Abstract:

The goal of CloudScale is to aid service providers in analysing, predicting and resolving scalability issues, i.e., support scalable service engineering. The project extends existing and develops new solutions that support the handling of scalability problems of software-based services.

This deliverable presents the work performed during the third and final period within Work Package 5, which was to develop an open access application well suited to showcase the benefits of the tools and methods developed in the project. The showcase application is called CloudStore.

This document explains the realization of the cloud-enabled version of CloudStore, the description of an expected usage evolution, and a measurement method for the capacity and elasticity of a system, and the distributed generation of load for them.

CloudStore has gained recognition as an asset in the cloud community, and has been submitted to SPEC as a candidate for standardisation.

Dissemination level

<table>
<thead>
<tr>
<th>PU</th>
<th>Public</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including Commission Services)</td>
<td></td>
</tr>
</tbody>
</table>

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 317704.
CloudScale consortium

CloudScale (Contract No. FP7-317704) is a Collaboration Project (STREP) within the 7th Framework Programme, Call 8, Priority 1.2 (Cloud Computing). The consortium members are:

- **SINTEF ICT**  
  (SINTEF, Norway)  
  NO-7465 Trondheim  
  Norway  
  www.sintef.com  
  **Project manager:** Richard T. Sanders  
  richard.sanders@sintef.no  
  +47 930 58 954  
  **Technical manager:** Gunnar Brataas  
  gunnar.brataas@sintef.no  
  +47 472 36 938

- **SAP AG**  
  (SAP, Germany)  
  69190 Walldorf, Germany  
  www.sap.com  
  Contact: Roozbeh Farahbod  
  roozbeh.farahbod@sap.com

- **Ericsson Nikola Tesla**  
  (ENT, Croatia)  
  Krapinska 42  
  10000 Zagreb, Croatia  
  www.ericsson.com/hr  
  Contact: Darko Huljenic  
  darko.huljenic@ericsson.com

- **XLAB, Slovenia**  
  Pot za Brdom 100  
  1000 Ljubljana, Slovenia  
  www.xlab.si  
  Contact: Jure Polutnik  
  jure.polutnik@xlab.si

- **Technische Universität Chemnitz (TUC, Germany)**  
  Straße der Nationen 62  
  09107 Chemnitz, Germany  
  www.tu-chemnitz.de  
  Contact: Steffen Becker  
  steffen.becker@informatik.tu-chemnitz.de

- **Kantega**  
  (Kantega, Norway)  
  Kirkegata 5  
  0153 Oslo  
  www.kantega.no  
  Contact: Hilde Nielsen  
  hilde.nielsen@kantega.no

1 The University of Paderborn (UPB) withdrew from the consortium in October 2014, as the research team there relocated to Chemnitz (TUC). TUC joined the consortium in October 2014. Kantega joined in February 2015 for compensating the reduced engagement of SAP.
# Table of contents

CloudScale consortium .......................................................... ii
Table of contents ........................................................................ iii
List of figures ............................................................................. iv
List of tables ............................................................................... iv
Executive summary ...................................................................... 1

1 Introduction ............................................................................ 3
  1.1 CloudScale motivation and background ................................. 3
  1.2 Summary ............................................................................ 4
  1.3 Structure of this document .................................................... 4
  1.4 Relationships with other deliverables .................................... 4
  1.5 Contributors ...................................................................... 4

2 Work performed and definitions ............................................ 5
  2.1 Progress and achievements in third period ........................... 5
    2.1.1 Publications and Standardisation .................................... 5
  2.2 Additional metrics ............................................................. 6
    2.2.1 Scalability with respect to cost ....................................... 6
    2.2.2 Mean time to quality repair .......................................... 6
    2.2.3 Scalability speed ......................................................... 6
    2.2.4 Number of SLO violations ........................................... 7
    2.2.5 Marginal cost ............................................................ 7

3 CloudStore .............................................................................. 8
  3.1 Current implementations review ......................................... 8
    3.1.1 Multitenant CloudStore ............................................... 8
    3.1.2 Relational CloudStore .................................................. 9
    3.1.3 noSQL CloudStore ....................................................... 9
    3.1.4 Composite service - Payment gateway ............................ 9
  3.2 Deployment ......................................................................... 10
  3.3 Measurements methodology .............................................. 11
    3.3.1 Choosing the configuration .......................................... 11
    3.3.2 Executing the measurement ......................................... 12
    3.3.3 Collecting and processing results ................................. 12

4 Measurements results analysis .............................................. 15
  4.1 AWS measurements ......................................................... 15
    4.1.1 Capacity ................................................................... 15
    4.1.2 Cost scalability .......................................................... 17
    4.1.3 Mean Time to Quality Repair (MTTQR) ......................... 18
    4.1.4 Scalability Speed (ScS) ............................................... 19
    4.1.5 Scalability Range (ScR) ............................................... 19
    4.1.6 Number of SLO Violations (NSLOV) .............................. 19
    4.1.7 Marginal cost ............................................................ 19
    4.1.8 Resource provisioning efficiency ................................. 20
  4.2 OpenStack measurements .................................................. 21
    4.2.1 Cost .......................................................................... 21
    4.2.2 Capacity .................................................................... 22
    4.2.3 Scalability with respect to cost .................................... 22
    4.2.4 Marginal cost ............................................................ 23
  4.3 Multitenant vs. tenant comparison ....................................... 24

5 Storyline and cost evaluation results ..................................... 26
  5.1 Storyline requirements ....................................................... 26
  5.2 Tipping points and cost-effective choices ............................ 27

6 Conclusion/Further work .......................................................... 29
List of figures

Figure 1 - The results of CloudScale ................................................................. 3
Figure 2 - CloudStores Meta Data Access Mechanism ........................................ 8
Figure 3 - External gateway service application ................................................ 10
Figure 4 - CloudStore architecture overview ..................................................... 11
Figure 5 - Number of SLO violations for every minute of scenario duration .......... 13
Figure 6 - Average CPU utilization of frontend instances ..................................... 13
Figure 7 - Response times for each operation ...................................................... 14
Figure 8 - Graph for capacity (#users) with respect to Cost (€/h) for Table 3 .......... 17
Figure 9 - Graph for Capacity (# users) versus Costs (€/h) for Table 2 .................. 17
Figure 10 - Scenario for MTTQR metric ............................................................... 18
Figure 11 - Graph showing SLO violations during measurement interval ............... 18
Figure 12 - Visualization of Resource Provisioning Efficiency metric ...................... 20
Figure 13 - Hoursly costs against capacity for openStack .................................. 23
Figure 14 - Change in monthly marginal cost against capacity .............................. 24
Figure 15 - Throughput vs. number of Tenants .................................................... 25
Figure 16 - Throughput and number of tenants vs. tenant size .............................. 25
Figure 17 - expected scalability and cost changes across Showcase development phases .... 26
Figure 18 - Cost (€/h) versus system capacity (# users) ....................................... 28
Figure 19 - Marginal cost (€/h) per user versus load (# users) .............................. 28

List of tables

Table 1 - Instance types for AWS cloud ............................................................. 12
Table 2 - Capacity measurements where database type is db.m3.large .................. 15
Table 3 - Capacity measurements ..................................................................... 16
Table 4 - Measurements with auto-scaling ......................................................... 16
Table 5 - Summary of measurement with ID 14 ................................................ 19
Table 6 - Cost model ......................................................................................... 21
Table 7 - Calculate cost per hour for running VM on OpenStack .......................... 22
Table 8 - OpenStack measurement results ......................................................... 22
Table 9 - Auxiliary table for the calculation of marginal costs .............................. 23
Table 10 - Expected costs and marginal costs per user, for different expected load values ........ 27
Executive summary

The main objective of work package 5 is to develop a showcase that lends itself well to demonstrate the benefits of the CloudScale tools and methods for (i.e. prediction, analysis, metrics, measurement, anti-patterns, etc.), offering a convincing story line to support dissemination and providing a platform for exploitation. The showcase aimed to be realistic, while completely open, thus overcoming any issues that the industrial use-case partners (SAP, ENT and Kantega) might have regarding confidentiality in connection with the evaluation work in WP4.

To this end, the project has developed a showcase we call CloudStore. This is a free and open scenario definition and implementation of a typical web application, with some inherent scalability and performance issues that demonstrate the merits of CloudScale. CloudStore also allows one to measure and compare different cloud providers, architectures and deployment in terms of capacity, elasticity and costs. CloudStore is based on the functional and non-functional specification of TPC-W².

The work performed during the third period of the Showcase work package focused on the final CloudStore implementations, creating a distributed and scalable load platform, defining the additional cloud-related metrics, and facilitating the usage of the Showcase as a tool for testing the CloudScale tools.

The CloudScale tools where applied to analyse CloudStore. The CloudStore implementation was migrated to both private and public clouds, namely to OpenStack and Amazon’s cloud platforms, and used to measure and compare different deployment options. In the final period we shifted focus from performance and capacity to elasticity and efficiency. The metrics were used to measure the different implementation and deployments of the CloudStore, and an analysis of comparative costs was performed in order to identify cost effective solution for the different stages of the storyline.

The source code and documentation for all the components of the CloudStore, simulated payment module and additional auxiliary tools for the measurements are made available through the project’s GitHub repository³, and include clear instructions of how to make use of them.

CloudStore has been submitted to SPEC as a tool for quantitative evaluation and analysis.

---

² TPC-W (transactional web e-Commerce benchmark) is a web server and database performance benchmark, proposed by the Transaction Processing Performance Council, see [www.tpc.org/tpcw](http://www.tpc.org/tpcw)
³ [github.com/CloudScale-Project/Showcase](https://github.com/CloudScale-Project/Showcase)
1 Introduction

1.1 CloudScale motivation and background

Cloud providers theoretically offer their customers unlimited resources for their applications on an on-demand basis. However, scalability is not only determined by the available resources, but also by how the control and data flow of the application or service is designed and implemented. Implementations that do not consider their effects can either lead to low performance (under-provisioning, resulting in high response times or low throughput) or high costs (over-provisioning, caused by low utilisation of resources).

CloudScale provides an engineering approach for building scalable cloud applications and services. Our objectives are to:

1. Make cloud systems scalable by design so that they can exploit the elasticity of the cloud, as well as maintaining and also improving scalability during system evolution. At the same time, a minimum amount of computational resources shall be used.
2. Enable analysis of scalability of basic and composed services in the cloud.
3. Ensure industrial relevance and uptake of the CloudScale results so that scalability becomes less of a problem for cloud systems.

CloudScale enables the modelling of design alternatives and the analysis of their effect on scalability and cost. Best practices for scalability further guide the design process.

The engineering approach for scalable applications and services will enable small and medium enterprises as well as large players to fully benefit from the cloud paradigm by building scalable and cost-efficient applications and services based on state-of-the-art cloud technology. Furthermore, the engineering approach reduces risks as well as costs for companies newly entering the cloud market.
1.2 Summary

This document expresses the achievements of Work Package 5 during the final period of the project.

1.3 Structure of this document

This document contains a description of the work performed during the last period, a short description of the metrics used, and a description of the last versions of the CloudStore and its components. It continues with the results of the measurements performed in the AWS public cloud, namely the missing elasticity and cost metrics, and the metrics for the OpenStack private cloud deployed at XLAB. Finally, the document includes the costs analysis for different deployments and infrastructures, closing with a short set of conclusions.

1.4 Relationships with other deliverables

- D1.3 – Design support, final version: Defines the measurements methodology
- D4.3 – Requirements and validation, final version, where the usage of the tools on the CloudStore is presented as part of the validation of the CloudScale method and its components.
- D5.1 – First version of Showcase: was the predecessor of D5.2, defines the storyline and the general goals and methodology for the work package
- D5.2 – Second version of Showcase: Description of the software implementation and deployment, AWS capacity measurements and tools usage description

1.5 Contributors

The following partners have contributed to this deliverable:

- XLAB
- SINTEF
- SAP
- TUC
- ENT
2 Work performed and definitions

2.1 Progress and achievements in third period

In the final period our main focus was to extend our results from Amazon to other cloud infrastructure, as well as producing additional metrics more focused on scalability and elasticity.

We continued working on AWS and OpenStack, and measuring former and new metrics defined in D1.3, while trying additional combinations of instance types for frontend and backend. During these measurements we were also fixing and optimizing the deployment scripts whenever it was needed. In period three we have also developed a payment service deployed as external service.

We also improved the collection of measurement results and improved the visualization of results, so that it is easier to interpret them and see what is happening with CloudStore and the underlying resources. Additional measurements also include testing different database sizes, for example with twice the number of customers and twice the number of books.

2.1.1 Publications and Standardisation

The results of our work in period three were also used for outreach and standardization actions. The paper "CloudStore – Towards Scalability Benchmarking in Cloud Computing" was accepted at the CloudForward 2015 conference, and was published at the Procedia Computer Science Volume 68, 2015, Pages 78–88 - 1st International Conference on Cloud Forward: From Distributed to Complete Computing. We have plans of submitting an extended journal article to a special issue of Future Generation Computer Systems (FGCS) in 2016.

In September 2015, SINTEF formally submitted CloudStore to SPEC as a tool for quantitative evaluation and analysis. We argued that CloudStore could reduce the research effort required to:

- Experiment with cloud metrics for scalability, elasticity and efficiency. The relationship between these metrics can be explored and details from these benchmarks can be compared.
- Benchmark (compare) different cloud computing services from Amazon, Google and Microsoft using cloud benchmarks.
- Benchmark different auto scaling policies to see which provides the best elasticity as well as efficiency.

In this way CloudStore can advance research on metrics for cloud computing as well as spur further competition between cloud computing vendors since one gains more ways of comparing their offerings than e.g. price per CPU.

In addition to the various implementations of the CloudStore service, there are corresponding Palladio models. Software engineers can use the models for conducting what-if analyses, e.g., to assess different deployments and elasticity configuration parameters. The models can also be used in the same manner as the CloudStore service: exploring metrics as well as cloud computing services. Moreover, since the CloudStore services and the corresponding Palladio models are comparable, detailed measurements can be supplemented with modelling of issues. This facilitates a model based approach, but without missing the concrete measurement feedback.

SPEC standardization of CloudStore will help to spread the knowledge about CloudStore. At the time of writing, the submission is in the process of being reviewed.
2.2 Additional metrics

In period two we focused on benchmarking the capacity of the CloudStore on the AWS cloud. In period three we also performed measurements on XLAB’s OpenStack private cloud. On AWS we measured and evaluated metrics from D1.3 focused on scalability and elasticity. These are:

- Capacity
- Scalability with respect to cost (CS)
- Mean time to Quality Repair (MMQR)
- Scalability Speed (ScS)
- Scalability Range (ScR)
- Number of SLO violations (NSLOV)
- Marginal Cost (MC)
- Resource provisioning efficiency (RPE)

Some of these metrics are dependent on each other. For instance Marginal cost is the relationship between the increases of cost against the increase of capacity, providing an estimate of the average cost of each additional user. Such metrics don’t require additional measurements, but do require additional calculation.

2.2.1 Scalability with respect to cost (CS)

Scalability with respect to cost describes how the Capacity increases depending on the (minimum) cost of the configuration. The metric is a function dependant on the cost, and how much capacity you can obtain for that given cost. In that sense, it is necessary to perform measurements for different deployment variables, since we cannot know a priori, for instance, if a several cheap instances will be more cost/effective than a few more expensive that imply the same cost. As such, the dots of cost vs. capacity are plotted on the graph, and the leftmost points are used to create a coverage function with the maximum amount of capacity that you can get for a given amount of money.

2.2.2 Mean time to quality repair (MTTQR)

Given a Usage Evolution with a work-load increase step (the amount being relevant to what can be expected in a real-life situation), and an elastic deployment with Auto Scaling, we measure the time it takes, on average, to return to a normal state without SLO Violations after that load burst. **Mean time to quality repair** metric is the time needed for the elastic system to return to normal (not violating the SLOs) after a sudden (predefined) increase in usage that is expected to reflect a real life situation.

It is useful to understand how the system copes with unforeseen bursts in load. Even if cannot handle the burst itself, we want the system to return to a normal behaviour as fast as possible.

2.2.3 Scalability speed (ScS)

We define Scalability speed as the workload range with maximal change rate for which a system can scale. This metric is a scalability metric which additionally considers the rate at which a system can scale. The metric defines that a system is able to achieve its SLOs while the workload increases at that maximal rate. The rate is defined by a maximum workload and an increase rate.

This metric focuses not on how far can a system scale to perform more work, but how fast that increase can be. This is important because even though there might be a baseline monthly growth in load that the system is handling very efficiently, it might have problems with the fast increase rates of bursts.
2.2.4 Number of SLO violations (NSLOV)

This metric can be used to compare the elasticity of two systems. The workload delta is specified as a factor (real number) as well. NSLOV reflects how often a system violates its SLOs when workload changes at a given rate, measured as a real number. For example, with a workload increase factor of 1.2, a perfectly elastic system would have 0 SLO violations per request, i.e NSLOV (1.2 req/s) = 0.

This metric is not very useful as a single value, since we don’t even know how many requests were made. It is valuable when comparing different implementations, deployment variants, technologies or infrastructures. For instance, if we are trying to decide whether to use MySQL or PostgreSQL in our solution, for a given infrastructure and given expected load, we can compare the number of SLO violation from both runs and decide that the one with lower violations performed better.

2.2.5 Marginal cost (MC)

Marginal costs, or Average Marginal Cost per User, are the operational costs of server one additional workload unit, which measures the efficiency of a cloud computing system. As exemplified before, it is calculated as the ratio between the delta in cost divided by the delta in capacity. This effectively calculates how much money it will cost to handle that many more users, averaged per user.

Given System 1 with Capacity 1000 and Cost 3.00€, and System 2 with Capacity 2000 and Cost 4.00€, it will cost 1 additional Euro \((\text{Cost}_2 - \text{Cost}_1)\) to handle 1000 additional users \((\text{Capacity}_2 - \text{Capacity}_1)\). The Average marginal cost per user, the average cost of having an additional user served by the service, is thus \((\text{Cost}_2 - \text{Cost}_1) / (\text{Capacity}_2 - \text{Capacity}_1) = 0.001€\).

The marginal cost can be calculated between the current capacity and the next step in capacity (cost of additional user from now on), but also from the capacity and cost of the minimal infrastructure to the current (to calculate the average cost of each of the existing users)

This value is particularly important to understand the costs per user of your product or service.
3 CloudStore

3.1 Current implementations review

In period three we didn’t change the CloudStore implementation from period two, because it was already good enough and scalable to use and run it in the cloud, at least within the given available infrastructure, and within the requirements of the storyline. What is new is the integration of payment gateway into CloudStore, which was added to demonstrate the utilization of composite services.

3.1.1 Multitenant CloudStore

Cloud environments reduce data centre operating costs through resource sharing and economies of scale. Infrastructure-as-a-Service is one example that leverages virtualization to share infrastructure resources. However, virtualization is often insufficient to provide Software-as-a-Service applications due to the need to replicate the operating system, middleware and application components for each customer. To overcome this problem, multi-tenancy has emerged as an architectural style that allows sharing a single Web application instance among multiple independent customers.

We decided to use the shared table approach because of its widely accepted usage. Consequently, the primary key has to be a combination of the tenantId and the entity specific id field. This was realized by making the CloudStore JPA compatible and using EclipseLink as persistence manager. EclipseLink has integrated support for Multi-tenancy. In addition to the existing CloudStore implementation the tenantId, retrieved from the meta-data manager, is added to every existing native SQL statement to ensure data isolation and thus privacy of the data.

![Figure 2 - CloudStores Meta Data Access Mechanism](image)

The CloudStore access the tenant meta-information by using a factory, which returns an interface abstracted reference to a tenant object. This makes the CloudStore portable for different tenant metadata access mechanisms. It would also enable the CloudStore to be deployed on environments without multi-tenancy support by implementing the access to the required information in a specific platform adapter (see Figure 2 - CloudStores Meta Data Access Mechanism).

Results of the comparison between tenant and multi-tenant capabilities are shown in Section 4.3.
3.1.2 Relational CloudStore

This version of CloudStore for public clouds is an adapted version of modernized version of CloudStore for usage of Amazon Web Services. We configured showcase to work with multiple databases. Instead of using a single database instance as in phase 0 and phase 1, we changed the CloudStore deployment to make use of the MySQL master-slave replication provided by Amazon’s RDS service. The “Relational Database Service” (RDS) is an Amazon’s solution to scale relational databases which currently supports only master-slave setup.

The main difference between master-slave and master-master replication is that with master-slave replication all write operations go to the master instance, while read operations are distributed across slaves by round-robin or other balancing strategy. With master-master replication all operations are evenly distributed across all database instances regardless of operation.

In period three we didn’t use master-slave configuration for testing, because our focus was on testing the frontend nodes and their capabilities, not database. We tried to bring our empirical tests closer as possible to the CloudStore model from WP3. CloudStore model assumption is, that database is set to infinity and it is not a bottleneck. We achieved that by increasing the resources (RAM, CPU) of database by vertical scaling not horizontal scaling. Horizontal scaling of database is possible only for read operations with master-slave replication. Master-master replication currently it is not supported on AWS.

3.1.3 noSQL CloudStore

We looked into the possibility of running the noSQL version of CloudStore in AWS, to which we turn to its DynamoDB service. DynamoDB is a key-value storage, itself a noSQL database, but noSQL support in CloudStore is made for MongoDB document-oriented noSQL database, so we were unable to test it within the AWS public cloud.

We also tried to make the new version of CloudStore that would work with DynamoDB. After trying and researching the possibilities we concluded that CloudStore is highly relational type of application and it doesn’t make sense to force relational data scheme into key-value data scheme, since the additional constrains to the data would end-up producing the very same data contention points that a relational database.

3.1.4 Composite service - Payment gateway

The TPC-W requirements specify the usage of an external payment service with very precise characteristics. To provide with a faithful implementation of the described requirements we created our own external gateway service, which simulates the behaviour of a real payment service. Our gateway service is very general and can be used to simulate not only payment services, but any kind of service whose response times are expected to follow a probabilistic function. The Payment gateway simulates the response times with different distributions and the dummy service can be used as an API.

Its implementation provides a graphical user interface to configure and use the system (Figure 3). It is a simple web application written in the Python programming language and it was deployed on Heroku PaaS service. We decided to deploy it on a different cloud such as Heroku because, according to the TPC-W specification, it needs to be a service external to the application. Heroku is a cloud platform based on a managed container system, with integrated data services and a powerful ecosystem, for deploying and running modern apps. Having the CloudStore running on the AWS public Cloud or our private OpenStack, and the gateway service deployed on Heroku’s PaaS provided with a realistic scenario of an external payment system.

At the same time, defining response times to follow a probabilistic function follows the approach taken at the Analyser for external services, allowing us to have a more precise and comparable simulations.
The application allows users to choose between 8 different distributions: gauss, logarithmic, Pareto, Weibull, gamma, exponential, constant, and uniform. For each distribution, the user can change the parameters that define the distribution behaviour. Users can just get a generated value of distribution for any particular purpose, or actually simulate the response time. The click on either button redirects the user on page to an API call. You can see the user interface in Figure 3.

An example of API call:
https://arcane-meadow-6418.herokuapp.com/gauss?mu=1&sigma=2&k=5&test=false

The CloudStore application makes direct use of the API (without the user interface) by calls to the API with this format.

3.2 Deployment

In period three our CloudStore deployment architecture and scripts haven’t changed much compared to the previous periods. The major difference in period three is that we added the payment gateway service into CloudStore (see Figure 4). Considering the deployment scripts we didn’t do major changes except fixing some bugs. The only new feature that we developed is the possibility of running one test multiple consecutive times.
In this period we focused on standardization of CloudStore as an industry standard for measuring and testing the clouds. With that manner we developed methodology for testing and measuring the CloudStore with defined metrics from D1.3.

3.3 Measurements methodology

The methodology that we use for measurements follows 4 steps:

- Choosing the configuration
- Executing the measurement
- Collecting and processing results
- Results interpretation

Each step is described in more details in the following sections.

3.3.1 Choosing the configuration

In this step we tried to choose configurations that would cover as many different types of instances as possible. In first batch of measurements we measured the T2 types of instances on AWS, which are a series of Virtual Machine configurations available from AWS, where we had a lot of problems, because T2s don’t have constant a CPU performance. Instead of that, they have burstable CPU performance which caused us the troubles with repeating the measurements.

In second batch of measurements we focused on M3 type of instances for AWS, which provide a constant performance and best price versus performance. In Table 1 are collected the instance types for database and frontend instances that we measured on AWS cloud:
<table>
<thead>
<tr>
<th>RDS instance type</th>
<th>Frontend instance type</th>
</tr>
</thead>
<tbody>
<tr>
<td>db.m3.medium</td>
<td>m3.medium</td>
</tr>
<tr>
<td>db.m3.large</td>
<td>m3.large</td>
</tr>
<tr>
<td>db.m3.xlarge</td>
<td>m3.xlarge</td>
</tr>
<tr>
<td>db.m3.2xlarge</td>
<td>m3.2xlarge</td>
</tr>
</tbody>
</table>

Table 1 - Instance types for AWS cloud

On OpenStack we tried to replicate the measurements done on AWS cloud, but got problems with provisioning so many resources for larger instance types, because our OpenStack infrastructure is shared and used by other employees also. That’s why we decided that we will try to replicate only the most relevant and important configurations. These are the db.m3.xlarge and db.m3.large for DB and m3.xlarge, m3.large for frontends.

3.3.2 Executing the measurement

First we measured the capacity with manual scaling just to get the feeling how many virtual users (VU) one configurations is capable of. After that we searched the more accurate capacity with bisection method. For example: if the capacity for configuration X was 2000 VU with 2000 simulated VU, in next step we increased the number of simulated VU on 4000. If the capacity was between 2000 VU and 4000 VU, we stopped with measurements, otherwise increased the simulated number of VU to 6000. We increased the number of simulated VU by 2000, because one JMeter instance can handle 2000 VU.

When we measured the capacity for each configuration, we ran the same configuration with number of VU as capacity was for that configuration, but with auto scaling enabled. We did over 50 measurements; the most important of them are collected in section 4.

On OpenStack we didn’t measure configurations with auto scaling enabled.

3.3.3 Collecting and processing results

After the measurement is finished our scripts connects to the JMeter virtual machines and collect data. This data is then processed in order to do some visualization for easier interpretation of results and to see what is happening with virtual machines during measurement. We collect data about CPU utilization of frontend instances and database, and data about operations and their response times from JMeter.

In the following figures you can see a sample of visual representation of different measurements:
Figure 5 - Number of SLO violations for every minute of scenario duration

Figure 6 - Average CPU utilization of frontend instances
The interpretation of the results is currently done manually, by examining the visualization graphs and reports.
4 Measurements results analysis

In this chapter we describe the results of measurements for public and private cloud. For public cloud we used Amazon Web Services and for private cloud we used XLABs private OpenStack cloud.

4.1 AWS measurements

In this section we describe the measurement setup, configurations and results. We list only the most important measurements to evaluate metrics from section 2.2.

We used the following setup for measuring the metrics from section 2.2:

- **Database**: db.m3.xlarge (4 vCPU, 15 GB), db.m3.large (2 vCPU, 7.5 GB)
- **Frontend**: m3.medium (1 vCPU, 3.75 GB)
- **Scaling policy**: from 1-15 frontend nodes
- **Scale up threshold**: 70% CPU utilization
- **Scale down threshold**: 20% CPU utilization

We also list the hourly price for each instance type:

- **m3.medium**: 0.067 €
- **db.m3.large**: 0.185 €
- **db.m3.xlarge**: 0.37 €

First we manually scaled the number of instances from 1 to 15 frontend instances. The database was always the same. We chose enough big instance type so it is not a bottleneck. With manual scaling we also scaled the connection pool size which maximum size is limited by RDS service.

4.1.1 Capacity

First we measured the capacity of **db.m3.large** instance type for database and **m3.medium** instance type for frontend, where we encountered that a bottleneck was on database with measurements with higher number of frontend nodes (Table 3).

<table>
<thead>
<tr>
<th>ID</th>
<th>RDS</th>
<th>FR type</th>
<th>#FR</th>
<th>Pool size</th>
<th>Duration</th>
<th>VU</th>
<th>CAP</th>
<th>Final cost</th>
<th>NSLOV</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>db.m3.large</td>
<td>m3.medium</td>
<td>1</td>
<td>600</td>
<td>16</td>
<td>2000</td>
<td>725</td>
<td>0,252</td>
<td>2 %</td>
</tr>
<tr>
<td>19</td>
<td>db.m3.large</td>
<td>m3.medium</td>
<td>2</td>
<td>300</td>
<td>16</td>
<td>2000</td>
<td>1438</td>
<td>0,319</td>
<td>3 %</td>
</tr>
<tr>
<td>20</td>
<td>db.m3.large</td>
<td>m3.medium</td>
<td>4</td>
<td>150</td>
<td>16</td>
<td>4000</td>
<td>2875</td>
<td>0,453</td>
<td>1 %</td>
</tr>
<tr>
<td>21</td>
<td>db.m3.large</td>
<td>m3.medium</td>
<td>6</td>
<td>100</td>
<td>16</td>
<td>6000</td>
<td>4313</td>
<td>0,587</td>
<td>14 %</td>
</tr>
<tr>
<td>22</td>
<td>db.m3.large</td>
<td>m3.medium</td>
<td>8</td>
<td>75</td>
<td>16</td>
<td>8000</td>
<td>5750</td>
<td>0,721</td>
<td>14 %</td>
</tr>
<tr>
<td>23</td>
<td>db.m3.large</td>
<td>m3.medium</td>
<td>10</td>
<td>60</td>
<td>16</td>
<td>8000</td>
<td>7250</td>
<td>0,855</td>
<td>15 %</td>
</tr>
<tr>
<td>24</td>
<td>db.m3.large</td>
<td>m3.medium</td>
<td>12</td>
<td>50</td>
<td>16</td>
<td>10000</td>
<td>7250</td>
<td>0,989</td>
<td>14 %</td>
</tr>
<tr>
<td>25</td>
<td>db.m3.large</td>
<td>m3.medium</td>
<td>15</td>
<td>40</td>
<td>16</td>
<td>10000</td>
<td>7188</td>
<td>1,19</td>
<td>73 %</td>
</tr>
</tbody>
</table>

**Table 2 - Capacity measurements where database type is db.m3.large**

The measurements are identified by an ID, and defined by a type of (RDS) database instance, a Frontend instance type (FR type), the number of such instances (#FR), the connection pool size for each, and duration of the measurements (in minutes), the number of virtual users (VU) for the load generation, the measured capacity (CAP), the calculated cost of all the instances, and the number of SLO violations (NSLOV) as a percent of the total.
Afterwards, we increased the performance of database so it is not a bottleneck anymore and measured the capacity (Table 2).

<table>
<thead>
<tr>
<th>ID</th>
<th>RDS</th>
<th>FR type</th>
<th># FR</th>
<th>Pool size</th>
<th>Duration</th>
<th>VU</th>
<th>CAP</th>
<th>Final cost</th>
<th>NSLOV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>1</td>
<td>1200</td>
<td>16</td>
<td>2000</td>
<td>800</td>
<td>0.437</td>
<td>57 %</td>
</tr>
<tr>
<td>2</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>2</td>
<td>600</td>
<td>16</td>
<td>2000</td>
<td>1438</td>
<td>0.504</td>
<td>13 %</td>
</tr>
<tr>
<td>3</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>3</td>
<td>400</td>
<td>16</td>
<td>4000</td>
<td>2125</td>
<td>0.571</td>
<td>32 %</td>
</tr>
<tr>
<td>4</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>4</td>
<td>300</td>
<td>16</td>
<td>4000</td>
<td>2875</td>
<td>0.638</td>
<td>14 %</td>
</tr>
<tr>
<td>5</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>5</td>
<td>240</td>
<td>16</td>
<td>6000</td>
<td>3563</td>
<td>0.705</td>
<td>25 %</td>
</tr>
<tr>
<td>6</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>6</td>
<td>200</td>
<td>16</td>
<td>6000</td>
<td>4250</td>
<td>0.772</td>
<td>34 %</td>
</tr>
<tr>
<td>7</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>7</td>
<td>171</td>
<td>16</td>
<td>8000</td>
<td>5250</td>
<td>0.839</td>
<td>22 %</td>
</tr>
<tr>
<td>8</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>8</td>
<td>150</td>
<td>16</td>
<td>8000</td>
<td>5750</td>
<td>0.906</td>
<td>15 %</td>
</tr>
<tr>
<td>9</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>9</td>
<td>133</td>
<td>16</td>
<td>8000</td>
<td>6250</td>
<td>0.973</td>
<td>7 %</td>
</tr>
<tr>
<td>10</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>10</td>
<td>120</td>
<td>16</td>
<td>10000</td>
<td>7188</td>
<td>1.04</td>
<td>12 %</td>
</tr>
<tr>
<td>11</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>11</td>
<td>109</td>
<td>16</td>
<td>10000</td>
<td>8126</td>
<td>1.107</td>
<td>5 %</td>
</tr>
<tr>
<td>12</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>12</td>
<td>100</td>
<td>16</td>
<td>10000</td>
<td>9063</td>
<td>1.174</td>
<td>4 %</td>
</tr>
<tr>
<td>13</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>13</td>
<td>92</td>
<td>16</td>
<td>12000</td>
<td>9375</td>
<td>1.241</td>
<td>9 %</td>
</tr>
<tr>
<td>14</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>15</td>
<td>80</td>
<td>16</td>
<td>10000</td>
<td>10000</td>
<td>1.375</td>
<td>0 %</td>
</tr>
</tbody>
</table>

Table 3 - Capacity measurements

Later on, we performed measurements with AWS auto-scaling, in order to measure the elasticity of the system with different configurations, available on Table 4.

<table>
<thead>
<tr>
<th>ID</th>
<th>RDS</th>
<th>FR type</th>
<th># FR</th>
<th>Pool size</th>
<th>Duration</th>
<th>VU</th>
<th>CAP</th>
<th>Increase rate</th>
<th>NSLOV</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>15</td>
<td>80</td>
<td>90</td>
<td>10000</td>
<td>166</td>
<td>111 VU/min</td>
<td>0 %</td>
</tr>
<tr>
<td>16</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>13</td>
<td>80</td>
<td>90</td>
<td>8000</td>
<td>8000</td>
<td>88 VU/min</td>
<td>0 %</td>
</tr>
<tr>
<td>17</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>11</td>
<td>80</td>
<td>90</td>
<td>6000</td>
<td>700</td>
<td>66 VU/min</td>
<td>0 %</td>
</tr>
<tr>
<td>26</td>
<td>db.m3.large</td>
<td>m3.medium</td>
<td>14</td>
<td>40</td>
<td>90</td>
<td>10000</td>
<td>611</td>
<td>111 VU/min</td>
<td>15 %</td>
</tr>
<tr>
<td>27</td>
<td>db.m3.large</td>
<td>m3.medium</td>
<td>13</td>
<td>40</td>
<td>90</td>
<td>8000</td>
<td>2889</td>
<td>88 VU/min</td>
<td>6 %</td>
</tr>
<tr>
<td>28</td>
<td>db.m3.large</td>
<td>m3.medium</td>
<td>10</td>
<td>40</td>
<td>90</td>
<td>6000</td>
<td>6000</td>
<td>66 VU/min</td>
<td>0 %</td>
</tr>
</tbody>
</table>

Table 4 - Measurements with auto-scaling

From these capacity measurements, resource scalability and cost scalability can be obtained, following the methodology better described in D1.3.

Additional measurements and experiments were performed combining different types for both the frontend and database instances, but for the purpose of this analysis we will focus on these, which focus on the scalability of a system as the number of frontend instances increases. These additional experiments can be found in D1.3, and in the paper “Resource and Cost Scalability Metrics for Cloud Computing” submitted for the 2016 Quality of Software Architectures ACM conference.
4.1.2 Cost scalability (CS)

According to definition for CS metric from D1.3 we represent this metric with a graph where we plotted the cost for running each configuration from Table 3 in correspondence to capacity (Figure 8). Labels on points mark the measurement ID. We did the same for measurements from Table 2 (shown in Figure 9).

![Cost vs Capacity](image)

*Figure 8 - Graph for capacity (#users) with respect to Cost (€/h) for Table 3*

![Cost vs Capacity](image)

*Figure 9 - Graph for Capacity (# users) versus Costs (€/h) for Table 2*

Being mostly linear with respect to the number of front-end instances, Figure 8 and Figure 9 closely resemble their respective Resource scalability graphs.
4.1.3 Mean Time to Quality Repair (MTTQR)

In order to evaluate the MTTQR metric we ran 2h scenario with auto scaling enabled and warm up time of 15 minutes to the half of capacity. Simulated capacity was 8000 VU. After 15 minutes we suddenly increased the load to full capacity and held the load for 1h 45 minutes (Figure 10). After the measurement was done we looked at the SLO violations graph and counted the minutes when SLO was violated, ignoring the first 15 minutes of warm up (Figure 11).

![Figure 10 - Scenario for MTTQR metric](image)

![Figure 11 - Graph showing SLO violations during measurement interval](image)

The evaluated metric is:

$$\text{MTTQR}_{120\text{min}} (88 \text{ VU/min}) = 34 \text{ min}$$
4.1.4 Scalability Speed (ScS)

We ran three measurements with auto scalability enabled with different number of simulated VU (different rate of increase of VU per minute). With measurement 15 from Table 2 we get number of SLO violations under 10% which is acceptable according to TPC-W specs, so the ScS is:

\[ \text{ScS}_{\text{xlarge}} = 111 \text{ VU/min} \]

If we evaluate the ScS metric for measurements from table 2 the ScS metric is:

\[ \text{ScS}_{\text{large}} = 88 \text{ VU/min} \]

because the first measurement with auto scaling enabled with number of SLO violation lower than 10% is measurement 27.

4.1.5 Scalability Range (ScR)

<table>
<thead>
<tr>
<th>ID</th>
<th>RDS</th>
<th>FR</th>
<th># FR</th>
<th>Pool size</th>
<th>Duration</th>
<th>VU</th>
<th>CAP</th>
<th>FR price</th>
<th>RDS price</th>
<th>Final price</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>db.m3.xlarge</td>
<td>m3.medium</td>
<td>15</td>
<td>80</td>
<td>16</td>
<td>10000</td>
<td>10000</td>
<td>0,067</td>
<td>0,37</td>
<td>1,375</td>
</tr>
</tbody>
</table>

Table 5 - Summary of measurement with ID 14

If we look at Table 5, we see that capacity was 10000 VU with 10000 VU simulated and there were 0 % NSLOV. That means that with more frontend nodes we could probably reach an even higher capacity, but we couldn’t measure that, due to AWS limitations in this configuration. AWS limitation for maximum number of VM is 20 per region. For that particular measurement we had 15 VMs for running the application and 5 VMs for running JMeter. Within these limitations we evaluated the Scalability Range metric as:

\[ \text{ScR} = 10000 \text{ VU} \]

4.1.6 Number of SLO Violations (NSLOV)

We counted the number of SLO violations during the measurement interval for each measurement. Column NSLOV in Table 3, Table 2 and Table 4 contains the percentage of all SLO violations during measurement interval. This raw measurement, even though it doesn’t provide a quantitative value of scalability, does provide us with a comparative measurement that can allow us to identify the solution, implementation or deployment that provides with better scalability for a given scenario. Just by running a particular Usage Evolution on two similar systems, we can just compare the number of SLO violation of each one and decide that the one with a consistent lower number of violations upon multiple runs is indeed scaling closer to the ideal. Nevertheless, the ratio between violations might not be meaningful. That is, the relationship between both measurements, other than the order, cannot be used to answer how much more or less scalable is one system in comparison with the other. Additionally, the comparative results are only valid for the given Usage Evolution, though we would expect the system with the best result to be also better in a similar scenario.

4.1.7 Marginal cost (MC)

To calculate marginal cost we must compare one measurement with measurement with increased capacity. For example, if we want to know how much we will pay for one additional user if we have available resources from measurement 12, we calculate that as the difference in cost between measurement 12 and measurement 13 divided by difference in capacity between measurement 12 and measurement 13:

\[ \frac{(1,241 - 1,174)}{(9375 - 9063)} = \frac{0,067}{312} = 0.00021 \text{ $/h} \]
Average Marginal Cost for measurements from Table 3 is:

\[ MC_{avg} = \frac{(1,375 - 0,437)}{(10000 - 800)} = \frac{0.938}{9200} = 0.00010 \text{ €/h} \]

Average Marginal Cost for measurements from Table 2 is:

\[ MC_{avg} = \frac{(1,19 - 0,252)}{(7188 - 725)} = \frac{0.938}{6463} = 0.000145 \text{ €/h} \]

4.1.8 Resource provisioning efficiency (RPE)

We measured the capacity with manual scaling where we scaled the frontend nodes from 1-15. Then we ran the measurements with auto scaling enabled and calculated Resource provisioning efficiency metric for first measurement with auto scaling enabled where NSLOV is lower than 10%, that is measurement 15 from Table 2.

![Figure 12 - Visualization of Resource Provisioning Efficiency metric](image)

In order to evaluate the RPE metric we measured the number of instances across time that were needed to support the increasing load. Next we calculated the minimum number of instances needed for the load present at any given time using Capacity information from Table 1. Finally we calculate the Overhead; that is, the difference between the actual number of instances and the minimum needed. The ratio (in percent) between ideal and real resource usage is the RPE of the system for that run.

We can calculate them to be:

\[ \text{RPE} = 83.92 \% \]

\[ \text{Cost efficiency metric} = 94.19 \% \]
4.2 OpenStack measurements

In this section we present the measurements of CloudStore on XLAB’s private OpenStack infrastructure. OpenStack Grizzly is running on 4 physical machines with Intel Xeon CPU each, 624 GB of cumulative RAM and 20 TB of cumulative HDD storage. We had a lot of problems with executing the measurements because of the storage is shared among many users. We had to execute the measurements at times when storage was less utilized.

On OpenStack we tried to replicate the measurements that we did on AWS. We didn’t measure two different databases, because on OpenStack we can manually set the maximum number of connections to database ourselves. On AWS we were using two different databases, because they have a limited maximum number of connections to database. The OpenStack Grizzly version that is used on XLAB’s infrastructure does not support auto scaling, so we evaluated only the capacity and scalability metrics, but not the elasticity ones.

4.2.1 Cost

In order to further obtain efficiency and other cost-related metrics that would be comparable to those obtain for AWS, we first needed to approximate the cost of an hour of a virtual machine in our infrastructure. We gathered the associated costs of running our testing data-centre for three years, time in which we expect to amortize the hardware, and then divide the costs by the percentage that such a virtual machine represents, divided by the amount of hours in those 3 years. Thus, we are assuming 100% perfect utilization without down-times during the entire 3 year period.

Cluster purchase price: 25000 EUR
Cluster capacity: 528 GB RAM, 46 CPU cores

<table>
<thead>
<tr>
<th></th>
<th>3 years</th>
<th>1 year</th>
<th>1 month</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server rack space</td>
<td>3'060€</td>
<td>1'020€</td>
<td>85€</td>
<td>0.115€</td>
</tr>
<tr>
<td>Electricity</td>
<td>7'488€</td>
<td>2'496€</td>
<td>208€</td>
<td>0.281€</td>
</tr>
<tr>
<td>Internet conn.</td>
<td>2'700€</td>
<td>900€</td>
<td>75€</td>
<td>0.101€</td>
</tr>
<tr>
<td>Maintenance</td>
<td>3'600€</td>
<td>1'200€</td>
<td>100€</td>
<td>0.135€</td>
</tr>
<tr>
<td>Amortization</td>
<td>25'000€</td>
<td>8'333€</td>
<td>694€</td>
<td>0.938€</td>
</tr>
<tr>
<td>Labor</td>
<td>90'000€</td>
<td>30'000€</td>
<td>2'500€</td>
<td>3.378€</td>
</tr>
<tr>
<td>Other costs</td>
<td>3'000€</td>
<td>1'000€</td>
<td>83€</td>
<td>0.115€</td>
</tr>
<tr>
<td><strong>Total cost</strong></td>
<td>134'848€</td>
<td>4'4949€</td>
<td>3'745€</td>
<td>5.062€</td>
</tr>
</tbody>
</table>

Table 6 – Cost model

The total cost to purchase 528 GB RAM cluster with 46 CPU cores is 25000 EUR. This costs model includes the rent for server rack space, cost of electricity and internet connection, maintenance and amortization costs, labour costs and other costs for administration of OpenStack. We collected the costs for the whole duration of the project (3 years) and then calculated the cost per hour by dividing 3 year cost with number of hours in the three years.

Next, we calculated how many virtual machines we can run on cluster with 528 GB RAM and 46 CPU cores, allowing for the overprovision of the CPUs (as does AWS).

- 4 GB VM (100% utilization) = 528/4 = 132 VMs
- 16 GB VM (100% utilization) = 528/16 = 33 VMs
We can now use this cost model to calculate the price per hour for running virtual machines that were used in measurements (Table 7).

<table>
<thead>
<tr>
<th></th>
<th>3 years</th>
<th>1 year</th>
<th>1 month</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 GB VM</td>
<td>1’021€</td>
<td>340€</td>
<td>28€</td>
<td>0.038€</td>
</tr>
<tr>
<td>16 GB VM</td>
<td>4’086€</td>
<td>1’362€</td>
<td>113€</td>
<td>0.153€</td>
</tr>
</tbody>
</table>

Table 7 – Calculate cost per hour for running VM on OpenStack

4.2.2 Capacity

<table>
<thead>
<tr>
<th>ID</th>
<th>DB</th>
<th>FR type</th>
<th># FR</th>
<th>Pool size</th>
<th>Duration</th>
<th>VU</th>
<th>CAP</th>
<th>Final cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>1</td>
<td>1200</td>
<td>16</td>
<td>2000</td>
<td>188</td>
<td>0.191</td>
</tr>
<tr>
<td>2</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>2</td>
<td>600</td>
<td>16</td>
<td>2000</td>
<td>1875</td>
<td>0.230</td>
</tr>
<tr>
<td>3</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>3</td>
<td>400</td>
<td>16</td>
<td>4000</td>
<td>2625</td>
<td>0.268</td>
</tr>
<tr>
<td>4</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>4</td>
<td>300</td>
<td>16</td>
<td>4000</td>
<td>3563</td>
<td>0.306</td>
</tr>
<tr>
<td>5</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>5</td>
<td>240</td>
<td>16</td>
<td>6000</td>
<td>3563</td>
<td>0.345</td>
</tr>
<tr>
<td>6</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>6</td>
<td>200</td>
<td>16</td>
<td>6000</td>
<td>6000</td>
<td>0.383</td>
</tr>
<tr>
<td>7</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>8</td>
<td>150</td>
<td>16</td>
<td>8000</td>
<td>6750</td>
<td>0.460</td>
</tr>
<tr>
<td>8</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>9</td>
<td>133</td>
<td>16</td>
<td>8000</td>
<td>7250</td>
<td>0.498</td>
</tr>
<tr>
<td>9</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>10</td>
<td>120</td>
<td>16</td>
<td>10000</td>
<td>8438</td>
<td>0.536</td>
</tr>
<tr>
<td>10</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>11</td>
<td>109</td>
<td>16</td>
<td>10000</td>
<td>8438</td>
<td>0.575</td>
</tr>
<tr>
<td>11</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>12</td>
<td>100</td>
<td>16</td>
<td>10000</td>
<td>4063</td>
<td>0.613</td>
</tr>
<tr>
<td>12</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>13</td>
<td>92</td>
<td>16</td>
<td>12000</td>
<td>5938</td>
<td>0.651</td>
</tr>
<tr>
<td>13</td>
<td>4 vCPU, 16 GB</td>
<td>1 vCPU, 4GB</td>
<td>15</td>
<td>80</td>
<td>16</td>
<td>10000</td>
<td>8438</td>
<td>0.728</td>
</tr>
</tbody>
</table>

Table 8 – OpenStack measurement results

As is visible from Table 8, capacity and number of SLO violations doesn’t raise or getting lower with more virtual machines. Our inspections showed that this happens because of slowness of distributed storage.

4.2.3 Scalability with respect to cost

The following graph depicts the increase in cost against the capacity. We can see that the cost grows slowly (double capacity with only a fraction of additional cost), while there is a jump in the cost growth (probably due to a lower-than-expected capacity for a particular configuration around 6500 virtual users)
4.2.4 Marginal cost

We calculate the marginal cost by jumps of 2 front instances, dividing the additional cost by the additional capacity (that is, the average cost for each additional virtual user).

From the capacity/cost table, we obtain this data:

<table>
<thead>
<tr>
<th>#VU</th>
<th>Cost</th>
<th>Monthly Cost</th>
<th>Marginal cost</th>
<th>Marginal Cost per month</th>
<th>#FR</th>
<th>Capacity</th>
<th>Capacity #FR+2</th>
<th>Cost #FR+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500</td>
<td>0.230 €</td>
<td>165.60 €</td>
<td>0.00004502 €</td>
<td>0.032 €</td>
<td>2</td>
<td>1875</td>
<td>3563</td>
<td>0.306 €</td>
</tr>
<tr>
<td>3500</td>
<td>0.306 €</td>
<td>220.32 €</td>
<td>0.00010267 €</td>
<td>0.074 €</td>
<td>4</td>
<td>3563</td>
<td>4313</td>
<td>0.383 €</td>
</tr>
<tr>
<td>6000</td>
<td>0.383 €</td>
<td>275.76 €</td>
<td>0.00005358 €</td>
<td>0.039 €</td>
<td>6</td>
<td>4313</td>
<td>5750</td>
<td>0.460 €</td>
</tr>
<tr>
<td>6500</td>
<td>0.460 €</td>
<td>331.20 €</td>
<td>0.00002827 €</td>
<td>0.020 €</td>
<td>8</td>
<td>5750</td>
<td>8438</td>
<td>0.536 €</td>
</tr>
<tr>
<td>8000</td>
<td>0.536 €</td>
<td>385.92 €</td>
<td></td>
<td></td>
<td>10</td>
<td>8438</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9 - Auxiliary table for the calculation of marginal costs

In Figure 14 a peak in marginal cost around 3500 virtual users is visible, which then declines as the number of users grows.
In order to calculate the average marginal costs within this capacity segment, we take the difference in monthly costs (385.92€ - 165.60) and divide it by the number of additional users (8438 / 1875), giving an average of 0.0336€ per user, per month. Similarly, we can calculate the hourly average marginal cost resulting in 0.00004663 €/h per user.

Comparing the costs tables we can see that for practically any given capacity, OpenStack is cheaper than AWS. Nevertheless, companies and specially start-ups might prefer to use the public cloud instead of incurring in any upfront costs before having their business validated in the market. Additionally, the public cloud might be particularly useful for handling bursts of load instead of having an infrastructure capable of handling bursts but that would be heavily underutilized most of the time.

4.3 Multitenant vs. tenant comparison

Having the Multitenant implementation of the CloudStore allowed us to compare the performance of both implementations in order to decide when it is more cost-effective to make use of multitenant implementations. Although the measurement results may have a relation to the capacity of a given system, the interpretation of costs should be in the focus of this investigation, and will be centred on answering the following question: “Given a certain setup, under which conditions is an application-based multi-tenancy approach more efficient than a virtualization-based approach, and vice versa?”

To produce the measurements, we followed the defined methods applied several times under different conditions. We performed 10 experiment series in total, five for each scenario. For every series, the size of each tenant was fixed to 250, 500, 750, 1000 or 1500 users, in order to measure the capacity of each system.

We started each experiment series with one active tenant and increased the amount of tenants stepwise until the application started to throw time out exceptions. To ensure equal conditions the databases were newly created, and filled with data before every run. Afterwards, the database management system and the VMs were restarted prior to starting the load driver. In the multi-tenancy scenario we restarted the VM as well. The warm-up phase was set to 10 minutes and the measurement period was 30 minutes.
Figure 15 - Throughput vs. number of Tenants
Figure 15 is the capacity measurement for both deployments with different number of virtual users. The CPU was the bottleneck that prevented achieving a higher throughput. For the same number of users, both the multitenant and the tenant versions have very similar results for lower numbers of tenants, but the multitenant version gained an advantage as the number of tenants increased.

But in order to understand these results we need to compare against the size of each tenant.

Figure 16 - Throughput and number of tenants vs. tenant size

By increasing the number of tenants for each tenant size, the maximum throughput was determined. The maximum throughput decreases with lower values for the tenant size in the case of virtualization whereas in the case of multi-tenancy throughput remains stable.

From Figure 16 we can see that multi-tenancy is less efficient in situations with more than 1000 users as then the amount of served tenants and the throughput is below the capabilities of virtualization. In the range between 250 and 1000 users per tenant, virtualization is a usable model for the given hardware configuration, if the amount of tenants to be served is below the curve for virtualization. Nevertheless, multi-tenancy is able to serve more tenants with a higher total throughput by using the same hardware in these boundaries.

The benefits of multi-tenancy become more significant in scenarios with 250 users or less. In these cases, the total throughput for virtualization was no longer limited by the CPU, instead the memory becomes the bottleneck. Multi-tenancy uses memory resources efficiently as it avoids allocating static memory for the application, application server and OS. Consequently multi-tenancy can still achieve a high throughput and good utilization of the CPU if the instances are of limited size.
5 Storyline and cost evaluation results

For the purpose of putting the showcase in the context of real-world application evolution we developed a storyline that simulates a start-up company which tries to position itself in the emerging market of e-Commerce. The story lasts for 3 years, in which the market evolves from young, through rapidly growing, to fully developed. In each phase the company is agile with adjusting the business strategies and adopting the system to the market needs.

The main business driver is surely the number of customers that correlates in the profits the company is making, which becomes under pressure when entering into the developed market phase with high competition, in which the company must rationalize its business, also with optimizing the cost of the system and its maintenance. The time-frame in which the market becomes stable and mature is normally longer, but for the purpose of the project, we have squeezed it into 3 years, where each year corresponds to situations that normally arise during the successful software life-cycle (prototyping, rapid adoption, and optimization).

The storyline includes the evolution of the CloudStore web application into a scalable and elastic cloud solution, as well as the increase in the number of daily clients, pushing the capacity demands of the solution with it. Once the implementation and deployment of the solution achieves the necessary levels of elasticity, the final goal will be to reduce costs in order to improve the business profitability.

![Figure 17 - expected scalability and cost changes across Showcase development phases](image-url)
5.1 Storyline requirements

During the description of the Storyline we described that the user-base and thus the system-load of our e-commerce application is expected to increase with time, while we expect to reduce the cost per client whenever possible. But we did not specify the number of users expected at any given time.

We will suppose now an exponential growth with a starting point of a user-base that results in a constant load of around 1000 users for year one, a load of 3000 users in year two, and 9000 users by year three. Even though the weekly and daily load can vary with bursts, we will look into the big picture assuming that the bursts will be of no more than 50% of the average, and that our implementation is scalable enough to respond to such changes. We can have confidence on this, because as measured with the Scalability Speed (from section 4.1.4), we know that we can manage changes in load of over 80 virtual users per minute.

For the sake of readability, we ignore the costs associated with the static content and the database.

5.2 Tipping points and cost-effective choices

From the measurements taken, we know that the cheapest configuration which allows us to serve 1000 concurrent virtual users without crossing the 10% time-outs defined in the SLO is to have two frontend m3.medium instances with a single db.m3.large instance for the database, with a total of around 0.32€ per hour or 230€ per month. Bursts are expected to occur sporadically, and be covered by additional duplicates of the m3.medium frontend, but most of the time we expect to be spending around 32 cents per hour for our infrastructure. We calculate the marginal costs for each additional user within bursts at around 0.00009325€ per hour, or roughly 0.07€ per month. This helps us define our strategy for user acquisition and fine-tune the pricing.

As we move forward in year two, our load increases and so does the number of instances that we need. 6 frontend m3.large instances with a single db.m3.large database are enough to handle the load of the 3000 users we expected for year two, with a cost of around 0.59€ per hour. But as the number of users continues to grow above 7000 users, this structure ceases to scale up (due to a bottleneck in the database).

As we approach the scalability range of this elastic deployment, we find the need to move to another one with a more powerful database instance. The cost of 11 m3.medium front ends and a db.m3.xlarge database is of 1.11€, but it allow us to serve over 8000 users and also scale further up, including the expected 9000 concurrent virtual users at the end of the storyline, with 13 front end instances and a cost of around 1.24€ per hour.

<table>
<thead>
<tr>
<th>#VU</th>
<th>Cost</th>
<th>Monthly Cost</th>
<th>Marginal cost</th>
<th>Monthly Marginal Cost</th>
<th>DB type</th>
<th>#FR</th>
<th>Capacity</th>
<th>Capacity #FR+2</th>
<th>Cost #FR+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.319 €</td>
<td>229.68 €</td>
<td>0.00009325 €</td>
<td>0.067 €</td>
<td>db.m3.large</td>
<td>2</td>
<td>1438</td>
<td>2875</td>
<td>0.453 €</td>
</tr>
<tr>
<td>2000</td>
<td>0.453 €</td>
<td>326.16 €</td>
<td>0.00009318 €</td>
<td>0.067 €</td>
<td>db.m3.large</td>
<td>4</td>
<td>2875</td>
<td>4313</td>
<td>0.587 €</td>
</tr>
<tr>
<td>3000</td>
<td>0.587 €</td>
<td>422.64 €</td>
<td>0.00009325 €</td>
<td>0.067 €</td>
<td>db.m3.large</td>
<td>6</td>
<td>4313</td>
<td>5750</td>
<td>0.721 €</td>
</tr>
<tr>
<td>5000</td>
<td>0.721 €</td>
<td>519.12 €</td>
<td>0.00008393 €</td>
<td>0.064 €</td>
<td>db.m3.large</td>
<td>8</td>
<td>5750</td>
<td>7250</td>
<td>0.855 €</td>
</tr>
<tr>
<td>7000</td>
<td>1.107 €</td>
<td>797.04 €</td>
<td>0.00007150 €</td>
<td>0.051 €</td>
<td>db.m3.xlarge</td>
<td>11</td>
<td>8126</td>
<td>9063</td>
<td>1.174 €</td>
</tr>
<tr>
<td>9000</td>
<td>1.241 €</td>
<td>893.52 €</td>
<td>0.00021440 €</td>
<td>0.154 €</td>
<td>db.m3.xlarge</td>
<td>13</td>
<td>9375</td>
<td>10000</td>
<td>1.375 €</td>
</tr>
</tbody>
</table>

Table 10 - Expected costs and marginal costs per user, for different expected load values

In Table 10 we can see the deployment details of the infrastructure needed to cope with the load as it evolves, as well as the actual capacity of that deployment and its cost. We also include the capacity and cost of having 2 additional Front End instances, with which we calculate the marginal cost per user as \((\text{Cost}_2-\text{Cost}_1)/(\text{Capacity}_2-\text{Capacity}_1)\); that is, how much more one needs to pay on average for each additional user above the current capacity.
Looking at the graph of costs against the progressive load in number of concurrent users we can see that the scalability of the system is rather linear.

![Cost / Number of users](image1.png)

**Figure 18 - Cost (€/h) versus system capacity (# users)**

But there is a piece of information that we are still missing. Looking at the marginal cost we see it to be quite almost constant while we have the db.m3.large database, and that as desired it improves with the db.m3.xlarge instance. Nevertheless, we seem to be approaching another bottleneck in the database connections, because the capacity between the 13 front-end instances and that of 15 instances only marginally improves the capacity, skyrocketing the marginal cost.

![Marginal Cost per month](image2.png)

**Figure 19 - Marginal cost (€/h) per user versus load (# users)**

This means that, even though we are handling our base load in a very cost-effective way, we are in a situation where Burst load will be very expensive and will potentially drive us to fail our service level objectives. A new update to our design is necessary, probably by increasing the number of connections to our database. Sadly we were unable to do this during the tests due to restrictions in the number of instances that is possible to create in Amazon, where we needed several instances just to produce the load.

Finally, we can compare the costs of running our application on either the AWS public cloud, or our private OpenStack cloud. The first one gives us an average of 0.00010 $/h, while OpenStack averages 0.00004663 €/h per user, or around half that of AWS. This confirmed our hypothesis that, in the long run, buying is cheaper than renting.
6 Conclusion/Further work

The CloudStore application, which we propose as a standard for measuring and comparing Cloud infrastructure, platforms and tools, proved to be useful to obtain important insights in the nature and behaviour of different cloud infrastructures. Interestingly enough, our measurements confirmed the hypothesis that, in the long run, buying is cheaper than renting.

The CloudStore tools helped detect problems with the implementation. In addition, the use of tools showed problems concerning the deployment of the application, whereby we determined the database and storage system to be bottleneck; this needed to be solved at the architectural level.

The main motivation for developing the showcase was to demonstrate the CloudScale tools and methods without needing to reveal business secrets concerning industrial applications. This objective was achieved. In addition the showcase could serve as a use case of its own, and could be used by any partner to perform validation of the tools. This was also done. Results of the validation of the CloudScale tools using the CloudStore show case are found in D4.3.

The work package produced additional results with value of their own. The Distributed JMeter implementation allows one to create load for testing in a scalable way, while the composite gateway is ideal for testing the impact of having external services of any kind, following an expected distribution for its response time. A methodology for measuring several of the capacity, scalability and elasticity metrics defined during the project is also presented, which can be used with the CloudStore or any other cloud application.

The showcase itself can be further improved, considering that its NoSQL implementation (currently based on Mongo) has severe performance problems, suggesting the need for a complete re-write of the solution with a different architecture and data philosophy, either on a document database like Mongo itself, or even a Key/value store such as Redis or Riak.