
<td>Core Web App – a part of the Base System</td>
</tr>
<tr>
<td>EC2</td>
<td>Amazon Elastic Cloud Computing</td>
</tr>
<tr>
<td>ELB</td>
<td>Amazon Elastic Load Balancer</td>
</tr>
<tr>
<td>JMX</td>
<td>Java Management Extension</td>
</tr>
<tr>
<td>RCM</td>
<td>Resource Consumption Manager</td>
</tr>
<tr>
<td>RDS</td>
<td>Amazon Relational Database Service</td>
</tr>
<tr>
<td>SCWA</td>
<td>SaaS Core Web App – main part of the TBRAM system</td>
</tr>
<tr>
<td>TBRAM</td>
<td>Tenant-based Resource Allocation Model</td>
</tr>
<tr>
<td>VM</td>
<td>virtual machine</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

In this chapter I describe the purpose of this master thesis (Section 1.1). In the next sections (Section 1.2, 1.4) I present the scope of the work as well as its limitations.

1.1 Purpose of the master thesis

It is very tempting for many companies to move their services into the cloud. They are advertised with rapid delivery and low costs of such a solution. However, in order an application to be cost-efficient it needs to manage its resources carefully. Development of systems working in cloud is exposed at risks. Over- and under-provisioning can lead to the revenue loss. There exists the need for solving these problems. As the trend of moving toward the cloud continuous the need for new system models becomes more and more important. In the nearest future these models can be of great value for many IT business companies.

We are still in early years of cloud computing development. Therefore, there are still a lot of topics that need a research. One of recent solutions for over- and underutilization problems may be a tenant-based resource allocation model (TBRAM) for SaaS applications. That solution was introduced and tested with regard to CPU and memory utilization by authors of [10]. They proved the validity of TBRAM by reduction of used server-hours as well as improving the resources utilization. However, the authors deployed their solution into a private cloud, which can only imitate public cloud environment. They tested cases with incremental and peak workload of the system. In this thesis paper I want to check whether the TBRAM is really worth to adhere. Examining that system in public and commercial cloud environment could deliver the answer for that question. Therefore, the main aim of the thesis is further TBRAM approach validation, as it was proposed in future research part of the base work [10]. If the results of the thesis will confirm usefulness of the model then it could be considered as the solution for previous mentioned provisioning problems.

The Cloud Computing is very popular topic in recent years. It can revolutionize whole IT market and that opportunity fascinates me. Maybe soon the SaaS model will become dominate software delivery paradigm, so I would like to understand it better.

I think that cost-effectiveness is very important factor when designing an SaaS application. That is because it focuses on creating applications that use their resources
efficiently which straightforwardly implies lower costs of running it [10]. I was always a big fan of software optimization. However, I often heard that it is pointless because hardware is becoming faster and faster so rapidly. I could understand their point that optimization is time consuming, error-prone and that a premature optimization is the *pure evil*. But, in the same time, I couldn't stand the waste of computing power and a thought, that something could run much faster with just a little more effort. I hope that SaaS paradigm and development for cloud environments would change a little our look at optimization. Ironically, the main force that prohibited optimization – money – now can become the leading force towards it.

### 1.2 Aim and objectives

In this work, I will built my authorial SaaS system in accordance to the TBRAM. Then I will deploy it into public and commercial Amazon's cloud environment. Then I will examine the model's influence on the system. The cost will be calculated based not only on used server-hours but also based on an actual billing statement. It will show not only more accurate model's validity estimations, but will also provide real cost information. It will hopefully show whether the tenant-based resource allocation model can improve cost-effectiveness of SaaS applications. What is more, my own load-balancing solution will be compared against the Amazon Elastic Load Balancer (ELB). Possibly, the cloud provider's load-balancing service will be sufficient or more cost-effective.

Within this research I want to find out if the TBRAM influence my authorial SaaS system. I want to see how it will tackle over- and underutilization problems when deployed into public cloud. I also want to check how that model affects the costs of running my system. Finally I want to know if the TBRAM can also improve cost-effectiveness of other SaaS cloud systems. Therefore, the main aim of this master thesis is:

**Examination of tenant-based resource allocation model's influence on cost-effectiveness of SaaS applications running in a public cloud.**
Objectives:

1. Comparison of cost-effectiveness between tenant-based (TBRAM) and tenant-unaware resource allocation models of SaaS applications.
2. Detecting possible influence of tenant-based resource allocation model on cost-effectiveness of tested SaaS applications.
3. Examining that influence in relation to tenant-unaware resource allocation model of SaaS applications.

1.3 Research questions / Hypotheses

The following research questions were stated to be answered by this research. Apart from them I stated also two mutually excluding hypothesis to verify.

Research questions:

RQ1: Does the tenant-based allocation model influence cost-effectiveness of SaaS applications?

RQ2: Does the tenant-based allocation model improve my authorial SaaS application?

RQ3: Can tenant-based resource allocation model improve cost-effectiveness of SaaS applications running in the public Cloud?

Null Hypothesis:

• Tenant-based allocation model does not influence costs of running SaaS applications in the cloud.

Alternative Hypothesis:

• Tenant-based allocation model can decrease costs of running SaaS applications in the cloud.

1.4 Limitations

This work is not focused on comparison between various public cloud infrastructures. Proposed solutions are deployed only on Amazon Web Services (AWS) cloud. In my opinion, deployment of a solution (based on above mentioned resource allocation model) on just one cloud environment is sufficient. It allows to check the model in practice and since the model does not contain any platform specific features, other cloud environments should not affect it dramatically. The AWS platform was
chosen by me because of rich monitoring services and low level access (Infrastructure-as-a-Service layer). These features are important in order to conduct proposed tests. Not without meaning is also the leading position of the Amazon company as a pioneer cloud computing provider. What is more, deploying the applications on other public clouds will significantly exceed my time and cost boundaries.

In this work I am implementing only several types of SaaS applications just to give a general overview. It is not the point of this thesis to test the model over all possible types of SaaS applications. The SaaS applications I chose are common for CRM or ERP applications, which in turn are very popular among existing SaaS solutions in cloud. Therefore, I think that implemented applications and obtained results can be representative.

Also, it is not the point of this thesis to compare cloud environment with traditional web hosting solutions. Although it is possible to deploy web pages and web applications in the cloud, I do not treat the cloud environment just like another web hosting service. I use cloud specific features like auto-scaling and virtual machines management. There are also many specific factors related to traditional web hosting that I am not using. That is why it would be hard to compare these two hosting environments. For that reason the comparison is out of the scope of this paper.
2 BACKGROUND

In this chapter I describe what the cloud computing actually is and what is my understanding of it (2.1). Then I explain the basics of Software-as-a-Service model and its relation to the cloud (2.2). The next section (2.3) treats about Java programming platform which I chose to implement the SaaS system. The two following sections (2.4, 2.5) describes problems with cloud economical model and multi-tenancy. The last section in this chapter (2.6) presents related works.

2.1 Cloud computing

Cloud computing has gained more and more attention recently. It refers to technology that enables virtualization of resources such as storage, processing and network bandwidth. It also describes on-demand Internet applications running as a services on that infrastructure. These applications are usually paid-per-use [3]. Every respecting IT-company started to think about providing its services in the cloud [12]. The reason for that is economical mainly: running an application on elastic, scalable cloud allow us to pay only for resources we are using at the moment. We don't have to pay for provisioning like in traditional hosting services [3]. The cloud is often viewed as a three-layer stack (Figure 1). It consists of Infrastructure-as-a-Service (IaaS), based on which Platform-as-a-Service (PaaS) is build and the Software-as-a-Service (SaaS) on top of that. SaaS refers to software delivery paradigm and it will be described in more details in the next section (2.2). IaaS represents the hardware virtualization and PaaS represents tools and APIs to build applications upon that.

![Cloud stack diagram]

Figure 1: Cloud stack
Cloud computing is not a new technology. It is rather mixture of technologies existing before, like: grid computing, utility computing, virtualization or autonomic computing [27]. Figure 2 presents that idea. Cloud computing performs calculations on distributed resources like in grid computing, but it is more advanced and flexible. It offers dynamic provisioning and sharing in hardware and software layers. Cloud computing also adopts utility computing business model with on-demand resources and utility based pricing. Cloud leverage virtualization of resources to achieve high abstraction levels and uses virtual machines (VM) to perform the tasks on. Finally, we can observe autonomic behaviour in the cloud in automatic scaling, but in overall it is still not completely autonomic. Many events still require human intervention.

![Diagram of cloud computing origins](image)

### Figure 2: Origins of cloud computing

#### 2.2 Software-as-a-Service

Software-as-a-Service (SaaS) is software delivery model in which entire application (software and data) is hosted in one place (usually in the cloud) [24]. The SaaS application is typically accessed by the users via a web browser. It is the top layer in cloud computing stack. SaaS evolved from SOA (Service Oriented Architecture) and manages applications running in the cloud. It is also seen as a model that extends the idea of Application Service Providers (ASP) [23]. ASP is primary centralized computing model from 1990's. SaaS platform can be characterized by: service provider development, Internet accessibility, off-premises hosting and pay-as-you-go pricing [9]. The SaaS platform supports hosting for many application providers. As oppose to ASP
model, SaaS provides fine-grained usage monitoring and metrics [16]. It allows tenants to pay accordingly to the usage of cloud resources. SaaS applications often conforms to multi-tenant architecture, which allows a single instance of a program to be used by many tenants (subscribers) [10]. That architecture also helps to serve more users because of more efficient resource management than in multiple instances approach [11].

2.3 Java Enterprise Edition

Java Enterprise Edition (J2EE) was extensively used to implement my authorial SaaS system. It supports development of enterprise scale applications including web services. J2EE is based on Java Standard Edition (Java SE) which is one of the leading, general purpose programming platform for many years. In J2EE software is written Java programming language and then configured using XML files. It is a general rule not only for pure J2EE components, but for many frameworks and technologies based on it.

I have chosen Java as a programming language platform for many reasons. Firstly, I have a good experience in programming in Java SE and Java EE. Secondly, thanks to its popularity, Java has great support for developers. It can help a lot when encountering problems with the technology. Because of its system independence, Java is supported by all main cloud infrastructure providers (even by Microsoft Azure). Many web frameworks supporting Plain Old Java Objects (POJO's) also exist like: Spring, Struts, Google Web Toolkit and many more. That feature makes an integration processes much easier. It is also worth to mention that great majority of Java related technologies and frameworks are tested in many enterprise applications and they are free of charge.

2.4 Cloud economy

Despite that in the cloud we can automatically receive on-demand resources we can still encounter problems related to inappropriate resource pool at the time. These are over- an underutilization which exists because of not fully elastic pay-per-use model used nowadays [21]. Over provisioning exhibits when, after receiving additional resources (in reply for peak loads), we keep them even if they are not needed any more. Thus we are affected from underutilization. Under provisioning (saturation) exhibits when we cannot deliver required level of service because of insufficient performance. This is also known as an overutilization. It leads to the customers turnover and revenue
losses [3]. For example Amazon Elastic Cloud Computing (EC2) service charge users for every partial hour they reserve each EC2 node. Paying for server-hours is common among cloud providers. That is why it is very important to utilize fully given resources in order to really pay just for what we use.

2.5 Multi-tenancy

We are still in early stages of cloud computing development. We cannot expect cost-effective pay-per-use model for SaaS applications after just deploying it in the cloud. What is more, automatic cloud scalability will not work efficiently that way [26]. To achieve desired scalability we need to design our SaaS application with that in mind. In order to do that, the application must be aware how it is used [15]. We can use multi-tenant architecture to manage the application behaviour. It allows to use a single instance of the program by many users. It works in similar way like a singleton class in object programming languages, which can supervise creation and life cycle of objects derived from that class. Supporting multiple users is very important design step for SaaS applications [6]. We distinguish two kinds of multi-tenancy patterns: multiple instances (every tenant has got its own instance running on shared resources) and native multi-tenancy (single instance running on distributed resources) [2, 6]. First pattern scale well for small number of users, but if there are more than hundreds it is better to use the second pattern.

2.6 Related work

Authors in [6] propose profiles approach to scaling in the cloud. They try to use best practices and their knowledge in order to create scalable profiles. The profile contains information that help to characterize a server in terms of its capabilities. When the scaling activity is fired it takes the profile information into account. In [11] authors propose a toolkit using Java mechanism to support multi-tenancy. They use context elements to track applications running on Java Virtual Machine. That in turn allows to distinguish each tenant. That information can be later used in order to estimate given tenant's resource usage. The tenant context can also be used for billing each tenant's activities. In [7] authors consider an intelligent resource mapping as well as an efficient VM management. This is a very important problem that greatly influence costs of running applications in cloud. In [12] authors describes components which influence
virtual machines performance. These are: measurement, modelling and resource management. They introduce a decomposition model for estimating potential performance loss while consolidating VMs. Amazon propose its Auto Scaling tool [1] to manage VM instances using predefined or user-defined triggers. It is the Amazon EC2 platform specific mechanism based on resource utilization. In this work I used the above mentioned works' findings to develop my own SaaS system.

In [10] authors implements a tenant-based resource allocation model for their SaaS application deployed in private Eucalyptus cloud. The authors performed tests with incremental and peak workload simulation. In the research they achieved significant reduce of server-hours compared to traditional resource scaling model. The tenant-based model improved also utilization of cloud resources by their SaaS system. Moreover, they introduce formal measures for under and over provisioning of virtual resources. The measures are designed specifically for SaaS applications with respect to CPU and memory utilization. In this work I am going to implement SaaS system based on their design and deploy it into the public AWS cloud environment. I will refer to their work as the *base paper* in the remaining parts of this thesis.
3 RESEARCH METHODOLOGY

To answer the research questions contained in this thesis an appropriate research was done. This chapter describes the way the research was planned and conducted. Section (3.1) presents proposed methodology. Section (3.2) focuses on used measures, which are: over- and underutilization, as well as the financial cost. In the last section (3.3) I describe the way the workload for the tests was generated.

3.1 Methodology

In order to answer the research questions I implemented an authorial multi-tenant SaaS system that benefits from J2EE capabilities. Then I deployed the system into the public AWS cloud. After that I used it as a test bed to conduct my experiments. Therefore, the used method is a quantitative research – experiment conducted according to methodology from the base paper [10].

Currently there are many cloud services providers. The main are: Amazon's AWS, Google Apps, Microsoft Azure and Salesforce.com. They have different characteristics and unfortunately they are incompatible with each other. The differences appear between available programming environments, operating systems or storage technologies [27]. Perhaps, even more important are the differences between offered resources, pricing and available cloud layers to use. According to [27] Windows Azure and Google Apps offer PaaS services layer and Salesforce.com offers SaaS (the highest layer). After examining each cloud providers’ offer I decided to use Amazon services because they are the only IaaS provider between the above mentioned. The infrastructure layer gives access to resources information required by this research. Amazon also offers free monitoring services and automatic scalability tools. Monitoring of VM statistics is the most important feature for me since I was going to test various factors with my SaaS system. Another advantage is support for J2EE technologies with which I feel the best as a programmer. There are also many free tools for Java platform to monitor utilization, like SIGAR and JMeter.

The next step was testing the cost-effectiveness of the SaaS applications. I decided to use a testing framework proposed in [10]. According to it I developed two versions of the SaaS system. First was using traditional resources scaling (based on number of users) with round-robin load balancing. The second one used proposed tenant-based
resource allocation system (TBRAM). I also used the Amazon CloudWatch cloud metering service for gathering required statistics. The CloudWatch was collecting predefined metrics as well as my own ones. For the cost calculation I simply used Amazon's on-line billing information about my cloud account. It provides actual and final costs of running users cloud applications.

When the system was already deployed into the cloud I conducted a series of tests following the testing framework. The tests were about generating workload for the system and observing its behaviour. Therefore, the independent variable in that test was the type of workload and number of simulated users. The dependent variables was CPU and memory consumption, number of used server-hours (in case of TBRAM system) and the cost of the cloud service.

After receiving the results I performed a statistical analysis to determine the answer for the main research question. The analysis also helped to generalize the results for other SaaS application of similar type.

Below is the list of methods I used:

• Implementation of tenant-unaware and then tenant-based resource allocation SaaS systems (with respect to tenant-based isolation, VM allocation and load-balancing).
• Deploying the systems in the public cloud (AWS).
• Testing the cost-effectiveness of the applications under certain workload.
• Statistical analysis of the results.
• Generalization of the results for the application type.

3.2 Measures

One of the most important factors that affects cost of a cloud solution is utilization of resources. Since in cloud computing we pay for what we use, it is vitally important to use given resources efficiently. Therefore measuring over- and underutilization provides a good view at cost-effectiveness of my system working in cloud. Additionally, I think that using billing information from Amazon can greatly improve the assessment of the costs. That is because it is not just another metric, but an actual financial cost that includes charges for all used cloud resources.

There are many performance characteristics that we can consider in SaaS system, like: CPU usage, memory consumption, bandwidth usage, number of I/O disk
operations, response time and others. The choice depends on the type of application we are about to test as well as on the characteristics we are interested to examine. To be consistent with the base article [10] I choose CPU usage and RAM memory consumption to determine virtual machines utilization. Nevertheless, I think these two metrics could be the most relevant in my case. The CPU usage rate can deliver better estimation of virtual machine utilization when combined with RAM memory consumption. We can imagine a case when the usage of CPU is low but because of the lack of memory we suffer from poor performance. That resource consumption model fits well to applications which performance is most dependent from calculation power and available memory. Also, these applications are not strongly database or network bandwidth centric. This kind of SaaS applications were implemented in this work.

3.2.1 Overutilization

The term “point of exhaustion” is often used in relation to overutilization. It is described as a point when some resource if fully utilized, for example 100 % CPU usage or all memory is consumed [20]. This definition tends to be accurate in many simple cases. However, in case of cloud computing that definition seem to be an oversimplification. Authors of [16] propose another definition, according to which the point of exhaustion is a maximal payload that can be assigned to a single virtual machine without decreasing its throughput in the same time. Readings above the exhaustion point describe a saturated machine. Thanks to that we can visualize an overutilization on a diagram (Figure 3). This new exhaustion point definition requires to measure a VM throughput together with the resource utilization (CPU or memory). My SaaS system uses the JMeter tool combined with CloudWatch network related metrics in order to calculate that. The system is stressed with HTTP requests generated by the tool. Then it calculates the throughput of the system by dividing the number of HTTP request by the time from start of the first request to the end of the last one. By measuring it this way we can include all the processing time between the requests as well. In previous works [16, 17, 19, 20, 22, 25] authors focused on throughput to discover inflection points. Whenever the throughput was dropping while the VM utilization was rising an inflection point was found. In my work I used the same approach.

My SaaS system used above mentioned inflection points to detect an overutilization of given virtual machine. All the VMs with Tomcat are monitored by gathering resource utilization metrics and throughput. The metrics I use are CPU usage and Java virtual
machine heap memory consumption. Based on that measures I can tell with good accuracy weather a VM is saturated in given moment or not.

![Figure 3: Inflection points detection](image)

Generally, when the VM is saturated the operating system processes start to use more and more resources making user's processes execution even slower. It has a negative influence on system responsiveness and therefore, on user experience. That is why it is always a good idea to avoid saturation. Even despite it does not have a direct influence on cost (we do not pay extra for high VM usage rate), it can drive to users turnover because of a poor performance.

### 3.2.2 Underutilization

From the economical point of view underutilization is just a waste of money. It means that we pay for something we do not need or not even use. From the end user perspective it is hardly noticeable, so from the provider perspective these extra money are spent almost on nothing. More formally, from the definition [5, 8] an underutilization describes a situation when some of the cloud resources are not being used by the working virtual machine. Of course it is almost impossible to assure 100% resource usage all the time so some kind of underutilization is inevitable.

Underutilization in a cloud can be measured by the amount of the resources available for use. According to [5] resource is wasted when we can reallocate given resource utilization into another VM without exceeding its maximal quantity allowed. It means that for example: the payload for two VMs could be easily allocated just in one VM making the other VM unused. In order to check if given VM can be allocated to
another one we need to calculate combinations of VMs according to some resource. One way to solve that problem is by using the knapsack algorithm. As proposed in the base article, I assign amount of used resources to the knapsack items' weights. As a value of an item I take available quantity of the resource in other VMs. In case of Java heap memory the item's value equals an amount of heap memory that still can be used (available memory). Thus, the most valuable items are the less used ones. The capacity of the knapsack is the amount of available resource of a VM we try to assign the workload to. By using this approach we obtain the maximum number of VM than can be potentially released. That VM number is used to measure underutilization. The lower that number is the better the resources are used.

3.2.3 Cost

Running the system in a public cloud gives us yet another way to assess the cost-effectiveness. Almost every action made in cloud is registered and added to our bill. We pay for sent Internet requests, storage, VM hours and many more. Therefore, the billing statement yields arguably the most accurate estimation of cost-effectiveness. At the end of the day it is the price we need to pay for our cloud service.

During the tests the Amazon CloudWatch service was collecting metrics about the cloud environment usage. Both SaaS systems (Base System and TBRAM) are tested against the same test plan (Section 5.2), so the requests number is exactly the same. The main difference between them can occur in the number of used virtual machines. That difference should be reflected on the bill statement. The comparison of costs of running the SaaS systems will show if there is any economical improvement with using the TBRAM approach over the traditional resource scaling approach.

3.3 Workload generation

In order to stress the SaaS systems during the tests I needed to generate workload for the servers. To do that I used a cluster of JMeter machines. Each of them could simulate hundreds of simultaneous users. The number of users varied across the simulation period. Similarly to the base article I decided to generate two types of workload: incremental and peak-based. In the incremental case the workload starts from zero and then incrementally rises up to the maximum level for given time period. It can simulate very steady and linear increase of tested SaaS system usage. The second case is much harder to efficiently provision. In the peak-based workload generation the number of
users starts always from zero for any given time period. Then in the middle it reaches the peak workload and starting to decrease back to zero till the end of the period. That can simulate very unequal load of the SaaS system's servers. In that case both scaling-up and scaling-down appears. It can represent the service which is heavily loaded at one part of the day and almost not used at others. Figure 4 shows the way of workload generation. We can see that each test consisted of three parts.

![Figure 4: Workload simulation](image)

The actual number of users to simulate depends on the type of tested servers. For the test purposes small Amazon virtual machine instances were used. According to [13, 25, 28] that type of server can handle 100 simultaneous users as maximum. That seemed to hold when I was testing the application in a local network. However, after running preliminary tests in cloud environment I established that small EC2 instances (m1.small) with my SaaS application reached their optimal performance for 50 simultaneous users. Therefore, each VM instance of the SaaS system was stressed with the workload up to the 50 concurrent users. Since the application was deployed within Tomcat container I could set the `maxThreads` parameter using the container's configuration file. The value of that parameter was left set to default 200. This is the maximum number of concurrent threads created by the container to answer workload demands. That number does not include other container's threads created when Tomcat starts. Table 1 shows in detail how many VM instances were used during the tests. The number of VM instances in the
table is relevant only to test of base SaaS system, when the instances number was set manually. For the tenant-aware SaaS system (TBRAM) that number was tuned dynamically by the SaaS system.

The ultimate goal of the simulation would be to run the test for one year. Then I could offer very accurate approximation of real life usage. Unfortunately, it would be too long for the time frames given for this thesis. The cost of such the simulation would also exceed many times my acquired funds. That is why I decided to run the test significantly shorter. The simulated period remained a full year. Since Amazon CloudWatch collects the metrics every minute I decided to simulate six hours with that time (1 min = 6 simulated hours). Using this scale, one day is 4 min and one month (30 days) is 4 * 30 = 120 min. Thus, to simulate an entire year I needed 24 hours of tests. Although, that time frames might seem a little bit short, in practice that is enough to simulate certain cloud specific phenomenons. In AWS cloud environment starting new EC2 instance is just the matter of several minutes counting from time of sending the request till the end of tomcat start-up procedure. Thus, we can have a new fully operational VM within minutes. The creation time depends of course from many things like current cloud utilization, chosen operating system or type and number of services we want to start during the booting process. In my case the whole VM creation procedure rarely exceeded 5 minutes and in case of SaaS platforms oscillated around 3 minutes. Thanks to that we can change the size of our instances fleet very dynamically. The entire test procedure was repeated for each workload generation type and for each SaaS system giving total tests duration of 96 hours.

Table 1: Simulation instances number

<table>
<thead>
<tr>
<th>Period</th>
<th>Simulation hrs</th>
<th>VM instances</th>
<th>Peak users number</th>
</tr>
</thead>
<tbody>
<tr>
<td>January – April</td>
<td>8</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>May – August</td>
<td>8</td>
<td>4</td>
<td>200</td>
</tr>
<tr>
<td>September – Dec.</td>
<td>8</td>
<td>8</td>
<td>400</td>
</tr>
</tbody>
</table>


4 SYSTEM DESIGN

In this chapter I describe a design of implemented SaaS systems. First of them is tenant-unaware base SaaS system (Chapter 4.1). Then I describe SaaS system conforming to tenant-based resource allocation model (Chapter 4.2). In the following chapters I give more details about used technologies (4.3, 4.4). I spend whole section to present the AWS cloud.

4.1 Base system

In this section I am describing the base tenant-unaware resource allocation SaaS system (Base System). It conforms to traditional approach to scaling resources in a cloud and is based on number of users of the system. Load-balancing technique leverage a simple but fast and even round-robin algorithm offered by Apache HTTP. In my work it substituted by Elastic Load Balancer service. According to [4] the ELB sends special requests to balancing domain's instances to check their statuses (health check). Then it round-robins among healthy instances with the less number of outstanding requests. In the process it does not take into consideration the instances resource usage of any kind. Although the name of this system suggest lack of awareness of tenants it concerns only resource allocation. The system was build according to Service Oriented Architecture (SOA) and native multi-tenancy pattern. First, it was implemented as a set of J2EE web applications using Spring and Struts frameworks. Several parts of the system were later transformed to web services using WSO2 and Axis2. Deploying application as a web service makes it independent from running platform. It also gives more flexibility with accessing the application.

To present the idea behind the system, I am going to show the whole test bed architecture first. It should provide a general overview of the system from high level perspective. It is shown in Figure 5.

In the figure we can see a group of Amazon EC2 (tomcat) instances. The number of the instances varies and it depends on number of simulated users. Each of the instances consists of VM with one Tomcat web container. In each Tomcat container my SaaS platform is deployed. The platform is the main part of the system as it makes foundries for SaaS applications. It also includes web services and common libraries.
The database is deployed in another VM running on Amazon Relational Database Service (RDS) instance. It is specially preconfigured machine to support relational databases. I chose MySQL Database Management Service (DBMS) among others, because it is free to use and sufficient for the test purposes. All the EC2 instances communicates with one database server on RDS. The database itself is not the subject of this research and it was configured to avoid being a bottleneck of the SaaS system.

To perform the test I used a set of JMeter machines. These are independent VMs containing Apache JMeter application designed for workload testing. That open source tool can be used to simulate heavy concurrent load over J2EE applications. It provides tools for analysing performance using graphical charts with performance metrics. JMeter was configured to create distributed requests in order to simulate workload usage. In distributed mode one VM is set to master and other VMs are set to slaves. The master was running the GUI version of the application while the slaves were started in server mode. JMeter machines were sending HTTP requests to the ELB. It is a load-balancing service provided by Amazon to work with cloud clusters. It can dynamically update its balancing domain and seamlessly scale to meet current demands. The load

Figure 5: Test bed architecture
balancer redirects requests to certain EC2 Tomcat instances. All the results are collected by the master JMeter machine.

During all the tests Amazon Cloud Watch service collected required data from the system components. It gathered information about services throughput from the load balancer. From EC2 instances it collected RAM and CPU consumption metrics. Finally, it monitored the database input/output operations. At this point it is worth to mention that not all the data were collected directly. Some statistics, like RAM consumption, are not available in CloudWatch. I used my own Resource Consumption Monitor to overcome that problem, which I describe later. All the statistics eventually were send to a desktop PC for further analysis.

Therefore, all the testing process was done in the cloud. It gave me many advantages. Using cluster of JMeter machines helped to simulate real workload, which could be troublesome to achieve without the cloud. Since all communication happened within one cloud, I was not charged for incoming/outgoing requests. What is more, I did not have to block my own workstation, so it can be used for other purposes.

4.1.1 SaaS platform

![SaaS platform architecture](image)

The SaaS platform (Figure 6) was the main part of my SaaS system. The task of the platform was to support deployment of plain web applications as a SaaS services. The idea behind the design of this part was inspired by the base article. Their SaaS platform was developed as a part of the Rapid Product Realization for Developing Markets Using Emerging Technologies research at Technologico de Monterrey University, Mexico.
Since I did not have such means and so much time, I made my SaaS platform much simpler and focused only at usability for my SaaS system.

The entry point to the platform was the Core Web App (CWA). As the name suggests, it was a web application that works as a gate. All interactions between outside environment and the parts of the platform were done through that element. Behind it there were applications responsible for users authorization, account management and logging. The platform contained also common Java libraries used by deployed applications. Configuration was made by the XML or plain text files. The platform exhibited web service interfaces to be consumed by outside applications. One example of such an interface was the interface for metering services. It allowed to monitor the usage of resources by the platform. That behaviour is depicted by the SOA element in Figure 6. On top of that all we can see SaaS applications. These were developed as normal Java web applications, but when deployed on the platform they gain an extra SaaS functionality. I implemented two of them: Sales application and Contacts Application. I think that two applications are enough to present the platform's functionality as well as the interactions between deployed applications. On more feature,
which is not showed in Figure 6, was the communication between the platform and external database server.

4.1.2 Database schema

The multi-tenant SaaS database was used by the SaaS platform applications. In case of TBRAM system it was also used by the main component – *SaaS Core Web App* (SCWA) - explained later in Section 4.2. The database consisted of information about tenants, sales and contacts. All the data were stored on external database server in the cloud.

As we can see in Figure 8 the database schema is very simple. It was designed just as data back-end for the system. It was not an aim of this work to test the database behaviour of any kind. No complex operations were performed over the tables. These were mostly simple insert, update, select and delete SQL operations. The database was accessed through the Hibernate framework to isolate the applications from used database engine. Exactly the same database was used throughout all the test. Before each test the database was reset to its initial state using a SQL script populating the tables.

![Figure 8: Multi-tenant SaaS database schema](image)

4.1.3 Resource Monitor

As described in previous chapter, to measure over- and underutilization I needed certain metrics from running VMs. Some of them were available directly through Amazon CloudWatch metering service. These were CPU usage, network in/out and number of requests per second in case of the ELB. The letter metric can be used to
calculate the throughput of the entire system. Other source of data for that metric came from JMeter tool. However, monitoring of RAM consumption and number of running threads is not provided by the CloudWatch. That is why I needed my own monitoring solution that sat between the monitoring domain and the CloudWatch. That was the Resource Consumption Monitor (RCM) service.

There exist two main approaches to the monitoring problem. First is a distributed approach. It is similar to Observer Pattern [14] known from object oriented programming patterns and best practices. In that case monitored VMs register themselves to the monitoring service and then publish metrics. The beauty and simplicity of the design is its main strength. It has however one big disadvantage from my point of view – each worker VM needs to be aware of the monitoring process. It is also hard to quickly notice a VM termination due to unexpected events or errors. That is why I chose the second, centralized approach. In that case the VMs with SaaS platforms are unaware of being monitored as it is beyond their consideration. The RCM was constantly monitoring the state of VMs by polling AWS cloud. After each interval (Polling interval) it collected the metrics from the monitored domain. After another interval (Publishing interval) it published gathered so far data to the CloudWatch service. Thanks to that all the VM's metrics were available in one place.

The RCM was a web application, but it could also be used as a standalone console Java application packed into an archive file (jar). It used the Java Management Extension (JMX) RMI based protocol which allows to request information about running Java Virtual Machine. I could (and I did) use my own web services to fulfil the same tasks, but since the entire SaaS system was based on Java I did not really need the flexibility that web services offer. Especially, that this flexibility comes with its price. First of all, the JMX packets are much smaller that competitive SOAP protocol's. Therefore it reduces network traffic and time needed to decode the packet. The next reason is a requirement for management of web services like Axis2. The JMX is built in Java Runtime Environment I used (JRE 1.7). Finally, the JMX technology is far more robust and advanced that I could ever make my web service within given time frames. It is also transparent for applications running on JVM. All we need to do is to add extra running parameters when starting the JVM.

To run the RCM I needed to set some parameters first. One of them was the running mode which tells the monitor whether to run in test mode (very frequent metrics
collecting, but without publishing them to the CloudWatch) or in *normal mode* (with synchronization to the CloudWatch). The chosen mode affected both (polling and publishing) intervals. In *test mode* the data were gathered every 5 seconds. In *normal mode* polling was set to 10 seconds and publishing to 1 minute. That setting matched the settings of CloudWatch service working in detailed mode. Using RCM I could also start or stop manually the monitoring of certain VM. To tell the RCM which instances to monitor I added special tag to VMs. The tagging is a feature in AWS cloud that helps to organize running instances. The most common usage of tags is for giving names to the instances which are often more meaningful than their ids.

Because I was sending my own metrics to the CloudWatch it was crucial that all the metrics (direct and my own) for given VM were taken in the same time. They could be published asynchronously, because they contained a timestamp tag. However, the measurement data itself needed to be synchronized in time. Otherwise they would be invalid. To avoid that I implemented a synchronization mechanism in the RCM. It was assuring that my own metering data are collected at the same moment as the direct CloudWatch ones.

The RCM was deployed on dedicated t1.micro EC2 instance, because it could not affect the work of virtual machines it monitored. Thanks to its web interface it could be managed from any computer via a web browser.

### 4.2 Tenant-aware system

The base SaaS system was implemented as a reference tenant-unaware resource allocation system. The main flaw of its design was rigid management of VM instances in the cloud. Thus, it could lead to serious over- and underutilization problems. Actually, we can confirm that by looking at the results chapter in this paper. One of the ways to tackle these issues was proposed by authors of the base article. It is called a tenant-based resource allocation model (TBRAM) for scaling SaaS applications over cloud infrastructures. By minimizing utilization problems it should hopefully decrease also the final cost of running the system in the cloud.

The TBRAM consists of three approaches that leverage multi-tenancy to achieve its goals. The first of them is tenant-based isolation, which separates contexts for different tenants. It was implemented with tenant-based authentication and data persistence as a part of the SaaS platform (Tomcat instances). That approach is described in section
The second way is to use tenant-based VM allocation (4.2.2). With that approach I was able to calculate the actual number of needed VMs by each tenant in given moment. The last but not least is the tenant-based load balancing (4.2.3). It allows to distribute virtual machines’ load with respect to certain tenant. An overview of the architecture is presented in Figure 7 below. The red dashed line in the picture denotes communication to web services. We can notice that the SCWA element in the Figure was the only change made to the original test bed. That element embraced proposed TBRAM approach.

4.2.1 Tenant-based isolation

To assure that system worked properly it needed to isolate one tenant from another. A situation when one tenant can access and affect data that do not belong to him/her is unacceptable in any commercial solution. The TBRAM approach proposes low level isolation as it improves its scalability [2]. The tenant-based isolation of TBRAM could be split into two implementations. One was based on data persistence and the other one was based on authentication mechanisms. In this place is worth to mention that tenant-based isolation was also used in the Base System. That was because both systems were using the same multi-tenant database. What is more that technique was practically affecting only the SaaS platform, so it is isolated from the SCWA concept. Thanks to that in both systems the SaaS platform was exactly the same, thus minimizing its influence on the results.

In the persistence layer the authors propose Shared Database – Shared Schema as it has the lowest hardware cost and the largest number of tenants per server [18]. To logically separate data the Tenant ID field is used for each database table. From technical point of view I used JoSQL libraries which let to perform SQL-like queries over Java collections. That libraries were used by Struts2 interceptors to achieve multi-tenant preprocessing. Java annotations were used to mark the places in code that needed this kind of tenant-based behaviour. That was arguably the most efficient way to implement multi-tenancy since the data were first fetched and then filtered. It could be achieved using SQL selection mechanisms. Interceptors as an implementation of aspect oriented programming postulates had many advantages as well. The main was that all the code was in one place but could affect any class marked with the annotation. Secondly, that annotation was the only change that needed to be make to an existing
application code to enable multi-tenancy. Therefore it could possibly be the most common way to add that tenant layer to existing applications.

Tenant-based authentication was the second concept used to achieve tenants isolation. As proposed by TBRAM it should be implemented into the core application of the system which is SCWA. During the authentication every user was linked to its Tenant ID. From now on the user could access only the data the certain tenant has rights to. Needless to say that she/he could not access any data before the authentication. The TBRAM also suggest to use an Access Control Lists (ACL), which I decided to omit as it introduces just unneeded complication for me. I decided to give full access to all SaaS applications to all users for simplicity. It was necessary to receive the tenant information from any point in the SaaS system. The TBRAM proposes a mechanism based on cookies and the servlet context. The authors [10] used a local Tomcat cluster to deploy their solution. In my case the SaaS system was deployed into the Amazon cloud infrastructure and that solution did not worked for me. I decided not to use Tomcat instances running in cluster mode. That was because I was worried about an overhead related to sharing session information between all cluster's nodes. If the nodes are running in different networks I think that can introduce non negligible influence. Currently, Tomcat 7 version supports all-to-all session replication. I used a special web service to serve the tenant information instead. It was more platform independent and it could work in both cases. Whenever any user was accessing given VM for the first time, the SaaS platform was checking in SCWA if that user was authenticated and authorized to do that. If it was, then its specific data ware saved locally in a session context so the next request from that user didn't require further communication to the SCWA's centralized web service. Therefore, only VMs that needed that specific information ware acquiring it. There are also other methods of session replication like session persistence (shared file system or database) which are outside the thesis scope.

According to TBRAM a Tenant Context object was conceptualized. It contained information about tenant ID, active users and their VM assignations etc. A Tenant Context Manager object in turn was used to manage all the underlying Tenant Context objects. Thanks to that an information about the tenant's state was available to all other services. The Tenant Context allowed to isolate each request sent to the platform based on given user's tenant information. The following Figure 9 shows the idea. We can see several users from two different tenants (subscribers). Despite they physically share the
same SaaS applications and the persistence layer they are still logically isolated by their tenant contexts. These context objects help to achieve native multi-tenancy of the applications. The users have no idea they are sharing the same resources.

4.2.2 Tenant-based VM allocation

Tenant-based VM allocation was used to determine the number of VM instances needed for a given tenant in a given moment. It combined the concept of profile approach with monitoring services implemented within the SaaS system. A profile used for the test bed in the base paper was a small virtual machine profile. It was meant to substitute the \textit{m1.small} EC2 instance in Amazon cloud. The profile was as follows: 1 CPU core, 1 GB of RAM, 800 MB to the JVM heap memory and 100 as the number of users the VM can handle \textit{(maxThreads = 200)}. In this work I used a similar profile since the SaaS

![Tenant context concept](image-url)
platform was deployed in actual *m1.small* instance. The main difference was the maximal number of users set to 50 in my case. This profile information together with current readings from metering services were used to calculate required number of VM instances.

The *Tenant Context Manager* was responsible for assigning the weights to each *Tenant Context*. These weights were later used for VM calculations. The TBRAM proposes the following formula:

\[
\text{Tenant Context weight} = \text{active users} \times \left( \frac{\text{heap size}}{\text{maxThreads}} \right)
\]

where *active users* are those whose session has not expired. *Heap size* is the amount of memory assigned to JVM (set in profile) and the *maxThreads* is a maximal allowed number of concurrent threads for the SaaS platform. The fragment in parenthesis could be treated as an average memory usage per thread for given profile. Therefore the formula above is an estimation of required memory for given number of active users.

The second formula is used to calculate the VM capacity:

\[
\text{VM capacity} = \text{heap size} - \left( \left( \frac{\text{heap size}}{\text{maxThreads}} \right) \times \text{platform threads} \right)
\]

The formula subtracts current memory usage from the maximum allowed amount described in the profile. The current memory consumption is calculated by multiplying number of SaaS platform's threads (when in idle) by the average memory per thread. From this formula we know how much memory is available solely for users of given SaaS platform. This is because from the amount of memory assigned to JVM some part is consumed by the Tomcat's and SaaS platform's threads just to start the service. All the following threads were created to serve each user. Thanks to that I was able to estimate an actual initial resources available.

The TBRAM suggest use of a knapsack algorithm to calculate the minimum number of instances needed to allocate current workload. This number was the only result yield by the algorithm since I was not interested in actual tenants assignations to available VMs. The algorithm used the values returned by the above formulas (Formula 1, 2). I used dynamic programming method to solve the knapsack problem quickly. This whole idea was conceptualized within *Tenant-Based VM Calculator*. The results of these calculations determined the number of VM instances requested from AWS cloud by the *VM Manager*. So the first source of information about needed number of instances came
from knapsack algorithm. Yet, it was not the only one. Sometimes even the most advanced estimations are inaccurate, thus leading to discrepancy between reality and its state kept by an application. That is why I decided to add also user factor. If several subsequent request dispatches failed then a new VM instance was requested from VM Manager on user's behalf.

4.2.3 Tenant-based load balancing

The last big part of TBRAM is a tenant-based load balancer. When it distributed workload among the instances it took into account the tenant aspect. Simple round-robin IP address based load balancer could spoil all the effort of the other parts of proposed model to isolate each tenant. Since users from the same tenant share certain tenant-related data it would be a good idea to dispatch their requests to the same VM (if possible). That could reduce amount of tenant data kept by each VM since some of them would be serving only a few tenants. That could also lead to better usage of servers cache mechanisms by concentrating on data that are really shared by number of tenant's users. That in turn could for example reduce a number of requests to a database engine.

The key task of the load balancer was to isolate requests from different tenants. The tenant-based load balancer worked in 7th layer of OSI model. It used information stored in the session context as well as local applications data to assign the load efficiently. The idea behind request scheduling is that requests from one tenant should be processed on the same VMs. If that was impossible, then it should at least try to limit their number so the requests were not scattered along the whole balancing domain. That could reduce context switching and allow to use previously cached data. The scheduling process used only current status data, so it belongs to dynamic load balancers family. The solution proposed by the TBRAM is based on adaptive models of load balancing.

As suggested in the base paper I made my load balancer a part of SCWA. It was a natural choice to put that element there, since all the requests came through it anyway (because of the centralized authorization service). My load balancer's design was similar to the proposed one. It consisted of five elements: Request Processor, Server Preparer, Cookie Manager, Response Parser and Tenant Request Scheduler. Each of them was responsible for specific function in processing pipeline sequence. The most important was the last part of processing assigned to Tenant Request Scheduler. The scheduling policy was saying that the subsequent requests from the same tenant should be dispatched to the same VM. If given VM was saturated, then the scheduler dispatched
the request to next available VM of that tenant. Finally, if no other VM was available, the scheduler requested a new VM from VM Manager.

The HTTP as the Internet protocol was designed to be stateless. It means that every request is independent. It starts from handshaking in order to establish a connection. Then data exchange appears for one or possibly more server's resources. After that the connection is closed. When user requests another resource the whole procedure repeats. However, there was a need to keep a track of users' actions for example to make functioning of shopping chart possible in online shops. Because of private IP addresses it was not feasible to recognize all users just based on that. This is where the session mechanism comes in with a help. In general it allows to keep user related data at the server side and therefore distinguish each unique user. It works just fine when there is only one server dealing with the given user because of limited session scope. If there are more servers that information needs to be shared somehow. One of the solutions for that problem is clustering of Tomcat servers. But even better solution is to dispatch given user's requests to a unique server as it eliminates the need of session sharing. For that purpose many available load balancers offer so called session stickiness or session affinity. This could be seen as a sort of higher layer which groups requests of given user within a session scope. When it comes to tenant-based load balancer it could be called tenant stickiness or affinity. We can imagine that as yet another layer above the session layer which groups requests from a given tenant.

4.2.4 VM Management

So far I described how the tenant-based system isolated tenants and how it balanced workload. But the system needed also a way to acquire and release cloud resources. More specifically – Amazon EC2 VM instances.

In my design I conceptualized VM Manager as a part of the system responsible for managing AWS resources. Its task was to keep the instances fleet size to match current load needs. The VM management layer consisted of two main elements: an actual manager and a cloud client. The manager was monitoring the usage of the current fleet. Based on data from Tenant-based VM Calculator as well as from users (user requested instances) it sent requests to the cloud client, which in turn communicated with the cloud through AWS Java API.

Basically there were two types of VM managers with corresponding cloud clients. This is because there are actually two ways to manage instances in the AWS cloud. First
one is the EC2 API which gives the finest granularity, but it comes with a price. We can easily create heterogeneous fleets of instances and start or stop any chosen instance. But it is like a double bladed sword. We also need to monitor the state of the fleet in case of accidents like unexpected shutdowns. It is also up to us to assure we will not send the same request to the cloud more than once. In many cases though it is preferable to choose the other way – Amazon AutoScaling (AS) service. In that approach we create homogeneous groups of instances called AutoScaling Groups. After we decide what types of instances we want to run all that left is to set how many instances we want to run. We can change that number later on to match our workload needs. The AS is then taking care of our fleet to keep its size to desired number. It can seamlessly remove unhealthy instances with new ones. When we scale down our AS group it chooses which VMs to stop. This might be seen as flaw, because there is no way to tell the AS service which instance to stop. The AS generally consider only the economical aspect of that operation. That means it shuts down the VM that running time is closer to reach a full hour. In another words, it stops the VM which used its paid hour the most. This behaviour is related to pay-per-use model when we pay for each started running hour of an instance. That mechanism helps to avoid situations when newly started or just exceeding a full hour VMs are shutdown. If we require more sophisticated mechanism we need to implement our own using EC2 API. The AS has one more interesting feature which is the cooldown period. This is a time period after the AS fleet resize action during which all subsequent request will be rejected. It can help to avoid too frequent changes to the fleet. We can set that parameter to suit our needs or alternatively we can decide not to honour it. Eventually, for the tests I chose the AS approach as it was robust and sufficient. For many simple scenarios using EC2 API would be just reimplementing the functionality that the AS already has. The AS has also a lot of other features that are outside the scope of this thesis.
4.3 Used technologies

In Table 2 below I present a summary of technologies used to implement my SaaS systems. All of them are free to use and some of them are open-source projects. As a programming language platform I used the newest version of Java virtual machine: JRE 1.7. It introduces minor security and language syntax improvements. I also needed a web container to serve web applications. I chose well known and settled Apache Tomcat in version 7, which also uses the newest Java virtual machine version. Nowadays, most of enterprise web applications are developed with use of many available web frameworks. Relying only on Java Servlets and JSP pages would be cumbersome, error prone and time consuming. The Struts 2 web framework has a very appealing feature which is interceptors support. Thanks to the interceptors stack we can easily extend web application's functionality with common behaviour. Therefore Struts 2 supports the Aspect Oriented Programming (AOP) model. I also used Spring 3 web framework, but in very limited way. Actually I used only two features of that framework. One was dependency injection (DI), which let me add dependencies via XML configuration file. The second feature was related to WSO2 web services technology. It allows to create web services directly from Spring action classes. The WSO2 uses Apache Axis 2 web services technology. This assured compatibility with my other web services which were created purely with Axis 2. To achieve multi-tenancy I used Java annotations in combination with JoSQL. The annotations are used to detect which parts of code need to be processed by Struts interceptors. I could use simple interface mechanism to achieve the same results, but annotations have more advantages. They are less intrusive to the code than an interface implementation, which in this case would have no methods. The second multi-tenancy enabler was JoSQL which allows to perform SQL-like queries over Java collections. Thanks to that I could achieve data filtration with respect to given tenant. As a persistence layer I used Hibernate 3 framework and occasionally Java Persistence API queries. The Hibernate used MySQL DBMS as a data back-end. Finally in order to communicate with Amazon AWS cloud infrastructure I used the AWS SDK for Java. I could use the JCloud framework, which can communicate with many cloud environments. However, since I focused only on Amazon cloud, using dedicated SDK let me control the infrastructure in more detail and use helpful specific features.
Amazon Web Services (AWS) is an IaaS cloud platform. It started in 2006 by serving IT infrastructure for business in form of web services. AWS is independent from programming language and operating system, like many IaaS providers. It enables seamless scalability of resources with use its data centres. Currently they are located in U.S., Ireland, Brazil, Singapore and Japan. AWS has a reach offer of products and services. The most important groups of them are:

- **Compute**
  
  The key service is Elastic Compute Cloud (EC2) which delivers computing infrastructure in the cloud. It is payed-for-use and very scalable. The scalability can be reached with use of Auto Scaling service to match the current load. There is also the Elastic Load Balancer (ELB) service to distribute network traffic across EC2 instances.

- **Networking**

  Amazon offers the Route 53 service as a highly available and scalable Domain Name System (DNS) for our applications. We can also use Virtual Private Cloud (VPC) that allows us to create private virtual network topology. It can substantially improve security of our cloud service.

### Table 2: Used technologies

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language platform</td>
<td>J2EE (JDK 1.7)</td>
</tr>
<tr>
<td>Web container</td>
<td>Apache Tomcat 7</td>
</tr>
<tr>
<td>Web framework</td>
<td>Struts 2</td>
</tr>
<tr>
<td>Web services</td>
<td>Apache Axis2</td>
</tr>
<tr>
<td>Dependency Injection (DI)</td>
<td>Spring 3</td>
</tr>
<tr>
<td>DI + web services</td>
<td>Spring 3 + WSO</td>
</tr>
<tr>
<td>Multi-tenancy layer</td>
<td>JoSQL + Java annotations</td>
</tr>
<tr>
<td>Persistence layer</td>
<td>Hibernate 3 + JPA</td>
</tr>
<tr>
<td>Cloud communication layer</td>
<td>AWS SDK for Java</td>
</tr>
</tbody>
</table>

4.4 **AWS Cloud**

Amazon Web Services (AWS) is an IaaS cloud platform. It started in 2006 by serving IT infrastructure for business in form of web services. AWS is independent from programming language and operating system, like many IaaS providers. It enables seamless scalability of resources with use its data centres. Currently they are located in U.S., Ireland, Brazil, Singapore and Japan. AWS has a reach offer of products and services. The most important groups of them are:

- **Compute**
  
  The key service is Elastic Compute Cloud (EC2) which delivers computing infrastructure in the cloud. It is payed-for-use and very scalable. The scalability can be reached with use of Auto Scaling service to match the current load. There is also the Elastic Load Balancer (ELB) service to distribute network traffic across EC2 instances.

- **Networking**

  Amazon offers the Route 53 service as a highly available and scalable Domain Name System (DNS) for our applications. We can also use Virtual Private Cloud (VPC) that allows us to create private virtual network topology. It can substantially improve security of our cloud service.
• **Storage**

There are two main services for storing our data. Simple Storage Service (S3) offers redundancy with storing any amount of data. Elastic Block Store (EBS) gives us a block level storage independent from EC2 instances. The latter one is more often used as a virtual hard disk drives system for a VM.

• **Databases**

We can store out data also in various databases. Relational Database Service (RDS) is used to host relational database in the cloud infrastructure. There are also NoSQL solutions like SimpleDB and DynamoDB. ElastiCache is yet another specific storage which works as in-memory cache in the cloud.

• **Deployment and Management**

When it comes to deployment and management we are also supported. CloudWatch offers monitoring of cloud resources. Elastic Beanstalk and CloudFormation help to quickly deploy even complicated systems into the cloud.

• **Application Services**

There is also a set of services created to help us with development of our applications. Simple Queue Service (SQS) can greatly improve reliability of an application by providing cloud based queue mechanism known for example from Unix systems. Simple Notification Service (SNS) and Simple Email Service (SES) give us scalable tools to send notifications and emails from the cloud.

There are also many other services like CloudFront that offers content delivery. Amazon recently started even a marketplace to sell software delivered by AWS. From my point of view one more feature of AWS was very appealing to me – the Java Developer Center. It consists of sample code and libraries as well as development tools and SDK to help with making our own applications. It also contains up to date documentation and many tutorials. We can additionally use the management console to control our cloud resources from the command line of our PCs.

EC2 service offers several types of instances for us to use. In case of standard types they varies from micro to extra large instances. We can choose one of the ways we want to acquire instances. **Reserved, On-demand** and **Spot instances** are the available options. During the tests I used only micro, small and medium on-demand instances with Linux and Windows operating systems. Their specification was as follows:
• **Micro Instance** (t1.micro) 613 MB of memory, up to 2 EC2 Compute Units (for short periodic bursts), EBS storage only, 32-bit or 64-bit platform

• **Small Instance** (m1.small) 1.7 GB of memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB of local instance storage, 32-bit or 64-bit platform

• **Medium Instance** (m1.medium) 3.75 GB of memory, 2 EC2 Compute Units (1 virtual core with 2 EC2 Compute Units), 410 GB of local instance storage, 32-bit or 64-bit platform
5 **Tests**

The aim of this thesis was to examine an influence of Tenant-Based Resource Allocation Model (TBRAM) on cost-effectiveness of SaaS system. Therefore, I designed and conducted tests in order to compare my SaaS systems built in accordance to TBRAM and without it. The comparison was made in terms of overutilization, underutilization and financial cost. To measure cost of running certain SaaS system in cloud I used billing information delivered by Amazon. To measure over- and underutilization is used metrics collected by Amazon CloudWatch monitoring service. Before I was able to do that I needed to set the entire test bed (Section 5.1) in Amazon cloud environment (AWS). Then the system was stressed with HTTP requests of generated workload. I used a cluster of JMeter machines performing their test plans (Section 5.2) to achieve that. There was just one test bed in general. The main difference was in the entry point to the SaaS system. In case of TBRAM it was the SCWA element including a load balancer, tenant context manager and virtual machines manager. In case of standard model it was just the Elastic Load Balancer (ELB) service from Amazon.

![Test bed deployment](image)

Figure 10: Test bed deployment
5.1 Test bed deployment

An overview of the test bed is showed in picture above (Figure 10). It consisted of Amazon EC2 instances (virtual machines) and services. Starting from the left side of the picture we can see a AS group of so called “JMeter machines”. These were virtual machines running Apache JMeter tool. They were organized in a cluster with one VM set a master and the rest as slaves. The number of VM varied depending on the number a simultaneous users to simulate. JMeter creates a single thread for each user so the consumption of computing resources strongly depends on their number. According to the program developers one or two machines with JMeter are sufficient to saturate even very powerful single server. I decided to use t1.micro Amazon EC2 Linux instances to host JMeter for several reasons. Firstly, they are free to use up to certain extent because they belong to AWS Free Tier. Even if they exceeded the free tier capabilities I did not count them to the costs of running the SaaS system. Secondly, t1.micro EC2 instances are powerful enough to simulate dozens up to even few hundreds users. During the preliminary test I established that micro EC2 instances had no problems with simulating 100 users. That was then set as the maximal number of users each JMeter slave machine simulated during the final tests. I also think they could simulate real simultaneous workload better than just one or two very powerful instances. The JMeter cluster was performing a test plan than walks through SaaS platform services. It sent HTTP requests to the entry point of the SaaS system and then collected responses to calculate the system throughput. Actually it was the JMeter master VM that collected the responses. Processing responses from few hundreds simultaneous users can be demanding from the hardware. I established that m1.small EC2 instance could handle the task, therefore that was the instance type of the master node. That node was running Windows 2008 Server for convenience of using JMeter's GUI.

What might be interesting, the JMeter master was the only instance running an operating system from Windows family in entire test bed. All others were running Amazon's version of Linux containing pre installed tools to manage AWS cloud. For Java based applications, Linux systems are known to run at least as fast as Windows systems. What is more, they are little cheaper to use since the price for most distributions does not contain license fees.

The entry point to the system was the Amazon Elastic Load Balancer (ELB) or the SCWA depending on allocation model being tested. In case of ELB it worked like
simple Apache based load balancer used in the base article. The ELB routed the web traffic in 4th OSI Internet stack layer. It used a simple algorithm like round-robin with session stickiness to balance the workload. However, ELB had some improvements regarding cloud environment. It could monitor the state of underlying balancing domain's instances. It could also seamlessly increase the number of machines it ran on in case of heavy loads. In case of TBRAM the entry point to the SaaS system was the SCWA. It consisted of three main components. Tenant-Based Load Balancer, Virtual Machines Manager (VM Manager) and Tenant Context Manager. From testing point of view only the two first components were essential. The load balancer worked in 7th OSI layer (application layer). It took into account a tenant that given user belonged to in order to redirect requests. The user's IP address was not sufficient for that purpose. Therefore, the ELB service could not be used. The VM Manager was used to monitor the balancing domain and start/stop EC2 instances to match current load. SCWA was hosted on a single m1.medium EC2 instance (3.75 GB of memory, 2 EC2 Compute Units). Since it was a single point that all the web traffic came through it needed to be powerful enough to handle it. The only difference between SaaS systems regarding tested allocation model was multi-tenancy load balancing support. All other parts remained the same.

From the load balancer, requests were directed to a AS group of EC2 worker instances. A single worker instance consisted of the SaaS platform deployed on a single Tomcat web container. SaaS platform contained services that given tenant could use. The platforms were running on m1.small EC2 instances which are sufficient and typically use to host web applications of similar size. I established that this type of instance could handle about 50 concurrent users of the SaaS platform. In case of standard resource allocation model the number of instances depended solely on the number of users and it was modified manually. In case of TBRAM the number of instances depended on current utilization state of machines and number of instances returned by VM Calculator component. Therefore it was dynamically changed to match the needs.

All the worker instances communicated with a database server. It was deployed in a special instance called Amazon Relational Database Service (RDS). That instance was running MySQL DBMS and it was independent from other system components. Since RDS is specialized for database access and the system was not very database intensive, I
used just one db.small instance. I order to improve the SaaS platforms data related performance, each of them kept a pool of database connections. After the preliminary tests I could confirm that the database would not be a bottleneck of the system. Even under heavy load it did not reach its maximal number of connections neither the CPU or memory usage.

While the tests were performed the Resource Monitor was collecting (via the JMX protocol) the RAM memory consumption of the worker EC2 instances. Then it was publishing them to the Amazon CloudWatch service. The Resource Monitor was running on a single t1.micro EC2 instance, because the task of collecting the data was not very resource consuming. The CloudWatch collected directly also other metrics, like CPU usage or network in/out transfers. This service, like most Amazon services, runs in the cloud and is seamlessly scaled horizontally to match current needs. It basically runs in one of two modes: normal or detailed monitoring. In normal mode the metrics collecting interval equals 5 minutes, whereas in detailed mode it equals 1 minute. During all the tests I used detailed monitoring to gather the more accurate data I could.

5.2 Test plan

All JMeter machines performed a set of HTTP requests to my SaaS systems according to a test plan. Figure 11 shows the flow of the test plan. In each row we can see steps belonging to certain SaaS application. The test plan starts from SCWA with the Login activity. On this stage certain tenant was linked to the logined user. For the test purposes this linking was done to a random tenant from the tenants database. Then the test flow moves to a second application, which is the Lead Application. Within this application several activities were performed like: searching for a lead, loading lead data and saving a new lead. Then similar activities were performed within a third application – Contacts Application. Finally the flow gets back to the SCWA to perform the Logout activity. This is the path that every simulated user walked through.

We can see on the next picture (Figure 5) a configuration screen from JMeter tool. We can see the test plan's tree of requests to perform. The test's parameters like number of users are not showed in the screen. The number of simulated users varied from 1 at the beginning to 400. The way the number of users reached the maximum value depended on a type of generated workload. Ramp-up period parameter can define a delay between starting one user after another. Loop count inform us how many times the
whole cycle should be repeated. It can also be set to “forever”. That means that after a
given thread (simulated user) finishes it starts once again for a new iteration. This way
we can reach only incremental growth of users number. Additionally, we are assured
that after all the simulated users were started (ramp-up period) their number will remain
the same until we shut the test down manually. It is helpful when we want to heavily
stress our system for a given period of time. In that case we can set start and end time of
the test.

During the final tests two types of workload were generated. As mentioned before,
that were incremental and peak-based workload simulations. Both of them consisted of
three phases. Despite the way the number of users varied during the tests they had one
more difference. In case of incremental workload generation the JMeter tool was set to
repeat the test loop forever. The same set up could not be applied to generate the peak-
based workload. In the second case I needed also a decrementing workload behaviour.
To achieve that I set the ramp-up period to the middle of given test phase. It means that
all the JMeter threads should have been started until the test was in the middle of its test
phase period. In this case it was after 4 hours of each test phase (test duration: 3 phases
* 8h = 24h). This however did not assured that all the threads would be simultaneously
running at the middle of the test part. Most likely the threads started at the beginning
would have finished its test paths. To solve this problem I needed to know how many
test loop iterations to set. I established during the preliminary tests that single SaaS
platform of my test bed could handle about 3470 test iterations at most (within 4 hours). Therefore the loop count parameter was set to 3470 assuring that the first JMeter thread will still be running at the middle of given test part. This parameter can be taken as a lower bound. If any of the SaaS platform VMs would process the requests slower than expected then the peak would be reached after the middle of the test phase. The consequence of such workload generation method was longer test phase duration in case of the system performance lower than expected. In the incremental tests each test phase was guaranteed to last the same time. In case of peak-based tests each test phase was guaranteed to perform the same number of iterations per JMeter machine. Thus, the workload generation methods had difference constraints (test duration or loop test iterations count).
6 Results and Analysis

This chapter presents the results of conducted research. First section (Section 6.1) presents the results in terms of server-hours, over- and underutilization as well as the cost. The next section (Section 6.2) provides more detailed results analysis. It focuses on the systems performance and costs distribution.

6.1 Results

When the tests were over it was time to collect measured data. As explained before, all of them were gathered by Amazon CloudWatch monitoring service. That tool allows to view some basic statistics of data in form of charts. However, in order to perform more advance analysis I needed to download the raw data for further processing. The results are presented in following tables (Table 3, 4).

<table>
<thead>
<tr>
<th>Simulated month</th>
<th>VMs</th>
<th>Server-hours</th>
<th>Combined-incremental</th>
<th>Combined-peak</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>incremental</td>
<td>Peak-based</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>UU (%)</td>
<td>OU (%)</td>
</tr>
<tr>
<td>January</td>
<td>2</td>
<td>1440</td>
<td>2460</td>
<td>38.75</td>
</tr>
<tr>
<td>February</td>
<td>2</td>
<td>1440</td>
<td>2580</td>
<td>20.83</td>
</tr>
<tr>
<td>March</td>
<td>2</td>
<td>1440</td>
<td>2460</td>
<td>0.00</td>
</tr>
<tr>
<td>April</td>
<td>2</td>
<td>1440</td>
<td>2580</td>
<td>0.00</td>
</tr>
<tr>
<td>May</td>
<td>4</td>
<td>2880</td>
<td>4200</td>
<td>19.38</td>
</tr>
<tr>
<td>June</td>
<td>4</td>
<td>2880</td>
<td>5160</td>
<td>9.79</td>
</tr>
<tr>
<td>July</td>
<td>4</td>
<td>2880</td>
<td>4920</td>
<td>10.63</td>
</tr>
<tr>
<td>August</td>
<td>4</td>
<td>2880</td>
<td>8040</td>
<td>10.83</td>
</tr>
<tr>
<td>September</td>
<td>8</td>
<td>5760</td>
<td>9840</td>
<td>31.61</td>
</tr>
<tr>
<td>October</td>
<td>8</td>
<td>5760</td>
<td>10320</td>
<td>34.48</td>
</tr>
<tr>
<td>November</td>
<td>8</td>
<td>5760</td>
<td>9840</td>
<td>19.90</td>
</tr>
<tr>
<td>December</td>
<td>8</td>
<td>5760</td>
<td>7440</td>
<td>21.39</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>40320</strong></td>
<td><strong>69840</strong></td>
<td><strong>Average</strong></td>
<td><strong>18.13</strong></td>
</tr>
</tbody>
</table>

The table above presents results of tests the tests performed over the Base System that used traditional resource scaling. The results comes from memory and CPU monitoring of the system. The first column contains the months of simulated year (24h of tests). In the second column we can see the number of virtual machines running each simulated month. This number is valid only for the incremental workload tests. As explained in the previous chapter, the peak-based tests were conducted in slightly different way. Instead of time constraints they were set to perform certain number of test plan iterations. We can notice the difference between the test types in server hours
provided by the VMs each simulated month. In case of the incremental test the value can be simplicity calculated by multiplying the number of VMs by the number of hours in the month (number of VMs * 24h * 30 days). In case of the peak-based test that calculation is not that straightforward because of test's time rescaling to simulated year constraints. This is because peak-based test were little longer than the original time frame of 24 hours (simulated year) per each test. The last four columns of the table contains Combined utilization. This term describes a situation during the tests when given VM was saturated or underutilized with respect to the both measures (CPU and memory). The combined utilization percentages were calculated with use of the following formulas from the base paper:

%UU (Underutilization) = (Combined UU/Measurements per hour) / server-hours. (3)

The above Formula 3 calculates the combined underutilization of given VM per each time period. It yields a percentage of wasted VMs out of all available in that time. We could also divide the number of measurements when a VM was underutilized by the number of all measurements to get the information how often the UU occurred. This number oscillated around 50% for both test types. The first way is more informative since it yields the size of the problem, not just the occurrence frequency. In case of overutilization I used the following formula:

%OU (Overutilization) = Combined OU / (Measurements per month * VMs number) (4)

Overutilization informs us about a percentage of VMs that were saturated each month. It is calculated with dividing the number of measurements that had inflection points by all measurements performed during that time period (measurements per month * VMs number). We can see that the OU almost did not appear in the tests. This is because the VMs workload was chosen with overutilization in mind. I did not wanted to saturate the VMs too much, but during the final tests the system behaved even better than during the preliminary test. That was probably caused by minor improvements in the system's code. Therefore saturation of the machines was even lower than I expected. Nevertheless, the tests of TBRAM system were conducted under exactly the same conditions, so I think it shouldn't bias the results. We can also observe the total number of server-hours provided by the SaaS platform VMs. It was 40320 and 69840 for the incremental and peak-based tests respectively.
The table above Table 4 shows results of the tests over the TBRAM system which took advantage of tenant-based load balancing and resource scaling. Since the VMs fleet size was adjusted to current needs dynamically there is no corresponding VMs number column with fixed size for each month. First thing we can notice is that total server-hours number was reduced by 19.94% and 30.21% for incremental and peak-based tests respectively. The %OU was marginally small like in the case of the Base System. There was however a difference in %UU between the systems. First of all we can notice that underutilization for incremental workload was smaller at the beginning of the simulated year than in the Base System. This is at least partially caused by the dynamic scaling method of TBRAM system. Whereas the Base System started with 2 VMs the second system could increase their number starting from just one machine. In the second part of the year the TBRAM did not act so well.

On the next chart (Figure 13) we can see the comparison of the Base System and the TBRAM system in terms of combined resource underutilization. The results comes from the incremental workload simulation tests. We can notice that during the first four months of simulated year the utilization problem did not existed in case of the TBRAM system. In the middle of the year the systems are comparable, but at the end of the year the Base System was significantly more efficient. In overall we can say that in case of the incremental tests both systems yield approximately the same resource waste (underutilization).

Table 4: Results of TBRAM system tests

<table>
<thead>
<tr>
<th>Simulated month</th>
<th>Server-hours</th>
<th>Combined-incremental</th>
<th>Combined-peak</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>incremental</td>
<td>Peak-based</td>
<td></td>
</tr>
<tr>
<td>January</td>
<td>720</td>
<td>1200</td>
<td>0.00</td>
</tr>
<tr>
<td>February</td>
<td>768</td>
<td>2040</td>
<td>0.00</td>
</tr>
<tr>
<td>March</td>
<td>1440</td>
<td>1920</td>
<td>0.00</td>
</tr>
<tr>
<td>April</td>
<td>1440</td>
<td>1440</td>
<td>0.00</td>
</tr>
<tr>
<td>May</td>
<td>1800</td>
<td>4440</td>
<td>2.78</td>
</tr>
<tr>
<td>June</td>
<td>2160</td>
<td>3960</td>
<td>10.56</td>
</tr>
<tr>
<td>July</td>
<td>2880</td>
<td>2280</td>
<td>15.83</td>
</tr>
<tr>
<td>August</td>
<td>3240</td>
<td>6000</td>
<td>1.19</td>
</tr>
<tr>
<td>September</td>
<td>3600</td>
<td>7680</td>
<td>29.17</td>
</tr>
<tr>
<td>October</td>
<td>3960</td>
<td>6720</td>
<td>40.02</td>
</tr>
<tr>
<td>November</td>
<td>4680</td>
<td>6000</td>
<td>54.45</td>
</tr>
<tr>
<td>December</td>
<td>5586</td>
<td>5064</td>
<td>51.59</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>32274</strong></td>
<td><strong>48744</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>17.13</strong></td>
<td><strong>1.42</strong></td>
<td><strong>14.51</strong></td>
</tr>
<tr>
<td><strong>T-student</strong></td>
<td><strong>0.1408</strong></td>
<td><strong>-0.0347</strong></td>
<td><strong>2.1854</strong></td>
</tr>
</tbody>
</table>
Now let us compare the systems in case of the peak-based workload tests. The following chart presents the results (Figure 14). Now the improvement from using the TBRAM is clearly visible. For most of the simulated year the TBRAM system was much more efficient than the Base System in terms of combined resource underutilization. Often the improvement was by 50% and more. The dynamic VM fleet tenant-based management showed its superiority when the workload was rapidly changing.

Figure 13: Resource underutilization during the incremental workload tests of the systems

Figure 14: Resource underutilization during the peak-based workload tests of the systems
It is important to notice that both averages for %UU are generally lower than in case of traditional scaling system. But, in order to confirm that one average is statistically different than the other I used a t-student test. In my case I wanted to check if the TBRAM system was a significant improvement to the Base System. The following t-student test parameters were used:

<table>
<thead>
<tr>
<th>N</th>
<th>degrees</th>
<th>accuracy</th>
<th>α</th>
<th>t₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>22</td>
<td>97.50%</td>
<td>0.025</td>
<td>2.074</td>
</tr>
</tbody>
</table>

where, \( N \) is the number of samples (months). \( Degrees \) stands for degrees of freedom of the t-test and it is equal to \( (N₁ + N₂ - 2 = 12 + 12 - 2 = 22) \). Unlike the authors of the base paper I chose the accuracy to be set to 97.5% rather than 99.5% because I think that test conducted in real public cloud environment is less predictable than a private cloud cluster. The \( t_α \) is a base parameter value from t-student distribution table for significance \( \alpha = 0.025 \). If the t-student value for given columns in both tables is greater than the base parameter (2.074) then we can say with 97.5% certainty that the column's averages are significantly different. We can see that the t-student value for the %UU in peak-based test is equal to 2.1854 (> 2.074). Therefore the TBRAM system statistically improved that characteristic. However, all the other averages were not improved according to the t-student test. The t-student values are given in the previous table in the last row. They were calculated by the following formula:

\[
t = \frac{x₁ - x₂}{\sqrt{\frac{s₁^2}{N₁} + \frac{s₂^2}{N₂}}}
\]  

where \( x₁ \) is an average and \( s₁ \) is a standard deviation of column in Table 3 where \( x₂ \) and \( s₂ \) are respective values for a column in Table 4.

Apart from the CloudWatch data I could also assess the systems on a financial basis. The Amazon's billing statement together with the AWS Simple Monthly Calculator were my data sources for the economical cost analysis. I was interested in the difference between the systems cost more than in total cost itself. Therefore in cost comparison I took into account only the parts that differs the both systems. That was the size of SaaS platform VMs fleet and the type of load balancer. All other test bed parts like the database, RCM or JMeter cluster were excluded from the price comparison. Since the
both test beds were stressed with exactly the same workload by performing the same test plan, the usage of database and other resources was the same. I think that excluding these elements can only increase the clarity of such a comparison. Finally, in the comparison I took into account only the cost of EC2 instances with the SaaS platform and the load balancers. Below I present Amazon EC2 prices for EU (Ireland) region for August 2012.

<table>
<thead>
<tr>
<th>EC2 instance type</th>
<th>Cost per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>0.085 USD</td>
</tr>
<tr>
<td>m1.medium</td>
<td>0.170 USD</td>
</tr>
<tr>
<td>ELB</td>
<td>0.028 USD</td>
</tr>
</tbody>
</table>

Table 7: AWS EC2 data transfer cost in EU region

<table>
<thead>
<tr>
<th>Data transfer type</th>
<th>Cost per GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>transfer out</td>
<td>0.120 USD</td>
</tr>
<tr>
<td>ELB in/out</td>
<td>0.010 USD</td>
</tr>
<tr>
<td>ELB data proc</td>
<td>0.008 USD</td>
</tr>
</tbody>
</table>

Table 6 shows the cost for every started hour of EC2 instance depending on its type. As explained in test bed deployment the SaaS platform VMs were deployed into m1.small EC2 instances. The SCWA element of TBRAM system was deployed into m1.medium instance as an equivalent of the Base System's ELB. The ELB is a special EC2 instance type and it is a part of AWS EC2 services. Table 7 in turn shows the cost per GB of data transferred through AWS. Normally we pay only for data transferred out of the cloud environment. In case of the ELB there is small difference, we pay also for data transferred in/out of the ELB (even when deployed into the same Availability Zone of a region) as well as for each GB processed by it. After this short introduction to payment details of AWS it is the time to present the actual economical difference between the systems. In Table 8 I present EC2 cost of the differing parts of both systems. We can notice that both workload types of tests where cheaper to conduct on TBRAM system with the overall cost reduction of 15.58%.

<table>
<thead>
<tr>
<th>Incremental</th>
<th>Peak-based</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.40 USD</td>
<td>21.74 USD</td>
<td>35.14 USD</td>
</tr>
<tr>
<td>TBRAM</td>
<td>11.82 USD</td>
<td>17.85 USD</td>
</tr>
<tr>
<td>Total</td>
<td>25.22 USD</td>
<td>39.59 USD</td>
</tr>
</tbody>
</table>

Table 8: EC2 cost difference between the systems
6.2 Analysis

6.2.1 Performance

The results presented in the previous chapter gave us the systems' behaviour comparison according to the base paper methodology. In this part of the paper I describe in more detail the systems performance recorded during the tests. To visualize the both systems behaviours I used graphical charts from the Amazon CloudWatch web application. Let us first take a look at the entry points to the systems which were the load balancers.

Figure 15: ELB performance during Base System incremental test

Figure 16: SaaS platform instances group performance during Base System incremental test

Figure 17: DB performance during Base System incremental test
In the charts (Figure 15, 16, 17) we can see the Base System components performance during the incremental workload test. The components are showed in sequence they were traversed by simulated client requests. That is from the ELB to the group of SaaS platform instances and eventually to the database. In the Figure 15 we can notice the moment when the ELB was scaled up. At about 08:00 August 26th the number of incoming requests to the ELB was still rising while the latency dramatically decreased because of the scaling activity. By looking at the sum of requests number (green line) we can distinguish three periods during which the VMs number was different (2, 4 and 8 VMs in this case). When the VMs number was increased we can notice rapid increase of requests handled by the ELB that appeared about 7am and 3pm on the chart. Figure 16 presents an aggregated performance of the SaaS platform instances group. We can also notice 3 distinguishable parts of the test. Each time new instances were started we can observe decreased overall utilization by the instances. This is because newly started VMs handle some part of the workload therefore decreasing the overall group utilization. The third chart (Figure 17) is just to show that the database was not a bottleneck during the test. We can see that the CPU utilization by the DB rarely exceeded 30%, even under the full system load at the end of the incremental test. Similarly the number of connections to the DB rarely exceeded 60 out of maximal 125. This was thanks to the strict connection release policy of the persistence layer.

Let us take a look now at the TBRAM system's performance during the incremental workload test. The following charts (Figure 18, 19, 20) presents the system's behaviour. The main difference between the charts is visible in the first chart (Figure 18). Instead of latency and request count statistics typical to ELB we see standard EC2 instance
metrics. We can observe strong relation between the CPU utilization and the throughput. On the second chart (Figure 19) we can see that periodical drops of the SaaS instances group resources utilization. They represent the moments where new instances were added to existing group. We do not see 3 distinguishable parts any more since the VMs were added dynamically when needed.

The last chart (Figure 20) is here to be consistent with the Base System charts. We can see that the maximal number of concurrent connections increased to about 85 while the maximal CPU utilization remained at the same level. Once again we do not see any symptoms of the database saturation. Therefore, I can assume that the DB was not a bottleneck for the system and did not negatively influenced the results. I based my assumption also on other DB metrics like read/write latency, IOPS and throughput.

Now it is time for the systems performance comparison during the peak-based workload test. The following two charts visualize the Base System behaviour (Figure 21, 22). Once again we can see three distinguishable parts of the test, only this time the parts duration period varies. The reason why is the test's constraint for the number of iterations rather than for the test duration. That was explained before in the test.
description part of this paper. By looking at the Figure 21 we can notice the relation between the peek number of simulated users (in the middle of each the test's part) and the ELB latency. The second chart (Figure 22) shows the SaaS platform instances group behaviour. We can observe the relation between the ELB's request count and the group's resources utilization.

Figure 21: ELB performance during the Base System peak-based test

Figure 22: SaaS platform instances group performance during the Base System peak-based test

Figure 23: SCWA performance during the TBRAM system peak-based test

Finally, let us take a look at the TBRAM system. The following two charts visualize the system performance similarly to the Base System (Figure 23, 24) In the first chart
(Figure 23) we can clearly see the SCWA element utilization reflected peak-based workload. We can observe 3 distinct parts of the test. The last part is little longer than the first two. That is even despite this test's characteristic were set up to make each of the parts last the same time period. As mentioned before, the peak-based tests were constrained with the number of iterations. Because during the third part of the test the system did not behaved as I expected (plateau instead of peak) the test duration was stretched out. We can see the plateau also on the second chart (Figure 24) during the third test's part. We can observe that the SaaS group was not saturated by the look at the metrics, so the group's performance was not the reason of the plateau.

The problem arose with finding the source of the plateau. My first guess was the database saturation. After all, the number of concurrent users increased very rapidly to 400 during the third part of the test. I was possible though to exceed the maximal number of DB's concurrent connections. Checking the DB's metrics however did not confirmed that theory. So the only part of the system that left was the JMeter cluster used to generate all the workload. I present the corresponding charts below:

![Chart showing average aggregated resource utilization by the JMeter auto-scaling group of instances.](image)

Figure 24: SaaS platform instances group performance during the TBRAM system peak-based test.

Figure 25 shows the average aggregated resource utilization by the JMeter auto-scaling group of instances. We need to keep in mind that the metrics where aggregated. It means that even the average metrics for the first two parts of the test looks similar, the throughput of the group was actually different. It depended mainly on the JMeter slave instances number in given test part, which was 1, 2 and 4 respectively. More to the point, we can clearly see that the utilization during the last test's part looks different. It does not, however, exhibits any saturation symptoms. Finally, let us examine the JMeter master instance performance.
In Figure 26 we can see the resource usage by the JMeter cluster’s master instance. On the chart we can see that during the second part of the test the JMeter instance was already close to saturation. During the last part it exhibits saturation symptoms because of the CPU overutilization. This kind of instance behaviour was not expected by me. The good part is that the very same instance was used in all other tests. Thus, the tests were conducted in the same conditions making the results valid. Note: Please ignore the network out (green line) peak at the beginning of the third part of the test. It was caused by the JMeter’s test data upload to the Amazon Simple Storage Service (S3) server. I took just a minute and did not affect the test.

6.2.2 Cost analysis

Before further cost analysis I would like to present a brief description of AWS pricing model. Even despite this model is mainly vendor specific, the general idea of pay-per-use is common between cloud computing providers.

One of the main reasons for using the cloud infrastructure is its flexibility. The same goes with the AWS pricing model. It can be described with two general ideas: pay-as-
you-go and pay-per-use. The first one means there is no need for long term contracts neither for any minimal commitment. The latter one is strongly related to utility computing roots of the cloud computing. It means that we pay just for what we used. This is generally true, but with certain granularity, for example: each started running hour of EC2 instance. In case of the AWS there is also no need to pay up-front for anything. We are also free to excess or under-utilize our resources without any additional fees. There are 3 fundamental characteristic for which we pay in the AWS. These are:

1. Compute
2. Storage
3. Transfer OUT

In case of computing we pay for each partial hour of our resources from start to stop of the instance. If it comes to storage we pay per each GB of stored data. There are many usage ranges with different prices. The more data we store the less per GB we pay. The last one is the transfer OUT which is generally considered as data transferred out of the AWS resources through the Internet. Transfers between our AWS resources do not count in the same way. If we communicate between the resources within the same Availability Zone (distinct physical location belonging to certain region) then we do not pay anything for that. What is also important is that we do not pay for any inbound traffic to our cloud resources. It does not matter whether the transfer IN comes form our other resources or from the Internet.

The AWS part that was used to compare my systems was the EC2 computing service. That included the instances running the SaaS platform, ELB and CloudWatch monitoring. It also included the AutoScaling service, but it is free to use as a tool. We pay just for the outputs of that tool, like increased number of running EC2 instances. Except the server-hours mentioned before, the EC2 cost depends on: instances type (m1.small, m1.medium and ELB on-demand instances in this case) and of course the instances number. When we use the ELB we are also charged for amount of data processed by the load balancer. One last EC2 service I was charged for was the CloudWatch running in detailed monitoring mode. The price of that service was one timed for both tested systems. Therefore I excluded it from the cost comparison.
Table 9: EC2 SaaS platform instances cost comparison

<table>
<thead>
<tr>
<th></th>
<th>incremental</th>
<th>Peak-based</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base System</td>
<td>10.20 USD</td>
<td>16.49 USD</td>
<td>26.69 USD</td>
</tr>
<tr>
<td>TBRAM</td>
<td>7.57 USD</td>
<td>12.24 USD</td>
<td>19.81 USD</td>
</tr>
<tr>
<td>Total</td>
<td>17.77 USD</td>
<td>28.73 USD</td>
<td>46.50 USD</td>
</tr>
</tbody>
</table>

Figure 27: The SaaS platforms costs comparison

Table 9 shows the cost of the both systems excluding the load-balancing cost (ELB or SCWA). That means that only costs of running the SaaS platform VMs were included. Figure 27 visualize the costs distribution for each test. The TBRAM cost is once again lower that the Base System's with 25.8% improvement this time. This holds even in case of incremental workload test when the TBRAM system did not statistically improved the underutilization. However, the cost reduction is the biggest for the peak-based workload tests. We need to remember however, that the SCWA contained also other parts of the system. Therefore the TBRAM system could not possibly work without that component. The next step though is to compare the separated costs of load-balancing. The results are presented in the table below.

Table 10: EC2 load balancing cost comparison

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>%system cost</th>
<th>cost</th>
<th>%system cost</th>
<th>cost</th>
<th>%system cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>incremental</td>
<td>Peak-based</td>
<td>Incremental</td>
<td>Peak-based</td>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>ELB</td>
<td>3.20 USD</td>
<td>23.88%</td>
<td>5.25 USD</td>
<td>24.14%</td>
<td>8.45 USD</td>
<td>24.04%</td>
</tr>
<tr>
<td>m1.medium</td>
<td>4.25 USD</td>
<td>35.97%</td>
<td>5.61 USD</td>
<td>31.43%</td>
<td>9.86 USD</td>
<td>33.24%</td>
</tr>
<tr>
<td>Total</td>
<td>7.45 USD</td>
<td>29.55%</td>
<td>10.86 USD</td>
<td>27.43%</td>
<td>18.31 USD</td>
<td>28.25%</td>
</tr>
</tbody>
</table>
In Table 10 we can see the cost of the ELB in Base System. The second system used m1.medium instance with the SCWA (that contained load balancer). In column named % system cost we can see what part of the system cost the load-balancing was. The ELB made 24.04% of the Base System cost, when the m1.medium instance made 33.24% of the TBRAM system cost. The ELB was also 14.2% cheaper in terms of USD price. This was because the cost of m1.medium instance per hour is over 6 times more than the ELB. We can see that, even despite additional data processing and transfer cost that the ELB introduces, the Amazon's load-balancing service was cheaper. The main difference was in the ELB's scaling up ability. I could notice the moment when the ELB scaled up during the preliminary tests. That was the trigger to make my own load balancer deployed into more powerful EC2 instance than the standard m1.small. That was clearly an over-provisioning for the time when the system load was low. The lack of scalability of the SCWA was the main reason for the higher load-balancing cost of the TBRAM system.
CONCLUSIONS

All the thesis objectives were fulfilled. After the results analysis I was ready to answer the research questions stated in this work. I could also reference my results to those from the base paper. Finally, I drew the conclusions.

RQ1: Does the tenant-based allocation model influence cost-effectiveness of SaaS applications?

The first research question was to find the TBRAM influence on cost-effectiveness of my SaaS system. In this paper I was looking at the cost-effectiveness from two different perspectives. First of them was based on server-hours number consumed by the systems during the test. The second one was based on the billing statement for AWS resources usage from Amazon. The TBRAM system used about 20% less server-hours in case of the incremental workload test and over 30% less in case of peak-based tests. That system was also over 15% cheaper than the Base System. Therefore I can say that the resource allocation based on tenants definitely influenced cost-effectiveness of the SaaS applications. So the answer for the first research RQ1 question is “YES”.

RQ2: Does the tenant-based allocation model improve my authorial SaaS application?

The second thing to examine was to check if and how the TBRAM improves my SaaS system. The results showed that this model statistically (with 97.5% accuracy) improved under-utilization of the cloud resources in case of peak-based workload. Other characteristic where generally slightly improved or the same as for the Base System. According to that I can also state that the TBRAM improved my authorial SaaS system. Therefore the answer for the second research question RQ2 is also “YES”.

RQ3: Can tenant-based resource allocation model improve cost-effectiveness of SaaS applications running in the public Cloud?

The ultimate goal of this thesis was to answer if the TBRAM can improve cost-effectiveness of other SaaS systems. The answer for the last research question requires a little more description. As it was just explained, the TBRAM reduced the cost of running my authorial SaaS system in the AWS cloud. It also improved the system in
case of underutilization. These are the advantages over the Base System conforming to traditional resource scaling. The TBRAM system introduced a twofold improvement: tenant-based load-balancing and tenant-based resource scaling. Dynamic resource scaling based on current tenants needs can significantly reduce server-hours used by multi-tenant SaaS cloud systems. It achieves that by avoiding the resource over-provisioning typical to the traditional resource scaling (based on current users number). Therefore, I think that most SaaS systems would benefit from the TBRAM in that matter, since it does not depend on the system's type. However, possible improvement of tenant-based load-balancing compared to traditional load-balancing (for example based on round-robin algorithm) depends more on the type of the SaaS system. If the SaaS system's applications are using significant amount of tenant related data, then dispatching each tenants workload to the same VMs can take advantage of server's caching. That can increase the server's performance. On the other hand, if the SaaS applications does not use tenant related data so extensively, then simple, fast and even load-balancing could be sufficient. Of course the results also depend on the TBRAM implementation. I think there are lot of more experienced cloud systems developers than I was. They possibly could benefit from the proposed model even more. In sum the TBRAM can improve cost-effectiveness of SaaS systems in cloud at least by more efficient resource management. Since computing fleet flexibility and pay-per-use pricing are integral parts of all cloud environments I think that the proposed model can reduce the costs. Thus, the answer for the main research question RQ3 is “YES”, too.

**Verification of the hypotheses**

The answers for the research questions helped me also to verify my previous hypotheses. Base on above answers I can reject the null hypotheses that claims the TBRAM does not influence the costs of running SaaS applications in cloud. Moreover, I can confirm the alternative hypotheses that claims the TBRAM can decrease these costs.

**Results comparison with the previous research**

This work was inspired and based on the base paper [10]. I wanted to check in practice if the model proposed in the base paper can really influence cost-effectiveness of SaaS systems running in public cloud. That was oppose to just testing the model in private Eucalyptus cloud. Comparing our results showed great similarities. In the base paper's research the authors achieved 32% server-hours reduction compared to
traditional resource scaling. I this work I achieved about 20% and 30% reduction in case of incremental and peak-based tests respectively. The better result for the peak-based test is caused mainly by the TBRAM underutilization improvement achieved for that type of workload. In the base work the model statistically improved also just the underutilization, but for both types of workload. Thus, I think that this work confirms the TBRAM benefits making it worth to consider even more.

Other findings

Designing, developing and deploying the SaaS systems into AWS cloud made me to draw some other conclusions, too. First of all, this research showed that TBRAM can improve the cost-effectiveness. However, this is just one side of a medal. Conformance to that model introduce non negligible development overhead. This is because we need to write the code for our own load balancer, VM manager (scaling) and resource monitor. These are not trivial elements to implement and have great influence on the overall system performance. They are also not easy to test. Because most of the system's components are independent and distributed, practically the only place they can be fully tested is the cloud we want to deploy it. Thus, they make our system more complex and error-prone. Without using the TBRAM we could simply leverage robust and elastic services delivered by the cloud provider like Elastic Load Balancer and CloudWatch monitoring in case of Amazon. So the model introduce additional costs for the system development and deployment. It is up to us to calculate whether that one time cost will return by the possible savings from decreased system's running costs.
SUMMARY AND FUTURE WORK

This work's aim was to check the proposed tenant-based resource allocation model in practice. In order to do that two SaaS systems were developed. First of them used traditional resource scaling based on number of users. It acted as a reference point for the second system. The second system enriched the base one with tenant-based load-balancing and resource scaling. Then the systems were deployed into AWS cloud. Next, both of them were tested with use of JMeter machines cluster performing the same test plan. During the tests the systems were monitored and Amazon CloudWatch service was collecting their metrics. After all, the monitoring data were analysed in terms of resource under- and overutilization. Also the cost information from Amazon billing statement was included to the systems comparison. The results confirmed previous research findings, where the model was tested in a private cloud. The TBRAM improved cost-effectiveness and resource utilization of my authorial SaaS system.

The research was conducted for certain SaaS system and cloud environment. The SaaS system representing the use of generic CRM application in one of the leading cloud provider's environment (AWS). Although the test bed was specific to used cloud, similar one could be easily deployed also into other cloud infrastructures. Altering the test bed parts could possibly yield different results, possibly even better than in this research. Therefore, there are still many things to check in order to further validate the TBRAM. The research that would require the less changes to existing system is to use different SaaS platform or applications. The test bed also could be modified. As mentioned before, the proposed model consisted of two main parts of the SCWA: load balancer and VM manager. I recommend to test the system with only one of those parts. For example substituting the tenant-based load balancer with the ELB. That Amazon's service proved to be a little bit cheaper. Also using just the tenant-based load balancer (without the VM manager) would show the real value of such a solution when compared to the ELB. Yet another idea is to leverage existing cloud services. Instead of using the SCWA component we could use Amazon AutoScaling triggers to achieve tenant-based behaviour. As an example we could set a request tenant cookie apart from standard session cookie (JSESSIONID). Then we can tell the ELB to dispatch requests based on that information. AutoScaling in turn could substitute the tenant-based VM manager SCWA part. Currently it is not possible to create auto-scaling triggers based on metrics.
from more than one resource. However, we can overcome this problem with creating our custom metrics. Therefore, we could create a metric that detects VMs saturation (more precisely: inflection points). Based on that we could create an exemplary auto-scaling policy to scale-up whenever the VM is saturated for certain period and to scale-down whenever the CPU usage is below 50%. This way we could achieve similar tenant-based resource scaling behaviour without the need to develop most of the SCWA components.

There are also many other things worth further research. I would recommend examining the system with use of distributed SCWA which could scale seamlessly like the ELB. More focus could be directed to the database. Using other, possibly noSQL engines, or distributed databases could be interesting. This work used CPU and memory utilization as VM's state metrics. Different metrics could be used to suit different needs, like measuring disk activity or system's threads number. Additionally another workload generation methods could be used to better simulate real usage scenarios. Finally, the whole test bed could be deployed into another cloud environments, like Windows Azure or Google Apps. That would probably require changes to existing system or even reimplementation.
9 REFERENCES


