Moving from Monolithic to Distributed Architecture

Economical and Performance-related Impact on an e-Learning Platform

YASSIN MAKHLOUFI

Major Thesis
Master’s Degree Project
Stockholm, Sweden 2015
Copyright © 2015
Yassin Makhloufi
Kungliga Tekniska Högskolan
Moving from Monolithic to Distributed Architecture: Economical and Performance-related Impact on an E-Learning Platform

Yassin Makhloufi

October 6, 2015

Abstract

When it comes to improving the scalability and performance of a web based application with an exponential growth of user numbers, a company has to face several crucial decisions in order to find the most efficient and economically most profitable system architecture. Those decisions also include the introduction of performance increasing methods and technologies. This thesis focuses on increasing the performance of an existing e-learning platform by analyzing and resolving weak-spots in the web server configuration. Subsequently, the initially monolithic system is transformed into a distributed and highly scalable architecture. As a last step, various database solutions have been found, tested, and compared in terms of their performance. Each modification is tested and measured by appropriate metrics. The outcome is a highly scalable system with a significant performance increase.
## Contents

Glossary

List of Figures vii

List of Tables vii

1 Introduction 1
   1.1 Background .............................................. 2
   1.2 Problem .................................................... 2
   1.3 Purpose .................................................... 3
   1.4 Objectives ............................................... 3
   1.5 Methodology .............................................. 3
   1.6 Ethical Aspects .......................................... 4

2 Theoretical Background 5
   2.1 Hosting .................................................... 5
      2.1.1 Traditional Hosting .................................. 5
      2.1.2 Cloud Platforms ..................................... 6
   2.2 Components of a Web Application Architecture .............. 6
      2.2.1 Web Server ........................................... 6
      2.2.2 Application Server ................................... 7
      2.2.3 Database Server ...................................... 8
      2.2.4 Static Content Storage .............................. 8
      2.2.5 Cache Server ......................................... 9
      2.2.6 Load Balancers ....................................... 9
   2.3 Server Distribution ....................................... 10
   2.4 Scaling .................................................... 11
      2.4.1 Vertical Scaling ..................................... 11
      2.4.2 Horizontal Scaling ................................. 12

3 The Platform 14
   3.1 Purpose of the platform .................................. 14
   3.2 Related Technology ....................................... 14
      3.2.1 Laravel Framework .................................. 14
      3.2.2 Ember.js Framework .................................. 15
      3.2.3 Apache2 Web Server .................................. 15
      3.2.4 MySQL .................................................. 15
      3.2.5 Memcached ............................................ 15
      3.2.6 Content Distribution Network ...................... 16
   3.3 Architecture ............................................. 16

4 Strategy 17
   4.1 Metrics .................................................... 17
      4.1.1 Response Time ....................................... 17
      4.1.2 Active Clients ....................................... 17
      4.1.3 Number of Responses ............................... 18
      4.1.4 Cost ................................................... 18
   4.2 Utilized Tools ........................................... 19
      4.2.1 Usage Analysis ..................................... 19
4.2.2 System Analysis ........................................... 19
4.2.3 Testing .................................................. 20
4.2.4 Miscellaneous .......................................... 20
4.3 Procedure .................................................. 20

5 Tests, Results .................................................. 22
  5.1 Design of Tests ........................................... 22
  5.2 Initial Monolithic Performance .......................... 27
  5.3 Upper User Limit ......................................... 29
  5.4 Further Web Server Optimization ....................... 31
  5.5 Code Optimization for Caching ......................... 36
  5.6 Distribution ............................................. 38
  5.7 Load Balancing .......................................... 40
  5.8 Database Architecture .................................. 42
    5.8.1 Single-Entity Database ............................... 42
    5.8.2 Galera Cluster ...................................... 42
    5.8.3 Amazon Aurora ...................................... 43
    5.8.4 Database Benchmarks ................................. 43

6 Comparison and Evaluation .................................. 45
  6.1 General Findings ......................................... 45
  6.2 Impact on the Tested Platform ......................... 46

7 Conclusion .................................................. 48

8 Future Work ................................................ 49

9 Appendices .................................................. 50

10 Bibliography ............................................... 51
Glossary

APC  Alternative PHP Cache. 9
Apdex  Application Performance Index. 17
API  Application Programming Interface. 6, 14, 15, 22, 34, 36, 40
ASP.NET  Microsoft’s Active Server Pages. 8, 14
CAU  Concurrently Active Users. 18, 21, 29, 47
CBMG  Customer Behavior Model Graphs. 22
CDN  Content Distribution (or Delivery) Network. 8, 9, 16, 39
CGI  Common Gateway Interface. 32
CMS  Content Management System. 14
CPH  Costs per Hour. 18, 19
CPU  Central Processing Unit. 29, 30, 43
CSS  Cascading Style Sheets. 6, 8
CSV  Comma-Separated Values. 22
DBMS  Database Management System. 8, 15
DNS  Domain Name System. 11
EBS  AWS - Elastic Block Storage. 27
EC2  AWS Elastic Computing Cloud. 16
GUI  Graphical User Interface. 24
HTML  Hypertext Markup Language. 6, 8, 9, 14, 34
HTTP  Hypertext Transfer Protocol. 6, 7, 22, 24, 32, 36
IaaS  Infrastructure as a Service. 6, 10, 43, 47, 48
Internet Backbones  Principal data routes between the most important ISP’s core routers. 5
JSON  JavaScript Object Notation. 6
JSP  Java Server Pages. 7
MPM  Multi-Processing Module. vii, 15, 29, 32–37, 46
MVC  Model-View-Control. 14, 15
PHP  PH: Hypertext Processor. 7, 8, 14, 15, 32
PHP-FPM  FastCGI Process Manager for PHP. vii, 32–35
RDS  AWS - Relational Database Service. 39, 42
REST  Representational State Transfer. 14, 34
RMI  Remote Method Invocation. 24
RSFL  Average Number of Responses per Second under Full Load. 18, 19, 28, 31, 32, 34, 35, 37, 38, 40, 41, 47
RT  Response Time. 17–19, 28, 32
SaaS  Software as a Service. 19, 22
SQL  Structured Query Language. 8
SSD  Solid State Drive. 26, 27, 44
URI  Uniform Resource Identifier. 9
XML  Extensible Markup Language. 6
List of Figures

1  Internet User Numbers over Time (ITU, 2010) .................. 1
2  Capacity Planning in Traditional Hosting. (Tavis & Fitzsimons, 2010) .................................................. 5
3  Market Share of Web Server Software (Netcraft, 2015) ........ 7
4  Vertical Scaling .................................................... 11
5  Horizontal Scaling .................................................. 12
6  Current Architecture .............................................. 16
7  Response Time Thresholds ........................................ 17
8  Client Number Thresholds ....................................... 18
9  Distribution of Load by Controller Methods .................... 23
10 Distribution of Requests by Controller Methods ................. 24
11 Responses of the Initial System’s Test .......................... 29
12 Hardware Utilization of Initial System (c4.xlarge) ............. 30
13 Change of Prefork Config ........................................ 30
14 Hardware Utilization with Optimized Prefork Config (c4.xlarge) . 31
15 Responses of Optimized Prefork Config .......................... 33
16 Change of Worker Config ........................................ 33
17 Change of PHP-FPM Config ...................................... 34
18 Change of the MPM-Event Configuration ........................ 36
19 Concept of the Cache Filter ..................................... 37
20 Decomposed Architecture ....................................... 39
21 Newly Implemented Architecture ................................ 41
22 MySQL vs. Aurora ............................................... 45

List of Tables

1  Request and Load Distribution (for a complete list, see appendix) .... 25
2  Variety Distribution of Payloads .................................. 26
3  Tested AWS Instance Types (Amazon, 2015) ...................... 27
4  Test Results of Initial Configuration ............................. 28
5  Test Results of optimized Prefork-MPM .......................... 32
6  Test Results of optimized Worker-MPM Configuration with a default PHP-FPM Configuration ......................... 34
7  Test Results of optimized Worker-MPM Configuration with an optimized PHP-FPM ....................................... 35
8  Test Results of optimized Event-MPM Configuration ............ 37
9  Test Results with Improved Caching .............................. 38
10 Test Results with a Distributed Architecture .................... 40
11 Test Results with Horizontally Scaled Application Servers ...... 41
12 Tested AWS Instance Types (Amazon, 2015) ...................... 44
13 Database Benchmark Results ...................................... 44
1 Introduction

When handling increasing usage numbers of a web application, software developers and system administrators have to face several crucial decisions in order to ensure a continuously smooth operation of the respective service. In the past, many popular services experienced problems regarding the scalability of their systems. During the past two decades, the number of people with internet access has increased exponentially (See Fig. 1). With the rise of the web in the late nineties, especially big players such as Google and Amazon had to find ways to keep their web platforms operational with such an increasingly growing number of users and a building up complexity of their services. eBay, for instance, had to completely change their architecture multiple times within the previous years in order to handle the exploding increase of user numbers. (Küpper, 2014)

![Figure 1: Internet User Numbers over Time (ITU, 2010)](image)

The ability to continuously provide a flawless access to a web based platform depends on many things. These include but are not limited to the complexity of computations which must be performed, the amount of data that has to be processed, the system’s architecture, the efficiency of how the underlying IT infrastructure is utilized and the quality of the deployed program code.

When done correctly, the performance of a web application should be at least proportionally scalable to its user numbers. This also means that the economical impact of increasing user numbers must be lower or equal to the increasing operational costs of the respective service, so that the platform becomes proportionally more profitable with an increasing number of users or even becomes percentually more lucrative with a higher usage.

The purpose of this thesis is to investigate ways to effectively increase the performance as well as the scalability of a web application by optimizing server
configurations, applying techniques of caching, using distribution and architectural changes and introducing new technologies. The findings will be applied in form of architectural restructuring, implementation of configuration changes and provision of new technologies for an existing e-learning platform. Finally the modifications will be tested and the results will be evaluated in comparison to the original system in terms of performance and capacity, scalability and economic efficiency.

1.1 Background

Popular web applications attract an often exponentially increasing number of users. A higher number of users is obviously greatly desirable for the vast majority of service providers, for it generally goes hand in hand with an increased turnover and thus results in a higher profit for the respective company. An increasing number of users is even more desirable for operators of products or services with a present network effect, since the platform’s value for its users depends on the number of others using it. (Shapiro & Varian, 2013) Examples for such successful services are Facebook\(^1\), Craigslist\(^2\) and AirBnB\(^3\).

But an increased usage of a web service, just as in the non-digital world, also results in a greater utilization of the underlying infrastructure, which has to be maintained, adjusted and scaled on demand. For online services, a higher number of users has a direct impact on multiple components of the servers’ hardware as well as on the network infrastructure. The more users are using a web platform, the more requests it has to deal with. This again can result in a severe congestion of the entire service. (Cardellini et al., 1999) As a result of a poor choice of hardware and infrastructure or simply through a misestimation of user numbers, the service could even temporarily be completely unreachable.

Therefore a higher number of users forces an operator to increase the performance and capacity of the servers and possibly of the network. Depending on the distribution of a service’s users, it can even come in handy to use multiple locations for the servers in order to provide a fast response time to all users, independently from their geographic location (Karaul et al., 2000).

1.2 Problem

At the considered platform (See 3 The Platform), load peaks are currently dealt with by vertically scaling up the virtual server hardware. Although the scaling process involves a short reboot of the server and thus a downtime of the platform, this issue has so far been ignored. When done in advance and at low-usage times, e.g. at night, the scaling can be performed with virtually no user noticing. Yet, due to additional contracts and new clients, the company expects much higher user numbers in near future, which does not only involve a higher probability of users accessing the platform at all times. Also, the increased number of users results in a higher unpredictability of future load peaks, which again will eventually lead to an unforeseen overload of the server. In order to

\(^1\)https://www.facebook.com/
\(^2\)https://www.craigslist.org/
\(^3\)https://www.airbnb.com/
counteract the overload, the server has to be scaled up. At high load times this means taking the server off-line for a short period of time. The result of that would be many users experiencing a service that does not work.

The current architecture not only leads to the previously described problem of the vertical scaling. It also prevents the company to isolate the system’s bottleneck when it comes to an overload. The monolithic architecture involves a failing of all components of the platform when the hardware’s performance or capacity is maxed out.

The system barely uses any kind of caching. An extensive use of caching can highly decrease the load of the application and database server. Although the load on the cache server increases, the ratio of load decrease on the database and application server pays off manyfold. Especially for this system, caching can be highly profitable, since many requests involve the same data as a response. Instead of computing this data piece over and over again, the result could be cached.

The biggest current problem is related to the application server’s configuration. The software is currently using the default configuration, which leads to the platform to be overloaded when having approximately 200-300 concurrently active users, independently from the configured hardware of the server.

1.3 Purpose

This thesis aims to investigate on measures to be taken in order to improve a web application’s performance and reduce its operating costs. The investigated measures will be applied onto an existing web-based e-learning platform and the impact on performance will be measured. The different set-ups will be compared in order to find an optimal solution for the platform by examining and evaluating the test results.

1.4 Objectives

The main objective of this thesis is to enable the platform to deal with an increasing amount of users with focus on economical reasonableness. The following goals shall be achieved and their impact shall be measured by sensibly chosen metrics:

- Increase of the system’s overall performance
- Increase of the system’s scalability
- Introduction of technologies which improve the systems performance, stability or fault tolerance.

1.5 Methodology

The objective of this thesis is to find and apply techniques and technologies in order to improve the performance and economical impact on a web application architecture. Therefore, the thesis’ structure includes a background literature
study, the reconfiguration of the currently used technology, the redesign on the architecture and the introduction of novel technologies. The results are tested, analyzed and evaluated.

The vast amount of different possible test configurations in terms of software set-ups and underlying hardware leads to the quantitative research method (Håkansson, 2013). Due to the experimental and testing character of this project, the positivism is chosen as the philosophical assumption. The experiments are used in order to collect and analyze data about the tested experimental set-ups, whose results are used to improve the later steps of the experimental procedure.

As a quality assurance, we build on validity, for the experimental set-up is tested for correctness, on reliability, for the accuracy of the tests has been verified and are consistent and on replicability, for the tests base on repeatable set-ups.

1.6 Ethical Aspects

A main concern of the application’s users is the security and confidentiality of their personal data and usage behavior. By the use of an infrastructure provider, i.e Amazon Web Services, the company is theoretically not completely in control of all data. Using a managed database gives the provider an immediate and direct access to all stored data without any possible control mechanisms from the rentee’s side. By operating provided virtual machine instances and the offered pre-configured machine images, one cannot exclude the provider’s possibility to access those machines or the data that is stored on them.

The provider could hypothetically have modified the virtual machines and thus gain permanent access to the systems. Since the servers exist in a virtualized form, non-encrypted persistent data could be read directly by the provider. Additionally, it is impossible to control a possible eavesdropping on non-secured connections, not only between the servers, but also between the platform and its users.

Additionally to the ethical concerns, the possible breach of privacy laws could also result in a loss of trust by the clients and thus, in a loss of revenue in the case of a data leakage. However, considering the even higher impact on the infrastructure’s provider and their business, and also their terms and conditions in which they accept to comply to EU privacy measures, an intentional leak through the provider is highly improbable.

An additional advantage for the considered platform is the fact that it does not store personal data besides the name and an email address. Thus, even with a data leak, it would be highly difficult to link those data sets to real persons.

https://aws.amazon.com/privacy/
2 Theoretical Background

2.1 Hosting

In the case of a public web based service, the platform is running on one or multiple servers which are reachable over the internet independently from the requester’s location. In the very most cases, those servers are located in a data center that usually has a direct connection to multiple Tier-1 Internet Backbones with a bandwidth of several GBit/s. These data centers are operated by major companies and provided with a reliable connection and power supply as well as high security measures. (Yau & Cheng, 2014)

2.1.1 Traditional Hosting

The used servers are traditionally dedicated hardware, that is either rented and operated by an external service provider, or is owned by the company itself and housed by a second company that operates a data center or rents parts of one. Dedicated hardware can give an operator a huge advantage when having to estimate the performance of the system regardless of the future use. Outside factors barely have any impact when using dedicated hardware. The operator can rely on the performance of the server that only handles requests to his own applications.

Additionally, no third party has any form of access to the data that is stored on such servers in order to steal or manipulate it without having physical access to the machines. This increases the trust in such a solution.

But one downside of traditional hosting is, the inflexibility of resource planning.

Figure 2: Capacity Planning in Traditional Hosting. (Tavis & Fitzsimons, 2010)
Time dependent regular load peaks cannot be dealt with dynamically, i.e. the capacity of all servers combined must always be able to handle the highest load peak, even though many resources will idle for the rest of the time (See Fig. 2).

2.1.2 Cloud Platforms

With the rise of cloud providers such as Amazon Web Services\(^5\), Microsoft’s Azure\(^6\) and the Google Cloud Platform\(^7\), a server landscape does not have to be managed by a web application’s operator himself, whose focus and expertise should be on his product. Instead, the company can use virtualized services such as virtual server instances and load balancers from those IaaS providers without having to care about the underlying hardware or network infrastructure.

A virtualized environment also enables the cloud customer to order and use or cancel virtualized hardware according to his demands in a manner of seconds. The physical hardware already exists and is connected to the IaaS provider’s infrastructure. Virtual resources can seamlessly be added and removed at any time. Although such a virtual server instance is in general costlier than a dedicated server with the same specifications, the flexibility and the reduction of IT management costs usually pay off manyfold.

2.2 Components of a Web Application Architecture

A web application typically and traditionally consists of a number of different components. Several of them can be located on the same server or be split onto different machines. The components have completely different functions and can each, on its own, highly impact a service, both positively and negatively, depending on the respective usage by the application, the component’s configuration, and the underlying hardware and network infrastructure.

2.2.1 Web Server

The first and most essential component of a web application’s system architecture is a web server. It accepts, processes and answers HTTP-requests. (Fielding et al., 1999) Answers to those requests usually come in the form of an HTML document with embedded addresses to other resources such as CSS, script files, images or multi-media contents, which again can be requested by a client and delivered by the web server. In case of a web server that acts as an API, a response might be provided as well formatted data, for instance as JSON or XML-messages, instead of content that is intended to be specifically human readable.

A web server is solely responsible for the handling of HTTP requests and only delivers static content. Yet, it can act as a gateway between the client who requests a dynamic document and an application server that computes such requests and possibly combines it with other actions such as database queries or caching before returning the requested dynamic content.

\(^{5}\)https://aws.amazon.com/  
\(^{6}\)https://azure.microsoft.com/  
\(^{7}\)https://cloud.google.com/
Although modern web server software usually additionally implements the capabilities of an application server, both are considered separate entities in this context, as they fulfill different purposes and ultimately can be run collaboratively on separate hardware entities. (Shklar & Rosen, 2009)

The most commonly used web server software is currently the Apache HTTP Server\(^8\) with a market share of 46.6% followed by Microsoft’s Internet Information Services\(^9\) with 29.3% and NGinX\(^10\) at 12.4% (See Fig. 3) Although these three solutions dominate the market, many other solutions exist and are being used.

### 2.2.2 Application Server

An application server is a program that processes requests based on persistent data combined with an individual payload. An application server can, for instance, process PHP or JSP documents, retrieve the necessary data from the file system, a database or even remote locations, and compute the result that is to be returned to the requesting client.

An application server adds dynamic functionalities to a website and allows not only parameter-based responses, but also user generated content and even interaction between multiple users through the web application. The most

---

\(^8\)https://httpd.apache.org/
\(^9\)https://www.iis.net/
\(^10\)http://nginx.org/
used server-side languages for websites are **PHP** (81.3%), followed by **ASP.NET** (16.7%) and Java (3.0%). (W3Techs, 2015) These languages equip a website with logics and allow to deploy interactive sites and complex services such as social networks or shopping sites.

### 2.2.3 Database Server

The very most websites need a dynamic possibility to persistently store and retrieve data. For basic and small amounts of information, the data can be stored easily and quickly on the application server’s file system or even on remote locations. For frequent changes, a database provides a much higher performance among many other advantages. Especially when it comes to data security, data indexing and complex transactions. For instance a relational database such as MySQL\(^{11}\) or Oracle RDBS\(^{12}\) can improve the overall system’s performance manyfold.

MySQL, followed by its fork MariaDB\(^{13}\), is by far the most used DBMS. (ScaleBase, 2014). Both are based on SQL and designed as a relational DBMS. But also systems which do not use SQL are widely used, such as MongoDB\(^{14}\), which can, in contrast to MySQL, handle, for instance, data locality and has a highly rich data model.

With respect to a web application, a database system especially needs to be highly available, reliable, scalable and efficient. These attributes depend not only on the choice of the DBMS, but also on its configuration and on the system that uses it, as well as on the infrastructure in which it is operated.

### 2.2.4 Static Content Storage

All the static data that is traditionally delivered by a web server, such as HTML, JavaScript files, CSS files, media files, etc. can be outsourced. This is, for instance, interesting to companies with combined web and application servers. Outsourcing static content to other locations directly decreases the server load of the application servers. When a website has a geographically well distributed target audience, and requests are coming from many different locations on multiple continents, the distributed server infrastructure of a CDN can also significantly decrease the response time for a website’s users.

A CDN, such as Akamai\(^{15}\) or AWS CloudFront\(^{16}\), usually operates a huge number of servers in several different geographical locations. This does not only reduce the latency to a website’s users but also increases the availability and performance of a web application. The use of a CDN can also highly decrease the traffic on the content provider’s infrastructure, which can possibly save costs. (Nygren et al., 2010)

\(^{11}\)https://www.mysql.com/
\(^{12}\)https://www.oracle.com/database/
\(^{13}\)https://mariadb.org/
\(^{14}\)https://www.mongodb.org/
\(^{15}\)https://www.akamai.com/
\(^{16}\)https://aws.amazon.com/cloudfront/
In order to use external static content storage, such as a CDN, the web server has to provide external instead of internal links embedded into the HTML document that is delivered to the users’ browsers. Those links induce the browsers to request the static content files from the external source instead of from the website’s server, which in turn reduces the traffic and the load of the website’s web servers.

2.2.5 Cache Server

Caching is one of the techniques used in order to reduce the general load of an application by temporarily storing the computation result of a request. If the same request reoccurs within a short (or pre-set) amount of time, the computation will be skipped and the previously cached result will be returned instead. This can significantly increase the performance of a web application as well as reduce the load on various entities of the system.

Depending on the implementation of the respective web application, multiple processes can be sped up through caching. If, for instance, a site delivers the same information to every user and the request URI matches, then the entire response can be computed once and the result can be returned to every user with the same request, without having to compute the request over and over again. For a finer granularity, an application can, for instance, cache single (serialized) objects or database query results as well.

A certain variety of caching technologies exists and can be applied on many systems. Depending on the implementation and available infrastructure, e.g. an external or internal Memcached server can be set up. Alternatively, there are file based and database oriented caching technologies and even specific frameworks in various programming languages which handle the caching for the programmer, such as APC.

In order to store a computation result in the cache, a key will be determined that describes the information that is to be stored. This key will be stored along with the data and can then be used in order to figure out if the requested information exists in the cache or if it has to be (re-)computed. Additionally, the expiry information will be stored along with the data. Depending on the type of data, the computation might have to be redone in a certain cycle in order to assure the actuality of the data.

2.2.6 Load Balancers

In the case of horizontal scaling (See 2.4.2 Horizontal Scaling), the requests that are sent to a service need to be distributed to multiple servers in order to scatter the website’s load. Therefore, the domain name of a website can point to a load balancing entity instead of a web server, which then redirects the requests to a web server that is supposed to handle the request.

A load balancer can use various decision criteria in order to choose the most appropriate server for a respective request, e.g. based on the target server’s load.

\[^{17}\text{http://www.memcached.org/}\]
Alternatively a load balancer can also evenly distribute the requests, either by random distribution or by round-robin scheduling.

Many IaaS providers offer load balancers in order to distribute the load of an application to a set of servers that then handle the respective requests. In many cases those are also highly dynamic, i.e. the capacity of the load balancer increases and decreases on demand.

2.3 Server Distribution

Even though it is possible to run all components of a web service on the same hardware entity, there are many advantages in moving away from such a monolithic architecture. Often the seamless operation of an application can be negatively affected by one single component which is either overloaded or does not function as expected. For a monolithic system this means that all other components will be affected as well, since they share the same hardware and network resources.

By contrast, distributing the components of a web application can bring significant advantages to its operator when it comes to increasing the scalability and availability of the system. In case the system runs into performance problems, only the hardware of the entity that is serving the bottleneck has to be adjusted instead of the entire system. Thus, if one component is reaching its performance limit, it can be scaled up without needlessly changing the hardware configuration of other components.

The latter could also possibly involve a data migration or other administrative tasks. Therefore the distribution of server components does not only reduce the direct hardware costs, but also helps avoiding unnecessary administrative and technical efforts which in turn always result in higher HR costs.

Another advantage is an increased availability of the system. If a component fails, not all other components are necessarily affected. Thus, if an optional component such as a cache server fails, the rest of the system will continue to work, even though possibly less efficient. The distribution also helps quickly identifying and solving failures. In a distributed system, a congestion or faulty component can quickly be spotted and possibly be exchanged or fixed easily. This increases the maintainability of a system significantly.

The classical three-tier architecture (Smith et al., 1998) for instance separates the client application from the business rules, which are executed on a remote entity, which again is separated from the data storage. That way, data is never accessed directly by the client and modifications on the system logics can be performed without affecting the other components. The distribution can be applied onto web applications to an extent of a 6-tier architecture (See 2.2 Components of a Web Application Architecture)
2.4 Scaling

When a website’s number of users increases or its complexity grows, the service can suffer from severe congestion (Cardellini et al., 1999). In order to tackle this problem, the infrastructure on top of which the service is running has to be scaled up to a level on which a seamless operation of the website to its peak times is possible, when a vast amount of requests hits the platform. Depending on which parts of the infrastructure are most prone to congestion, these components have to be improved in terms of performance or capacity.

Congestions are a result of multiple limitations of the utilized hardware and software. They lead to certain challenges which have to be dealt with in order to keep a service up and running. Especially computational power, as well as working memory, storage space and network bandwidth are often a bottleneck when it comes to the continuous smooth functioning of the respective platform that is dealing with increasing load.

An unhandled bottleneck can cause multiple problems; among them a slow response time and a bad user experience. It can even result in a complete collapse of the service in the case of a critical component failure or an overload in monolithic architecture (See 2.3 Server Distribution). This in turn could lead to massive economical losses for a company.

2.4.1 Vertical Scaling

For a monolithic or closely coupled system with only one major hardware entity - virtual or dedicated - an overload or a congestion forces the operator of the service to vertically scale up that entity in terms of performance, capacity and/or storage capabilities (See Fig. 4). This means, that the hardware performance has to be increased either by adding additional resources, such as RAM modules or hard drives or by moving the system to entirely new hardware. Even if only one single hardware component does not provide a sufficient performance or capacity, mostly all components have to be exchanged, since dedicated servers are generally offered in predefined configuration packages. This in turn is expensive due to the effort in the system’s migration as well as the fact that a server with improved hardware is in general much more expensive.

Figure 4: Vertical Scaling

Another major drawback of vertical scaling is the migration problem that it brings due to the necessity to operate two servers simultaneously. In case of a migration to new hardware, the new and the old server have to be running concurrently until the entire migration is completed and all possible routing caches such as DNS entries are updated. In the case of a local hardware upgrade or a
virtual server, it usually has to be completely shut down in order to change the (virtualized or physical) hardware configuration. To prevent a downtime of the website, an intermediate server has to run in parallel and the data has to be transferred to a new one. To maintain a synchronized database and a seamless transition between the systems brings its own problems due to possibly existing transactional data on the application layer. Yet, this practice has to be followed in order to avoid down times of the service. The resulting inflexibility hinders a company from dynamically adding and removing resources on demand. Therefore, the vertical scaling is usually only done in one direction, i.e. up. Thus, the wasted capacity (See 2.1.1 Traditional Hosting) is, in result, a waste of financial resources.

Additionally, a vertical scaling obviously has an upper limit. The hardware can possibly reach a point where only highly expensive mainframe computers would be able to handle the load, which again also have an upper limit at some point.

On the other hand, vertical scaling can use virtualization technology more effectively as it provides more resources for the hosted operating systems to share. (Küpper, 2014)

2.4.2 Horizontal Scaling

Horizontal scaling is much more flexible when it comes to the handling of load peaks. Many services have to deal with a non-consistent load of their servers. The load of a web service can vary by the time of the day, the day of the week or even the month of a year. Online shops, for instance, are facing a much higher use of their websites in the time before Christmas. (Bellenger et al., 2011) Confronted with this situation, many companies chose to design their applications accordingly.

Horizontal scaling leverages the possibilities of distributed architectures to make a system highly scalable by duplicating certain components of a system (See Fig. 5) and distributing the load onto those. (Küpper, 2014) The number of copies is hereby not limited by the architecture. Only network configurations or infrastructure related reasons can restrict the system from exceeding a certain number. For instance, an application server (See 2.2.2 Application Server) can be duplicated and the load can be distributed using a load balancer (See 2.2.6 Load Balancers). The results of identical requests to different application server instances would then be the same as long as all application servers access the same external resources, such as database or cache servers.

Figure 5: Horizontal Scaling
This approach allows the operators of services to flexibly react to unexpected load peaks. If a load balancer observes a significant increase of the existing servers’ load, it can add multiple additional server instances of the respective component in order to better distribute the requests and decrease the individual load. Also, a horizontal scaling can significantly lower the costs for companies with regular increases and decreases of their service’s usage numbers. A time schedule based addition or reduction of the instance numbers prevents the company from maintaining and paying for a performance level equal or higher than the highest regular load, due to the pay-as-you-go pricing model of cloud providers in which a client only pays for those resources which are being used at the time.

Nonetheless, horizontal scaling poses some problems of its own, especially when it comes to a system with closely coupled components. Particularly a database, which would be replicated by horizontally scaling a single-entity-system, will have to stay continuously synchronous. There are many approaches for a distributed database architecture, whereas a few technologies and methods have proven to be reliable, effective and efficient.

Horizontal scaling enables a company to operate distributed applications in a more effective manner regarding the utilized hardware. Both, the partitioning as well as the pooling of resources are more efficient. (Sosinsky, 2010)
3 The Platform

The platform on which the findings of this thesis will be applied and tested is an educational e-learning platform that is in productive use in multiple countries. It is currently in the state of crucial technological changes in terms of architecture, used technologies as well as its infrastructure, which is the company’s main motivation of supporting and supervising this project.

3.1 Purpose of the platform

The platform offers educational contents to students in the form of textual and illustrated descriptions, as well as multimedia contents with a focus on the curricula of school years 7 to 9. Using this platform, students can follow the contents tuned to their very curriculum in their preferred language with a variety of possible subtitles. This enables students with different native languages to follow the learning contents, especially when such students recently moved to a different country and still have difficulties with the local language.

Additionally to the learning contents, students can complete quizzes related to the followed lessons in multiple difficulty levels in order to test and verify their understanding of the respective topic. The progress of the students can be followed and statistically evaluated by parents and teachers in order to improve teaching at school and homework support at home.

3.2 Related Technology

The system was originally built on the Drupal\textsuperscript{18} CMS and has recently been translated into a custom application based on the Laravel\textsuperscript{19} MVC-framework. It is now built as a state-of-the-art one-page-application with an Ember.js\textsuperscript{20} front end. The following technology and configuration describes the initial state of the system before this project’s findings have been applied.

3.2.1 Laravel Framework

Laravel is a PHP based MVC Framework, initially released in June 2011.\cite{surguy2013} For the past four years (as of 2015), Laravel has been getting increasingly more attention among web developers. Laravel re-uses existing components and allows a developer to leverage a complex yet clear layer on top of PHP, which eases the building of web applications significantly. Laravel combines many of the best features of CodeIgniter, Yii, ASP.NET MVC, Ruby on Rails and Sinatra, which makes it a very powerful development tool. \cite{saunier2014}

The entire system, which this thesis is based on, consists of two Laravel instances. One for the delivery of the initial HTML document and all related resources such as the JavaScript frameworks and a second instance that provides a RESTful API, that gives the web application access to the server side.

\textsuperscript{18}\url{https://www.drupal.org/}
\textsuperscript{19}\url{http://laravel.com/}
\textsuperscript{20}\url{http://emberjs.com/}
functionality including the persistent data storage and caching layers. Both instances are run on the same application server, while the very most load lies on the instance that serves the API.

3.2.2 Ember.js Framework

Ember is a relatively new JavaScript MVC framework, which was initially released in 2011. It integrates the highly efficient Handlebars templating engine which automatically updates the page when data is changed behind the scenes.

“Ember.js is a JavaScript MVC framework that handles important tasks like managing code modules, maintaining state, and expediting reliable data flow. It provides the patterns, components, and scaffolding you need to build ambitious web applications.”

(Skeie, 2014)

A one-page application, that uses an MVC framework such as Ember.js, is capable of offloading all the layout and displaying computations onto the client machine. This does not only reduce the load on the web and application servers of the web platform but also significantly decreases the network traffic between the user’s web browser and the server. Only the necessary data, such as user and content objects, are transferred, and the transfer can even be compressed using gzip/deflate which is supported by all modern web browsers. Traditional responses of a web server contain usually a vast amount of redundant data and already transferred content which can be reduced by this procedure.

3.2.3 Apache2 Web Server

As a web server software, the platform is running on an Apache2 web server which acts as both, a web server and an application server. The Apache server handles the requests of clients, the connection to the database and the cache server. As application server, the software is configured to use the PHP5-module with the Prefork-MPM.

3.2.4 MySQL

The persistent data is stored in a local MySQL database. The database is structured in a relational way such that the result of a database query can be aggregated before it is returned to the application server. Since a DBMS is optimized for database requests, in general this is computationally much faster compared to an aggregation on the application layer.

3.2.5 Memcached

The cache being used in order to reduce the computation and querying of the same requests and their results is based on a memcached server. It can be read from and written to by the application server which computes the results of database queries or system objects.
3.2.6 Content Distribution Network

An external CDN is used for the storage and delivery of multi-media contents. The videos presented on the web platform are being linked externally to their locations on the CDN’s servers, which offloads the media-traffic onto their infrastructure and reduces the load and therefore increases the availability and stability of the platform itself. As a CDN provider, the platform uses Amazon AWS Cloudfront. Yet, the CDN does not host all possible static information but only the videos.

3.3 Architecture

The platform that is being regarded in this thesis initially consists of two entities: the CDN and the platform server - both hosted by AWS. The latter incorporates the web server, the application server, the database and the cache on the same server.

![Figure 6: Current Architecture](image)

The server is hosted as a single EC2 instance. This makes the architecture effectively monolithic (See Fig. 6). Due to having low user numbers in the range of about one hundred concurrently active clients at peak times, scalability has not been an urgent issue up to this point. Higher loads could easily be predicted well in advance. At occurring load peaks, the server had been scaled up vertically in advance, so that the required performance could be provided when necessary.
4 Strategy

In order to achieve the objectives, appropriate metrics have to be chosen which will be used to design tests. The results of these tests will then be evaluated in order to determine what impact the respective changes and optimizations had on the system as well as to decide which of them are most beneficial.

4.1 Metrics

The following metrics describe a way to measure the performance and the scalability of the various configurations to be tested.

4.1.1 Response Time

The response time (RT) describes the time that elapses between the moment of sending a request and receiving a response to that very request. With an increasing hardware load of a server, the RT can grow to an extent that the system becomes unusable. This is a result of more requests being made than responses can be sent, which leads to the requests being queued for a longer time. Directly related to the Response Time is the Apdex - the Application Performance Index (Sevcik, 2005).

![Impact of Response Time on User Experience](image)

**Figure 7: Response Time Thresholds**

It assumes that after the RT exceeds a certain threshold, a user gets "unsatisfied" and after a second higher one, he gets "frustrated". Taking research findings (Hoover, 2006) and the company’s goal regarding user satisfaction into account, the first threshold has been chosen at 1 second and the second threshold at 4 seconds (See Fig. 7).

4.1.2 Active Clients

When mentioning clients in the context of web platforms, the term is usually interchangeable with "user" - at least within the scope of this work. Thus, when speaking of active clients, meant is the number of concurrently active users
browsing and using the platform. An increasing number of active users leads to an increase of the system load and thus to an increase of the RT.

Figure 8: Client Number Thresholds

Taking into account the previously mentioned thresholds, it results in two metrics (See Fig. 8): the number of CAU up to which the average RT is satisfying ($U_S$) and the number of CAU up to which the average RT is tolerated ($U_T$).

4.1.3 Number of Responses

Another indicator for the performance potential of a system is the average number of responses (RSFL) it can generate in one second under full load. Although this number should be approximately proportional to $U_S$ and $U_T$, it gives us another metric that does not directly depend on the RT but rather on the actual amount of work the platform can handle.

4.1.4 Cost

The last but obviously very important factor is the cost of running the platform in a certain configuration. This includes costs for servers and infrastructure, measured as costs per hour (CPH), which will be calculated in US-Dollars.

In order to compare the previously presented metrics to each other in a reasonable manner, the costs are presented as a function of them. This results in three artificial economically comparable metrics, the 'One-Dollar-Hour' metrics:
\[ O_{US} \triangleq \frac{U_s}{CPH} \] The number of concurrently active clients up to which the average RT is satisfying and which can be maintained for one continuous hour at the cost of 1 USD.

\[ O_{UT} \triangleq \frac{U_t}{CPH} \] The number of concurrently active clients up to which the average RT is tolerable and which can be maintained for one continuous hour at the cost of 1 USD.

\[ O_{RSFL} \triangleq \frac{RSFL}{CPH} \] The average number of responses per second under full load which can be maintained for one continuous hour at the cost of 1 USD.

Certainly, these numbers only describe metrics in order to compare different configurations on a cost level. They allow us to determine which configuration is economically superior in the hypothetical case of a perfect horizontal scaling possibility of the respective configuration without costs for overhead. In order to have a final value that tells us actually a specific comparison value, \( O_{US} \), \( O_{UT} \) and \( O_{RSFL} \) are expressed as percentages of the respective maximum in the current test set. They are then equally weighted summed up and the arithmetical mean is computed, which gives results in a number from 0 to 1, with 1 as the best possible value. This gives us an economical percentual value, by which we can easily determine the best economical choice:

\[
O_{TOTAL} = \frac{O_{US\_current}}{O_{US\_MAX}} + \frac{O_{UT\_current}}{O_{UT\_MAX}} + \frac{O_{RSFL\_current}}{O_{RSFL\_MAX}}
\]

This metric can of course only be used to compare test results of the same test set, while using the same maximum numbers for determining the ratios.

4.2 Utilized Tools

In order to perform tests, measure the impact of changes, modify configurations and monitor the system’s behavior, the following software and tools have been used.

4.2.1 Usage Analysis

Since the regarded system is online and actively used, it already has existing usage data. For the design of realistic tests, the average user’s behavior has been analyzed using the SaaS of Google Analytics\(^{21}\) and New Relic\(^{22}\). These tools give a very detailed insight into the sequence, frequency and response times of the actions the users perform.

4.2.2 System Analysis

For the monitoring and analysis of certain user behaviors’ impact on the system, as well as the tests’ effect on the servers’ load, the used tools are New Relic,\(^{21}\) https://www.google.com/analytics/  
\(^{22}\) https://newrelic.com/
Nagios\(^{23}\), AWS CloudWatch\(^{24}\) and the Linux command line tools HTop\(^{25}\) and Iotop\(^{26}\).

### 4.2.3 Testing

For the tests, Apache’s JMeter\(^{27}\) has been used in order to simulate the users’ actual behavior. The software can simulate increasing and decreasing client numbers, as well as provide elaborated statistics of a performed tests, using the plugins ExtraSet\(^{28}\) and StandardSet\(^{29}\). For the database benchmarks, the tool sysbench\(^{30}\) has been used in its version 0.4.12.

### 4.2.4 Miscellaneous

Some other software that earned its right to be mentioned in this document for it enabling me to do all this work:

- Debian\(^{31}\): The servers’ operating system
- Ubuntu Mate\(^{32}\): The author’s operating system
- Microsoft Windows\(^{33}\): The author’s second operating system
- LibreOffice\(^{34}\) Writer/Calc: For tables, calculations and diagrams
- Microsoft Office\(^{35}\) Excel/Word: Same purpose
- Shutter\(^{36}\): For copy/pasting diagrams and saving them as image files.
- LaTeX, MiKTeX\(^{37}\) and Texmaker\(^{38}\): For making this document look neat.
- Vim-Nox\(^{39}\): For being awesome

### 4.3 Procedure

The procedure consists of multiple steps. It will involve measures to improve the performance and the scalability, which are the main goals of this project.

\(^{23}\)https://www.nagios.org/
\(^{24}\)https://aws.amazon.com/cloudwatch/
\(^{25}\)http://hisham.hm/htop/
\(^{26}\)http://guichaz.free.fr/iotop/
\(^{27}\)https://jmeter.apache.org/
\(^{28}\)http://jmeter-plugins.org/wiki/ExtrasSet/
\(^{29}\)http://jmeter-plugins.org/wiki/StandardSet/
\(^{30}\)https://github.com/akopytov/sysbench
\(^{31}\)https://www.debian.org/
\(^{32}\)https://ubuntu-mate.org/about/
\(^{33}\)https://www.microsoft.com/en-us/windows
\(^{34}\)https://www.libreoffice.org/
\(^{35}\)https://products.office.com/en-us/home
\(^{36}\)http://shutter-project.org/
\(^{37}\)http://miktex.org/
\(^{38}\)http://www.xm1math.net/texmaker/
\(^{39}\)http://packages.ubuntu.com/precise/vim-nox
1. Before anything can be tested, the tests have to be created. Therefore the existing user data is analyzed and appropriate tests are designed which can simulate actual user behavior. The tests will be performed as a stress test by simulating vast amounts of concurrently active users until the server becomes overloaded.

2. The second step focuses on the various possible hardware configurations for the monolithic architecture. The goal is to determine the economically best possible hardware, and to find a reference value for all future modifications on the system which will allow an easy comparison towards the impact’s extent of the respective undertaken changes.

3. As a third step, the most urgent issue will be dealt with: the upper limit of 200-300 CAU, which is seemingly independent from the underlying hardware. When scaled up vertically, the system persists in not being able to handle the amount of requests. Investigation on server side and respective counter measures will be undertaken.

4. After the previous issue turns out to be solvable, the focus will be put onto the optimization of the web server configuration in order to improve the performance of the software in terms of thread handling and computation of requests. The various possible configurations will be tested on all available and reasonable hardware configurations for enabling an elaborated and complete comparison to the initial configuration.

5. After finding the optimal web server configuration, the code will be examined and optimized in terms of caching in order to reduce the computations on the database and application servers.

6. When an optimal architecture and configuration is found, the system will be distributed in order to enable horizontal scaling and to profit from the other advantages that come with it. The distributed system will again be optimized in terms of the respectively most cost efficient hardware configuration for each component and the costs of the distributed system will be put into relation to the maintainable CAU numbers.

7. Due to the nature of a monolithic system, the database as a single component will again face the drawbacks, which the entire initial system was exposed to, including the inability of horizontal scaling. Therefore possible solutions will be investigated and tested in terms of performance, stability and economical efficiency.

8. As a last step, the results of the previous measures will be compared, analyzed and evaluated in order to determine the best possible architecture, hardware and software configuration.
5 Tests, Results

The Results of the procedure are presented in this chapter. Due to the nature of this research approach, the respective following steps depend on the results and interpretation of the previous steps. Therefore the discussion and evaluation will be presented respectively after each step’s data analysis and aggregation at the end.

5.1 Design of Tests

When dealing with load tests, there is a huge variety of tools, which can ease the designing and performing of such tests. For instance, Italian researchers have developed a set of tools called WALTy (Ruffo et al., 2004), which facilitate what-if analyses of possible future user behavior based on Customer Behavior Model Graphs (CBMG). Other tools, such as Specweb2005\textsuperscript{40} and WebStone\textsuperscript{41} offer the possibility to perform standardized benchmarking tests of web servers. Yet, such software is not applicable in this context, since for one, exactly ordered client behavior such as depicted in CBMGs is not of importance in this scope and would reasonlessly complicate the testing. Standardized benchmarking tests are also not directly relevant due to the fact that the already hardware-optimized AWS instances will be used. Thus, the testing in this project bases on actual user behavior of the very website that is to be optimized. The behavior has been captured and replayed, where parameters have been randomized and modified in order to simulate varying user behavior.

For simulating the user behavior, the software JMeter is being used. It allows to generate pre-defined HTTP requests with dynamic payload, randomized orders, multi-threaded concurrent simulation of many users, and the graphical and tabular presentation of results in form of diagrams and CSV files.

In order to simulate user behavior, it must of course be analyzed first. According to our results collected by Google Analytics, the average user performs one manual action per minute which results in one server request per minute that is being sent to the API.

The analysis of the code and system’s behavior revealed that the client software that runs in the users’ respective browser performs automatic polls of user data every ten seconds additionally to the manual actions of the users. This results in an average of 7 requests to the server per minute per user.

The transactions invoked by the users and processed on server side have been statistically recorded by the used SaaS NewRelic. They show the distribution of user requests and their resulting load on the server (See Figure 9 and 10).

As can be seen in the results of this analysis, the load is mostly generated by the controller method “user@show”, which is a result of the continuous automatic polling of the user data. For the accurate simulation of the mean user

\textsuperscript{40}https://www.spec.org/web2005/
\textsuperscript{41}http://www.mindcraft.com/webstone/
behavior, though, the distribution of requests rather than the resulting load is the important factor (See Fig. 10).

For the simulation of the requests, the configuration of JMeter is set in a way such that the distribution of requests is reflected by a randomized weighted choice of actions each thread can perform. Essentially, each thread simulating one user iterates over the list of possible actions and performs them with their respective probability determined by the mean distribution of requests (See Tab. 1). Statistically this is a highly accurate way of simulating real behavior, which is ensured by performing and recording actual requests and repeating those in the tests (Halili, 2008). In this project, requests with a request ratio of less than 0.1% have been disregarded due to lack of relevance.

The number of threads is controlled by a stepping thread group, which increases the number of simulated users by one per second until a pre-set threshold is reached which is chosen to be high enough to max out the platform’s performance capabilities. Subsequently the number of threads is firstly held constant for a pre-set time and then reduced in the same manner as increased before.

The results collected by JMeter which are important in the scope of this research include:
• The number of threads over time
• The number of received responses per second over time
• The average response time over time
• The distribution of different HTTP response codes over time

The response codes tell us if a request has been successful, if a possible client side error occurred or if the server did not answer or could not process the respective request.

For the testing, one machine alone can barely simulate the vast amount of requests, which can be generated by thousands of users. Therefore, Apache JMeter offers the possibility to be run in server mode without a GUI. This provides the additional capability of running distributed load tests with many testing nodes and one master node that aggregates the results and triggers the commands to the nodes using RMI (Nevedrov, 2006). In the scope of this thesis, ten testing nodes with one master node have been set up and used, running on AWS server instances.
<table>
<thead>
<tr>
<th>Action</th>
<th># Reqs</th>
<th>% Reqs</th>
<th>Load (ms)</th>
<th>Load (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/1.0.users.show</td>
<td>208655.00</td>
<td>89.70%</td>
<td>47999999</td>
<td>87.9524%</td>
</tr>
<tr>
<td>/1.0.lessons.show</td>
<td>5008.00</td>
<td>2.15%</td>
<td>564000</td>
<td>1.0334%</td>
</tr>
<tr>
<td>/1.0.quizReports.store</td>
<td>3861.00</td>
<td>1.66%</td>
<td>1570000</td>
<td>2.8768%</td>
</tr>
<tr>
<td>/1.0.quizzes.show</td>
<td>3274.00</td>
<td>1.41%</td>
<td>16000000</td>
<td>2.9317%</td>
</tr>
<tr>
<td>/1.0.subjects.index</td>
<td>2589.00</td>
<td>1.11%</td>
<td>433000</td>
<td>0.7934%</td>
</tr>
<tr>
<td>/1.0.videoReports.store</td>
<td>2529.00</td>
<td>1.09%</td>
<td>213000</td>
<td>0.3903%</td>
</tr>
<tr>
<td>/1.0.subjects.index</td>
<td>1010.00</td>
<td>0.43%</td>
<td>691000</td>
<td>1.2661%</td>
</tr>
<tr>
<td>/1.0.lessons.index</td>
<td>981.00</td>
<td>0.42%</td>
<td>265000</td>
<td>0.4856%</td>
</tr>
<tr>
<td>/1.0.schoools.show</td>
<td>922.00</td>
<td>0.40%</td>
<td>199000</td>
<td>0.3646%</td>
</tr>
<tr>
<td>/1.0.courses.index</td>
<td>891.00</td>
<td>0.38%</td>
<td>55500</td>
<td>0.1017%</td>
</tr>
<tr>
<td>/1.0.schoolClasses.show</td>
<td>744.00</td>
<td>0.32%</td>
<td>671000</td>
<td>1.2295%</td>
</tr>
<tr>
<td>/1.0.tasks.index</td>
<td>298.00</td>
<td>0.13%</td>
<td>73600</td>
<td>0.1349%</td>
</tr>
<tr>
<td>/index.php</td>
<td>285.00</td>
<td>0.12%</td>
<td>44400</td>
<td>0.0814%</td>
</tr>
<tr>
<td>/1.0.subjects.show</td>
<td>277.00</td>
<td>0.12%</td>
<td>23300</td>
<td>0.0427%</td>
</tr>
<tr>
<td>/1.0.schoools.index</td>
<td>272.00</td>
<td>0.12%</td>
<td>32300</td>
<td>0.0592%</td>
</tr>
<tr>
<td>/1.0.courses.show</td>
<td>263.00</td>
<td>0.11%</td>
<td>32200</td>
<td>0.0590%</td>
</tr>
<tr>
<td>/1.0.users.index</td>
<td>262.00</td>
<td>0.11%</td>
<td>31200</td>
<td>0.0572%</td>
</tr>
<tr>
<td>/1.0.disciplines.show</td>
<td>233.00</td>
<td>0.10%</td>
<td>17300</td>
<td>0.0317%</td>
</tr>
</tbody>
</table>

Table 1: Request and Load Distribution
(for a complete list, see appendix)

The payload that is chosen for the respective requests is not static but depends on the number of simulated users in that particular test. The reason is a caching that is implemented in the code. Although faintly used, the caching could highly affect the test results. If all simulated users would perform the same requests, the probability of being provided with a cached response is much higher compared to randomly chosen requests. Therefore, and due to the fact that a later step is the actual change of code in the favor of caching, the payload is randomly chosen out of a set of possibilities whose size depends on the simulated number of users (See Tab. 2).

Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRV</td>
<td>Number of active test nodes</td>
</tr>
<tr>
<td>USR</td>
<td>Number of tested users per server</td>
</tr>
<tr>
<td>LPS</td>
<td>Number of lessons per session</td>
</tr>
<tr>
<td>UPG</td>
<td>Number of users per group (or classroom)</td>
</tr>
<tr>
<td>OCF</td>
<td>Overlapping content factor (how many groups do same lessons in an overlapping session)</td>
</tr>
</tbody>
</table>

The number of active test nodes varies, depending on the performance of the respective configuration that is to be stress tested. E.g. a configuration with low-end hardware has been tested with fewer requests than configurations which turned out to be able to handle many hundreds of users. The number of tested users per test node has been set to 200-400, due to networking constraints of the testing nodes. The average number of lessons per session has been set to 5, which is a result of user surveys. The average number of students in one group or classroom has been set to 20 and the overlapping content factor to 3, which
<table>
<thead>
<tr>
<th>Payload</th>
<th>Number of different data sets</th>
<th>Controller Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>userIDs</td>
<td>( \text{USR, SRV} )</td>
<td>UserIDs@show</td>
</tr>
<tr>
<td>LessonIDs</td>
<td>( \text{USR, SRV, LPS, UPG, OCF} ), because the number of lessons per session, users per group and the overlapping content across groups has to be taken into account</td>
<td>Lessons@show, Quizzes@show</td>
</tr>
<tr>
<td>SubjectIDsSets</td>
<td>All there are, because there are only 47 and several are called per user per session</td>
<td>Subjects@index</td>
</tr>
<tr>
<td>SchoolIDs</td>
<td>( \text{USR, SRV, UPG} ), because one classroom registers at the same time, tops (which is when this is called)</td>
<td>Schools@show</td>
</tr>
<tr>
<td>CourseIDsSets</td>
<td>All there are, because there are only 4</td>
<td>Courses@index</td>
</tr>
<tr>
<td>ClassIDs</td>
<td>( \text{USR, SRV} ), Assuming max one entire class registers at once</td>
<td>Classes@show</td>
</tr>
<tr>
<td>SubjectIDs</td>
<td>All of them, because they will not be cached, so it makes no difference which one</td>
<td>Subjects@show</td>
</tr>
<tr>
<td>CourseIDs</td>
<td>All there are, because there are only 4</td>
<td>Courses@show</td>
</tr>
<tr>
<td>SearchKeywords</td>
<td>All of them, because there is no caching strategy for the school search yet. Data is too non-static</td>
<td>Schools@index</td>
</tr>
<tr>
<td>UserIDSets</td>
<td>All of them, because they will not be cached, so it makes no difference which one</td>
<td>Users@index</td>
</tr>
<tr>
<td>DisciplineIds</td>
<td>All there are, because there are only 8</td>
<td>Disciplines@show</td>
</tr>
</tbody>
</table>

Table 2: Variety Distribution of Payloads

bases on the assumption that every lesson is watched in average by three classes within a reasonable caching time.

Since each thread simulates one user and iterates over the list of possible actions, the thread will be put to sleep after each iteration for 8.5 seconds. This is done due to the fact that in average every client sends in average seven requests per minute.

For the testing of different hardware configurations, the system will be deployed on all possible AWS instance types, which are reasonable in this scope. Those have been chosen to be the general purpose computation-optimized instance types. They all offer a different virtualized hardware and thus show different results in the tests. The instance types can be categorized into four basic sets (Amazon, 2015):

- M3: Based on Intel Xeon E5-2670 v2 (Ivy Bridge) processors with SSD-backed storage
<table>
<thead>
<tr>
<th>Instance Type</th>
<th>#vCPUs (Clock)</th>
<th>Mem</th>
<th>Storage (GB)</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>m3.medium</td>
<td>1 (2.5 GHz)</td>
<td>3.75 GB</td>
<td>1 x 4 SSD</td>
<td>Moderate</td>
</tr>
<tr>
<td>m3.large</td>
<td>2 (2.5 GHz)</td>
<td>7.5 GB</td>
<td>1 x 32 SSD</td>
<td>Moderate</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4 (2.5 GHz)</td>
<td>15 GB</td>
<td>2 x 40 SSD</td>
<td>High</td>
</tr>
<tr>
<td>m3.2xlarge</td>
<td>8 (2.5 GHz)</td>
<td>30 GB</td>
<td>2 x 80 SSD</td>
<td>High</td>
</tr>
<tr>
<td>m4.large</td>
<td>2 (2.4 GHz)</td>
<td>8 GB</td>
<td>EBS Only</td>
<td>Moderate</td>
</tr>
<tr>
<td>m4.xlarge</td>
<td>4 (2.4 GHz)</td>
<td>16 GB</td>
<td>EBS Only</td>
<td>High</td>
</tr>
<tr>
<td>m4.2xlarge</td>
<td>8 (2.4 GHz)</td>
<td>32 GB</td>
<td>EBS Only</td>
<td>High</td>
</tr>
<tr>
<td>m4.4xlarge</td>
<td>16 (2.4 GHz)</td>
<td>64 GB</td>
<td>EBS Only</td>
<td>High</td>
</tr>
<tr>
<td>m4.10xlarge</td>
<td>40 (2.4 GHz)</td>
<td>160 GB</td>
<td>EBS Only</td>
<td>10 Gigabit</td>
</tr>
<tr>
<td>c4.large</td>
<td>2 (2.9 GHz)</td>
<td>3.75 GB</td>
<td>EBS Only</td>
<td>Moderate</td>
</tr>
<tr>
<td>c4.xlarge</td>
<td>4 (2.9 GHz)</td>
<td>7.5 GB</td>
<td>EBS Only</td>
<td>High</td>
</tr>
<tr>
<td>c4.2xlarge</td>
<td>8 (2.9 GHz)</td>
<td>15 GB</td>
<td>EBS Only</td>
<td>High</td>
</tr>
<tr>
<td>c4.4xlarge</td>
<td>16 (2.9 GHz)</td>
<td>30 GB</td>
<td>EBS Only</td>
<td>High</td>
</tr>
<tr>
<td>c4.8xlarge</td>
<td>36 (2.9 GHz)</td>
<td>60 GB</td>
<td>EBS Only</td>
<td>10 Gigabit</td>
</tr>
<tr>
<td>c3.large</td>
<td>2 (2.8 GHz)</td>
<td>3.75 GB</td>
<td>2 x 16 SSD</td>
<td>Moderate</td>
</tr>
<tr>
<td>c3.xlarge</td>
<td>4 (2.8 GHz)</td>
<td>7.5 GB</td>
<td>2 x 40 SSD</td>
<td>Moderate</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>8 (2.8 GHz)</td>
<td>15 GB</td>
<td>2 x 80 SSD</td>
<td>High</td>
</tr>
<tr>
<td>c3.4xlarge</td>
<td>16 (2.8 GHz)</td>
<td>30 GB</td>
<td>2 x 160 SSD</td>
<td>High</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>32 (2.8 GHz)</td>
<td>60 GB</td>
<td>2 x 320 SSD</td>
<td>10 Gigabit</td>
</tr>
</tbody>
</table>

Table 3: Tested AWS Instance Types (Amazon, 2015)

- M4: Based on Intel Xeon E5-2676 v3 (Haswell) processors with support for enhanced networking
- C3: Based on Intel Xeon E5-2680 v2 (Ivy Bridge) processors with SSD-backed storage
- C4: Based on Intel Xeon E5-2666 v3 (Haswell) processors with support for enhanced networking

All categories have multiple performance levels, which are all tested for the performance-related and economical comparison. Based on the implementation, configuration and type of applications that are run on a server, a vertical scaling may be economically more efficient than the pure horizontal scaling on the lowest performance level. Thus, the economically most optimal instance type has to be found, which can then on demand be scaled up horizontally. The hardware configuration of the different instance types depends on the level (medium, large, xlarge, etc) and on the previously mentioned category (See Tab. 3).

These 19 instance types have to be tested for the highest efficiency regarding the related platform that runs on it.

5.2 Initial Monolithic Performance

The initial configuration that is currently used on the live system has been tested and is being evaluated in this chapter. The results are used as a baseline
<table>
<thead>
<tr>
<th>Instance</th>
<th>Uₚ</th>
<th>UₚT</th>
<th>RSFL</th>
<th>USD/h</th>
<th>OₚUS</th>
<th>OₚUT</th>
<th>OₚRSFL</th>
<th>OₚTOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>m3.medium</td>
<td>40</td>
<td>110</td>
<td>9</td>
<td>$0.079</td>
<td>506.33</td>
<td>1392.41</td>
<td>113.92</td>
<td>59.174%</td>
</tr>
<tr>
<td>m3.large</td>
<td>195</td>
<td>245</td>
<td>20</td>
<td>$0.158</td>
<td>1234.18</td>
<td>1550.63</td>
<td>126.58</td>
<td>80.248%</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>200</td>
<td>270</td>
<td>22</td>
<td>$0.315</td>
<td>634.92</td>
<td>857.14</td>
<td>69.84</td>
<td>43.314%</td>
</tr>
<tr>
<td>m3.2xlarge</td>
<td>205</td>
<td>275</td>
<td>22</td>
<td>$0.632</td>
<td>324.37</td>
<td>435.13</td>
<td>34.81</td>
<td>21.895%</td>
</tr>
<tr>
<td>m4.large</td>
<td>190</td>
<td>260</td>
<td>21</td>
<td>$0.150</td>
<td>1266.67</td>
<td>1733.33</td>
<td>140.00</td>
<td>86.956%</td>
</tr>
<tr>
<td>m4.xlarge</td>
<td>215</td>
<td>265</td>
<td>22</td>
<td>$0.300</td>
<td>716.67</td>
<td>883.33</td>
<td>73.33</td>
<td>46.268%</td>
</tr>
<tr>
<td>m4.2xlarge</td>
<td>210</td>
<td>270</td>
<td>22</td>
<td>$0.600</td>
<td>350.00</td>
<td>450.00</td>
<td>36.67</td>
<td>23.098%</td>
</tr>
<tr>
<td>m4.4xlarge</td>
<td>205</td>
<td>265</td>
<td>23</td>
<td>$1.200</td>
<td>170.83</td>
<td>220.83</td>
<td>19.17</td>
<td>11.566%</td>
</tr>
<tr>
<td>m4.10xlarge</td>
<td>200</td>
<td>275</td>
<td>22</td>
<td>$3.000</td>
<td>66.67</td>
<td>91.67</td>
<td>7.33</td>
<td>4.577%</td>
</tr>
<tr>
<td>c4.large</td>
<td>210</td>
<td>250</td>
<td>22</td>
<td>$0.141</td>
<td>1489.36</td>
<td>1773.05</td>
<td>156.03</td>
<td>95.851%</td>
</tr>
<tr>
<td>c4.xlarge</td>
<td>215</td>
<td>265</td>
<td>22</td>
<td>$0.281</td>
<td>765.12</td>
<td>943.06</td>
<td>78.29</td>
<td>49.397%</td>
</tr>
<tr>
<td>c4.2xlarge</td>
<td>220</td>
<td>270</td>
<td>22</td>
<td>$0.562</td>
<td>391.46</td>
<td>480.43</td>
<td>39.15</td>
<td>25.043%</td>
</tr>
<tr>
<td>c4.4xlarge</td>
<td>225</td>
<td>270</td>
<td>23</td>
<td>$1.125</td>
<td>200.00</td>
<td>240.00</td>
<td>20.44</td>
<td>12.796%</td>
</tr>
<tr>
<td>c4.8xlarge</td>
<td>225</td>
<td>275</td>
<td>22</td>
<td>$2.250</td>
<td>100.00</td>
<td>122.22</td>
<td>9.78</td>
<td>6.341%</td>
</tr>
<tr>
<td>c5.large</td>
<td>200</td>
<td>250</td>
<td>20</td>
<td>$0.129</td>
<td>1550.39</td>
<td>1937.98</td>
<td>155.04</td>
<td>99.789%</td>
</tr>
<tr>
<td>c5.xlarge</td>
<td>225</td>
<td>255</td>
<td>22</td>
<td>$0.516</td>
<td>426.36</td>
<td>523.26</td>
<td>42.64</td>
<td>27.275%</td>
</tr>
<tr>
<td>c5.2xlarge</td>
<td>220</td>
<td>270</td>
<td>22</td>
<td>$1.032</td>
<td>222.87</td>
<td>266.47</td>
<td>21.32</td>
<td>13.929%</td>
</tr>
<tr>
<td>c5.4xlarge</td>
<td>240</td>
<td>275</td>
<td>22</td>
<td>$2.064</td>
<td>92.05</td>
<td>130.81</td>
<td>10.66</td>
<td>6.506%</td>
</tr>
</tbody>
</table>

| Maximum    | 1550.39 | 1937.98 | 156.03 |

Table 4: Test Results of Initial Configuration

for determining the extent of a modification’s performance increase and will be used for a final comparison in terms of economical impact and overall improvement of the system.

The initial configuration of the system has been tested with up to 600 simulated users only, due to the fact, that a known limit of 200-300 users could be handled at the current state. The results of the tests prove that the limit is actually independent from the underlying hardware.

The platform has been tested on all 19 available hardware options on AWS. The results show an increase of the performance when vertically scaled up in terms of computation power and available working memory. The cheapest and computationally weakest instance (m3.medium) could handle a minimum of 40 concurrent users with a satisfactory RT and respectively 110 concurrent users with a tolerating RT. The server could only compute an average of 9 responses per second. After doubling the hardware power (m3.large), the platform was able to more than double its UT and RSFL and even multiply its US even fivefold. Yet, after increasing the hardware performance again by doubling it (m3.xlarge), the performance increase was only slight and barely noticeable. All other tested instance types show a similar performance, independently from the increase in computation power. (See Table 4)

Considering the results, it becomes obvious, that the c3.large instance is currently the best choice in terms of economical efficiency. Although other instance types perform slightly better in total, the respective price decreases their economical value significantly.
In any case, the results clearly show the inability of the system to scale after reaching the value of about 22 responses under full load. Additionally to the valid responses, the graph (See Fig. 11) shows a number of invalid responses, such as SocketExceptions, SocketTimeouts or ConnectExceptions, which indicate faulty answers, no answers at all or even the client’s incapability to connect to the server in the first place.

5.3 Upper User Limit

The previous test clearly depicts the problem of the upper CAU limit. In case the number of concurrent users rises over about 250, the system will become slow or even unusable. Therefore, measures have to be taken in order to remove this limit and as a result enable scalability of the system in the first place. Therefore, the server configuration has been examined and modified.

The Apache web server software provides many modules for different purposes. When it comes to the handling of client requests, the software uses so called Multi-Processing Modules (MPM). It deals with the process and thread handling, assigns tasks to threads and handles open connections and queued requests. As per default, the software came with the Prefork-MPM pre-installed.

Prefork is mostly designed for applications with non-thread-safe libraries. Here, one parent process manages the pool of server processes, which in turn handle incoming requests (TheApacheSoftwareFoundation, 2015b). Yet, this module does not utilize the multi-threading capabilities of the underlying hardware at all but instead uses processes with much more overhead than threads. Memory limits can be exceeded more quickly and the CPU has to compute more overhead. Especially since the default configuration assumes a low-end hardware,
the number of processes which handle the requests is highly limited. As the tests show, the server hardware was never fully exploited to any time using the default configuration. The CPU as well as the working memory were mostly idle (See Fig. 12). This is the first point of modification, which is supposed to increase the system’s performance by enabling the web server to use more processes (See Fig. 13).
Therefore the number of maximum used processes has been increased to an extent, which is appropriate for high-end servers. Significantly increasing the potential of the server enabled the effective use of the underlying hardware (See Fig. 14) and additionally, unambiguously increased the number of concurrent clients (See Tab. 5). After applying the configuration modifications, the upper user limit is now depending on the used hardware instead of a fixed limit of about 250. This clearly provides a solution of the previously existing problem of not being able to scale.

![CPU usage](image1)

![Physical memory](image2)

![Disk I/O utilization](image3)

![Network I/O (Mb/s)](image4)

Figure 14: Hardware Utilization with Optimized Prefork Config (c4.xlarge)

The clearly best choice in terms of economical efficiency would be the c4.xlarge instance type, when a hypothetically flawless horizontal scaling were possible. The $O_{UT}$ value is by far the highest among all tested instance types and the and $O_{RSFL}$ value is only outperformed by c3.large. The $O_{TOTAL}$ value gives us a percentual value, which aggregates all three other values into one, by giving each of them a percentual value against the maximum in its column, then adding the three values and building the arithmetical mean. Here, c4.xlarge has an $O_{TOTAL}$ value of over 83%, which depicts its superiority in this configuration.

5.4 Further Web Server Optimization

Now being able to scale the system up, even though only vertically possible at the current state, we dove deeper into the configuration and optimization of the Apache server. Being able to scale vertically was only the first step in using the hardware potential.
Table 5: Test Results of optimized Prefork-MPM

An additional problem to the inability of leveraging multi-threading with the existing solution based on an optimized Prefork is it being prone to deliver HTTP-Errors in the form of socket and connection errors when it starts to get overloaded and is no longer able to handle the current amount of requests (See Fig. 15). In such a case, the server starts to drop requests and refuse connections. This leads to a bad user experience, since dropped requests are worse than longer response time. Dropped requests should be avoided, if possible and reasonable.

Further investigation showed that the MPM can be changed with an entirely different implementation of thread handling. The Apache Worker-MPM can be optimized in a much better way (Liu et al., 2003). It implements a hybrid multi-process multi-threaded server. Threads serve requests and are controlled by multiple server processes, each with many threads, which retain a similarly stable environment as a process-based server (TheApacheSoftwareFoundation, 2015c). However, the Worker-MPM does not run on the conventional PHP5-Apache-module. Therefore, the entire application server has been switched from MPM-Prefork with the PHP5-module to the Worker-MPM using the FastCGI Process Manager for PHP (PHP-FPM).

The default configuration of the Worker-MPM unfortunately did not bring the expected increase in performance or scalability, but quite the opposite. Also not solving the problem of refused client requests, the performance was back on a similar level compared to the initial configuration, implementing a user limit of about 250 with the difference that the lowest hardware configuration could handle almost double as many users with a satisfactory RT. But when scaling...
up, the numbers stayed very similar to the initial results and the scalability was again not possible. Therefore, the Worker-MPM configuration has been analyzed and adapted to a high-end server’s needs (See Fig. 16), which ultimately involved again significantly increasing the number of possible concurrent processes and threads, as well as the number of threads a process can hold and requests a thread can process. This again resulted in a significant performance boost as well as again in a vertical scalability (See Tab. 6).

Since the system is now using a configuration based on the Worker-MPM, the focus has to be put onto the other part, besides the MPM, which is the PHP-FPM configuration. Specifically the thread handling has been modified until an optimal thread/process ratio has been found and the number of requests per thread was ideal (See Fig. 17).

The result was a configuration that was not only able to handle a manyfold of the initial number of users, but also solved the problem of rejecting connections,
<table>
<thead>
<tr>
<th>Instance</th>
<th>Us</th>
<th>U_T</th>
<th>RSFL</th>
<th>USD/h</th>
<th>O_US</th>
<th>O_UT</th>
<th>O_RSFL</th>
<th>O_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>m3.medium</td>
<td>33</td>
<td>88</td>
<td>10</td>
<td>$0.079</td>
<td>417.72</td>
<td>1113.92</td>
<td>126.58</td>
<td>42.034%</td>
</tr>
<tr>
<td>m3.large</td>
<td>120</td>
<td>270</td>
<td>23</td>
<td>$0.158</td>
<td>759.49</td>
<td>1708.86</td>
<td>145.57</td>
<td>59.814%</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>350</td>
<td>460</td>
<td>45</td>
<td>$0.315</td>
<td>1111.11</td>
<td>1460.32</td>
<td>142.86</td>
<td>63.443%</td>
</tr>
<tr>
<td>m3.2xlarge</td>
<td>605</td>
<td>835</td>
<td>72</td>
<td>$0.632</td>
<td>957.28</td>
<td>1321.20</td>
<td>113.92</td>
<td>54.150%</td>
</tr>
<tr>
<td>m4.large</td>
<td>200</td>
<td>325</td>
<td>28</td>
<td>$0.150</td>
<td>1333.33</td>
<td>1491.67</td>
<td>143.33</td>
<td>62.830%</td>
</tr>
<tr>
<td>m4.xlarge</td>
<td>445</td>
<td>590</td>
<td>56</td>
<td>$0.300</td>
<td>1483.33</td>
<td>941.67</td>
<td>87.50</td>
<td>40.843%</td>
</tr>
<tr>
<td>m4.2xlarge</td>
<td>635</td>
<td>895</td>
<td>86</td>
<td>$0.600</td>
<td>1108.86</td>
<td>1491.67</td>
<td>143.33</td>
<td>62.830%</td>
</tr>
<tr>
<td>m4.4xlarge</td>
<td>895</td>
<td>1130</td>
<td>105</td>
<td>$1.200</td>
<td>745.83</td>
<td>941.67</td>
<td>87.50</td>
<td>40.843%</td>
</tr>
<tr>
<td>m4.10xlarge</td>
<td>760</td>
<td>1110</td>
<td>96</td>
<td>$3.000</td>
<td>253.33</td>
<td>370.00</td>
<td>32.00</td>
<td>14.872%</td>
</tr>
<tr>
<td>c4.large</td>
<td>215</td>
<td>355</td>
<td>32</td>
<td>$0.141</td>
<td>1524.82</td>
<td>2517.73</td>
<td>226.95</td>
<td>98.406%</td>
</tr>
<tr>
<td>c4.xlarge</td>
<td>450</td>
<td>590</td>
<td>60</td>
<td>$0.281</td>
<td>1601.42</td>
<td>2099.64</td>
<td>213.52</td>
<td>92.493%</td>
</tr>
<tr>
<td>c4.2xlarge</td>
<td>765</td>
<td>1110</td>
<td>98</td>
<td>$0.562</td>
<td>1361.21</td>
<td>1975.09</td>
<td>174.38</td>
<td>80.094%</td>
</tr>
<tr>
<td>c4.4xlarge</td>
<td>990</td>
<td>1380</td>
<td>119</td>
<td>$1.125</td>
<td>880.00</td>
<td>1226.67</td>
<td>105.78</td>
<td>50.094%</td>
</tr>
<tr>
<td>c4.8xlarge</td>
<td>720</td>
<td>950</td>
<td>101</td>
<td>$2.250</td>
<td>320.00</td>
<td>422.22</td>
<td>44.89</td>
<td>18.844%</td>
</tr>
<tr>
<td>c5.large</td>
<td>205</td>
<td>220</td>
<td>24</td>
<td>$0.129</td>
<td>1589.15</td>
<td>1705.43</td>
<td>186.05</td>
<td>82.982%</td>
</tr>
<tr>
<td>c5.xlarge</td>
<td>360</td>
<td>490</td>
<td>48</td>
<td>$0.258</td>
<td>1395.35</td>
<td>1899.22</td>
<td>186.05</td>
<td>81.514%</td>
</tr>
<tr>
<td>c5.2xlarge</td>
<td>685</td>
<td>885</td>
<td>80</td>
<td>$0.516</td>
<td>1327.52</td>
<td>1656.98</td>
<td>155.04</td>
<td>72.341%</td>
</tr>
<tr>
<td>c5.4xlarge</td>
<td>720</td>
<td>990</td>
<td>98</td>
<td>$1.032</td>
<td>697.67</td>
<td>959.30</td>
<td>94.96</td>
<td>41.170%</td>
</tr>
<tr>
<td>c5.8xlarge</td>
<td>730</td>
<td>960</td>
<td>104</td>
<td>$2.064</td>
<td>353.68</td>
<td>465.12</td>
<td>50.39</td>
<td>20.920%</td>
</tr>
</tbody>
</table>

| Maximum   | 1601.42 | 2517.73 | 226.95 |

Table 6: Test Results of optimized Worker-MPM Configuration with a default PHP-FPM Configuration

![Figure 17: Change of PHP-FPM Config](image)

when the hardware was overloaded. As can be seen, the performance of the server greatly improved and the economic capabilities increased significantly (see Tab. 7). Although not showing significant performance improvements compared to the set-up with an optimized Prefork configuration, the improved user experience with respect to the now not-rejected connections makes this configuration a considerably better choice compared to the previous one.

Since the system builds on a RESTful API, that does not render HTML and other visual data, and thus usually does not have to serve many requests per client in a short amount of time, the third and last available MPM has to be considered as well: the Event based MPM. It is based on the Worker MPM
but is designed to handle more requests to be served simultaneously. This can happen by offloading processing work to supporting threads, while the main threads can work on new requests. Using Event-MPM, all "left-open" connections, which are idly waiting for additional requests from the same client, are not blocking any threads that could otherwise do some actual computation work (TheApacheSoftwareFoundation, 2015a).

Again, following the previous testing work flow, the configuration provided by default has been tested and, as expected, it showed a much worse performance compared to the optimized configurations of Prefork or Worker. Therefore it has been optimized again for a high-end hardware (See Fig. 18) together with the already used PHP-FPM configuration which came to use in the test setup for the Worker module, the results prove the Apache Foundation’s claim of MPM being capable of handling more threads than the other two modules with solving the keep-alive problem.

The results of the last test show the capability of the system to handle up to 2750 concurrent users with just one single instance (c4.4xlarge) in a monolithic system. This increase allows the platform to be scaled up in a highly economic and efficient manner, when it comes to vertical scaling (See Tab: 8). The previously existing user limit has been pushed to an eight fold of its initial value. Even when expecting an exponential growth, the platform could be operated with a monolithic architecture for much longer than before.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Uₜ</th>
<th>Uₜ</th>
<th>RSFL</th>
<th>USD/h</th>
<th>Oₜ</th>
<th>Oₜ</th>
<th>Oₜ</th>
<th>Oₜ</th>
<th>Oₜ</th>
<th>Oₜ</th>
</tr>
</thead>
<tbody>
<tr>
<td>m3.medium</td>
<td>45</td>
<td>75</td>
<td>10</td>
<td>$0.079</td>
<td>569.62</td>
<td>949.37</td>
<td>126.58</td>
<td>39.828%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m3.large</td>
<td>220</td>
<td>250</td>
<td>22</td>
<td>$0.158</td>
<td>1592.28</td>
<td>139.24</td>
<td>62.709%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>380</td>
<td>530</td>
<td>45</td>
<td>$0.315</td>
<td>1206.35</td>
<td>142.86</td>
<td>61.694%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m3.2xlarge</td>
<td>700</td>
<td>950</td>
<td>85</td>
<td>$0.632</td>
<td>1107.59</td>
<td>134.49</td>
<td>56.582%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m4.large</td>
<td>205</td>
<td>320</td>
<td>28</td>
<td>$0.150</td>
<td>1592.28</td>
<td>139.24</td>
<td>62.709%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m4.xlarge</td>
<td>480</td>
<td>665</td>
<td>56</td>
<td>$0.300</td>
<td>1206.35</td>
<td>142.86</td>
<td>61.694%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m4.2xlarge</td>
<td>890</td>
<td>1180</td>
<td>112</td>
<td>$0.600</td>
<td>1107.59</td>
<td>134.49</td>
<td>56.582%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m4.4xlarge</td>
<td>1750</td>
<td>2350</td>
<td>210</td>
<td>$1.200</td>
<td>1458.33</td>
<td>175.00</td>
<td>73.920%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m4.10xlarge</td>
<td>1620</td>
<td>2150</td>
<td>178</td>
<td>$3.000</td>
<td>227.76</td>
<td>59.33</td>
<td>26.460%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c4.large</td>
<td>305</td>
<td>320</td>
<td>32</td>
<td>$0.141</td>
<td>2163.12</td>
<td>226.95</td>
<td>96.489%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c4.xlarge</td>
<td>530</td>
<td>705</td>
<td>64</td>
<td>$0.281</td>
<td>1886.12</td>
<td>227.76</td>
<td>95.497%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c4.2xlarge</td>
<td>1100</td>
<td>1420</td>
<td>128</td>
<td>$0.562</td>
<td>1957.30</td>
<td>227.76</td>
<td>96.828%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c4.4xlarge</td>
<td>230</td>
<td>250</td>
<td>23</td>
<td>$1.125</td>
<td>1733.33</td>
<td>209.78</td>
<td>89.661%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c4.8xlarge</td>
<td>1680</td>
<td>2200</td>
<td>188</td>
<td>$2.250</td>
<td>746.67</td>
<td>83.56</td>
<td>36.634%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c5.large</td>
<td>230</td>
<td>250</td>
<td>23</td>
<td>$0.129</td>
<td>1782.95</td>
<td>178.29</td>
<td>79.136%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c5.xlarge</td>
<td>1470</td>
<td>1960</td>
<td>180</td>
<td>$1.032</td>
<td>1424.42</td>
<td>174.42</td>
<td>72.532%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c5.2xlarge</td>
<td>230</td>
<td>250</td>
<td>23</td>
<td>$0.129</td>
<td>1782.95</td>
<td>178.29</td>
<td>79.136%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c5.4xlarge</td>
<td>800</td>
<td>980</td>
<td>90</td>
<td>$0.516</td>
<td>1550.39</td>
<td>174.42</td>
<td>72.532%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c5.8xlarge</td>
<td>1950</td>
<td>2650</td>
<td>225</td>
<td>$2.064</td>
<td>944.77</td>
<td>109.01</td>
<td>47.451%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Maximum | 2163.12 | 2526.69 | 227.76 |

Table 7: Test Results of optimized Worker-MPM Configuration with an optimized PHP-FPM
5.5 Code Optimization for Caching

Due to time constraints at the development phase, the code has initially not been optimized in terms of the reuse of already computed data. Therefore, the code improvement towards caching is considered to have quite some potential.

Caching can be useful especially considering the possibility of non-ideal code in which the developers did not make use of eager-loading (Hejlsberg et al., 2010) which resulted in an inefficient use of database queries and object computations (Bauer & King, 2006).

Regarding the API that is supposed to be improved, most data is not depending on session variables, setting the authentication and authorization aside. As a result, most retrieved data by clients is equal, independently from the user who requests the information. This, of course, only regards HTTP-GET requests and no modification requests (POST, PUT, PATCH, DESTROY). Therefore, we can assume that the caching of responses to such requests can highly decrease the load on the server side.

For the application of caching, the first step was creating a filter-class, that acts as a middleware for HTTP requests, implemented on the most efficient server configuration (MPM-Event). The filter is able to read, manipulate and respond to requests that are made to the server (See Fig. 19).

Essentially the now implemented cache filter receives a request and checks if the response to that request has already been stored earlier in the cache and returns that cached response if it exists. If not, the request is forwarded to the program that computes the response and returns it to the cache filter, which firstly stores it in the cache and then returns it to the requester.

Some requests cannot be cached, as they differ slightly from user to user or depending on the time of the request. However, parts of the computation can be cached, which is done in the object controllers.

The effect has been tested with the four instances which prove to be the most
<table>
<thead>
<tr>
<th>Instance</th>
<th>Uₚ</th>
<th>Uₜ</th>
<th>RSFL</th>
<th>USD/h</th>
<th>OₚS</th>
<th>OₜU</th>
<th>ØRSFL</th>
<th>ØTOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>m3.medium</td>
<td>75</td>
<td>100</td>
<td>10</td>
<td>$0.079</td>
<td>949.37</td>
<td>1265.82</td>
<td>126.58</td>
<td>49.829%</td>
</tr>
<tr>
<td>m3.large</td>
<td>225</td>
<td>240</td>
<td>22</td>
<td>$0.158</td>
<td>1424.05</td>
<td>1518.99</td>
<td>139.24</td>
<td>62.658%</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>420</td>
<td>530</td>
<td>46</td>
<td>$0.315</td>
<td>1333.33</td>
<td>1682.54</td>
<td>146.03</td>
<td>64.154%</td>
</tr>
<tr>
<td>m3.2xlarge</td>
<td>740</td>
<td>1020</td>
<td>90</td>
<td>$0.632</td>
<td>1170.89</td>
<td>1613.92</td>
<td>142.41</td>
<td>60.088%</td>
</tr>
<tr>
<td>m4.large</td>
<td>240</td>
<td>315</td>
<td>28</td>
<td>$0.150</td>
<td>1600.00</td>
<td>2100.00</td>
<td>186.67</td>
<td>79.654%</td>
</tr>
<tr>
<td>m4.xlarge</td>
<td>460</td>
<td>540</td>
<td>56</td>
<td>$0.300</td>
<td>1533.33</td>
<td>1800.00</td>
<td>186.67</td>
<td>74.876%</td>
</tr>
<tr>
<td>m4.2xlarge</td>
<td>870</td>
<td>1300</td>
<td>110</td>
<td>$0.600</td>
<td>1450.00</td>
<td>2166.67</td>
<td>183.33</td>
<td>77.494%</td>
</tr>
<tr>
<td>m4.4xlarge</td>
<td>1900</td>
<td>2500</td>
<td>210</td>
<td>$1.200</td>
<td>1583.33</td>
<td>2083.33</td>
<td>175.00</td>
<td>77.460%</td>
</tr>
<tr>
<td>m4.10xlarge</td>
<td>1600</td>
<td>2100</td>
<td>170</td>
<td>$3.000</td>
<td>533.33</td>
<td>700.00</td>
<td>56.67</td>
<td>25.735%</td>
</tr>
<tr>
<td>c4.large</td>
<td>280</td>
<td>340</td>
<td>32</td>
<td>$0.141</td>
<td>1985.82</td>
<td>2411.35</td>
<td>226.95</td>
<td>95.779%</td>
</tr>
<tr>
<td>c4.xlarge</td>
<td>500</td>
<td>680</td>
<td>60</td>
<td>$0.281</td>
<td>1779.36</td>
<td>2419.93</td>
<td>213.52</td>
<td>90.489%</td>
</tr>
<tr>
<td>c4.2xlarge</td>
<td>1130</td>
<td>1530</td>
<td>125</td>
<td>$0.562</td>
<td>2010.68</td>
<td>2722.42</td>
<td>222.42</td>
<td>99.335%</td>
</tr>
<tr>
<td>c4.4xlarge</td>
<td>1900</td>
<td>2750</td>
<td>220</td>
<td>$1.125</td>
<td>1688.89</td>
<td>2444.44</td>
<td>195.56</td>
<td>86.651%</td>
</tr>
<tr>
<td>c4.8xlarge</td>
<td>1700</td>
<td>2200</td>
<td>190</td>
<td>$2.250</td>
<td>755.56</td>
<td>977.78</td>
<td>84.44</td>
<td>36.900%</td>
</tr>
<tr>
<td>c3.large</td>
<td>220</td>
<td>270</td>
<td>23</td>
<td>$0.129</td>
<td>1705.43</td>
<td>2093.02</td>
<td>178.29</td>
<td>80.087%</td>
</tr>
<tr>
<td>c3.xlarge</td>
<td>440</td>
<td>540</td>
<td>48</td>
<td>$0.258</td>
<td>1705.43</td>
<td>2093.02</td>
<td>186.05</td>
<td>81.225%</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>900</td>
<td>1150</td>
<td>100</td>
<td>$0.516</td>
<td>1744.19</td>
<td>2228.68</td>
<td>193.80</td>
<td>84.668%</td>
</tr>
<tr>
<td>c3.4xlarge</td>
<td>1600</td>
<td>2300</td>
<td>190</td>
<td>$1.032</td>
<td>1550.39</td>
<td>2228.68</td>
<td>184.11</td>
<td>80.032%</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>1950</td>
<td>2700</td>
<td>225</td>
<td>$2.064</td>
<td>944.77</td>
<td>1308.14</td>
<td>109.01</td>
<td>47.690%</td>
</tr>
</tbody>
</table>

Table 8: Test Results of optimized Event-MPM Configuration

Figure 19: Concept of the Cache Filter

effective ones towards computational and economical efficiency. To get the best choice of instance types, the previous tests have been analyzed and it has been found that, with optimal configurations, the instance types c4.large, c4.xlarge, c4.2xlarge and c4.4xlarge show the highest potential in terms of economically related performance (See 4.1.4 Cost). The tests show the impact of the elaborately introduced caching in multiple places of the program (See Tab. 9). The
Table 9: Test Results with Improved Caching

<table>
<thead>
<tr>
<th>Instance</th>
<th>(U_S)</th>
<th>(U_T)</th>
<th>RSFL</th>
<th>USD/h</th>
<th>(O_{US})</th>
<th>(O_{UT})</th>
<th>(O_{RSFL})</th>
<th>(O_{TOTAL})</th>
</tr>
</thead>
<tbody>
<tr>
<td>c4.large</td>
<td>270</td>
<td>540</td>
<td>50</td>
<td>$0.141</td>
<td>1914.89</td>
<td>3829.79</td>
<td>354.61</td>
<td>88.970%</td>
</tr>
<tr>
<td>c4.xlarge</td>
<td>750</td>
<td>1100</td>
<td>85</td>
<td>$0.281</td>
<td>2669.04</td>
<td>3914.59</td>
<td>302.49</td>
<td>93.925%</td>
</tr>
<tr>
<td>c4.2xlarge</td>
<td>1550</td>
<td>2150</td>
<td>200</td>
<td>$0.562</td>
<td>2758.01</td>
<td>3825.62</td>
<td>355.87</td>
<td>99.242%</td>
</tr>
<tr>
<td>c4.4xlarge</td>
<td>2450</td>
<td>3050</td>
<td>280</td>
<td>$1.125</td>
<td>2177.78</td>
<td>2711.11</td>
<td>248.89</td>
<td>72.719%</td>
</tr>
<tr>
<td>Maximum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2758.01</td>
<td>3914.59</td>
<td>355.87</td>
<td></td>
</tr>
</tbody>
</table>

system is now able to handle up to 3000 users simultaneously, still using one monolithic entity.

While the c4.4xlarge can handle the most concurrent users as a single entity, the most efficient one is by far the c4.2xlarge instance. It outperforms all previous test configurations in all three values (\(O_{US}\), \(O_{UT}\) and \(O_{RSFL}\)) and only the c4.xlarge and c4.large instances in the same configuration show a slightly better \(O_{UT}\) value.

5.6 Distribution

The main reason for splitting the system into its components is enabling a horizontal scaling of the application server. Additionally, this should not only remove the upper performance limit which is in place when scaling vertically, but should also remove one single-point-of-failure in the system, i.e. if an optional component such as the cache server fails for whatever reason, the other servers will continue to work seamlessly.

Therefore the goal is to distribute the system as far as it is useful. The intention is not to split all the components where ever possible, but rather to provide an optimal configuration for this platform in terms of performance and scalability. An additional advantage of a separated and horizontally scalable system for this platform is the possibility to increase the computational power and capacity of the hardware without having to shut down the system awhile.

An ideal architecture for a web based application separates the following entities from each other onto separate servers (AmazonWebServices, 2015):

- Web Server
- Application Server
- Static Content
- Database
- Cache

AWS additionally suggests to use load balancers between the web and application layer, in order to distribute the load evenly and guarantee a maximal availability (AmazonWebServices, 2015). Yet, in this project, the separation of
web and application server is not envisaged. Therefore, the monolithic system will be split into multiple parts, but as for the platform’s configuration, the separation of web and application server is spared. Thus, the new architecture uses separate servers for cache, database and application server, the latter also hosting the web server. The horizontal scaling of the system will be discussed in the next section (See 5.7 Load Balancing). Since the static content was already outsourced at the beginning, the CDN is and stays a separate entity in this architecture (See Fig. 20).

![Figure 20: Decomposed Architecture](image)

Due to the use of multiple hardware entities, the cost for the various configurations has to be re-calculated. The load simulation will again involve the testing of the four most efficient instance types (c4.large, c4.xlarge, c4.2xlarge and c4.4xlarge) for the application server, as the c4.4xlarge has proven to not be very economical. The database server is set up as a single MySQL RDS instance and the cache as a 1-node ElastiCache Cluster, both as m3.medium instances in AWS. The performance of those two entities is set to be constant, so that the different application server configurations can be measured. Before performing the actual tests, it was made sure that neither the cache server nor the database server will be a performance bottleneck to the system by choosing hardware with an appropriate performance and capacity.

Because the database and the cache will use constant hardware configurations, their cost is constant as well, i.e. 0.103$ per hour for the cache (m3.medium) and 0.220$ per hour for the database server, which results in 0.323$ per hour in constant costs. Additionally, the costs for the test set-up of the application servers are the following:

- 0.141$ (c4.large) + 0.323$ = 0.464$
<table>
<thead>
<tr>
<th>Instance</th>
<th>$U_S$</th>
<th>$U_T$</th>
<th>RSFL</th>
<th>USD/h</th>
<th>$O_{US}$</th>
<th>$O_{UT}$</th>
<th>$O_{RSFL}$</th>
<th>$O_{TOTAL}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>c4.large</td>
<td>150</td>
<td>210</td>
<td>18</td>
<td>$0.464$</td>
<td>323.28</td>
<td>452.59</td>
<td>38.79</td>
<td>78.753%</td>
</tr>
<tr>
<td>c4.xlarge</td>
<td>220</td>
<td>290</td>
<td>28</td>
<td>$0.604$</td>
<td>364.24</td>
<td>480.13</td>
<td>46.36</td>
<td>88.732%</td>
</tr>
<tr>
<td>c4.2xlarge</td>
<td>370</td>
<td>450</td>
<td>44</td>
<td>$0.885$</td>
<td>418.08</td>
<td>508.47</td>
<td>49.72</td>
<td>96.966%</td>
</tr>
<tr>
<td>c4.4xlarge</td>
<td>540</td>
<td>810</td>
<td>64</td>
<td>$1.448$</td>
<td>372.93</td>
<td>559.39</td>
<td>44.20</td>
<td>92.700%</td>
</tr>
</tbody>
</table>

| Maximum    | 418.08 | 559.39 | 49.72 |

Table 10: Test Results with a Distributed Architecture

- $0.281 \times (c4.xlarge) + 0.323 = 0.604$
- $0.562 \times (c4.2xlarge) + 0.323 = 0.885$
- $1.125 \times (c4.4xlarge) + 0.323 = 1.448$

The distribution of the architecture is expected to improve the performance, since the computations of the database and cache are now outsourced to other hardware entities. Therefore the application server merely has to compute the client’s requests and send some other requests to the database and cache server, which should offload parts of the computations.

However, as the results show (See Tab. 10), the distribution had a highly negative effect on the performance. After investigation, it became clear the problem was the latency between the database server and the application server. Due to the fact that the database is highly inefficient and therefore the implementation of the API involves several dozens of queries per user request, even a low latency (about 7 ms) can affect the results significantly. The web server threads, which are sending the database queries, stay in a blocking-state until they receive an answer. Now, that the response time for each query has multiplied manyfold, already a few hundred concurrent clients can result in the very most threads being in a state of waiting for database responses.

But setting aside the significant loss in performance, the distributed architecture now allows us to horizontally scale the web and application servers.

### 5.7 Load Balancing

Since the architecture now allows a horizontal scaling, a load balancer guarantees the even distribution of load on the its instances (See Fig. 21).

In order to provide appropriate test results, this set-up has been tested with one, two, four, eight and sixteen parallel application servers, all respectively using the same instance type. Because the c4.2xlarge instance type has proven to be the most efficient one in terms of cost related to the performance. The costs of the different configurations are composed as follows:

- $1 \times 0.562 + 0.323 = 0.885$
- $2 \times 0.562 + 0.323 = 1.447$
Figure 21: Newly Implemented Architecture

<table>
<thead>
<tr>
<th>Count</th>
<th>U_S</th>
<th>U_T</th>
<th>RSFL</th>
<th>USD/h</th>
<th>O_US</th>
<th>O_UT</th>
<th>O_RSFL</th>
<th>O_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>250</td>
<td>300</td>
<td>28</td>
<td>0.885</td>
<td>282.49</td>
<td>338.98</td>
<td>31.64</td>
<td>81.543%</td>
</tr>
<tr>
<td>2</td>
<td>450</td>
<td>540</td>
<td>49</td>
<td>1.447</td>
<td>310.99</td>
<td>373.19</td>
<td>33.86</td>
<td>88.907%</td>
</tr>
<tr>
<td>4</td>
<td>900</td>
<td>1100</td>
<td>95</td>
<td>2.571</td>
<td>350.06</td>
<td>427.85</td>
<td>36.95</td>
<td>99.642%</td>
</tr>
<tr>
<td>8</td>
<td>1500</td>
<td>1900</td>
<td>180</td>
<td>4.819</td>
<td>311.27</td>
<td>394.27</td>
<td>37.35</td>
<td>93.690%</td>
</tr>
<tr>
<td>16</td>
<td>2800</td>
<td>3600</td>
<td>320</td>
<td>9.315</td>
<td>300.59</td>
<td>386.47</td>
<td>34.35</td>
<td>89.390%</td>
</tr>
</tbody>
</table>

Table 11: Test Results with Horizontally Scaled Application Servers

- $4 \times 0.562$ + 0.323$ = 2.571$
- $8 \times 0.562$ + 0.323$ = 4.819$
- $16 \times 0.562$ + 0.323$ = 9.315$

The costs again include the constant costs for the database and the cache server. Although the configuration with a single instance has already been tested before, the simulation will be repeated here, due to the additional load balancer. This will give us a baseline for the following test results (See Tab. 11):

Looking at the results of a single c4.2xlarge instance shows the impact of introducing a load balancer to the system. U_S, U_T as well as RSFL are all three significantly reduced compared to the same simulation without a load balancer. However, the load balancer finally enabled a horizontal scaling. The other test results show the possibility to scale the system to up to 3600 users, using 16 concurrent application servers. The highest efficiency can be reached with the use of 4 concurrently operating application servers. An upper limit of concurrently active users could not be found, though. This brings us to the conclusion
that the horizontal scalability of the application servers is now possible.

5.8 Database Architecture

The Database was in its initial state not only closely coupled to the other components while being located on the same hardware entity, but also consisted of only one MySQL server instance. But even now, after separating this component and putting it onto an own hardware entity, in case it comes to an overload, the scalability problem arises again for the database. It can be scaled up vertically, but this involves the already mentioned drawbacks and problems. Therefore other possible solutions have been investigated and found.

There are two basic approaches for horizontally scalable, (i.e. distributed) database concepts: replication and partitioning. For replication based distributed databases, the main design issue is the necessity for data consistency on all copies, i.e. all copies of the same data item must have the same value at the same time. Therefore synchronization protocols have to be employed which use for instance a master node, that propagates all changes to the slave nodes. (Özsu & Valduriez, 2011) Fragmented (i.e. partitioned) databases bring the problem that more complex read-queries have to be split and forwarded to the nodes with the respective data and the results have to be aggregated before delivering the result. This again is rather inefficient.

The three solutions which are being compared are i) a conventional high-end single database server with a single instance, ii) the rather new Galera technology, running on three separate medium-level servers using writeSet replication and the recently introduced Aurora technology by AWS.

5.8.1 Single-Entity Database

The current configuration is built on a single instance database server. As previously discussed, the drawbacks of a single-entity system including the limited scalability and existence of a single point of failure force us to look upon other solutions, which solve this problem. The current usage of the system, however, allows at least a tenfold vertical scaling of the current database server, considering the available hardware at AWS.

Considering the current architecture, the only possible solutions can build on MySQL compatible Databases, since the application is directly geared to it. The refactoring of all database queries is not only time consuming but at the current stage also economically impossible. Therefore the single-entity database will be installed as an Amazon RDS instance and vertically scaled throughout the testing in order to calculate the performance in comparison to the other two options.

5.8.2 Galera Cluster

The Galera Cluster technology is based on writeSet replication, i.e. a master-master replication rather than a conventional Master-Slave architecture. It allows clients to use any node of a cluster for both, read and write operations. This
removes the problem that the entire system is just as strong as the write-master
node and greatly increases the stability of the entire database structure. Using
Galera Clusters removes the slave-lag that is involved in conventional master-
slave replication systems and prevents the loss of transactions. The technology’s
nature provides both a read and a write scalability and greatly decreases client
latencies (Wit & Selij, 2014).

Unfortunately a Galera Cluster is not offered by any IaaS provider due to the
fact, that for one, it is a quite new technology and rather untested in production
environments and second, prove to be quite unstable. Yet, the huge benefits en-
courage a deeper look into the technology.

For the database-cluster, the servers each build on a MariaDB implementa-
tion of the Galera technology. Since the nature of a Galera-Cluster requires a
number of at least three nodes, the benchmarks will involve two settings, one
with three and one with five nodes.

5.8.3 Amazon Aurora

Amazon’s Aurora database engine has been announced in 2014 and officially
started in Summer 2015. Although closed-source, it promises the performance
and reliability of commercial databases at the price of open-source databases.
It is a relational database engine that is compatible with MySQL. According
to AWS, Aurora provides up to five times the performance of MySQL on the
same hardware. Taking this into account, the costs for running the platform’s
database could be lowered significantly, if the technology proves to be usable.

Due to its compatibility to MySQL, the migration of data is easy and possi-
ble without hurdles. For the benchmark, an equal set-up to the MySQL server
has been created for an objective and fair comparison.

5.8.4 Database Benchmarks

For the benchmarks of the different database set-ups, the software sysbench has
been used. The tests included transactions on a test-set of 1,000,000 rows and
have been executed from a high-end server, whose hardware was independent
from the database servers and far too powerful to reach its limit during the
benchmarks. Thus, sysbench has been executed remotely while running 128
threads in parallel.

Previous monitoring of the database has shown a weakness towards the mem-
ory consumption on the database server, while the CPU was generally not even
close to reach its capacity limit. Thus, the tests involved the use of memory-
optimized instance types. For the Aurora and MySQL servers, the tests included
an execution of several hundreds of thousands of transactions within 60 seconds
firstly on an r3.large instance and then on an r3.xlarge instance (See Tab. 12),
which both are based on an Intel Xeon E5-2670 v2. The Galera cluster has been
tested as a set-up of respectively 3 and 5 nodes on the r3.large instance type.

During the tests, the immaturity of the Galera technology became clear. When
<table>
<thead>
<tr>
<th>Instance Type</th>
<th>#vCPUs (Clock)</th>
<th>Mem</th>
<th>Storage (GB)</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>r3.large</td>
<td>2 (2.5 GHz)</td>
<td>15.25 GB</td>
<td>1 x 32 SSD</td>
<td>Moderate</td>
</tr>
<tr>
<td>r3.xlarge</td>
<td>4 (2.5 GHz)</td>
<td>30.5 GB</td>
<td>1 x 80 SSD</td>
<td>Moderate</td>
</tr>
</tbody>
</table>

Table 12: Tested AWS Instance Types (Amazon, 2015)

<table>
<thead>
<tr>
<th>DB Type</th>
<th>Instance</th>
<th>Price/h</th>
<th>read/write per sec</th>
<th>transactions per sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aurora</td>
<td>r3.large</td>
<td>$0.320</td>
<td>579</td>
<td>11010</td>
</tr>
<tr>
<td>MySQL</td>
<td>r3.large</td>
<td>$0.265</td>
<td>647</td>
<td>12301</td>
</tr>
<tr>
<td>Aurora</td>
<td>r3.xlarge</td>
<td>$0.640</td>
<td>1064</td>
<td>20212</td>
</tr>
<tr>
<td>MySQL</td>
<td>r3.xlarge</td>
<td>$0.530</td>
<td>1248</td>
<td>23714</td>
</tr>
<tr>
<td>Galera</td>
<td>3 x r3.large</td>
<td>$0.585</td>
<td>1856</td>
<td>25983</td>
</tr>
<tr>
<td>Galera</td>
<td>5 x r3.large</td>
<td>$0.975</td>
<td>3001</td>
<td>39224</td>
</tr>
</tbody>
</table>

Table 13: Database Benchmark Results

running multiple threads concurrently on the Galera cluster, several transaction conflicts occurred, although the Galera technology promises to correct those conflicts. However, the conflicts led to frequent errors in the benchmark, as several transactions could not be verified on one server after they had been written to another one. The write-tests did not deliver any useful results for the Galera set-up and its benchmarks have been limited to read-only tests.

As the benchmark results show (See Tab. 13), the Aurora database does not perform as well as promised by AWS, but quite the opposite: it has been outperformed by the conventional MySQL database in every test setting. As already mentioned, due to the immaturity of the technology, the Galera cluster could not be properly tested in terms of write-performance. Yet, due to the fact that a Galera-based solution is not offered as a managed solution, it would involve a high administrative effort for a company, which wants to use it in a production environment. The system has to be kept up to date and secure, which not only involves the database software but also the entire server and its operating system on which the cluster nodes are running.

Therefore, as long as the technology is not completely sophisticated and matured, one should better count on conventional solutions. As for the Aurora technology, it might be much more efficient when deployed with synchronous replication. However in the scope of this project, this is neither necessary nor useful due to the high load on the application server and the comparatively low load on the database. So according to the benchmark results (See Fig. 22), the set-up with a conventional MySQL database instance is the most effective solution, although it involves a necessity for vertical scaling, when the user numbers outgrow the respective current performance. Yet, a rare modification of the database with an offline-time of at most a few seconds can be approved for the other advantages this decision brings with it.
6 Comparison and Evaluation

Aggregating the results of the previous steps, we can conclude both, the best possible solution and direct impact on the platform that was subject to the tests, as well as general findings that can be applied onto other platforms or projects.

6.1 General Findings

The test results lead us to certain findings, which can be used in order to improve the performance and costs of any client-server software. This is not limited to web applications and platforms but also includes native software, that communicates with one or more server-instances. The most important findings can be summarized as follows:

1. When using a server that does not have to handle many requests per second for each client, the configuration of server process and thread handling becomes a critical part of making a system scalable and increasing its performance. An inefficient thread handling can result in the hardware not being used to its full extent, which is equal to a waste of resources. This is a result of many idle threads and processes, that block other threads and processes, which otherwise could be worked off. More specifically, for web applications, when the connections to the clients are kept open by using the keep-alive option of HTTP 1.1, the non-optimal configuration of the web server software can result in severe bottlenecks. Those can easily be avoided by increasing the number of threads and processes in order to leverage the capabilities of modern high-end server hardware.

2. Server-side caching can have an enormous impact on the performance of an application. That applies especially to services with data that is modified infrequently. However, a tradeoff has to be made when using caching, due to the fact that cached data might not be up-to-date. Therefore, the cached data piece either has to be flushed from the cache on modification,
which results in a lower performance on server-side. The other option would be to risk and accept a certain level of inactuality of the presented data. The tradeoff would involve the decision of which data piece should be cached for which duration, which should not be cached at all and when the cache should be flushed.

3. Separation of different services of an application has a huge potential when it comes to availability and scalability. Data redundancy can easily be implemented in a distributed system where single entities can be replicated and thus, the load can be distributed. However, the separation has a significant impact on the performance of a system, especially when additional entities such as load balancers are introduced. Compared to closely coupled components, the performance becomes even worse when the entities maintain a high communication between themselves. This is a result of significantly increased response latencies. The performance can here even shrink to 10% or less of its value compared to a monolithic architecture.

4. New database technologies can possibly increase the performance and lower the cost of persistent data storage. However, not all new technologies keep what they promise. The here tested technologies, Galera Clusters and Amazon Aurora both did not bring to proof their claim of an enhanced performance on the same hardware compared to a conventional MySQL database. Especially the Galera technology is not yet ready to be used in a production environment for its lack of transaction safety as demonstrated and proven in the 5.8.4 Database Benchmarks. Although Amazon’s Aurora database engine is provided at 82% of the price for MySQL instances on the same hardware, Amazon’s claim to provide a 5-times higher performance could not be verified, but quite the opposite has been shown in our test results. Namely MySQL indeed performed better when taking into account both, the costs as well as the performance, however Aurora is generally only offered on rather high-cost hardware, which is not a useful option for systems with a rather low database usage.

6.2 Impact on the Tested Platform

The architectural changes and tests on the e-learning platform led to a both, economical as well as performance related impact on several levels:

1. The initial system as is was before any kind of modification has been performed was only able to handle up to 270 concurrently active users. Due to that limitation, which was in place independently from the underlying hardware, the economically most efficient choice was to choose the instance type c3.large for the then monolithic system.

2. By investigating and improving the configuration of the Prefork MPM module, the limit could be removed. After optimizing the module, the system was able to serve up to 1670 concurrent clients on an instance type of m4.4xlarge. This could be achieved by vertically scaling the underlying hardware.
3. A further research about Apache’s multi-processing modules led to the implementation, optimization and test of the two other modules, Worker, and Event. After modification of their respective configuration, both showed a better performance than the Prefork module, and both did not refuse any connections as Prefork did. the Event-module performed slightly better than the Worker module, by serving up to 2750 concurrent users (Worker: 2700) with a faintly higher efficiency of the most efficient instance (c4.2xlarge) in all three metrics (OUS, OUT and ORSFL).

4. Using the best of the three multi-processing-units, the Event based module, the performance has again been tested after optimizing the code for caching of rarely changed data pieces. This increased the performance again to a level, such that up to 3050 concurrent users could use the platform on a c4.4xlarge instance. Here the c4.2xlarge instance has again proven to be the economically most efficient one.

5. Building on the previous results, the database, cache and application server have been separated to different servers in order to reduce the load on the application server by outsourcing the database and cache computations. This, however, proved to be a misconception due to the increased latency between application server and database. For the system currently bases on a highly inefficient database structure, each request produces many database queries, which slow the system down significantly with the higher latency between the two servers. On the other hand, the distribution enabled a horizontal scaling of the application server in the next step. The decrease in performance, though, led to a decreased maximum number of concurrent users in this state: with a c4.4xlarge instance, maximal 810 concurrent users could be served, while OUS, OUT and ORSFL dropped to about 15% of their previous value.

6. Introducing a load balancer as an additional layer again reduced the performance and thus the capability of handling many concurrent users. OUS, OUT and ORSFL dropped to 350, 428 and 37, respectively. However, after horizontally scaling up the application server, it became obvious, that there was no visible performance limit left. The highest efficiency was reached with 4 concurrent servers, which allowed to serve 1100 concurrent users. When using 16 concurrent application servers, the limit rose to 3600 CAU, while only losing a few points in economical efficiency compared to the configuration with 4 servers.

7. The attempt to introduce a distributed database system resulted in the insight that a system with such a low database use, compared to the computations on the application server, is well suited with a single MySQL database instance, which prove to be by far the best possible solution, keeping in mind that a managed database highly reduces its administration costs and only a limited number of possibilities are given on the used IaaS.
7 Conclusion

The undertaken measures led to the desired result of an improvement of the performance. The system is now able to handle virtually unlimited numbers of users. The company does not expect to exceed 1000 concurrent users within the next few months, which is given by the solutions provided in this thesis. Hereby the company can even choose between a highly optimized monolithic architecture or a less efficient, though much more scalable distributed architecture, which supports at least 3600 concurrent users, as modelled in this project. The user-limit has been increased by at least 1300%. Considering the optimized monolithic architecture, the economical efficiency has been improved by about 100%, although the distributed architecture even shows a reduced efficiency of about 23% compared to the initial system with the option of horizontal scalability.

An additional gain is the knowledge about the web server configuration and its workings with special regard to the running system as well as an insight into possible database solutions. The database did not show much potential to be optimized, however, the separation in the proposed distributed architecture enables to use theoretically any database solution available, although the tested solutions did not prove to be better than the already used one.

All in all, the thesis can be considered successful in its attempt to increase the system’s overall performance by optimizing the web server configuration and implementing caching. Also the findings of this project enabled both, a vertical as well as a horizontal scaling and puts both solutions into comparison. Using technologies such as load balancers and a highly available IaaS provider put the system in an innovative architecture and environment, which ensures stability and the option to quickly react on system failures. However, the attempt to introduce novel database technologies unfortunately did not lead to an improvement of the system but rather motivated the use of the current database configuration.
8 Future Work

As possible future work projects, the database could be optimized in terms of its structure. Due to the fact that the database originates in a Drupal installation, which was the first implementation of the platform, the database consists of several hundred tables. This is a result of Drupal’s inefficient use of relational databases and their potential when combined with implementations of object relations.

Another possible future work could be to extend the findings of this work in terms of the used metrics, so that those can be used to ultimately measure and compare different web platforms with each other, which have similar functionalities. The One-Dollar-Hour metrics can be used to describe the cost efficiency of such web services and might be usable to determine the economical and scalability related value of a system.

Also an option would be the investigation about different database models. This could include the testing of NoSQL databases for parts of the persistent data storage, especially for those, which are subject to heavy and frequent data modifications and insertions.

As a last possibility, the web server software could be separated from the application server. Then, other technologies could be investigated and applied to the system, for instance the Apache2 web server could be changed for NGinX and tested for its performance impact.
9 Appendices

As a digital appendix, the following documents are available per email request to yassinm@kth.se:

- The implementation of the cache filter
- The elaborate results of the stress tests in form of diagrams and spreadsheets
- The aggregated results and their calculated metrics as spreadsheets
- The configuration of the test plans
- The Latex source files
10 Bibliography


