Bachelor Degree Project

Zero-Downtime Deployment in a High Availability Architecture

- Controlled experiment of deployment automation in a high availability architecture.

Author: Axel Nilsson
Supervisor: Johan Hagelbäck
Semester. HT 2018
Subject: Computer Science

Abstract

Computer applications are no longer local installations on our computers. Many modern web applications and services rely on an internet connection to a centralized server to access the full functionality of the application. High availability architectures can be used to provide redundancy in case of failure to ensure customers always have access to the server. Due to the complexity of such systems and the need for stability, deployments are often avoided and new features and bug fixes cannot be delivered to the end user quickly. In this project, an automation system is proposed to allow for deployments to a high availability architecture while ensuring high availability. The purposed automation system is then tested in a controlled experiment to see if it can deliver what it promises. During low amounts of traffic, the deployment system showed it could make a deployment with a statistically insignificant change in error rate when compared to normal operations. Similar results were found during medium to high levels of traffic for successful deployments, but if the system had to recover from a failed deployment there was an increase in errors. However, the response time during the experiment showed that the system had a significant effect on the response time of the web application resulting in the availability being compromised in certain situations.

Contents

1 Introduction

Many computer applications have gone from being local installations on our computers to web applications hosted on remote servers. While there are many benefits to this strategy, one downside is the availability due to the fact that we often need to have a connection to the application server to use the full functionality of the application or service.

There are several ways to achieve high availability, but often the aim is to create stability and redundancy in the system [1]. A simple way to keep a system stable is to avoid making changes to it. Additionally, to the high availability requirement, there is often a need to deliver new features and bug fixes quickly, but making changes to a system can be a risk and might introduce instability to the system.

This project looks at the use of automation in the deployment process to allow for zero-downtime deployment in a high availability architecture. Could such a system allow for deployment approaches like continuous delivery or even continuous deployment while maintaining the promise of high availability?

1.1 Background

This project involves several different concepts and methods where we look closer at what happens when these are combined. Starting with the architecture followed by the deployment and its surrounding concepts.

When discussing *availability* in computing, it is generally looked at as the amount of time where a service is working as intended and can respond to its consumers over a period of time [2]. The amount of time where the service is available is often referred to as *uptime*, and the reverse where the service is not available is known as *downtime*. Therefore, *high availability* is where a system has an uptime which is significantly higher than the amount of downtime and where the service can promise a high measure of availability. In practice, high availability can be achieved by removing any single points of failure and ensure redundancy and overhead within the system [3]. When a failure does occur, planned or unplanned, the system has to react and redirect the traffic intended for the failed part of the system to another part of the system which can perform the same task [2].

Zero-downtime deployment is when a new version of the application can be introduced into the production environment without making the user

aware of any downtime [4]. This allows for deployments of new features and bug fixes to take place more often without the users noticing. Common techniques for implementing zero-downtime deployment are *Blue-Green Deployment* [5] and *Canary Releasing* [6]. Zero-downtime deployment techniques are commonly used when implementing rapid deployment approaches like *continuous deployment*.

Continuous deployment builds upon *continuous delivery* which in turn builds upon *continuous integration*. *Continuous integration* is an approach to software development where changes are continuously integrated into the code base and then tested using automatic tests creating a quick feedback loop [7]. Continuous integration builds upon the idea that a painful task should be performed often to minimize the complications each time the task is performed [8]. *Continuous delivery* builds upon continuous integration to ensure the software is always releasable [9]. This is done through packaging of the software and more testing, sometimes including deployment to a staging environment or other test environments, allowing for a one push deployment to production [9][10]. *Continuous deployment* takes that final deployment step and fully automates it. Every time a change passes all steps the continuous delivery pipeline it is automatically deployed directly to production [9][10].

Other concepts used are *configuration management* which is a way of systematically managing changes to computer systems [11], *software provisioning* which means preparing servers with the software needed to run our application [12], and *deployment* which means putting the application on a server or updating the existing application to a newer version, replacing the old one. Many of these behaviors are combined to create automation tools like Ansible, Chef, or Puppet which can perform these tasks [12].

1.2 Related work

No research studying the same problem as this project was found during the literature review. There where a decent amount of existing research around how high availability systems can be managed. Many of these articles and papers focused on one of two subjects. The first subject was cloud computing and how to orchestrate virtual machines in the cloud and what requirements are expected of both the application and a potential platform to host these applications in a way to ensure high availability [13, 14]. The other articles focused on high availability through the use of open source virtualization

platforms and how to create frameworks for high availability using virtualization together with tools like Heartbeat or Pacemaker [15, 16, 17]. Both methods generally use some type of availability cluster to create redundancy in the system and then use tools to route traffic between them and monitor the services.

Many of the articles [13, 14, 16] mention the *five-nines* or 99.999% availability as the goal for availability and one even defines it as the requirement for what should be considered high availability. This measurement is done by measuring the amount of time the service is available during a set period of time, generally over a whole year.

Zero downtime deployment methods like *Blue-Green Deployment* [5] and *Canary Releasing* [6] and how they work and are structured are covered on popular tech blogs like Martin Fowler's website. However, no research comparing these techniques or testing their claims could be found and no research studying the adaptation of these techniques for a high availability architecture was found.

1.3 Problem formulation

Carl Caum said in his blog post about the difference between continuous delivery and continuous deployment "Continuous deployment should be the goal of most companies that are not constrained by regulatory or other requirements." [10]. In a high availability architecture, the main requirement is to ensure high availability of the service. The simplest way to ensure high availability is to avoid changes to the system since a static system is less likely to suddenly become faulty. However, this method does not allow for quick bug fixes to be applied after a deployment and might even go against the business needs if their needs are to rapidly add new features.

If changes were allowed onto the system more often rather than being avoided, the system would gain the benefits of rapid deployment. Constantly introducing change could also put the system at risk of failing its high availability requirement. But though automation of the deployment process, traffic routing, backups, and rollback action together with an adapted zero-downtime deployment technique could allow for fast and consistent configuration and deployment of the application while maintaining high availability.

The research question this project aims to answer is: *Can automation of the software deployment process allow for deployments without compromising the requirement of high availability in a high availability architecture?* If this is true, there should be no significant change in availability during a deployment compared to normal operations. This should remain true even if there is a problem with the new version of the application.

1.4 Motivation

Both high availability and having the ability to deploy often have their own distinct benefits, and if combined could have a monetary benefit to a company.

A service being unavailable could bring with it a set of problems. For example, if an e-commerce website is unavailable, none of their customers can buy anything from the company's website which would likely lead to a monetary loss [18]. High availability could also provide a commercial advantage since it affects the user experience. A service with too much downtime could create an incentive for users to start looking at competing solutions.

The main advantage of being able to deploy to production often through an automated system is adaptability. This means that new features created by the developers can be pushed to production much faster and start creating value for the business. Bugs or even security vulnerabilities can also be resolved much quicker. Just like continuous integration aims to fix the pains of integrating software by doing it more often [7], deploying more often forces developers and maintainers to create a system which can handle change often, instead of being static and hoping nothing goes wrong.

Combining both high availability and the adaptability of more deployments could allow for a business to reap the benefits of both, possibly creating a monetary gain.

1.5 Objectives

Table 1.1 displays the different steps needed to perform this experiment. Objectives O1 to O5 is about creating the automation system and implement the different subsystems and methods. Objectives O6 to O8 are about deploying the system to a controlled environment and then design, implement, and perform the experiment on this system.

O1	Create a sample web service application
O2	Create underlying configuration management to support web application
O3	Implement internal traffic routing
O4	Implement post deployment tests and monitoring
O5	Implement failover/recovery system
O6	Deploy system to environment
O7	Design and implement experiment
O8	Perform the experiment

Table 1.1: Table of all objectives and their identifier

The goal of this project is to implement and test a solution to the problem. The results should then show if the system solves the problem or not. Since commonly used zero-downtime deployment techniques already exist and can possibly be adapted to the specific architecture used in the implementation. It is therefore expected that the results will show the deployment system working and allowing for change to be deployed to a deployment environment without affecting the availability of the service. There is a risk of the system having flaws making it hard to come to a good conclusion to the research question. Another risk is if the system proves difficult to test making results difficult to analyze or even making them useless.

1.6 Scope/Limitation

For deployments to happen as often as they do when implementing continuous deployment or continuous delivery it is important that the application has passed through a rigorous testing pipeline before being deployed to a production environment. Deploying an untested or badly tested application to production is risky and will most likely not produce very good results. But in this project, the focus is on the deployment process. Because of this no testing prior to the deployment phase is included.

The system will not be universal. It focuses on a single operating system with a specific software platform and database management system using specific tools. Not all systems and platforms work well in a distributed system and therefore might yield different results. The architecture itself may vary quite a lot since it can be created in various ways which can also affect

the results.

The focus of this project is the deployment of the software running on the servers. Orchestration and provisioning of servers are outside of the scope of this project.

1.7 Target group

The target group for this project is developers, system administrators, and the organizations they work for. Traditionally developers work to produce changes to a system in the form of bug fixes or new features, while the system administrators want to limit change in favor of stability. We, therefore, have two factions with the same business goals, but different ideas on how these goals are met.

DevOps and the agile methodology aims to mitigate these issues and point developers and administrators in the same direction [19]. This is very prominent when working with a high availability system where changes are avoided to mitigate the risk of downtime. In this project, the focus is to create a system to allow for small incremental changes to be performed often in a safe way. This gives developers and system administrators not only a common goal to deliver business value safely to the consumer but also makes them work together in the same direction instead of against each other.

1.8 Outline

Following this chapter is the **Method** chapter where the choice of method is presented and how it will be used to answer the research question. The validity and reliability of the results produced by the chosen method and ethical considerations are discussed.

Then the **Implementation** is presented. It starts by looking at what infrastructure and environment will be used. Then implementations of the automation system and the sample application is presented together with a tour of the deployment process. It ends with a description of the tools used to conduct the experiment.

Next, the **Results** of the experiment are presented. This is followed by an **Analysis** where the results go through statistical testing and an analysis is performed to make sense of the gathered data.

The **Discussion** chapter then talks about what discoveries were made during this project and what the answer to the research question is and how it related to existing research.

It ends with a **Conclusion** with a retrospective on the project and the results together what could have been done differently and how future research could build and expand on the results of this project.

2 Method

This project used a controlled experiment to test the hypothesis that automation can allow for changes to be deployed to a high availability cluster without diminishing the availability of the system.

The experiment consisted of several tests using a stress testing tool, the high availability system deployed on virtual machines, and the automatic deployment system. The web service was deployed in the high availability architecture. When the application was fully deployed and had reached a working state with all machines, the point is set as the starting point for all tests. Before each test, the state of all machines had to be returned to the same starting point.

The dependent variable was the measurements gathered by the stress testing tools from each test. This will give us values from the clients' point of view on the error rate during each scenario and what changes, if any, there are in the response time. Each test used two independent variables: *deployment scenario* and *traffic level*.

There are three *deployment scenarios* used in these tests: *no deployment, successful deployment,* and *failing deployment*. The first scenario is the control and is referred to as the *no deployment* scenario. During this scenario, the tests are performed without using the automatic deployment system. This scenario was to mimic normal operations where no changes are being applied to an online system creating control results to which results from the other scenarios can be compared. The other two scenarios both use the automatic deployment system during the tests. The second scenario known as the *successful deployment* scenario is where the system deploys a working version of the application to the servers. The third scenario is known as the *failing deployment* scenario. Here the deployment system deliberately deploys a faulty version of the application to make the smoke tests fail to in turn force the automation system to perform a rollback.

Since availability can be affected by the amount of traffic due to exhausted resources [3] we need to control the amount of traffic used in each test. Because of this three traffic levels: *low, medium,* and *high* where defined and used in the experiment. What should be considered low or high is dependant on what resources are available. Therefore, these values had to be defined based on the implementation. The first value to be defined is the *medium level* which is roughly on the point where the system during a *no*

deployment scenario starts experiencing errors. The *low level* should then be considerably lower levels of traffic compared to medium, and *high level*, in turn, should be considerably higher than medium. The exact values used in the experiment is covered in the implementation chapter.

All combinations of the independent variables were used to create tests, resulting in 9 tests. Each test is performed 10 times to create a decent set of results on which we can good average values and also perform statistical testing.

2.1 Reliability and Validity

The results produced in this experiment should be reproducible using exactly the same automation system, application, infrastructure, and system resources as is described in the Implementation chapter. However, this is only one solution out of infinite possible solutions using different tools, methods, pattern, and architectures to solve the same problem. The solution in this project tries to use common practices and adapt them to the situation to produce a good answer to the research question. But any change in the implementation would most likely produce changes in the results.

2.2 Ethical Considerations

No personal or sensitive data is used in this experiment, and all traffic is synthetically generated and contains only information relevant to the experiment. There should not be any ethical issues which might arise by conducting this experiment.

3 Implementation

This chapter presents the automation system, the application and the environment and infrastructure where it all is deployed. The testing tools and how they are implemented and used in the experiment is also presented.

3.1 Environment & Infrastructure

To perform the experiment, the automation system developed in this project had to be used in an environment where it could deploy the application to a suitable architecture. Virtual machines were created using Vagrant and VirtualBox and then configured and provisioned with the configuration management tool Ansible. The final step for making the servers ready for the tests is to do a first deployment of the application and make sure the application is working as intended.

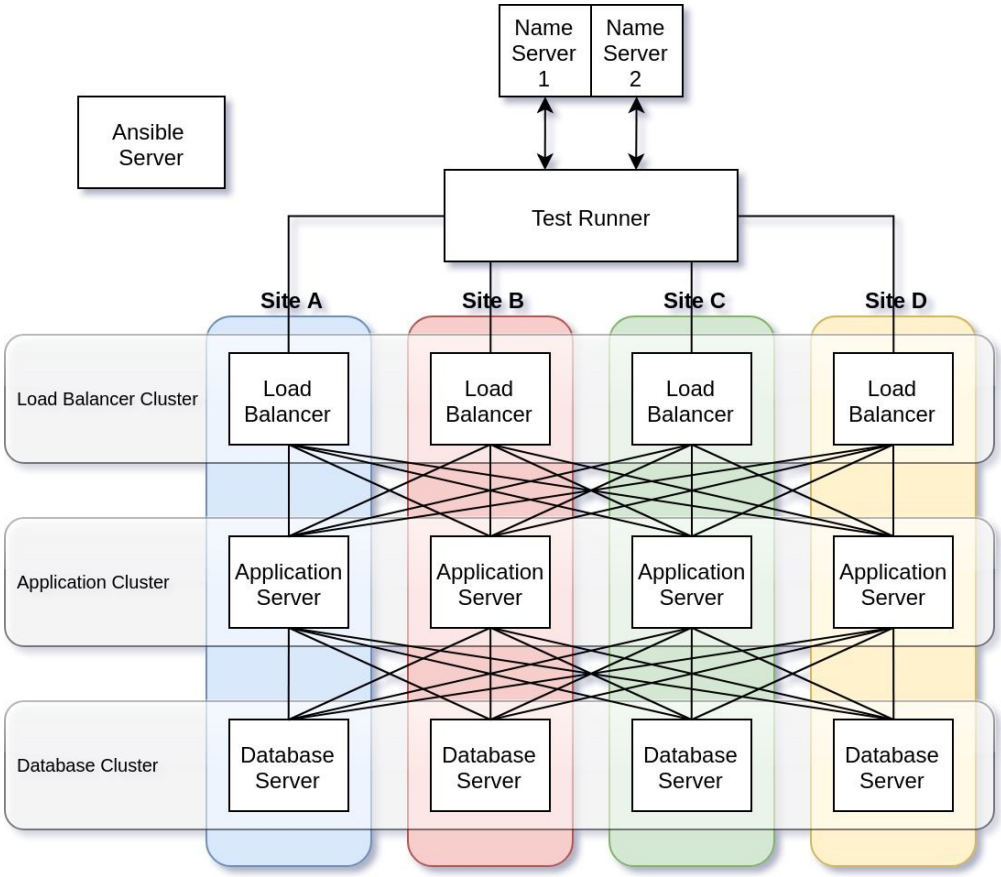

Figure 3.1: Diagram over the infrastructure.

Figure 3.1 shows the infrastructure used in the experiment. At the top in the middle is the *Test Runner* machine. This machine is responsible for running the experiment and acting like the consumer of the web service by generating the traffic and logging the results using a stress testing tool. There are also two name servers running bind9 in a master/slave configuration to be used by the Test Runner to get the IP address of one of the load balancer servers. Last is the Ansible Server which is responsible for executing the deployments during testing.

The remaining machines are divided into 4 sites named *Site A, Site B, Site C,* and *Site D* where each site run three servers: one *load balancer*, one *application server*, and one *database server*. Every server is also part of a cluster together with all other machines running the same software, creating a high availability cluster where if one node in the cluster were to fail, another node can perform the same tasks. There are three clusters in this system, a *load balancer cluster*, an *application cluster*, and a *database cluster*. *Figure 3.1* also show all ways traffic can move between clusters during normal operations.

All virtual machines are hosted on a single host to minimize disturbances from other machines on the network. The host is running Arch Linux with a minimal amount of software running which could affect the performance of the virtual machines. The virtual machines are all running Ubuntu 16.04 Server and their assigned resources are detailed in *Table 3.1*.

Type	vCPU	Memory	Essential Software
Load Balancer	1	256 MB	HAProxy
Application Server	1	256 MB	NodeJS, Test Application
Database Server	1	1024 MB	Java, Apache Cassandra
Name Server	1	256 MB	bind9
Ansible Server	1	512 MB	Ansible
Test Runner	2	1024 MB	Java, Taurus, JMeter

Table 3.1: Summary of the resources and software associated with each type of server.

3.2 High Availability Load Balancing

Starting from the top, how does the consumer know which load balancer to contact, and what happens if the one they usually contact goes down? These problems are solved with round-robin DNS which is a technique for load balancing using the domain name system [20]. Instead of only returning the same IP address each time the records are requested, the name server instead has a list of addresses and returns them one after another for each request. DNS creates a central point of contact to the system without creating a single point of failure since DNS is by design distributed [21].

Round-robin DNS is not a perfect system since it, for example, will continue to send the address of a host which is not available and since caching is an important part of DNS it cannot react to quick change very well [21]. This is why the DNS always resolves with an address to one of the load balancers. The load balancers are running a software called HAProxy [23] which unlike round-robin DNS can respond quickly to change by letting an administrator add and remove servers without having to restart any servers. It also supports multiple load balancing modes, sticky sessions and more.

3.3 Application

The *application cluster* runs a web application written in NodeJS with the Express framework. The application itself is a very simple CRUD application in the form of an HTTP API sending and receiving JSON data. This is done to make it easy to analyze the response from the service and know if the database is working as intended or not.

The database used in this project is Apache Cassandra [24]. This is a database management system created for working in a decentralized environment as the one in this project by being able to replicate data between nodes and failed nodes can be replaced without producing downtime [24].

3.4 Automation System

The automation system was created using Ansible, which is an automation tool from Red Hat. It uses a push method to distribute changes and tasks from a central machine using SSH and Python [25]. Using Ansible, changes can be pushed out whenever is desired by running a playbook consisting of a set of instructions and declarations to be executed on the servers. Ansible keeps track of the machines using an inventory file where they can be grouped and

be assigned variables to make it easy to manage multiple types of servers.

3.5 Deployment

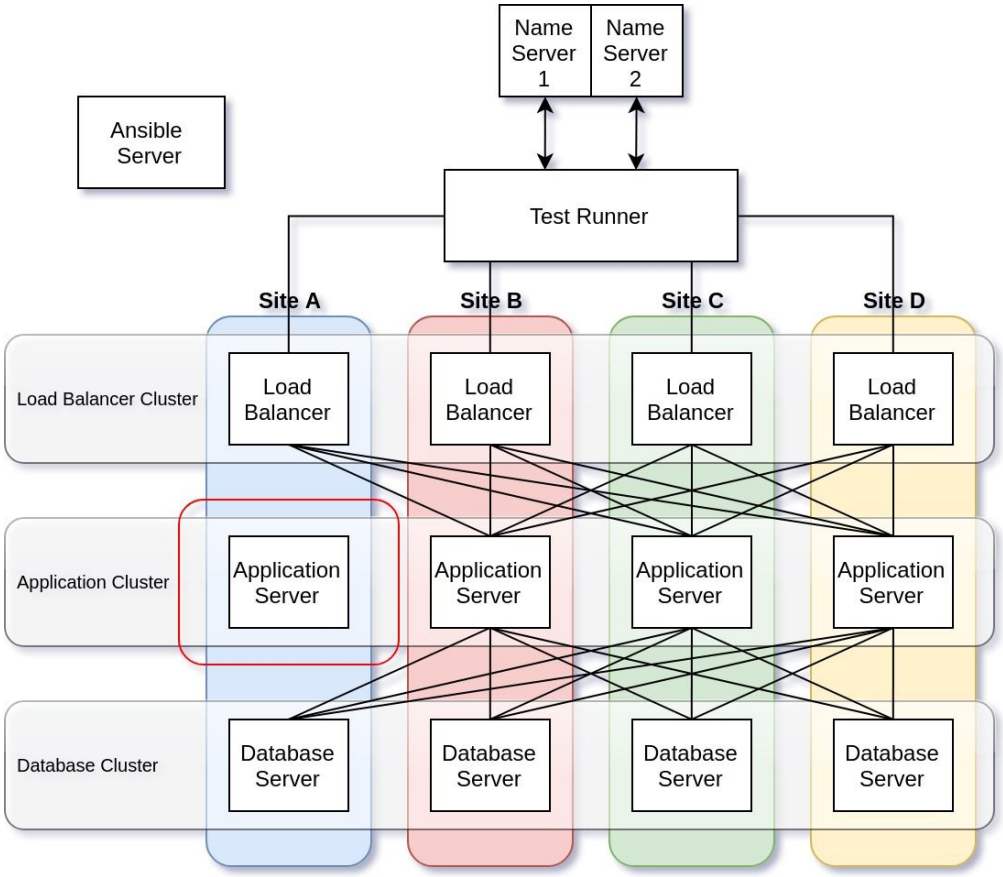

Figure 3.2: Chart over the infrastructure during deployment showing how traffic is routed to other nodes.

As shown in *Figure 3.2* compared to *Figure 3.1* when performing a deployment on Site A, all connections to the application server in Site A have been removed. Since it is known which server will become unavailable, the load balancers can be informed in advance to avoid routing traffic to an unavailable application server. The logical thing to do then is to continue doing the same thing for each site. However, this creates an issue with versioning since some servers will be on a newer version while others still remain on the old version making compatibility with the database complicated if there are any schema changes in the new version of the application. If an application server of version 1.0 tries to put data in a version 2.0 database the schemas might not match and will then produce an

error. This could be solved by writing the code and changing the schema with backward compatibility in mind [4]. One problem is how to recover from a failure if some machines are on the new version and have received data from a consumer. This could be solved by having scripts convert from the new version to the old version, but there is a possibility that it will lead to lost information since columns might be missing in the old schema. Also having to maintain a reverse-migration script creates a risk of human errors destroying or corrupting data.

The solution to the data consistency problem takes inspiration from a known zero-downtime deployment method called *blue-green deployment* where the production environment is divided into two groups: one blue, and one green [5]. At all times at least one of the environments are available to the consumer while the other one might not be. When a deployment is performed it is done in the offline environment while the online environment continues serving the customers. After the deployment is finished and the tests show a working system the traffic is redirected to the updated environment. If a problem is discovered with the newly deployed software traffic can instead be redirected back to the old version of the system. Leaving half the system on the old version of the software allows for quick rollback to the old version if a deployment fails. One downside with this method is that half the system is unavailable during large parts of the deployment which can lead to a lack of computational resources.

For the database, it would be logical to divide the cluster in a similar way to how it was done with the application server, but this proved to be very difficult in Apache Cassandra. Instead, the division is created by creating a new table and moving all data from the old table to the new table but leaving the old table for the old version of the software to use. If a problem is discovered during deployment it is most likely detected during the deployment to the first two sites and we can simply remove the new table and move back to the old without having to recover anything. If the deployment does go through the first two sites then the small amount of data added to the old database since the new database was created is transferred over to the new table.

Figure 3.3 illustrates how this is implemented within this project's infrastructure and shows why a minimum of four sites is required. For example, if a new version is deployed to the system it is first done on site A and site B with traffic being routed to site C or site D. The reason why four

sites are needed is to allow for redundancy even during a deployment when only two sites are available to consumers. If the deployment is working as intended then the live traffic is sent to site A and B while site C and D are being updated and later being added together with the other two sites.

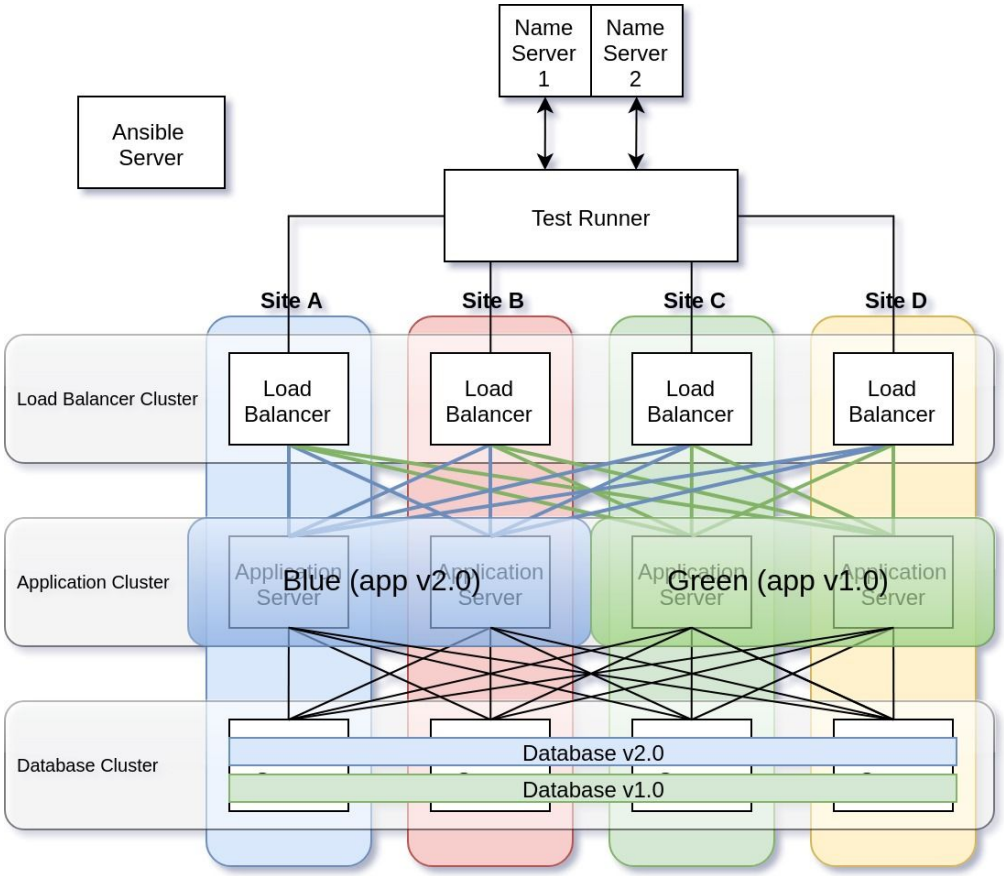

Figure 3.3: Chart over the infrastructure showing the blue greed division during deployment.

This method relies on the idea that fatal issues are most likely going to surface during deployment to site A and B. If there happens to be an issue when deploying to site C or D, the automation system will not add the application server to the system, and someone will have to manually fix it. This is fine since we still have at least two systems working.

Figure 3.4 described all steps performed by the deployment system during the deployment process. To make testing easier there is no webhook used and the confirmation for a full deployment will always confirm to continue with a full deployment during the tests.

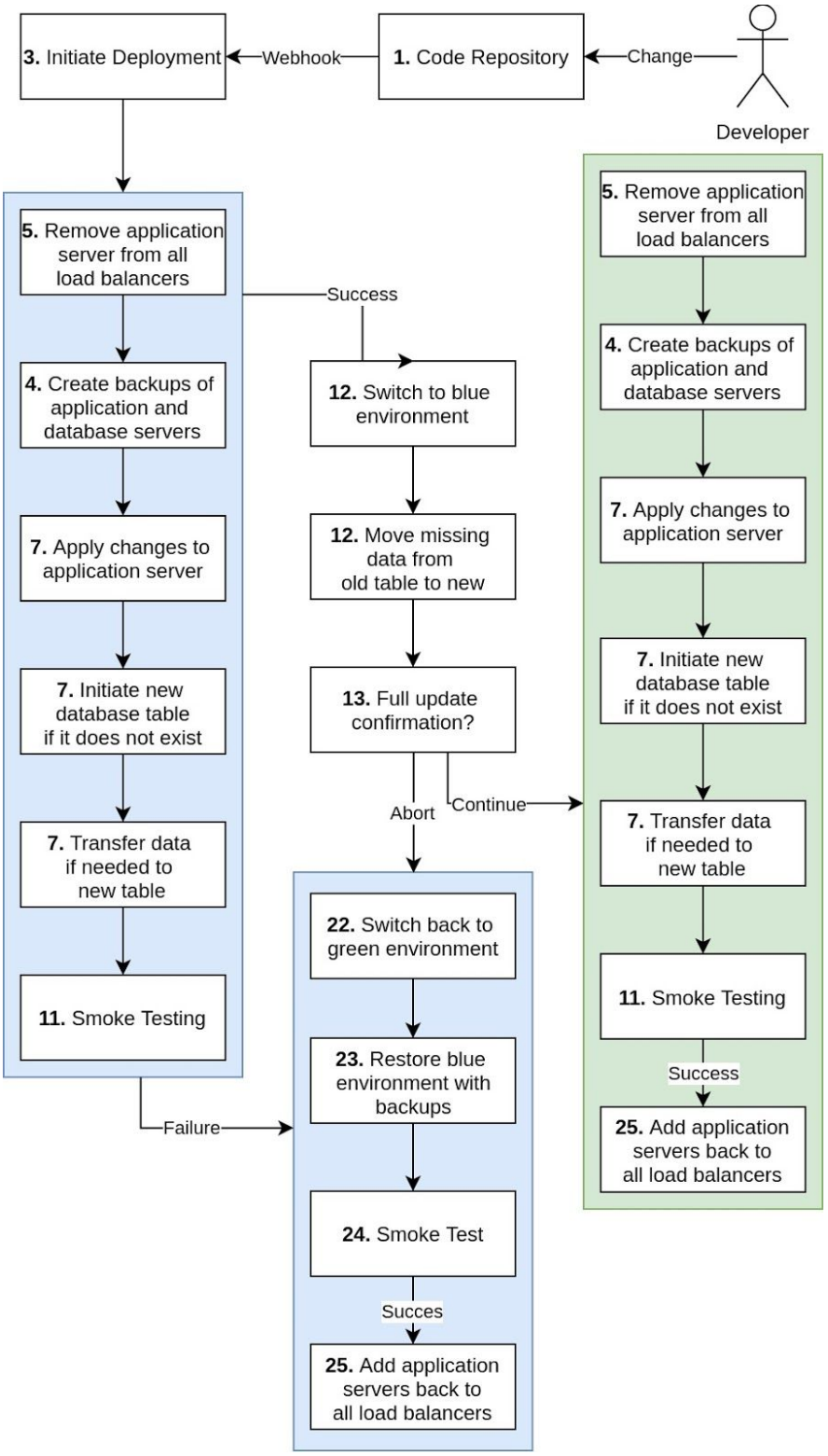

Figure 3.4: Chart describing the different steps of the deployment process

3.6 Experiment Tools

The experiment was conducted using the testing tool Taurus with JMeter as the test runner. These tools are used to stress test web services allowing for testing of the system during different levels of simulated load and produces a report on the system performance from the clients' point of view.

Level	**Virtual Users**	**Throughput**
Low	50	100
Medium	260	260
High	500	300

Table 3.2: Table of the different levels of traffic used during the experiment

Table 3.2 illustrates the different levels of traffic used during the tests. The level of traffic is controlled by the number of virtual users concurrently making requests to the system, and what throughput they should try to achieve. Each test runs for 5 minutes together with a 30 seconds ramp-up period. The scenario played out in the test only contains a single GET request to *http://test.axnion.xyz/item* which will tell the application server to fetch all item data from the database and return the information making use of both the application and database servers. When the test was complete the data was sent to Blazemeters. From there a .csv file with data from the test can be downloaded and the data extracted and archived.

4 Results

Here the results are presented from the experiment. More detailed data from each test execution can be found under *Appendix 1*.

4.1 Error Rate

Table 4.1 describes the relationship between each deployment scenario and the different amounts of traffic being sent to the system. The values represent the average percentage of requests not answered or request resulting in an error message from the servers. The data is also visualized as diagrams in *Figures 4.1, 4.2,* and *4.3*.

% of requests resulting in an error	No Deployment	Successful Deployment	Failed Deployment
Low Traffic	0.015%	0.008%	0.025%
Medium Traffic	0.014%	0.639 %	1.2 %
High Traffic	11.921%	19.107%	17.893 %

Table 4.1: Matrix describe the average percentage of requests which resulted in an error during test execution.

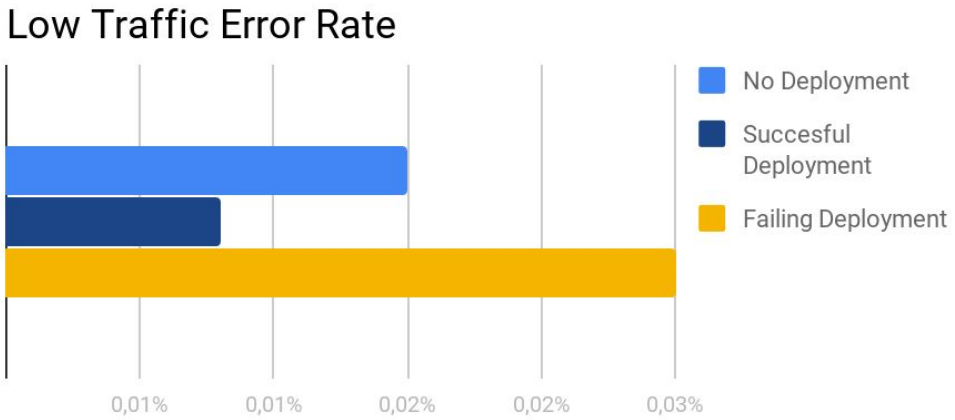

Figure 4.1: Diagram of average error rate during low traffic.

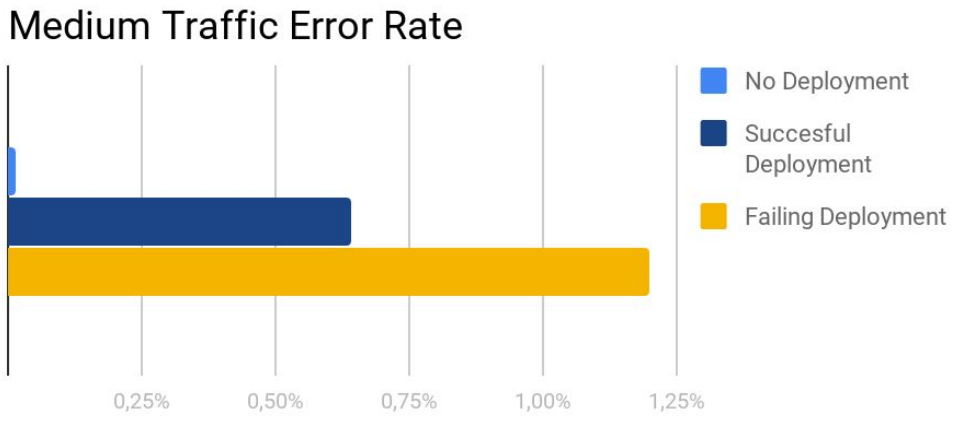

Figure 4.2: Diagram of average error rate during medium traffic.

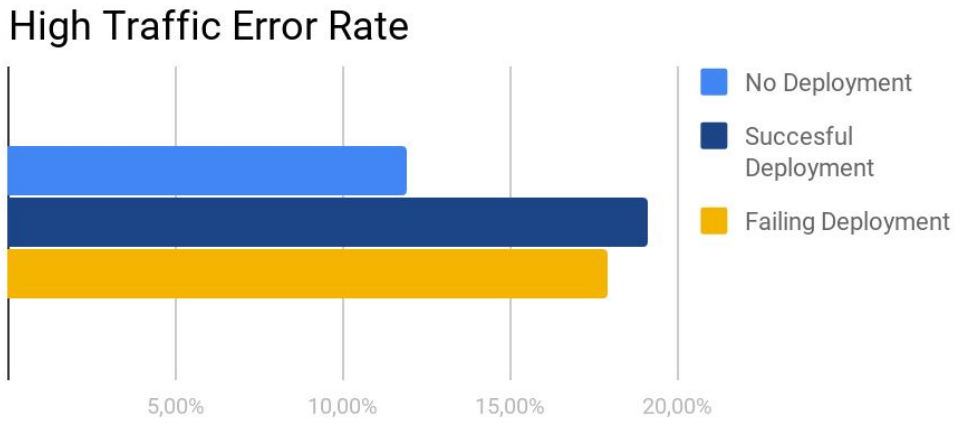

Figure 4.3: Diagram of average error rate during high traffic.

4.2 Response Time

Another datapoint collected with the testing tool was the time it took for a request to result in a response, what is known as response time. *Table 4.2* presents the average of the collected average response times during the execution of each experiment and *Table 4.3* shows the average of the highest recorded response times from the experiments. Both tables follow the same matrix pattern as *Table 4.1*. The data is again visualized using bar charts. The average response time is shown in *Figure 4.4* and the maximum response time is shown in *Figure 4.5*.

Avg Response Time (milliseconds)	No Deployment	Successful Deployment	Failed Deployment
Low Traffic	188	271	245
Medium Traffic	2298	4581	4914
High Traffic	8068	9685	9871

Table 4.2: Matrix describing the average response time (in milliseconds) for each type of deployment scenario with each level of traffic.

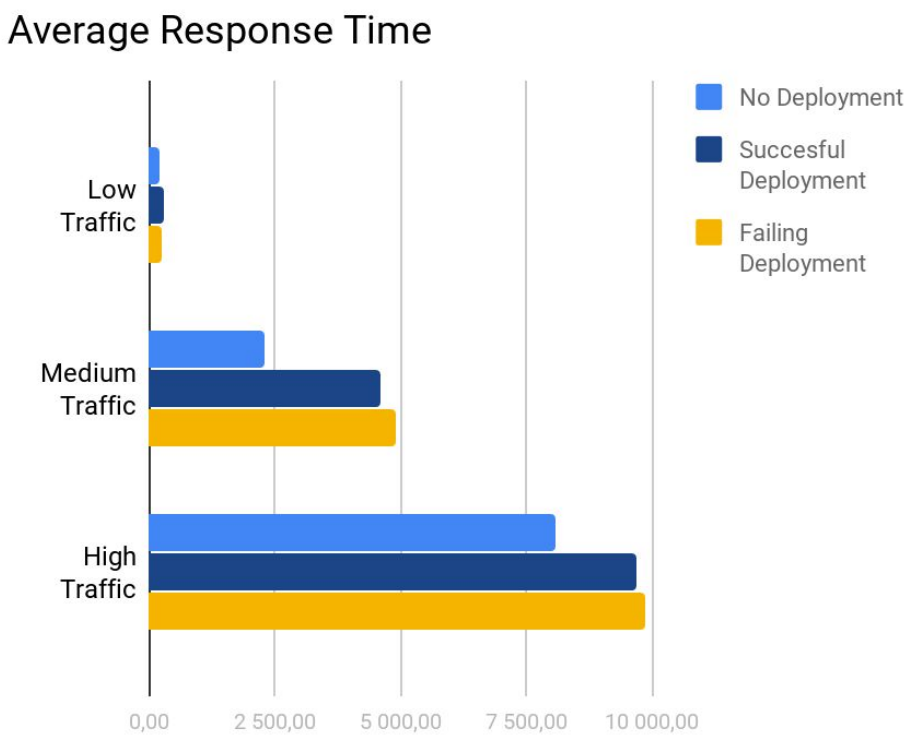

Figure 4.4: Diagram of average response time in the different scenarios.

Maximum Response Time	No Deployment	Successful Deployment	Failed Deployment

(milliseconds)			
Low Traffic	729	2007	1238
Medium Traffic	4685	45375	49602
High Traffic	68114	65279	68037

Table 4.3: Matrix describing the maximum response time (in milliseconds) for each type of deployment scenario with each level of traffic.

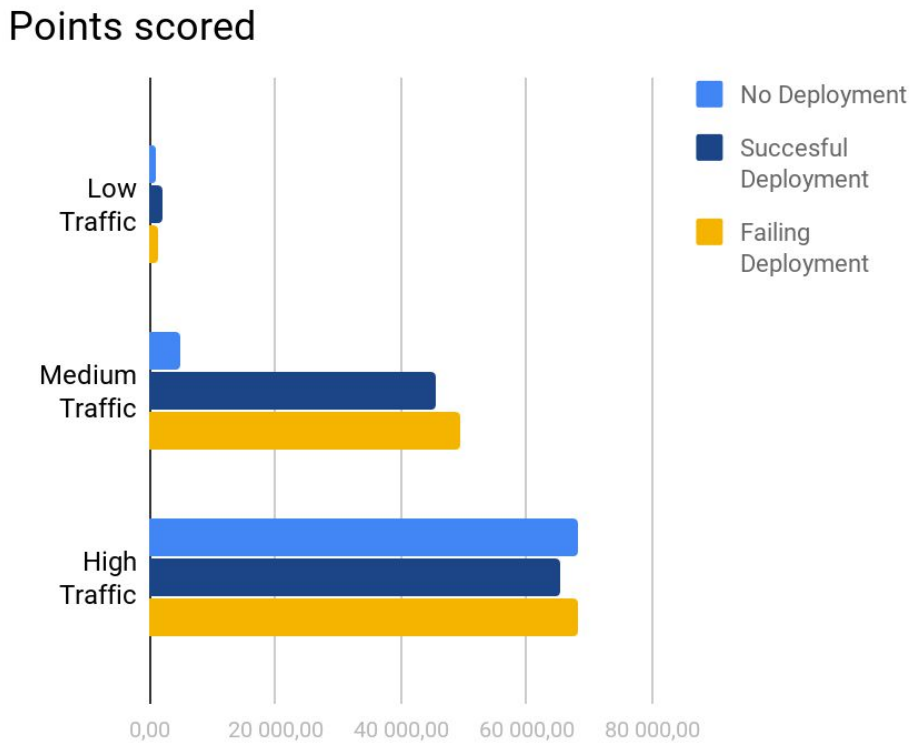

Figure 4.5: Diagram of maximum response time in the different scenarios.

5 Analysis

This chapter looks at the data from the results. Statistical tests are performed on the data sets to better understand the data and analyze it to understand how the system performed.

5.1 Statistical Testing

The statistical test compares values from the two deployment scenarios with the values from the scenario without a deployment. This was done on all three levels of traffic. The tests were performed with values from error rate, average response time, and maximum response time.

A T-test was used since there is no need to see the statistical significance of a successful and a failed deployment meaning the test only has to be performed with two values. The significance level is set to 0.05 and the values were taken from the tables in *Appendix 1*. Each box is color coded where red means there is no statistical significance and green that there is a statistical significance.

In this case, a statistical significance would mean that there is a significant enough difference between no deployment and each of the deployment scenarios. If the results are showing little or no statistical significance it would suggest the system is working as intended.

P-Value	Successful Deployment	Failed Deployment
Low Traffic	0.404	0.655
Medium Traffic	0.053	0.000
High Traffic	0.124	0.000

Table 5.1: The resulting P-value of the error rate

P-Value	Successful Deployment	Failed Deployment
Low Traffic	0.000	0.000
Medium Traffic	0.000	0.000

High Traffic	0.008	0.008

Table 5.2: The resulting P-value of the average response time

P-Value	Successful Deployment	Failed Deployment
Low Traffic	0.000	0.000
Medium Traffic	0.000	0.000
High Traffic	0.142	0.070

Table 5.3: The resulting P-value of the maximum response time

5.2 Error Rate

The averages in *Table 4.1* suggests that during low traffic the system can perform a deployment without an increase in errors. Since the blue-green deployment method does temporarily remove half of the application servers for large parts of the deployment process, the availability of the system could be jeopardized due to lack of resources. But during low traffic, the remaining two servers seem to do well enough to able to respond to all requests. The results from both successful deployments and failing deployment show no statistically significant difference when compared to the no deployment scenarios.

The results of the tests during medium and high traffic levels are different from the low traffic scenarios. The averages from *Table 4.1* show an increase in the average error rate, both during successful and failing deployments. However, according to the statistical tests, only the failing deployments seems to have a statistically significant difference compared to normal operations. This does show an increase in errors when the system has to recover and perform a rollback from a failed deployment and could indicate a flaw in the implementation.

5.3 Response time

The error rate only showed a statistical significance during medium and high traffic with a failing deployment. However, the response times were much more consistent on all traffic levels.

During low traffic levels, both the average response time and the maximum response time does seem to increase slightly, but the statistical tests do show that the difference is significant enough. A similar pattern is seen during medium and high traffic where the average response times have almost doubled during deployment compared to no deployment.

With the system under high load, the maximum response times stay very consistent during the tests, with an average around 68 seconds. The results from the medium traffic levels during deployment show a similar maximum response time happening during a deployment on medium traffic, but in that case at around 50 seconds. This number makes sense since the load balancers have a timeout value to the client at 50 seconds, meaning if the application server does not respond to the load balancer within 50 seconds the load balancer will respond with an error. Exactly why it takes 18 seconds extra for the load balancer to respond during high load is still unknown, but it is possible that even the load balancers are having trouble responding to the high traffic with error messages. Both normal operations and both deployment scenarios reach a similar maximum response time. Because of this, there is no statistical significance between them.

6 Discussion

According to the statistical tests from the experiment, there was no statistically significant change in the error rate values of a system during normal operations compared to a system performing a successful deployment. However, the deployment scenarios still create a significantly higher average error rate during medium to high levels of traffic compared to a service during normal operations. It could be that during a deployment at medium to high traffic the service becomes less stable, resulting in more fluctuations in the error rate. Unlike successful deployment, during a failed deployment at medium or high levels of traffic, there is a statistical significance when compared to the control values. This is most likely a flaw in the implementation of the rollback action.

Going solely by the error rates, it looks to be possible to use automation to deploy a web application to a high availability architecture while keeping the promise of high availability, at least during low levels of traffic. During higher levels of traffic a successful deployment can be performed with this automatic deployment system without affecting availability, but if the deployment fails, there will be a slight increase in error rate.

The Node application servers used in the implementation were much more resilient to crashing during high load than was expected. This meant that even during high load there would generally still be a response, it would just take longer before the response was given back to the load balancer to send back to the user. This means that most of the errors were produced due to the application servers not being able to respond before the 50-second timeout ran out on the load balancers. We could, therefore, alter the number of errors by changing the timeout values on the load balancers.

The response time tells a different story than the error rate and makes the answer to the research question less clear. The statistical tests make it clear that there is always a statistically significant change in the response time. But just because there is a significant change in response time does not mean that availability is affected. It depends on what kind of system it is and what is considered an acceptable response time. During low traffic, the average response time went up from 188 milliseconds without deployment to an average of 271 milliseconds during a successful deployment. While the statistical tests do say that the difference between the two data sets are

significant, depending on the application that change would be hardly noticeable. If the application is a website then most users would not notice the difference between 188 milliseconds to 271 milliseconds. If the application worked with timing critical real-time data then such an increase in response time might prove devastating.

Another dependency on the change in response time is computational resources. More resources behind the load balancers would give the ability to respond to more requests per second which would improve response time. The response time numbers discussed in the previous paragraph was at low levels of traffic, but during medium levels of traffic, it goes from the average response time of 2298 milliseconds to 4581 milliseconds for a successful deployment and 4914 milliseconds for a failed deployment. The average response time during a deployment is almost double the response time during normal operations. By scaling up the system resources to where what is now considered medium levels of traffic would then be considered low. Therefore, depending on the requirements of the system, what is considered high availability when it comes to the response time is subjective making it hard to draw a conclusion from the response time results.

Comparing the results of this experiment with other research dealing with high availability is hard since others often measure availability in a different way. Existing research in high availability often looks at larger aspects like orchestration and cloud, resulting in their time perspective being a lot broader than what is possible in this project and they generally measure availability in minutes of downtime per year or as the percentage of availability [13, 14, 16]. Running, monitoring, and maintaining the software system over a whole year would not be practical for this project and scaling it down would be very hard since, for example, restarting a server once will take the same amount of time if the experiment is running for an hour or if the experiment was running for a year, but the resulting percentage of downtime would differ immensely.

7 Conclusion

Can automation of the software deployment process allow for continuous deployment without compromising the requirement for high availability in a high availability architecture? Yes, provided there are enough computational resources to be able to respond to every request in what is considered a timely manner, even when only half the system is available.

However, something like continuous deployment where new releases are constantly being deployed might not be a good idea if availability is important. Since continuous deployment would mean that a deployment could happen at any time, it could happen during a time when the application receives the most traffic. If there is a need to have a consistent response time even during peak traffic, half the system has to be able to take care of the whole traffic load. This would mean that more than half of the system will never be utilized during normal operations. But we would not want that hit to happen during peak hours. Instead, daily deployment during times where the traffic is at its lowest might be a better solution. Half the system could then handle the traffic without any significant change in response time or availability while keeping the number of resources needed to a minimum.

7.1 Improvements

The main issue with this project was how to measure availability in a way that worked in a shorter time frame. Availability in existing research was generally measured in minutes per year [13, 14, 16] which is not a very practical way to measure availability in a short period of time. Using error rate of each request does work, but only shows part of the picture. Some modern web application platforms do not always crash during high load, they instead become flooded with requests to which they cannot respond to in time resulting in an increase in response time. Error rate could be coupled with response time by setting a limit to how long a request can wait before a response has to be received or it will count as an error. The problem with this is that what is considered the maximum response time has to be defined.

Another issue which has to be improved is the implementation. According to the tests, there seems to be a flaw with the rollback feature since it produced more errors than it should. This feature could be improved to allow for failed deployment to happen during medium or high traffic without increasing the error rate.

7.2 Future work

There are many areas of this project that can be investigated further or changed. For example, how would canary releasing compare to the blue-green technique used? How could containers be utilized in this situation, for example using Kubernetes instead of traditional load balancer clusters? What has been presented in this project is only one solution using one set of methods. There are endless possible combinations to test and compare to the results produced in this project.

Another angle which is overlooked in this project is cost. Is it worth creating the system proposed here or are you better off creating a smaller system and accept that there will be a slight monetary loss during downtime? For example, if all you need is an application server, database server, and maybe a load balancer for the possibility of future expansion, when is it then worth it to implement this system which requires at a minimum of four times the amount of servers?

References

[1] D. Hegoda, (2015, Aug. 17) High Availability Deployment Architecture. [Online]. Available: http://dasunhegoda.com/high-availability-deployment-architecture/1003

[2] E. Heidi, (2016, Nov. 4) What is High Availability?. [Online]. Available: https://www.digitalocean.com/community/tutorials/what-is-high-availability

[3] J.W. Rawles, (2001, Jul). Oracle9i Real Application Cluster. [Online]. Available: https://docs.oracle.com/cd/A91202_01/901_doc/rac.901/a89867/pshavdtl.htm

[4] M. Grzejszczak, (2016, Jun. 28). Zero-Downtime Deployment With a Database. [Online]. Available: https://dzone.com/articles/zero-downtime-deployment-with-a-database-1

[5] M. Fowler, (2010, Mar. 1). BlueGreenDeployment. [Online]. Available: https://martinfowler.com/bliki/BlueGreenDeployment.html

[6] D. Sato, (2014, Jun. 25) CanaryRelease [Online]. Available: https://martinfowler.com/bliki/CanaryRelease.html

[7] M. Fowler, (2006, May. 1) Continuous Integration. [Online]. Available: https://www.martinfowler.com/articles/continuousIntegration.html

[8] E. Bottcher, (2010, May. 26) Continuous Integration - If something hurts, do it more often. [Online]. Available: http://evan.bottch.com/2010/05/26/continuous-integration-if-something-hurts-do-it-more-often/

[9] M. Fowler, (2013, May. 30) ContinuousDelivery. [Online]. Available: https://martinfowler.com/bliki/ContinuousDelivery.html

[10] C. Caum, (2013, Aug. 30) Continuous Delivery Vs. Continuous Deployment: What's the Diff?. [Online]. Available:

https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff

[11] E. Heidi, (2016, Mar. 24) An Introduction to Configuration Management. [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-configuration-management

[12] N. Gibbs, (2015, Apr. 13) What's Deployment Versus Provisioning Versus Orchestration? [Online]. Available: http://codefol.io/posts/deployment-versus-provisioning-versus-orchestration

[13] R. Moreno-Vozmediano, R. S. Montero, E. Huedo, and I. M. Llorente, "Orchestrating the Deployment of High Availability Services on Multi-zone and Multi-cloud Scenarios," in *Int. J. Grid Util. Comput.*, vol. 16, no. 1, pp. 39–53, Mar. 2018.

[14] P. Endo, M. Rodrigues, G. Gonçalves, J. Kelner, D. Sadok, and C. Curescu, "High availability in clouds: systematic review and research challenges," *J Cloud Comp*, vol. 5, no. 1, pp. 1–15, 2016.

[15] L. Perkov, N. Pavkovic, J. Petrovic, "High-Availability Using Open Source Software", *MIPRO, 2011 Proceedings of the 34th International Convention*, pp.167–170, 2011.

[16] E. Braastad, "Management of high availability using virtualization", H. Haugerud, Ed., ed: Høgskolen i Oslo. Avdeling for ingeniørutdanning, 2006.

[17] S. Kaur, K. Kaur, and D. Singh, "A framework for hosting web services in cloud computing environment with high availability," ed, 2012, pp. 1-6.

[18] Y. Fountis, (2017, Mar. 2) Demystifying High Availability Architecture. [Online]. Available: https://pressidium.com/blog/2017/high-availability-architecture/

[19] L. E. Lwakatare, P. Kuvaja, and M. Oivo, "Relationship of devops to agile, lean and continuous deployment: A multivocal literature review study,"

Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 10027. pp. 399–415, 2016 [Online]. Available: http://dx.doi.org/10.1007/978-3-319-49094-6_27

[20] O.S. Tezer, (2014, Feb. 20) How To Configure DNS Round-Robin Load-Balancing For High-Availability. [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-configure-dns-round-robin-load-balancing-for-high-availability

[21] Cloudflare, (n.d.) What is DNS? How Does A Domain Name Server Work?. [Online]. Available: https://www.cloudflare.com/learning/dns/what-is-dns/

[22] Cloudflare, (n.d.) Round-Robin DNS. [Online]. Available: https://www.cloudflare.com/learning/dns/glossary/round-robin-dns/

[23] HAProxy Website, (n.d.) HAProxy. [Online]. Available: http://www.haproxy.org/

[24] Apache Cassandra Website, (n.d.). *Apache Cassandra*. [Online]. Available: http://cassandra.apache.org/

[25] Ansible Website, (n.d.). How Ansible Works. [Online]. Available: https://www.ansible.com/overview/how-ansible-works

A Appendix 1

This appendix contains information gathered from the test reports when running the experiment.

#	Samples	Avg Rspns Time (ms)	Min Rspns Time (ms)	Max Rspns Time (ms)	Avg Thruput (rqst/s)	Error Count	Error Rate (%)
1	31315	134	26	452	108.54	0	0
2	31339	181	26	805	110.07	2	0.006
3	31281	221	28	753	107.11	0	0
4	31383	198	31	784	107.33	12	0.038
5	31334	211	30	859	109.52	0	0
6	31295	202	28	798	107.53	0	0
7	31357	190	26	606	106.14	20	0.064
8	31286	194	28	906	107.26	0	0
9	31386	208	14	797	107.42	14	0.045
10	31321	138	26	528	109.12	0	0
avg	31330	188	26	729	108	48	0.015

Table A.1: Test results from low traffic and no deployment.

#	Samples	Avg Rspns Time (ms)	Min Rspns Time (ms)	Max Rspns Time (ms)	Avg Thruput (rqst/s)	Error Count	Error Rate (%)
1	30339	261	27	1925	101.59	7	0,023
2	30168	271	26	1989	97.58	0	0
3	30324	252	26	1992	105.51	8	0,026

4	30256	275	26	1920	102.25	0	0
5	30204	282	27	2049	99.47	0	0
6	30299	268	27	2115	104.35	0	0
7	30039	278	27	2083	91.27	0	0
8	30260	251	27	2145	102.37	9	0,03
9	30180	275	26	1840	98.35	0	0
10	30204	295	27	2013	99.47	0	0
avg	30227	271	27	2007	100.22	2	0.008

Table A.2: Test results from low traffic and successful deployment.

#	Samples	Avg Rspns Time (ms)	Min Rspns Time (ms)	Max Rspns Time (ms)	Avg Thruput (rqst/s)	Error Count	Error Rate (%)
1	30704	234	26	1225	93.42	0	0
2	30596	252	29	1329	103.55	0	0
3	30648	235	27	1249	106.33	9	0,029
4	30652	243	27	1300	106.45	0	0
5	30710	238	29	1092	94.01	0	0
6	30671	236	27	1238	103.02	8	0,026
7	30635	239	25	1179	105.53	0	0
8	30773	251	28	1312	97.12	6	0,19
9	30572	250	27	1249	102.42	0	0
10	30576	270	26	1211	102.55	0	0
avg	30654	245	27	1238	101.44	2	0.025

Table A.3: Test results from low traffic and failing deployment.

#	Samples	Avg Rspns Time (ms)	Min Rspns Time (ms)	Max Rspns Time (ms)	Avg Thruput (rqst/s)	Error Count	Error Rate (%)
1	32848	2541	65	6187	107.59	0	0
2	37020	2239	37	4575	115.02	0	0
3	36733	2244	28	4351	116.12	3	0,08
4	36354	2263	30	4835	112.44	2	0,06
5	36433	2272	31	4411	116.43	0	0
6	36580	2263	30	4563	124.08	0	0
7	36138	2289	35	4567	117.29	0	0
8	36243	2282	33	4107	122.47	0	0
9	36301	2282	30	4551	110.03	0	0
10	35904	2306	32	4703	108.8	0	0
avg	36055	2298	35	4685	115.03	1	0.014

Table A.4: Test results from medium traffic and no deployment.

#	Samples	Avg Rspns Time (ms)	Min Rspns Time (ms)	Max Rspns Time (ms)	Avg Thruput (rqst/s)	Error Count	Error Rate (%)
1	18907	4573	28	45247	63.55	0	0
2	19892	4214	26	38399	64.39	70	0,352
3	18293	4747	25	49087	70.42	0	0
4	20839	4112	31	50079	78.23	56	0,269
5	19739	4301	28	38847	72.24	0	0

6	19345	4400	26	41375	68.21	162	0,837
7	19618	4405	28	50047	66.28	86	0,438
8	17038	5223	28	50047	61.46	231	1,356
9	17191	5246	29	50047	53.59	525	3,053
10	18715	4592	26	40575	63.48	15	0,08
avg	18958	4581	28	45375	66.19	115	0.639

Table A.5: Test results from medium traffic and successful deployment.

#	Samples	Avg Rspns Time (ms)	Min Rspns Time (ms)	Max Rspns Time (ms)	Avg Thruput (rqst/s)	Error Count	Error Rate (%)
1	17961	4817	28	50047	63.53	351	1,954
2	17396	4485	28	50047	63.55	373	2,144
3	15477	5659	26	50079	50.06	284	1,835
4	19856	4338	27	50015	65.44	311	1,566
5	17694	5042	24	50047	58.36	188	1,063
6	16400	5347	26	50015	50	68	0,415
7	20532	4175	26	47615	65.38	0	0
8	17277	4982	28	48095	68.08	168	0,972
9	17725	5055	28	50047	57.25	261	1,472
10	16959	5241	27	50015	57.31	99	0,584
avg	17728	4914	27	49602	59.90	210	1.2

Table A.6: Test results from medium traffic and failing deployment.

#	Samples	Avg Rspns	Min Rspns	Max Rspns	Avg Thruput	Error Count	Error Rate

		Time (ms)	Time (ms)	Time (ms)	(rqst/s)		(%)
1	20295	7431	27	68095	72.27	2348	11,569
2	22625	7209	26	68351	77.21	2644	11,686
3	16846	9895	24	68031	51.48	2335	13,861
4	20201	7485	28	68031	67.41	2298	11,376
5	21690	6963	27	68031	68.08	2463	11,355
6	18064	8269	25	68223	66.19	1848	10,23
7	18892	8893	26	68031	61.08	2364	12,513
8	19107	8857	25	68287	57.9	2375	12,43
9	16114	8992	26	68031	49.23	2206	13,69
10	21927	6688	27	68031	73.25	2302	10,498
avg	19576	8068	26	68114	64.41	2318	11.921

Table A.7: Test results from high traffic and no deployment.

#	Samples	Avg Rspns Time (ms)	Min Rspns Time (ms)	Max Rspns Time (ms)	Avg Thruput (rqst/s)	Error Count	Error Rate (%)
1	17008	8365	29	68031	59.59	1963	11,542
2	13836	12170	29	50303	58.09	1721	12,439
3	15993	10704	30	68031	55.44	1842	11,518
4	16208	8955	25	66047	50.55	1830	11,291
5	18935	9169	28	68031	63.19	2667	14,085
6	16558	9711	27	68031	52.56	2087	12,604
7	14417	11438	27	68031	54.28	2496	17,313

8	16522	9077	29	60031	53.39	2276	13,776
9	17992	9385	25	68223	62.41	2732	15,185
10	17989	7877	29	68031	62.32	2036	11,318
avg	16546	9685	28	65279		2165	19.107

Table A.8: Test results from high traffic and successful deployment.

#	Samples	Avg Rspns Time (ms)	Min Rspns Time (ms)	Max Rspns Time (ms)	Avg Thruput (rqst/s)	Error Count	Error Rate (%)
1	13097	12738	28	68031	50.28	2249	17,172
2	16717	10084	27	68031	60.58	2555	15,284
3	15298	11097	28	68031	54.18	2462	16,094
4	19112	8436	28	68031	72.15	2601	13,609
5	18343	8869	26	68095	64.45	2620	14,283
6	20759	7384	28	68031	64.37	2512	12,101
7	14456	11115	27	68031	56.26	2491	17,232
8	16278	10633	28	68031	54.27	2608	16,022
9	17814	9195	29	68031	56.26	2375	13,332
10	18401	9154	30	68031	56.33	2539	13,798
avg	17028	9871	28	68037	58.91	2501	17.893

Table A.9: Test results from high traffic and failing deployment.